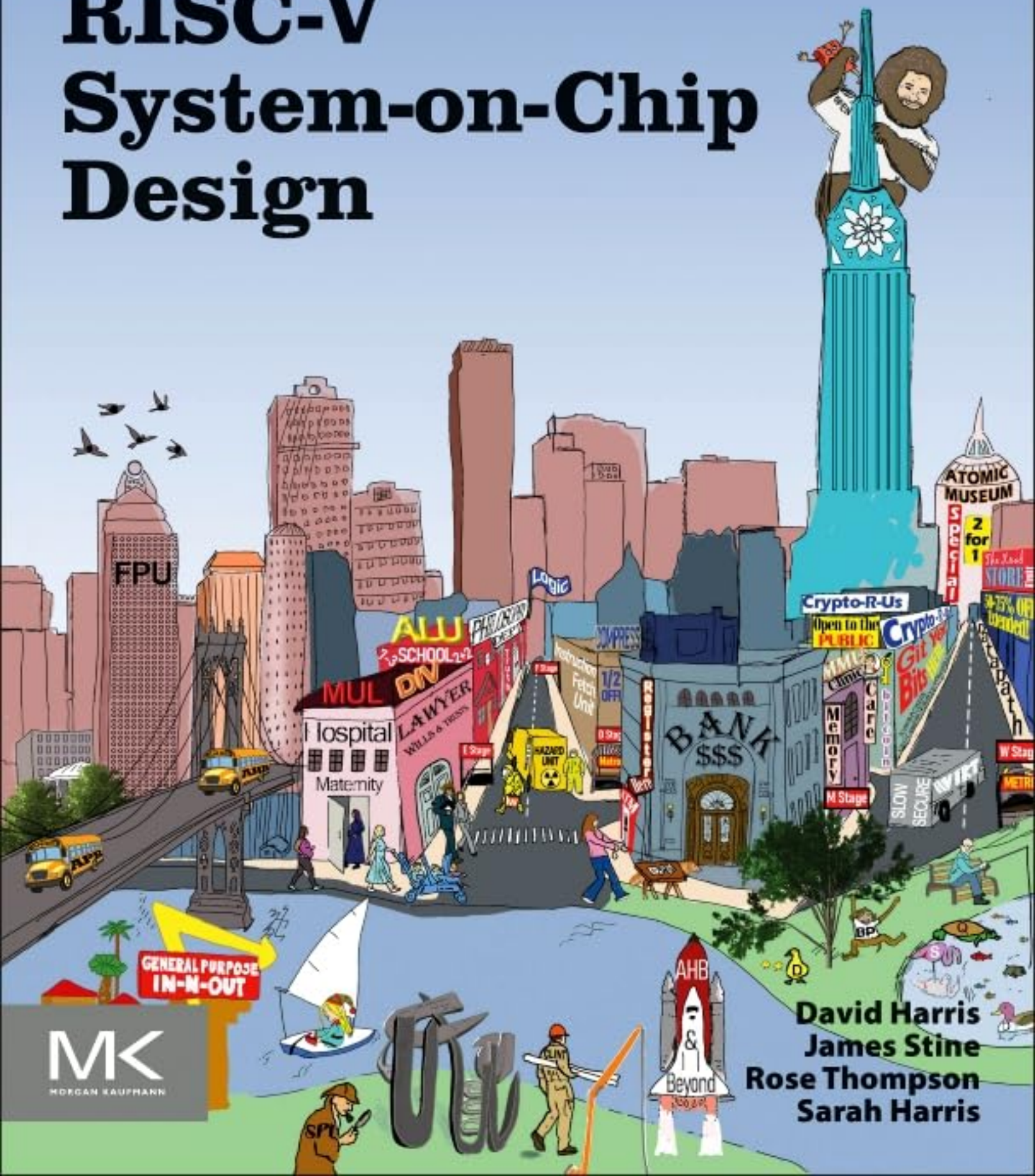


RISC-V System-on-Chip Design



David Harris
James Stine
Rose Thompson
Sarah Harris

MK
MORGAN KAUFMANN

<https://pages.hmc.edu/harris/ddca/rvsocd.html>
<https://github.com/openhwgroup/cvw>

RISC-V System-on-Chip Design

David Harris
James Stine
Rose Thompson
Sarah Harris



ELSEVIER



MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

For accessibility purposes, images in electronic versions of this book are accompanied by alt text descriptions provided by Elsevier. For more information, see <https://www.elsevier.com/about/accessibility>.

Books and Journals published by Elsevier comply with applicable product safety requirements. For any product safety concerns or queries, please contact our authorised representative, Elsevier B.V., at productsafety@elsevier.com.

Publisher's note: Elsevier takes a neutral position with respect to territorial disputes or jurisdictional claims in its published content, including in maps and institutional affiliations.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Front cover art is by Sarah Harris.

Back cover image is a die photograph of the FE310, released in 2016 as the world's first commercially-available RISC-V chip. Image © 2016 SiFive, reproduced with permission.

ISBN: 978-0-323-99498-9

For information on all Morgan Kaufmann publications visit our website at <https://www.elsevier.com/books-and-journals>

Publisher: Peter Linsley
Acquisitions Editor: Stephen Merken
Editorial Project Manager: Vicky Sharma
Production Project Manager: Prasanna Kalyanaraman
Cover Designer: Greg Harris

Typeset by TNQ Tech



Contents

Preface	xix
How to Use This Book	xxiii
Acknowledgements	xxv
About the Authors	xxvii
Foreword	xxix
Chapter 1 A Brief History of Computer Design	1
1.1 Moore's Law and Beyond	2
1.2 System-on-Chip	6
1.3 Birth of Computing	9
1.4 Mainframes and Minicomputers	13
1.5 Microprocessors	16
1.6 CISC and RISC	20
1.7 RISC-V	26
1.8 International Economic and Security Competition	26
1.9 Summary	29
Chapter 2 Introduction to RISC-V	31
2.1 RISC-V Assembly Language	32
2.1.1 <i>R-Type Instructions</i>	32
2.1.2 <i>I-Type Instructions</i>	34
2.1.3 <i>S-Type Instructions</i>	34
2.1.4 <i>B-Type Instructions</i>	35
2.1.5 <i>J-Type Instructions</i>	35
2.1.6 <i>U-Type Instructions</i>	35
2.1.7 <i>Other Instructions</i>	36
2.1.8 <i>RV64I Additional Instructions</i>	36
2.1.9 <i>Register Conventions</i>	37
2.1.10 <i>Endianness</i>	37
2.1.11 <i>Pseudoinstructions</i>	37
2.1.12 <i>C to Assembly Language Examples</i>	38
2.2 RISC-V Machine Language	43

2.3	Have a Hart	45
2.3.1	Privilege Modes	46
2.3.2	Control and Status Registers	46
2.3.3	Processes and Threads	46
2.3.4	Cores and Harts	47
2.3.5	Exceptions, Interrupts, and Traps	49
2.3.6	Execution Environment	49
2.4	Memory Map	51
2.5	RISC-V Extensions and Profiles	52
2.5.1	RISC-V Profiles	53
2.6	Comparison with Other Architectures	54
2.7	RISC-V Microarchitecture	56
2.7.1	Single-Cycle Microarchitecture	56
2.7.2	Pipelined Microarchitecture	65
2.7.3	Verification	67
2.8	Wally System-on-Chip	70
2.8.1	Wally Block Diagram	73
2.8.2	Wally Configurations	73
2.8.3	Wally Code Base	74
2.8.4	Verification	76
2.8.5	Benchmarking	76
2.8.6	Implementation	77
2.9	RISC-V Market	80
2.10	Summary and a Look Ahead	85

Chapter 3 RISC-V Software Tool Flow 89

3.1	Platform Requirements and Tool Overview	90
3.2	Getting Started	90
3.3	GCC Assembler and Spike Simulator	91
3.3.1	A Quick Tutorial	91
3.3.2	Linking and ELF	93
3.3.3	Application Binary Interface	95
3.3.4	Supervisor Binary Interface	99
3.3.5	Machine Architecture	99
3.3.6	Driving GCC	101
3.3.7	Assembler Directives	103
3.3.8	Spike Simulation	105
3.3.9	Performance Measurement	107
3.3.10	Putting It All Together	107
3.4	GCC GNU C Compiler	114
3.4.1	A First C Program	115
3.4.2	Inline Assembly Language	120
3.4.3	Mixed C and Assembly Language	121
3.4.4	Standard Library and Startup Code	122

3.4.5	<i>Compiler Explorer</i>	123
3.5	Sail Simulation	124
3.6	QEMU Simulation	127
3.6.1	<i>Simulating with QEMU</i>	127
3.6.2	<i>Debugging With GDB</i>	128
3.7	Test Suites	133
3.7.1	<i>riscv-arch-test</i>	133
3.7.2	<i>RISCOF</i>	137
3.7.3	<i>wally-riscv-arch-test</i>	138
3.8	Summary	140
Chapter 4 HDL Design Practices		141
4.1	HDL Design Methodology	142
4.1.1	<i>Design Strategies</i>	142
4.1.2	<i>Version Control</i>	144
4.1.3	<i>Reuse</i>	144
4.2	Hardware Design with SystemVerilog	145
4.2.1	<i>Modules</i>	145
4.2.2	<i>Combinational Logic</i>	148
4.2.3	<i>Sequential Logic</i>	153
4.2.4	<i>Verilog Syntax</i>	168
4.2.5	<i>Configurable Hardware</i>	173
4.3	Wally Style Guidelines	181
4.3.1	<i>Signals</i>	181
4.3.2	<i>Spacing and Indentation</i>	185
4.3.3	<i>Synchronous Design</i>	187
4.3.4	<i>Modules</i>	188
4.3.5	<i>Configurable Hardware</i>	189
4.3.6	<i>Comments</i>	189
4.4	Wally Configuration	190
4.5	Summary	191
Chapter 5 Design Verification		193
5.1	Verification Strategies	195
5.1.1	<i>Test Vector Generation</i>	196
5.1.2	<i>Reference Models</i>	198
5.1.3	<i>Unit vs. System Testing</i>	198
5.1.4	<i>Emulation</i>	198
5.1.5	<i>Technology Readiness Levels</i>	199
5.1.6	<i>Design and Verification Approaches</i>	200
5.2	Design Verification Plan	202
5.3	Lint	203
5.4	Testbenches	204

5.5	Logic Simulators	207
5.5.1	<i>Questa</i>	207
5.5.2	<i>Verilator</i>	210
5.5.3	<i>VCS</i>	212
5.6	Lock-Step-Compare	212
5.6.1	<i>RVVI-TRACE Interface</i>	214
5.7	Assertions	214
5.7.1	<i>Immediate Assertions</i>	215
5.7.2	<i>Concurrent Assertions</i>	216
5.8	Code Coverage	217
5.9	Functional Coverage	221
5.9.1	<i>riscvISACOV</i>	222
5.9.2	<i>Directed Tests</i>	222
5.9.3	<i>Constrained Random Tests</i>	226
5.10	Formal Verification	226
5.11	Wally Verification	227
5.11.1	<i>Design Verification Plan</i>	227
5.11.2	<i>Reference Model</i>	229
5.11.3	<i>Top-Level Testbench and Testvectors</i>	229
5.11.4	<i>Lint</i>	230
5.11.5	<i>Simulation</i>	231
5.11.6	<i>Regression</i>	233
5.11.7	<i>Debugging Hints</i>	235
5.11.8	<i>Floating-Point Unit Testing</i>	239
5.11.9	<i>Lock-Step-Compare</i>	240
5.11.10	<i>Assertions</i>	241
5.11.11	<i>Code Coverage</i>	241
5.11.12	<i>Functional Coverage</i>	245
5.12	Summary	247
Chapter 6	Logic Synthesis	249
6.1	Performance, Power, and Area (PPA)	250
6.1.1	<i>Performance</i>	250
6.1.2	<i>Power</i>	255
6.1.3	<i>Area</i>	257
6.2	Fabrication Technologies	257
6.2.1	<i>Inverter Characteristics</i>	259
6.2.2	<i>Datapath Units</i>	260
6.2.3	<i>Memories</i>	266
6.3	Running Synthesis	268
6.4	Reviewing Log Files	269
6.4.1	<i>synth.out</i>	269
6.4.2	<i>Mapped Output Files</i>	270
6.4.3	<i>Quality of Results (QoR) Reports</i>	275

6.4.4	<i>Performance Reports</i>	277
6.4.5	<i>Power Reports</i>	282
6.4.6	<i>Area Reports</i>	284
6.4.7	<i>Other Reports</i>	284
6.5	Wally Synthesis Results	287
6.5.1	<i>Synthesis Configurations</i>	287
6.5.2	<i>Area and Delay vs. Target Clock Period</i>	287
6.5.3	<i>Area and Delay vs. Configurations</i>	291
6.5.4	<i>Area and Delay vs. Features in rv64gc</i>	295
6.6	Summary	297
Chapter 7 Pipelined Core		299
7.1	Single-Cycle Core	301
7.1.1	<i>Architectural State</i>	302
7.1.2	<i>Immediate Extension</i>	303
7.1.3	<i>Arithmetic Circuitry</i>	303
7.1.4	<i>Controller Design</i>	311
7.1.5	<i>Instruction Fetch Unit</i>	314
7.1.6	<i>Load/Store Unit</i>	314
7.1.7	<i>Sample Program</i>	316
7.2	Pipelining the RV32I Core	317
7.2.1	<i>Synchronous Memories</i>	319
7.2.2	<i>Hazards</i>	321
7.3	SystemVerilog	328
7.4	Test Plan	329
7.5	Summary	333
Chapter 8 Privileged Operations		335
8.1	Principles	336
8.1.1	<i>Security and Protection</i>	336
8.1.2	<i>Traps</i>	337
8.1.3	<i>Control and Status Registers (CSRs)</i>	338
8.2	RISC-V Practices	339
8.2.1	<i>Privilege Modes</i>	340
8.2.2	<i>Privileged Instructions</i>	340
8.2.3	<i>Control and Status Registers (CSRs)</i>	342
8.2.4	<i>Reset Behavior</i>	349
8.2.5	<i>Traps</i>	350
8.2.6	<i>Trap Handler</i>	357
8.3	Test Plan	360
8.3.1	<i>Generating Exceptions</i>	362
8.3.2	<i>Generating Interrupts</i>	363

8.4	Wally Implementation	364
8.4.1	Wally Privileged Unit	364
8.4.2	Privileged Instructions	370
8.4.3	Control and Status Registers	370
8.4.4	Traps	371
8.4.5	Pipeline Hazards	373
8.4.6	Instruction Fetch Unit (IFU) and Privileged Changes	377
8.4.7	Endian Swapper	378
8.5	Summary	380
Chapter 9 Bus Interface		383
9.1	Principles	383
9.1.1	Bidirectional vs. Unidirectional Buses	385
9.1.2	Handshaking	386
9.1.3	Multiple Peripherals	386
9.1.4	Multiple Controllers	387
9.1.5	Other Bus Features	389
9.1.6	High-Speed Off-Chip Links	390
9.2	AMBA Buses	391
9.2.1	APB	392
9.2.2	AHB	395
9.2.3	AXI	404
9.3	Other SoC Buses	410
9.4	Test Plan	410
9.5	Wally Implementation	411
9.5.1	LSU and IFU AHB Interface	412
9.5.2	External Bus Unit	415
9.5.3	System-on-Chip	417
9.5.4	Combined Bus and Tightly Integrated Memories	420
9.5.5	Bus Hazards	423
9.6	Summary	427
Chapter 10 Caches		429
10.1	Principles	430
10.1.1	Cache Performance	430
10.1.2	Locality	432
10.1.3	Direct-Mapped Cache	432
10.1.4	Cache Writes	434
10.1.5	Associative Caches	436
10.1.6	Cache Design Tradeoffs	438
10.1.7	Replacement Policy	439
10.1.8	Cache Flush and Invalidation	441

10.2	RISC-V Practices	442
10.3	Test Plan	442
10.4	Wally Implementation	443
10.4.1	<i>Interfaces</i>	445
10.4.2	<i>Cache Way Module</i>	446
10.4.3	<i>Cache FSM</i>	454
10.4.4	<i>pLRU Implementation</i>	456
10.4.5	<i>Top-Level Cache Logic</i>	459
10.4.6	<i>Pipeline Stalls</i>	466
10.4.7	<i>Exceptions and Interrupts</i>	466
10.5	Summary	467
Chapter 11 Memory Management Unit		469
11.1	Principles	469
11.1.1	<i>Virtual Memory</i>	469
11.1.2	<i>Physical Memory Checking</i>	477
11.2	RISC-V Practices	478
11.2.1	<i>RISC-V Virtual Memory</i>	478
11.2.2	<i>RISC-V Memory Protection</i>	489
11.2.3	<i>Memory Exceptions</i>	494
11.2.4	<i>Fences</i>	495
11.2.5	<i>HPTW</i>	495
11.3	Test Plan	495
11.4	Wally Implementation	495
11.4.1	<i>MMU</i>	497
11.4.2	<i>HPTW</i>	507
11.4.3	<i>MMU Faults</i>	511
11.4.4	<i>Fence Instruction</i>	511
11.5	Summary	512
Chapter 12 Load/Store Unit		515
12.1	Wally Load/Store Unit Overview	515
12.2	Data Cache Integration	518
12.2.1	<i>Cache Line Fetches</i>	520
12.2.2	<i>Cache Line Writeback</i>	523
12.2.3	<i>Burst EBU</i>	525
12.3	Memory Management and HPTW Integration	526
12.3.1	<i>PMA Checker</i>	527
12.3.2	<i>PMP Checker</i>	529
12.3.3	<i>TLB and Cache Integration</i>	529
12.3.4	<i>HPTW Integration</i>	531

12.4	Stalls, Flushes, and Traps	532
12.4.1	LSU Stall Generation	532
12.4.2	LSU Trap Generation	533
12.4.3	Disabling Interrupts	533
12.4.4	LSU Response to Stalls and Flushes	533
12.5	Examples	534
12.5.1	Cache Load Hit	534
12.5.2	Cache Store Hit	534
12.5.3	Cache Miss	536
12.5.4	Uncached Access	537
12.5.5	TLB Miss	540
12.5.6	TLB Access Fault	543
12.6	Test Plan	544
12.7	Summary	544

Chapter 13 Instruction Fetch Unit 545

13.1	Branch Prediction Principles	545
13.1.1	Static Branch Prediction	548
13.1.2	Simple Dynamic Branch Prediction	548
13.1.3	Gshare	551
13.1.4	Branch Target Buffer	554
13.1.5	Return Address Stack	556
13.1.6	TAGE	557
13.2	Test Plan	558
13.2.1	Correctness	558
13.2.2	Performance	558
13.3	Wally Implementation	559
13.3.1	Instruction Cache	560
13.3.2	PC Logic	560
13.3.3	Branch Predictor	562
13.3.4	ITLB Misses	574
13.3.5	Reset, Stalls, Flushes, and Traps	574
13.4	Summary	576

Chapter 14 Extensions: C (Compressed) 579

14.1	RISC-V Practices	579
14.1.1	Zc* Extension	580
14.2	Test Plan	582
14.3	Wally Implementation	582
14.3.1	Instruction Decompression	582
14.3.2	PC Logic	585
14.3.3	Spills	586
14.4	Summary	590

Chapter 15 Extensions: M (Multiply and Divide)	591
15.1 Principles	591
15.1.1 Signed and Unsigned Arithmetic	591
15.1.2 Carry-Save Addition	592
15.1.3 Multiplication	593
15.1.4 Division	602
15.2 RISC-V Practices	604
15.2.1 Signed and Unsigned Operations	604
15.2.2 Division Special Cases	604
15.2.3 Software Arithmetic Implementations	605
15.2.4 Zmmul	606
15.3 Test Plan	606
15.4 Wally Implementation	607
15.4.1 Multiplication	607
15.4.2 Division and Remainder	610
15.4.3 Multiply and Divide Unit	613
15.4.4 Hazard Unit	614
15.5 Summary	615
 Chapter 16 Extensions: F/D/Q/Zfh/Zfa (Floating-Point)	 617
16.1 Principles	617
16.1.1 Normalized Floating-Point Formats	618
16.1.2 Special Cases	620
16.1.3 Rounding	622
16.1.4 Flags	626
16.1.5 Fixed-Point Notation	628
16.2 Floating-Point Unit Design	630
16.2.1 Unpack Unit	631
16.2.2 Fused Multiply-Add	633
16.2.3 Division and Square Root	637
16.2.4 Conversion	639
16.2.5 Comparison	640
16.2.6 Postprocessing Unit	641
16.3 SoftFloat	641
16.3.1 SoftFloat Demo	643
16.3.2 fpcalc	643
16.4 RISC-V Practices	646
16.4.1 Floating-Point Control and Status Register	646
16.4.2 Registers and NaN Boxing	648
16.4.3 NaNs in RISC-V	648
16.4.4 Floating-Point Move Instructions	649
16.4.5 Zfa Additional Floating-Point Instructions	649
16.5 Test Plan	651
16.5.1 riscv-arch-test	651
16.5.2 TestFloat	652

16.6	Wally Implementation	655
16.6.1	<i>Floating-Point Unit</i>	656
16.6.2	<i>Unpacking Unit</i>	659
16.6.3	<i>Fused Multiply-Add</i>	660
16.6.4	<i>Divide and Square Root</i>	660
16.6.5	<i>Convert</i>	661
16.6.6	<i>Move</i>	662
16.6.7	<i>Compare</i>	662
16.6.8	<i>Sign Injection</i>	662
16.6.9	<i>Classify</i>	662
16.6.10	<i>Zfa Additional Floating-Point Instructions</i>	663
16.6.11	<i>Postprocessing</i>	666
16.6.12	<i>Privileged Unit Changes</i>	666
16.6.13	<i>Integer Execution Unit Changes</i>	666
16.6.14	<i>Hazard Unit Changes</i>	667
16.6.15	<i>LSU Changes</i>	669
16.7	Summary	669
Chapter 17 Extensions: A (Atomic)		673
17.1	Principles	673
17.1.1	<i>Concurrency</i>	674
17.1.2	<i>Coherence</i>	676
17.1.3	<i>Consistency</i>	677
17.1.4	<i>Synchronization</i>	682
17.1.5	<i>Deadlock, Livelock, and Starvation</i>	696
17.2	RISC-V Practices	698
17.2.1	<i>Consistency Models</i>	698
17.2.2	<i>Fences</i>	699
17.2.3	<i>Atomic Instructions</i>	700
17.2.4	<i>Atomic Subsets</i>	705
17.2.5	<i>Profile Extensions</i>	705
17.3	Test Plan	706
17.4	Wally Implementation	706
17.4.1	<i>Atomic Module</i>	708
17.4.2	<i>Timing Diagrams</i>	711
17.5	Summary	712
Chapter 18 Extensions: Zb* & Zk* (Bit Manipulation and Cryptography)		715
18.1	Cryptography Principles	716
18.1.1	<i>Finite Fields</i>	716
18.1.2	<i>AES: Advanced Encryption Standard</i>	722
18.1.3	<i>Hash Functions</i>	731

18.2	RISC-V Practices	734
18.2.1	<i>Zba: Address Generation</i>	734
18.2.2	<i>Zbb: Basic Bit Manipulation</i>	736
18.2.3	<i>Zbc: Carry-less Multiplication</i>	738
18.2.4	<i>Zbs: Single-Bit Operations</i>	739
18.2.5	<i>Zbkb: Bit Manipulation for Cryptography</i>	739
18.2.6	<i>Zbkx: Crossbar Permutations</i>	741
18.2.7	<i>Zknd and Zkne: AES Decryption and Encryption</i>	742
18.2.8	<i>Zknh: SHA-2 Instructions</i>	747
18.3	Test Plan	748
18.4	Wally Implementation	748
18.4.1	<i>ALU Modifications for Bit Manipulation</i>	748
18.4.2	<i>ALU Modifications for Cryptography</i>	757
18.4.3	<i>Controller Modifications</i>	764
18.5	Summary	765
Chapter 19 Other Extensions		767
19.1	Cache Management: Zicbom, Zicboz, Zicbop	767
19.1.1	<i>RISC-V CMO Instructions</i>	767
19.1.2	<i>Test Plan</i>	769
19.1.3	<i>Wally Implementation</i>	769
19.2	Misaligned Loads and Stores: Zicclsm	775
19.2.1	<i>Test Plan</i>	776
19.2.2	<i>Wally Implementation</i>	776
19.3	TLB Management: Svinval	777
19.4	Pause Hint: Zihintpause	779
19.5	Conditional Instructions: Zicond	779
19.5.1	<i>Test Plan</i>	781
19.5.2	<i>Wally Implementation</i>	782
19.6	Custom Extensions	783
19.7	Summary	784
Chapter 20 Peripherals		784.e1
20.1	GPIO	784.e2
20.1.1	<i>Memory Map</i>	784.e3
20.1.2	<i>Loopback Testing</i>	784.e5
20.1.3	<i>Test Plan</i>	784.e5
20.1.4	<i>Wally Implementation</i>	784.e6
20.2	CLINT	784.e8
20.2.1	<i>Memory Map</i>	784.e8
20.2.2	<i>Test Plan</i>	784.e9
20.2.3	<i>Wally Implementation</i>	784.e10

20.3	PLIC	784.e10
20.3.1	<i>Memory Map</i>	784.e11
20.3.2	<i>Test Plan</i>	784.e12
20.3.3	<i>Wally Implementation</i>	784.e13
20.4	UART	784.e13
20.4.1	<i>Memory Map</i>	784.e16
20.4.2	<i>Test Plan</i>	784.e18
20.4.3	<i>Wally Implementation</i>	784.e19
20.5	SPI	784.e19
20.5.1	<i>Memory Map</i>	784.e20
20.5.2	<i>Loopback Testing</i>	784.e22
20.5.3	<i>Test Plan</i>	784.e23
20.5.4	<i>Wally Implementation</i>	784.e23
20.6	Summary	784.e23
Chapter 21 Benchmarking		784.e24
21.1	CoreMark	784.e24
21.1.1	<i>Porting CoreMark</i>	784.e26
21.1.2	<i>CoreMark Compiler Flags</i>	784.e27
21.1.3	<i>Instruction Set Impact</i>	784.e27
21.2	Embench	784.e27
21.2.1	<i>Porting Embench</i>	784.e34
21.2.2	<i>Embench Flags</i>	784.e35
21.2.3	<i>Instruction Set Impact</i>	784.e36
21.2.4	<i>Troubleshooting</i>	784.e39
21.3	Performance Verification	784.e39
21.3.1	<i>Cache Performance Verification</i>	784.e40
21.3.2	<i>Branch Predictor Performance Verification</i> ...	784.e41
21.4	Summary	784.e47
Chapter 22 Linux		784.e48
22.1	Principles	784.e48
22.1.1	<i>Embedded Linux Hardware Requirements</i> ...	784.e50
22.1.2	<i>Bootloader</i>	784.e50
22.1.3	<i>Device Tree</i>	784.e50
22.1.4	<i>Kernel</i>	784.e51
22.1.5	<i>Root Filesystem</i>	784.e51
22.1.6	<i>Init</i>	784.e52
22.1.7	<i>Shell</i>	784.e52
22.2	Buildroot Linux on Wally	784.e52
22.2.1	<i>Building Buildroot Linux</i>	784.e53
22.2.2	<i>Source, Binary, and Disassembled Files</i>	784.e53
22.2.3	<i>Linux Memory Map</i>	784.e56

22.2.4	<i>Linux Boot Overview</i>	784.e57
22.2.5	<i>Zero Stage Bootloader</i>	784.e57
22.2.6	<i>OpenSBI Firmware</i>	784.e60
22.2.7	<i>Kernel</i>	784.e63
22.2.8	<i>Root File System (rootfs)</i>	784.e65
22.2.9	<i>Device Tree</i>	784.e66
22.2.10	<i>Init Process</i>	784.e66
22.2.11	<i>Trap Handlers</i>	784.e68
22.3	Booting Linux	784.e70
22.3.1	<i>Booting Linux on QEMU</i>	784.e70
22.3.2	<i>Booting Linux on Wally using a Logic Simulator</i>	784.e73
22.3.3	<i>Modifying Buildroot</i>	784.e75
22.4	Summary	784.e76
Chapter 23 FPGA Implementation		784.e78
23.1	FPGA Principles	784.e79
23.1.1	<i>FPGAs</i>	784.e79
23.2	Wally FPGA Implementation	784.e80
23.2.1	<i>SD Card</i>	784.e83
23.2.2	<i>Arty A7 Setup</i>	784.e84
23.2.3	<i>Supporting Other Boards</i>	784.e96
23.2.4	<i>Wally Zero Stage Bootloader</i>	784.e99
23.3	FPGA Results	784.e99
23.3	Interesting Bugs	784.e100
23.3.1	<i>Bug 1: AHB Bursts</i>	784.e101
23.3.2	<i>Bug 2: Concurrent Exceptions</i>	784.e102
23.3.3	<i>Critical Path Analysis</i>	784.e102
23.4	Summary	784.e105
Appendix A Wally Synopsis		784.e106
A.1	Wally Repository	784.e106
A.2	Wally Configurations	784.e106
A.3	Wally Regression and Test Suites	784.e109
A.4	Key Wally Commands	784.e109
A.5	Wally Diagrams	784.e109
Appendix B Hitchhiker's Guide to Linux		784.e153
B.1	Connecting to a Linux Server	784.e153
B.1.1	<i>X2Go</i>	784.e153
B.1.2	<i>Changing Your Password</i>	784.e154
B.2	Working With Files	784.e154

B.3	More Handy Commands	784.e154
B.3.1	<i>Cast of Characters</i>	784.e157
B.3.2	<i>Standard Input and Output</i>	784.e157
B.3.3	<i>Common Commands</i>	784.e158
B.3.4	<i>Working With Processes</i>	784.e159
B.3.5	<i>Compressing, Bundling, and Transferring Files</i>	784.e161
B.4	Linux Productivity	784.e161
B.4.1	<i>Multiple Terminal Windows</i>	784.e161
B.4.2	<i>Text Editors</i>	784.e162
B.4.3	<i>The Linux File System</i>	784.e163
B.4.4	<i>Environment Variables</i>	784.e164
B.4.5	<i>Web Browsing</i>	784.e164
B.4.6	<i>Document Viewing and Printing</i>	784.e165
B.4.7	<i>Installing Packages</i>	784.e165
B.5	Scripting and Programming	784.e165
B.5.1	<i>Shell Scripts</i>	784.e165
B.5.2	<i>Python</i>	784.e166
B.5.3	<i>C</i>	784.e167
B.5.4	<i>Makefiles</i>	784.e167

Appendix C Version Control Using Git 784.e170

C.1	Gitting Started	784.e171
C.2	Setting Up a Repository	784.e172
C.3	Basic Git Flow	784.e172
C.4	Snapshots and HEAD	784.e175
C.5	Merge Conflicts	784.e175
C.6	Branching and Merging	784.e176
C.7	Tags	784.e178
C.8	Staging and Undo	784.e179
C.9	Submodules	784.e180
C.10	Other Git Capabilities	784.e181
C.11	Summary	784.e182

Appendix D Tcl Book of Armaments 784.e185

D.1	Tcl Usage	784.e185
D.1.1	<i>Arguments and Variables in Tcl</i>	784.e186
D.1.2	<i>Nested Commands</i>	784.e187
D.1.3	<i>Summary of the Basic Tcl Command Syntax</i>	784.e187
D.2	Additional Commands	784.e188
D.2.1	<i>Lists</i>	784.e188
D.2.2	<i>Outputting Data to the Screen With Tcl</i>	784.e189
D.2.3	<i>File, Glob, and Other File-Related Commands</i>	784.e189
D.2.4	<i>Control Flow</i>	784.e189

D.2.5	<i>Procedures</i>	784.e191
D.2.6	<i>Running Tcl Scripts</i>	784.e191
D.2.7	<i>Aliases</i>	784.e192
D.2.8	<i>Exiting Tcl</i>	784.e192
Appendix F	Floating-Point Implementation	784.e193
F.1	Fused Multiply-Add	784.e193
F.1.1	<i>Multiply Significands</i>	784.e193
F.1.2	<i>Product and Addend Signs</i>	784.e193
F.1.3	<i>Add Exponents</i>	784.e193
F.1.4	<i>Alignment Shift</i>	784.e194
F.1.5	<i>Add Aligned Significands</i>	784.e195
F.1.6	<i>LZA: Leading Zero Anticipator</i>	784.e198
F.1.7	<i>Multipath FMA Optimization</i>	784.e200
F.2	Floating-Point Division and Square Root	784.e201
F.2.1	<i>Recurrence Division Algorithm</i>	784.e202
F.2.2	<i>Radix-2 Recurrence Division</i>	784.e207
F.2.3	<i>Radix-4 Recurrence Division</i>	784.e213
F.2.4	<i>Divider Latency</i>	784.e219
F.2.5	<i>Quotient Conversion</i>	784.e220
F.2.6	<i>Early Termination</i>	784.e222
F.2.7	<i>Subnormal Floating-Point Preprocessing</i>	784.e223
F.2.8	<i>Floating-Point Postprocessing</i>	784.e224
F.2.9	<i>Integer Division</i>	784.e225
F.2.10	<i>Square Root</i>	784.e234
F.2.11	<i>Unified Divide and Square Root</i>	784.e249
F.2.12	<i>Multiplicative Algorithms for Division and Square Root</i>	784.e257
F.3	Conversion	784.e258
F.3.1	<i>Result Exponent</i>	784.e258
F.3.2	<i>Shift Amount Selection</i>	784.e261
F.3.3	<i>Examples</i>	784.e262
F.4	Postprocessing	784.e263
F.4.1	<i>Shift Calculation</i>	784.e263
F.4.2	<i>Initial Result Mux</i>	784.e265
F.4.3	<i>Normalization Shift</i>	784.e265
F.4.4	<i>Correction Shift</i>	784.e265
F.4.5	<i>Sign Logic</i>	784.e267
F.4.6	<i>Rounding</i>	784.e268
F.4.7	<i>Special Cases and Flags</i>	784.e269
Bibliography		785
Index		799
RISC-V Instruction Set Summary		815

Preface

We wrote this textbook to close the gap in computer engineering education between principles of computer architecture and practices to build real microprocessors and systems-on-chip (SoCs). Many excellent books address computer architecture, operating systems, and digital design, but none that we know of connect these ideas together to reach an SoC capable of running an operating system (OS). Moreover, the open RISC-V instructions set architecture (ISA), pioneered at the University of California, Berkeley, has energized the computer architecture field by offering a modern instruction set suitable for hardware implementation in commercial systems, avoiding the intellectual property barriers of ARM, the historical baggage of x86, and the warts of earlier RISC architectures such as MIPS and SPARC. This book shows you how to apply contemporary professional practices to design, verify, and implement a RISC-V SoC with a processor core, memories, and peripherals to support its booting Linux.

This book is intended for a graduate or advanced undergraduate course or for self-study. It assumes that you have had introductory courses on digital design and computer architecture at the level of *Digital Design and Computer Architecture: RISC-V Edition* or *Computer Organization and Design: RISC-V Edition*. Specifically, you should be comfortable reading and writing C, assembly language, and Verilog and working in a Linux environment. You may get even more out of this book if you've taken a second course at the level of *Computer Architecture: A Quantitative Approach* or studied operating systems, compilers, field programmable gate arrays (FPGAs), or chip design.

While most textbooks offer simplified processor designs to illustrate general principles, this book takes the next step to explore optimized implementations of a complete processor. Our CORE-V Wally processor and SoC is part of the free and open-source OpenHW Foundation suite of RISC-V processors. Wally is described in SystemVerilog because it is an industry standard understood by most engineers and is supported by nearly all commercial simulation and synthesis tools. Wally is configurable in most regards, ranging from a tiny RV32E embedded core to a full-featured SoC, which is capable of booting Linux, with an RV64GC core, caches, virtual memory, and peripherals. This book delves into design issues such as bus interfaces, exceptions, privileged operations,

We named our RISC-V SoC Wally in homage to the Harvey Mudd College mascot Wally Wart, which personifies the flat, square protrusions that adorn many brutalist-style buildings at Harvey Mudd College, and also in honor of Pixar's adorable WALL-E robot.



division, and floating-point arithmetic that are glossed over in most texts. Although detailed implementation issues could become tedious, they are where most practicing engineers spend most of their time. We aim to give insight into good designs and the choices that lead to such designs.

The book applies professional tools and methodologies that students will see in industry. These include SystemVerilog for design, verification, and synthesis, Spike and QEMU for RISC-V functional simulation, Siemens Questa or Verilator for SystemVerilog simulation, Verilator for lint, Synopsys Design Compiler for chip implementation, and Xilinx Vivado for FPGA implementation. The system uses a Linux development environment, including GCC tools for programming and debugging in C and assembly language, Git for version control, and make-based scripts to drive compilation, simulation, and synthesis. The SoC itself also can boot Linux based on the embedded Buildroot flow, including the boot loader, kernel, device drivers, and command prompt. It has open-source implementations on an FPGA and on the Google Skywater 130-nm CMOS process design kit. All of the design files are open-source and available in a Git repository, although some commercial tools require licenses to run.

This book is divided into four parts:

- ▶ **Introduction:** The big picture, guided by an example of designing, verifying, and implementing a simplified single-cycle RISC-V processor
- ▶ **Tools:** C, Assembly, and SystemVerilog compilers, simulators, validation, synthesis
- ▶ **RISC-V Microarchitecture:** Pipelining, privileged operation, bus interface, caches, memory management, branch prediction, compressed instructions, multiplication and division, floating-point, atomic instructions, bit manipulation instructions, and other extensions
- ▶ **Implementation:** Peripherals, benchmarking, Linux boot, and FPGA implementation

This book took three years of development and another year of production. There are many more important topics in computer architecture and SoC design that were omitted from this edition for schedule reasons; we hope to address them in a future volume:

- ▶ **Multicore:** Multilevel cache coherency, synchronization, interprocessor communication
- ▶ **Microarchitecture:** Superscalar, out-of-order, deeper pipelines
- ▶ **RISC-V Extensions:** Vector, hypervisor, debug, trace
- ▶ **Verification:** Functional coverage metrics, hardware emulation

- ▶ **SoC Design:** Intellectual property block design and integration, AXI interfaces, accelerators, memory controllers, network-on-chip
- ▶ **SoC Implementation:** Timing and power optimization, clock gating, clock domain crossings, external interfaces, phase-locked loops (PLLs), verification, design-for-test, tapeout

Additional materials related to the book are available on the Companion Site: <https://www.elsevier.com/books-and-journals/bookcompanion/9780323994989>.

Please note that chapters 20–23 and Appendices A–D and F are available on the above Companion Site. These chapters and appendices are also in the electronic version of the book.

For qualified instructors, additional materials are available on the Instructor Site: <https://educate.elsevier.com/9780323994989>.

How to Use This Book

This book is most satisfying to use in conjunction with the CORE-V Wally processor source code, along with the compiler toolchain, simulation environment, and synthesis scripts. The complete code base can be cloned from github.com/openhwgroup/cvw. This repository also has documentation on how to install Wally and supporting tools.

The book comes with a set of lecture slides and labs that may be freely used or modified for non-commercial purposes as long as the source is credited. These supplements and an errata list are also available on the GitHub site.

OPEN SOURCE

All of the design files, including the Wally SoC, verification scripts, and implementation scripts, are provided open source under the Solderpad Hardware License version 2.1 (solderpad.org/licenses/SHL-2.1). We hope that others will enhance and extend the design and contribute to a vibrant open RISC-V ecosystem in collaboration with the OpenHW Foundation.

BUGS

Despite our best efforts, we are confident that this book and the Wally SoC still have bugs. Moreover, the processor relies on numerous external tools and repositories, all of which are likely to change and break things over time (a phenomenon called *bit rot*).

The textbook errata is on the GitHub site. We appreciate your corrections via a pull request to this page:

github.com/openhwgroup/cvw/blob/main/errata.md

Wally uses standard GitHub issues and pull requests for bug reports and fixes. We appreciate contributors.

FURTHER READING

This book has been influenced by many great works in computer architecture and systems. Hennessy and Patterson's *Computer Architecture: A Quantitative Approach* revolutionized the field when it was first introduced and remains the bible for principles of computer architecture. The *RISC-V Instruction Set Manual* by Waterman and Asanović is not just the authoritative definition of the RISC-V architecture but also an outstanding model of a specification that is succinct yet provides insight

into the factors influencing design decisions. The Berkeley CS152 *Computer Architecture* course has an excellent web page that gave us many ideas. SiFive's datasheets for various RISC-V chips define de-facto standard peripherals. Bryant and O'Hallaron's *Computer Systems: A Programmer's Perspective* has excellent coverage of the interaction between software and hardware. *Operating Systems: Three Easy Pieces* has particularly good coverage of concurrency and atomic operations. Ercegovac and Lang's *Digital Arithmetic* is the leading general book on computer arithmetic, and Muller's *Handbook of Floating-Point Arithmetic* has the most comprehensive coverage of floating-point hardware and software.

We learned a great deal writing *Digital Design and Computer Architecture, RISC-V Edition*, and the processor in this book is an outgrowth of the simplified processor in that introductory book. That book provides sufficient background for readers to understand this book. If you are interested in chip design and hardware implementations of arithmetic and memories, *CMOS VLSI Design* by Weste and Harris covers these topics in great detail.

Acknowledgements

The Wally RISC-V processor development has been a team effort with contributions from many students at Harvey Mudd College, Oklahoma State University, the University of Nevada, Las Vegas, and the University of Engineering and Technology, Lahore. The authors thank our students who contributed to the Wally processor and the chapters and to reviewing this book. Katherine Parry was the main architect of the floating-point unit. Cedar Turek, Alessandro Maiuolo, and Kevin Kim contributed to the divide/square root unit. Jordan Carlin and Jessica Liu contributed to quad and Zfa additional floating-point. Kelvin Tran and Ryan Swann developed the crypto unit. Noah Boorstin, Kaveh Pezeshki, Ben Bracker, and Skylar Litz brought up Linux on Wally and contributed to the Linux chapter. Ben also implemented and tested peripherals. Thomas Fleming and Jessica Torrey developed the memory management unit. Kip Mac-sai-Goren developed and debugged privileged tests, especially for the memory management and physical memory protection units. Kip and Kevin Kim developed the bit manipulation unit. Nic Lucio developed peripheral tests. Madeleine Masser-Frye and Shreya Sanghai optimized critical paths in Design Compiler synthesis. Daniel Torres, Elizabeth Hendenberg, and Abraham Casteneda ported the benchmarks. Daniel also brought up RISCOF tests. Jacob Pease contributed to the SDC peripheral and Linux boot. Naiche Whyte-Aguayo developed the SPI peripheral. Limnanthes Serafini did cache performance verification and parameterization. Teo Ene contributed and mentored in many areas. The HMC and OSU SoC Design classes of 2023 contributed coverage tests for their class projects, along with Harshini Srinath. Other students contributing to the project have included Jarred Allen, Ethan Falicov, Thomas Kidd, Tony Lei, Domenico Ottolia, and Udeema Shakya.

Jordan Carlin has made extraordinary contributions to this work in countless dimensions, especially related to the flow and verification, improving the manuscript, and leading teams. He has become the OpenHW Foundation's most prolific contributor and also contributes to the Sail model and other teams at RISC-V International. As one of the CORE-V Wally committers, his eagle eyes have improved hundreds of pull requests.

Hamza Ali, Marina Bellido, Kaden Cassidy, Noah Fotenos, Corey Hickson, Isabella Hottenrott, Hamza Jamal Vikram Krishna, Muhammad

Abdullah, Roman De Santos, Georgia Tai, Daniel Torres, and Muhammad Zain at HMC and UET made notable contributions to functional verification.

The OpenHW Foundation has welcomed CORE-V Wally to its robust family of open RISC-V cores. We especially appreciate the advice and support from Mike Thompson, Duncan Bees, Rick O'Connor, and Flo Wohlrab.

We have benefited from advice and support from many leaders of the RISC-V community, including Andrew Waterman, Krste Asanović, Andrea Gallo, David Patterson, Megan Lehn, Pawan Kumar, and Jeremy Bennett.

We have particularly enjoyed collaborating on verification with several teams, including Simon Davidmann, Lee Moore, Aimee Sutton, and Duncan Graham at Synopsys; Umer Shahid, Bilal Zafar, and Huda Sajjad at 10x Engineers; Adnan Hamid and David Kelf at Breker Verification Systems; and Allen Baum, Derek Hower, and James Ball with the RISC-V Certification Steering Committee.

Our work in microarchitecture has been influenced heavily by our experience designing microprocessors at Intel, Hewlett-Packard, Broadcom, Air Force Research Laboratory, and elsewhere.

We are indebted to our reviewers: Andrea Bartolini, Duncan Bees, Javier Bruguera, Daniel Chaver, Simon Davidmann, Ashish Dixit, Michael Engel, Andrea Gallo, Firas Hassan, Brian "Redbeard" Herrington, Andreas Koch, Bill McSpadden, Lee Moore, Simon Moore, Robert Mullins, Torbjørn Ness, John Nestor, Xiaoning Qi, Alex Solomatnikov, Dwight Sunada, Mike Thompson, and Peiyi Zhao.

About the Authors



David Harris is the Harvey S. Mudd Professor of Engineering Design at Harvey Mudd College. He received his Ph.D. from Stanford University and S.B. and M.Eng. degrees from MIT. His interests are in the area of microprocessor design, computer arithmetic, and high speed and low power circuits. David is the author of *Digital Design and Computer Architecture*, *CMOS VLSI Design*, *Logical Effort*, and *Skew-Tolerant Circuit Design*, and he holds sixteen patents in the field. He has designed chips at Intel, Broadcom, Sun Microsystems, Hewlett-Packard, and elsewhere. David's other interests include writing Southern California hiking guidebooks, building experimental aircraft, and traveling with his family.



James Stine is the Edward Joullian Professor in Engineering in the School of Electrical and Computer Engineering at Oklahoma State University. He received his Ph.D. in electrical engineering from Lehigh University. He was previously with the Illinois Institute of Technology in Chicago, IL. He specializes in research and teaching in VLSI, digital arithmetic, computer system architecture, and digital design. He is also a leading developer of commercial and academic system on chip design flows for Google, Cadence Design Systems, Synopsys, Mentor Graphics and several public-domain tools. James's interests include spending time with his family.



Rose Thompson is a processor architect at Skyworks Solutions, Inc. She received her Ph.D. in Electrical Engineering from Oklahoma State University and two B.S. degrees in Electrical Engineering and Computer Engineering from the University of Washington. She has also designed chips at the Air Force Research Laboratory. Her professional interests include SoC design and verification, custom instruction set architectures, branch prediction, memory systems, and secure computing. Rose also enjoys biking, hiking, rock climbing, and playing the piano.



Sarah Harris is a Professor of Electrical and Computer Engineering at the University of Nevada, Las Vegas. She received her M.S. and Ph.D. at Stanford University. Before joining UNLV in 2014, she was a faculty member at Harvey Mudd College for ten years. She has also worked at Hewlett Packard, NVIDIA, and the Technical University of Darmstadt. She is the co-author of three other textbooks including *Digital Design and Computer Architecture: RISC-V Edition*. Her research interests include computer architecture and applications of embedded systems and machine learning to biomedical engineering and robotics. Sarah also enjoys playing music and spending time with her family.

Foreword

I'm honored and rather humbled to have been invited to write the foreword for this *RISC-V System-on-Chip Design* textbook. But, how did we get here? How is it possible to have a truly open-source system-on-chip (SoC) design? How does this textbook help readers achieve this outcome?

Open-source history traces back to the early days of computing when researchers and developers shared code to propel innovation. In the 1960s and 1970s, the concept began to solidify with the emergence of time-sharing systems and the advent of the Unix operating system, which was distributed along with its source code. In the early 1980s, as proprietary software proliferated, the open-source movement took on a more structured approach with the founding of the Free Software Foundation by Richard Stallman in 1985. Stallman's GNU Project was pivotal because it sought to develop a completely free Unix-like operating system. The 1990s marked a fundamental turning point for open-source software with the development of the Linux kernel by Linus Torvalds in 1991. Linux, combined with GNU tools, provided a complete open-source operating system that gained rapid popularity among developers and enterprises. GitHub, launched in 2008, has also revolutionized open-source development by providing a platform for collaborative coding and distribution. In contemporary engineering, open source plays an integral role in cloud computing, artificial intelligence, and the Internet of Things (IoT).

The success of open-source software has inspired a revolution in open-source hardware, exemplified by the RISC-V architecture. Developed at Berkeley in 2010 by Professor Krste Asanović, his graduate students, including Andrew Waterman and Yunsup Lee, and Professor Dave Patterson as a fifth-generation reduced instruction set computing architecture, RISC-V has become the fastest-growing computer architecture in the industry. To foster open community-based collaboration and manage overall development of the RISC-V instruction set architecture (ISA) specifications, the project was spun out of the University of California, Berkeley, and the RISC-V Foundation was formed in 2015. Working with Krste, Dave and executives from top technology companies, including Google, NVIDIA, Western Digital, and many more, I had the privilege of leading the effort to incorporate the RISC-V Foundation nonprofit organization and serve as executive director for close to 5 years. The open nature of the RISC-V ISA specifications has had a profound impact on the semiconductor industry, introducing a new level of flexibility and innovation by allowing companies to tailor their processors

to specific applications without needing cumbersome licensing agreements. This textbook is an excellent example of the breadth of the RISC-V architecture's overall impact, with the majority of the CORE-V Wally SoC development flow and key building blocks being available via open source.

With the RISC-V ISA specifications having gained significant industry traction, the need for high-quality, well-written, SystemVerilog-based, open-source RISC-V cores became of interest to several key industry players. This need drove the creation of the OpenHW Foundation, which was established in 2019 as a not-for-profit global organization aimed at fostering the collaborative development of high-quality, production-worthy, open-source hardware designs, particularly RISC-V processors. After my tenure running the RISC-V Foundation, I was asked to spearhead the creation of the OpenHW Foundation, and together with industry leaders from companies such as Alibaba, Bluespec, Imperas, NXP, SiLabs, and Thales, the OpenHW Foundation was formed with initial CORE-V Family projects based on the excellent work of Professor Luca Benini, Frank Gürkaynak, and the rest of the PULP Platform team at ETH Zurich. The CORE-V Family has grown to include 10 different cores, including CORE-V Wally (cvw), of varying sizes and capabilities for use in application and embedded class systems. These CORE-V cores have been instrumental in demonstrating the capabilities and benefits of open-source hardware collaboration. Hundreds of millions of these cores have been sold commercially in SoCs across the industry. By leveraging the collective expertise of its community, OpenHW has contributed to the wider acceptance and implementation of RISC-V cores in various applications, from automotive to IoT and embedded systems.

This textbook is an excellent way for readers to become immersed in the vast area of open-source, the RISC-V ISA, and processor SoC design methodologies. The book is best used in conjunction with the CORE-V Wally processor source code and related tools, available in the OpenHW CORE-V Wally GitHub repository. To get started, readers are guided through an example of designing, verifying, and implementing a simplified single-cycle RISC-V processor before diving into the five-stage, 64-bit, Linux-capable Wally processor.

This textbook provides comprehensive coverage of both open-source and commercial compilers, simulators, and synthesis tools, including SystemVerilog, which is the de facto SoC design language used in the semiconductor industry. The authors have taken a practical approach to using both open-source tools such as GCC, Verilator, Spike, Sail, and Quick Emulator (QEMU) alongside tools from commercial electronic design automation companies, including Synopsys, Siemens, and Cadence. This open-source commercial tool pragmatic approach is required to achieve SoC designs suitable for deep submicron implementations and

reflects the reality of the design environment graduating engineers will be working in after completing their studies.

The RISC-V microarchitecture section of the textbook provides fine-grain detail regarding processor design considerations, including pipelining, buses, caches, and memory as well as the various RISC-V instruction extensions. This is all brought together through a “soft-physical” implementation using a field programmable gate array, which demonstrates the use of SoC peripherals, benchmarking, and booting Linux. The deep dive the authors take into processor design topics, such as buses, privileged modes, virtual memory, and floating-point, which are often glossed over in other engineering textbooks, provides excellent learning for students because these are the topics with which most practicing engineers spend a majority of their time. Some 4 years in the making, this *RISC-V System-on-Chip Design* textbook is destined to become a “must-have” tome covering processor and SoC design for years to come.

Good luck with your open-source RISC-V development journey!

Rick O'Connor

Founding OpenHW Foundation CEO

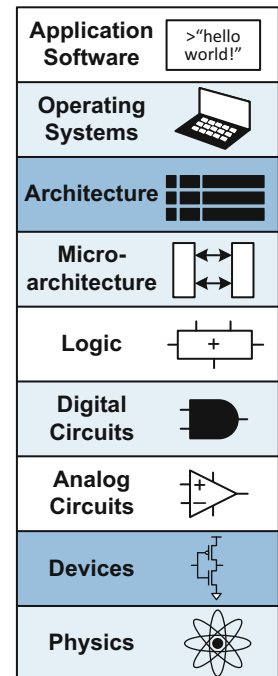
Founding RISC-V Foundation Executive Director

A Brief History of Computer Design

Computers are among the most significant inventions in the history of technology. Their cost, size, power consumption, and performance have improved exponentially, with greater impact and for a longer duration than any other human invention. A microprocessor, the brain of a computer, is smaller than a jellybean. A simple microprocessor can be manufactured on a chip for one-hundredth of a cent. Microprocessors are built from electrically controlled switches called transistors. Decades of miniaturization have shrunk transistors such that more than ten billion can be packed onto a square-centimeter chip. The transistors can switch on or off in a few picoseconds, consuming less than a femtojoule of energy each time. A *system-on-chip* (SoC) integrates multiple microprocessors and memories with various communication links, peripherals, and accelerators, enabling low-cost mobile and wearable devices. These astonishing hardware advances have transformed nearly every aspect of society in the past half century, including medicine, communication, transportation, consumer products, energy, warfare, education, and work. Computers have put much of humanity's collective knowledge at our fingertips while upending how we interact in ways with which society is still grappling.

Meanwhile, the free and open-source software movement has been transforming software by sharing source code with a license that allows others to freely use and improve it. For example, open-source software and standards such as Linux, TCP/IP, Apache HTTPD, and Python underlie much of the operation of the Internet. The movement has historical roots, such as the Automobile Manufacturers Association openly sharing automotive innovations in the first half of the 20th century and IBM's release of its operating system (OS) code in the 1950s and 1960s. Richard Stallman created the Free Software Foundation in 1985, and the Open Source Initiative began in 1998. However, computer architectures have historically been proprietary, protected by patents and licensing agreements that limit adoption and extract royalties from manufacturers. To advance a widely adopted open-source architecture, a team at Berkeley developed a fifth-generation reduced instruction set computer (RISC)

- 1.1 **Moore's Law and Beyond**
- 1.2 **System-on-Chip**
- 1.3 **Birth of Computing**
- 1.4 **Mainframes and Minicomputers**
- 1.5 **Microprocessors**
- 1.6 **CISC and RISC**
- 1.7 **RISC-V**
- 1.8 **International Economic and Security Competition**
- 1.9 **Summary**



RISC-V, pronounced “risk five,” is the fifth RISC architecture developed at Berkeley (the University of California, Berkeley), with their first RISC designs being developed in the 1980s.

architecture called *RISC-V* in 2010 and established the nonprofit RISC-V Foundation (now called RISC-V International) as a collaborative community. The nonproprietary RISC-V architecture facilitates innovation, and RISC-V has become the fastest-growing computer architecture, drawing enormous attention from both industry and academia.

Most computer engineering students take courses in computer architecture, and many have the opportunity to build toy processors, but very few get to build a real full-featured microprocessor. This book takes you step by step through the design of a complete open-source RISC-V processor and SoC using contemporary design tools and methods. By the end of this book, you should have a solid understanding of computer architecture theory and practice and be ready to make your own contributions to the vibrant field of processor and SoC design.

Advances in transistors, computer hardware, and computer software have become intertwined in a *virtuous cycle* in which improvements in one lead to improvements in the others. This chapter sets the context for the book with a brief history of transistors and computer design, with emphasis on how advances in implementation technology have driven architecture and microarchitecture. The next chapter introduces the RISC-V architecture and previews the rest of the book.

1.1 MOORE’S LAW AND BEYOND

The semiconductor revolution began in 1947 when John Bardeen, Walter Brattain, and William Shockley invented the first working point-contact transistor at Bell Lab. *Transistors* are semiconductor devices with a control terminal that influences the flow of current between two other terminals. They operate as amplifiers or electrically controlled switches and are a key building block of digital logic. In 1958, Jack Kilby and Robert Noyce coined the first *integrated circuits* by placing multiple transistors on a single substrate. Fig. 1.1 shows the first planar integrated circuit prototype built by Noyce and his team at Fairchild in 1960. Bardeen, Brattain, Shockley, and Kilby all received Nobel Prizes in physics for their world-changing contributions.

An integrated circuit is now manufactured on a large circular silicon wafer using a series of deposition, masking, and etching steps to create transistors and interconnecting wires. *Photolithography* (Greek for *light-stone-writing*) is a key manufacturing step in which a photoresist is exposed to light through a mask to define the shapes of transistors and wires. As manufacturing improvements allow for smaller transistors, they become faster (less distance for electrons to flow), lower power (fewer electrons flowing), and cheaper (more fit on a wafer). The wafer is cut up, or *diced*, into rectangles called *dice* or *chips*, which are tested and packaged.

In 1965, Gordon Moore observed that the number of transistors in an integrated circuit was growing exponentially. This observation, now

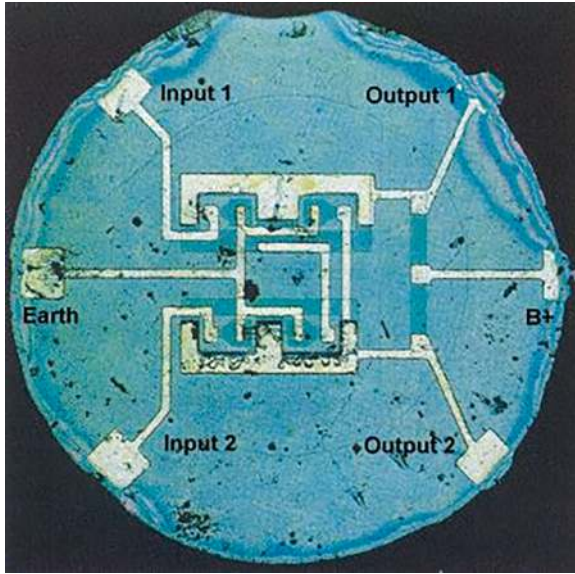
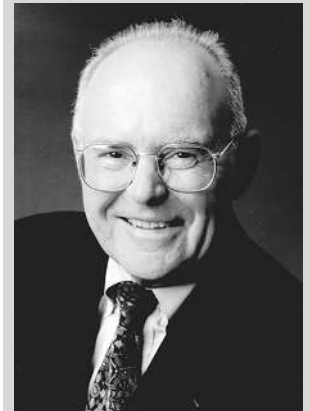


Fig. 1.1 First planar integrated circuit. (ON Semiconductor Corp., 1961 [onsemi61]. Reprinted with permission.)

known as *Moore's Law*, portended technical advances unrivaled by any other human achievement [Moore65, Moore03]. Since its inception, the semiconductor industry has increased the number of transistors on a chip by a ludicrous 11 orders of magnitude. Fig. 1.2 shows how microprocessor transistor counts have risen at a remarkably steady 40% compound annual growth rate for five decades, doubling almost every two years. In 2024 a cell phone chip such as the Apple A17 Pro had approximately 19 billion transistors on a 1 square centimeter chip for an estimated \$40 manufacturing cost. In other words, the cost of a transistor is now about 2 nanodollars! However, no exponential curve can continue indefinitely. As shown in Table 1.1, Moore's Law has been slowing in recent years because of the rising cost and technical difficulty of manufacturing atomic-scale transistors far smaller than the wavelength of visible light. Only a handful of companies worldwide can invest the tens of billions of dollars necessary to develop cutting-edge manufacturing processes and factories, so the logic fabrication industry has largely consolidated to TSMC, Samsung, UMC, Global Foundries, SMIC, and Intel. Despite the small number of companies involved, global semiconductor sales rose from \$33 billion in 1987 to approaching \$700 billion in 2025, as shown in Fig. 1.3.

Gordon Moore (1929–2023) was a cofounder and CEO emeritus of Intel Corporation. He is best known for articulating Moore's Law, which drove the advancement of semiconductors for decades. Moore grew up in northern California and then attended San Jose State, Berkeley, and Caltech, receiving a PhD in chemistry. He cofounded Fairchild Semiconductor and then Intel. After retiring, Moore and his wife became philanthropists supporting environmental conservation and scientific research. Moore was a passionate fisherman, which inspired his conservation efforts.



(© 2006. Image courtesy of Intel Corporation. Reprinted with permission.)

Moore’s law: The number of transistors on microchips doubles every two years

Our World
in Data

Moore’s law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing - such as processing speed or the price of computers.

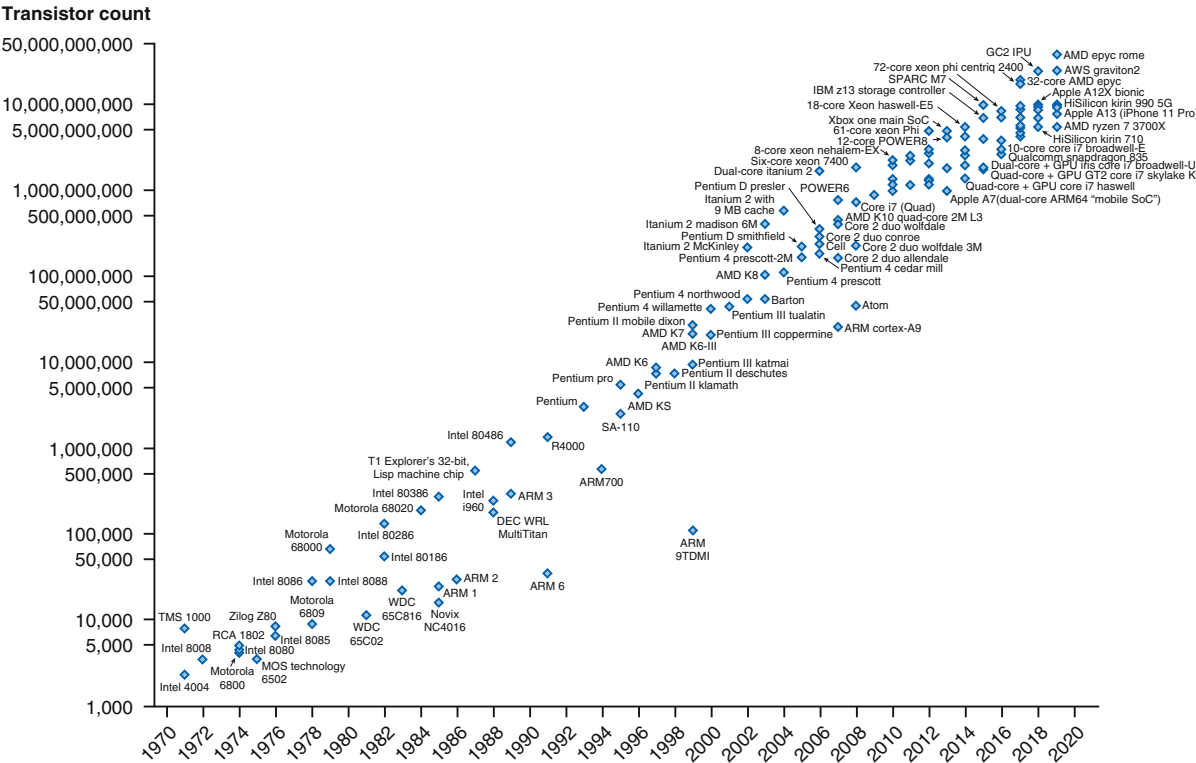


Fig. 1.2 Microprocessor transistor counts. (© 2020 Hannah Ritchie and Max Roser [Ritchie20]. This figure is openly licensed under CC BY 4.0.)

Table 1.1 Chip and transistor costs. [Zafar19]

Process Node	16 nm	10 nm	7 nm	5 nm	3 nm
Initial production	2014	2016	2018	2020	2022
Chip area (mm ²)	125	88	83	85	85
Billions of transistors	3.3	4.3	6.9	10.5	14.1
Gross dice / wafer	478	686	721	707	707
Net dice / wafer	360	512	546	530	510
Wafer cost (\$)	5912	8389	9965	12500	15500
Die cost (\$)	16.43	16.37	18.26	23.57	30.45
Transistor cost (nanobucks)	4.98	3.81	2.65	2.25	2.16

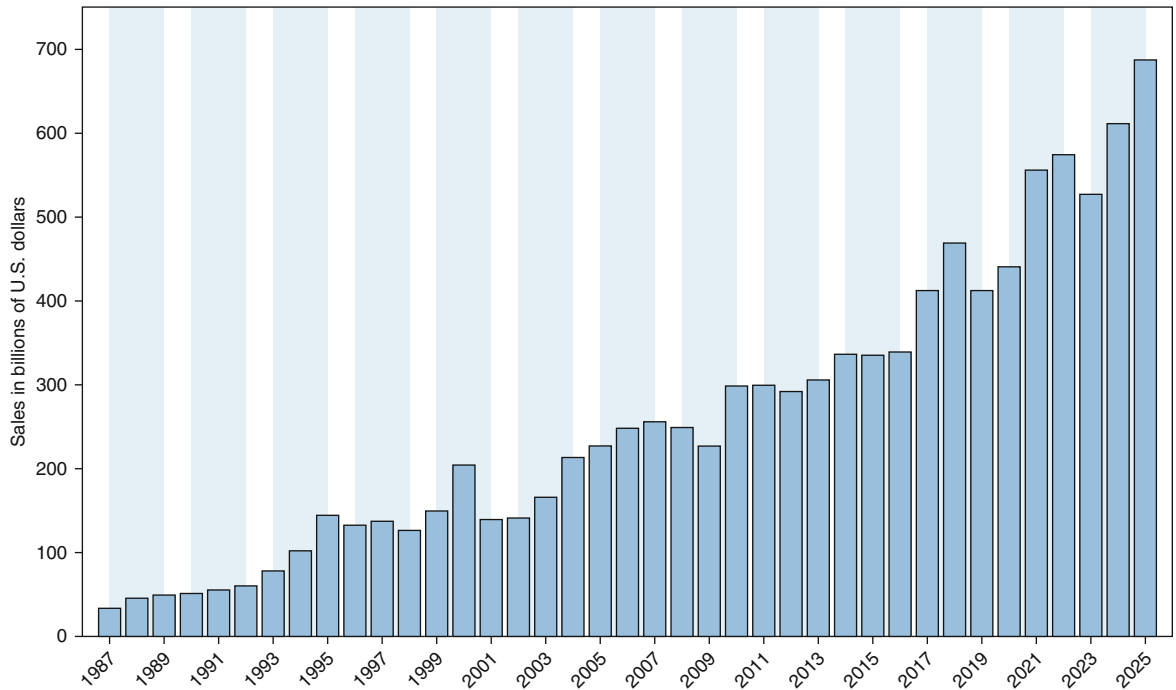


Fig. 1.3 Global semiconductor sales. [market25]

A corollary of Moore's Law is *Dennard's Law*, which describes how transistor speed and power consumption improve as transistors shrink [Dennard74, Bohr07]. Dennard scaling predicted that if a new process reduces transistor dimensions, supply voltage, and threshold voltage by 30%, then a chip of the same size can pack twice as many transistors that operate at 1.4 times the speed and half the power per transistor. Each such scaling is called a *process node*. Process nodes have historically been identified by their *feature size*, the gate length of a minimum-sized transistor. For example, a 180-nm process uses masks and photolithography to define a transistor with a channel length of 180 nm from source to drain. Process nodes from the mid-1990s to 2013 advanced from 180 nm to 130 nm, 90 nm, 65 nm, 45 nm, 32 nm, and 22 nm. To reduce leakage current at 22 nm or below, transistors became three-dimensional structures with gates wrapping around the channel. Now the process node no longer correlates with any physical feature, but manufacturers still have named their improved nodes 16 nm, 10 nm, 7 nm, 5 nm, 3 nm, 2 nm, and so on.

Dennard scaling tapered off around 2004 because continuing to reduce threshold voltage causes subthreshold leakage current to rise exponentially, and reducing gate oxide thickness below about 12 angstroms (4 atomic layers) allows large numbers of electrons to quantum-mechanically tunnel

through the insulating gate, resulting in unacceptable gate leakage current. Since then, transistor power and performance have gradually improved because of device innovations such as *strained silicon*, *high-k metal gates*, *FinFETs*, and *gate-all-around transistors* as well as dimensional scaling.

1.2 SYSTEM-ON-CHIP

The incredible progress in miniaturization drives chip designs, especially processors. A computer's *instruction set architecture (ISA)*, or simply *architecture*, is the programmer's view of a computer, including the instructions, registers, and memory accessing modes. If two computers have the same ISA and supporting capabilities, a program that runs on one will also run on the other. Such compatibility allows customers to upgrade to a new computer that runs their existing programs better. A computer's *microarchitecture* is the interconnection of registers, memories, and execution units that implement the architecture. As more transistors become available, computers can adopt microarchitectures with larger data sizes and more parallelism to run a program faster. Thus, processors have grown from operating on 8- to 16- to 32- to 64-bit values, from using microcoded to pipelined to superscalar to out-of-order microarchitectures, and from having a single core to using multiple cores.

Chips are sometimes categorized as *general purpose* (e.g., microprocessors, graphics processing units, or field-programmable gate arrays) versus *application-specific integrated circuits (ASICs)* (e.g., hard disk read channel controllers, Wi-Fi modems, Bitcoin mining engines). The distinction has blurred as ASICs incorporate more programmable components. Modern integrated circuits are often called *systems-on-chip (SoCs)* because they combine many processors, memories, accelerators, busses, and other components. For example, a contemporary cell phone chip such as the Apple A15 in Fig. 1.4 contains at least two high-performance and four energy-efficient application processors, tens of megabytes of level 2 and 3 caches, many graphics processing units and neural engine cores, video and image signal processing accelerators, and interfaces to a display and external memory. The components are clocked at several gigahertz (GHz) and collectively perform tens of trillions of math operations per second. Keeping all the components running simultaneously would overheat the chip, so only a portion of the chip operates at any given time; the remainder is known as *dark silicon*. The specialized *accelerators* are 2 to 3 orders of magnitude more power efficient than general-purpose processors [Taylor12], so tasks are offloaded to these accelerators when practical. The system is so complex that a small onboard microprocessor is used to manage power and clocks to all the other components.

The increasing number of transistors on a chip has led to rising development costs, as shown in Fig. 1.5. Chip design teams have grown from one or two people in the 1970s to hundreds of people spread across

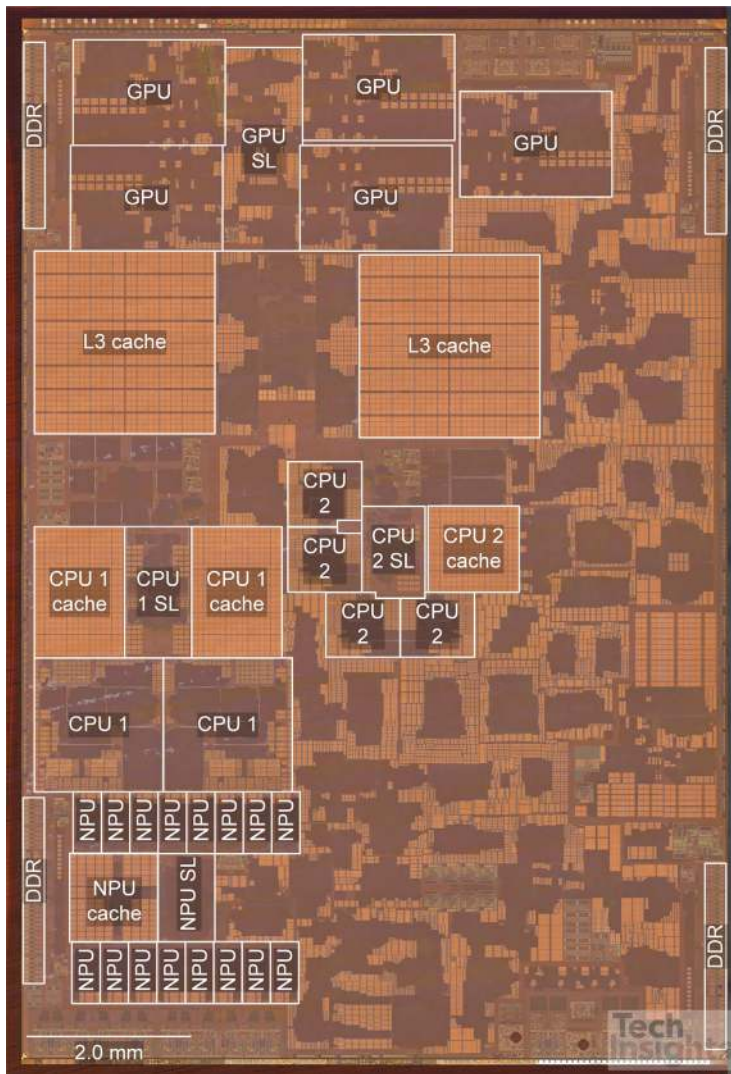


Fig. 1.4 Apple A15 Bionic die photo. (Image courtesy of TechInsights Inc. [iphone13]. Reprinted with permission.)

multiple divisions and companies for a modern SoC. Nevertheless, the number of transistors per designer has increased by five orders of magnitude, meaning engineers must be far more productive. In the 1990s, designers moved from transistor-level schematic design to *hardware description languages (HDLs)* such as Verilog or VHDL. An HDL allows a designer to specify logic functions using textual descriptions of its behavior or structure, thus increasing the level of abstraction and reducing the time to specify a function. Electronic design automation (EDA) tools

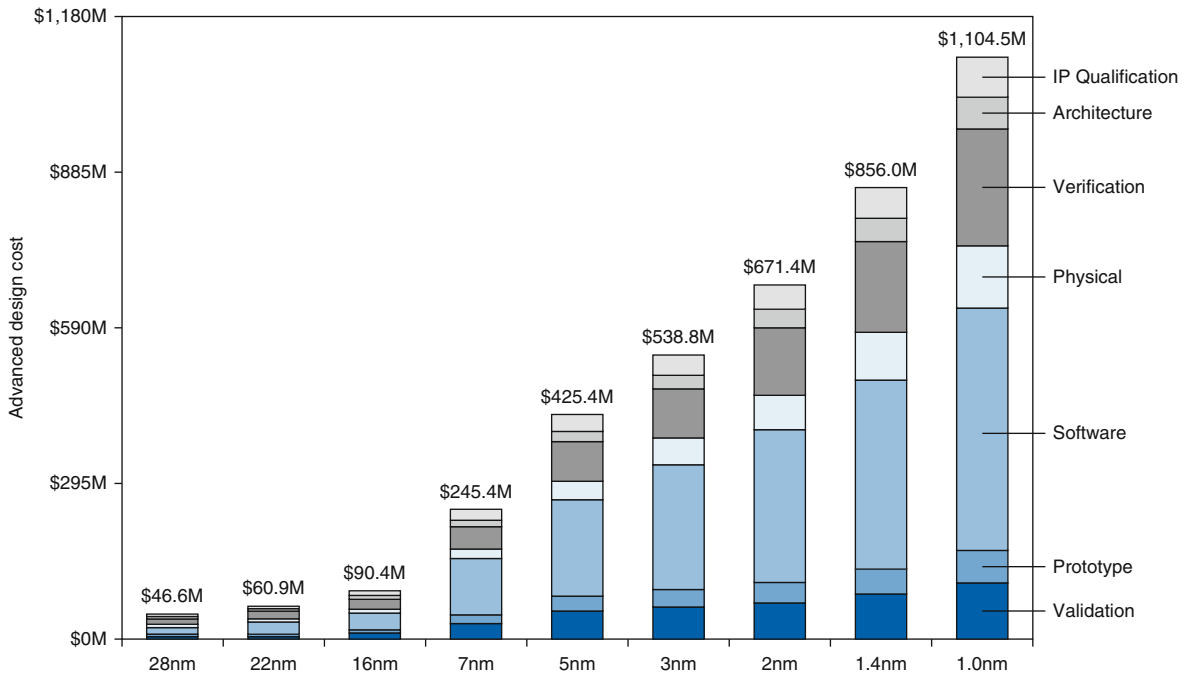


Fig. 1.5 SoC design cost. (Figure courtesy of International Business Strategies, Inc., September 2024. Reprinted with permission.)

such as Synopsys Design Compiler convert these HDL designs into gates and transistors, which is steadily improving productivity and quality of results. The industry has moved toward *design reuse*, in which *intellectual property (IP)* blocks such as processors, memories, accelerators, peripherals, and standard cell libraries are reused from a previous product or purchased from a third party. Designs have become so complex that verifying their correctness is more time-consuming and expensive than doing the design; IP blocks that come with robust test frameworks help reduce verification costs too. Despite all these advances, the cost of a new SoC design continues to rise exponentially. New SoCs in cutting-edge processes must target high-volume multi-billion-dollar markets, and the chip design industry has consolidated to a smaller number of companies (notably Intel, Samsung, Apple, Qualcomm, Broadcom, AMD, Texas Instruments, NVIDIA, MediaTek, STMicroelectronics, Infineon, NXP, Google, Meta, Amazon, Microsoft, and Renesas) capable of making these huge investments. The industry has learned that software is harder than hardware, and now the largest cost and source of errors in an SoC is the software running on it.

Especially because of these rising software development costs, the open-source movement has proven critical to the computer industry [opensource.org]. The Linux OS is the most prominent example; this robust kernel is the most widely used in the world, running on billions of processors from Android cell phones to warehouse-scale compute farms

[Usage22, Moody01]. More than 4,000 people contribute to the Linux code base each year, with the largest contributions coming from highly competitive for-profit companies such as Intel and RedHat [Larabel19]. Closely associated are many other open-source software tools such as the GNU Compiler Collection, Python, and PHP. Some open-source developers are driven by altruism, idealism, and curiosity, but others are capitalists striving to maximize profit. The computer industry has discovered that open-source software offers competitive advantages of flexibility, interoperability, security, and collaboration that often outweigh the benefits of keeping code proprietary. These open-source programs, such as the Android OS, run on many SoCs.

Open-source software has inspired an open-source hardware movement [oshwa.org]. Processor architectures, such as the Intel and ARM architectures, have historically been proprietary. However, RISC-V is a free and open computer architecture that has rapidly gained traction in the semiconductor industry. Chapter 2 describes the RISC-V architecture in detail.

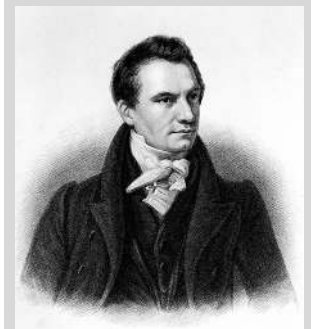
1.3 BIRTH OF COMPUTING

Fig. 1.6 lays out some notable milestones in the development of computing.

Charles Babbage conceived the first digital computer in the 1830s. An English mathematician and Fellow of the Royal Society, Babbage was frustrated by the countless errors found in tables of mathematical functions in his day, finally exclaiming, “I wish to God these calculations had been executed by steam” [Eves69]. From 1823 until 1833, he designed the Difference Engine, an intricate array of 10-toothed gears that could rotate to perform additions, multiplications, and finite difference computations for tables of logarithms and trigonometric functions. The machine proved too difficult for Babbage and his machinists to build. Babbage eventually abandoned his efforts and moved on to the more general Analytical Engine with programmable instructions and a 1000-element memory that anticipated modern computers (Fig. 1.7). Running out of funding to complete the machine, Babbage died a broke and disappointed man in 1871. In 1991, engineers built two full-size operational Difference Engines from Babbage’s plans. The 5-ton machines stand 11 feet long and 7 feet high. One of them can be seen at the Science Museum in London.

By World War II, militaries had urgent need for computation, such as ballistic firing tables needed to aim battleship turrets at targets 25 miles away. The first *computers* were people, usually young women, working in large teams to calculate and check these tables. Researchers strove to develop electromechanical computers using relays, electrically controlled switches, to automate such calculations. German scientist Konrad Zuse independently reinvented the notion of programmable computers and realized that binary numbers were most suitable because a switch can be in one

Charles Babbage (1791–1871) attended Cambridge University and married Georgiana Whitmore in 1814. He invented the Analytical Engine, the world’s first mechanical computer. He also invented the cowcatcher and the universal postage rate and was interested in lock picking but abhorred street musicians. He was elected to the Royal Society in 1816 and served as the Lucasian chair of mathematics at Cambridge from 1828 to 1839.



(© 1833 Wellcome Collection, by R. Roffe [Babbage]. This image is openly licensed under CC BY 4.0.)

1830-1871	Analytical Engine	First digital computer (<i>Charles Babbage</i>)
1941	Z3	First general-purpose computer (<i>Conrad Zuse</i>)
1942	ABC	First electronic computer (<i>Atanasoff, Berry</i>)
1943	Colossus	First programmable computer (<i>Bletchley Park</i>)
1944	Mark I	Separate program and data (<i>Harvard</i>)
1945	ENIAC	Digital electronic, gp & programmable (<i>U. Penn</i>)
1948	Baby	First to have a stored program (<i>Manchester U.</i>)
1949	EDSAC	First practical stored program (<i>Cambridge U.</i>)
1951	EDVAC	Follow on to ENIAC, added a stored program (<i>U. Penn</i>)
1961	Stretch	First transistor-based computer (<i>IBM</i>)
1964	System/360	First commercial success (<i>IBM</i>)
1970	PDP-11	Leading minicomputer (<i>DEC: Digital Equip. Corp.</i>)
1971	4004	First 4-bit microprocessor, on a single chip (<i>Intel</i>)
1977	Apple II	Low-cost 6502 makes computers widely available
1981	IBM PC	Start of the PC revolution based on Intel's 8088
1982-1986	RISC	RISC-I (<i>Berkeley</i>), MIPS I (<i>Stanford</i>), ARM I, SPARC
1992	Alpha 21064	200 MHz RISC processor 3x speed of Pentium (<i>DEC</i>)
1993	Pentium	66 MHz CISC: RISC wins RISC vs. CISC debate (<i>Intel</i>)
1994-2006	MHz Wars	Super-pipelining, high power
2006	Core	Multi-core (<i>Intel</i>)
2010	RISC-V	First popular open-source architecture (<i>Berkeley</i>)

Fig. 1.6 Timeline of computer design.

of two states. His Z3 computer, built in Berlin in 1941, was the first functional general-purpose computer, but it was destroyed in a bombing raid. Meanwhile, Howard Aiken at Harvard, in collaboration with IBM, built the 5-ton Mark I (Fig. 1.8) in 1944 [Aiken46], and George Stibitz at Bell Telephone Labs built the nonprogrammable Complex Number Calculator in 1940.

Relays were both slow and unreliable. John Atanasoff at Iowa State College pioneered the application of vacuum tubes to perform digital calculations from 1937 to 1942. He and his student Clifford Berry built the

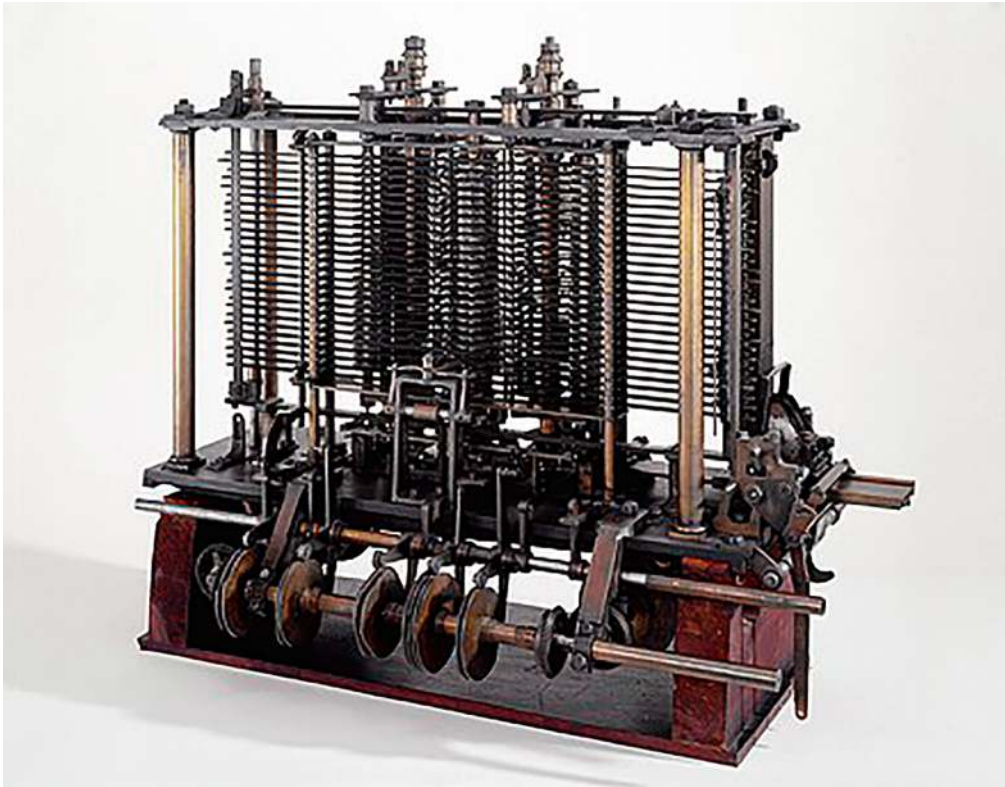


Fig. 1.7 Babbage's Analytical Engine, under construction at the time of his death in 1871. (This image is openly licensed under CC BY-SA 2.0.)

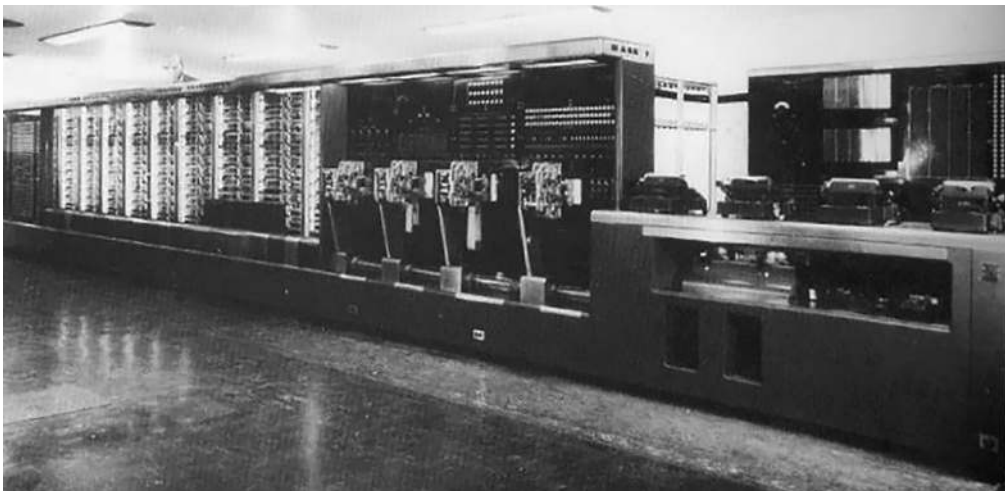


Fig. 1.8 Harvard Mark I. (Image courtesy of Harvard University [MarkI]. Reprinted with permission.)

Vacuum tubes evolved from incandescent lightbulbs and look similar, with a vacuum-sealed glass container and electrodes. Like lightbulbs, vacuum tubes allowed current to flow in only one direction, which enabled the flow of current to be electrically controlled. The ENIAC used thousands of 6SN7 dual triode vacuum tubes for flip-flops, each 4 inches tall and dissipating up to 5 W. On average, one ENIAC tube burned out every two days.



6SN7 vacuum tube. (Vostok, 1980 [6SN7]. Public Domain: CC0 1.0 Universal.)

Grace Hopper found the first actual computer bug in the Harvard Mark II in 1947, a dead moth shorting one of the relays, as shown in Fig. 1.10.

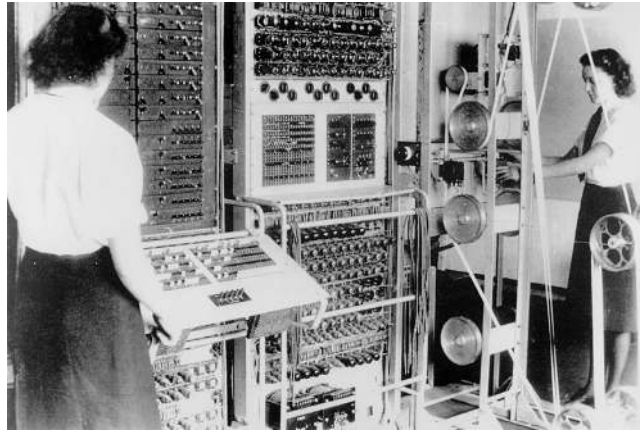


Fig. 1.9 Dorothy Du Boisson and Elsie Booker operating the Colossus in 1943. (Image courtesy of the National Archives (UK), 1943 [colossus]. Public Domain.)

first special-purpose digital electronic computer in 1942, the ABC (Atanasoff-Berry Computer) before getting called to serve in the war. In 1943, British codebreakers at Bletchley Park built the Colossus, the first programmable digital electronic computer, to decrypt German communication, arguably turning the tide of the war (Fig. 1.9). It used 1600 vacuum tubes and was programmed with plugs and switches rather than an internal stored program. Colossus was classified, so its significance was not widely recognized until much later. Another influential vacuum tube computer was the Electronic Numerical Integrator and Computer (ENIAC), built in 1945 by John Mauchly and J. Presper Eckert at the University of Pennsylvania. It ran at 5 kHz, weighed 30 tons, and consumed 174 kW. In collaboration with distinguished mathematician John von Neumann, they proposed a follow-on with a stored program called the Electronic Discrete Variable Automatic Computer (EDVAC) [vonNeumann45], but it did not begin operation until 1951. The Manchester University “Baby” computer in 1948 was the first to have a stored program. It was built by F.C. Williams and Tom Kilburn with influences from Alan Turing and Max Newman. The Electronic Delay Storage Automatic Calculator (EDSAC), developed by Maurice Wilkes at Cambridge University in 1949, is recognized as the first stored-program computer to go into regular use.

Like lightbulbs, vacuum tubes are prone to burning out over time, so keeping a computer with thousands of tubes running for an extended period was difficult. William Shockley, John Bardeen, and Walter Brattain invented the transistor in 1948 at Bell Labs (another Nobel Prize-winning discovery), and transistors were soon applied to digital computers. Gene Amdahl designed IBM’s first transistorized computer, the 33-foot long 21-kW IBM 7030 “Stretch” behemoth, with 150,000 germanium bipolar transistors, that was the world’s fastest computer in 1961. It was used by

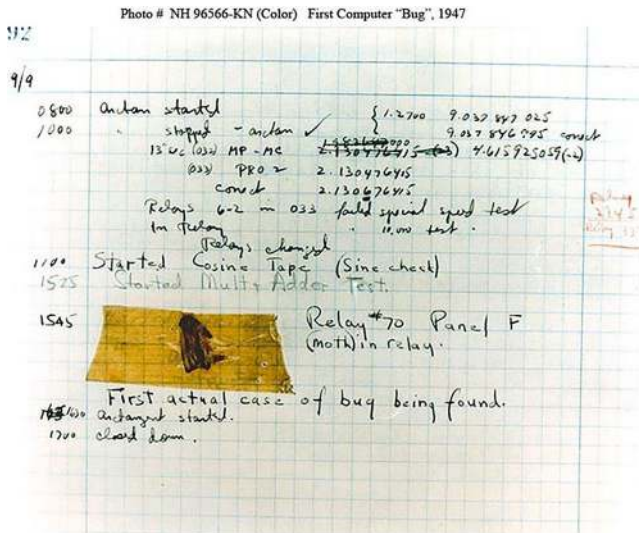


Fig. 1.10 First computer bug. (Image courtesy of the Naval Surface Warfare Center, 1945 [bug]. Public Domain.)

national labs in the United States for atomic weapons simulations. Although it was commercially disappointing, Stretch pioneered pipelining, multiprogramming, memory protection, and generalized interrupts that were later used on the System/360 computers.

1.4 MAINFRAMES AND MINICOMPUTERS

By the 1960s, large-scale *mainframe computers* became the focus of commercial computing. The name comes from the large cabinet, called the main frame, that holds the processor and memory. They initially operated in *batch* mode, operating on programs and data provided on punched cards or magnetic tape for applications such as census or payroll processing. Later, *timesharing* systems supported multiple users with interactive terminals, bringing applications such as airline booking and software development. Mainframes were so costly that they were mostly under centralized control of large organizations. Smaller *minicomputers* emerged in the 1960s and 1970s, typically costing tens of thousands of dollars, a price point that smaller organizations and universities could afford.

IBM bet its future on the System/360 series of computers in the 1960s (Fig. 1.11) with a common ISA shared by many microarchitectural implementations with different price/performance tradeoffs. IBM invested an astonishing \$5 billion in development (half as much as the US spent on the Manhattan Project to develop the atomic bomb). Production delays nearly bankrupted the company, but the problems were resolved, and the wildly successful product line set the standard for *mainframe*

John von Neumann (1903–1957) was a mathematician and a founding figure in computing. Born in Budapest Hungary, von Neumann was a child prodigy. By 8 years of age, he was learning calculus and could speak six languages. He received his PhD in mathematics and soon joined the faculty at Princeton University's Institute for Advanced Study. Von Neumann made numerous fundamental contributions to pure and applied mathematics, established a mathematical framework for quantum mechanics, founded the field of game theory, and contributed to the development of the nuclear and hydrogen bombs and the notion of Mutual Assured Destruction. He is one of the founders of the field of computer architecture, publishing the idea of a stored program architecture. He died of cancer at 53 years of age, likely because of his work on the Manhattan Project.



(© 2008 Los Alamos National Laboratory [vonNeumann]. Reprinted with permission.)

Alan Turing (1912–1954) was a British mathematician and a founder of the fields of computer science and artificial intelligence (AI). Born in London, he attended the University of Cambridge and then received a PhD in mathematics from Princeton University in 1938. Turing developed the notion of a universal computing machine capable of carrying out any algorithm, now known as the Turing machine. He published a seminal proof that not all problems are computable. Turing returned to Britain to serve as a cryptanalyst during World War II and broke the German's Enigma Code. Historian Harry Hinsley estimated that the cryptanalysts may have shortened the war by two years and saved 14 million lives. Turing contributed to several early computer designs, wrote a program in 1950 to play chess, and coined the Turing Test for AI. Following the war, Turing was persecuted by the British government for being homosexual and had his security clearance stripped in 1952. He died of cyanide poisoning in 1954, which was ruled a suicide. The British government finally formally apologized for Turing's treatment in 2009.



(Alan Turing passport photo, age 16 years, 1928 or 1929 [Turing]. Public Domain.)



Fig. 1.11 IBM System/360 Model 85. (Image courtesy of IBM Corporation [IBM64]. Reprinted with permission.)

computers. Some models sold for up to \$2 million. They spanned a wide range of performances, from the Model 30 with 8 KiB of magnetic core memory and 34.5 Kinstructions/second to the Model 91 with up to 6 MiB of memory and 16.6 Minstructions/second, filling a large room and consuming 94 KW of power.

As shown in Fig. 1.12, each bit of core memory was built from a ferrite donut hand woven into a mat of wires to flip orientation when a current was applied. Logic circuits were built with Solid Logic Technology (SLT), a precursor to integrated circuits [Davis64]. Fig. 1.13 shows an SLT module in which a transistor and several diodes and resistors are arranged on a shared substrate under a protective glass coat. Texas Instruments introduced the SN74xx-series of *small-scale integration* (SSI) chips in 1966, containing a handful of logic gates in a plastic package. These chips and their successors became widely used in both mainframes and minicomputers.

System/360 was succeeded by the compatible System/370 series in 1970, the System/390 series in 1990, and the z/Architecture in 2000 to provide software continuity over six decades. The z16 processor in 2022 has up to 200 cores, each running at up to 5.2 GHz, filling up to four refrigerator-sized frames, and it is still compatible with the original System/360 architecture. System/360 brought computing to the business mainstream and opened the ecosystem for many competitors, including Digital Equipment Corporation and Data General Corporation.

In the 1970s, Digital Equipment Corporation's 18-bit PDP-11 (Fig. 1.14) became the leading *minicomputer*, selling some 600,000 units at starting prices of \$12,000. Early computers were built from racks of

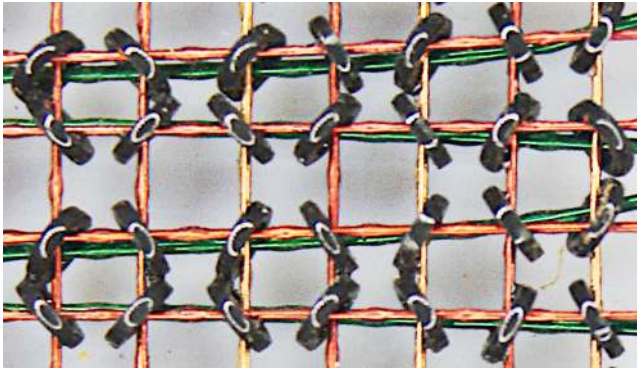


Fig. 1.12 Core memory. (Image courtesy of Ken Shirriff [memory]. Reprinted with permission.)

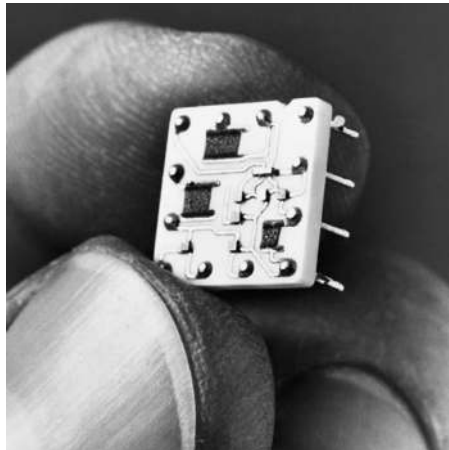


Fig. 1.13 IBM Solid Logic Technology. (Image courtesy of IBM Corporation [SLT]. Reprinted with permission.)

printed circuit boards containing hundreds of 74xx-series TTL logic gates. The PDP-11 soon found 18-bit addresses too constraining, so DEC followed up with the extremely successful 32-bit VAX *superminicomputer* that offered a large address space, virtual memory, floating-point operations, and vector processing.

The development of mainframes and minicomputers drove time-sharing OS advances. The Multics project, started in 1964 at MIT in collaboration with General Electric and Bell Labs, pioneered ideas in time-sharing, multiprocessing, virtual memory, security, and file systems. Multics ran behind schedule and required expensive hardware. When Bell Labs withdrew from the project, some of their developers took ideas from Multics to build a simpler OS capable of running on cheaper minicomputers such as a PDP-7 or PDP-11. These developers were Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy, and Joe

Grace Hopper (1906–1992) was a pioneering computer scientist. She was born in New York City, received her PhD in mathematics from Yale in 1934, and joined the faculty at Vassar College. During World War II, she volunteered to join the Navy Reserve, where she programmed the Mark I computer to calculate the Navy's gunnery tables. In the 1950s, she proposed programming in high-level languages using English words and developed the COBOL language and compiler. She continued her Naval service and led initiatives to standardize the Navy's computer languages, retiring as a Rear Admiral (formerly called Commodore) at 79 years of age. She was awarded the National Medal of Technology and the Presidential Medal of Freedom for her service.



Rear Admiral Grace Hopper (Image courtesy of the United States Navy DN-SC-84-05971, 1984 [Hopper]. Public Domain.)

Tracy Kidder's *The Soul of a New Machine* [Kidder81] gives a captivating account of the human side of developing the Data General Eclipse MV/8000 minicomputer, a 32-bit VAX competitor.

Ken Thompson (1943–) and Dennis Ritchie (1941–2011) received the Turing Award in 1983 for their contributions to operating systems, particularly their implementation of Unix at Bell Labs. Thompson received his bachelor of science and master's degrees in electrical engineering and computer science from Berkeley. Ritchie wrote his PhD thesis at Harvard University (but never bothered to pick up his degree). They both went directly to Bell Labs in the late 1960s.



Thompson (left) and Ritchie (right). (Photo taken 1973 [ThompsonRitchie]. Public Domain.)



Fig. 1.14 DEC PDP-11/70. (Photo by Dave McGuire, 2009 [PDP-11]. Public Domain.)

Ossanna, who developed the Unix OS around 1970 [Ritchie84]. At the time, operating systems were usually written in assembly language. From 1972 to 1973, the team developed the C programming language and used it to rewrite Unix at a higher level. Unix became a broadly used OS for technical computing. However, it suffered from licensing issues and competing incompatible implementations. In 1991, Linus Torvalds began reimplementing Unix from scratch as open-source software. His Linux OS has become the leading Unix-like OS. Appendix B is a getting-started guide for chip designers using Linux, and [Chapter 22](#) describes booting Linux in hardware.

1.5 MICROPROCESSORS

Robert Noyce and Gordon Moore founded Intel in 1968. Their first product was the 3101 static RAM chip shown in [Fig. 1.15](#). The chip had a whopping capacity of 64 bits, each clearly visible in the die photo, which was sufficient to begin the transition from magnetic core memories to integrated circuit RAM. In 1969 a Japanese calculator company called Busicom asked Intel to develop a 12-chip set for a desktop calculator. Ted Hoff was assigned to the design at Intel and soon realized that he could build the calculator more economically with a programmable processor. Too young to know it couldn't be done, Hoff and Federico Faggin designed Intel's 4-bit programmable 4004 in 1971. This chip is recognized as the world's first commercial microprocessor, in which all of the circuits were on a single chip. It used 2,300 transistors in a 10- μ m silicon-gate pMOS process. [Fig. 1.16](#) shows the 4004 die photo; some of the 4-bit datapaths are visible, such as the four repeating columns in the lower left. The layout was created from a red masking film called *rubylith*, cut by hand with a utility knife on a light table.

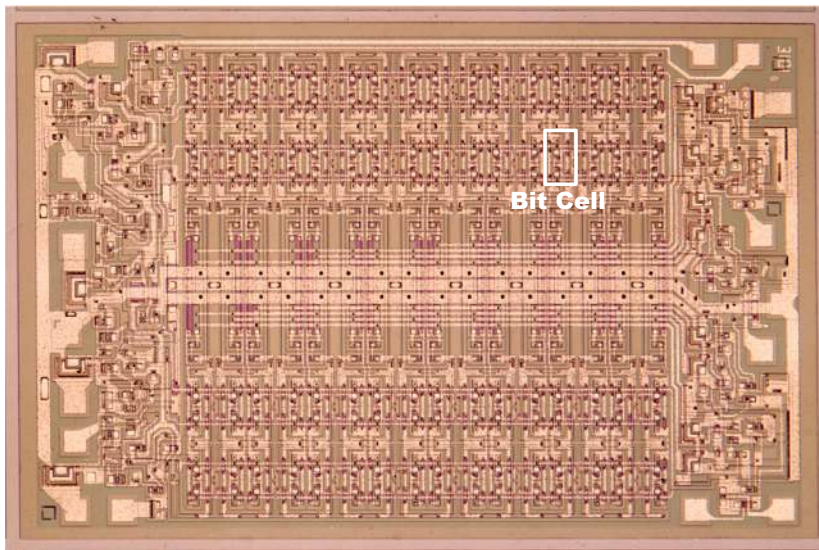
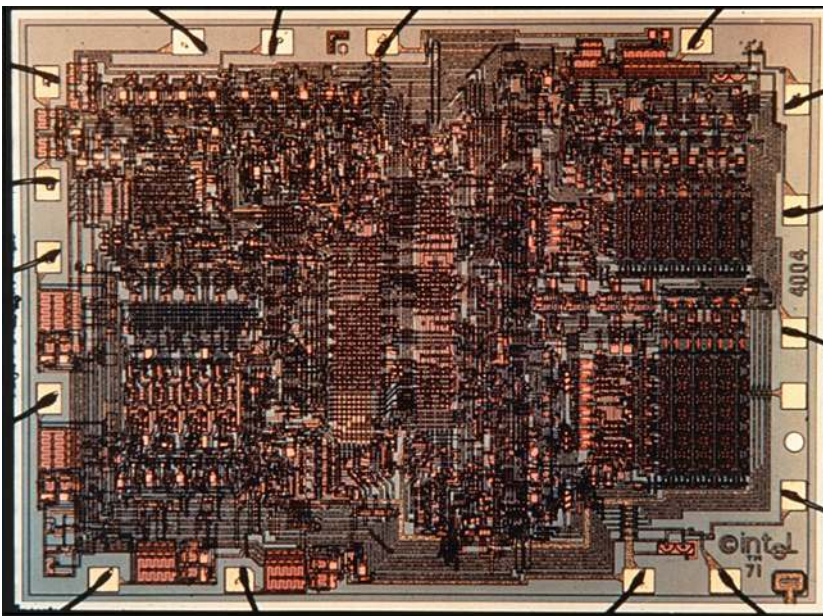
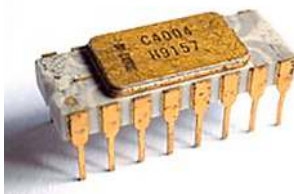


Fig. 1.15 3101 64-bit SRAM. (Image courtesy of Intel Corporation [Intel69]. Reprinted with permission.)



(a)



(b)

Fig. 1.16 Intel 4004 microprocessor: (a) die and (b) packaged chip. (Images courtesy of Intel Corporation [Intel71, Intel4004]. Reprinted with permission.)

The microprocessor revolution accelerated the progress of computing as integrated circuits rapidly improved transistor counts, speed, and cost. Soon processors became cheap enough that individuals could afford personal computers (PCs). Steve Wozniak and Steve Jobs founded the Apple Computer Company in 1976 to sell their Apple I computer, followed by the \$1,300 Apple II in 1977, which was based on the 6502 microprocessor (Fig. 1.17). IBM introduced the PC in 1981 (Fig. 1.18). Because Intel's 8088 was available off the shelf in quantity at a good price, IBM's PC used that processor, which had a 16-bit instruction set but a low-cost 8-bit data bus. IBM was unwilling to gamble on young Intel as its sole supplier of processors, so it arranged a second-source agreement allowing another startup, AMD, to manufacture compatible chips. The PC was



Fig. 1.17 Apple II computer. (© 2016 Pratyeka [Appelle]. This image is openly licensed under CC BY-SA 2.0.)



Fig. 1.18 IBM PC. (Figure courtesy of IBM Corporation [IBMPC]. Reprinted with permission.)

priced starting at \$1,565. IBM hoped to sell 200,000 units a year but was soon selling that many per month, and the PC revolution took off. The 8088 was a cost-reduced version of Intel's 8086, which had a 16-bit data bus. The 80286 added 24-bit addresses and a memory management unit. The 80386 stepped up to a 32-bit instruction set. Fig. 1.19 shows the 80386 die photo. The 32-bit integer datapath is on the left. The external bus unit is on the upper right. A microcode programmable logic array (PLA) is in the lower right. The middle part of the chip has synthesized random logic. The 80486 added pipelining, a cache, and a floating-point unit. The Pentium introduced the execution (or *issuing*) of two instructions at once, called *2-way superscalar issue*, and the Pentium Pro/II/III/IV added wider out-of-order issue. A victim of its tremendous success, Intel's x86

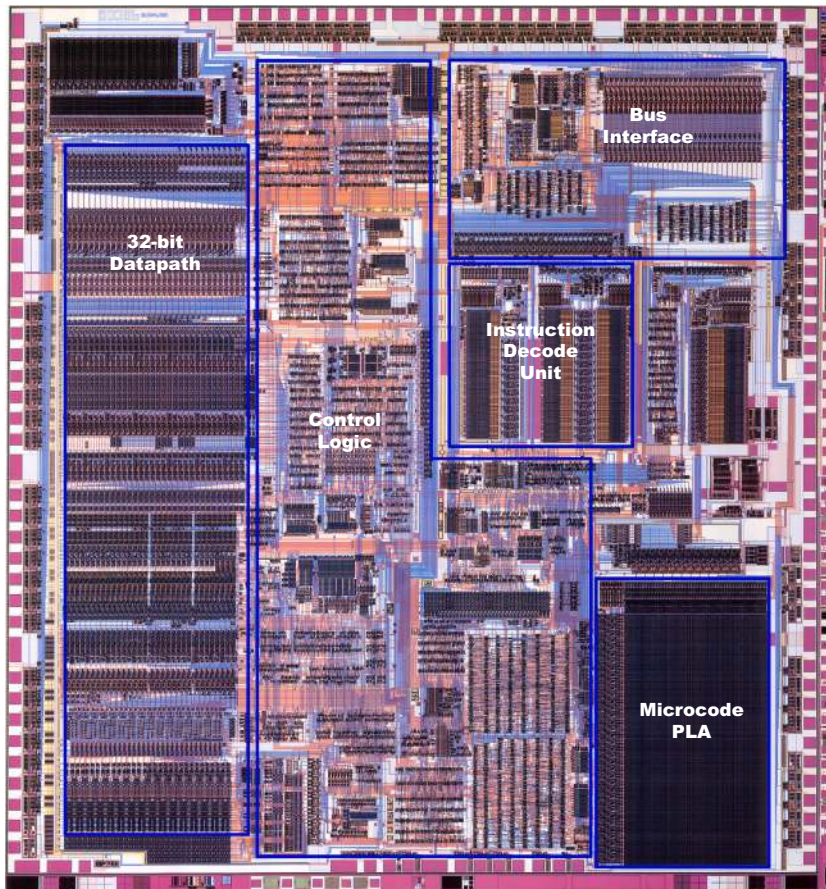


Fig. 1.19 Intel 80386 photo. (Image courtesy of Intel Corporation [Intel]. Reprinted with permission.)

architecture became a Frankenstein monster of backward-compatible variable-length instructions accumulated over the decades.

1.6 CISC AND RISC

In the 1970s, some computer architects sought to maximize the capability of assembly language instructions. For example, the x86 REP MOVSB instruction moves an entire variable-length zero-terminated string from one address to another. Greenblatt [Greenblatt74] describes an entire machine optimized for running LISP. In contrast, David Patterson and David Ditzel laid out a 1980 manifesto [Patterson80] for *reduced instruction set computers (RISC)*. They argued that “the ultimate test for cost-effectiveness is the speed at which an implementation executes a given algorithm.” Traditional *complex instruction set computers (CISC)*, such as S/360, PDP11, and x86, had long design times, high rates of design errors, costly state to save during context switches, and long clock cycles. Only a quarter of the available instructions in some instruction sets account for 99% of the instructions used in real programs. Patterson and Ditzel proposed simplifying to a minimal set of instructions necessary to compile code and making these instructions run as fast as possible using a minimum number of clock cycles at a high clock rate. For example, RISC processors use a *load/store architecture* in which instructions operate on data in a small set of fast registers and use explicit load and store instructions to move data between memory and these registers.

Many teams began developing RISC processors. Patterson’s team at Berkeley developed the 32-bit RISC-I processor in 1982 [Sequin82]. The 32 rows of the datapath are visible in the die photo in Fig. 1.20. The large array on the left is the register file, with the address decoders at the top. The columns on the right are the integer execution unit datapath. The control circuits are in the upper right, implemented with some PLAs. John Hennessy’s team at Stanford developed the MIPS-I processor in 1982–1984 [Hennessy82]. Steve Furber and Sophie Wilson at Acorn Computer developed the ARM1 processor in 1985. Hennessy and Patterson’s *Computer Architecture, a Quantitative Approach* [Hennessy90] became the leading text on computer architecture, and the latest edition is a must-read for all serious students of computer architecture today.

Microprocessor design entered an intensely competitive period from 1986 to 2002, as shown in Fig. 1.21, when companies raced to claim market share by offering the best performance. The time to execute a program depends on the product of the number of instructions, the cycles per instruction, and the cycle time. Manufacturers chased Moore’s Law aggressively to increase transistor speed and quantity, knowing their competitors were doing the same. Rising transistor counts allowed microprocessor designers to build more sophisticated microarchitectures, adopting features such as caches, branch prediction, superscalar, and out-of-order

The IBM 801, an experimental machine developed in the 1970s, is recognized as the first RISC processor. The architect, John Cocke, received the Turing Award for his contributions.

David Patterson (1947–) has been a professor of computer science at Berkeley since 1976. He coined RISC with John Hennessy in 1984. It was later commercialized by Sun Microsystems as the SPARC architecture. Patterson helped develop the RISC-V architecture and continues to play an integral role in its development.



(Image courtesy of David Patterson. Reprinted with permission.)

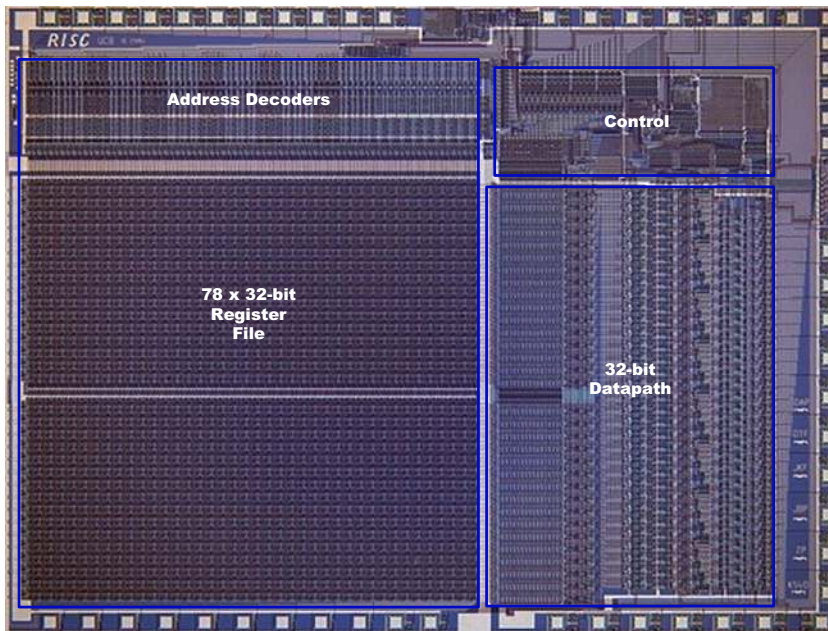


Fig. 1.20 RISC-I die photo. (Image courtesy of David Patterson. Reprinted with permission.)

execution pioneered by mainframes and minicomputers, driving down the cycles per instruction. Faster transistors and pipelining boosted clock speeds. Clever circuit designers innovated to reduce the number of gates between pipeline stages to improve clock speed even further. Digital Equipment Corporation introduced their Alpha 21064 RISC processor [Dobberpuhl92] running at 200 MHz in 1992, a year before Intel released their CISC (x86) Pentium chip running at 66 MHz. This stunning 3x clock rate advantage ended the debate of RISC vs. CISC. Intel became competitive again by translating its complex instructions into a sequence of one or more RISC-like microinstructions and then executing these microinstructions on a fast, efficient datapath, first seen in the Pentium Pro [Colwell05].

During the 1990s, processors were largely marketed based on clock rate. Intel, AMD, Alpha, MIPS, SPARC, and others competed for the highest clock rate during the dramatic “Megahertz Wars.” For example, Intel’s P6 microarchitecture came out at 200 MHz in the 1995 Pentium Pro processor in a 0.35- μm process and was aggressively pushed up to 1.13 GHz by the 1999 Pentium III in a 0.13- μm process. Intel then switched to their Netburst microarchitecture, reaching 3.4 GHz by 2002 for the Pentium 4 in the same 0.13- μm process. These clock frequencies were reached using superpipelining, with 14 pipeline stages in the Pentium III and 20 to 31 or more in the Pentium 4. The longer pipelines led to more pipeline hazards, so performance did not improve as much as clock rate

John Hennessy (1952–) is a professor of electrical engineering and computer science at Stanford University, where he served as president from 2000 to 2016. He coined RISC with David Patterson. He also developed the MIPS computer architecture and cofounded MIPS Computer Systems in 1984. The MIPS processor was used in many commercial systems, including products from Silicon Graphics, Nintendo, and Cisco. John Hennessy and David Patterson were given the Turing Award in 2017 for pioneering a quantitative approach to the design and evaluation of computer architectures.



(Image courtesy of John Hennessy. Reprinted with permission.)

increased, and in some cases, performance even decreased. Presently, pipeline depth ranges from 2 stages in low-cost microcontrollers to 5 to 8 stages in energy-efficient processors to 13 to 20 stages in high-performance processors.

Although long pipelines and high clock rates gave marginal performance benefits, they cost enormous amounts of power, often more than 100 W, making them expensive to cool and poorly suited to laptops. In a notorious keynote address at the International Solid-State Circuits Conference in 2001, Intel CTO Pat Gelsinger plotted the power density of processors over the years [Gelsinger01]. As shown in Fig. 1.22, he observed that “if scaling continues at present pace, by 2005, high-speed processors would have [the] power density of [a] nuclear reactor, by 2010, a rocket nozzle, and by 2015, [the] surface of sun.” Investors may have taken this badly; Intel stock declined more than 30% in the subsequent months. But Gelsinger was arguing that scaling had to change. Processors had reached the *power wall*, in which clock frequency is constrained by power dissipation. In 2006, Intel abandoned the power-hungry Netburst microarchitecture and switched to the Core microarchitecture that attempts to get performance from parallelism (running multiple threads at once) using multiple power-efficient cores derived from the Pentium Pro. Fig. 1.23 shows that the rate of performance increase leveled off dramatically after microprocessors hit the power wall and the end of Dennard scaling around

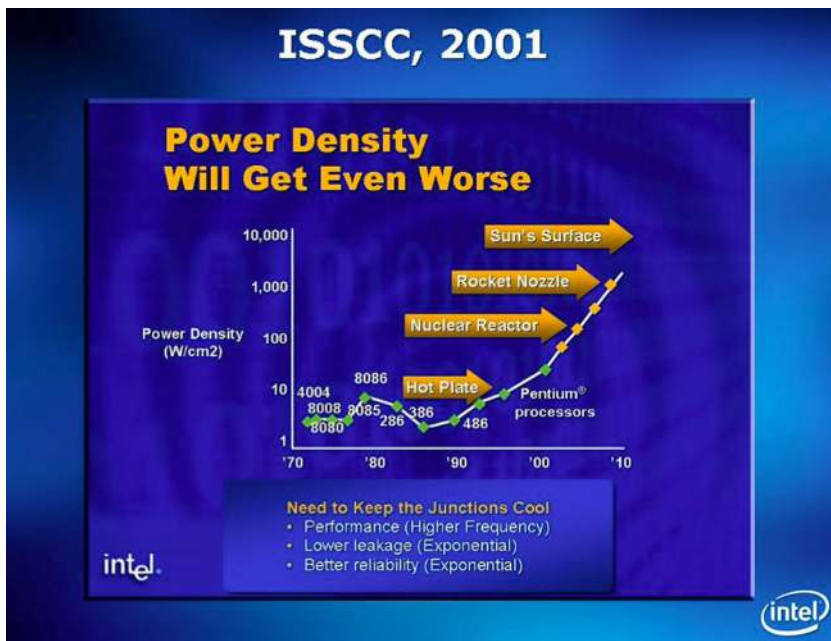


Fig. 1.22 Processor power density. (Figure courtesy of Intel Corporation [Gelsinger06]. Reprinted with permission.)

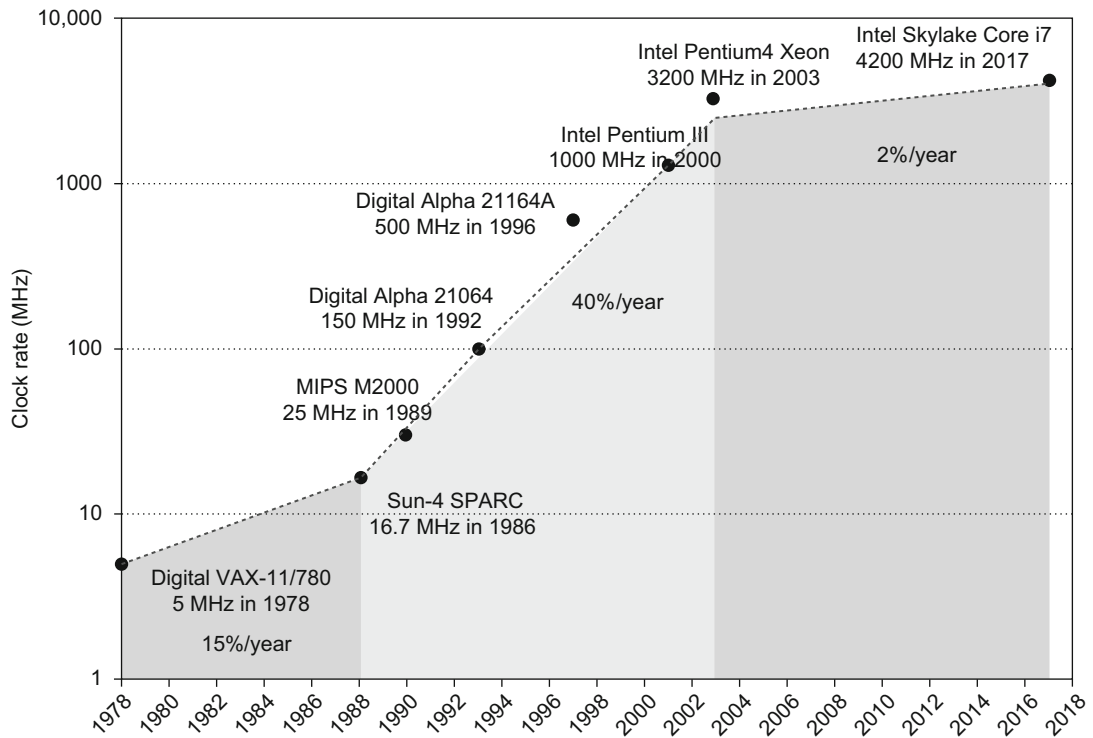


Fig. 1.23 Microprocessor clock speeds. (Figure courtesy of Elsevier [Hennessy19]. Reprinted with permission.)

2003, and Intel claimed most of the CPU market, reducing pressure for innovation. AMD took a sudden performance lead in 2020, and the renewed competition has been good for the industry and customers. Fig. 1.24 shows Intel's 13th-generation Core processor, Raptor Lake, with 24 total cores (16 energy-efficient cores and 8 high-performance cores running up to 5.7 GHz), 30 MiB of level 3 cache, graphics and video accelerators, and memory, display, and PCI expansion bus controllers, all fabricated on a 257-mm² die in the Intel7 process.

In recent years as innovation in PC processors has slowed, competition has shifted to the mobile device sector. Nearly all cell phones and tablets presently use ARM-based processors, which are also used in Apple computers and Raspberry Pi computers. ARM licenses its processor IP to companies such as Qualcomm, Samsung, MediaTek, and Apple, which design the rest of the SoC for mobile devices. Fig. 1.25 shows the logic board from the Apple iPhone 13 Pro, with an Apple A15 Bionic SoC. Recall the A15 die photo from Fig. 1.4.

Currently, more than 25 billion ARM processors ship each year, or more than 3 per human on earth, and most of these are embedded in systems where the customer may be unaware a computer exists. ARM makes

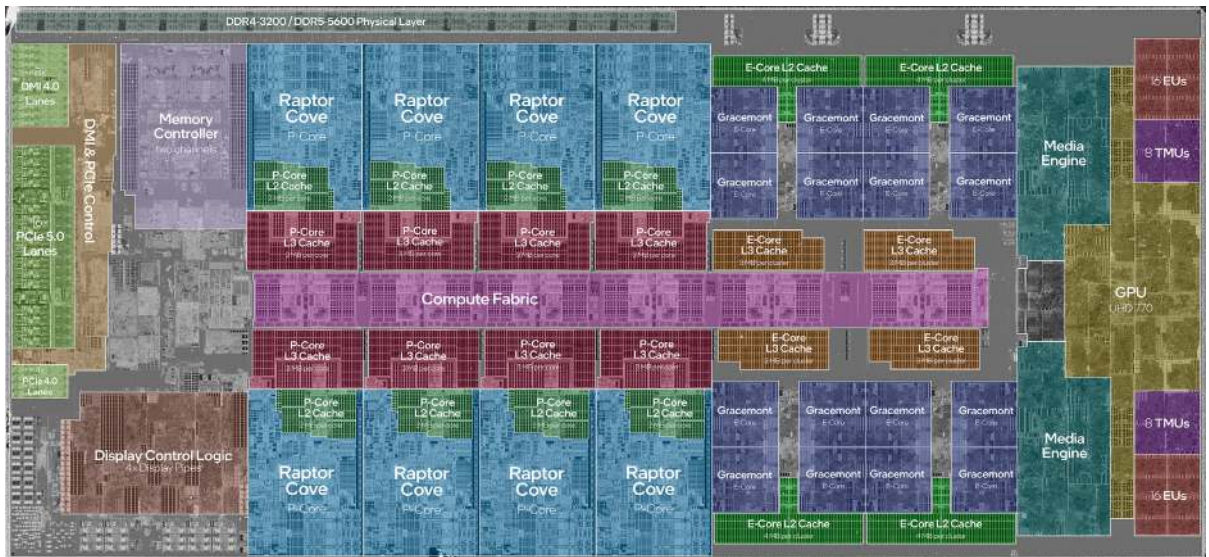


Fig. 1.24 Intel Raptor Lake Core i9-13900K die, with 8 high performance and 16 energy-efficient cores. (Image by JmsDoug and Fritzchens Fritz, 2023 [corei9]. Public Domain: CC0 1.0 Universal.)

Apple iPhone Pro Board Shot

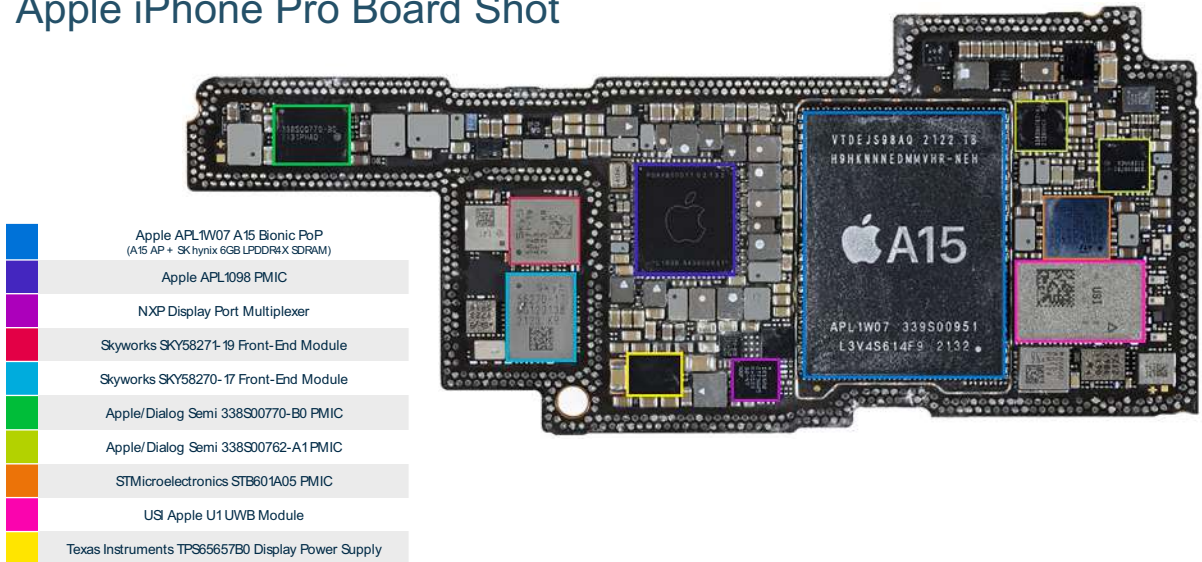


Fig. 1.25 Apple iPhone13 logic board with A15 processor. (Image courtesy of TechInsights Inc. [iphone13]. Reprinted with permission.)

its revenue from licensing processors, so the company is known for jealously guarding its IP, to the point that it is difficult for universities to use the architecture in education, and ARM sued one of their largest customers, Qualcomm, in 2022 over a licensing dispute.

1.7 RISC-V

Since the RISC revolution, computer architects have gained more experience with architectural tradeoffs. The original ARM architecture made several choices that make high-performance implementations more difficult, including conditional execution, too few registers, flexible addressing modes using three source registers, and an optional shift before the arithmetic/logic unit (ALU). In 2010, Krste Asanović, Andrew Waterman, Yunsup Lee, David Patterson, and others at Berkeley developed a fifth-generation RISC architecture, dubbed RISC-V. It has many similarities to the MIPS instruction set but avoids the pitfalls known from MIPS and ARM (see [Section 2.6](#)). Most importantly, the RISC-V architecture is an open standard with no licensing fees, and it has a defined process to continue to evolve under the stewardship of the nonprofit organization RISC-V International. At the time of this writing, RISC-V has a tiny share of the overall processor market, but it has tremendous momentum. By 2024, more than 13 billion RISC-V cores shipped, including over a billion in NVIDIA GPUs. In 2024, the SHD Group forecast RISC-V revenue growing at a compound annual rate of 47% to \$92 billion in 2030 [SHD24], as shown in [Fig. 1.26](#). Based on current trends, it is likely that x86 processors will continue to run laptop and desktop computers, ARM will continue to run phones and tablets (perhaps with some competition from RISC-V), and RISC-V will displace most of the diverse proprietary architectures serving other markets such as embedded processors and data centers. According to Asanović at the 2022 RISC-V Summit, “RISC-V is inevitable.” And according to RISC-V International’s former CTO Mark Himmelstein at the 2023 RISC-V Summit, “All your core are belong to us.”

1.8 INTERNATIONAL ECONOMIC AND SECURITY COMPETITION

In the 1970s, a handful of talented engineers could start a semiconductor company in a small building. A burst of startups in the 1970s and 1980s, centered in Silicon Valley but also active around the world, fed the highly innovative industry. Any company that wanted a fab could build its own. The exponentially increasing costs of developing advanced semiconductor technology have forced a shakeout, first eliminating smaller companies, then smaller countries, and now smaller continents. This drove manufacturing to foundries that produce chips for other companies. For example, in the United States, AMD and IBM, with large teams of brilliant technologists funded by formidable revenues from their microprocessor

Krste Asanović started RISC-V as a summer project at Berkeley. He is a professor of computer science at Berkeley and served as chairman of the board for RISC-V International. He is also the cofounder and chief architect of SiFive, a company that develops RISC-V chips, boards, and supporting tools.



(Image courtesy of SiFive. Reprinted with permission.)

This quote refers to an early-2000s meme from the badly translated Zero Wing video game, “All your base are belong to us.”)

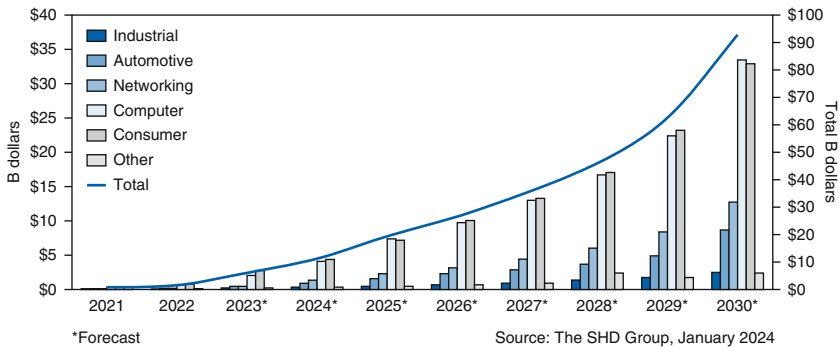


Fig. 1.26 Forecast RISC-V market growth. (Figure courtesy of The SHD Group [SHD24]. Reprinted with permission.)

and mainframe computer lines, ceased to be able to justify the cost of their fab development partnership. In 2008 and 2014, AMD and IBM sold their semiconductor business to the United Arab Emirates sovereign wealth fund which was seeking to diversify beyond oil. The resulting company, GlobalFoundries, manufactures chips in Singapore, Germany, and the United States. By 2015, Europe's largest semiconductor manufacturer, ST Microelectronics, was unable to keep up with the transition to a 14-nm FinFET process. They continue production, but not in the most advanced and profitable nodes. Japan, once home to many important semiconductor companies, fell to 2% of global foundry capacity by 2020. For decades, Intel led the industry, often a year or two ahead of the competition. Intel stumbled getting its 10-nm process into production and now trails behind the major foundries.

Fig. 1.27 shows the market share of the \$102 billion global semiconductor foundry industry as of 2022 [IDC23]. Taiwan Semiconductor Manufacturing Company (TSMC) is by far the dominant foundry, producing chips for most of the biggest companies, including Apple, Qualcomm, AMD, NVIDIA, Broadcom, and MediaTek. Samsung (South Korea) is the second largest foundry, serving external customers as well as its enormous internal cell phone business. UMC (United Microelectronics Corporation in Taiwan) and GlobalFoundries (international) compete for the remaining business, largely by being lower-cost suppliers. Intel (United States) is the last manufacturer primarily building internal products (microprocessors and field programmable gate arrays), but it is starting to offer foundry services.

Countries that control semiconductor design and manufacturing have tremendous economic and security power. Innovations in technology have been a major driver of world economic growth, and countries such as the United States, South Korea, Taiwan, Israel, and India have benefited disproportionately. Artificial intelligence (AI) has the potential of being the next huge growth opportunity. NVIDIA's market value tripled to more than

Andrew Waterman is chief engineer at SiFive, a company he co-founded with Krste Asanović in 2015 to provide low-cost RISC-V cores and custom chips. He earned his PhD in computer science from Berkeley in 2016, where, weary of the vagaries of existing instruction-set architectures, he co-designed the RISC-V ISA and the first RISC-V cores.



(Image courtesy of Andrew Waterman. Reprinted with permission.)

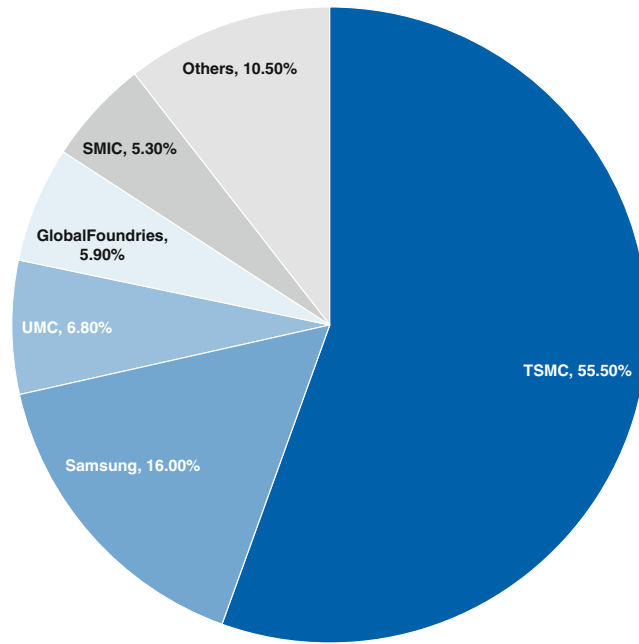


Fig. 1.27 Global semiconductor foundry market share (2022).

\$3 trillion in 2024 because its chips are in great demand for machine learning. Vast amounts of the world's information pass through cell phone and fiberoptic networks. Governments have influenced the design of communication chips and systems to surreptitiously collect information. In 2018, the United States and other countries banned cellular providers from using Huawei and ZTE chips because of Chinese espionage concerns, and in 2023, China retaliated by banning government officials from using Apple iPhones. After Russia invaded Ukraine, Western countries attempted to enforce export controls to reduce Russia's access to chips used in their weapons systems. In a sign of growing economic competition, in 2023, the United States, Japan, and the Netherlands imposed export controls prohibiting the sale of advanced photolithography equipment to China, and China retaliated by cutting off exports of gallium, germanium, and other key materials used in chip manufacturing.

The two largest semiconductor producers are in areas of great geopolitical instability. South Korea is in range of conventional or nuclear attack by North Korea. China claims ownership of Taiwan and has been making heavy military investments, threatening invasion. Because many countries depend on TSMC for producing chips critical to both military and commercial activity, this has become a matter that threatens the national and economic security of most developed nations. The United States eventually recognized this risk, and a divided Congress passed the bipartisan

CHIPS Act in 2022, a rare act of industrial policy allocating \$280 billion over a decade to spur chip research and production in the United States. The European Union, United Kingdom, and Japan have passed similar legislation to invest in this essential industry.

The open RISC-V architecture promises to change the power that the proprietary x86 and ARM architectures have held over international trade and perhaps the associated power the United States has held in the PC industry.

1.9 SUMMARY

Since the invention of the integrated circuit in 1958, chips have advanced to the point that we can build a chip with tens of billions of transistors for tens of dollars. The transistors switch in picoseconds using subfemtojoules of energy, allowing us to build mobile devices with a full day's battery life. Early electromechanical, vacuum tube, and transistor computers have given way to microprocessors, which have revolutionized nearly every aspect of society, from communication to medicine and from entertainment to warfare. The proprietary x86 and ARM architectures have long dominated personal computing and mobile devices. Since its inception in 2010, the open RISC-V architecture has grown at an unprecedented rate, now shipping more than 10 billion cores per year. RISC-V enables everything from simple microcontrollers that cost microcents to drop into an SoC, to application processors running Linux, to high-performance servers. It has also spurred development of a rich open-source software ecosystem.

The next chapter introduces the RISC-V architecture, including the base integer instruction set, privileged operations, the memory map, and standard extensions. It shows simplified single-cycle and pipelined implementations of a subset of the base instruction set, illustrating microarchitecture, logic design, and logic verification.

Introduction to RISC-V

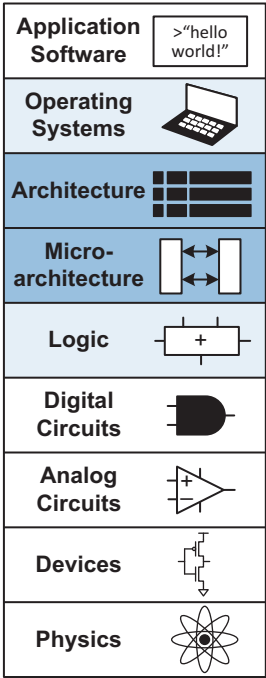
2

RISC-V is a fifth-generation *reduced instruction set computer* (RISC) architecture initially developed at Berkeley and now stewarded by the nonprofit RISC-V International. Its streamlined design reflects lessons from three decades of RISC processor sign, notably that clever optimizations in one era become implementation obstacles in a later era. Most importantly, RISC-V is a free and open standard that has led to widespread adoption and vibrant competition and innovation.

RISC-V is a load/store *instruction set architecture* (ISA) with 32-bit instructions, 32 general-purpose registers of width XLEN, and a program counter of width XLEN. The registers are named $x0$ to $x31$, and $x0$ is hardwired to 0. The base integer ISA comes in RV32I and RV64I flavors, corresponding to $XLEN = 32$ or 64 . RV32I has 37 essential instructions summarized in Table I.1 (on the last pages of the book). RV64I adds 12 more instructions to handle 64-bit (double-word) loads and stores and to operate on only the bottom 32 bits of a 64-bit register.

The *architectural state* of a processor is the minimum set of information to determine what a program will do. For RISC-V, the architectural state consists of the memory, program counter, and registers, including optional *control and status registers* (CSRs). A processor can run many *threads*. Each thread has its own values of the program counter and registers, called the thread’s *context*, but shares memory. The *operating system* (OS) gives the appearance of running all threads concurrently by switching from one to another every few milliseconds. This *context switch* involves saving the context of one thread to memory and reloading the program counter and registers with the context of the next thread.

- 2.1 RISC-V Assembly Language
- 2.2 RISC-V Machine Language
- 2.3 Have a Hart
- 2.4 Memory Map
- 2.5 RISC-V Extensions and Profiles
- 2.6 Comparison with Other Architectures
- 2.7 RISC-V Microarchitecture
- 2.8 Wally System-on-Chip
- 2.9 RISC-V Market
- 2.10 Summary and a Look Ahead



RV32I also has `fence`, `ecall`, and `ebreak` instructions, bringing the official total to 40, but these can be ignored in a simple implementation.

A computer saves power by going into *sleep* mode. It saves the context of the current thread to memory and then powers off the processor while keeping the memory powered up. It can rapidly wake from sleep by powering up the processor and reloading the thread context from memory. A computer may reduce power consumption to zero by going into *hibernate* mode. It saves the entire architectural state, including the contents of memory, to disk, then powers off the entire system, including memory as well as the processor. It takes much longer to wake from hibernation because it has to reload the architectural state from disk.

Observe that the architectural state comprises the memory and the current thread's context.

Microprocessors are often classified as *application processors* or *embedded processors*. Application (“apps”) processors generally run applications on a multitasking OS with a graphical user interface on a screen; these are the main processors in mobile devices, personal computers (PCs), and servers. Embedded processors are often invisible to the user; they may connect a switch to the window's motor controller in your car, make annoying noises in a child's toy, or manage voltage regulators and clock generators behind the scenes for the application processors in a cell phone system-on-chip (SoC). Because RV32 addresses are only 32 bits, the architecture has a 4-GiB address space. This is more than sufficient for embedded applications that often have kilobytes of code and data but is too small for many systems with a modern OS. Thus, application processors commonly use RV64.

The RISC-V architecture is officially defined at riscv.org/specifications/ratified, including the *Unprivileged Specification* that covers the main instruction set and extensions and the *Privileged Specification* that covers privilege modes, traps, and CSRs [RISCV-Unprivileged, RISCV-Privileged]. This section summarizes the RV32I and RV64I base instruction sets; see the Unprivileged Specification for more details. The Privileged Specification and standard extensions are covered in later chapters. The Unprivileged Specification is one of the best-written architecture specifications in existence, being succinct, clear, and annotated with commentary to explain the guiding principles behind the architectural choices. Those with serious interest in RISC-V should read these specifications rather than relying on the summaries in this book.

This chapter introduces the RISC-V architecture, including its assembly and machine language. It then illustrates the major design steps using toy single-cycle and pipelined RISC-V processors before introducing CORE-V Wally (cvw), a configurable RISC-V pipelined processor capable of running Linux. Wally supports the rich set of extensions in the RVA22S64 application processor profile, as well as virtual memory, caches, and branch prediction.

2.1 RISC-V ASSEMBLY LANGUAGE

The RV32I instruction set is summarized in Table I.1 (on the last pages of the book). The `op`, `funct3`, and `funct7` fields relate to the machine language encoding, which is discussed in Section 2.2. Instructions are classified into six types based on their operands and encoding, as shown in Table 2.1.

2.1.1 R-Type Instructions

R-type instructions have only register operands: two source registers and one destination register. For example, `add x3, x1, x2` computes the sum of

Table 2.1 Instruction types.

Type	Description	# Source Registers	# Destination Registers	Immediate
R	Register-type	2	1	N/A
I	Immediate-type	1	1	12-bit
S	Store-type	2	0	12-bit
B	Branch-type	2	0	13-bit branch offset
J	Jump-type	1	1	21-bit jump offset
U	Upper immediate-type	1	1	20-bit upper immediate

the values in source registers $x1$ and $x2$ and puts the result in destination register $x3$. R-type instructions are executed by the integer *arithmetic/logic unit* (ALU), and they include:

- ▶ Arithmetic operations: `add`, `sub`
- ▶ Logical operations: `and`, `or`, `xor`
- ▶ Shift operations: `sll`, `srl`, `sra`
- ▶ Comparison operations: `slt`, `sltu`

Arithmetic and logical operations are the same, whether their operands are treated as unsigned or two's complement signed numbers. Recall that a *two's complement* signed binary number has a negative weight for the most significant bit. For example, the bits 1110_2 interpreted as a 4-bit *unsigned* number have a value of $1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 14_{10}$. The same bits interpreted as a 4-bit signed two's complement number have a value of $1 \times -(2^3) + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -2_{10}$. The bits $0001_2 = 1_{10}$ as either unsigned or two's complement. For example, `add 1110, 0001` = `1111`, which is 15_{10} as unsigned or -1_{10} as two's complement, and is correct in either interpretation.

Shift operations multiply or divide by powers of 2 by shifting left or right. Shift left logical (`sll`) fills the bottom bits with 0s. Shift right logical (`srl`) fills the upper bits with 0s. Shift right arithmetic (`sra`) fills the upper bits with the old sign bit. A left or right shift by k is equivalent to a multiply or divide by 2^k . When dividing, use `srl` if the number is unsigned or `sra` if it is signed. For example, `sll 1110, 0001` = `1100`. As a two's complement number, the result is $-2 \ll 1 = -2 \times 2^1 = -4$. As an unsigned number, the result is $14 \ll 1$, which overflows to 12. `srl 1110, 0001` = `0111`. As an unsigned number, the result is $14 \gg 1 = 14/2^1 = 7$. `sra 1110, 0001` = `1111`. As a two's complement number, the result is $-2 \gg 1 = -2/2^1 = -1$.

Set less than (`slt`) comparison sets the result to 1 if the first operand is less than the second or 0 otherwise. Comparisons and branches (`slt`, `blt`, `bge`) ordinarily treat register values as signed two's complement integers, but the unsigned flavors (`sltu`, `bltu`, `bgeu`) treat them as unsigned integers instead. For example, `slt 1110, 0001` = 1 because $-2 < 1$, but `sltu 1110, 0001` = 0 because $14 \geq 1$.

2.1.2 I-Type Instructions

I-type instructions take a source register, a 12-bit signed *immediate* (i.e., a constant in the instruction that is immediately available without accessing memory), and a destination register. The immediate is sign-extended to XLEN before use. These instructions include:

- ▶ Loads: `lw, lh, lhu, lb, lbu`
- ▶ Jumps: `jalr`
- ▶ Arithmetic operations: `addi`
- ▶ Logical operations: `andi, ori, xori`
- ▶ Shift operations: `slli, srli, srai`
- ▶ Comparison operations: `slti, sltiu`

Note that `subi` is not needed because it can be performed by an `addi` with a negative immediate.

Beware that `sltiu` sign-extends the immediate to 32 bits before treating it as an unsigned number for comparison!

Loads read a value from memory and place it in the destination register. The memory address is computed by adding the sign-extended immediate to a base address in the source register. *Subword loads* come in normal and unsigned versions. The normal version (`lb/lh`) sign-extends the byte or half-word to the full XLEN length. The unsigned version (`lbu/lhu`) zero-extends.

Jump and link register (`jalr`) computes an address in the same way as loads: as the sum of the base address in the source register, and a sign-extended immediate. It jumps to this address. It also saves PC+4 in the destination register to remember where to return to later; this is called *linking*.

Other I-type arithmetic, logical, shift, and comparison instructions resemble R-type instructions but use the source register as the first operand and the immediate as the second operand. Immediate shift instructions (`slli`, `srli`, `srai`) shift by the unsigned number in the lower 5 bits of the immediate.

2.1.3 S-Type Instructions

S-type instructions are stores. They take two source registers and a 12-bit signed immediate. Like loads, they add the immediate to the first source register to compute an address. Then they store the contents of the second source register into that address. These instructions include store word/half/byte: `sw/sh/sb`.

2.1.4 B-Type Instructions

B-type instructions are branches. Like S-type instructions, they also take two source registers and a 13-bit signed immediate. They compare the two source registers. If the condition is satisfied, they take the branch by sign-extending the immediate (also called the *branch offset*) and adding it to the PC. Branches include branch if equal/not equal (*beq/bne*) and branch if less than/greater than or equal in both signed (*blt/bge*) and unsigned (*bltu/bgeu*) versions. Because the 13-bit branch offset will always be a multiple of 2, and thus its least significant bit will always be 0, only its upper 12 bits are encoded in the instruction's immediate field. The processor must append the implicit least significant 0 before sign-extending and adding it to the PC. So, the branch offset spans the range $[-4096, 4094]$.

2.1.5 J-Type Instructions

The only J-type instruction is *jump and link* (*jal*), which makes an unconditional PC-relative jump and saves PC+4 in the destination register as the return address. *jal* takes a destination register and a 21-bit two's complement immediate (*jump offset*). Like branches, *jal* only encodes the upper 20 bits of the jump offset in the instruction because the lsb is always 0. When executing *jal*, the processor appends a 0 to the encoded 20 bits of the immediate before sign-extending it. Thus, the jump offset spans the range $[-1048576, 1048574]$. *jal* adds this jump offset to the PC and jumps to that address. For unconditional jumps (*j*), *x0* is used as the destination register to discard the return address.

2.1.6 U-Type Instructions

Like J-type instructions, U-type instructions also use a destination register and a 20-bit immediate. They shift the immediate left by 12 bits into the upper bits of a 32-bit word, filling the lower bits with 0s.

The load upper immediate (*lui*) instruction places the shifted immediate in the destination register. In combination with *addi*, this is useful for creating 32-bit constants.

Remember that 12-bit immediates are sign-extended, which could throw off the upper bits by 1. For example, to load the constant 0x00003800 into *x1*, use the following sequence. -2048 in 12-bit two's complement is 0x800, which is sign-extended to 0xFFFF800.

```
lui  x1, 0x00004      # x1 = 0x00004000
addi x1, x1, -2048    # x1 = 0x00004000 + 0xFFFF800 = 0x00003800
```

Ordinary RISC-V instructions are 4 bytes long, but compressed instructions are 2 bytes, so instructions may be on 2-byte boundaries. They are never at odd addresses, so the least significant bit (lsb) of the branch offset is always 0. RISC-V gains additional branching range by omitting the branch offset's lsb when encoding this immediate in the instruction.

The C language and RISC-V assemblers use 0x and 0b prefixes to indicate hexadecimal and binary, respectively.

The more intuitive sequence of `lui x1, 0x00003; addi x1, x1, 0x800` would produce `0x00002800` instead of the intended result.

The *add upper immediate to PC* (`auipc`) instruction adds the shifted immediate to the program counter and places it in a destination register.

`auipc rd, 0` is used to copy the PC into `rd`. `auipc` is also useful for computing arbitrary offsets relative to the program counter for either loads/stores or jumps. For either case, use `auipc` to add the upper 20 bits of the offset to the PC into a temporary register. Then, for a PC-relative load or store, use the 12-bit immediate offset to add the low bits to the temporary register to form the address. For a long PC-relative jump, use `jalr` to add the lower 12 bits. For example, if the PC is initially at `0x80001000`, to jump ahead by `0x10002456` to `0x90003456`, use the following code sequence:

Notice how `x1` is conveniently used as a temporary register because it will be overwritten with the return address.

```
auipc x1, 0x10002    # x1 = 0x80001000 + 0x10002000 = 0x90003000
jalr  x1, 0x456(x1)  # Jump to 0x90003000 + 0x00000456 = 0x90003456
                        # x1 = PC + 4 = 0x80001008
```

2.1.7 Other Instructions

The base instruction set defines `ecall` and `ebreak` for OS calls and debugger breakpoints, as discussed in [Section 2.3.6](#). If there is no privileged `mtvec` trap vector control register, these instructions do nothing.

The base instruction set also defines the `fence` instruction for memory ordering. A simple single-core in-order processor such as Wally inherently executes memory accesses in program order and thus can ignore `fence`. [Section 17.2.2](#) discusses fences for concurrent systems.

2.1.8 RV64I Additional Instructions

Table I.2 (on the last pages of the book) summarizes the additional instructions in RV64I.

RV64I adds `ld` and `sd` to load and store 64-bit *double words*. It also adds `lwu` to load an unsigned word into the bottom 32 bits of a 64-bit register and fill the top 32 bits with 0s. Furthermore, shift amounts are extended to 6 bits.

In C-like languages, the 32-bit `int` data type is common even in 64-bit code. 32-bit `ints` are represented in 64-bit registers with bit 31 (the sign bit) extended into the upper bits (63:32). Ordinary 64-bit add, subtract, and shift instructions can produce incorrect results when the inputs and outputs are interpreted as 32 bits. These would require costly masking sequences to fix, so RV64 provides add, subtract, and shift instruction versions with a `w` suffix (which we call RW- and IW-type instructions) to support 32-bit operations. Any result bits overflowing the 32-bit word are discarded, and then the 32-bit result is sign-extended to fill the 64-bit register.

Consider adding the following two 32-bit integers on a 64-bit ALU. The 32-bit result should be 0:

```
0x8000_0000 + 0x8000_0000
= 0x0000_0000
```

These numbers are represented in 64-bit form by sign-extending from 32 to 64 bits: `0xFFFF_FFFF_8000_0000`.

Computing the addition on a 64-bit ALU:

```
0xFFFF_FFFF_8000_0000 +
0xFFFF_FFFF_8000_0000
= 0xFFFF_FFFF_0000_0000.
```

But when interpreted as a 32-bit result, the answer should be 0, so the upper bits are discarded and the 32-bit result is sign-extended, which instead results in `0x0000_0000_0000_0000`, as desired.

2.1.9 Register Conventions

The `x0` register is also called `zero` and is hardwired to 0 because it is such a commonly used constant. In principle, registers `x1-x31` could be used for any purpose. However, RISC-V establishes some conventions given in Table I.4 (on the last pages of the book) so that different programmers can write different blocks of code that play together nicely. `x1` is called `ra` and is used to hold the *return address* when making a function call with a `jal` instruction. `x2` is called `sp` and points to the *top of the stack*. The `s0-s11` *saved* registers are used for local variables that must preserve their value across function calls, while the `t0-t6` *temporary* registers hold intermediate results that can be overwritten during function calls. Function calls pass their inputs in the `a0-a7` *argument* registers and return their result in `a0`.

The strange ordering of register names was selected so that RV[32/64]E systems with only 16 registers (`x0-x15`) would still have some registers available for each purpose.

2.1.10 Endianness

A processor's *endianness* defines how it stores the bytes within a word, as shown in Fig. 2.1. In little-endian, the least significant byte in a 32-bit word is stored at an offset of 0 from the start of a word address, and the most significant byte is stored at an offset of 3. In big-endian, the order is reversed. For example, consider storing the word `0x1234F0E0` at address `0x8`, as shown in Fig. 2.1. In a big-endian machine, address `0x8` contains byte `0x12`, and address `0xB` contains `0xE0`. In a little-endian machine, address `0x8` contains `0xE0`, and `0xB` contains `0x12`. A `lb t0, 8(x0)` instruction that reads a byte from address 8 will return `0x12` for big-endian and `0xE0` for little-endian machines.

RISC-V is little-endian by default. The `status` register can optionally switch the processor into big-endian mode for loads and stores, as described in Section 8.2.3.1. Instructions are always fetched in little-endian form.

2.1.11 Pseudoinstructions

RISC-V is missing many instructions that would be useful to a programmer, but these can be implemented using existing instructions.

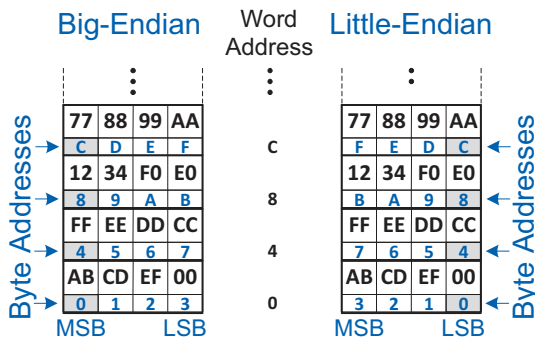


Fig. 2.1 Big and little-endian memory addressing.

A full list of pseudoinstructions can be viewed at github.com/riscv-non-isa/riscv-asm-manual/blob/main/src/asm-manual.adoc#pseudoinstructions.

For example, we might want a *load immediate* instruction `li rd, imm` that loads an immediate into register `rd`. For small immediates, this can be implemented using the real RISC-V instruction `addi rd, x0, imm`, which adds the immediate to 0 and places it in `rd`. Thus, `li` is called a *pseudoinstruction*. The programmer can use the pseudoinstruction, but the assembler replaces it with the real `add` instruction. Similarly, the *branch on less than or equal* pseudoinstruction `ble rs1, rs2, label` is replaced with the real instruction `bge rs2, rs1, label`; the *jump* pseudoinstruction `j label` is replaced by `jal x0, label`; and the *move* pseudoinstruction `mv rd, rs1` is replaced by `addi rd, rs1, 0`. Table I.7 (on the last pages of the book) summarizes common RISC-V pseudoinstructions, and is worth reviewing.

2.1.12 C to Assembly Language Examples

Programs are typically written in a high-level language and then compiled into assembly language. C is popular for system programming because it supports pointers, so programmers can directly access locations in memory. Other popular languages such as C++ and Java are closely related. This section reviews how C programs are translated to assembly language to help you think about both languages.

In each of the following programs, assume the following variables have been declared and that they are kept in specific registers. For example, `a` is in register `s0`, `b` is in register `s1`, and so on.

```
int a, b, c;      // a, b, c, in s0, s1, s2
int i, j;         // i, j in s3, s4
int grades[100]; // Base address of grades in s5
char name[80];   // Base address of name in s6
```

Code Example 2.1 shows how arithmetic in C translates into an R-type instruction in assembly language. Code Example 2.2 shows a logical instruction and how to use a constant expressed in hexadecimal. Code Example 2.3 shows how to break more complex arithmetic into a sequence of simpler steps; notice how it uses a temporary register `t0` to hold the intermediate result.

Code Example 2.1 SIMPLE ADDITION

C Code	Assembly Language
<code>a = b + c;</code>	<code>add s0, s1, s2</code> <code># a = b + c</code>

Code Example 2.2 ANDING WITH A CONSTANT

C Code	Assembly Language
<code>a = b & 0xFF;</code>	<code>andi s0, s1, 0xFF</code> <code># 0xFF = 11111111</code>

Code Example 2.3 TEMPORARY REGISTER (`t0`) HOLDING INTERMEDIATE CALCULATION

C Code	Assembly Language
<code>a = a + b - c;</code>	<code>add t0, s0, s1</code> <code># t0 = a + b</code> <code>sub s0, t0, s2</code> <code># a = t0 - c</code>

Code Example 2.4 shows an `if` statement. The assembly language branches over the body if the opposite condition is true (`bne`, for `==`). Note the use of a label indicating where to resume. Code Example 2.5 shows an `if/else` statement. The assembly language branches to `else` if the opposite condition is true. Otherwise, it does the body of the `if` and then jumps past the `else`.

Code Example 2.4 `if` STATEMENT

C Code	Assembly Language
<code>if (i == j)</code> <code>a = 1;</code>	<code>bne s3, s4, done</code> <code># Skip if i != j</code> <code>li s0, 1</code> <code>done:</code>

Code Example 2.5 `if/else` STATEMENT

C Code	Assembly Language
<code>if (i == j)</code> <code>a = 1;</code>	<code>bne s3, s4, else</code> <code># i != j</code> <code>li s0, 1</code> <code># a = 1</code> <code>j done</code>
<code>else</code> <code>a = 2;</code>	<code>else: li s0, 2</code> <code># a = 2</code> <code>done:</code>

Code Example 2.6 shows a `while` loop to add up even numbers from 0 to 10. The code initializes `a` and `i`, then loads the comparison value (10) into a temporary register. It exits the `while` loop if the opposite condition is satisfied: `bgt` (branch if greater than) for `<=` (less than or equal

comparison). Otherwise, it executes the body of the loop and then jumps back to check the condition again.

Code Example 2.6 while LOOP TO ADD EVEN NUMBERS FROM 0 TO 10

C Code	Assembly Language
<pre>a = 0; i = 0; while (i <= 10) { a = a + i; i = i + 2; }</pre>	<pre>li s0, 0 # a = 0 li s3, 0 # i = 0 li t0, 10 # Need 10 for compare while: bgt s3, t0, done # Done if i > 10 add s0, s0, s3 # a = a + i addi s3, s3, 2 # i = i + 2 j while # Repeat while loop done:</pre>

Code Example 2.7 shows a for loop to compute $a = 2^b$. A for loop has four parts: the initialization, comparison, postoperation, and body. The assembly language first does the initialization ($i = 1$). Then, at a label, it checks the opposite condition ($>$) to exit the loop. It performs the body ($a = a * 2$) and then the postoperation ($i++$) and then jumps back to the label to repeat the loop.

Code Example 2.7 for LOOP TO COMPUTE $A = 2^B$

C Code	Assembly Language
<pre>a = 1; for (i = 1; i <= b; i++) a = a * 2;</pre>	<pre>li s0, 1 # a = 1 li s3, 1 # i = 1 for: bgt s3, s1, done # Done if i > b slli s0, s0, 1 # a = a * 2 addi s3, s3, 1 # i++ j for # Repeat loop done:</pre>

In C, & is read as “address of.” Hence, &grades[i] is the address of the *i*th element of the grades array.

Code Example 2.8 accesses fixed elements of an array using lw and sw with constant offsets. An int is 4 bytes, so grades[1] is found 4 bytes beyond the beginning of the array. Code Example 2.9 shows a loop accessing and incrementing elements of an array. The address of grades[i] is 4**i* bytes past the beginning, so the loop uses a shift to multiply *i* by 4, then adds this offset to the base address to find the address of grades[i].

Code Example 2.8 ACCESSING SINGLE ELEMENTS OF AN INTEGER ARRAY

C Code	Assembly Language
<code>grades[1] = grades[0] + 17;</code>	<pre> lw t0, 0(s5) # t0 = grades[0] addi t0, t0, 17 # t0 = t0 + 17 sw t0, 4(s5) # grades[1] = t0 </pre>

Code Example 2.9 `for` LOOP GRADE INFLATION

C Code	Assembly Language
<pre> for (i = 0; i < 100; i++) grades[i] = grades[i] + 14; </pre>	<pre> li s3, 0 # i = 0 li t0, 100 # t0 = 100 for < for: bge s3, t0, done # Done if t0 >= 100 slli t1, s3, 2 # t1 = i * 4 add t1, s5, t1 # t1 = &grades[i] lw t2, 0(t1) # t2 = grades[i] addi t2, t2, 14 # t2 = grades[i] + 14 sw t2, 0(t1) # grades[i] = t2 addi s3, s3, 1 # i++ j for # Repeat done: </pre>

Code Example 2.10 illustrates string handling by computing the length of a string. Recall that C strings are arrays of characters (bytes) terminated by a 0 (NULL) byte. Also recall that an expression in C is considered FALSE if 0 and TRUE if nonzero. Hence `while(name[i])` loops until reaching the NULL byte at the end of the string. The assembly language uses the `lb` instruction to access `name[i]`.

Code Example 2.10 `while` LOOP THAT COUNTS THE NUMBER OF LETTERS IN A NAME

C Code	Assembly Language
<pre> i = 0; while (name[i]) i++; </pre>	<pre> li s3, 0 # i = 0 while: add t0, s6, s3 # t0 = &name[i] lb t1, 0(t0) # t1 = name[i] beq t1, zero, done # (name[i]==0)? addi s3, s3, 1 # i++ j while # Repeat done: </pre>

Code Example 2.11 shows a factorial function, which illustrates passing an argument, returning a result, and saving and restoring registers on the stack. The argument `n` arrives in `a0`. The function uses two local variables in `s0` and `s1`, so it saves the old values of these registers on the stack

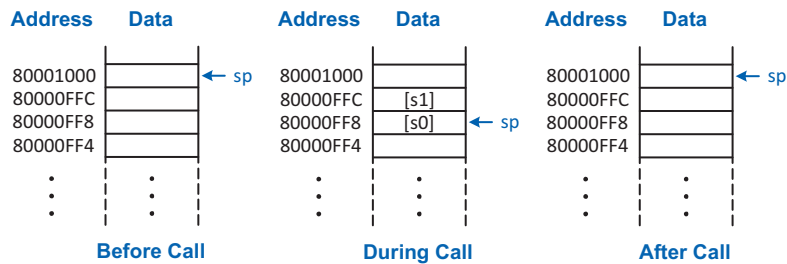


Fig. 2.2 Stack for factorial function from Code Example 2.11.

before changing them and restores these registers at the end. The stack pointer moves down and up by 8 bytes (two 4-byte words) to make room for these registers, as shown in Fig. 2.2. Like all functions, `fact` places the return value (in this case, `result`) in `a0` before returning to the caller using the `ret` pseudoinstruction.

`fact` uses the `mul` (multiply) instruction from the M extension.

Code Example 2.11 FACTORIAL FUNCTION

C Code	Assembly Language
<pre>int fact(int n) { int result = 1; // In s0 int i; // In s1 for (i = 1; i <= n; i++) result = result * i; return result; }</pre>	<pre>fact: addi sp, sp, -8 # Room on stack sw s0, 0(sp) # Save s0 sw s1, 4(sp) # Save s1 li s0, 1 # result = 1 li s1, 1 # i = 1 for: bgt s1, a0, done # i > n? mul s0, s0, s1 # result *= i addi s1, s1, 1 # i++ j for # Repeat done: mv a0, s0 # Return result lw s1, 4(sp) # Restore s1 lw s0, 0(sp) # Restore s0 addi sp, sp, 8 # Restore stack ret # Return</pre>

An optimizing compiler will observe that there are no function calls within `fact`, so `result` and `i` can be kept in temporary registers rather than needing to save and restore `s0` and `s1`, as shown in Code Example 2.12.

Code Example 2.12 OPTIMIZED FACTORIAL FUNCTION

C Code	Assembly Language
<pre>int fact(int n) { int result = 1; // In t0 int i; // In t1 for (i = 1; i <= n; i++) result = result * i; return result; }</pre>	<pre>fact: li t0, 1 # result = 1 li t1, 1 # i = 1 for: bgt t1, a0, done # i > n? mul t0, t0, t1 # result *= i addi t1, t1, 1 # i++ j for # Repeat done: mv a0, t0 # Return result ret # Return</pre>

Code Example 2.13 illustrates the factorial function again, this time written recursively. Before the function can call itself, it must save the previous return address and argument *n* on the stack.

Code Example 2.13 RECURSIVE FACTORIAL FUNCTION

C Code	Assembly Language
<pre>int factr(int n) { if (n <= 1) return 1; else return n*factr(n-1); }</pre>	<pre>factr: li t0, 1 # For compare bgt a0, t0, else # n > 1 li a0, 1 # Return 1 ret # Return else: addi sp, sp, -8 # Room on stack sw ra, 0(sp) # Save return adr sw a0, 4(sp) # Save n addi a0, a0, -1 # n-1 jal factr # Call factr(n-1) lw t0, 4(sp) # Restore n lw ra, 0(sp) # Restore ret adr addi sp, sp, 8 # Restore stack mul a0, t0, a0 # n * factr(n-1) ret</pre>

2.2 RISC-V Machine Language

Machine language is the binary representation of assembly language. Every assembly language instruction corresponds to a 32-bit machine language instruction stored in the computer's program memory. Fig. I.1 (on the last pages of the book) summarizes the six instruction formats used by RV{32/64}I and an R4 format used for floating-point multiply-accumulate instructions.

RV32I instructions are distinguished by their *op*, *funct3*, and *funct7* fields, as given in Table I.1. The least significant 7 bits of an instruction always contain *op*, which defines the type of instruction so that an instruction decoder knows how to interpret the remaining bits. For example, all R-type instructions have *op* = 51 (0x33). The processor must examine the *funct3* and *funct7* fields to determine which R-type instruction is being executed.

The *rs1*, *rs2*, and *rd* fields are always found in the same bit positions of the instruction if they are needed. Thus, these fields can connect directly to address inputs of the register file without a need for multiplexing, saving time and delay compared with other ISAs.

The immediate bits are sometimes scrambled around the instruction in an odd way, as shown in Fig. 2.3, to keep the other fields in the same locations and to encode each immediate bit in the same instruction bit when possible. This bit swizzling does not impact performance because the immediate bits can be rearranged and extended while the register file is reading a source register.

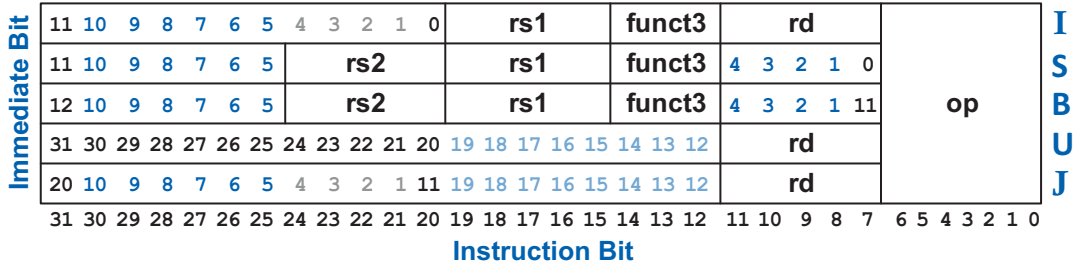


Fig. 2.3 Immediate bits in the instruction.

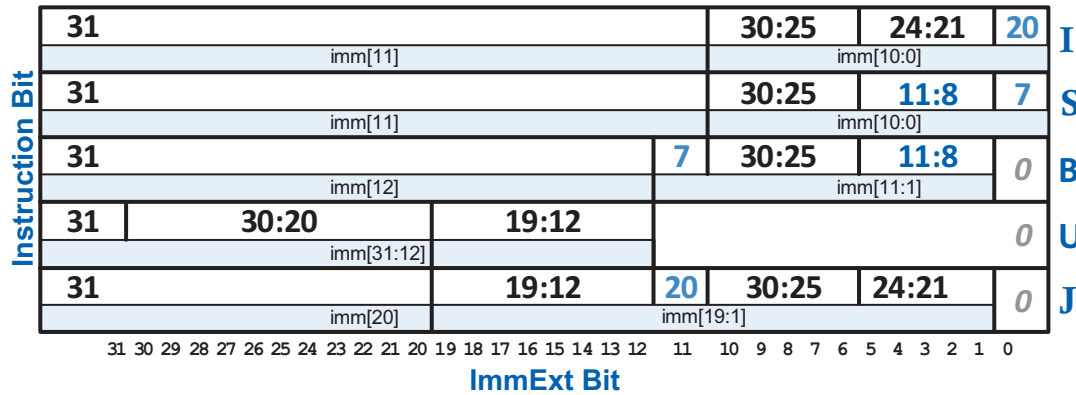


Fig. 2.4 Extension: formation of ImmExt from instruction (Instr) bits.

Fig. 2.4 shows how the extended immediate (ImmExt) is formed (bit-swizzled and extended) from the instruction bits (Instr) in each instruction format. The white and blue squares show how the instruction and immediate bits, respectively, form the 32-bit sign-extended immediate ImmExt. The blue italicized zeros show where ImmExt is hardwired to 0 instead of coming from Instr. Encoding a given immediate bit in the same instruction bit across formats, when possible, minimizes extension hardware; multiplexers can then select a given ImmExt bit from only a few options. For example, $\text{ImmExt}[31] = \text{Instr}[31]$ for all instruction types. $\text{ImmExt}[20] = \text{Instr}[20]$ for U-type instructions, or $\text{Instr}[31]$ for other types. $\text{ImmExt}[0] = \text{Instr}[20]$ for I-type, $\text{Instr}[7]$ for S-type, or 0 for other types.

The most time-consuming part of extending the immediate is sign-extending its most significant bit (msb) into the upper bits because it must drive many loads (i.e., has a large *fanout*). Thus, RISC-V always places the immediate's msb in bit 31 of the instruction, so it is available without further decoding. RV64 also extends bit 31 into bits 63:32 of ImmExt.

The next two examples illustrate how converting between assembly and machine language is a methodical process.

Example 2.1 CONVERT ASSEMBLY TO MACHINE LANGUAGE

Convert `sub x3, x4, x5` to machine language.

Solution: According to Table I.1, `sub` is an R-type instruction with `op = 0b0110011`, `funct3 = 0b000`, and `funct7 = 0b0100000`. The source registers are `rs1 = 4 = 0b00100`, `rs2 = 5 = 0b00101`, and `rd = 3 = 0b00011`. Placing these fields in the order given in the R-type format of Fig. I.1, we obtain:

funct7	rs2	rs1	funct3	rd	op	
0100000	00101	00100	000	00011	0110011	= 0x405201B3

Machine language instructions are generally written in hexadecimal for brevity.

Example 2.2 CONVERT MACHINE TO ASSEMBLY LANGUAGE

Convert RISC-V machine instruction `0x03E90523` to assembly language.

Solution: The instruction in binary is `0000 0011 1110 1001 0000 0101 0010 0011`. The bottom 7 bits are `0100011`, which, according to Table I.1, is an S-type instruction. Rewrite the spaces in the binary number to align with the S-type fields:

imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op
0000001	11110	10010	000	01010	0100011

The S-type instruction with `funct3 = 000` is `sb`. `rs1 = x18 = s2` and `rs2 = x30 = t5`. Cobble together the two halves of the immediate to get `000000101010 = 42`. Hence, the instruction is:

`sb t5, 42(s2)`

You can find a convenient on-line RISC-V assembler at riscvasm.lucasteske.dev. Type in some assembly language code, optionally including pseudoinstructions, and the page will check your syntax and give you the machine language and disassembled code.

2.3 Have a Hart

A *hardware thread* has physical registers containing its context (values of the program counter and registers) and hardware to fetch and execute instructions independently of other hardware threads. A *single-threaded* core has one hardware thread, and a *multithreaded* or *multicore* processor has multiple hardware threads.

A RISC-V *hart* is an abstraction of a *hardware thread*. A hart may be physical, virtual, or simulated. For example, a *physical hart* actually is a hardware thread. An OS may create the illusion of running several threads concurrently by time-multiplexing many *virtual harts* onto a smaller number of hardware threads. A simulator or emulator provides one or more *simulated harts*, possibly running on entirely different underlying hardware. From the program's perspective, all of these kinds of harts are indistinguishable from a true hardware thread except, perhaps, in performance.

Privilege modes provide security and prevent programs from interfering with each other. RISC-V harts may run in machine, supervisor, or user privilege modes. In addition to the ordinary registers, harts have CSRs, some of which are only accessible from higher privilege modes. When a special event such as a system call, illegal instruction, or interrupt occurs, the hart takes a trap to handle the event, often via an execution environment (EE) such as the OS or firmware running at a higher privilege level.

2.3.1 Privilege Modes

RISC-V processors boot in *machine mode*, which is the highest privilege mode and has full control over all processor capabilities. Processors may also optionally support *user* and *supervisor modes*, which provide security by restricting a program's access to certain portions of the memory and CSRs. Processor traps provide controlled access to higher privilege modes.

For example, whereas an OS such as Linux typically runs in supervisor mode, application programs run in user mode. A malicious program shouldn't be able to modify the OS or steal data from another program. Thus, the user-level program can only invoke the OS through a *system call* using the `ecall` instruction, which causes a trap, transferring control to the OS trap handler. The trap handler and system call validates the parameters to reject illegal requests, such as opening a file for which the program lacks permission.

2.3.2 Control and Status Registers

Each privilege mode is associated with a set of CSRs that holds information such as flags from a floating-point operation, which RISC-V features are enabled, and where to go when a trap occurs. The CSRs also include performance counters to track things such as the number of cycles the processor has run, the number of instructions that have retired (i.e., finished successfully), and other implementation-dependent data.

2.3.3 Processes and Threads

A *process* is a running program. In an application processor with a multi-tasking OS, each user process generally has the illusion of owning the entire computer even though there are usually more processes than cores. The processor switches between processes every few milliseconds to give the appearance that all processes are running concurrently. One user process cannot directly interfere with the memory used by another (unless it requests a region of shared memory from the OS).

A process may be divided into one or more *software threads* that share a common memory and code space. Software threads appear to run concurrently. They are useful when a program needs to do several things

The RISC-V Privileged Specification, which describes privilege modes, CSRs, instructions, and traps, is longer and more complex than the main instruction set specification [RISCV_Privileged]. [Chapter 8](#) further describes privileged features.

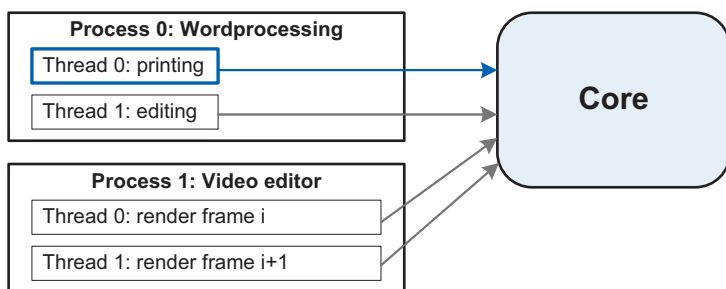


Fig. 2.5 Core running two processes, with two threads per process.

at once (e.g., a word processor spooling to a printer while you are still editing) and to improve performance (e.g., a video editor rendering many frames concurrently on many cores). Fig. 2.5 shows a core running these two processes. Only one thread (in this case, Thread 0, highlighted in blue) runs on the core at any given time. POSIX threads (*pthreads*) are a common type of software thread. Pthreads provide a programming interface to create and destroy threads and synchronize data between threads.

2.3.4 Cores and Harts

A *core* is an individual processor on a chip that fetches and executes instructions. At a minimum, it includes the context (PC, registers, and CSRs) and functional units needed to run a thread. A *multithreaded* core has multiple hardware copies of the PC and registers so that it can quickly switch between executing one thread and another by multiplexing between the contexts for each thread while sharing the common execution units. A *multicore* processor has two or more cores on the same chip, along with an interconnection network and usually an upper-level cache.

Examples of harts, some of which are illustrated in Fig. 2.6, are:

- ▶ In a multicore processor, each RISC-V core is a *physical hart*.
- ▶ A k -way multithreaded core has k physical harts that operate in a time-interleaved fashion. A processor with 4 cores each capable of 2-way multithreading has 8 physical harts.
- ▶ An OS may time-multiplex multiple user-level *virtual harts* onto a smaller number of physical harts. A typical OS has one user-level virtual hart for each user-level software thread. The OS *scheduler* receives timer interrupts, causing it to *context switch* between threads every few milliseconds by saving the context of the current virtual hart to memory and loading the context of the next virtual hart onto the physical hart.

A microprocessor is often partitioned into core and *uncore* portions, with the uncore containing peripherals, shared memory or upper-level caches, and external interfaces.

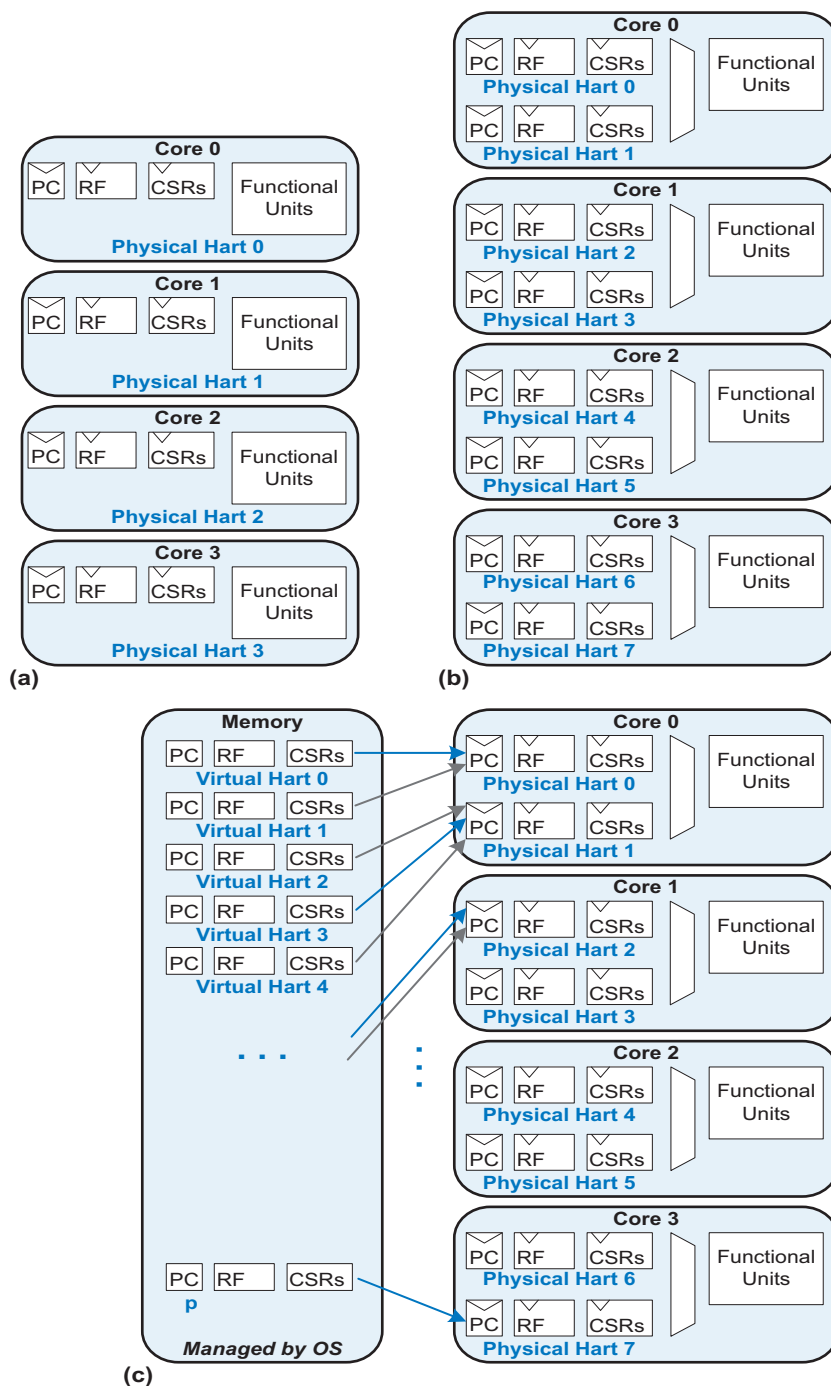


Fig. 2.6 Example harts: (a) multicore, (b) 2-way multithreaded cores, and (c) virtual harts.

- ▶ An emulator such as Spike or QEMU can run RISC-V programs using *simulated harts* on a computer with a different ISA.

2.3.5 Exceptions, Interrupts, and Traps

In RISC-V, an *exception* is an unusual condition caused by a program, such as an illegal instruction, an access to nonexistent memory, or an OS call. An *interrupt* is an asynchronous external event such as a timer reaching some trip point or a serial port indicating that a new piece of data has arrived. Both exceptions and interrupts cause a *trap*, which is a transfer of control from the current program to a *trap handler*, which is code that handles the event. Traps interact closely with the CSRs and also raise the privilege level, going to machine or supervisor mode. When a trap to machine mode occurs, the current program counter is stored in the *machine exception program counter* (`mepc`) CSR. A code indicating the trap's cause is stored in the *machine cause* (`mcause`) CSR. Supplementary information, such as the illegal instruction or illegal address being accessed, may be stored in the *machine trap value* (`mtval`) CSR. The processor changes to machine mode and jumps to the address specified by the *machine trap-vector base-address* (`mtvec`) CSR. Traps to supervisor mode are similar and use `sepc`, `scause`, `stval`, and `stvec`.

The trap handler is code that responds appropriately, for example by performing the OS call, placing the new serial data into a buffer, or terminating the program executing an illegal instruction. A machine- or supervisor-level trap handler returns to the original program using the `mret` or `sret` instruction, which returns to the address in `mepc` or `sepc` and restores the previous privilege level. Traps are discussed further in [Chapter 8](#).

2.3.6 Execution Environment

A hart runs within an *execution environment* (*EE*) such as an OS. An *execution environment interface* (*EI*) allows programmers to interact with its EE and defines the initial state of the program, the number and types of harts, including the privilege modes and instruction set they support, the memory map, and how traps are handled. An EE can be implemented in hardware, software, or a mix of the two.

The simplest EE is *bare metal*, in which the processor runs in machine mode (M-mode) with no OS. The programmer has direct access to all features and memory. Physical harts directly correspond to physical cores. There are no virtual harts and no protection between programs.

[Fig. 2.7](#) shows a more sophisticated EE for a Linux-based OS on a processor that supports machine, supervisor, and user modes (M-, S-, and U-modes, respectively). The user can run multiple concurrent U-mode applications, such as a terminal, email client, compiler, and music player.

This book does not cover hypervisors in detail. But, in general, a *hypervisor* runs more than one *guest OS*, time-multiplexing the physical harts to give each OS its own virtual harts, called *guest harts*. For example, VirtualBox is a hypervisor that allows Mac users to run Windows or Linux as a guest OS. Each OS in turn time-multiplexes multiple user-level virtual harts onto these guest harts.

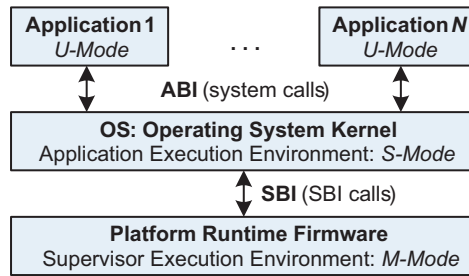


Fig. 2.7 Linux-based execution environments (EEs).

The OS, also called the *kernel*, runs in S-mode and provides each application with its own application EE that has its own virtual memory space so that one application can't interfere with another application's memory.

Fig. 2.7 shows two common EEs, an *application binary interface* (ABI) and a *supervisor binary interface* (SBI). An ABI enables applications to interact with their EE, and an SBI enables the OS to interact with its EE, also called its runtime environment: peripherals, timers, and so on. As will be discussed further in Section 3.3.3, the ABI defines a set of *system calls* for applications to make requests to the OS for managing processes, memory, and files and synchronizing user-level threads. The OS kernel, in turn, uses SBI calls to access peripherals and services such as setting the system timer or starting, stopping, and synchronizing cores, as discussed in Section 3.3.4. The SBI allows the same OS to run on different RISC-V chips by only modifying the platform runtime firmware.

Software has no way to tell whether it is running on a physical, virtual, or simulated hart, except perhaps by performance. For example, consider a Linux OS running on a 4-core dual-threaded processor that has 8 physical harts, because it has 8 distinct contexts (PCs, register files, and CSRs) kept in hardware. The Linux OS runs on these physical harts in supervisor mode. Each user-level (software) thread is assigned to a virtual hart. The OS creates and destroys virtual harts as it spawns and kills software threads, and the OS scheduler time-multiplexes (periodically switches) which virtual hart is running on a given physical hart so that an unlimited number of threads can have the illusion of running on their own processor even though a finite number of physical harts exist. As long as the OS allocates time fairly, each U-mode virtual hart makes progress independent of the others.

As specified by the EEI, a hart typically makes system calls to its EE by placing arguments in the `a0...a7` registers and performing `ecall`. A hart may also transfer control to a debugger by performing `ebreak`. Each of these instructions causes an exception and jumps to the exception handler at `mtvec` or `stvec`.

A purely software EE is a RISC-V *simulator* or *emulator*, such as Spike, Sail, or QEMU, which runs RISC-V programs on underlying hardware such as an x86-based computer. In this case, there are no physical RISC-V cores, so the harts are simulated.

Sometimes the term *simulator* is used for a tool that models only the harts and RAM, while an *emulator* also models *peripherals*, that is, input/output (I/O) devices such as keyboards, serial UART connections, and so forth. Spike is a RISC-V simulator that can run programs without peripherals, and QEMU and Sail are emulators that model the CLINT timer, PLIC interrupt controller, and UART serial port needed to boot Linux. The term *simulator* is used generically to describe all such tools.

Whereas fast simulators such as QEMU use *binary translation* to convert a program from the instruction set being simulated to the native instruction set, simpler simulators interpret programs one instruction at a time. QEMU runs approximately 1 billion instructions per second. Spike runs about 100 million instructions per second. Sail is focused on provable correctness rather than performance and runs only about 1 million instructions per second.

A mixed hardware/software EE can provide capabilities not directly available from the hardware. For example, consider a user-mode program running on hardware that lacks a floating-point unit (FPU). If the program tries to execute a floating-point instruction, an illegal-instruction exception occurs. The exception handler could perform the floating-point arithmetic in software, and then the user-mode program could continue.

2.4 MEMORY MAP

The *memory map*, a key part of the EE, defines the addresses at which memory and I/O peripherals are located. A representative memory map, used in the Wally processor, is shown in Fig. 2.8 (and in Table A.2 in Appendix A). This memory map contains a small boot ROM, a larger RAM, and various peripherals, including:

- ▶ CLINT: Core Local Interrupter, for handling software and timer interrupts
- ▶ PLIC: Platform-Level Interrupt Controller, for multiplexing external interrupts
- ▶ GPIO: General-purpose input/output, for reading and writing binary values from pins
- ▶ UART: Universal asynchronous receiver/transmitter, for serial communication, such as printing to a terminal
- ▶ SPI: Serial Peripheral Interface, for higher-speed clocked serial communication
- ▶ SDC: Secure Digital (SD) Card controller

Spike, Sail, and QEMU are discussed further in Chapter 3.

The field of memory systems is inconsistent on whether address 0 is shown at the top or bottom of a memory diagram. This textbook uses the bottom.

RAM	87FFFFFF
	80000000
GPIO	100600FF
	10060000
SPI	10040FFF
	10040000
UART	10000007
	10000000
PLIC	0FFF0000
	0C000000
CLINT	0200FFFF
	02000000
SDC	00013FFF
	00013000
Boot ROM	00001FFF
	00001000

Fig. 2.8 Wally default memory map.

Wally’s memory map is compatible with the SiFive FU540 processor and the QEMU *virt* machine (*virt* is a QEMU emulation model with peripherals suitable for booting Linux). An exception is that the UART is at 0x10010000 on the FU540 rather than 0x10000000 on Wally and *virt*. See Table 6 of [FU540]. Section 4.4 describes the Wally configuration files, which can be used to add or remove peripherals or change the memory map.

The RISC-V reset vector is platform dependent. In most Wally configurations, the reset vector is 0x80000000 because programs typically start at this address. The testbench automatically (automatically) loads a program into RAM at this address at reset. In the Buildroot Linux configuration and QEMU, the reset vector is 0x1000 to start a Linux boot loader preloaded in boot ROM. On the SiFive FU540 chip, the reset vector is 0x1004 for a bootloader.

Configurations without external RAM map the on-chip memories, data tightly-integrated memory (DTIM), and instruction read-only memory (IROM) to address 0x80000000 instead. Typically, the program begins at 0x80000000 and data follows.

On reset, the PC begins at an address called the *reset vector*, and some instructions must be available at that address. In a real system, the reset vector points to the boot ROM, which contains a *boot loader* that loads the next stage of an OS from the SD card (or flash memory or hard disk, if they are supported). In a simulator, the reset vector could point directly to RAM that was preinitialized on reset with the program of interest.

2.5 RISC-V EXTENSIONS AND PROFILES

The base instruction sets are known as RV32I or RV64I. These operate on 32- or 64-bit registers, respectively. The RISC-V Unprivileged Specification defines several other ratified extensions (see Table 2.2). The M extension supports multiply, divide, and remainder instructions. The F, D, and Q extensions support single-, double-, and quad-precision floating-point math, respectively. All of these could be emulated in software but are much faster with hardware support. The A extension introduces atomic instructions used for synchronization between multiple RISC-V harts. Most application processors support the M, F, D, and A extensions and CSRs and instruction fences, which is written as RV64IMAFD_Zicsr_Zifencei. A processor with all these extensions is abbreviated as RV32G or RV64G.

Table 2.2 Common RISC-V extensions.

Extension	Instructions
A	Atomic read-modify-write for synchronization
B	Bit manipulation (B = Zba + Zbb + Zbs)
C	Compressed (16-bit)
E	Embedded (reduce register set to x0-x15)
F, D, Q	Single-, double-, and quad-precision floating-point math
M	Multiply, divide, and remainder
Zicsr	CSRs and privileged instructions
Zifencei	<code>fence.i</code> instruction/data synchronization
V	Vectors data parallelism
U	User mode
S	Supervisor mode, for OS and virtual memory
H	Hypervisor for multiple concurrent operating systems
G	I, M, F, D, A, Zicsr, Zifencei

Nonvolatile program memory can be the most expensive component in an embedded processor, so these processors usually support a compressed instruction set (similar to the ARM Thumb instructions) that packs most frequently used instructions into only 16 bits. The RISC-V C extension provides these compressed instructions, reducing instruction bits fetched by up to 25% or 30%. RV64GC = RV64IMAFDC_Zicsr_Zifencei contains all of the standard extensions.

RV32E targets tiny embedded cores. The RV32I register file comprises about half the area of a minimal core, so RV32E eliminates registers x16–x31 to save a quarter of the core area.

The Zicsr extension introduces CSRs and instructions to read and write these registers; this is necessary to support traps, multiple privilege modes, and other configuration options. The Zifencei standard extension adds the `fence.i` instruction to synchronize writes and fetches for instruction memory. This is needed by operating systems that load programs into memory.

RISC-V also defines an unratified base instruction set RV128I supporting 128-bit memory addresses by using 128-bit registers, anticipating that the 64-bit addresses of RV64I (which support 16 exabytes of memory) will eventually become insufficient for large computers. RV128I adds quadword loads and stores and instructions to operate on the bottom 64 bits of a 128-bit register. No commercial need yet exists for 128-bit addresses, and the standard may evolve when these large systems are developed.

A large and rapidly expanding set of other extensions have been ratified, such as Zk* cryptographic instructions, Zicbo* cache management instructions, Sstc supervisor timer compare, and so forth.

RISC-V vendors are free to develop their own proprietary nonstandard extensions. Such extensions can provide a competitive advantage. However, vendors also have an incentive to publish and standardize their extensions to increase software adoption.

Some RISC-V systems provide *accelerators* for specialized tasks. An accelerator might be a fixed-function unit such as an H.265 video decompressor or a programmable unit such as a graphics processor. An accelerator typically communicates with the cores through shared memory.

2.5.1 RISC-V Profiles

RISC-V has a great deal of configurability and extensibility. Initially, it used ISA strings such as RV64IMAC to define the supported extensions, but these are becoming unwieldy as more extensions are added with longer names, resulting in strings such as RV64IMAFDC_Zicsr_Zifencei_Zicclsm_Zba_Zkt_Svnapot. RISC-V defines *profiles* to capture standard collections of extensions. Software developers can specify a profile with a minimum feature set needed to run the software. As of the time of this writing, the following profiles have been ratified [RISCV-Profiles]. Table A.6 details the extensions supported in each profile.

All of the RISC-V extensions are optional. A program may read the *machine instruction set architecture* (misa) CSR (see Section 8.2.3.2) to determine which letter extensions are available and then either use an instruction from an extension or call a library function to emulate the missing instruction. RISC-V International is in the process of defining a standard way to determine whether other named extensions (e.g., Zifencei) are supported.

“There is only one mistake that can be made in computer design that is difficult to recover from—not providing enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every computer.” [Bell75]

- ▶ RV120U32: Basic unprivileged RV32I
- ▶ RV120U64: Basic unprivileged RV64I
- ▶ RVA20U64: Application processor RV64GCU
- ▶ RVA20S64: Application processor RV64GCUS
- ▶ RVA22U64: Application processor RV64GCBU
- ▶ RVA22S64: Application processor RV64GCBUS with rich OS
- ▶ RVA23U64: Application processor RV64GCBVU with rich OS
- ▶ RVA23S64: Application processor RV64GCBVUSH with rich OS
- ▶ RVB23U64: Customized processor RV64GCBU
- ▶ RVB23S64: Customized processor RV64GCBUS

All profiles are little-endian. The profiles strongly recommend raising an illegal instruction exception on unimplemented opcodes.

Notice that the profiles standardize U- and S-mode features. Only a small amount of portable firmware runs in machine mode, so M-mode features are not standardized.

Table A.6 summarizes the various required and optional extensions for each profile. Of course, a core may have other extensions beyond the required and optional ones in the profile, but software written for that profile will not take advantage of those extensions. The table also indicates the chapter that describes the extension if it is presently supported by Wally.

RISC-V has many minor optional behaviors, such as whether it has hardware performance monitors (Zihpm), whether virtual memory page table entry dirty bits are set in hardware or via a software exception (Svade), and whether cache blocks are 64 bytes (Zic64b). The RISC-V profiles gave these behaviors names in March 2023 so that platforms can rely on consistent behavior.

2.6 COMPARISON WITH OTHER ARCHITECTURES

RISC-V is a fifth-generation RISC architecture, with a basic instruction set similar to other RISC processors. Its most important strength is that it is an open architecture that companies can implement without license fees and researchers can freely improve. A minor strength is that the machine language encodings keep all register fields and the sign bit of the immediate in the same position for all instructions, slightly reducing hardware and the decode critical path. RISC-V avoids various features that have not aged well in previous architectures, including those in ARM, MIPS, SPARC, and x86 architectures.

Such overly clever features in the 32-bit ARMv7 architecture include support of *conditional execution* of instructions, allowing a variable shift before ALU and memory instructions, and ALU and store instructions supporting three source registers. These features lead to more complex instruction encoding, leaving only enough remaining bits for encoding 16 registers, which has proved to be a performance limiter. The shifter is lightly used but impacts the critical path for all instructions. The conditional execution and third register file port increase the expense of building superscalar out-of-order processors. RISC-V eliminates all of these features. ARM recognized the same limitations of its instruction set and fixed some of them with its

64-bit ARMv8 ISA in 2013. ARMv8 has 32 64-bit registers and generally has many similarities to the RISC-V ISA. The 64-bit ARMv8 architecture has an irregular machine language encoding that is cruel for humans to manually translate; this is not a problem for assemblers, although it makes it difficult to teach in an introductory course. ARM processors are presently used in almost all mobile devices. ARM does not manufacture its own chips but rather licenses its synthesizable cores to other companies building SoCs. It also sells architecture licenses, so companies can design their own ARM cores. Apple has demonstrated with its M1 and subsequent chips that ARM processors can match or exceed the performance of x86 processors while consuming much less power. ARM's revenue is based on royalties from licensees, so it tightly controls access. ARM sued one of its biggest customers, Qualcomm, over a licensing dispute.

MIPS is similar to RISC-V. However, the MIPS architecture has a *branch delay slot*, meaning the instruction after the branch is always executed even if the branch is taken. This improved performance of the original pipelined implementation because it avoided flushing the next instruction when a branch was taken, but it proved difficult for microarchitects and compilers in higher-performance implementations [Hennessy19]. As ARM grew to dominate the market, MIPS market share has steadily declined and is now negligible.

SPARC has *register windows*, in which the hardware automatically rotates through a larger set of registers during function calls, so that the program does not have to save and restore registers on the stack. This worked well for benchmark programs on a pipelined microarchitecture with a poor compiler but is unnecessary with a better compiler, degrades performance on multitasking loads that have to save the registers during context switches, and is costly in high-performance implementations [Magnussen97]. SPARC is now primarily used by Oracle servers and has a tiny market share.

The Intel x86 complex instruction set computer (CISC) architecture evolved piecemeal from the 8088/8086 to today's modern processors. The complicated legacy instructions, variable-length encoding, and limited set of registers make high-performance x86 microarchitectures difficult. Since the Pentium Pro in the mid-1990s, Intel decoded the complex instructions into simpler micro-ops (RISC-like instructions) and then executed these micro-ops on superscalar out-of-order datapaths [Colwell02]. The variable-length decoder and micro-op cache add hundreds of thousands of transistors compared to a RISC processor but deliver comparable performance [Isen09]. Nearly all Windows PCs use x86 processors. Intel and AMD have a duopoly on the architecture and can, thus, charge high prices.

The greatest present drawbacks of RISC-V are its lack of binary compatibility with existing x86 and ARM software, its less-mature software and verification ecosystem, and its less-mature suite of implementations. However, software, verification, and implementations are advancing rapidly. The open architecture has potential to surpass its proprietary competitors in many markets. This is a new golden age to be a processor designer.

2.7 RISC-V MICROARCHITECTURE

Recall that microarchitecture is the interconnection of gates and registers that implement an architecture. Many microarchitectures may exist for a given architecture. Design teams write a *microarchitecture specification* describing the microarchitecture in enough detail that a logic designer can write hardware description language (HDL) code based on the spec. This section and the next explore microarchitectures for a simplified RV32I core that supports the following instructions:

- ▶ ALU Instructions: `add, sub, and, or, slt, addi, andi, ori, slti`
- ▶ Memory Instructions: `lw, sw`
- ▶ Branch Instructions: `beq, jal`

All microarchitectures must include the architectural state (memory, PC, and registers). Some microarchitectures may include other nonarchitectural state, such as pipeline registers, caches, or branch history, to make the processor run faster. However, this information could be flushed, and the processor will still restart correctly based on the architectural state. Different microarchitectures offer different cost/performance tradeoffs while running the same ISA.

Performance is measured as the *central processing unit's* (CPU's) time T_{CPU} to run a program of interest with a certain number of instructions. The total program execution time is given by Eq. (2.1) in terms of number of instructions, average number of *cycles per instruction* (CPI), and cycle time (T_c) as:

$$T_{CPU} = \#instructions \times CPI \times T_c \quad (2.1)$$

This section describes single-cycle and pipelined microarchitectures. The single-cycle microarchitecture executes one instruction in each clock cycle; that is, $CPI = 1$. It is simple but slow because the clock period is long. The pipelined microarchitecture uses a 5-stage pipeline so that five instructions can overlap their execution and the clock period T_c can be much shorter. This comes at the expense of pipeline registers and hardware to detect for dependencies and resolve them through forwarding and stalls; the stalls increase the CPI somewhat. The microarchitectures are described with block diagrams, truth tables, and HDL (SystemVerilog) code. We also describe a simple test program to verify basic functionality.

2.7.1 Single-Cycle Microarchitecture

Fig. 2.9 shows a block diagram of a single-cycle microarchitecture. The core is partitioned into an *instruction fetch unit* (IFU), *integer execution unit* (IEU), and *load/store unit* (LSU). The IFU contains the PC, IROM, and the logic to increment the PC or handle branches and jumps. The IEU contains the *integer register file* (IRF), the datapath with an ALU,

The microarchitectures are adapted from Chapter 7 of *Digital Design and Computer Architecture, RISC-V Edition*. The main differences are that the ALU and branch comparator have been reorganized to save hardware and that the hierarchical partitioning was changed to facilitate expanding to a full-featured processor.

the branch comparator, and multiplexers to handle arithmetic instructions and memory/branch address generation, and a controller to decode instructions. The LSU contains the *data tightly-integrated memory* (DTIM) used for loads and stores. In these schematics, heavy lines indicate busses of size XLEN (32 for RV32 or 64 for RV64), medium lines indicate narrower multibit signals, and light lines indicate 1-bit signals. The processor fetches an instruction every cycle. Loads and stores each access memory in a single cycle.

The single-cycle processor has two memories and a register file. The register file has three ports: read ports 1 and 2 and write port 3. It receives a 5-bit address for each port (A1, A2, and A3) specifying one of the $2^5 = 32$ registers. The read ports read the specified registers onto the read data ports (RD1 and RD2). The write port needs to know where to write (A3), what to write (WD3), whether to write (WE3, write enable), and when to write (on the rising edge of clock). The memories have a single port. The instruction memory only reads, so it is called the IROM. The DTIM may read or write depending on the write enable WE signal. The write enables for the register file and data memory come from the controller.

The comparator (comp) asserts Eq if $R1 == R2$, for beq tests. The sign extender unit (extend) forms the 32-bit ImmExt from the immediate fields of the instruction (see Figs. 2.3 and 2.4 and Section 7.1.2). The ALU has an adder and various other logical operators. The adder directly drives $IEUAdr = PC + ImmExt$ to the LSU and IFU for fast memory address calculation. ALUResult is selected from the various possible operations according to Table 2.3. The ALUControl signal comprises the {Sub, ALUOp} bits that indicate whether to subtract and whether to consider the funct3 field to determine the operation.

2.7.1.1 Sample Program

Consider running the program from Code Example 2.14 on the RISC-V single-cycle computer starting at address 0x1000. Each line contains the assembly language instruction, the instruction's address, and the machine language encoding.

Code Example 2.14 SAMPLE PROGRAM

Assembly Language	Address	Machine Language
start: lw x3, 4(x1)	# 0x1000	0x0040A183
sub x5, x3, x2	# 0x1004	0x402182B3
beq x0, x0, 0x1000	# 0x1008	0xFE000CE3

All numbers in Code Example 2.14 are 32-bit values and are expressed in hexadecimal; the 0x prefix is sometimes omitted for brevity. Binary control field values are indicated

A processor with a unified memory containing instructions and data is called a *von Neumann architecture*, based on John von Neumann's report on the EDVAC [vonNeumann45]. A processor with separate instruction and data memories is called a *Harvard architecture*, based on the Harvard Mark I that kept data in electronic storage and programs on a paper tape [Aiken46]. von Neumann architectures are more flexible because they allow for not only loading but also modifying programs, but they are slower because separate cycles are needed to access an instruction and then the data. Modern computers generally gain the advantages of both architectures by using a single unified main memory but separate instruction and data caches. The separate caches give single-cycle access to both instructions and data, while the unified main memory means that ordinary load and store instructions can be used to copy or modify instructions in the same way as data. Computers need a mechanism to update the instruction cache when changes are made to cached instructions. RISC-V uses the fence.i instruction to force this synchronization, as described in Section 10.4.2.6.

The DTIM is called *tightly integrated* because it connects directly to the IEU address and data busses without the overhead of an external bus protocol.

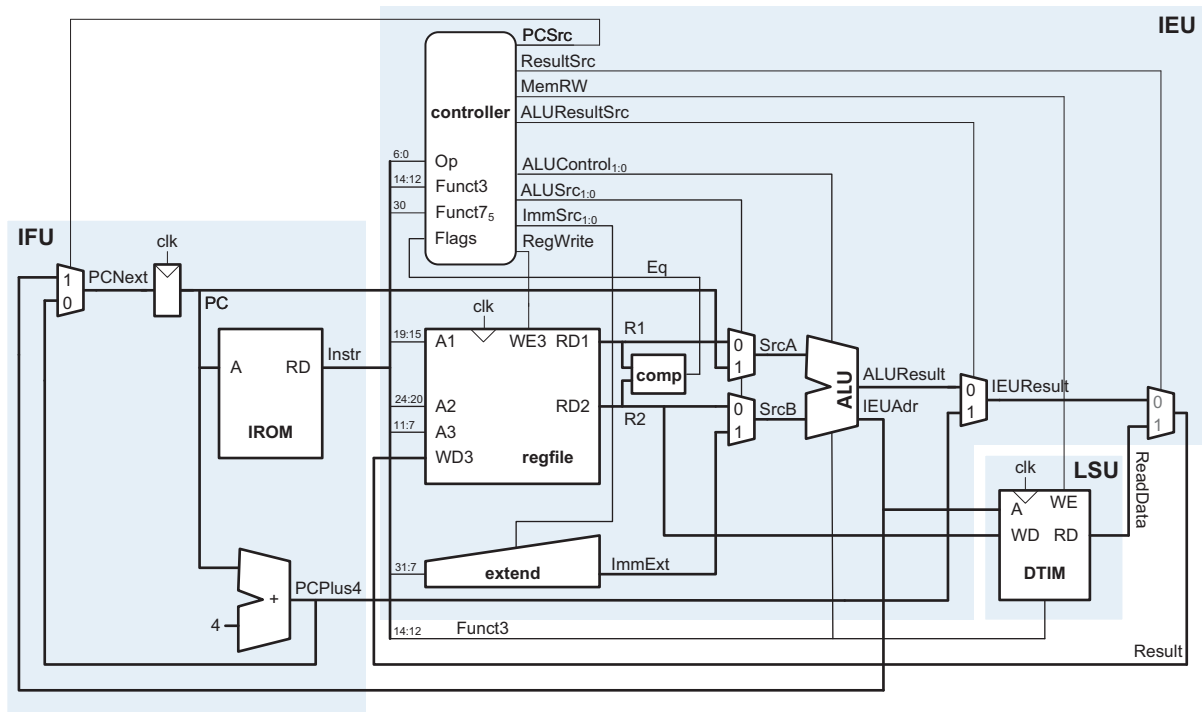


Fig. 2.9 Single-cycle processor block diagram.

Table 2.3 ALU operation.

ALUControl[1] (Sub)	ALUControl[0] (ALUOp)	Funct3	ALUResult
0	0	X	ADD
0	1	000	ADD
0	1	110	OR
0	1	111	AND
1	1	000	SUB
1	1	010	SLT

with a subscript 2. Suppose $x1$ initially contains 0x2000, $x2$ contains 7, and memory address 0x2004 contains the value 0x2A. The code executes as follows:

Cycle 1: $lw\ x3, 4(x1)$

- PC = 0x1000, so the instruction memory (the IROM) reads the lw instruction (0x0040A183) from address 0x1000.

- ▶ $rs1 = 00001_2 = 1$ (from instruction bits 19:15). The register file receives this number at its A1 port and reads the contents of x1 (0x2000) onto R1.
- ▶ At the same time, the extend unit extends the immediate field (4) to 32 bits: $ImmExt = 0x000000004$.
- ▶ The controller sets $ALUSrc = 10_2$ to choose $SrcB = ImmExt$ and $SrcA = R1$. The ALU adder computes $IEUAdr = 0x2000 + 4 = 0x2004$.
- ▶ The data memory in the LSU (the DTIM) reads from this address and returns $ReadData = 0x2A$.
- ▶ The controller sets $ResultSrc = 1$ to choose $Result = ReadData$ and sets $RegWrite = 1$ to write 0x2A into register 3 (obtained from rd in bits 11:7 of the instruction) on the rising edge of the clock at the end of the cycle.
- ▶ Meanwhile, $PCPlus4$ is computed as $0x1000 + 4 = 0x1004$, and the $PCSrc$ control signal is 0, so $PCNext = PCPlus4$ is written to PC at the end of the cycle.

Cycle 2: `sub x5, x3, x2`

- ▶ $PC = 0x1004$, so the instruction memory reads the `sub` instruction (0x402182B3) from address 0x1004.
- ▶ The register file receives $A1 = 3$ and $A2 = 2$. It thus reads $SrcA = R1 = 0x2A$ and $SrcB = R2 = 7$.
- ▶ The ALU subtracts: $ALUResult = 0x2A - 7 = 0x23$.
- ▶ The $Result$ mux chooses $Result = ALUResult = 0x23$ via $IEUResult$ to send it to the register file. (Note that the data memory reads from this address, too, but the read data is not used.)
- ▶ The register file writes 0x23 to register 5 and increments PC to 0x1008.

Cycle 3: `beq x0, x0, 0x1000`

- ▶ $PC = 0x1008$, so the instruction memory reads the `beq` instruction (0xFE000CE3) from this address.
- ▶ The register file receives $A1 = 0$ and $A2 = 0$. Register 0 is hardwired to 0.
- ▶ Thus, the comparator compares 0 to 0 and asserts the `Eq` flag to indicate they are equal. Based on the instruction (`beq`) and this `Eq` flag, the controller sets $PCSrc = 1$.
- ▶ Meanwhile, the extend unit produces $ImmExt = 0xFFFFFFFF8 = -8$. The ALU's adder computes $IEUAdr = PC + ImmExt = 0x1008 + -8 = 0x1000$.
- ▶ The PC multiplexer chooses $PCNext = IEUAdr$. $RegWrite$ is 0, so no registers are written.
- ▶ At the end of the cycle, PC is updated to 0x1000. The processor repeats this loop indefinitely.

2.7.1.2 Controller Design

As we saw in the sample program, the controller must produce appropriate control signals to drive the multiplexers and write enable signals in the datapath to execute the current instruction. Fig. 2.10 shows a block diagram of the controller. The main decoder computes most of the control signals based on the `op` field of the instruction, as given in Table 2.4.

PC normally increments by 4, but it skips to the `IEUAdr` computed by the ALU for jumps and taken branches. The `PCSrc` logic determines whether this skip should occur.

`ALUControl` has an `ALUOp` bit and a `Sub` bit. `Sub` is asserted for subtracts and set less than instructions:

$$\text{ALUControl}[1] = \text{Sub} = \text{ALUOp} \ \& \ ((\text{Funct3} == 3'b000) \ \& \ \text{Funct7b5} \ \& \ \text{Op}[5] \ | \ (\text{Funct3} == 3'b010))$$

$$\text{ALUControl}[0] = \text{ALUOp}$$

Distinguishing subtraction from `add`, `addi`, and other ALU instructions requires checking bits from `funct3`, `funct7`, and the opcode. This chapter uses a lazy decoder that only examines enough bits to distinguish the supported instructions; it produces unpredictable results for illegal instructions. The full Wally decoder uses a precise decoder that raises an illegal-instruction exception for any illegal instruction when exceptions (`Zicsr`) are supported.

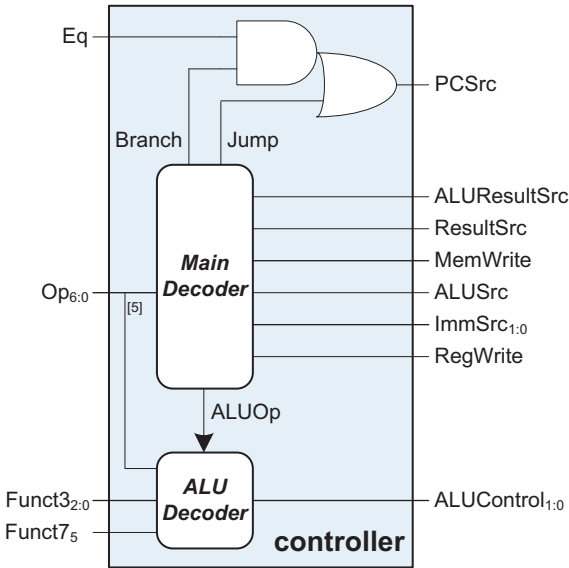


Fig. 2.10 Single-cycle processor controller.

Table 2.4 Main decoder truth table.

Instruction	Op	RegWrite	ImmSrc	ALUSrc	ALUOp	ALUResultSrc	MemWrite	ResultSrc	Branch	Jump
lw	0000011	1	00	01	0	0	0	1	0	0
sw	0100011	0	01	01	0	0	1	0	0	0
R-type ALU	0110011	1	xx	00	1	0	0	0	0	0
I-type ALU	0010011	1	00	01	1	0	0	0	0	0
beq	1100011	0	10	11	0	0	0	0	1	0
jal	1101111	1	11	11	0	1	0	0	0	1

2.7.1.3 SystemVerilog

Code Example 2.15 describes the single-cycle processor in SystemVerilog. SystemVerilog is covered in detail in [Chapter 4](#), but even if you are unfamiliar with the details, you should be able to relate the modules and their inputs and outputs to the objects in the block diagrams.

This code is available in the CORE-V-Wally repository described in [Section 3.2](#) in `cvw/examples/verilog/riscvsingle`.

Code Example 2.15 SIMPLIFIED SINGLE-CYCLE RISC-V PROCESSOR

```
// riscvsingle.sv
// RISC-V single-cycle processor
// David_Harris@hmc.edu
// Sarah.Harris@unlv.edu
// 2021

module riscvsingle(
    input logic clk, reset,
    output logic [31:0] WriteData, IEUAdr,
    output logic MemWrite);

    logic [31:0] PC, PCPlus4, Instr, ReadData;
    logic PCSrc;

    ifu ifu(.clk, .reset, .PCSrc, .IEUAdr, .Instr, .PC, .PCPlus4);
    ieu ieu(.clk, .reset, .Instr, .PC, .PCPlus4, .PCSrc, .MemWrite, .IEUAdr, .WriteData,
        .ReadData);
    lsu lsu(.clk, .MemWrite, .IEUAdr, .WriteData, .ReadData);
endmodule

module ifu(
    input logic clk, reset,
    input logic PCSrc,
    input logic [31:0] IEUAdr,
    output logic [31:0] Instr, PC, PCPlus4);

    logic [31:0] PCNext;
    // Next PC logic
    flopr #(32) pcreg(clk, reset, PCNext, PC);
    adder pcadd4(PC, 32'd4, PCPlus4);
    mux2 #(32) pcmux(PCPlus4, IEUAdr, PCSrc, PCNext);
    irom irom(PC, Instr);
endmodule
```

Continued

Code Example 2.15 SIMPLIFIED SINGLE-CYCLE RISC-V PROCESSOR—Cont'd

```

module irom(input logic [31:0] a,
            output logic [31:0] rd);

    logic [31:0] ROM[63:0];

    initial
        $readmemh("riscvtest.memfile",ROM);

    assign rd = ROM[a[7:2]]; // Word aligned
endmodule

module ieu(
    input logic clk, reset,
    input logic [31:0] Instr,
    input logic [31:0] PC, PCPlus4,
    output logic PCSrc, MemWrite,
    output logic [31:0] IEUAdr, WriteData,
    input logic [31:0] ReadData);

    logic RegWrite, Jump, Eq, ALUResultSrc, ResultSrc;
    logic [1:0] ALUSrc, ImmSrc;
    logic [1:0] ALUControl;

    controller c(.Op(Instr[6:0]), .Funct3(Instr[14:12]), .Funct7b5(Instr[30]), .Eq,
                .ALUResultSrc, .ResultSrc, .MemWrite, .PCSrc,
                .ALUSrc, .RegWrite, .ImmSrc, .ALUControl);
    datapath dp(.clk, .reset, .Funct3(Instr[14:12]),
                .ALUResultSrc, .ResultSrc, .ALUSrc, .RegWrite, .ImmSrc, .ALUControl, .Eq,
                .PC, .PCPlus4, .Instr, .IEUAdr, .WriteData, .ReadData);
endmodule

module controller(
    input logic [6:0] Op,
    input logic Eq,
    input logic [2:0] Funct3,
    input logic Funct7b5,
    output logic ALUResultSrc,
    output logic ResultSrc,
    output logic MemWrite,
    output logic PCSrc,
    output logic RegWrite,
    output logic [1:0] ALUSrc, ImmSrc,
    output logic [1:0] ALUControl);

    logic Branch, Jump;
    logic Sub, ALUOp;

    logic [10:0] controls;

    // Main decoder
    always_comb
        case(Op)
            // RegWrite_ImmSrc_ALUSrc_ALUOp_ALUResultSrc_MemWrite_ResultSrc_Branch_Jump
            7'b0000011: controls = 11'b1_00_01_0_0_0_1_0_0; // lw

```


Code Example 2.15 SIMPLIFIED SINGLE-CYCLE RISC-V PROCESSOR—Cont'd

```

7'b0100011: controls = 11'b0_01_01_0_0_1_0_0_0; // sw
7'b0110011: controls = 11'b1_xx_00_1_0_0_0_0_0; // R-type
7'b0010011: controls = 11'b1_00_01_1_0_0_0_0_0; // I-type ALU
7'b1100011: controls = 11'b0_10_11_0_0_0_0_1_0; // beq
7'b1101111: controls = 11'b1_11_11_0_1_0_0_0_1; // jal
default:    controls = 11'bx_xx_xx_x_x_x_x_x_x; // Non-implemented instruction
endcase

assign {RegWrite, ImmSrc, ALUSrc, ALUOp, ALUResultSrc, MemWrite,
       ResultSrc, Branch, Jump} = controls;

// ALU Control Logic
assign Sub=ALUOp & ((Funct3 == 3'b000)&Funct7b5&Op[5])|(Funct3 == 3'b010); // Subtract or SLT
assign ALUControl = {Sub, ALUOp};

// PCSrc logic
assign PCSrc = Branch & Eq | Jump;
endmodule

module datapath(
    input  logic      clk, reset,
    input  logic [2:0] Funct3,
    input  logic      ALUResultSrc, ResultSrc,
    input  logic [1:0] ALUSrc,
    input  logic      RegWrite,
    input  logic [1:0] ImmSrc,
    input  logic [1:0] ALUControl,
    output logic      Eq,
    input  logic [31:0] PC, PCPlus4,
    input  logic [31:0] Instr,
    output logic [31:0] IEUAdr, WriteData,
    input  logic [31:0] ReadData);

    logic [31:0] ImmExt;
    logic [31:0] R1, R2, SrcA, SrcB;
    logic [31:0] ALUResult, IEUResult, Result;

    // Register file logic
    regfile rf(.clk, .WE3(RegWrite), .A1(Instr[19:15]), .A2(Instr[24:20]),
              .A3(Instr[11:7]), .WD3(Result), .RD1(R1), .RD2(R2));
    extend ext(.Instr(Instr[31:7]), .ImmSrc, .ImmExt);

    // ALU logic
    cmp cmp(.R1, .R2, .Eq);
    mux2 #(32) srcamux(R1, PC, ALUSrc[1], SrcA);
    mux2 #(32) srcbmux(R2, ImmExt, ALUSrc[0], SrcB);
    alu alu(.SrcA, .SrcB, .ALUControl, .Funct3, .ALUResult, .IEUAdr);
    mux2 #(32) ieureultmux(ALUResult, PCPlus4, ALUResultSrc, IEUResult);
    mux2 #(32) resultmux(IEUResult, ReadData, ResultSrc, Result);
    assign WriteData = R2;
endmodule

module regfile(
    input  logic      clk,
    input  logic      WE3,
    input  logic [ 4:0] A1, A2, A3,

```

Continued

Code Example 2.15 SIMPLIFIED SINGLE-CYCLE RISC-V PROCESSOR—Cont'd

```

input logic [31:0] WD3,
output logic [31:0] RD1, RD2);

logic [31:0] rf[31:1];
// 3-ported register file
// Read 2 ports combinationaly (A1/RD1, A2/RD2)
// Write port 3 on rising edge of clock (A3/WD3/WE3)
// Register 0 hardwired to 0

always_ff @(posedge clk)
    if (WE3) rf[A3] <= WD3;

assign RD1 = (A1 != 0) ? rf[A1] : 0;
assign RD2 = (A2 != 0) ? rf[A2] : 0;
endmodule

module extend(
input logic [31:7] Instr,
input logic [1:0] ImmSrc,
output logic [31:0] ImmExt);

always_comb
    case(ImmSrc)
        // I-type
        2'b00: ImmExt = {{20{Instr[31]}}, Instr[31:20]};
        // S-type (stores)
        2'b01: ImmExt = {{20{Instr[31]}}, Instr[31:25], Instr[11:7]};
        // B-type (branches)
        2'b10: ImmExt = {{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0};
        // J-type (jal)
        2'b11: ImmExt = {{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0};
        default: ImmExt = 32'bx; // Undefined
    endcase
endmodule

module cmp(
input logic [31:0] R1, R2,
output logic      Eq
);

    assign Eq = (R1 == R2);
endmodule

module alu(
input logic [31:0] SrcA, SrcB,
input logic [1:0] ALUControl,
input logic [2:0] Funct3,
output logic [31:0] ALUResult, IEUAdr);

logic [31:0] CondInvb, Sum, SLT;
logic        ALUOp, Sub, Overflow, Neg, LT;
logic [2:0]  ALUFunct;

assign {Sub, ALUOp} = ALUControl;
// Add or subtract
assign CondInvb = Sub ? ~SrcB : SrcB;

```

Code Example 2.15 SIMPLIFIED SINGLE-CYCLE RISC-V PROCESSOR—Cont'd

```

assign Sum = SrcA + CondInvb + Sub;
assign IEUAdr = Sum; // Send this out to IFU and LSU
// Set less than based on subtraction result
assign Overflow = (SrcA[31] ^ SrcB[31]) & (SrcA[31] ^ Sum[31]);
assign Neg = Sum[31];
assign LT = Neg ^ Overflow;
assign SLT = {31'b0, LT};

assign ALUFuncnt = Funct3 & {3{ALUOp}}; // Force ALUFuncnt to 0 to Add when ALUOp = 0
always_comb
    case (ALUFuncnt)
        3'b000: ALUResult = Sum;           // add or sub
        3'b010: ALUResult = SLT;           // slt
        3'b110: ALUResult = SrcA | SrcB;   // or
        3'b111: ALUResult = SrcA & SrcB;   // and
        default: ALUResult = 'x;
    endcase
endmodule

module lsu(
    input logic clk, MemWrite,
    input logic [31:0] IEUAdr, WriteData,
    output logic [31:0] ReadData);

    logic [31:0] RAM[63:0];

    assign ReadData = RAM[IEUAdr[7:2]]; // Word aligned
    always_ff @(posedge clk)
        if (MemWrite) RAM[IEUAdr[7:2]] <= WriteData;
endmodule

```

2.7.2 Pipelined Microarchitecture

Fig. 2.11 shows how to convert the single-cycle processor into a classic five-stage pipeline by adding registers between stages. The processor executes an instruction across these stages as follows:

- ▶ **Fetch:** Reads the instruction from the instruction memory (IROM)
- ▶ **Decode:** Decodes the control signals and reads the source registers from the register file
- ▶ **Execute:** Performs the ALU operation
- ▶ **Memory:** Accesses the data memory to read or write data
- ▶ **Writeback:** Writes the result to the register file

Each signal is now annotated with a suffix corresponding to its pipeline stage. Notice how the control signals are also pipelined so they

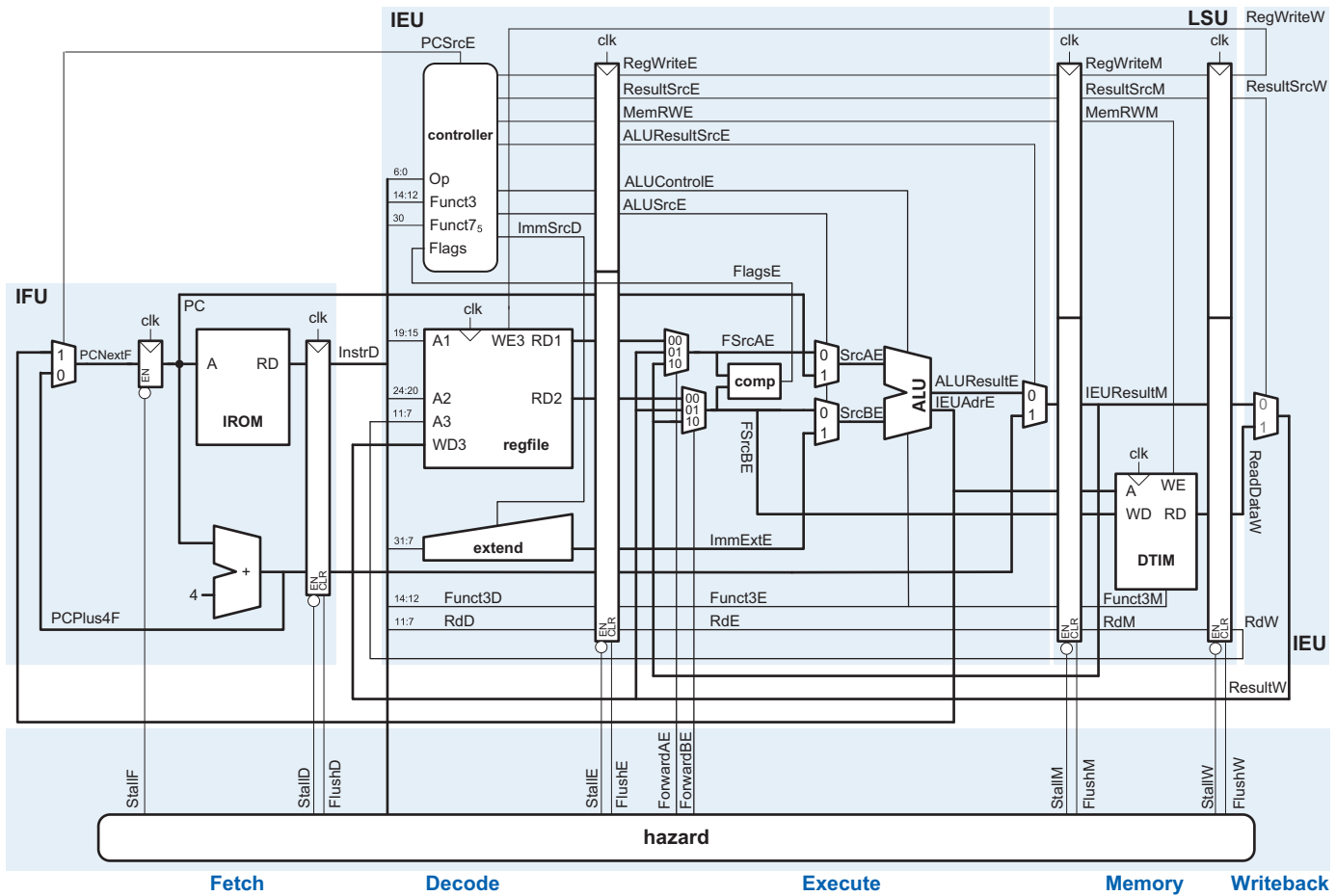


Fig. 2.11 Pipelined processor.

advance along with the instructions they control. Also note how the destination register `rd` must be pipelined so that the result is written to the proper register in the Writeback stage. The register file writes in the first half of a cycle and reads in the second half, so a value can be written and read back out in the same cycle. Only one major operation (ALU, memory, or register file access) occurs in each stage, so the cycle time is substantially faster—ideally, 5x if not for the sequencing overhead of the extra registers and the inability to perfectly balance each stage of the pipeline.

The pipelined processor design is complicated by *pipeline hazards* that arise when one instruction depends on another that is still in the pipeline. The two main types of hazards are data hazards and control hazards. *Data hazards* are caused by pipeline delays in reading and writing registers. Some data hazards can be solved by *forwarding* results from a previous instruction, and others require *stalling* the pipeline. *Control hazards* occur when the decision of what instruction to fetch next has not been made by the time the fetch takes place. A branch predictor addresses control hazards by guessing which instruction should be fetched next; if the guess is wrong, the processor must *flush* the incorrect instructions and restart with the correct one. Fig. 2.11 shows the forwarding multiplexers (muxes) and a hazard unit to control the muxes and issue stalls and flushes. Hazard logic is discussed further in Section 7.2.2.

2.7.3 Verification

All hardware requires logic verification to catch the bugs and gain some confidence that the design is correct. Chapter 5 describes logic verification in more depth. This section uses a simple ad-hoc test program listed in Code Example 2.16 that exercises all types of instructions, including taken and not-taken branches. If the program runs correctly, it should store the value 7 into address 96 and then 25 into address 100. A store with any other address or value indicates failure.

The IEU (integer execution unit) operates in the Decode, Execute, and Writeback stages of the pipelined processor.

A mark of an inexperienced engineer is to expect one's code is correct on the first try.

Code Example 2.16 TEST PROGRAM

```

# riscvtest.s
# Sarah.Harris@unlv.edu
# David_Harris@hmc.edu
# 2025
#
# Test the RISC-V processor.
# add, sub, and, or, slt, addi, lw, sw, beq, jal
# If successful, it should write the value 25 to address 100

```

	RISC-V Assembly	Description	Address	Machine Code
main:	addi x2, x0, 5	# x2 = 5	0	00500113
	addi x3, x0, 12	# x3 = 12	4	00C00193
	addi x7, x3, -9	# x7 = (12 - 9) = 3	8	FF718393
	or x4, x7, x2	# x4 = (3 OR 5) = 7	C	0023E233
	and x5, x3, x4	# x5 = (12 AND 7) = 4	10	0041F2B3
	add x5, x5, x4	# x5 = (4 + 7) = 11	14	004282B3
	beq x5, x7, end	# Shouldn't be taken	18	02728863
	slt x4, x3, x4	# x4 = (12 < 7) = 0	1C	0041A233
	beq x4, x0, around	# Should be taken	20	00020463
	addi x5, x0, 0	# Shouldn't happen	24	00000293
around:	slt x4, x7, x2	# x4 = (3 < 5) = 1	28	0023A233
	add x7, x4, x5	# x7 = (1 + 11) = 12	2C	005203B3
	sub x7, x7, x2	# x7 = (12 - 5) = 7	30	402383B3
	sw x7, 84(x3)	# [96] = 7	34	0471AA23
	lw x2, 96(x0)	# x2 = [96] = 7	38	06002103
	add x9, x2, x5	# x9 = (7 + 11) = 18	3C	005104B3
	jal x3, end	# Jump to end, x3 = 0x44	40	008001EF
	addi x2, x0, 1	# Shouldn't happen	44	00100113
end:	add x2, x2, x9	# x2 = (7 + 18) = 25	48	00910133
	sw x2, 0x20(x3)	# Mem[100] = 25	4C	0221A023
done:	beq x2, x2, done	# Infinite loop	50	00210063

The hexadecimal machine code is placed in a text file called `riscvtest.memfile`. The IROM loads this file at startup (see the `irom` module in [Code Example 2.15](#)).

[Code Example 2.17](#) lists the SystemVerilog testbench. The testbench instantiates the top-level module, which could be either the single-cycle or pipelined processor. The testbench generates a clock repeating every 10 time units and applies a reset for the first two clock cycles. The processor runs the test program from the IROM. If all goes well, the testbench detects the write of 25 to address 100 and prints success. Similarly, it prints failure if an incorrect write occurs. The testbench will hang if no writes occur.

Code Example 2.17 SYSTEMVERILOG TESTBENCH

```

module testbench();
  logic      clk;
  logic      reset;

  logic [31:0] WriteData, DataAdr;
  logic      MemWrite;

  // Instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // Initialize test
  initial
  begin
    reset <= 1; # 22; reset <= 0;
  end

  // Generate clock to sequence tests
  always
  begin
    clk <= 1; # 5; clk <= 0; # 5;
  end

  // Check results
  always @(negedge clk)
  begin
    if (MemWrite) begin
      if (DataAdr == 100 & WriteData == 25) begin
        $display("Simulation succeeded");
        $stop;
      end else if (DataAdr != 96) begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule

```

Fig. 2.12 shows waveforms from the simulation. The 32-bit busses are displayed in hexadecimal. The testbench toggles `clk` with a 10-ns period and asserts `reset` for the first two cycles, initializing PC to 0. You can watch PC and `Instr` step through the instructions in accordance with the program from Code Example 2.16. Observe at time 110 ns how PC jumps from instruction address 0x20 to 0x28 because of the `beq` around. Also observe how `SrcA`, `SrcB`, and `Result` line up with the test program. For example, at 45 ns when `PC` = 8, the program adds $12 + (-9) = 3$. `SrcA` is 0x0000000C = 12₁₀; `SrcB` is 0xFFFFFFF7, which is -9 as a two's complement number; `Result` = 3. At time 195 ns, `MemWrite` rises, and the program correctly writes 0x00000019 = 25₁₀ to address 0x00000064 = 100₁₀ and prints "Simulation succeeded".

To simulate with Questa or ModelSim, type:

```

$ cd cvw/examples/
verilog/riscvsingle
$ vsim -do riscvsingle.do

```

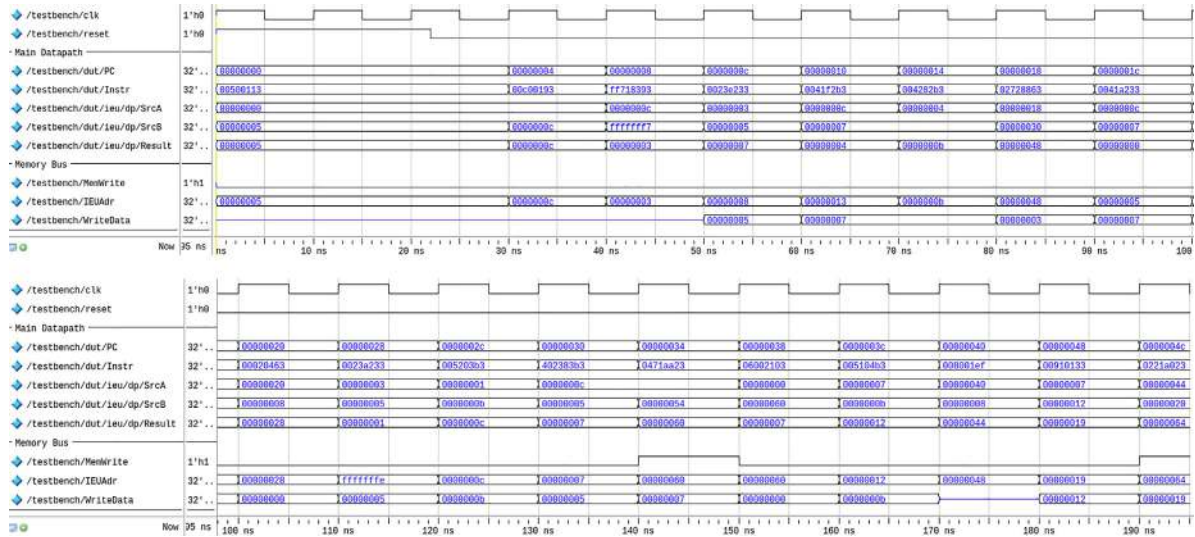



Fig. 2.12 riscvsingle simulation (first and second half).

2.8 WALLY SYSTEM-ON-CHIP

Wally is named in honor of the Harvey Mudd College mascot, Wally Wart, which personifies the flat, square protrusions that adorn the brutalist-style buildings at Harvey Mudd College.



Art by Sarah Harris (adapted from Jeff on Pinterest).

This book explores RISC-V SoC design using an open-source configurable pipelined processor affectionately known as Wally. Wally is described in SystemVerilog. The features are parameterized, and the values are set in a configuration file. Configurations range from a simple RV32E embedded core to a full-featured RV64GC application processor with caches, branch prediction, and virtual memory. Configurable features include those in Table 2.5.

Wally presently does not support:

- ▶ Advanced microarchitecture: superscalar, out-of-order, multithreading, multicore
- ▶ Clock gating and other power reduction techniques
- ▶ Certain extensions
 - V: Vector instructions
 - Zvk*: Vector cryptographic instructions
 - H: Hypervisor
 - Debug and trace interfaces
 - Various recent smaller extensions

Wally (formally CORE-V Wally: cvw) is a project of the OpenHW (Open Hardware) Foundation, which curates and supports robust open-source

Table 2.5 Wally configurable features.

Configurable Feature	Description
XLEN	32 or 64
Primary extensions	
A	Atomic
B	Bit manipulation
C	Compressed
D	Double-precision floating-point
E	Embedded 16 registers
F	Single-precision floating-point
Zk*	Cryptography
M	Multiplication and division
Q	Quad-precision floating-point
Additional extensions	
Zifencei	fence.i synchronization
Zfhh	Half-precision floating-point
Zfa	Additional floating-point
Zmmul	Integer multiplication without division
Zbcb	Carry-free multiplication
Zca, Zcb, Zcf, Zcd	Additional compressed instructions
Zicbom, Zicboz, Zicbop	Cache Management Operations
Zicclsm	Misaligned Loads and Stores
Zihintpause	Pause Hint
Zicond	Conditional zero
Privileged modes and features	
Zicsr	CSRs
Zicntr, Zihpm	Performance counters
Sm1p12	Machine mode (M) version 1.12
Ss1p12	Supervisor mode (S) version 1.12
U	User mode (U)

(Continued)

Table 2.5 Wally configurable features. Cont'd

Configurable Feature	Description
Sstc	Supervisor timer compare
Reset vector address	Typically 0x1000 or 0x80000000
PMP	Physical memory protection
Interrupts	Vectored or nonvectored
Sv32, Sv39, Sv48	Virtual memory
TLB sizes	Instruction and data TLB (ITLB and DTLB) sizes
Svnapot	Huge pages
Svpbmt	Page-based memory types
Svadu	A/D hardware updates
Svinal	TLB Management
Caches	I\$ & D\$ capacity, associativity
Branch Prediction	Gshare or predict not taken
Peripherals	
CLINT	Core Local Interruptor with timers
PLIC	Platform Level Interrupt Controller
UART	Universal asynchronous receiver/transmitter serial port
GPIO	General-purpose input/output pins
SPI	Serial Peripheral Interface
SDC	SD Card
DTIM / IROM	Data tightly-integrated memory, instruction ROM

cores. Many of the cores have evolved from academic projects to industry-supported open products. Other OpenHW cores include:

- ▶ CV32A60AX: RV32IMCA application class 6-stage pipeline based on Ariane
- ▶ CV32E40PV2: RV32IMFC microcontroller 4-stage pipeline based on RI5CY
- ▶ CV32E20: RV32[E][I][M]C simple, low-power microcontroller based on lowRISC Ibex

The Wally repository and how to access it are discussed in [Section 3.2](#).

2.8.1 Wally Block Diagram

[Fig. 2.13](#) shows a high-level block diagram of Wally. It uses the same 5-stage pipeline described in [Section 2.7.2](#). The SoC core connects via a bus interface to the uncore block containing peripherals and memory.

The major blocks in the Wally core (module name: `wallypipelinedcore`) are given in [Table 2.6](#). Main modules have a light blue background and sub-modules are shown within them.

These units are discussed in detail in the remainder of the book. Integer instructions have single-cycle latency and throughput. Loads, multiplies, and most floating-point instructions have single-cycle throughput but 2-cycle latency. A load immediately after a store also must stall for a cycle. Division and cache misses have long latency, and they stall the pipeline. Mispredicted branches are discovered in the Execute stage and thus must flush the next two instructions. Writes to the CSRs flush the pipeline to ensure consistency.

The large memory arrays in the caches and branch predictor are implemented as single-ported edge-triggered RAMs. The smaller multi-ported register files and TLB content-addressable memories (CAMs) are implemented using flip-flops or latches because many libraries for field programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs) lack specialized cells for these structures.

2.8.2 Wally Configurations

[Table A.1](#) lists standard configurations for which Wally has been tested, namely: `rv32e`, `rv32i`, `rv32imc`, `rv32gc`, `rv64i`, and `rv64gc`. The embedded configuration `rv32e` showcases the smallest size and highest frequency achievable in the TSMC 28-nm technology, with no on-chip memories. The simple CPU configurations, `rv32i` and `rv64i`, run only the integer instruction set and have small on-chip memories; for example, the `rv32i` configuration could run simple programs to control other portions of an SoC. The microcontroller configuration `rv32imc` adds a bus, peripherals, privileged modes, and compressed instructions; this configuration is common as a low-cost stand-alone chip to control appliances, toys, and vehicle systems. The application processor configurations, `rv32gc` and `rv64gc`, add the full set of instruction extensions, caches, branch prediction, supervisor mode, and virtual memory needed to boot Linux or similar operating systems. In particular, the `rv64gc` configuration supports the RVA22S64 profile. The more complex designs have a much lower frequency because the trap and floating-point logic is on the critical path. The area increases substantially as memories, floating-point, and branch prediction are added.

Popular open-source licenses include Apache, BSD, GNU, and MIT [opensource.org]. The GNU license is a copyleft license that requires all derivative works be distributed under the same terms, so proprietary derivatives are forbidden. The other licenses are essentially “anything goes,” requiring only attribution and a lack of warranty. Apache is more complex than MIT or BSD but explicitly addresses patent rights. The SolderPad license is an open hardware flavor of the Apache license stewarded by the Free and Open Source Silicon Foundation. Wally uses the SolderPad license because it permissively supports hardware and software development while protecting from patent abuse and because it is the standard for the OpenHW Foundation.

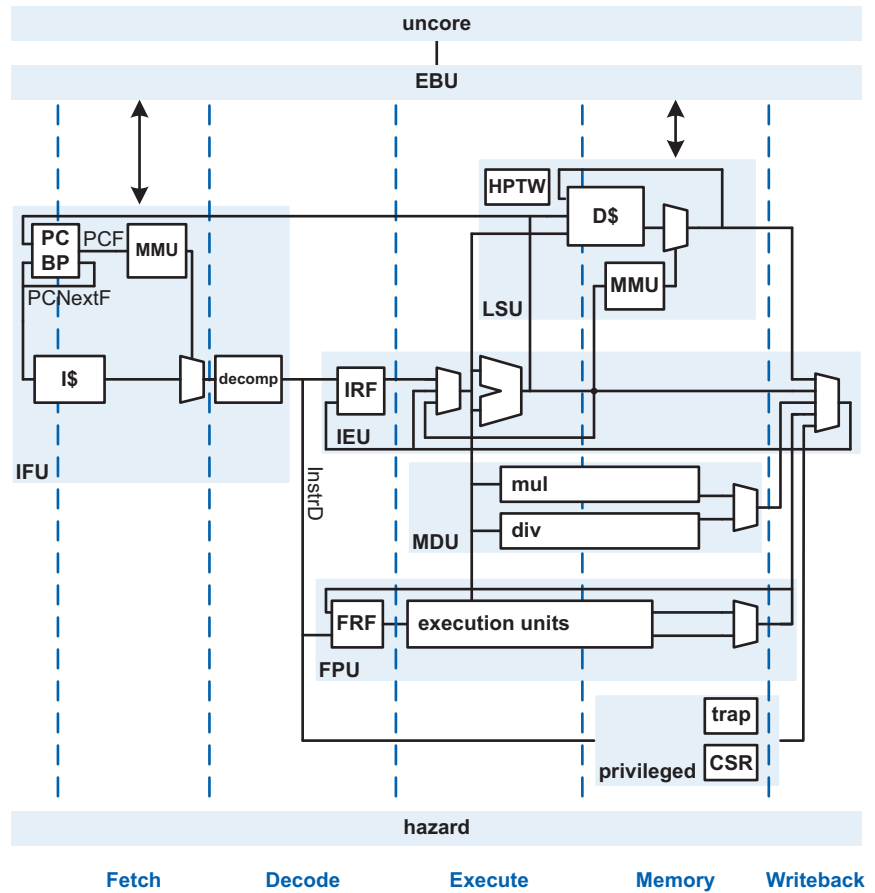


Fig. 2.13 Wally block diagram.

2.8.3 Wally Code Base

The Wally code is maintained in a Git repository at github.com/openhwgroup/cvw. Section 3.2 describes how to clone the repository and compile test cases. Fig. A.1 shows the organization of the repository.

The version described in this textbook is Wally 1.0. We anticipate Wally will continue to improve over time, and changes from Wally 1.0 will be described in the repository. Check out version 1.0 to see the processor exactly as it is in this book.

Table 2.6 Major blocks in the Wally core.

Module Name	Description
ifu	Instruction fetch unit (IFU)
cache or irom	Instruction cache (I\$) or instruction ROM (IROM)
mmu	Memory management unit (MMU)
bpred	Branch predictor
decompress	Decompressor for compressed instructions
in multiple modules	PC logic
ieu	Integer execution unit (IEU)
regfile	Integer register file (IRF)
ieureultmux	Result multiplexer (mux)
controller	Controller, including forwarding and mux selects
alu	Arithmetic logic unit (ALU)
mdu	Multiply/divide unit (MDU)
mul	Multiplier
div	Divider
fpu	Floating-point unit (FPU)
fpreg	Floating-point register file (FRF)
fma	Floating-point multiply/accumulate (FMA)
fdivsqrt	Floating-point divide/square root
fcvt	Floating-point convert
fcmp	Floating-point compare
lsu	Load/store unit (LSU)
cache or dtim	Data cache (D\$) or DTIM
mmu	Memory management unit (MMU)
hptw	Hardware page table walker (HPTW)
ebu	External bus unit (EBU)
hazard	Hazard unit
privileged	Privileged unit
csr	Control and status registers (CSRs)
trap	Trap unit

Experience teaches us that untested features usually have bugs. The verification suites are not comprehensive enough to cover all corner cases, especially of the privileged architecture, so we are confident that bugs remain.

2.8.4 Verification

Wally is verified with a variety of test cases, as listed in [Table 2.7](#). These tests are discussed further in [Sections 3.7](#), [5.7](#), and [A.3](#).

2.8.5 Benchmarking

Given that the logic is correct (or at least passes all test cases), the next step is to optimize for performance. Performance depends on a processor's cycle time and a program's number of instructions and execution cycles, as described in [Eq. \(2.1\)](#). We use benchmarking to measure the number of cycles and, in the next section, explore logic synthesis timing analysis to obtain cycle time. Compiler optimization flags affect the number of instructions. Benchmarks consist of a group of programs that are used to compare the performance of different processors. This section summarizes scores for the classic CoreMark and newer Embench benchmarks. [Chapter 21](#) describes these benchmarks in more depth.

Table 2.7 Test Suites used to verify Wally.

Test Suite	Description
riscv-arch-test	Shows that each instruction in isolation performs its basic functions; provided by RISC-V International's Architecture Test Group
wally-riscv-arch-test	Tests functions not covered by riscv-arch-test, particularly for the MMU and privileged operations
Coverage tests	Exercises code coverage by showing that the lines of Verilog have been exercised by the tests. Functional coverage tests are in progress.
TestFloat	Shows that FPU passes a comprehensive set of tests; from Berkeley
Linux boot	Boots Linux in simulation and on an FPGA, confirming that Wally correctly executes 550 million instructions, including using privileged modes, CSRs, interrupts, peripherals, and virtual memory

Table 2.8 compares CoreMark scores for Wally and other embedded processors. One CoreMark is one iteration per second of the benchmark. Scores are usually reported as CoreMark/MHz to normalize for clock frequency. Wally has similar performance to ARM’s Cortex M4 and OpenHW Foundation’s RI5CY. Cadence’s VeeR-EH2 is a dual-issue superscalar processor and has nearly twice the CoreMark performance of Wally.

Table 2.9 compares Embench 1.0 scores for Wally and other embedded processors, optimized for code size (using compiler flag -Os) and speed (using compiler flag -O2). Size and speed are normalized to the best achieved by an ARM Cortex M4, showing that Wally has similar code density and is about 9% faster at the same clock rate. Even though VeeR-EH2 is a superscalar processor with a tremendous CoreMark score, it is only 22% faster than Wally in Embench, showing that Embench is a more realistic benchmark that is harder to game. RI5CY speed appears suspicious, being equal for -Os and -O2.

2.8.6 Implementation

Wally is designed to balance performance, power, and area rather than favor one at great expense to the others. The SystemVerilog has been synthesized in 130-, 90-, and 28-nm complementary metal-oxide-semiconductor (CMOS) processes and on three FPGAs. The results were reviewed to detect and eliminate grossly inefficient structures caused by poor coding. The design was also tuned to shorten the cycle time by balancing the amount of logic in each pipeline stage and by optimizing cycle-limiting paths. Implementation is discussed further in Chapters 6 and 23.

We hope Wally will continue to evolve. It could be a good platform for evaluating implementation tradeoffs in RISC-V functional units. For example, we hope it can serve as a baseline against which improved implementations can be compared. Most of the SystemVerilog was written by five people working part time over 3 years, and we are confident that many important optimizations still should be made.

Table 2.8 CoreMark scores.

Processor	CoreMark/MHz
Wally (rv32imb)	3.36
Cortex M4	3.42 ^a
RI5CY	3.19 ^b
VeeR-EH2	6.3 ^c
VeeR-EL2	3.9

^awww.eembc.org/viewer/?benchmark_seq=1701

^bwww.pulp-platform.org/docs/RossiGdanskOrconf09-18.pdf

^criscv.org/wp-content/uploads/2019/12/12.11-14.20a3-Bandic-WD_SweRV_Cores_Roadmap_v4SCR.pdf

Table 2.9 Embench scores.^a

Processor	Architecture	Compiler Directives	Size Optimized -Os		Speed Optimized -O2	
			Speed/MHz	Size	Speed/MHz	Size
Wally	RV32IMBC	GCC 13.2.0 -ffunction-sections -fdata-sections	1.01	1.01	1.09	1.18
Cortex M4	ARMV7M	GCC 9.2 -ffunction-sections	0.90	1.00	1.00	1.15
RISCY	RV32IMC	GCC 9.2	0.93	1.05	0.93	1.27
VeeR-EH2	RV32IMACZb	GCC 10.2 -ffunction-sections	1.18	1.11	1.33	1.26

^agithub.com/embench/embench-iot-results

2.8.6.1 FPGA

Chapter 23 describes Wally implementations on a Xilinx FPGA on the Arty A7, VCU108, and VCU118 boards. Fig. 23.3 shows the Arty A7 board annotated with I/O and memory components. The FPGA boots Linux and prints boot messages over the USB cable to a terminal.

2.8.6.2 SoC

Chapter 6 describes logic synthesis with Design Compiler to convert the SystemVerilog description of the processor into a set of logic gates and memories placed on a chip. Fig. 2.14 shows a placement of the rv32gc configuration for a chip in the TSMC 28hpc+ 28-nm process. The components are placed during physical synthesis, but routing has not been performed. The chip area is 0.7 mm on a side. The memory arrays account for the largest part of the area. Each 16-KiB cache has four ways, each comprising four 1-KiB 64×128 data arrays and one 64×22 tag array. The logic blocks are color coded by their unit. Physical synthesis places cells to reduce delay on the critical path, so we observe interesting tentacles where the IFU, LSU, and privileged unit closely interact with the 16 physical memory protection (PMP) checkers. The FPU has the least communication with other blocks, so it is in the corner, near the IEU and LSU. This floorplan is automatically generated during synthesis; each run is different.

Table A.1 summarizes the maximum frequency for each configuration in TSMC28, ranging from 2.99 GHz for rv32e to 0.8 GHz for rv64gc. In this process, a fanout-of-4 inverter (FO4) gate delay is 11.6 ps for typical silicon at nominal (0.9 V) voltage and 25°C, which is a handy metric for normalizing frequency in a process-independent fashion. Hence, the clock

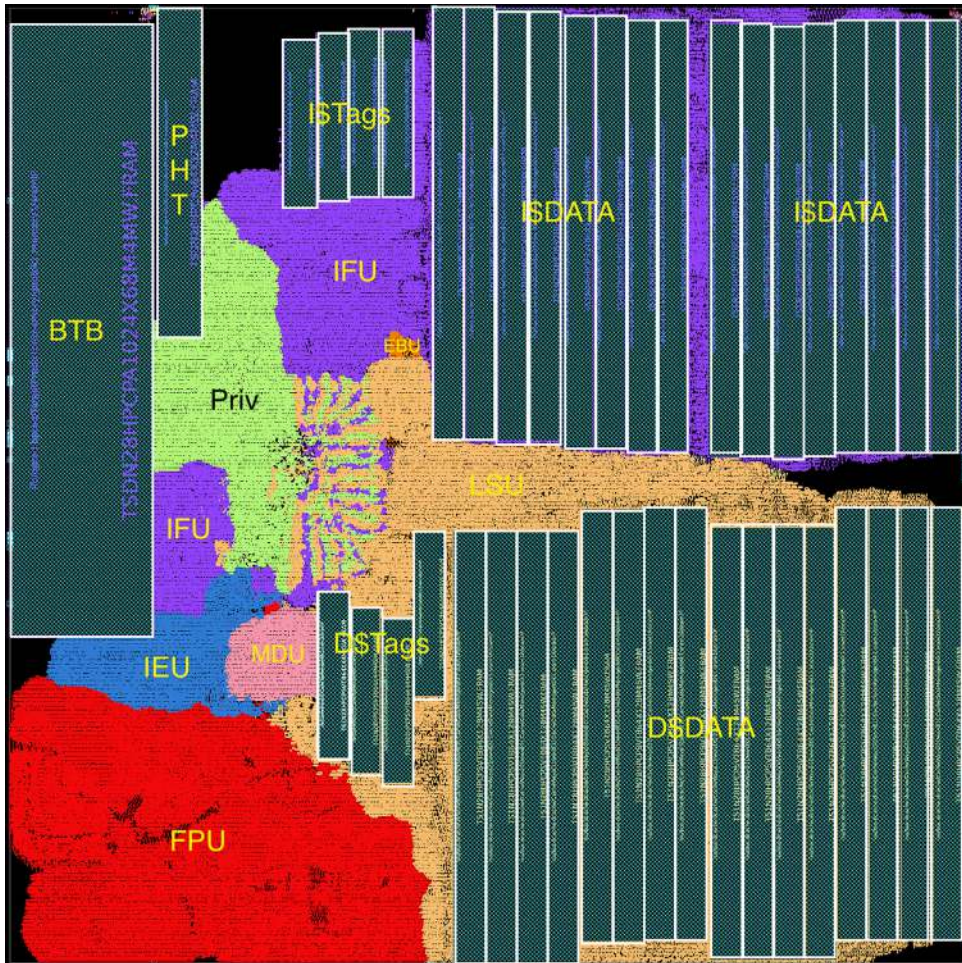


Fig. 2.14 rv32gc wallypipelinedcore layout in TSMC 28 nm.

period (reciprocal of frequency) ranges from 29 to 108 FO4 delays for the various configurations. The 5-stage pipelines are not as aggressive as contemporary high-performance processors with more pipeline stages that target closer to 50 FO4 delays per cycle.

A complete SoC needs bond pads or solder bumps to connect signals to the external world. These signals include clock, reset, peripheral pins, and the AHB bus, as well as power and ground. The core is so small in 28 nm that it is dwarfed by the pads.

2.9 RISC-V MARKET

The RISC-V market has grown explosively since the architecture's inception at Berkeley in 2010. As a clean and open architecture, RISC-V is addressing a huge need in the semiconductor industry. By the end of 2021, more than 12 billion RISC-V cores had shipped in commercial products [RV22]. In 2023, the SHD Group forecasted that the RISC-V market would grow at a 40% compound annual growth rate through 2030, with applications in hard drives, modems, televisions, artificial intelligence, Internet of Things, gaming consoles, data centers, and smartphones.

Most RISC-V cores are developed as intellectual property (IP) blocks in a hardware description language and are licensed or available open source to SoC designers. Intel became a premier member of RISC-V International in 2022 and established a billion-dollar fund to support chip innovation, including fabrication of chips with RISC-V processors. The RISC-V market is evolving rapidly, and it is too early to predict market winners. At the time of this writing, some of the commercial vendors with cores or SoCs for sale or in development include Alibaba, Andes, Codaip, Esperanto, Microchip, Rivos, Semidynamics, SiFive, Synopsys, Tenstorrent, and Ventana. Some of the most widely known open-source cores include the Cadence VeeR (originally Western Digital SweRV) family, Alibaba XuanTie family, and OpenHW Foundation CORE-V family.

Whereas RV32 cores are often used as low-power embedded processors, RV64 cores are often used as multicore application processors with memory-management units running Linux or Android. [Table 2.10](#) through [Table 2.13](#) summarize key attributes of some of these cores. By the time you read this, there will undoubtedly be interesting new cores.

Some RISC-V cores are targeting larger SoCs. For example, Esperanto's ET-SoC-1 is a 7-nm chip with 1088 energy-efficient RV64 ET-Minion cores with vector units and 160-MiB of SRAM for machine learning. The Ventana Veyron V2 data center server SoC contains up to 192 high-performance RV64GCV cores, tightly linked to domain-specific accelerators.

Table 2.10 Commercial RV64 multicore application processors.

Vendor	SiFive	SiFive	SiFive	Andes	Synopsys	Western Digital	Ventana
Product	P650 ^a	P550 ^b	U54 ^c	AX45MP ^d	ARC RPX105 ^e	SweRV EHX3 ^f	Veyron V2
RV64 ISA	RVA22S64	GCV	GC	GCP	RVA23S64	GBC	RVA23
μArchitecture	4 issue Out-of-order	3 issue Out-of-order	1 issue Pipelined	2 issue In-order	2-issue In-order	2-issue In-order	Up to 15 issue Out-of-order
Pipeline stages	13	13	5	8	10	9	
L1 I\$/D\$ (KiB)	32/32	32/32 64 B lines 4 ways	32/32 64 B lines 2–8 ways Configurable as TIM	8–64 64 B lines 1–4 ways + local memory	Up to 64/64	32/32 64 B lines	512/128
L2\$ (KiB)	256	256 64 B lines 8 ways	128	128–2048 64 B lines 16 ways	Up to 128	128–512 64 B lines 8 ways	1536
L3\$ (MiB)	16	1–8	No	No	Up to 16	0–4 16 ways	128/cluster
Branch predictor	Yes	Yes	512-entry BHT 28-entry BTB 6-entry RAS	BHT BTB RAS	Yes	Yes	Yes
Multiplier latency			1 cycle	2 cycles			
Divider latency			3–64 cycles Variable				
MMU	SV39/48 PMP	SV39/48 PMP	SV39 PMP8	SV39/48 32–512 entry TLB PMP16	SV39	SV39	
Privilege levels	MSUH	MSU	MSU	MSU	MSU	MSU	MSUH
Bus	ACE/CHI/AXI	AXI4	AHB/AXI	AXI		AXI, TileLink	CHI 512 GiB/s
Multicore	Up to 16	Up to 8	Up to 4	Up to 4	Up to 16	2/4/8	3.2 cores/cluster 192 cores total
Frequency (GHz)			1.5 in 28hpc			1.8 in 7 nm	3.6 in 4 nm
Area (mm ²)			0.538 in 28hpc				
CoreMark/MHz			3.16	5.5			
SPECINT2006/ GHz	11	8.65				6.2	
Nationality	USA	USA	USA	Taiwan	USA	USA	USA

^awww.sifive.com/cores/performance-p650^bsifive.cdn.prismic.io/sifive/b4f3f7c7-560f-4ecd-857d-715cc6be237b_p550-8mc-data-sheet-2022.pdf^cwww.sifive.com/cores/u54^dwww.andestech.com/en/products-solutions/andescor-processors/riscv-ax45mp/^ewww.synopsys.com/dw/ipdir.php?ds%40arc-v-rpx-100^ftatic.sched.com/hosted_files/riscvsummit2021/38/ehx3_open_noNDA_Nov_18_2021_rev8b.pdf

Table 2.11 Commercial RV32 single-core embedded processors.

Vendor	SiFive	SiFive	SiFive	Andes	Andes	Synopsys	Synopsys
Product	E76 ^a	E31 ^b	E20 ^c	N25F ^d	N22	RHX-100 ^e	RMX-100 ^f
RV32 ISA	IMAFC	IMAC	IMAC	GCB	[I/E]MAC	GCV	IC<MFD>
μArchitecture	2 issue Superscalar	1 issue Pipelined	1 issue Pipelined	1 issue Pipelined	1 issue Pipelined	2 issue Superscalar	1 issue Pipelined
Pipeline Stages	8	5	2	5	2	10	3
L1 I\$/D\$ (KiB)	32 /32 64B lines 2/4 way + TIM	16 KiB I\$/ITIM 64 B line 2 way 64 KiB DTIM	None	8–64 1–4 way + local mem	1–32 1–2 way + local mem	4–128 + DTIM	2–64 I\$ + DTIM
Branch predictor	1.3 KiB BHT 4-entry BTB 2-entry IJTP 3-entry RAS	512-entry BHT 28-entry BTB 6-entry RAS	None	256-entry BHT 32–256 entry BTB 4-entry RAS	optional	Yes	
Multiplier latency	3 cycles	2 cycles	5 cycles	2 cycles			1–9 cycles
Divider latency	3–32 cycles	2–33 cycles	Up to 32 cycles			16 cycles	
MMU	PMP8	PMP8	None	PMP	PMP	SV32/PMP	PMP
Privilege levels	MU	MU	M	MU	M, optional U	MSU	MSU
Bus	AHB/AXI	AHB/AXI	AHB/AXI	AHB/AXI	AHB/AXI	AHB/AXI	AHB/AXI
Frequency (GHz) (28hpc)	1.6 (28hpc)	0.9 (28hpc)	0.725 (28hpc)				
Area (mm ²) (28hpc)	0.14	0.187	0.046				
CoreMark/MHz	5.7	3.0	2.5				
Country	USA	USA	USA	Taiwan	Taiwan	USA	USA

^ausermanual.wiki/m/51f036f7010a69b4bccfd2838ef299e126f18cbb65b682eb9076b35dedbc0e79^busermanual.wiki/Document/SiFiveE31Manualv2p0.20928298.pdf^cstatic.dev.sifive.com/SiFive-E20-Manual-v1p0.pdf^dwww.andestech.com/tw/產品與解決方案/andescore-processors/riscv-n25f/^ewww.synopsys.com/dw/ipdir.php?ds%4arc-v-rhx-100^fwww.synopsys.com/dw/ipdir.php?ds%4arc-v-rmx-100

Table 2.12 Open-source RV64 application processors.

Vendor	OpenHW	T-Head	Berkeley	Madras	lowRISC	BlackParrot (U. Washington+ Boston U.)	This Book (OpenHW)
Product	CVA6 ^a (PULP Ariane)	C910 ^b	BOOMv3 ^c	Shatki C ^d	Muntjac ^e	BlackParrot ^f	Wally (64gc)
RV64 ISA	GC	GC	GC	GC	IMAC	IMAFD	GC
μArchitecture	1 issue Pipelined	3 issue Out-of-order	4 issue Out-of-order	1 issue Pipelined	1 issue Pipelined	1 issue Pipelined	1 issue Pipelined
Pipeline stage	6	12	~ 14	5	5	8	5
L1 I\$/D\$ (KiB)	Up to 32/32	32–64	32/32	Configurable	16/16	32/32	Configurable
L2\$ (KiB)		128–8192	512	N/A	64/core	yes	N/A
Branch predictor	BHT BTB RAS	BHT BTB RAS	TAGE	Gshare	BHT BTB RAS	BHT BTB RAS	Gshare
Multiplier latency	2 cycles			Configurable	2–5 cycles	3 cycles	2 cycles
Divider latency	Up to 64 cycles				34–66 cycles	Configurable	Configurable
MMU	SV39 PMP	SV39 512–1024 entry TLB PMP16		SV39/48 PMP	SV39	SV39	SV39/48 PMP16/64
Privilege levels	MSU	MSU	MSU	MSU	MSU	MSU	MSU
Bus	AXI	AXI		TileLink/AXI	TileLink/AXI	Wishbone	AHB-Lite
Multicore		Up to 4			Yes	Yes	
Frequency (GHz)	1.5 (22 FDX)	2–2.5 (TSMC12nm)	1 (FinFET)		0.08 (Kintex 7 FPGA)		0.9 (TSMC 28hpc)
Area (mm ²)	0.23 (175 kgate)	0.398					0.5
CoreMark/ MHz	2.5	7.1	6.2	2.84	3.01	3.04	3.06
Language	SystemVerilog	Verilog	Chisel3	BlueSpec	SystemVerilog	SystemVerilog	SystemVerilog
Nationality	Switzerland	China	USA	India	UK	USA	USA

^agithub.com/openhwgroup/cva6^bgithub.com/T-head-Semi/openc910^cgithub.com/riscv-boom/riscv-boom^dgithub.com/shaktiproject/cores/c-class/-/tree/master^egithub.com/lowRISC/muntjac^fgithub.com/black-parrot/black-parrot

Table 2.13 Open-source RV32 embedded processors.

Vendor	OpenHW	lowRISC	Chips Alliance	Chips Alliance	Madras	Papon	This Book (OpenHW)
Product	CV32E40P ^a (R15CY)	Ibex ^b (Zero-riscy/ OpenHW/ CV32E20)	VeeR EH2 ^c (Western Digital SweRV EH2)	VeeR EL2 ^d (Western Digital SweRV EL2)	Shatki E ^e	VexRiscv ^f	Wally (32imc)
RV32 ISA	IMFC	{I/E}MCB	IMAC	IMC	IMAC	GC	IMC
μArchitecture	1 issue Pipelined	1 issue Pipelined	2 issue Superscalar dual-threaded	1 issue Pipelined	1 issue Pipelined	1 issue Pipelined	1 issue Pipelined
Pipeline Stage	4	2 or 3	9	4	3	5	5
L1 I\$/D\$ (KiB) and/or DTIM/IROM	N/A	Optional 4 KiB I\$	Optional I\$ + local mem	Optional I\$ 32 or 64 B lines 2 or 4 ways + local mem		1–32 Up to 8 ways + local mem	DTIM + IROM
Branch predictor (entries)	N/A	N/A	32–4096 BHT 32–512 BTB 2–8 RAS	32–4096 BHT 32–512 BTB 2–8 RAS		BHT BTB	N/A
Multiplier latency	1– 5	1	3	1	4	3	2
Divider latency	3– 35	37	34	34	32	34	Configurable
MMU	N/A	PMP16	N/A	N/A	PMP16	SV39 PMP	
Privilegelevels	M	MU	M	M	MU	MSU	M
Bus	OBI	Custom OBI	AXI4	AXI4 or AHB-Lite	AXI4-Lite	AXI	AHB
Frequency (GHz)		0.5 (65 nm)					1.9
Area (mm ²)	70k gate	15– 61k gate	0.067 (16 nm)	0.023 (16 nm)			0.043 (28nm)
CoreMark/MHz	3.19	0.9– 3.1	6.3	4.3			3.34
Language	SystemVerilog	SystemVerilog	SystemVerilog	SystemVerilog	BlueSpec/ Verilog	SpinalHDL	SystemVerilog
Nationality	International	International	USA	USA	India	Switzerland	USA

^agithub.com/openhwgroup/cv32e40p^bgithub.com/lowRISC/ibex^cgithub.com/chipsalliance/Cores-VeeR-EH2^dgithub.com/chipsalliance/Cores-VeeR-EL2^egitlab.com/shaktiproject/cores/e-class/-/tree/master^fgithub.com/SpinalHDL/VexRiscv

2.10 SUMMARY AND A LOOK AHEAD

This book is unusual in that it teaches you microprocessor and SoC design in the context of a real processor implementation. Each principle covered in the book is illustrated in the context of the Wally RISC-V processor. Logic verification is the most difficult and time-consuming part of processor design, so each feature begins with a test plan.

This chapter introduced the RISC-V instruction-set architecture. As a fifth-generation RISC architecture, RISC-V defines the RV32I instruction set with 40 simple instructions and a 32-entry register file of 32-bit registers. Register `x0` is tied to 0. The other register uses are flexible in principle, but the RISC-V ABI convention consistently uses certain registers for certain purposes, such as local variables (`s0-s11`), function arguments and return result (`a0-a7`), the stack pointer (`sp`), function call return address (`ra`), and temporary results (`t0-t6`). All instructions are encoded in a handful of 32-bit formats. The opcode and source/destination registers are in the same location for every format, facilitating a simple and fast decoder. The instructions are summarized below. Most operate on registers or immediates. R-type arithmetic and logical instructions operate on two source registers and place the result in a destination register. I-type arithmetic and logical operations are similar but use one source register and one 12-bit immediate. RISC-V is a load/store architecture, so explicit load and store instructions are used to move data between memory and registers. Conditional branch and unconditional jump instructions control program flow.

- ▶ **R-type arithmetic/logical:** `add, sub, and, or, xor, sll, srl, sra, slt, sltu`
- ▶ **I-type arithmetic/logical:** `addi, andi, ori, xori, slli, srli, srai, slti, sltui`
- ▶ **Loads and stores:** `lw, lh, lhu, lb, lbu, sw, sh, sb`
- ▶ **Branches:** `beq, bne, blt, bge, bltu, bgeu`
- ▶ **Jumps:** `jr, jalr`
- ▶ **Upper immediates:** `lui, auipc`
- ▶ **System instructions:** `ecall, ebreak, fence`

RV64I extends the register file to 64 bits and adds a handful of additional instructions to load/store double words and to operate on the bottom 32 bits of a register. Instructions remain 32 bits long. RISC-V avoids ideas that once seemed clever in other instruction sets but later proved problematic for high-performance implementations.

RISC-V also defines a privileged architecture with three privilege levels, traps, and CSRs. The memory map is system dependent, but a

typical system has main memory starting at address 0x80000000 and perhaps a boot memory at 0x1000, along with various memory-mapped peripherals such as a timer, interrupt controller, and serial port. RISC-V has many standard extensions such as floating-point, which allows the architecture to scale from a simple embedded microcontroller core to a full-featured application processor. RISC-V profiles define a collection of extensions so that a programmer has a standardized target for development.

This chapter illustrated microarchitecture, logic design, SystemVerilog, testbenches, and verification with simple single-cycle and pipelined implementations of a toy processor supporting a subset of RV32I. It introduced Wally, a configurable open-source RISC-V processor and SoC capable of scaling from a tiny RV32I core to a full-featured SoC capable of booting Linux. The remainder of this book illustrates RISC-V SoC design by building up Wally's capabilities.

The next four chapters develop the tool flows for programming and hardware description. [Chapter 3](#) begins with installing the RISC-V compiler and simulator tool chains and the Wally repository. It then covers low-level RISC-V programming in assembly language and C, including simulating with Spike, QEMU, and Sail. Modern digital systems are specified and synthesized using hardware description languages (HDLs), with SystemVerilog leading in industry. [Chapter 4](#) addresses HDL practices, starting with a SystemVerilog tutorial and then covering design methodology and style guidelines. [Chapter 5](#) describes logic verification and debugging using the Wally code base. [Chapter 6](#) describes logic synthesis, mapping Wally configurations onto 130-, 90-, and 28-nm processes to quantify the speed and size of the SoC.

The largest part of this book covers the design of full-featured pipelined processors, including all the critical details usually glossed over in architecture texts. Most chapters begin with principles applicable across computer architectures. They then show how the principles are applied in practice in the RISC-V architecture. They present a test plan showing how the features are verified and then illustrate a real implementation with Wally. [Chapter 7](#) implements the full RV{32/64}I instruction set, first on a single-cycle core and then with a 5-stage pipeline demonstrating hazards, forwarding, stalls, and flushes. Both designs assume ideal local instruction and data memories. [Chapter 8](#) adds the privileged architecture, including privilege modes, traps, and CSRs.

The next five chapters focus on the memory system. [Chapter 9](#) adds a bus interface to fetch instructions and data from real external memories. External memory is slow, so [Chapter 10](#) describes caches to speed up access to frequently used memory. [Chapter 11](#) describes memory management, including virtual memory and memory protection, essential for modern operating systems. [Chapters 12 and 13](#) put this all together to describe the load/store unit (LSU) and instruction fetch unit (IFU) that can access data from caches or internal memory or over the bus while

supporting virtual memory and protection. The IFU also contains a branch predictor, which must be highly accurate to give good performance in a pipelined processor.

The next six chapters describe the optional RISC-V extensions. [Chapter 14](#) covers the compressed 16-bit instruction set, with a decoder that decompresses into standard RISC-V instructions executing on the regular datapath. [Chapter 15](#) covers the integer multiply and divide unit (MDU). [Chapter 16](#) and [Appendix F](#) describe the floating-point unit (FPU), including rarely addressed details of designing and verifying floating-point multiply/add, divide/square root, and other units. [Chapter 17](#) covers atomic instructions, which are important for synchronizing multicore systems. [Chapter 18](#) describes the bit manipulation and cryptography units. [Chapter 19](#) addresses a variety of other smaller extensions.

The final chapters address system design. [Chapter 20](#) describes common peripherals, including timers, interrupt controllers, and serial ports needed to boot Linux. [Chapter 21](#) runs benchmarks to measure the processor performance. [Chapter 22](#) examines how Linux boots on a full-featured processor, and [Chapter 23](#) synthesizes Wally onto a field-programmable gate array to boot Linux on physical hardware.

The appendices cover practicalities of tools and development environments for the SoC designer. [Appendix A](#) provides a synopsis of the Wally design. [Appendix B](#) introduces Linux, especially regarding features helpful for the SoC workflow. [Appendix C](#) covers version control with Git. [Appendix D](#) covers TCL, used to control tools such as logic synthesis. The RISC-V instruction set is summarized for quick reference on the last pages of this book.

This project has kept us and our students awake late at night and early in the morning, wrestling with fascinating puzzles and challenging debug. It has been a thrill to draw together computer architecture, mathematics, operating systems, and digital design to build and optimize a complex system. We hope you share our excitement about demystifying the implementation of a real processor and SoC. The best way to learn this material is to make your own changes and improvements to the system.

Despite the hard work of a good team over three years, we are confident that bugs remain undetected. We appreciate your input pointing out and fixing these errors. GitHub facilitates an open-source development flow in which many people around the world can contribute to a shared code base.

We also see this book as a stake in the ground documenting a reasonable baseline for essential components of a pipelined processor. We look forward to reading your discoveries of better approaches.

RISC-V Software Tool Flow

This chapter lays out the tool flow to work with the RISC-V *toolchain*, which consists of compilers, simulators, and other software development tools run successively to develop and test programs. Fig. 3.1 shows components of the RISC-V toolchain. The GCC (GNU Compiler Collection) assembler and compiler convert assembly language and C source code files into object files in Executable and Linkable Format (ELF). The GCC linker combines object files into an executable file, also in ELF. The executable file can be disassembled back into assembly language. A RISC-V simulator such as Spike or Sail runs the ELF and can log the results.

The chapter begins by describing platform requirements in Section 3.1 and how to clone the Wally git repository in Section 3.2. Section 3.3 describes how to write assembly language code for a bare metal platform and then assemble, link, and disassemble it with GCC driven by a Makefile. It also shows how to simulate the program using Spike. Section 3.4 then describes how to write, run, and disassemble C programs and how to mix assembly and C code using the same tools. Sections 3.5 and 3.6 introduce the formal Sail RISC-V model and simulator and the speedy QEMU

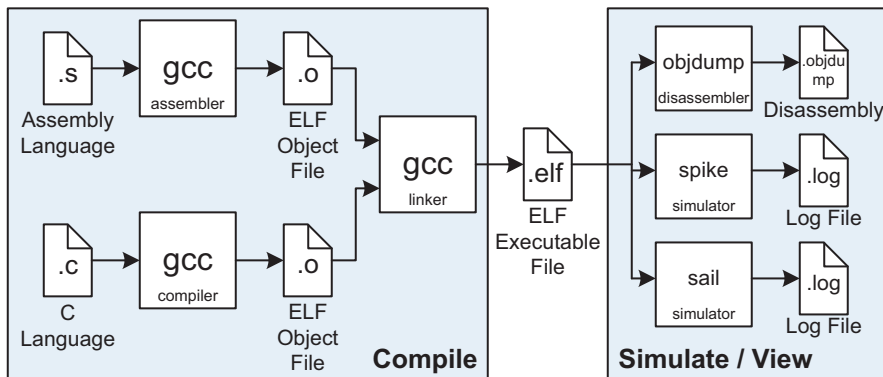


Fig. 3.1 RISC-V toolchain.

- 3.1 [Platform Requirements and Tool Overview](#)
- 3.2 [Getting Started](#)
- 3.3 [GCC Assembler and Spike Simulator](#)
- 3.4 [GCC GNU C Compiler](#)
- 3.5 [Sail Simulation](#)
- 3.6 [QEMU Simulation](#)
- 3.7 [Test Suites](#)
- 3.8 [Summary](#)

Application Software	>"hello world!"
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

The term *compiler* generally refers to a tool that converts a program from a high-level language, such as C, into an executable. The compiler can also refer to just the first step of creating an executable, which is converting the program into assembly; the *assembler* converts the assembly into machine code, and the *linker* then groups programs and libraries into a single executable.

An *instruction set simulator (ISS)*, such as Spike, simulates how instructions running on a core change the architectural state and memory. It does not model peripherals, nor does it attempt to predict the number of cycles the program takes. In contrast, an *emulator*, such as QEMU, also models the behavior of peripherals but still does not accurately predict the number of execution cycles. An emulator is needed to boot Linux because this operating system (OS) depends on peripherals such as a timer, as well as terminal input/output. A *logic simulator*, also called a *cycle-accurate simulator*, such as Verilator or Questa, simulates the actual hardware as defined by the SystemVerilog code; it captures exactly how many cycles are required, accounting for microarchitectural details such as pipeline hazards, cache misses, or branch mispredictions. Logic simulators are orders of magnitude slower than Spike or QEMU, but they are critical in verifying the correct implementation. It is useful to compare the results of an ISS or emulator with a logic simulator to verify the SystemVerilog.

(Quick Emulator). [Section 3.7](#) concludes by describing how to use these tools to build test suites for RISC-V implementations.

3.1 PLATFORM REQUIREMENTS AND TOOL OVERVIEW

Wally uses a Linux-based development flow. Although the tools may work under a variety of shells, the directions are tested on the Bash shell, which is the default on most Linux distributions. The main tools, which are freely available, are:

- ▶ **GCC:** The *software toolchain* for compiling programs to run on Wally
- ▶ **Spike:** RISC-V *instruction set simulator (ISS)*
- ▶ **Sail:** RISC-V ISS, formal architecture model
- ▶ **QEMU:** RISC-V *emulator* for simulating a core with peripherals
- ▶ **Verilator:** Logic simulator and lint

Optional tools are:

- ▶ **Questa:** Logic simulator (Siemens)
- ▶ **VCS:** Logic simulator (Synopsys)
- ▶ **Design Compiler:** Logic synthesis tool for custom chips (Synopsys)
- ▶ **ImperasDV:** RISC-V simulator for lockstep simulation (Synopsys)
- ▶ **Vivado:** Logic synthesis tool for FPGAs (AMD-Xilinx)

This chapter describes the tools for compiling and simulating RISC-V programs using GCC, Spike, Sail, and QEMU. [Chapters 4 and 5](#) introduce hardware design and verification using SystemVerilog and logic simulation and show how to compare the results to the Sail reference. [Chapter 6](#) discusses synthesis targeting a custom chip (application-specific integrated circuit [ASIC]) using Design Compiler, and [Chapter 23](#) describes FPGA implementation using AMD's Vivado. A limited version of Vivado is free, but Questa and Design Compiler require licenses from Siemens and Synopsys, respectively. Both licenses are expensive and only available to companies and universities, not individuals.

This chapter also shows how to use a group of programs, called *test suites*, to verify functionality. Wally uses the following test suites: official RISC-V tests (riscv-arch-test), additional tests we have developed for Wally (wally-riscv-arch-test), and tests for floating-point operations (TestFloat).

3.2 GETTING STARTED

Wally and its tools may be installed and used in one of three ways:

- ▶ On a personal Linux computer, having administrative access to install tools. Clone the repository into a home directory.

- ▶ On a shared Linux server, where an administrator installs shared tools and users clone the Wally repository into their home directories. We expect this to be the most common case for universities and companies.
- ▶ Users install a Docker container that includes Ubuntu and some pre-installed tools.

The directions are likely to evolve over time, so they are part of the README in the CORE-V-Wally (cvw) repository at github.com/openhwgroup/cvw.

The directions include *New User Setup* and *Toolchain Installation and Configuration*. On a shared Linux server, each user performs the New User Setup, and the administrator performs Toolchain Installation. On a personal Linux computer, the user does both.

Wally uses the environment variables `$RISCV` to point to the shared RISC-V tools (typically in `/opt/riscv` or `~/riscv`) and `$WALLY` to point to the cvw repository in a home directory (typically `~/cvw`). The `setup.sh` script sets these variables and others to run RISC-V tools.

The New User Setup directions involve running `make` to build the tests, which takes a while. Then run `regression-wally`. If all goes well, you'll see it print "Success" on each test and then "SUCCESS! All tests ran without failures."

If installing the toolchain on your system is problematic (because of lack of administrator privileges, unsupported OS, general Linux software woes, and so on), the cvw repository `docs/docker.md` describes how to install a Docker container with many of the tools.

3.3 GCC ASSEMBLER AND SPIKE SIMULATOR

GCC is a high-quality and free open-source toolchain that includes cross-compilers to build RISC-V programs from another platform such as an Intel-based Linux computer. The toolchain and its associated software include an assembler and C compiler (`gcc`); the core C library (`newlib` or `glibc`); a debugger (`gdb`); and binary utilities, including a disassembler (`objdump`). Spike is a free and open-source RISC-V ISS originally developed at Berkeley. This section shows how to write, assemble, link, and simulate assembly language programs.

3.3.1 A Quick Tutorial

This section shows how to compile a program using GCC, disassemble it with `objdump`, and then simulate it on Spike. [Code Example 3.1](#) shows a simple RISC-V assembly language program running on bare metal.

Spike is specific to RISC-V. All other tools can be used with other instruction sets or hardware.

The Wally RISC-V flow described in this chapter is tested on currently supported versions of Ubuntu LTS, Red Hat Enterprise Linux, AlmaLinux and Rocky Linux (free clones of Red Hat with no support), Debian Stable, and OpenSUSE Leap.

As of mid-2025, Synopsys and Cadence support their integrated circuit design CAD (computer-aided design) tools on Red Hat Enterprise Linux 8.4+ and 9.1+. Thus, Red Hat is suitable for organizations hosting CAD tools on the same server. Red Hat is expensive for universities, but AlmaLinux and Rocky are free and compatible Red Hat clones. We use AlmaLinux 9.5 in our development. Finding a Linux platform supported by all the necessary and rapidly evolving tools is an ongoing hassle for hardware developers.

Code Example 3.1 `example.S`

```
.section .text.init
.globl rvtest_entry_point # Label rvtest_entry_point viewable by other files
rvtest_entry_point:      # Label expected by linker that indicates start of program
    li a0, 42            # a0 = 42

self_loop:
    j self_loop          # Infinite loop
```

In this book's command line examples, \$ indicates the Linux prompt. Just type the commands in blue; do not type \$.

The program will load 42 into register a0, then enter an infinite loop. The assembler directives starting with a dot define the initial text section (where the program begins) and a global label `rvtest_entry_point` that the linker expects at the start of the program.

Assemble the program by typing the following at a command line, which is fully explained in [Section 3.3.6](#):

```
$ cd $WALLY/examples/asm/example
$ riscv64-unknown-elf-gcc -o example -march=rv32i -mabi=ilp32 -nostartfiles -mcmodel=medany
-T./../link/link.ld example.S
```

C source code, assembly, object, and executable files end in `.c`, `.S`, `.o`, and `.elf` (or nothing), respectively.

Now use `objdump` to *disassemble the object file*, that is, view the compiled machine code in its assembly equivalent. For example, at address `0x80000000`, the machine language instruction `0x02a00513` disassembles to `li a0, 42`.

All GCC tools require the `riscv64-unknown-elf-prefix`.

The compiler name is a hassle to type. It follows a GNU Target Triplet standard of `cpu-vendor-abi` before the tool name. The `cpu` is `riscv64`, although GCC with `multilib` supports any of the RV32 and RV64 flavors using the `-march` flag. For RISC-V, the GCC vendor is normally called *unknown* because GCC is not specific to a particular flavor of Linux. The application binary interface (ABI) is defined by the ELF file. `riscv64-unknown-elf-gcc` uses a lightweight standard C library called *newlib*.

```
$ riscv64-unknown-elf-objdump -D example
```

```
example:      file format elf32-littleriscv
```

```
Disassembly of section .text:
```

```
80000000 <rvtest_entry_point>:
80000000:      02a00513          li      a0,42
```

```
80000004 <self_loop>:
80000004:      0000006f          j      80000004 <self_loop>
```

```
Disassembly of section .riscv.attributes:
```

```
00000000 <.riscv.attributes>:
0:      1941          addi    s2,s2,-16
2:      0000          unimp
...
```

Simulate the program with Spike using the `-d` flag for interactive debug mode. Expect a warning about undefined symbols. Press enter to advance through the program one line at a time. Spike inserts some startup code at address 0x1000 to initialize `a0` and `a1` to some values we don't care about and then jumps to the program at 0x80000000. Typing `reg 0 a0` will print the value of `a0` in hart 0¹, which should be 0x2a (42₁₀) after the `li` instruction. Press `q` to quit when done.

LLVM Clang is the major open-source alternative to GCC. Both are mature mainstream compiler tool chains that produce excellent code. LLVM is more modular. GCC was the first to support RISC-V, and it continues to be a default choice for RISC-V development.

```
$ spike --isa=rv32i -d example
warning: tohost and fromhost symbols not in ELF; can't communicate with target
:
: core 0: 0x00001000 (0x00000297) auipc    t0, 0x0
:
: core 0: 0x00001004 (0x02028593) addi    a1, t0, 32
:
: core 0: 0x00001008 (0xf1402573) csrr     a0, mhartid
:
: core 0: 0x0000100c (0x0182a283) lw      t0, 24(t0)
:
: core 0: 0x00001010 (0x00028067) jr      t0
:
: core 0: 0x80000000 (0x02a00513) li      a0, 42
: reg 0 a0
0x0000002a
:
: core 0: 0x80000004 (0x0000006f) j      pc + 0x0
:
: q
```

3.3.2 Linking and ELF

GCC compiles source code files into *object* files and then automatically *links* one or more object files, libraries, and startup code together to form an *executable* file. The object files contain machine language code and also *unresolved external references* to other object files or libraries. Libraries contain standard functions such as `printf` or `abs`. To save space, the linker only links in the required parts of the library. Startup code may initialize the registers, including the stack and global pointers `sp` and `gp`, as well as set up an exception handler. The linker *relocates* the object files into a file with instructions at consecutive addresses and resolves the references between files to create the executable file.

`riscv64-unknown-elf-gcc` and most Linux-based systems represent both object and executable files as ELF files, which contain a 52- or 64-byte header followed by program and section header tables and the code and data that these headers point to. The file header begins with the magic number 0x7F 45 4C 46, indicating that it is an ELF file, followed by bytes specifying {32/64}-bit format, little or big endianness, the code ISA (instruction set architecture), the entry point address where the program should begin executing, and the size and location of the program and section header tables. Sections are marked by assembler directives, which begin with a

“GNU, a recursive acronym meaning GNU’s not Unix, is the Unix-like OS Richard Stallman set out in 1983 to develop. The aim was to make GNU entirely free (freedom-respecting) software, so it would enable people to shun non-free software and recover lost freedom. GNU became self-hosting after its combination in 1992 with the kernel Linux. See gnu.org/gnu/the-gnu-project.html.”

—Richard Stallman, 2022

¹Spike supports simulating multihart processors, but this simulation only has hart 0.

period, and they include information needed to link object files together into executables, including code (.text), initialized variables (.data), the symbol table (.symtab) that lists the program's labels, and other named sections such as an attributes section (.riscv.attributes). [Section 3.3.7](#) describes assembler directives further.

The `readelf` command prints the *metadata* from the ELF file. The first 16 *magic bytes* identify the type of file. The 45 4c 46 bytes are ASCII for “ELF”. Subsequent metadata bytes describe the format and structure of the ELF file. The program uses 32-bit code (indicated by ELF32) in little-endian format, starting at 0x80000000 (indicated by Entry point address). The section headers list various sections and their locations in the ELF file. The .text section holds the program code. The .symtab section (the symbol table) lists the addresses of symbols (labels) in the program. The .riscv.attributes section (shown at the bottom Attribute Section) indicates that the code requires at least RV32I version 2.0 (indicated by Tag_RISCV_arch: “rv32i2p1”).

```
$ riscv64-unknown-elf-readelf -a example
```

ELF Header:

```

Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:                               ELF32
Data:                                   2's complement, little endian
Version:                             1 (current)
OS/ABI:                               UNIX - System V
ABI Version:                          0
Type:                                 EXEC (Executable file)
Machine:                              RISC-V
Version:                              0x1
Entry point address:                  0x80000000
Start of program headers:             52 (bytes into file)
Start of section headers:             4344 (bytes into file)
Flags:                                0x0
Size of this header:                  52 (bytes)
Size of program headers:              32 (bytes)
Number of program headers:            1
Size of section headers:              40 (bytes)
Number of section headers:            6
Section header string table index: 5

```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	80000000	001000	000008	00	AX	0	0	4
[2]	.riscv.attributes	RISCV_ATTRIBUTE	00000000	001008	00001a	00		0	0	1
[3]	.symtab	SYMTAB	00000000	001024	000070	10		4	5	4
[4]	.strtab	STRTAB	00000000	001094	00002e	00		0	0	1
[5]	.shstrtab	STRTAB	00000000	0010c2	000033	00		0	0	1

Key to Flags:

```

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
D (mbind), p (processor specific)

```

There are no section groups in this file.

```

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x80000000 0x000008 R E 0x1000

Section to Segment mapping:
Segment Sections...
00      .text

There is no dynamic section in this file.

There are no relocations in this file.

The decoding of unwind sections for machine type RISC-V is not currently supported.

Symbol table '.symtab' contains 7 entries:
 Num:   Value   Size Type      Bind   Vis      Ndx Name
   0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
   1: 80000000     0 SECTION LOCAL  DEFAULT    1 .text
   2: 00000000     0 SECTION LOCAL  DEFAULT    2 .riscv.attributes
   3: 00000000     0 FILE    LOCAL  DEFAULT  ABS cc21kp4v.o
   4: 80000004     0 NOTYPE  LOCAL  DEFAULT    1 self_loop
   5: 80000000     0 NOTYPE  GLOBAL  DEFAULT    1 rvtest_entry_point
   6: 80001000     0 NOTYPE  GLOBAL  DEFAULT    1 _end

No version information found in this file.
Attribute Section: riscv
File Attributes
  Tag_RISCV_arch: "rv32i2p1"

```

3.3.3 Application Binary Interface

As introduced in [Section 2.3.6](#), an *application binary interface* (ABI) is the standard that defines how programs communicate during function calls and system calls, including:

- ▶ The binary format of executables and object and library files
- ▶ Which instruction set (ISA) is supported
- ▶ The size, layout, and alignment of data types
- ▶ The calling convention for how arguments are passed to functions in registers or on the stack
- ▶ OS calling conventions
- ▶ Code models of how instructions access a label in the program

An ABI is similar to an API (*application programming interface*). An API defines functions and variables exposed by library code to other programs. For example, the C library API provides functions to open and read files, print to the screen, and perform trigonometric math. An ABI defines conventions used by the compiler when building a program, such as which registers are used to pass inputs to a function.

The RISC-V ABI [RISCV-ABI] supports ELF for executable and object files. It defines a variety of 32- and 64-bit RISC-V ABI calling conventions

Richard Stallman (1953–) is a pioneer of the free software movement and the founding developer of the GNU OS and Free Software Foundation (FSF). Stallman attended Harvard University and became a programmer at the MIT Artificial Intelligence Laboratory. After denouncing proprietary software as vicious and unjust, he defined the *four essential freedoms of free software* [Stallman86].

0. The freedom to run the program as you wish, for any purpose.
1. The freedom to study how the program works and change it so it does your computing as you wish.
2. The freedom to redistribute copies so you can help your neighbor.
3. The freedom to distribute copies of your modified versions to others. By doing this you can give the whole community a chance to benefit from your changes.

Stallman is the founding developer of GCC, the GDB debugger, make, and the emacs text editor.



(Photograph courtesy Richard Stallman. Reprinted with permission.)

summarized in Table 3.1. The name of these data type ABIs use the following convention: i = integer, l = long, and p = pointer. So, ilp32 means that integers, pointers, and longs are each 32 bits; lp64 means that longs and pointers are 64 bits (but ints are still 32 bits). Data types are aligned to address boundaries that are a multiple of their size; for example, address 0x3002 is legal for `char` and `short` but not for `int` or `long`. GCC needs to know the ABI to generate proper code, such as using `lw` vs `ld` to load a `long`. GCC takes a command argument such as `-mabi ilp32` to specify the ABI.

The RISC-V base integer calling convention (ilp32 or lp64) defines how functions pass arguments and use registers. In these ABIs, a function passes up to eight arguments in registers `a0–a7`. These registers need not be preserved. Functions can return results in `a0` (and `a1` if the result is 2XLEN wide). If a function needs more arguments than can be passed in `a0–a7`, the remainder are passed on the stack. The *saved registers*, `s0–s11`, are preserved across function calls and are suitable for local variables; the callee must save and restore any of these registers that it wants to use. `s0` may also be used as a frame pointer to access variables on the stack frame. Registers `t0–t6` are not preserved across a function call and are suitable for temporary (intermediate) results. Register `sp` (`x2`) is the stack pointer, and `ra` (`x1`) holds the return address during a function call. Register `gp` (`x3`) is the *global pointer*, used to access global variables. Table I.4 (on the last pages of the book) summarizes these register uses. The floating-point ABIs (ilp32f, ilp32d, and lp64d) pass `float` or `double` arguments in the floating-point registers `fa0–fa7`.

When running on a platform with an OS and supervisor mode, user-mode programs call the OS by placing the `syscall` number in `a7` and any other arguments in `a0–a5` and then use `ecall` (environment call instruction) to cause a

Table 3.1 RISC-V ABI data type sizes for various calling conventions (in bits).

Data Type	ilp32	ilp32f	ilp32d	lp64	lp64d
int	32				
long	32			64	
pointers	32			64	
char	8				
short	16				
float	n/a	32		n/a	32
double	n/a		64	n/a	64

trap and elevate privilege to supervisor mode. The trap handler in the OS (typically, Linux) kernel receives this `ecall` and looks at `a7` and the arguments to perform the appropriate system call, then places any return result in `a0`, and performs `sret` to return to the user-level program. For example, the `open` system call takes two arguments (in `a0` and `a1`) that specify a pointer to the file path name and flags conveying information such as whether to create a new file if it doesn't already exist. The call returns a *file descriptor*, which is an index into the OS's table of open files; the program can then pass this descriptor to other system calls such as `read`, `write`, or `close`.

The global pointer `gp` points to the static data segment where global variables are held.

```
int open(const char *pathname, int flags);
```

The RISC-V ABI prefers to access global variables using an immediate offset from the global pointer because this is tight and fast code. The two's complement immediate is only 12 bits, so if there are too many global variables, they cannot all be accessed with this short offset (-2048 to $+2047$ bytes). Instead, the code must explicitly calculate the address. RISC-V presently defines two code models: *small* (medlow) and *medium* (medany). The small model uses 32-bit *absolute addresses* and thus is suitable for RV32. It is not position independent, so the linker must change addresses when relocating code. The medium model, suitable for RV64, generates *PC-relative addresses*. It uses the `auipc` instruction to add a signed 32-bit offset to the program counter, and thus can access variables and code within ± 2 GiB of the current instruction. It is position-independent, so the code does not have to be changed if it is relocated.

Code Example 3.2 and Fig. 3.2 illustrate the code generated for a simple C program under these two code models. The compiler placed `main` at address `0x10104`, the global variable `a` at `0x129c0`, and the global variable

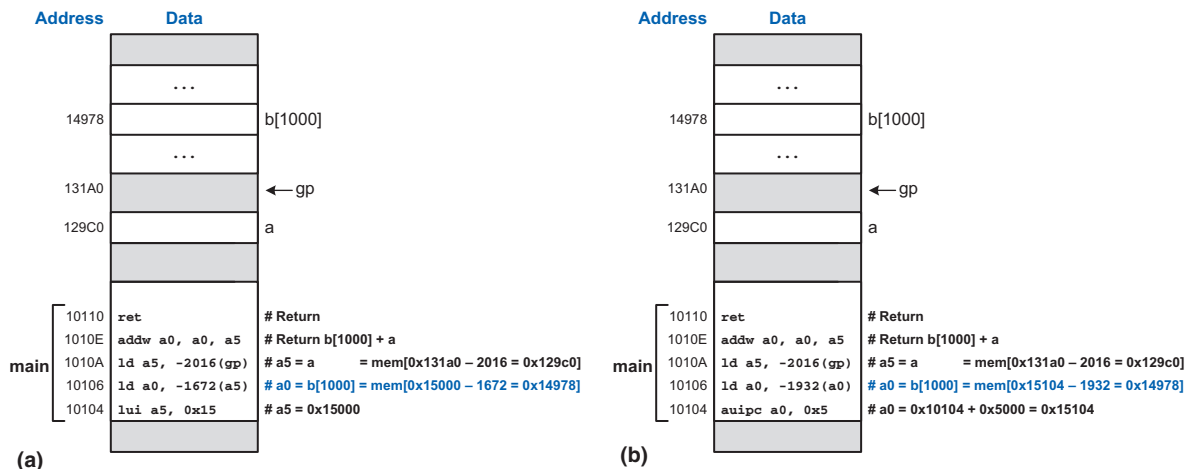


Fig. 3.2 Accessing global variables in (a) medlow and (b) medany code models.

`b[1000]` at `0x14978`. The global pointer is initialized to `0x131a0`. `a` is at 2016 bytes before `gp` ($0x131a0 - 0x129c0 = 0x7E0 = 2016$), so `a` is accessed in both code models with an immediate offset from the global pointer: `ld a5, -2016(gp)`. The `b` array has 2000 8-byte entries and thus is too large to access with an immediate offset.

In `medlow`, the absolute address of `b[1000]` is obtained by `lui` placing `0x15000` in `a5`, followed by `ld` with an offset of $-1672 = -0x688$, giving the absolute address $0x15000 - 0x688 = 0x14978$. In `medany`, the relative address of `b[1000]` is $0x14978 - 0x10104 = 0x4874$ bytes beyond the program counter (PC). This relative address is computed by adding `0x5000` to the PC and then subtracting `0x78c` ($0x5000 - 0x78c = 0x4874$). First, `auipc` adds `0x5000` to the PC ($a0 = 0x10104 + 0x5000 = 0x15104$). Then, `ld` accesses `b[1000]` with an offset of $-1932 = -0x78C$, again giving the address of `b[1000]` ($0x15104 - 0x78c = 0x14978$).

Code Example 3.2 CODE GENERATION WITH MEMORY MODELS

```
// mcmodel.c
long a;
long b[2000];

int main(void)
{
    return a + b[1000];
}

// Makefile
all:
    riscv64-unknown-elf-gcc -O3 -march=rv64gc -mcmodel=medlow mcmodel.c -o mcmodel_medlow
    riscv64-unknown-elf-objdump -d mcmodel_medlow > mcmodel_medlow.objdump
    riscv64-unknown-elf-gcc -O3 -march=rv64gc -mcmodel=medany mcmodel.c -o mcmodel_medany
    riscv64-unknown-elf-objdump -d mcmodel_medany > mcmodel_medany.objdump

// mcmodel_medlow.objdump
000000000010104 <main>:
    10104: 67d5                lui     a5,0x15
    10106: 9787b503           ld      a0,-1672(a5) # 14978 <__global_pointer$+0x17d8>
    1010a: 8201b783           ld      a5,-2016(gp) # 129c0 <a>
    1010e: 9d3d              addw    a0,a0,a5
    10110: 8082              ret

// mcmodel_medany.objdump
0000000000010104 <main>:
    10104: 00005517           auipc   a0,0x5
    10108: 87453503           ld      a0,-1932(a0) # 14978 <__global_pointer$+0x17d8>
    1010c: 8201b783           ld      a5,-2016(gp) # 129c0 <a>
    10110: 9d3d              addw    a0,a0,a5
    10112: 8082              ret
```


Although programs normally follow the ABI, the ISA does not require this. For example, a hardware implementation cannot assume that the return address for a function call is always in `x1`.

3.3.4 Supervisor Binary Interface

The OS kernel can invoke platform runtime firmware via the supervisor binary interface (SBI). This firmware abstracts platform-dependent operations so that the OS does not need to change from one platform to another. Like ABI system calls, each SBI call is identified by a call number. Examples of SBI calls include the following:

- ▶ Set timer
- ▶ Send inter-processor interrupt (IPI, also called software interrupt) to another hart
- ▶ Remote fences to synchronize with other harts
- ▶ Start or stop harts
- ▶ Reboot system
- ▶ Read performance counters
- ▶ Check versions of the hardware and SBI implementation

To make an SBI call, the OS places the SBI call number in `a7` and `a6`, arguments in other `a` registers, and uses the `ecall` instruction, which causes an exception and elevates the privilege to machine mode. The trap handler in the SBI platform runtime firmware receives this `ecall`; does the necessary machine-mode work; places the error code, if any, in `a0` and the return result in `a1`; and performs `mret` to return to the OS. For example, the `sbi_set_timer` function sets the current value of the timer. It is called with an EID (extension ID) of `0x54494D45` in `a7`, an FID (function ID) of `0` in `a6` and a new 64-bit time value in `a0` (and `a1` on RV32). The timer is part of a memory-mapped peripheral such as the Core-Local Interruptor (CLINT) which can only be accessed in machine mode and whose address may be system dependent. For example, the SBI call to set the timer is:

```
struct sbiret sbi_set_timer(uint64_t stime_value);
```

See the SBI documentation [RISCV-SBI] for a list of SBI calls and their encodings. SBI calls are primarily made by the OS, so programmers rarely use them.

3.3.5 Machine Architecture

RISC-V has many ISA flavors, so GCC needs to be told which one to target. RISC-V defines the ISA with a base instruction set and letters or

Recall that the G (general) extension is shorthand for supporting all of these extensions: I (base integer), M (multiply/divide), A (atomic), F (floating-point), and D (double) extensions.

Wally has various predefined configurations described in [Section 2.8.2](#) and [Tables A.1](#) and [A.3](#), such as `rv32gc`. These configurations define the ISA and also Wally-specific implementation options such as the peripherals and cache organization.

strings describing the extensions, such as `RV32GC` (see [Section 2.5](#)). GCC uses the `-march` (machine architecture) flag to indicate which instructions the compiler can produce, such as `-march=rv32gc`. Note the letters are lowercase for GCC. Whereas in `rv32imac`, the `mul` instruction is legal, in `rv32i` using `mul` produces an “Unrecognized opcode” compiler error because the processor hardware does not support the multiply/divide extension (M). Similarly, the C statements `int a, b, c; a = b * c;` compile into `mul s0, s1, s2` in `rv32imac`, but in `rv32i`, they compile into a library call to a multiplication routine using a sequence of shifts and adds.

[Table 3.2](#) lists a variety of common machine architectures and their associated ABIs. The blue entries also correspond to default Wally configurations from [Tables A.1](#) and [A.3](#), although Wally can support any machine architecture by setting the desired XLEN and supported extensions in the configuration file.

The machine architecture and associated ABI, which gives the size of data types ([Table 3.1](#)), are passed to GCC on the command line using flags such as `-march=rv32i -mabi=ilp32` or `-march=rv64gc -mabi=lp64d`. GCC also supports longer architecture names containing other extensions separated by underscores, such as `-march=rv64gc_zba_zbb_svpbmt`.

Table 3.2 Machine architectures.

Architecture	Synonym	ABI
<code>rv32e</code>		<code>ilp32e</code>
<code>rv32i</code>		<code>ilp32</code>
<code>rv32im</code>		<code>ilp32</code>
<code>rv32iac</code>		<code>ilp32</code>
<code>rv32imac</code>		<code>ilp32</code>
<code>rv32imafc</code>		<code>ilp32f</code>
<code>rv32imafdc</code>	<code>rv32gc</code>	<code>ilp32d</code>
<code>rv64i</code>		<code>lp64</code>
<code>rv64im</code>		<code>lp64</code>
<code>rv64ic</code>		<code>lp64</code>
<code>rv64iac</code>		<code>lp64</code>
<code>rv64imac</code>		<code>lp64</code>
<code>rv64imafdc</code>	<code>rv64gc</code>	<code>lp64d</code>

3.3.6 Driving GCC

Let's look a little more deeply into using GCC. `riscv64-unknown-elf-gcc` is both a C compiler and assembler. It is invoked as a single command, but under the hood it performs four steps:

- ▶ **Preprocess:** Insert `#include` files and substitute `#define` macros with their values
- ▶ **Compile:** Turn each C program into assembly language
- ▶ **Assemble:** Turn each assembly program into an ELF object file
- ▶ **Link:** Join all the object files together, relocate sections to desired addresses, resolve cross-references, and produce an ELF executable file

Recall that a basic GCC compiler invocation, described further in [Table 3.3](#), is as follows:

```
$ riscv64-unknown-elf-gcc -o example -march=rv32i -mabi=ilp32
-mcmodel=medany -nostartfiles -T../link/link.ld example.S
```

Table 3.3 GCC options.

GCC Argument	Description
<code>example.S</code>	The assembly language source file to compile, indicated by the <code>.S</code> suffix; suffix <code>.c</code> would indicate a C source file
<code>-o example</code>	The executable output file is named <code>example</code>
<code>-march=rv32i</code>	Specifies the machine architecture as <code>rv32i</code> ; other options include <code>rv32g</code> , <code>rv32imc</code> , and so on
<code>-mabi=ilp32</code>	Specifies the ABI compatible with the machine architecture, as given in Table 3.2
<code>-mcmodel=medany</code>	Specifies medium code mode with 32-bit relative addresses; <code>-mcmodel=medlow</code> specifies small code mode with 32-bit absolute addresses
<code>-nostartfiles</code>	Prevents the assembler from inserting startup code, so only the actual code in <code>example.S</code> is assembled
<code>-T</code>	Specifies the path to the linker script, with the script shown in Code Example 3.3

Code Example 3.3 shows the `link.ld` script invoked above. The linker script specifies that the program starts at a symbol called `rvtest_entry_point` and that the `.text` (program) segment begins at address `0x80000000`. Subsequent sections are aligned to multiples of `0x1000` (4 KiB), which is the size of a page if virtual memory is used (see Chapter 11).

Code Example 3.3 `link.ld`

```
OUTPUT_ARCH( "riscv" )
ENTRY(rvtest_entry_point)

SECTIONS
{
    . = 0x80000000;
    .text : { *(.text.init) }
    . = ALIGN(0x1000);
    .tohost : { *(.tohost) }
    . = ALIGN(0x1000);
    .data : { *(.data) }
    .data.string : { *(.data.string)}
    . = ALIGN(0x1000);
    .bss : { *(.bss) }
    _end = .;
}
```

If `-march` is not specified, the machine architecture is system dependent, likely `rv64gc`, which produces compressed instructions. If the linker script is not specified, GCC will default to starting the program at `0x10078`. The following example shows this output. Observe how the file format is `elf64`, which means the program uses the RV64 ISA, and a 16-bit compressed instruction is used for the jump.

```
$ riscv64-unknown-elf-gcc -o example -nostartfiles example.S
$ riscv64-unknown-elf-objdump -d example

example:      file format elf64-littleriscv

Disassembly of section .text:

0000000000010078 <rvtest_entry_point>:
 10078:      02a00513      li      a0,42

000000000001007c <self_loop>:
 1007c:      a001              j      1007c <self_loop>
```

Typing the compile command is tedious and error prone, and reconstructing all the compiler flags later would be difficult. We avoid these problems by creating a Makefile along with the source code, as given in [Code Example 3.4](#).

Code Example 3.4 MAKEFILE

```
example.objdump: example
    riscv64-unknown-elf-objdump -D example > example.objdump

example: example.S
    riscv64-unknown-elf-gcc -o example -march=rv32i -mabi=ilp32 -mmodel=medany \
        -nostartfiles -T../link/link.ld example.S

clean:
    rm -f example example.objdump
```

The Makefile contains a *recipe* defining how to make the outputs. The files listed after the colons are dependencies. `example.objdump` depends on the executable file `example`, which in turn depends on `example.S`. If `example.S` has changed or `example` doesn't exist, make uses `gcc` to compile `example.S` into `example`. Then if `example` has changed, make uses `objdump` to disassemble `example` into `example.objdump`. Appendix B.5.4 describes Makefile syntax. To regenerate `example` and `example.objdump`, type:

```
$ make
```

To delete the generated files (`example` and `example.objdump`), type:

```
$ make clean
```

Other GCC flags that appear in this chapter include:

- ▶ `-g` Generate debugging symbols to show the source code while debugging
- ▶ `-static` Link to libraries statically, avoid dynamic libraries

3.3.7 Assembler Directives

Assembly language files include *instructions*, *comments*, *labels*, and *directives*. For RISC-V GCC, single-line comments begin with `//` or `#`. Multiline comments begin with `/*` and end with `*/`. Labels end with a colon (`:`). Labels don't directly generate code but are used to determine the address for loads, stores, branches, and jumps.

In code, assembler directives begin with a period, but when discussing sections, periods are often omitted. For example, we may refer to the text section, `.text` section, or `.text` interchangeably.

`.bss` (also simply, `bss`) stands for *block starting symbol*. Historically, it meant a block of uninitialized data that was indicated (started) by a symbol (label). Most systems these days initialize the `.bss` section to 0 to avoid nondeterministic behavior caused by random values in uninitialized variables. Nevertheless, a prudent programmer always initializes variables before using them.

Assembler directives indicate section boundaries and specify variables and constants. Table 3.4 lists some of the common directives [RISCV-ASM]. Section assembler directives mark the beginning of that section. Text sections contain program code. Data sections contain variables. By convention, initialized global variables such as `int salary = 110000; go` in an ordinary data section while uninitialized global variables such as `double matrix[3][3]; go` in a `bss` data section. Some directives have

Table 3.4 RISC-V assembler directives.

Assembler Directive	Description
Sections	
<code>.text</code>	Text (program code)
<code>.data</code>	Data: initialized variables
<code>.rodata</code>	Read-only data: initialized constants
<code>.bss</code>	Data: uninitialized variables
<code>.symtab</code>	Symbol table, including addresses of labels
<code>.section name</code>	User-defined section
<code>.end</code>	Optional end of file
Data	
<code>.byte x1, x2, ...</code>	8-bit comma-separated values
<code>.half/.short/.2byte x1, x2, ...</code>	16-bit comma-separated values
<code>.word/.long/.4byte x1, x2, ...</code>	32-bit comma-separated values
<code>.dword/.8byte x1, x2, ...</code>	64-bit comma-separated values
<code>.string "str"</code>	Null terminated string
<code>.space size</code>	Leave room for <i>size</i> bytes
<code>.fill num, size, value</code>	Fill memory with <i>num</i> entries of <i>size</i> bytes with <i>value</i>
Other	
<code>.align size</code>	Align next code or data address to a power of 2 <i>size</i>
<code>.global/.globl</code>	Define a global symbol visible to the linker
<code>.equ const, value</code>	Define a constant <i>const</i> to be <i>value</i>
<code>.macroendm</code>	Define a macro

multiple names, shown separated by / in the table; for example, `.word` or `.4byte` both generate 32-bit numbers.

3.3.8 Spike Simulation

Spike was the original *golden reference model* of the RISC-V instruction set architecture: it served both as a simulator and as a reference against which other RISC-V hardware or software implementations can be compared. Sail has replaced Spike for this purpose, but Spike is more convenient to use. Both are instruction set simulators, so they only simulate the function of the instructions (not the actual hardware) and, thus, are not cycle accurate. They ignore delays such as cache misses, memory transactions, and so on. Spike shockingly lacks a user manual, but brief instructions can be found at [Spike].

To simulate in interactive debug mode, run Spike with `-d`.

```
$ spike --isa=rv32gc -d example
```

All values in Spike are in hexadecimal. To see the contents of integer register `t0` on hart 0, type:

```
:reg 0 t0
```

This also works for control and status registers (CSRs):

```
:reg 0 mstatus
```

To see the contents of a single- or double-precision floating-point register, replace `reg` with `fregs` or `fregd`. To read the contents of physical memory at address `0x80000004`, type:

```
:mem 80000004
```

Table 3.5 summarizes common Spike commands, and the listing below it illustrates more Spike capabilities (but some would hang on this example because it doesn't write to memory).

```
: until pc 0 80000004
: until reg 0 a0 2a
: until mem 0 80000020 1
: while mem 0 80000020 deadbeefdeadbeef
: run
: <Ctrl-c>
: q
```

Spike is named for the golden spike that completed the transcontinental railroad. The term *golden* is contentious because it implies perfection, and even the best reference model never perfectly matches a nontrivial written specification.

Spike supports other commands such as system calls through the HTIF, but they are poorly documented. Spike uses another program called proxy kernel (`pk`) to handle system calls using the host OS. Spike also interfaces with `gdb` through `OpenOCD` for debugging, but this has been unreliable in our environment. If your program is complicated enough to need these features, you may be better off running with `QEMU`, described in [Section 3.6](#).

The address of `tohost` doesn't matter; Spike detects a write to an address with a label of this name in the symbol table.

Table 3.5 Spike commands.

Command	Description
Enter	Press enter key to step through instructions one at a time
reg 0 t0	Print the contents of register <code>t0</code> on hart 0
reg 0 mstatus	Print the contents of the <code>mstatus</code> CSR on hart 0
mem 80000004	Print the contents of memory at physical address <code>0x80000004</code>
mem 0 80000008	Print the contents of memory at Core 0's virtual address <code>0x80000008</code>
until <condition>	Run until a condition is satisfied
while <condition>	Run while a condition is satisfied
run	Run indefinitely, printing PC and instruction as it goes
rs	Run silently, without printing
Ctrl-c	Pause the simulation and reenters interactive debug mode
q	End the simulation
help	List available commands

A RISC-V program used for simulation or testing typically ends with an infinite loop, so it doesn't wander into undefined instructions. Therefore, a Spike simulation will also end by looping indefinitely. Spike provides a basic *host target interface* (HTIF) which, among other features, allows a program to signal completion. To do so, a program must declare an 8-byte `.dword` at a label called `tohost`. In Spike, a write of 1 to this address causes the program to terminate successfully. A write of 3 causes it to print `*** FAILED ***` and terminate. A write of any even number to the HTIF triggers a system call, with the number interpreted as a pointer to an array in memory containing the system call number and arguments. For example, system call number 64 is `sys_write`, which takes three arguments: a file descriptor, a pointer to a string, and the length of the string (see github.com/riscv-software-src/riscv-isa-sim/blob/master/fesvr/syscall.cc). File descriptors of 0, 1, and 2 correspond to `stdin`, `stdout`, and `stderr`, respectively.

Spike also has a *signature* mechanism to write the contents of a block of memory to a file upon completion. Typically, the signature block should have a series of words that are only correct if the program ran properly. The signature block in a `.data` section is delimited by `begin_signature:` and `end_signature:` labels. Run Spike with the command below to write the values in the signature block to a file named `example.signature.output`, with 4 bytes per line in hexadecimal. The user can compare this file with an expected output to determine whether the program ran correctly.

```
$ spike +signature=example.signature.output +signature-granularity=4 example
```

In this case, the signature file would be omitted because `example` doesn't have a signature block. [Code Example 3.5](#) in [Section 3.3.10](#) is a more interesting program that writes a signature.

3.3.9 Performance Measurement

RISC-V has several CSRs used for performance measurement. `instret` indicates how many instructions have been retired (i.e., completed). `cycle` indicates how many clock cycles have passed. The `csrr reg, CSR` pseudoinstruction reads a CSR into `reg`, an integer register. The example in the next section illustrates using `instret` to count the number of instructions executed by a function call.

In Spike, `cycle` and `instret` both advance by one each cycle and have the same value because one instruction is executed in each cycle. In a real processor with a CPI other than 1.0, the two will differ.

3.3.10 Putting It All Together

The program `$WALLY/examples/asm/sumtest` illustrates an RV64 program linking multiple files with multiple sections, using assembler directives, writing to the host interface, and generating a signature with Spike. The example involves the two source files given in [Code Examples 3.5](#) and [3.6](#).

`sumtest.S` initializes the stack pointer `sp`. Note that `la` is a pseudoinstruction that loads an address into a register with an `auipc` and `addi` sequence to compute the address relative to the PC. Also note the double-word (8-byte) `ld` and `sd` loads and stores in the RV64 program. The program reads the value of `N` (4) from the data segment and then calls the function `sum(N)` defined in the `sum.S` file, which adds up the numbers from 1 to `N`. `sumtest` also counts the number of instructions retired before and after the call and takes their difference to measure the call performance. It writes the return result and instruction count to the signature memory and then writes 1 to `tohost` to indicate completion before entering an infinite loop at `self_loop`.

RISC-V has a configurable and implementation-specific set of performance counters to measure microarchitectural events, such as cache misses or branch mispredictions. These are discussed further in [Section 8.4.3.1](#).

Code Example 3.5 `sumtest.S`

```
// sumtest.S
// David_Harris@hmc.edu 24 December 2021

.global rvtest_entry_point
rvtest_entry_point:
    la sp, topofstack      # Initialize stack pointer
    la t0, N               # Get address of N in data
    ld a0, 0(t0)           # Load N
    csrr s8, instret        # Count # instructions completed before call made
    jal sum                # Call sum(N)
    csrr s9, instret        # Count # instructions completed after call made
    sub s9, s9, s8          # Calculate # instructions completed during call
    la t0, begin_signature # Address of signature
    sd a0, 0(t0)           # Store sum(N) in signature
    sd s9, 8(t0)           # Record performance (# instructions executed during sum)

write_tohost:
    la t1, tohost          # Load address of tohost label into t1
    li t0, 1               # 1 for success, 3 for failure
    sd t0, 0(t1)           # Send success code

self_loop:
    j self_loop            # Wait

.section .tohost
tohost:                   # Write to HTIF
    .dword 0
fromhost:
    .dword 0

.data
N:
    .dword 4              # N = 4

.EQU XLEN,64
begin_signature:
    .fill 2*(XLEN/32),4,0xdeadbeef # Initialize signature memory with 0xdeadbeef
end_signature:

# Initialize stack with room for 512 bytes
.bss
    .space 512
topofstack:
```

Each 32-bit value in the signature is initialized to the hexadecimal number 0xdeadbeef, which should be different than any sensible signature later written by the program.

`sum(N)` adds the numbers from 1 to `N` and returns the result. The `sum` label is declared global so that the other file `sumtest.S` can reference it. The `sum` function uses `s0` and `s1`, so it begins by saving them on the stack and then restores them at the end. According to the RISC-V ABI, `sum` looks for `N` in `a0` during the function and returns the result in `a0` at the end of the function.

Code Example 3.6 `sum.S`

```
// sum.S
// David_Harris@hmc.edu 24 December 2021
// Add numbers from 1 to N.

// Result in s0, i in s1, N in a0, return answer in a0
//
// long sum(long N) {
//     long result, i;
//     result = 0;
//     for (i=1; i<=N; i++) result = result + i;
//     return result;
// }
.global sum
sum:
    addi sp, sp, -16    # Make room to save s0 and s1 on the stack
    sd s0, 0(sp)
    sd s1, 8(sp)

    li s0, 0            # result = 0
    li s1, 1            # i = 1
for: bgt s1, a0, done    # Exit loop if i > N
    add s0, s0, s1      # result = result + i
    addi s1, s1, 1      # i++
    j for               # Repeat

done:
    mv a0, s0           # Put result in a0 to return
    ld s0, 0(sp)        # Restore s0 and s1 from stack
    ld s1, 8(sp)
    addi sp, sp, 16
    ret                 # Return from function
```

[Code Example 3.7](#) lists the expected results written to the signature memory. Counting instructions, we see the call to `sum` takes $13+4N$ instructions, including the `jal` and one of the `csrr` instructions. Remember that `bgt` executes one last time when the for loop exits. So, for $N = 4$, the expected return value is $\text{sum}(4) = 1+2+3+4 = 10 = 0xA$, and the number of instructions executed is $29 = 0x1D$.

Code Example 3.7 `sumtest.reference_output`

```
0000000000000000A
0000000000000001D
```

The Makefile in [Code Example 3.8](#) compiles `sumtest` for `rv64gc`. `make sim` runs Spike to generate `sumtest.signature.output`. It then compares against `sumtest.reference_output`. If the files are not identical, it quits with an exit code; otherwise, it prints `Signature matches! Success! Build and test using the following commands:`

Notice how the Makefile in [Code Example 3.8](#) declares a variable `TARGET` with the name of the program so we could rename the program without retyping `sumtest` many times. In contrast, [Code Example 3.4](#) had the file name example in many places, each of which would need to be changed.

```

$ make clean # Optionally delete old files
rm -f sumtest sumtest.objdump sumtest.signature.output

$ make # Build and disassemble sumtest; similar to example except -march & -mabi
riscv64-unknown-elf-gcc -g -o sumtest -march=rv64gc -mabi=lp64d -mcmodel=medany \
  -nostartfiles -T../link/link.ld sumtest.S sum.S
riscv64-unknown-elf-objdump -D sumtest > sumtest.objdump

$ make sim # Simulate sumtest in spike, write signature.output, compare with reference_output
spike +signature=sumtest.signature.output +signature-granularity=8 sumtest
diff --ignore-case sumtest.signature.output sumtest.reference_output || exit
echo "Signature matches! Success!"
Signature matches! Success!

```

Code Example 3.8 MAKEFILE

```

TARGET = sumtest

$(TARGET).objdump: $(TARGET)
    riscv64-unknown-elf-objdump -D $(TARGET) > $(TARGET).objdump

$(TARGET): $(TARGET).S sum.S
    riscv64-unknown-elf-gcc -g -o $(TARGET) -march=rv64gc -mabi=lp64d -mcmodel=medany \
        -nostartfiles -T../link/link.ld $(TARGET).S sum.S

sim:
    spike +signature=$(TARGET).signature.output +signature-granularity=8 $(TARGET)
    diff --ignore-case $(TARGET).signature.output $(TARGET).reference_output || exit
    echo "Signature matches! Success!"

clean:
    rm -f $(TARGET) $(TARGET).objdump $(TARGET).signature.output

```

make `sim` uses a subtle Unix idiom. The OR operator (`||`) is lazy; it only evaluates as much as necessary. So if the first input is TRUE, it returns TRUE without evaluating the second input. In this case, `diff` returns TRUE if the two files are identical, and the lazy OR ignores the `exit` command. Hence, if the files are identical, the Makefile prints `Success!` If the files are different, the `||` operator invokes `exit` to stop make without printing success.

Code Example 3.9 shows the disassembled code of the `sumtest` program. Observe that the format is `elf64` and that the code includes 16-bit compressed instructions, such as the `li` at `0x80000003c`. Observe how the `la` (load address) pseudoinstruction assembles into `auipc` and `addi`. The stack pointer is initialized to `0x80003200`, which is at the top (highest address) of the `.bss` section. Observe how the linker combines `sumtest` and `sum` into a single continuous text section. The data section places `N` at `0x80002000` and a 16-byte signature at `0x80002008`.

Code Example 3.9 `sumtest.objdump`

```
sumtest:      file format elf64-littleriscv
```

```
Disassembly of section .text:
```

```
0000000080000000 <rvtest_entry_point>:
```

```

80000000:      00003117      auipc      sp,0x3
80000004:      20010113      addi       sp,sp,512      # 80003200 <_end>
80000008:      00002297      auipc      t0,0x2
8000000c:      ff828293      addi       t0,t0,-8      # 80002000 <N>
80000010:      0002b503      ld         a0,0(t0)
80000014:      c0202c73      rdinstret  s8
80000018:      02c000ef      jal        ra,80000044 <sum>
8000001c:      c0202cf3      rdinstret  s9
80000020:      418c8cb3      sub        s9,s9,s8
80000024:      00002297      auipc      t0,0x2
80000028:      fe428293      addi       t0,t0,-28     # 80002008 <begin_signature>
8000002c:      00a2b023      sd         a0,0(t0)
80000030:      0192b423      sd         s9,8(t0)

```

```
0000000080000034 <write_tohost>:
```

```

80000034:      00001317      auipc      t1,0x1
80000038:      fcc30313      addi       t1,t1,-52     # 80001000 <tohost>
8000003c:      4285         li         t0,1
8000003e:      00533023      sd         t0,0(t1)

```

```
0000000080000042 <self_loop>:
```

```
80000042:      a001         j          80000042 <self_loop>
```

```
0000000080000044 <sum>:
```

```

80000044:      1141         addi       sp,sp,-16
80000046:      e022         sd         s0,0(sp)
80000048:      e426         sd         s1,8(sp)
8000004a:      4401         li         s0,0
8000004c:      4485         li         s1,1

```

```
000000008000004e <for>:
```

```

8000004e:      00954563      blt        a0,s1,80000058 <done>
80000052:      9426         add        s0,s0,s1
80000054:      0485         addi       s1,s1,1
80000056:      bfe5         j          8000004e <for>

```

```
0000000080000058 <done>:
```

```

80000058:      8522         mv         a0,s0
8000005a:      6402         ld         s0,0(sp)
8000005c:      64a2         ld         s1,8(sp)
8000005e:      8082         ret

```

```
Disassembly of section .tohost:
```

```
0000000080001000 <tohost>:
```

```
...
```

```
0000000080001008 <fromhost>:
```

```
...
```

Disassembly of section .data:

```
0000000080002000 <N>:
    80002000:    0004                0x4
    80002002:    0000                unimp
    80002004:    0000                unimp
    ...

0000000080002008 <begin_signature>:
    80002008:    deadbeef            jal    t4,7ffdd5f2 <XLEN+0x7ffdd5b2>
    8000200c:    deadbeef            jal    t4,7ffdd5f6 <XLEN+0x7ffdd5b6>
    80002010:    deadbeef            jal    t4,7ffdd5fa <XLEN+0x7ffdd5ba>
    80002014:    deadbeef            jal    t4,7ffdd5fe <XLEN+0x7ffdd5be>
```

Disassembly of section .bss:

```
0000000080003000 <_end-0x200>:
```

Observe how `objdump` disassembles the data section into gibberish. Using `-d` instead of `-D` only shows disassembled sections expected to contain instructions and does not show the values in the data section.

[Code Example 3.10](#) lists the ELF metadata of the `sumtest` file. The file has `.text`, `.tohost`, `.data`, and `.bss` sections specified in the assembly language files, as well as RISC-V attributes, debug symbols, and the symbol table generated by the linker. The symbol table contains the addresses of all the labels from the text and data segments. The compiler targeted version 2.0 of the RV64GC instruction set and 1.11 of the privileged specification, as indicated by the attributes section at the end of the file.

Code Example 3.10 `sumtest` ELF METADATA

```
$ riscv64-unknown-elf-readelf -a sumtest

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class:                   ELF64
  Data:                     2's complement, little endian
  Version:                  1 (current)
  OS/ABI:                   UNIX - System V
  ABI Version:              0
  Type:                     EXEC (Executable file)
  Machine:                  RISC-V
  Version:                  0x1
  Entry point address:      0x80000000
  Start of program headers: 64 (bytes into file)
  Start of section headers: 9904 (bytes into file)
  Flags:                    0x5, RVC, double-float ABI
  Size of this header:      64 (bytes)
  Size of program headers:  56 (bytes)
  Number of program headers: 2
  Size of section headers:  64 (bytes)
  Number of section headers: 15
  Section header string table index: 14
```


Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info	Align
[0]	0000000000000000	NULL	0000000000000000	00000000
[1]	.text	PROGBITS	0000000080000000	00001000
	00000000000000060	0000000000000000	AX 0 0	2
[2]	.tohost	PROGBITS	0000000080001000	00002018
	00000000000000010	0000000000000000	0 0 0	1
[3]	.data	PROGBITS	0000000080002000	00002000
	00000000000000018	0000000000000000	WA 0 0	1
[4]	.bss	NOBITS	0000000080003000	00002018
	00000000000000200	0000000000000000	WA 0 0	1
[5]	.riscv.attributes	RISCV_ATTRIBUTE	0000000000000000	00002028
	00000000000000037	0000000000000000	0 0 0	1
[6]	.debug_line	PROGBITS	0000000000000000	0000205f
	00000000000000128	0000000000000000	0 0 0	1
[7]	.debug_line_str	PROGBITS	0000000000000000	00002187
	0000000000000003e	0000000000000001	MS 0 0	1
[8]	.debug_info	PROGBITS	0000000000000000	000021c5
	00000000000000050	0000000000000000	0 0 0	1
[9]	.debug_abbrev	PROGBITS	0000000000000000	00002215
	00000000000000028	0000000000000000	0 0 0	1
[10]	.debug_aranges	PROGBITS	0000000000000000	00002240
	00000000000000060	0000000000000000	0 0 0	16
[11]	.debug_str	PROGBITS	0000000000000000	000022a0
	0000000000000004a	0000000000000001	MS 0 0	1
[12]	.symtab	SYMTAB	0000000000000000	000022f0
	000000000000002a0	0000000000000018	13 25 8	
[13]	.strtab	STRTAB	0000000000000000	00002590
	0000000000000008a	0000000000000000	0 0 0	1
[14]	.shstrtab	STRTAB	0000000000000000	0000261a
	00000000000000096	0000000000000000	0 0 0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
 L (link order), O (extra OS processing required), G (group), T (TLS),
 C (compressed), x (unknown), o (OS specific), E (exclude),
 D (mbind), p (processor specific)

There are no section groups in this file.

Program Headers:

Type	Offset	VirtAddr	PhysAddr
	FileSiz	MemSiz	Flags Align
LOAD	0x00000000000001000	0x0000000080000000	
	0x0000000000000060	R E 0x1000	
LOAD	0x00000000000002000	0x0000000080002000	
	0x0000000000000018	0x0000000000001200	RW 0x1000

Section to Segment mapping:

```
Segment Sections...
00      .text
01      .data .bss
```

There is no dynamic section in this file.

There are no relocations in this file.

The decoding of unwind sections for machine type RISC-V is not currently supported.

Symbol table '.symtab' contains 28 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000080000000	0	SECTION	LOCAL	DEFAULT	1	.text
2:	0000000080001000	0	SECTION	LOCAL	DEFAULT	2	.tohost
3:	0000000080002000	0	SECTION	LOCAL	DEFAULT	3	.data
4:	0000000080003000	0	SECTION	LOCAL	DEFAULT	4	.bss
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	.riscv.attributes
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	.debug_line
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	.debug_line_str
8:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	.debug_info
9:	0000000000000000	0	SECTION	LOCAL	DEFAULT	9	.debug_abbrev
10:	0000000000000000	0	SECTION	LOCAL	DEFAULT	10	.debug_aranges
11:	0000000000000000	0	SECTION	LOCAL	DEFAULT	11	.debug_str
12:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	cc0f1QM8.o
13:	0000000080003200	0	NOTYPE	LOCAL	DEFAULT	4	topofstack
14:	0000000080002000	0	NOTYPE	LOCAL	DEFAULT	3	N
15:	0000000080002008	0	NOTYPE	LOCAL	DEFAULT	3	begin_signature
16:	0000000080000034	0	NOTYPE	LOCAL	DEFAULT	1	write_tohost
17:	0000000080001000	0	NOTYPE	LOCAL	DEFAULT	2	tohost
18:	0000000080000042	0	NOTYPE	LOCAL	DEFAULT	1	self_loop
19:	0000000080001008	0	NOTYPE	LOCAL	DEFAULT	2	fromhost
20:	0000000000000040	0	NOTYPE	LOCAL	DEFAULT	ABS	XLEN
21:	0000000080002018	0	NOTYPE	LOCAL	DEFAULT	3	end_signature
22:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	ccZtnxq7.o
23:	000000008000004e	0	NOTYPE	LOCAL	DEFAULT	1	for
24:	0000000080000058	0	NOTYPE	LOCAL	DEFAULT	1	done
25:	0000000080000044	0	NOTYPE	GLOBAL	DEFAULT	1	sum
26:	0000000080000000	0	NOTYPE	GLOBAL	DEFAULT	1	rvtest_entry_point
27:	0000000080003200	0	NOTYPE	GLOBAL	DEFAULT	4	_end

No version information found in this file.

Attribute Section: riscv

File Attributes

Tag_RISCV_arch: "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0"

Tag_RISCV_priv_spec: 1

Tag_RISCV_priv_spec_minor: 11

3.4 GCC GNU C COMPILER

The GCC tools are known for their excellent C compiler. This section shows how to *cross-compile* C programs for RISC-V. Cross-compile means to generate RISC-V assembly on a processor with a different architecture, such as x86. We disassemble the program and examine its memory map and the effects of optimization flags. Then we look at how to mix C

and assembly by using inline assembly statements and by linking functions written in the different languages.

3.4.1 A First C Program

Code Example 3.11 shows a simple program, `sum.c`, that illustrates a C program to add the numbers from 1 to `N`. It also illustrates printing to the console, via the HTIF, and verifying results. The program is built using GCC and run with Spike. The program adds the numbers from 1 to 4 and successfully prints the sum of 10. It prints the number of cycles and instructions retired during the `sum` function call by sandwiching that call between the `setStats(1)` and `setStats(0)` calls. These calls have some latency, so `mcycle` and `minstret` are not sampled at exactly the same time and are conservative. Run the `sum.c` program on Spike as follows:

```
$ cd $WALLY/examples/C/sum
$ make
riscv64-unknown-elf-gcc -o sum -g -O \
  -march=rv64gc -mabi=lp64d -mcmodel=medany \
  -nostartfiles -T../common/test.ld -I../common \
  sum.c ../common/crt.S ../common/syscalls.c
riscv64-unknown-elf-objdump -S sum > sum.objdump
$ spike sum
s = 10
mcycle = 31
minstret = 38
```

Code Example 3.11 `sum.c`

```
// sum.c
// David_Harris@hmc.edu 24 December 2021
// Simple illustration of compiling C code

#include <stdio.h> // Supports printf
#include "util.h"  // Supports verify

long sum(long N) { // Adds up numbers from 1 to N
    long result, i;
    result = 0;
    for (i=1; i<=N; i++) {
        result = result + i;
    }
    return result;
}

int main(void) {
    int s[1], expected[1];
    setStats(1); // Record initial mcycle and minstret
    s[0] = sum(4); // Call assembly language sum function to compute 1 + 2 + 3 + 4 = 10
    setStats(0); // Record elapsed mcycle and minstret
    printf("s = %d\n", s[0]); // Print sum of numbers, which should be 10
    expected[0] = 10; // Set expected value
    return verify(1, s, expected); // Check the 1 element of s matches expected. 0 means success
}
```

GCC normally produces only the executable output file. Adding the `--save-temps` flag to the compile statement saves the preprocessor, compiler, and assembler intermediate outputs as `.i`, `.s`, and `.o` files, respectively, which may give you more insight into the whole process.

The optimization flag is the capital letter “O”: `-O` (not the number zero: `-0`).

Code Example 3.12 shows the Makefile with the compiler flags. The backslash `\` is a continuation character allowing a command to exceed one line. `-g` generates debug symbols, and `-O` performs basic optimization. The compiler needs the `-march`, `-mabi`, and `-mmodel` flags to generate proper code for `rv64gc` with the `lp64d` ABI. The next line of flags allows the program to run on bare metal without an OS. Specifically, `-nostdlib` disables linking both the C standard library `libc.a` and the startup code; it is a superset of `-nostartfiles`. `-static` prevents dynamic linking with shared libraries, so we can see the entire program in the ELF object dump. `-lm` links the `libm.a` math library if necessary. `-fno-tree-loop-distribute-patterns` disables an optimization that would replace loops with `memcpy` or `memset` from the disabled standard library. The `-T` and `-I` flags specify the linker script and include the file search path. Because we requested `-nostdlib`, we also must compile and link the `crt.S` minimal startup file and `syscalls.c` library (borrowed from the `riscv-software-src/riscv-tests` repository) to do `printf` and `exit` with the HTIF.

Optimization flags include:

- O Perform simple optimizations to reduce code size and execution time.
- O2 Perform all optimizations that don’t bloat the code (compiling takes longer).
- O3 Perform all optimizations, including those that increase code size, to decrease execution time.

Code Example 3.12 MAKEFILE FOR C PROGRAMS

```
TARGET = sum

$(TARGET).objdump: $(TARGET)
    riscv64-unknown-elf-objdump -S $(TARGET) > $(TARGET).objdump

$(TARGET): $(TARGET).c Makefile
    riscv64-unknown-elf-gcc -o $(TARGET) -g -O\
        -march=rv64gc -mabi=lp64d -mmodel=medany \
        -nostdlib -static -lm -fno-tree-loop-distribute-patterns \
        -T./common/test.ld -I./common \
        $(TARGET).c ../common/crt.S ../common/syscalls.c
clean:
    rm -f $(TARGET) $(TARGET).objdump
```

The disassembled program is too long to show here in full because of the startup and library code, but you can view it in `sum.objdump`. The code begins at `0x80000000` in the `.text.init` section at the `_start` label defined in `crt.S`. This startup code initializes all registers to 0 and turns on the floating-point unit (FPU). It sets the trap vector `mtvec` to be the address

of the `trap_entry` function. If a trap (interrupt or exception) occurs, the `trap_entry` function causes the program to exit with an error code. The startup code initializes the global pointer, thread pointer, and stack pointer registers, `gp`, `tp`, and `sp`, to RAM above the program section. It then calls `_init` in `syscalls.c`, which calls `main()` from hart 0. Fig. 3.3 shows a memory map of the program.

`main()` calls `sum(4)`. Code Example 3.13 shows the disassembled code for `sum` with optimizations turned off (no `-O` flag) and Fig. 3.4 shows the stack usage. We added the comments to help explain the code. The program allocates 48 bytes (6 double words) on the stack. It saves `s0` on the stack and then uses `s0` as a frame pointer that points to the bottom of `sum`'s stack frame (i.e., the original top of stack before the function

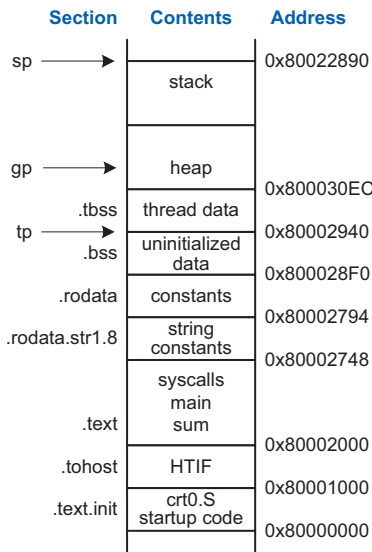


Fig. 3.3 Program memory map.

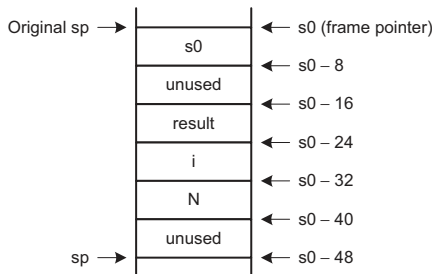


Fig. 3.4 `sum` stack usage (optimization turned off).

was called). It keeps local variables on the stack instead of in registers and accesses them relative to the frame pointer `s0`. The code is grossly inefficient.

Code Example 3.13 `sum` DISASSEMBLY (OPTIMIZATION TURNED OFF)

```

long sum(long N) {
    8000231a: 7179          addi    sp,sp,-48    # Room for 6 double-words on stack
    8000231c: f422          sd      s0,40(sp)    # Save s0 on stack
    8000231e: 1800          addi    s0,sp,48     # s0 = frame ptr (old top of stack)
    80002320: fca43c23     sd      a0,-40(s0)    # Save N on stack
    long result, i;
    result = 0;
    80002324: fe043423     sd      zero,-24(s0) # result = 0
    for (i=1; i<=N; i++) {
    80002328: 4785          li      a5,1
    8000232a: fef43023     sd      a5,-32(s0)    # i = 1
    8000232e: a829          j      80002348 <sum+0x2e> # Check if done
        result = result + i;
    80002330: fe843703     ld      a4,-24(s0)    # a4 = result
    80002334: fe043783     ld      a5,-32(s0)    # a5 = i
    80002338: 97ba          add     a5,a5,a4     # result = result + i
    8000233a: fef43423     sd      a5,-24(s0)    # Put result back in memory
    for (i=1; i<=N; i++) {
    8000233e: fe043783     ld      a5,-32(s0)    # a5 = i
    80002342: 0785          addi    a5,a5,1       # i = i + 1
    80002344: fef43023     sd      a5,-32(s0)    # Put i back in memory
    80002348: fe043703     ld      a4,-32(s0)    # a4 = i
    8000234c: fd843783     ld      a5,-40(s0)    # a5 = N
    80002350: fee7d0e3     bge     a5,a4,80002330 <sum+0x16> # Loop if N >= i
    }
    return result;
    80002354: fe843783     ld      a5,-24(s0)    # a5 = result
}
    80002358: 853e          mv      a0,a5         # Return value = result
    8000235a: 7422          ld      s0,40(sp)    # Restore s0 from stack
    8000235c: 6145          addi    sp,sp,48     # Clean up stack pointer
    8000235e: 8082          ret

```

Code Example 3.14 shows the disassembled code for `sum` with basic optimizations (-O) turned on. Now the program doesn't use the stack at all. It receives `N` in `a0` and stores `N+1` in `a4` as a completion condition. It keeps `i` in `a5` and uses `a0` for `result`, which saves copying a different register to `a0` to return `result` at the end. It loops until `i` equals `N+1` and then returns. The code is much faster. Observe that the disassembler has some difficulty associating the assembly language with the original C code because the optimizer reordered some of the code for performance.

Code Example 3.14 `sum` DISASSEMBLY (OPTIMIZATION TURNED ON WITH `-o`)

```

0000000080002000 <sum>:
#include "util.h" // Supports verify

long sum(long N) {
    long result, i;
    result = 0;
    for (i=1; i<=N; i++) {
        80002000: 00a05b63          blez    a0,80002016 <sum+0x16>    # Return 0 if N <= 0
        80002004: 00150713          addi    a4,a0,1                # a4 = N+1
        80002008: 4785              li      a5,1                  # a5 = i = 1
        result = 0;
        8000200a: 4501              li      a0,0                  # result = 0
        result = result + i;
        8000200c: 953e              add     a0,a0,a5              # result= result + i
    for (i=1; i<=N; i++) {
        8000200e: 0785              addi    a5,a5,1              # i++
        80002010: fee79ee3         bne     a5,a4,8000200c <sum+0xc> # Loop if i != (N+1)
        80002014: 8082              ret                                # Return result
        result = 0;
        80002016: 4501              li      a0,0                  # result = 0
    }
    return result;
}
    80002018: 8082              ret                                # Return result

```

When `-O2` or `-O3` optimization is enabled, the compiler precomputes the result of calling `sum(4)` as 10 and replaces the `sum` function entirely with assigning `a1` to 10 as shown in [Code Example 3.15](#). The call to `verify` is also eliminated because the compiler precomputes that `verify` will return 0 (indicating success).

Code Example 3.15 `sum(4)` DISASSEMBLY (OPTIMIZATION TURNED ON WITH `-O2`)

```

8000271a:    45a9              li      a1,10

```

`main()` uses the `verify` function to compare an array of integer results to expected values; in this case, there is only one array element, which is $1+2+3+4 = 10$. It then returns with a code of 0 for success or 1 for failure. After `main` completes, `init` calls `exit` with the code for success or failure. Spike terminates silently if the code is 0 or prints `*** FAILED *** (tohost = 1)` on failure.

Table 3.6 Cycles to execute `sum(4)`.

Optimization Level	<code>mcycle</code>	<code>minstret</code>	True Cycles = True # Instructions
None	115	132	62
-O	31	38	21
-O2 or -O3	11	16	1

The `asm` statement needs to determine which register contains a variable in the C program. `%0` substitutes the argument, which is `"=r"(cycles)`, meaning the register containing the cycles variable. Therefore, the `csrr` instruction specifies the destination register as the register holding the variable `cycles`, which the compiler selected to be `a1`. If the program used more registers, `cycles` might be allocated to a different register, and the `asm` statement would pick the correct register. The `volatile` keyword prevents the compiler from optimizing away the assembly language instruction.

Table 3.6 shows the number of cycles and instructions retired to run `sum`, as reported by the program at various optimization levels. Spike simulates one instruction per cycle, so the two columns ideally would be the same, but the program has some overhead to call `setStats` and read each performance counter, and the overhead differs slightly between counters because of the order in which the counters are read. The table also shows the true number of instructions used by `sum`, measured by manually counting during the Spike simulation. These performance counters are more useful for longer programs where the overhead is negligible.

3.4.2 Inline Assembly Language

Code Example 3.16 illustrates *inline assembly language* used to add a single assembly language instruction that can't be implied by standard C code. It demonstrates the `asm` statement, which reads the `mcycle` CSR (0xB00). The result goes in the register holding the variable `cycles`. Code Example 3.17 shows the disassembly. The `asm` statement produces `csrr a1, mcycle`, where `a1` is the register holding the variable `cycles`.

Code Example 3.16 `inline.c`: SHOWS INLINE ASSEMBLY LANGUAGE

```
// inline.c
// David_Harris@hmc.edu 12 January 2022
// Illustrates inline assembly language

#include <stdio.h>

int main(void) {
    long cycles;
    asm volatile("csrr %0, 0xB00" : "=r"(cycles)); // read mcycle register
    printf ("mcycle = %ld\n", cycles);
}
```

Code Example 3.17 DISASSEMBLY OF `inline.c`

```

int main(void) {
    80002000: 1141                addi    sp,sp,-16
    80002002: e406                sd      ra,8(sp)
    long cycles;
    asm volatile("csrr %0, 0xB00" : "=r"(cycles)); // Read mcycle register
    80002004: b00025f3            csrr    a1,mcycle
    printf("mcycle = %ld\n", cycles);
    80002008: 00000517            auipc   a0,0x0
    8000200c: 70850513            addi    a0,a0,1800 # 80002710 <atol+0x6a>
    80002010: 540000ef            jal    ra,80002550 <printf>
}
    80002014: 4501                li      a0,0
    80002016: 60a2                ld      ra,8(sp)
    80002018: 0141                addi    sp,sp,16
    8000201a: 8082                ret

```

3.4.3 Mixed C and Assembly Language

When a C program needs to integrate many lines of assembly language, the easiest approach is to put the C and assembly language in different files and have them communicate through function calls via the ABI's calling convention. [Code Example 3.18](#) shows `sum_mixed.c`, which calls the `sum` function written in assembly language from [Code Example 2.5](#). The code is located in `$WALLY/examples/C/sum_mixed`. The C file declares `sum` as `extern`, so the linker knows to look for it elsewhere. The `sum.S` file declared the label `sum` as `global`, so the linker finds it. According to the ABI, the C program passes `N` to `sum` in `a0` and gets the result back in `a0`. Recall that `sum.S` saved and restored `s0` and `s1` on the stack so that they are preserved across the function call. The optimizer does not modify the assembly language code.

Code Example 3.18 `sum_mixed.c`

```

// sum_mixed.c
// David_Harris@hmc.edu 12 January 2022
// Call assembly language from C

#include <stdio.h>           // Defines printf
#include "util.h"           // Defines setStats and verify
extern int sum(int);        // Prototype for the sum function written an external assembly file

int main(void) {
    int s[1], expected[1];

    setStats(1);            // Record initial mcycle and minstret
    s[0] = sum(4);          // Call assembly language sum function to compute 1 + 2 + 3 + 4 = 10
    setStats(0);            // Record elapsed mcycle and minstret
    printf("s = %d\n", s[0]); // Print sum of numbers, which should be 10
    expected[0] = 10;       // Set expected value
    return verify(1, s, expected); // Check the 1 element of s matches expected. 0 means success
}

```

3.4.4 Standard Library and Startup Code

C programmers use a variety of standard libraries, such as `stdio.h`, `string.h`, `time.h`, and `math.h` that include basic functions such as `printf()`, `strlen()`, `time()`, `sqrt()`, or `cos()`. GCC needs to know how to implement these functions. The implementation is platform specific; for example, printing is different to a Linux console versus a serial COM port, reading the time depends on the system timer, and `math` is different for processors with or without floating-point instructions.

Whereas the `libc` standard library defines most of these functions, `libm` defines math functions. Together, these are called the *C standard library*. Historically, `libc` and `libm` were separated because the math libraries could be large and not all programs needed them. `libc` is automatically linked unless `-nostdlib` is specified when compiling. `libm` is only linked if the `-lm` flag is provided. The libraries can be *statically linked*, such that a copy is incorporated into the program, or *dynamically linked*, such that only one copy of the standard library is in memory and many different programs can call this library. Dynamic linking saves memory but requires some OS support. Hence, platforms such as Linux and Windows tend to use dynamic linking, while simple embedded processors use static linking. `libc.so` and `libm.so` are the *shared object* dynamically linked libraries, while `libc.a` and `libm.a` are the statically linked *archive* libraries.

Many implementations of the C standard library exist. Some are proprietary and vendor specific, such as the Microsoft C runtime library for Windows on x86 processors. The leading open-source implementations are `glibc` and `newlib`. `glibc` is often used for Linux systems that already have it available as a shared library. `newlib` has more permissive licensing terms and thus is often preferred for embedded software. When installing *riscv-gnu-toolchain*, the `$WALLY/bin/wally-tool-chain-install.sh` script compiled the RISC-V GCC cross-compiler with `newlib`.

The `newlib` standard library is precompiled so that GCC doesn't have to recompile all of the library functions every time it compiles a program. Instead, GCC simply links the program to the precompiled libraries in `libc.a` and `libm.a`. The best implementation of a library depends on the machine architecture. For example, `sqrt()` could be implemented with `fsqrt.d` in `rv64gc` but would require a lengthy series of integer operations in `rv32i`. GCC solves this with the *multilib* feature, which precompiles many flavors of `newlib` targeting different machine architectures. For example, the installation script compiled all the flavors listed in [Table 3.2](#). GCC automatically links the best library flavor consistent with the `-march` of the program being compiled.

In [Section 3.4](#), the C Makefiles used the `-nostdlib` argument for GCC to avoid the startup code standard libraries, and instead have included `crt.S` and `syscalls.c` from `$WALLY/examples/C/common`. It is informative

to read through each of these files. In brief, `crt.S` is an assembly language startup routine that initializes all registers to 0, enables floating-point, initializes a trap vector, sets up a global pointer and stack pointer, and defines the `.tohost` label to terminate a program in simulation. `syscalls.c` defines the `printf` function, which invokes `putchar`, which makes a system call to write a character to the console via `stdout`. `syscalls.c` defines a few other essential library functions such as `memcpy()`, `memset()`, `strlen()`, `strcmp()`, `strcpy()`, and `exit()`.

3.4.5 Compiler Explorer

The Compiler Explorer at godbolt.org is a handy tool to quickly understand how code compiles without having to set up a Makefile and type a series of commands. Choose the architecture and compiler (e.g., RISC-V rv32gc gcc 10.2.0), and optionally specify compiler flags such as `-O` or `-O3`. Type your C code in the left pane. The disassembled output will appear in the right pane. Fig. 3.5(a) shows how a recursive factorial function is compiled with `-O` optimization. Fig. 3.5(b) shows how the compiler is clever enough to eliminate the recursive function call with `-O2` optimization.

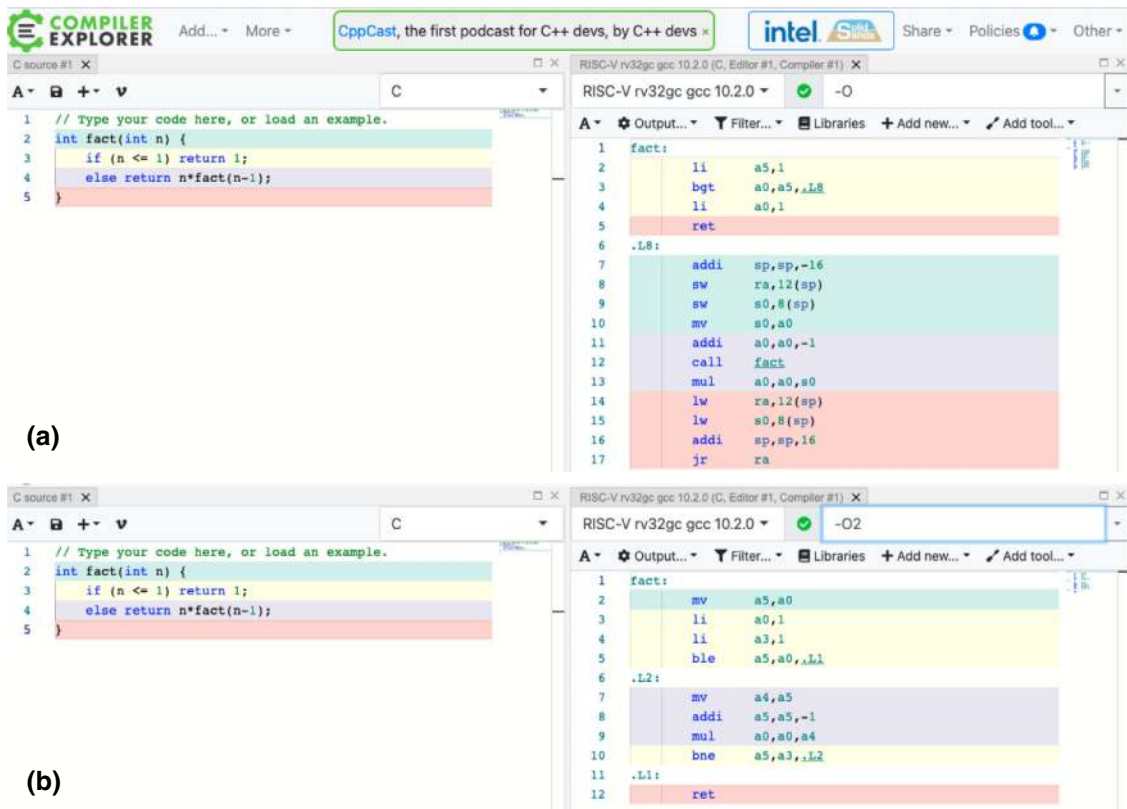


Fig. 3.5 Compiler Explorer factorial output: (a) `-O`, and (b) `-O2`. (© 2024 Matt Godbolt. Reprinted with permission.)

Sail is written in an object-oriented functional programming language called OCaml, which is a derivative of a language called ML used in theorem provers. The Sail compiler can be downloaded from its GitHub page, [github.com/rem-s-project/sail](https://github.com/REMS-project/sail).

Sail descriptions of MIPS, ARM, and IBM Power ISAs also exist. The holy grail of formal language specification and theorem proving would be to prove that an HDL implementation of the ISA is logically correct. Much research remains before this can be automated for complex cores.

`sail_riscv_sim` uses a JSON configuration file to control supported RISC-V extensions and other implementation specific parameters. The default configuration can be viewed with the `--print-default-config` flag. The simulator defaults to RV64 with most extensions enabled and matches Spike's implementation choices. To specify the configuration, for example the Wally rv64gc configuration, run:

```
sail_riscv_sim
--config $WALLY/tests/
riscov/sail_cSim/
rv64gc.json $WALLY/
examples/asm/sumtest/
sumtest
```

3.5 SAIL SIMULATION

The official formal and executable specification of the RISC-V architecture is written in a language called Sail [RISCV-SAIL]. Indeed, if the English specification is ever ambiguous or deviates from the Sail specification, Sail takes precedence. [Code Example 3.19](#) shows the Sail description of RISC-V I-type instructions. Sail is a pseudocode-like language for describing ISAs. Sail serves several purposes:

- ▶ Documents the behavior of each instruction in an ISA in a precise and readable way.
- ▶ Automatically creates a simulator that exactly matches the specification.
- ▶ Generates definitions for theorem provers aiming to prove properties about an ISA or implementation.

Sail produces a RISC-V simulator called `sail_riscv_sim`. The simulator is compiled in C++ based on the Sail specification, so it is also known as `sail_cSim`. We use the term “Sail” colloquially for this simulator that is automatically produced from the Sail language description of RISC-V.

The following example (see the next two pages) shows an excerpt of `sail_riscv_sim` running `sumtest` from [Section 3.3.10](#) and logging the signature to `sumtest.sig`. On each instruction, Sail prints the instruction number, processor mode (e.g., [M] for machine), program counter, and machine and assembly language instruction. It then prints any registers or memory that are modified. The signature `testsum.sig` contains the sum (0A) and cycle count (1D) from the `begin_signature` memory, represented as 8-byte double-words printed 4 bytes per line.

The Sail RISC-V simulator runs at about 300,000 instructions per second, and substantially slower for floating-point instructions. It is a bottleneck for the RISCOF test signature generation, as will be described in [Section 3.7.2](#).

Code Example 3.19 SAIL DESCRIPTION OF RISC-V I-TYPE INSTRUCTIONS

```
// Excerpted from:
// github.com/riscv/sail-riscv/blob/master/model/riscv_insts_base.sail
function clause execute (ITYPE (imm, rs1, rd, op)) = {
  let immext : xlenbits = sign_extend(imm);
  X(rd) = match op {
    RISCV_ADDI => X(rs1) + immext,
    RISCV_SLTI => zero_extend(bool_to_bits(X(rs1) <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(X(rs1) <_u immext)),
    RISCV_ANDI => X(rs1) & immext,
    RISCV_ORI  => X(rs1) | immext,
    RISCV_XORI => X(rs1) ^ immext
  };
  RETIRE_SUCCESS
}
```

```

$cd $WALLY/examples/asm/sumtest
$sail_riscv_sim sumtest --test-signature sumtest.sig --trace-all
using sumtest.sig for test-signature output.
ELF Entry @ 0x80000000
HTIF located at 0x80001000
begin_signature: 0x80004008
end_signature: 0x80004018
CSR mstatus (0x300) <- 0x00000000A00000000
CSR misa (0x301) <- 0x800000000034112F
CSR mcause (0x342) <- 0x0000000000000000

[0] [M]: 0x00000000000001000 (0x00000297) auipc x5, 0x0
x5 <- 0x00000000000001000

. . . More bootloader code, jumping to 0x80000000

[5] [M]: 0x00000000080000000 (0x00005117) auipc x2, 0x5
x2 <- 0x00000000080005000

[6] [M]: 0x00000000080000004 (0x20010113) addi x2, x2, 0x200
x2 <- 0x00000000080005200

[7] [M]: 0x00000000080000008 (0x00004297) auipc x5, 0x4
x5 <- 0x00000000080004008

[8] [M]: 0x0000000008000000C (0xFF828293) addi x5, x5, -0x8
x5 <- 0x00000000080004000

[9] [M]: 0x00000000080000010 (0x0002B503) ld x10, 0x0(x5)
mem[R,0x00000000080004000] -> 0x0000000000000004
x10 <- 0x0000000000000004

[10] [M]: 0x00000000080000014 (0xC0202C73) csrres x24, instret, x0
CSR instret (0xC02) -> 0x0000000000000000A
x24 <- 0x0000000000000000A

[11] [M]: 0x00000000080000018 (0x02C000EF) jal x1, 0x2c
x1 <- 0x0000000008000001C

[12] [M]: 0x00000000080000044 (0x1141) c.addi x2, -0x10
x2 <- 0x000000000800051F0

[13] [M]: 0x00000000080000046 (0xE022) c.sdsp x8, 0x0(x2)
mem[W,0x000000000800051F0] <- 0x0000000000000000

[14] [M]: 0x00000000080000048 (0xE426) c.sdsp x9, 0x8(x2)
mem[W,0x000000000800051F8] <- 0x0000000000000000

[15] [M]: 0x0000000008000004A (0x4401) c.li x8, 0x0
x8 <- 0x0000000000000000

[16] [M]: 0x0000000008000004C (0x4485) c.li x9, 0x1
x9 <- 0x0000000000000001

[17] [M]: 0x0000000008000004E (0x00954563) blt x10, x9, 0xa

[18] [M]: 0x00000000080000052 (0x9426) c.add x8, x9
x8 <- 0x00000000000000001

[19] [M]: 0x00000000080000054 (0x0485) c.addi x9, 0x1
x9 <- 0x00000000000000002

[20] [M]: 0x00000000080000056 (0xBFE5) c.j -0x8

```

```

. . . More loops of sum.S

[34] [M]: 0x0000000080000058 (0x8522) c.mv x10, x8
x10 <- 0x000000000000000A

[35] [M]: 0x000000008000005A (0x6402) c.ldsp x8, 0x0(x2)
mem[R,0x00000000800051F0] -> 0x0000000000000000
x8 <- 0x0000000000000000

[36] [M]: 0x000000008000005C (0x64A2) c.ldsp x9, 0x8(x2)
mem[R,0x00000000800051F8] -> 0x0000000000000000
x9 <- 0x0000000000000000

[37] [M]: 0x000000008000005E (0x0141) c.addi x2, 0x10
x2 <- 0x0000000080005200

[38] [M]: 0x0000000080000060 (0x8082) c.jr x1

[39] [M]: 0x000000008000001C (0xC0202CF3) csrwr x25, instret, x0
CSR instret (0xC02) -> 0x0000000000000027
x25 <- 0x0000000000000027

[40] [M]: 0x0000000080000020 (0x418C8CB3) sub x25, x25, x24
x25 <- 0x000000000000001D

[41] [M]: 0x0000000080000024 (0x00004297) auipc x5, 0x4
x5 <- 0x0000000080004024

[42] [M]: 0x0000000080000028 (0xFE428293) addi x5, x5, -0x1c
x5 <- 0x0000000080004008

[43] [M]: 0x000000008000002C (0x00A2B023) sd x10, 0x0(x5)
mem[W,0x0000000080004008] <- 0x000000000000000A

[44] [M]: 0x0000000080000030 (0x0192B423) sd x25, 0x8(x5)
mem[W,0x0000000080004010] <- 0x000000000000001D

[45] [M]: 0x0000000080000034 (0x00001317) auipc x6, 0x1
x6 <- 0x0000000080001034

[46] [M]: 0x0000000080000038 (0xFCC30313) addi x6, x6, -0x34
x6 <- 0x0000000080001000

[47] [M]: 0x000000008000003C (0x4285) c.li x5, 0x1
x5 <- 0x0000000000000001

[48] [M]: 0x000000008000003E (0x00533023) sd x5, 0x0(x6)
htif[0x0000000080001000] <- 0x0000000000000001
htif-syscall-proxy cmd: 0x0000000000000001
mem[W,0x0000000080001000] <- 0x0000000000000001

SUCCESS

$ more sumtest.sig
0000000a
00000000
0000001d
00000000

```

3.6 QEMU SIMULATION

QEMU is a powerful open-source emulator that handles many different computer architectures including RISC-V. It performs just-in-time translation from the architecture being simulated to the architecture of the host computer running QEMU (e.g., from RISC-V to x86). Thus, it can run on the order of 1 billion instructions per second, which is fast enough to comfortably boot an OS and an order of magnitude faster than Spike. It also interfaces with GDB to facilitate debugging, and emulates peripherals (e.g., a timer) that are important for booting an OS. QEMU is not a golden model for RISC-V, but no known discrepancies exist.

QEMU is variously pronounced kyoo-EM-yoo, kee-moo, kwemu, or Q.E.M.U. It is an acronym for Quick EMulator.

3.6.1 Simulating with QEMU

Table 3.7 lists the options used in this chapter for running QEMU. To simulate `sumtest.c` using QEMU, do the following:

```
$ cd $WALLY/examples/asm/sumtest
$ qemu-system-riscv64 -S -M virt -nographic -bios none -kernel sumtest
```

The following excerpt (below) shows common QEMU commands, as summarized in Table 3.8. Just after invoking QEMU, it will appear to hang. Pressing Ctrl-a and then c enters the monitor. `info registers` prints the contents of all registers, showing that the PC is at 0x1000, where it runs a small bootloader as Spike does. `x` displays memory. Recall from Code Example 3.9 that the signature is two double words stored at address 0x80002008. Using `x /2xg 0x80002008`, we see the signature is initialized to 0xdeadbeefdeadbeef. Press c to continue. QEMU will automatically halt at the infinite self_loop, as you can confirm with `info registers` showing the PC at 0x80000042. Now the signature shows a sum of 0x0a (10) and an `instret` value of 0x48470! The sum is correct, but the number of instructions retired is wildly incorrect. QEMU by default relies on the host processor's performance counters, which run at a much different rate. Finally, q terminates the simulation.

```
Ctrl-a c
QEMU 6.0.91 monitor - type 'help' for more information
(qemu) info registers
pc          0000000000001000
... <other registers>
(qemu) x /2xg 0x80002008
0000000080002008: 0xdeadbeefdeadbeef
(qemu) c
(qemu) info registers
pc          0000000080000042
... <other registers>
(qemu) x /2xg 0x80002008
0000000080002008: 0x000000000000000a 0x00000000000048470
(qemu) q
```

As you are learning to use QEMU, it will often seem to lock up when you press the wrong keys. Kill it with `killall` in another terminal window:

```
$ killall qemu-system-riscv64
```


Table 3.7 QEMU run options.

Run Option	Description
-S	Start QEMU with the PC paused at the beginning of the program
-M virt	Use a generic virtual RISC-V platform instead of a specific board
-nographic	Disable graphical window support and redirect the virtual serial port to display on the console
-bios none	Skip loading the OpenSBI BIOS and run in bare metal mode
-kernel sumtest	Load <code>sumtest.c</code> program into memory so it's ready to execute
-icount 0	Enable the virtual instruction counter to get accurate cycle counts
-gdb tcp::1234	Interface QEMU with GDB to single step through the program

Table 3.8 QEMU commands.

Command	Description
Ctrl-a, then c	Enter monitor (right after QEMU starts)
info registers	Print all general-purpose registers
x <address>	Display memory at <address>
x /2xg <address>	Display two double words of memory in hexadecimal format
c	Continue running program
q	Terminate simulation

Learn more from QEMU's documentation at www.qemu.org/docs/master. The Invocation and QEMU Monitor sections are of particular relevance.

We could use the `-icount 0` option to enable the virtual instruction counter to get accurate cycle counts, at a cost in simulation speed. Omit the `-S` flag to start immediately. After these modifications, shown below, see that `instret` is `0x1d`, in agreement with Spike.

```
$ qemu-system-riscv64 -M virt -nographic -bios none -kernel
sumtest -icount 0
Ctrl-a c
QEMU 6.0.91 monitor - type 'help' for more information
(qemu) x /2xg 0x80002008
0000000080002008: 0x000000000000000a 0x000000000000001d
(qemu) q
```

3.6.2 Debugging With GDB

QEMU can interface with the GDB debugger to single step through a program or use breakpoints. The interface is made with a TCP network connection to the QEMU virtual machine.

The authoritative book on GDB is freely available [Stallman23].

Invoke QEMU with a `-gdb tcp::<port>` option. Usually port 1234 is fine, but if somebody else is using it, pick a different port in the range 1024–49151.

```
$ qemu-system-riscv64 -S -M virt -nographic -bios none -kernel sumtest -gdb tcp::1234
```

Then invoke GDB in another terminal, connect it over the same port, and load the symbols from `sumtest` using this command:

```
$ riscv64-unknown-elf-gdb -ex "target remote :1234" -iex "file sumtest"
```

The following transcript (see the next two pages) shows an example debugging session, and [Table 3.9](#) summarizes common GDB commands. With the symbol table, we can refer to an address by name, such as `&begin_signature`. In the transcript, we see the simple bootloader jump to the main program at `0x80000000`, and GDB prints the associated source code. At the end of the transcript, we see the signature contains the expected sum of `0x0a` and the registers contain expected values.

Table 3.9 GDB commands.

Command	Description
<code>info all-reg</code>	Print contents of all integer registers
<code>x <address></code>	Print memory at address
<code>x <name></code>	Print memory starting at name in symbol table
<code>stepi</code>	Single step through program
Enter	Press enter key to repeat previous command
<code>b <function></code>	Set a breakpoint at a function, specified by its name
<code>b <address></code>	Set a breakpoint at instruction address
<code>c</code>	Continues running
Ctrl-c	Pause program
<code>set \$pc=<address></code>	Set PC to <address> (for example, to <code>0x1000</code> to restart program)
<code>q</code>	End debug session

GNU gdb (GDB) 10.1

... This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv64-unknown-elf".

...
Reading symbols from sumtest...

Remote debugging using :1234

0x0000000000001000 in ?? ()

(gdb) info all-reg

zero	0x0	0
ra	0x0	0x0
sp	0x0	0x0
gp	0x0	0x0
tp	0x0	0x0
t0	0x0	0

...
--Type <RET> for more, q to quit, c to continue without paging--q

Quit

(gdb) display /i \$pc

1: x/i \$pc

=> 0x1000: auipc t0,0x0

(gdb) x/2xg &begin_signature

0x80002008: 0xdeadbeefdeadbeef 0xdeadbeefdeadbeef

(gdb) step

Cannot find bounds of current function

(gdb) stepi

0x0000000000001004 in ?? ()

1: x/i \$pc

=> 0x1004: addi a2,t0,40

(gdb)

0x0000000000001008 in ?? ()

1: x/i \$pc

=> 0x1008: csrr a0,mhartid

(gdb)

0x000000000000100c in ?? ()

1: x/i \$pc

=> 0x100c: ld a1,32(t0)

(gdb)

0x0000000000001010 in ?? ()

1: x/i \$pc

=> 0x1010: ld t0,24(t0)

(gdb)

0x0000000000001014 in ?? ()

1: x/i \$pc

=> 0x1014: jr t0

(gdb)

rvtest_entry_point () at sumtest.S:6

6 la sp, toposfstack # Initialize stack pointer

1: x/i \$pc

=> 0x80000000 <rvtest_entry_point>: auipc sp,0x3

(gdb)

0x0000000080000004 in rvtest_entry_point () at sumtest.S:6

6 la sp, toposfstack # Initialize stack pointer

1: x/i \$pc

=> 0x80000004 <rvtest_entry_point+4>: addi sp,sp,512

(gdb)

7 la t0, N # Get address of N in data

1: x/i \$pc

=> 0x80000008 <rvtest_entry_point+8>: auipc t0,0x2

(gdb)

0x000000008000000c 7 la t0, N # Get address of N in data

1: x/i \$pc

=> 0x8000000c <rvtest_entry_point+12>: addi t0,t0,-8

(gdb)

```

8          ld a0, 0(t0)          # Load N
1: x/i $pc
=> 0x80000010 <rvtest_entry_point+16>:  ld      a0,0(t0)
(gdb)
9          csrr s8, instret      # Count instructions before call
1: x/i $pc
=> 0x80000014 <rvtest_entry_point+20>:  rdinstret s8
(gdb)
10         jal sum              # Call sum(N)
1: x/i $pc
=> 0x80000018 <rvtest_entry_point+24>:  jal      ra,0x80000044 <sum>
(gdb)
sum () at sum.S:16
16         addi sp, sp, -16      # Make room to save s0 and s1 on the stack
1: x/i $pc
=> 0x80000044 <sum>:                addi      sp,sp,-16
(gdb) b self_loop
Breakpoint 1 at 0x80000042: file sumtest.S, line 23.
(gdb) c
Continuing.

Breakpoint 1, self_loop () at sumtest.S:23
23         j self_loop          # Wait
1: x/i $pc
=> 0x80000042 <self_loop>:  j      0x80000042 <self_loop>
(gdb) x/2xg &begin_signature
0x80002008:  0x0000000000000000a      0x000000017658e47a8
(gdb) info all-reg
zero      0x0                0
ra         0x8000001c         0x8000001c <rvtest_entry_point+28>
sp         0x80003200         0x80003200
gp         0x0                0x0
tp         0x0                0x0
t0         0x1                1
t1         0x80001000         2147487744
t2         0x0                0
fp         0x0                0x0
s1         0x0                0
a0         0xa                10
a1         0x87000000         2264924160
a2         0x1028             4136
(gdb) set $pc=0x1000
(gdb) b sum
Breakpoint 1 at 0x80000044: file sum.S, line 16.
(gdb) c
Continuing.

Breakpoint 1, sum () at sum.S:16
16         addi sp, sp, -16      # make room to save s0 and s1 on the stack
(gdb) stepi
17         sd s0, 0(sp)
(gdb) q
A debugging session is active.

    Inferior 1 [process 1] will be detached.

Quit anyway? (y or n) y
Detaching from program: /home/ben/cvw/examples/asm/sumtest/sumtest, process 1
Ending remote debugging.
[Inferior 1 (process 1) detached]

```

To simulate a C program, change `-g` to `-gdwarf-2` in the Makefile to generate appropriate debugging symbols for C variables. Then type the following commands:

```
$ cd $WALLY/examples/C/sum
$ qemu-system-riscv64 -M virt -nographic -bios none -kernel sum -S -gdb tcp::1234
```

In another terminal, type:

```
$ riscv64-unknown-elf-gdb -iex "file sum" -ex "target remote:1234"
```

See the following transcript. QEMU hangs on the `printf` statement.

```
GNU gdb (GDB) 10.1
...
Reading symbols from sum...
Remote debugging using :1234
0x0000000000001000 in ?? ()
(gdb) b main
Breakpoint 1 at 0x8000201a: main. (2 locations)
(gdb) c
Continuing.

Breakpoint 1, main () at sum.c:17
17 int main(void) {
(gdb) s
19     setStats(1);
(gdb) n
20     s[0] = sum(4);
(gdb) s
sum (N=N@entry=4) at sum.c:11
11     for (i=1; i<=N; i++) {
(gdb) p N
$1 = 4
(gdb) s
12         result = result + i;
(gdb) p i
$2 = 1
(gdb) s
11     for (i=1; i<=N; i++) {
(gdb) p result
$3 = 1
(gdb) s
12         result = result + i;
(gdb)
11     for (i=1; i<=N; i++) {
(gdb)
12         result = result + i;
(gdb)
11     for (i=1; i<=N; i++) {
(gdb)
12         result = result + i;
(gdb)
11     for (i=1; i<=N; i++) {
(gdb)
main () at sum.c:21
21     setStats(0);
```

```
(gdb) p s[0]
$4 = <optimized out>
(gdb)
$5 = <optimized out>
(gdb) n
22     printf("s = %d\n", s[0]);
(gdb) n
^C
Program received signal SIGINT, Interrupt.
0x00000000800020c2 in syscall (arg2=7, arg1=<optimized out>, arg0=1, which=64)
    at ../common/syscalls.c:28
28     while (fromhost == 0)
(gdb)
```

3.7 TEST SUITES

RISC-V International’s Architecture Test Special Interest Group (SIG) provides the RISCOF RISC-V Compatibility Framework, which includes the `riscv-arch-test` suite of tests that exercise every unprivileged RISC-V instruction. Passing these tests is a requirement to use the RISC-V trademark, but the tests are not sufficient to prove an implementation is completely correct. RISCOF compiles the `riscv-arch-test` tests and simulates them with `Sail` to produce signatures for each test. [Section 5.6](#) describes verifying the SystemVerilog model by running the same tests and comparing the signatures.

At the time of this writing, further test suites and formal verification methods are rapidly evolving. In the interim, we have created the `wally-riscv-arch-test` suite with additional tests, especially of privileged instructions, the memory management unit, and peripherals, described in [Chapters 8, 11, and 20](#).

Other more specialized Wally tests include floating-point tests, benchmarks, and booting Linux, described in [Chapters 16, 21, and 22](#) and [Appendix F](#).

Most of the test suites come as source code, which requires compilation. To compile all the test cases, go to the top-level `cvw` directory and type:

```
$ make --jobs
```

[Section 5.11.1](#) describes how to run a regression to apply these tests to the Wally SystemVerilog model.

3.7.1 riscv-arch-test

`riscv-arch-test` provides test cases for the RV32I and RV64I instruction sets and various ratified extensions including C, M, F, D, Zb*, and Zk*, along with some basic privileged instruction tests [`riscv-arch-test`].

RISCOF is written in Python and enables testing of RISC-V processors and comparing their results to a reference model such as `Sail` or `Spike`.

Passing the RISCOF tests and having the results approved by RISC-V International is a prerequisite to licensing the RISC-V trademarks in connection with the design.

`make --jobs` launches jobs in parallel on a multicore computer, which is much faster than just `make`.

Each instruction has an assembly language test file that exercises the instruction and generates a signature containing the expected results. RISCOF compiles the test programs with GCC, runs them using Spike and Sail, and verifies that the signatures match.

riscv-arch-test is incorporated into the cvw repository as a git submodule so that improvements in the suite can easily be pulled into cvw.

3.7.1.1 Architectural Test Pool

`$WALLY/addins/riscv-arch-test/riscv-test-suite` contains the *architectural test pool* of tests for all the base RISC-V architectures including RV32I and RV64I. A tree of the most important files is given in Fig. 3.6. The current set of tests is designed to run in machine mode, so the top-level architectural test suite directories include `rv32i_m` and `rv64i_m`. Each test suite contains subdirectories such as `I`, `C`, `M`, and `F` for the core integer instructions and each of the optional extensions. It also contains a `privilege` directory that tests `ebreak`, `ecall`, and misaligned stores, loads, branches, and jumps. Each of these directories contains a `src` directory with assembly language test files for each test objective, such as `I/src/add-01.S`, that tests `add` instructions.

Linux lists a folder's hierarchical contents using the `tree` command, as shown in Fig. 3.6 when running `tree` in the `$WALLY/addins/riscv-arch-test/riscv-test-suite` directory. By the time you read this, more extensions will be in the test pool.

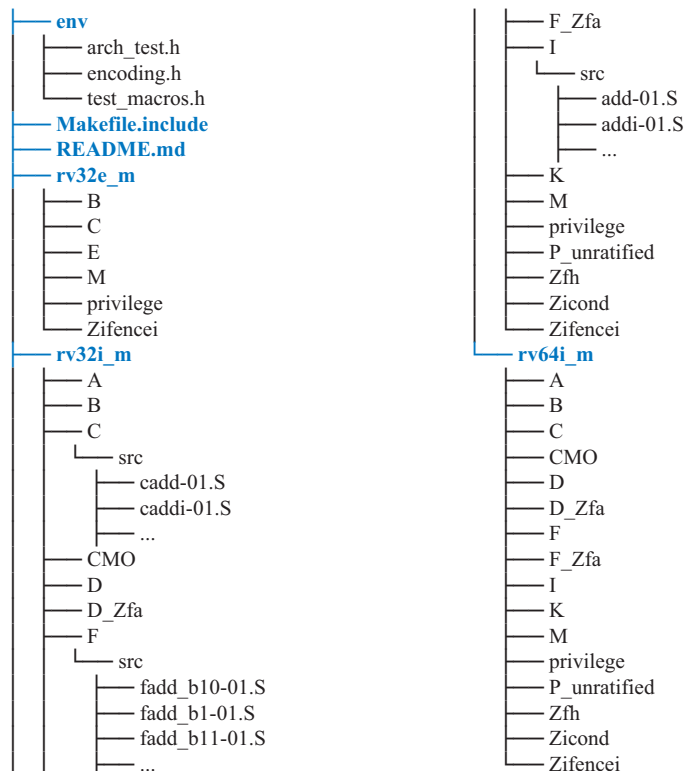


Fig. 3.6 riscv-test-suite organization.

3.7.1.2 Test File Structure

If you need to write additional tests, the structure of the test case source files is important. [Code Example 3.20](#) shows an excerpt of the `rv32i_m/I/src/add-01.S` file containing various macros. The code example has explanatory comments and disassembled code added in blue showing what the macros produce. The file tests two additions ($0x7ffffff + 1$ and $0x20000 + 0x20000$), writes the results to signature memory, and calls the HTIF to halt.

Users interested in testing instructions can gloss over most macros and just focus on the test cases, but if you are curious, you can learn about the macros at github.com/riscv-non-isa/riscv-arch-test/blob/main/spec/TestFormatSpec.adoc.

Code Example 3.20 `rv32i_m/src/add-01.S` EXCERPT

```
#include "model_test.h"
#include "arch_test.h"
RVTEST_ISA("RV32I")

.section .text.init
.globl rvtest_entry_point
rvtest_entry_point:
# 80000000:    feedc0b7          lui     ra,0xfeedc          # Initialize registers to predictable values
# 80000004:    ead08093          addi    ra,ra,-339 # feedbead <_end+0x7eed63a9>
# 80000008:    ff76e137          lui     sp,0xff76e
# 8000000c:    f5610113          addi    sp,sp,-170 # ff76df56 <_end+0x7f768452>
...
RVMODEL_BOOT          # Does nothing
RVTEST_CODE_BEGIN
RVTEST_CASE(0,"//check ISA:=regex(. *32.*);check ISA:=regex(. *I.*);def TEST_CASE_1=True;",add)

RVTEST_SIGBASE( x3,signature_x3_1) # Set x3 (gp) to point to signature memory
800000f8:    00000013          nop
800000fc:    00000013          nop
80000100:    00006197          auipc   gp,0x6
80000104:    01418193          addi    gp,gp,20 # 80006114 <signature_x3_0>
80000108:    00000013          nop
8000010c:    00000013          nop
80000110:    00000013          nop
80000114:    00000013          nop
80000118:    00000013          nop
8000011c:    00000013          nop
inst_0: # Test add x24, x4, x24 with x4 = 0x7fffffff, x24 = 0x00000001
// rs2 == rd != rs1, rs1==x4, rs2==x24, rd==x24, rs1_val > 0 and rs2_val > 0, rs2_val == 1,
// rs1_val == (2**(xlen-1)-1), rs1_val != rs2_val, rs1_val == 2147483647
// opcode: add ; op1:x4; op2:x24; dest:x24; op1val:0x7fffffff; op2val:0x1
TEST_RR_OP(add, x24, x4, x24, 0x80000000, 0x7fffffff, 0x1, x3, 0, x18)
80000120:    80000237          lui     tp,0x80000
80000124:    fff20213          addi    tp,tp,-1 # 7fffffff <absimm+0x3ad0dad0>
80000128:    00100c13          li      s8,1
8000012c:    01820c33          add     s8,tp,s8
80000130:    0181a023          sw     s8,0(gp)
```



```

inst_1: # Test add x28, x10, x10 with x10 = 0x00020000
// rs1 == rs2 != rd, rs1==x10, rs2==x10, rd==x28, rs1_val > 0 and rs2_val < 0, rs2_val == -257,
// rs1_val == 131072
// opcode: add ; op1:x10; op2:x10; dest:x28; op1val:0x20000; op2val:0x20000
TEST_RR_OP(add, x28, x10, x10, 0x40000, 0x20000, 0x20000, x3, 4, x18)
80000134:      00020537      lui      a0,0x20
80000138:      00020537      lui      a0,0x20
8000013c:      00a50e33      add      t3,a0,a0
80000140:      01c1a223      sw       t3,4(gp)
...
RVTEST_CODE_END
RVMODEL_HALT                                # Writes successful completion to HTIF
# 800032bc:      00100093      li       ra,1
# 800032c0 <write_tohost>:
800032c0:      00001297      auipc    t0,0x1
800032c4:      d412a023      sw       ra,-704(t0) # 80004000 <tohost>
800032c8:      ff9ff06f      j        800032c0 <write_tohost>
RVTEST_DATA_BEGIN                          # Room for initialized data values (for loads and stores)
.align 4
rvtest_data:
.word 0xbabecafe
RVTEST_DATA_END

RVMODEL_DATA_BEGIN                         # Start of signature
rvtest_sig_begin:
sig_begin_canary:
CANARY;

signature_x3_0:
.fill 0*(XLEN/32),4,0xdeadbeef            # Unused

signature_x3_1:
.fill 17*(XLEN/32),4,0xdeadbeef           # Room for 17 32-bit results

RVMODEL_DATA_END                           # End of signature

```

Code Example 3.21 shows the signatures produced by Sail and Spike, found in `tests/riscov/work/riscv-arch-test/rv32i_m/I/src/add-01.S/{ref/Reference-sail_c_simulator.signature and dut/DUT-spike.signature}`. Thus, the expected signature is `0x80000000` and `0x00040000`. To change the tests, replace the code between `RVTEST_SIGBASE` and `RVTEST_CODE_END` and increase the amount of signature storage at the `.fill` statement as necessary.

Code Example 3.21 `tests/riscov/work/riscv-arch-test/rv32i_m/I/src/add-01.S/ref/reference-sail_c_simulator.signature`

```

80000000
00040000

```

3.7.2 RISCOF

RISCOF (github.com/riscv-software-src/riscof) is a set of scripts that compile test cases and run them on Sail and Spike to generate signatures, which can then be used to compare signatures produced by an HDL processor model. RISCOF was developed by the Indian Institute of Technology (IIT) Madras and Incore and has become part of the RISC-V International riscv-arch-test flow. The Wally toolchain includes RISCOF. RISCOF depends on configuration files in `$WALLY/tests/riscof/spike`. It places the compiled tests and signatures in `$WALLY/tests/riscof/work`.

RISCOF is invoked by running `make` from `$WALLY/tests/riscof`. This Makefile is also indirectly invoked when running `make` from `$WALLY` or `$WALLY/sim`.

The Makefile compiles suites of tests listed in [Table 3.10](#). Each test references a configuration in the `riscof` directory, which in turn points to a YAML file in the `riscof/spike` subdirectory containing the instruction set architecture (RV{32/64}), extensions, and so on. [Code Example 3.22](#) shows part of the `rv32gc` YAML configuration. The configuration has one core that supports RV32GC (RV32IMAFDCZicsr) and various other extensions such as bit manipulation. The YAML file configures the value of the machine instruction set architecture register (`misa`) to indicate standard extensions supported (see [Section 8.2.3.2](#)): A, C, D, F, I, M, and XLEN = 32. RISCOF examines all the tests in the specified directory, making a list in `riscof/riscof_work/{CONFIG}/test_list.yaml` of those with an `RVTEST_ISA` macro compatible with the ISA configuration.

YAML (YAML Ain't Markup Language) is a generic language for describing hierarchical configuration information. Various RISC-V tools are configured with YAML files. YAML has similarities to JSON, but hierarchies are expressed by indenting rather than nested braces.

Code Example 3.22 `tests/riscof/spike/spike_rv32gc_isa.yaml` CONFIGURATION (FOR `config32.ini`)

```
hart_ids: [0]
hart0:
  ISA: RV32IMAFDCZicsr_Zicond_Zifencei_Zfa_Zfh_Zca_Zcb_Zba_Zbb_Zbc_Zbkb_Zbkc_Zbkx_Zbs_Zknd_Zkne_Zknh
  physical_addr_sz: 32
  User_Spec_Version: '2.3'
  supported_xlen: [32]
  misa:
    reset-val: 0x40001120
  rv32:
    accessible: true
    mxl:
      implemented: true
    type:
      warl:
        dependency_fields: []
        legal:
          - mxl[1:0] in [0x1]
        wr_illegal:
          - Unchanged
```

Table 3.10 Wally RISCOF test suites.

Suite	Configuration	Directory	Description
arch32	config32.ini	riscv-arch-test	Standard 32-bit compatibility tests
arch64	config64.ini	riscv-arch-test	Standard 64-bit compatibility tests
arch32e	config32e.ini	riscv-arch-test	Standard 32-bit compatibility tests for rv32e
wally32	config32.ini	wally-riscv-arch-test	Custom 32-bit tests: privileged, peripherals
wally64	config64.ini	wally-riscv-arch-test	Custom 64-bit tests: privileged, peripherals

Running RISCOF can take more than an hour on a slow machine. The largest part of the time comes from the slow Sail simulator producing signatures. Unfortunately, RISCOF is not yet smart enough to only process test cases that have changed.

Fig. 3.7 shows arch32 excerpts of the report generated by RISCOF. RISCOF compiles and runs the tests on both a “Device under test” (Spike) and a reference model (Sail_CSim) and checks that their signatures (`DUT-spike.signature` and `Reference-sail_c_simulator.signature`) match. It also invokes `elf2hex` to convert the ELF files into memfiles suitable to load into a Verilog testbench.

The Makefile copies the compiled tests and signatures into the `work` directory structured as shown in Fig. 3.8. The `dut` and `ref` directories each have an ELF, memfile, and signature. The `ref` directory also has an `.objdump`, which is convenient when debugging, and `.addr` and `.lab` files to associate labels with addresses in the Verilog testbench.

3.7.3 wally-riscv-arch-test

The `$WALLY/tests/wally-riscv-arch-test` directory is derived from `riscv-arch-test` but contains test cases written for this book that are not part of the `riscv-arch-test` suite. They include:

- ▶ Privileged tests, including traps, CSRs, and the MMU
- ▶ Peripheral tests

Sail does not presently model the peripherals used to generate interrupts, so it cannot be used to generate the expected values. Instead, scripts copy a `reference_output` file from `wally-riscv-arch-test` to the corresponding `riscov/work` directory. This means that the developer must predict the expected values and that there is no reference model to catch errors in these predictions.

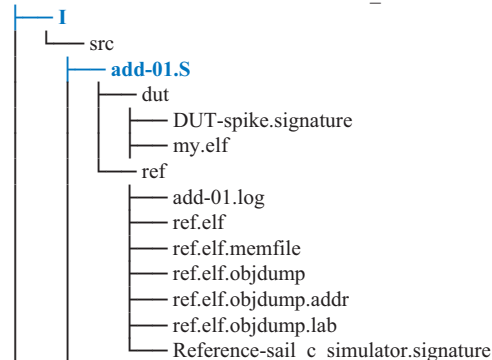
```

make -C ../../tests/riscov/
make[1]: Entering directory '/home/harris/cvw/tests/riscov'
mkdir -p ./riscov_work
mkdir -p ./work
mkdir -p ./work/riscv-arch-test
mkdir -p ./work/wally-riscv-arch-test
sed 's,(0),/home/harris/cvw/tests/riscov,g;s,(1),32gc,g' config.ini > config32.ini
sed 's,(0),/home/harris/cvw/tests/riscov,g;s,(1),64gc,g' config.ini > config64.ini
sed 's,(0),/home/harris/cvw/tests/riscov,g;s,(1),32e,g' config.ini > config32e.ini
riscov run --work-dir=./riscov_work --config=config32.ini --suite=../../addins/riscv-arch-test/riscv-test-suite/ --env=../../addins/riscv-arch-test/riscv-test-suite/env --no-browser
INFO | ***** RISCOF: RISC-V Architectural Test Framework 1.25.2 *****
INFO | using riscv_isac version : 0.11.0
INFO | using riscv_config version : 3.5.0
INFO | Reading configuration from: /home/harris/cvw/tests/riscov/config32.ini
INFO | Preparing Models
INFO | Input-ISA file
INFO | Loading input file: /home/harris/cvw/tests/riscov/spike/spike_rv32gc_isa.yaml
INFO | Load Schema /home/harris/.local/lib/python3.8/site-packages/riscv_config/schemas/schema_isa.yaml
INFO | Processing Hart: hart0
INFO | Initiating Validation
INFO | No errors for Hart: 0 :)
INFO | Updating fields node for each CSR
INFO | Initiating WARL legality checks.
INFO | Initiating post processing and reset value checks.
INFO | Initiating validation checks for indexed csrs
INFO | Initiating validation checks for shadow fields
INFO | Performing Checks on PMP CSRs
INFO | Dumping out Normalized Checked YAML: /home/harris/cvw/tests/riscov/riscov_work/spike_rv32gc_isa_checked.yaml
INFO | Input-Platform file
INFO | Loading input file: /home/harris/cvw/tests/riscov/spike/spike_platform.yaml
INFO | Load Schema /home/harris/.local/lib/python3.8/site-packages/riscv_config/schemas/schema_platform.yaml
INFO | Initiating Validation
INFO | No Syntax errors in Input Platform Yaml. :)
INFO | Dumping out Normalized Checked YAML: /home/harris/cvw/tests/riscov/riscov_work/spike_platform_checked.yaml
INFO | Generating database for suite: /home/harris/cvw/addins/riscv-arch-test/riscv-test-suite
INFO | Database File Generated: /home/harris/cvw/tests/riscov/riscov_work/database.yaml
INFO | Env path set to/home/harris/cvw/addins/riscv-arch-test/riscv-test-suite/env
INFO | Running Build for DUT
INFO | Running Build for Reference
INFO | Selecting Tests.
INFO | Running Tests on DUT.
INFO | Running Tests on Reference Model.
INFO | Initiating signature checking.
INFO | Following 387 tests have been run :
INFO | TEST NAME : COMMIT ID : STATUS
INFO | /home/harris/cvw/addins/riscv-arch-test/riscv-test-suite/rv32i_m/1/src/add-01.S : -: Passed
INFO | /home/harris/cvw/addins/riscv-arch-test/riscv-test-suite/rv32i_m/1/src/add1-01.S : -: Passed
...
INFO | Test report generated at /home/harris/cvw/tests/riscov/riscov_work/report.html.
rsync -a ./riscov_work/rv32i_m/ ./work/riscv-arch-test/rv32i_m/ || echo "error suppressed"
...
riscv64-unknown-elf-elf2hex --bit-width 32 --input ../../tests/riscov/work/riscv-arch-test/rv32i_m/1/src/add-01.S/ref/ref.elf --output
../../tests/riscov/work/riscv-arch-test/rv32i_m/1/src/add-01.S/ref/ref.elf.memfile
...

```

Fig. 3.7 RISCOF log for arch32 suite.

cvw/tests/riscov/work/riscv-arch-test/rv32i_m



Similar directories for C, M, F, D, etc.

Similar directories for each instruction

Spike device under tests files
 Signature from Spike
 ELF for Spike
 Sail reference files
 Log file from Sail simulation
 ELF for Sail
 Machine language code from ELF for Verilog sim
 Disassembled ELF
 Addresses of labels in ELF for Verilog sim
 Names of labels in ELF for Verilog sim
 Signature from Sail

Fig. 3.8 RISCOF work output.

To add new tests to regression in `tests/wally-riscv-arch-test`:

- ▶ Create the test assembly language `.S` file following the form of other tests.
 - ▶ Preserve the existing `riscv-arch-test` macros.
 - ▶ Edit the `RVTEST_ISA` and `RVTEST_CASE` lines to match your required extensions (e.g., `RV64I_Zicsr`).
- ▶ Add the tests to a test suite in `tests.vh`.
- ▶ Add the test suite to the testbench if it is new.
- ▶ Add the test suite to regression.
- ▶ If Sail can simulate the test correctly (e.g., doesn't depend on interrupts or peripherals):
 - ▶ Run `make` in the `sim` directory, invoking RISCOF to compile the tests and generate `Reference-sail_c_simulator.signature`.
- ▶ Otherwise:
 - Include `"NO_SAIL=True"` in the `RV_TEST_CASE` (see privileged tests for an example).
 - ▶ Write the expected values in `riscv-test-suite/rv{32/64}i_m/<testtype>/references/testname.reference_output`.
 - ▶ Run `make` in the `sim` directory, invoking RISCOF to compile the tests without running Sail.
- ▶ Run `regression-wally` to run the tests.
- ▶ Inspect the object dump and signature in `tests/riscov/work/wally-riscv-arch-test`.

3.8 SUMMARY

This chapter has described the RISC-V toolchain, simulators, and test suites used in an open-source Linux-based development environment widely used by professional developers. GNU's `gcc` provides an assembler, linker, and C compiler that support many architectures including RISC-V. It outputs RISC-V programs as ELF files that also contain headers, debug information, and the symbol table. `objdump` disassembles ELF files to assembly language.

Spike, Sail, and QEMU are three standard RISC-V simulators. Spike was the original golden RISC-V simulator and remains the easiest to use. Sail is a newer simulator that also serves as RISC-V's formal executable architecture specification. QEMU is an extremely fast open-source emulator, and it interfaces nicely with GDB for debugging. This chapter also showed how to use the toolchain and simulators to compile RISC-V verification test suites and generate expected results.

HDL Design Practices

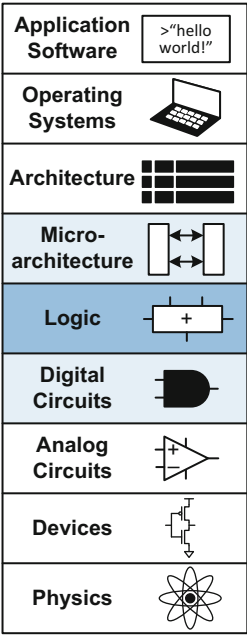
4

- 4.1 HDL Design Methodology
- 4.2 Hardware Design with SystemVerilog
- 4.3 Wally Style Guidelines
- 4.4 Wally Configuration
- 4.5 Summary

Digital designers need high-productivity design methods to develop complex systems-on-chip (SoCs) in a reasonable amount of time. Currently, most designs are specified using *hardware description languages (HDLs)*, such as SystemVerilog and VHDL, that offer a textual representation of hardware. The HDL code can be simulated in a *logic simulator*, such as Siemens Questa, Synopsys VCS, Cadence Xcelium, or open-source Verilator, to *verify* that it performs the intended function and then synthesized with a *logic synthesis* tool, such as Synopsys Design Compiler, AMD Vivado, or Intel Quartus, to map the design onto a library of standard cells for a custom chip, also called an ASIC (application-specific integrated circuit), or onto configurable logic in an FPGA (field programmable gate array).

An advantage of HDLs over schematics is that the logic can be specified at a higher level of abstraction called *behavioral* or *register transfer level (RTL)*. An RTL description focuses on the registers and the datapath elements, such as adders, shifters, and multiplexers, rather than individual logic gates or transistors. For example, the designer can write “+” for addition instead of drawing a chain of logic gates that perform addition, or they could describe an FSM based on its state transitions rather than in terms of Boolean equations. Moreover, the logic synthesis tool can automatically pick the most appropriate implementation for given constraints. For example, adders can be built using ripple carry, carry lookahead, or prefix tree architectures, with increasing speed but also increasing cost. Given timing constraints, a logic synthesis tool will try to pick the lowest-cost implementation that meets timing.

The Wally processor uses SystemVerilog [IEEE1800-2023], a modern flavor of Verilog. Verilog is the most widely used HDL in the industry. SystemVerilog fixes some syntactical annoyances of Verilog (e.g., the reg data type, which might or might not be a register) and adds powerful capabilities for verification that are beyond the scope of this book. SystemVerilog is nearly interchangeable with Verilog and is well supported in



modern tools, so there is little reason to write new code in classic Verilog anymore. SystemVerilog has many nonsynthesizable constructs for writing testbenches, and the SystemVerilog standard is not explicit about what is synthesizable or not. This chapter emphasizes the standard idioms to describe hardware. At times, this book uses the names *Verilog* and *SystemVerilog* interchangeably.

The leading competitor to Verilog is VHDL [IEEE1076-2019], which once was dominant in Europe and at US military contractors but is steadily declining in market share [Golson16]. The synthesizable subset of VHDL is essentially interchangeable with SystemVerilog, but idiosyncrasies of the language lead to about 50% more code to describe the same hardware. SystemC [IEEE1666-2011] is another widely supported language suitable for simulation and synthesis. It is a C++ class library, offering a higher level of abstraction that can reduce design time for sophisticated users. Leading simulation and synthesis tools support a mix of all three languages. Other HDLs such as Chisel, SpinalHDL, and Bluespec promise to offer higher productivity but have not yet gained large market share.

This chapter begins with an overview of methodologies for designing with HDLs. It provides a tutorial about designing hardware using SystemVerilog, including combinational and sequential logic and memories. It examines how to use SystemVerilog to describe configurable systems such as the Wally processor. It then discusses coding style guidelines used on Wally. [Chapters 5](#) and [6](#) address logic verification and synthesis with SystemVerilog.

4.1 HDL DESIGN METHODOLOGY

An SoC is too big for a single person to design in a competitive timeframe and too complex to keep in one's head in its entirety. Designers have developed several techniques for managing this complexity.

4.1.1 Design Strategies

Sophisticated design teams write specifications before writing code. The specifications define the system functions and design hierarchy. A good specification helps all members of a team understand how the system functions and where different features are implemented. Getting the hierarchy right the first time reduces effort repartitioning the system later. However, the ultimate goal of the project is working code, not polished documentation, so managers must remember that the specifications are merely a means toward an end. Less experienced designers often lack a frame of reference to write a proper specification and have to write some code first to understand the system before documenting it later. Startup companies also commonly omit specifications, rushing to get a minimally viable

product before funding runs out. Startups often have to bet on the skill of a small team of designers who keep the design in their heads, hoping that none of the designers get hit by a truck before the product is ready.

After the requirements are defined, start by thinking of the design with block diagrams. Begin with a top-level block diagram and draw the inputs and outputs. Early in the design phase, it may be most practical to focus on the major input and output busses and leave their associated control signals for later. Then, sketch the sub-blocks and how they connect to the primary inputs and outputs and to internal signals between sub-blocks. Repeat this process hierarchically for each sub-block until it is simple enough to understand. Often, different blocks are owned by different designers and communicate via interfaces specified early in the design process. Later chapters of this book show hierarchical block diagrams of Wally's units and subsystems.

After you get down to simple blocks, design them in terms of hardware components such as registers, multiplexers, ALUs, memories, and combinational logic blocks. The most common beginner mistake is to write HDL like a program, and the more programming experience one has, the easier it is to make this mistake.

To obtain good power, performance, and area, and to minimize design errors, designers use only a synthesizable subset of SystemVerilog where they can readily understand the hardware implied by their code. For each hardware component, write the corresponding idiom from [Section 4.2](#). If you aren't sure what hardware you intend to imply with some code, you are unlikely to get good results from that code. [Section 4.3](#) addresses style guidelines that avoid common Verilog pitfalls and produce clear code.

Two of the most common design methodologies are waterfall and agile. *Waterfall* project management is a traditional approach drawn from mature industries such as construction. The project is divided into tasks, such as conception, initiation, design, building, testing, and deployment. Each step is reviewed and then frozen before moving on to the next step. It works well when the time and cost of each step can be predicted well from experience and when risk is low. Software projects were historically based on the waterfall model; for example, the US Defense Department DOD-STD-2167A standard of 1985 mandated this approach for defense contracts. Because testing of the overall design does not occur until much later in the project when using this method, major flaws may not be revealed until late in the process when they are most expensive to fix, and sometimes the problems in the underlying design are insurmountable so late in the process [Keating02]. In contrast, *agile* project management is an iterative design process that emphasizes delivering frequent working prototypes, getting rapid customer feedback, and embracing changing requirements. Key features are implemented first, problems are discovered

early, and the design adapts to work around the problems. Except when an SoC is a minor update to an existing product, agile methods tend to be most appropriate for SoC design. Wally was developed using an agile methodology.

4.1.2 Version Control

Version control is the practice of tracking and managing changes to code. Version control systems maintain a database, called a *repository*, of all of the code files, tracking every modification. The main branch of the code is intended to be stable and robust. Developers create other branches as they write and test new features, then merge them into the main branch when ready. Merging is smart enough that different developers can modify different parts of a file concurrently without stomping on each other. If a bug is introduced, version control makes it easy to roll back to the last working version of the system and to identify exactly what changes have been made since that working version. Open-source development is driven by public repositories managed by a team of *committers* who review and accept contributions.

Version control is essential for complex projects, and Git has become the version control system of choice for most organizations that aren't deeply entrenched in some other tool. Wally uses Git for version control, as discussed in Appendix C. GitHub is a free and widely used cloud service for hosting Git repositories.

Even when you are working on a small project alone, version control provides file backups, an easy way to roll back to the last working design, and a repository to store and share your work. When developers get accustomed to version control, they rarely go back.

4.1.3 Reuse

The most effective strategy for controlling design cost is to reuse proven components. The SoC design team typically licenses cell, memory, and input/output (I/O) libraries from the chip foundry or reuses libraries developed internally. The team then adds components such as processor cores, peripherals, and accelerators. ARM and RISC-V are leading SoC processors. In 2023, SoCs shipped more than 30 billion ARM cores [Statista25]. As of 2022 an estimated 10 billion RISC-V cores had shipped [RISCV23], and RISC-V is increasing exponentially. The cores range in complexity from tens of thousands to tens of millions of transistors and vary in performance over two orders of magnitude. Peripherals include double-data-rate (DDR) and Flash memory controllers, radios for Wi-Fi, Bluetooth, or cellular communication, wired Ethernet and serial links, timers, and analog/digital and digital/analog data converters. Accelerators include

A student with a rich set of code in their public GitHub repositories and a history of making contributions to other projects has a substantial edge in an interview!

graphics processing units (GPUs); video codecs; and specialized blocks for artificial intelligence, cryptography, and networking. Accelerators operate in parallel with processors and typically offer two orders of magnitude better performance and energy efficiency than general-purpose processors programmed for the same task. Most of these components are licensed from third parties or reused from a company’s proprietary library. The design team may add some “special sauce” blocks customized to the target application and then integrates all the parts onto an SoC, verifies the system, and develops software to run on the chip.

4.2 HARDWARE DESIGN WITH SYSTEMVERILOG

A good hardware designer visualizes the desired hardware (e.g., registers, memories, FSMs, and combinational logic blocks) and then writes HDL idioms that imply such hardware. This section reviews these idioms. See also [Harris21] for a side-by-side introduction to SystemVerilog and VHDL. The most common novice error when using hardware description languages is to think of them as software programming and to simply write code rather than visualize hardware.

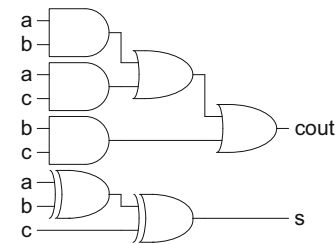
We design logic *hierarchically* or *structurally* by decomposing a complex system into modules with well-defined inputs and outputs. Each of the modules is recursively decomposed into smaller modules until the modules are simple enough that they can be described *behaviorally*.

Behavioral HDL (RTL) code defines the function performed by the module. The designer must first consider whether a piece of behavioral logic is combinational or sequential. Whereas *combinational logic* outputs depend only on current input values, *sequential logic* outputs depend on current and previous input values and thus have memory. For ease of design, we rely on the *synchronous sequential discipline*, in which the only sequential elements are edge-triggered flip-flops or memories.

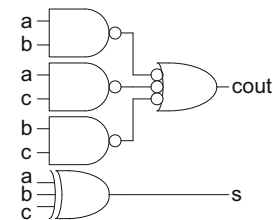
4.2.1 Modules

Code Example 4.1 shows the SystemVerilog for a full adder. It begins with the keyword `module` and then the name of the module (`fulladder`). It declares the three inputs and two outputs. Note that signals in synthesizable SystemVerilog are declared of type `logic`, which can represent 0, 1, z (floating), or x (uninitialized or invalid). The outputs are described with `assign` statements specifying Boolean equations using `&` (and), `|` (or), `^` (xor). **Fig. 4.1** shows various possible implementations. The synthesis tool chooses the lowest-cost solution that meets timing requirements in a target cell library.

SystemVerilog has many features not covered in this tutorial. Some, such as `$display`, are intended for writing testbenches and are not synthesizable. Others, such as `reg` and `wire`, are historical baggage and shouldn’t be used in current designs. Except in testbenches, we recommend using the synthesizable subset of the language described in this chapter. A substantial fraction of the Verilog code you will discover in an Internet search is bad or has subtle errors, so it is much better to think of the hardware you want and then write the appropriate idiom. If you aren’t sure of what hardware you want, you are likely to write bad code too.



(a)



(b)

a	b	c	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

a	b	c	cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(c)

Fig. 4.1 fulladder implementations: (a) basic logic gates, (b) optimized logic gates, and (c) FPGA lookup tables.

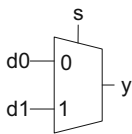


Fig. 4.2 mux2.

Code Example 4.1 fulladder

```
module fulladder(input  logic a, b, c,
                 output logic s, cout);

    assign s = a ^ b ^ c;
    assign cout = (a & b) | (a & c) | (b & c);
endmodule
```

Code Example 4.2 shows the SystemVerilog for a 64-bit 2:1 multiplexer module. Observe that some of the inputs are 64 bits wide. It uses the *ternary operator* `? :` to choose the output *y* to be either *d1* or *d0* based on the value of the select line, *s*. The comments show a logically equivalent but less readable description with Boolean equations, illustrating the `~` (not) operator and replicating *s* into a 64-bit bus. **Fig. 4.2** shows the circuit implied. It could be implemented in many ways, such as 64 1-bit 2:1 muxes, 128 2-input AND gates and 64 2-input OR gates, 192 2-input NAND gates, or 64 3-input FPGA lookup tables.

Code Example 4.2 mux2

```
module mux2(input  logic [63:0] d0, d1,
            input  logic      s,
            output logic [63:0] y);

    assign y = s ? d1 : d0;

    /* Alternative implementation using gates
    logic [63:0] s64;

    assign s64 = {64{s}}; // Replicate s 64 times
    assign y = ~s64 & d0 | s64 & d1; */
endmodule
```

Often it is helpful to create a *parameterized* module with adjustable bus width. **Code Example 4.3** shows a parameterized 2:1 mux with a default width of 8 bits. Parameterized modules can save time and code and enable code reuse; `mux2` can be used for any 2:1 mux, regardless of bus width.

Code Example 4.3 PARAMETERIZED mux2

```
module mux2 #(parameter WIDTH = 8) (
    input  logic [WIDTH-1:0] d0, d1,
    input  logic      s,
    output logic [WIDTH-1:0] y);

    assign y = s ? d1 : d0;
endmodule
```

Code Example 4.4 shows how to *instantiate* the default (8-bit bus) version of `mux2` and a 12-bit version. Each mux is instantiated by writing the *module name* (`mux2`) followed by the *instance name* (in this case, `muxa` or `muxb`) followed by the inputs and outputs. An instantiation is like grabbing parts (in this case, two muxes) from the hardware bin and then connecting their inputs and outputs as desired.

Code Example 4.4 INSTANTIATING PARAMETERIZED MODULES

```
mux2 muxa(d0, d1, s, y);           // 2:1 mux with default 8-bit busses
mux2 #(12) muxb(data0, data1, sel, yout); // 2:1 mux with 12-bit busses
```

Code Example 4.5 shows four SystemVerilog representations of a 4:1 multiplexer with a default width of 64 bits. The first is a structural representation, in which the 4:1 mux is built from three 2:1 muxes, as shown in Fig. 4.3. The width is passed as a parameter to each `mux2` to override the

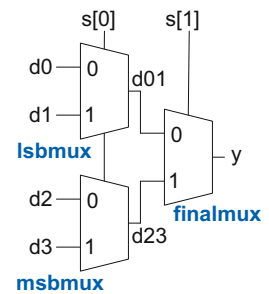


Fig. 4.3 `mux4` built from 2:1 multiplexers.

Code Example 4.5 `mux4`

```
module mux4 #(parameter WIDTH = 64) (
    input  logic [WIDTH-1:0] d0, d1, d2, d3,
    input  logic [1:0]      s,
    output logic [WIDTH-1:0] y);

    logic [WIDTH-1:0] d01, d23;

    // Structural implementation using 2:1 muxes
    mux2 #(WIDTH) lsbmux(d0, d1, s[0], d01);
    mux2 #(WIDTH) msbmux(d2, d3, s[0], d23);
    mux2 #(WIDTH) finalmux(d01, d23, s[1], y);

    /* Alternative behavioral implementation using assign statement
    assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0); */

    /* Alternative behavioral implementation using case statement
    always_comb
    case(s)
        0: y = d0;
        1: y = d1;
        2: y = d2;
        3: y = d3;
    endcase */

    /* Alternative behavioral implementation using if statements
    always_comb
    if      (s == 2'b00) y = d0;
    else if (s == 2'b01) y = d1;
    else if (s == 2'b10) y = d2;
    else      y = d3; */
endmodule
```

Wally Structural Coding Style

Wally favors a structural coding style that decomposes datapath modules into simple blocks, such as registers, multiplexers, memories, and Boolean equations, to emphasize the hardware implied by the code. These lower-level modules are instantiated within a higher-level or top-level module in the hierarchy. Experienced designers who readily visualize the hardware implied by Verilog statements often use more of a behavioral style for brevity. Beginning designers who don't want to think about the hardware implied by Verilog statements also often use a behavioral style and end up writing code that implies wasteful hardware or that won't synthesize.

Instantiating vs. Calling

In a typical programming language, a function *calls* another function. The second function puts information into a stack frame, performs its task, and then cleans up the stack frame and vanishes. In contrast, a hardware description language *instantiates* modules, that is, creates an instance of the module that permanently exists within the design, as opposed to function calls, which only exist while they are being called. The mux4 in [Code Example 4.5](#) can be built from three mux2 modules, which exist on a chip whether or not the inputs are changing.

default value. The internal signals d01 and d23 must be declared in the mux4 module as well. The alternate representations are commented out. The second 4:1 multiplexer representation uses an `assign` statement, the third uses an `always` block with a `case` statement, and the fourth uses an `always` block with `if/else` statements. These approaches are discussed in [Section 4.2.2](#). This example illustrates comments, which can proceed from `//` to the end of the line or from `/*` until the next `*/`. It also illustrates using `[]` to select bits from within a larger bus.

4.2.2 Combinational Logic

Recall that the output of *combinational logic* depends only on the current inputs. SystemVerilog describes combinational logic with `assign` statements or `always_comb` blocks.

4.2.2.1 Combinational Logic Using `assign` Statements

Boolean equations are described using an `assign` statement. [Table 4.1](#) lists the various Verilog operators and their order of precedence. The precedence follows the usual standard of AND before OR and `*` before `+`. Use parentheses to change this order or if you don't want your reader to have to remember the order of more obscure operations.

Multiple `assign` statements all operate in parallel, just like the hardware they imply. Hence, the order of `assign` statements in a module does not affect operation. This is a major difference between HDLs and conventional sequential programming languages.

Boolean operators `&`, `|`, and `|` can work in either bitwise or reduction form. *Bitwise* form takes two N -bit inputs and produces an N -bit output. For example, `y = a & b;` performs the bitwise AND of `a` and `b` and puts the result in `y`, where `a`, `b`, and `y` are all N -bit signals. If `a = 1010` and `b = 0111`, then `y = 0010`. The *reduction* form takes an N -bit input and produces a single-bit output; for example, `w = &a;` performs the AND of all bits in the N -bit signal `a` and puts the result in the 1-bit signal `w`. If `a = 1010`, `w = 0` because not all bits of `a` are TRUE. The equality comparator in [Code Example 4.6](#) checks if `a` is equal to `b` by bitwise XNORing each of the four columns of bits to see if each column is equal, then using a reduction AND to see if all four columns are equal, as shown in [Fig. 4.4](#).

Code Example 4.6 eqcmp

```
module eqcmp(input  logic [3:0] a, b,
             output logic      eq);

    assign eq = &(a ~^ b); // Equivalent to assign eq = (a == b)
endmodule
```

Table 4.1 SystemVerilog operators.

Precedence	Operator	Meaning
Highest	<code>~</code>	NOT
	<code>* / %</code>	Multiply, divide, modulo
	<code>+ -</code>	Add, subtract
	<code><< >></code>	Logical left/right shift
	<code><<< >>></code>	Arithmetic left/right shift
	<code>< <= > >=</code>	Relative comparison
	<code>== !=</code>	Equality comparison
	<code>& ~&</code>	AND, NAND
	<code>^ ~^</code>	XOR, XNOR
	<code> ~ </code>	OR, NOR
Lowest	<code>?:</code>	Conditional (ternary)

Verilog also has `&&`, `||` (logical AND, OR) operators that take 2 multibit inputs and return a 1-bit output. As in C, they interpret a multibit input as FALSE if zero and TRUE if nonzero. Hence, `011 && 111 = 1` because 011 and 111 are both nonzero. Some Verilog coding styles use logical operators for control logic and bitwise operators for datapaths. Our style uses bitwise operators exclusively and never attempts to interpret a multibit input as TRUE or FALSE.

Arithmetic operations can be performed with various power, performance, and area (PPA) trade-offs. The hardware for an N -bit add or subtract (+ or -) operation depends on the speed requirements but involves at least N full adders (see [Weste12] for an extensive discussion of alternatives). Relative comparison (`<`, `>`, and so on) is almost as expensive as subtraction. As seen earlier, equality comparison is cheaper, requiring only N XNOR gates and one N -input AND. Constant shifts are free (just wires), but variable shifts require many multiplexers and are fairly costly (see Section 7.1.3.3). Multiplication by a power of two is just a shift, but general multiplication is quite costly, involving an array of partial product generators and an array of adders to sum the partial products as discussed in Section 15.1.3. Division and modulus by powers of two are equivalent to shifts or truncation and are best expressed that way. Division and modulo by other amounts are expensive and are avoided in a digital system when feasible; see Section 15.1.4 for CMOS division circuits.

Cost is proportional to chip area.

4.2.2.2 Combinational Logic Using `always_comb` Statements

Combinational logic can also be described with `if` statements (convenient for next state logic in FSMs) and `case` statements (convenient for things resembling truth tables). These statements can only be placed inside an `always` block. In classic Verilog, `always` blocks require a sensitivity list that is notoriously prone to accidentally imply sequential logic instead of combinational logic or cause mismatches between simulation and

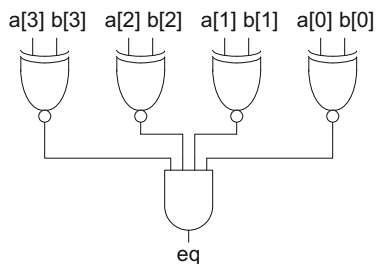


Fig. 4.4 Equality comparator.

synthesized logic, so SystemVerilog introduced the `always_comb` statement that will warn you if you make a mistake.

Code Example 4.5 showed a multiplexer optionally implemented with `case` and `if` statements. These are more verbose than `assign` statements or a structural design. Code Example 4.16 shows a better application of `if` statements to FSM next state logic.

A seven-segment display is shown in Fig. 4.5. For example, to display the digit 3, the display should light up segments a, b, c, d, and g, as shown. Code Example 4.7 shows a seven-segment-display decoder that receives a digit and determines which segments to light up. It is implemented with a `case` statement, which is far easier than grinding out seven Boolean equations by hand and coding `assign` statements for each output. Note that the `default` statement is necessary so that the outputs are well-defined even when an illegal input such as 1111 is applied; otherwise, the logic would remember the old output, which implies a latch (called *latch inference*) and is not combinational logic or synchronous sequential logic.

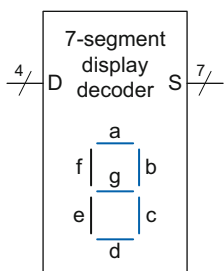


Fig. 4.5 Seven-segment display.

Code Example 4.7 SEVEN-SEGMENT DISPLAY DECODER

```
module sevenseg(input  logic [3:0] digit,
                output logic [6:0] segments);

    always_comb
    case(digit)
        // abc_defg
        0:    segments = 7'b111_1110;
        1:    segments = 7'b011_0000;
        2:    segments = 7'b110_1101;
        3:    segments = 7'b111_1001;
        4:    segments = 7'b011_0011;
        5:    segments = 7'b101_1011;
        6:    segments = 7'b101_1111;
        7:    segments = 7'b111_0000;
        8:    segments = 7'b111_1111;
        9:    segments = 7'b111_0011;
        default: segments = 7'b000_0000;
    endcase
endmodule
```

Placing underscores (_) in constants helps readability; they are ignored by simulation and synthesis tools.

Code Example 4.8 implements a priority circuit using a `casez` statement, which is like `case` but allows don't cares (?) in the inputs. The priority circuit grants priority (outputs 1) to the most-significant input that requests priority (inputs 1). The comments show a second equivalent implementation with `if/else` statements. Both styles are comparably easy to read and write and imply the same hardware. Fig. 4.6 shows the priority circuit schematic.

Code Example 4.8 PRIORITY CIRCUIT

```
module priorityckt(input  logic [3:0] a,
                  output logic [3:0] y);

  // casez implementation
  always_comb
  casez(a)
    4'b1???: y = 4'b1000;
    4'b01???: y = 4'b0100;
    4'b001?: y = 4'b0010;
    4'b0001: y = 4'b0001;
    default: y = 4'b0000;
  endcase

  /* if implementation:
  always_comb
  if      (a[3]) y = 4'b1000;
  else if (a[2]) y = 4'b0100;
  else if (a[1]) y = 4'b0010;
  else if (a[0]) y = 4'b0001;
  else          y = 4'b0000; */
endmodule
```

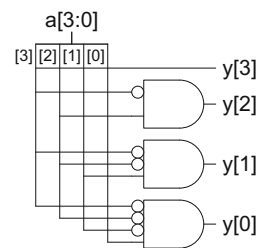


Fig. 4.6 Priority circuit.

4.2.2.3 Xs and Zs, Oh My

SystemVerilog `logic` signals have four states: 0, 1, x, and z. x indicates unknown, and z indicates *floating* (high impedance).

Unknown (x) values have several possible causes. A logic signal with no driver or multiple conflicting drivers has an unknown value. The output of a sequential element such as a flip-flop or memory cell is unknown until a value is written. If a gate receives an x or z input, it will produce an x output when it can't determine the correct output value. For example, $1'b1 \& 1'bx = x$. But $1'b0 \& 1'bx = 0$ because anything AND 0 is 0.

The presence of x in simulation is generally an indication of a logic problem in a design and should be tracked down and fixed. An exception is that memories are generally initialized to x. However, this is a problem when uninitialized memory is read, and this x percolates into logic that affects the state of the system. For example, if Wally reads an uninitialized

The Questa simulator, which is discussed further in Chapter 5, displays x as a red line and z as a blue line.

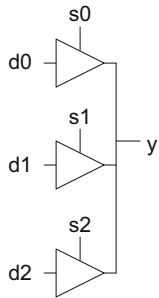


Fig. 4.7 Tristate distributed multiplexer.

Note that *y* is declared *tri* rather than *logic*. *logic* signals can only have a single driver. Tristate busses have multiple drivers, so they should be declared as a *tri* net. Typically, exactly one driver on a net is active at a time, and the net takes on that value. Otherwise, the net floats (*z*). Also note that a *tri* output from a module can be used as a *logic* input to another module.

Note that '*z*' makes all the bits of *y* float, as discussed in Section 4.2.5.2.

memory address, the *x* values can propagate through the hardware page table walker and hang the simulation.

SystemVerilog has another data type called *bit* that only represents 0 and 1. When *logic* is converted to *bit*, *x* and *z* map to 0. As shown in Section 4.2.3.3, Wally uses *bit* instead of *logic* in random access memory (RAM) to prevent uninitialized RAM from corrupting the system and to cut simulation memory usage (because *bit* takes half as much space as *logic*).

Some designers have tried to cleverly use *x* as a mechanism to indicate problems, such as an illegal instruction. Unfortunately, simulations may be optimistic or pessimistic in a way that hides real problems or indicates phantom problems, so it's best not to rely on *x* to detect design errors [Sutherland13].

A *tristate bus* has multiple drivers controlled such that at most one is active. If the bus has one active driver, it takes the value from that driver. If no driver is active, the bus floats. Tristate drivers are modeled with *z*. For example, Code Example 4.9 describes a distributed multiplexer. It defines a tristate buffer and then allows a bus *y* to be driven by *d0*, *d1*, or *d2* based on *s0*, *s1*, or *s2*, as shown in Fig. 4.7. Each tristate output is floating (*z*) when not enabled. If exactly one of the tristates is enabled, it will overpower the other floating tristates and drive its value onto the bus. If multiple tristates attempt to drive the bus to different values, the bus will get *x* to indicate contention. If none of the tristates are enabled, the bus will be floating.

Code Example 4.9 TRISTATE DISTRIBUTED MULTIPLEXER

```
module tristate #(parameter WIDTH=32) (
    input  logic [WIDTH-1:0] a,
    input  logic             en,
    output tri  [WIDTH-1:0] y);

    assign y = en ? a : 'z;
endmodule

module distributedmux(
    input  logic [3:0] d0, d1, d2,
    input  logic      s0, s1, s2,
    output tri  [3:0] y);

    tristate #(4) t0(d0, s0, y);
    tristate #(4) t1(d1, s1, y);
    tristate #(4) t2(d2, s2, y);
endmodule
```

Fig. 4.8 shows the waveforms from the distributed mux. *d0* is held at 0000. *d1* is held at 0101. *d2* is unknown. Following the waveform from left to right: when none of the *s* signals are asserted, the output *y* floats;

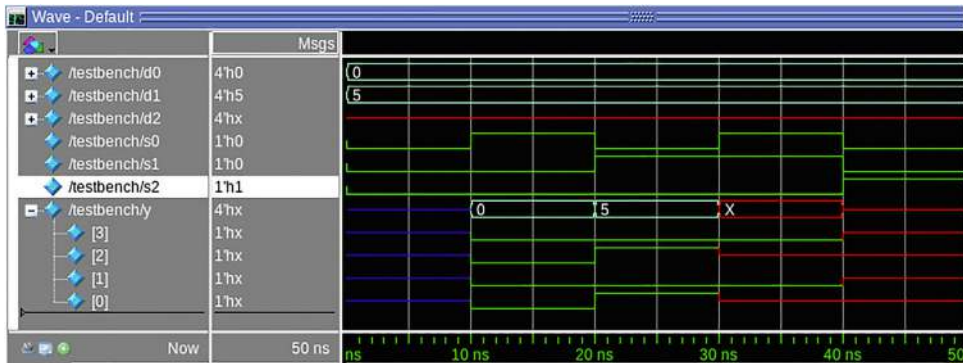


Fig. 4.8 Distributed multiplexer waveforms.

when s_0 is asserted, y is 0000; when s_1 is asserted, y is 0101; when s_0 and s_1 are both asserted, y is 0x0x because bits 0 and 2 are simultaneously driven high and low by tristates in contention; when s_2 is asserted, y is unknown because d_2 is unknown.

This coding style is not recommended within a chip. Tristate busses are slow and can have contention if one driver turns ON before another turns OFF. Many cell libraries don't even have tristate cells. Instead, tristate drivers are more appropriate for I/O pins that must access an off-chip shared bus or for shared bidirectional pins. [Code Example 4.10](#) shows how to use tristate drivers for bidirectional general-purpose I/O, and [Fig. 4.9](#) shows the synthesized circuit. Observe that the bidirectional pin is declared `inout` and must be a `tri` net because it could have multiple drivers.

Code Example 4.10 BIDIRECTIONAL I/O

```
module gpio #(parameter WIDTH=16) (
  input logic [WIDTH-1:0] GPIOOutVal, GPIOEn,
  output logic [WIDTH-1:0] GPIOInVal,
  inout tri [WIDTH-1:0] GPIOPin);

  assign GPIOInVal = GPIOPin;
  tristate #(1) ts[WIDTH-1:0](GPIOOutVal, GPIOEn, GPIOPin);
endmodule
```

The tristates are defined as an array of WIDTH 1-bit modules because each bit needs an independent enable. Arrays of modules are described further in [Section 4.2.5.5](#).

4.2.3 Sequential Logic

The output of *sequential logic* depends on previous as well as current inputs. Common forms of sequential logic include flip-flops, finite state machines, counters, shift registers, and memories. We are primarily interested

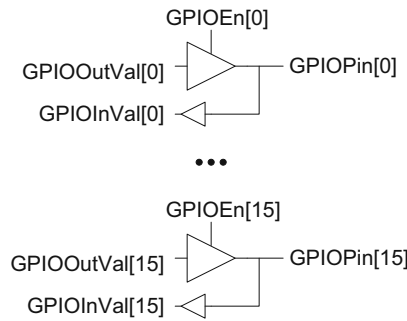


Fig. 4.9 Bidirectional I/O.

in *synchronous sequential logic* in which all outputs change on the edge of a common clock. SystemVerilog uses an `always_ff` block to describe flip-flops.

4.2.3.1 Flip-Flops

The SystemVerilog idiom for an ordinary *flip-flop* (also called a *D flip-flop*, *flop*, or *positive edge-triggered flip-flop*) is given in [Code Example 4.11](#). On the rising edge of the clock, a flip-flop copies the input to the output. The `always_ff` construct will perform the operation in the body when the condition inside the sensitivity list (in this case, `posedge clk`) is true. So `q` gets the value of `d` at the rising edge of the clock. The `<=` operator is called a *nonblocking assignment* and is often pronounced “gets.” All nonblocking assignments happen in parallel. A group of one or more flip-flops sharing a common clock and a bus of inputs and outputs is called a *register* as shown in [Fig. 4.10](#). This flop module is parameterized so that the number of flip-flops is configurable.

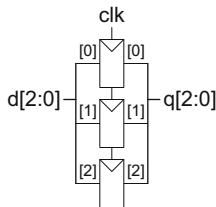


Fig. 4.10 3-bit register.

Code Example 4.11 flop

```
module flop #(parameter WIDTH = 3) (
  input  logic      clk,
  input  logic [WIDTH-1:0] d,
  output logic [WIDTH-1:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

Combinational logic in an `always_comb` block is best described with blocking assignments (`=`) when one line depends on the previous. Non-blocking assignments also produce the correct result, but the simulation

will be slower because the `always` block may need to be evaluated multiple times.

Sequential logic is best described with nonblocking assignments (`<=`) because `<=` captures the way real hardware operates in parallel rather than consecutively. For example, in the following code, on the rising edge of the clock, `a` is copied to `b`, and then the new value of `b` is copied to `c`, so both `b` and `c` receive the value of `a`, and the circuit implies only one flip-flop. If each `=` is replaced with `<=`, `c` will get the old value of `b` concurrently with `b` getting the old value of `a`, so the code implies two flip-flops. Fig. 4.11 shows the hardware implied by the blocking and non-blocking code.

(a) Bad synchronizer using blocking assignments

```
always_ff @(posedge clk) begin
    b = a; // Blocking
    c = b; // Blocking
end
```

(b) Good synchronizer using nonblocking assignments

```
always_ff @(posedge clk) begin
    b <= a; // Non-blocking
    c <= b; // Non-blocking
end
```

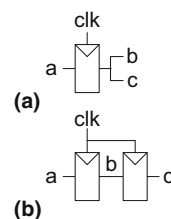


Fig. 4.11 (a) Blocking: one flip-flop and (b) Nonblocking: two flip-flops.

4.2.3.1.1 Resettable Flops

At startup, the output of a flip-flop is unpredictable. SystemVerilog represents this unknown state with an `x`. Most systems require that some or all flip-flops receive a reset at startup to place them in a known state. Code Example 4.12 shows the idioms for asynchronously and synchronously resettable flip-flops. Whereas *asynchronously resettable flip-flops* are reset as soon as the reset signal rises, *synchronously resettable flip-flops* only respond when the clock rises. Synchronous reset is also known as *clear*.

Synchronous reset can be tricky if a system has gated clocks that might not all fire during reset. This could be solved by a more complex clock gater that always fires when reset is applied. Asynchronous reset avoids this issue and works even if some clocks come from external sources that aren't yet running. However, asynchronous reset requires substantially more transistors [Weste12], along with a complex distribution tree to get the reset to all flops at nearly the same time. Most FPGAs have asynchronous reset inputs built in to all flip-flops, so asynchronous

Asynchronous resets must be glitch-free to prevent a spurious reset.

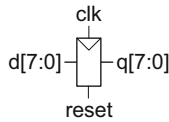


Fig. 4.12 Resettable flip-flop.

Reset can be eliminated for registers that always get a value before they are used. For example, the register file does not need to be reset because a program should not depend on the values of registers before they are initialized. In simulation, registers without a reset start with an unknown value (x) until given a value. Occasionally this x propagates through other logic, leading to subtle simulation problems. x also can be a clue of a bug in simulation, and nonreset registers can obscure this clue. Wally resets all flip-flops to avoid these difficulties.

reset is preferred on FPGAs. ASIC designers reduce cost by using synchronous reset or eliminating reset entirely on some flip-flops. Synchronous clear is also used to flush a stage in a pipelined processor by deasserting the write control signals. Fig. 4.12 shows a flip-flop with a reset terminal.

Code Example 4.12 ASYNCHRONOUSLY AND SYNCHRONOUSLY RESETTABLE FLIP-FLOPS

```
module flopr #(parameter WIDTH = 8) (
    input  logic      clk, reset,
    input  logic [WIDTH-1:0] d,
    output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset) // Asynchronous reset
        if (reset) q <= 0;
        else      q <= d;
endmodule

module flopc #(parameter WIDTH = 8) (
    input  logic      clk, reset,
    input  logic [WIDTH-1:0] d,
    output logic [WIDTH-1:0] q);

    always_ff @(posedge clk) // Synchronous reset
        if (reset) q <= 0;
        else      q <= d;
endmodule
```

4.2.3.1.2 Enabled Flops

Enabled flip-flops only copy the input to the output on the rising edge of the clock when the enable is true. The enable can be used to stall a pipeline stage in a processor or to save power. Code Example 4.13 shows the idiom for an enabled flip-flop. Code Example 4.14 puts everything together for a flip-flop with enable, reset, and clear. Fig. 4.13 shows flip-flops with these terminals.

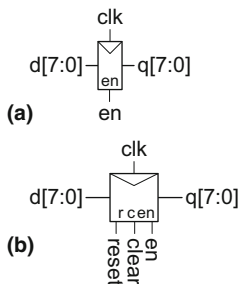


Fig. 4.13 (a) Enabled flip-flop and
(b) Enabled flip-flop with reset
and clear.

Code Example 4.13 ENABLED FLIP-FLOP

```
module flopen #(parameter WIDTH = 8) (
    input  logic      clk, en,
    input  logic [WIDTH-1:0] d,
    output logic [WIDTH-1:0] q);

    always_ff @(posedge clk)
        if (en) q <= d;
endmodule
```

Code Example 4.14 FLIP-FLOP WITH ENABLE, ASYNCHRONOUS RESET, SYNCHRONOUS CLEAR

```

module flopenrc #(parameter WIDTH = 8) (
  input  logic      clk, reset, clear, en,
  input  logic [WIDTH-1:0] d,
  output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)  q <= 0;
    else if (en)
      if (clear) q <= 0;
      else      q <= d;
endmodule

```

4.2.3.1.3 Latches

A *transparent* or *D latch* passes D through to Q as long as clock is asserted and retains Q otherwise. Code Example 4.15 shows the SystemVerilog idiom for a latch using the `always_latch` statement. Transparent latches are not edge triggered and thus generally should be avoided in a synchronous sequential design methodology, but they are needed in special-purpose circuits such as clock gaters. Fig. 4.14 shows the schematic symbol for a latch, which omits the triangle representing the edge-triggered clock.

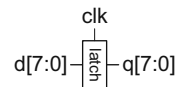
Code Example 4.15 TRANSPARENT LATCH

```

module latch #(parameter WIDTH = 8) (
  input  logic      clk,
  input  logic [WIDTH-1:0] d,
  output logic [WIDTH-1:0] q);

  always_latch
    if (clk) q <= d;
endmodule

```

**Fig. 4.14** Transparent latch.**4.2.3.2 Finite State Machines**

Finite state machines (FSMs) are often used as controllers. They keep track of the state of a system or subsystem and calculate the next state depending on the current state and the inputs. FSMs are categorized as Moore or Mealy machines. In Moore machines, the outputs depend only on the current state. In Mealy machines, the outputs depend on the current state and inputs.

Fig. 4.15 shows generic block diagrams for both a Moore and Mealy FSM, each with M bits of input, N bits of output, and k bits of state. Both FSMs have a k -bit register to hold the current state, and the combinational next state logic computes the k bits of next state (inputs to the register)

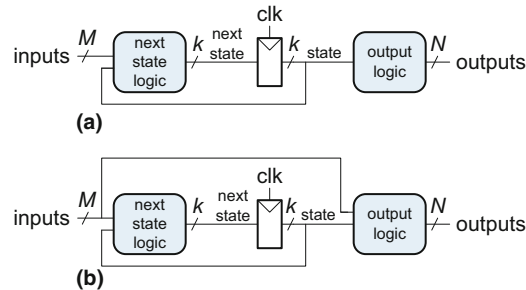


Fig. 4.15 FSM block diagrams: (a) Moore and (b) Mealy.

based on the inputs and current state. The output logic is combinational logic that computes the outputs based on the current state (for a Moore FSM) or the current state and inputs (for a Mealy FSM). SystemVerilog describes the FSM using the same format, with a register and two blocks of combinational logic. The next state logic is often described using a `case` statement that depends on the current state. Verilog's `enum` keyword allows us to give names instead of numbers to the states, making the FSM easier to read during design and simulation.

For example, Fig. 4.16 shows a conceptual Moore state transition diagram for a two-bit branch predictor FSM. It has an output `p` (predict taken), which is 1 when the branch should be predicted as taken, and an input `t` (taken), which indicates when the branch actually was taken. Reset and clock are not considered inputs for the next state logic. The two bits of state give four states: Strongly Not Taken (SNT), Weakly Not Taken (WNT), Weakly Taken (WT), and Strongly Taken (ST). The FSM resets into the Weakly Taken state and increases or decreases its prediction based on whether the branch was actually taken. Code Example 4.16 shows the SystemVerilog code for the FSM. The four states are enumerated. Because the enumeration encoding is not stated explicitly, it is in natural order, with SNT = 00 and ST being 11. The resettable state register is described with an `always_ff @(posedge clk)` statement. The next state logic is described with a `case` statement depending on the current state, containing `if` statements depending on the input(s). The output logic depends only on the current state and is conveniently described with an `assign` statement.

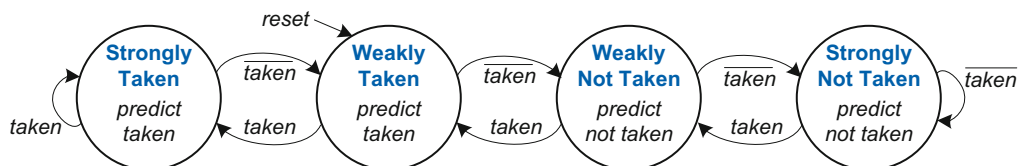


Fig. 4.16 Two-bit branch predictor state transition diagram.

Fig. 4.17 shows a schematic of the FSM. Logic synthesis automatically derives the Boolean expressions and maps them onto appropriate gates, saving the engineer from that time-consuming and error-prone task.

Code Example 4.16 TWO-BIT BRANCH PREDICTOR FSM

```
module twobitbranchpredictor(input  logic clk, reset,
                           input  logic t,
                           output logic p);

    typedef enum logic [1:0] {SNT, WNT, WT, ST} statetype;

    statetype state, nextstate;

    // State register
    always_ff @(posedge clk, posedge reset)
        if (reset) state <= WT;
        else      state <= nextstate;

    // Next state logic
    always_comb
        case (state)
            SNT: if (t) nextstate = WNT;
                 else  nextstate = SNT;
            WNT: if (t) nextstate = WT;
                 else  nextstate = SNT;
            WT:  if (t) nextstate = ST;
                 else  nextstate = WNT;
            ST:  if (t) nextstate = ST;
                 else  nextstate = WT;
            default: nextstate = WT;
        endcase

    // Output logic
    assign p = (state == WT) | (state == ST);
endmodule
```

In Code Example 4.16, `typedef enum` defines a new *enumerated type* called `statetype`, which has an underlying type of `logic [1:0]`. This new type enables the user to use enumerated values, in this case SNT, WNT, WT, and ST, instead of binary encodings for the states (i.e., 00, 01, 10, and 11). This increases readability and ease of coding.

Code Example 4.17 shows a more complex FSM illustrating how to merge the state register with the next state logic, how to incorporate a counter into the FSM, how to use `begin/end` to take multiple actions, and how Verilog keeps the old value of signals when they aren't changed

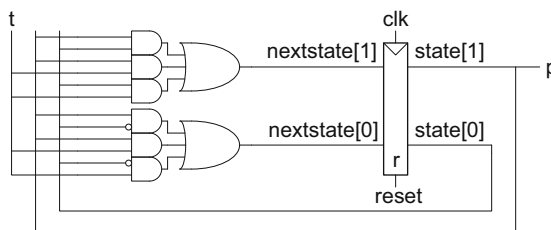


Fig. 4.17 Two-bit branch predictor logic.

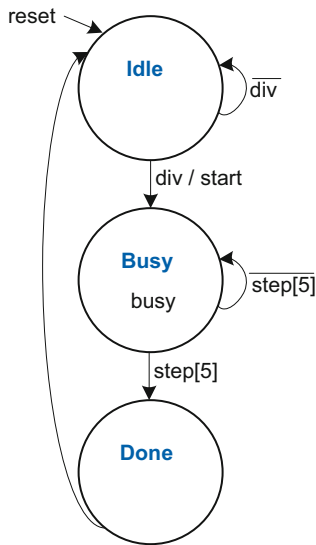


Fig. 4.18 Divider FSM.

A group of statements within `begin/end` delimiters is known as a compound statement. Using `begin/end` is analogous to using curly brackets `{ }` in C and Java.

In Fig. 4.18, checking `step[5]` in the Busy state is the same as checking `(step == 6'd32)`, but it may produce less logic because of its 1-bit comparison instead of the latter's 6-bit comparison.

in sequential logic. As shown in Fig. 4.18, the FSM produces `start` and `busy` signals for a divider that takes 32 clock cycles. When the divider is Idle and the `div` input is asserted, the FSM asserts `start` for one cycle and enters the Busy state. The `busy` signal is held high for 32 cycles, and then the FSM enters the Done state for one cycle before returning to Idle. The next state logic is implicit inside the `always_ff`. If the state is not changed, it keeps its old value. This is only legal in an `always_ff` because keeping an old value implies sequential logic rather than combinational logic. In some conditions, both the `state` and `step` signals are updated. Verilog provides the `begin/end` construct to contain multiple statements within the body of a block. The divider controller could be viewed as two interacting FSMs, one to track the state and the other to count steps. However, the two are merged here for brevity. Fig. 4.19 shows the hardware implied by this SystemVerilog. Avoid coding FSMs like this until you are experienced enough to be able to visualize the hardware you are implying.

Code Example 4.17 DIVIDER CONTROL

```

module dividercont(input  logic clk, reset,
                  input  logic div,
                  output logic start, busy);

    typedef enum logic [1:0] {Idle, Busy, Done} statetype;
    statetype state;
    logic [5:0] step;

    always_ff @(posedge clk)
        if (reset) state <= Idle;
        else case (state)
            Idle: if (div) begin
                    state <= Busy;
                    step  <= 1;
                end
            Busy: begin
                    if (step[5]) state <= Done;
                    step <= step + 1;
                end
            Done: state <= Idle;
        endcase

    assign start = div & (state == Idle);
    assign busy  = (state == Busy);
endmodule
  
```

4.2.3.3 Memories

Memories are organized as arrays of 2^M -words $\times$ N -bits. The memory receives an address indicating which word to read or write. Digital systems

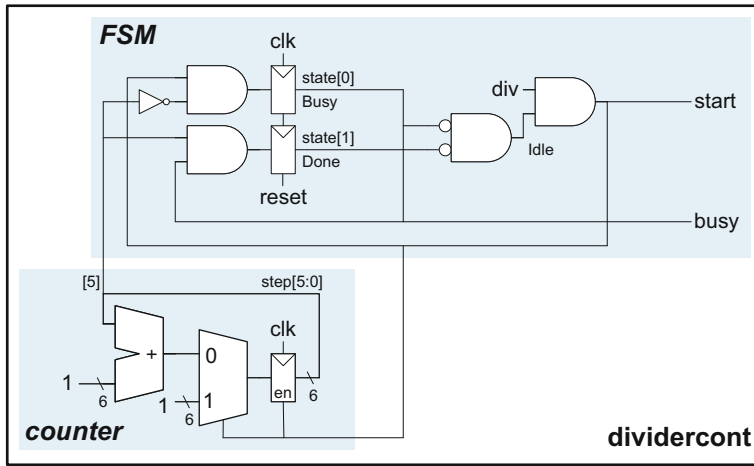


Fig. 4.19 Divider control schematic.

have many types of memories, and the designer must select an appropriate memory with the desired number of read and write ports, capacity ($2^M \times N$), and timing. Designing a custom memory is tricky, takes time, and introduces risk of error, so the designer is often constrained to pick memories from a limited library of options.

A *read port* consists of an address input *a* and a read data output *rd*. Memory reads can be either *asynchronous* or *synchronous*, as shown in Fig. 4.20. With asynchronous (also called *combinational*) read, the address stabilizes at some point, and the read data becomes valid after a propagation delay. With synchronous (also called *edge-triggered*) read, the address stabilizes at least some setup time before the rising edge of the clock. The read data arrives some clock-to-Q propagation delay after the clock edge. Memory writes are always synchronous because they need a timing edge indicating when to write (sometime after the address and write data are stable). We use synchronous memories in this textbook (starting in Section 7.2.1) because, for all but very small memories, they are typically smaller and faster than memories with asynchronous reads.

A *write port* consists of an address input *a* specifying where to write, a write data input *wd* specifying what to write, a write enable input *we* specifying whether to write, and a clock *clk* specifying when to write. Thus, write ports are inherently synchronous.

A *read/write port* has the same connections as a write port but also has a read data output. It writes when *we* is asserted and always reads.

Two types of memory exist, read-only memory (ROM) and RAM, which are discussed next. This book names memories based on the number of ports and how many can be used for read, write, or read/write.

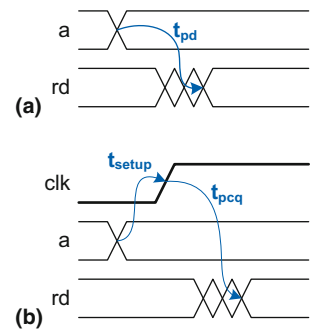


Fig. 4.20 Memory read timing:
(a) asynchronous and
(b) synchronous.

e and b suffixes are used for chip and byte enables, respectively, that enable access to the memory (chip) overall and enable writing specific bytes within a word. For example, a basic single-ported RAM is called `ram1p1rw` because it has one port (1p), which is read/write (1rw). A dual-ported RAM with byte enables and one read and one write port is called `ram2p1r1wb`.

4.2.3.3.1 ROMs

A *read-only memory* (ROM) retains its contents after being powered off and back on. It typically has a single read port. The contents may be programmed into the ROM at design time (e.g., by the presence or absence of transistors in an ASIC or by the configuration of an FPGA memory). Other ROMs are built from flash memory and can be reprogrammed, albeit in a slow and cumbersome way. Such flash ROMs are often used to hold upgradable firmware.

Code Example 4.18 shows the idiom for a 2^M -word $\times$ N -bit ROM with one asynchronous read port, and Fig. 4.21 shows the symbol for the ROM. This module has two parameters: M and N . `mem` is declared as a 2-dimensional array of 2^M entries, each of size N . As discussed in Section 4.2.2.3, `mem` is declared as `bit` rather than `logic`, so uninitialized entries default to 0 rather than `x`. This reduces simulator memory requirements and avoids subtle bugs with uninitialized memory values corrupting a simulation. The `initial $readmemh` statement populates the memory array from a file. For example, Code Example 4.19 shows the `romfile.dat` for an 8-word $\times$ 16-bit ROM. It has only 6 lines, so the last two entries of the ROM default to 0000. Each line has 16 bits specified with four hexadecimal nibbles. For example, entry 2 is 0xA5A5 (binary 1010_0101_1010_0101). If the system lacks a ROM generator, logic synthesis might map this module onto a network of combinational logic producing the specified outputs.

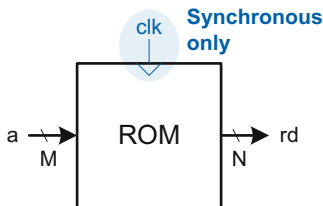


Fig. 4.21 ROM (`rom1p1r`) with 1 read port.

Note the exponentiation operator `**` in the ROM example: $2^{**}M = 2^M$ (=8 if $M = 3$). SystemVerilog also has a handy `$clog2` (ceiling of $\log_2$) operator that is handy for calculating the number of bits needed to address a memory. For example, a memory containing 32 words needs $\$clog2(32) = 5$ address bits.

Code Example 4.18 ASYNCHRONOUS $2^M \times N$ -BIT ROM

```
module rom1p1r #(parameter M=3, N=16) (
    input  logic [M-1:0] a,
    output logic [N-1:0] rd);

    bit [N-1:0] mem[2**M];

    initial $readmemh("romfile.dat", mem);

    assign rd = mem[a]; // Asynchronous read
endmodule
```

Code Example 4.19 SAMPLE romfile.dat

```
0000
FFFF
A5A5
0001
DEAD
BEEF
```

FPGAs often build larger ROMs from synchronous block RAMs and thus expect synchronous read. Synchronous read is similar to the combinational read shown in [Code Example 4.18](#) but would be coded as follows:

```
always_ff @(posedge clk) rd <= mem[a];
```

4.2.3.3.2 RAMs

A *random access memory* (RAM) is a historic and confusing term for a memory that can be read and written, generally with a delay that is short and independent of the address being accessed. Midsized memories are generally implemented with *static RAM* (SRAM), which retains its value as long as power is applied. SRAM is built from 6-transistor memory cells shown in [Fig. 4.22](#). To write the cell, the bitline and its complement are driven to opposite values, and the wordline is pulsed to overpower the cross-coupled inverters. To read the cell, bitline and its complement are precharged high, and then wordline is pulsed to pull down one of the bitlines. The cell requires periphery circuits to precharge and sense bitlines, as well as a decoder. Thus, SRAM arrays typically come as a library component with a specified capacity, shape, and timing characteristics. They use a clock to sequence the precharge and sensing, so they read synchronously. Very small memories can be implemented with flip-flops or latches when the overhead of the periphery circuits is excessive or no suitable library component is available. Such memories generally read combinatorially through a multiplexer tree.

[Code Example 4.20](#) shows the SystemVerilog idiom for a 2^M -word $\times$ N -bit single-ported asynchronous RAM with one port capable of reading

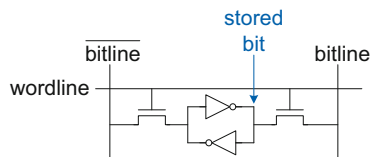


Fig. 4.22 6T SRAM bit cell.

and writing (`ram1p1rw_async`). The RAM reads asynchronously onto the read data port (`rd`) some propagation delay after the address (`a`) stabilizes, as shown in Fig. 4.20(a). If the write enable (`we`) is asserted, it also synchronously writes the write data (`wd`) to the same address (`a`) on the rising edge of the clock.

Code Example 4.20 ASYNCHRONOUS RAM

```
module ram1p1rw_async #(parameter M = 6, N = 32) (
    input  logic      clk,
    input  logic      we,
    input  logic [M-1:0] a,
    input  logic [N-1:0] wd,
    output logic [N-1:0] rd);

    bit [N-1:0] mem[2**M];

    assign rd = mem[a];
    always_ff @(posedge clk)
        if (we) mem[a] <= wd;
endmodule
```

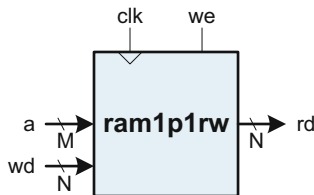


Fig. 4.23 RAM with 1 read/write port.

Synchronous read is again coded:

```
always_ff @(posedge clk) rd <= mem[a];
```

Fig. 4.23 shows a symbol for a RAM. The same symbol is used for both asynchronously and synchronously readable RAM so the designer must pay close attention to which form is used.

4.2.3.3.3 Byte Write Enables

Systems often need to write only some of the bytes within a memory word. For example, a cache memory array might have 128 words of 256 bits each. A write to the cache might need to update 32, 16, or 8 bits within the word. A brute force solution is to read the entire word, modify certain bytes using multiplexers, and write the word back. This may take an extra clock cycle. If a memory with a per-byte write enable is available, a more efficient solution is to only write the necessary bytes. Code Example 4.21 shows how to describe a RAM with one write enable signal per byte rather than a single one for the entire word. The design assumes N is a multiple of 8. For example, if $N = 32$, the memory has $32/8 = 4$ byte enables `we[3:0]` controlling which bytes

(bits [31:24], [23:16], [15:8], or [7:0]) are written of the 32-bit word at address `a`.

Code Example 4.21 RAM WITH BYTE ENABLES

```
module ram1plrwb #(parameter M = 6, N = 32) (
    input  logic      clk,
    input  logic [N/8-1:0] we,
    input  logic [M-1:0]  a,
    input  logic [N-1:0]  wd,
    output logic [N-1:0]  rd);

    bit [N-1:0] mem[2**M];
    genvar i;

    always_ff @(posedge clk)
        rd <= mem[a];

    for (i = 0; i < N/8; i++) // Byte writes
        always_ff @(posedge clk)
            if (we[i]) mem[a][i*8 +: 8] <= wd[i*8 +: 8];
endmodule
```

Code Example 4.21 introduces two new pieces of Verilog syntax. The `for` loop is called a *generate loop* and implies multiple pieces of hardware as discussed in Section 4.2.5.4. In this example, it creates the byte-enabled columns of the RAM. The SystemVerilog `+:` operator is called an *indexed part-select*. It selects a specified number of bits starting at a specified offset. For example, `wd[i*8 +: 8]` selects 8 bits of `wd` starting at column `i*8`. When `i = 2`, it selects `wd[23:16]`. It is equivalent to `wd[i*8 + 7 : i*8]`, but SystemVerilog synthesis prefers the indexed part-select notation.

4.2.3.3.4 Chip Enable

RAMs often provide a chip enable (`ce`) to save energy when the RAM is inactive. When enabled, the RAM behaves normally. When disabled, the RAM ignores all other inputs and retains the last read value `rd`. Code Example 4.22 shows SystemVerilog code for a RAM with a single read/write port with a chip enable and byte enables for writes.

Code Example 4.22 SYNCHRONOUS RAM WITH BYTE AND CHIP ENABLES

```

module ram1p1rwbe #(parameter M = 6, N = 32) (
    input  logic          clk,
    input  logic          ce,
    input  logic [N/8-1:0] we,
    input  logic [M-1:0]  a,
    input  logic [N-1:0]  wd,
    output logic [N-1:0]  rd);

    bit [N-1:0] mem[2**M];
    genvar i;

    always_ff @(posedge clk)
        if (ce) rd <= mem[a];

    for(i = 0; i < N/8; i++) // Byte writes
        always_ff @(posedge clk)
            if (ce & we[i]) mem[a][i*8 +: 8] <= wd[i*8 +: 8];
endmodule

```

4.2.3.3.5 Multiported RAMs

A multiported RAM has more than one read or write port. [Code Example 4.23](#) shows the SystemVerilog idiom for a synchronous two-port RAM with one read and one write port (`ram2p1r1w`). The port number is appended to each signal associated with it. So, port 1 is the read port (`a1`, `rd1`), and port 2 is the write port (`a2`, `wd2`, `we2`). [Code Example 4.24](#) shows the idiom for a synchronous two-port RAM, with each port capable of reading and writing (`ram2p2rw`). [Code Example 4.25](#) shows the idiom for a three-ported *register file*, which is a small memory with two asynchronous read ports and a synchronous write port (`rf3p2r1w`). This example uses `logic` rather than `bit`, so a simulation can detect bad programs that use uninitialized registers. [Fig. 4.24](#) shows symbol representations of each.

Code Example 4.23 DUAL-PORT RAM WITH ONE READ AND ONE WRITE PORT

```

module ram2p1r1w #(parameter M = 6, N = 32) (
    input  logic          clk,
    input  logic          we2,
    input  logic [M-1:0] a1, a2,
    input  logic [N-1:0] wd2,
    output logic [N-1:0] rd1);

    bit [N-1:0] mem[2**M];

    always_ff @(posedge clk) begin
        rd1 <= mem[a1];
        if (we2) mem[a2] <= wd2;
    end
endmodule

```

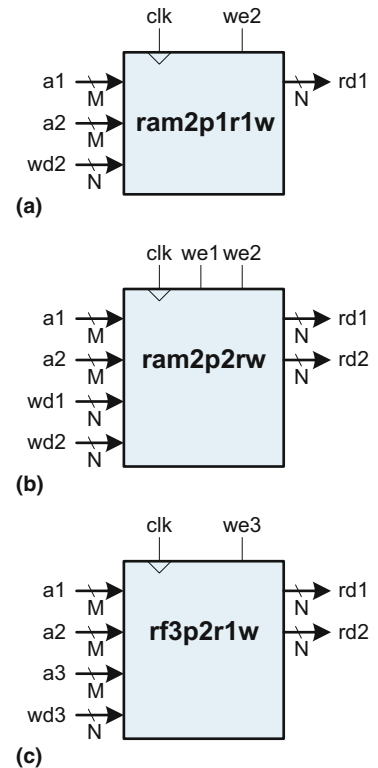


Fig. 4.24 (a) 2-port 1r1w RAM, (b) 2-port 2rw RAM, and (c) 3-port 2r1w register file.

Code Example 4.24 DUAL-PORT RAM WITH TWO READ/WRITE PORTS

```

module ram2p2rw #(parameter M = 6, N = 32) (
    input  logic      clk,
    input  logic      we1, we2,
    input  logic [M-1:0] a1, a2,
    input  logic [N-1:0] wd1, wd2,
    output logic [N-1:0] rd1, rd2);

    bit [N-1:0] mem[2**M];

    always_ff @(posedge clk) begin
        rd1 <= mem[a1];
        rd2 <= mem[a2];
        if (we1) mem[a1] <= wd1;
        if (we2) mem[a2] <= wd2;
    end
endmodule

```

Code Example 4.25 TRIPLE-PORTED REGISTER FILE WITH TWO READ AND ONE WRITE PORTS

```

module rf3p2rlw #(parameter M = 5, N = 32) (
    input  logic      clk,
    input  logic      we3,
    input  logic [M-1:0] a1, a2, a3,
    input  logic [N-1:0] wd3,
    output logic [N-1:0] rd1, rd2);

    logic [N-1:0] mem[2**M]; // Logic (not bit) to model uninitialized registers

    // Asynchronous read on ports 1, 2
    assign rd1 = mem[a1];
    assign rd2 = mem[a2];

    // Synchronous write on port 3
    always_ff @(posedge clk)
        if (we3) mem[a3] <= wd3;
endmodule

```

4.2.3.3.6 Library Considerations

ASIC vendor memory libraries vary widely in richness and configurability. Whereas a basic memory library has only single-port RAMs, a better library may have RAMs with one read and one write port. For high speed, RAM arrays generally can't be larger than about 128 rows $\times$ 256 bits. However, larger RAMs can be constructed from multiple subarrays.

In an ASIC, the prior RAM idioms imply flip-flop-based memories. To target SRAM, an ASIC requires an explicit instantiation of a particular SRAM (see [Section 6.5.3](#)). When Wally is targeted to an FPGA (see [Chapter 23](#)), the RAM idioms are mapped to *block RAM* when possible.

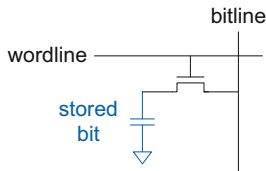


Fig. 4.25 1T DRAM cell.

Smaller memories are less area efficient because the decoder and column circuitry overhead is a greater proportion of the total area.

Custom designers can create a wide range of memories including many-ported register files and *content-addressable memories* (CAMs), but ASIC designers often have to synthesize these structures from flip-flops. Note that SRAM is about 10 times denser per bit than flop-based memory, and dynamic RAM (DRAM) is about 10 times denser than SRAM.

Very large memories generally use DRAM because of its density advantages. DRAM is built from a cell with one transistor and one capacitor as shown in [Fig. 4.25](#). It requires complex periphery circuits and must be periodically read and refreshed before the charge on the capacitor leaks away. Commodity DRAM chips are more than an order of magnitude cheaper per bit than SRAM, but DRAM has longer latency, has much less bandwidth, and requires a controller to manage both refreshes and the high-speed off-chip interface. A double data rate (DDR) DRAM controller is tricky to design and would be the subject of its own textbook; most design teams purchase memory controllers as prevalidated IP blocks (intellectual property blocks). Some processes offer embedded DRAM on an SoC; this is less expensive per bit than SRAM but not as cheap as commodity DRAM.

4.2.4 Verilog Syntax

This section summarizes various other commonly used Verilog constructs, including numbers, bit swizzling, x and z values, delays, port connections, the preprocessor, and assertions.

4.2.4.1 Numbers

The format for declaring *integer literal constants* (numbers) is `N'Bvalue`, where `N` is the size in bits and `B` is a letter indicating the base, with `b` for binary, `d` for decimal, `h` for hexadecimal, and, less commonly used, `o` for octal. If the base is omitted, it defaults to decimal. Omitting the size leads to subtle gotchas and should generally be avoided [Sutherland06]. The number is padded with leading zeros to bring it to the specified size. Underscores are ignored and can be helpful to break long numbers into more readable chunks. [Table 4.2](#) gives examples of representing constants.

An exception to the rule of always including the bit size for constants is when filling a signal with a repeated value of 0, 1, x, or z. For example, to fill the 5-bit data signal with all 1s (11111_2), use `assign data = '1`; To fill it with all 0s (00000), zs (`zzzzz`), or xs (`xxxxx`), use `assign data = '0`, `assign data = 'z`, or `assign data = 'x`, respectively. This syntax is particularly convenient when the bit size is parameterized.

Table 4.2 SystemVerilog numbers.

Number	Bits	Base	Decimal Value	Stored
3'b101	3	2	5	101
8'b1010_1011	8	2	171	10101011
8'b11	8	2	3	00000011
3'd6	3	10	6	110
8'hAB	8	16	171	10101011

4.2.4.2 Bit Swizzling

Bit swizzling is the process of selecting, replicating, and concatenating (joining together) signals and constants to form new signals. The {} operator concatenates bits. To replicate bits, use curly brackets around the number of times to replicate the bit; for example, {5{a[2]}} replicates bit 2 of signal a, a[2], five times. [Code Example 4.26](#) shows more examples of swizzling.

Code Example 4.26 BIT SWIZZLING

```
// Create 64-bit bus of all 0 or all 1 depending on the value of s
logic [63:0] s64;
assign s64 = {64{s}};

// Create 8-bit bus of all x or all 1s
logic [7:0] data0, data1;
assign data0 = 'x;      // data0 = xxxxxxxx
assign data1 = '1;      // data1 = 11111111

// Concatenate two 32-bit values to form a 64-bit value
logic [31:0] upper, lower;
logic [63:0] entire;
assign entire = {upper, lower};

// Form bus containing constants and parts of other busses
// y = a[2] a[2] b[2] b[1] b[0] a[1] a[0] 1 0 1
logic [2:0] a, b;
logic [9:0] y;
assign y = {{2{a[2]}}, b, a[1:0], 3'b101};

// Sign- and zero-extend signal c
// csignext = c[3] c[3] c[3] c[3] c[3:0]
// czeroext = 0    0    0    0    c[3:0]
logic [3:0] c;
logic [7:0] csignext, czeroext;
assign csignext = {{4{c[3]}}, c};
assign czeroext = {4'b0, c};

// Add two 32-bit numbers to produce a 32-bit sum and a 1-bit carry out
logic [31:0] a, b, sum;
logic      cout;
assign {cout, sum} = a + b;
```

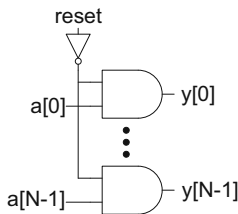


Fig. 4.26 Gating circuit.

4.2.4.3 One-Bit with Many-Bit Gating

Often many bits need to be *gated* by one bit. For example, we might want to force an N -bit bus a to all 0s during a reset condition. Verilog does not directly support combining one and many bits with a bitwise operator, but we can use either a conditional operator or swizzling as shown in [Code Example 4.27](#). We prefer the conditional operator for brevity, although it is important to remember that it reduces to N AND gates rather than an N -bit 2:1 multiplexer. Both idioms produce the logic shown in [Fig. 4.26](#).

Code Example 4.27 GATING

```
logic      reset;
logic [N-1:0] a, y1, y2;
assign y1 = reset ? '0 : a; // Conditional operator: y1 = 000... upon reset
assign y2 = {N{~reset}} & a; // Bit swizzling:      y2 = 000... upon reset
```

4.2.4.4 Delays

Verilog supports delays in simulation, indicated by `#`. [Code Example 4.28](#) shows the single-ported RAM from [Code Example 4.20](#) annotated with delays of 2 time units propagation delay on the `rd` output and 1 time unit of clock-to-q delay for writing the memory.

Code Example 4.28 RAM WITH DELAYS

```
module ram1plr_async_delay #(parameter M = 6, N = 32) (
    input  logic      clk,
    input  logic      we,
    input  logic [M-1:0] a,
    input  logic [N-1:0] wd,
    output logic [N-1:0] rd);

    bit [N-1:0] mem[2**M];

    assign #2 rd = mem[a];
    always_ff @(posedge clk)
        if (we) mem[a] <= #1 wd;
endmodule
```

Delays are necessary in testbenches to toggle signals, such as clocks, in the desired sequence, as will be illustrated in [Code Example 5.1](#). Delays can be helpful (but not essential) for debugging to visualize the register outputs changing after the clock edge. Although Verilog has a `timescale` directive to indicate the value of each time unit, it's best to think of delays as just being relative and having arbitrary units. Delays are ignored during synthesis because the delay of a circuit depends on its physical implementation and varies with voltage, temperature, and manufacturing variations.

Delays are not necessary outside of testbenches. Some logic simulators, including Verilator, are significantly faster when all signals toggle on the same clock edge. The industry has mostly converged on a coding style with no `#` delays outside of testbenches, and Wally follows this style.

4.2.4.5 Port Connections

So far, we have connected ports by using the same order in the module instantiation as in the module declaration. This approach is succinct but can be difficult to read because you may have to look at both the instantiation and declaration to understand which signals are going to which ports. Moreover, if another designer reorders the ports in the module declaration, the instantiation might have subtle bugs because signals go to the wrong ports.

SystemVerilog has several other methods of specifying port connections illustrated in [Code Example 4.29](#). The module describes a 4-bit register built from four 1-bit resettable flip-flops. `f0` is *connected by ordered list*. `f1` is *connected by name*. For example, the `clk` port of this flip-flop is connected to the `clk` input of `reg4`. The `d` port of this flip-flop is connected to the `x[1]` input of `reg4`. This solves the above issues with ordered lists but is substantially more typing. In this example, the signals are written one per line, just to be even less succinct. `f2` addresses this with *implicit port connections*. When the signal name is the same in the declaration and instantiation, it can just be listed once. For example, `.clk` connects the `clk` port of `f2` to the `clk` input of `reg4` without having to type it twice. The data terminal is still connected by name because it has a different name in the declaration (`d`) and the instantiation (`x[2]`). The ports could have been listed in any order and would still connect properly, although it is good practice to keep them in the same order. Finally, `f3` illustrates implicit port connections with `.*`, which automatically connects all of the ports that have the same name in the declaration and instantiation. In this case, `reset` and `clk` are connected using `.*`.

Code Example 4.29 PORT CONNECTIONS

```
module reg4(input  logic      clk,
            input  logic      reset,
            input  logic [3:0] x,
            output logic [3:0] y);

    flopr #(1) f0(clk, reset, x[0], y[0]);
    flopr #(1) f1(.clk(clk),
                  .reset(reset),
                  .d(x[1]),
                  .q(y[1]));
    flopr #(1) f2(.clk, .reset, .d(x[2]), .q(y[2]));
    flopr #(1) f3(.d(x[3]), .q(y[3]), .*);
endmodule
```

For brevity, we prefer ordered lists for simple modules such as flip-flops and muxes. We connect by name with implicit port connections for more complex modules. We find `.*` makes code difficult to read because it is harder to trace how signals propagate through the module hierarchy when they aren't listed in the instantiation, so we generally avoid it. However, `.*` can be helpful in a testbench or top-level module where signals are all connected by name, but the large number of names are tedious to type.

Unused outputs can be discarded by omission, that is, by leaving that signal blank when connected by name, for example, `.outsigname()`.

4.2.4.6 Preprocessor

Like the C language, Verilog has a *preprocessor* that reads and modifies files before regular processing begins. Some of the common *compiler directives* it handles are ``define` and ``include`.

4.2.4.6.1 ``define`

``define` is a *text macro substitution* directive. It is commonly used to define global constants and improve code readability, similar to `#define` in C and Java. For example, ``define XLEN 64` replaces `XLEN` with `64` everywhere it is found in the Verilog code. `XLEN` can then be easily changed to `32` by changing it only in the ``define`.

Code Example 4.30 shows an excerpt from a header file that uses ``define` to set `XLEN` to `32` and `PPN_BITS` to an expression that depends on `XLEN`, which evaluates to `22` in this case. Macros with a tick (```) before their name are substituted into code verbatim during preprocessing. For example, everywhere ``PPN_BITS` appears in other code is replaced by `(`XLEN==32 ? 22 : 42)`, which in turn is replaced by `(32==32 ? 22 : 42)`. Macro expressions should have parentheses around them to avoid order-of-operations issues when substituted into code [Wilson10].

Code Example 4.30 `config.vh`

```
`define XLEN 32
`define PPN_BITS (`XLEN==32 ? 22 : 44)
...
```

The order of preprocessing and compiling files is not defined by SystemVerilog. If a macro is ``defined` in more than one place with different values, its behavior can be nondeterministic.

4.2.4.6.2 ``include`

``include` inserts the entire contents of a file, similar to `#include` in C and Java. The included file is called a *header* file and is conventionally suffixed with `.vh` or `.svh`. A SystemVerilog file would incorporate `config.vh` above with the following:

```
`include "config.vh"
```

4.2.4.7 Assertions

SystemVerilog provides assertions to check properties during simulation. Assertions are of the form `assert (condition) else $error("message");`. If the condition is not satisfied, the message is printed. [Code Example 4.31](#) checks that ``S_SUPPORTED` is only 1 if ``U_SUPPORTED` is 1, too.

Substitute `$warning` for `$error` if the problem is less severe.

Code Example 4.31 ASSERTION

```
always_comb
    assert (`U_SUPPORTED | (`S_SUPPORTED == 0)) else $error ("S supported only if U is too");
```

4.2.5 Configurable Hardware

A complex SoC is usually configurable to support different needs. For example, the size of a branch predictor buffer should be configurable so that it can easily be changed without rewriting code. Wally takes such configuration to an extreme, making almost everything configurable, including XLEN (RV32 or RV64) and the supported extensions. This way, one code base can be used to support many configurations, from RV32E to RV64GC.

SystemVerilog has flexible ways to configure hardware based on compile-time constants, including disabling unused signals, setting the width of signals, and conditionally instantiating or generating multiple copies of a piece of hardware. We initially illustrate configurations with ``define`. This method does not work for SoCs with multiple cores that have different configurations because the order of defining the macros is nondeterministic. Therefore, we then introduce a more general method of passing parameters hierarchically.

4.2.5.1 Disabling Signals

[Code Example 4.32](#) shows a macro used to disable a feature in a certain configuration. The file includes `config.vh`, which contains the configuration macros. When `XLEN = 32`, `WriteMSTATUSH = CSRMWriteM & (CSRAdrM == MSTATUS)`. If `XLEN` is 64, `WriteMSTATUSH` is tied to 0 (and likely optimized away, along with any affected logic) because the `mstatus` register does not exist in this configuration.

Code Example 4.32 DISABLE FEATURE WITH A MACRO

```
`include "config.vh"
assign WriteMSTATUSH = CSRMWriteM & (CSRAdrM == MSTATUS) & (`XLEN == 32);
```

4.2.5.2 Values of Configurable Width

When a signal width is parameterized, the width of its value must be adjusted to match. Unfortunately, SystemVerilog does not accept expressions in bit widths; ('XLEN-1)'b0 is not recognized. The examples that follow show some ways to express values of configurable width using bit swizzling.

```
// Create XLEN-bit bus of all 0 or all 1 depending on s
logic [`XLEN-1:0] sx;
assign sx = {'XLEN{s}}; // Or: assign sx = s ? '1 : '0;

// Sign-extend Instr[31:20] to `XLEN bits
logic [`XLEN-1:0] extend;
assign extend = {{{`XLEN-12}{Instr[31]}}, Instr[31:20]};

// Place LT in the bottom bit and WIDTH-1 0s in the upper bits
logic [`XLEN-1:0] SLT;
assign SLT = {{{`XLEN-1}{1'b0}}, LT};
```

The number 0 always fills all bits and can be used safely in place of '0.

SystemVerilog also defines '0, '1, 'x, and 'z to place that value in each bit of the destination, thus filling the destination with all 0s, 1s, Xs, or Zs. [Code Example 4.9](#) uses 'z to specify the floating output of a tristate of configurable width.

4.2.5.3 Conditional Generates

Conditional generate statements allow hardware to be created based on a condition. [Code Example 4.33](#) shows how to instantiate a multiply/divide (`muldiv`) module only when the 'M_SUPPORTED configuration value is 1. Otherwise, the `MulDivResultW` and `DivBusyE` signals are tied to 0, so they aren't floating, and no hardware is created during synthesis.

Conditional generates may contain almost any Verilog statements, including signal declarations, module instantiations, continuous assignments, and `always` blocks. The condition must be known at compile time because hardware is either generated or not generated; it can't wink in and out of existence based on the value of a changing signal. Generates produce another level of hierarchy in simulation if they contain any modules or internal signals; it is good practice to give this level a name with a colon (:) after the `begin` statement. For example, if this module is called `top`, then the `mdu` module is `top.mdugen.mdu`. If the generate loop is not named, the simulator will name it automatically, such as `top.genblk0.mdu`.

Verilog has an ``ifdef` compiler directive to conditionally compile code. Unused signals could be conditionally eliminated with ``ifdef` rather than just set to 0. However, we find ``ifdef` clutters the code excessively and complicates simulation because signals will exist in certain configurations and vanish in others. We prefer driving unused signals to 0 and allowing logic synthesis to optimize them away.

Code Example 4.33 CONDITIONALLY INSTANTIATED MODULE
(WITHIN `wallypipelinedhart`)

```

...
logic [`XLEN-1:0] MulDivResultW;
logic          DivBusyE;

/* Optional generate/endgenerate statement: generate */
if (`M_SUPPORTED) begin: mdugen
    mdu mdu(.clk, .reset, .ForwardedSrcAE, .ForwardedSrcBE,
            .Funct3E, .Funct3M, .MulDivE, .W64E,
            .MulDivResultW, .DivBusyE, .StallM, .StallW, .FlushM, .FlushW);
end else begin // No M instructions supported
    assign MulDivResultW = 0;
    assign DivBusyE = 0;
end
/* Optional generate/endgenerate statement: endgenerate */

```

Conditional generates (and generate loops described in [Section 4.2.5.4](#)) may optionally be surrounded by the generate and endgenerate statements as shown in comments in [Code Example 4.33](#). We omit them because brevity is the soul of wit.

4.2.5.4 Generate Loops

A *generate loop* creates multiple pieces of hardware based on a loop variable. [Code Example 4.34](#) shows a *priority circuit* of configurable size N that generates a *thermometer code* using a generate loop. It detects the least significant 1 digit in the input and puts 1s in all the less significant output bits and 0s in the upper output bits. For example, if $a = 01010000$, $y = 00001111$ because the four least significant input bits are 0 and the fifth input bit is 1. The generate loop implies a chain of $N-1$ AND gates and N inverters, as shown in [Fig. 4.27\(a\)](#). The loop index i is declared as a *genvar*, which is not a physical signal in the circuit but rather used to count the hardware being generated. As with [Code Example 4.33](#), the generate loop is given an optional name: *therm*. Most synthesis tools, particularly Synopsys Design Compiler, are smart enough to recognize this priority circuit and optimize it to a tree structure rather than a ripple structure when speed matters, as shown in [Fig. 4.27\(b\)](#), thus reducing the delay from $N-1$ to $\log_2 N$ AND gates along the critical path.

Code Example 4.34 PRIORITY THERMOMETER CIRCUIT

```

module prioritythermometer #(parameter N = 8) (
    input  logic [N-1:0] a,
    output logic [N-1:0] y);

    // Create thermometer code mask
    genvar i;
    assign y[0] = ~a[0];
    for (i = 1; i < N; i++) begin: therm
        assign y[i] = y[i-1] & ~a[i];
    end
endmodule

```

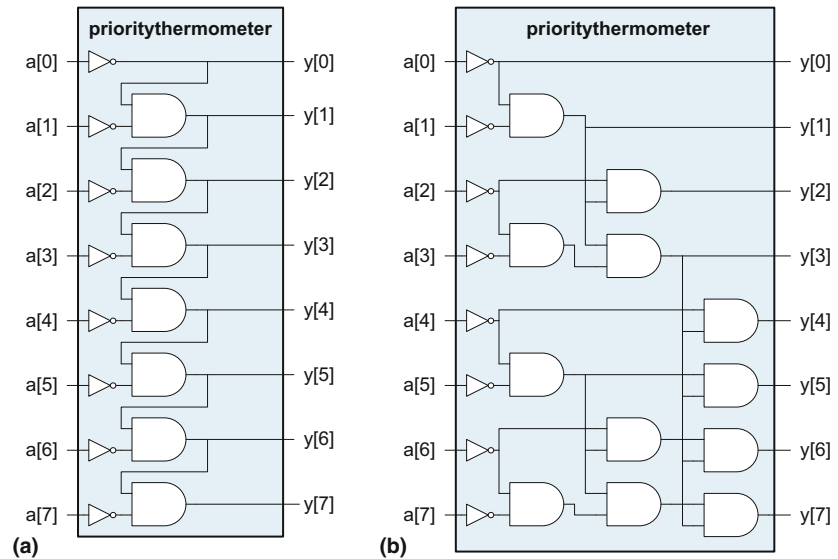



Fig. 4.27 Priority circuit: (a) Direct ripple implementation and (b) Faster tree implementation.

4.2.5.5 Arrays of Modules

To instantiate identical modules, SystemVerilog's array of instances notation is more succinct than a generate loop. [Code Example 4.35](#) illustrates the `tlbram` instantiating an array of `TLB_ENTRIES` identical `tlbramline` modules. SystemVerilog connects signals to arrays as you would hope; for example, when the `Matches` signal (of width `TLB_ENTRIES`) is connected to the one-bit `tlbramline` array `re` port, each instance of the array receives the corresponding bit of `Matches`. The `q` outputs of the `tlbramline` array are ``PPN_BITS+10` bits wide and are bundled together into the `RamRead` array which has `TLB_ENTRIES` entries of width ``PPN_BITS+10` bits. [Fig. 11.21](#) shows all these connections. The example also illustrates using module parameters (`TLB_ENTRIES`) and global configuration parameters (``PPN_BITS` and ``XLEN`).

This example also illustrates a *distributed AND-OR mux* read circuit, which is used in many other components such as the cache reads in [Section 10.4.5.1](#). Unlike the `ram1plrw` from [Code Example 4.20](#) that takes a binary address and uses array notation `rd <= mem[a]` to read from that address, the `tlbram` reads all entries and zeroes out the entries that don't match. Specifically, `tlbram` takes a one-hot `Matches` input that indicates which entry to read. Each `tlbramline` ANDs its `Match` signal with the line to produce `RamRead`, which is 0 when its `Match` signal is 0. The `or_rows` module in `tlbram` then performs a bit-wise OR on the `RamRead` signals to produce a ``PPN_BITS+10`-bit `PageTableEntry` signal.

Code Example 4.35 **tlbram WITH tlbramline AND or_rows**

```

`include "config.vh"

module tlbram #(parameter TLB_ENTRIES = 8) (
    input  logic          clk, reset,
    input  logic [XLEN-1:0] PTE,
    input  logic [TLB_ENTRIES-1:0] Matches, WriteEnables,
    output logic [PPN_BITS-1:0] PPN,
    output logic [7:0] PTEAccessBits,
    output logic [TLB_ENTRIES-1:0] PTE_Gs);

    logic [PPN_BITS+9:0] RamRead[TLB_ENTRIES];
    logic [PPN_BITS+9:0] PageTableEntry;

    // RAM implemented with array of flops and AND/OR read logic
    tlbramline #(`PPN_BITS+10) tlbramline[TLB_ENTRIES-1:0]
        (.clk, .reset, .re(Matches), .we(WriteEnables),
         .d(PTE[PPN_BITS+9:0]), .q(RamRead), .PTE_G(PTE_Gs));
    or_rows #(TLB_ENTRIES, `PPN_BITS+10) PTEOr(RamRead, PageTableEntry);

    // Rename the bits read from the TLB RAM
    assign PTEAccessBits = PageTableEntry[7:0];
    assign PPN = PageTableEntry[PPN_BITS+9:10];
endmodule

module tlbramline #(parameter WIDTH = 22) (
    input  logic          clk, reset,
    input  logic          re, we,
    input  logic [WIDTH-1:0] d,
    output logic [WIDTH-1:0] q,
    output logic          PTE_G);

    logic [WIDTH-1:0] line;

    flopenr #(WIDTH) pteflop(clk, reset, we, d, line);
    assign q = re ? line : 0;
    assign PTE_G = line[5]; // Send global bit to CAM as part of ASID matching
endmodule

module or_rows #(parameter ROWS = 8, COLS=2) (
    input  var logic [COLS-1:0] a[ROWS],
    output logic [COLS-1:0] y);

    genvar row;
    if(ROWS == 1)
        assign y = a[0];
    else begin
        logic [COLS-1:0] mid[ROWS-1:1];
        assign mid[1] = a[0] | a[1];
        for (row = 2; row < ROWS; row++)
            assign mid[row] = mid[row-1] | a[row];
        assign y = mid[ROWS-1];
    end
endmodule

```

4.2.5.6 Configuration With Parameter Structure

``define` is the simplest way to configure code. However, it does not scale well when a piece of code is reused in different configurations. For example, consider building a chip with five Wally cores, four of which are rv64gc application processors and one that is an rv32e microcontroller for power and clock management. The different cores have identical Verilog but use different `config.vh` files. When the second config file is read, it overwrites macros from the first one, leading to unpredictable behavior.

The scalable solution for SoCs is to use Verilog's parameter capabilities, passing different parameters to the different cores. The top-level module in turn passes parameters to the submodules. A core may have dozens or hundreds of parameters, so typing out each parameter in every module is too cumbersome. Instead, we use Verilog's structure syntax to pass a single structure containing many parameters.

The following examples illustrate defining and using parameter structures in Wally. For brevity, they only show three of the parameters: `XLEN`, `MISA`, and `M_SUPPORTED`. The examples use a fair amount of new SystemVerilog syntax.

In [Code Example 4.36](#), `cvw.sv` creates a *package* called `cvw` that defines the `cvw_t` struct with all of the parameters. Packages hold definitions that can be imported into other modules. The struct contains many elements of various data types. It is declared as `packed`, meaning that it comprises consecutive bitfields. The `int` fields are each 32 bits and `logic` is 1 bit, so the illustrated portion of `cvw_t` is 65 bits.

Code Example 4.36 `src/cvw.sv` PARAMETER STRUCTURE DEFINITION PACKAGE

```
package cvw;

typedef struct packed {
    int      XLEN;           // Machine width (32 or 64)
    int      MISA;          // Machine Instruction Set Architecture
    ...
    logic     M_SUPPORTED; // Mul/div extension supported
} cvw_t;

endpackage
```

In [Code Example 4.37](#), the rv64gc `config.vh` defines configuration parameters that are specific to rv64gc, such as the `XLEN` and `misa`. The `localparam` statement declares named constant values, possibly dependent on other parameters. It also includes a `config-shared.vh` file ([Code Example 4.38](#)) that defines constants and parameters that are derived

from the `config.vh`-specific parameters. For example, `rv64gc` = RV64I-MAFDC (with supervisor and user modes) has `XLEN = 64` and a `misa` setting A, C, D, F, I, M, U, and S. The 12th bit of `MISA` indicates whether the M (multiply/divide) extension is supported. `rv32i/config.vh` in [Code Example 4.39](#) contains a different configuration with `XLEN = 32` and a `MISA` setting of only I. `M_SUPPORTED` is 1 for `rv64gc` and 0 for `rv32i`. The tedious `parameter-defs.vh` file in [Code Example 4.40](#) creates a `cvw_t` structure named `P` (short for *Parameters*) with all of its fields initialized by the `localparam` values from the configuration file.

Code Example 4.39 `config/rv64gc/config.vh` CONFIGURATION DEFINITIONS

```
localparam XLEN = 32'd64;
localparam MISA = (32'h00000104 | 1<<5 | 1<<3 | 1<<18 | 1<<20 | 1<<12 | 1<<0);
...
`include "config-shared.vh"
```

Code Example 4.38 `config/shared/config-shared.vh` CONFIGURATION DEFINITIONS

```
localparam M_SUPPORTED = ((MISA >> 12) % 2 == 1);
```

Code Example 4.39 `config/rv32i/config.vh` CONFIGURATION DEFINITIONS

```
localparam XLEN = 32'd32;
localparam MISA = 32'h00000100;
...
`include "config-shared.vh"
```

Code Example 4.40 `config/shared/parameter-defs.vh` PARAMETER INITIALIZATION

```
localparam cvw_t P = '{
  XLEN :      XLEN,
  MISA :      MISA,
  ...
  M_SUPPORTED : M_SUPPORTED
};
```

Next, we parameterize the SystemVerilog files using `cvw_t P`. Each file adds `import cvw::*; #(parameter cvw_t P)` to the module declaration line so that it reads the `cvw` package into its local namespace (to define `cvw_t`)

and accepts the parameter structure *P*. In [Code Example 4.41](#), `wallypipelinedsoc.sv` is the top-level module. It instantiates `wallypipelinedcore`, passing the parameter *P* down to this module. Note that it also defines a reset synchronizer that does not require any parameters. In [Code Example 4.42](#), `wallypipelinedcore` is the processor core with the various functional units. Observe how it defines a bus of width `XLEN` using the `P.XLEN` syntax instead of ``XLEN` to access this element of the parameter structure. It also instantiates the integer execution unit (IEU) and conditionally instantiates the multiply/divide unit (MDU) when the *M* extension is supported, passing the parameters down to each of these units.

Code Example 4.41 `src/wally/wallypipelinedsoc.sv` TOP-LEVEL MODULE

```
module wallypipelinedsoc import cvw::*; #(parameter cvw_t P) (
    input  logic          clk,
    input  logic          reset_ext,
    output logic          reset,
    ...

    synchronizer resetsync(.clk, .d(reset_ext), .q(reset));
    wallypipelinedcore #(P) core(.clk, .reset, ...);
    ...

endmodule
```

Code Example 4.42 `src/wally/wallypipelinedcore.sv` TOP-LEVEL MODULE

```
module wallypipelinedcore import cvw::*; #(parameter cvw_t P) (
    input  logic          clk,
    input  logic          reset,
    ...
    logic [P.XLEN-1:0] MDUResultW;

    ieu #(P) ieu(.clk, .reset, ...);
    if (P.M_SUPPORTED) begin:mdu
        mdu #(P) mdu(.clk, .reset, ..., MDUResultW);
    end
    ...

endmodule
```

Simulation requires a testbench to initialize the configuration structure *P* based on the desired configuration and to instantiate the top-level module with these parameters, as shown in [Code Example 4.43](#). [Section 6.3](#) shows how synthesis uses a *wrapper* to load the desired configuration.

In an SoC containing multiple different cores, replace `parameter_defs.vh` with files defining different parameters for each core.

Code Example 4.43 TESTBENCH IN `testbench/testbench.sv`

```
`include "config.vh"

import cvw::*;

module testbench;

    `include "parameter-defs.vh"

    wallypipelinedsoc #(P) dut(.clk, .reset, ...);
    ...

endmodule
```

4.3 WALLY STYLE GUIDELINES

“A foolish consistency is the hobgoblin of little minds.”

—Ralph Waldo Emerson

Most organizations and experienced designers establish a set of HDL style guidelines for large projects with multiple contributors to speed development, maintain consistency, and reduce errors. Many reasonable choices exist, but consistency is valuable for readability across the design, and the designer must be empowered to break these guidelines when the need arises. This chapter summarizes the guidelines used in this book for signal names, white space, modules, and configuration.

Two other extensive Verilog style guides are available from the following organizations:

- ▶ **lowRISC:** github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md
- ▶ **Freescale:** people.ece.cornell.edu/land/courses/ece5760/Verilog/FreescaleVerilog.pdf

4.3.1 Signals

Signal naming conventions are important in all programming, and particularly in HDL. SystemVerilog is case sensitive; for example, `RegWrite`, `regwrite`, and `regWrite` are not the same signal. If capitalization is inconsistent, the developer may have to search across files to see how a particular signal is capitalized or may introduce bugs where the same signal is capitalized differently in different places and doesn’t connect. Pipelined systems

These style guidelines apply to synthesizable code.

Testbenches are an entirely different matter and generally look more like software.

Unlike Verilog, VHDL is case insensitive, which can cause problems with EDA (electronic design automation) tools such as Synopsys Design Compiler and Xilinx Vivado. Most EDA tools use *Tool Command Language* (Tcl) to process a design’s RTL, such as defining a clock or specifying timing constraints. Tcl is case sensitive, so it is entirely possible for the EDA tool to not find the VHDL signal name if capitalization is not consistent throughout the VHDL files.

use suffixes to keep track of a signal’s pipeline stage and prefixes to distinguish related signals from different blocks. In SystemVerilog, nearly all signals are declared of type `logic`. Ports are connected by order for simple generic modules and by name for more complex modules, preferably keeping the name the same across the hierarchy.

4.3.1.1 Capitalization

Table 4.3 lists various conventions for capitalizing signal names [Name-Conv]. *flatcase* works well for short simple names, such as in the C standard library, but is difficult to read for more complex signal names common in HDL design. *snake_case* involves reaching for the underscore key with the right pinky finger, which we find wearisome when typing. *camelCase* has the advantage that it requires no capitals for single-word variable names yet remains highly readable for multiword names, so this is a personal favorite of the authors when programming. However, *UpperCamelCase* is common in HDL design, looks natural in block diagrams, and was the choice of Patterson and Hennessy in the diagrams for *Computer Organization and Design*, from which Wally is indirectly descended [Hennessy20]. *MACRO_CASE* is the most difficult to type but makes constants and macros stand out from variables. In an empirical study, Binkley found that *camelCase* leads to higher accuracy and faster recognition than *snake_case* [Binkley09].

Guideline 4.1 Use flatcase for `clk` and `reset`.

These signals appear in almost every module and should have easy names.

Guideline 4.2 Use UpperCamelCase for signals.

Other conventions are reasonable, but it’s important to pick one and stick with it. Wally inherits this convention choice from prior books.

Guideline 4.3 Use *MACRO_CASE* for constants and parameters.

Most developers use this convention.

Table 4.3 Capitalization conventions.

Example	Convention Name
<code>regwrite</code>	<i>flatcase</i>
<code>regWrite</code>	<i>camelCase</i> / <i>dromedaryCase</i>
<code>RegWrite</code>	<i>UpperCamelCase</i> / <i>PascalCase</i>
<code>reg_write</code>	<i>snake_case</i>
<code>REG_WRITE</code>	<i>MACRO_CASE</i>

4.3.1.2 Suffixes and Prefixes

In pipelined systems, the same signal often exists in multiple pipeline stages and must be named in a way to distinguish the stage. HDL designers normally do this with a suffix. For example, the Wally pipeline has Fetch, Decode, Execute, Memory, and Writeback stages. The suffix may be one letter (F, D, E, M, or W) or an abbreviation (FET, DEC, EXE, MEM, WRB). Single-letter abbreviations are more succinct, but complex systems with many pipeline stages may need longer stage suffixes for clarity. Signals outside the pipeline (e.g., in the bus interface or hardware page table walker) have no suffix.

While pipeline registers accept inputs from one pipeline stage and produce outputs for the next stage and, normally, combinational logic accepts inputs and produces outputs within the same pipeline stage, some components operate across multiple pipeline stages. Such components are a common source of bugs, and care must be taken in signal naming. Some components that necessarily break these rules include the register file (reading in the Decode stage and writing in the Writeback stage), the external bus interface (accepting requests from the Fetch and Memory stages), and the hazard unit (combining hazards from many stages and sending stalls and flushes to each stage). For such components, signals generated within the pipeline use the suffix of the pipeline stage in which they were produced. Signals generated outside the pipeline (i.e., in the hazard unit) use the suffix of the stage in which they will be used.

Guideline 4.4 Use suffixes on all signals associated with pipeline stages.

For the pipelined Wally processor, use F, D, E, M, and W suffixes.

Guideline 4.5 Use prefixes to distinguish related signals from different sources.

Sometimes signals are prefixed with their source unit to distinguish similar signals from different places. For example, the datapath uses `IEUResultM`, `MDUResultW`, and `SCResultW` to distinguish results from the IEU, MDU, and store conditional units. Source units are typically abbreviated in all caps (e.g., IEU, IFU, LSU, FPU, SC, and MDU).

4.3.1.3 Declaring Signals

Guideline 4.6 Declare all internal signals, including 1-bit signals.

Undeclared signals are a source of bugs.

Guideline 4.7 Only use (bidirectional) inout pins at the top-level chip module. Define them as `inout tri`.

Bidirectional pins should only be used when they are an essential part of an external I/O standard, such as general-purpose I/O (GPIO).

Internally, use unidirectional buses and multiplexers rather than bidirectional busses and tristates.

Guideline 4.8 Declare all other signals as `logic`.

The `logic` datatype can be used anywhere except on tristate nets, supports `x` and `z`, and avoids the ugly rules of `wire` and `reg` in old-school Verilog.

Guideline 4.9 Declare busses as little-endian.

For consistency with prevailing custom, declare a bus as `[31:0]` rather than `[0:31]`.

Guideline 4.10 Avoid passing arrays through ports if possible but declare the port as `var logic` if necessary.

Arrays are usually used to represent memories and are kept internal to a module. In unusual cases when arrays have to be passed through ports, SystemVerilog wants the `var logic` syntax shown in [Code Example 4.44](#), which is otherwise outside the scope of this chapter.

[Code Example 4.44](#) illustrates many of these guidelines.

Code Example 4.44 IFU SIGNAL DECLARATIONS (PARTIAL)

```
module ifu import cvw::*; #(parameter cvw_t P) (
    input    logic          clk, reset,                // Lower case
    input    logic          StallF, StallD, StallE, StallM, StallW, // Suffix
    input    logic          FlushF, FlushD, FlushE, FlushM, FlushW, // Suffix
    input    logic [P.XLEN-1:0] SATP_REGW,             // Little-endian
    input    logic [1:0]    STATUS_MPP,
    input var logic [7:0]    PMPCFG_ARRAY_REGW[P.PMP_ENTRIES-1:0], // Passing array
    output   logic          InstrAccessFaultF);
    logic [P.XLEN-1:0]      PCNextF;
    logic                   BranchMisalignedFaultE;
    ...
```

4.3.1.4 Port Connections

Guideline 4.11 Connect ports by order for simple generic modules such as flip-flops and multiplexers.

The connections are usually obvious, and this convention is most succinct, as shown for `mux2` in [Code Example 4.45](#).

Guideline 4.12 Connect ports by name for other modules.

Use `.sig` to connect a port named `sig` to an external signal of the same name. Use `.sig(name)` to connect a port named `sig` to an external signal called `name`, as shown for ports such as `PAdr` connecting to the `dmmu` in [Code Example 4.45](#).

Guideline 4.13 For non-generic modules used only once, use the same name for ports and signals connecting to those ports.

Tracing logic is easiest when signals don't change names unnecessarily at module boundaries. It also allows `.sig` notation in the port connections, which is most succinct. For example, the `dmmu` module in [Code Example 4.45](#) does this for many ports such as `clk` and `PrivilegeModeM`.

Guideline 4.14 Avoid `.*` notation except in wrapper modules.

When a module uses `.*`, the reader needs to open up the module declaration to identify the inputs and outputs, which makes tracing logic more cumbersome. An exception is a top-level or wrapper module where nearly all signals have the same name, and humans find them tedious to read.

Guideline 4.15 Pack multiple port connections on one line of the instantiation.

Brevity improves the ability to see the big picture. For example, in [Code Example 4.45](#), all of the port connections (`ALUResultE`, ..., `IEUResultE`) are on one line.

The *top-level* module is the highest module in the design hierarchy and is typically a structural module that instantiates modules. A *wrapper* module maps signals to desired interfaces, for example, pins on an FPGA that are connected to the system clock, switches, and so on.

Code Example 4.45 PORT CONNECTIONS

```

mux2 #(P.XLEN) ieuresultmux(ALUResultE, AltResultE, ALUResultSrcE, IEUResultE);
mmu #(.TLB_ENTRIES(P.DTLB_ENTRIES), .IMMU(0)) dmmu(
    .clk, .reset, .SATP_REGW, .STATUS_MXR, .STATUS_SUM,
    .STATUS_MPRV, .STATUS_MPP,
    .PrivilegeModeW, .DisableTranslation(SelHPTW),
    .PAdr(PreLSUPAdrM), .VAdr(IEUAdrM),
    ...
    .PMPCFG_ARRAY_REGW, .PMPADDR_ARRAY_REGW);

```

4.3.2 Spacing and Indentation

These guidelines aim to use whitespace in a consistent way to improve readability without bloating code.

Guideline 4.16 Indent with spaces instead of tabs.

Different editors interpret tabs differently.

Guideline 4.17 Indent by two spaces for each level of nesting.

Two and four spaces are common conventions. We prefer two because space is at a premium in a textbook layout.

Guideline 4.18 Align signal declarations within a block.

Signals are more readable when they are aligned.

Guideline 4.19 Leave a space around operators and after `if/case/for` statements.

This space is conventional and improves readability.

Guideline 4.20 Avoid space before semicolons, commas, closing parentheses, and open parentheses in module instantiations, and inside bus width declarations.

Extra space here doesn't help readability.

Guideline 4.21 Only use `begin` and `end` when more than one statement is required in a block.

Avoid unnecessary syntax.

Guideline 4.22 Pack the `begin` statements on the same line where feasible.

Avoid code length bloat.

Guideline 4.23 Limit lines to a maximum of 95 characters.

This is the most that fits on a line in this textbook layout and is also a comfortable number to fit in a text editor. The historic line length of 80 characters is also reasonable but no longer necessary with wider screens.

Guideline 4.24 Use blank lines sparingly to logically separate portions of code.

Place blank lines after internal signal declarations, between module declarations, and to divide unrelated portions of code.

Guideline 4.25 Use banner comments sparingly to separate major portions of code in a complex module when using hierarchy isn't a good alternative.

Banners take up space but could help the reader recognize different portions of code.

Code Example 4.46 shows an example that follows these guidelines.

Code Example 4.46 `redundantmul`

```

module redundantmul #(parameter WIDTH = 8)(
    input  logic [WIDTH-1:0]  a,b,
    output logic [2*WIDTH-1:0] out0, out1);

    if (P.DESIGN_COMPILER == 1) begin: mul
        logic [2*WIDTH-1+2:0] tmp_out0;
        logic [2*WIDTH-1+2:0] tmp_out1;
        DW02_multp #(WIDTH, WIDTH, 2*WIDTH+2) mul(
            .a, .b, .tc(1'b0), .out0(tmp_out0), .out1(tmp_out1));
        assign out0 = tmp_out0[2*WIDTH-1:0];
        assign out1 = tmp_out1[2*WIDTH-1:0];
    end else begin: mul
        assign out0 = a * b;
        assign out1 = 0;
    end
endmodule

```

4.3.3 Synchronous Design

Guideline 4.26 Use synchronous sequential design.

Every component should be a positive edge-triggered flip-flop, a standard memory, or combinational logic. All components should share a common clock named `clk`. Avoid latches, negative edge-triggered flip-flops, and asynchronous components. Exceptions to synchronous design should only occur when essential (e.g., at an interface with an asynchronous peripheral) and should be carefully documented with comments and reviewed.

Large systems with multiple subsystems inevitably have more than one clock domain. For example, a RISC-V core might run at a different frequency than a Wi-Fi packet processor or GPU or DRAM controller. Use synchronous sequential design within a clock domain. Clock domain crossings, sending signals between clock domains, are beyond the scope of this book.

Guideline 4.27 Instantiate memories and flip-flops using modules.

This makes it easier to identify the number of bits of state in the system and to swap out synchronous or asynchronous reset, scan, or low-power features.

Guideline 4.28 Do not use `#` delays except in the testbench.

Some simulators are significantly faster when all signals toggle on the cycle boundary.

An exception is that FSM state registers often use enumerated state types that are incompatible with the logic datatype of a flip-flop module. These state registers are defined with `always_ff` in the FSM (see [Code Example 4.16](#)).

4.3.4 Modules

Guideline 4.29 Place each module in a file with the same name as the module.

This is a conventional way to organize large code projects so finding modules is easy. An exception is that closely related small modules, such as generic multiplexers, share one file.

Guideline 4.30 The module name is immediately followed by the “(“ with no spaces, and the input and output ports begin on the same line as the module name when not parameterized.

This style is succinct and clear.

Guideline 4.31 When a module is parameterized, place the parameters and “(” on the same line as a module declaration. Place input and output ports on subsequent lines, indented by two spaces. Place the closing “);” on the same line with the last port declaration.

This style is succinct and keeps indents small.

Guideline 4.32 One `input` or `output` keyword is on each line in the module declaration, and ports with the same polarity (input or output) and width are declared on the same line. Text is aligned horizontally for each line: `input/output`, `bitwidth`, and signal names.

This style makes it easy to group and parse interface signals.

Guideline 4.33 Use behavioral coding style for leaf cells including generic blocks and control blocks.

Control blocks are most easily described behaviorally.

Guideline 4.34 Use structural coding style as much as possible for datapath blocks.

Describe a datapath with multiplexers, flip-flops, and other functional blocks. Push nontrivial control into a separate module. Sometimes a datapath needs `assign` statements for bit swizzling.

Guideline 4.35 Use hierarchy so that modules remain small and easy to understand.

This guideline sometimes trades off against using too much hierarchy that makes logic difficult to trace and wastes designer time declaring ports. We have reviewed a commercial microprocessor that has approximately 200 substantial modules averaging about 4,000 lines each.

Guideline 4.36 Avoid programming constructs that obscure the hardware being implied.

Recall that behavioral modules describe how they should function (e.g., using `if`, `case`, or `assign` statements). Structural modules are built from smaller modules.

Industrial code often uses behavioral style in datapaths when the designers and readers are experienced enough to readily visualize the hardware implied.

For example, SystemVerilog has constructs called *functions* and *tasks* that resemble function calls in C rather than hardware modules. These can be useful in testbenches, but our coding style avoids them in synthesizable blocks.

4.3.5 Configurable Hardware

Guideline 4.37 Define generic configurable modules where appropriate.

This includes mux and flop modules, but also more complex blocks such as a memory management unit (MMU) and cache elements that can be shared.

Guideline 4.38 Use a parameter structure to pass parameters to the top-level module and its submodules. The top-level module reads its parameters from a `config.vh` file.

Guideline 4.39 Use `#parameter` to define parameters for individual modules.

Guideline 4.40 Module interfaces should remain the same whether or not a feature is enabled.

Tie unused outputs to 0 with a conditional generate statement. Avoid using ``ifdef` to omit unused signals entirely; simulation files are more difficult to write when signals don't exist in certain configurations.

Guideline 4.41 Omit the optional `generate/endgenerate` keywords around conditional generates and generate loops.

Brevity! This is a controversial guideline; some teams require `generate/endgenerate`.

Guideline 4.42 When a generate block contains instance or signal declarations, name the block with a colon after the `begin` keyword.

This gives the signal or instance a consistent name to use when debugging.

4.3.6 Comments

Guideline 4.43 Place a banner comment at the beginning of each file.

Include the filename, author's name and email address, date, purpose of the module, and license. Including your name and date establishes your copyright ownership.

Guideline 4.44 Use comments to explain the big picture rather than restate what the code does.

Comments at the beginning of a module that explain the principles of operation are most valuable. Good variable name choices eliminate the need for many line-by-line comments.

Bad Example:

```
assign ab = ~a;           // Flip the bits
assign minusa = ab + 1; // Add one
```

Good example:

```
assign minusa = ~a + 1; // Take the 2's complement to negate a
```

4.4 WALLY CONFIGURATION

The Wally (cvw) repository organization is listed in [Fig. A.1](#).

As described in [Section 2.8.2](#), Wally is configurable to scale from a tiny rv32e core to simple rv{32/64}i cores to an rv32imc microcontroller with peripherals to rv{32/64}gc application processors with floating-point, branch prediction, caches, and virtual memory, all using the same code base. Each configuration is defined by:

```
$WALLY/config/<configname>/config.vh
```

In addition, Wally has many derived configurations, which are perturbations of the base configurations. They are used to test optional features, such as whether caches are enabled, how branch predictors perform with different table sizes, and whether various combinations of floating-point data sizes can be supported. These derived configurations are summarized in [Table A.5](#) and defined in:

```
$WALLY/config/deriv/<configname>/config.vh
```

The derived configurations are generated with the `bin/derivgen.pl` script using the `config/derivlist.txt` table. They are regenerated by running `sim/make deriv` or `sim/make`.

Two notable derived configurations are `buildroot` and `fpga`, intended to boot Linux in simulation and on an FPGA. These are variants of `rv64gc`, with the reset vector pointing to a boot ROM at `0x1000` with a zero-stage boot loader.

Simulation and synthesis scripts point to the desired configuration. The `config.vh` file defines many parameters used in SystemVerilog generate blocks. [Table A.3](#) lists key configuration parameters for each configuration, and [Table A.4](#) lists additional configuration parameters.

The source code is in `$WALLY/src`, with subdirectories for each unit. `testbench.sv` has a module called `riscvassertions` that checks legal values for configuration parameters. Examples of checks include:

- Optional supported modes are self-consistent (e.g., D requires F).

- ▶ Parameters have a limited set of legal values (e.g., `PMP_ENTRIES = 0, 16, or 64`).
- ▶ Cache organization is legal (e.g., way size doesn't exceed VM page size).

4.5 SUMMARY

Hardware description languages (HDLs) are used to specify the design of an SoC in a form that is both human and machine readable. The HDL code is simulated for logic verification and synthesized onto a chip or FPGA. Wally is written in SystemVerilog, the most widely used HDL.

SystemVerilog is a complex language, but only a tiny subset is used to describe synthesizable hardware. A good designer visualizes the desired type of logic, then writes SystemVerilog idioms that imply such logic. The designer should first determine whether a component is combinational or sequential logic. In combinational logic, the output depends on the current value of inputs. In sequential logic, the output depends on current and previous inputs; in other words, it has state.

SystemVerilog has several idioms for combinational logic. `assign` statements define Boolean or arithmetic equations. `always_comb` blocks can contain `case` statements or `if/else` statements. `case` statements define combinational logic with truth tables. `if/else` can be helpful for complex control logic. Synthesis excels at optimizing logic onto gates, so the designer can focus on writing a clear and concise description of what the logic should do and let the tool perform the Boolean optimization.

Synchronous sequential logic is built from edge-triggered flip-flops and combinational logic. The `always_ff @(posedge clk)` idiom describes edge-triggered flip-flops. FSMs can be described with a bank of flip-flops to hold the state, and combinational logic blocks to compute the next state and output. The next state code may be merged into the `always_ff` block for brevity. Memories, such as RAMs, ROMs, and register files, are a special case of sequential logic that contain an array of flip-flops or bit cells indexed by an address.

Larger modules are built from smaller modules, taking advantage of hierarchy and reuse. Parameterized modules can adjust bus widths based on the parameter value. Conditional generates, described by an `if` statement whose condition depends on a configuration parameter, instantiate modules only under certain conditions (e.g., when a feature is supported). Generate loops, described by a `for` statement, instantiate multiple pieces of hardware. SystemVerilog also has an array notation that is more succinct when the hardware is identical.

Compared to Verilog, SystemVerilog has a much larger set of features for writing testbenches to test hardware. These features include reading test vectors from files, checking results against expectations, and reporting

discrepancies. Indeed, SystemVerilog has an entire object-oriented feature set for modeling and verification comparable to C++. However, this book focuses on widely used constructs and the basic features needed for testbenches.

Good style is important for readable and maintainable HDL code. Wally uses coding guidelines similar to many other large SystemVerilog systems. It adopts UpperCamelCase signal names, with single-letter suffixes to indicate the pipeline stage. Except in generic or multiply instantiated modules, the same name is used for a signal throughout the hierarchy of modules. Ports are connected by order for simple modules (flip-flops and muxes), with the `.sig` notation for more complex modules, or with `.sig(name)` when a signal must change names where it enters a generic module. We find that a dense coding style improves readability by letting the developer see more of the code base on a screen. Wally is heavily configurable, allowing it to scale from a tiny rv32e core to a full-featured rv64gc application processor. The configurations avoid unnecessary hardware, keeping the size and critical path as tight as possible.

Design Verification

5

“Everyone knows debugging is twice as hard as writing a program in the first place.”

—Brian Kernighan, inventor of the C language (1974)

“If you didn’t test it, it doesn’t work.”

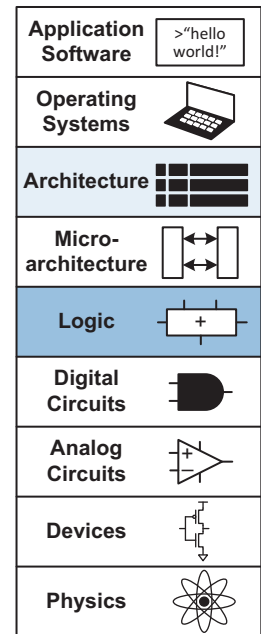
—Bob Colwell, chief architect on Pentium Pro through Pentium 4 [Colwell02]

Logic verification is the process of checking that logic conforms to the specification. Verification is generally more time-consuming and expensive than design. Verification engineers make up about half of an SoC development team. Verification engineers need a broad understanding of the system to define tests that exercise the full range of features and because errors they discover might come from anywhere. The mantra of SoC design is “If it is not verified, it will not work.”

Logic verification involves simulating the design with a *testbench* that applies a set of *test vectors* to the *device under test (DUT)* and checks that results match a reference model. Verification challenges include the following:

- ▶ Defining what features need to be tested
- ▶ Determining what test vectors to use
- ▶ Creating a reliable reference model
- ▶ Locating and fixing bugs when they are detected
- ▶ Coping with the enormous runtimes of large simulations

5.1	Verification Strategies
5.2	Design Verification Plan
5.3	Lint
5.4	Testbenches
5.5	Logic Simulators
5.6	Lock-Step-Compare
5.7	Assertions
5.8	Code Coverage
5.9	Functional Coverage
5.10	Formal Verification
5.11	Wally Verification
5.12	Summary



The Universal Verification Methodology (UVM) [IEEE 1800.2-2017] is a standardized method of logic verification with SystemVerilog that is widely accepted in industry. It takes an object-oriented approach involving drivers, monitors, stimulus generators, and scoreboards. The methodology is sufficiently complex that it is primarily used by verification specialists and is beyond what can be covered in this chapter. *Getting Started with UVM: A Beginner's Guide* [Cooper13] is a reasonable place to start learning UVM.

SystemVerilog has become the standard language for verification. The language is complex, and engineers use different subsets of the language for different purposes, as summarized in [Table 5.1](#).

[Section 5.1](#) begins by discussing verification strategies, from exhaustive and ad-hoc strategies that might be suitable for a class project or initial prototype to coverage-driven verification that systematically identifies what features must be covered, then generates directed and random tests to cover these features. Basic tests may run for minutes to hours, and comprehensive testing can take years, so tests are often run in hardware on a field programmable gate array (FPGA) emulator that is orders of magnitude faster than a software simulator. Technology Readiness Levels (TRLs), originally developed by the National Aeronautical and Space Administration (NASA) to manage multibillion dollar space programs, are a way of characterizing the maturity of design verification. Design and verification go hand in hand. Good logic designers test as they go and keep basic tests always passing rather than throwing untested RTL over the wall to a verification engineer.

A *design verification plan* (*DV plan*) is a structured document defining the coverage requirements, as outlined in [Section 5.2](#). It must cover both the trees and the forest: it must have enough detail that a verification engineer can translate each requirement into a machine-readable test but also be organized in a way that the reader can be confident that all the system's functions are covered.

Designers commonly run a *lint* tool, described in [Section 5.3](#), to detect static code problems such as mismatched port sizes. It is much faster to fix these problems immediately than to wait for one to cause a subtle bug and then track down the root cause by studying waveforms. [Section 5.4](#)

Table 5.1 SystemVerilog language subsets.

Subset	Section	Purpose
Synthesizable	4.2	Register transfer level (RTL) design
Testbench	5.4	Initialize memory and read tests from files, check results, log discrepancies
Assertions (SVA)	5.7	Specify properties a design must satisfy
Functional coverage (FC)	5.9	Coverpoints and covergroups to measure how thoroughly RTL is verified
Object-oriented	n/a	Generic reusable testbenches, particularly with the Unified Verification Methodology (UVM) library; also used in functional coverage

introduces a simple testbench, and [Section 5.5](#) shows how to simulate the testbench using the open-source Verilator simulator as well as the commercial Questa and VCS simulators.

Some verification methodologies generate a *memory signature* where key results are written to memory and checked against expectations. This approach can miss bugs that do not corrupt the signature. A more robust approach, described in [Section 5.6](#), is *lock-step-compare*, in which the entire architectural state of the system is compared against an independent reference model after each step.

[Section 5.7](#) describes *assertions*, which check properties, such as whether a configuration is legal or whether a bus protocol is violated. Assertions can be a fast way to identify problems where they occur in the RTL rather than waiting for the problem to corrupt the architectural state.

Code coverage, discussed in [Section 5.8](#), measures whether the simulations fully exercise every part of every line of code in the RTL. This is necessary but not sufficient. For example, code coverage would not detect that a feature is entirely absent from the code. *Functional coverage*, discussed in [Section 5.9](#), tests that all features are exercised. The DV plan is translated into a large number of *coverpoints*, such as exercising each instruction with each possible register and corner case values. The coverpoints are organized into *covergroups*, such as one group for each instruction. When the DV plan has been carefully reviewed and converted to coverpoints and enough test vectors have been simulated to reach 100% code and functional coverage while completely agreeing with the reference model, it is reasonable to declare that the system is adequately verified.

It would be even better if one could mathematically prove that the RTL correctly implemented a specification, such as the Sail architectural model. Such *formal verification (FV)* has long been a subject of research and is gradually becoming practical for an increasing set of use cases, as briefly described in [Section 5.10](#).

[Section 5.11](#) concludes with details of Wally's verification. In the context of RISC-V processor verification, the DUT is the SystemVerilog RTL model of the processor, the test vectors are RISC-V programs exercising various features, and the reference model is Sail, Spike, or ImperasDV. The architectural state of the RTL model is compared to the lock-step-compare reference model or the functional coverage monitors through a standard protocol called the *RISC-V Verification Interface (RVVI)*. Test vectors include the riscv-arch-test compatibility test suite, custom directed tests, and constrained random tests.

5.1 VERIFICATION STRATEGIES

A testbench loads and applies *test vectors* to a DUT and checks the results. Creating good test vectors and a reference model of expected results is a key challenge of logic verification. Verification strategies, discussed in

this chapter, must address the following questions: When is it best to test each unit individually and then assemble components and test the whole system, and when is the overhead of the unit testbench high enough that it is more expedient to skip unit testing? Simulations run slowly but give complete visibility into the design to locate bugs. Hardware emulation runs orders of magnitude faster but may have restricted visibility. Design and verification are an iterative process, steadily improving over the lifetime of the system. How does one characterize the maturity of the system at each phase? How do design and verification engineers collaborate to minimize the overall development effort?

5.1.1 Test Vector Generation

If students were asked to verify a simple module with only a few inputs and outputs on a homework assignment, they would likely create test vectors listing every possible input and the expected output. For example, a full adder has three inputs and two outputs, so it could be fully tested with $2^3 = 8$ vectors. Unfortunately, such *exhaustive testing* becomes intractable even for moderately interesting blocks such as an adder with two 64-bit inputs, which requires 2^{128} test vectors.

The next level of sophistication is ad-hoc testing using a modest set of test vectors informed by an understanding of the design. The RISC-V test program from [Code Example 2.16](#) is another example of *ad-hoc* or *exploratory* test vectors. The program was crafted by hand to test each instruction in the implemented instruction set, to verify that both taken and untaken branches work, and to check that a value could be stored to memory and correctly loaded back. Ad-hoc testing is a good way to quickly get basic confidence that a system performs its key functions. However, it does not give high confidence that the system works for the complete range of inputs.

Booting Linux is another readily available ad-hoc test that exercises some of the complicated interactions between instructions, exceptions, caches, and virtual memory. However, booting Linux is no substitute for rigorously testing each feature individually [Thompson23].

Directed testing attempts to systematically cover *corner cases*. In this context, corner cases are inputs that would be most likely to trigger unusual behavior. For example, for a 64-bit adder, corner cases for each input could include $\{0, 1, 2, 3, 2^{32}, 2^{63}-1, 2^{63}, 2^{63}+1, 2^{64}-2, 2^{64}-1, \text{ and two random values in the middle of the range}\}$. The adder can be tested with the *cross-product* of these 12 corner cases, that is, where every combination of the corners is used as the two inputs. If the adder works correctly for all these $12 \times 12 = 144$ corner cases, it is likely to work for intermediate values as well unless the system has unusually subtle design bugs.

Intel's infamous Pentium floating-point divide bug (FDIV) in 1994 was caused by a bad tool that optimized out 5 of 1,066 minterms from

Corner cases are not always enough. Imagine a carry lookahead adder coded by hand. It would be easy to introduce a typo in one of the columns that works for these corner cases but does not work for all cases. Similarly, a manufacturing defect in one of the transistors that causes column 12 of the adder to depend only on input A but not input B would go undetected by these cases.

a quotient digit selection programmable logic array (PLA), resulting in an error that occurred around the 9th or 10th decimal digit in about 1 division of every 9 billion for randomly selected inputs [Halfhill95]. Directed tests would be unlikely to catch such a subtle error. Moreover, testing was performed on the quotient selection equations, which were actually correct, rather than the transistor-level output of the PLA tool, which had the missing minterms. Intel initially dismissed the bug as unimportant, affecting an average spreadsheet user once every 27,000 years. However, Intel's leading customer, IBM, concluded the bug could occur once every 24 days in more demanding financial spreadsheets, which caused Intel to stop shipping Pentium-based computers. Intel recalled the defective processors at a cost of \$475 million and replaced them with corrected chips.

Directed tests work well for some modules, but the cross-product can still be too large for modules with many inputs, and determining sensible corners can be difficult for other modules. *Coverage-driven testing* is a technique that defines *coverage points* (also called *coverpoints*) that need to be tested. For example, a processor might define basic coverage points to be every source and destination register, every instruction opcode, and extreme values (0, 1, -1 , max positive, max negative, and so on). The riscv-arch-test suite satisfies coverage points for each instruction. It writes a signature to show that the test worked.

A *constrained random test generator* produces test vectors, consisting of random instructions. Constrained random test generation is particularly good at hitting hard-to-find cases, which are also called *corner cases* in a somewhat different usage of the term. For example, it is difficult to write a directed test that causes an interrupt while the instruction fetch unit is stalled waiting for the load/store unit to service a translation lookaside buffer miss. But a constrained random test generator that produces millions or billions of code sequences exercising interrupts and virtual memory may stumble across an instruction sequence that hits this corner case. Moreover, experience shows that the constrained random test generator often unearths bugs in areas unforeseen by the DV plan. How do we know if we have generated enough code sequences? The DV plan specifies code and functional coverage requirements, and we generate random tests until coverage reaches 100% or it becomes easier to write some directed tests for coverpoints that the random generator is not reaching.

In practice, verification engineers use a mix of all these techniques. For example, Wally was tested with simple ad-hoc programs early in development to get basic functionality. Then it was tested with riscv-arch-test that involves constrained random tests for a set of coverage points. At the time of this writing, riscv-arch-test doesn't cover much of the privileged specification, so we also write ad-hoc and directed tests for these features.

5.1.2 Reference Models

Creating a reference model for a new system can be an enormous task. Although it does not need to be cycle accurate and handle pipelining or out-of-order execution, it still must faithfully implement every feature of the architecture. Moreover, an architecture sometimes has unspecified behavior; for example, RISC-V highly recommends but does not require an illegal-instruction fault on an undefined opcode. The reference model and the DUT must produce the same results in these situations. The reference model and RTL should be written by different teams to reduce the chance that both models contain the same conceptual or implementation errors.

RISC-V is fortunate to be mature enough to have multiple reference models, notably Sail, Spike, and ImperasDV. Sail is the official executable reference model and now takes priority over the written architecture specification in the event of disagreement. However, Sail does not yet have sufficient support for interrupts and related peripherals to be an adequate reference for certain privileged behavior. Spike is a more mature simulator with interrupt support and is the reference for some OpenHW Group cores. ImperasDV is a commercial reference model, widely used in industry.

Other RISC-V simulators include QEMU, Whisper, and IDL.

5.1.3 Unit vs. System Testing

Development teams must weigh the pros and cons of unit versus system testing. In a hierarchical development approach, different teams could develop different units, such as the integer execution unit (IEU), floating-point unit (FPU), instruction fetch unit (IFU), and load/store unit (LSU). Each unit could be verified in isolation with unit-level testing. Then the units could be integrated and verified with system-level testing. Ideally, the individual units would already be correct, so system-level testing would only catch integration bugs. Moreover, testing a unit is faster than testing the system because it involves fewer lines of code and fewer tests irrelevant to the unit.

Unfortunately, testbench development is quite time-consuming. Creating test vectors tailored to unit interfaces can also be difficult. For example, it is much easier to load a program into memory and run a system-level test on an entire processor than to determine all of the control, data, and stall signals going into and coming out of an LSU.

Consequently, teams tend to emphasize testing relatively large blocks with simple interfaces. In an SoC, tests may focus on each different core or accelerator and on overall SoC integration, rather than on units within a core.

5.1.4 Emulation

An SoC with thousands to millions of lines of RTL generally simulates with a system clock of several kilohertz or even single hertz. Running large

simulations is painful, especially if these simulations are repeated after each design change. Simulations are accelerated by running many jobs in parallel on different cores, but this takes an expensive compute farm and even more expensive simulator licenses. In contrast, if the RTL is synthesized onto one or more FPGAs, it can run at megahertz rates. Such *hardware emulation* can accelerate verification by orders of magnitude.

Moderately complicated systems fit on one large FPGA that costs thousands or tens of thousands of dollars. FPGA-based emulation involves a moderate amount of infrastructure development to apply and check test vectors via the chip interface. The design may take minutes to hours to synthesize, place and route, and download onto the FPGA. Tests that require days in simulation may finish in seconds on the FPGA. However, if the outputs do not match expectation, it may take hours to isolate the bug using a clunky on-chip logic analyzer interface.

Commercial SoCs are usually too big to emulate on a single FPGA. They often use commercial emulation systems, such as Cadence Palladium or Synopsys ZeBu, containing racks of FPGAs, RAM, and input/output (I/O) to handle hundreds of billions of logic gates. Associated software automatically partitions the RTL onto many FPGAs and streamlines the debug process. These systems cost millions of dollars and are a precious resource for large-scale verification.

5.1.5 Technology Readiness Levels

NASA developed a TRL scale in the 1970s to discuss the level of technical maturity for space systems. The scale was adopted by the US Defense Department and European Space Agency and is now used more broadly in research and development outside space systems. Whereas academia usually develops prototypes in TRL 1–4, industry commercializes products in TRL 7–9. TRLs 4–7 are sometimes called the *Valley of Death*, where neither researchers nor industry prioritizes funding.

The OpenHW Group uses the TRL scale to characterize microprocessors, as shown in [Table 5.2](#). Most cores aim for TRL5, comparable to new commercially licensed processor IP. The OpenHW Group aims for collaborations between industry and academia to bridge the Valley of Death.

OpenHW hosts many open-source cores, including Wally, officially called CORE-V Wally (cvw). Wally reached TRL3 by passing the riscv-arch-tests and booting Linux in simulation. It then reached TRL4 with synthesis results and an FPGA implementation booting Linux in the lab. To reach TRL5 (eventually), Wally will need to meet the following guidelines:

- ▶ Design validation plans for all features of the design under test
- ▶ 100% functional coverage of all items in the DV plan
- ▶ 100% code coverage (line, condition, branch, FSM)
- ▶ All tests passing in regression
- ▶ No open issues

CORE-V is a family of open-source RISC-V cores hosted by the OpenHW Foundation, including cva6, cvw, cve2, and cve4.

Table 5.2 Technology Readiness Levels (TRLs) for OpenHW cores. (Adapted and reprinted with permission from The OpenHW Foundation.)

TRL	OpenHW Utilization
1. Basic Principles Observed	OpenHW research projects may target TRL-1 as project output, e.g., to develop novel approaches to core or accelerator architecture.
2. Concept Formulation	- Core IP or accelerator development projects are typically initiated as TRL-2 concepts, identifying principles and applications of the IP. - The OpenHW Project Concept Gate output includes a TRL-2 description of the Core IP.
3. Proof of Concept	- Core IP or accelerator development projects will pass through TRL-3 as the (RTL) design completes. - Proof of concept is shown by core compilation and demonstration of basic operations (e.g., Linux booted, CoreMark results, hello-world).
4. Component Prototype	Core IP or accelerator projects will pass through TRL-4 as they produce preliminary PPA results (via synthesis scripts for FPGA or ASIC) and/or run preliminary application code, such as an accelerator running machine learning code.
5. Subsystem Designed and Tested	Core IP projects reach TRL-5 as they complete full verification. The OpenHW RTL Freeze checklist process verifies that the design is fully ready for industrial adoption.
6. Functional (Alpha) Prototype	OpenHW IP that is integrated into an MCU system or other device reaches TRL-6 as prototype silicon is fabricated and demonstrated on a development board or other platform.
7. Field Demonstration Prototype	OpenHW IP that is integrated into an MCU system or other device reaches TRL-7 as prototype silicon is fabricated, deployed, and demonstrated in the field.

5.1.6 Design and Verification Approaches

Back in the bad old days of the 1990s, design and verification were often rigidly separated. Designers wrote RTL, perhaps ran basic unit testing, and then threw the code “over the wall” to a verification team, often in a different country. The verification team would locate bugs and throw them back to the designer to fix. This process was slow because of the loose coupling between teams and tended to increase verification cost because the designer made decisions without awareness of how they would affect the verification effort. As verification cost came to exceed

design cost, teams adopted more collaborative approaches. Successful SoC projects now begin with the goal of verification in mind.

One verification strategy is to develop the test cases before or concurrently with the hardware description language (HDL) so that the HDL can be tested as soon as it is written. Designers use a suite of *regression* tests to make sure that new changes don't break previously working features. These tests may be a group of known inputs and expected outputs or, for a processor, a specified program with expected results. Designers may run an abbreviated regression test every time they change code and a longer regression nightly.

A pitfall strategy for inexperienced designers is to get HDL to pass test cases by trial and error rather than by designing logically from the specifications. Test cases are rarely completely comprehensive, and it is likely that undiscovered bugs remain when the HDL is just tweaked until the tests pass.

Verification strategies can be classified as *always-broken* or *always-working* [Keating07]. In the always-broken strategy, a team of designers builds individual units and debugs them using unit tests. Then they integrate the units into an entire system and run regression on the system. Inevitably, upon integration, difficult bugs surface because of interdependencies and take a long time to debug. During this time, the design is broken, and it is difficult for others to continue development because they lack a functional prototype to improve. During integration, while the system is broken, the unit owners individually add new features to units and run unit testing but not system testing. Eventually, when the system works briefly, the painful integration process repeats. Overall regression is always broken except during brief intervals after successful integration steps.

In the always-working strategy, the team goes through the difficult integration just once, preferably on as small a set of features as possible, and gets those features working. Then team members incrementally add new features and perform system-level regression before checking the features back into the design to ensure the changes don't break existing features. Overall, regression is always working after the initial prototype is developed. New features and tests are gradually added after the enhanced system is debugged by individual contributors. When a proposed feature breaks regression, it is relatively easy to fix because the designer knows exactly what change caused the problem. In the real world, this is sometimes impractical, due to adding a very large new feature, restructuring the design or test cases, or because of license, hardware, or software problems that temporarily stall regression. However, when these problems arise, the team should prioritize getting back to a working code base as quickly as possible.

Test-driven development [Beck03] is a development process in which the test cases are written before the code. It can be viewed as a disciplined form of always-working verification. The general process is:

1. Add a new test for added capability.
2. Run all the tests and confirm the new one fails.
3. Make a code change to implement the added capability.
4. Run all the tests and confirm the new one succeeds.
5. Refactor the code to remove duplication.

Tracking down a complex bug in a large RTL system through simulation is so time consuming and painful that developers strive to minimize doing this. Wally has gravitated toward an always-working test-driven methodology. Whenever possible, small changes are made and immediately tested to confirm that they work and don't break other features. If a change introduces a bug, it may be faster to discard the change and re-create it carefully in smaller steps than to diagnose the bug in a waveform viewer. The developer makes the changes in a development (dev) branch of their fork of the repository, then issues a pull request to incorporate the changes into the main branch when the feature is complete and tested.

5.2 DESIGN VERIFICATION PLAN

A design verification plan (DV plan or test plan) needs to be organized in a way that human reviewers can determine if all features of the system are tested and in a way that the verification engineer can write tests to cover each feature and measure whether each feature was covered. This generally involves a high-level statement enumerating the major areas (e.g., base instruction set, unprivileged extensions, privileged instructions, privileged extensions), pointing to spreadsheets that list each feature and subfeature. For example, a spreadsheet for the M extension could contain a feature for each instruction. Each feature would contain subfeatures, such as the following:

Division instructions have additional subfeatures for specific corner cases, such as division by 0 and overflow.

- ▶ Using all registers for each source and destination
- ▶ Using all combinations where source and destination register match
- ▶ Corner cases of inputs: (0, 1, 2, random number, max positive, max negative, -1)

The spreadsheet should contain the following columns for each feature and subfeature.

- ▶ A reference to a specification

- ▶ A description of the feature to be verified
- ▶ A description of the required stimulus to exercise the feature
- ▶ A method of checking the correct operation of the feature
- ▶ The coverage specification that indicates that the feature was actually verified
- ▶ A reference to the coverage specified in the previous point

See the Wally DV plan in [Section 5.11.1](#) for more examples.

5.3 LINT

Debugging a large design is time-consuming. Engineers save a lot of time by detecting problems during static code analysis (compilation) rather than through simulation. HDL simulators such as Questa indicate some errors and warnings that should always be corrected, but dedicated code analysis tools called *linters* are more comprehensive and faster. Some of the types of warnings flagged by lint tools include:

- ▶ Unsynthesizable logic
- ▶ Unintended latches and combinational loops
- ▶ Undeclared, unused, and undriven signals
- ▶ Mismatched port sizes
- ▶ Always and case statement gotchas such as missing defaults
- ▶ Poor coding style
- ▶ Deprecated Verilog constructs
- ▶ Mismatched vector sizes in assignment

Many lint tools exist, with Ascent Lint from Real Intent and Synopsys SpyGlass Lint being current commercial market leaders. Design teams often produce customized lint rule sets reflecting their organization's coding standards. Wally uses Verilator lint because it is free and good enough to catch many bugs.

Lint tools produce spurious warnings on certain types of code. For example, Verilator warns about incomplete `case` statements without a default even if the value is set elsewhere in an `always_comb` block. To turn off a specific lint warning, place the following comment in the System-Verilog code. `CASEINCOMPLETE` is the name of the warning being waived.

```
// verilator lint_off CASEINCOMPLETE
```

It is wise to turn the warning back on after the code that would cause the warning:

```
// verilator lint_on CASEINCOMPLETE
```

A good designer keeps the design lint clean every time it is checked in to reduce bugs and prevent warnings that confuse other designers.

5.4 TESTBENCHES

Testbenches apply stimulus to a DUT, and preferably check that the results match expectations. SystemVerilog offers a vast set of nonsynthesizable features for writing testbenches, including generating clocks, reading files, and printing messages to the screen. An entire object-oriented portion of the language caters to software engineers writing large verification testbenches [IEEE 1800-2017], but this is beyond the scope of this chapter.

A simple self-checking SystemVerilog testbench performs the following functions, as illustrated in Fig. 5.1:

- ▶ Instantiate DUT.
- ▶ Generate clock and reset inputs.
- ▶ Read a set of *test vectors* (inputs and expected outputs) from a file.
- ▶ Apply the inputs on successive clock cycles and check the outputs.
- ▶ Log any mismatches.

Code Example 5.1 shows a testbench for the full adder from Code Example 4.1. The DUT has three inputs and two outputs. The testbench declares internal signals, including the clock, reset, inputs, outputs, and expected output values. It also has two variables to keep track of the current vector number and number of errors seen so far and an array of test

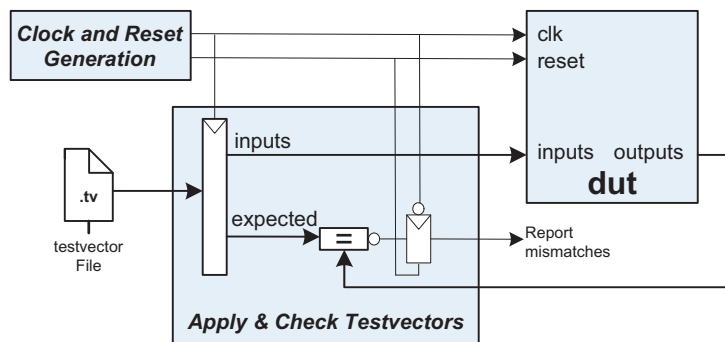


Fig. 5.1 Self-checking testbench.

Code Example 5.1 fulladder TESTBENCH

```

module testbench();
    logic      clk, reset;
    logic      a, b, c, s, cout, sexpected, coutexpected;
    logic [31:0] vectornum, errors;
    logic [4:0] testvectors[10000:0];

    // Instantiate device under test
    fulladder dut(a, b, c, s, cout);

    // Generate clock
    always
    begin
        clk = 1; #5; clk = 0; #5;
    end

    // At start of test, load vectors and pulse reset
    initial
    begin
        $dumpfile("fulladder.vcd");
        $dumpvars;
        $readmemb("fulladder.tv", testvectors);
        vectornum = 0; errors = 0;
        reset = 1; #22; reset = 0;
    end

    // Apply test vectors on rising edge of clk
    always @(posedge clk)
    begin
        #1; {a, b, c, coutexpected, sexpected} = testvectors[vectornum];
    end

    // Check results on falling edge of clk
    always @(negedge clk)
    begin
        if (~reset) begin // Skip during reset
            if (s != sexpected | cout != coutexpected) begin // Check result
                $display("Error: inputs = %b", {a, b, c});
                $display("  outputs cout s = %b%b (%b%b expected)", cout, s, coutexpected, sexpected);
                errors = errors + 1;
            end
            vectornum = vectornum + 1;
            if (testvectors[vectornum] === 'bx) begin
                $display("%d tests completed with %d errors",
                    vectornum, errors);
                $stop;
            end
        end
    end
endmodule

```

vectors. The array is 5 bits wide to accommodate the inputs and outputs and 10,001 entries deep so it has plenty of space even if one modifies the file to use many more vectors. The `always` block uses delays to produce a clock continuously oscillating every 10 ticks. At the start of simulation,

the testbench reads the test vectors from `fulladder.tv`, interpreting them as binary numbers. It asserts `reset` for slightly more than 2 cycles (22 simulation ticks). It then applies test vectors just after each rising clock edge and checks the outputs on the falling clock edge when `reset` is not asserted. If either output disagrees with expectations, it reports the failing vector and increments the error count. In any case, it then advances to the next test vector. If that vector is all `x`, then the testbench has completed all of the vectors in the file, and the simulation should stop.

Observe that the simulation uses several new commands. The `$dumpfile` and `$dumpvars` write a Verilog Change Dump (.vcd) file used in Section 5.5.2 to view waveforms after a Verilator simulation; they are unnecessary for other simulators. `$readmemb` reads a text file in binary format into a memory array. `$readmemh` does the same in hexadecimal format. `$display` writes to the console. `$stop` halts a simulation. `initial` statements are run once at the start of a simulation. `always` statements without a sensitivity list are repeated forever. Testbenches use `===` and `!==` instead of `==` and `!=` for equality and inequality comparison to work properly on signals that might include `x` or `z`.

Code Example 5.2 shows the test vector file with an exhaustive set of vectors. The file lists the three inputs and two outputs. The optional underscores are included for readability but are ignored by the testbench. The file can have comments starting with `//`.

Code Example 5.2 `fulladder.tv`

```
// abc_cout s
000_00
001_01
010_01
011_10
100_01
101_10
110_10
111_11
```

Questa prefers `$stop` when the testbench is done applying vectors; `$finish` terminates without an opportunity to generate coverage statistics or inspect waveforms. However, Verilator prefers to halt the testbench with `$finish`.

A testbench may halt with `$stop` or `$finish`. `$stop` suspends the simulation. This is useful in a graphic user interface environment where the user may want to inspect the waveforms for troubleshooting. `$finish(errcode)` terminates the simulation and prints a diagnostic message according to Table 5.3. If `errcode` is omitted, it defaults to 1.

Table 5.3 `$finish` codes.

errcode	Diagnostic Message
0	None
1	Simulation time and location of <code>\$finish</code>
2	Simulation time, location of <code>\$finish</code> , and memory statistics

5.5 LOGIC SIMULATORS

A logic simulator applies test vectors to a model of a system and computes the outputs. We use SystemVerilog to describe both the system model and input generation. Our system model is coded with synthesizable Verilog because it describes hardware, whereas the test vectors are generated by a nonsynthesizable testbench. Traditional *event-driven logic simulation* maintains a list of signal changes called *events* that happen at a particular time. Each event causes the logic that depends on the signal to be reevaluated. For example, when the code on the right-hand side of an `assign` statement or a `<=` or `=` in an `always_comb` block changes, the signal on the left-hand side is recomputed. If the left-hand signal changes, this creates a new event that gets added to the list. If the assignment has a delay, the new event will be added after that delay. Otherwise, the new event will be added after an infinitesimal *delta* delay that appears to be zero in simulation yet keeps events ordered to reflect cause and effect. Older simulators are interpreted, but most modern simulators compile the Verilog into a faster executable form.

Leading commercial HDL simulators include Questa from Siemens (formerly called ModelSim, from Mentor Graphics), Cadence Xcelium, and Synopsys VCS. All are mature products that support SystemVerilog, VHDL, and SystemC. Verilator is a leading free open-source Verilog/SystemVerilog simulator.

5.5.1 Questa

Simulate the `fulladder` testbench in the Questa GUI using:

```
$ cd $WALLY/examples/verilog/fulladder
$ vsim fulladder.sv
```

Choose Compile → Compile... and select `fulladder.sv` to compile. Then choose Simulate → Start Simulation..., expand the `work` library, and select `testbench`. Click on Optimization Options and change to “Apply full visibility to all modules (full debug mode)” so that Questa doesn’t optimize away the internal signals; then click OK.

In the Objects pane, select all of the signals. Choose Add → To Wave or drag them to the waveform window to add them.

Type `run 100` or `run all` to run the simulation. `run 100` runs the simulation for 100 simulation ticks (10 clock cycles), which is enough to run all 8 test vectors for this example. The testbench from [Code Example 5.1](#) generates a clock to sequence the inputs, and pulses `reset` for two cycles at the start. It then applies all 8 possible combinations of `a`, `b`, and `c`, and checks

If you are a heavy Questa user, be sure to download the manuals, which describe many features beyond what there is space to cover here. The Questa SIM User’s Manual is a good place to start.

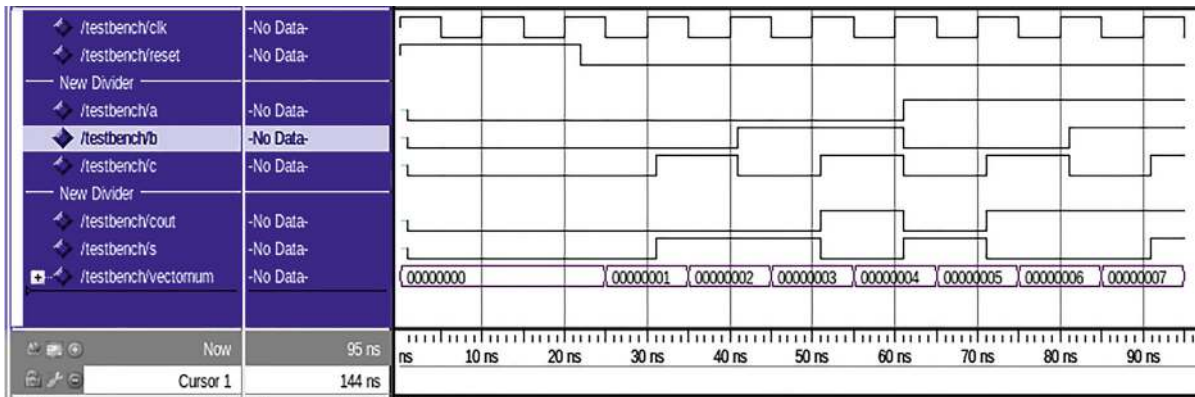


Fig. 5.2 fulladder simulation waveforms in Questa.

that `s` and `cout` are correct. You will see the following report and waveforms resembling Fig. 5.2. You can type `quit` to terminate Questa.

```
VSIM > run 100
#      8 tests completed with          0 errors
# ** Note: $stop : /home/harris/cvw/examples/verilog/fulladder/fulladder.sv(42)
#      Time: 95 ns Iteration: 1 Instance: /testbench
# Break at /home/harris/cvw/examples/verilog/fulladder/fulladder.sv line 42
VSIM2 > quit
```

If you need to fix a bug, reopening Questa and repeating all these clicks becomes tedious. You can recompile with the compile menu. Then repeat to simulation with:

```
VSIM > restart -f # Restarts the simulation without losing your waveforms
VSIM > run 100
```

Better yet, you can create a script called a *do file* that compiles and simulates for you. Code Example 5.3 shows a *do file* for the full adder. First, it uses the `vlog`, `vopt`, and `vsim` commands to compile, optimize, and start the simulation. By default, the compiled design is placed in a directory called `work`. The *do file* adds all the interface signals to the waveform window and then runs until the simulation completes.

```
$ vsim -do fulladder.do
```

Code Example 5.3 `fulladder.do`

```
# fulladder.do

# Compile, optimize, and start the simulation
vlog fulladder.sv
vopt +acc work.testbench -o workopt
vsim workopt

# Add waveforms and run the simulation
add wave *
run -all
view wave
```

If you don't need the GUI to debug, you can run in batch mode, which is faster. [Code Example 5.4](#) shows a batch mode script that doesn't add waveforms and quits when the simulation is complete. This is handy for regressions, which run many simulations and report whether they passed. Note the `-c` flag to `vsim` to invoke a batch mode simulation.

```
$ vsim -c -do fulladder-batch.do
```

Code Example 5.4 `fulladder-batch.do`

```
# fulladder-batch.do
vlog fulladder.sv
vopt +acc work.testbench -o workopt
vsim workopt
run -all
quit
```

To avoid having to remember the `vsim` command and its arguments, we can write a 1-line script containing the command. Name it `sim-fulladder-batch`, change its permissions to be executable (`chmod 755`), and invoke the script with:

```
$ ./sim-fulladder-batch
```

If the simulation is simple, you can use the Questa `qrun` command to automate all of this batch simulation instead of writing a do file. `qrun` identifies the top-level testbench based on the module hierarchy and compiles, optimizes, and simulates the file.

```
$ qrun fulladder.sv
```

If you don't have an X2Go or VNC connection to your server, you can be productive with command line tools using an `ssh -Y` connection. Batch simulations work great from the command line. The Questa GUI will open over `ssh -Y` but is painfully slow to use.

5.5.2 Verilator

Verilator is an open-source SystemVerilog simulator. Verilator converts the SystemVerilog code into a C program that simulates its behavior, then compiles and runs the C program. Because it is open source, Verilator is a key component of open-source hardware design. Verilator is gaining traction at companies as well because simulator licenses are expensive and because Verilator can be as fast or faster than commercial simulators.

Verilator was conceived by Paul Wasson at Digital Equipment Corporation in 1994 and was released as open source in 1998. Wilson Snyder has been the lead developer since 2001. Initially, it had many limitations, such as handling only a subset of Verilog and only supporting testbenches written in SystemC. During the time this book was in development, Verilator became capable of simulating SystemVerilog testbenches and all of Wally's features, and many of Verilator's functional and performance bugs were fixed. Verilator can run all of the regression tests on the Wally testbench. Verilator does not yet support UVM (see [Table 5.1](#)), but work is in progress.

Verilator involves three steps: compiling the Verilog into an executable file, running the executable to simulate the testbench, and optionally viewing waveforms. Invoke these steps with the following:

```
$ cd $WALLY/examples/verilog/fulladder
$ verilator --binary --top-module testbench --trace fulladder.sv
$ obj_dir/Vtestbench
```

Verilator takes a list of Verilog files to compile (`fulladder.sv`). The `--top-module` argument usually indicates the testbench. The `--binary` argument asks Verilog to generate C code to model the behavior of Verilog and compile it into a binary executable file. The optional `--trace` argument enables generating a `.vcd` (Verilog Change Dump) file for a waveform viewer when the testbench invokes `$dumpvars`. Verilator creates `obj_dir` containing the executable (`Vtestbench`) along with the C code and supporting files.

Alternatively, a script invokes both of these commands, so you don't need to remember the arguments. See the output printed below, showing Verilator compile the files and run the simulation. The testbench prints the vectors with the inputs and expected outputs. The simulation also records the signals to `fulladder.vcd`.

```

$ ./verilate
make: Entering directory '/home/harris/cvw/examples/verilog/fulladder/obj_dir'
ccache g++ -O3 -I. -MMD -I/usr/local/share/verilator/include -
I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=1 -
DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-
shadow -Wno-sign-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-
parameter -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused-variable -
DVL_TIME_CONTEXT -fcoroutines -c -o verilated.o
/usr/local/share/verilator/include/verilated.cpp
...
ccache g++ -O3 -I. -MMD -I/usr/local/share/verilator/include -
I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=1 -
DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-
shadow -Wno-sign-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-
parameter -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused-variable -
DVL_TIME_CONTEXT -fcoroutines -c -o Vtestbench__ALL.o Vtestbench__ALL.cpp
echo "" > Vtestbench__ALL.verilator_deplist.tmp
g++ -fuse-ld=mold verilated.o verilated_vcd_c.o verilated_timing.o verilated_threads.o
Vtestbench__ALL.a -pthread -lpthread -latomic -o Vtestbench
rm Vtestbench__ALL.verilator_deplist.tmp
make: Leaving directory '/home/harris/cvw/examples/verilog/fulladder/obj_dir'

- V e r i l a t i o n   R e p o r t: Verilator 5.025 devel rev v5.024-12-ga9664b8d1
- Verilator: Built from 0.013 MB sources in 3 modules, into 0.049 MB in 11 C++ files needing
0.000 MB
- Verilator: Walltime 4.949 s (elab=0.000, cvt=0.212, bld=4.734); cpu 0.000 s on 1 threads;
allocated 19.750 MB
cycle: 00000001 vectornum 00000000 testvectors[vectornum]: 00000
cycle: 00000002 vectornum 00000000 testvectors[vectornum]: 00000
cycle: 00000003 vectornum 00000001 testvectors[vectornum]: 00101
cycle: 00000004 vectornum 00000002 testvectors[vectornum]: 01001
cycle: 00000005 vectornum 00000003 testvectors[vectornum]: 01110
cycle: 00000006 vectornum 00000004 testvectors[vectornum]: 10001
cycle: 00000007 vectornum 00000005 testvectors[vectornum]: 10110
cycle: 00000008 vectornum 00000006 testvectors[vectornum]: 11010
cycle: 00000009 vectornum 00000007 testvectors[vectornum]: 11111
cycle: 0000000a vectornum 00000008 testvectors[vectornum]: 00000
cycle: 0000000b vectornum 00000009 testvectors[vectornum]: 00000
10 tests completed with 0 errors
- fulladder.sv:49: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.025 devel
- Verilator: $finish at 120ps; walltime 0.001 s; speed 0.000 s/s
- Verilator: cpu 0.000 s on 1 threads; allocated 249 MB

```

The waveforms recorded in the VCD file can be displayed with commercial or open-source viewers. For example, invoke the open-source GTKWave viewer using:

```
$ gtkwave -f fulladder.vcd
```

Click into the TOP -> testbench hierarchy to see a list of internal signals; then click to add them to the waveform viewer, as shown in [Fig. 5.3](#).

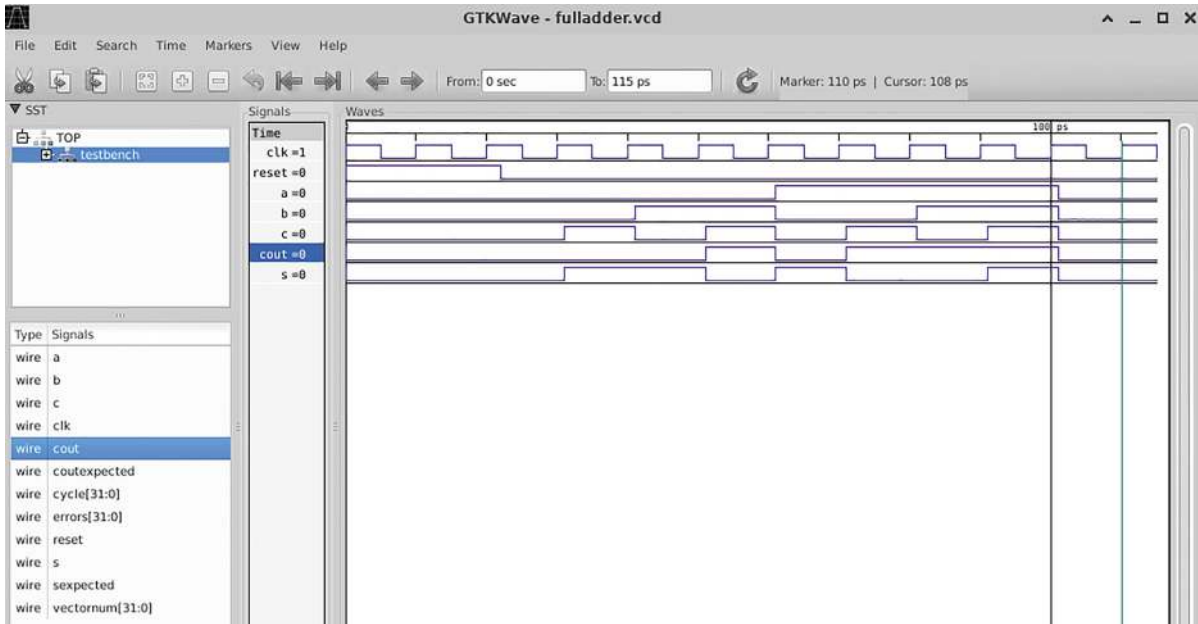


Fig. 5.3 fulladder simulation waveforms in GTKWave.

5.5.3 VCS

Synopsys VCS is another commercial HDL simulator. Like Verilator, it compiles HDL code into an executable, which is then run. By default, the executable is named `simv`. Invoke VCS with the following:

```
$ cd $WALLY/examples/verilog/fulladder
$ vcs -sverilog -full64 fulladder.sv
$ ./simv
```

5.6 LOCK-STEP-COMPARE

Signature checking has several limitations in verification. A program typically produces one signature word after a sequence of instructions, and that might not capture all design errors. For example, an early version of riscv-arch-test floating-point tests created a signature with the result of the instruction but not the floating-point flags, thus missing bugs in the flag logic. Moreover, a system test such as running a benchmark or booting Linux doesn't produce a signature or only indicates success (e.g., a login prompt) after millions of instructions. A more robust verification methodology is *lock-step-compare*, in which a testbench logs all the

architectural state that changes when an instruction completes and compares against a reference model, reporting every discrepancy. The state includes the program counter (PC), integer and floating-point registers, control and status registers (CSRs), and memory. In this way, deviations are discovered immediately rather than possibly being hidden or causing a subtle failure much later. A key requirement of operation is to inject the same interrupts into the reference model as the DUT, so they remain in *lockstep*, following the same program counter sequence and producing exactly the same architectural state after each instruction. Another challenge is that the RISC-V specification is both configurable and ambiguous in many places, so the reference model must be configured to exactly match the behavior of the DUT when multiple behaviors are legal. The reference model must not only agree on XLEN and supported extensions but also on subtle behavior such as whether the `UXL` bits of the `mstatus` register are writable, such that user-mode XLEN can be changed at runtime.

ImperasDV is a widely used commercial lock-step-compare tool. It requires a tracer in the testbench to log the changing state using the RVVI-TRACE standard interface [RVVI-TRACE]. The DUT runs programs in the usual fashion, while the Imperas reference model runs the same programs, takes the same interrupts, reads the same values from memory and peripherals, and reports tracer discrepancies, as shown in Fig. 5.4 (github.com/riscv-verification/riscvISACOV/blob/v20231026/source/coverage/RV32I_coverage.svh). Synopsys acquired Imperas in 2024.

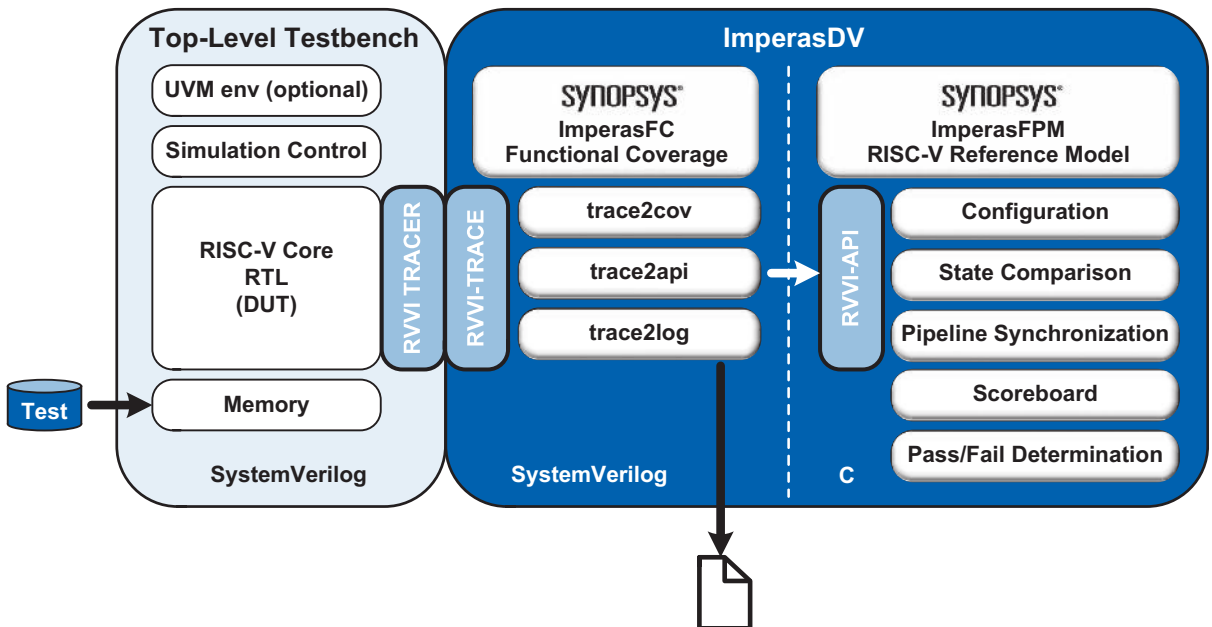


Fig. 5.4 ImperasDV lock-step-compare system. (Figure adapted and reprinted with permission, courtesy of Imperas.)

Table 5.4 RVVI-TRACE signals.

Signal	Size	Meaning
clk	1	System clock updating state
insn	32	Current instruction
trap	1	Trap rather than retiring instruction
pc_rdata	XLEN	PC of instruction
x_wdata	$XLEN \times 32$	New value(s) being written to integer register file
x_wb	32	Which integer registers are being written
f_wdata	$FLEN \times 32$	New value(s) being written to floating-point register file
f_wb	32	Which floating-point registers are being written
csr	$XLEN \times 4,096$	New value(s) being written to CSRs
csr_wb	4,096	Which CSRs are being written

5.6.1 RVVI-TRACE Interface

RVVI-TRACE defines a SystemVerilog interface, capturing the state changes that occur during each instruction. Wally uses RVVI both to communicate with ImperasDV during lockstep simulation and to provide the state to riscvISACOV functional coverage. Key RVVI fields are listed in Table 5.4.

`testbench/common/wallyTracer.sv` sends register changes (register files and CSRs) from the Writeback stage to the RVVI-TRACE interface. It also pipelines other signals (such as `PCF`, `InstrF`, and `TrapM`) into the Writeback stage, accounting for stalls and flushes, and sends them to RVVI-TRACE.

5.7 ASSERTIONS

SystemVerilog Assertions (SVAs) check that the RTL conforms to a specification. They warn of trouble long before it propagates to the architectural state. Thus, they speed debug by pinpointing the root cause of a problem. Some of the most common places for assertions are in FIFOs (first in, first out queues), bus interfaces, and state machines. Assertions are also used in functional coverage and formal verification to specify properties that must be checked. Assertions are classified as *immediate* and *concurrent*, as discussed briefly next. See the SystemVerilog specification for syntax nuances or [Mehta20] for an in-depth tutorial about using SVA for verification.

5.7.1 Immediate Assertions

Immediate assertions test an expression at the time they are executed. They are placed in `initial` or `always` (`always_comb`/`always_ff`) blocks, which determines when they execute. Assertions are written so that the expression is `TRUE` unless there is a defect in the design. The syntax of an assertion is as follows:

```
assert (expression) <statement1> <else statement2>
```

If expression evaluates to `TRUE` (1), `statement1` is executed; otherwise, expression must be 0, `x`, or `z`, and `statement2` is executed. If `statement1` is omitted, the assertion is silent when the expression is `TRUE`. If `else statement2` is omitted, the assertion calls `$error` when expression is not `TRUE`.

The most common statements print messages associated with levels of severity.

- ▶ `$fatal(errcode, message)`
- ▶ `$error(message)`
- ▶ `$warning(message)`
- ▶ `$info(message)`

Each of these statements prints the message. `$fatal` also terminates the simulation by calling `$finish(errcode)`, with the codes described in [Table 5.3](#). Assertions typically use `$error` to report a violation.

[Code Example 5.5](#) gives examples of immediate assertions in the `riscvassertions` module, which is part of the Wally testbench. The module contains many assertions to ensure the configuration parameters describe legal architectural and microarchitectural choices. The assertion statements are in an `initial` block, so they are checked just once at the beginning of simulation. For example, the RISC-V architecture specification allows 0, 16, or 64 entries in the Physical Memory Protection (PMP) checker. Virtual memory is only available on systems with supervisor mode because it depends on the supervisor-mode `satp` CSR, discussed further in [Chapter 11](#). The Wally load/store unit (LSU) microarchitecture reads a maximum of `XLEN` bits from the bus or tightly integrated memory but can read longer lines from a data cache. Thus, for example, Wally RV32 only supports `FLEN = 64`-bit double-precision floating-point if the cache is enabled. If any of these configuration parameters are unsupported, the simulation terminates with a fatal error and an explanation of the problem. Hence, the assertions enforce legal configurations as part of the code base rather than part of the documentation that a user might

lose or not read. For example, if the config file sets `PMP_ENTRIES` to 1, the assertion will terminate the simulation with:

```
# ** Fatal: Illegal number of PMP entries: PMP_ENTRIES must be 0, 16, or 64
```

Code Example 5.5 riscvassertions.sv

```
module riscvassertions import cvw::*; #(parameter cvw_t P);
  initial begin
    assert (P.PMP_ENTRIES == 0 || P.PMP_ENTRIES==16 || P.PMP_ENTRIES==64)
      else $fatal(1, "Illegal number of PMP entries: PMP_ENTRIES must be 0, 16, or 64");
    assert (P.S_SUPPORTED || P.VIRTMEM_SUPPORTED == 0)
      else $fatal(1, "Virtual memory requires S mode support");
    assert (P.DCACHE_SUPPORTED || ~P.F_SUPPORTED || P.FLEN <= P.XLEN)
      else $fatal(1, "Data cache required to support FLEN > XLEN because
AHB/DTIM bus width is XLEN");
    ...
  end
```

Sometimes the assertion may fire while `expression` is glitching, producing a faulty error during a transient invalid state that resolves to a good state. *Deferred immediate assertions* add the `final` keyword, which causes the assertion to wait until `expression` settles before evaluating.

```
assert final (expression) <statement1> <else statement2>
```

5.7.2 Concurrent Assertions

Concurrent assertions describe behavior that spans clock cycles. They are called concurrent because they execute in parallel with the rest of the design. They use a succinct and powerful notation suited to describe multicycle events in synchronous sequential systems. The SystemVerilog 1800-2017 specification devotes more than a hundred pages to concurrent assertions, but this section just touches on a simple example.

Concurrent assertions are expressed using *properties*, which in turn may depend on *sequences*. Properties use the *implication operator* `|=>` to imply a relationship between an *antecedent* and a *consequent*. If the antecedent is satisfied, the concurrent assertion checks the consequent and reports if it is violated.

For example, consider checking a property of a bus protocol. According to the AHB standard ([AHB21], [Section 3.7.1](#)), when the `HREADY` signal is low, indicating that a peripheral needs a wait state, the controller must not change the transfer type `HTRANS` from one bus cycle to the next, except:

- ▶ The controller can change `HTRANS` from `IDLE` to `NONSEQ`.

- ▶ If the controller was busy in the previous cycle of a burst transfer, HTRANS can go from BUSY to SEQ.
- ▶ During reset, the protocol does not apply.

Code Example 5.6 gives an example of a concurrent assertion to check this property. The assertion has the name `transaction_type_stability` used in reporting. The concurrent assertion is evaluated on the rising edge of the HCLK bus clock. It is disabled during bus reset. Otherwise, if HREADY is low (antecedent), one of three consequents must be true: HTRANS is stable (has the same value as the previous cycle), HTRANS was IDLE on the previous cycle, or HTRANS was BUSY on the previous cycle during a burst.

Code Example 5.6 `fulladder-batch-coverage.do`

```
// Example courtesy Umar Yaqoob and Bilal Zafar of 10xEngineers.
transaction_type_stability: assert property (
  @(posedge HCLK) disable iff (!HRESETn)
  !HREADY | => $stable(HTRANS) |
    ($past(HTRANS) == AHB_IDLE) |
    (($past(HTRANS) == AHB_BUSY) & (HBURST != 3'b000))
);
```

5.8 CODE COVERAGE

A logic simulator can collect and report coverage metrics for HDL code. A minimum level of coverage is that the test vectors should activate every line of code, called *line coverage*. A next level of coverage is to ensure every expression in a logic equation and every condition in a case or branch has been exercised, called *statement*, *expression*, *condition*, *FSM*, and *branch coverage*.

For example, Code Example 5.7 shows how to log code coverage into a Unified Coverage Database (UCDB) file using Questa. Compared to Code Example 5.4, the do file adds a `+cover=sbecf` to the `vopt` command to measure statement, branch, expression, condition, and FSM coverage. It runs `vsim` with the `-coverage` flag. Finally, it saves all the coverage information from `/testbench/dut` into `fulladder.ucdb`.

Code Example 5.7 `fulladder-batch-coverage.do`

```
vlog fulladder.sv
vopt +acc work.testbench -o workopt +cover=sbecf
vsim workopt -coverage
run -all
coverage save -instance /testbench/dut fulladder.ucdb
quit
```

`/testbench/dut` in [Code Example 5.7](#) refers to the fulladder module. Lines 50-51, containing the fulladder body, are listed in [Code Example 5.8](#).

Code Example 5.8 `fulladder.sv` BODY

```
50: assign s = a ^ b ^ c;
51: assign cout = (a & b) | (a & c) | (b & c);
```

Statement coverage counts whether each statement in the code has been executed. It is similar to *line coverage* but more specific because statements could span more than one line, and more than one statement could be placed on a line. *FSM coverage* checks that all states have been reached and all transitions between states have been exercised. *Branch coverage* checks if the Boolean expression in control structures (e.g., if, case) have evaluated to both TRUE and FALSE. Condition and expression coverage check that each input of a Boolean expression is exercised TRUE and FALSE while the other inputs are in a state that make the output depend on the input being exercised. *Condition coverage* applies to the condition in an if statement, and *expression coverage* applies to the right-hand side of assignment statements. For example, in the statement $y = a \& b \mid c$, y only depends on a when $b = 1$ and $c = 0$. Hence, the full expression coverage requires exercising six cases:

- ▶ $a = 0 / 1$ when $b = 1, c = 0$
- ▶ $b = 0 / 1$ when $a = 1, c = 0$
- ▶ $c = 0 / 1$ when a and b are not both 1

Invoke the simulation with the following `vsim` command, then use `vcover` to produce a detailed human-readable report from the UCDB:

```
$ vsim -c -do fulladder-batch-coverage.do
$ vcover report -details fulladder.ucdb > fulladder_coverage_details.rpt
```

The report in [Code Example 5.9](#) shows that the full adder has six expressions (e.g., $(a \& b)$, $(a \wedge b)$, etc.) and two statements (lines 50–51 assigning s and $cout$), as highlighted in blue. The simulation applied the `fulladder.tv` vectors from [Code Example 5.2](#) to exhaustively check all 8 combinations of a , b , and c , so the full adder’s expressions and statements are fully covered.

Code Example 5.9 fulladder-coverage_details.rpt

```

QuestaSim-64 vcover 2022.4_2 Coverage Utility 2022.12 Dec 9 2022
Start time: 10:58:48 on May 23,2023
vcover report -details fulladder.ucdb
Coverage Report by instance with details

=== Instance: /dut
=== Design Unit: work.fulladder
Expression Coverage:
  Enabled Coverage          Bins   Covered   Misses   Coverage
  -----
  Expressions              6      6        0    100.00%

=====Expression Details=====

Expression Coverage for instance /dut -

  File fulladder.sv
  -----Focused Expression View (Bimodal)-----
Line   50 Item    1 ((a ^ b) ^ c)
Expression totals: 3 of 3 input terms covered = 100.00%

  -----Focused Expression View-----
Line   51 Item    1 (((a & b) | (a & c)) | (b & c))
Expression totals: 3 of 3 input terms covered = 100.00%

Statement Coverage:
  Enabled Coverage          Bins    Hits    Misses   Coverage
  -----
  Statements              2      2        0    100.00%

=====Statement Details=====

Statement Coverage for instance /dut -

  Line      Item          Count    Source
  ----      -
  File fulladder.sv
  50          1            9
  51          1            9

Total Coverage By Instance (filtered view): 100.00%

```

To see an example of incomplete coverage, comment out the sixth and seventh lines of `fulladder.tv`, as shown in [Code Example 5.10](#). Now the tests no longer contain a vector that makes `cout = 1` due to `a` being 1 (the last row has `cout = 1`, but `b` and `c` are also 1 so `cout` doesn't depend on `a`). Rerun `vsim` and regenerate the coverage report, as shown in [Code Example 5.11](#). Now the `cout` logic at line 51 is incompletely covered. Specifically, the vectors no longer are sufficient to show that setting input `a = 1` affects `cout`. Row 2 of the Focused Expression View indicates that `a` is never 1 when `b`

and *c* are in a state that affects the output. The *nonmasking condition* is tricky to interpret in this report but essentially means when $b \wedge c$ is 1 such that *cout* is influenced by *a*. The fulladder shows 91.66% coverage overall.

Code Example 5.10 MODIFIED fulladder.tv

```
// abc_cout s
000_00
001_01
010_01
011_10
100_01
//101_10
//110_10
111_11
```

Code Example 5.11 fulladder-coverage_details.rpt (MISSING 101 AND 110 TEST CASES)

Coverage Report by instance with details

=== Instance: /dut
=== Design Unit: work.fulladder

Expression Coverage:

Enabled Coverage	Bins	Covered	Misses	Coverage
-----	----	----	----	-----
Expressions	6	5	1	83.33%

=====Expression Details=====

Expression Coverage for instance /dut -

File fulladder.sv

-----Focused Expression View (Bimodal)-----

Line 50 Item 1 ((a ^ b) ^ c)

Expression totals: 3 of 3 input terms covered = 100.00%

-----Focused Expression View-----

Line 51 Item 1 (((a & b) | (a & c)) | (b & c))

Expression totals: 2 of 3 input terms covered = 66.66%

Input Term	Covered	Reason for no coverage	Hint
-----	-----	-----	-----
a	N	'_1' not hit	Hit '_1'
b	Y		
c	Y		

Rows:	Hits	FEC Target	Non-masking condition(s)
-----	-----	-----	-----
Row 1:	1	a_0	(~(b & c) && b), (~(b & c) && c)
Row 2:	***0***	a_1	(~(b & c) && ~(a & c) && b), (~(b & c) && ~(a & b) && c)
Row 3:	1	b_0	(~(a & c) && a), (~((a & b) (a & c)) && c)
Row 4:	1	b_1	(~(b & c) && ~(a & c) && a), (~((a & b) (a & c)) && c)
Row 5:	1	c_0	(~(a & b) && a), (~((a & b) (a & c)) && b)
Row 6:	1	c_1	(~(b & c) && ~(a & b) && a), (~((a & b) (a & c)) && b)

```

Statement Coverage:
  Enabled Coverage      Bins      Hits      Misses      Coverage
  -----
  Statements            2        2        0      100.00%

=====Statement Details=====

Statement Coverage for instance /dut -
  Line      Item      Count      Source
  ----      -
  File fulladder.sv
  50          1          6
  51          1          6

Total Coverage By Instance (filtered view): 91.66%

```

In a larger design, multiple independent tests are run, often in parallel. To combine each of their UCDB files to get overall coverage, use:

```
$ vcover merge -out combined.ucdb test1.ucdb test2.ucdb
```

As discussed in [Section 5.1.5](#), Technology Readiness Level 5 (TRL5) requires 100% code coverage to ensure the entire code has been exercised. This typically involves writing a set of directed tests that cover the obvious cases (e.g., tests each instruction), using a random test generator to stress unobvious cases, and then writing more custom directed tests to check stranger cases (e.g., illegal instructions).

Nevertheless, it is often impossible to actually exercise all of the statements because certain parts of the code may be unreachable, particularly if the code is configurable. For example, a 32-bit configuration cannot exercise 64-bit instructions. The designer adds *coverage exclusions* to tell the coverage tool to ignore certain uncovered cases that the designer is satisfied are unreachable. [Section 5.11.11.2](#) gives an example of Wally coverage exclusions. Adding exclusions requires caution because poorly written or outdated exclusions can hide legitimate bugs.

5.9 FUNCTIONAL COVERAGE

Code coverage measures whether all code is exercised, while functional coverage measures whether all features have been exercised [Mehta20]. Functional coverage includes *architectural* coverage, which applies to any core supporting a particular extension, and *microarchitectural* coverage, which is specific to an implementation, stressing features such as pipeline hazards, interrupts, cache and bus systems, and branch prediction.

Functional coverage begins with the DV plan defining what features need to be covered. For example, in RISC-V, every instruction should be covered, using every register and corner cases appropriate to that instruction. Experience shows that coverage that is easy to describe doesn't catch many bugs, and coverage that catches hard bugs takes a lot of thought to describe.

SystemVerilog `covergroup` and `coverpoint` statements define how the coverage is measured. A `covergroup` consists of related `coverpoints` sampled at a particular time, such as `coverpoints` for a particular instruction. The `coverpoints` define the specific coverage, such as using every destination register. Each `coverpoint` has one or more bins. For example, the destination register `coverpoint` has 32 bins, one for each possible register. Wally uses a modified version of the Imperas `riscvISACOV` `coverpoints`, discussed below.

Functional coverage measurement involves running tests while the simulator monitors which `coverpoint` bins and assertions were hit. By also running the tests in lockstep mode against a reference model, you can ensure that the RTL produces the correct results for each of these `coverpoints`.

Functional coverage tests can be directed or random. Many features are easy to test with directed tests, but some corner cases are hard to reach and are easier to cover with constrained random tests. When functional coverage reaches 100%, the verification engineer knows enough tests have been run and all features in the DV plan have been exercised.

5.9.1 `riscvISACOV`

`riscvISACOV` is a set of SystemVerilog architectural functional `coverpoints` developed by Imperas (now Synopsys). They cover most of the common RISC-V extensions. A set of RV32I `coverpoints` are on a public repository [github.com/riscv-verification/riscvISACOV], and a larger set are available under a commercial license.

`riscvISACOV` has coverage definition files for every extension it supports (e.g., RV32I, RV64I, RV64M, etc.). Each file contains a `covergroup` for each instruction in that extension. Each `covergroup` contains various `coverpoints`. Table 5.5 summarizes `coverpoints` used for RV{32/64}I instructions. For example, each instruction is tested with every source and destination register and with various corner cases of signs and values. Not all instructions use all `coverpoints`; for example, R-type instructions don't use immediates, I-type instructions don't use `rs2`, shifts don't cover negative shift amounts, and only branches use branch offsets.

`riscvISACOV` uses the RVVI interface (see Section 5.6.1) to access the architectural state. The `ins` variable contains the current instruction sampled on the rising edge of the clock when it completes, including information such as the name, source and destination register identifiers and values, and whether a trap occurs. Fig. 5.5 lists an excerpt of the `riscvISACOV` RV32I add `covergroup` and its `coverpoints`. Only a few lines of repetitive definitions are shown.

5.9.2 Directed Tests

Test generators produce directed tests to cover the easy functional coverage cases systematically and efficiently.

Table 5.5 riscvISACOV RV{32/64}I coverpoints.

Name	Description	Number of Bins
cp_asm_count	Is the instruction used?	1
cp_rd	Each register is exercised	32
cp_rs1		32
cp_rs2		32
cp_rd_sign	Positive, negative	2
cp_rs1_sign		2
cp_rs2_sign		2
cp_offset		2
cp_imm_sign	0, positive, negative	3
cr_rs1_rs2	Cross-product of source signs	4
cr_rs1_imm		6
cmp_rd_rs1_eq	Destination and source uses same regs	32
cmp_rd_rs2_eq		32
cmp_rd_rs1_rs2	All three registers are the same	32
cmp_rd_rs1_eqval	Contents of registers match	2
cmp_rd_rs2_eqval		2
cp_rd_toggle	Each bit is 0 or 1	2XLEN
cp_rs1_toggle		2XLEN
cp_rs2_toggle		2XLEN
cp_rd_maxvals	0, 1, 2XLEN-1, $2^{XLEN-1}-1$, $2^{XLEN-1}+1$, $2^{XLEN}-1$	6
cp_rs1_maxvals		6
cp_rs2_maxvals		6
cp_gpr_hazard	None, RAW, WAR, WAW	4
cp_mem_hazard	None, RAW	2
cp_mem_unaligned	Yes, no	2


```

covergroup add_cg with function sample(ins_rv32i_t ins);
    option.per_instance = 1;
    option.comment = "Add";

    cp_asm_count : coverpoint ins.ins_str == "add" iff (ins.trap == 0) {
        option.comment = "Number of times instruction is executed";
        bins count[] = {1};
    }
    cp_rd : coverpoint ins.get_gpr_reg(ins.current.rd) iff (ins.trap == 0) {
        option.comment = "RD (GPR) register assignment";
    }
    cp_rd_sign : coverpoint int'(ins.current.rd_val) iff (ins.trap == 0) {
        option.comment = "RD (GPR) sign of value";
        bins neg = {[0:-1]};
        bins pos = {[1:$]};
    }
    cp_rs1 : coverpoint ins.get_gpr_reg(ins.current.rs1) iff (ins.trap == 0) {
        option.comment = "RS1 (GPR) register assignment";
    }
    cp_rs1_sign : coverpoint int'(ins.current.rs1_val) iff (ins.trap == 0) {
        option.comment = "RS1 (GPR) sign of value";
        bins neg = {[0:-1]};
        bins pos = {[1:$]};
    }
    cp_rs2 : coverpoint ins.get_gpr_reg(ins.current.rs2) iff (ins.trap == 0) {
        option.comment = "RS2 (GPR) register assignment";
    }
    cp_rs2_sign : coverpoint int'(ins.current.rs2_val) iff (ins.trap == 0) {
        option.comment = "RS2 (GPR) sign of value";
        bins neg = {[0:-1]};
        bins pos = {[1:$]};
    }
    cr_rs1_rs2 : cross cp_rs1_sign, cp_rs2_sign iff (ins.trap == 0) {
        option.comment = "Cross coverage of RS1 sign and RS2 sign";
    }
}

`ifdef COVER_LEVEL_COMPL_EXT
    cmp_rs1_rs2_eq : coverpoint ins.current.rs1 == ins.current.rs2 iff (ins.trap == 0) {
        option.comment = "Compare RS1 and RS2 register assignment";
        bins x0 = {1} iff (ins.current.rs1 == "x0");
        bins x1 = {1} iff (ins.current.rs1 == "x1");
        bins x2 = {1} iff (ins.current.rs1 == "x2");
        ...
        bins x31 = {1} iff (ins.current.rs1 == "x31");
    }
`endif
}

```

Fig. 5.5 riscvISACOV RV32I add covergroup.

```

cmp_rs1_rs2_eqval : coverpoint ins.current.rs1_val == ins.current.rs2_val
iff (ins.trap == 0) {
    option.comment = "Compare RS1 and RS2 register values";
    bins rs1_eqval_rs2 = {1};
    bins rs1_neval_rs2 = {0};
}

cp_rd_toggle : coverpoint unsigned'(ins.current.rd_val) iff (ins.trap == 0) {
    option.comment = "RD Toggle bits";
    wildcard bins bit_0_0 = {32'b????????????????????????????0?};
    wildcard bins bit_1_0 = {32'b????????????????????????????0?};
    wildcard bins bit_2_0 = {32'b????????????????????????????0?};
    ...
    wildcard bins bit_31_0 = {32'b0?????????????????????????????};
    wildcard bins bit_0_1 = {32'b????????????????????????????1?};
    wildcard bins bit_1_1 = {32'b????????????????????????????1?};
    wildcard bins bit_2_1 = {32'b????????????????????????????1?};
    ...
    wildcard bins bit_31_1 = {32'b1?????????????????????????????};
}

cp_rs1_toggle : coverpoint unsigned'(ins.current.rs1_val) iff (ins.trap == 0) {
    ...
}

cp_rs2_toggle : coverpoint unsigned'(ins.current.rs2_val) iff (ins.trap == 0) {
    ...
}

cp_rd_maxvals : coverpoint unsigned'(ins.current.rd_val) iff (ins.trap == 0) {
    option.comment = "RD Max values";
    wildcard bins zeros = {0};
    wildcard bins min = {32'b10000000000000000000000000000000};
    wildcard bins max = {32'b01111111111111111111111111111111};
    wildcard bins ones = {32'b11111111111111111111111111111111};
}

cp_rs1_maxvals : coverpoint unsigned'(ins.current.rs1_val) iff (ins.trap == 0) {
    ...
}

cp_rs2_maxvals : coverpoint unsigned'(ins.current.rs2_val) iff (ins.trap == 0) {
    ...
}

`endif

`ifdef COVER_LEVEL_DV_UP_BAS
cmp_rd_rs1_eq : coverpoint ins.current.rd == ins.current.rs1 iff (ins.trap == 0) {
    option.comment = "RD and RS1 register (assignment) WAR Hazard";
    bins x0 = {1} iff (ins.current.rd == "x0");
    bins x1 = {1} iff (ins.current.rd == "x1");
    ...
}

```

Fig. 5.5 riscvISACOV RV32I add covergroup. Cont'd

```

cmp_rd_rs1_eqval : coverpoint ins.current.rd_val == ins.current.rs1_val iff (ins.trap == 0) {
    option.comment = "Compare RD and RS1 register values";
    bins rd_eqval_rs1 = {1};
    bins rd_neval_rs1 = {0};
}
cmp_rd_rs2_eq : coverpoint ins.current.rd == ins.current.rs1 iff (ins.trap == 0) {
    ...
`endif
}
cmp_rd_rs2_eqval : coverpoint ins.current.rd_val == ins.current.rs2_val iff (ins.trap == 0) {
    ...
}
cp_gpr_hazard : coverpoint check_gpr_hazards(ins.hart, ins.issue) iff (ins.trap == 0) {
    option.comment = "GPR Hazard";
    bins hazards[] = {NO_HAZARD, RAW_HAZARD, WAR_HAZARD, WAW_HAZARD};
}
`endif
endgroup

```

Fig. 5.5 riscvISACOV RV32I add covergroup. Cont'd

These paths may change as testing evolves.

The `addins/cvw-arch-verif/bin/testgen.py` script produces directed tests in assembly language. They are placed in `addins/cvw-arch-verif/tests`. `testgen.py` reads the coverage files from the `fcov` directory to determine which directed tests to create for which instructions. See [Section 5.11.12](#) for how to run the directed test generator.

5.9.3 Constrained Random Tests

At the time of this writing, Wally is not regularly using riscv-dv, and scripts have broken.

riscv-dv is a widely used open-source constrained random test generator for RISC-V [github.com/chipsalliance/riscv-dv]. It was originally developed at Google and is now maintained by Chips Alliance. It consists of Python scripts that invoke SystemVerilog UVM modules that emit assembly language test files. riscv-dv is included as a submodule in Wally.

5.10 FORMAL VERIFICATION

A holy grail for logic verification would be to mathematically prove that HDL code faithfully implements a machine-readable specification. Such theorem provers are known as *formal verification* (FV) and are an active area of research. Some commercial offerings exist, but they presently require guidance from an experienced application engineer and are difficult to apply to the entirety of a complex design. Seligman provides an introduction to FV [Seligman23].

The general concept of FV is to annotate a DUT with SVA directives (see [Section 5.7](#)). Inputs and certain internal nodes are given `assume` directives to constrain their values. Outputs and other internal nodes are given `assert` directives to check their values. The chain of logic from assertions back to assumptions is called the *cone of influence*. The FV tool analyzes the logical relationships in the RTL to attempt to find a counterexample (consistent with the input assumptions) that violates the output assertions. If no counterexample exists, the assertion is proved. The number of cases to analyze increases exponentially with the size of the cone of influence, so FV becomes computationally intractable for blocks with large cones.

Algorithmic improvements have shrunk the cones requiring analysis, and methodology advancements have improved guidance about what blocks are suitable for FV and how to write the SVA directives. FV is likely to be practically relevant to an increasing set of designs in the future. For now, it is beyond the scope of this book.

For example, the OpenHW Group ran FV on the CV32E40P v2.0.0 core to compare it against the Sail specification, which was translated into a set of assertions using a proprietary tool. The process discovered many interesting bugs. However, the relatively simple core (4-stage RV32IMC) would have taken an impractical amount of compute time to completely prove that the RTL matched the Sail specification.

5.11 WALLY VERIFICATION

Wally targets TRL3 verification of all configurations and TRL5 verification of the rv64gc configuration. [Section 5.11.1](#) describes the overall DV plan to achieve these readiness levels. [Section 5.11.2](#) discusses how Sail and ImperasDV are used as reference models. [Sections 5.11.3 to 5.11.6](#) discuss the testbench, test vectors, lint, simulation, and regression testing used to achieve TRL3 and to test every time the code changes, as well as for larger nightly regression. Code changes usually introduce bugs, so [Section 5.11.7](#) describes debugging techniques to quickly locate the failure and get to the root cause. [Section 5.11.8](#) addresses floating-point unit-level testing. Next, we examine deeper testing for the rv64gc configuration. [Section 5.11.9](#) describes lockstep simulation with ImperasDV. [Sections 5.11.10 to 5.11.12](#) explore assertions, code coverage, and functional coverage for rv64gc.

5.11.1 Design Verification Plan

[Table 5.6](#) summarizes the elements of the Wally DV plan. The table lists each type of test that is run and the section that describes that test. For TRL3, it lists the applicable configurations. The primary configurations are summarized in [Tables A.1, A.3, and A.4](#), and the derivative configurations are summarized in [Table A.5](#) (see [Appendix A](#)). TRL5 verification

Table 5.6 Wally DV plan.

Tests	Section	TRL3	TRL5
Verilator Lint	5.3, 5.11.4	All configs	rv64gc
riscv-arch-test	3.7	All configs	rv64gc
wally-riscv-arch-test	3.7	All configs	rv64gc
TestFloat	5.11.8, 16.5.2	rv{32/64}gc + derived	rv64gc
CoreMark	21.1	Many configs	rv64gc
Embench	21.2	rv32*	n/a
Cache PV	21.3.1	rv{32/64}gc	rv64gc
BranchPred PV	21.3.2	rv{32/64}gc	rv64gc
Linux Boot	22.3.2	rv64gc	rv64gc
ImperasDV Lock-Step	5.6, 5.11.9		rv64gc
FPGA Linux Boot	23.2		rv64gc
Code Coverage	5.11.11		rv64gc
Functional Coverage	5.11.12		rv64gc

runs on the rv64gc configuration. Benchmarks and performance verification (PV) tests are also done to detect performance bugs.

The DV plan will continue to evolve with the code. The most up-to-date DV plan is in the cvw repository under `docs/testplans`.

Recall that the tests must be compiled before running simulations for the first time:

```
$ cd $WALLY
$ make --jobs
```

Invoke Wally's full set of verification tests (except FPGA Linux Boot) using:

```
$ regression-wally --nightly
```

5.11.2 Reference Model

RISCOF uses Sail as the reference model to generate reference signatures. Recall from [Section 3.7.2](#) that RISCOF crosschecks Sail signatures against Spike. The wally-riscv-arch-test tests that use CSRs, interrupts, and peripherals that are not supported by Sail set the `NO_SAIL=True` directive and use a manually written reference signature. The flow of GCC, RISCOF, Sail, and Verilator compile and simulates regression tests entirely with open-source tools.

Wally also uses the commercial ImperasDV as a reference model for rv64gc lock-step-compare. ImperasDV confirms that the manually written signatures are correct, catches subtle discrepancies that might not appear in a signature, and detects bugs in Sail. For example, ImperasDV detected an incorrect implementation of the Zfa `fcvtmod.w.d` instruction in Sail.

5.11.3 Top-Level Testbench and Testvectors

The regression tests include Verilator lint, riscv-arch-test, and wally-riscv-arch-test. They are run concurrently to complete as fast as possible (usually several minutes). The riscv-arch-test *pool* of programs is divided into suites (e.g., RV32 integer instructions, or RV64 multiply/divide instructions), each of which contain files (e.g., `rv32i_m/I/src/add-01.S`), each of which contain tens to thousands of tests (e.g., corner cases for operands and source and destination registers). A key issue is that most of the files are small and run in less than a second, but the startup time to launch the simulator and compile Wally takes many seconds. Therefore, the testbench is designed to simulate all of the files in a suite, one after another, to reduce the startup overhead.

Testbenches are in `$WALLY/testbench`. The main testbench is called `testbench.sv`. It uses various helper modules in the `common` subdirectory. The testbench takes a `TEST` argument from the command line, which specifies the test suite. `testbench` performs the following functions:

- ▶ Selects a suite of tests from `tests.vh` based on the `TEST` argument
- ▶ Instantiates `wallypipelinedsoc` as the DUT
- ▶ Generates a clock and pulses reset
- ▶ Verifies that the configuration is self-consistent using the `riscvassertions` module
- ▶ Provides instrumentation to decode the names of instructions and labels
- ▶ For each program in the selected suite
 - ▶ Loads the test program at `0x80000000` in hexadecimal format from `test.elf.memfile`

- ▶ Runs tests until the program reaches an infinite loop or `write_tohost`.
- ▶ Flushes the data cache back to memory
- ▶ Compares a specified range of memory to `test.signature.output`
- ▶ Reports success or failure

The test suites defined in `testbench/tests.vh` and summarized in [Table A.7](#) include test cases from the `riscv-arch-test` and `wally-riscv-arch-test` suites described in [Section 3.7](#). Note that one RISCOP test suite contains many testbench suites. For example, the RISCOP `arch32` suite contains `arch32i`, `arch32c`, `arch32m`, and so on. Details about the tests in each suite are described in subsequent chapters.

Interrupts are generated from software with test programs that configure the CLINT, PLIC, and GPIO as described in [Section 8.3.2](#).

The testbench can also load and run Coremark and Embench, as described in [Chapter 21](#). It also runs the coverage test suite and is used to collect code coverage measurements.

The buildroot test case is different than the others. It simply runs the first 1.6 million instructions of the Buildroot Linux boot and verifies that it does not hang. [Section 22.3.2](#) describes running the Linux boot in lockstep with ImperasDV to verify that it behaves correctly; this is a multiday simulation.

5.11.4 Lint

After making significant edits, start by running `lint` and fixing any warnings, as described in [Sections 5.1](#) and [5.3](#). The `lint-wally` script runs `lint` on `rv32e`, `rv64gc`, `rv32gc`, `rv32imc`, `rv32i`, `rv64i`, and `fdqh_rv64gc` configurations to detect configuration-dependent problems. It uses the `-I` flag to point to the desired configuration and the `testbench/wallywrapper.sv` *wrapper module* to load the configuration parameters and instantiate the top-level `wallypipelinedsoc`.

```
$ lint-wally
```

5.11.5 Simulation

Wally uses the `wsim` wrapper script to invoke simulation using Verilator, Questa, or VCS. Usage is as follows:

```
usage: wsim [-h] [--elf ELF] [--sim {questa,verilator,vcs}] [--tb
{testbench,testbench_fp}] [--gui] [--ccov] [--fcov] [--args ARGS] [--params PARAMS]
[--define DEFINE] [--vcd] [--lockstep] [--lockstepverbose] [--rvvi] config [testsuite]

positional arguments:
  config                Configuration file
  testsuite             Test suite or path to .elf file

options:
  -h, --help            Show this help message and exit
  --elf ELF, -e ELF     ELF File name; use if name does not end in .elf
  --sim {questa,verilator,vcs}, -s {questa,verilator,vcs}
                        Simulator
  --tb {testbench,testbench_fp}, -t {testbench,testbench_fp}
                        Testbench
  --gui, -g             Simulate with GUI
  --ccov, -c            Code Coverage
  --fcov, -f           Functional Coverage using cvw-arch-verif
  --args ARGS, -a ARGS Optional arguments passed to simulator via $value$plusargs
  --params PARAMS, -p PARAMS Optional top-level parameter overrides of the form param=value
  --define DEFINE, -d DEFINE Optional define macros passed to simulator
  --vcd, -v            Generate testbench.vcd
  --lockstep, -l       Run ImperasDV lock, step, and compare
  --lockstepverbose, -lv Run ImperasDV lock, step, and compare with tracing enabled
  --rvvi, -r          Simulate RVVI hardware interface and ethernet
```

Example 5.1 USING `wsim` TO RUN SIMULATIONS

Simulate integer tests on the `rv64gc` configuration in Questa:

```
$ wsim rv64gc arch64i
<messages starting up Questa and compiling Verilog>
# rv64i_m/I/src/add-01.S succeeded.  Brilliant!!!
# rv64i_m/I/src/addi-01.S succeeded.  Brilliant!!!
...
# rv64i_m/I/src/xori-01.S succeeded.  Brilliant!!!
# SUCCESS! All tests ran without failures.
# ** Note: $stop : /home/harris/cvw/testbench/testbench.sv(424)
# Time: 4449160 ns Iteration: 1 Instance: /testbench
# Break at /home/harris/cvw/testbench/testbench.sv line 424
# Stopped at /home/harris/cvw/testbench/testbench.sv line 424
# End time: 04:08:23 on May 06,2024, Elapsed time: xxx
# Errors: 0, Warnings: 0
```

Run integer tests in Questa GUI for debugging:

```
$ wsim rv64gc arch64i --gui
```

Run floating-point testbench for addition tests. Many other derived configurations are available.

```
$ wsim f_rv64gc add --tb testbench_fp
<messages starting up Questa and compiling Verilog>
# Running f32_add_rne.tv vectors
# Running f32_add_rz.tv vectors
# Running f32_add_ru.tv vectors
# Running f32_add_rd.tv vectors
# Running f32_add_rnm.tv vectors
#
# All Tests completed with          0 errors
```

Verilator requires the user to increase the stack size before the first `wsim` run using `ulimit`. This is done in `setup.sh`.

Simulate multiply/divide instructions on `rv32i` configuration in Verilator, generating VCD file to view waveforms later. The VCD file can occupy gigabytes, so delete it when you are done.

```
$ wsim -s verilator rv32imc arch32m --vcd
<messages about compiling files>
g++ -fuse-ld=mold wrapper.o verilated.o verilated_dpi.o verilated_vcd.c.o
verilated_timing.o verilated_threads.o Vtestbench__ALL.a -pthread -
lpthread -latomic -o Vtestbench
rm Vtestbench__ALL.verilator_deplist.tmp
make[1]: Leaving directory
'/home/harris/cvw/sim/verilator/wkdir/rv32imc_arch32m'
- V e r i l a t i o n   R e p o r t: Verilator 5.025 devel rev v5.024-12-
ga9664b8d1
- Verilator: Built from 308.865 MB sources in 244 modules, into 9.008 MB in
33 C++ files needing 76.297 MB
- Verilator: Walltime 9.125 s (elab=0.068, cvt=0.201, bld=8.733); cpu 0.000 s
on 1 threads; allocated 73.996 MB
mkdir -p /home/harris/cvw/sim/verilator/logs
wkdir/rv32imc_arch32m/Vtestbench +TEST=arch32m
rv32i_m/M/src/div-01.S succeeded. Brilliant!!!
rv32i_m/M/src/divu-01.S succeeded. Brilliant!!!
...
rv32i_m/M/src/mulhu-01.S succeeded. Brilliant!!!
SUCCESS! All tests ran without failures.
- /home/harris/cvw/testbench/testbench.sv:422: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.025 devel
- Verilator: $finish at 799ns; walltime 0.126 s; speed 0.000 s/s
- Verilator: cpu 0.000 s on 1 threads; allocated 4914 MB
make: Leaving directory '/home/harris/cvw/sim/verilator'
```

View waveforms from Verilator simulation:

```
$ gtkwave -f $WALLY/sim/verilator/testbench.vcd
```

Simulate integer tests on the rv64gc configuration in VCS:

```
$ wsim -s vcs rv64gc arch64i
Read memfile ../../tests/riscov/work/riscv-arch-test/rv32i_m/I/src/add-01.S/ref/ref.elf.memfile
rv32i_m/I/src/add-01.S succeeded. Brilliant!!!
...
Read memfile ../../tests/riscov/work/riscv-arch-test/rv32i_m/I/src/xori-01.S/ref/ref.elf.memfile
rv32i_m/I/src/xori-01.S succeeded. Brilliant!!!
SUCCESS! All tests ran without failures.
$finish called from file "/home/harris/cvw/testbench/testbench.sv", line 427.
$finish at simulation time 1034140
Info-[HSE-SIMPROF] Simulating for the full-feature CPU time profiler
V C S S i m u l a t i o n R e p o r t
Time: 1034140
CPU Time: xxx seconds; Data structure size: 264.1Mb
Tue May 7 12:59:36 2024
```

`wsim` launches the selected simulator in a simulator-dependent fashion. For example, to launch Questa:

```
$ wsim rv32gc arch32i --sim questa
Config=rv32gc tests=arch32i sim=questa gui=False args=""
params="" define=""
Running Questa with command: cd $WALLY/sim/questa;
MTI_VCO_MODE=64 vsim -c -do "do wally.do rv32gc arch32i
testbench --args \" \" --params \" \" --define \" \" "
```

`wally.do`, a *do file*, is a TCL file instructing Questa to compile, optimize, and simulate the code. It uses the `+TEST` flag to override a testbench variable called `TEST` with the name of the test suite. The code is placed in a library that is named based on the configuration and test suite so that many simulations can be performed concurrently during regression. `wally.do` also loads up waveforms using `wave.do`, which organizes commonly used signals into convenient groups. These scripts pass the configuration with the `+incrdir` argument so the compiler includes the appropriate `config.vh` file.

5.11.6 Regression

The regression-`wally` script launches `lint-wally` and all the test suite simulations in parallel and reports any errors. It uses Verilator because this simulator is free and fast. Table A.7 lists the various test suites run on various configurations. Some suites are repeated; for example, the `arch32i`

integer tests are run on rv32i, rv32imc, and rv32gc to check all these configurations. If a test suite fails, go into the `logs` subdirectory and look at the `<config>_<suite>.log` file. Fix all compiler warnings and errors. If there is a real logic problem in the Verilog, you may need to simulate it using `wsim` to track it down. To keep the repository in always-working state, it is essential to run `regression-wally` before committing any changes.

```
$ regression-wally
```

5.11.6.1 Nightly Regression

Regression testing confirms that a recent change has not broken the system. The basic Wally regression tests (`regression-wally`) run architecture compatibility tests on the main Wally configurations. Experience has shown that it is important to run a larger suite of tests on a regular basis. These take longer to run, so they are often run automatically while the designer is away and are called *nightly regression*.

Wally's nightly regression includes:

- ▶ `regression-wally` expanded to run applicable tests on all derived configurations (Section 4.4)
- ▶ `lint-wally` on all derived configurations
- ▶ Normal `regression-wally` on Verilator, Questa, and VCS
- ▶ ImperasDV lock-step tests for the rv64gc configuration on riscv-arch-test, wally-riscv-arch-test, coverage, and Linux boot
- ▶ TestFloat on all floating-point configurations
- ▶ Functional coverage tests
- ▶ Cache performance validation tests
- ▶ CoreMark and Embench benchmarks

Eventually it should also include branch predictor performance validation tests.

Invoke nightly regression using:

```
$ regression-wally --nightly
```

5.11.6.2 Nightly Build

The `nightly_build.py` script clones a fresh copy of the cvw repository, makes all of the test cases, and runs these regression tests using

regression-wally --nightly. It sends an email to a user or mailing list with a summary of the results, highlighting any failures. Invoke using:

```
$ nightly_build.py --path ~/nightly --target all --tests all
--send_email ben@hmc.edu, alyssa@hmc.edu
```

--path specifies where to check out the repository and build the tests. --target all specifies building both Linux and RISCOF tests. --tests all runs regression-wally --nightly --buildroot. --send_email is followed by a comma-separated list of email addresses. The results are saved in nightly/<date>/results. If particular tests fail, consult the logs in nightly/<date>/logs.

The local copies of the repository and old logs can take a significant amount of disk space. nightly_build.py deletes old files, keeping only the most recent copy of the repository and the last 30 days of logs.

To automatically run nightly_build.py at 9 PM every evening, edit the cron table using:

```
$ crontab -e
```

Paste the following line into the crontab editor:

```
0 21 * * * curl -L https://raw.githubusercontent.com/openhwgroup/cvw/
refs/heads/main/bin/nightly_build.py | /opt/riscv/riscv-python/bin/
python - --path PATH/T0/NIGHTLY --target all --tests all --send_email
ben@hmc.edu, alyssa@hmc.edu
```

5.11.7 Debugging Hints

This section introduces a few bugs into the design and then shows how to systematically find them. The bugs will include:

- ▶ A syntax error caught by lint
- ▶ A compilation error caught by Questa
- ▶ A logic error in the design caught by a regression failure

Make the following edits in \$WALLY/src/:

- ▶ In ieu/ieu.sv, misspell one of the input ports as inputt.
- ▶ In ieu/datapath.sv, change the output of the fbemux from ForwardedSrcBE to ForwardedSrcAE (which is also the output of faemux, resulting in contention).
- ▶ In ieu/alu.sv, change the line for the OR gate from `A | CondMaskInvB` to `A + CondMaskInvB`.

The error messages, line numbers, wave.do file, and code for this example may change as tool releases and Wally continue to evolve. You may need to use your judgement to debug.

Now try to find these bugs. First run lint:

```
$ lint-wally
Error: ../src/ieu/ieu.sv:31: syntax error, unexpected logic, expecting ')' or ','
  inputt logic      clk, reset,
      ^~~~~
%Error: ../src/ieu/ieu.sv:33: syntax error, unexpected input
  input logic [31:0]   InstrD,
      ^~~~~
```

Focus on the first error reported and don't get distracted by the many other errors. Carefully reading the message, we see an error right after `inputt` that points us to fix the misspelling.

Fix and run lint again, and it should pass. Now try running regression.

```
$ regression-wally
...
wsim --sim questa rv32gc arch32m >
/home/harris/cvw/sim/questa/logs/rv32gc_arch32m.log: Failures detected in output
  Check /home/harris/cvw/sim/questa/logs/rv32gc_arch32m.log
...
Regression failed with 72 failed configurations
```

Given that almost every test failed, there's clearly something majorly wrong with the code. Look at any one of the failing log files and scroll through until you find the error reported:

```
$ tail $WALLY/sim/questa/logs/rv32gc_arch32m.log
...
# ** Error (suppressible): ../src/ieu/datapath.sv(108): (vopt-7045)
'ForwardedSrcAE' is driven by more than one continuous assignment. See
../src/ieu/datapath.sv(109).
```

Go to `src/ieu/datapath.sv` and look or search for components driving `ForwardedSrcAE`. The error message points to lines 108 and 109. You'll see that it is the output of two multiplexers. Fix this by changing the output of the `fbemux` back to `ForwardedSrcBE`.

Run regression again to find the next issue. Some of the tests pass and some fail, indicating an error affecting certain tests. There's no need to wait for everything to run; press Ctrl-c to halt when you are satisfied there is a problem.

```
$ regression-wally
rv32e passed lint
rv64gc passed lint
wsim --sim questa rv32i arch32i >
/home/harris/cvw/sim/questa/logs/rv32i_arch32i.log: Failures detected in output
Check /home/harris/cvw/sim/questa/logs/rv32i_arch32i.log
wsim --sim questa rv32imc arch32c >
/home/harris/cvw/sim/questa/logs/rv32imc_arch32c.log: Failures detected in output
Check /home/harris/cvw/sim/questa/logs/rv32imc_arch32c.log
wsim --sim questa rv32imc wally32periph >
/home/harris/cvw/sim/questa/logs/rv32imc_wally32periph.log: Success
wsim --sim questa rv32imc arch32m >
/home/harris/cvw/sim/questa/logs/rv32imc_arch32m.log: Success
...
```

Pick either the first failure or a failure in a test suite that you are most familiar with. The arch32i tests are a good choice because they relate to simple integer instructions that we understand well. Look at the log file for the problem:

```
$ less $WALLY/sim/questa/logs/rv32imc_arch32i.log
...
# Read memfile ../../addins/riscv-arch-test/work/rv32i_m/I/lw-align-01.elf.memfile
# rv32i_m/I/lw-align-01 succeeded. Brilliant!!!
# Read memfile ../../addins/riscv-arch-test/work/rv32i_m/I/or-01.elf.memfile
# Error on test rv32i_m/I/src/or-01.S result 2: adr = 80006118 sim (D$) 00000004
signature = 00000002
# ** Note: $stop : /home/harris/cvw/testbench/testbench.sv(884)
# Time: 758300 ns Iteration: 1 Region: /testbench/genblk3
# Break at /home/harris/cvw/testbench/testbench.sv line 884
# Stopped at /home/harris/cvw/testbench/testbench.sv line 884
...
```

This shows that the problem is in the `rv32i_m/I/or-01` test suite during test number 2 (indicated by the “2:”). The expected signature is 00000002, but the simulation computes 00000004. Start by studying the test suite to understand the failing test. Look in `tests/riscv/work/riscv-arch-test/rv32i_m/I/src/or-01.S/ref/ref.elf.objdump` to see the disassembled file with corresponding source code lines. You’ll see lines of code initializing registers and then setting `gp` to `signature_x3_0` (0x80006114). Then you’ll see the test cases. The failing test is reproduced below. The key instruction at 0x80000118 computes 2 OR 2. The answer should be 2 (consistent with the signature reported in the error message above).

```
80000130 <inst_1>:
80000130:      00200313          li      t1,2
80000134:      00200313          li      t1,2
80000138:      006368b3          or      a7,t1,t1
8000013c:      011a223          sw      a7,4(gp)
```

We will need to look at the simulation waveforms to troubleshoot. To make the failure occur early in the simulation, edit `tests.vh` and place the failing simulation at the beginning of the suite of tests:

```
string arch32i[] = '{
  `RISCVARCHTEST,
  "rv32i_m/I/or-01",    # Added this test at start
  "rv32i_m/I/add-01",
  ...
}
```

Now manually invoke the failing simulation with a GUI so we can inspect the waves. Note that the `do` file requires the configuration (`rv32imc`) and the name of the test suite (`arch32i`).

```
$ wsim --gui rv32imc arch32i
```

The simulation will run for a little while. You may see harmless errors about no object matching certain signal names because some configurations do not have all signals. It should fail with the same error as above. Explore the different groups of waves. Scroll to the start and click to put the yellow cursor at the start. Expand the Memory Stage wave group and click on `IEUAdrM`, the load/store address in the Memory stage. In the search box in the toolbar, enter the address of the bad store (80006118) and click the adjacent Search Next icon. Zoom in until the waves are legible. Expand the “alu execution stage” wave group and look back two cycles to see `SrcAE` and `SrcBE` are 00000002 and `ALUResultE` is 00000004. Now we need to figure out why `ALUResultE` is incorrect. Click on `ALUResultE` and use the Add → To Wave → Contributing Signals menu to add all the signals causing `ALUResultE`. `PreALUResult` should show up. Add contributing signals for `PreALUResult`, and `FullResult` should show up. Add the contributing signals for `FullResult`. Fig. 5.6 shows all of these signals, including `A = 2`, `CondMaskInvB = 2`, `ALUSelect = 6`, and `FullResult = 4`.

Browse or search in a text editor to find the portion of code that computes this result, shown below. We see for `ALUSelect = 6` that the system selects `FullResult = A + CondMaskInvB`. This is the error. Fix it by replacing the blue `+` with `|`.

```
always_comb
case (ALUSelect)
  3'b000: FullResult = Sum; // add or sub
  3'b001: FullResult = Shift; // sll, sra, or srl
  3'b010: FullResult = {{(WIDTH-1){1'b0}}, LT}; // slt
  3'b011: FullResult = {{(WIDTH-1){1'b0}}, LTU}; // sltu
  3'b100: FullResult = A ^ CondMaskInvB; // xor, xnor, binv
  3'b101: FullResult = ... : Shift; // bext (or IEU shift)
  3'b110: FullResult = A + CondMaskInvB; // or, orn, bset
  3'b111: FullResult = A & CondMaskInvB; // and, bclr
endcase
```

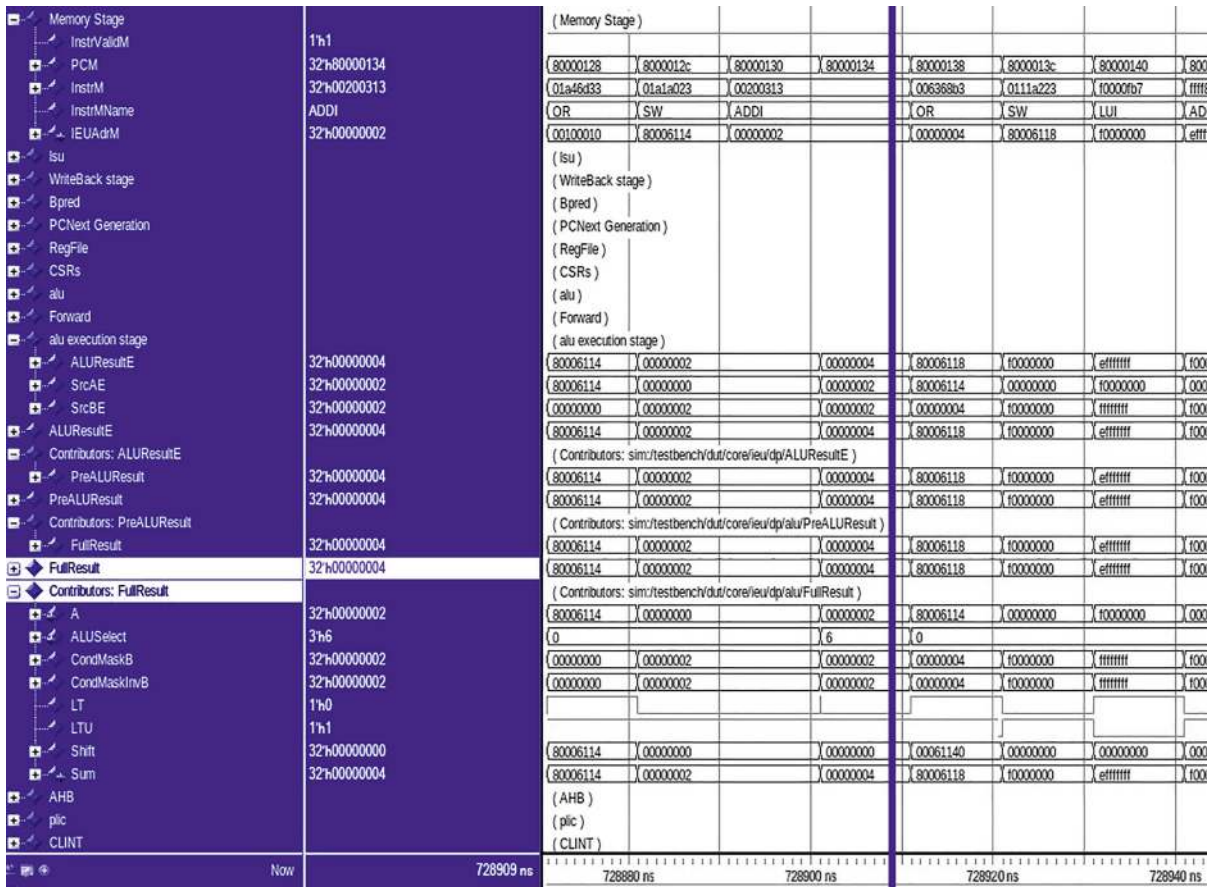


Fig. 5.6 rv32imc arch32i simulation waveforms.

Rerun regression-wally to confirm that this has fixed the problem.

5.11.8 Floating-Point Unit Testing

The riscv-arch-test vectors exercise the floating-point unit (FPU). The Berkeley TestFloat vectors, described in [Section 16.5.2](#), exercise the FPU in different ways that have detected bugs not found by riscv-arch-test. They also test quad precision, which is not part of riscv-arch-test at the time of this writing.

TestFloat has a large number of vectors, so it is well-suited to unit-level testing. Such testing is faster because it only runs the FPU execution units, rather than the entire core, and because it applies vectors directly to the execution units rather than running supporting code to move values into registers and write signatures to memory.

FPU unit testing uses the unit-level `testbench/testbench-fp.sv` and vectors from `tests/fp/vectors`. See [Section 16.5.2](#) for directions on how to invoke the tests. `regression-wally --nightly` runs `TestFloat` for all supported precisions (half-, single-, double-, and quad-precision) on both primary and derived configurations.

5.11.9 Lock-Step-Compare

ImperasDV is a commercial tool that requires a license. It is not part of the Synopsys academic bundle. This section assumes that you have a license and have installed the Imperas tools in `$RISCV/ImperasDV-OpenHW`.

As discussed in [Section 5.6](#), ImperasDV performs lock-step-compare of running an Executable and Linkable Format (ELF) file on the Verilog model and on an Imperas reference model. It captures any discrepancies between the state of the system, including registers, the PC, CSRs, and memory. The reference model watches memory/peripheral reads from the Verilog to return the same values. It also watches the Verilog to take interrupts at the same time. This technique has located at least 20 bugs in Wally [Thompson23].

To run a single ELF file, use `wsim` with the `--lockstep` option:

```
$ wsim rv64gc $WALLY/tests/riscov/work/riscv-arch-test/rv64i_m/I/src/add-01.S/ref/ref.elf
--lockstep
...
# Info (IDV) -----
# Info (IDV) ImperasDV INITIALISED
# Info (IDV) -----
...
# Info (IDV) ImperasDV VERIFICATION REPORT
...
# Info (IDV) Total compares      : 24,371
# Info (IDV) Mismatches         : 0
```

Use `--lockstepverbose` instead to record each instruction that is committed. The output is large, so pipe it to a log file:

```
$ wsim rv64gc $WALLY/tests/riscov/work/riscv-arch-test/rv64i_m/I/src/add-01.S/ref/ref.elf
--lockstepverbose > log
```

The following excerpt from the log file shows execution of the first instruction: `lui x16, 0x7d5c0`, highlighted in blue. It is represented in machine language as `7d5c0837` at address `80000000`. After the instruction runs, `x16` changes to `000000007d5c0000`.

```
# Info (RISCV_PMP) CPU 'refRoot/cpu': PMP PRIV=rwx 0x00000000: 0xffffffff (mode Machine)
# Info 1: 'refRoot/cpu', 0x0000000080000000(rvtest_init): Machine 7d5c0837 lui x16,0x7d5c0
# Info MEMX 0x80000000 0x80000000 2 0837
# Info MEMX 0x80000002 0x80000002 2 7d5c
# Info x16 0000000000000000 -> 000000007d5c0000
```

Section 22.3.2 describes how to run the Linux boot in lockstep with ImperasDV.

`regression-wally --nightly` runs lockstep simulations on all of the wally-riscv-arch-test and coverage tests, as well as the Linux boot.

5.11.10 Assertions

At the time of this writing, the only Wally assertions are in `riscvassertions.sv`, as described in Section 5.7.1. They check that the configuration options in the selected `config.vh` are self-consistent.

5.11.11 Code Coverage

Wally measures code coverage on the `rv64gc` configuration. It runs the functional coverage tests (described in Section 5.11.12) along with some extra hand-written directed tests for code coverage. The directed tests exercise features such as illegal instructions and each entry of the TLB and PMP. These extra tests must be compiled first:

```
$ cd $WALLY/tests/coverage
$ make --jobs
```

Then run regression with the `--ccov` flag.

```
$ regression-wally --ccov
```

Regression runs all the usual `rv64gc` tests, plus the extra `coverage64gc` tests just compiled. It also applies exclusions from the `$WALLY/sim/coverage-exclusions-rv64gc.do` file. It puts the results in `$WALLY/sim/questa/cov` and `$WALLY/sim/covhtmlreport/index.html`.

The HTML report is a good place to see the coverage percentages for the top-level modules. `rv64gc_coverage_hierarchical.rpt` lists each type of coverage for each instance in the design. Most modules are fully covered, so the `rv64gc_uncovered_hierarchical.rpt` is more useful, showing only the instances with incomplete coverage. For each top-level module, the `rv64gc_uncovered_<module>.rpt` contains a detailed report sufficient to identify the uncovered code.

The `coverage-exclusions-rv64gc.do` file waives coverage in a variety of areas. In particular, the load/store unit (LSU) and instruction fetch unit (IFU) share common memory management unit (MMU) and cache modules, which are configured to support reads and writes for the LSU and instruction fetches for the IFU. Unsupported access types cannot occur, so they must be excluded from coverage. Questa excludes coverage based on ranges of lines. This is unstable in that code changes that change line numbering break the exclusions in ways that are difficult to maintain. Wally defines a `GetLineNum` function that looks for a particular string in the code and substitutes the corresponding line number. For example,

Hint: in Microsoft's Visual Studio Code, install the HTML Preview extension. Then when you open `index.html`, click on the Show Preview button in the upper right to view the HTML web page.

cache fsm.sv has a comment `//exclusion-tag: cache AnyMiss`. The coverage exclusion file searches for that line and excludes certain coverage related to cache misses.

5.11.11.1 Coverage Example: Illegal Instruction Test Case

As an example, consider test coverage for the controller decoding branch instructions. According to Table I.1 (on the last pages of the book), branches have `Op = 1100011` and `Func3` of anything except 010 and 011. The relevant section of the controller is shown in [Code Example 5.12](#). Line 211 is bolded.

Code Example 5.12 controller.sv BRANCH DECODING

```
assign BFuncD = (Func3D[2:1] != 2'b01); // Legal branches
always_comb begin
    ControlsD = `CTRLW'b0_000_00_00_000_0_0_0_0_0_0_0_0_0_0_0_0_1; // Default: Illegal instr.
    case(OpD)
        ...
        7'b1100011: if (BFuncD) // Branches
            ControlsD = `CTRLW'b0_010_11_00_000_1_0_0_0_0_0_0_0_0_0_0_0;
```

Branches with an illegal `Func3D` should throw an Illegal Instruction exception, but these ill-formed branches are not part of the riscv-arch-test suite. Therefore, we add a line to `tests/coverage/ieu.S` with `Op = 1100011` and `Func3 = 010` to check that illegal branches are detected. Zeroing all the other bits of the instruction that the decoder ignores gives an illegal instruction of `0x00002063`. This line is shown in [Code Example 5.13](#). Note that the `.word` assembler directive is used because there is no assembly language opcode for an illegal branch.

Code Example 5.13 tests/coverage/ieu.S ILLEGAL BRANCH

```
.word 0x00002063 // Illegal branch instruction
```

To see that this test is important, comment out the line and invoke `make` in the `tests/coverage` directory to recompile. Then run `regression-wally --ccov`. [Code Example 5.14](#) shows an excerpt of the `rv64gc_uncovered_ieu.rpt` produced, showing that one branch is not covered. The uncovered branch is at line 211, with an All False Count of 0. This means that `if (BFuncD)` is never false when `OpD` is 1100011. Restoring the commented out line from `ieu.S` creates a branch test where `BFuncD = 0`, exercising a malformed illegal branch.

Code Example 5.14 `sim/cov/rv64gc_uncovered_iou.rpt`

```
=====
=== Instance: /core/iou/c
=== Design Unit: work.controller
=====

Branch Coverage:
  Enabled Coverage          Bins      Hits      Misses   Coverage
  -----
  Branches                  44       43        1    97.72%
  ...
-----IF Branch-----
  211                        15647      Count coming in to IF
  211                2      15647
                        ***0***      All False Count
Branch totals: 1 hit of 2 branches = 50.00%
```

5.11.11.2 Coverage Example: Unreachable Logic Exclusion

As another example, consider adding an exclusion for unreachable logic. The instruction and data caches share the same code. As shown in bolded line 112 of [Code Example 5.15](#), `cache fsm` receives an `InvalidateCache` signal used to invalidate the instruction cache; this signal is never asserted for data caches. Therefore, it is impossible for tests to exercise this `cache fsm` code for the data cache.

Code Example 5.15 `cache fsm.sv` EXCERPT

```
always_comb begin
  NextState = STATE_ACCESS;
  case (CurrState)
    STATE_ACCESS: if(InvalidateCache)
      NextState = STATE_ACCESS;
      else if (FlushCache & ~READ_ONLY_CACHE)
        NextState = STATE_FLUSH;
      else if (AnyMiss & (READ_ONLY_CACHE | ~LineDirty))
        NextState = STATE_FETCH;
      else if (AnyMiss & ~READ_ONLY_CACHE)
        NextState = STATE_WRITEBACK;
      else
        NextState = STATE_ACCESS;
    STATE_FETCH: if (CacheBusAck)
      NextState = STATE_WRITE_LINE;
      else
        NextState = STATE_FETCH;
```

[Code Example 5.16](#) shows an excerpt of `rv64gc_lsu_uncovered.rpt` illustrating this problem. The report indicates that most of the `if` conditions are activated hundreds or thousands of times, but item 2 of line 112 is never exercised, as highlighted in blue.

Code Example 5.16 `sim/cov/rv64gc_lsu_uncovered.rpt`

```

=====
=== Instance: /core/lsu/bus/dcache/dcache/cache fsm
=== Design Unit: work.cache fsm
=====
Branch Coverage:
  Enabled Coverage          Bins      Hits      Misses   Coverage
  -----
  Branches                  27       26         1    96.29%

=====Branch Details=====
Branch Coverage for instance /core/lsu/bus/dcache/dcache/cache fsm

  Line      Item                      Count      Source
  ----      -
  -----CASE Branch-----
  111                      460817    Count coming in to CASE
  112                      405537
  117                      21035
  119                      10397
  120                      21468
  123                      32
  126                      1183
  129                      1137
  133                      28
Branch totals: 8 hits of 8 branches = 100.00%

  -----IF Branch-----
  112                      405537    Count coming in to IF
  112                      ***0***
  113                      556
  114                      21033
  115                      49
  116                      383899
Branch totals: 4 hits of 5 branches = 80.00%

```

To waive covering this line in the data cache but not instruction cache, we add a unique comment to the end of line 112:

```
// exclusion-tag: dcache InvalidateCheck
```

Next, we add a line to the `coverage-exclusions-rv64gc.do` file, as shown in [Code Example 5.17](#) highlighted in blue. It uses a custom function called `GetLineNum` to search for the line of `cache fsm.sv` containing this exclusion tag comment. It applies an exclusion with the `-scope` option to only waive coverage on the data cache, not the instruction cache. The `-item b 2` option excludes the uncovered branch from the coverage report.

This line is actually already in the do file, so the unreachable logic is waived and not reported as uncovered.

Code Example 5.17 `sim/coverage-exclusions-rv64gc.do`

```
coverage exclude -scope /dut/core/lsu/bus/dcache/dcache/cachefsm -linerange
[GetLineNum ../src/cache/cachefsm.sv "exclusion-tag: dcache
InvalidateCheck"] -item b 2
```

5.11.12 Functional Coverage

To collect functional coverage, Wally needs the covergroup files and test cases. This is a work in progress at the time of this writing.

Wally presently focuses on architectural functional coverage. It uses `cvw-arch-verif`, which is an extension of the nonproprietary `riscvISACOV` functional covergroups. Wally runs Questa simulations of test programs in lockstep with ImperasDV to detect logic bugs while measuring coverage. The goal is to have adequate covergroups for every RISC-V feature, test programs that give 100% coverage of these covergroups, and no logic bugs.

To create the covergroups and test programs and then run them on Wally:

```
$ cd $WALLY/addins/cvw-arch-verif
$ make --jobs
$ regression-wally --fcov
```

`make` first invokes `covergroupgen.py`, which reads the `.csv` files from the `testplans` directory specifying which coverpoints apply to which unprivileged instructions in each extension for RV32 and RV64. It generates functional coverage groups in the `fcov` subdirectory. `make` then runs `testgen.py`, which reads these covergroups and produces assembly language tests in the `tests` subdirectory to hit each coverpoint.

`regression-wally --fcov` runs these programs from the `tests` directory in lockstep while collecting coverage on a list of covergroups from the `fcov` directory specified in `$WALLY/config/<configuration>/coverage.svh`. It produces `.ucdb` files for each test, merges them, and puts reports in the `work` directory.

An excerpt of the coverage reports is shown in [Code Examples 5.18 and 5.19](#). These reports show some RV32I instructions with 100% coverage and some with partial coverage. For example, the `beq` hits 100% of the bins in each coverpoint except that it does not yet test backward branches.

Code Example 5.18 \$WALLY/addins/cvw-arch-verif/work/summary_rv32gc.txt

Covergroup Coverage:				
Covergroups	79	na	na	69.70%
Covergroup Bins	30958	20426	10532	65.97%
Covergroup	Metric	Goal	Bins	Status
Covergroup instance obj_add	100.00%	100	-	Covered
Covergroup instance obj_addi	100.00%	100	-	Covered
Covergroup instance obj_and	100.00%	100	-	Covered
Covergroup instance obj_andi	100.00%	100	-	Covered
Covergroup instance obj_auipc	88.57%	100	-	Uncovered
Covergroup instance obj_beq	96.42%	100	-	Uncovered

Code Example 5.19 \$WALLY/addins/cvw-arch-verif/work/report_rv32gc.txt

TYPE /RISCV_coverage_pkg/RISCV_coverage__1/beq_cg	96.42%	100	-	Uncovered
covered/total bins:	404	405	-	
missing/total bins:	1	405	-	
% Hit:	99.75%	100	-	
Coverpoint cp_asm_count	100.00%	100	-	Covered
Coverpoint cp_rs1	100.00%	100	-	Covered
Coverpoint cp_rs2	100.00%	100	-	Covered
Coverpoint cmp_rd_rs1_eqval	100.00%	100	-	Covered
Coverpoint cp_rs1_corners	100.00%	100	-	Covered
Coverpoint cp_rs2_corners	100.00%	100	-	Covered
Coverpoint cmp_rs1_rs2	100.00%	100	-	Covered
Coverpoint cp_rs1_sign	100.00%	100	-	Covered
Coverpoint cp_rs2_sign	100.00%	100	-	Covered
Coverpoint cp_offset	50.00%	100	-	Uncovered
covered/total bins:	1	2	-	
missing/total bins:	1	2	-	
% Hit:	50.00%	100	-	
bin neg	0	1	-	ZERO
bin pos	539	1	-	Covered
Coverpoint cp_rs1_toggle	100.00%	100	-	Covered
Coverpoint cp_rs2_toggle	100.00%	100	-	Covered
Cross cr_rs1_rs2_corners	100.00%	100	-	Covered
Cross cr_rs1_rs2	100.00%	100	-	Covered

By the time you read this, the unprivileged instructions are likely to be fully covered. Functional coverage will eventually include covergroups and tests for privileged features, including virtual memory, PMP, and CSRs.

5.12 SUMMARY

Logic verification now accounts for nearly twice as much effort as all the other elements of chip design combined. Although customers expect software to contain bugs and are accustomed to frequent updates, they need hardware to work reliably. Serious hardware bugs that require creating a new mask set and fabricating new chips cost millions of dollars in engineering costs and even more in opportunity costs when a chip is late to market. Defining the set of features that must be tested, the test vectors that exercise these features, and the expected results is harder than coding the features themselves.

Lint tools save a great deal of debugging time by warning the designer of bad code, such as undriven signals, combinational loops, and mismatched port sizes. The next verification step is to simulate the design in a testbench that applies test vectors and logs any discrepancies. Verification engineers also measure code coverage to ensure that the test vectors exercise every part of the code. Regression testing runs the full suite of tests after each code change to ensure that a new feature doesn't break the existing system; this is important to know right away because it is much easier for a developer to fix bugs in code as it is being written than for somebody else unfamiliar with the buggy code to isolate and fix the bug later.

The RISC-V Architecture Test Special Interest Group provides the `riscv-arch-test` suite of programs that exercise each unprivileged instruction and produce a signature for each test. Wally's testbench loads each program into memory and runs it on the Verilog, then checks that the signature in memory matches expectations. Wally also uses its own `wally-riscv-arch-test` suite to exercise privileged instructions, CSRs, exceptions, and the virtual memory system.

Wally also uses the Imperas lock-step-compare to run programs and check that the architectural state (memory, PC, registers, and CSRs) in the Verilog model matches the Imperas reference model after each instruction. This methodology has caught many subtle errors that are difficult to detect with signature-based verification.

Despite best efforts at verification, any system complicated enough to be interesting almost certainly has bugs lurking. An experienced design team has the humility to recognize human imperfection. Complex systems often have firmware-programmable configuration (known as "chicken bits") to disable or work around components that turn out to be defective.

Logic Synthesis

6

Logic synthesis converts register transfer level (RTL) to hardware by mapping the RTL (Verilog or VHDL) onto a library of cells. Synthesis attempts to minimize area and power while meeting timing constraints. Thus, the key metrics are performance, power, and area (PPA). More advanced (smaller feature size) fabrication processes offer better PPA, fueling the semiconductor industry’s virtuous cycle of better products at lower prices.

Logic synthesis begins by reading RTL files, constraints, standard cell libraries, and a technology file. Constraints include the floorplan (including chip area and blockages), switching activity factors, target clock frequency, clock skew, and characteristics of input and output pins (including delays, drivers, or loads). The *standard cell library* contains gates such as inverters, NAND2-NAND4 (2- to 4-input NAND gates), NOR2-NOR4 (2- to 4-input NOR gates), compound AND-OR-Invert gates, and flip-flops. A robust cell library has multiple drive strengths for each cell to efficiently drive different loads while optimizing PPA. An SoC also needs a library of memories and a library of input/output pad cells. Each cell in the library is characterized with delay and power for various loads under different voltage and temperature operating conditions. The cell also contains physical views suitable for placement and routing to build a chip. The technology file contains design rules such as the number of metal layers and the width and spacing of the metal and vias between layers.

Next, the synthesis tool *analyzes* the design to find syntax errors and translates it into an HDL-independent intermediate format. Then it *elaborates* the design to map it onto a technology-independent generic library including registers and arithmetic operations. It *compiles* the design to map it onto the actual cell library and optimizes the choice of cells attempting to meet timing while minimizing chip area. The compiler may *flatten* the design to optimize better across module boundaries, but this produces a giant blob of gates that is difficult for the designer to understand or improve. A *physical synthesis* tool also places cells and estimates wire lengths because wires significantly affect timing. The synthesis tool produces a mapped *netlist* that specifies the cells and their connections; timing,

- 6.1 Performance, Power, and Area (PPA)
- 6.2 Fabrication Technologies
- 6.3 Running Synthesis
- 6.4 Reviewing Log Files
- 6.5 Wally Synthesis Results
- 6.6 Summary

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

area, and power reports; and possibly placement information. After synthesis, a *place and route* (P&R) tool performs detailed placement and routing to produce a manufacturable chip layout.

As with most other CAD (computer-aided design) tools, logic synthesis is controlled by a Tcl (Tool Command Language) script that specifies files, constraints, and steps to perform.

This chapter begins with an overview of fabrication technologies, including three example CMOS (complementary metal oxide semiconductor) processes (Skywater 130 nm, Skywater 90 nm, and TSMC 28 nm: SKY130, SKY90, and TSMC28) and the characteristics of inverters, datapath components, and memories. It describes how to run the industry-standard Synopsys Design Compiler synthesis tool and interpret the results. Finally, it describes synthesizing the rv32e configuration of Wally onto SKY130 and TSMC28 and synthesizing the larger configurations onto TSMC28 with RAM (random access memory) libraries.

See Fig. 7.13 for a general block diagram of the rv32e configuration. However, note that rv32e does not include a DTIM or IROM; instead it goes to the AHB bus to access memory.

6.1 PERFORMANCE, POWER, AND AREA (PPA)

The primary figures of merit for synthesis are performance, power, and area (PPA). *Performance* is indicated by the clock period at which the system operates. Increasing performance improves the responsiveness and capabilities of a system. Products such as personal computer (PC) microprocessors are marketed largely on clock speed, and modest improvements can lead to billions of dollars of revenue. *Power* is the number of watts drawn; it includes both *active power* while the system is running and *leakage power* consumed when the clock is stopped. Power determines the battery size and life of mobile systems. Power is also important for systems plugged into the wall because it determines the cooling cost and environmental footprint. *Area* is the chip space used by the design, which also indicates the design cost. It is expressed in square microns for ASICs (application-specific integrated circuits) or numbers of logic elements and memories for field programmable gate arrays (FPGAs). Area dictates how many features (cores, memories, accelerators, and so on) can be placed on a profitable chip. These figures of merit are influenced by the RTL design, the fabrication technology and cell library (see Section 6.2), and the logic synthesis tool. The performance generally trades off against power and area.

6.1.1 Performance

Most synthesized designs are synchronous sequential circuits composed of combinational logic and clocked memory elements, in which all the clocked elements in a block share a common clock with a specified target frequency. The clocked elements are either flip-flops or other edge-triggered memories. The *clock period* is the time from one rising clock edge to the next. The *clock frequency* is the reciprocal of the clock period.

For example, if the clock period is 0.5 ns (0.5×10^{-9} s), the clock frequency is 2 GHz (2×10^9 Hz).

On the rising edge of the clock, each flip-flop's outputs may change. These new values propagate through cones of logic to reach the inputs to the next flip-flops. The new values must settle sometime before the next rising edge of the clock. This is called the *setup time constraint*. If new values reach the next flip-flop too soon after the clock edge, they might slip through and corrupt the result. This is called the *hold time constraint*. The rest of this section reviews these constraints and describes how logic synthesis timing analysis checks the constraints.

6.1.1.1 Timing Constraints

Fig. 6.1 shows the timing characteristics of synchronous sequential building blocks. The horizontal axis indicates time. Fig. 6.1(a) illustrates combinational logic (CL) with input(s) A and output(s) Y . The timing diagram shows that A has some initial value and changes to some new value. Y starts at an initial value and takes some time to respond. After a contamination delay t_{cd} , Y might start to change. For some time, the value of Y is unknown, indicated by the crosshatch pattern. After a propagation delay t_{pd} , Y has definitely settled to the new value. The contamination and propagation delays are the minimum and maximum values, indicated by gray and blue, respectively. They may differ because of input patterns, rising or falling edges, operating conditions such as voltage and temperature, manufacturing variation that affects transistor currents or capacitances, and aging that degrades transistors over time. Fig. 6.1(b) illustrates a flip-flop with a clock clk , input D , and output Q . Just as a camera needs the subject to be still when the shutter is pressed to prevent blurring, the flip-flop needs the input to be stable around the rising edge of the clock to be sampled reliably. Specifically, the input must stabilize at least t_{setup} before the rising clock edge and remain stable for at least t_{hold} after the rising clock edge. Like combinational logic, the output will not start changing until some contamination delay t_{ccq} and will settle to a final value after some propagation delay t_{pcq} from clock to Q .

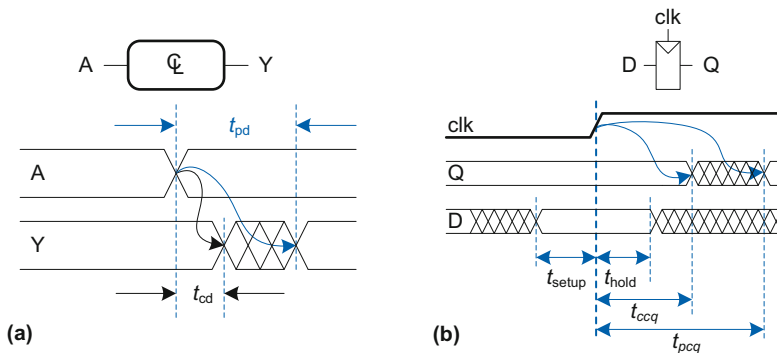


Fig. 6.1 Timing characteristics: (a) combinational logic and (b) flip-flops.

The *setup time constraint* determines the minimum reliable clock period, as illustrated in Fig. 6.2. A path in a synchronous sequential system starts at a register R1, goes through combinational logic, and ends at a register R2. The registers are made of one or more flip-flops sharing a common clock. When the clock rises, the first register's output $Q1$ will stabilize after t_{pcq} . The combinational logic then takes t_{pd} to reach the second register's input $D2$. This input must stabilize t_{setup} before the next clock edge. The clock period T_c must be long enough for all of this:

$$T_c \geq t_{pcq} + t_{pd} + t_{setup} \quad (6.1)$$

The *hold time constraint* determines whether an input to a flip-flop might change too soon after the clock edge, causing the system to malfunction, as illustrated in Fig. 6.3. On the rising edge of the clock, the first register output $Q1$ might change after t_{ccq} . The combinational logic might change after t_{cd} , reaching the input $D2$ to the second register. This input must not change for at least a hold time t_{hold} after the rising edge of the clock lest R2 sample the wrong value, become slower, or go metastable.

$$t_{ccq} + t_{cd} > t_{hold} \quad (6.2)$$

In a real system, clocks do not all switch at exactly the same time. The switching time might vary because of systematic errors (e.g., different length clock wires going to two registers), random manufacturing variation (e.g., a faster clock buffer in one path than another), and environmental variation (e.g., a voltage spike causing the clock buffers to momentarily speed up, shortening a clock cycle). These effects are collectively called *clock skew*.

Fig. 6.4 redraws the setup time constraint accounting for clock skew. Each clock edge has hashed lines indicating the range of times that it might

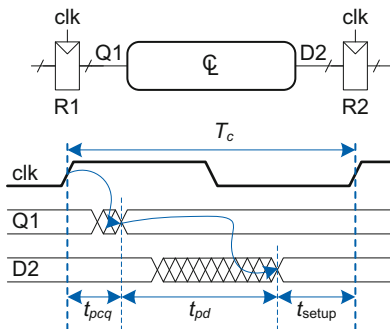


Fig. 6.2 Setup time constraint.

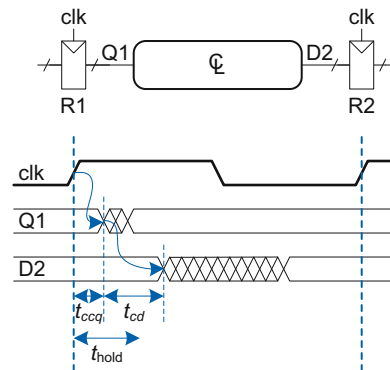


Fig. 6.3 Hold time constraint.

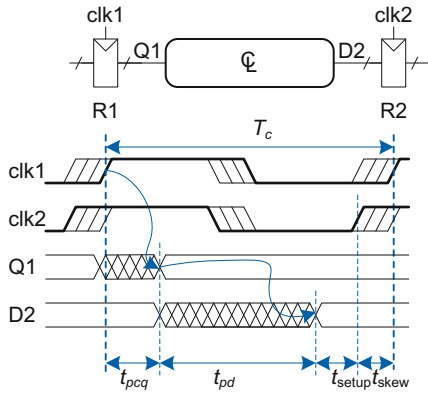


Fig. 6.4 Setup time constraint with clock skew.

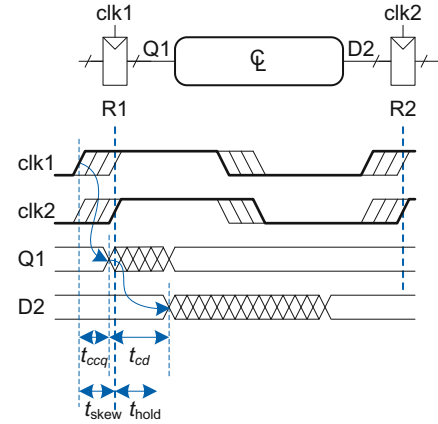


Fig. 6.5 Hold time constraint with clock skew.

switch. The clock skew t_{skew} is defined as the range between the earliest and latest that a clock edge might switch. In the worst case, clk1 rises on the latest skewed clock edge, and clk2 rises on the earliest skewed edge. After propagation delays through R1 and the combinational logic, D2 must set up in time. The clock skew makes the setup constraint more difficult to satisfy.

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} + t_{\text{skew}} \quad (6.3)$$

Fig. 6.5 redraws the hold time constraint accounting for clock skew. In the worst case, clk1 rises early and clk2 rises late. The skew effectively increases the hold time.

$$t_{ccq} + t_{cd} > t_{\text{hold}} + t_{\text{skew}} \quad (6.4)$$

Setup and hold times and clock skew on an SoC are all measured in picoseconds (10^{-12} s). Nevertheless, they are vitally important to achieve multigigahertz operation.

6.1.1.2 Static Timing Analysis

Synthesis performs *static timing analysis* (STA) to determine the worst-case timing by adding up the propagation delays along each path between clocked elements, which starts at the rising edge of the clock (*startpoint*), considered time = 0, passes through the Q output of the flip-flop, and propagates through a chain of combinational logic. The path must arrive at the D input of the next flip-flop (*endpoint*) some setup time before the next rising edge of the clock. The *arrival time* is the time at which the endpoint settles ($t_{cq} + t_{pd}$), and the *required time* is the clock period less the setup time ($T_c - t_{\text{setup}}$). The *slack* is the difference between the required time and the arrival time. If the slack is positive, the circuit operates properly at the desired frequency. If it is negative, the circuit cannot achieve the desired frequency.

In static timing analysis, t_{cq} is the maximum clock-to-q delay referred to earlier as t_{pcq} .

The *worst negative slack* (WNS) is the most negative slack of all paths, or zero if none are negative. The *total negative slack* (TNS) is the sum of all the negative slacks at each endpoint.

The *critical path* is the one with the WNS, limiting the clock period. The synthesis tool strives to reduce this critical path to improve performance. For example, if a gate must drive a high fanout (e.g., a select signal driving a 64-bit multiplexer), synthesis can choose a cell with a high *drive strength* that is larger but faster at driving the heavy load. If one signal arrives late on the critical path, the synthesis tool might precompute the output assuming that the signal is 0 and again assuming it is 1, then select the output from these two possibilities when the late signal arrives. Similarly, whereas a critical path might use a fast prefix adder, a noncritical path might use a smaller, cheaper ripple carry adder. Hence, critical paths tend to use faster structures that cost more area, and noncritical paths use the smallest implementations. Some processes offer low threshold voltage (low- V_t) transistors with better speed but exponentially more leakage power. Synthesis inserts low- V_t cells selectively on the critical paths. In a well-balanced system, many paths are nearly equally long, and there is no single outlier that could be optimized to speed up the whole system.

If synthesis cannot meet the target clock period, the *logic design engineer* may modify the RTL, simplifying the critical path or moving logic between pipeline stages. The *physical design engineer* may tweak the synthesis constraints, such as flattening the module hierarchy to enable more optimizations, preplacing modules so that critical cells are near each other, tuning the clock network, or enabling low- V_t cells. If none of these efforts succeed, the chip will still operate correctly but at a reduced clock frequency. Moreover, static timing analysis assumes worst-case environment and manufacturing. The chip will often run faster than analysis predicted.

For hold times, a similar analysis is performed using contamination delays (the minimum time through the combinational logic), but this analysis is independent of clock frequency. Hold time analysis is essential to ensure signal transitions do not compromise the endpoint flip-flop's sampled value. Hold time failures are catastrophic because they cause the chip to run unreliably at any speed; thus, these failures cannot be fixed by simply changing the clock speed. The hardest case to satisfy is back-to-back flip-flops with no logic between them ($t_{cd} = 0$). Synthesis sometimes must insert delay buffers to ensure hold time is satisfied. Alternatively, FPGAs often prevent hold time problems by providing a flip-flop that is so slow (big t_{ccq}) that the system is guaranteed to work, albeit slowly.

External inputs and outputs are assigned required arrival times relative to the clock edge to satisfy setup and hold time constraints.

PC enthusiasts know they can often overclock their CPU (central processing unit), especially if they have a good cooling system.

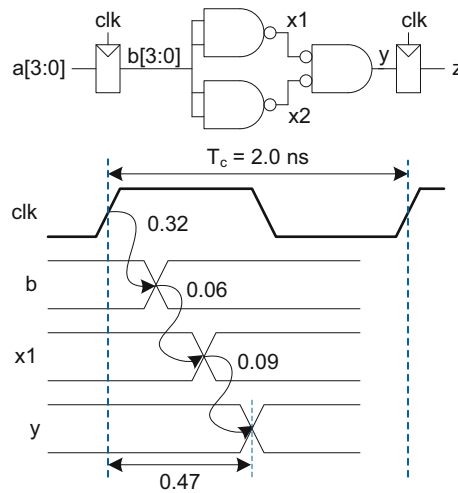


Fig. 6.6 Schematic annotated with timing.

Code Example 6.1 describes a synchronous sequential system with a 4-input AND gate between two flip-flops. Fig. 6.6 shows the schematic, where the AND gate is implemented with 2-input NAND and NOR gates. Suppose the flip-flops have a $t_{cq} = 0.32$ ns and a $t_{setup} = 0.13$ ns, and the NAND and NOR gates have propagation delays of 0.06 and 0.09 ns, respectively, as annotated in the schematic. If the target clock period is 2 ns, y has a required time of $2.0 - 0.13 = 1.87$ ns and an arrival time of $0.32 + 0.06 + 0.09 = 0.47$ ns. The path has slack of $1.87 - 0.47 = 1.4$ ns.

Code Example 6.1 EXAMPLE OF TIMING ANALYSIS

```
module example (
    input logic      clk,
    input logic [3:0] a,
    output logic      q);

    logic [3:0] b;
    logic      y;

    flop #(4) f1(clk, a, b);
    assign y = &b; // 4-input AND
    flop #(1) f2(clk, y, q);
endmodule
```

6.1.2 Power

As digital circuits toggle between 0 and 1, they consume *dynamic* power to charge and discharge capacitance on the nodes. Even when they are idle, they draw some *leakage* power because transistors never turn completely OFF.

Power is energy per time. If a system consumes energy E per clock cycle and the clock period is T_c , the power is

$$P = E/T_c \quad (6.5)$$

The clock frequency f is the reciprocal of the period T_c , so power is proportional to frequency:

$$P = Ef \quad (6.6)$$

All transistors and wires on a chip have some capacitance, generally on the order of a femtofarad for micron-scale structures ($1 \text{ fF} = 10^{-15} \text{ F}$). As you may recall from a physics course, the energy required to charge a capacitor from 0 to V_{DD} is proportional to the capacitance and quadratic with the voltage. This is called *dynamic energy* because it is caused by signals changing. Half of this energy is stored in the capacitor, and the other half is dissipated as heat during the charging process. When the capacitor discharges from V_{DD} back to 0, the remaining energy stored in the capacitor is dissipated as heat.

$$E_{dyn} = CV_{DD}^2 \quad (6.7)$$

A clock signal that rises and falls once every clock cycle with frequency f therefore consumes

$$P_{dyn} = E_{dyn}f = CV_{DD}^2f \quad (6.8)$$

In general, the dynamic power is proportional to the *activity factor* α , which measures the number of times the signal rises and falls each cycle. A clock has an activity factor of $\alpha = 1$ because it rises and falls once a cycle. A data signal that rises or falls only once in a clock cycle switches half as often and thus draws half the power, so $\alpha = 1/2$. A truly random data signal that rises or falls in half the cycles and keeps its old value in the other half has $\alpha = 1/4$. Most signals switch even less often; for example, only one or a few bits of the program counter change in a typical cycle, and the upper bits of an integer are often all 0s. Empirically, $\alpha = 0.1$ is a reasonable estimate for the activity of data signals, corresponding to a signal that rises or falls in 20% of the cycles. Activity factors are also called *toggle rates*.

$$P_{dyn} = \alpha CV_{DD}^2f \quad (6.9)$$

Dynamic power is computed for each component based on that component's activity factor and capacitance. The power of an entire block or chip is the sum of the power consumed by each component.

Transistors that are supposed to be OFF actually draw some leakage current I_{leak} . This leakage is predominantly subthreshold leakage between the source and drain, but tiny transistors also have some tunneling current through the gate as electronics quantum mechanically teleport across the ultrathin gate insulator. The leakage power is proportional to the leakage current and the supply voltage

$$P_{leak} = I_{leak}V_{DD} \quad (6.10)$$

In some processes and cells, the leakage current differs for HIGH vs. LOW outputs. However, over a large circuit, it is reasonable to estimate leakage as the average of these two cases.

When a chip is operating at a high frequency f , the dynamic power is usually at least an order of magnitude greater than the leakage power, so leakage can be ignored. However, when the chip is idle, it often stops the clock, so $f = 0$, and leakage accounts for all of the power. Mobile devices that interact with a user spend most of their time waiting for the user, and hence their standby battery life is determined by the leakage power.

Energy depends on the number and size of components, so it correlates with area. For example, the Wally floating-point divide/square root unit consumes an average of $0.28 \text{ fJ}/\mu\text{m}^2/\text{cycle}$ in the TSMC 28HPC+ (TSMC28) process under typical conditions. Hence, minimizing area is a rough proxy for minimizing power.

Modern SoCs use other knobs to optimize power [Keating07, Weste12]. Cell libraries with multiple drive sizes and threshold voltages allow the designer to trade dynamic or leakage power for speed on the critical path. Clock gating and power gating turn off dynamic and leakage power, respectively, of idle subsystems. Dynamic voltage and frequency scaling reduce the supply voltage and clock frequency to save power when a component does not need to operate at peak performance. Adaptive voltage scaling measures the real-time chip performance and adjusts the supply voltage to reduce the power wasted by unnecessarily large guard bands for manufacturing and environmental variations.

6.1.3 Area

Synthesis makes many design decisions trading performance against area. For example, whereas a ripple carry adder is slow but small, a prefix adder is larger but faster. Synopsys Design Compiler provides *DesignWare* arithmetic components that automatically select a good implementation based on timing constraints. As another example, cell libraries contain flavors of cells with different *drive strengths*. A cell with a larger drive strength has wider transistors that deliver more current and can drive a given capacitive load faster. However, the cell also takes more area and power and presents a larger capacitive load on the previous stage.

6.2 FABRICATION TECHNOLOGIES

Integrated circuit fabrication technologies are characterized by their *process node* or *feature size*. The $10\text{-}\mu\text{m}$ node through the 28-nm node, which were developed from 1972 until 2009, are known as *planar processes*, in which transistors are built from layers on a flat silicon wafer, including the silicon source, drain, and channel, an insulating gate dielectric, and a conductive gate. The process node of a planar process refers to the drawn length of the transistor gate. Processes typically scaled length and width by a factor of $1/\sqrt{2} \approx 0.7$ from one node to the next, such that each node packed twice as many transistors in the same area [Dennard74]. Smaller

transistors were not only cheaper but also faster and used less energy, leading to a cycle that drove the semiconductor revolution for four decades. According to Dennard scaling theory, delay scales proportionally to feature size and power (per gate) scales quadratically with feature size.

Vigorous worldwide competition led to new nodes every 2 to 3 years. Fig. 6.7 shows the progression of Intel process nodes from 10 μm to 10 nm and beyond, indicating a thousand-fold feature size reduction and a million-fold increase in transistor density over five decades or a doubling of transistor density every 2.5 years.

As transistors shrunk, the power supply voltage also scaled down to avoid damaging the tiny devices. To maintain performance, the transistor threshold voltage also decreased. Unfortunately, the leakage of OFF transistors increases exponentially as threshold voltage decreases. Below the 28-nm node, planar processes struggled to improve performance without excessive leakage power. Chip manufacturers turned to FinFETs (fin-shaped field-effect transistors), in which transistors are turned on their side and the gate is wrapped around three sides to reduce leakage. The process node is no longer linked to any physical dimension, but smaller nodes are now a marketing term for general advances in materials and fabrication that improve performance or power. For example, TSMC describes that, compared to its 5-nm process, its 3-nm process has 33% better transistor density and is 10% to 15% faster at the same power or 25% to 30% lower power at the same speed [Shilov20].

The technical details of more advanced nodes are closely guarded trade secrets. The 28-nm node is the latest node that universities can presently access at moderate cost with reasonable amounts of paperwork. For

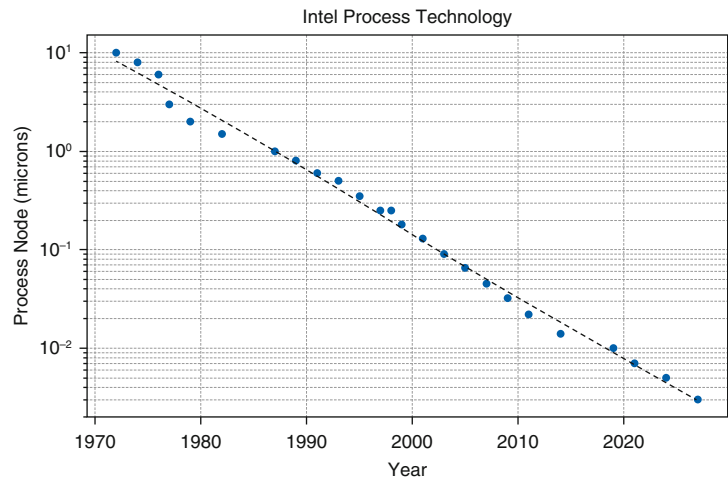


Fig. 6.7 Planar process feature size progression. (Data from [WikichipProcTech].)

example, IMEC supports universities and companies prototyping on the TSMC 28HPC+ process; the nondisclosure agreements require about three months to complete, and a batch of a hundred unpackaged 1-mm² chips costs about \$12,000. The NDA (nondisclosure agreement) forbids sharing certain details, so this book only shows overall results, not detailed 28-nm timing reports.

This book also illustrates synthesis in an older node with open-source access so that users can more easily reproduce the book's results. Cypress Semiconductor spun off its foundry business, which Skywater Technology acquired in 2017. Skywater partnered with Efabless and Google to provide open-source access and free chip fabrication for universities. The Skywater 130-nm process design kit (SKY130 PDK) is available at github.com/google/skywater-pdk. The Oklahoma State University cell library for Skywater 130 is available at foss-eda-tools.googleusercontent.com/skywater-pdk/libs/sky130_osu_sc_t12.

SKY130 has a discontinuity in the transistor I-V curve model that sometimes produces simulation artifacts. Skywater has a more advanced 90-nm process (SKY90) with more robust transistor models that was intended for open-source access in 2022. The scripts in this chapter support both SKY90 and SKY130. However, all results shown in this chapter are from a prerelease version of SKY90. Unfortunately, it appears that Google has abandoned the open-source release of the SKY90 PDK.

6.2.1 Inverter Characteristics

Table 6.1 summarizes key characteristics of these processes. Power and speed are given under typical process conditions at nominal supply voltage and room temperature (25°C). The track pitch is the width plus spacing of

Table 6.1 Typical inverter characteristics.

Characteristic	Units	SKY130	SKY90	TSMC28
Nominal supply voltage	V	1.8	1.2	0.9
Track pitch	μm	0.37	0.35	0.090
Cell height	tracks	12	8	9
INVX1 FO4 delay	ps	99.5	43.3	11.6
INVX1 area	μm ²	4.4	1.96	0.252
INVX1 leakage	nW	0.13	2.0	1.1
INVX1 input capacitance	fF	2.8	3.49	0.54

a layer 1 or 2 metal wire (metal1 or metal2), which can usually be found in the technology LEF (.tlef) file in the PDK. The cells in a library are typically the same height so that they snap together easily; this height is usually 7-12 metal tracks. An INVX1 has unit drive strength, while an INVX2 has twice the drive strength. An FO4 inverter is an inverter driving four identical copies of itself. The FO4 delay is the delay of such an inverter and a handy metric for the overall performance of a process. The theory of logical effort [Sutherland99] indicates that gates in well-designed fast circuits have delays comparable to an FO4 inverter.

6.2.1.1 Process Variation

All results in this book are reported under typical process, voltage, and temperature (PVT) conditions defined earlier. Real circuits must operate over a range of temperatures; for example, commercial-grade parts are tested in an environment from 0° to 70°C, while military-grade parts are tested from -55° to 125°C. The transistor junction temperatures can be substantially higher when the chip is drawing a lot of power. The supply voltage often varies by 10% or more because of voltage regulator error and power supply distribution noise. Random fluctuations in the manufacturing process produce transistors that may be faster or slower than typical (TT). Fast (FF) transistors have lower threshold voltages and hence more leakage, while slow (SS) transistors have the opposite. Leakage increases exponentially with temperature.

Table 6.2 summarizes INVX1 delay and leakage across these process, voltage, and temperature (PVT) corners, showing that leakage increases by two orders of magnitude from typical to hot and fast. Delay varies by roughly a factor of two between fast and slow extremes. Products must leave *margin* so that they achieve specifications despite these variations.

6.2.2 Datapath Units

Designers typically estimate the delay, area, and energy of functional units and memories when planning a design. Moreover, because fabrication technologies are constantly advancing, it is useful to express delay and area in units that are independent of technology so that insights from one design remain relevant in the next design. A typical metric is to list all delays relative to an FO4 inverter. Functional unit delay tends to track FO4 inverter delay remarkably well, although paths dominated by long wires do not.

Wally's `bin/libppa.pl` script parses Liberty files to extract delay, leakage, area, and capacitance.

Table 6.2 INVX1 delay and leakage variations.

Process	Voltage (V)	Temperature (°C)	SKY130		SKY90		TSMC28	
			FO4 Delay (ps)	INVX1 Leakage (nW)	FO4 Delay (ps)	INVX1 Leakage (nW)	FO4 Delay (ps)	INVX1 Leakage (nW)
TT	Nominal	−40			41.1	0.1		
TT	Nominal	25	99.5	0.013	43.3	2.0	11.6	1.1
TT	Nominal	85			44.4	15.5	11.6	12.2
TT	Nominal	125			45.1	43.3		
TT	−10%	25	118.7	0.008	50.6	1.3	14.4	0.7
TT	+10%	25	92.6	0.017	38.2	3.0	9.9	1.6
SS	Nominal	−40					13.0	0.0
SS	Nominal	25			60.7	0.3		
SS	Nominal	125					12.8	23.0
SS	−10%	−40	206	0.000	68.4	0.0	16.7	0.0
SS	−10%	25			70.3	0.2		
SS	−10%	125	182	0.28	71.0	8.1	15.1	17.6
FF	Nominal	−40					10.8	0.1
FF	Nominal	25			31.0	13.4		
FF	Nominal	125					10.8	80.5
FF	+10%	−40	64.6	0.025	26.7	1.4	9.0	0.2
FF	+10%	25			27.9	20.9		
FF	+10%	125	68.5	211.2	29.2	268.2	9.4	111.8

To characterize the delay, area, power, and energy for each block, many synthesis runs are performed, sweeping the target frequency around the maximum achievable. Fig. 6.8 shows the results for a 32-bit adder in SKY90. As target frequency increases, the achieved delay initially decreases, and the area increases (gray dots). Beyond some target frequency, the delay ceases improving significantly, and the area and delay become noisy (blue dots) as Design Compiler struggles to achieve an impossible target. Selecting the rightmost gray dot gives a reasonably stable design

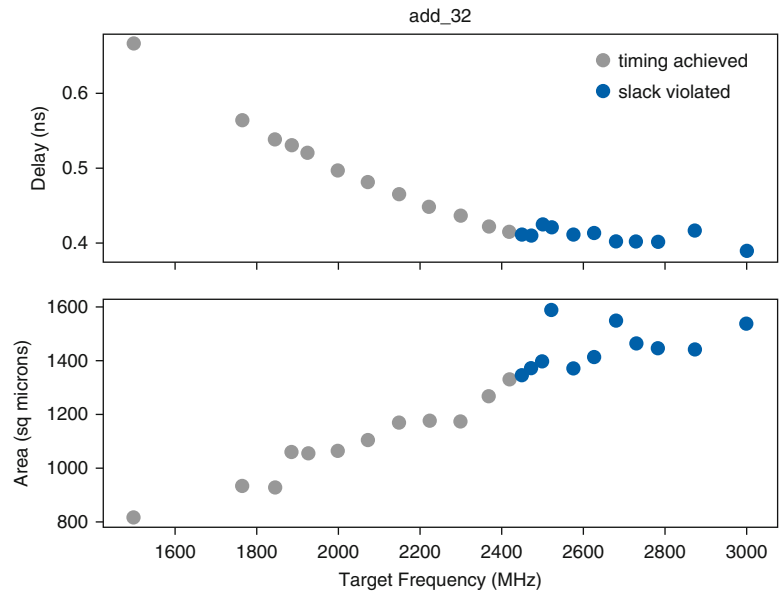


Fig. 6.8 32-bit adder area and delay in SKY90.

Table 6.3 32-bit fast adder characteristics at TT 25°C.

Characteristic	Units	SKY130	SKY90	TSMC28
Delay	ps	891	449	100
	FO4	9.0	10.4	8.6
Area	μm ²	2,581	1,441	209
Leakage Power	nW	18	714	1,060
Dynamic Energy	fJ/op	685	658	82

point to measure the best achievable delay and the resulting area and energy given that delay, listed in Table 6.3. We will refer to these as *fast add32 PPA*.

Dynamic energy is the total energy for the unit to perform one operation. It is measured using an activity factor of 1 for clocked nets and 0.1 for all other nets. Adders are combinational, so none of their nets are clocked. Dynamic energy is computed as the dynamic power times the clock period. Leakage power is measured with a static probability of 0.5, meaning each net has an equal chance of being 0 or 1. These activity factors are the Synopsys Design Compiler default values, listed as `power_default_toggle_rate` and `power_default_static_probability`.

Fig. 6.9 explores the PPA of other datapath building blocks, including comparators, priority encoders, simple left shifters, multipliers,

It is suspicious that SKY130 and SKY90 have similar dynamic energies (see Table 6.3) because energy is usually lower at a smaller feature size. It raises questions about whether the libraries are characterized accurately.

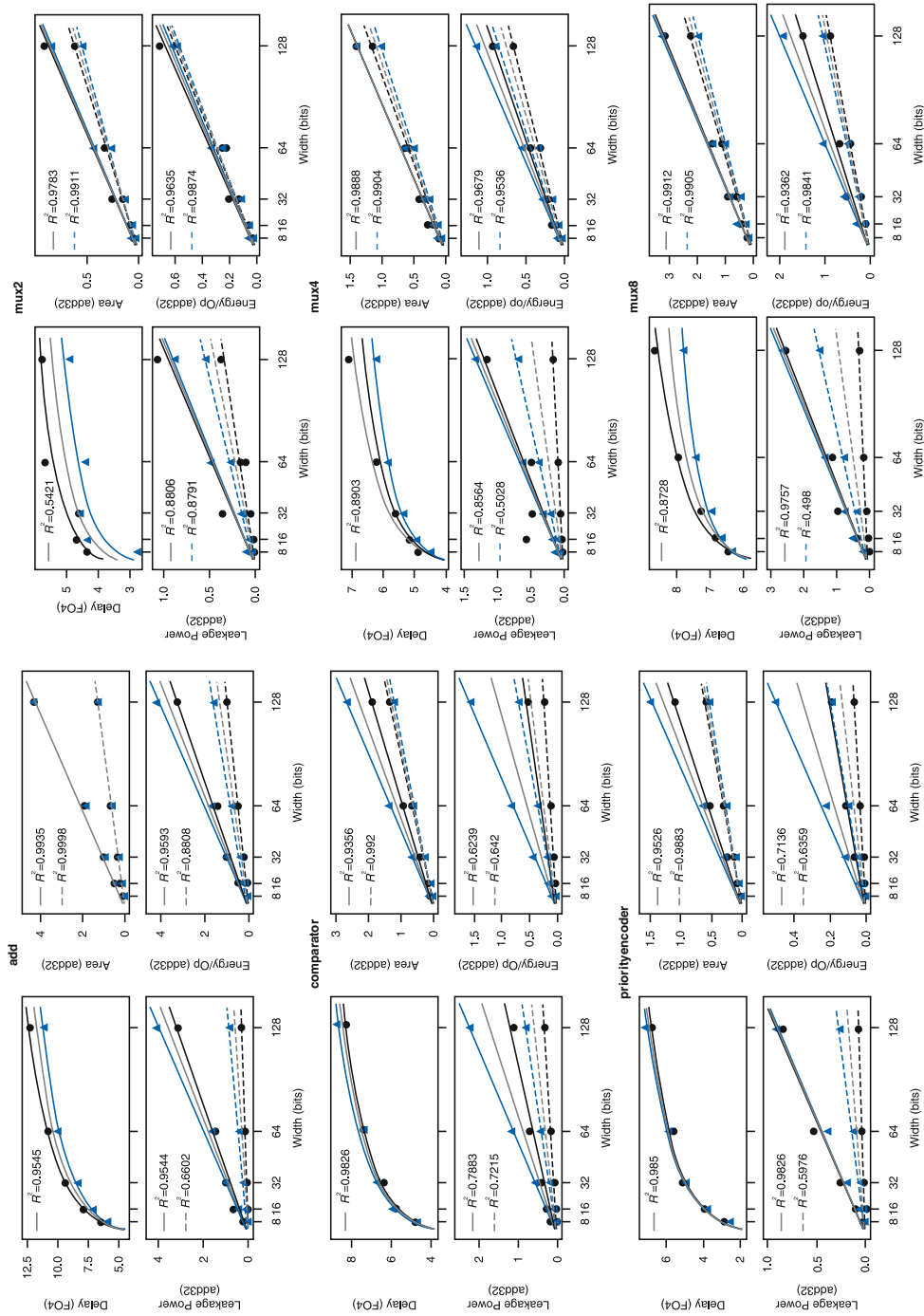


Fig. 6.9 Datapath component PPA.

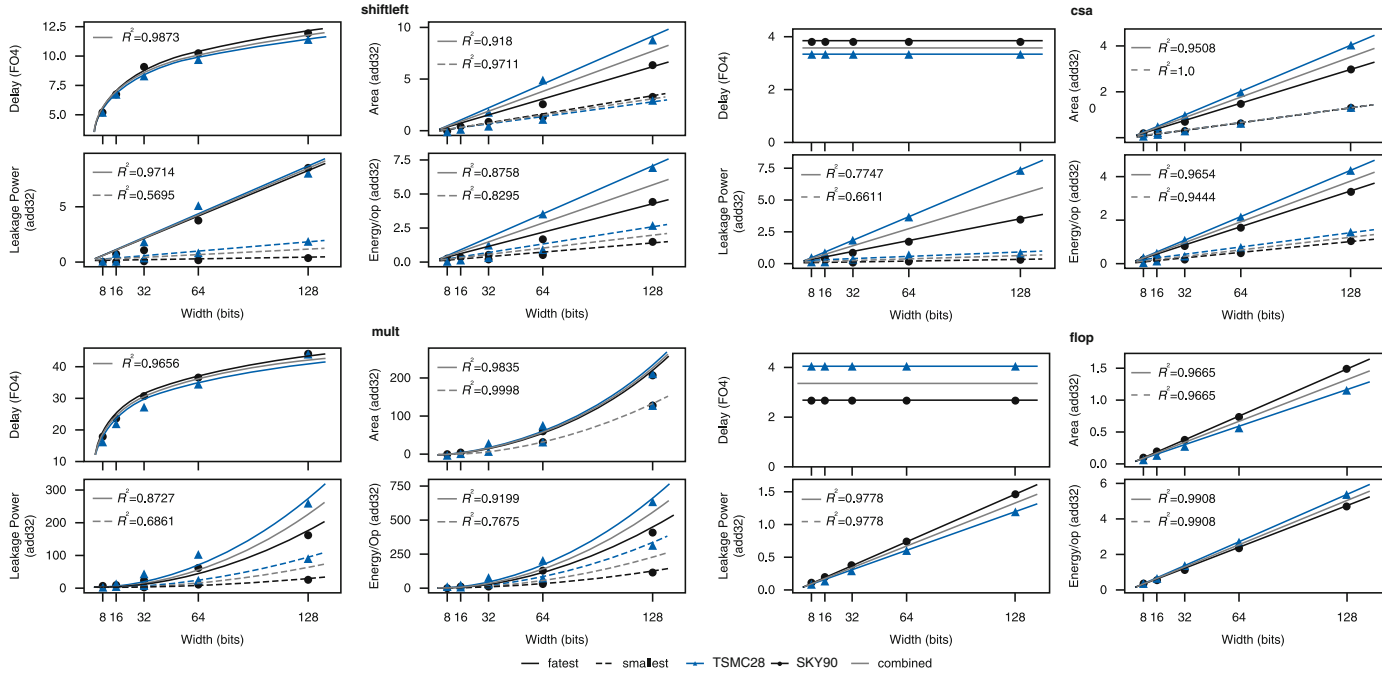


Fig. 6.9 Datapath component PPA. Cont'd

multiplexers, carry-save adders, and flip-flops. Each block has four graph panels, each containing synthesis results for 90- and 28-nm technologies, along with an average of the two. The x-axis lists size (8–128 bits). The delay panel shows the best achievable delay, as a function of size, expressed in FO4 inverter delays. The area panel shows the area as a function of size. The solid lines show the area for the design with the best achievable delay, and the dashed lines show the smallest achievable area when delay doesn't matter. For example, an adder uses a prefix structure to achieve best delay, but it uses a much more compact ripple structure if delay doesn't matter. Similarly, the leakage power and energy panels show results for best achievable delay and for smallest achievable area.

The R^2 fit values for the average delay, area, leakage, and dynamic energy are shown on the plots. For most results, they are close to 1, indicating that the average curve fit is a good match to both 90- and 28-nm results from 8 to 128 bits.

Table 6.4 summarizes the fits as a function of the number of inputs N . Most components have a best achievable delay involving a constant term and a term growing logarithmically with the component size. Carry-save adders (CSAs) and flip-flops naturally have a delay independent of the number of inputs. Area, leakage, and energy typically scale linearly with the normalized size $S = N/32$ except that they are quadratic for multipliers.

Table 6.4 Datapath element performance, power, and area (PPA).

Element	Best Delay (FO4)	At Best Delay (Relative to Fast add32)			At Lowest Area (Relative to Fast add32)		
		Area	Leakage	Energy	Area	Leakage	Energy
add	$1.9 + 1.4 \log_2 N$	$1.0S$	$0.89S$	$0.91S$	$0.32S$	$0.15S$	$0.33S$
comparator	$2.1 + 0.91 \log_2 N$	$0.58S$	$0.44S$	$0.27S$	$0.33S$	$0.15S$	$0.12S$
priorityencoder	$0.0 + 0.98 \log_2 N$	$0.32S$	$0.22S$	$0.087S$	$0.14S$	$0.044S$	$0.033S$
shiftright	$0.46 + 1.60 \log_2 N$	$1.9S$	$2.1S$	$1.4S$	$0.77S$	$0.28S$	$0.48S$
mult	$0.0 + 6.0 \log_2 N$	$13S^2$	$13S^2$	$33S^2$	$7.8S^2$	$3.5S^2$	$14S^2$
mux2	$2.6 + 0.41 \log_2 N$	$0.21S$	$0.23S$	$0.16S$	$0.14S$	$0.11S$	$0.14S$
mux4	$3.1 + 0.50 \log_2 N$	$0.35S$	$0.31S$	$0.26S$	$0.27S$	$0.11S$	$0.19S$
mux8	$4.9 + 0.46 \log_2 N$	$0.80S$	$0.66S$	$0.43S$	$0.53S$	$0.23S$	$0.24S$
csa	3.6	$0.87S$	$1.3S$	$0.95S$	$0.33S$	$0.15S$	$0.31S$
flop	3.3	$0.33S$	$0.33S$	$1.3S$	$0.33S$	$0.33S$	$1.3S$

Note that multiplexer delay is measured from the select input and grows with the number of inputs and bus width (N) because the select signal must drive many bits. The delay from the data input is essentially independent of the number of bits.

Key takeaways are that most datapath components have a delay that grows with the log of the width, except that flip-flops and carry-save adders have a delay of 3-4 FO4 independent of the number of inputs. Microarchitects can make early critical path estimates from these component delays. Datapath component area grows linearly with width (surprisingly, even shifters), except that multipliers grow quadratically. Microarchitects can also make early size estimates to help evaluate tradeoffs; for example, a comparator is about half the size of a fast adder, and a shifter is about twice the size.

6.2.3 Memories

Obtaining SRAM generators can be difficult, especially for academic designers. No Skywater generator is available. The TSMC generator entails various licensing hassles, and it produces a high aspect ratio with large column circuits and poor area utilization, so the area advantage is lower than expected. At the time of this writing, OpenRAM did not produce satisfactory RAM decoders in our processes of interest.

If a multiport RAM is unavailable, one can synthesize a register file from latches rather than flip-flops to save area. For example, the RISC-V Ibex core optionally uses latches in the register file. They do not play nicely with FPGAs or Verilator, which are optimized for flop-based designs (see ibex-core.readthedocs.io/en/latest/03_reference/register_file.html).

On-chip RAM can be built from flip-flops or static RAM (SRAM) cells. Flip-flops are easy to use and are readily available with standard logic synthesis, but their area and power consumption are far greater than those of SRAM. For example, in TSMC28, the flip-flop cell area is $1.89 \mu\text{m}^2$, while the SRAM bit cell area is $0.13 \mu\text{m}^2$. However, SRAM requires additional *peripheral circuitry*, including a precharge circuit for bitlines, sense amplifiers, and pitch-matched decoders. It is created with SRAM generators, provided by the fab or third-party IP vendors.

Similarly, on-chip ROM can be synthesized from combinational logic gates that provide the desired input/output relationship, or it can be created with ROM generators. We find that the TSMC28 ROM generators have so much overhead that logic synthesis normally produces smaller and faster results for a 128×32 -bit boot ROM.

Wally by default describes memories with `always_ff` statements as in [Section 4.2.3.3.2](#). When the `USE_SRAM` configuration is set, memories of specific sizes from [Table 6.5](#) are replaced with a memory wrapper module instantiation. The wrapper in turn should instantiate a compiled SRAM macro for the target process. The names of these cells may be considered confidential by the vendor, so they are not specified in this textbook or the code.

[Table 6.5](#) compares the flip-flop and memory modules used in Wally. The flop-based designs are much larger and more power hungry but are faster. SRAM has greatest density and power benefits for large arrays where the peripheral area is modest compared to the bit cell array. As shown in [Fig. 4.23](#), flop-based memories can have either an asynchronous or synchronous read. The SRAM modules are inherently synchronous with a read time involving an address setup t_{setup} before the clock edge and a clock-to-Q delay t_{cq} to the read data output. Wally uses memories with

Table 6.5 Memory block characteristics.

RAM	Application	Process	Style	Size (μm^2)	t_{setup} addr (ps)	t_{cq} read (ps)	Leakage (μW)	Dynamic Energy (pJ/access)
ram1p1rwbe_64x128 64×128 single port	Cache Data	SKY130	Flop	465k	152	1412	1.4	177
		SKY90	Flop	196k	46	536	38	74
		TSMC28	Flop	33k	13	212	162	26
			SRAM	7k	107	365	15	9
ram1p1rwbe_64x44 64×44 single port	Cache Tags	SKY130	Flop	164k	152	1354	0.56	61
		SKY90	Flop	69k	46	506	14	26
		TSMC28	Flop	12k	13	192	58	8
			SRAM	3k	107	305	6	4
ram2p1r1wbe_64x32 64×32 dual port	Branch history table (folded from 1024×2)	SKY130	Flop	117k	152	1287	0.41	45
		SKY90	Flop	49k	46	490	10	19
		TSMC28	Flop	9k	13	187	43	6
			SRAM	8k	50	459	20	4
ram2p1r1wbe_1024x68 1024×68 dual port	Branch target buffer	SKY130	Flop	2,245k	152	1,946	12	1270
		SKY90	Flop	1,653k	46	740	332	626
		TSMC28	Flop	261k	13	380	1,390	217
			SRAM	48k	72	550	84	10

synchronous read, as given in Code [Example 4.22](#), allowing either type of memory to be used in the same pipeline.

The SRAM numbers in [Table 6.5](#) come from the memory compiler datasheet directory. The size comes from the Area entry. t_{cq} is the access time (t_{cd}) entry. t_{setup} is the address setup (t_{as}) entry. Leakage is the total leakage current (in μA) multiplied by the 0.9 V supply voltage. Dynamic energy is the read dynamic current (in $\mu\text{A}/\text{MHz}$) multiplied by V_{DD} times the period of 1 MHz (1 μs). For example, the 64×128 -bit RAM has leakage current of 16.77 μA and thus leakage power of $16.77 \mu\text{A} \times 0.9 \text{ V} = 15.1 \mu\text{W}$. It has dynamic read current of 11.5 $\mu\text{A}/\text{MHz}$ and thus dynamic energy of $11.5 \mu\text{A}/\text{MHz} \times 0.9 \text{ V} \times 10^{-6} \text{ s} = 10.35 \text{ pJ/read}$.

Dynamic energy per read is actually multiplied by the read frequency (in MHz), but that cancels with the read period (in μs). For example, if a read occurs at a frequency of 2 MHz, the read period is $1/(2 \text{ MHz}) = 1/2 \mu\text{s}$. So, the energy per read is: $11.5 \mu\text{A}/\text{MHz} \times 2 \text{ MHz/read} \times (0.9 \text{ V}) \times (1/2 \mu\text{s})$, which is still 10.35 pJ/read.

The flop-based numbers in Table 6.5 come from Design Compiler synthesis reports. Edit the RAM module default parameters to specify the size. The design is synthesized with an impossibly fast target (10 GHz) to estimate the best speed.

```
$ make synth DESIGN=ram1p1rwbe TECH=sky90 CONFIG=syn_rv64gc FREQ=10000 USESRAM=0
```

The area and leakage numbers come straight from synthesis reports. The dynamic energy is the reported dynamic power (in mW) multiplied by the 100-ps target period. The t_{cq} read delay comes from the timing report. t_{setup} is obtained as the setup time of a flip-flop from another synthesis run such as `wallypipelinedcore` (see Section 6.3).

6.3 RUNNING SYNTHESIS

Run synthesis with:

```
$ cd $WALLY/synthDC
$ make synth DESIGN=wallypipelinedcore TECH=sky130 CONFIG=syn_rv32e FREQ=330
```

The CONFIG options are described in Section 6.5.1.

The run results can be large. You should delete the contents of the run directory (`rm -rf runs`) periodically to save disk space unless you need to refer to old runs.

The Makefile uses `.synopsys_dc.setup` and `scripts/synth.tcl`, both of which are necessary to read to follow this section. The Makefile sets the top-level design to `wallypipelinedcore` and configuration to `rv32e`. The target technology is SKY130. The target frequency is 330 MHz, a little faster than achievable. Synthesis takes only a few minutes to run because the `rv32e` configuration is simple. In contrast, synthesizing the `rv64gc` configuration with 16-KiB caches can take more than an hour. Watch for warnings and errors in the messages printed.

The synthesis results are stored in `runs` within a directory named for the design, configuration, technology, frequency, date, and git hash for future reference. The scripts copy all the Verilog code into a subdirectory called `hdl`. The first time synthesis is run, it analyzes the Liberty (`.lib`) file, which describes the cell library, and caches the analyzed library in a directory called `alib-52`.

The scripts copy the specified configuration into the `config` subdirectory. As discussed in Section 4.2.5.6, the parameterized top-level module requires a wrapper to load the `config.vh` file and `parameter-defs.vh`. This file (e.g., `wallypipelinedcorewrapper`) is automatically generated by `synth.tcl` using `wrapperGen.py` in the `run hdl` subdirectory if the top-level design module contains the keyword `cvw_t`. `synth.tcl` synthesizes the wrapper to pass the parameters to the top-level module but generates reports for the top-level module (e.g., `wallypipelinedcore`).

The `synth.tcl` script mostly uses commands that are custom to Design Compiler, but it also uses native Tcl commands to read environment variables, copy files, and perform actions conditionally. The key Design Compiler commands are `analyze`, `elaborate`, and `compile_ultra`, which read the Verilog and run synthesis.

If you have access to TSMC28 libraries, run synthesis to get much higher frequency and lower area than in the SKY130 process. Invoke with:

```
$ make synth DESIGN=wallypipelinedcore TECH=tsmc28 CONFIG=syn_rv32e FREQ=3000
```

With suitable technology files (called Milkyway and TLU+), Design Compiler can run *physical synthesis*, creating an internal model of cell placement at the same time it maps Verilog onto logic gates. This allows Design Compiler to estimate wire lengths and optimize appropriately, producing area and delay that agree with final placement and routing results to within about 5%. This synthesis is also called *topographical mode*. TSMC28 results in this chapter are produced using topographical mode for best accuracy. The Makefile specifies `TECH=tsmc28` for no physical synthesis or `TECH=tsmc28psyn` for physical synthesis.

When the USESRAM configuration is set, the synthesis scripts will also load the memory libraries from [Section 6.2.3](#) for most compact cache RAM, and branch predictor tables. Invoke with:

```
$ make synth DESIGN=wallypipelinedcore TECH=tsmc28psyn CONFIG=syn_sram_rv32gc FREQ=3000 USESRAM=1
```

The Design Compiler User Guide is proprietary. Licensed institutions can download it from solvnetsynopsys.com. If you are serious about optimizing your synthesis results, it is worth reading.

6.4 REVIEWING LOG FILES

cd into `$WALLY/synthDC/runs/RUN`, where `RUN` is the name of the synthesis you just completed. The directory contents are shown in [Fig. 6.10](#). The rest of this section describes the logs from `wally-pipelinedcore_rv32e_orig_sky130nm_330_MHz...` generated in [Section 6.3](#).

The `config` directory contains the configuration file (e.g., `config/rv32e/config.vh`) used in synthesis. The other files shown in [Fig. 6.10](#) are described in the following sections.

6.4.1 `synth.out`

`synth.out` is the output log file that gives warnings and error messages. [Fig. 6.11](#) shows some of the messages in the log file for each of the stages of compilation. Mapping and optimization complete many passes to improve WNS and area. See the added comments in blue to indicate what is occurring at each step.

To check out the Verilog code used in a previous run, use `git checkout HASH`, where `HASH` is a code such as `b4ff90e1` at the end of the name of the run directory. The code is in `src`. You can reproduce the simulation by running `make synth` again with the parameters from the run directory name, for example, `make synth b4ff90e1`. Run `git checkout main` to return to the main branch.

The reports in this section were generated around January 2024. In spring 2024, a Flush signal slightly increased the length of the critical path to the external bus unit (EBU). We anticipate that path may be fixed later. We also anticipate rv64gc area will increase slightly as extensions such as Zfa are implemented.

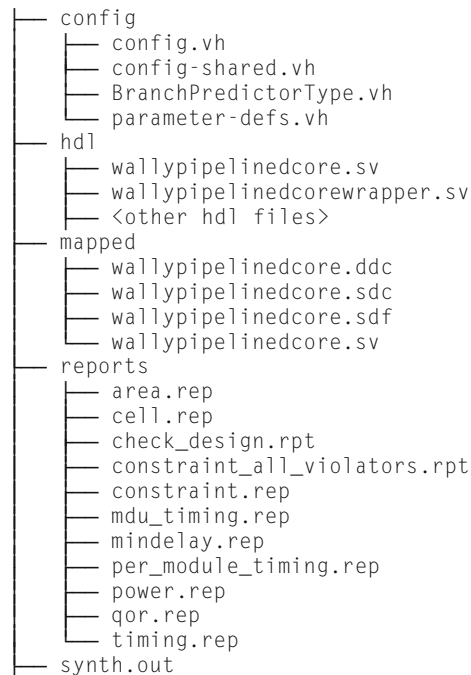


Fig. 6.10 run directory contents.

6.4.2 Mapped Output Files

The mapped directory has several outputs of synthesis.

This section uses `DESIGN` to indicate the top-level module name, which, in this example, is `wallypipelinedcore`.

Design Compiler warns that `clk` is a high-fanout net because it drives so many flip-flops. Placement tools will handle it with a special clock-routing algorithm designed to balance delay.

- ▶ **DESIGN.sdc:** Synopsys Design Constraints file; uses Synopsys Tcl commands to provide the design constraints, including the clock period, driver and load cells, delays from primary inputs and outputs, and maximum fanout
- ▶ **DESIGN.sdf:** Standard Delay Format file; gives the design timing suitable for further timing analysis
- ▶ **DESIGN.ddc:** an encrypted binary database file; contains the synthesis results that Synopsys tools load for further analysis
- ▶ **DESIGN.sv:** contains the synthesized Verilog netlist. By default, our scripts run hierarchical synthesis, preserving the same module hierarchy and signal names as in the original Verilog. This makes interpreting the results feasible, but it also prevents cross-module optimizations, which is described in [Section 6.5.2](#).

```
# Synthesis Synopsys Flow
. . .
analyze -f sverilog -lib WORK $my_verilog_files      # Compile all source files
Compiling source file ./runs/wallypipelinedcore_rv32e_orig_sky130nm_330_MHz_2023-11-03-07-
38__5e84d5e6/hdl/wallypipelinedcorewrapper.sv
Opening include file ./runs/wallypipelinedcore_rv32e_orig_sky130nm_330_MHz_2023-11-03-07-
38__5e84d5e6/hdl/config.vh
Compiling source file ./runs/wallypipelinedcore_rv32e_orig_sky130nm_330_MHz_2023-11-03-07-
38__5e84d5e6/hdl/wallypipelinedcore.sv
. . .
# Target SKY130 (130 nm library), TT, 1.08 V, 25 C
Presto compilation completed successfully.
Loading db file
'/opt/riscv/cad/lib/sky130_osu_sc_t12/12T_ms/lib/sky130_osu_sc_12T_ms_TT_1P8_25C.ccs.db'
Loading db file '/cad/synopsys/SYN/libraries/syn/dw_foundation.sldb'
elaborate $my_toplevel -lib WORK
Loading db file '/cad/synopsys/SYN/libraries/syn/gtech.db'
Loading db file '/cad/synopsys/SYN/libraries/syn/standard.sldb'
  Loading link library 'sky130_osu_sc_12T_ms_TT_1P8_25C.ccs '
  Loading link library 'gtech'
Running PRESTO HDLC
Presto compilation completed successfully. (wallypipelinedcore)
Elaborated 1 design.
Current design is now 'wallypipelinedcorewrapper'.
Information: Building the design 'wallypipelinedcore' instantiated from design
'wallypipelinedcorewrapper' with the parameters
. . .
Inferred memory devices in process      # Found flip-flops in design
in routine flopenl_WIDTH32 line 33 in file
'./runs/wallypipelinedcore_rv32e_orig_sky130nm_330_MHz_2023-11-03-07-38__5e84d5e6/hdl/flopenl.sv'.
=====
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=====
| q_reg | Flip-flop | 32 | Y | N | N | N | N | N | N |
=====
. . .
Linking design 'wallypipelinedcorewrapper'
Using the following designs and libraries:
-----
sky130_osu_sc_12T_ms_TT_1P8_25C.ccs (library)
/opt/riscv/cad/lib/sky130_osu_sc_t12/12T_ms/lib/sky130_osu_sc_12T_ms_TT_1P8_25C.ccs.db
dw_foundation.sldb (library) /cad/synopsys/SYN/libraries/syn/dw_foundation.sldb
. . .
=====
| DesignWare Building Block Library | Version | Available |
=====
| Basic DW Building Blocks | R-2020.09-DWBB_202009.5 | * |
| Licensed DW Building Blocks | R-2020.09-DWBB_202009.5 | * |
=====
# Problems are related to unused signals in the configurable design
Information: There are 2472 potential problems in your design. Please run 'check_design' for more
information. (LINT-99)
. . .
Information: Uniquified 2 instances of design 'flopenl_WIDTH32'. (OPT-1056)
# Uniquify instances to optimize each one separately
Beginning Pass 1 Mapping
-----
Processing 'ifu'      # Map modules onto cell library
. . .
```

Fig. 6.11 synth.out log.

```

Updating timing information
Information: Updating design information... (UID-85)
Information: Removing unused design 'mux2_WIDTH3_1'. (OPT-1055)
Information: The register 'ifu/PCPDReg/q_reg[0]' is a constant and will be removed. (OPT-1206)
# Remove unused registers
Beginning Mapping Optimizations (Ultra High effort) # Start synthesis
-----
Mapping Optimization (Phase 1)
Mapping Optimization (Phase 2)
Mapping Optimization (Phase 3) # First try is big and slow
-----
      ELAPSED      AREA      WORST NEG      TOTAL      DESIGN
      TIME        AREA      SLACK    SETUP    RULE COST
      -----
0:00:11    104050.7      0.37      196.9      0.0
0:00:11    103514.9      0.80      460.9      0.0
      -----
Beginning Constant Register Removal
-----
Information: The register 'ieu/c/controlregW/q_reg[1]' is a constant and will be removed. (OPT-
1206)
# No reason for flops on constant values.
. . .
Beginning Global Optimizations
-----
Numerical Synthesis (Phase 1)
Numerical Synthesis (Phase 2)
Global Optimization (Phase 1)
Global Optimization (Phase 2)
. . .
Information: Added key list 'DesignWare' to design 'alu_32_165949'. (DDB-72)
# Designware Libraries have optimized components
Beginning Delay Optimization # Get faster
-----
      0:00:12    96295.4      0.76      406.6      0.0      384.4135
      0:00:14    97328.8      0.26      124.8      0.0      389.7814
      0:00:14    97328.8      0.26      124.8      0.0      389.7814
      0:00:14    97328.8      0.26      124.8      0.0      389.7814
Information: The register 'dut/ieu/c/controlregW/q_reg[2]' is a constant and will be removed.
(OPT-1206)
Information: The register 'dut/ieu/c/controlregW/q_reg[0]' is a constant and will be removed.
(OPT-1206)
      0:00:14    97102.2      0.26      124.8      0.0      389.5597
      0:00:14    96738.9      0.43      172.3      0.0      388.6815
Beginning WLM Backend Optimization # WLM is wire load model
-----
Beginning Leakage Power Optimization (max_leakage_power 0)
-----
      ELAPSED      AREA      WORST NEG      TOTAL      DESIGN
      TIME        AREA      SLACK    SETUP    RULE COST
      -----
      0:00:28    96382.3      0.14      55.4      0.0      472.3618
. . .
Optimization Complete
-----
Memory usage for this session 165 Mbytes.
Memory usage for this session including child processes 165 Mbytes.
CPU usage for this session 40 seconds ( 0.01 hours ).
Elapsed time for this session 42 seconds ( 0.01 hours ).
# Elapsedtime < CPU usage if using multicore processing

```

Fig. 6.11 synth.out log. Cont'd

Code Example 6.2 shows the hazard unit's behavioral Verilog (on page 274), and Code Example 6.3 shows the synthesized netlist (on page 275). Many of the signals have been optimized out of the netlist because the rv32e configuration does not have a multiply and divide unit (MDU), floating-point unit (FPU), or privileged unit. Hence many inputs, including MDUStallD, FPUStallD, and all the trap and control and status register (CSR) signals, are tied to 0 and can be ignored.

Design Compiler has a Design Vision tool that displays netlists as schematics. Invoke it using:

```
$ design_vision-xg
```

Use File → Read and open the DDC file (e.g., in runs/<current_run>/mapped/wallpipelinedcore.ddc). Navigate to the hazard unit (hazard) in the Logical Hierarchy pane, then use Schematic → New Schematic View. Double click on the symbol in the schematic to see the gates inside. Fig. 6.12 shows the synthesized schematic. Observe that it exactly matches the netlist from Code Example 6.3.

The #1 delays in the Verilog in Code Example 6.2 are included for simulation to help see causality. They are ignored by synthesis.

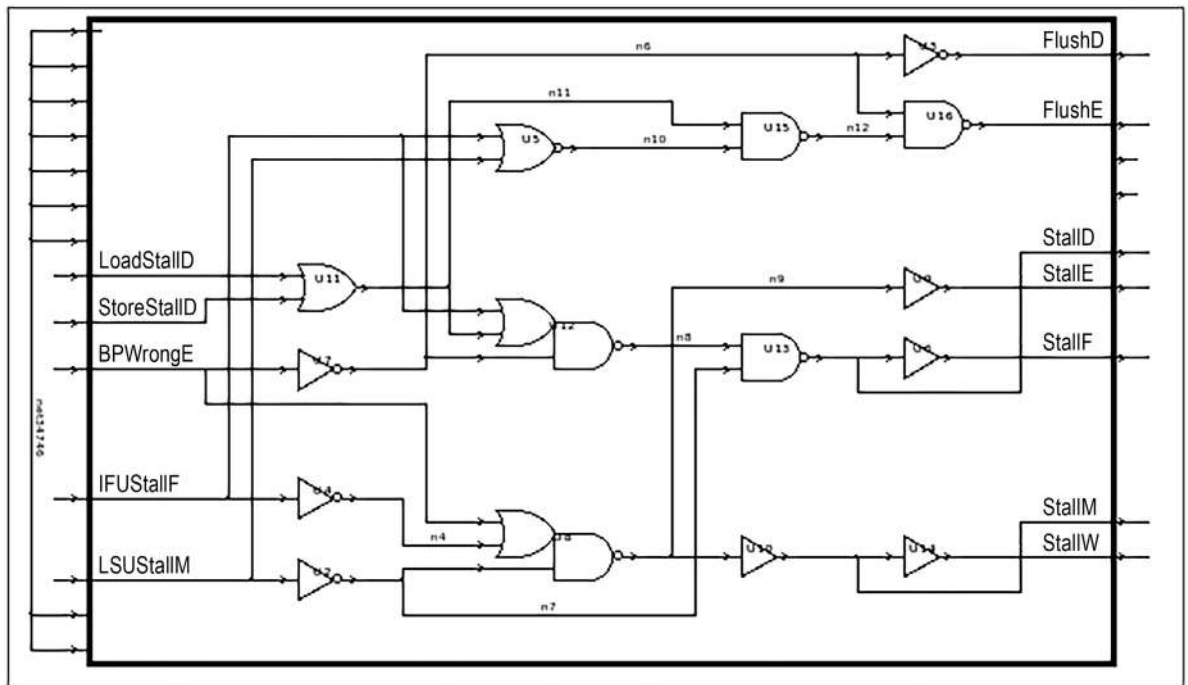


Fig. 6.12 Hazard unit (hazard) synthesized schematic.

Code Example 6.2 hazard.sv

```

module hazard (
    // Detect hazards
    input logic BPWrongE, CSRWriteFenceM, RetM, TrapM,
    input logic LoadStallD, StoreStallD, MDUStallD, CSRRdStallD,
    input logic LSUStallM, IFUStallF,
    input logic FCvtIntStallD, FPUStallD,
    input logic DivBusyE, FDivBusyE,
    input logic EcallFaultM, BreakpointFaultM,
    input logic wfiM, IntPendingM,
    // Stall & flush outputs
    output logic StallF, StallD, StallE, StallM, StallW,
    output logic FlushD, FlushE, FlushM, FlushW
);

    logic StallFCause, StallDCause, StallECause, StallMCause, StallWCause;
    logic LatestUnstalledD, LatestUnstalledE, LatestUnstalledM, LatestUnstalledW;
    logic FlushDCause, FlushECause, FlushMCause, FlushWCause;

    logic WFIStallM, WFIInterruptedM;

    // WFI logic
    assign WFIStallM = wfiM & ~IntPendingM;           // WFI waiting for an interrupt or timeout
    assign WFIInterruptedM = wfiM & IntPendingM;       // WFI detects a pending interrupt.

    // Stalls and flushes
    assign FlushDCause = TrapM | RetM | CSRWriteFenceM | BPWrongE;
    assign FlushECause = TrapM | RetM | CSRWriteFenceM |(BPWrongE & ~(DivBusyE | FDivBusyE));
    assign FlushMCause = TrapM | RetM | CSRWriteFenceM;
    assign FlushWCause = TrapM & ~WFIInterruptedM;

    // Stall causes
    assign StallFCause = '0;
    assign StallDCause = (LoadStallD | StoreStallD | MDUStallD | CSRRdStallD | FCvtIntStallD |
                          FPUStallD) & ~FlushDCause;
    assign StallECause = (DivBusyE | FDivBusyE) & ~FlushECause;
    assign StallMCause = WFIStallM & ~FlushMCause;
    assign StallWCause = (IFUStallF & ~FlushDCause) | (LSUStallM & ~FlushWCause);

    // Stall each stage for cause or if the next stage is stalled
    assign #1 StallF = StallFCause | StallD;
    assign #1 StallD = StallDCause | StallE;
    assign #1 StallE = StallECause | StallM;
    assign #1 StallM = StallMCause | StallW;
    assign #1 StallW = StallWCause;

    // Detect the first stage that is not stalled
    assign LatestUnstalledD = ~StallD & StallF;
    assign LatestUnstalledE = ~StallE & StallD;
    assign LatestUnstalledM = ~StallM & StallE;
    assign LatestUnstalledW = ~StallW & StallM;

    // Flush stage if the previous stage is the last one stalled or the system has reason to flush
    assign #1 FlushD = LatestUnstalledD | FlushDCause;
    assign #1 FlushE = LatestUnstalledE | FlushECause;
    assign #1 FlushM = LatestUnstalledM | FlushMCause;
    assign #1 FlushW = LatestUnstalledW | FlushWCause;
endmodule

```

Code Example 6.3 `wallypipelinedcore.sv` NETLIST HAZARD MODULE

```

module hazard ( BPWrongE, CSRWriteFenceM, RetM, TrapM, LoadStallD, StoreStallD,
               MDUStallD, CSRRdStallD, LSUStallM, IFUStallF, FCvtIntStallD, FPUStallD,
               DivBusyE, FDivBusyE, EcallFaultM, BreakpointFaultM, wfiM, IntPendingM,
               StallF, StallD, StallE, StallM, StallW, FlushD, FlushE, FlushM, FlushW
);
  input BPWrongE, CSRWriteFenceM, RetM, TrapM, LoadStallD, StoreStallD,
        MDUStallD, CSRRdStallD, LSUStallM, IFUStallF, FCvtIntStallD, FPUStallD,
        DivBusyE, FDivBusyE, EcallFaultM, BreakpointFaultM, wfiM,
        IntPendingM;
  output StallF, StallD, StallE, StallM, StallW, FlushD, FlushE, FlushM,
        FlushW;
  wire n4, n6, n7, n8, n9, n10, n11, n12;

  sky130_osu_sc_12T_ms__inv_1 U2 ( .A(LSUStallM), .Y(n7) );
  sky130_osu_sc_12T_ms__inv_3 U3 ( .A(n6), .Y(FlushD) );
  sky130_osu_sc_12T_ms__inv_1 U4 ( .A(IFUStallF), .Y(n4) );
  sky130_osu_sc_12T_ms__nor2_1 U5 ( .A(IFUStallF), .B(LSUStallM), .Y(n10) );
  sky130_osu_sc_12T_ms__buf_1 U6 ( .A(StallD), .Y(StallF) );
  sky130_osu_sc_12T_ms__inv_2 U7 ( .A(BPWrongE), .Y(n6) );
  sky130_osu_sc_12T_ms__oai21_1 U8 ( .A0(BPWrongE), .A1(n4), .B0(n7), .Y(n9) );
  sky130_osu_sc_12T_ms__buf_4 U9 ( .A(n9), .Y(StallE) );
  sky130_osu_sc_12T_ms__buf_2 U10 ( .A(n9), .Y(StallM) );
  sky130_osu_sc_12T_ms__or2_1 U11 ( .A(LoadStallD), .B(StoreStallD), .Y(n11) );
  sky130_osu_sc_12T_ms__oai21_1 U12 ( .A0(IFUStallF), .A1(n11), .B0(n6), .Y(n8) );
  sky130_osu_sc_12T_ms__nand2_1 U13 ( .A(n8), .B(n7), .Y(StallD) );
  sky130_osu_sc_12T_ms__buf_2 U14 ( .A(StallM), .Y(StallW) );
  sky130_osu_sc_12T_ms__nand2_1 U15 ( .A(n11), .B(n10), .Y(n12) );
  sky130_osu_sc_12T_ms__nand2_1 U16 ( .A(n6), .B(n12), .Y(FlushE) );
endmodule

```

6.4.3 Quality of Results (QoR) Reports

Quality of results (QoR) reports summarize key timing and area results. Timing metrics include the critical path slack (WNS), TNS, and any hold time violations. Area metrics include the number and area of combinational, sequential, and buffer/inverter cells. For example, `qor.rep` in [Fig. 6.13](#) shows timing and area results for Wally `rv32e` in the SKY130 process. As highlighted in blue text, the critical path goes through 37 levels of logic. The target clock period is 3.03 ns, but the design has 0.15 ns of negative slack, so it does not meet timing and has a best achievable clock period of 3.18 ns. The area is 95,313 μm^2 , or about 0.3 mm on a side. The processor is built from 8,226 cells, of which 1,173 are sequential (flip-flops), 2,069 are inverters or buffers, and the rest are logic gates. Synthesis took about 2 minutes. Synthesis involves a certain amount of random choice, and thus results vary slightly from run to run.

```

*****
Report : qor
Design : wallypipelinedcore__165949
Version: R-2020.09-SP5-5
Date   : Fri Nov  3 07:39:11 2023
*****

Timing Path Group 'clk'
-----
Levels of Logic:          37.000000
Critical Path Length:     3.030967
Critical Path Slack:      -0.152755
Critical Path Clk Period: 3.030303
Total Negative Slack:     -57.774506
No. of Violating Paths:   710.000000
Worst Hold Violation:     0.000000
Total Hold Violation:     0.000000
No. of Hold Violations:   0.000000
-----

Cell Count
-----
Hierarchical Cell Count:    78
Hierarchical Port Count:   6697
Leaf Cell Count:           8226
Buf/Inv Cell Count:        2069
Buf Cell Count:            335
Inv Cell Count:            1734
CT Buf/Inv Cell Count:     0
Combinational Cell Count:  7053
Sequential Cell Count:     1173
Macro Count:               0
-----

Area
-----
Combinational Area:        52579.678278
Noncombinational Area:    42734.023876
Buf/Inv Area:              10614.396889
Total Buffer Area:         2400.485956
Total Inverter Area:       8213.910933
Macro/Black Box Area:     0.000000
Net Area:                  0.000000
-----

Cell Area:                  95313.702154
Design Area:                95313.702154

Design Rules
-----
Total Number of Nets:      8614
Nets With Violations:      0
Max Trans Violations:      0
Max Cap Violations:        0
-----

Design WNS: 0.152755 TNS: 57.774506 Number of Violating Paths: 710
Design (Hold) WNS: 0.000000 TNS: 0.000000 Number of Violating Paths: 0
-----

```

Fig. 6.13 qor.rep.

6.4.4 Performance Reports

The synthesis script uses the `report_timing` command to generate `timing.rep` that lists the worst overall timing path and `per_module_timing.rep` that lists the longest path through each of the various modules. For rv32e, the longest path involves a branch comparison in the Execute stage triggering a branch that leads to a stall or flush. In SKY90 and TSMC28, the stall endpoint is longest. In SKY130, the flush path is 2 ps slower than the stall path, but the flush path is more complicated to describe because it involves canceling an EBU transaction. Therefore, this section describes the stall critical path.

Fig. 6.14 annotates the stall path listed in the `timing.rep` report in Fig. 6.15 using SKY130. Fig. 6.14 is a diagram of the rv32e core adapted from Fig. 7.13. All times are in nanoseconds and capacitances in picofarads, as

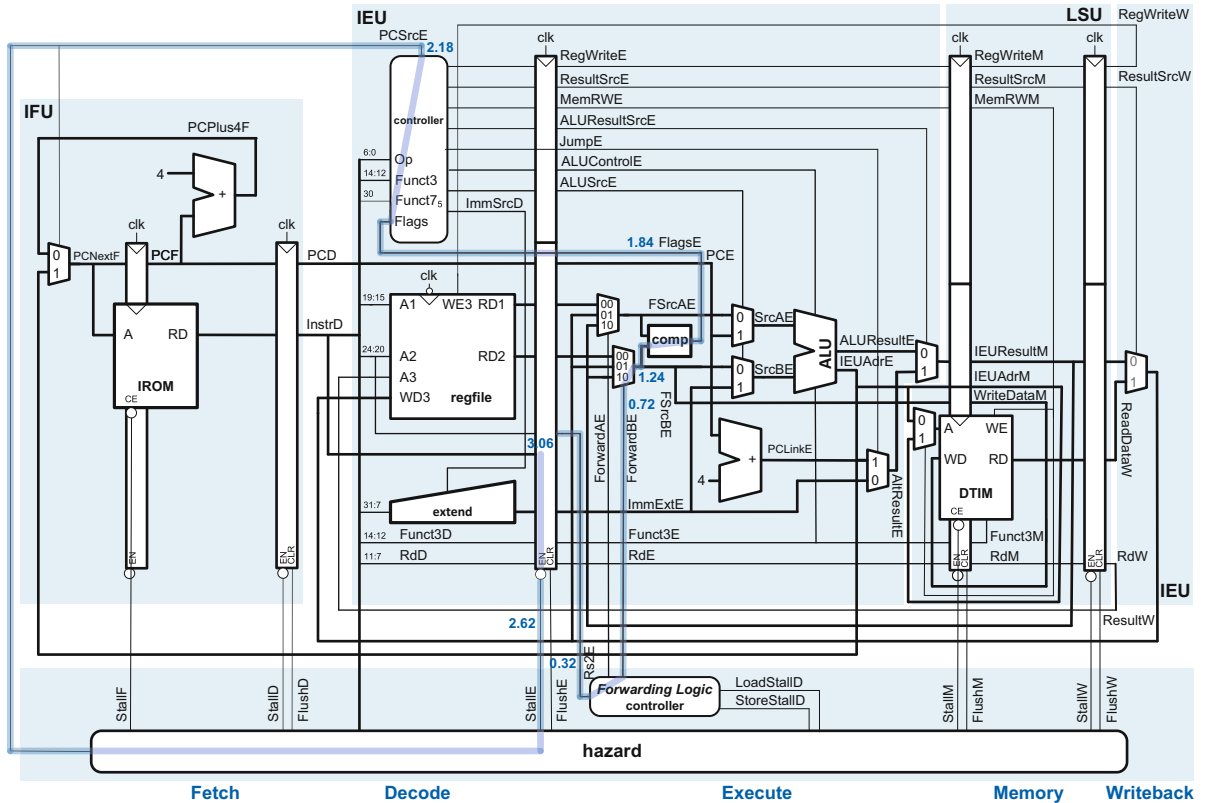


Fig. 6.14 rv32e core with critical path annotated in blue.

A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: PVT_1P8V_25C Library: skyl30_osu_sc_12T_ms_TT_1P8_25C.ccs

Wire Load Model Mode: top

Startpoint: ieu/dp/Rs2EReg/q_reg[0]
 (rising edge-triggered flip-flop clocked by clk)
 Endpoint: ifu/InstrEReg/q_reg[31]
 (rising edge-triggered flip-flop clocked by clk)
 Path Group: clk
 Path Type: max

Point	Fanout	Cap	Trans	Incr	Path
clock clk (rise edge)				0.000000	0.000000
clock network delay (ideal)				0.000000	0.000000
ieu/dp/Rs2EReg/q_reg[0]/CK (skyl30_osu_sc_12T_ms__dff_1)		0.000000	0.000000	#	0.000000 r
ieu/dp/Rs2EReg/q_reg[0]/Q (skyl30_osu_sc_12T_ms__dff_1)		0.080838	0.320341		0.320341 f
ieu/dp/Rs2EReg/q[0] (net)	4	0.013949	0.000000		0.320341 f
ieu/dp/Rs2EReg/q[0] (flopenrc_WIDTH5_3)			0.000000		0.320341 f
ieu/dp/Rs2E[0] (net)		0.013949	0.000000		0.320341 f
ieu/dp/Rs2E[0] (datapath__395949)			0.000000		0.320341 f
ieu/Rs2E[0] (net)		0.013949	0.000000		0.320341 f
ieu/fw/Rs2E[0] (forward)			0.000000		0.320341 f
ieu/fw/Rs2E[0] (net)		0.013949	0.000000		0.320341 f
ieu/fw/U36/Y (skyl30_osu_sc_12T_ms__xnor2_1)			0.089581	0.111006	0.431348 r
ieu/fw/n19 (net)	1	0.002578	0.000000		0.431348 r
ieu/fw/U35/Y (skyl30_osu_sc_12T_ms__nand2_1)			0.056313	0.070386	0.501734 f
ieu/fw/n54 (net)	2	0.004396	0.000000		0.501734 f
ieu/fw/U34/Y (skyl30_osu_sc_12T_ms__inv_1)			0.038368	0.051942	0.553675 r
ieu/fw/n21 (net)	1	0.002665	0.000000		0.553675 r
ieu/fw/U10/Y (skyl30_osu_sc_12T_ms__nand2_1)			0.035893	0.048400	0.602075 f
ieu/fw/n4 (net)	1	0.002499	0.000000		0.602075 f
ieu/fw/U9/Y (skyl30_osu_sc_12T_ms__nor2_1)			0.119280	0.118555	0.720630 r
ieu/fw/ForwardBE[1] (net)	2	0.005788	0.000000		0.720630 r
ieu/fw/ForwardBE[1] (forward)			0.000000		0.720630 r
ieu/ForwardBE[1] (net)		0.005788	0.000000		0.720630 r
ieu/dp/ForwardBE[1] (datapath__395949)			0.000000		0.720630 r
ieu/dp/ForwardBE[1] (net)		0.005788	0.000000		0.720630 r
ieu/dp/fbemux/s[1] (mux3_WIDTH32_0)			0.000000		0.720630 r
...					
ieu/dp/fbemux/y[26] (mux3_WIDTH32_0)				0.000000	1.237343 f
ieu/dp/n126 (net)		0.009574	0.000000		1.237343 f
ieu/dp/comp/b[26] (comparator_WIDTH32)			0.000000		1.237343 f
...					
ieu/dp/comp/U272/Y (skyl30_osu_sc_12T_ms__oi21_1)			0.052593	0.063912	1.839484 f
ieu/dp/comp/flags[0] (net)	1	0.002488	0.000000		1.839484 f
ieu/dp/comp/flags[0] (comparator_WIDTH32)			0.000000		1.839484 f
ieu/dp/FlagsE[0] (net)		0.002488	0.000000		1.839484 f
ieu/dp/FlagsE[0] (datapath__395949)			0.000000		1.839484 f
ieu/FlagsE[0] (net)		0.002488	0.000000		1.839484 f
ieu/c/FlagsE[0] (controller__395949)			0.000000		1.839484 f
ieu/c/FlagsE[0] (net)		0.002488	0.000000		1.839484 f
ieu/c/branchflagmux/d1[0] (mux2_WIDTH1)			0.000000		1.839484 f
ieu/c/branchflagmux/d1[0] (net)		0.002488	0.000000		1.839484 f
ieu/c/branchflagmux/U1/Y (skyl30_osu_sc_12T_ms__nand2_1)			0.055557	0.051331	1.890814 r
ieu/c/branchflagmux/n3 (net)	1	0.002578	0.000000		1.890814 r

Fig. 6.15 timing.rep.

ieu/c/branchflagmux/U4/Y (sky130_osu_sc_12T_ms__nand2_1)	0.052831	0.063409	1.954223	f
ieu/c/branchflagmux/y[0] (net)	1	0.004722	0.000000	1.954223 f
ieu/c/branchflagmux/y[0] (mux2_WIDTH1)			0.000000	1.954223 f
ieu/c/BranchFlagE (net)		0.004722	0.000000	1.954223 f
ieu/c/U33/Y (sky130_osu_sc_12T_ms__xnor2_1)		0.056597	0.100257	2.054480 f
ieu/c/n22 (net)	1	0.002418	0.000000	2.054480 f
ieu/c/U34/Y (sky130_osu_sc_12T_ms__oai21_1)		0.128619	0.122617	2.177096 r
ieu/c/PCSrcE (net)	2	0.004966	0.000000	2.177096 r
ieu/c/PCSrcE (controller__395949)			0.000000	2.177096 r
ieu/PCSrcE (net)		0.004966	0.000000	2.177096 r
ieu/PCSrcE (ieu__395949)			0.000000	2.177096 r
PCSrcE (net)		0.004966	0.000000	2.177096 r
ifu/PCSrcE (ifu__395949)			0.000000	2.177096 r
ifu/PCSrcE (net)		0.004966	0.000000	2.177096 r
ifu/U3/Y (sky130_osu_sc_12T_ms__buf_1)		0.054478	0.107858	2.284954 r
ifu/BPWrongE (net)	1	0.004894	0.000000	2.284954 r
ifu/BPWrongE (ifu__395949)			0.000000	2.284954 r
BPWrongE (net)		0.004894	0.000000	2.284954 r
hzu/BPWrongE (hazard__395949)			0.000000	2.284954 r
hzu/BPWrongE (net)		0.004894	0.000000	2.284954 r
hzu/U6/Y (sky130_osu_sc_12T_ms__inv_2)		0.037596	0.048914	2.333868 f
hzu/n5 (net)	4	0.011301	0.000000	2.333868 f
hzu/U7/Y (sky130_osu_sc_12T_ms__nand2_1)		0.037941	0.046295	2.380163 r
hzu/n6 (net)	1	0.002578	0.000000	2.380163 r
hzu/U8/Y (sky130_osu_sc_12T_ms__nand2_1)		0.100068	0.096502	2.476665 f
hzu/StallM (net)	4	0.011132	0.000000	2.476665 f
hzu/U13/Y (sky130_osu_sc_12T_ms__buf_2)		0.054383	0.144729	2.621394 f
hzu/StallE (net)	4	0.016288	0.000000	2.621394 f
hzu/StallE (hazard__395949) <-			0.000000	2.621394 f
StallE (net)		0.016288	0.000000	2.621394 f
ifu/StallE (ifu__395949)			0.000000	2.621394 f
ifu/StallE (net)		0.016288	0.000000	2.621394 f
ifu/U6/Y (sky130_osu_sc_12T_ms__inv_2)		0.073454	0.077434	2.698829 r
ifu/n256 (net)	6	0.015610	0.000000	2.698829 r
ifu/InstrEReg/en (flopenr_WIDTH32_1)			0.000000	2.698829 r
ifu/InstrEReg/en (net)		0.015610	0.000000	2.698829 r
ifu/InstrEReg/U8/Y (sky130_osu_sc_12T_ms__nand2_1)		0.055165	0.067654	2.766483 f
ifu/InstrEReg/n3 (net)	1	0.004619	0.000000	2.766483 f
ifu/InstrEReg/U9/Y (sky130_osu_sc_12T_ms__inv_2)		0.065370	0.071717	2.838200 r
ifu/InstrEReg/n21 (net)	1	0.013498	0.000000	2.838200 r
ifu/InstrEReg/U4/Y (sky130_osu_sc_12T_ms__inv_6)		0.078840	0.089298	2.927498 f
ifu/InstrEReg/n106 (net)	32	0.077421	0.000000	2.927498 f
ifu/InstrEReg/U108/Y (sky130_osu_sc_12T_ms__oai22_1)		0.120045	0.128082	3.055580 r
ifu/InstrEReg/n66 (net)	1	0.002852	0.000000	3.055580 r
ifu/InstrEReg/q_reg[31]/D (sky130_osu_sc_12T_ms__dff_1)		0.120045	0.000000	3.055580 r
data arrival time				3.055580
clock clk (rise edge)		3.030303	3.030303	
clock network delay (ideal)		0.000000	3.030303	
ifu/InstrEReg/q_reg[31]/CK (sky130_osu_sc_12T_ms__dff_1)		0.000000	3.030303	r
library setup time		-0.152149	2.878155	
data required time			2.878155	

data required time			2.878155	
data arrival time			-3.055580	

slack (VIOLATED)			-0.177425	

Fig. 6.15 timing.rep. Cont'd

defined in the Liberty (.lib) file. The path involves a branch being taken and then fetching the next instruction accordingly. It starts at the `Rs2E` source register and ends at the `InstrEReg` register: the forwarding logic selects an input to the branch comparator, which generates branch flags; the control unit then calculates $PCSrc = 1$, which the hazard unit uses to compute the `StallE` input to `InstrEReg`. Many nearly equal critical paths exist, so if you repeat this experiment, you can expect a different endpoint and a few picoseconds variation in timing.

The report has one line (point) for each cell or net (wire) along the path. Cells are described by the library name, cell type, and drive strength. For example, `sky130_osu_sc_12T_ms__inv_4` is an inverter from the OSU SKY130 12-track medium-speed standard cell library with a drive strength of 4. Higher drive cells are faster at driving large loads but are larger, have more input capacitance, and consume more power. Design Compiler optimizes drive strength to try meeting the target cycle time at the lowest area cost. The *fanout* and *cap* columns list the number of cells and total capacitance that a net drives. *Trans* (transition) time indicates how long a cell output takes to rise or fall, typically from 20% to 80% of V_{DD} . *Incr* gives the incremental delay contributed by that cell. The *Path* column sums all incremental delays along the path so far. The *r* or *f* at the end of the line indicates whether the signal is rising or falling.

As highlighted in blue in Fig. 6.15, this report shows that the clock rises at time 0. After a 0.32-ns clock-to-Q delay through the D flip-flop (dff), the `Rs2E[0]` signal falls. This ripples through five gates (`xnor2`, `nand2`, `inv`, `nand2`, `nor2`) to compute the `ForwardBE[1]` forwarding multiplexer control signal at 0.72 ns. The forwarding mux output is available at 1.24 ns. The 32-bit comparator involves 10 levels of logic, completing at 1.84 ns. The output passes through various levels of the hierarchy and changes names to `ieu/FlagsE[0]`; this name changing doesn't cause delay but is shown to help you follow the logic. The branch logic computes `PCSrcE`, which rises at 2.18 ns, indicating that the branch is taken. The `StallE` signal falls at 2.62 ns because the taken branch does not stall the Execute stage but causes a flush instead. Finally, after traversing the enable/clear logic, the `D` input of the `InstrEReg` flip-flop arrives at 3.06 ns. The flip-flop has a setup time of 0.15 ns. The target clock period in this example is 3.03 ns (333 MHz), so the `D` value is required at $3.03 - 0.15 = 2.88$ ns. Hence, the circuit has negative slack, and the target frequency cannot be met: $slack = 2.88 - 3.06 = -0.18$ ns. The actual achievable frequency is $1/(3.03 + 0.18 \text{ ns}) = 311$ MHz.

The `synth.tcl` script generates a variety of other timing reports specific to various modules, which are helpful when optimizing a particular

An astute reader may notice that the timing report shows `InstrEReg`'s `D` input rising, but this scenario causes `D` to fall (because a pipeline flush clears the contents of the register to 0). This is called a *false path* because it is logically impossible. The timing analyzer does not understand such logical correlations between signals and simply looks for the worst-case scenario. It is possible to annotate paths as false to reduce timing conservatism, but the falling `D` path is unlikely to be significantly better.


```

Startpoint: lsu/ReadDataMWReg/q_reg[5]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint:  lsu/ReadDataMWReg/q_reg[5]
            (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: min

```

Point	Fanout	Cap	Trans	Incr	Path
clock clk (rise edge)				0.000000	0.000000
clock network delay (ideal)				0.000000	0.000000
lsu/ReadDataMWReg/q_reg[5]/CK (sky130_osu_sc_12T_ms__dff_1)			0.000000	0.000000 #	0.000000 r
lsu/ReadDataMWReg/q_reg[5]/Q (sky130_osu_sc_12T_ms__dff_1)			0.047256	0.196276	0.196276 r
lsu/ReadDataMWReg/q[5] (net)	2	0.003854		0.000000	0.196276 r
lsu/ReadDataMWReg/U11/Y (sky130_osu_sc_12T_ms__mux2_1)			0.035681	0.024804	0.221080 r
lsu/ReadDataMWReg/n6 (net)	1	0.002566		0.000000	0.221080 r
lsu/ReadDataMWReg/q_reg[5]/D (sky130_osu_sc_12T_ms__dff_1)			0.035681	0.000000	0.221080 r
data arrival time					0.221080
clock clk (rise edge)				0.000000	0.000000
clock network delay (ideal)				0.000000	0.000000
lsu/ReadDataMWReg/q_reg[5]/CK (sky130_osu_sc_12T_ms__dff_1)				0.000000	0.000000 r
library hold time				-0.050442	-0.050442
data required time					-0.050442
data required time					-0.050442
data arrival time					-0.221080
slack (MET)					0.271522

Fig. 6.16 mindelay.rep.

module. The `report_timing -nworst` option can request the top *N* critical paths for a design, but often the other paths are just variations of the same path involving different bits of a bus.

The timing analyzer computes minimum delays through components when checking hold time. To get a min-delay report that shows hold time checks, uncomment the `report_timing` line in `synth.tcl` that includes `-delay_type min`. The `mindelay.rep` file in Fig. 6.16 shows the shortest path and hold time constraint. This path is from the output back to the enabled input of the `ReadDataMWReg`. The hold time is -0.050 ns, and the contamination delay (earliest arrival time) is 0.221 ns, so the hold time is easily met with 0.271 ns of slack.

Design Compiler refers to clock skew as *clock uncertainty*, which degrades both max-delay and min-delay constraints because the clock might arrive at a flip-flop either early or late. Change the uncertainty to 0.1 ns in `synth.tcl` by modifying `set my_uncertainty 0` to `0.1`, and then rerun synthesis. Fig. 6.17 shows the setup slack has worsened by about 0.12 ns compared to Fig. 6.15 because of the uncertainty. Observe that the endpoint has also changed to a different register and that the arrival time is now 3.08 ns rather than 3.05 ns because of some randomness in

The `set_critical_range` command tells synthesis to keep some margin such that less critical paths don't accidentally appear as the most critical. This helps the designer recognize which paths are truly critical and which were just relaxed until they became critical. The `synth.tcl` file sets a critical range of 50 ps to emphasize the real critical paths by attempting to give 50 ps of slack to noncritical paths. Turning off `set_critical_range` before the final synthesis run could save a little area and power for these less critical paths.

```

ieu/dp/ImmExtEReg/q_reg[18]/D (sky130_osu_sc_12T_ms__dff_1) 0.121576 0.000000 3.081019 r
data arrival time 3.081019

clock clk (rise edge) 3.030303 3.030303
clock network delay (ideal) 0.000000 3.030303
clock uncertainty -0.100000 2.930303
ieu/dp/ImmExtEReg/q_reg[18]/CK (sky130_osu_sc_12T_ms__dff_1) 0.000000 2.930303 r
library setup time -0.152505 2.777798
data required time 2.777798
-----
data required time 2.777798
data arrival time -3.081019
-----
slack (VIOLATED) -0.303221

```

Fig. 6.17 timing.rep excerpt showing setup time with uncertainty.

Point	Fanout	Cap	Trans	Incr	Path
clock clk (rise edge)				0.000000	0.000000
clock network delay (ideal)				0.000000	0.000000
lsu/ReadDataMWReg/q_reg[2]/CK (sky130_osu_sc_12T_ms__dff_1)			0.000000	0.000000 #	0.000000 r
lsu/ReadDataMWReg/q_reg[2]/Q (sky130_osu_sc_12T_ms__dff_1)			0.041536	0.191974	0.191974 r
lsu/ReadDataMWReg/q[2] (net)	2	0.003163		0.000000	0.191974 r
lsu/ReadDataMWReg/U8/Y (sky130_osu_sc_12T_ms__mux2_1)			0.034672	0.023973	0.215947 r
lsu/ReadDataMWReg/n3 (net)	1	0.002566		0.000000	0.215947 r
lsu/ReadDataMWReg/q_reg[2]/D (sky130_osu_sc_12T_ms__dff_1)			0.034672	0.000000	0.215947 r
data arrival time					0.215947
clock clk (rise edge)				0.000000	0.000000
clock network delay (ideal)				0.000000	0.000000
clock uncertainty				0.100000	0.100000
lsu/ReadDataMWReg/q_reg[2]/CK (sky130_osu_sc_12T_ms__dff_1)				0.000000	0.100000 r
library hold time				-0.050201	0.049799
data required time					0.049799
data required time					0.049799
data arrival time					-0.215947
slack (MET)					0.166148

Fig. 6.18 mindelay.rep showing hold time with uncertainty.

the synthesis process. Fig. 6.18 shows that hold slack has also decreased by 0.105 ns compared to Fig. 6.16 (from 0.271 to 0.166 ns).

During placement and routing, CAD tools may insert delay buffers to fix hold time violations.

6.4.5 Power Reports

The power.rep file in Fig. 6.19 shows the power in each module. Dynamic power is categorized as *switching power* and *internal power*. Switching power accounts for the load capacitance (gates and wires driven by each cell),

```

*****
Report : power
        -hier
        -analysis_effort low
Design : wallpipelinedcore__395949
Version: R-2020.09-SP5-5
Date   : Thu Nov 16 07:18:39 2023
*****

Library(s) Used:

    sky130_osu_sc_12T_ms_TT_1P8_25C.ccs (File:
/opt/riscv/cad/lib/sky130_osu_sc_t12/12T_ms/lib/sky130_osu_sc_12T_ms_TT_1P8_25C.ccs.db)

Operating Conditions: PVT_1P8V_25C   Library: sky130_osu_sc_12T_ms_TT_1P8_25C.ccs
Wire Load Model Mode: top

Global Operating Voltage = 1.8
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW   (derived from V,C,T units)
    Leakage Power Units = 1nW

```

Hierarchy	Switch Power	Int Power	Leak Power	Total Power	%
wallpipelinedcore__395949	0.820	5.302	715.939	6.122	100.0
hzu (hazard__395949)	7.04e-03	1.90e-03	1.674	8.95e-03	0.1
ebu.ebu (ebu_XLEN32_PA_BITS34_AHBW32)	3.79e-02	0.177	10.097	0.215	3.5
lsu (lsu__395949)	5.16e-02	0.609	32.021	0.661	10.8
ieu (ieu__395949)	0.458	3.299	602.129	3.758	61.4
ifu (ifu__395949)	0.264	1.213	69.435	1.477	24.1

Fig. 6.19 power.rep.

and internal power accounts for the capacitance inside each cell. Both are expressed in units of mW. The leakage power is given in units of nW. The power is calculated at the target clock frequency even if the circuit has negative slack.

Observe that the leakage power is 715 nW compared to dynamic power of $(5.302 + 0.820) = 6.122$ mW. In other words, leakage power is 0.012% of the dynamic power. When a system is operating at full speed, leakage is typically negligible. However, leakage influences battery life when the system is quiescent.

Power is estimated using Design Compiler's default activity factor of 1.0 (100%) for clocks and 0.1 (10%) for non-clock signals. It also assumes equal probability of 0 and 1 states on each node to compute leakage power. Activity factors can be determined more accurately by running a simulation of the synthesized netlist on a real workload and recording the number of times each node toggles in a VCD (Verilog Change Dump) file. See `wsim -vcd` and the `MAKE_VCD` mode in `testbench.sv`. However, the results are highly sensitive to the workload; for example, the CoreMark benchmark has no floating-point instructions and thus may predict no activity in the FPU.

Power is computed at the target clock frequency and hence changes with the frequency target. Energy per cycle or operation is a more insightful metric because it is independent of frequency. Energy per cycle is obtained by multiplying the power by the target clock period (the reciprocal of frequency). In this case, the power is 6.122 mW, and the target frequency is 333 MHz, so the energy is $6.122 \times 10^{-3} / 333 \times 10^6 = 18$ pJ/cycle.

6.4.6 Area Reports

Fig. 6.20 shows an area report that gives a breakdown for each module. As highlighted in blue, the total area is 95,064 μm^2 , or about 0.1 mm^2 ; 71% of the area is in the integer execution unit (`ieu`), and more than half of that area is the register file (`ieu/dp/regf`). The total area is based on cell area, assuming wires can run over top. Place and route tools typically need an extra 10%–30% area to have the flexibility to place critical cells near each other and avoid routing congestion.

6.4.7 Other Reports

The reports directory has a number of other handy outputs.

- ▶ **check_design.rpt** lists potential problems with the design. Because Wally is configurable, it lists hundreds of unconnected ports. Many inputs are unused in `rv32e` because the features that would use them do not exist. For example, the `TimerIntM` timer interrupt pin is unconnected because there is no privileged unit to accept interrupts.
- ▶ **constraint_all_violators.rep** lists all endpoints that have negative slack.
- ▶ **constraint.rep** summarizes unsatisfied constraints, particularly maximum delay.

Report : area

Design : wallypipelinedcore__395949

Version: R-2020.09-SP5-5

Date : Thu Nov 16 07:18:39 2023

Library(s) Used:

sky130_osu_sc_12T_ms_TT_1P8_25C.ccs (File:
/opt/riscv/cad/lib/sky130_osu_sc_t12/12T_ms/lib/sky130_osu_sc_12T_ms_TT_1P8_25C.ccs.db)

Number of ports: 6891

Number of nets: 13479

Number of cells: 8265

Number of combinational cells: 7014

Number of sequential cells: 1173

Number of macros/black boxes: 0

Number of buf/inv: 2060

Number of references: 11

Combinational area: 52330.105886

Buf/Inv area: 10525.996487

Noncombinational area: 42734.023876

Macro/Black Box area: 0.000000

Net Interconnect area: undefined (No wire load specified)

Total cell area: 95064.129763

Total area: undefined

The above information was reported from the logical library. The following are from the physical library:

Hierarchical area distribution

Hierarchical cell	Global cell area		Local cell area			Design
	Absolute Total	Percent Total	Combi-national	Noncombi-national	Black-boxes	
wallypipelinedcore__395949	95064.1298	100.0	40.0488	0.0000	0.0000	
wallypipelinedcore__395949						
ebu.ebu	4124.5380	4.3	949.9380	0.0000	0.0000	
ebu_XLEN32_PA_BITS34_AHBW32						
ebu.ebu/IFUInput	2252.9892	2.4	30.2808	0.0000	0.0000	
controllerinput_PA_BITS34						
ebu.ebu/IFUInput/genblk1.RestorMux	412.6980	0.4	412.6980	0.0000	0.0000	
mux2_WIDTH43						
ebu.ebu/IFUInput/genblk1.SaveReg	1810.0104	1.9	746.2752	1063.7353	0.0000	
flopenr_WIDTH43						
ebu.ebu/LSUInput	252.0144	0.3	252.0144	0.0000	0.0000	
controllerinput_PA_BITS34_SAVE_ENABLED0						
ebu.ebu/ebufsmarb	669.5964	0.7	228.5712	0.0000	0.0000	ebufsmarb
ebu.ebu/ebufsmarb/BeatCounter	295.4820	0.3	60.5616	0.0000	0.0000	
counter_WIDTH4						
...						
hzu	98.6568	0.1	98.6568	0.0000	0.0000	
hazard__395949						
ieu	67129.1145	70.6	25.3968	0.0000	0.0000	
ieu__395949						
ieu/c	2178.2640	2.3	555.3108	0.0000	0.0000	
controller__395949						
ieu/c/branchflagmux	23.4432	0.0	23.4432	0.0000	0.0000	
mux2_WIDTH1						
ieu/c/controlregD	59.5848	0.1	27.3504	32.2344	0.0000	

Fig. 6.20 area.rep.

flopenrc_WIDTH1_0						
ieu/c/controlregE	984.1260	1.0	403.9068	580.2192	0.0000	
flopenrc_WIDTH35						
ieu/c/controlregM	431.7456	0.5	173.8704	257.8752	0.0000	
flopenrc_WIDTH25						
ieu/c/controlregW	124.0536	0.1	59.5848	64.4688	0.0000	
flopenrc_WIDTH5_5						
ieu/dp	64108.8488	67.4	336.5076	0.0000	0.0000	
datapath__395949						
ieu/dp/IEUResultMReg	1695.7248	1.8	664.2240	1031.5009	0.0000	
flopenrc_WIDTH32_2						
. . .						
ieu/dp/WriteDataMReg	1714.2840	1.8	682.7832	1031.5009	0.0000	
flopenrc_WIDTH32_1						
ieu/dp/altresultmux	389.2548	0.4	389.2548	0.0000	0.0000	
mux2_WIDTH32_1						
ieu/dp/alu	8880.0887	9.3	4587.5412	0.0000	0.0000	
alu_32_395949						
ieu/dp/alu/sh	4292.5476	4.5	4292.5476	0.0000	0.0000	
shifter__395949						
ieu/dp/comp	1995.1140	2.1	1995.1140	0.0000	0.0000	
comparator_WIDTH32						
ieu/dp/ext	465.4452	0.5	465.4452	0.0000	0.0000	
extend__395949						
ieu/dp/faemux	902.5632	0.9	902.5632	0.0000	0.0000	
mux3_WIDTH32_1						
ieu/dp/fbemux	904.0284	1.0	904.0284	0.0000	0.0000	
mux3_WIDTH32_0						
ieu/dp/ieuresultmux	615.8724	0.6	615.8724	0.0000	0.0000	
mux2_WIDTH32_0						
ieu/dp/regf	34373.1031	36.2	13977.5196	20395.5835	0.0000	
regfile_32_1						
ieu/dp/resultmuxW	601.2204	0.6	601.2204	0.0000	0.0000	
mux5_WIDTH32						
ieu/dp/srcamux	593.4060	0.6	593.4060	0.0000	0.0000	
mux2_WIDTH32_3						
ieu/dp/srcbmux	615.8724	0.6	615.8724	0.0000	0.0000	
mux2_WIDTH32_2						
ieu/fw	816.6048	0.9	816.6048	0.0000	0.0000	forward
ifu	16242.7191	17.1	2312.0856	0.0000	0.0000	
ifu__395949						
ifu/AlignedInstrRawDFlop	1701.0972	1.8	669.5964	1031.5009	0.0000	
flopenl_WIDTH32_1						
ifu/FlushInstrEMux	278.3880	0.3	278.3880	0.0000	0.0000	
mux2_WIDTH32_5						
ifu/InstrEReg	1710.3768	1.8	678.8760	1031.5009	0.0000	
. . .						
ifu/pcreg	1641.5124	1.7	642.2460	999.2665	0.0000	
flopenl_WIDTH32_0						
lsu	7429.0526	7.8	243.7116	0.0000	0.0000	
lsu__395949						
lsu/AddressMReg	1720.1448	1.8	688.6440	1031.5009	0.0000	
flopenrc_WIDTH32_7						
lsu/ReadDataMReg	1474.9681	1.6	443.4672	1031.5009	0.0000	
. . .						

Fig. 6.20 area.rep. Cont'd

lsu/subwordread	713.0640	0.8	713.0640	0.0000	0.0000
subwordread_LLEN32					
lsu/subwordwrite	450.7932	0.5	450.7932	0.0000	0.0000
subwordwrite_LLEN32					
lsu/swbytemask	58.6080	0.1	58.6080	0.0000	0.0000
swbytemask_32_0					
-----			-----		
Total			52330.1059	42734.0239	0.0000

Fig. 6.20 area.rep. Cont'd

6.5 WALLY SYNTHESIS RESULTS

This section describes area and delay results from synthesizing various configurations of `wallypipelinedcore` with different constraints and synthesis options. The results are presented for SKY130, SKY90, and TSMC 28SKY130. The data is gathered under typical conditions (TT process, nominal voltage, 25°C), as in [Section 6.2.1](#).

6.5.1 Synthesis Configurations

The primary configurations have large memories, so they can run arbitrary test programs during logic verification. For example, `rv32i` has an 8-MiB DTIM. Design Compiler does not gracefully build such a large memory from flip-flops. The runtime and memory are huge, and the synthesis usually fails. Therefore, we use derived configurations, that is, configurations with some tweaks such as smaller memories, summarized in [Table 6.6](#). For Skywater, the derived `syn_*` configurations reduce the size of the DTIM, IROM, and cache and branch predictor memories. For TSMC, the derived `syn_sram_*` configurations only reduce the DTIM/IROM size and build the cache and branch memories from SRAM. Other derived configurations are used to turn off features to explore critical paths, as described in [Section 6.5.4](#).

[Table 6.7](#) summarizes the core memory configurations for each case. `rv32e` has only a register file. `rv32i`, `rv64i`, and `rv32imc` have register files, DTIM, and IROM. `rv32gc` has caches, branch predictors, TLBs, and PMP. Note that uncore memories and peripherals are not part of these results.

6.5.2 Area and Delay vs. Target Clock Period

[Fig. 6.21\(a\)](#) shows how the area and delay vary with target clock period for the `rv32e` configuration in the SKY130 technology. Synthesis achieves its timing goals at long target cycle times, as indicated by the gray dots. As the cycle time decreases, the area increases slightly because synthesis trades area for faster and more parallel logic. However, [Fig. 6.20](#) shows that 38% of the area is consumed by the register file, with flip-flop area being independent of cycle time, so overall area does not increase too much with

Table 6.6 Derived configurations for synthesis.

Derived Configuration	Base Configuration	Tweaks
syn_rv32e	rv32e	Reduced memory size per Table 6.7 to keep runtime and memory reasonable for processes without SRAM
syn_rv32i	rv32i	
syn_rv32imc	rv32imc	
syn_rv32gc	rv32gc	
syn_rv64i	rv64i	
syn_rv64gc	rv64gc	
syn_sram_rv32e	rv32e	Use SRAM library for TSMC28
syn_sram_rv32i	rv32i	
syn_sram_rv32imc	rv32imc	
syn_sram_rv32gc	rv32gc	
syn_sram_rv64i	rv64i	
syn_sram_rv64gc	rv64gc	
syn_rv64gc_pmp0	rv64gc	Gradually disable features to explore critical paths
syn_rv64gc_noPriv	syn_rv64gc_pmp0	
syn_rv64gc_noFPU	syn_rv64gc_noPriv	
syn_rv64gc_noMulDiv	syn_rv64gc_noFPU	
syn_rv64gc_noAtomic	syn_rv64gc_noMulDiv	
syn_sram_rv64gc_pmp0	syn_sram_rv64gc	Gradually disable features to explore critical paths while using SRAM library
syn_sram_rv64gc_noPriv	syn_sram_rv64gc_pmp0	
syn_sram_rv64gc_noFPU	syn_sram_rv64gc_noPriv	
syn_sram_rv64gc_noMulDiv	syn_sram_rv64gc_noFPU	
syn_sram_rv64gc_noAtomic	syn_sram_rv64gc_noMulDiv	

frequency. At shorter cycle times, synthesis is unable to achieve the target cycle time, as indicated by the blue dots. The achieved cycle time and area plateau at a point nearly independent of the target cycle time, although there is some synthesis noise resulting in slight variations from one point

Table 6.7 Core memory configurations for synthesis.

Element	Configurations	USES _{SRAM}	Size	Implementation
D\$, I\$	rv{32/64}gc	0	1 way each 512 B/way 8 64B lines 8 44 or 22-bit tags	[rv32gc: $8 \times (512 + 22)$ = 4,272 flops rv64gc: $8 \times (512 + 44)$ = 4,448 flops] × 2 caches
		1	4 ways each 4 KiB/way 64 512-bit lines 64 22/44-bit tags	[4 ram1p1rwbe_64x128 1 ram1p1rwbe_64x44] × 4 ways × 2 caches
BHT	rv{32/64}gc	0	2^5 entries × 2 bits	64 flops
		1	2^{10} entries × 2 bits	1 ram2p1r1wbe_64x32
BTB	rv{32/64}gc	0	2^5 entries × 36 or 68 bits	1,152 – 2,176 flops
		1	2^{10} entries × 68 bits	1 ram2p1r1wbe_1024x68
DTIM	rv32i, rv32imc	N/A	128 words × 32 bits	4,096 flops
	rv64i	N/A	64 words × 64 bits	4,096 flops
DTLB, ITLB	rv{32/64}gc	N/A	32 entries each × 61 or 106 bits	1,952 – 3,392 flops
PMP	rv{32/64}gc	N/A	16 entries × 64 bits	1,024 flops
Register File	rv32e	N/A	16 entries × 32 bits	512 flops
	rv32{i/imc/gc}	N/A	32 entries × 32 bits	1,024 flops
	rv64{i/gc}	N/A	32 entries × 64 bits	2,048 flops

to another. Normalizing delay in terms of fanout-of-4 (FO4) inverter delays helps compare designs across processes. The best achievable cycle time is 28-32 FO4.

Fig. 6.21(b and c) show area and delay versus clock period for SKY90 and TSMC28. The TSMC28 feature size is about 5 times smaller than SKY130. The area is about 15 times smaller, and the clock period is about 8 times shorter. The best achievable cycle time is similar for all designs when normalized to FO4 delays.

Ideal Dennard scaling would predict that a 5x feature size reduction makes a design 25x smaller and 5x faster; market requirements favor improving performance, typically at a cost of more area.

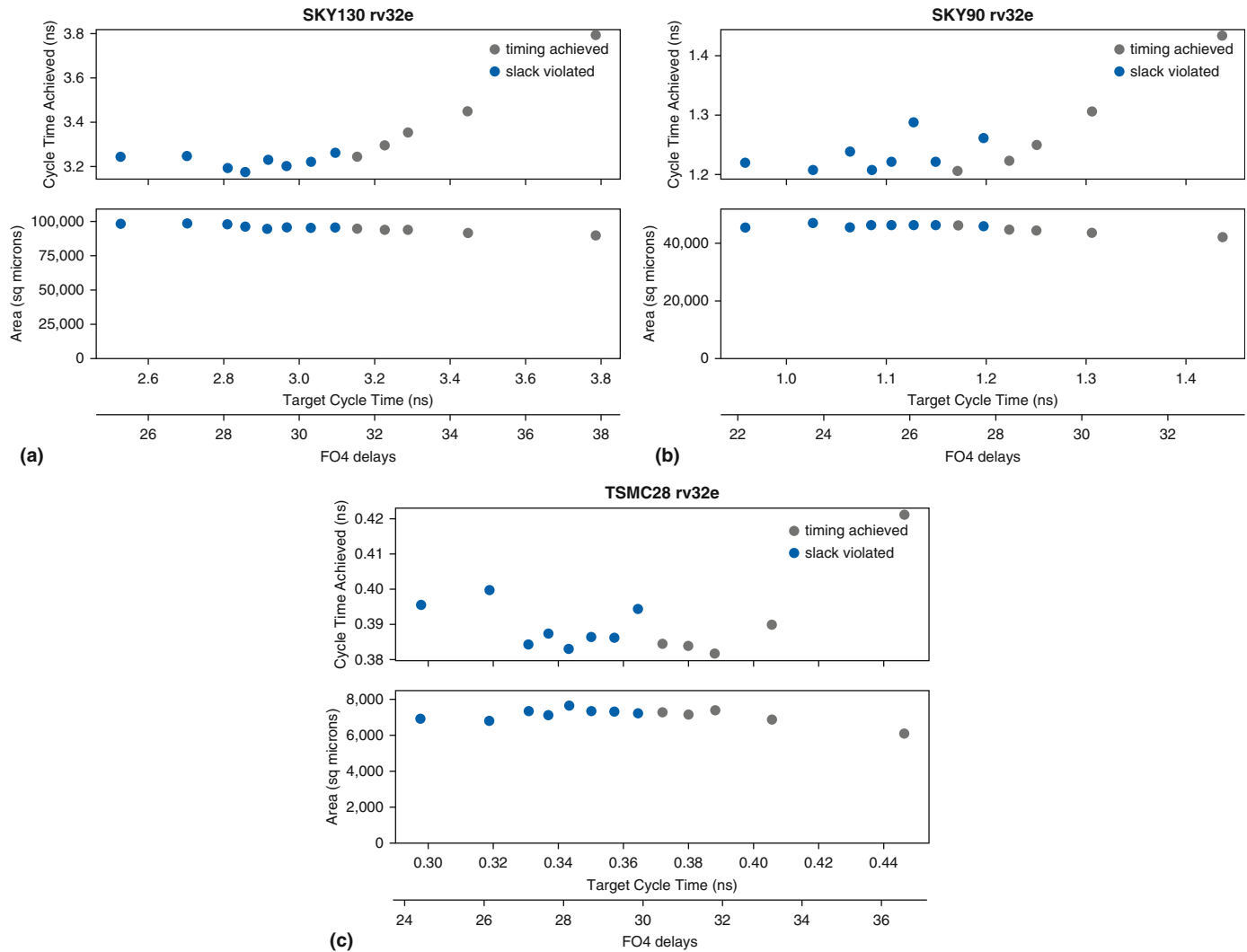


Fig. 6.21 walltypelinedcore rv32e area and delay vs. target clock period for (a) SKY130, (b) SKY90, and (c) TSMC28.

The results and plots are generated with the following commands:

```
$ cd $WALLY/synthDC
$ ./wallySynth.py --freqsweep 330 --tech sky130
$ ./wallySynth.py --freqsweep 870 --tech sky90
$ ./wallySynth.py --freqsweep 2800 --tech tsmc28psyn
$ ./extractSummary.py
```

wallySynthAll.py
automates running all of the
simulations shown here.

The `--freqsweep` option requests a series of target frequencies centered around the specified value (e.g., 870 MHz = 1.15 ns). Use `wallySynth.py -h` to get more information on wallySynth options. The synthesis logs, reports, and outputs are placed in the `runs` subdirectory. `extractSummary.py` generates plots in `wallyplots/freqsweep*` from all of the runs, including these sweeps and other experiments described later in this section. Expect some messages because the other experiments have not yet been run. Linux users may display the plots using:

extractSummary.py
requires several Python
libraries to be installed: sudo,
pip3, install,
matplotlib, scipy,
sklearn, and adjustText.

```
$ feh wallyplots/freqSweep_sky90_rv32e.png
```

Fig. 6.22 shows the same results when the `--maxopt` flag is added to the `wallySynth.py` invocation. Curiously, TSMC28 gets markedly faster, but SKY90 does not improve much. The `--maxopt` flag turns on the following Design Compiler optimizations to get better area and delay at the cost of more runtime:

- ▶ `flatten`: Allow optimizations across all modules but lose module names
- ▶ `optimize_registers`: Move logic across pipeline stages to improve critical paths

```
$ <remove or archive old files in the runs directory so they don't interfere>
$ ./wallySynth.py --freqsweep 350 --tech sky130 --maxopt
$ ./wallySynth.py --freqsweep 870 --tech sky90 --maxopt
$ ./wallySynth.py --freqsweep 3000 --tech tsmc28psyn --maxopt
$ ./extractSummary.py
```

6.5.3 Area and Delay vs. Configurations

Fig. 6.23 shows area and delay for the various configurations. Note that TSMC28 uses full-sized RAMs, while Skywater uses reduced-size flop-based memories. The results, plots, and area tables are generated with the following commands. The target frequency is set beyond what is achievable, to estimate the minimum cycle time.

```
$ ./wallySynth.py --configsweep --tech sky130 --targetfreq 1000
$ ./wallySynth.py --configsweep --tech sky90 --targetfreq 3000
$ ./wallySynth.py --configsweep --tech tsmc28psyn --targetfreq 10000 --usesram
$ ./extractSummary.py --sky130freq 1000 --sky90freq 3000 --tsmc28freq 10000
```

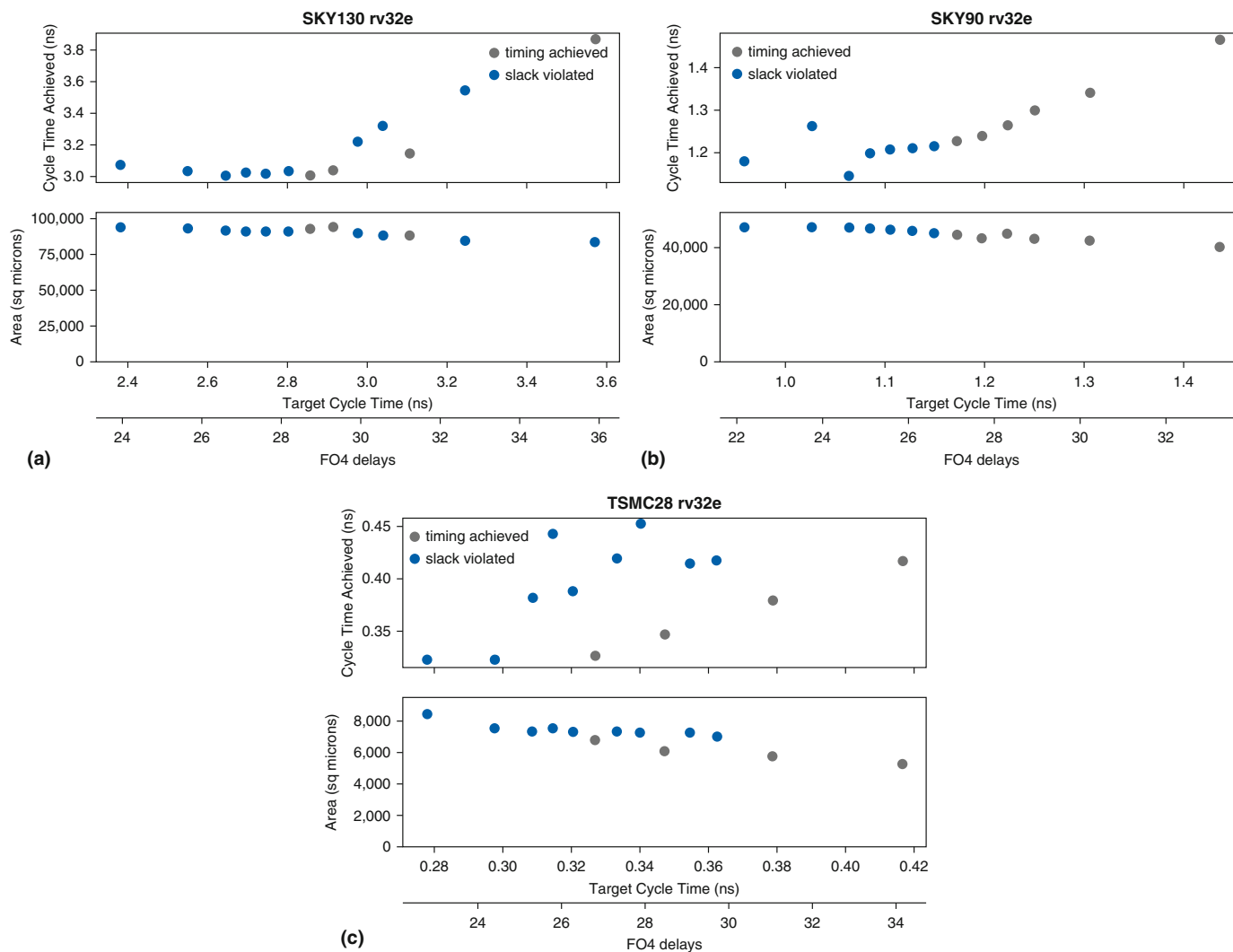


Fig. 6.22 wallypipelinedcore rv32e area and delay vs. target clock period with maximum optimization for (a) SKY130, (b) SKY90, and (c) TSMC28.

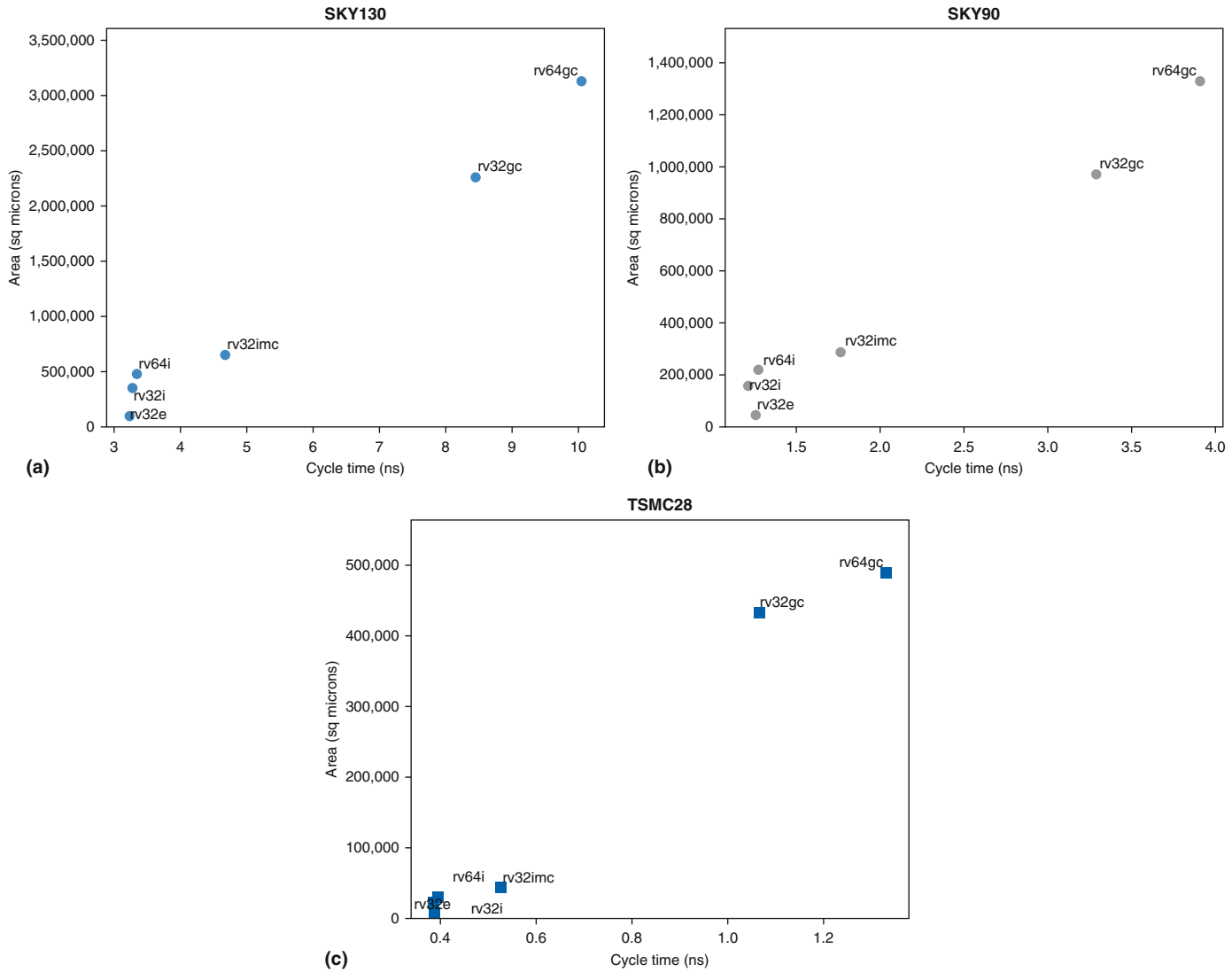


Fig. 6.23 wallypipelinedcore area and delay vs. configuration for (a) SKY130, (b) SKY90, and (c) TSMC28.

Table 6.8 summarizes the area breakdown by module for each configuration in TSMC28 with real memories ($USES_{RAM} = 1$). The rv{32/64}gc configurations are far larger than the others because of the large caches and branch predictor. The rv32e IEU is 30% smaller than rv32i because it has 16 rather than 32 registers. The rv64 IEU is about twice the size of rv32 because of the wider datapaths. The rv32 and rv64 LSU and FPU are comparable in size because most of the structures are independent of XLEN. In some configurations, a module is more critical than in others and hence trades greater area for greater speed. The full-featured rv64gc application processor is 11x larger than the rv32imc embedded core and 72x larger than the minimal rv32e core.

Table 6.9 breaks down area by cell type, and Table 6.10 counts the number of cells of each type. In the rv{32/64}gc configurations, the cache and branch predictor memories account for 65%–70% of the area. These configurations have 42 RAMs: (4 data subarrays and one tag array in each way) $\times$ 4 ways/cache $\times$ two caches, along with a BHT and BTB. Flip-flops add 15%–36% to the logic cell count but 60%–100% more area because they are larger than the average logic cell. Inverters and buffers also add 15%–30% to the logic cell count but only 10%–15% more area because they are smaller than the average logic cell.

Table 6.11 lists the most critical path for each configuration.

Fig. 2.14 shows a placement of the rv32gc walltypipelinedcore. Observe how the RAMs account for most of the area and how the units that communicate on critical paths are placed close to each other.

Synthesizing with `--maxopt` does not make much difference for the various configurations in rv64gc. It detects that rv32i and rv64i don't drive outputs and, thus, optimizes away the entire processor. It also runs much more slowly and sometimes crashes on designs with flop-based memories.

The rv64gc path through the PMP checker from the `pmpaddr` register (signal name: `PMPAdr`) is unexpected. We anticipate that the PMP path would start with the TLB physical address translation. This is a sign that something might be wrong in the synthesis experiment; it remains an open question at the time of this writing.

Table 6.8 walltypipelinedcore area by module (μm^2 , TSMC28, $USES_{RAM} = 1$, no MAXOPT, high speed).

Module	rv32e	rv32i	rv32imc	rv32gc	rv64i	rv64gc
ifu	1,071	906	1,967	208,720	1,605	221,465
ieu	4,951	6,522	6,673	7,618	13,645	16,795
lsu	470	15,328	15,325	164,555	16,071	183,752
hazard	33	11	56	66	16	70
ebu	294		261	318		475
privileged			12,040	16,001		18,729
mdu			8,358	4,107		9,883
fpu				31,776		38,575
Total	6,831	22,836	44,810	433,212	31,355	489,759

6.5.4 Area and Delay vs. Features in rv64gc

Fig. 6.24 shows area and delay for rv64gc as various features are turned off, as described in Table 6.12. The results and plots and area tables are generated with the following commands:

```
$ ./wallySynth.py --featuresweep --tech sky130 --targetfreq 1000
$ ./wallySynth.py --featuresweep --tech sky90 --targetfreq 3000
$ ./wallySynth.py --featuresweep --tech tsmc28psyn --targetfreq 10000 --usesram
$ ./extractSummary.py --sky130freq 1000 --sky90freq 3000 --tsmcfreq 10000
```

Table 6.9 wallypipelinedcore area by component type (μm^2 , TSMC28, USESRAM = 1, no MAXOPT).

Components	rv32e	rv32i	rv32imc	rv32gc	rv64i	rv64gc
RAMs	0	0	0	313,997	0	313,997
Flip-flops	2,866	11,010	19,216	38,203	13,920	54,878
Inv/Buf	463	1,820	2,382	10,277	2,555	15,737
Logic	3,502	10,006	23,211	70,733	14,880	105,146

Table 6.10 wallypipelinedcore cell count by component type (TSMC28, USESRAM = 1, no MAXOPT).

Components	rv32e	rv32i	rv32imc	rv32gc	rv64i	rv64gc
RAMs	0	0	0	42	0	42
Flip-flops	1,180	5,641	8,636	15,925	7,021	22,404
Inv/Buf	1,101	5,410	5,038	24,157	6,734	34,900
Logic	4,204	12,766	29,680	94,975	18,162	143,283
Total	6,485	23,817	43,354	135,099	31,917	200,629

Table 6.11 wallypipelinedcore critical path summary by configuration (TSMC28, USESRAM = 1, no MAXOPT).

Configuration	Critical Path
rv32e	Execute stage: forward register, compare, taken branch, flush pipeline
rv32i	Execute stage: forward register, compare, taken branch, PC register
rv32imc	Execute stage: forward register, integer multiply
rv32gc	Memory stage: HPTW, VAdr, ReadDataWord, AMO, LineWriteData
rv64i	Execute stage: forward register, compare, taken branch, PC register
rv64gc	Fetch stage: PMPAdr, PMP check, InstrAccessFault

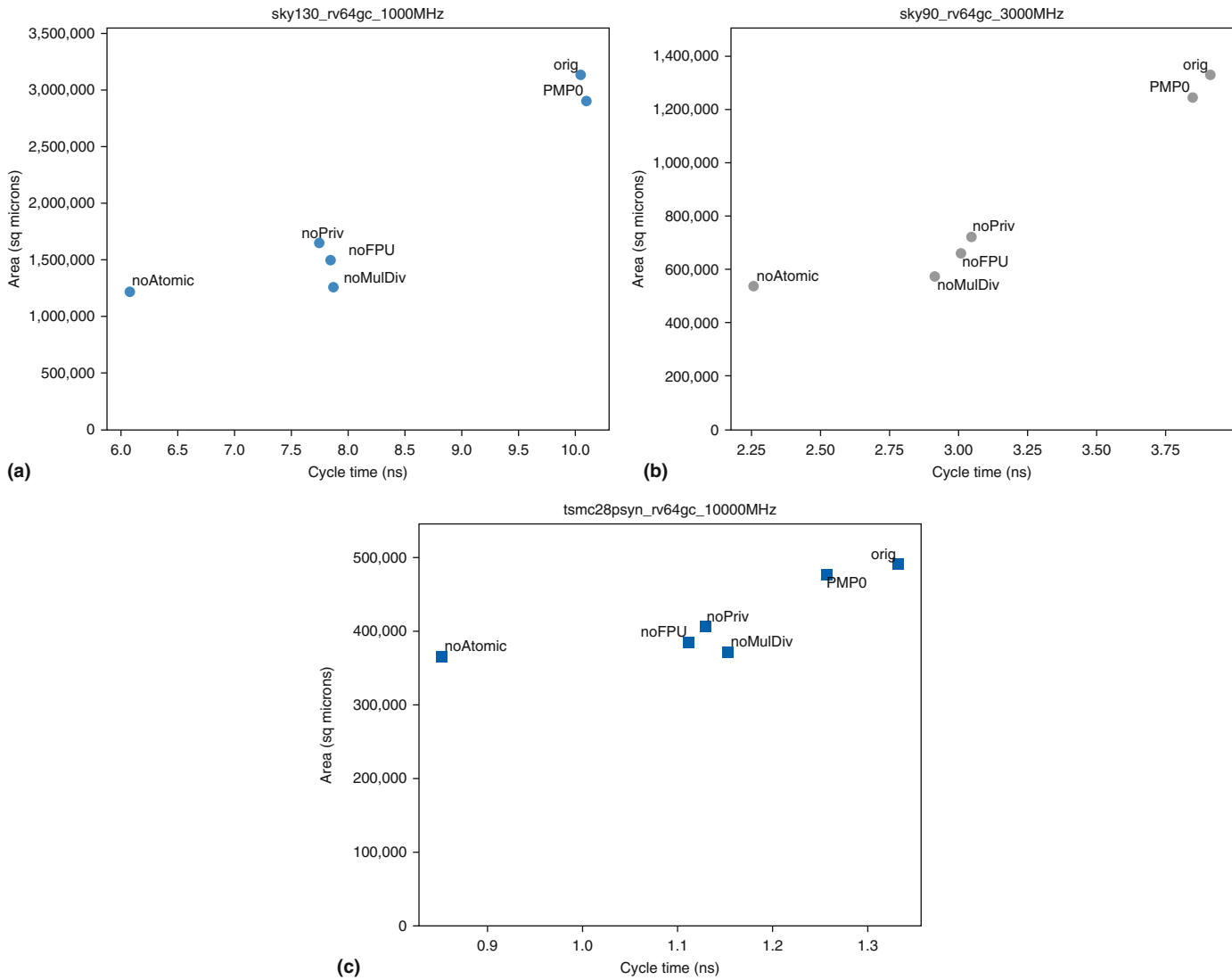


Fig. 6.24 rv64gc wallpipelinedcore area and delay vs. features for (a) SKY130, (b) SKY90, and (c) TSMC28.

Table 6.12 wallypipelinedcore rv64gc feature deactivation experiment (TSMC28, USESRAM = 1, no MAXOPT).

Feature	PMP Entries	Privileged Unit	FPU	MDU	Atomic Ops
orig	16	Y	Y	Y	Y
PMP0	0	Y	Y	Y	Y
noPriv	0	N	Y	Y	Y
noFPU	0	N	N	Y	Y
noMulDiv	0	N	N	N	Y
noAtomic	0	N	N	N	N

The SKY90 and TSMC28 results generally show the cycle time improving and area shrinking as features turn off. The area benefits are more pronounced on SKY130 and SKY90 because they use tiny flop-based memories; in TSMC28, the large RAMs account for most of the area.

6.6 SUMMARY

Logic synthesis maps synthesizable HDL code onto a cell library, minimizing area subject to a clock frequency constraint. This chapter examines the capabilities of three CMOS technologies. The older Skywater Technology 130- and 90-nm processes (SKY130 and SKY90) are low-cost technologies suited to microcontrollers and other chips of modest complexity. The newer TSMC 28 nm process (TSMC28) is the last of the mainstream planar processes before the advent of FinFETs, costly multipatterning, and draconian nondisclosure agreements (NDAs) that appeared starting with the 16-nm process. TSMC 28 is available for academic research and modern enough to be relevant for some commercial products. It also has an industry-strength set of libraries, including a memory compiler that can produce the SRAM memories needed for Wally's caches and branch predictor. The 28-nm process is 3-4x faster and offers about 8x better density and dynamic energy than the 90-nm process.

This chapter has illustrated how to use a Make-based script to synthesize Wally using the industry-standard Synopsys Design Compiler and how to interpret performance, power, and area (PPA) reports. In the 28-nm process, rv32e consumes about 6,500 logic cells (0.007 mm²) and operates up to 2.8 GHz, limited by the Execute stage branch logic. rv64gc consumes about 200,000 logic cells and 42 RAM macros (0.5 mm²) and operates at 800 MHz, limited by the time for the TLB and PMP to detect an illegal physical address, generate a fault, and update the program counter to fetch the next instruction from the trap handler.

In SKY130, it is unexpected that PMP0 is slower than the baseline design (orig) and that noFPU and noMulDiv are slower than noPriv, calling into question the quality of the library models. Sometimes, you get what you pay for.

Pipelined Core

7

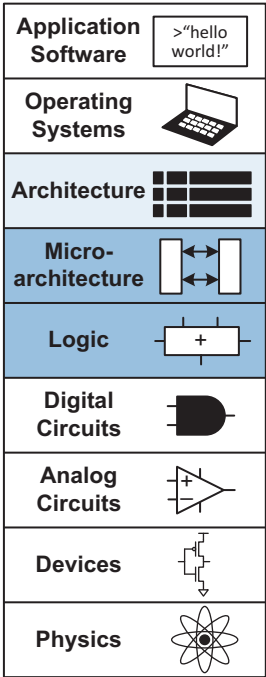
- 7.1 Single-Cycle Core
- 7.2 Pipelining the RV32I Core
- 7.3 SystemVerilog
- 7.4 Test Plan
- 7.5 Summary

This chapter builds a simple configurable pipelined core capable of running RV32I or RV64I, integer instructions. Compared to the toy microarchitectures in [Section 2.7](#), it implements the full integer instruction set. [Fig. 7.1](#) shows the high-level block diagram with the unimplemented portions grayed out. The core contains ideal instruction and data memories (instruction read-only memory [IROM], data tightly integrated memory [DTIM]) and has no bus interface for external memory or peripherals. The trivial branch predictor always predicts branch not taken. There is no privileged unit, so the processor always runs in machine mode and does not take traps, support control and status registers (CSRs), or provide virtual memory. None of the other extensions, such as multiply/divide or floating-point, are supported in the core described in this chapter. All these features will be added to the core in subsequent chapters.

Recall that standard RISC-V instructions are 32 bits long. Operands are XLEN bits, where $XLEN = 32$ in RV32 and $XLEN = 64$ in RV64. [Table 7.1](#) lists the various integer instructions common to the RV32I and RV64I instruction set as well as the extra instructions in RV64I. [Tables I.1](#) and [I.2](#) (on the last pages of the book) provide more details for these instructions. The processor in this chapter supports all these instructions. All instructions in a category are handled in approximately the same way.

[Section 7.1](#) begins by building a single-cycle RV{32/64}I core that supports all these base instructions and executes each instruction in one cycle. [Section 7.2](#) then converts the single-cycle design into a five-stage pipeline by adding pipeline registers and hazard logic. [Section 7.3](#) explores how this core is implemented in SystemVerilog, and [Section 7.4](#) defines the test plan for these configurations. Subsequent chapters fill in the remaining features:

- ▶ 8: Privileged operations
- ▶ 9–13: Memory, bus, and cache systems
- ▶ 14: Compressed instructions
- ▶ 15: Multiply and divide instructions



Tightly integrated memories (TIMs) are located near the processor and can typically be accessed within one processor clock cycle. They are also referred to as tightly coupled (TCMs), closely-coupled (CCMs), or scratchpad memories.

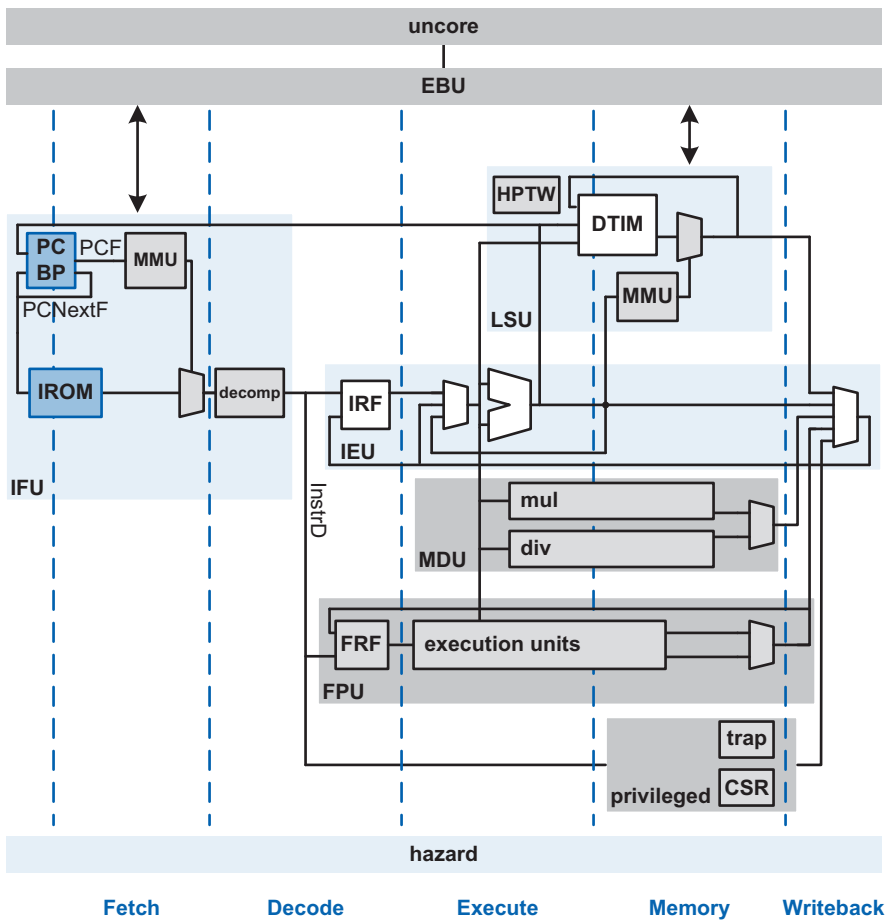


Fig. 7.1 RV{32/64}I high-level block diagram.

Table 7.1 Integer instructions.

Category	Op	Format	RV{32/64}I Integer Instructions	RV64I Extra Instructions
R-Type ALU	0x33/0x3B	R	add, sub, and, or, xor, sll, srl, sra, slt, sltu	addw, subw, sllw, srlw, sraw
I-Type ALU	0x13/0x1B	I	addi, andi, ori, xori, slli, srli, srai, slti, sltiu	addiw, slliw, srliw, sraiw
Loads	0x03	I	lw, lh, lhu, lb, lbu	ld, ldw
Stores	0x23	S	sw, sh, sb	sd
Branches	0x63	B	beq, bne, blt, bge, bltu, bgeu	
Jump and Link	0x6F	J	jal	
Jump and Link Register	0x67	I	jalr	
Load Upper Immediate	0x37	U	lui	
Add Upper Immediate PC	0x17	U	auipc	

- ▶ 16: Floating-point instructions
- ▶ 17: Bit-manipulation and cryptography instructions
- ▶ 18: Atomic instructions
- ▶ 19: Other extensions
- ▶ 20: Peripherals

Chapters generally start with the principles applicable to many computer architectures and then examine the practices specific to RISC-V. They then define the test plan and finally provide a microarchitecture specification with the implementation details.

7.1 SINGLE-CYCLE CORE

Recall the simplified RV32I single-cycle core from Fig. 2.9, which supported only the `add`, `sub`, `and`, `or`, `slt`, `addi`, `andi`, `ori`, `slti`, `lw`, `sw`, `beq`, and `jal` instructions. This section details a full-featured single-cycle RV{32/64}I core shown in Fig. 7.2 that adds support for the rest of the instructions in Table 7.1, including the following:

- ▶ More R- and I-type ALU instructions, including RV64's W instructions, which operate on 32-bit values
- ▶ Byte, half-word, and double-word loads and stores
- ▶ More flavors of branches
- ▶ Upper immediate instructions: `lui` and `auipc`
- ▶ Jump and link register: `jalr`

On the surface, Fig. 7.2 closely resembles the simplified core from Fig. 2.9. The prominent structural change is a new `AltResult` multiplexer, which chooses an alternate value to write to the register file: either the immediate for `lui` or `PCPlus4` for jump return addresses. Under the hood are several other changes:

- ▶ The datapath is configurable to be either 32 or 64 bits for RV32 or RV64.
- ▶ The ALU is enhanced to support the additional operations (`xor`, `set less than unsigned`, `shifts`). It takes an extra bit of control to specify RV64's RW/IW instructions, such as `addw` and `addiw`.
- ▶ The LSU adds circuitry to perform subword loads and stores.
- ▶ The comparator generates more flags (`equal`, `less than`, `less than unsigned`) to handle all the flavors of branches, and the controller picks the appropriate flag.
- ▶ The extender is enhanced to support U-format instructions.
- ▶ The controller adds entries for the new instruction categories: `jalr`, `lui`, and `auipc`.

In Chapter 9, we add a bus to access peripherals and off-chip memory. Then in Chapters 10–13, we replace the instruction and data memories with caches holding the most recently used portions of the off-chip memory, as well as add the `fence.i` instruction to synchronize the instruction cache after stores to program memory.

Multiplexers and registers are named by their output signal. For example, the `AltResult` multiplexer outputs the `AltResult` signal.

This design is slightly different than the processor in *Digital Design and Computer Architecture* [Harris21], which uses the ALU for branch comparisons and a separate adder for branch address calculations. The revised organization, which uses the ALU for address calculation and a dedicated branch comparator, saves a bit of hardware when supporting all instructions and decreases critical path delay in small configurations.

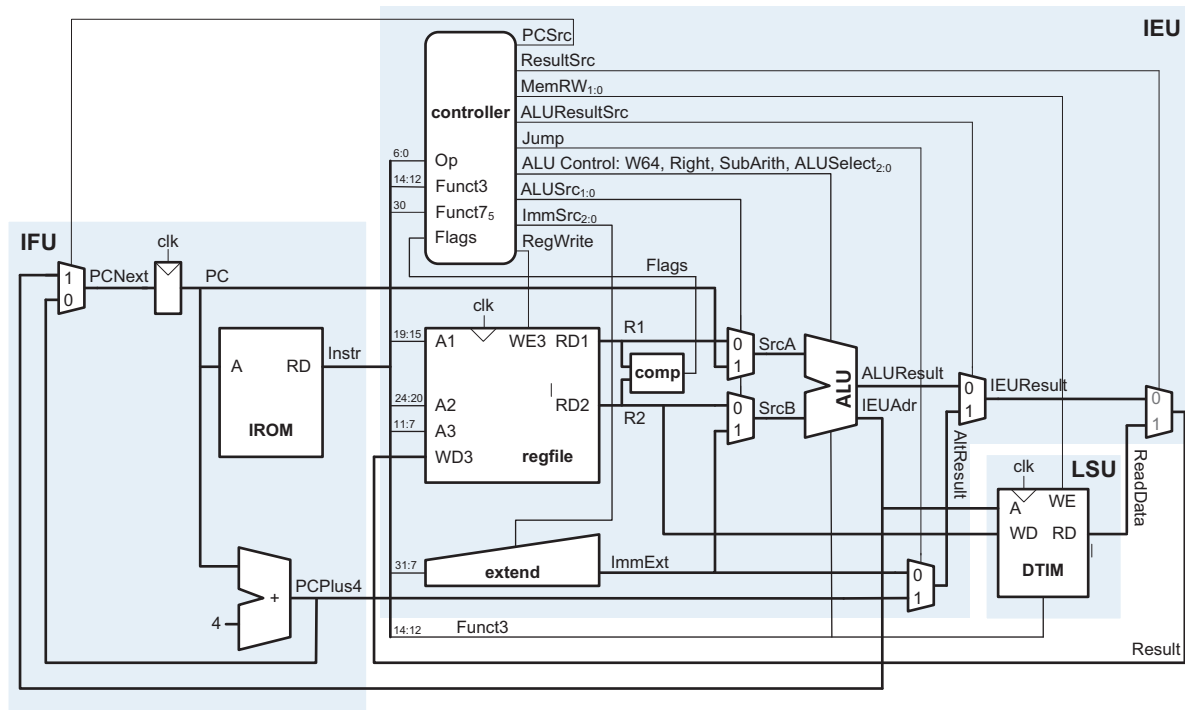


Fig. 7.2 Single-cycle core.

This section first discusses the single-cycle core by defining the architectural state that must exist in any RISC-V implementation: the program counter (PC) and registers. We then explore the `extend` block that extracts and extends the immediate from various formats of instructions. Next, we describe the arithmetic circuitry including the ALU and comparator. We then discuss the controller, which computes the multiplexer select and write enable signals to steer the datapath. Finally, we examine how to support subword loads and stores in the DTIM.

7.1.1 Architectural State

Any processor must have a certain amount of *architectural state* to remember the information visible to the programmer. In RISC-V, the architectural state comprises the PC, which contains the address of the current instruction, and the integer register file (IRF), which contains registers `x1-x31`. The single-cycle microarchitecture has this architectural state, instruction and data memories, and no other memory elements.

Because this chapter only deals with integer operations, we also refer to the IRF as simply the register file.

The IRF has three ports, so it can read two source registers and write a destination register in one cycle. Ports 1 and 2 are read ports, each with a 5-bit address ($A1/A2$) that specifies which of the 32 registers to read, and an $XLEN$ -bit read data port ($RD1/RD2$), which delivers the contents of the specified register. Port 3 is a write port where $A3$ specifies the register and $WD3$ specifies the data to write; $WE3$ specifies whether to write and a clock specifies when to write (on the rising edge). Register 0 is hardwired to 0, and writes to it are ignored.

In this implementation, the instruction memory is read-only and is preloaded with the program before the processor starts executing. The data memory is readable and writable.

7.1.2 Immediate Extension

Another source of operands for many instructions comes from a 12-, 13-, 21-, or 32-bit immediate that is encoded in a 12- or 20-bit immediate field that is scattered around the instruction, as described in Section 2.2. The extender block extracts this immediate and appropriately sign-extends its most significant bit, which is always in instruction bit 31, to $XLEN$ bits to create an extended immediate, $ImmExt$, as shown in Table 7.2 and Figs. 2.3 and 2.4. In RV64, the immediate is also sign-extended into bits 63:32 of $ImmExt$. The extender is just a big multiplexer controlled by an $ImmSrc$ signal that depends on the instruction format, as will be described in Section 7.1.4.1.

7.1.3 Arithmetic Circuitry

The processor has three arithmetic blocks:

- ▶ **ALU:** Performs an arithmetic, shift, or logical operation and address calculation
- ▶ **Comparator:** Compares two registers to produce flags for branches
- ▶ **Incrementer:** Increments the PC by 4 to calculate the next PC

Table 7.2 Extend operation.

$ImmSrc$	$ImmExt$	Type	Description
000	$\{\{20\{Instr[31]\}\}, Instr[31:20]\}$	I	12-bit signed
001	$\{\{20\{Instr[31]\}\}, Instr[31:25], Instr[11:7]\}$	S	12-bit signed
010	$\{\{20\{Instr[31]\}\}, Instr[7], Instr[30:25], Instr[11:8], 1'b0\}$	B	13-bit signed
011	$\{\{12\{Instr[31]\}\}, Instr[19:12], Instr[20], Instr[30:21], 1'b0\}$	J	21-bit signed
100	$\{Instr[31:12], 12'b0\}$	U	20-bit signed

Throughout this book, core diagrams such as Fig. 7.2 use the following conventions:

- ▶ **Module names:** bold font. These match the HDL module names (e.g., `hazard` for hazard unit, `lsu` for the load/store unit [LSU]).
- ▶ **Signal names:** UpperCamel-Case. They match the HDL signal names.
- ▶ **Descriptions:** bold italic font.
- ▶ **Highlighted hardware/signals:** shown in bright blue.
- ▶ **Unused or optional hardware/signals:** shown in gray.
- ▶ **Bus widths:** single wires use thin lines, multibit buses ($<XLEN$) use medium lines, $XLEN$ buses use thick lines, and $>XLEN$ buses use very thick lines.

RISC-V immediate encodings are strange to look at but are optimized to minimize the cost and delay of the sign extension multiplexers and to avoid interfering with the other instruction fields.

The ALU has two outputs: `ALUResult` (for integer arithmetic/logical operations) and `IEUAdr` (for address calculations). The ALU can perform any of the RISC-V integer arithmetic/logical operations, including `add`, `sub`, `and`, `or`, `xor`, `set-less-than` (`slt` or `sltu`), and shifts (`sra`, `srl`, or `sll`). It also adds a base register to an immediate offset for memory, branch, and jump address calculations.

While the ALU is computing a branch target address, the branch comparator produces equal (`eq`) and less than (`lt`) flags for branches. The `lt` flag may come from signed or unsigned comparison, depending on the flavor of branch.

Table 7.3 lists the operations performed by the arithmetic circuitry for each type of instruction. The ALU computes `ALUResult` by performing addition or another operation (`op`) that depends on the ALU instruction being executed. The first source is either `R1` from the register file or `PC`, and the second source is either `R2` from the register file or the sign-extended immediate `ImmExt`. The integer execution unit (IEU) `Result` multiplexer (shown as two multiplexers in Fig. 7.2) chooses the `ALUResult`, the immediate, or `PC + 4` (`PCPlus4`) to write to the register file. To minimize delay, the ALU also provides its adder output early (before the ALU's internal multiplexer, see Fig. 7.3) as `IEUAdr`, which is sent to the LSU and IFU (instruction fetch unit) as an address for memory operations, branches, and jumps.

7.1.3.1 ALU Design

Fig. 7.3 shows a diagram of the ALU. The ALU computes all operations in parallel and selects `ALUResult` from among them with a large multiplexer. The adder optionally inverts the `B` input to perform `sub`, `slt`, or `sltu`. The

Table 7.3 Arithmetic circuits summary.

Category	<code>ALUResult</code>	<code>IEUResult</code>	<code>IEUAdr</code> (and Destination)	Comparator
R-Type ALU	<code>R1 op R2</code>	<code>ALUResult</code>	<code>x</code>	
I-Type ALU	<code>R1 op ImmExt</code>	<code>ALUResult</code>	<code>x</code>	
Loads/Stores	<code>X</code>	<code>x</code>	<code>R1 + ImmExt → LSU</code>	
Branches	<code>X</code>	<code>x</code>	<code>PC + ImmExt → IFU</code>	<code>R1 cmp R2</code>
<code>jal</code>	<code>X</code>	<code>PCPlus4</code>	<code>PC + ImmExt → IFU</code>	
<code>jalr</code>	<code>X</code>	<code>PCPlus4</code>	<code>R1 + ImmExt → IFU</code>	
<code>lui</code>	<code>X</code>	<code>ImmExt</code>	<code>x</code>	
<code>auipc</code>	<code>PC + ImmExt</code>	<code>ALUResult</code>	<code>x</code>	

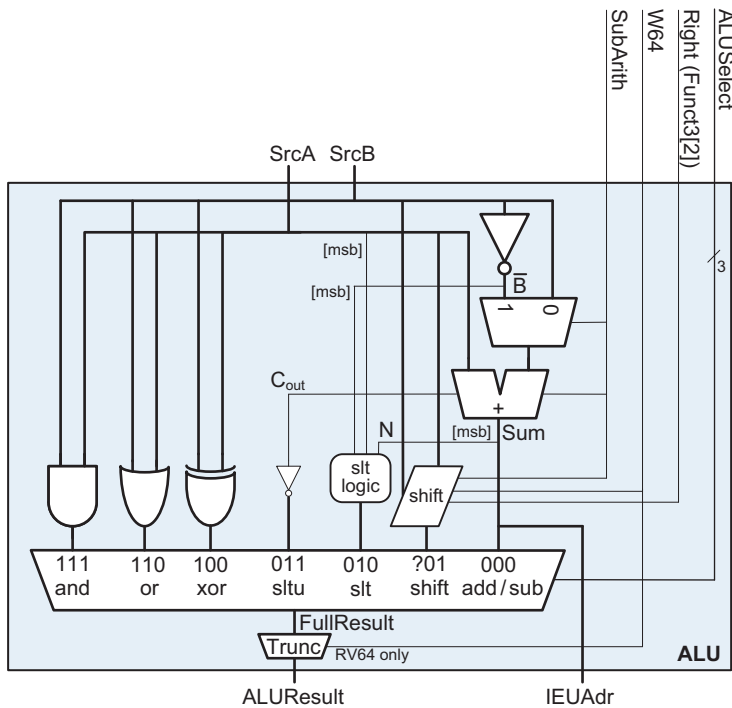


Fig. 7.3 ALU schematic.

The IEUAdr output is faster than ALUResult because it skips the multiplexer delay. This reduces the critical path when accessing memory.

shifter performs logical left shifts and logical and arithmetic right shifts (*sll*, *srl*, and *sra*). In RV64, it also handles W-type shifts on 32-bit words. The *slt* and *sltu* functions compute the *lt* and *ltu* flags and place them in the least significant bit of the result, with 0s in the upper bits. Bitwise gates perform *and*, *or*, and *xor*. ALU components are discussed further in the next sections.

7.1.3.2 Add/Subtract

The adder in the ALU performs addition or subtraction based on the control signal *SubArith*. When *SubArith* = 0, it adds, computing $\text{Sum} = \text{SrcA} + \text{SrcB}$. When *SubArith* = 1, it subtracts, computing $\text{Sum} = \text{SrcA} + \text{SrcB} + 1 = \text{SrcA} - \text{SrcB}$, using the two's complement identity that $-X = \overline{X} + 1$. Adder architectures and their area/delay tradeoffs have been heavily researched; see [Weste12] for an overview. It is usually expedient to describe the adder with + in the HDL and allow the logic synthesis tool to select the best adder given the constraints. If the adder is timing critical, synthesis generally produces a parallel prefix tree with $O(\log_2 N)$ delay.

7.1.3.3 Shifter

The shifter performs the three flavors of shifts and also handles W-type 32-bit shifts for RV64. Consider an N -bit shifter that shifts $A_{N-1:0}$ by $k \in [0, N-1]$ bits to produce $Y_{N-1:0}$. The shift amount k is a $\log_2 N$ -bit number. The main shifter architectures are funnel and barrel shifters [Weste12]. An N -bit *funnel shifter* creates a $(2N-1)$ -bit input word and then right shifts to produce an N -bit result, like a funnel necking down from wide to narrow. On the other hand, an N -bit *barrel shifter* takes an N -bit input and rotates it to produce an intermediate N -bit result. It then applies a mask to handle sign or zero extension on the bits that should be discarded. Funnel shifters are simpler for RISC-V because the integer instruction set does not have rotate instructions.

A funnel shifter creates a source word $Z_{2N-2:0}$, shifts it right by k , and delivers the bottom N bits as the result Y . To perform shift right logical (*srl*), place A in the bottom N bits of Z and 0s in the top $N-1$ bits of Z . Then shift it right by k , the desired shift amount. Thus, a shift by 0 produces $Y = A_{N-1:0}$. A shift by 1 produces $Y = \{0, A_{N-1:1}\}$. A shift by $N-1$ produces $N-1$ 0s followed by A_{N-1} . Shift right arithmetic (*sra*) is the same except the 0s are replaced by the sign bit A_{N-1} . Shift left logical (*sll*) places A in the most significant N bits of Z and 0s in the bottom $N-1$ bits of Z and then right shifts by the bit-wise complement of k ($\bar{k}$), which is $N-1-k$. For example, a left shift by 0 becomes a right shift by $N-1$ (i.e., all 1s: 111...1), which discards all the 0s and produces $Y = A$. Likewise, a left shift by 1 becomes a right shift by $N-2$ (i.e., 1111...10), which produces $Y = \{A_{N-2:0}, 0\}$.

Fig. 7.4 shows a 64-bit funnel shifter that can handle the various flavors of RV64I shifts. This design uses a multiplexer to choose 127 bits of Z according to Table 7.4 followed by six levels of 2:1 multiplexers to

Imagine a barrel shifter is a wooden barrel on its side with the input bits painted around the circumference. Rotating the barrel and reading the bits starting with the new spot facing up gives the rotated output bit sequence.

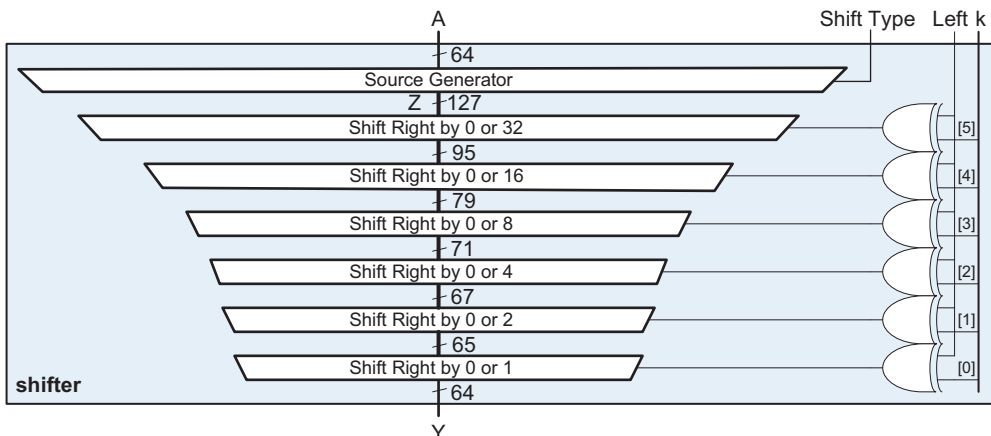


Fig. 7.4 64-bit funnel shifter.

Table 7.4 64-bit shifter source generator and shift amount.

Shift Type	Z	Shift Amount
srl	{63'b0, $A_{63:0}$ }	k
sra	{{63{ A_{63} }}, $A_{63:0}$ }	k
sll	{ $A_{63:0}$, 63'b0}	$\bar{k}$
srlw	{95'b0, $A_{31:0}$ }	k
sraw	{64'b0, {31{ A_{31} }}, $A_{31:0}$ }	k
sllw	{32'b0, $A_{31:0}$, 63'b0}	$\bar{k}$

Table 7.5 32-bit shifter source generator and shift amount.

Shift Type	Z	Shift Amount
srl	{31'b0, $A_{31:0}$ }	k
sra	{{31{ A_{31} }}, $A_{31:0}$ }	k
sll	{ $A_{31:0}$, 31'b0}	$\bar{k}$

shift right by each power of two. Observe how the bus width funnels down from 127 bits to 64 bits as the shift proceeds. The XOR gates invert the shift amount for left shifts. Some implementations use 4:1 or 8:1 multiplexers to reduce the number of logic levels. Table 7.5 shows a simpler source generator for RV32 that only produces a 63-bit Z and does not support word operations (srlw, sraw, and srlw).

Example 7.1 FUNNEL SHIFTER

How does a 64-bit funnel shifter perform the 32-bit arithmetic left shift (sllw) $0x00000024 \ll 4$? Find Z, the shift amount, and Y.

Solution: The inputs are $A = 0x0000000000000024$ and $k = 0b000100$. The funnel shifter forms the 127-bit input word Z according to Table 7.4: $Z = \{32'b0, A_{31:0}, 63'b0\} = 0x00000000000000120000000000000000$. The shift amount is $\bar{k} = 0b111011 = 59$. The funnel shifter shifts Z right by the shift amount, producing $0x0000000000000240$.

Beware that N for negative flag and N for the number of bits in a word are unrelated uses of the same variable name.

7.1.3.4 Set Less Than

`slt` and `sltu` instructions compute $\{C, Sum\} = A - B = A + (\overline{B} + 1)$ and check if the result is negative, which indicates that $A < B$. They can make the sign determination based on the two most significant bits of the difference $\{C, Sum_{N-1}\}$, which are called the C (*carry out* or, simply *carry*) and N (*negative*) flags: $N = Sum_{N-1}$.

For unsigned comparison, A and B are both positive. If the carry out is 0, $A - B$ is negative (so A is less than B). Thus, $slt_u = \overline{C}$.

For signed comparison, we must consider overflow. When overflow occurs, the result's sign bit (and thus, N) is wrong. When subtracting, overflow occurs when two conditions are true: (1) A and B have opposite signs, which, when subtracting, is effectively adding values of the same sign: $(A_{N-1} \oplus B_{N-1})$, and (2) the result's sign ($N = Sum_{N-1}$) differs from A 's sign: $(A_{N-1} \oplus N)$. Thus, the *overflow* flag $V = (A_{N-1} \oplus B_{N-1})(A_{N-1} \oplus N)$. The true result is negative when $N \oplus V$. In other words, when overflow occurs, the N flag gave the wrong value, so it must invert. This simplifies to $slt = N \oplus V = A_{N-1}\overline{B_{N-1}} + A_{N-1}N + \overline{B_{N-1}}N$.

7.1.3.5 W-Type Instructions

For W-type RV64 instructions, the full 64-bit result should be truncated to 32 bits and then sign extended into the upper half. The 32-bit Trunc multiplexer shown in Fig. 7.5 performs this operation by choosing the upper half of `ALUResult` from either the full result or by sign-extending bit 31. In RV32, no W-type instructions exist, and this Trunc multiplexer is omitted.

7.1.3.6 Comparator

The dedicated branch comparator produces the `eq` and `lt/ltu` flags for branches based on two source registers. Wally uses a dedicated comparator rather than reusing the adder and `slt` logic because comparison is on the critical path of a simple pipelined processor and because the adder is also needed to compute the branch destination. The comparator is faster than the regular ALU for several reasons:

- The branch comparator inputs come from the register file, rather than the `Src{A/B}` and `SubArith` muxes.

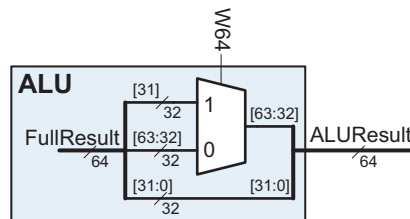


Fig. 7.5 Trunc multiplexer.

- ▶ Computing flags is less expensive than producing a full XLEN-bit result.
- ▶ The flags are available without waiting for ALU multiplexing after the adder.

A brute force method is to subtract $A - B$ and compute `lt` and `ltu` as in [Section 7.1.3.4](#). The `eq` flag should also be set when the difference is 0. This method is overkill because the actual answer of the subtraction is unnecessary.

The design can be optimized using a parallel prefix circuit, as described by Equations 7.1–7.6. The circuit first computes whether a column's inputs are equal (E_i) or unsigned less than (L_i). It then merges these bitwise results to obtain group results for columns i through j ($E_{i:j}$, $L_{i:j}$). Columns $i:j$ are equal ($E_{i:j} = 1$), when both the upper group of bits ($E_{i:k}$) AND the lower group of bits ($E_{k-1:j}$) are equal (Equation 7.3). Columns $i:j$ of A are less than columns $i:j$ of B when the upper bits are less ($L_{i:j}$) OR the upper bits are equal AND the lower bits are less ($E_{i:k}L_{k-1:j}$); see Equation 7.4. After computing these signals for all columns, the prefix circuit outputs the equal (`eq`) and less than unsigned (`ltu`) flags.

The prefix signals, $E_{i:j}$ and $L_{i:j}$, are each 1-bit signals indicating whether bits $i:j$ of A and B are equal or less than unsigned, respectively: $E_{i:j} = (A_{i:j} == B_{i:j})$ and $L_{i:j} = (A_{i:j} < B_{i:j})$.

Precomputation

$$(7.1) \quad E_i = \overline{A_i \oplus B_i} \quad \text{Bits } i \text{ of } A \text{ and } B \text{ are equal}$$

$$(7.2) \quad L_i = \overline{A_i}B_i \quad \text{Bit } i \text{ of } A < \text{bit } i \text{ of } B \text{ (viewed as unsigned numbers)}$$

Prefix Recursion

$$(7.3) \quad E_{i:j} = E_{i:k}E_{k-1:j} \quad \text{Bits } i:j \text{ are equal}$$

$$(7.4) \quad L_{i:j} = L_{i:k} + E_{i:k}L_{k-1:j} \quad \text{Bits } i:j \text{ are less than unsigned}$$

Output Logic

$$(7.5) \quad \text{eq} = E_{N-1:0} \quad A == B$$

$$(7.6) \quad \text{ltu} = L_{N-1:0} \quad A < B \text{ unsigned}$$

Signed comparison can be performed with the same hardware as unsigned comparison by flipping the most significant bit of A and B . The principle is that unsigned numbers are in the range $[0, 2^N - 1]$, and signed numbers are in the range $[-2^{N-1}, 2^{N-1} - 1]$. Flipping the most significant bit is equivalent to adding 2^{N-1} to both positive and negative two's complement numbers and converts them to unsigned inputs (i.e., shifts them to the range $[0, 2^N - 1]$) without changing which one is less.

The comparator can use bubble-pushing and alternating stages to perform the prefix recursion with only 1 level of AOI/OAI (AND-OR-INVERT/OR-AND-INVERT) gates.

A prefix adder that performs $A + B$ calculates the precomputation signals (column propagate and generate) as $P_i = A_i \oplus B_i$ and $G_i = A_i B_i$. Because this prefix subtractor is performing $B + \bar{A}$, $P_i = \bar{A}_i \oplus B_i = \bar{A}_i \oplus \bar{B}_i$ and $G_i = \bar{A}_i B_i$.

Example 7.2 COMPARING 32-BIT TWO'S COMPLEMENT NUMBERS

Compare the following 32-bit two's complement numbers by inverting the most significant bit (msb) of each one and then performing unsigned comparison: -7 and 3, -5 and -10.

Solution: -7 is 0xFFFFFFF9, and 3 is 0x00000003. Flipping the most significant bit of each number and performing unsigned comparison results in: 0x7FFFFFFF9 < 0x80000003, which is correct ($-7 < 3$). -5 is 0xFFFFFFF3, and -10 is 0xFFFFFFF6. Flipping the most significant bit of each number and performing unsigned comparison results in: 0x7FFFFFF3 ≥ 0x7FFFFFF6, which is correct ($-5 ≥ -10$).

Thus, the comparator handles both signed and unsigned comparison by adding two XOR gates to conditionally flip the most significant bits of each input for signed comparison and then using the unsigned comparison described earlier. The $1t/1tu = L_{N-1:0}$ output is now correct for either signed or unsigned comparison. Thus, an $N = 2^\ell$ -bit comparator's critical path involves an XOR to handle signed/unsigned comparison, two 2-input gates for precomputation, and ℓ levels of AND and AND-OR logic to compute $L_{N-1:0}$ and $E_{N-1:0}$ ($1t/1tu$ and eq).

The comparator recursion closely resembles subtraction. Consider computing $\{C, Sum\} = B - A - 1 = B + \bar{A}$. For unsigned numbers, $C = 1$ if and only if $A < B$. The prefix recursion for $B + \bar{A}$ is shown in Equations 7.7–7.14. It is identical to the comparator recursion from Equations 7.1–7.6 except P corresponds to E and G corresponds to L .

Precomputation

$$(7.7) \quad P_i = \bar{A}_i \oplus B_i$$

Column i propagates a carry

$$(7.8) \quad G_i = \bar{A}_i B_i$$

Column i generates a carry

Prefix Recursion

$$(7.9) \quad P_{i:j} = P_{i:k} P_{k-1:j}$$

Bits $i:j$ propagate a carry

$$(7.10) \quad G_{i:j} = G_{i:k} + P_{i:k} P_{k-1:0}$$

Bits $i:j$ generate a carry

Output Logic

$$(7.11) \quad Sum_i = P_i \oplus G_{i-1:0}$$

$$(7.12) \quad C = G_{N-1:0}$$

$$(7.13) \quad eq = (Sum == 0) = P_{N-1:0}$$

$A == B$

$$(7.14) \quad 1tu = G_{N-1:0}$$

$A < B$ unsigned

To highlight the similarities and differences between the subtractor and comparator, Fig. 7.6 shows diagrams of the subtractor-based (Equations 7.7–7.14) and optimized (Equations 7.1–7.6) prefix network unsigned comparator designs, also referred to as simply *prefix comparators*. For

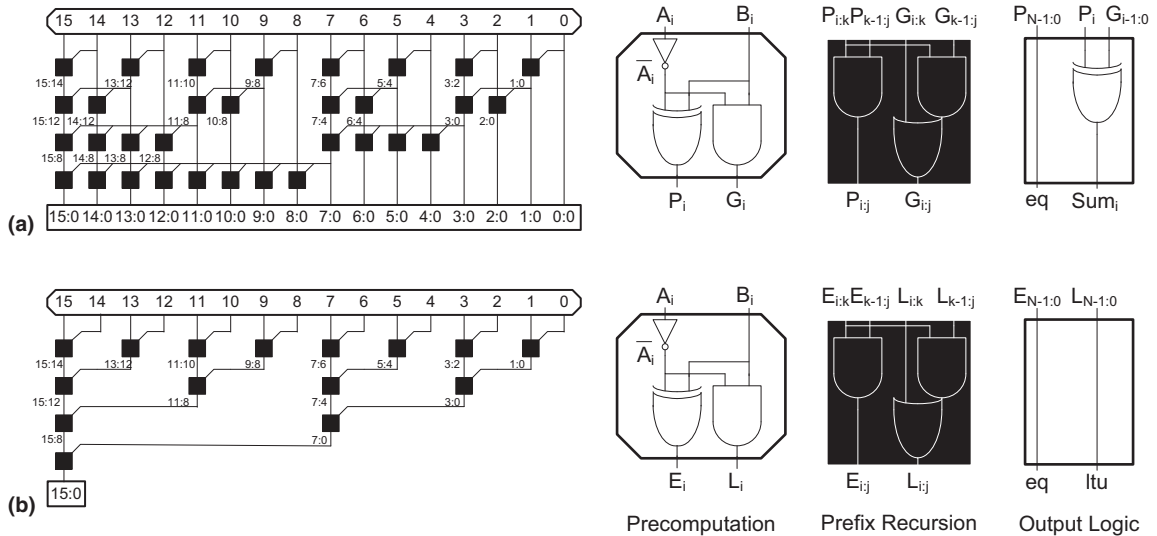


Fig. 7.6 Prefix comparators: (a) subtract, (b) optimized.

both networks, the precomputation is the same, with N AND, XOR, and NOT gates. The optimized comparator produces only the final `eq` and `ltu` results, so it does not need to calculate the intermediate column results or the `Sum` output. Thus, whereas the subtractor has $N \log_2 N / 2$ recurrence cells (black boxes), the comparator only has N recurrence cells, and the N final XORs to produce the `Sum` in the subtractor are not needed in the comparator.

Despite these design optimizations, Design Compiler is smart enough to synthesize high-level operations onto well-optimized DesignWare blocks, so we do not need to write prefix equations to get satisfactory results. Thus, Wally's `comparator` module in [Code Example 7.1](#) expresses the comparison operations in Verilog using `==` and `<` statements rather than prefix equations. The Wally `comparator` logic shown below calls the output `lt`, whether it is signed or unsigned.

Code Example 7.1 COMPARATOR LOGIC USING `==` AND `<`

```
// For signed numbers, flip msb before comparing
assign af = {a[WIDTH-1] ^ sgnd, a[WIDTH-2:0]};
assign bf = {b[WIDTH-1] ^ sgnd, b[WIDTH-2:0]};
assign eq = (a == b); // Equality
assign lt = (af < bf); // Signed or unsigned less than
```

7.1.4 Controller Design

[Fig. 7.7](#) shows a block diagram of the controller. The controller has a main decoder that computes most of the control signals based on an instruction's

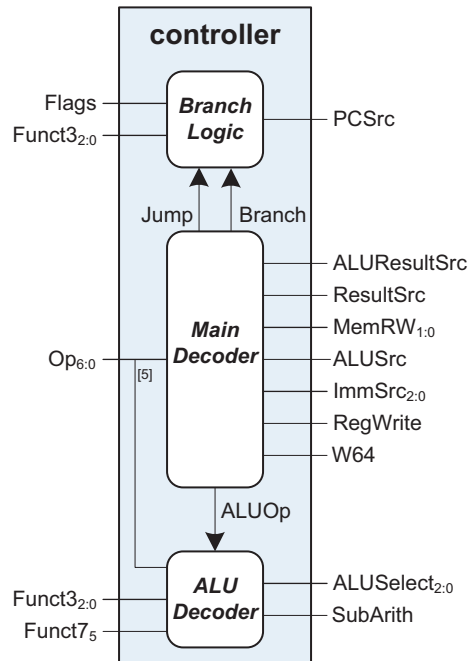


Fig. 7.7 Controller.

op field. The controller also has branch logic to compute **PCSrc** based on the branch/jump type and flags, and an ALU decoder.

7.1.4.1 Main Decoder

The don't care (x) entries in Table 7.6 do not matter but are treated as 0 in Wally to avoid illegal values in simulation.

The main decoder truth table is given in Table 7.6. Each row corresponds to a category of instructions specified by the **op** field (**Instr[6:0]**). The rows in blue text are only required for RV64I. The columns indicate the output signals. **ALUSrc** is a 2-bit value {**ALUSrcA**, **ALUSrcB**} driving the two source multiplexers. **MemRW** is also a two bit value {**MemRead**, **MemWrite**} that determines whether to read, write, or leave the memory idle.

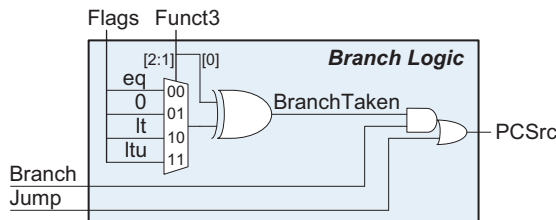
7.1.4.2 Branch Logic

The comparator implementation from Section 7.1.3.6 handles signed and unsigned comparison with the same logic, so **lt** is also used for **ltu** in the branch logic.

The controller's branch logic, shown in Fig. 7.8, outputs the PC source signal **PCSrc**, which chooses the next PC as **PC + 4** when **PCSrc = 0** and the branch or jump target address (**IEUAdr**) otherwise. The main controller sends the branch logic the **Branch** and **Jump** signals, which indicate whether the instruction is a branch or jump, the **funct3** field from the instruction, which indicates the type of branch, and the {**eq**, **lt**, **ltu**} flags from the comparator. Given the **funct3** and flags, the branch logic determines

Table 7.6 Main decoder.

Category	op	Branch	Jump	ALUSrc	ImmSrc	ALUOp	W64	ALUResultSrc	ResultSrc	RegWrite	MemRW
R-type ALU	0x33	0	0	00	xxx	1	0	0	0	1	00
I-type ALU	0x13	0	0	01	000	1	0	0	0	1	00
RW-type ALU	0x3B	0	0	00	xxx	1	1	0	0	1	00
IW-type ALU	0x1B	0	0	01	000	1	1	0	0	1	00
Loads	0x03	0	0	01	000	0	0	x	1	1	10
Stores	0x23	0	0	01	001	0	0	x	x	0	01
Branches	0x63	1	0	11	010	0	0	x	x	0	00
jal	0x6F	0	1	11	011	0	0	1	0	1	00
jalr	0x67	0	1	01	000	0	0	1	0	1	00
lui	0x37	0	0	xx	100	x	0	1	0	1	00
auipc	0x17	0	0	11	100	0	0	0	0	1	00

**Fig. 7.8** Branch logic.

whether the branch should be taken according to [Table 7.7](#). For example, a `bne` will be taken if $R1 \neq R2$, indicated by an `eq` flag of 0. This table simplifies to a 4:1 multiplexer to select the flag of interest followed by an XOR to invert the flag if necessary. The branch logic then sets `PCSrc` if the instruction is either a jump or a conditional branch that should be taken.

7.1.4.3 ALU Decoder

The controller's ALU decoder determines the ALU's operation by producing `ALUSelect` and `SubArith`, which is 1 when subtracting or arithmetic right shifting. Recall from [Fig. 7.3](#) that the ALU also requires `W64` and `Right` control signals. `W64` comes from the main decoder, and `Right` = `Funct3[2]`, which is 1 when right shifting.

Table 7.7 BranchTaken determination.

Branch Type	Funct3	BranchTaken
beq	000	eq
bne	001	$\overline{\text{eq}}$
<undefined>	01x	0
blt	100	lt
bge	101	$\overline{\text{lt}}$
bltu	110	ltu
bgeu	111	$\overline{\text{ltu}}$

This decoder is *lazy* in that it only looks at bit 5 of Funct7 rather than checking `Funct7 == 0100000`. When traps are supported, RISC-V strongly recommends throwing an illegal-instruction exception if `Funct7 != 0100000`. Thus, when `ZICSR_SUPPORTED = 1`, the decoder is stricter, at the cost of a few extra gates.

Note that `sub` also needs to look at bit 5 of the opcode to distinguish `sub` from `addi`, which has the same Funct3 field and no Funct7.

ALUSelect is Funct3 for ALU instructions or 000 for `auipc` to add the PC and the upper-immediate.

SubArith indicates the ALU is doing a subtraction (including set less than) or arithmetic right shift. This is determined from the `funct` fields (Funct3 and Funct7) when ALUOp is 1:

```
slt      = (Funct3 == 010)
sltu     = (Funct3 == 011)
sra      = (Funct3 == 101 & Funct7[5])
sub      = (Funct3 == 000 & Funct7[5] & Op[5])
SubArith = ALUOp & (slt | sltu | sra | sub)
```

7.1.5 Instruction Fetch Unit

The *instruction fetch unit* (IFU) in Fig. 7.2 comprises the PC register, IROM, PC incrementer, and PCNext multiplexer. The IROM is preloaded with the program, and stores to program addresses are disregarded. The incrementer adds 4 to find the address of the next instruction. The multiplexer picks between `PC + 4` and the branch or jump destination in `IEUAdr`.

Chapter 9 describes how to fetch instructions from an external memory over a bus interface, and Chapter 13 describes how to add an instruction cache and branch predictor to improve performance.

7.1.6 Load/Store Unit

The LSU contains a DTIM with single-cycle access. In this section, we assume that memory reads occur asynchronously and that writes occur on the rising edge of the clock. Section 7.2.1 introduces more practical memories with synchronous reads.

The DTIM internally accesses entire XLEN-bit words or double words. To handle an instruction such as `lb`, `lbu`, `lh`, or `lhu`, the DTIM reads the full word, extracts the appropriate byte or halfword, places it in the least significant bits of the result, and then sign extends to full length. To handle `sb` or `sh`, the DTIM replicates the subword into all positions of the data write signal's word. It asserts byte write signals (`ByteMask`) to indicate which bytes within the word should be written based on the size of the store (indicated by `Funct3`) and the offset within the word (in the least significant bits of `Adr`). These operations are referred to as *subword loads and stores*. For RV64, `lw`, `lwu`, and `sw` are also subword operations because the native word size is 64 bits.

Fig. 7.9 shows the LSU, including subword load and store blocks for RV64. The RV32 design is similar but drops support for `sd`, `ld`, and `lwu`, and only produces a 4-bit `ByteMask` and 32-bit `WriteData` and `ReadData`.

The LSU supports subword operations, but the random access memory (RAM) always reads and writes full-length *native* words. When the core doesn't have floating-point hardware, the native length is `LLEN = XLEN`. When floating-point is added in Chapter 16, the native length

Another way to perform a subword write to memory is to read the word from memory, modify it, and write it back. This is slower and consumes more power. The hardware to implement bit or byte enables is modest, so most static random access memory (SRAM) generators and field programmable gate array (FPGA) memories support such enables.

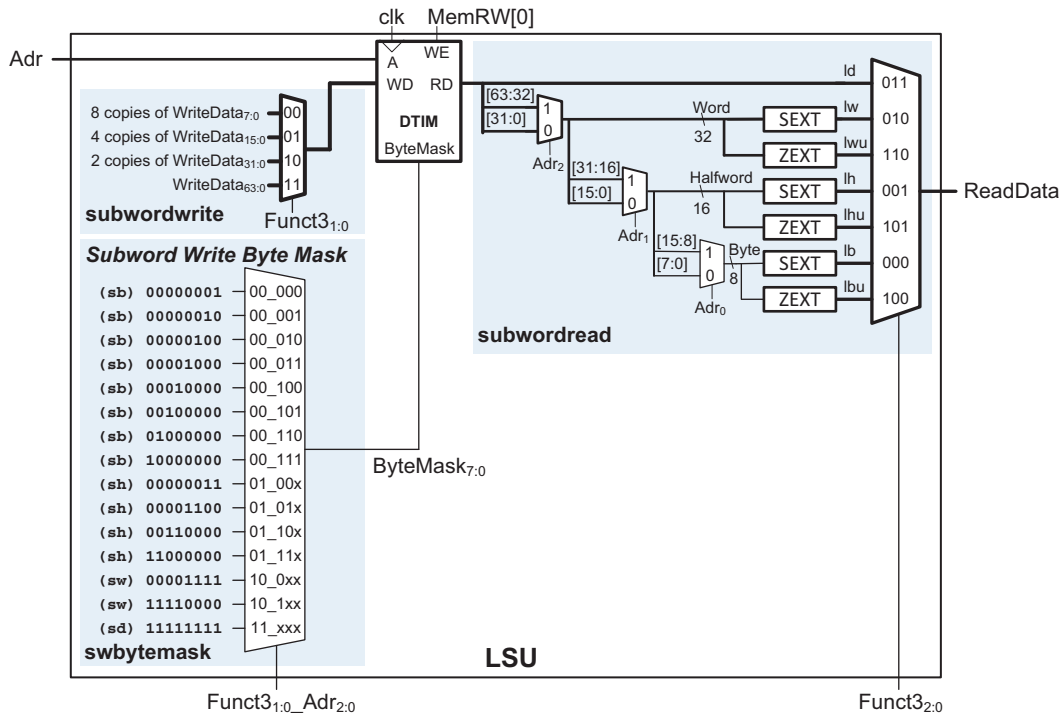


Fig. 7.9 RV64 LSU.

becomes $LLEN = \max(XLEN, FLEN)$. Read $LLEN$ as the “longer” of $XLEN$ and $FLEN$.

For example, a RV32GC configuration has $XLEN = 32$ but $FLEN = 64$, so the native accesses are 64-bit words. The `ReadData` and `WriteData` bus widths are this native length.

Section 8.4.7 adds big-endian support. Section 9.5 describes how to access data from an external memory over a bus interface. Chapter 12 covers a fully-featured LSU with a data cache and memory management unit (MMU). Chapter 17 describes the atomic memory extensions.

7.1.7 Sample Program

To visualize the processor in action, consider running the program in Code Example 7.2, starting at address `0x80000000`, and assume address 4 contains 42.

Code Example 7.2 `ch7.s` SAMPLE PROGRAM

```
start:
0x80000000: lw x1, 4(x0)           00402083
0x80000004: sw x1, 8(x0)           00102423
0x80000008: add x2, x1, x1          00108133
0x8000000c: beq x1, x2, done        00208463
loop:
0x80000010: jal x0, loop           0000006f
done:
```

Fig. 7.10 shows the `lw` instruction running on the single-cycle processor, with the relevant datapath signals highlighted in blue.

- ▶ **Fetch instruction:** The PC is initially at `0x80000000`. The IROM reads from this address to find $Instr = 0x00402083$.
- ▶ **Decode instruction:** The instruction fields are sent to the controller, register file, and extender. The controller produces the appropriate control signals.
- ▶ **Get source operands:** The register file reads $R1 = 0$ from register `x0` and reads garbage for `R2`. The extender produces $ImmExt = 4$. The source multiplexers choose $SrcA = 0$ from `x0` and $SrcB = 4$ from the immediate.
- ▶ **Execute:** The ALU adds these two sources to produce $ALUResult$ and $IEUAdr = 4$.
- ▶ **Read memory:** The DTIM reads from this address to find $ReadData = 42$.

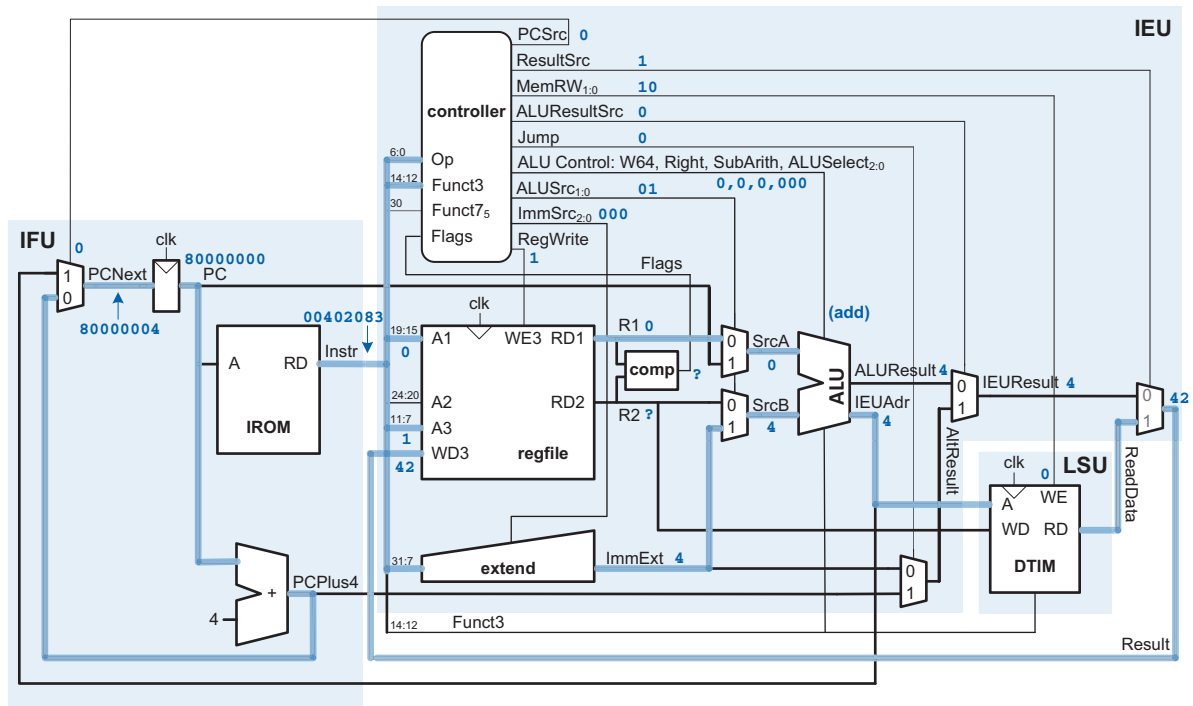


Fig. 7.10 Single-cycle processor executing `lw` instruction.

- ▶ **Write result:** The result multiplexer chooses `Result = 42`. The register file writes 42 into `x1`.
- ▶ **Increment PC:** Meanwhile, the PC incrementer computes `PCPlus4 = 80000004`, which is selected as `PCNext`.

On the next cycle, PC is 80000004, so the IROM reads `Instr = 00102423`. Predict the rest of the results until you are comfortable with the processor operation.

7.2 PIPELINING THE RV32I CORE

The cycle time of the single-cycle processor is limited by the longest path to execute an instruction, including fetching the instruction, reading from the register file, performing an ALU operation, accessing data memory, and writing the result back to the register file. Pipelined processors shorten the cycle time and improve throughput by separating the processor into multiple pipeline stages; hence, the next instruction can start before the current one completes. This introduces a set of *hazards* when one instruction depends on the results of a previous instruction that has not yet completed.

PCInkE = PCE + 4 is recomputed in the Execute stage rather than being pipelined along from PCPlus4F through Decode and Execute stage registers. The cost of duplicating the PC incrementer is less than the cost of storing an additional XLEN bits in the Fetch and Decode pipeline registers.

- ▶ **A hazard unit**, which resolves pipeline hazards using stalls and flushes
- ▶ **Pipeline registers** with enable and clear inputs to stall or flush the pipe
- ▶ **Bypass multiplexers** to forward results from previous instructions



Fig. 7.11 Wally pipelined processor.

- ▶ **Forwarding logic** controller to drive bypass multiplexers
- ▶ **Control and data signals are staged** through the pipeline to remain associated with their instruction
- ▶ **Signal name suffixes** (F, D, E, M, and W) indicate the pipeline stage

The IROM, register file, and DTIM read asynchronously during the Fetch, Decode, and Writeback stages, respectively. DTIM and register file writes occur in the Writeback stage. The DTIM is written at the beginning of the Writeback stage (at the rising clock edge). But register file writes occur later, at the falling clock edge in the middle of the Writeback stage, because `ResultW` isn't available until after that stage's initial rising clock edge. Thus, register file writes occur in the first half of the Writeback stage, and asynchronous register file reads occur in the second half of the Decode stage (after register file writes complete).

7.2.1 Synchronous Memories

All but the smallest memories use synchronous reads for performance and area efficiency. This section modifies the pipelined processor from Fig. 7.11 to have synchronous IROM and DTIM memories. These memories are enabled with the `DTIM_SUPPORTED` and `IROM_SUPPORTED` configuration parameters in `config.vh`. The LSU and IFU will later add bus-based memories and caches in Chapters 9–13.

Fig. 7.12 compares the LSU timing with asynchronous and synchronous DTIMs. The top half shows the address signals of a basic LSU in the Execute and Memory stages, `IEUAdrE` and `IEUAdrM`. The bottom half shows the timing for either asynchronous read (Fig. 7.12(a)) and as was assumed in Fig. 7.11) or synchronous read (Fig. 7.12(b)). For both, first a read occurs from address 1 (A1) during cycle 2 followed by a write to address 2 (A2) on the rising edge at the start of cycle 4. In the asynchronous read of Fig. 7.12(a), the address `IEUAdrM` stabilizes at the beginning of the Memory stage, after which the read occurs asynchronously. The write address and write data then set up before the end of the Memory stage and the write occurs synchronously on the rising edge of the clock at the start of the Writeback stage, in Cycle 4. Fig. 7.12(b) uses a memory with synchronous read. Writes still occur at the start of the Writeback stage, but reads occur on the rising edge of the clock at the start of the Memory stage. Hence, the address must set up earlier for reads so that the read address, `DTIMAdr`, is available before the rising edge of the clock at the beginning of the Memory stage. Thus, the `DTIMAdr` mux chooses the DTIM address as `IEUAdrE` for reads and `IEUAdrM` for writes.

The DTIM address (A) must setup in the Execute stage for reads and in the Memory stage for writes. So, performing a read after a write causes a pipeline hazard because they both need to access the address port in the

Chapter 8 introduces CSRs and traps. In a processor with traps, it is important to update the state of the system *after* all hazards have been resolved. Otherwise, there is a risk that part of the state has changed before a trap occurs, while another part did not, leaving the system in an inconsistent state. Waiting to *commit* (complete) all changes to the architectural state (registers, memory, and CSRs) until the Writeback stage avoids such problems.

Similar to the register file that reads in Decode and writes in Writeback, the synchronous DTIM reads in Memory and writes in Writeback. There is no completely consistent place to show either of these memories in the pipeline diagram. Both are drawn where the read occurs, and we must remember that writes occur later.

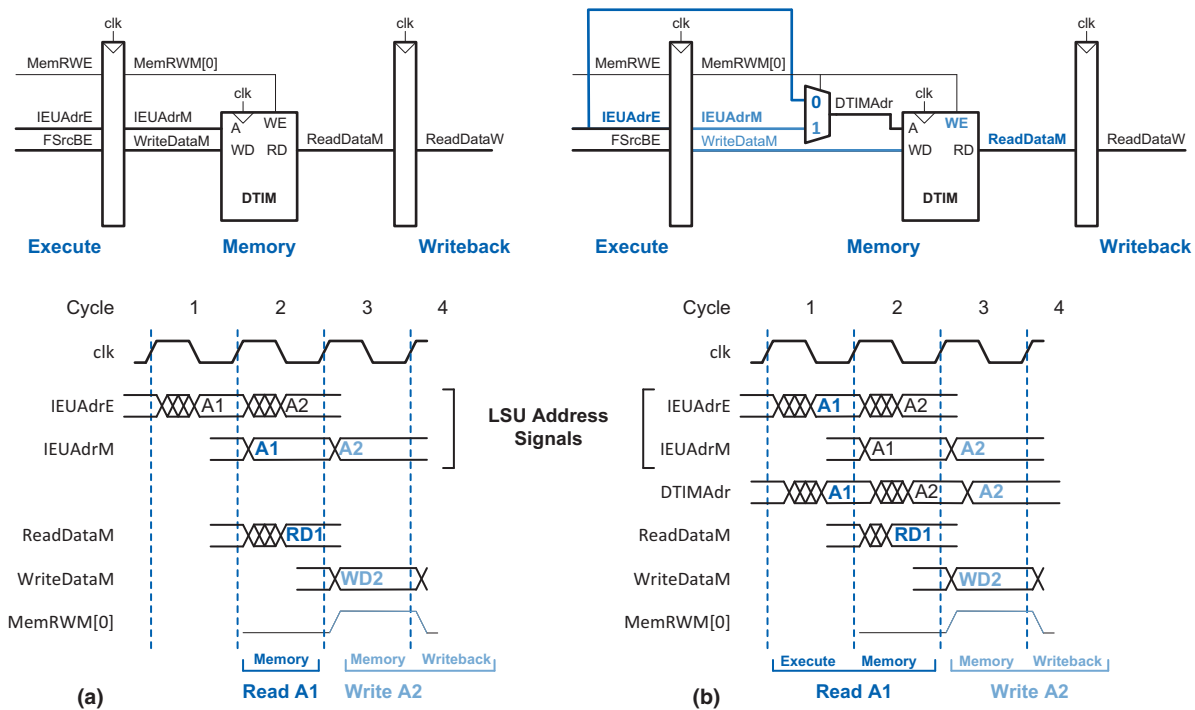


Fig. 7.12 DTIM timing: (a) asynchronous read and (b) synchronous read.

same cycle – that is, while the store is in the Memory stage and the load is in the Execute stage. This is discussed further in [Section 7.2.2](#).

[Fig. 7.13](#) shows the improved pipeline diagram using synchronous memories. Observe that the IROM and DTIM are shifted left to indicate that the read address must set up before the clock edge. The DTIM adds a DTIMAdr mux. Both memories add a chip enable to hold the old read value during a stall.

In summary, an access to the DTIM involves work in the Execute, Memory, and Writeback stages, as shown in [Table 7.8](#).

Table 7.8 LSU DTIM access steps.

Stage	DTIM Read	DTIM Write
Execute	Compute and set up memory address: IEUAdrE = base address + offset	Compute and set up memory address: IEUAdrE = base address + offset
Memory	Read ReadDataM synchronously from IEUAdrE	Set up IEUAdrM, WriteDataM, MemRWM
Writeback	Write ReadDataW into register file	Write WriteDataM synchronously to DTIM

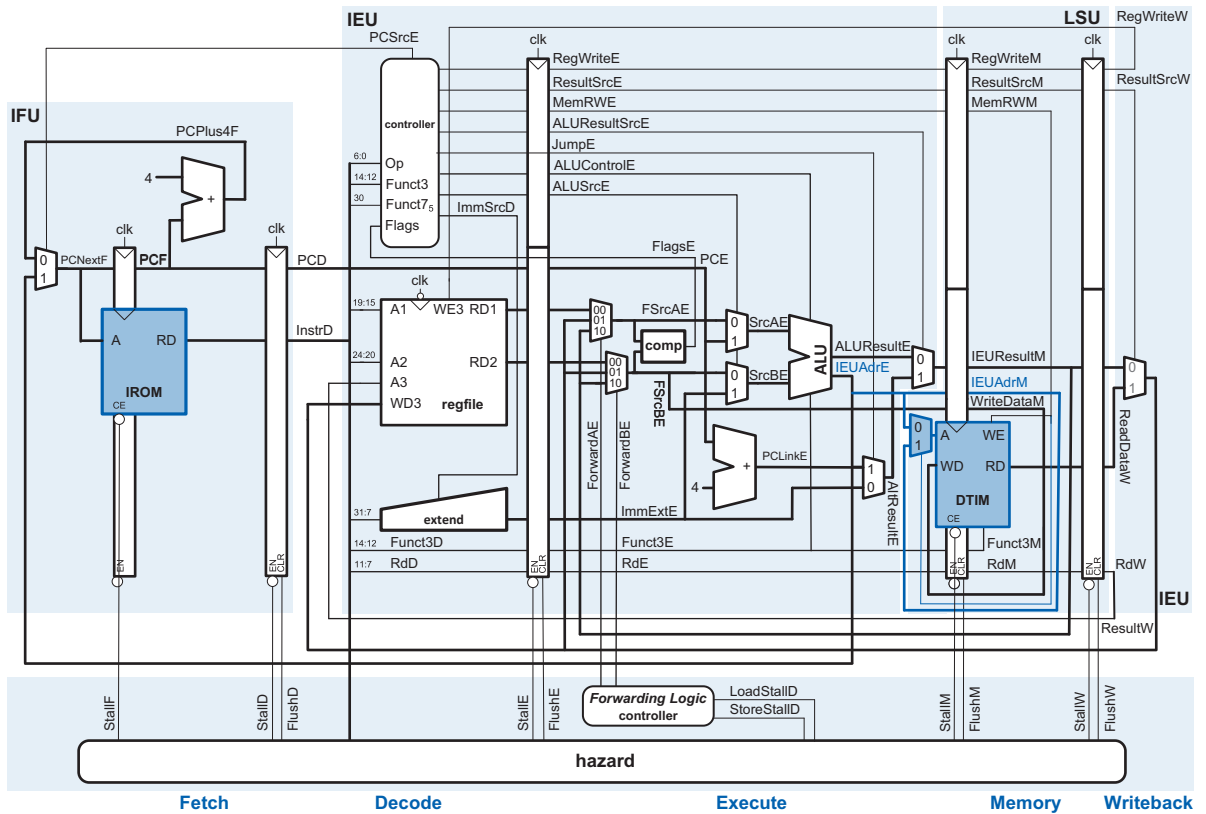


Fig. 7.13 Improved Wally pipelined processor with synchronous memories.

The IROM has a similar issue as the DTIM: the address for synchronous reads must setup before the Fetch stage. Hence, the IROM must be read with `PCNextF` rather than `PCF`. Therefore, reset is applied to the `PCNextF` mux rather than the `PCF` register, as shown in Fig. 7.14. To reduce clutter, the reset to `PCNextF` is not shown in other figures, such as Fig. 7.13.

7.2.2 Hazards

Pipeline hazards occur when an instruction depends on the results of another instruction that has not yet completed. They can be classified as data hazards and control hazards. In a *data hazard*, an instruction needs a value from a previous instruction that has not yet completed. In a *control hazard*, caused by branches and jumps, the decision of which instruction to fetch next has not been made by the time the fetch should occur. Some data

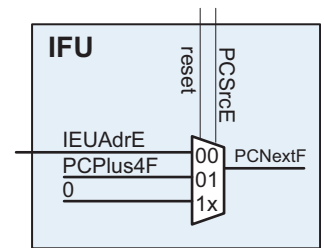


Fig. 7.14 `PCNextF` with reset.

hazards can be solved by *forwarding* (also called *bypassing*) a result from one instruction to the next before that result has been written back to the register file. If the result isn't available to forward by the time it's needed, the processor *stalls* instead, waiting for the result. Control hazards are addressed by predicting which instruction to fetch. If the prediction proves correct, execution continues normally. If the prediction was wrong, the processor flushes the incorrectly fetched instructions, by clearing their pipeline registers, and starts again with the correct instruction.

7.2.2.1 Hazard Example

Fig. 7.15 shows an example of a program that causes hazards on a pipelined processor. The program is on the left, with each row corresponding to an instruction. The pipeline stages are shown on the right in the cycle each one operates for a given instruction.

The `addi` instruction has no dependencies. In cycle 1, the instruction memory (IM) fetches the `addi`. In cycle 2, the register file/sign extender reads the operands 0 and 42. In cycle 3, the ALU executes $42 + 0 = 42$. In cycle 4, the data memory is unused. In cycle 5, the register file writes 42 to `t0`.

The `ori` instruction depends on `t0`. In cycle 3, it reads `t0` from the register file but gets an outdated value because the `addi` instruction won't write `t0` until cycle 5. This is called a *read after write (RAW)* data hazard. The

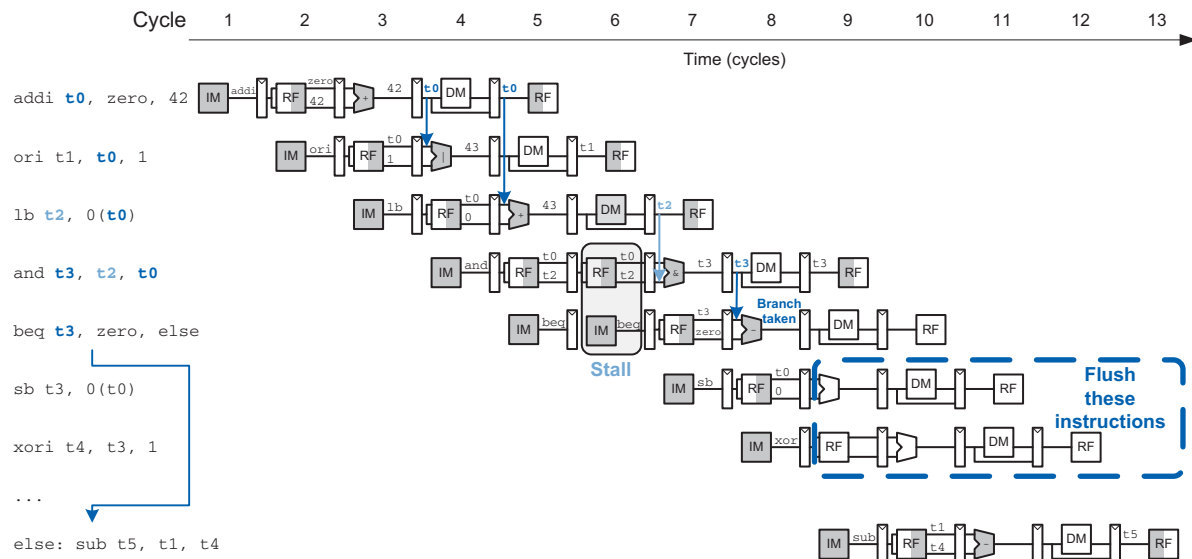


Fig. 7.15 Program with hazards.

pipeline solves the hazard in cycle 4 by forwarding t_0 from the Memory stage of the `addi` to the Execute stage of `ori` so the correct value is used.

The `lb` instruction also depends on t_0 , so there is another read after write hazard. This time, the pipeline solves the hazard in cycle 5 by forwarding t_0 from the Writeback stage of the `addi` to the Execute stage of `lb`.

The `and` instruction depends on both t_0 from `addi` and t_2 from `lb`. `addi` writes t_0 to the register file in the first half of cycle 5 (i.e., at the falling edge of clock), so `and` reads t_0 from the register file correctly in the second half of cycle 5, and no forwarding is necessary. Unfortunately, t_2 comes from the data memory and is not yet available when `and` wants it by the start of cycle 6. Thus, the hazard unit must stall the pipeline during cycle 6: `and` remains in the Decode stage, and `beq` remains in the Fetch stage. On cycle 7, t_2 is now available and is forwarded from the Writeback stage of `lb` to the Execute stage of `and`.

The `beq` instruction must predict whether to take the branch. Branch predictors are discussed in depth in [Section 13.1](#), but our simple pipelined processor always predicts that the branch is not taken and adds 4 to PC as usual. Thus, it fetches `sb` in cycle 7 and `xori` in cycle 8. By cycle 8, the `beq` executes and determines if $t_3 = 0$. If so, as we assume in [Fig. 7.15](#), the branch should have been taken. The hazard unit flushes the `sb` and `xori` instructions and fetches `sub` in cycle 9. Flush is performed by clearing the flushed instruction's pipeline registers or at least all those that contain write enables such as `RegWrite` or `MemWrite`, so that the flushed instruction changes no state and behaves as a `nop`.

7.2.2.2 Pipeline Registers

As shown in [Fig. 7.13](#), each pipeline register takes a stall and flush signal. For example, the register between the Decode and Execute stages takes `StallE` and `FlushE` to stall or flush the instruction entering the Execute stage. Implicitly, all registers are also synchronously resettable to begin at a known initial state. Thus, pipeline registers typically use flip-flops with enable, reset, and clear (`flopencr`) as shown in [Fig. 7.16](#). `Stall` drives the enable (`en`) pin. `Flush` drives the clear pin. Synchronous `reset` has highest priority, initializing `Q` to 0 even if `en` = 0. Enable has next priority, recirculating `Q` when disabled.

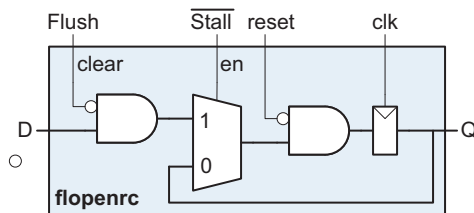


Fig. 7.16 Pipelined register: `flopencr` register with enable, reset, and clear.

Clear has the lowest priority in this implementation, although clear and reset could be ORed together in an alternate register design.

Enable uses a multiplexer in this implementation, although it could be ANDed with `clk` in an alternate register design. Such clock gating requires a latch on the enable to prevent clock glitching, as well as CAD tools to model the gated clock delay for setup and hold time analysis.

7.2.2.3 Forwarding Logic

The pipelined processor adds forwarding logic to the controller, as shown in Fig. 7.17, to forward results from later stages (Memory and Writeback) to the Execute stage. The forwarding logic outputs are combinational. The pipeline forwards a result to the Execute stage if a source register in the Execute stage (Rs1E or Rs2E) matches the valid destination register in a later stage (RdM or RdW). A destination register is valid if it is nonzero (writes to register 0 are ignored) and will be written. If the source register matches destinations in multiple stages, the pipeline picks the earliest one, which corresponds to the most recent instruction. The forwarding logic also detects hazards that it cannot solve by forwarding. These hazards, such as load and store dependencies, are sent to the hazard unit to stall or flush the pipeline.

Forwarding is done with multiplexers for SrcA and SrcB as shown in Fig. 7.13. The forwarding logic computes the forwarding multiplexer select signals Forward{A/B}E. Specifically, the ForwardAE logic for SrcA is:

```
if      ((Rs1E == RdM) & RegWriteM) & (Rs1E != 0) # Mem stage
    ForwardAE = 10
else if ((Rs1E == RdW) & RegWriteW) & (Rs1E != 0) # WB stage
    ForwardAE = 01
else    ForwardAE = 00                                # No forwarding
```

The ForwardBE logic for SrcB is analogous but involves Rs2E instead of Rs1E.

The result of a load instruction is not available until the end of the Memory stage. A dependent instruction requires the result by the start of the Execute stage. Therefore, a load immediately followed by use requires a one-cycle stall. This is called a *structural hazard* because the structure of the pipeline cannot forward the load result to the next instruction. This can be detected in the Decode stage (LoadStallD) by checking if the instruction in the Execute stage is a load (indicated by bit 1 of the memory read/write enable signal, MemRWE[1] = 1) and if the load destination register matches either source register in the Decode stage (MatchDE = 1).

```
MatchDE = (Rs1D == RdE | Rs2D == RdE)
LoadStallD = MemRWE[1] & MatchDE
```

As discussed in Section 7.2.1, the synchronous DTIM needs the read address to setup in the Execute stage and the write address to setup in the Memory stage. This creates another structural hazard that a DTIM store cannot immediately be followed by a load because both would need the address port at the same time. This StoreStallD is detected

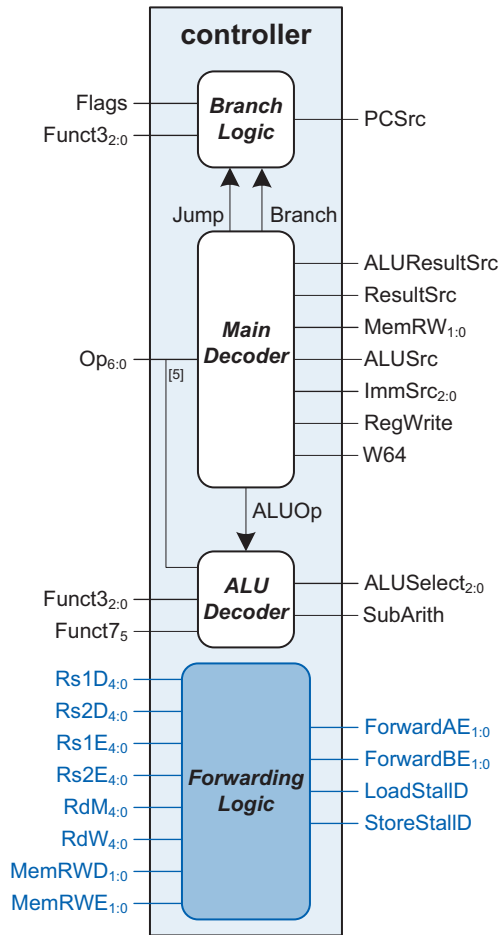


Fig. 7.17 Forwarding logic in controller.

when the instruction in the Execute stage is a store ($\text{MemRWE}[0]$) and the instruction in the Decode stage is a load ($\text{MemRWD}[1]$).

$$\text{StoreStallD} = \text{MemRWE}[0] \ \& \ \text{MemRWD}[1]$$

7.2.2.4 Hazard Unit

The hazard unit monitors various hazards that cannot be resolved by forwarding, such as load-use dependencies and taken branches. Fig. 7.18 shows the general architecture of the hazard unit. The unit receives various hazard signals from the IEU such as PCSrc , which is 1 when a branch is taken, and LoadStall or StoreStall , which are from the forwarding logic within the controller. Using these hazard inputs, the Flush Cause Generation block computes FlushXCause signals for each pipeline stage X that

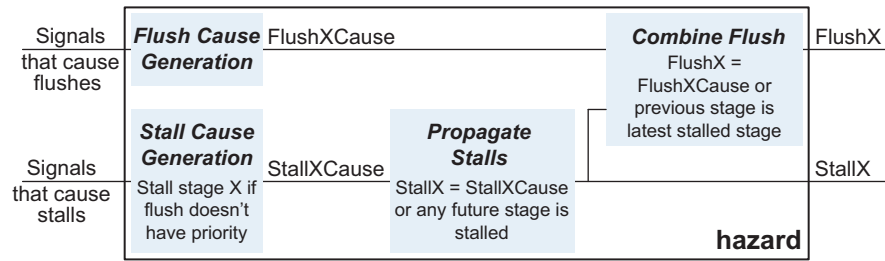


Fig. 7.18 Hazard unit architecture.

might be flushed. Similarly, the Stall Cause Generation block computes `StallXCause` signals. When a stage stalls, all previous stages must also stall to prevent piling up. The Propagate Stalls block stalls all stages prior to a stalled stage. Finally, when a pipeline stage is stalled, the next stage should be flushed unless that stage is also stalled. The Combine Flush block flushes all stages with a `FlushXCause` or after the latest stalled stage.

Branches and jumps are taken in the Execute stage. So, upon a taken branch or jump, two instructions (those in the Fetch and Decode stages) must be flushed from the pipeline. `PCSrcE` indicates that a branch or jump is taken (in addition to selecting `PCNextF = IEUAdrE`, the calculated target address). Thus, for the simple pipelined processor in this chapter, `FlushDCause` and `FlushECause` are raised for jumps and taken branches.

```
FlushDCause = PCSrcE
FlushECause = PCSrcE
```

Flushes have priority over stalls, so flushes gate the stall cause. The pipeline stalls in the Decode stage for a load or store stall, unless the stage is flushed by the earlier logic (`FlushDCause` or `FlushECause`).

```
StructuralStallD = LoadStallD | StoreStallD
StallDCause = StructuralStallD & ~FlushDCause
```

Fig. 7.19 shows the hazard unit for the simple pipelined processor. In this version, it only causes Decode and Execute stage flushes and Decode stage stalls; it will be extended in later chapters. The OR gates in the Propagate Stalls logic backpropagates a stall to earlier stages. The AND gates in Combine Flush detect the latest unstalled stage, and the OR gates flush this stage and any others where a flush is caused. For example, if only Fetch and Decode are stalled, Execute is the latest unstalled stage and must be flushed.

The hazard unit is one of the most difficult parts of the processor to get right, especially when we add traps, multicycle bus and cache operations, etc. It has been the source of many subtle bugs during the Wally design, which means it might still have other subtle bugs lurking. Unfortunately, we know of no systematic way to design a provably correct hazard unit.

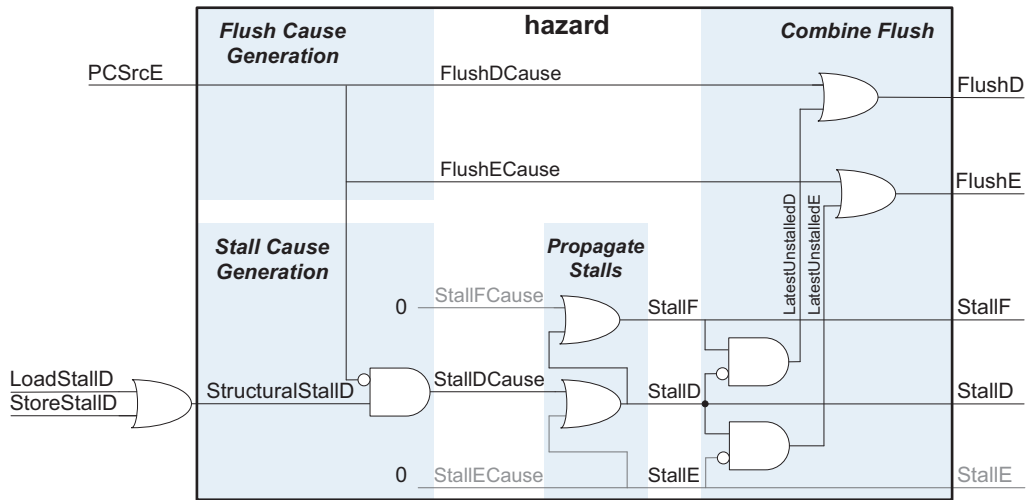


Fig. 7.19 Hazard unit stall and flush logic for the simple pipelined processor.

We will expand the hazard unit in later chapters as we add features such as traps, branch prediction, cache misses, CSRs, and multicycle operations. These added features change the Flush Cause and Stall Cause Generate logic, but they use the same principles for propagating stalls and detecting the latest unstalled stage in the Propagate Stalls and Combine Flush logic.

The hazard unit inputs in Fig. 7.19 are not shown in Figs. 7.11 and 7.13 because of space limitations.

7.2.2.5 Pipelined Processor Performance

Performance is measured by a processor's execution time when running a program or a set of programs. Execution time depends on the number of instructions executed, the clock cycle time (T_C), and the clock *cycles per instruction* (CPI), that is, the average number of clock cycles required to execute a single instruction. For an ideal pipelined processor, CPI is 1 because one instruction completes per clock cycle. However, hazards and stalls degrade CPI. Execution time is calculated as:

$$(7.15) \text{ Execution Time} = \text{Number of instructions} \times CPI \times T_C$$

For a given processor and suite of programs, referred to as a benchmark, the cycle time and number of instructions are constant, so CPI determines performance.

For example, consider CoreMark, an industry standard benchmark discussed further in Section 21.1. Running 10 iterations of the main loop in CoreMark results in the instruction mix and CPI given in Table 7.9

Table 7.9 CoreMark instruction mix.

Type	Number of Instructions	Percent of Total Instructions	CPI
R- or I-type ALU	1,320,259	49.7	1
Jumps / returns	46,750	1.8	3
Branches	588,905	22.2	1.84
Loads / stores	698,738	26.3	1.41
Total	2,654,652	100	1.32

when running on Wally. The benchmark is compiled for rv32im with no branch prediction support (`BPRED_SUPPORTED = 0`) to match the static branch predictor from this chapter.

```
$ cd $WALLY/benchmarks/coremark
$ make ARCH=rv32im_zicsr CONFIG=nobpred_rv32gc run
```

Adding a branch predictor in [Section 13.3.3](#) improves the branch CPI. Adding bit manipulation instructions reduces the instruction count. Together, these improve the overall CPI to 1.14 as described in [Section 21.1](#).

The instruction mix comes from the performance counter results. Branches are taken 41.9% of the time. Our static branch predictor predicts not taken; therefore, misprediction rate is 41.9%. Each misprediction requires an additional two cycles to flush the two instructions in the Fetch and Decode stages after the branch target address is calculated in the Execute stage. Jumps always require three cycles: one to execute the jump and two to flush the next two instructions. The performance counters indicate that 40.1% of loads cause a use after load stall. Thus, the CPI for R-/I-type ALU operations is 1, jumps is 3, branches is $1 + 0.419 \times 2 = 1.84$, and loads/stores is $1 + 0.401 \times 1 = 1.40$. The overall CPI is then a weighted sum of these CPIs: $\text{CPI} = (0.497 \times 1) + (0.018 \times 3) + (0.222 \times 1.84) + (0.263 \times 1.41) = 1.32$. This approximates CoreMark's reported CPI: $\# \text{ clock cycles} / \# \text{ instructions} = 3,36,604 / 2,654,652 = 1.27$.

7.3 SYSTEMVERILOG

The Wally configurable pipelined processor is implemented in SystemVerilog. It uses `$WALLY/config/rv64i/config.vh` to specify the set of supported features described in this chapter for a 64-bit processor. The rv32i

configuration is similar. Wally's rv32e configuration is also similar, but minimizes size by shrinking the register file from 32 to 16 entries and replacing the IROM and DTIM with an AHB interface to off-chip memory described in [Section 9.5](#). The configurable source code is in `$WALLY/src`.

The full source code contains conditional generate statements to produce only the configured hardware. However, it has modules and ports sufficient to support all configurations. It is worth reading and relating to the diagrams in this chapter and the coding guidelines from [Section 4.3](#) to get an understanding of the processor design.

7.4 TEST PLAN

The instructions are tested using the riscv-arch-test suite [riscv-arch-test]. This is an evolving set of tests provided by RISC-V International to offer a basic level of confidence that an implementation conforms to the RISC-V specification.

The configuration in this chapter supports the rv32i_m/I, rv32e_m/E, or rv64i_m/I suite of tests. These include a collection of tests for each of the RV32I or RV64I unprivileged instructions. Each instruction has a series of tests produced by the RISC-V constrained random test generator, `riscv-ctg`. The tests perform an operation and then write a result called the *signature* to memory. If the full set of signatures produced by the device under test matches a reference signature, the test suite passes.

The constrained random test generator is driven by a configuration file that lists the constraints for each instruction [crtg]. [Code Example 7.3](#) lists the constraints for the `addi` instruction. It tests all the different registers being used for `rs1` and `rd`, including cases where `rs1` and `rd` are the same register. To test a range of operand values, it selects values including 0, 1, the most negative, the most positive, walking ones, walking zeros, alternating bits, and random values, including operand pairs that are equal, unequal, and that have each of the four patterns of signs. Walking ones start with a value that is all 0s except the least significant bit. The 1 then shifts (walks) to the left (i.e., 00...001, 00..010, 00...100, etc.). Walking zeros are the similar but start with all 1s and the 0 walks to the left.

Code Example 7.3 **addi** TEST VECTOR CONSTRAINTS

```

addi:
    config:
        - check ISA:=regex(. *I.*)
    opcode:
        addi: 0
    rsl:
        <<: *all_regs
    rd:
        <<: *all_regs
    op_comb:
        <<: *ifmt_op_comb
    val_comb:
        <<: [ *ifmt_val_comb_sgn, *base_rslval_sgn, *ifmt_base_immval_sgn]
    abstract_comb:
        'sp_dataset(xlen,["rsl_val",("imm_val",12)])': 0
        <<: [*rslval_walking, *ifmt_immval_walking]

ifmt_val_comb_sgn: &ifmt_val_comb_sgn
    'rsl_val == imm_val': 0
    'rsl_val != imm_val': 0
    'rsl_val > 0 and imm_val > 0': 0
    'rsl_val > 0 and imm_val < 0': 0
    'rsl_val < 0 and imm_val > 0': 0
    'rsl_val < 0 and imm_val < 0': 0

base_rslval_sgn: &base_rslval_sgn
    'rsl_val == (-2**(xlen-1))': 0
    'rsl_val == 0': 0
    'rsl_val == (2**(xlen-1)-1)': 0
    'rsl_val == 1': 0

ifmt_base_immval_sgn: &ifmt_base_immval_sgn
    'imm_val == (-2**(12-1))': 0
    'imm_val == 0': 0
    'imm_val == (2**(12-1)-1)': 0
    'imm_val == 1': 0

rslval_walking: &rslval_walking
    'walking_ones("rsl_val", xlen)': 0
    'walking_zeros("rsl_val", xlen)': 0
    'alternate("rsl_val",xlen)': 0

```

Table 7.10 lists the number of test cases per instruction for RV32I, RV32E, and RV64I. These statistics can be found using `$WALLY/bin/testcount.pl`. The tests are continually evolving, and the test count has increased slightly since this data was last collected in 2022. All these load and store tests are aligned so they do not cause misalignment faults (see Section 8.2.5). RV64 has more test cases because there are more operand values to cover.

Table 7.10 riscv-arch-test I cases.

Instruction	RV32I Tests	RV32E Tests	RV64I Tests
add-01	588	583	679
addi-01	561	561	619
addiw-01			625
addw-01			678
and-01	584	588	677
andi-01	562	554	619
auipc-01	64	63	63
beq-01	583	590	679
bge-01	590	586	674
bgeu-01	726	728	821
blt-01	580	586	674
bltu-01	726	727	814
bne-01	584	585	678
fence-01	1		1
jal-01	32	16	32
jalr-01	33	27	33
lb-align-01	33	18	32
lbu-align-01	32	17	33
ld-align-01			32
lh-align-01	32	16	32
lhu-align-01	32	16	32
lui-01	63	63	63
lw-align-01	32	16	32
lwu-align-01			32
misalignl-jalr	1		1
or-01	587	589	678

(Continued)

Table 7.10 riscv-arch-test I cases. Cont'd

Instruction	RV32I Tests	RV32E Tests	RV64I Tests
ori-01	560	557	622
sb-align-01	70	78	135
sd-align-01			132
sh-align-01	71	71	133
sll-01	89	88	153
slli-01	88	89	151
slliw-01			150
sllw-01			153
slt-01	584	581	676
slti-01	560	561	617
sltiu-01	699	697	759
sltu-01	722	722	819
sra-01	90	90	154
srai-01	87	87	152
sraiw-01			152
sraw-01			154
srl-01	92	89	153
srli-01	90	88	153
srliw-01			152
srlw-01			154
sub-01	592	584	670
subw-01			678
sw-align-01	68	68	132
xor-01	588	583	675
xori-01	566	559	624
Total	12,642	12,521	17,836

The riscv-arch-test suite has several limitations. It does not test interactions between instructions, such as those that cause pipeline hazards. It also has limited coverage of the instruction operands. Wally uses other functional coverage tests to fill these gaps.

See [Section 5.11.6](#) for instructions of how to run a regression including these tests.

7.5 SUMMARY

This chapter has developed a processor that runs the RV{32/64}I base instruction set. The processor is partitioned into an instruction fetch unit (IFU), integer execution unit (IEU), and load/store unit (LSU). The processor has no external memory interface, so it uses an instruction ROM (IROM) in the IFU to hold instructions and data tightly-integrated memory (DTIM) in the LSU to hold data. The architectural state comprises the program counter (PC) and the 31 registers in the integer register file (IRF).

The IFU contains the IROM, organized as a 32-bit wide read-only memory. It also contains an incrementer to add 4 to the PC and a mux to select $PC + 4$ or the branch/jump target address from the IEU.

The IEU contains a datapath and a controller. The datapath has the IRF, extension logic to assemble an immediate from various bits of the instruction, and an ALU capable of addition/subtraction, shifts, and logical operations. Branch comparison is on the critical path for simple processors, so the datapath has a fast branch comparator separate from the ALU. The datapath also has a variety of multiplexers to choose sources and results appropriately. The controller contains combinational logic that decodes the `op` and `funct` fields of the instruction to determine write enables and multiplexer selects.

The LSU contains the DTIM, organized as an XLEN-bit wide RAM with a single read/write port and byte write enables. It has subword read multiplexers controlled by the least significant address bits and the access type to choose a subset of the word and sign or zero-extend the subword to XLEN-sized ReadData. It also has subword write logic to replicate the smaller WriteData value and turn on the appropriate byte enables.

We begin with a single-cycle core, then convert it into a pipelined design by adding pipeline registers and hazard/forwarding logic and adapting timing for synchronous memories (DTIM and IROM). The five stages of the pipeline are:

Fetch: Read instruction from IROM and update PC

Decode: Decode instruction to compute control signals, read registers from IRF

Execute: Execute ALU, branch, and jump instructions and calculate an address for loads/stores

Memory: Read or write DTIM if applicable

Writeback: Update IRF with new result

Pipelining greatly increases the clock frequency at a modest increase in complexity. Hazards are the main challenge for a pipelined processor. When one instruction depends on the result of a previous instruction that is not yet written back to the register file, the forwarding logic, in many cases, resolves the problem by forwarding the result of the previous instruction to multiplexers at the start of the Execute stage for the current instruction. Unfortunately, load instructions have a two-cycle latency, and their result is not available until the end of the Memory stage. Hence, the hazard unit stalls the pipeline for a cycle when a load is immediately followed by a dependent instruction. A store followed by a load also causes a stall because of a structural hazard with the memory's address port. Finally, the direction of a branch is not known until the comparison is in the Execute stage. The processor continues fetching consecutive instructions as if the branch is not taken, so it must flush the next two instructions when it discovers that a branch should be taken.

The Wally pipelined processor is described using SystemVerilog in the Wally repository. To instantiate only the hardware needed for the simple processor described in this chapter, use the `rv32i`, `rv32e`, or `rv64i` configurations. These configurations have been verified using the integer tests in the `riscv-arch-test` suite. The next chapters add features to this pipelined processor.

Privileged Operations

8

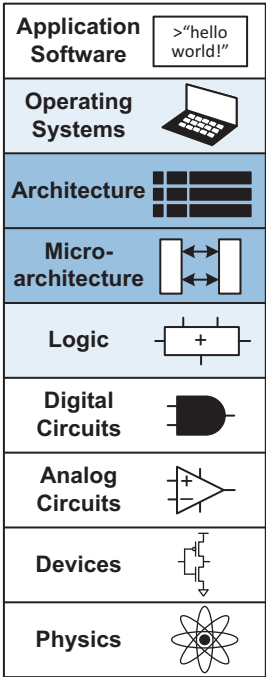
- 8.1 Principles
- 8.2 RISC-V Practices
- 8.3 Test Plan
- 8.4 Wally Implementation
- 8.5 Summary

In the basic integer core described so far, a program can access any feature and any location in memory. This raises two problems: a faulty program can crash the machine, and a malicious program can compromise the machine. Processors add privileged modes to provide protection and security to prevent these problems.

Processors also must be able to respond to unusual program events called *exceptions* or *faults* (e.g., trying to access nonexistent memory) and external events called *interrupts* (e.g., a keystroke or periodic timer). The processor responds with a *trap* that saves the current program counter (PC), raises the privilege mode if necessary, and jumps to a *trap handler* that is located at a specified memory address. The trap handler may save other state, respond to the trap, restore the saved state, and restore the PC to continue the program.

This chapter introduces privileged operations and traps as well as the *control and status registers* (CSRs) and related instructions needed to support them. [Section 8.1](#) gives an overview of the principles and motivations for privileged operations. [Section 8.2](#) then discusses RISC-V privileged features, including privileged modes, privileged instructions, CSRs, traps, and a sample trap handler. [Section 8.3](#) describes the test plan. [Section 8.4](#) explores an implementation of the privileged features. Memory management, including virtual memory and physical memory protection (PMP), is such a large and important part of privileged operations that it is mostly covered in [Chapter 11](#).

Other privileged features include debug mode and hypervisors. *Debug mode* allows a debugger to step through a program one instruction at a time and inspect all of the architectural state. This is inherently a privileged operation because the debugger could access sensitive information. A *hypervisor* switches among multiple guest operating systems running on *virtual machines*, giving each the illusion of owning the entire machine. For example, I am writing this book on a Mac but drawing the illustrations in Visio, which only runs on Windows. I use Parallels to create a virtual Windows machine on the Mac, with a hypervisor switching between the



A trap handler can also be referred to as an *exception handler*, *interrupt handler*, or simply *handler*.

Control and status registers (CSRs) are sometimes referred to simply as registers for brevity.

The RISC-V privileged specification is nearly as long as the unprivileged instruction set architecture (ISA) specification, and much more intricate. It is the authoritative source and covers privileged operations in more detail than this chapter could; it is essential reading for developers concerned about these features (see riscv.org/specifications/ratified/).

two. The privileged architecture supports trapping on certain instructions and CSR accesses to support virtual machines. The Hypervisor (H) extension improves virtual machine performance by avoiding the need for some of these traps. Neither debug mode nor hypervisor features are covered in this book.

8.1 PRINCIPLES

The functions of a privileged unit are to provide security and protection and to handle traps. A set of privileged instructions and CSRs supports these functions.

8.1.1 Security and Protection

Security and protection are closely related but distinct terms. In this chapter, we define *security* as the general problem of preventing unauthorized agents from accessing a computer system and detecting and mitigating damage if access does occur. *Protection* refers to specific hardware and software mechanisms to prevent both malicious and accidental access to resources.

Early computers only ran one program at a time, but developers soon discovered a need to share the computer among multiple programs and users. The first general-purpose time sharing was the Compatible Time Sharing System (CTSS) running on a modified IBM 7094 [Corbato62]. Designers immediately discovered the need to protect one program from another and added architectural features to address this need. Intel introduced memory management and protection in its 80286 processor in 1982, but developers of operating systems were slow to extend their code to take advantage of these features because of the challenge of maintaining backward compatibility. For example, Microsoft operating systems such as Windows 95 were notoriously unstable because a bug in one program could bring down the whole system. Almost twenty years after the hardware support for protection on PCs became available, Microsoft Windows 2000 took advantage of it to give mainstream users a moderately stable computing experience.

Malware has become an enormous economic drain and national security threat. Cybercriminals and governments use malware to steal personal information, hold corporate data ransom, and attack critical infrastructure such as power plants and military labs. Around 2010, a computer worm called Stuxnet, attributed to Israel and the United States, reportedly wiped out 20% of Iran's nuclear centrifuges by causing them to spin out of control, slowing its weapons program. In 2015 and 2016, Russian hackers repeatedly attacked Ukraine's power grid and shut off power in the capital city Kyiv. The FBI recorded \$16B of cybercrime losses reported by US businesses and individuals in 2024 [IC3-21]; this is likely just the tip of the iceberg. In 2024, the US National Security Agency (NSA)

“There are only two types of companies: those that have been hacked and those that don't know they have been hacked.”—John Chambers, CEO, Cisco

announced that Chinese hackers had gained access to control portions of US critical infrastructure, including energy, transportation, communication, and water treatment for at least the past five years, and had stepped up cyberattacks on US critical infrastructure ahead of potential conflict in the Pacific or over Taiwan.

Malware threats include confidentiality, integrity, and availability attacks. A *confidentiality* attack steals sensitive information such as credit card numbers. An *integrity* attack tampers with data, such as changing grades on a transcript or encrypting a hospital's patient records until a ransom is paid. An *availability* attack denies service, such as bringing down a website.

Any device connected to the internet is now under nearly constant attack by automated scripts probing for access. Most attacks target known vulnerabilities, such as weak passwords, bugs in outdated programs, or gullible employees. Rigorous information technology (IT) policies of using unique secure passwords, deploying firewalls and antivirus software, quickly patching security holes, and training employees about suspicious messages can mitigate some of these attacks. The most sophisticated attackers, often state sponsored, also seek out new bugs and vulnerabilities that are unknown to vendors and users. Such *zero-day* exploits are used to take control of phones, computers, and critical infrastructure for spying and economic warfare. Zombie armies of compromised computers are also used to probe other computers or launch distributed denial of service attacks.

Securing a large amount of code, especially code from untrusted sources, is very difficult. Larger code bases have more bugs that can be exploited. A first step is to divide a system into at least two *privilege modes*. Most processes operate in user mode (U-mode). U-mode programs can only access resources such as memory and files granted to them by the OS kernel. They are protected from accessing other resources, so even if a U-mode program is malicious or buggy, it cannot read or corrupt resources owned by another program or the OS. When a process needs to request more resources, it must call the OS, which runs in supervisor mode (S-mode). The OS verifies that the request is valid and assigns the resources, then returns to the U-mode portion of the OS kernel. Now, the attack surface is limited to the smaller OS kernel rather than the entire set of programs on the machine.

8.1.2 Traps

Recall that a processor takes a trap to stop normal program execution and jump to a trap handler when an interrupt or exceptional program event occurs. Traps are an important protection mechanism, providing a secure method to invoke system calls at a higher privilege level and a way to stop programs that attempt to access unauthorized data or code.

RISC-V only supports precise traps.

Traps are categorized as precise or imprecise. A precise trap is one associated with a specific instruction such that all previous instructions have retired (completed), and no later instructions have modified architectural state or memory. In contrast, an imprecise trap is one in which the pipeline cannot be restarted at a specific instruction, for example, if some instructions have retired out of order when the trap occurs.

To illustrate imprecise programs, consider the following program:

```
0x1000: div t1, t2, t3
0x1004: add s0, s0, s1
```

Suppose the `div` instruction takes 20 cycles, and the independent `add` instruction takes 1 cycle. Suppose the core allows the `add` to write its result to the register file and retire before the `div` retires. Then suppose a timer interrupt occurs after the `add` but before the `div` retires, causing an imprecise trap. After the trap handler completes, to what PC should it return? If it returns to 0x1000, the `div` will reissue properly, but `add` will execute a second time and produce an incorrect result in `s0`. If it returns to 0x1008, the `div` instruction will not have executed at all, leaving an incorrect result in `t1`.

Precise traps allow a program to resume from a well-defined place. To precisely handle traps, a pipeline should have a retire or commit stage at the end to delay modifying architectural state until all previous instructions have retired. For in-order processors, the writeback stage at the end of the pipeline serves this role. Out-of-order processors are more complicated and may use a reorder buffer to delay committing results to architectural state until all previous instructions have completed.

For example, if a load causes a virtual memory page fault, the trap handler should fetch the missing page and then resume the load without having other partially completed instructions leaving the system in an inconsistent state. Similarly, if a debugger takes a trap when reaching a breakpoint, all the instructions before the breakpoint should be complete, and none of the instructions after the breakpoint should appear to have been started. Handling *imprecise* traps in a way other than terminating the program is complicated and sometimes impossible, so most processors generate precise traps.

8.1.3 Control and Status Registers (CSRs)

Most processors have CSRs that report or change the behavior of the processor. Examples of CSRs are:

- ▶ Registers to control traps, such as enabling/disabling interrupts or saving the program counter
- ▶ Registers to enable/disable optional features
- ▶ Performance counters to assist profiling programs

8.2 RISC-V PRACTICES

RISC-V provides protection and security by supporting up to three privilege modes: *machine*, *supervisor*, and *user*, also referred to as *M-mode*, *S-mode*, and *U-mode*, respectively. As described in [Section 8.2.1](#), these modes enable protection by allowing only higher privilege modes access to some operations, CSRs, and portions of memory so that malicious programs at lower privilege levels cannot access or corrupt critical data or code. The privileged specification also adds a set of instructions (see [Section 8.2.2](#)) to support traps and memory synchronization and a set of CSRs (see [Section 8.2.3](#)), some of which are only accessible in a given privilege mode. [Section 8.2.4](#) defines the state of the system on reset.

The base RISC-V ISA has only M-mode and lacks CSRs. The Zicsr extension provides CSRs, and the U and S extensions provide the additional privilege modes. For quick reference on the last pages of this book, [Table I.8](#) summarizes privileged instructions, and [Tables I.9–I.13](#) summarize the CSRs and traps.

The privileged specification also defines the causes and handling of traps, discussed further in [Sections 8.2.5](#) and [8.2.6](#). As an overview, traps to machine mode operate as follows:

1. Elevate the privilege to M-mode.
2. Save the current program counter in the `mepc` (machine exception PC) CSR.
3. Place a code indicating the reason for the trap in `mcause`.
4. Optionally record additional information about the trap in `mtval`.
5. Record the previous privilege mode and interrupt enable state in `mstatus`.
6. Disable interrupts.
7. Jump to a *trap vector* address (address of the trap handler code) stored in `mtvec`.
8. The trap handler responds to the trap ([Section 8.2.6](#)).
9. The trap handler then uses the `mret` (M-mode exception return) instruction to return to the address in `mepc` and to restore the privilege mode and interrupt enable.

[Fig. 8.1](#) shows the CSRs and trap return instruction (`mret`) used in an M-mode trap.

Most traps in Linux are handled by the S-mode OS. An M-mode trap handler could reroute traps to S-mode (in software), but it is faster for the hardware to directly *delegate* the trap. The `mdeleg` and `mideleg` CSRs (machine exception and interrupt delegation registers) indicate which

The RISC-V privileged specification has evolved over time and is still evolving, so it is important to know which version of the extensions are supported. The Sm1p12 and Ss1p12 extensions indicate that Version 1.12 of the machine- and supervisor-mode specifications are supported.

Note that each flavor of the `epc` is set to the PC of the instruction that traps. For some traps such as interrupts or page faults, the trap handler returns to that PC to rerun that instruction. For other traps, such as an environment call (`ecall`), the trap handler adds 4 to the `epc` so that it returns to the next instruction.

CSRs	
mepc	PC where trap occurred
mcause	Trap cause
mtval	More info about trap (optional)
mstatus	Previous privilege mode and interrupt enable state
mtvec	Instruction address of trap handler

Trap return instruction

mret	Return to mepc and Restore previous privilege mode and interrupt enable
------	--

Fig. 8.1 CSRs and trap return instruction (mret) used in M-mode trap.

exceptions and interrupts should be delegated to S-mode. This way, when such a trap occurs, it can immediately go to the S-mode (OS) trap handler instead of the M-mode handler. S-mode traps are similar to M-mode traps but use a corresponding set of registers, such as `sepc` and `stvec`, and a supervisor trap handler that returns using `sret`.

The current privilege mode is stored in hardware, as described in Section 8.4.1, but it is not directly visible to a program. However, the privilege mode may be changed by the `mret` and `sret` instructions, as discussed in this chapter. The mode is generally obvious: whereas user programs run in U-mode, the OS runs in S-mode and makes SBI calls to M-mode.

Table 8.1 RISC-V privilege modes.

Code	Mode
00	User
01	Supervisor
11	Machine

8.2.1 Privilege Modes

Table 8.1 lists the RISC-V privilege modes and their encodings. The processor boots in M-mode. Simple embedded microcontrollers (“bare metal”) only use M-mode. Secure microcontrollers add U-mode to separate untrusted application code from a smaller trusted secure kernel. Processors running sophisticated operating systems such as Linux also add a supervisor mode (S-mode), which supports virtual memory and provides security while allowing an M-mode supervisor binary interface (SBI) to abstract platform-specific details from the OS, such as sending interprocessor interrupts or rebooting the computer. Section 2.3.6 described the SBI.

8.2.2 Privileged Instructions

Both the RISC-V privileged and unprivileged specification define instructions to interact with privileged features. The unprivileged specification defines the following instructions:

- ▶ `csr*` CSR access instructions (including `csrrw`, `csrrs`, and `csrrc`)
- ▶ `ecall` Environment call
- ▶ `ebreak` Debugger breakpoint

The RISC-V privileged specification introduces the following instructions:

- ▶ `mret` Return from M-mode trap
- ▶ `sret` Return from S-mode trap
- ▶ `wfi` Wait for interrupt
- ▶ `sfence.vma` Synchronize virtual memory by flushing TLB entries

`sfence.vma` is described in [Section 11.2.4](#), and the other instructions are described here.

8.2.2.1 CSR Access Instructions

RISC-V defines a family of instructions to read and modify the CSRs: `csrrw`, `csrrs`, and `csrrc`. Each reads the value of the CSR into `rd`. Then the instructions, respectively, write, set, or clear bits of the CSR specified by `rs1`.

The instructions also come in immediate forms `csrrwi`, `csrrsi`, and `csrrci`, that use a 5-bit unsigned immediate instead of `rs1` to indicate which bit to write, set, or clear.

Programs commonly want to just read or just write the CSRs. `csrr rd, csr` is a pseudoinstruction, *CSR read*, for reading `csr` into `rd`; it translates into the actual RISC-V instruction `csrrs rd, csr, x0`. To write a CSR, pseudoinstruction *CSR write* may be used. `csrw csr, rs1` writes `rs1` into `csr`; it translates into the actual RISC-V instruction `csrrw x0, csr, rs1`.

Pseudoinstructions `csrs/csrc` are also commonly used. They set/clear bits in the CSR as specified by bits that are high in `rs1`. `csrs csr, rs1` makes `csr = csr | rs1`, and `csrc csr, rs1` makes `csr = csr & ~rs1`.

8.2.2.2 Exception Generating Instructions

Two instructions purposefully trigger an exception: `ecall` or `ebreak`. `ecall` triggers an exception to call the OS. For example, in a system with all three privilege modes, a U-mode program uses `ecall` to invoke the S-mode OS through the application binary interface (ABI), and the OS uses `ecall` to invoke the firmware through the supervisor binary interface (SBI).

`ebreak` triggers a breakpoint exception to transfer control to a debugger. A debugger sets a breakpoint by replacing the corresponding instruction with an `ebreak` and recording the replaced instruction. When the program reaches the breakpoint, the `ebreak` trap occurs, entering the debugger code. The debugger may display the current PC and register values. To single-step through the instruction at the breakpoint, the

These and other common pseudoinstructions are listed in [Table L.7](#) (on the last pages of the book).

debugger restores the instruction at the breakpoint, inserts an `ebreak` at the next instruction, and returns to the program.

8.2.2.3 Trap Return Instructions

The `mret` and `sret` instructions return from M- and S-mode trap handlers, respectively. `mret` sets the program counter to the return value stored in `mepc` and restores the previous privilege mode and interrupt enable state from the `mstatus` register. Specifically, it changes the privilege mode to `mstatus.MPP` (machine previous privilege mode) and `mstatus.MIE` (machine interrupt enable) to `mstatus.MPIE` (machine previous interrupt enables). These status bits are described in [Section 8.2.3.1](#).

`mret` then sets `mstatus.MPP` to the least-privileged supported mode (U-mode, if supported) and sets `mstatus.MPIE`. If `mstatus.MPP` was not M-mode, `mret` also clears `mstatus.MPRV` (the Modify PRiVilege mode bit), as discussed in [Section 11.2.1.5](#). `mret` causes an illegal-instruction exception if attempted from S- or U-mode.

`sret` is analogous to `mret` but returns to `sepc` and manipulates the SPP, SIE, and SPIE bits of the `sstatus` register. `sret` causes an illegal-instruction exception if attempted from U-mode or while `mstatus.TSR = 1` (TSR is an acronym for trap on `sret`). `sret` then sets `mstatus.SPP` to U-mode and `mstatus.SPIE` to 1 and clears `mstatus.MPRV`.

Bitfields of a register are indicated by `.<bitfield name>` after the register. For example, `mstatus.MPP` indicates the MPP bitfield within the `mstatus` CSR.

`wfi` will resume program execution when an interrupt is pending even if interrupts are globally disabled. In this case, the pending interrupt is not taken, but the code after `wfi` can poll interrupt sources and respond right away, without needing to save and restore registers as is done in a trap handler. Thus, the system can improve response time by globally disabling interrupts before a `wfi`.

8.2.2.4 Wait for Interrupt

Most processors spend most of their time in an *idle loop*, waiting for an interrupt that indicates the user has pressed a key, moved the mouse, or tapped the screen, or a timer is scheduling the program's next action. The wait for interrupt instruction (`wfi`) is used to save power while the processor is in such an idle loop. After a `wfi` instruction executes, the processor stops executing instructions until an interrupt is pending. `wfi` preferably puts the processor in a low power state, stopping clocks to most of the core and possibly lowering the power supply voltage. To facilitate a hypervisor switching between guest operating systems, if `wfi` is executed in U- or S-mode and `mstatus.TW = 1` (`TW` = timeout wait) or `wfi` is executed in U-mode and S-mode is supported, `wfi` will time out after an implementation-specific delay (possibly 0) by triggering an illegal-instruction exception.

8.2.3 Control and Status Registers (CSRs)

RISC-V supports up to 4,096 CSRs, which are XLEN bits wide. Some CSRs require 64 bits of information and have an associated `h` upper half for RV32. CSRs are specified by 12-bit addresses:

- ▶ The two most significant bits of the address are 00 or 01 for standard read/write registers and 11 for read-only registers.

- ▶ The next two bits are 00 for U-mode, 01 for S-mode, and 11 for M-mode registers. CSRs can only be accessed from a privilege level greater than or equal to their mode.

Table I.9 (see the last pages of the book) summarizes the standard RISC-V CSRs, excluding those associated with hypervisors and debug/trace. The CSR type is M/S/U (machine/supervisor/user) for the mode and RO or RW for read-only/read-write.

Trap registers are described in Section 8.2.5. The supervisor address translation and protection (satp) register for virtual memory is described in Section 11.2.1.1. Physical memory protection (PMP) registers are described in Section 11.2.2.2. Floating-point CSRs are described in Section 16.4.1. The remaining CSRs are described next.

8.2.3.1 Status Register

The 64-bit status register (`mstatus` or `sstatus`) holds information such as the previous privilege level, interrupt enables, floating-point/vector enables, timeout wait, and XLEN for U- and S-mode. The status register is the most complex CSR. All the bits of the status register are accessible in M-mode using the `mstatus` register, and a subset is accessible in S-mode using the `sstatus` register. Fig. 8.2 shows the bitfields of the 64-bit `mstatus` register for RV64.

Fig. 8.3 shows the same bitfields packed into two 32-bit registers, `mstatus` and `mstatush`, for RV32; the relative positions are the same except that SD (state dirty) is shifted into bit 31 to make checking dirty state easy and SXL/UXL are not supported.

Fig. 8.4 shows `sstatus` for RV64; the MIE, MPiE, MPP, MPRV, TVM, TW, TSR, SXL, SBE, and MBE fields are inaccessible from S-mode, but the other fields are the same as `mstatus`. `sstatus/h` for RV32 is the same as `sstatus` except SD is again shifted into bit 31. The bitfields are described next.

MPP/SPP: Previous Privilege

The 2-bit MPP field holds the previous privilege mode from before a trap into M-mode (MPP = 00, 01, or 11 for U-, S-, or M-mode, respectively). The `mret` instruction restores the privilege mode to the value saved in MPP. Likewise, the 1-bit SPP field holds the previous privilege mode before a trap into S-mode (SPP = 0 or =1 for U- or S-mode, respectively), and `sret` restores the privilege mode to the value saved in SPP.

MIE/SIE and MPiE/SPiE: Global Interrupt Enables and Previous Interrupt Enables

A trap handler must disable interrupts, at least initially, so that it is not interrupted while processing an interrupt. The `mstatus.MIE` field enables (1)

This book does not discuss or support CSRs associated with hypervisors or debug/trace, including `dcsr`, `dpc`, `dscratch0`, `dscratch1`, `scontext`, `mtinst`, and `mtval2`.

The privileged specification provides an elaborate description of what occurs when a program writes to reserved CSRs or writes illegal values. This chapter glosses over these special cases. However, occasionally acronyms will appear:

- ▶ **WPRI:** Writes Preserve values, Reads Ignore values (for reserved fields)
- ▶ **WLRL:** Write Legal/Read Legal values only (programs must write/read legal values)
- ▶ **WARL:** Write Any/Read Legal values (writing illegal values is allowed, but they won't stick)

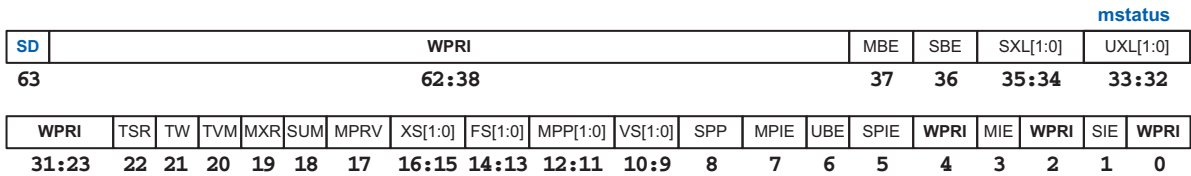


Fig. 8.2 RV64 mstatus register.

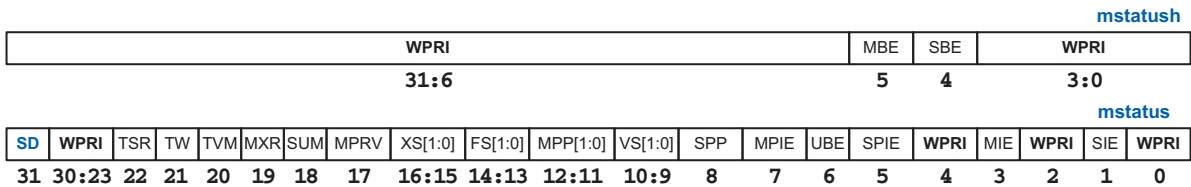


Fig. 8.3 RV32 mstatush and mstatus registers.

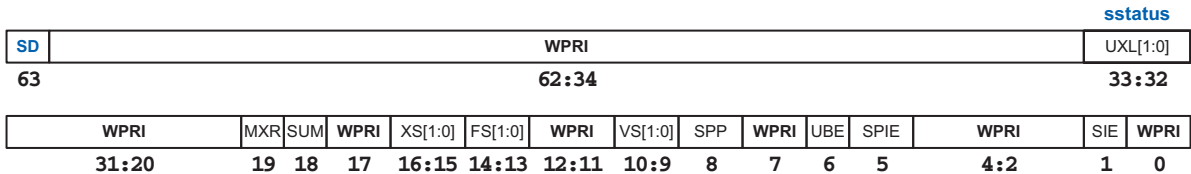


Fig. 8.4 RV64 sstatus register.

Do not confuse the `mie` machine interrupt enable register (containing per-interrupt enables, see Fig. 8.7) with the similarly named `MIE` global interrupt enable field of the `mstatus` (machine status) register (see Figs. 8.1–8.3 and Table I.10).

or disables (0) all M-mode interrupts for a hart operating in M-mode. In M-mode, interrupts delegated to S-mode are always disabled regardless of `SIE`.

When a trap to M-mode occurs, the old value of `MIE` is copied to `MPIE`, and `MIE` is cleared, as described earlier, so that a recursive trap cannot immediately occur. The `mret` instruction restores `MIE` to the value in `MPIE` to resume normal operation. Therefore, the trap handler generally does not need to manipulate the `mstatus.MIE` bit in software.

Likewise, the `SIE` field enables (1) or disables (0) all interrupts delegated to S-mode for a hart operating in S-mode. In S-mode, M-mode interrupts are always globally enabled regardless of `MIE`, although the Machine Interrupt Enable register (`mie`) may be used to disable individual interrupt sources. When a trap to S-mode occurs, `SIE` is copied to `SPIE` and is then cleared. `sret` restores `SIE` to the value in `SPIE`.

MBE/SBE/UBE: Endianness

These bits control whether loads and stores are treated as little-endian (0) or big-endian (1) in M-/S-/U-mode. Instructions are always little-endian so that a disassembler can unambiguously interpret the instruction. An implementation may hardwire these bits to specific values or allow them to change under software control.

Endianness is defined in [Section 2.1.10](#).

FS/VS/XS/SD: Floating-Point/Vector/Extension/State Status

Floating-point and vector extensions introduce new registers. If these registers are in use, the trap handler must save and restore them during context switches, which is a costly operation. The FS and VS fields indicate whether floating-point or vector registers are *dirty* (different from the last saved version) and need saving. The XS field can do the same for a user-defined extension.

[Table 8.2](#) describes the status codes. For example, when FS = 00 (off), the floating-point unit is disabled, and operations accessing floating-point state cause an illegal instruction trap. Enable floating-point by setting FS = 01 (initial) in software.

On reset, typically `mstatus.FS = 00`, and floating-point is disabled. The OS may enable floating-point during the boot process. When writing a bare-metal M-mode program with floating-point instructions, don't forget to turn on floating-point, lest your program take an illegal-instruction trap.

```
lui t0, 2           # 1 in bit 13
csrs mstatus, t0    # Set FS = 01 = initial
```

The hardware sets FS = 11 (dirty) when the processor writes a floating-point register or `fcsr`. When FS indicates a dirty state, the OS must save the floating-point registers and `fcsr` during a context switch. Once the state is saved, FS can be marked as clean (FS = 10), and future context switches do not need to save the floating-point state unless it is written again.

SD (state dirty) is set if FS, VS, or XS is 11. When SD is 0, no state is dirty and context switches do not need to save or recall any floating-point, vector, or other extension state.

Table 8.2 State status codes.

Status	FS/VS Meaning
00	Off
01	Initial
10	Clean
11	Dirty

TW: Timeout Wait

When TW = 1 and the system is in U- or S-mode, a `wfi` instruction will produce an illegal-instruction exception if it does not complete within an implementation-defined bounded time. This allows the hypervisor to switch to a different guest OS rather than wasting time sleeping in the current OS.

MPRV, MXR, SUM: Modify Effective PRiVilege, Make eXecutable Readable, Supervisor User Memory

These bits influence address translation and protection as described in [Section 11.2.1.5](#).

TVM, TSR: Trap Virtual Memory, Trap on sret

Both of these bits facilitate efficient hypervisors, which are not discussed in detail in this book. But generally, when `TVM = 1`, accessing `satp` or executing `sfence.vma` in S-mode raises an illegal-instruction exception. When `TSR = 1`, executing `sret` in S-mode raises an illegal-instruction exception. A hypervisor trap handler could receive these instructions and manage virtual memory or operating systems returns appropriately.

Table 8.3 XLEN codes.

Code	XLEN
01	32
10	64
11	128

uxl/sxl: User/Supervisor XLEN

For RV64, these fields control the effective XLEN in U- or S-mode according to [Table 8.3](#), allowing the processor to emulate RV32. These fields are often hardwired to 10 for processors that do not need to change their width on the fly. The M-mode XLEN is stored in the two most significant bits of the `misalr` register, which is discussed next.

8.2.3.2 misalr Register

The machine ISA (`misalr`) register bitfields indicate which standard extensions are supported by the processor, as given in [Table I.12](#) (on the last pages of the book). The corresponding bit is set for each supported extension.

The association of letters to features is a bit too cute, and RISC-V has long ago run out of letters. Additional extensions have longer names starting with Z (for any mode) or S (for supervisor mode), described in [Table A.6](#). At the time of this writing, a standard way to discover which extensions are available in a hart is still being defined. Linux tests for the presence of some extensions by trying them during boot, taking an illegal instruction trap if the extension is unavailable, and generating a table of which extensions are available.

8.2.3.3 Machine Information Registers

The read-only machine information registers include `mvendorid`, `marchid`, `mimpid`, `mhartid`, and `mconfigptr`. All these registers can be 0 in a simple single-core processor or may have more information to identify the design, hart number, and configuration.

JEDEC assigns the Vendor ID for a fee. RISC-V International assigns a unique Arch ID to a processor architecture upon request. A list of all the RISC-V architecture IDs is at github.com/riscv/riscv-isa-manual/.

8.2.3.4 Counters

RISC-V has up to 32 64-bit performance counters to keep track of various events. The first three are standard, measuring the number of clock cycles, number of timer ticks, and number of instructions retired. The others are platform dependent and optionally configurable; they could track events

such as cache accesses and misses, branch predictions and mispredictions, pipeline stalls, and so on. These counters are helpful to profile program behavior and identify program or microarchitectural areas for optimization.

Performance monitoring CSRs (see [Table I.9](#)) include the following:

- ▶ Up to 32 performance counters: `mcycle/h`, `minstret/h`, and `mhpmpcounter3/h`—`mhpmpcounter31/h`
- ▶ 32-bit `mcounteren` and `scounteren` that determine which privilege modes can read each of the counters
- ▶ `mhpmevent3/h`—`mhpmevent31/h` (hardware performance monitor event) registers to optionally select which events are observed by which counters
- ▶ 32-bit `mcountinhibit` (machine counter inhibit) register to disable counters and, thus, reduce energy consumption of unused counters

In RV64, each performance counter holds a single 64-bit value (i.e., `mhpmpcounter3`). In RV32, each performance counter is divided into low and high 32-bit halves (i.e., `mhpmpcounter3` and `mhpmpcounter3h`).

RISC-V also provides read-only views of the performance counters for less-privileged access: `cycle/h`, `time/h`, `instret/h`, and `hpmcounter3/h`—`hpmcounter31/h`. The first three counters are special. Counter 0 (`cycle`) tracks the number of clock cycles elapsed on the current hart. Counter 1 (`time`) tracks a system timer that may operate at a constant speed independent of each hart's clock. Counter 2 (`instret`) tracks the number of instructions retired (completed) by a hart. The other 29 counters may be hardwired to specific events, configurable among many events based on `mhpmevent3`—`31`, or simply not implemented.

`time/h` is a memory-mapped register, not a CSR.

Each counter can be read and written from M-mode (i.e., with `mcycle`, `mhpmpcounter`) or read-only from any mode (i.e., with `cycle`, `hpmcounter`). Performance counters raise security risks such as timing attacks, in which information is deduced by measuring the speed of variable latency operations such as cache accesses or division. Thus, the `mcounteren` and `scounteren` bits can selectively turn off access to counters from S- and U-mode, respectively. These enables could be read-only as 0 if the counters are only accessible in M-mode.

The `time` register comes from a system timer rather than a per-hart timer, so there is no `mtime` register to which a user could write the time. The system timer is often in the Core Local Interruptor (CLINT) peripheral, which will be discussed further in [Section 20.2](#).

The `rdcycle rd` pseudoinstruction is equivalent to `csrr rd, cycle` to read the current cycle count. Similarly, `rdtime` and `rdinstret` pseudoinstructions read the other special timers. In RV32, they also have an `h` form (`rdcycleh`, `rdtimeh`, and `rdinstreth`) to read the upper 32 bits.

Overflow of 64-bit counters is not a practical concern in present systems. If a counter increments every cycle at 1 GHz, it would have to run continuously for 585 years before overflowing.

The base RISC-V instruction set has no counters, even with the Zicsr extensions. The Zicntr extension provides the standard `cycle`, `time`, and `instret` family of counters. The Zihpm extension provides one or more platform-specific hardware performance monitor counters. The Sscounter-enw extension indicates that `scounteren` is writable (as opposed to read-only 0), and thus counters can be accessible from U-mode.

8.2.3.5 CSR Interaction with Peripherals

The timer value and most interrupts come from peripherals outside the RISC-V core. In many systems, these peripherals are called the *Core Local Interruptor* (CLINT) and the *Platform-Level Interrupt Controller* (PLIC). The CLINT contains the system timer, which may run at a constant rate independent of any hart's sleeping or variable frequency. It also contains timer compare registers and software interrupt registers on a per-hart basis to generate machine timer and software interrupts for each hart. The PLIC prioritizes and routes interrupts from other peripherals to the external machine and supervisor interrupt inputs of each core. These peripherals are accessed using memory-mapped input/output (I/O) rather than through CSRs. However, reading the `time` register returns a value from the CLINT timer. Each hart's `mip` and `sip` register fields are affected by the interrupt inputs from peripherals. The CLINT and PLIC are described in more depth in [Chapter 20](#).

8.2.3.6 Environment Configuration Registers

Some RISC-V extensions raise security risks, especially when the OS or firmware is not designed to support them. For example, the Sstc extension defines a `stimercmp` CSR. If the M-mode firmware has not been enhanced to save and restore `stimercmp` when switching between S-mode processes, such processes could exchange covert information by writing and reading `stimercmp`. Similarly, with Zicbom, a user-level process could use `cbo.inval` on a dirty cache line to leave memory in an inconsistent state. The `{m/s}envcfg` registers allow the system to disable such extensions at lower privilege levels when they are not safely supported.

The machine and supervisor environment configuration registers `menvcfg` and `senvcfg` enable the availability of certain features in lower privilege modes. [Fig. 8.5](#) shows the bitfields in these registers. `menvcfg` is always 64 bits, so `menvcfgh` is used to access the upper half in RV32. The `STCE`, `PBMTE`, and `ADUE` bits control S-mode features, so they only exist in `menvcfg`. The other bits apply to both S- and U-modes and exist in both `menvcfg` and `senvcfg`. In a system with S- and U-modes, the enables must be set in both `menvcfg` and `senvcfg` to enable the feature for U-mode. The bitfields of unsupported extensions are hardwired to 0.

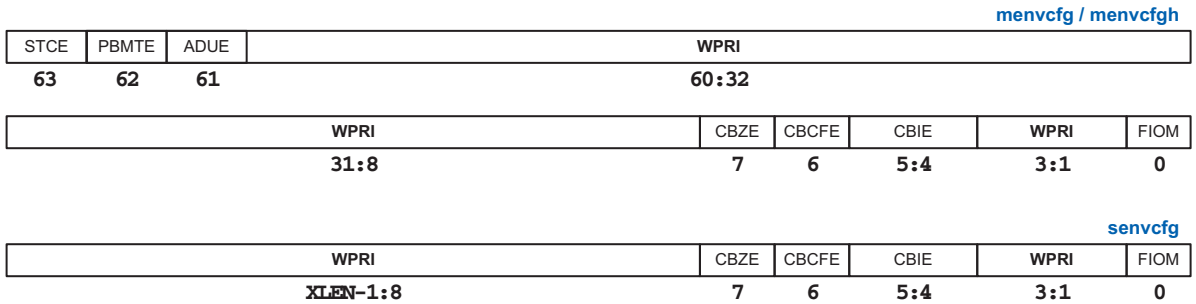


Fig. 8.5 `menvcfg/h` and `senvcfg` registers.

When the Sstc Supervisor Timer Compare extension is supported (see [Section 8.2.5.4](#)), `menvcfg.STCE` enables access to the `stimecmp` register in S-mode. If `menvcfg.STCE = 0`, accessing `stimecmp` in S-mode causes an illegal-instruction exception, and the `mip.STIP` and `sip.STIP` bits revert to their behavior as if Sstc were not implemented.

When the Svpbmt Page Based Memory Types extension is supported (see [Section 11.2.2.1.1](#)), `menvcfg.PBMT` enables this feature. If `menvcfg.PBMT = 0`, the core behaves as if Svpbmt is not supported.

When the Svadu Hardware A/D Update extension is supported (see [Section 11.2.1.4](#)), `menvcfg.ADUE` (A/D Update Enable) enables this feature. If `menvcfg.ADUE = 0`, the core takes a page fault when it needs to update the A or D bits of a page table entry.

When Zicbo* Cache Management Operation extensions are supported, the `CBZE`, `CBCFE`, and `CBIE` bits enable support for the `cbo.zero`, `cbo.flush`, and `cbo.inval` instructions as discussed in [Section 19.1](#).

When the FIOM bit is set, a `fence` instruction in a privilege level below M-mode is modified so that I/O fences also imply order for main memory accesses, as described in [Section 17.2.2](#).

The RISC-V privileged specification also defines the `mseccfg` CSR for security configuration. It controls the Zkr random number generation and Smepmp Supervisor Memory Execution Prevention extensions. Wally does not support either of these extensions, so the register is hardwired to 0.

8.2.4 Reset Behavior

On reset, a core begins in M-mode. The PC is set to an implementation-defined reset vector, typically 0x1000 or 0x80000000. `mcause` is set to a value indicating the cause of the reset or 0 if the system does not distinguish reset conditions. `mstatus.MIE` and `mstatus.MPRV` are cleared, and the `pmp` registers are all disabled by setting the A and L fields to 0. Other hart state is undefined, so the system must initialize all registers and CSRs before using them.

8.2.5 Traps

RISC-V traps are classified as either *exceptions*, which are caused by instructions, or *interrupts*, which are caused by external events such as a key press. Some exceptions are also referred to as *faults*. Recall that traps by default are handled in M-mode but can be delegated in hardware to an S-mode handler, instead of the M-mode handler needing to route the trap to the OS.

Table 8.4 summarizes the standard RISC-V traps, listed in priority from highest to lowest. If multiple traps occur on the same instruction, the highest priority is triggered. For a trap into mode x (machine or supervisor), the `xcause`, `xepc`, and `xtval` registers are written. The most significant bit of the `xcause` register is 1 for an interrupt and 0 for an exception. The trap code is written into the low bits of `xcause`. The exception program counter, `xepc`, is set to the address (PC) of the current instruction, the one that generated the exception or that was interrupted. The trap value register `xtval` can either be cleared to 0 or written with trap-specific information to help the software trap handler (see Table 8.4). The processor then changes PC to a trap handler address specified in `xtvec`. All addresses are virtual when virtual memory is enabled (see Chapter 11).

Traps that are handled in the same privilege mode where they were caused (machine to machine or supervisor to supervisor) are called *horizontal* traps. Traps to a higher privilege level are *vertical* traps. Traps never go to a lower privilege level.

When a trap to M-mode occurs, the privilege mode and machine interrupt enable (`mstatus.MIE`) are stored to `mstatus.MPP` and `mstatus.MPIE`, respectively, so that `mret` can later return to the correct state. The privilege mode is set to M-mode, and `mstatus.MIE` is cleared so that the trap won't be immediately interrupted again. The CSRs associated with M-mode traps are as follows:

`mstatus.MIE` is also known as a *global interrupt enable* because clearing it disables all M-mode interrupts. For example, the hardware clears this bit when taking a trap to disable all interrupts during the trap handler. The `mie` register provides per-interrupt enable bits.

- ▶ `mstatus`: Interrupt enable, previous interrupt enable, previous privilege mode bits
- ▶ `mcause`: Cause of interrupt or exception (see Table 8.4)
- ▶ `mtval`: Value that gives extra trap information (see Table 8.4)
- ▶ `mtvec`: Address (also called vector) of trap handler
- ▶ `mepc`: Exception PC, holding return address from trap
- ▶ `mscratch`: Typically holds the exception stack pointer
- ▶ `mip`: Interrupt pending that lists which interrupts are currently pending
- ▶ `mie`: Interrupt enable that lists which interrupts are taken (enabled) or ignored (disabled)

Table 8.4 Trap cause (`xcause`) and value (`xtval`) in priority order.^a

	Trap	Cause (<code>xcause</code>)		Value (<code>xtval</code>)
		Interrupt Bit (msb)	Trap Code	
Interrupts	M-mode external interrupt	1	11	0
	M-mode software interrupt	1	3	0
	M-mode timer interrupt	1	7	0
	S-mode external interrupt	1	9	0
	S-mode software interrupt	1	1	0
	S-mode timer interrupt	1	5	0
Exceptions	Instruction page fault	0	12	PC
	Instruction access fault	0	1	PC
	Illegal instruction	0	2	Instruction
	Instruction address misaligned	0	0	Target Address
	Breakpoint	0	3	PC
	<code>ecall</code> from U-mode	0	8	0 (<code>a0 = arg</code>)
	<code>ecall</code> from S-mode	0	9	0 (<code>a0 = arg</code>)
	<code>ecall</code> from M-mode	0	11	0 (<code>a0 = arg</code>)
	Store/AMO address misaligned	0	6	Store Address
	Load address misaligned	0	4	Load Address
	Store/AMO page fault	0	15	Store Address
	Load page fault	0	13	Load Address
	Store/AMO access fault	0	7	Store Address
	Load access fault	0	5	Load Address

^aStore and load misaligned faults become lowest priority when misaligned cacheable accesses are handled in hardware using the Zicclsm extension (see [Section 19.2](#)).

Similarly, when a trap to S-mode occurs, the privilege mode and supervisor interrupt enable `mstatus.SIE` are stored to `mstatus.SPP` and `SPIE`, respectively; the privilege mode is set to S-mode; and `mstatus.SIE` is cleared. S-mode traps use a similar set of CSRs but lack views of the M-mode state.

To precisely handle traps, a pipeline should have a *retire* or *commit* stage at the end to delay modifying architectural state until all previous instructions have retired. For in-order processors, the writeback stage at the end of the pipeline serves this role. Out-of-order processors are more complicated and may use a *reorder buffer* to delay committing results to architectural state until all previous instructions have completed.

8.2.5.1 Precise Traps

RISC-V traps are precise. Recall that a *precise* trap is one associated with a specific instruction such that all previous instructions are retired, and no later instructions are retired.

RISC-V exceptions are precise: all instructions prior to the exception complete, and neither the instruction causing the exception nor any subsequent instructions change the architectural state of the system before the exception is handled. This is accomplished by flushing the instruction that caused the exception and any subsequent instructions that are in-flight in the pipeline.

RISC-V interrupts are also precise. Although interrupts are not caused by a particular instruction, they are handled between instructions, with all instructions up to some point retired and no instructions after that point modifying architectural state. Interrupts are handled within a bounded amount of time from the triggering condition.

8.2.5.2 Exceptions

Exceptions are caused by instructions that encounter a special situation. The causes include illegal instructions, accesses to unavailable or misaligned addresses, *ecall*/*ebreak* instructions, and virtual memory page faults (see Table 8.4), as described in this section.

The `medeleg` (exception delegation) register controls which exceptions are delegated to S-mode. It contains a bit for each of the standard exception causes from Table 8.4, as shown in Fig. 8.6, where each grey bit corresponds to an exception cause (unused bits, such as 10 and 14, are set to 0). A one in a given bit delegates that exception to the S-mode handler. For example, if bit 2 of `medeleg` is 1, illegal instruction exceptions call the S-mode handler instead of the M-mode handler. However, traps never reduce privilege levels, so an exception occurring in M-mode will always trap to the M-mode handler.

For example, in a system with a Linux-like S-mode OS, `medeleg` normally delegates at least page faults, U-mode OS calls, and breakpoints to S-mode because these exceptions should be handled by the OS. If they weren't delegated, the M-mode handler would have to pass them along to the OS.

Illegal Instructions

A hart throws an *illegal-instruction exception* when it tries to execute an instruction that either doesn't exist or is unavailable in the current privilege

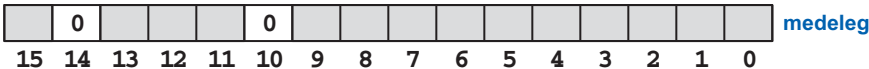


Fig. 8.6 medeleg exception delegation register.

mode. The 0x00000000 instruction is always illegal, and many other instructions are not implemented in the base ISA or standard extensions. Some instructions such as `mret` are unavailable in lower privilege levels. Accessing an unimplemented CSR or a privileged CSR from a lower privilege level also throws an illegal-instruction exception.

Access Faults

A hart throws an *access fault* when it tries to fetch, read, or write from an inaccessible memory location. Examples include reading from a non-existent memory location, writing to a read-only memory (ROM), making word accesses to a peripheral that only handles byte accesses, or accessing a protected region of memory from an insufficient privilege level. Accessible regions are defined by *physical memory attributes* (PMA, essentially generalized memory address decoders) and *physical memory protection* (PMP) registers described in [Section 11.2.2](#). A PMA checker checks for allowed accesses.

Address Misaligned

Most memories can only directly handle accesses from an address that is *naturally aligned*, that is, a multiple of the access size. For example, loading a 4-byte word from 0x1000 is fine, but loading that word from 0x1001 is not because it involves bytes from the words at both 0x1000 and 0x1004. Compilers appreciate this limitation and normally align data correctly, but bad or unusual code might attempt a misaligned access.

A hart may throw a *load or store address-misaligned* exception when it attempts a memory operation to an address that is not a multiple of the access size. For example, a load word instruction to address 0x1002 is misaligned because 0x1002 is not a multiple of 4 bytes. The trap handler could kill the program or it could handle the exception by issuing a pair of aligned accesses and then shift, mask, and merge to deliver the misaligned result and return to the program.

A hart will throw an *instruction address-misaligned* exception when it attempts to branch or jump to an illegal address. Note that branch and jump offsets are implicitly a multiple of 2 and that `jalu` sets the least significant bit of the destination address to 0, so instruction addresses are inherently a multiple of 2. When the Compressed extension is not supported (see [Chapter 14](#)), instructions are 4 bytes long and must be word aligned, so a branch or jump to an odd multiple of 2 causes an exception. When the Compressed extension is supported, instructions only require halfword alignment so instruction address-misaligned exceptions cannot occur.

With the Zicclsm extension (ZICCLSM_SUPPORTED), cacheable main memory regions of memory support misaligned accesses in hardware, as described in [Section 19.2](#). Such accesses may take longer but do not throw a load or store address-misaligned exception.

ecall and ebreak

ecall and ebreak instructions are used for OS calls and to invoke debugger breakpoints. They throw the corresponding exception to elevate the privilege and transfer control.

Page Faults

When virtual memory is enabled, attempting to access a virtual page that is not in physical memory throws a *page fault*. The OS can then fetch the virtual page from storage and place it in physical memory before reattempting the access. Virtual memory also allows pages to be designated as user-only or supervisor-only, and code-only or data-only; violating these protections causes a page fault that might terminate the program. Virtual memory is discussed further in Section 11.2.1.

8.2.5.3 Interrupts

RISC-V defines three types of interrupts (timer, software, and external—TI, SI, and EI, respectively) at two privilege levels, machine and supervisor. Interrupts are managed with the mip and mie *machine interrupt pending* and *enable* registers shown in Fig. 8.7. For example, mip.MEIP is set when a machine external interrupt is pending, mie.MEIE is set to enable machine external interrupts.

Like exceptions, interrupts can also be delegated to the OS (S-mode handler) by setting bits in the mideleg (interrupt delegation) register. mideleg contains a bit for each of the standard interrupt causes from Table 8.4, as shown in Fig. 8.8. For example, bits 1, 5, and 9 correspond to S-mode software, timer, and external interrupts (SSI, STI, SEI).

Only 6 bits of the interrupt registers are not hardwired to 0 (Figs. 8.7 and 8.8). All bits tied to 0 are optimized away.

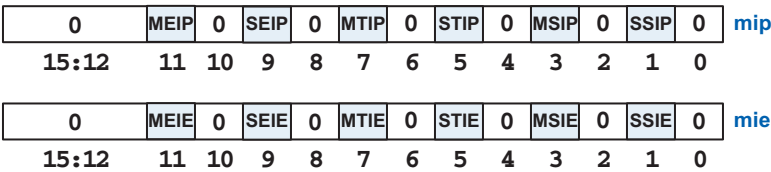


Fig. 8.7 mip and mie registers.

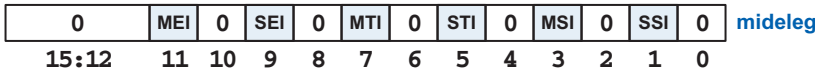


Fig. 8.8 mideleg interrupt delegation register.

The machine and supervisor interrupt nomenclature is confusing. Machine and supervisor interrupts have no fundamental hardware differences, but supervisor interrupts are conventionally delegated to the S-mode handler, while machine interrupts go to the M-mode handler. Thus, in a system with an S-mode OS, `mideleg` is set to 0x222 to delegate supervisor external, timer, and software interrupts to the S-mode handler. [Section 22.2.11](#) describes Linux interrupt handling further.

S-mode code uses the `sip` and `sie` registers which are subsets of `mip` and `mie` that contain only the S-mode bits. Only the bits of `sip` that are delegated are visible: `sip = mip & mideleg`. `sip.SSIP` (S-mode software interrupt pending) is writable in order to clear the interrupt after it is handled; the other `sip` bits are read-only because they are caused by external agents.

When multiple interrupts are pending, RISC-V processors prioritize M-mode interrupts before S-mode and external, then software, and then timer interrupts. This generally causes the most important interrupts to be serviced first.

8.2.5.4 Interrupt Triggering and Generation

Interrupts can be triggered by peripherals or by writing bits in the `mip` CSR, as shown in [Fig. 8.9](#) and as described further below. `mip.SEIP`, `SSIP`, and `STIP` are writable, allowing M-mode code to send interrupts to S-mode. All six bits are readable, returning the values on the right side of the figure.

External interrupts are produced by peripherals to indicate that an event has occurred, such as data being ready from a serial port. A RISC-V core normally has two external interrupt pins (`MExtInt` and `SExtInt`) for the two privilege levels. External circuitry arbitrates among interrupts from all the various peripherals to drive these two pins. In a typical system, the PLIC performs this arbitration and drives the interrupt pins for

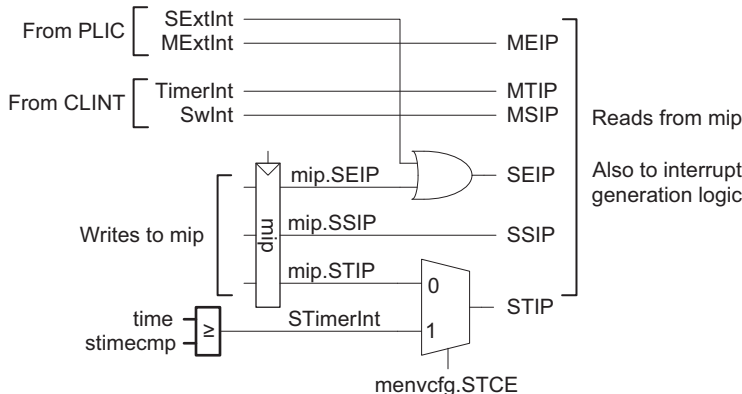


Fig. 8.9 Interrupt sources.

In a system with no S-mode, S-mode interrupts are typically not produced but would go to an M-mode trap handler if they occurred.

RISC-V implementations may choose which interrupts can be delegated. Like most implementations, Wally only permits delegating supervisor interrupts to S-mode.

each hart. The trap handler queries the PLIC to determine which peripheral caused the interrupt, then handles the interrupt from that peripheral and writes a peripheral register to clear the interrupt. S-mode external interrupts can also be triggered by M-mode software making `mip.SEIP = 1`, as shown in Fig. 8.9.

Software interrupts are produced by programs to communicate between harts. In a typical system, the CLINT generates an M-mode software interrupt `SwInt` for a hart when another hart writes to a memory-mapped bit in the CLINT. The trap handler then writes to that memory-mapped bit in the CLINT to clear the machine software interrupt. If the interrupt should be handled by the supervisor, the M-mode handler sets `mip.SSIP` to trigger the S-mode interrupt, which is delegated to the supervisor handler, which then clears `sip.SSIP`.

Timer interrupts provide a heartbeat for the OS scheduler to switch between tasks. The CLINT produces a machine timer interrupt `TimerInt` when the timer exceeds some `mtimecmp` compare value. Some timer interrupts are handled directly by the M-mode firmware, and others are delegated to the supervisor by writing `mip.STIP`. The trap handler then writes the CLINT `mtimecmp` register to a value exceeding the current time to clear the interrupt and prepare for the next event needed by the M-mode firmware or S-mode OS.

Fig. 8.10, shows the logic to generate an interrupt, `InterruptM`. A hart takes an interrupt if interrupts are globally enabled and if any interrupt is both pending in `mip` and enabled in `mie`. The global enable depends on whether the interrupt is delegated to S-mode or not. Interrupts are always globally enabled in U-mode. Delegated interrupts are globally enabled in

It is cumbersome for the timer interrupt to trap to a machine mode handler that sets `STIP` that causes a trap to the supervisor mode handler. The RISC-V `Sstc` extension adds a `stimecmp` register to produce supervisor timer interrupts directly from hardware when `time ≥ stimecmp`. `menvcfg.STCE` must be set to 1 to enable this capability.

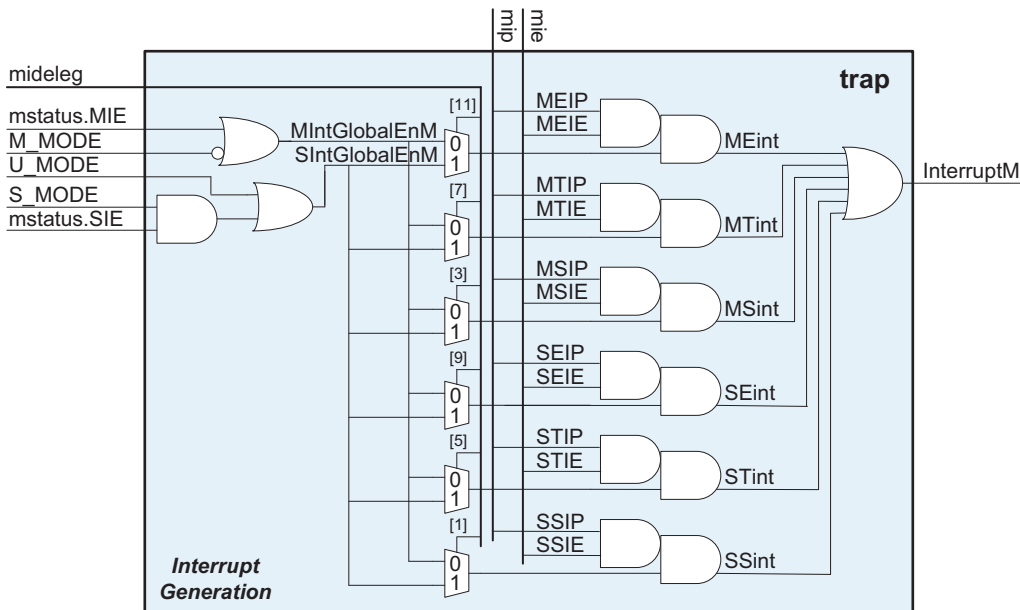


Fig. 8.10 Interrupt generation.

S-mode when `mstatus.SIE` is set and are globally disabled in M-mode. Undelegated interrupts are globally enabled in S-mode and in M-mode when `mstatus.MIE` is set. In other words, undelegated interrupts are enabled when `mstatus.MIE = 1` (regardless of mode) or in any mode other than M-mode.

8.2.5.5 Interrupt Vectoring

The addresses in `mtvec` or `stvec` are the trap vector (i.e., trap handler address) for M- or S-mode. Recall that the hart jumps to the trap vector when a trap occurs. This trap vector must be a multiple of 4, so the bottom two bits are ignored and treated as 0.

To handle interrupts faster, RISC-V optionally allows the core to jump to a different address depending on the interrupt source. This is called *vectored interrupts*. RISC-V enables interrupt vectoring by making the bottom two bits of the trap vector be 01. Exceptions still jump directly to the trap vector (discarding the least significant bit), but interrupts jump to the trap vector plus four times the trap cause (i.e., the trap cause shifted left by 2 bits). For example, a machine software interrupt has a cause of 3, so it jumps to the trap vector + 12. Processors with vectored interrupts typically require that the trap vector is aligned to a multiple of 64 (i.e., its lower 6 bits are 0) so that the vector addition reduces to concatenating the shifted cause with the upper bits of the trap vector address.

8.2.6 Trap Handler

[Code Example 8.1](#) illustrates a sample RV64 M-mode trap handler, which handles two types of traps: timer interrupts and OS calls (`ecall`). It ignores any other traps. If the trap is a timer interrupt, the trap handler adds more time to the timer compare register. If the trap is an `ecall`, it looks for an argument in `a0`. If the argument is 0–3, it changes to that privilege mode (0 = user, 1 = supervisor, 3 = machine) by changing `mstatus.MPP` before issuing `mret`. If the argument is 4, it terminates the program by calling `write_tohost`. For testing purposes, the handler also logs each trap by writing `mcause` and `mtval` to signature memory each time a trap occurs.

Run the trap using:

```
$ cd $WALLY/examples/asm/trap
$ make
$ make sim
```

The RISC-V Advanced Interrupt Architecture (AIA) extension (Smaia/Ssaia) was ratified in June 2023, with additional interrupt features focused on the needs of large, high-performance RISC-V systems. It replaces the PLIC with a more general interrupt controller that supports message-signaled interrupts from PCI Express and similar devices and reduces the interrupt overhead for systems with hypervisors.

There is only room for one instruction at the trap vector, so this instruction is normally a jump to the actual trap handler.

Exceptions are not vectored because they are less sensitive to latency. For example, a timer interrupt must be handled quickly, but an illegal-instruction exception does not.

As introduced in [Chapter 5](#), *signature memory* is memory set aside for the test program to write. After tests are completed, this memory should have a specific *signature*, that is, memory values that show that the tests were completed successfully. The signature depends on the test being run.

Code Example 8.1 `trap.S` TRAP INITIALIZATION AND HANDLER

```

// trap.S
// David_Harris@hmc.edu 11 May 2022
//
// Demonstrates setting up traps and invoking a trap handler.
// The trap handler accepts timer interrupts and ecalls.
// Saves cause and mtval into the signature
// ecall takes one argument in a0
// 0: Change privilege mode to user
// 1: Change privilege mode to supervisor
// 3: Change privilege mode to machine
// 4: Terminate program using write_tohost code

.EQU MTIME,    0x200bff8
.EQU MTIMECMP, 0x2004000

.global rvtest_entry_point

rvtest_entry_point:
    la sp, topofstack           # Initialize stack pointer (not used)
    la s6, begin_signature      # s6 points to signature memory

    # Set up timer
    jal set_timecmp

    # Set up interrupts
    la t0, trap_handler
    csrw mtvec, t0              # Initialize mtvec to trap_handler
    csrw mideleg, zero          # Don't delegate interrupts
    csrw medeleg, zero          # Don't delegate exceptions
    li t0, 0x080                # mie bit 7 = machine timer interrupt
    csrw mie, t0                # Enable machine timer interrupt
    la t0, topoftrapstack
    csrw mscratch, t0           # mscratch holds trap stack pointer
    csrsi mstatus, 0x8          # Turn on mstatus.MIE global interrupt enable

main:
    # Change to user mode
    li a0, 0                    # a0 = 0: argument to enter user mode
    ecall                       # System call to enter user mode

    # Wait for timer interrupts
    li t0, 0x1000               # Loop counter start value

loop:
    addi t0, t0, -1             # Decrement counter
    bne t0, zero, loop          # And repeat until zero

done:
    li a0, 4                    # a0 = 4: argument to terminate program
    ecall                       # System call to terminate program
    j self_loop                 # Wait forever (not executed)

set_timecmp:                    # Set timer compare to 800 ticks later
    la t0, MTIME
    la t1, MTIMECMP
    ld t0, 0(t0)                # Read current timer
    addi t0, t0, 0x800           # Increment timer

```

```

        sd t0, 0(t1)                # Set MTIMECMP = MTIME + 0x800
        ret

.align 2
trap_handler:
    csrrw tp, mscratch, tp          # Trap handlers must be aligned to multiple of 2^2 = 4
    sd t0, 0(tp)                    # Load trap handler stack pointer, tp
    sd t1, -8(tp)                   # Swap mscratch and tp
    csrr t0, mcause                  # Save t0 and t1 on the stack
    csrr t1, mtval
    sd t0, 0(s6)                    # Check the cause
    sd t1, 8(s6)                    # And the trap value
    addi s6, s6, 16                  # Write mcause and mtval to the signature
    bgez t0, exception              # If msb is clear, it is an exception

interrupt:
    jal set_timecmp                 # Must be a timer interrupt
    j trap_return                   # Increment compare for next timer interrupt
                                    # Clean up and return

exception:
    csrr t1, mepc                   # Add 4 to mepc so exception returns to 1 instruction
    addi t1, t1, 4                  # after the excepting instruction
    csrrw mepc, t1
    li t1, 8                        # Is it an ecalls trap?
    andi t0, t0, 0xFC               # If cause = 8, 9, or 11, it is an ecalls
    bne t0, t1, trap_return          # Ignore other exceptions

ecall_trap:
    li t0, 4
    beq a0, t0, write_tohost         # a0 = 4: terminate program
    bltu a0, t0, changeprivilege     # a0 = 0-3: change privilege level
    j trap_return                   # Ignore other ecalls

changeprivilege:
    li t0, 0x00001800               # Mask off mstatus.MPP in bits 11-12
    csrrc mstatus, t0
    andi a0, a0, 0x003              # Only keep bottom two bits of argument
    slli a0, a0, 11                 # Move into mstatus.MPP position
    csrs mstatus, a0                # Set mstatus.MPP with desired privilege

trap_return:
    ld t1, -8(tp)                   # Return from trap handler
    ld t0, 0(tp)                    # Restore t1 and t0
    csrrw tp, mscratch, tp          # Restore tp
    mret                            # Return from trap

write_tohost:
    la t1, tohost
    li t0, 1
    sd t0, 0(t1)                    # 1 for success, 3 for failure
                                    # Send success code

self_loop:
    j self_loop                     # Wait - infinite loop

```

```

.section .tohost
tohost:
# Write to HTIF (host target interface) to terminate program
    .dword 0
fromhost:
    .dword 0
.EQU XLEN, 64
begin_signature:
    .fill 6*(XLEN/32),4,0xdeadbeef
end_signature:
# Initialize stack with room for 512 bytes each for program and trap handler
.bss
    .space 512
topofstack:
# And another stack for the trap handler
.bss
    .space 512
topoftrapstack:

```

The system call (`ecall`) to change privilege mode in [Code Example 8.1](#) is very useful in test code, but it would be dangerous in a real system!

8.3 Test Plan

The privileged test plan involves testing exceptions, interrupts, CSR accesses, and privileged instructions.

At the time of this writing in 2025, `riscv-arch-test` has a small set of privileged tests including misaligned loads, stores, and branches/jumps as well as `ecall` and `ebreak`. Wally has additional privileged tests in `tests/wally-riscv-arch-test`. The memory management tests related to virtual memory, PMP checking, and PMA checking are described in [Chapter 11](#). Traps related to atomic instructions are described in [Chapter 17](#). The other tests are described in this section. The test plan is described in detail in the `cvw` repository under `docs/testplans`.

[Table 8.5](#) summarizes the privileged instructions and the tests that exercise them. [Table 8.6](#) summarizes the CSRs and the tests that exercise

Table 8.5 Summary of privileged instruction tests.

Instruction	Testing
<code>ecall</code>	WALLY-trap-01
<code>ebreak</code>	WALLY-trap-01
<code>mret</code>	WALLY-trap-01
<code>sret</code>	WALLY-trap-s-01
<code>wfi</code>	WALLY-wfi-01 and WALLY-status-tw-01
<code>sfence.vma</code>	Not explicitly tested yet

Table 8.6 Summary of CSR tests.

CSR	Test(s)
Counters	Not tested
fflags	FPU tests in Chapter 16
frm	FPU tests in Chapter 16
fcsr	FPU tests in Chapter 16
sstatus	See Table 8.7
sie	WALLY-sie-01
stvec	WALLY-stvec-01 and WALLY-trap-s-01
sscratch	WALLY-trap-s-01
sepc	WALLY-trap-s-01, WALLY-trap-u-01
scause	WALLY-trap-s-01, WALLY-trap-u-01
stval	WALLY-trap-s-01, WALLY-trap-u-01
sip	Not explicitly tested
satp	WALLY-mmuv-sv32-01, WALLY-mmuv-sv39-01, WALLY-mmuv-sv48-01
senvcfg	Bits tested in the context of features they enable
mvendorid	WALLY-minfo-01
marchid	WALLY-minfo-01
mimpid	WALLY-minfo-01
mhartid	WALLY-minfo-01
mconfigptr	WALLY-minfo-01
mstatus	See Table 8.7
misa	WALLY-misa-01
medeleg	WALLY-trap-s-01, WALLY-trap-u-01
mideleg	WALLY-trap-s-01, WALLY-trap-u-01
mie	WALLY-mie-01
mtvec	WALLY-mtvec-01, WALLY-trap-01
mscratch	WALLY-trap-01
mepc	WALLY-trap-01
mcause	WALLY-trap-01

(Continued)

Table 8.6 Summary of CSR tests. Cont'd

CSR	Test(s)
mtval	WALLY-trap-01
mip	Not explicitly tested
menvcfg	Bits tested in the context of features they enable
pmpcfg[]	WALLY-pmp-01
pmpaddr[]	WALLY-pmp-01

The privileged tests are among the more difficult tests to write. Enumerating the cases to test is lengthy, and checking that the tests cover all of these cases requires care. Our reference model of Spike (see [Section 3.3.8](#)) lacks the peripherals to generate some of the interrupts, so the tests could not be checked against a reference model, increasing the chance of making the same error misinterpreting the specification in the test case as in the RTL. The privileged specification leaves many choices to the implementation, and these tests are only written to check Wally's implementation, not all valid implementations. Even for Wally, these tests only cover basic privileged functions. We believed these tests were better than anything else we could find at the time they were written in 2022 but expect more comprehensive tests will eventually be available from the RISC-V Architecture Test Special Interest Group.

them. The `mstatus/sstatus` CSRs contain many bitfields; the tests for each bitfield are summarized in [Table 8.7](#).

All privileged tests are in `tests/wally-riscv-arch-test/riscv-test-suite/rv{32,64}i_m/privilege`.

`riscv-arch-test` intends to support U- and S-mode tests in directories named `rv{32,64}i_{u,s}`. This naming structure is not yet implemented, so the non-M-mode tests are placed in `rv{32,64}i_m/privilege/testname-{u,s}-01`. Configurations that do not support U- or S-mode should omit the associated tests.

The privileged unit had a high number of difficult bugs for a variety of reasons:

- ▶ Complex specification with many easily-overlooked corner cases
- ▶ Exceptions and CSR accesses disrupt pipeline behavior
- ▶ This unit interacts with peripherals, which were also lightly tested

The Wally privileged tests include the `WALLY-TEST-LIB-32/64.h` file. This file includes a generic trap handler that logs a signature for each trap, including `mcause` and optionally `mtval` and certain `mstatus` bits. It also contains a collection of functions and macros to cause traps and change privilege modes. Most tests call these functions directly. The MMU/PMA/PMP tests contain a table of tests, and calls a function called `run_test_loop` that applies each test. The following sections describe how test programs trigger the various traps.

8.3.1 Generating Exceptions

RISC-V defines 14 standard exceptions as listed in [Table 8.4](#). [Table 8.8](#) gives an example of how each exception is triggered. Note that exceptions caused by atomic instructions are covered in [Section 17.2.3.1](#).

Table 8.7 Summary of `mstatus` bitfield tests.

Field	Test
SD	WALLY-status-fp-enabled-01
MBE	WALLY-endianness-01
SBE	WALLY-endianness-01
SXL	WALLY-status-xlen-01
UXL	WALLY-status-xlen-01
TSR	WALLY-trap-sret-01
TW	WALLY-status-tw-01
TVM	WALLY-mmuv-svxx-01
MXR	WALLY-mmuv-svxx-01
SUM	WALLY-mmuv-svxx-01
MPRV	WALLY-mmuv-svxx-01
XS	<not supported>
FS	WALLY-status-fp-enabled-01
MPP	WALLY-trap-01
VS	<not supported>
SPP	WALLY-trap-s-01
MPIE	WALLY-trap-01
UBE	WALLY-endianness-01
SPIE	WALLY-trap-s-01
MIE	WALLY-mie-01
SIE	WALLY-sie-01

8.3.2 Generating Interrupts

Six interrupts exist, as listed in [Table 8.4](#): timer, software, and external interrupts, each handled in M- or S-mode. [Table 8.9](#) gives examples of how each interrupt can be triggered by a test program. Recall from [Section 8.2.5.4](#) that some interrupts are caused by peripherals and others are caused by writing to the `mip` or `sip` registers.

With the Zicclsm extension (see [Section 19.2](#)), misaligned loads and stores to cachable memory are handled by hardware rather than trapping, but misaligned accesses to peripherals still generate an exception. Therefore, these exceptions are tested by accessing the CLINT.

Table 8.8 Methods of causing exceptions.

Exception	Examples of Exception Generation	
Instruction address misaligned	This exception is not tested because privileged tests run with the Compressed extension, which can never generate misaligned instruction addresses.	
Instruction access fault	<code>jr zero</code>	// Jump to unimplemented address
Illegal instruction	<code>.word 0</code>	// 0x00000000 is an illegal instruction
Breakpoint	<code>ebreak</code>	
Load address misaligned	<code>li t3, 0x2000001</code> <code>lw t4, 0(t3)</code>	// Peripheral access must be aligned // Load from a misaligned address
Load access fault	<code>lw t1, 0(zero)</code>	// Load from unimplemented address
Store address misaligned	<code>li t3, 0x2000001</code> <code>sw t4, 0(t3)</code>	// Peripheral access must be aligned // Store to a misaligned address
Store access fault	<code>sw t4, 0(zero)</code>	// Store to unimplemented address
<code>ecall</code> from U-mode	<code>GOTO_U_MODE</code> <code>ecall</code>	// A macro that switches to U-mode
<code>ecall</code> from S-mode	<code>GOTO_S_MODE</code> <code>ecall</code>	// A macro that switches to S-mode
<code>ecall</code> from M-mode	<code>GOTO_M_MODE</code> <code>ecall</code>	// A macro that switches to M-mode
Instruction page fault	See MMU tests in Section 11.3	
Load page fault	See MMU tests in Section 11.3	
Store page fault	See MMU tests in Section 11.3	

8.4 WALLY IMPLEMENTATION

[Fig. 8.12](#) shows the Wally core with privileged support. Compared to [Fig. 7.1](#), it adds the privileged unit, which includes CSRs and trap logic.

The privileged unit adds architectural state: the CSRs and the 2-bit privilege mode. Privileged instructions operate in the Memory stage and update the architectural state on the rising edge of the clock at the start of the Writeback stage.

8.4.1 Wally Privileged Unit

[Fig. 8.13](#) shows a block diagram of the privileged unit, including major busses connecting components. The major components are the `csr` unit,

which holds the CSRs; the `trap` unit that issues traps; the `privmode` unit that tracks the privilege mode; and `privdec`, which decodes privileged instructions. These blocks all operate in the Memory stage, and the `csr` and `privmode` blocks update state at the start of the Writeback stage. The privileged pipeline registers (`privpiperegs`) delay faults that occur in the Decode and Execute stages so that they are processed in the Memory stage instead.

The `privdec` module decodes the instruction `InstrM` and sends appropriate control signals to the other modules.

Table 8.9 Examples of causing interrupts.

Interrupt	Program to Generate an Interrupt
S-mode software	Write a 1 to <code>sip.SSIP</code> . <pre>li t3, 0x2 # SSIP is bit 1 csrs sip, t3 # Set supervisor software interrupt pending</pre>
M-mode software	Write the MSIP bit of the CLINT. <pre>la t3, 0x02000000 # MSIP register in CLINT li t4, 1 # 1 in the lsb sw t4, 0(t3) # Write MSIP bit of the mip register</pre>
S-mode timer	If <code>Sstc</code> is not supported, write a 1 to the <code>mip.STIP</code> . <pre>li t3, 0x20 # STIP is bit 5 csrs mip, t3 # Set supervisor time interrupt pending</pre> If <code>Sstc</code> is supported, set <code>menvcfg.STCE</code> and write <code>stimecmp</code> (like M-mode timer below)
M-mode timer	Set <code>mtimecmp</code> to <code>mtime</code> + <some offset> by writing to the CLINT and wait. # The following code works for both RV32 and RV64. # RV64 alone would be easier using double-word adds and stores <pre>li t3, 0x30 # Desired offset from the present time mv a3, t3 # Stop waiting for interrupt after this many loops la t4, 0x02004000 # mtimecmp register in CLINT la t5, 0x0200BFF8 # mtime register in CLINT lw t2, 0(t5) # Low word of mtime lw t6, 4(t5) # High word of mtime add t3, t2, t3 # Add desired offset to the current time bgtu t3, t2, nowrap # Check new time exceeds current time (no wraparound) addi t6, t6, 1 # If wrap, increment most significant word sw t6, 4(t4) # Store into most significant word of mtimecmp nowrap: sw t3, 0(t4) # Store into least significant word of mtimecmp time_loop: addi a3, a3, -1 bnez a3, time_loop # Wait for interrupt to occur or loop to time out</pre>

(Continued)

Table 8.9 Examples of causing interrupts. Cont'd

Interrupt	Program to Generate an Interrupt
S-mode external	Use the same code as Machine external interrupts (see below) except set the S priority threshold to 0, set the M priority threshold to 7, and enable source 3 in S-mode at address 0x0C002080.
M-mode external	External interrupts depend on a peripheral to cause them. Wally presently can receive external interrupts from the GPIO and UART peripherals, routed through the PLIC to the ExtInt pin. This set of tests relies on the GPIO to cause an interrupt. The first section of this code configures both the PLIC and GPIO to allow for an external interrupt. This involves raising GPIO priority above the M-mode priority threshold, enabling the GPIO, and enabling input pins as interrupters within the GPIO. <pre> # ===== Configure PLIC ===== # Assume a3 is set to number of iterations to wait for interrupt # M priority threshold = 0 la t3, 0xC200000 # PLIC IntThreshold[M-mode] li t4, 0 sw t4, 0(t3) # PLIC IntThreshold[M-mode] = 0 # S priority threshold = 7 la t3, 0xC201000 # PLIC IntThreshold[S-mode] li t4, 7 sw t4, 0(t3) # PLIC IntThreshold[S-mode] = 7 # source 3 (GPIO) priority = 1 la t3, 0xC000000 # PLIC IntPriority base address li t4, 1 sw t4, 0x0C(t3) # PLIC IntPriority[3] 1 # enable source 3 interrupts in M-mode la t3, 0x0C002000 # PLIC IntEnable[M-mode] li t4, 0b1000 sw t4, 0(t3) # PLIC IntEnable[M-mode][3] = 1 la t3, 0x10060000 # GPIO base address li t4, 0x1 sw t4, 0x08(t3) # GPIO.output_en = 1 enable pin 1 as an output sw t4, 0x04(t3) # GPIO.input_en = 1 also enable pin 1 as an input # for loopback test sw zero, 0x1C(t3) # Clear GPIO.rise_ip sw zero, 0x24(t3) # Clear GPIO.fall_ip sw zero, 0x2C(t3) # Clear GPIO.high_ip sw zero, 0x34(t3) # Clear GPIO.low_ip sw t4, 0x28(t3) # GPIO.high_en = 1 enable interrupt on high input sw t4, 0x0C(t3) # GPIO.output_val = 1 raise pin 1 (cause interrupt) m_ext_loop: addi a3, a3, -1 bnez a3, m_ext_loop # Wait a3 iterations to see if interrupt fires </pre>

GPIO	GPIO Low Interrupt Pending	10060034
	GPIO High Interrupt Pending	1006002C
	GPIO High Interrupt Enable	10060028
	GPIO Fall Interrupt Pending	10060024
	GPIO Rise Interrupt Pending	1006001C
	GPIO Output Value	1006000C
	GPIO Output Enable	10060008
	GPIO Input Enable	10060004
PLIC	...	
	Interrupt Threshold (S-mode)	0C201000
	Interrupt Threshold (M-mode)	0C200000
	Interrupt Enable	0C002000
CLINT	Interrupt Priority	0C000000
	...	
	mtime _{63:32}	0200BFFC
	mtime _{31:0}	0200BFF8
	mtimecmp _{63:32}	02004004
	mtimecmp _{31:0}	02004000

Fig. 8.11 Memory-mapped registers used for interrupt generation.

The CSR unit (*csr*) contains all the CSRs and has the following functions:

- ▶ Reads and writes the CSRs over the *CSRReadValW* and *SrcAM* signals, respectively, connecting to the IEU.
- ▶ During traps, it does the following:
 - o Sends delegation and interrupt register values (*mideleg/mideleg*, and *mip/mie*) to the trap unit (*trap*) and reads the trap cause from the trap unit
 - o Sends *xstatus* bits to *trap* and *privmode*, the privileged mode unit (*privmode*), which holds the current privilege mode
 - o May record *InstrM* (in *xtval*) and *PCM* (in *xepc*) from the instruction fetch unit (IFU) and *IEUAdrM* (in *xtval*) from the IEU
 - o Uses the CSRs (*xtvect*) to determine the trap destination, *UnalignedPCNextF*
- ▶ Sends or receives the remaining signals (discussed in future chapters).
 - o Sends *pmp* and *satp* CSRs to the LSU to control memory management (see [Chapter 12](#))
 - o Sends the *frm* (floating-point rounding mode) CSR to control the FPU, which returns the *SetFFlagsM* signals (see [Chapter 16](#))

The code must change if the peripheral base addresses move or the generally-purpose I/O (GPIO) or platform-level interrupt controller (PLIC) are not available to produce and route external interrupts.

Note that interrupt tests are not simulated in Sail because Sail does not have the same peripherals to generate interrupts. [Chapter 20](#) describes Wally's peripherals.

Recall from the Wally memory map in [Fig. 2.8](#) that the PLIC and CLINT registers are mapped to addresses starting at 0x0C000000 and 0x02000000, respectively. [Fig. 8.11](#) shows the memory-mapped registers used in the [Table 8.9](#) examples. A full list of registers is found in [Chapter 20](#).

Additions to diagrams are highlighted in bright blue with a blue outline. For example, [Fig. 8.12](#) shows the addition of the privileged unit. Modules in gray have not yet been added.

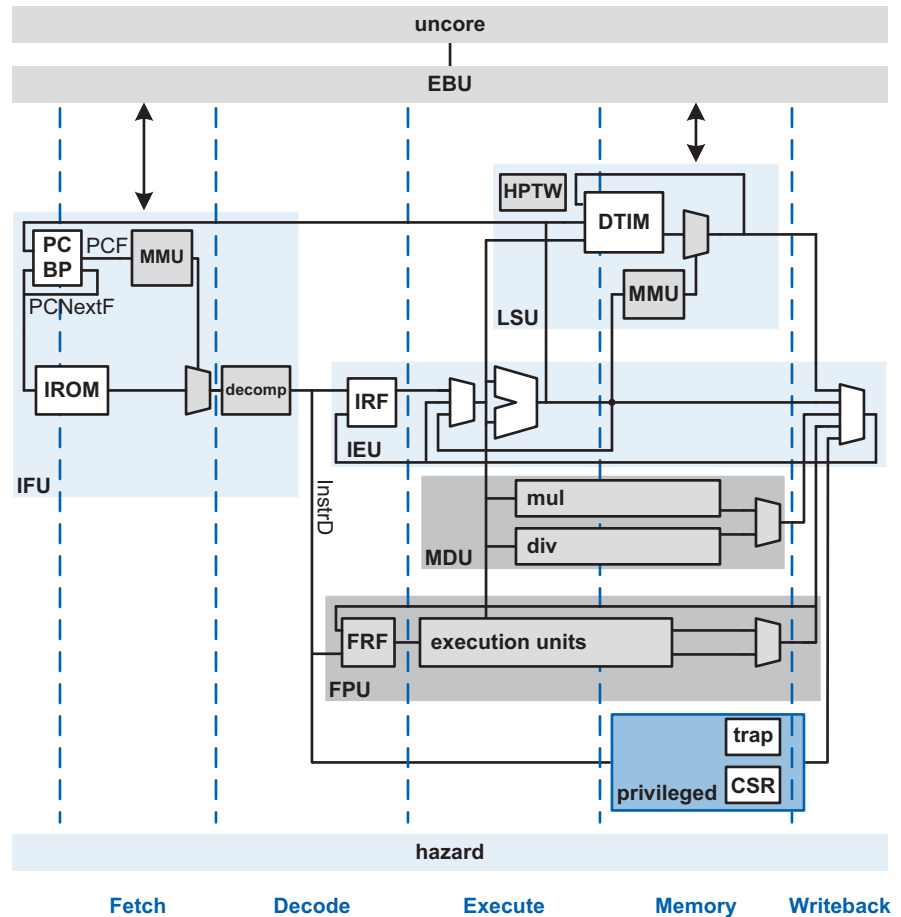


Fig. 8.12 Wally high-level block diagram, highlighting privileged support.

- o The CLINT provides the `MTIME_CLINT` signal for the timer (see [Section 20.2](#))

The `trap` module receives exceptions and interrupts; determines what to take, considering the current privilege mode (`PrivilegeModeW`); and responds with `TrapM`, `DelegateM`, and `CauseM`.

The `privmode` module changes the privilege in response to traps and `mret/sret` instructions.

Recall that Wally is configurable, and all features beyond the basic instruction set can be turned on or off at compile time using configuration options. Privileged support is enabled by setting the `ZICSR_SUPPORTED` flag to 1 in `config.vh`. When `ZICSR_SUPPORTED = 0`, all the privileged signals and architectural state are tied to 0 and optimized away by logic synthesis.

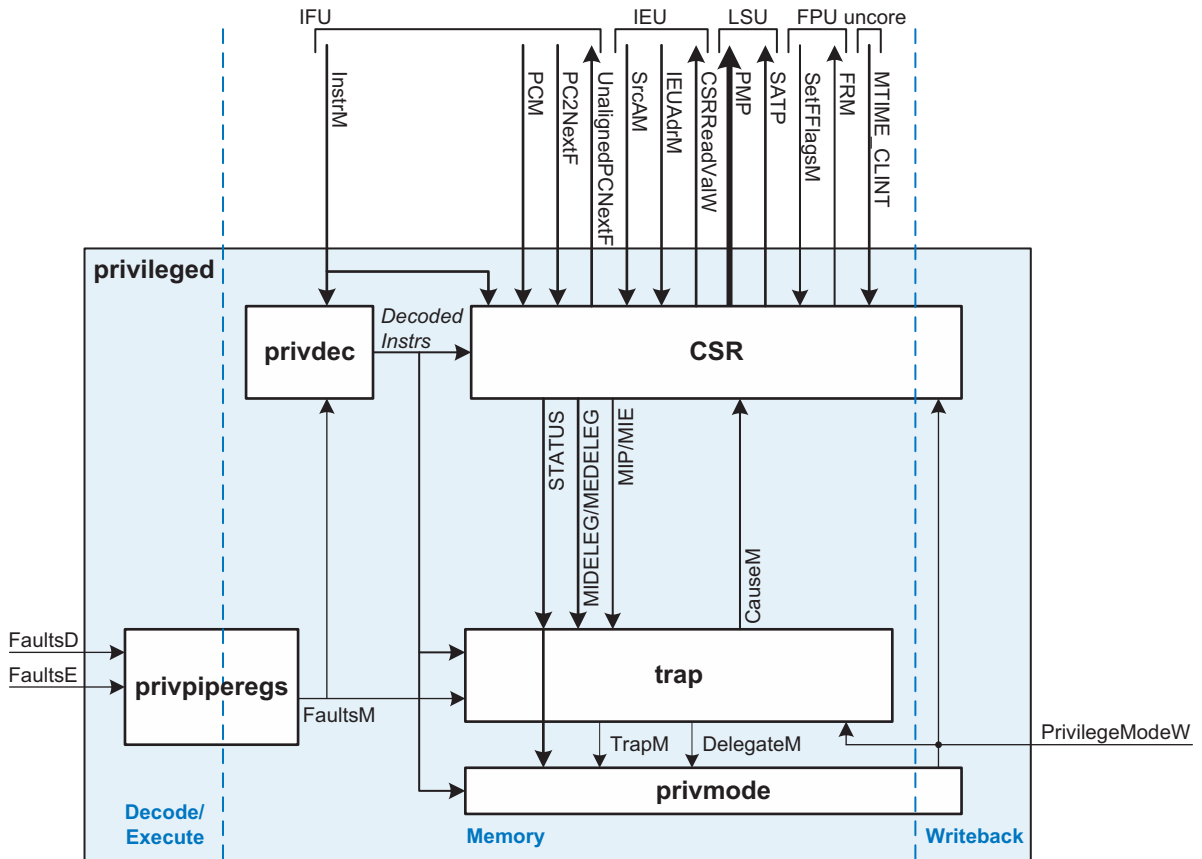


Fig. 8.13 Privileged unit block diagram.

Various features within the privileged unit are further enabled with flags, including:

- ▶ `S_SUPPORTED` Supervisor mode
- ▶ `ZICOUNTERS_SUPPORTED` Performance counters
- ▶ `VECTORED_INTERRUPTS_SUPPORTED` Vectored interrupts
- ▶ `BIGENDIAN_SUPPORTED` Big-endian loads and stores
- ▶ `WFI_TIMEOUT_BIT` WFI times out after $2^{\text{WFI_TIMEOUT_BIT}}$ cycles

The `Sstvecd` extension indicates that an implementation also supports direct (unvectored) interrupts with the bottom two bits of the trap vector being 00. RVA20S64 and later profiles expect `Sstvecd`, and Wally always supports it, independent of `VECTORED_INTERRUPTS_SUPPORTED`.

Each CSR submodule contains a register for each CSR, address decoders, write enables, and read multiplexers. If none of the submodules detect a legal access, the `csr` module causes an illegal-instruction exception.

8.4.3.1 Performance Counters

When `ZICSR_SUPPORTED` and `ZICNTR_SUPPORTED` are both 1, Wally provides 64-bit performance counters to count events. The number of counters is specified by the `COUNTERS` configuration variable, to a maximum of 32. The active counters are defined in [Table 8.10](#); any others are hardwired to 0.

`testbench.sv` prints the performance counters at the end of a test if the `PrintHPMCounters` parameter is 1. Benchmarks in [Chapter 21](#) parse the log file and extract performance numbers.

8.4.3.2 Machine Information Registers

Wally is assigned the OpenHW Foundation `mvendorid` of 0x602, the RISC-V International `marchid` of 36, and an `mimpid` of 0x100 (for Wally version 1.0.0). The single hart has `mhartid` = 0 and `mconfigptr` = 0.

8.4.3.3 `misa` and `XLEN`

Wally treats `misa` as a read-only register, so features cannot be disabled on the fly. In particular, `misa.XLEN` is hardwired, so an RV64 machine cannot run RV32I code. Similarly, `mstatus.SXL` and `mstatus.MXL` are hardwired to `misa.XLEN`.

The `Ssu64xl` extension indicates that 64-bit U-mode is available. Wally inherently supports this extension on any RV64 configuration.

8.4.4 Traps

The `trap` module manages pending exceptions and interrupts, global and local interrupt enables, and delegation registers. Key outputs include:

- ▶ `TrapM` Trap should occur
- ▶ `CauseM` Cause of highest-priority trap
- ▶ `DelegateM` Trap should be delegated to S-mode

[Fig. 8.14](#) shows a schematic of the trap unit. `mie`, `mip`, and `mideleg` are 12-bit inputs from the CSRs. Based on the privilege mode and global interrupt enable signals, the unit determines whether M-mode and S-mode interrupts are enabled. Considering whether a particular interrupt is delegated to S-mode, the trap unit determines which interrupt types are enabled. Precise interrupts are always associated with a specific PC so that the handler can return properly. Therefore, when the instruction in the Memory stage is not valid, interrupts are disabled until the next valid instruction arrives (`InstrValidM` = 1). Meanwhile, the trap unit determines which interrupts are pending (`PendingIntsM`) based on the `mie` and `mip` registers. The interrupts that are both pending and enabled are valid

Table 8.10 Performance counters.

CSR	Count	Enable	Section with Description
B00	Cycles elapsed	<always>	—
B01	Unused		—
B02	Instructions retired	InstrValidM	8.2.3.4
B03	Total branches	BranchM	13.3.3.3
B04	Total jumps	JumpM & ~ReturnM	13.3.3.3
B05	Total returns	ReturnM	13.3.3.3
B06	Branch predictor fetched wrong instruction	BPWrongM	13.1.4, 13.3.3
B07	Branch prediction wrong direction	BPPredWrongM	13.1.2, 13.3.3.1
B08	Branch/jump prediction wrong target address	BTAWrongM	13.3.3.2
B09	Return address stack wrong address	RASPredPCWrongM	13.3.3.4
B0A	Instruction class predictor wrong	IClassWrongM	13.3.3.3
B0B	Load stalls	LoadStallM	7.2.2.3
B0C	<Deprecated>		
B0D	Data cache accesses	DCacheAccess	10.4.3
B0E	Data cache misses	DCacheMiss	10.4.3
B0F	Data cache miss cycles	DCacheStallM	10.4.1
B10	Instruction cache accesses	ICacheAccess	10.4.3
B11	Instruction cache misses	ICacheMiss	10.4.3
B12	Instruction cache miss cycles	ICacheStallF	13.3.5
B13	CSR writes executed	CSRWriteM	8.4.5.4
B14	fence.I executed	InvalidateICacheM	13.3.1.1
B15	sfence.VMA executed	sfencevmaM	11.4.4
B16	Number of interrupts taken	InterruptM	8.4.4
B17	Number of exceptions taken	ExceptionM	8.4.4
B18	Division stall cycles (Integer & FP)	DivBusyE FDivBusyE	15.4.2 & 16.6.4

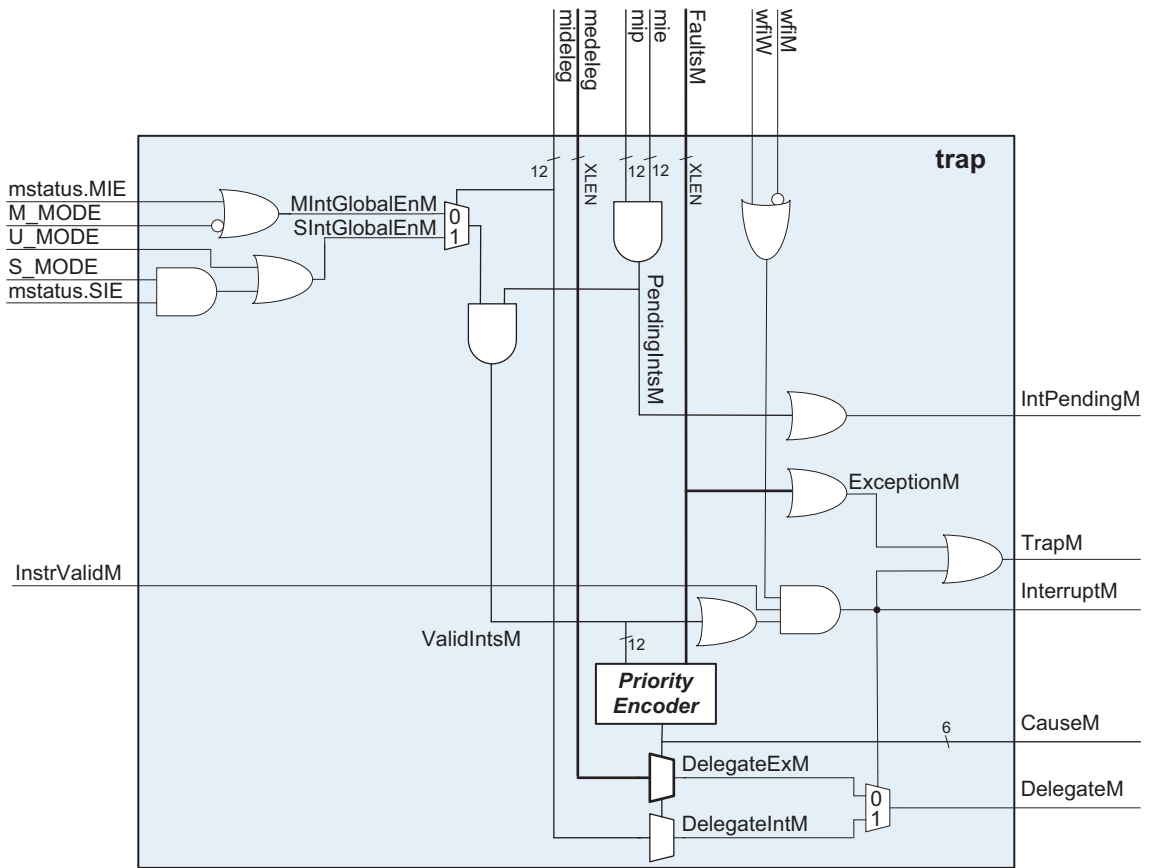


Fig. 8.14 Trap unit.

(`ValidIntsM`). A priority encoder considers all the valid traps to determine a `CauseM` code for the highest priority trap. If any trap occurs, `TrapM` is asserted. Depending on the cause and type of the trap, the unit examines the corresponding bit of the `mideleg` or `medeleg` register to assert `DelegateM` if the trap should be delegated to S-mode. `IntPendingM` is also raised when any interrupt is pending (to end a `wfi`), even if it is not enabled.

Exceptions are detected according to Table 8.11.

8.4.5 Pipeline Hazards

The privileged unit flushes the pipeline on traps, CSR writes, `xret` (`mret`, `sret`), and fences to ensure subsequent instructions see the correct state. It also stalls the pipeline during CSR reads and `wfi`. Fig. 8.15 shows the hazard unit. Compared to the baseline unit in Fig. 7.19, new signals are highlighted in blue. Each new flush and stall cause is discussed in the following subsections.

Fence instructions, described in Sections 8.4.5.5 and 17.2.2, ensure an ordering of memory accesses.

The hazard unit will continue to be expanded in future chapters to support new features.

Table 8.11 Exception sources.

Exception	Source	Section with Description
Instruction page fault	mmu	11.4.3
Instruction access fault	mmu	11.4.3
Illegal instruction	privdec	8.4.2
Instruction address misaligned	ifu	8.2.5.1
Breakpoint	privdec	8.4.2
ecall	privdec	8.4.2
Load address misaligned	mmu	11.4.3
Store/AMO address misaligned	mmu	11.4.3
Load page fault	mmu	11.4.3
Store/AMO page fault	mmu	11.4.3
Load access fault	mmu	11.4.3
Store/AMO access fault	mmu	11.4.3

8.4.5.1 Precise Traps

Recall that RISC-V traps are precise, so the instructions before the trap complete and the instructions after the trap are flushed and do not affect architectural state. The simplest way to ensure consistency in the architectural state is to update all state at the same time. The register file is written in the Writeback stage, so Wally updates all other state at this time. In particular, instructions changing the privilege mode or CSRs write this state on the rising edge of the clock at the start of the Writeback stage.

Exceptions may occur as late as the Memory stage; load/store faults are not known until this time. Again, it is simplest to have all traps take place at the same place in the pipeline and as late as possible before architectural state is updated, meaning traps occur in the Memory stage and change state at the rising edge of the clock at the start of the Writeback stage. When a trap occurs, the instruction in the Memory stage and all subsequent instructions in earlier stages are flushed. In Fig. 8.15, `TrapM` causes the full pipeline flush. On the rising edge of the clock at the start of the Writeback stage, CSRs are updated (including `epc`, `tval`, `cause`, and `status`), and the privilege mode and PC are changed to take the trap.

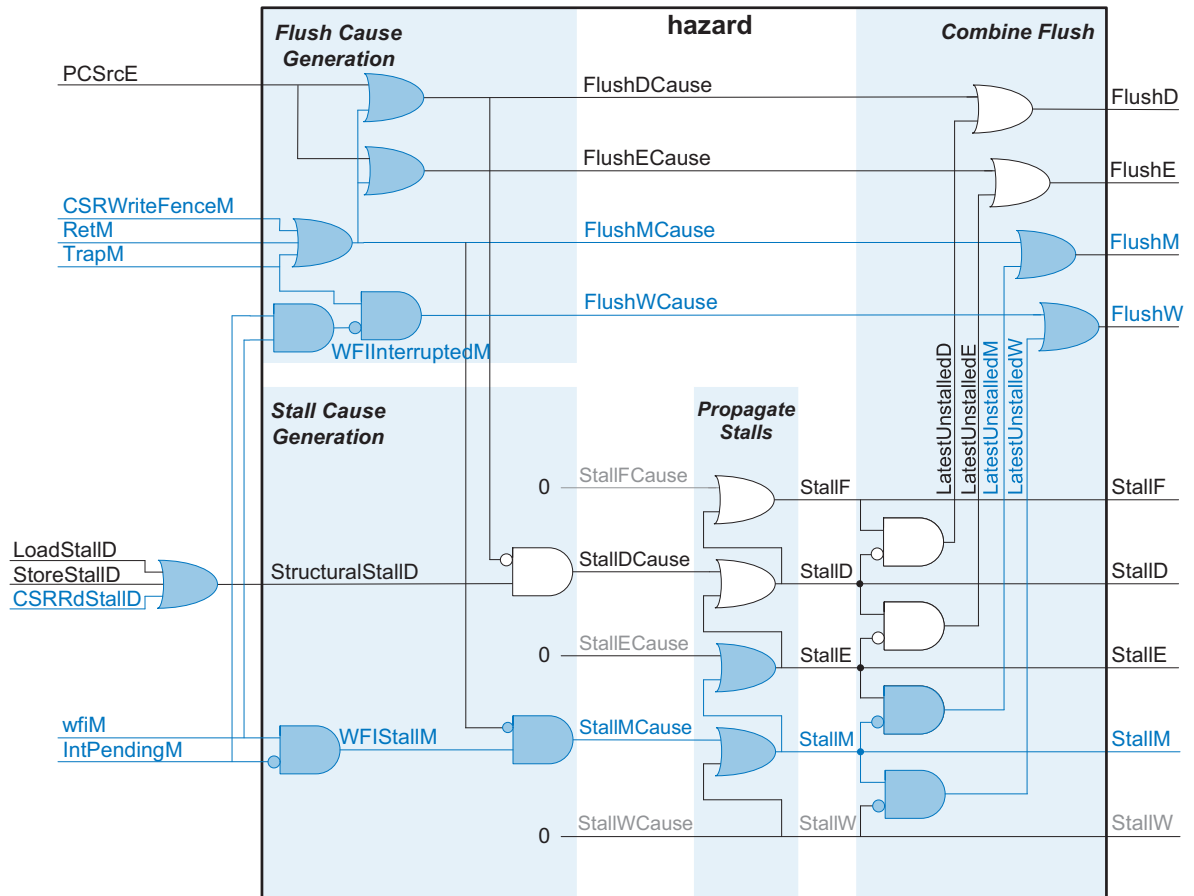


Fig. 8.15 Hazard unit, with privileged changes highlighted in blue.

8.4.5.2 mret and sret

Like traps, `mret` and `sret` change the PC and privilege mode on the rising edge of the clock at the start of the Writeback stage. Hence, they must flush all subsequent instructions in the pipeline. In Fig. 8.15, `RetM` causes a flush of all but the Writeback stage.

8.4.5.3 CSR Reads

CSR reads occur in the Memory stage so they obtain the latest CSR values, including any changes that may have been triggered by the previous instruction, such as CSR writes or floating-point operations that change flags. Therefore, like load and store structural hazards, CSR reads have a 2-cycle latency from issue to use, so the pipeline must stall in the Decode stage when the next instruction source register depends on the CSR read

destination register. In Fig. 8.15, `CSRReadStallD` stalls the Decode stage for a CSR read hazard.

8.4.5.4 CSR Writes

CSR writes also change architectural state and thus must occur on the rising edge of the clock at the start of the Writeback stage. Writes to CSRs change the way a program behaves, such as turning on virtual memory translation or changing the rounding mode of floating-point instructions. The simplest way to ensure subsequent instructions reflect the new behavior is to flush and reissue the instructions after a CSR write.

The controller asserts `CSRWriteM` while a CSR write is in the Memory stage and in turn raises `CSRWriteFenceM`. In response, the hazard unit flushes the Decode, Execute, and Memory stages, as shown in Fig. 8.15.

8.4.5.5 Fences

Fence instructions are synchronization events that ensure memory consistency. `fence.i` and `sfence.vma` ensure consistency of the instruction memory and virtual memory translations, respectively.

In systems with no cache, all writes go to memory, which is inherently self-consistent. In systems with instruction and data caches, a store may go to the data cache but not immediately to main memory, so data may be inconsistent between the cache and main memory. An instruction read from that address will obtain a stale value. `fence.i` makes sure that consistent instruction values are read. The simplest way to implement `fence.i` is to flush the data cache (so all stores are written to main memory) and invalidate the instruction cache, as discussed in Sections 10.4.2.6 and 10.4.5.4.

Similarly, after a write to a page table, the translation lookaside buffer (TLB) may contain stale entries. The simplest way to implement `sfence.vma` is to invalidate the entire TLB as discussed in Section 11.2.4.

Like CSR writes, fences assert `CSRWriteFenceM` to flush and refetch all instructions following the fence.

8.4.5.6 wfi

The wait for interrupt module (`wfi`) waits until an interrupt is pending or until it times out and causes an illegal-instruction exception. `WFIStallM` in the hazard unit (Fig. 8.15) is 1 while the `wfi` is active and no interrupts are pending. In Fig. 8.15, `WFIStallM` drives `StallM` to keep `wfi` in the Memory stage until an interrupt is pending or a timeout raises `TrapM`. When an interrupt becomes pending, `WFIStallM` deasserts and `WFIInterruptedM` asserts, to keep the `wfi` instruction from flushing as it advances to the Writeback stage and commits. `InterruptM` and `TrapM` are asserted, flushing the instructions after `wfi` and jumping to the trap handler. `WFIStallM` would preferably gate clocks to the unused parts of the core, but Wally does not yet implement clock gating.

`fence.i` is defined in the `Zifencei` extension (configured when `ZIFENCEI_SUPPORTED = 1` in `config.vh`). `fence.i` (or `cbo.flush` in Zicbom, see Section 19.1.1.1) is required for coherence in a system with a cache. However, it can be treated as a `nop` in cacheless systems.

`sfence.vma` is part of the privileged specification and is supported when virtual memory is enabled (configured when `VIRTMEM_SUPPORTED = 1` in `config.vh`).

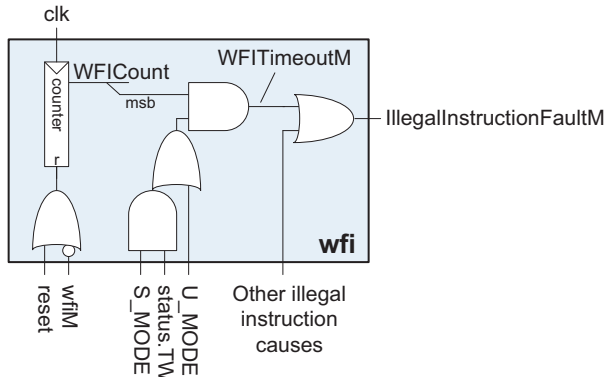


Fig. 8.16 `wfi` timeout logic.

Fig. 8.16 shows the `wfi` timeout logic in the `privdec` module for a system supporting S-mode. When `wfi` reaches the Memory stage, the hazard unit asserts `StallM` to stall the pipeline until an interrupt is pending or a timeout occurs. The `WFI` counter begins counting from zero. When the most significant bit is set and the system is in either U- or S-mode with `status.TW` set, it asserts `WFI` timeout, which causes an illegal-instruction exception. As a result, the processor flushes `wfi` from the pipeline and enters the trap handler.

The `WFI_TIMEOUT_BIT` configuration parameter determines the number of cycles to wait before timing out. It has a default value of 16, meaning `wfi` times out after 2^{16} cycles.

In a system that does not support S-mode, the timeout only occurs in U-mode when `status.TW` is set.

8.4.6 Instruction Fetch Unit (IFU) and Privileged Changes

Fig. 8.17 enhances the PC logic to support privileged operations: traps, `mret/sret`, and refetching instructions after a pipeline flush. The privileged changes in the `privileged` module are highlighted in blue. The multiplexer in gray is instantiated only when vectored interrupts are supported.

In the original Wally pipelined processor (Fig. 7.11), the `PCSrcE` mux selected `PCNextF` from either `PCPlus4F` or `IEUAdrE` (the branch or jump target address). In Fig. 8.17, the output of the `PCSrcE` mux is now called `PC1NextF`. If fences or CSRs are supported, the `CSRWriteFenceM` signal flushes the pipeline as described in Sections 8.4.5.4 and 8.4.5.6 and reloads the program counter with the instruction after the CSR write or fence. The flush is triggered in the Memory stage, so the next instruction is in the Execute stage and its address is in `PCE`. Thus, the `PCWriteFenceM` multiplexer chooses `PC2NextF` from either `PC1NextF` or `PCE`.

The next multiplexer handles traps and trap returns. During a trap, the PC is loaded from the `TrapVectorM`, which is either `mtvec`

Solution: The memory contents for each mode are shown in Fig. 8.18. In little-endian, the least significant byte of $x1$ is stored in the lowest address. In big-endian, the least significant byte is stored in the highest address. The contents of the registers are shown in Table 8.12.

Little-Endian Memory Contents	Byte Address	Big-Endian Memory Contents
FE	1007	10
DC	1006	32
BA	1005	54
98	1004	76
76	1003	98
54	1002	BA
32	1001	DC
10	1000	FE

Fig. 8.18 Memory contents for endian example.

Table 8.12 Register contents for endian example.

Load	Register	Little Endian	Big Endian
ld x2, 1000(x0) x2		FEDCBA98_76543210	FEDCBA98_76543210
lwu x3, 1000(x0) x3		00000000_76543210	00000000_FEDCBA98
lwu x4, 1004(x0) x4		00000000_FEDCBA98	00000000_76543210
lbu x5, 1001(x0) x5		00000000_00000032	00000000_000000DC

Observe that a full-length write in either endianness followed by a read in the same endianness returns the original value. However, subword loads return different portions of the full word depending on the endianness. A program can use this behavior to deduce the endianness of the machine.

Fig. 8.19 shows the RV64 LSU with the optional `endianswap` modules in gray. It is similar to the LSU from Fig. 7.9 except that it swaps endianness before writing to memory and after reading from memory. Big-endian subword reads must also swap the order of the subword by inverting the address. The `BigEndianM` signal is asserted when the `mstatus.MBE`, `SBE`, or `UBE` bit is set corresponding to the current privilege mode. The `endianswap` module exchanges byte 0 with byte 7, 1 with 6, and so forth. RV32 is the same except that `endianswap` only exchanges byte 0 with byte 3 and byte 1 with byte 2 in a word. The endian logic adds a multiplexer delay to the Memory stage read and write paths, which might impact cycle time.

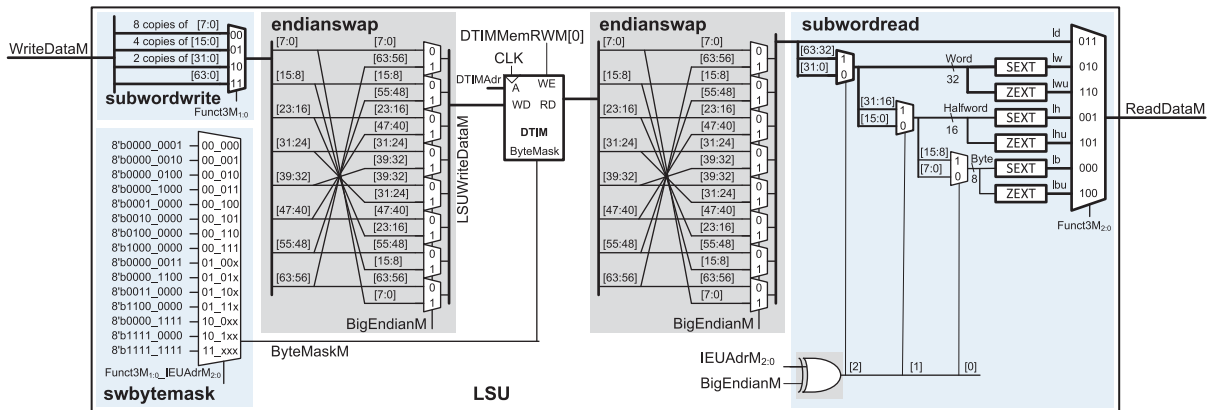


Fig. 8.19 RV64 LSU with endianswap modules.

8.5 SUMMARY

A robust microprocessor needs to protect itself from crashes or compromise caused by buggy or malicious programs. Bad instructions cause exceptions, such as illegal-instruction, access-fault, address-misaligned, or page-fault exceptions. System calls (`ecall`) and debugger breakpoints (`ebreak`) are also exceptions. The microprocessor also receives interrupts from asynchronous events, such as timers, interprocessor communication, or external peripherals. Exceptions and interrupts are collectively called *traps*. Traps leave the current program and jump to a trap handler. Processors add multiple privilege modes for robustness and security. Programs run in user mode, in which mode they cannot interfere with each other. The trap handler is part of the firmware that runs in machine mode, in which it has full power to allocate machine resources and kill faulty programs. Application processors add an intermediate supervisor mode, to run an OS such as Linux without giving the entire OS kernel full M-mode powers. Each privilege mode is associated with a set of CSRs to configure traps, enable features, or monitor performance counters.

The `mret` and `sret` instructions return from M- and S-mode trap handlers. `csrrw`, `csrrs`, and `csrrc` instructions read and write, set, or clear CSRs, respectively. The `wfi` instruction can be used to save power by placing the processor in a low-power state until an interrupt occurs.

In a system without S-mode, traps always elevate the privilege to M-mode. In a system with S-mode, traps default to M-mode, but traps can be delegated to the S-mode OS via the `medeleg` and `mideleg` registers. A set of CSRs communicates information about the trap. Let $x = m$ (machine) or s (supervisor), the privilege mode of the trap handler. When a trap occurs, the core enters x mode. It saves the old program counter in `xepc`, disables interrupts, places a code in `xcause` that indicates the reason for the trap,

places additional information such as the bad address or instruction in `xtval`, and jumps to a trap handler at `xtvec`. The trap handler typically swaps the stack pointer (`sp`) with an exception stack pointer in the `xscratch` register, then saves the integer registers onto the exception stack so that it can use the registers without corrupting the program that was running. It then reads `xcause` to determine the cause of the trap and respond accordingly. For example, it may perform a system call, respond to the timer or external interrupt, or kill a program that made a bad memory access. The trap handler then restores the registers and stack pointer. Finally, it performs the `xret` instruction, which reenables interrupts, restores the old privilege level, and returns to the old program counter stored in `xepc`.

CSRs with names beginning with `m` can only be accessed from M-mode, so lower privilege programs cannot seize control of the processor. The `mstatus` register is particularly important; it contains bitfields to globally enable interrupts, specify the privilege mode to which the trap handler should return, configure endianness, and so forth. Other machine CSRs contain codes defining the vendor, architecture, implementation, hart IDs, and information about the supported extensions. `mcycle` and `minstret` are standard performance monitor counters that track the number of cycles and instructions that have elapsed. The platform-dependent `mhpmonitor` hardware performance monitor counters keep track of the number of times other events occur, such as cache misses or branch mispredicts.

Similarly, CSRs starting with `s` can only be accessed from S-mode or above. The `satp` supervisor address translation and protection CSR controls virtual memory, which is described in depth in [Chapter 11](#).

Wally adds a privileged unit to track the privilege level, store the CSRs, and handle traps and privileged instructions. Traps, `xret` instructions, CSR writes, and fences flush the pipeline to discard partially completed instructions and ensure subsequent instructions see the updated privilege and CSRs.

If big-endian mode is supported, the LSU adds endianswap muxes to reverse the bytes written and read from the data memory. Instructions are always stored in little-endian order.

Wally defines an extensive set of wally-riscv-arch-test programs to test privilege modes, traps, and CSRs.

Bus Interface

9

A processor’s capabilities are extended by interfacing it with external memory and peripherals such as a keyboard, mouse, or display. Digital systems use *bus interfaces* to connect components with standardized communication protocols. For example, processor cores use one or more standard SoC buses to communicate with memory and peripherals. Using a standard SoC bus makes it practical to attach off-the-shelf peripherals designed for that bus.

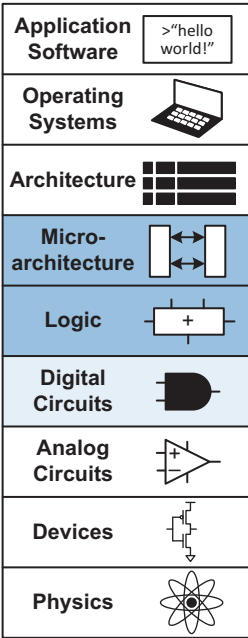
This chapter begins by exploring the principles driving SoC bus design. It then surveys a variety of standard SoC buses, including AHB (Advanced High-performance Bus) and APB (Advanced Peripheral Bus), which are used in Wally, as well as AXI (Advanced eXtensible Interface), TileLink, and others.

The chapter then extends the Wally processor by adding an external bus unit (EBU) that controls an AHB bus. Whereas the Wally processor from [Chapter 7](#) used idealized instruction and data memories (IROM [instruction read-only memory] and DTIM [data tightly integrated memory]) located inside the core, the EBU added in this chapter provides access to RAM (random access memory) and ROM (read-only memory) located outside the processor in the uncore portion of the SoC, as highlighted in bright blue at the top of [Fig. 9.1](#).

9.1 PRINCIPLES

A bus connects one or more *controllers* to one or more *peripherals* or *responders*. The peripheral might be a memory, such as SRAM (static RAM), DRAM (dynamic RAM), or ROM, or an I/O (input/output) device such as a timer, interrupt controller, general-purpose input/output (GPIO) device, serial port, or data converter. In most bus protocols, the controller sends an *address* to specify which peripheral and which memory location or register

- 9.1 Principles
- 9.2 AMBA Buses
- 9.3 Other SoC Buses
- 9.4 Test Plan
- 9.5 Wally Implementation
- 9.6 Summary



[Chapter 20](#) describes off-chip peripheral interfaces using SPI and UARTs. High bandwidth off-chip interfaces such as DDR, PCIe, USB, and Ethernet are beyond the scope of this book.

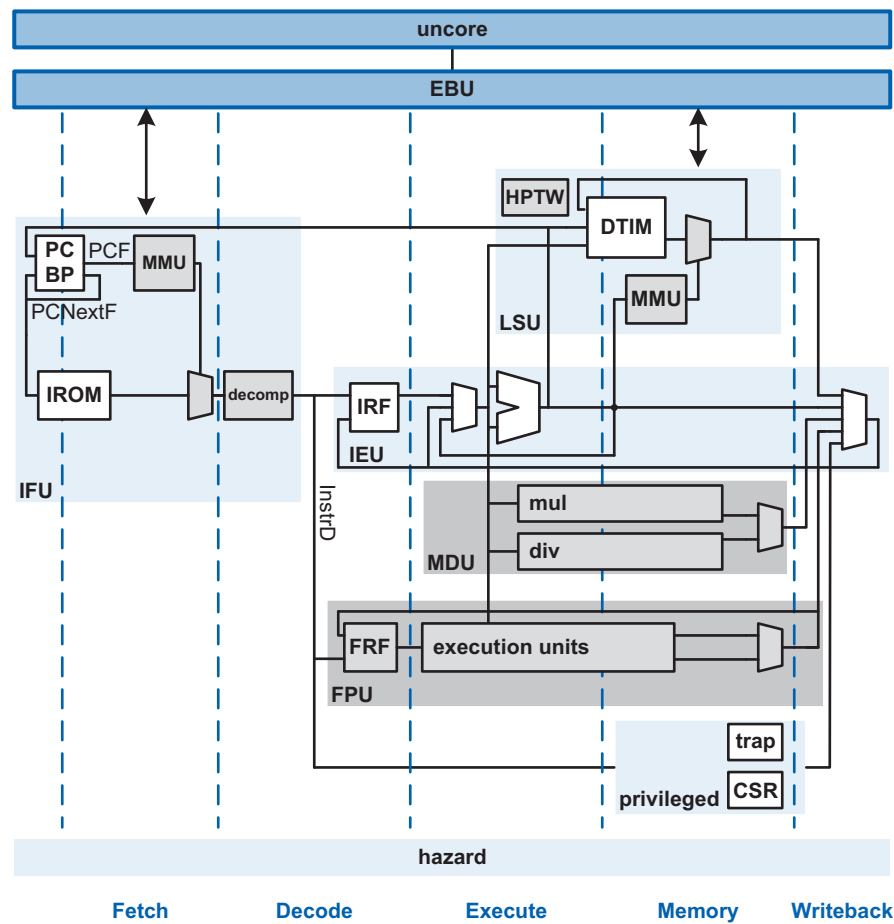


Fig. 9.1 RV{32/64} high-level block diagram highlighting the EBU and uncore.

The *uncore* part of an SoC is, as expected, everything except the processor core. It includes the bus interfaces, memory, and peripherals.

within that peripheral to access. The bus also transmits *data* that the controller reads from or writes to the peripheral.

Fig. 9.2 builds up a sequence of progressively more practical buses, showing the interface signals and timing. The simplest example of a bus is an *asynchronous read-only bus*, like the instruction memory interface from Fig. 7.2. As shown in Fig. 9.2(a), the controller sends an *address* (A), and a peripheral responds with *read data* (RD) within a bounded amount of time. However, most memories and peripherals are *synchronous* for ease of implementation: they need the address to set up before a clock edge, and they return the read data after that clock edge, as shown in Fig. 9.2(b). Peripherals other than ROMs and sensors need a read/write interface. They need to know what to write (*write data*, WD), where to read/write (*address*, A), whether to write (*write enable*, WE), and when to read/

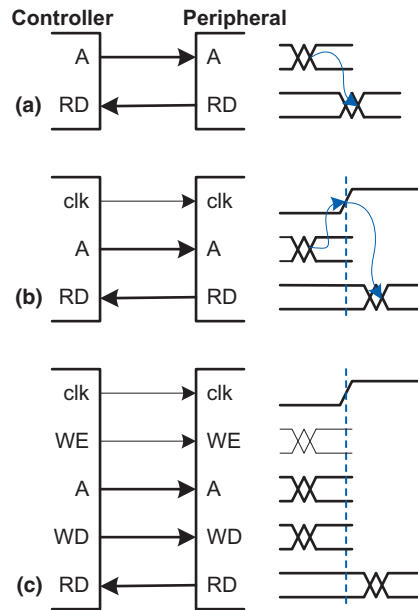


Fig. 9.2 Basic bus interfaces: (a) asynchronous read-only, (b) synchronous read-only, and (c) synchronous read/write

write (clk). Hence, in the *synchronous read/write* bus shown in Fig. 9.2(c), the controller sends the address (A), write data (WD), and write enable (WE) before the clock edge. The peripheral then returns read data (RD) after the clock edge. When WE is asserted, the controller also writes to the peripheral just after the clock edge.

9.1.1 Bidirectional vs. Unidirectional Buses

The bus in Fig. 9.2(c) has separate unidirectional RD and WD (read and write data) ports. Pins on a chip and wires on a circuit board are relatively expensive, so designers were historically motivated to combine these two ports into a single bidirectional port, as shown in Fig. 9.3. A tristate buffer, with alternating enable input polarity, is added on both the controller (left) and peripheral (right) sides. When WE is 1, the controller writes the data from signal D to the peripheral; otherwise, the controller reads data from the peripheral on signal D.

In an SoC, far more wires are available. Moreover, long on-chip wires need repeaters (buffers) to achieve reasonable speed, making bidirectional operation impractical. Hence, SoC buses universally favor unidirectional

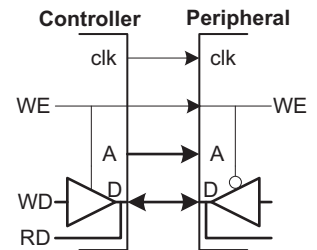


Fig. 9.3 Synchronous read/write bus with a bidirectional data port.

Busess are also referred to as links.

Busess formerly used the nomenclature *master* and *slave*, and you will still see these terms in standards documents. These terms carry dehumanizing and derogatory connotations, so the industry is moving toward deprecating this language [Ellis21, OSHA22]. There is not yet agreement on new terminology. In 2021, ARM began using the terms *manager* and *subordinate* or *requester* and *completer*. Other applications, such as SCSI buses, use *initiator* and *target* for these two roles. In the absence of a new standard, this book uses the terms *controller* and *peripheral* (or *device*).

RISC-V generalizes the address decoder as a *physical memory attributes (PMA)* unit that also determines whether the address is cacheable, what transfer widths are allowed, and so on, as described in Section 11.2.2.1.

buses. Tristate buffers driving on-chip bidirectional buses are typically forbidden because of their poor performance and tool support.

Unidirectional point-to-point off-chip buses also can run at a far higher speed than bidirectional multidrop buses, which have multiple peripherals that connect to the bus at different points along the bus wires. It is now cheaper to achieve a required bandwidth by running a moderate number of wires that transmit data serially at high speed in each of several unidirectional links than to run a larger number of wires at lower speed in a single bidirectional link. Section 9.1.6 discusses off-chip buses further.

9.1.2 Handshaking

A controller only needs to access the bus some of the time. To save power, it can provide a *request* (REQ) line that is only asserted when it puts a valid address on the bus.

Some peripherals take more than one bus cycle to respond. For example, DRAM is fast for sequential reads but slow for reads to random addresses. A bus often provides an *acknowledge* (ACK) or *ready* (RDY) signal back from the peripheral indicating that the transfer is complete. This request/acknowledge sequence is known as *handshaking*. Fig. 9.4 shows the timing for a peripheral with a 2-cycle read access time. Observe how RDY remains low for a cycle before rising on the second cycle when the read data (RD) is available.

9.1.3 Multiple Peripherals

In a practical system, the controller communicates with multiple peripherals. SoCs typically use *memory-mapped I/O* to select a particular peripheral based on the address range. The SoC contains an *address decoder* to produce select signals to choose which peripheral or memory to read based on the address. The SoC *memory map* defines the addresses of each peripheral.

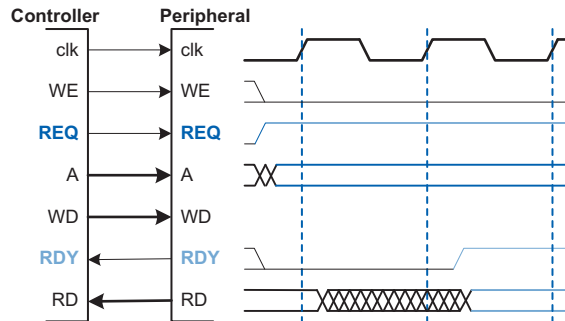


Fig. 9.4 Handshake timing for multicycle access.

Fig. 9.5 shows buses that link multiple peripherals with *shared* and *point-to-point* data ports. Both versions require an address decoder. The shared bus allows multiple peripherals to drive the RD and RDY lines using tristates, as was shown in Fig. 9.3. In contrast, the point-to-point design requires a read multiplexer to choose the read data (RD) and ready (RDY) signals from the selected peripheral. Shared buses minimize the number of wires, but point-to-point buses are much faster and eliminate potential contention among peripherals. Shared buses have many problems in SoCs, including accurately modeling delay, verifying contention, and lacking repeaters to drive long wires. Therefore, SoC on-chip buses use point-to-point wiring.

9.1.4 Multiple Controllers

Larger systems also have multiple controllers that may share the same bus. One or more controllers may place a request on any given cycle by

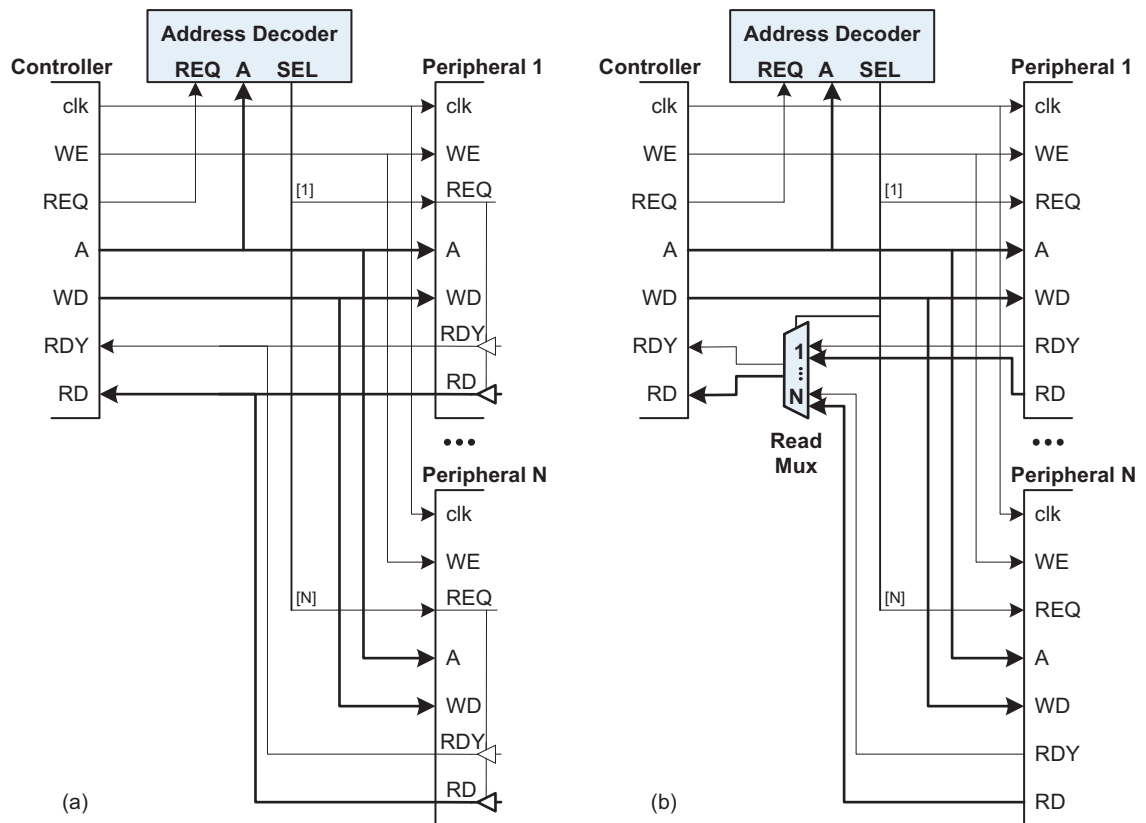


Fig. 9.5 Bus with multiple peripherals: (a) shared data ports and (b) point-to-point data ports with read multiplexer.

asserting their REQ. An *arbiter* picks among the requests by granting access to at most one controller and chooses the address (A), write data (WD), and write enable (WE) from that controller, as shown by the Grant Mux in Fig. 9.6. This synchronous bus uses a common clock for all components, which is not shown.

Arbiters can use various methods to pick which request to grant, including static, round-robin, time-division multiple access, and dynamic priority algorithms. In *static priority*, each controller has a fixed priority, and the requesting controller with the highest priority is granted access. This could starve a low-priority controller if a high-priority controller is making continuous requests. In *round robin*, access is granted in a circular manner, so every controller will eventually get access. This is fair but could cause long delays on a critical request. In *time division multiple access (TDMA)*, controllers are assigned time slots of varying length depending on their bandwidth requirements. In *dynamic priority*, the priority is adjusted in real time based on the system needs.

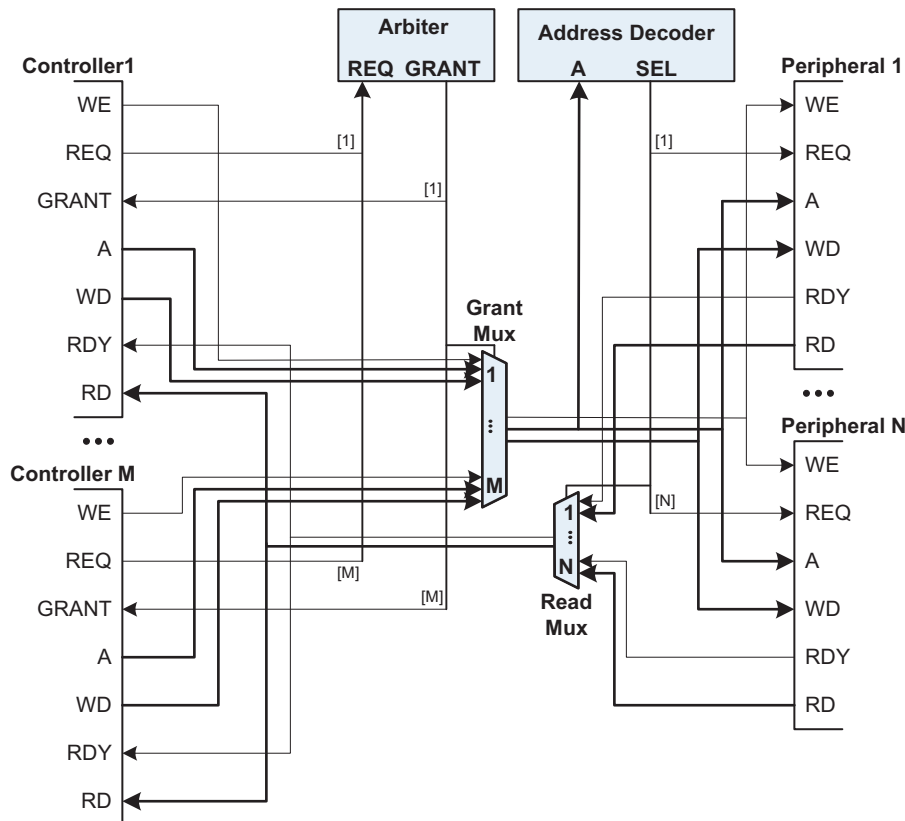


Fig. 9.6 Bus with multiple controllers and arbitration.

Systems with multiple controllers may need to *synchronize* between controllers so that only one can access a resource at a time. Synchronization is performed with *atomic* operations described in [Chapter 17](#). These operations often read, modify, and then write a location in memory in such a way that no other controller can interrupt the operation. Thus, buses often have a *locking* signal so that a controller that is granted access can complete the atomic operation before another controller gains access.

In a multicontroller system, a device may act as both a peripheral and controller. When the device is a controller, it may perform *direct memory access* (DMA), reading or writing memory without going through the processor. DMA introduces challenges maintaining *coherence* between memory and caches such that any store to an address will eventually be seen by subsequent loads from the same address. For example, if a device writes directly to an address in DRAM but a cache already holds an older value of the contents of that address, reads to the cache will not see the new value written by the device, and the memory system is incoherent. [Section 19.1](#) describes the RISC-V cache management operation (CMO) extension, which includes additional RISC-V instructions to invalidate and clean the data cache to maintain coherence with a DMA controller. A more detailed discussion of DMA coherence is beyond the scope of this book.

9.1.5 Other Bus Features

Additional bus features include support for multiple transfer modes (e.g., subword writes, burst transfers, and split transfers), cache coherency, and bridges between buses. High-performance systems may abandon buses altogether in favor of higher-speed networks. This section gives an overview of these features.

Buses often support various transfer modes. Most buses can write data smaller than the full word size. They provide either a `Size` or `ByteEn` signal that indicates which bytes to write. To reduce the overhead of repeated arbitration, buses also often define *burst* transfers consisting of a sequence of reads or writes, often to sequential addresses. Finally, when a data transfer is slow, it may tie up a bus for many processor cycles. For example, when a DRAM read is made to a new row, the bus may take many tens of cycles to respond. *Split transfer* buses tag the transfer with a unique ID and then open the bus for other operations. Sometime later, the bus returns the data and tag to the controller that initiated the request.

Complex systems often use more than one type of bus interface. For example, a system may have a high-speed memory bus and a low-speed bus for other (including legacy) peripherals. A *bridge* interfaces two buses to translate the interaction from one bus to another and so that their protocols are each satisfied.

Some buses, such as ACE and TileLink Cached, incorporate *cache coherence* capabilities directly into the bus protocols to support *shared-memory multiprocessors* in maintaining *coherence* across their cache memories.

High-performance systems often need more bandwidth than a single bus can provide. For example, they may need to allow multiple processors to concurrently access multiple banks of level 3 cache memory. A *Network on Chip* (NoC) provides a flexible and high-performance interconnect. Types of NoC architectures include *crossbars*, where every component can communicate with any other component; *indirect networks*, in which components communicate through a series of switches; and *meshes* or *rings*, in which every component can communicate with its neighbors.

9.1.6 High-Speed Off-Chip Links

The bandwidth of off-chip links has grown along with processor performance. AT&T developed the first commercial modem in 1958 for the SAGE system that connected radar systems across the United States to detect intrusions by Soviet nuclear bombers. The modem ran over the phone lines at 110 baud (symbols per second), transmitting 80 bits per second (bps), which is 10 characters per second, between mainframe computers and teletype terminals.

Bandwidth has increased rapidly since those first links. In 2025, personal computers are equipped with USB-C ports running at 40 Gbps to external devices and HDMI 2.2 ports supporting 96 Gbps using inexpensive cables. On the motherboard, a dual-channel DDR5-6400 memory system transmits approximately 400 Gbps between DRAM and a CPU (central processing unit), and PCIe 5.0 x16 transmits up to 1 Tbps between a graphics card and CPU. In data centers, Ethernet runs at 800 Gbps between cabinets. Wireless networks have also become very fast, with Wi-Fi 7 and 5G cellular theoretically capable of more than 10 Gbps.

Although long-distance networks use fiber-optic cables driven by lasers, shorter links are primarily electrical over copper wires. As the data moves along the copper at nearly the speed of light, transmission line effects are critical, and the wires must be properly *terminated* with a *matched impedance*, making point-to-point links essential. The links use *differential signaling*, in which the data is represented as a difference in voltage between a pair of wires, rather than as a single-ended signal relative to a ground line. Differential signaling reduces *crosstalk*, *common-mode noise*, and *electromagnetic interference* because current is balanced, flowing out and back along the pair rather than returning over a shared ground line.

Some high-speed links such as DDR (double-data-rate) memory use *source-synchronous clocking*, in which a clock is sent along with a parallel bundle of data bits. The clock rises and falls each cycle, generally at a rate as fast as the transmission line can reliably support. If the memory bits only switched once per clock cycle, they would only use half of the available bandwidth of the line. Hence, *DDR links* send data on both rising and falling edges of the clock (thus, its name: double-data-rate) to use the same toggle rate as the clock and fully saturate the transmission line's capabilities.

Source-synchronous speeds are limited by mismatches in flight time between bits in the bundle. These mismatches are influenced by the line lengths, impedances, and noise. *High-speed serial links* avoid these mismatches by sending a serial bitstream over a differential pair encoded in such a way that the bitstream conveys both clock and data. The receiver performs *clock/data recovery* (CDR) to determine when to sample the serial stream, then decodes the data bits. For example, *8b/10b* encoding sends 8 bits of data in a 10-bit symbol such that the transmitted bitstream toggles at least once every 5 bits even when the unencoded bitstream is a continuous pattern of 0s or 1s. The redundant bits can also provide error detection and correction. A phase-locked loop (PLL) in the CDR detects these transitions to align the sampling clock. Similarly, 10-Gb Ethernet uses *64b/66b* encoding, PCIe 5 uses *128b/130b* encoding, and USB 3.1 uses *128b/132b* encoding to guarantee transitions while reducing the overhead. PCIe 5 x16 uses 16 serial *lanes* each operating at 32 GHz (64 Gbps).

PCIe (PCI Express), DDR memory, USB, and Ethernet each have specifications that are many hundreds of pages long, and they require a complex controller and *physical interface* (PHY) beyond the scope of this book. SoC designers purchase these controllers and PHYs as *IP* (*intellectual property*) blocks. The controllers may be integrated onto an ASIC (application-specific integrated circuit) or FPGA (field programmable gate array) and connected to the rest of the SoC via AXI or another standard on-chip bus. The SoC physical interface involves custom mixed-signal circuitry such as impedance-controlled differential output drivers and phase-locked loops. These are also sold by IP vendors.

[Dally98] provides a good introduction to high-speed communication. All these sophisticated techniques for fast off-chip links cost chip area and power. Within the SoC, it remains more cost-effective to use wide buses, instead of serial buses, to deliver high local bandwidth.

9.2 AMBA BUSES

ARM's Advanced Microcontroller Bus Architecture (AMBA) has been a leading family of open-standard on-chip buses used in both ASICs and FPGAs since 1996. The three major AMBA flavors are:

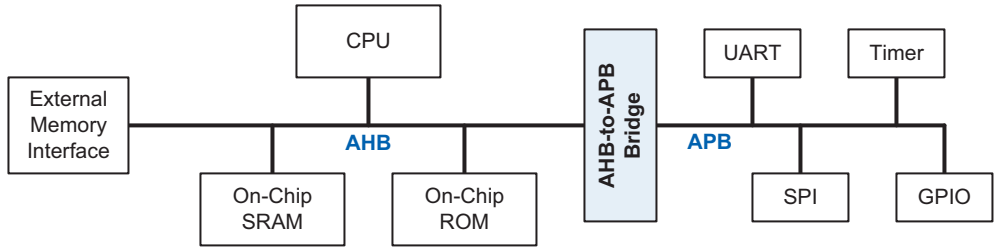


Fig. 9.7 AHB/APB system.

David Flynn was the original architect of ARM's synthesizable processors and the AMBA on-chip interconnect. He received the IEEE/RSE James Clerk Maxwell Medal in 2019 for these contributions as well as for his contributions to microprocessor technology and deployment. He also coauthored the *Low Power Methodology Manual* [Keating07], which defined industry-standard methods of clock and power gating. Flynn was an ARM fellow and worked at the company from 1991 to 2018. He earned his BSc in computer science from Hatfield Polytechnic and his doctorate in electrical engineering from Loughborough University.

- ▶ APB (Advanced Peripheral Bus): Simple unipipelined
- ▶ AHB (Advanced High-performance Bus): Pipelined
- ▶ AXI (Advanced eXtensible Bus): Multichannel, high-speed

Fig. 9.7 shows an example system that uses AHB and APB. Whereas the high-speed CPU and on-chip memories use AHB, the slower peripherals use the simpler APB bus. The AHB-to-APB bridge translates signals from one protocol to the other. AHB can also be used as an interface to off-chip memory.

The following subsections describe APB [APB21], AHB [AHB21], and AXI [AXI21]. Both APB and AHB use two phases for address and data transfer, but AHB is faster because the phases of consecutive operations can overlap. AXI is now the dominant standard for systems with multiple controllers or high-speed DRAM interfaces, but the 500-page specification [AXI21] is beyond the scope of this chapter.

9.2.1 APB

Fig. 9.8 shows an APB bus connecting a controller to one or more peripherals. The controller is an AHB-to-APB bridge. The standard APB signal names are similar to the generic ones from Fig. 9.5 but are prefixed by *P*, as listed in Table 9.1 for the main APB signals except *PCLK* and *PRESETn*. The *PRESETn* signal resets all peripherals when asserted low. The address decoder, which is part of the controller, generates the *PSEL[N:1]* signal for the various peripherals and the mux. The APB mux returns the data and ready signals from the selected peripheral. *PADDR* may be up to 32 bits, and *PRDATA/PWDATA* may be 8, 16, or 32 bits.

Each transfer has two phases: *Setup* and *Access*. The Setup phase takes one clock cycle, during which the controller drives the *PADDR*, *PWRITE*, and *PSEL* signals onto the bus along with *PWDATA* for a write. The Access phase takes one or more clock cycles, during which the controller holds these Setup phase signals steady and asserts *PENABLE*; the peripheral performs the read or write. Upon a read, the peripheral responds with *PRDATA*. The peripheral raises *PREADY* when it is done reading or writing. Typically,

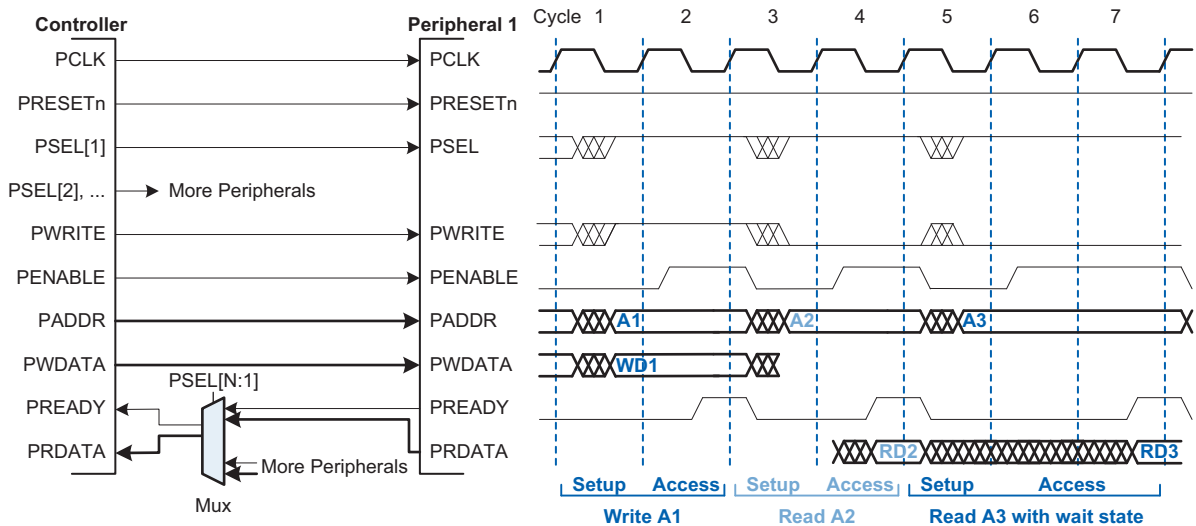


Fig. 9.8 APB interface.

the Access phase is one cycle long, but a slow peripheral could hold `PREADY` low for one or more *wait states* before responding. A peripheral that always responds in a single Access phase cycle can tie its `PREADY` high. If no peripheral is selected, the APB mux asserts `PREADY` so the bus remains in a ready state.

The waveforms in Fig. 9.8 illustrate read and write transfers. In cycles 1 and 2, the APB bus performs a write transfer with no wait states. In cycles

Table 9.1 Main APB signals.

Signal	Description	Width (bits)	Phase	Source
PADDR	Address to read or write	32	Setup	Controller
PRDATA	Read data from peripheral	32	Access	Peripheral
PWDATA	Write data to peripheral	32	Setup	Controller
PWRITE	Write enable	1	Setup	Controller
PENABLE	Enable read or write	1	Access	Controller
PREADY	Peripheral asserts when Access phase completes	1	Access	Peripheral
PSTRB	Write byte enable	4	Setup	Controller

3 and 4, the bus performs a read with no wait states. In cycles 5–7, the bus performs a read with one wait state.

Not shown in Fig. 9.8 are the optional `PSTRB` (APB strobe) and `PSLVERR` (APB peripheral error) signals. `PSTRB[3:0]` is a byte write enable, with one bit per byte. A peripheral may assert `PSLVERR` when an error occurs.

Example 9.1 GPIO USING APB

Design a 32-bit GPIO peripheral that uses an APB interface. The peripheral has three 32-bit memory-mapped I/O registers: `GPIOOUT`, `GPIOIN`, and `GPIOEN`. `GPIOOUT` is the output register (for writing to the GPIO pins); `GPIOIN` is the read-only input register (for reading the values on the GPIO pins); and `GPIOEN` configures the GPIO pins as outputs or inputs (1 = output, 0 = input), with each bit controlling a tristate enable connected to the GPIO pins. These registers are mapped to a 2-bit address: `GPIOOUT` is at address 00, `GPIOEN` at 01, and `GPIOIN` at 10.

Solution: Fig. 9.9 shows the 32-bit GPIO peripheral. The core of the peripheral is a tristate buffer that enables writing `GPIOOUT` onto the GPIO pins for each bit of `GPIOEN[31:0]` that is asserted. The pins can always be read onto `GPIOIN`. A 2:3 address decoder determines which register is being accessed. When `PWRITE`, `PSEL`, and `PENABLE` are asserted, the `GPIOOUT` or `GPIOEN` register is written when `PADDR[1:0]` is 00 or 01, respectively. The GPIO multiplexer is controlled by the address (`PADDR[1:0]`) to read the appropriate register. The peripheral has no wait states; thus, it always asserts `PREADY`.

ASIC and FPGA implementations use specially designed tristate buffers (also called *drivers*) capable of driving large capacitive PCB (printed circuit board) wire loads. They are instantiated in the top-level module, and therefore the `gpio_apb` module has three interface signals: `GPIOEN`, `GPIOOUT`, and `GPIOIN`.

Fig. 20.3 shows a full-featured GPIO peripheral with an APB interface.

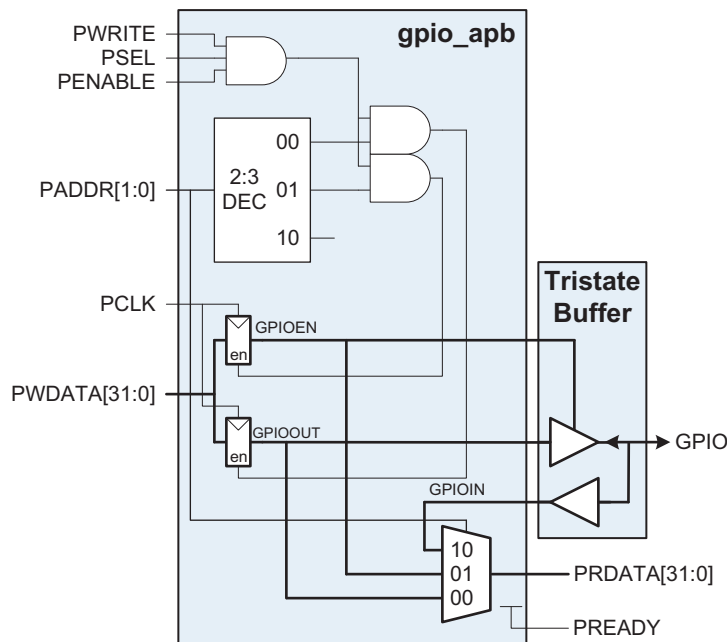


Fig. 9.9 GPIO peripheral with APB interface.

9.2.2 AHB

Like APB, AHB uses two phases of operation and takes at least two cycles for a transfer, with the option of additional wait states. However, AHB doubles the throughput by pipelining, that is, overlapping the two phases. AHB refers to the two phases as Address (Adr) and Data phases, which are similar to APB's Setup and Access phases, as listed in Table 9.2.

Fig. 9.10 shows an AHB bus connecting a controller to one or more peripherals. The address decoder generates $HSEL[N:1]$ for N peripherals and the AHB mux. $HADDR$ may be up to 64 bits, and $HRDATA/HWDATA$ are typically 32, 64, 128, or 256 bits, permitting wider transfers than APB. AHB adds an $HTRANS$ signal to indicate the type of transfer, as summarized in Table 9.3. Typically, $HTRANS$ is IDLE when the bus is not in use, or NONSEQ for ordinary transfers. Section 9.2.2.5 introduces SEQ burst transfers.

Fig. 9.10 illustrates a write to address A1; then a read from A2; then another read from A3, which adds a wait state. In the Address phase, the controller places the address $HADDR$ and control signals $HWRITE$ and $HTRANS$ on the bus, and the address decoder drives $HSEL$ to the appropriate peripheral. In the Data phase, the controller sends $HWDATA$, or the peripheral

ARM uses inconsistent terminology between APB and AHB.

AHB was originally developed to support multiple controllers and split transactions [AMBA99], but a simpler flavor called AHB-Lite omits this support. Systems requiring these additional features soon migrated to the more capable AXI bus, leaving AHB-Lite as the favored flavor of AHB. ARM recognized this in the recent AHB5 specification, which is based on AHB-Lite with optional extra features related to security and exclusive transfers. This book uses the term *AHB* to refer to AHB5 and focuses on the required features that are common to AHB-Lite and AHB5.

Table 9.2 Comparison of APB and AHB terminology.

Term	APB	AHB
Controller	Requester	Manager
Peripheral	Completer	Subordinate
Phase 1	Setup	Address (or Adr)
Phase 2	Access	Data
Signals	Use prefix P	Use prefix H

Table 9.3 HTRANS transfer types.

HTRANS	Type	Description
00	IDLE	No data transfer
01	BUSY	Like IDLE, but in the middle of a burst
10	NONSEQ	Single transfer or first transfer of a burst
11	SEQ	Subsequent transfers of a burst

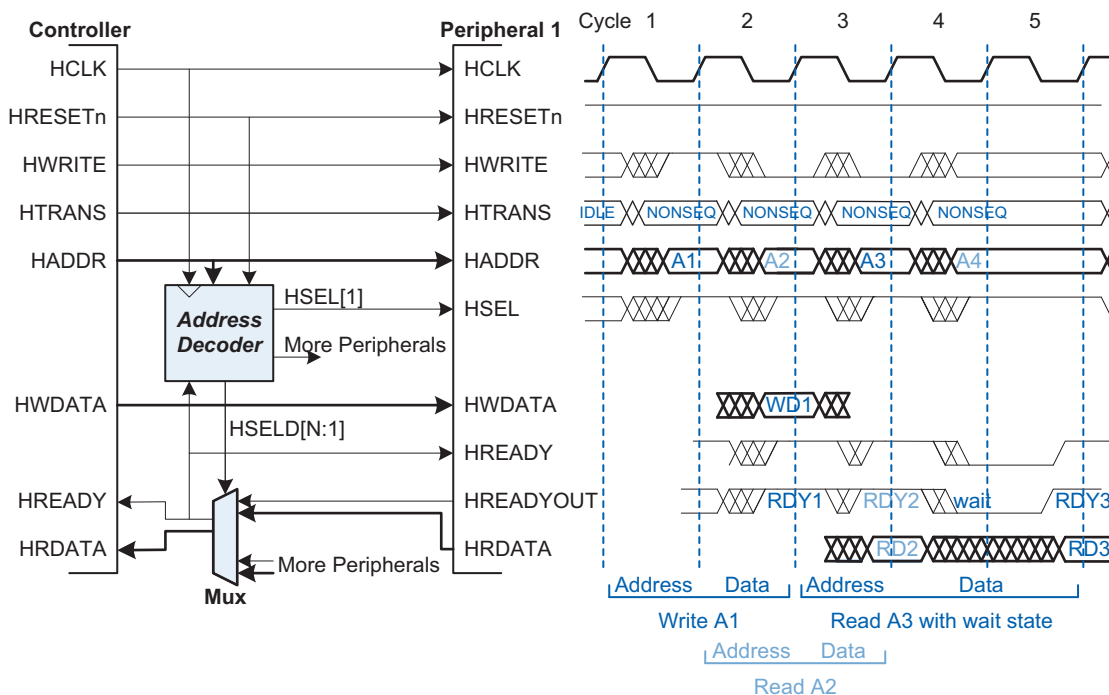


Fig. 9.10 AHB interface in the uncore.

responds with `HRDATA`. The peripheral also raises `HREADYOUT` to signal that the Data phase is complete, or the peripheral may keep it low to insert one or more wait states before completing. The Address phase of the next transfer begins when the previous transfer is in its Data phase. This requires an extra register to pipeline `HADDR` and the control signals. Because of this pipelining, AHB has nearly twice the throughput of APB.

The AHB multiplexer chooses the `HREADYOUT` and, on a read, `HRDATA` from the selected peripheral. It chooses based on `HSELD`, a version of `HSEL` delayed to align with the Data phase. As in APB, when no `HSEL` signals are asserted, the AHB mux asserts `HREADY` to prevent the bus from hanging. This is called the *default subordinate* behavior. AHB also sends `HREADY` to the peripherals to mark the completion of one transaction and start of the next. When `HTRANS` is `NONSEQ` or `SEQ` and a peripheral receives `HREADY` and `HSEL`, the peripheral must capture the Address phase inputs so that it can generate the addresses for the next transactions.

Fig. 9.11 shows a schematic of the address decoder and multiplexer, including the comparators to check `HADDR` against a peripheral's address range to find `HSEL`, the AND-OR muxing for `HRDATA` and `HREADY`, the `HSELD` delay register, and the default `HREADY` logic. The Wally PMA (physical memory attribute) checker, `pmachecker`, located in the memory

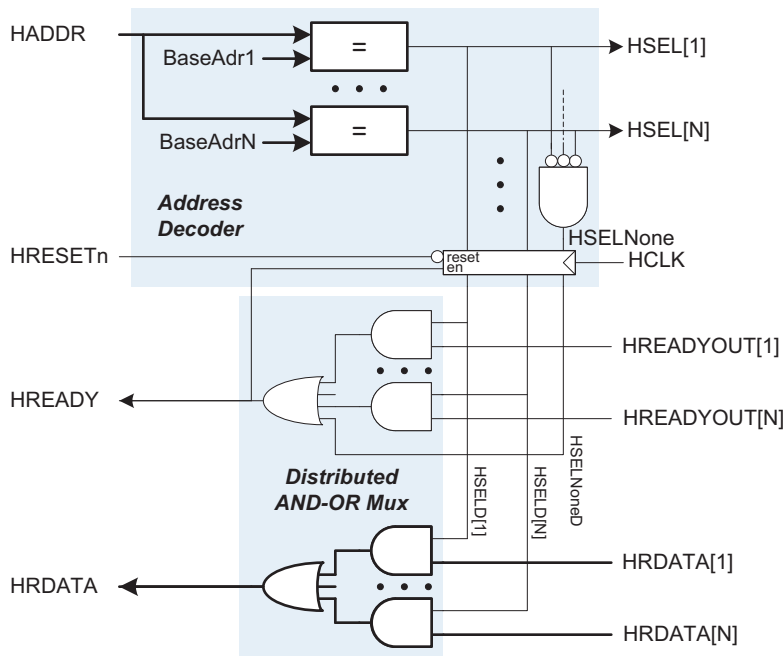


Fig. 9.11 AHB address decoder and multiplexer.

management unit (MMU) (Fig. 11.16), expands the address decoders to also check access types and sizes.

9.2.2.1 Additional AHB Signals

A full AHB link has several other signals shown in Fig. 9.12 that were left out of Fig. 9.10 for simplicity. Table 9.4 lists the main AHB signals other than HCLK and HRESETn. The controller sends HSIZE, HWSTRB, HBURST, HPROT, and HMASTLOCK and optionally HEXCL, HMASTER, and HNONSEC. All of these signals are transmitted in the Address phase except HWSTRB, which is sent in the Data phase with HWDATA. The peripheral replies with HRESP and, optionally, HEXOKAY to indicate success of an exclusive transfer. HRESP is a 2-bit signal that shows the status of the transfer (00: OKAY, 01: ERROR, 10: RETRY, 11: SPLIT).

HSIZE indicates the size of the transfer, as defined in Table 9.5, not to exceed the width of the data buses. HWSTRB (write strobe) conveys byte write enables to write only certain bytes within a larger transfer. For example, Table 9.6 shows two ways to write a single byte using an AHB bus. It assumes a little-endian memory system with a 32-bit bus and writes 0x42 to address 0x1002. The first option sends a whole word and only enables 1 byte in the word to be written, and the second option makes a single-byte transfer.

For Wally, HADDR width (see Table 9.4) is ADDR_WIDTH = PA_BITS (physical address bits), which is 34 bits for RV32 and 56 bits for RV64, as will be discussed in Chapter 11.

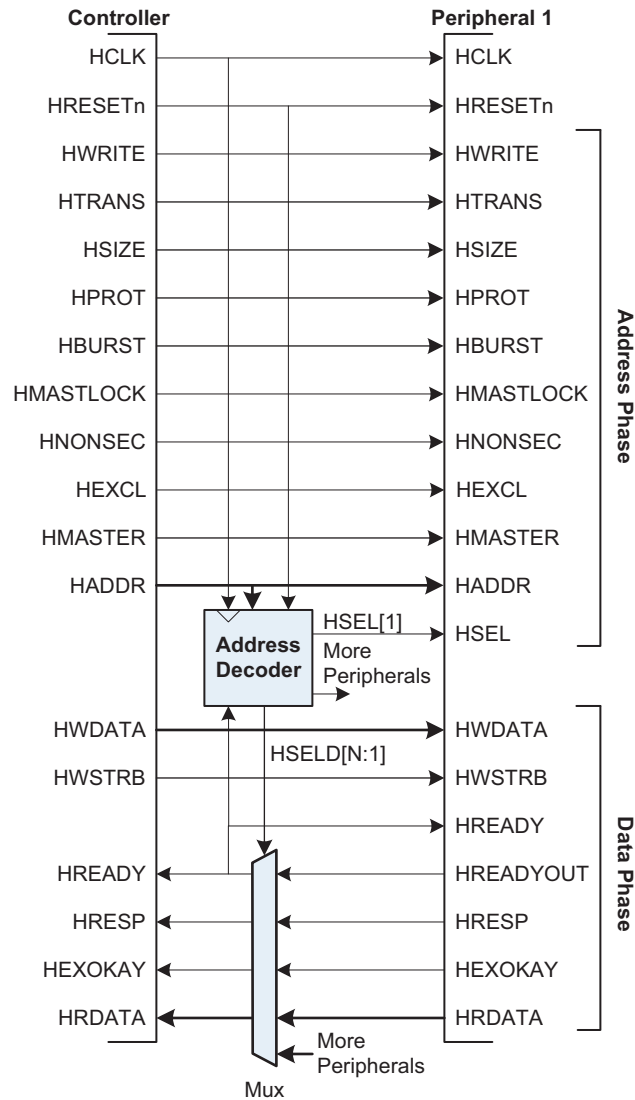


Fig. 9.12 Full AHB interface.

Table 9.4 Main AHB signals.

Signal	Description	Width (bits)	Phase	Source
HADDR	Address to read or write	ADDR_WIDTH	Address	Controller
HRDATA	Read data from peripheral	AHBW	Data	Peripheral
HWDATA	Write data to peripheral	AHBW	Data	Controller
HWRITE	Write enable	1	Address	Controller
HREADY	Peripheral asserts when Data phase completes	1	Data	Peripheral
HTRANS	Type of transaction (see Table 9.3)	2	Address	Controller
HSIZE	Size of transaction (see Table 9.5)	3	Address	Controller
HWSTRB	Write strobe: indicates which bytes to write	AHBW/8	Data	Controller
HBURST	Information about bursts (see Section 9.2.2.5)	3	Address	Controller
HPROT	Protection information	4	Address	Controller
HMASTLOCK	Master lock for atomic instructions	1	Address	Controller
HEXCL	Exclusive transfer for atomic instructions (optional)	1	Address	Controller
HMASTER	Master for atomic instructions (optional)	0-8	Address	Controller
HNONSEC	Identifies nonsecure transactions (optional)	1	Address	Controller
HRESP	Peripheral transmits status of transfer	1	Data	Peripheral
HEXOKAY	Exclusive transfer okay: is 1 when successful (optional)	1	Data	Peripheral

Table 9.5 HSIZE transfer sizes.

HSIZE[2:0]	Number of Bits	Size
0	8	Byte
1	16	Half word
2	32	Word
3	64	Double word
4	128	
5	256	
6	512	
7	1024	

Table 9.6 Two options for a 1-byte transfer on an AHB bus.

Signal	Option 1	Option 2
HADDR	0x1000	0x1002
HSIZE	2 (4 bytes)	0 (1 byte)
HWSTRB	0b0100	0b0001
HWDATA	0x00420000	0x00000042

Many systems don't need all the signals listed in Table 9.4. The controller can tie HSIZE to the bus size (e.g., 2 for a 32-bit bus), HPROT = 3 (ignored by most peripherals), HBURST = 0 (no burst), and HMASTLOCK = 0, and omit optional signals. In a system where peripherals don't generate errors, the peripheral can tie HRESP to 0 and the controller can ignore HRESP.

HTRANS[1] indicates that the transaction is NONSEQ or SEQ.

HPROT conveys optional information about protection, including cacheability, bufferability, and privilege/user and data/instruction modes. HBURST gives hints about burst transfers as will be discussed in Section 9.2.2.5. HMASTLOCK (master lock) and the optional HEXCL/HMASTER (exclusive/master) signals can be used for atomic operations. The optional HNONSEC signal identifies nonsecure transactions for systems with security extensions.

9.2.2.2 AHB Peripherals

An AHB peripheral must perform several functions to interface with the bus:

- ▶ Detect new transactions.
- ▶ Retime writes, so that HWDATA arrives in the Data phase rather than in the Address phase.
- ▶ Read or write registers in the peripheral.
- ▶ Produce HRDATA in the Data phase.
- ▶ Generate the HREADYOUT, HRESP, and HEXOKAY acknowledgement.

A peripheral starts a new transaction when the bus contains a valid transaction, the peripheral is selected, and the previous transaction is completing. The peripheral detects these conditions by asserting an internal signal, `initTrans`, when HTRANS is NONSEQ or SEQ, HSEL is asserted, and HREADY is asserted from the previous transaction. When `initTrans` asserts, the peripheral either reads or writes depending on the value of HWRITE.

When interacting with a peripheral that has no wait states, a controller reads from the peripheral on the rising edge after the Address phase and HRDATA is available during the Data phase. However, a write to the peripheral occurs on the rising edge *after* the Data phase because HWDATA is not available until the Data phase. Hence, when HREADY is asserted, HADDR and HWRITE must be captured at the end of the Address phase to retain their values for writes in the Data phase.

A peripheral may tie HREADYOUT to 1 if it never needs wait states. Otherwise, the peripheral only asserts HREADYOUT when the access is complete.

9.2.2.3 Example: AHB SRAM Interface

Fig. 9.13 shows a single-port synchronous SRAM with an AHB interface. It asserts `initTrans` when the RAM is selected (`HSEL = 1`), a transaction is pending (`HTRANS[1] = 1`), and the previous transaction is completing (`HREADY = 1`). `memRead` or `memWrite` is asserted when `initTrans` is 1. A register delays HADDR and `memWrite` by a cycle to align with HWDATA arriving in the Data phase for writes and to retain the values for transactions with wait states. A multiplexer chooses the RAM address from HADDR on reads with

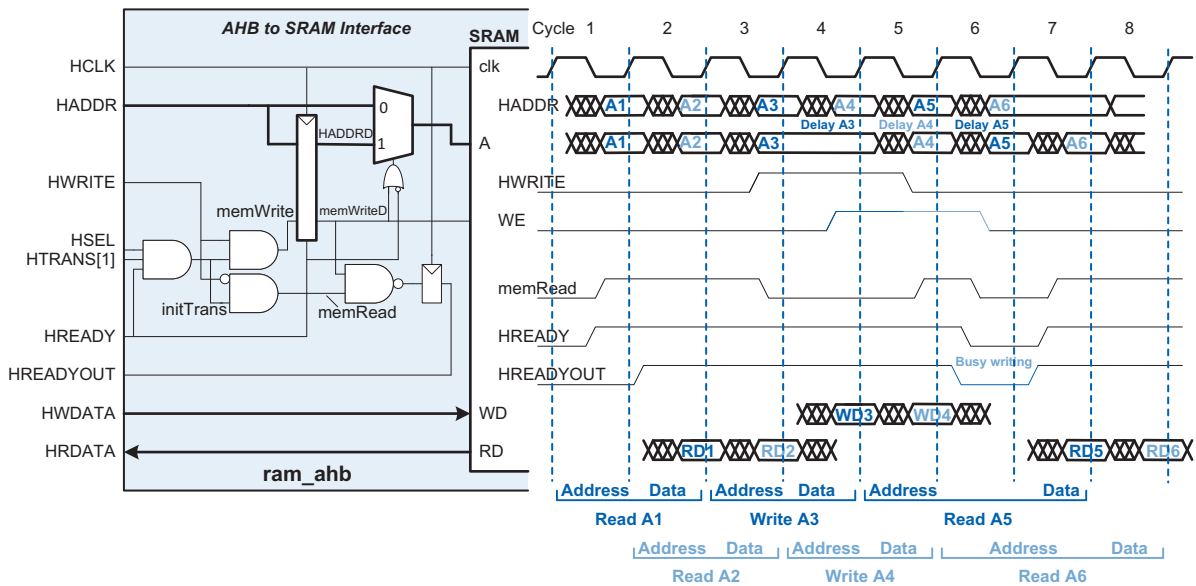


Fig. 9.13 AHB SRAM interface.

no wait state or from the delayed `HADDRD` for writes and for reads with one or more wait states.

Reads occur on the rising edge at the end of the Address phase. Writes occur on the rising edge at the end of the Data phase, so their address and write enable must be delayed a cycle. Thus, the SRAM can accommodate consecutive reads and consecutive writes and a read followed by write — but not a write immediately followed by a read because the SRAM has a single read/write port and cannot accept the read and write addresses on the same cycle. Therefore, the SRAM interface inserts a single wait state by lowering `HREADYOUT` for one cycle for a write followed by read.

The waveforms in [Fig. 9.13](#) show two consecutive reads to A1 and A2, two consecutive writes to A3 and A4, and then two more reads to A5 and A6. The read after write has a wait state created by lowering `HREADYOUT`. As a result, `HREADY` and `memRead` must wait, delaying the read from A5.

9.2.2.4 Example: AHB-to-APB Bridge

[Fig. 9.14](#) shows an AHB-to-APB bridge. The left side behaves as a peripheral on an AHB bus. The right side behaves as a controller on an APB bus. The bridge translates AHB transactions into APB transactions, adjusting the phases of the signals before returning `HREADYOUT`. The bridge supports a configurable number of peripherals. It expects select signals for each peripheral from the AHB address decoder. The bridge contains the APB read mux to choose the results from the selected peripheral. The bridge accepts

A simple SRAM expects the write data at the same time as the address and control signals. The AHB protocol complicates the interface because write data arrives a cycle late. This timing is an artifact of AHB's roots as the native memory interface of early ARM processors [Flynn97]. For example, the ARM6 had a 3-stage pipeline in which the address arrived a cycle before the write data.

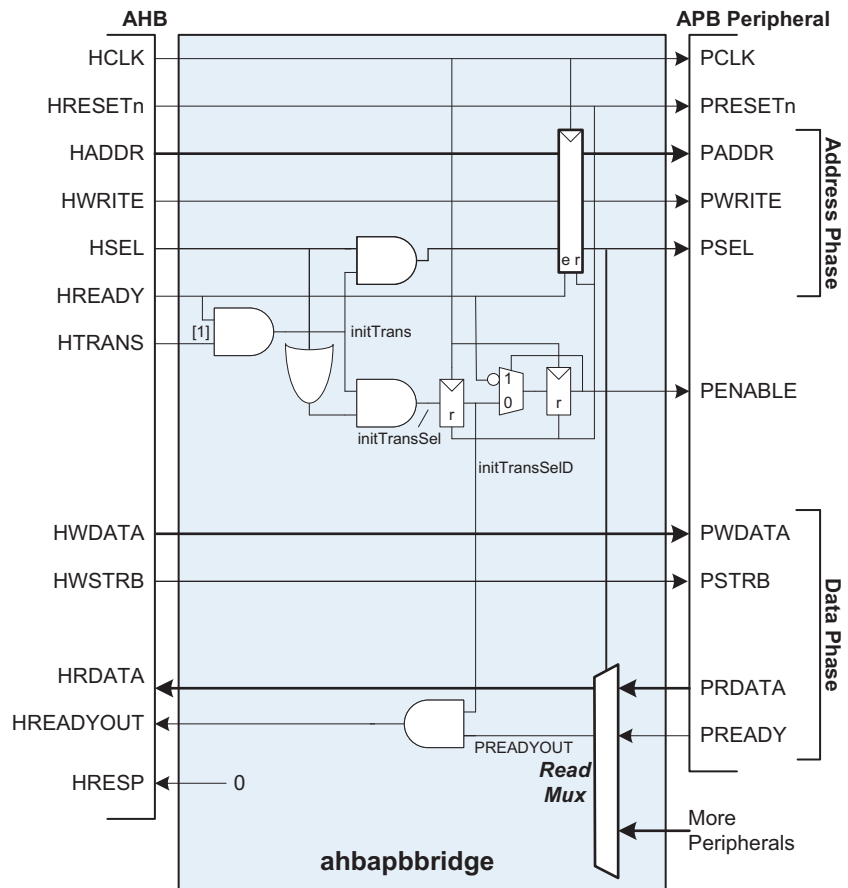


Fig. 9.14 AHB-to-APB bridge.

the full width of the data buses and uses **HWSTRB** byte enables for subword writes.

Fig. 9.15 shows how an AHB read and write without wait states would be translated to APB signals. **PRDATA** (APB read data) is available in the cycle after **PADDR** changes, so no adjustments need to be made on reads from the APB peripheral. On a write, because APB peripherals are not pipelined, **HADDR**, **HWRITE**, and **HSEL** are delayed by a cycle to align with **HWDATA**. The resulting delay in **PREADY** would also cause **HREADYOUT** to be delayed by a cycle (not shown).

9.2.2.5 Burst Transfers

DRAM is organized into rows called *pages*, which are typically 4–16 Kib in size. DRAM reads a full page of memory into its sense amplifiers and

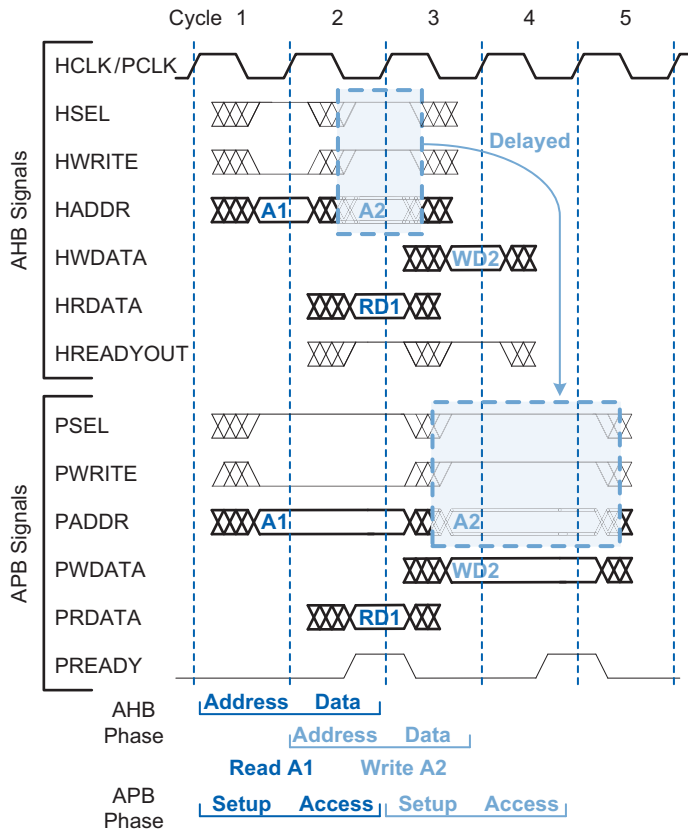


Fig. 9.15 Translation of AHB to APB signals.

returns words much faster from that open page than from a new page. Thus, a DRAM controller is much faster at bursts of accesses to consecutive memory locations than accesses to random memory locations.

AHB uses SEQ transactions and the H_{BURST} signal to tell a peripheral that such a burst is occurring so that some peripherals can respond faster. The AHB standard defines various H_{BURST} encodings for fixed and variable length, with addresses that increment or wrap around an aligned line. Simple peripherals ignore the SEQ and H_{BURST} hints and just respond to $HADDR$.

For example, Fig. 9.16 shows a 4-beat write burst followed by a single read. The H_{BURST} signal is set to INCR during the write burst to indicate addresses are incrementing. On the first beat, H_{TRANS} is NONSEQ, and on subsequent beats, it is SEQ. H_{SIZE} is 2 (010), so each beat transfers one 32-bit word, and the 4 beats transfer 128 bits all together.

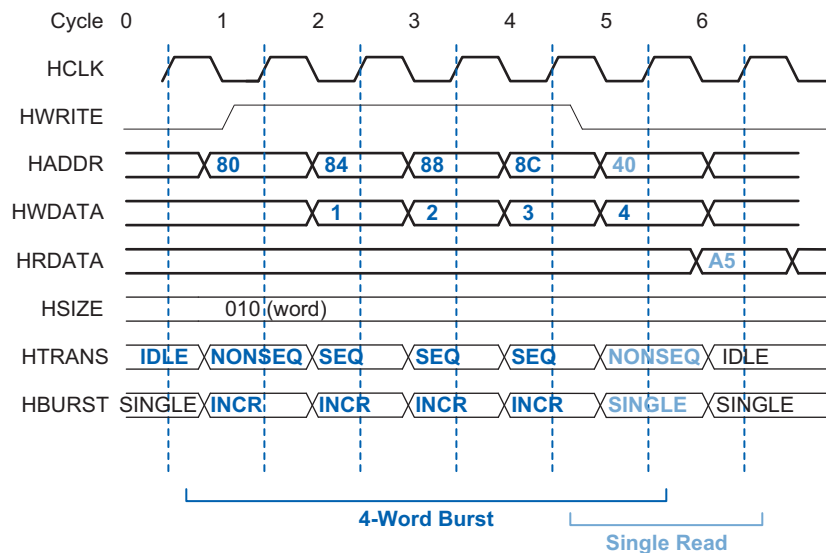


Fig. 9.16 AHB burst transfer.

9.2.2.6 Multilayer AHB

Multicontroller AHB systems use independent AHB buses called *layers* for each controller. The layers are connected to peripherals through an interconnect matrix with decoders for each controller and arbiters managing access to each peripheral, as shown in Fig. 9.17.

9.2.3 AXI

AHB is designed for the address phase to take a single clock cycle. The response may take multiple cycles, but one transaction must complete before the next begins. However, modern SoCs must handle multicycle

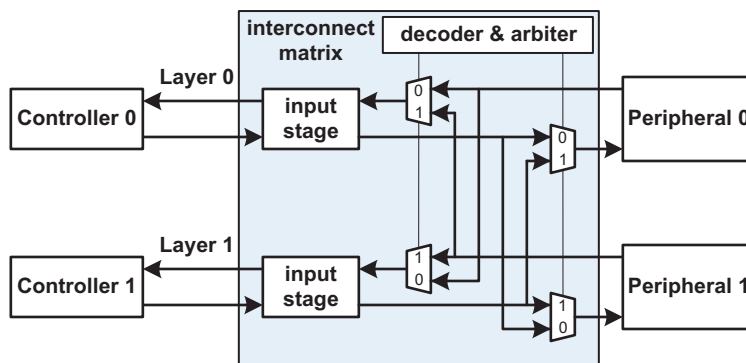


Fig. 9.17 Multilayer AHB. (Adapted from [MultilayerAHB04].)

on-chip delays, multiple cores accessing the same peripherals and memory, cache coherence, and devices acting as both a controller and peripheral. As clock frequencies have increased, the flight time to send information across an SoC has become multiple clock cycles. An SoC also often has many heterogeneous cores communicating with one or more DDR DRAM controllers as well as accelerators and other on-chip peripherals. The DRAM can service multiple concurrent requests, and access to an open page of DRAM is much faster than access to a new page. To maintain cache coherence, one core's cache may need to snoop transactions from other cores. Direct memory access (DMA) devices may act as a peripheral when being configured by a core, then as a controller when issuing memory requests.

The Advanced eXtensible Interface (AXI) addresses all these issues by providing independent pipelined address and data channels operating at high clock rates that handle multiple pending requests and out-of-order responses [AXI21]. AXI uses the terms *manager* and *subordinate* in place of controller and peripheral. It was designed for multiple managers and subordinates connected by flexible interconnect matrices that support cache coherency and DMA devices. Each channel can have arbitrary latency, so pipeline registers can be added to meet clock speed targets. AXI handles large bursts, such as transferring entire cache lines between memory and a core; the manager only sends the burst's start address, and the subordinate autoincrements the address for subsequent beats. Byte strobes enable unaligned data transfers. Simple bridges connect AXI to AHB or APB. AXI is now the leading standard for SoCs with high-performance multicore processors. It is also widely used on FPGAs, especially for high-bandwidth IP blocks such as memory controllers. This section briefly outlines the AXI4 protocol.

AXI4 is based on five independent transaction channels. Table 9.7 summarizes the main AXI signals on the five channels [AXI21]. For a read transaction, shown in Fig. 9.18(a), a manager sends address and control information on the *read address channel* to a subordinate. The subordinate responds with a burst of data on the *read data channel*, possibly taking multiple beats (data transfers) to convey the information. For a write transaction, shown in Fig. 9.18(b), a manager sends address and control information on the *write address channel* to a subordinate. It also sends a burst of data on the *write data channel*. The write data burst can come after the address. The subordinate acknowledges on the *write response channel*. Each of the device's five channels may be routed through an interconnection matrix arbitrating among multiple managers and subordinates.

AXI uses a global clock called `ACLK` and an active low reset `ARESETn`. Each channel has a *source* that sends most of the information and asserts `VALID` and a *destination*, which replies with a `READY` signal. Every

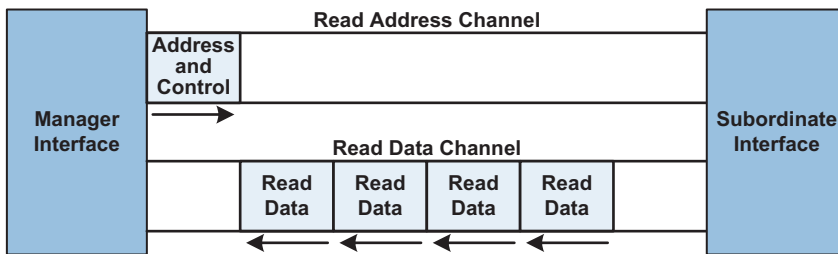
Table 9.7 AXI channel signals.

Signal	Description	Width (bits)	Source
Read Address Channel			
ARID	Identification tag	ILEN	Manager
ARADDR	Starting address	ALEN	Manager
ARLEN	Burst length in beats: $ARLEN + 1$	8	Manager
ARSIZE	Bytes in a beat: $2^{ARSIZE} = 1 \dots 128$	3	Manager
ARBURST	Burst mode (01 = increment)	2	Manager
ARVALID	Channel signals are valid	1	Manager
ARREADY	Transfer on channel can be accepted	1	Subordinate
Read Data Channel			
RID	Identification tag	ILEN	Subordinate
RDATA	Read data	2^{3-10} (8-1,024)	Subordinate
RRESP	Status (00 = normal successful transfer)	2	Subordinate
RLAST	Last beat in the transaction	1	Subordinate
RVALID	Channel signals are valid	1	Subordinate
RREADY	Transfer on channel can be accepted	1	Manager
Write Address Channel			
AWID	Identification tag	ILEN	Manager
AWADDR	Starting address	ALEN	Manager
AWLEN	Burst length in beats: $AWLEN + 1$	8	Manager
AWSIZE	Bytes in a beat: $2^{AWSIZE} = 1 \dots 128$	3	Manager
AWBURST	Burst mode (01 = increment)	2	Manager
AWVALID	Channel signals are valid	1	Manager
AWREADY	Transfer can be accepted	1	Subordinate
Write Data Channel			
WID	Identification tag	ILEN	Manager
WDATA	Write data	2^{3-10} (8-1024)	Manager

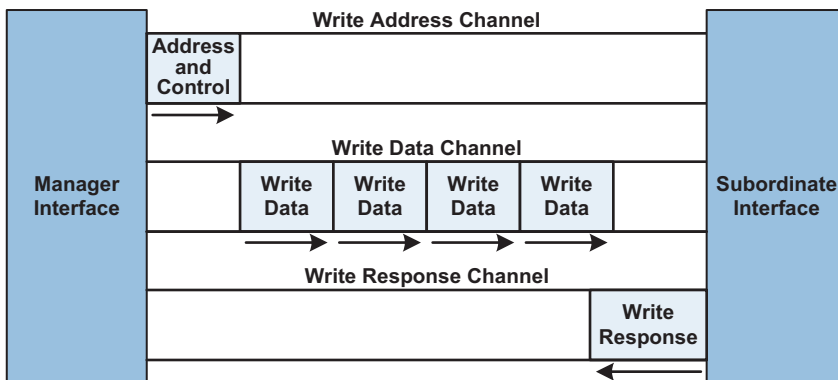
(Continued)

Table 9.7 AXI channel signals. Cont'd

Signal	Description	Width (bits)	Source
WSTRB	Write strobe: which bytes hold valid data	WDATA/8	Manager
WLAST	Last beat in the transaction	1	Manager
WVALID	Channel signals are valid	1	Manager
WREADY	Transfer on channel can be accepted	1	Subordinate
Write Response Channel			
BID	Identification tag	ILEN	Subordinate
BRESP	Status (00 = normal successful transfer)	2	Subordinate
BVALID	Channel signals are valid	1	Subordinate
BREADY	Transfer on channel can be accepted	1	Manager



(a)



(b)

Fig. 9.18 AXI transactions: (a) read and (b) write. (Adapted from [AXI21])

transaction includes an identification tag to distinguish out-of-order responses. AXI also supports other optional control signals that convey atomic transactions, cache coherence, security, quality of service, regions, and user-defined control. The widths of the ID, address, and data buses are system-dependent (ILEN, ALEN, and 8–1024 bits).

Every channel has `VALID` and `READY` handshaking signals, thus providing two-way flow control so both the manager and subordinate can control the rate of information movement. The signals are active high, so they are asserted when 1. The channel source places information on the channel and raises the `VALID` signal to begin a transaction. The destination asserts its `READY` signal to indicate that it is able to accept a transaction; it may do so even before seeing `VALID`. The transaction occurs on the next rising `ACLK` edge when both `VALID` and `READY` are asserted. The source can then lower `VALID`.

Fig. 9.19 illustrates the handshaking; signals sent by the manager are blue, and those sent by the subordinate are black. Single-headed arrows point to signals that may be asserted before or after the signal at the start of the arrow. Double-headed arrows point to signals that must be asserted only after the signal at the start of the arrow asserts.

Fig. 9.19 (a) shows the read transaction handshake involving the Read Address and Read Data channels:

- The manager places information on the Read Address channel and asserts `ARVALID` without waiting for `ARREADY`.
- The subordinate raises `ARREADY` when it is ready to accept the address. This could be before or after `ARVALID`.
- When the subordinate sees both `ARVALID` and `ARREADY`, it accepts the Read Address transaction. When information is available on the

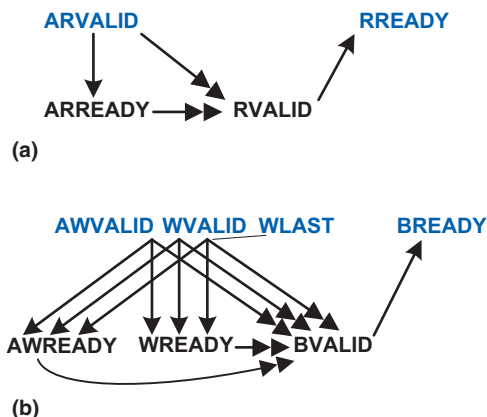


Fig. 9.19 AXI handshaking: (a) read and (b) write. (Adapted from [AXI21].)

Read Data channel, which may take some time, the subordinate asserts `RVALID` without waiting for `RREADY`. The subordinate sets `RRESP = 0` for a successful normal transaction, or nonzero otherwise.

- ▶ The manager raises `RREADY` when it is ready to accept the Read Data. This could be before or after `RVALID`.

Fig. 9.19(b) shows the write transaction handshake involving the Write Address, Write Data, and Write Response channels. Note that the subordinate may receive information on the Write Address and Write Data channels in either order because of different numbers of register stages in the channel or uncertain delays through the interconnect matrix.

- ▶ The manager places information on the Write Address channel and asserts `AWVALID`. It also places the first beat of a burst on the Write Data channel and asserts `WVALID`. The manager continues sending beats on consecutive cycles and asserts `WLAST` on the final cycle. The manager does not wait for `AWREADY` or `WREADY` to drive the valid (`AWVALID`/`WVALID`) signals.
- ▶ The subordinate raises `AWREADY` and `WREADY` when it is ready to accept the address and data, respectively. This could be before or after `AWVALID` and `WVALID`.
- ▶ When the subordinate sees `AWVALID` and `AWREADY`, it accepts the Write Address transaction.
- ▶ When the subordinate sees `WVALID` and `WREADY`, and `WLAST` arrives, it accepts the Write Data transaction.
- ▶ When the subordinate has accepted both Write Address and Write Data transactions for the same ID tag, it places `BRESP` for that ID on the Write Response channel and raises `BVALID`.
- ▶ The manager raises `BREADY` when it is ready to accept the response. This could be before or after `BVALID`.

If sending information on a channel takes multiple cycles, a lag occurs between the time a subordinate deasserts `READY` and the time the manager sees that the subordinate cannot receive more transactions. The subordinate either needs an input buffer to accept transactions during this lag or must handshake by waiting to assert `READY` until a `VALID` transaction arrives on the channel. Waiting on handshakes over a multicycle channel keeps the bus idle most of the time, so buffering is necessary for good performance.

AXI5 adds various optional capabilities (e.g., loopback, poison bits, and ARM-specific features) to the AXI4 protocol described earlier. AXI4-Lite is a subset of AXI limited to in-order transactions comprising single-beat bursts of 32 or 64 bits. A simple bridge can connect AXI-Lite subordinates to an AXI bus. The AXI Coherence Extensions (ACE)

protocol provides more hardware support for cache coherency. The Coherent Hub Interface (CHI) extends ACE for large, scalable communication networks with directory-based cache coherence.

AXI is ideal for high-performance on-chip communication because it is easy to support wide buses on chip, and they can be pipelined for speed. AXI's bursts and out-of-order capabilities are well-suited for linking cores to on-chip memory controllers. However, a 32-bit AXI5 bus involves 224 interface pins [AXI21], which is far too many for a chip-to-chip interface on a simple microcontroller. AXI also entails greater implementation complexity, such as buffering out-of-order responses. AHB is a better fit for small cores with simple peripherals.

9.3 OTHER SoC Buses

Several other buses are found on SoCs.

Wishbone is an open-standard SoC bus developed by Silicore Corporation and maintained by OpenCores for portable IP cores [Wishbone10]. It has similarities to AHB but doesn't delay write data by a cycle. It lacks the cache coherency features of AXI and other buses. Wishbone is compatible with many free OpenCores, but it is not used as much in commercial systems.

The OpenHW group uses the Silicon Labs Open Bus Interface [OBI22] in several of its RISC-V processors. It resembles a simplified AXI interface, with an address and response channel.

CoreConnect, developed by IBM, is another commercial SoC bus. It includes a high-speed processor local bus (PLB) with a bridge to a simpler on-chip peripheral bus (OPB). It is available for free but requires a license from IBM. Xilinx FPGAs used CoreConnect IPs before switching to AXI; little CoreConnect IP is now available.

SiFive developed an open-standard SoC bus called TileLink for ease of implementation and to provide high speed and cache coherence [TileLink20]. A key difference between AHB and TileLink is that AHB allows one controller to directly communicate with multiple peripherals, but TileLink allows only point-to-point links between a single controller and a single peripheral. TileLink adds a switch component when it needs to connect multiple agents or peripherals. TileLink's point-to-point links facilitate high-speed SoC networks in which the link is pipelined to permit multiple messages in-flight concurrently. However, SiFive has found that customers preferred the broadly accepted AHB or AXI bus rather than a novel bus.

9.4 TEST PLAN

The AHB bus has primarily been tested through normal use of memory and peripherals.

The `ahb64_ram_{0/1/2}_{0/1}_rv64gc` tests stress the variable latency AHB interface and burst mode operation. The first number gives the

Chapter 10 describes caches.

RAM latency (0, 1, or 2 extra cycles of memory latency), and the second number indicates whether burst mode cache line fills are enabled (1) or not (0). These tests run `fadd_b11-01`, where the active data exceeds the cache sizes and, thus, generates memory traffic. A more comprehensive test plan would be beneficial.

9.5 WALLY IMPLEMENTATION

The basic pipelined processor core from Fig. 7.13 has no bus interface because its memories (DTIM and IROM) are inside the core and it does not access peripherals. This section adds an *uncore* module with memory and peripherals that communicate with the core over an AHB bus. The memory and peripherals are collectively called AHB *devices*. When the `BUS_SUPPORTED` configuration variable in `config.vh` is 1, Wally adds an AHB interface to the load/store unit (LSU) and instruction fetch unit (IFU) to access peripherals or memory other than the DTIM or IROM. Wally also adds an *external bus unit* (EBU) that arbitrates between requests from the LSU and IFU and passes the selected request onto the uncore's AHB bus.

Wally's AHB interface is shown in Fig. 9.20. Wally includes AHB, EBU, and uncore modules when `BUS_SUPPORTED = 1`. The `wallypipelinedsoc` top-level module comprises the `wallypipelinedcore` and `uncore` modules. `wallypipelinedcore` is the processor core, which includes the LSU, IFU, and EBU (modules `lsu`, `ifu`, and `ebu`). The `uncore` includes on-chip RAM (`ram_ahb`) and ROM (`rom_ahb`) and an optional AHB

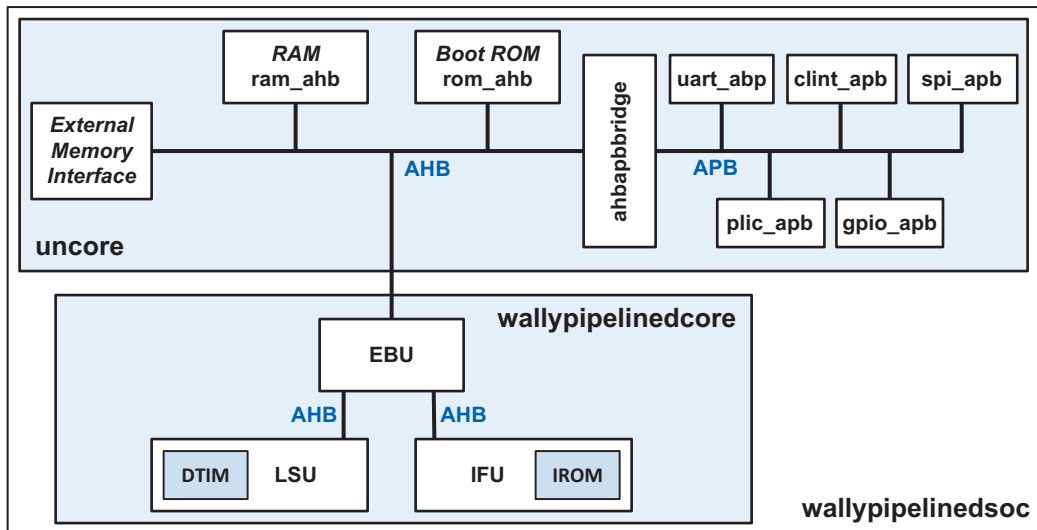


Fig. 9.20 Wally AHB interface and top-level system diagram.

Table 9.8 Wally bus memory configuration options.

Configuration Parameter	Meaning	Address
UNCORE_RAM_SUPPORTED	RAM in the uncore on AHB	0x80000000
BOOTROM_SUPPORTED	ROM in the uncore on AHB for bootloader	0x1000
EXT_MEM_SUPPORTED	AHB interface to RAM outside the uncore (e.g., via a DDR memory controller on an FPGA)	0x80000000

interface to off-chip memory. [Table 9.8](#) summarizes the configuration parameters for these bus-based memories. The uncore also has an AHB-to-APB bridge (`ahbapbbridge`) to various peripherals described in [Chapter 20](#). In the `rv{32/64}i` configuration (or to increase performance in configurations with a bus), the LSU and IFU contain a DTIM and IROM, as described in [Chapter 7](#). [Chapters 12 and 13](#) add optional caches.

To enable bus access, the LSU and IFU add AHB interfaces that include an FSM to sequence the AHB phases and a register to capture `HRDATA`. The core adds an EBU to arbitrate between LSU and IFU requests. Wally adds a new top-level `wallypipelinedsoc` module that contains the `wallypipelinedcore` and the uncore memories and peripherals.

9.5.1 LSU and IFU AHB Interface

This section describes a configuration where the LSU and IFU add AHB bus interfaces to access uncore memory instead of or in addition to the DTIM and IROM memories located in the core. The EBU arbitrates between LSU and IFU bus requests so that they can both access uncore AHB devices. The bus `HRDATA` and `HWDATA` widths are `XLEN`.

[Fig. 9.21](#) shows the LSU with a bus interface. Compared to [Figs. 7.13 and 8.19](#) where the LSU had a DTIM, this updated LSU keeps the subword and biendian support but removes the DTIM and adds the AHB signal pins and an `ahbinterface` module that has fetch and write buffers and a `busfsm` control state machine. The fetch buffer holds the `HRDATA` read from the bus, and the write buffer holds the values driving `HWDATA` and `HWSTRB` onto the bus. The `busfsm` walks through the phases of the AHB transaction and generates AHB control signals and the fetch buffer enable at appropriate times. There is no need for the DTIM `A` (address) multiplexer of [Fig. 7.13](#) because the bus transaction starts in the Memory stage for both loads and stores; so `IEUAdrM` ties directly to `LSUHADDR`.

The `rv32imc` configuration has a DTIM and IROM as well as a bus to uncore peripherals. It does not include external (uncore) memories.

[Fig. 9.21](#) omits several AHB control signals for brevity. `HBURST` is hardwired to 0 for now; [Section 12.2](#) introduces burst-mode cache line fetches and writebacks. `HPROT` is tied to 4'b0011 and `HMASTLOCK` is tied to 0 because protection and bus locking are not supported. `HRESP` is ignored because the memory system does not produce any error conditions. The uncore module generates `HSEL` based on `HADDR` and configured memory ranges for memories and peripherals, as discussed in [Section 9.5.3.1](#).

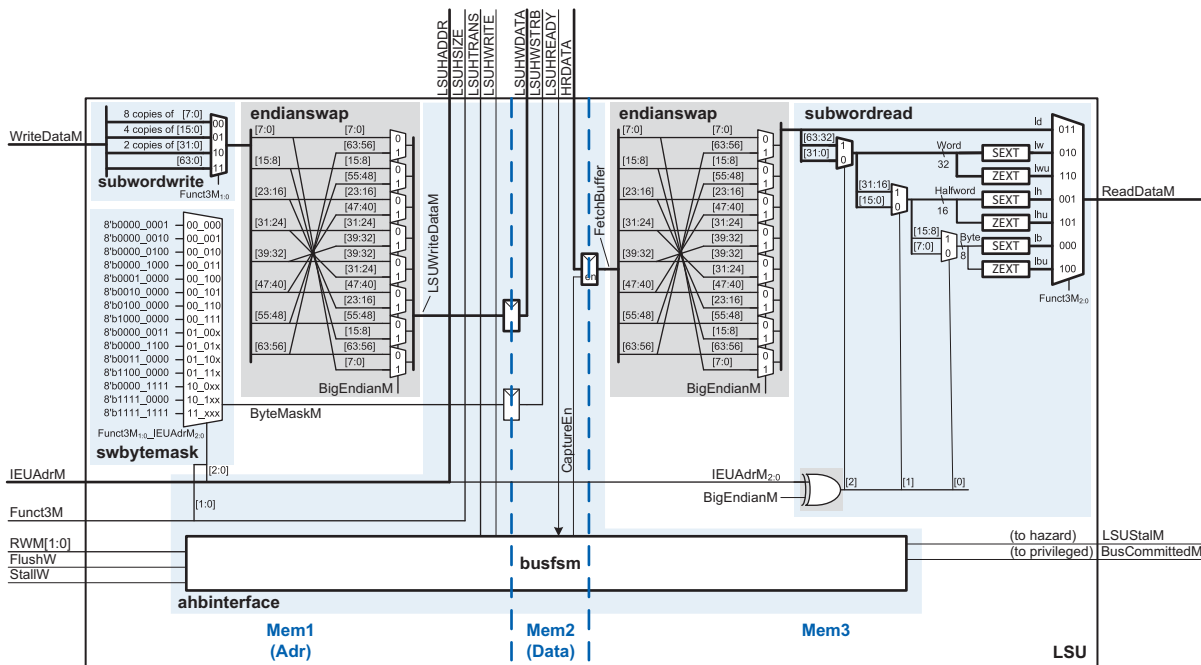


Fig. 9.21 LSU with AHB interface (lsu module configured with ``BUS_SUPPORTED = 1`, ``DTIM_SUPPORTED = 0`, ``DCACHE_SUPPORTED = 0`).

Recall from Table 7.8 that an LSU DTIM access involves work in the Execute, Memory, and Writeback stages but only stalls for hazards. In contrast, an access to bus-based memory always takes at least two extra cycles, as shown in Fig. 9.22 for `ld` followed by additional instructions (`i1`, `i2`, ...). The LSU stalls the CPU during these extra cycles by setting `BusStall = 1`, so the LSU with an AHB interface behaves as a multicycle (nonpipelined) system. Table 9.9 summarizes the steps the LSU takes to access the AHB bus. The Memory stage is stalled while it takes at least three cycles:

The Mem2 and Mem3 phases could be merged, but this might increase the cycle time of the processor. In a processor with a DTIM/IROM or cache, bus accesses are infrequent, so it is better to have a fast cycle time than a lower bus latency.

Cycle	F	D	E	M	W	State/AHB	BusStall
0	<code>ld</code>						0
1	<code>i1</code>	<code>ld</code>					0
2	<code>i2</code>	<code>i1</code>	<code>ld</code>				0
3	<code>i3</code>	<code>i2</code>	<code>i1</code>	<code>ld</code>		Adr	1
4	<code>i3</code>	<code>i2</code>	<code>i1</code>	<code>ld</code>		Data	1
5	<code>i3</code>	<code>i2</code>	<code>i1</code>	<code>ld</code>		Mem3	0
6	<code>i4</code>	<code>i3</code>	<code>i2</code>	<code>i1</code>	<code>ld</code>		0

Fig. 9.22 `ld` instruction using the AHB bus.

Table 9.9 LSU AHB access steps.

Stage	Read	Write
Execute	Compute memory address $IEUAdrE = \text{base address} + \text{offset}$	
Mem1/AHB Adr	Drive $HADDR = IEUAdrM$; generate other AHB control signals Writes also perform subword write muxing	
Mem2/AHB Data	Read $HRDATA$ synchronously from AHB device	Drive $HWDATA$ onto AHB
Mem3	Capture $HRDATA$ into $ReadDataM$ at start of cycle, then perform subword read and endian muxing	Write $HWDATA$ into device
Writeback	Write $ReadDataW$ into register file	(Nothing)

The AHB bus is pipelined: the current request's Adr phase overlaps with the previous request's Data phase. Wally could leverage this pipelining to eliminate the two bus stall cycles and, thus, reduce AHB memory access latency. But this would require modifying the standard 5-stage pipeline so that memory accesses occur in 2 cycles instead of 1. Right now, the Fetch or Memory stage receives its address at the beginning of the stage and produces its read or write data at the end. To take advantage of AHB's pipelining, the address would need to be calculated a cycle earlier (which would increase the LSU's $IEUAdrE$ critical path) or the data would need to be read or written a cycle later (which would add an extra cycle to the LSU's Load Delay Hazard). These modifications could be made, but bus transactions are infrequent because most accesses are filtered by the cache, and Amdahl's Law says to make the common case fast. So, it is better to maintain a lower cycle time than pay the price for a lower bus latency.

- ▶ In the first phase, the AHB Adr phase (also called Mem1), $HREADY$ (in this case, $LSUHREADY$), may be low, which indicates the bus cannot take a request. The LSU holds the AHB Adr phase signals steady until $LSUHREADY$ is high.
- ▶ In the second phase, the AHB Data phase (also called Mem2), the device reads or writes. A slow device may stretch this phase beyond one cycle by keeping $LSUHREADY$ low.
- ▶ The third phase (Mem3) captures $HRDATA$ in the `FetchBuffer` register and muxes the `FetchBuffer` bits for either subword accesses or, if needed, to reorder bytes for endianness to produce $ReadDataM$.

The LSU's AHB bus interface has a three-state FSM controller (`busfsm`) shown in Fig. 9.23 that sequences the steps from Table 9.9 for both stores and loads. State names match the AHB specification's Adr and Data phases with the added third state, Mem3. All memory requests start in the Adr phase. If read or write is requested, `busfsm` asserts `BusStall` to stall the pipeline while the bus services the request. When the bus is ready, `busfsm` asserts $HTRANS = 2$ (NONSEQ) to initiate a single AHB read or write and advances to the Data state. While in the Data state, the FSM asserts `BusCommitted` to prevent the privileged unit from taking interrupts until the bus transaction completes. When the bus raises $HREADY$ to complete the Data phase, `busfsm` advances to Mem3 and raises `CaptureEn` on reads to enable the fetch buffer to sample $HRDATA$. While in Mem3, `BusStall` is lowered, so the pipeline can advance. The `Stall`, `BusStall`, and `BusCommitted` signals are discussed further in Section 9.5.5.

The IFU bus interface is similar to the LSU interface but is 32-bit little-endian read-only. Thus, it drops $HWDATA$, $HWRITE$, $HWSTRB$, and the subword and endian logic, as shown in Fig. 9.24, and $HSIZE$ is 010 to always fetch 4-byte instructions. The IFU `busfsm` receives `StallID` instead of `StallW`. In the

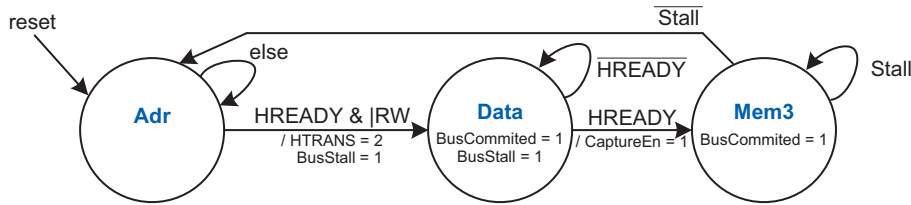


Fig. 9.23 busfsm state machine; Stall = StallW for the LSU and StallD for the IFU.

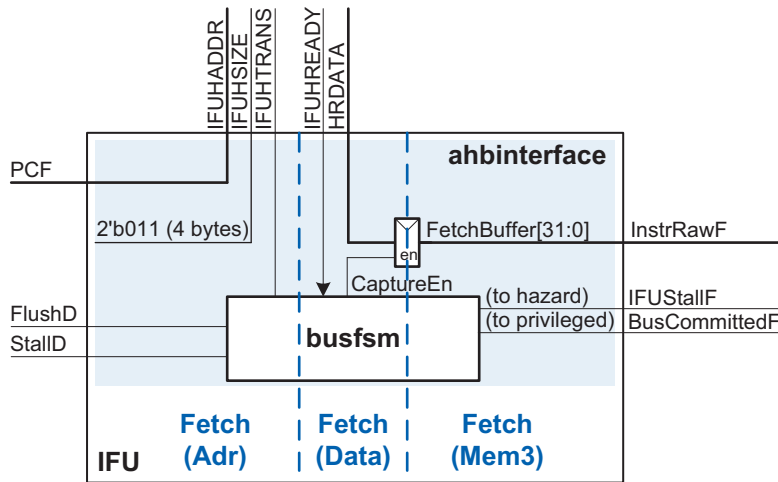


Fig. 9.24 IFU AHB interface (ifu module configured with ``BUS_SUPPORTED = 1`, ``IROM_SUPPORTED = 0`, ``ICACHE_SUPPORTED = 0`).

Without some sort of local memory, the processor runs slowly because each instruction or data access takes at least 3 cycles over the bus. [Chapter 12](#) describes the full LSU with a cache and bus. The cache services most accesses, while the bus fetches cache misses from RAM and accesses peripherals in the uncore.

Mem3 stage, the IFU outputs `InstrRawF` from the fetch buffer. The AHB signals from each unit have corresponding IFU or LSU prefixes, and the EBU must arbitrate between requests from the two units, as discussed next.

9.5.2 External Bus Unit

The external bus unit (EBU) implements a subset of the multilayer AHB protocol to arbitrate between the IFU and LSU AHB interfaces (also called controllers). LSU requests get priority. The selected controller's request is placed on the bus to the uncore. As shown in [Fig. 9.25](#), the EBU consists of an FSM arbiter (`ebufsmarb`), muxes to choose the selected controller, and registers to retain the other controller's request. [Section 12.2.3](#) extends the EBU to handle burst transactions for fast cache line fills.

When `LSUSelect` is 1, the muxes send the LSU signals to the bus. Otherwise, if `IFUSelect` is 1, the mux sends the IFU signals. If neither is selected, the bus receives 0, making it idle. The read data bus (`HRDATA`) goes to both the IFU and LSU. Only the LSU writes to external memory,

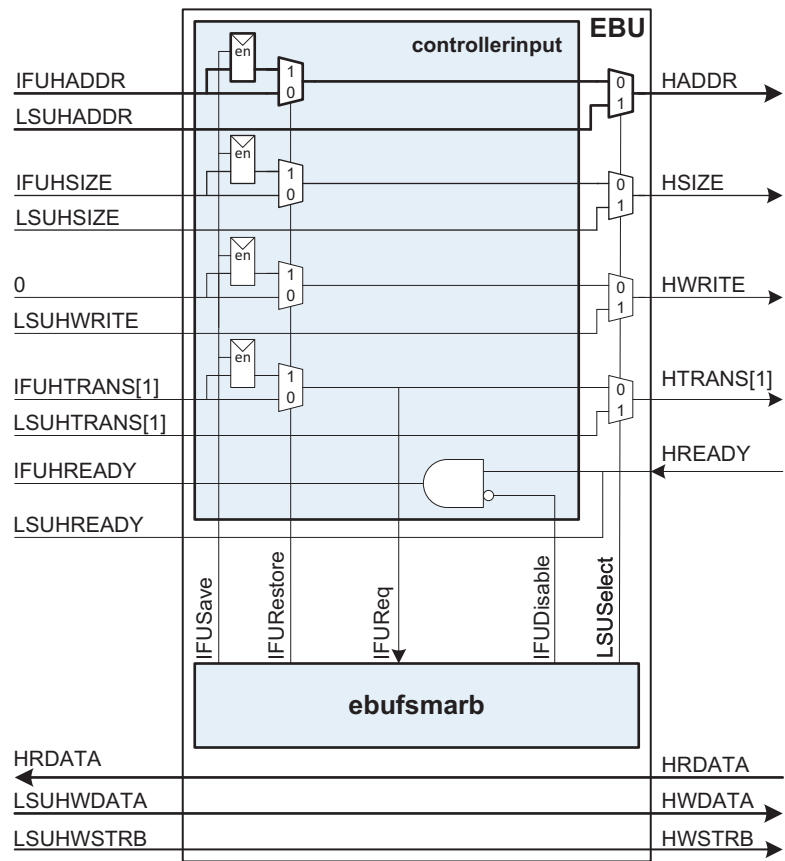


Fig. 9.25 External bus unit (EBU).

so **HWDATA** only connects to the LSU, and the IFU never asserts **IFUHWWRITE**, so that signal (just above **LSUHWWRITE**) is tied to 0.

The **ebufsmarb**, shown in Fig. 9.26, arbitrates when both the LSU and IFU make requests at the same time. It always gives priority to the LSU to prevent deadlocking the pipeline. Only two states are required, an Idle state to indicate either 1 or 0 requests are active and an Arbitrate state, which selects the LSU. It is a Mealy machine with outputs that are a function of the current state and inputs. When two requests occur in the Idle state, the FSM transitions to Arbitrate and selects the LSU. Because the IFU detects **HREADY** = 1, it assumes the Adr phase of the request was taken and moves on to the Data phase. So, the **ControllerInput** module must save the IFU's Adr phase **HADDR** and control signals for later. Data phase signals are inherently delayed, so they need not be stored and replayed. When the device completes the memory request, **HREADY** asserts, and the FSM returns

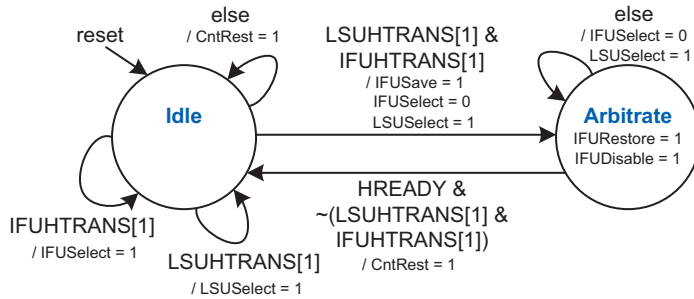


Fig. 9.26 EBU FSM arbiter (ebufsmarb), with the LSU having priority over the IFU.

to Idle. Upon returning to Idle, the IFU Adr phase is replayed by driving the stored IFU request onto the bus.

Fig. 9.27 shows the timing of a concurrent IFU and LSU read followed by two more IFU reads:

- ▶ In cycle 0, both the IFU and LSU make requests by driving `IFUHTRANS` and `LSUHTRANS` to `NONSEQ (2)`. The FSM chooses the LSU, raising `LSUSelect` to grant the LSU bus access and drive the LSU Adr phase signals onto the bus: `HADDR = LA1`. The IFU request is saved inside the `controllerinput` register with `IFUSave = 1`.
- ▶ In cycle 1, the FSM transitions to the Arbitrate state. `HREADY` is 1, indicating that the device has returned `HRDATA`. `LSUHREADY` asserts to capture `HRDATA` into the LSU. Meanwhile, the EBU selects the saved IFU request's Adr phase signals (`IA1`) to drive onto the AHB bus. `IFUHREADY` is 0 to continue stalling the IFU.
- ▶ In cycle 2, the FSM transitions to the IDLE state. The device responds to the IFU request with `HRDATA` for `IA1` and `HREADY = 1`.
- ▶ In cycles 3 and 4, two more requests originate from the IFU but, because the LSU is no longer making requests, no arbitration is needed.

When the data cache is added to the LSU, cache misses will generate AHB burst requests. Burst operations require the EBU to count each word of the transaction and ensure the lower priority IFU is not interrupted during a burst.

9.5.3 System-on-Chip

As shown in Fig. 9.20, the Wally SoC (wallypiplinedsoc) includes both the processor core and the uncore module that contains memory, peripherals, and an external memory interface connected to the core via the bus. Chapter 20 describes the peripherals further. The external memory interface could be an off-chip AHB device such as a DDR4 memory controller. The uncore module also contains address decoding and device multiplexing. Fig. 2.5 shows Wally's default memory map.

HRDATA and HWDATA bus widths are AHBW, which is set to the natural word size XLEN in Wally configurations. [Section 11.2.1](#) discusses virtual and physical addressing and explains that RISC-V uses 34-bit physical addresses for RV32 and 56-bit physical addresses for RV64. HADDR is a physical address of width PA_BITS (34 or 56). Recall that programs use 32- or 64-bit addresses. In systems with no MMU, RV32 prepends two leading 0s to create 34-bit addresses, and RV64 drops the 8 most significant bits to create 56-bit addresses. Systems with an MMU that supports virtual memory translate XLEN-bit virtual addresses to PA_BITS physical addresses (see Chapter 11).

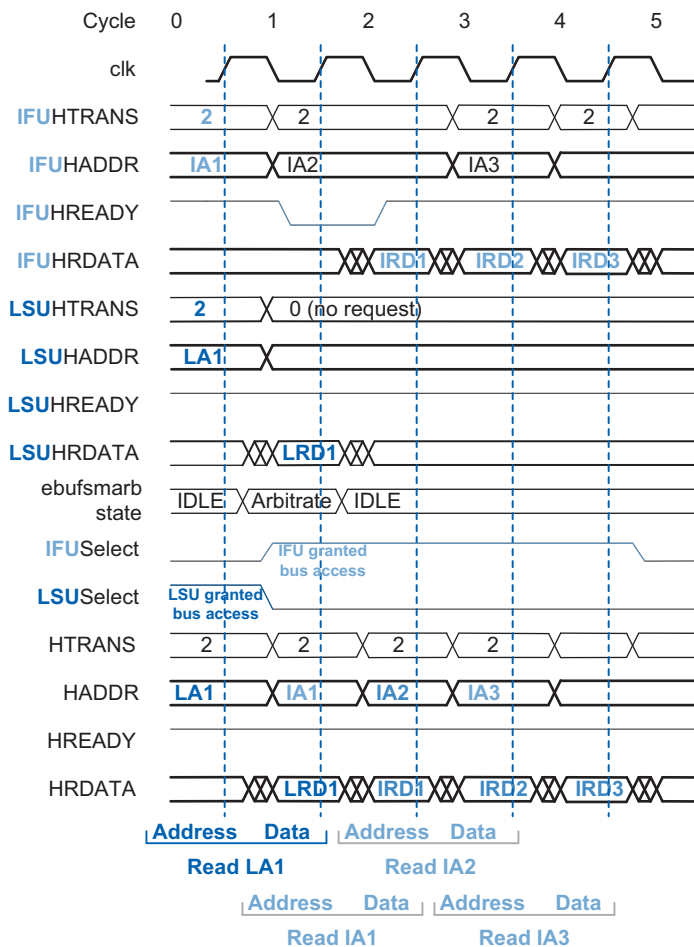


Fig. 9.27 Timing diagram for IFU and LSU arbitration of concurrent requests.

9.5.3.1 Address Decoding and Device Mux

As shown in [Fig. 9.12](#), AHB requires an address decoder and multiplexer to select and choose from among multiple devices. [Fig. 9.28](#) shows the uncore, which includes the address decoder, mux, and several peripherals, emphasizing the HADDR, HREADY, HREADYOUT, and HSEL connections.

Recall that AHB is pipelined into Adr and Data phases and that a transaction ends when the selected device asserts its HREADY signal in its Data phase. Because of pipelining, the HREADY signal in the Adr phase of one transaction comes from the Data phase of the previous transaction using the delayed HSEL signal, HSELD.

The decoder uses the configuration parameters `<DEVICE>_SUPPORTED`, `<DEVICE>_BASE`, and `<DEVICE>_RANGE` to determine which devices (e.g.,

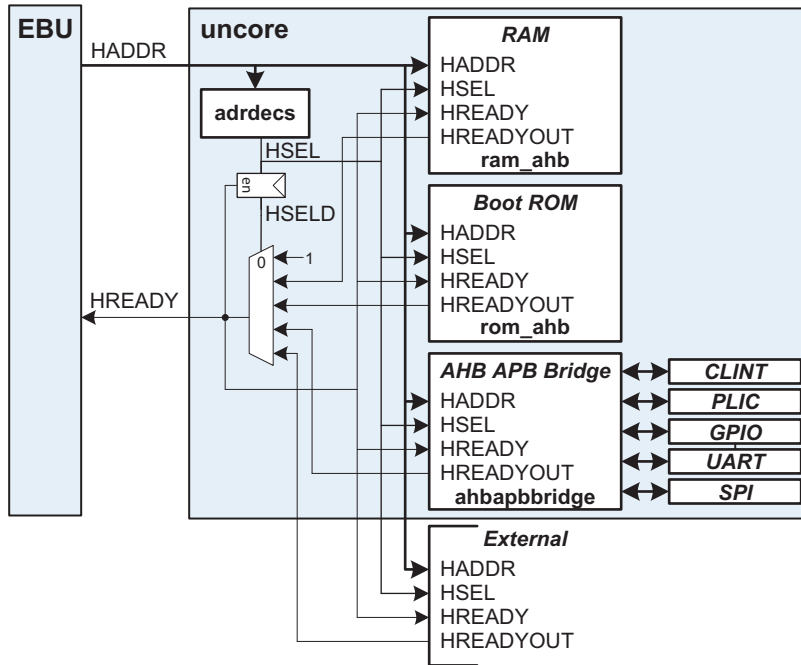


Fig. 9.28 Uncore interface from EBU to devices.

Boot ROM, uncore RAM, GPIO) are supported and where they appear in the memory map.

The address decoders module (*addrdecs*) receives *HADDR* in the *Adr* phase and produces a one-hot *HSEL* signal, with 1 bit per device. The module contains an address comparator for each device; an implementation is shown in [Section 11.4.1.1](#). *HSEL*[0] is asserted when *HADDR* doesn't match any device address. When the previous transaction's Data phase completes, as indicated by that device asserting *HREADY*, the current transaction enters the Data phase and latches *HSEL* to *HSELD*. The currently selected device reads or writes during the Data phase and raises its *HREADYOUT* when complete. The *HREADY* multiplexer picks *HREADYOUT* from the selected device (based on *HSELD*) and drives *HREADY* back to the EBU and to all the devices. If no device is selected, the bus is idle and immediately asserts *HREADY* to avoid locking up. The result multiplexer, not shown in this diagram, also selects *HRDATA* and *HRESP* from the selected device.

Example 9.2 ROM AND RAM READS WITH WAIT STATES

Suppose an AHB bus is initially idle, then reads from the ROM immediately followed by a read from the RAM, and then returns to idle. Suppose the ROM and RAM are slow, with 2 and 1 extra wait states, respectively. Let the ROM and RAM be devices

1 and 2. Sketch a timing diagram of HSEL, HSELD, HREADYOUT (from the ROM and RAM), and HREADY.

Solution: Fig. 9.29 shows a timing diagram.

- ▶ In cycle 0, the bus is idle: HSEL[0] = 1 to indicate no requests.
- ▶ In cycle 1, when the ROM transaction Adr phase begins, the controller asserts HSEL[1] to indicate a ROM access. HREADY was 1 because the bus was idle.
- ▶ In cycle 2, the ROM's Data phase begins but does not complete. In addition, the RAM Adr phase begins and asserts HSEL[2] to indicate a RAM access, but the access cannot begin until HREADY rises.
- ▶ In cycles 2–4, the slow ROM is completing its Data phase and the mux selects HREADY as the ROM's HREADYOUT using HSELD (a delayed version of HSEL), which indicates the peripheral in the Data phase.
- ▶ In cycle 4, the ROM finally asserts HREADYOUT (and, thus, HREADY).
- ▶ In cycle 5, the RAM's Data phase begins, but the slow RAM does not assert HREADYOUT until cycle 6. Meanwhile, the bus becomes idle and asserts HSEL[0] to indicate no new requests in cycles 5 and beyond.
- ▶ In cycles 5 and 6, the mux selects HREADY as the RAM's HREADYOUT. The RAM finally asserts HREADYOUT in cycle 6.

Upon reset, both HSEL[0] and HSELD[0] must be set to 1 to assert HREADY and, thus, prevent the bus from locking up.

9.5.4 Combined Bus and Tightly Integrated Memories

An embedded processor typically doesn't have caches but instead has a small IROM and DTIM in the core as well as a bus to peripherals or external memories. For example, the Wally rv32imc configuration is organized this way. The address decoder must distinguish between accesses to these TIMs versus bus accesses and use a mux to access them accordingly.

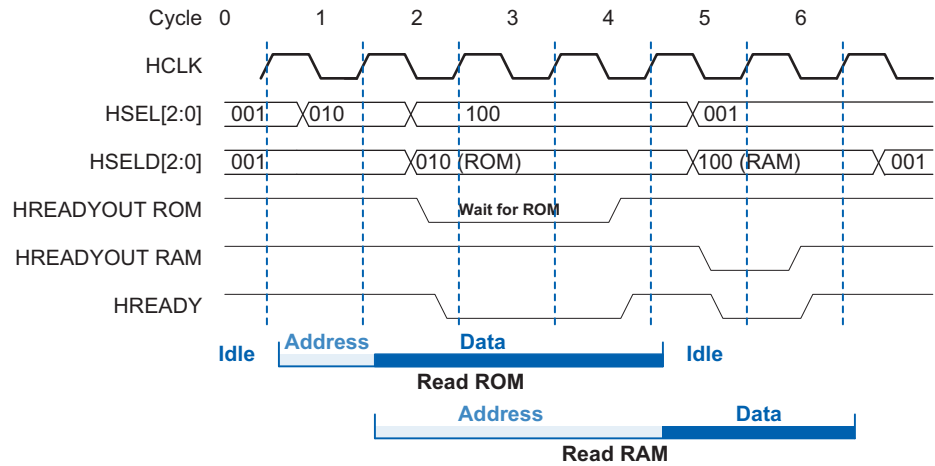


Fig. 9.29 AHB timing diagram of consecutive reads to different devices with wait states.

Wally handles this decoding using a PMA checker in the MMU, as described in Sections 11.4.1.1 and 12.3.1. Fig. 9.30 shows a diagram of the LSU with both a DTIM and AHB bus. The PMA checker's address decoder outputs `SeIDTIM`, which controls a multiplexer to select between the DTIM's read data (`DTIMReadDataM`) or the AHB's fetch buffer (`FetchBuffer`). `SeIDTIM` also demuxes the read/write signal `RWM` to either the DTIM or `busfsm`.

Fig. 9.31 abstracts the LSU to emphasize the data and address paths. The `subwordwrite`, `subwordread`, and `endianswap` modules are abbreviated as `SWW`, `SWR`, and `ES`. The `busfsm` is hidden in the `ahbinterface`. The processor stalls in the Memory stage during the multicycle bus access. The three (or more) AHB cycles (`Adr`, `Data`, `Mem3`) are not shown, and the core resumes in the Memory stage when the stall is released during `Mem3`.

The IFU uses the same approach to select between the IROM and the Bus, as shown in Fig. 9.32. A separate PMA checker in the IFU's MMU outputs `SeIIROM` to select the IROM or the Bus. Unlike the LSU's DTIM, the IROM only reads instructions. `PCNextF` directly drives the IROM read address port.

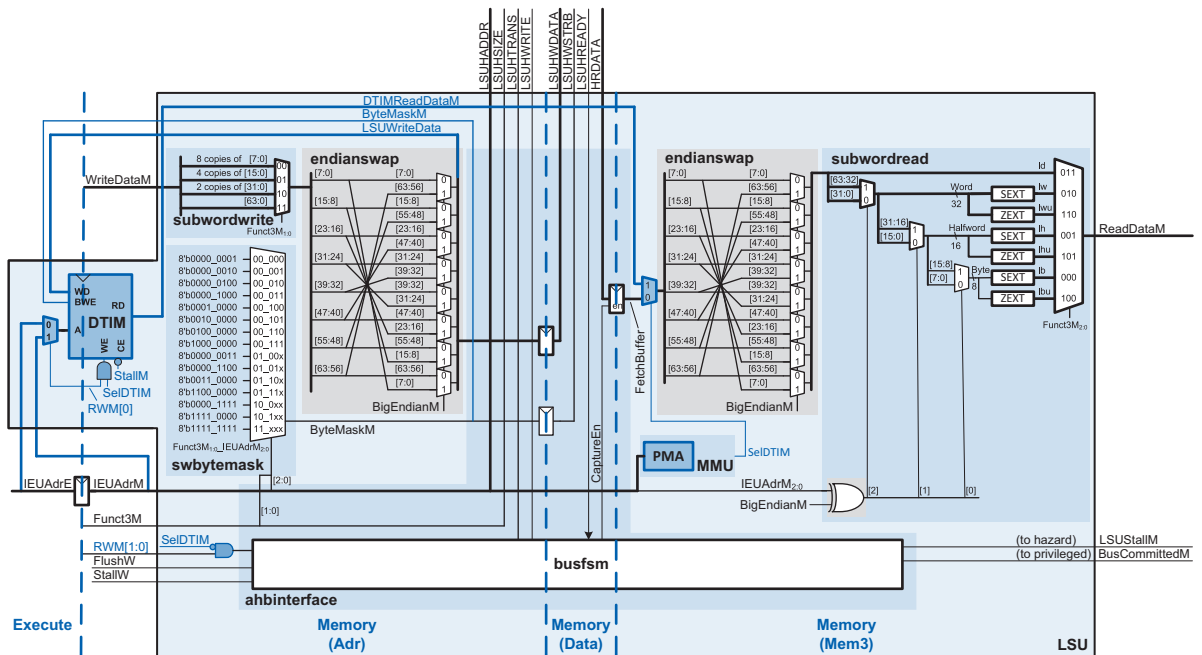


Fig. 9.30 LSU with DTIM and bus.

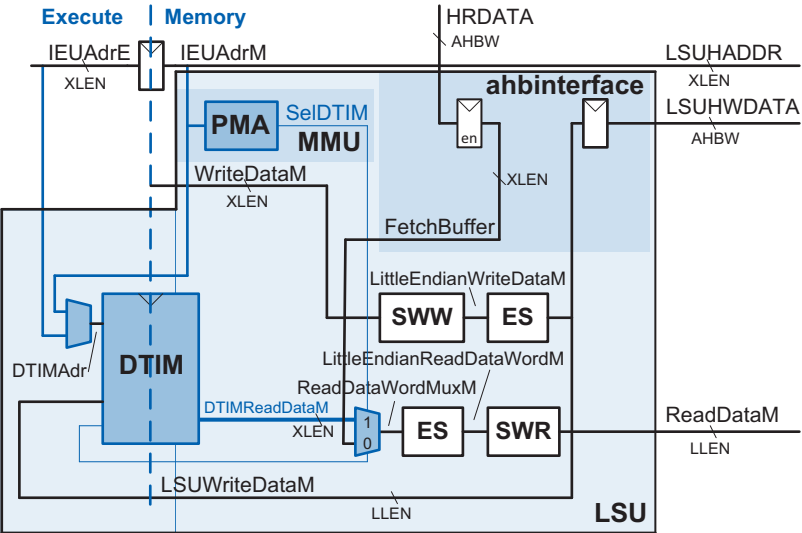


Fig. 9.31 Abstracted LSU with DTIM and bus.

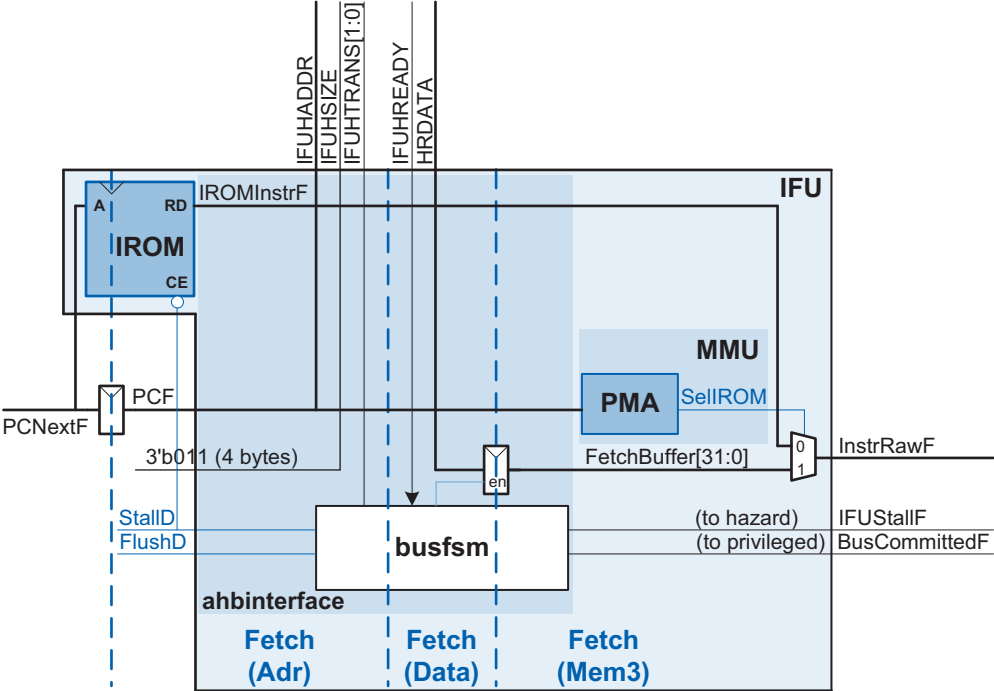


Fig. 9.32 IFU with IROM and bus.

9.5.5 Bus Hazards

Each AHB interface in the IFU and LSU takes multiple cycles to complete a bus transaction. The `busfsm` from Fig. 9.23 must *interlock* with the hazard unit to correctly stall the pipeline and delay interrupts occurring in the middle of a bus transaction.

Because each bus transaction takes multiple cycles, the LSU or IFU `busfsm` generates `BusStall` while in the midst of a transaction. Specifically, `BusStall` asserts when the bus transitions from the `Adr` phase into the `Data` phase and remains high until the `Data` phase completes. When `BusStall` is lowered, the hazard unit unstalls the pipeline, and the IFU or LSU enters its `Mem3` stage, producing a valid instruction `InstrRawF` or capturing the load result in `ReadDataM`.

In the LSU, `LSUStallM` = `BusStall` from its `busfsm`. Similarly, the IFU's `IFUStallF` = `BusStall` from its `busfsm`. The hazard unit raises `StallW` to stall the entire pipeline when either unit stalls, as shown in Fig. 9.33. Traps take priority and should flush the pipeline rather than processing the LSU/IFU stall. However, traps are only allowed at the start (i.e., in the `Adr` phase) of a bus transaction. The `CommittedM/F` signals, discussed later in this section, postpone interrupts arriving in the midst (in the `Data` phase) of a bus transaction.

Recall that when both the IFU and LSU need to access the bus, the EBU prioritizes the LSU. In that case, when the LSU completes its request and lowers `LSUStallM`, the pipeline remains stalled until the IFU's request is finished. Because `StallW` is still asserted, the LSU's `busfsm` remains in `Mem3` to prevent reissuing the bus request. After the IFU's `busfsm` transitions to `Mem3`, both units lower their `BusStall` signals and the pipeline resumes. Other hazards such as CSR writes may also stall the pipeline, keeping the `busfsm` in `Mem3` to prevent reissuing the bus request.

The `busfsm`'s `BusCommitted` signal delays pending interrupts until a bus transaction completes. `BusCommitted` must remain high while waiting in `Mem3` so that the interrupt does not occur until the next instruction. `BusCommitted` is zero only during the `Adr` phase, which allows interrupts to occur before either the LSU or IFU starts a bus access. If a trap occurs while the IFU or LSU is in the `Adr` phase, the bus transaction is canceled by driving `HTRANS` to 0.

An *interlock* disallows certain states to occur that would conflict. For example, an interrupt that occurs during a bus transaction must be delayed until after the bus transaction completes.

`LSUStallM` also asserts on cache and hardware page table walker stalls (see Section 12.4.1). Similarly, `IFUStallF` also asserts on cache stalls (see Section 13.3.5) and spills (see Section 14.3.3).

Recall that a stall at a given stage stalls all previous stages. Thus, a stall in the Writeback stage stalls the entire pipeline.

The bus stall could be expressed as follows:

```
StallWCause = (IFUStallF &
~FlushFCause | LSUStallM)
& ~FlushWCause
```

A Fetch flush (`FlushFCause`) has priority over an IFU stall (`IFUStallF`), and a Writeback flush (`FlushWCause`) has priority over an LSU stall. `FlushFCause` is not explicitly defined but is equivalent to `FlushDCause` because a flush in one stage also flushes all previous pipeline stages. Similarly, `FlushDCause` always asserts whenever `FlushWCause` asserts. Thus, the bus stall simplifies to the following, as also shown in Fig. 9.33:

```
StallWCause = (IFUStallF &
~FlushDCause | LSUStallM)
& ~FlushWCause
```

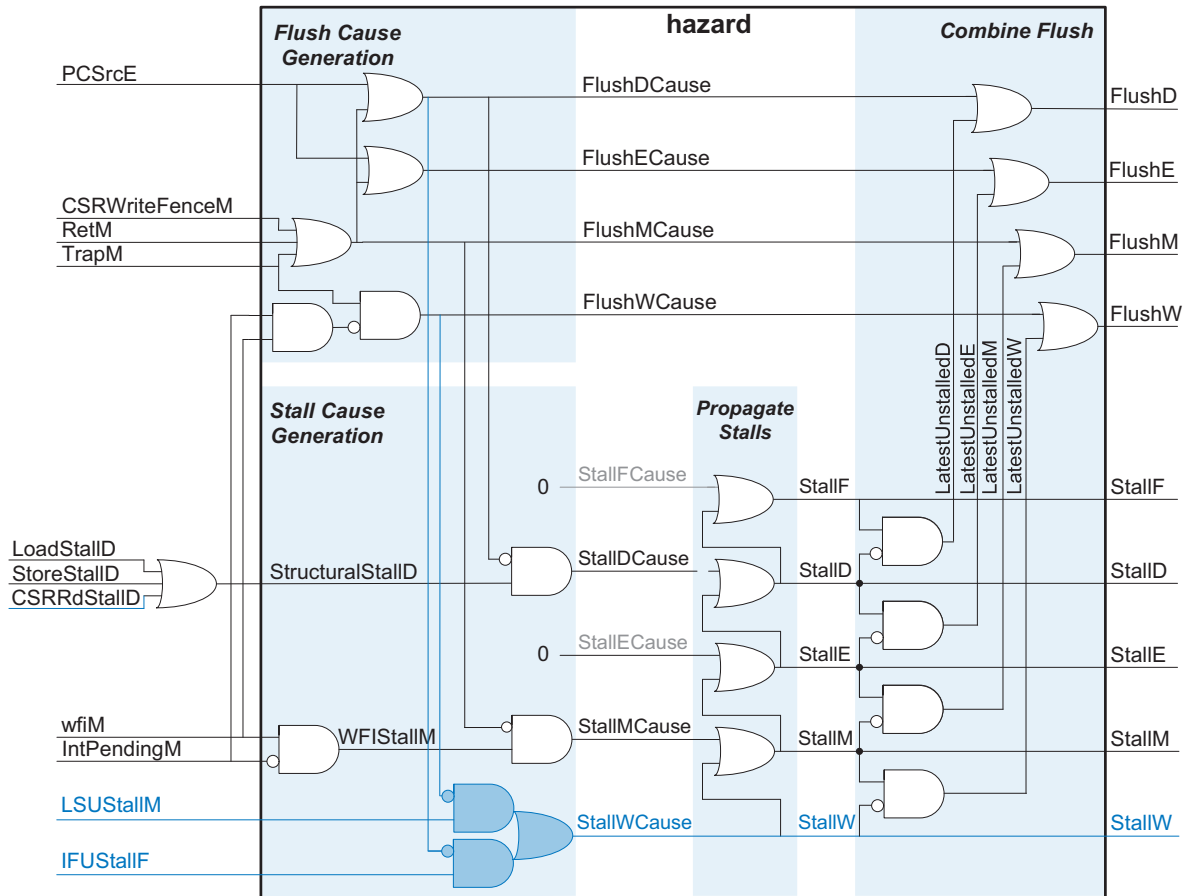


Fig. 9.33 Hazard unit, with changes due to the AHB highlighted in blue.

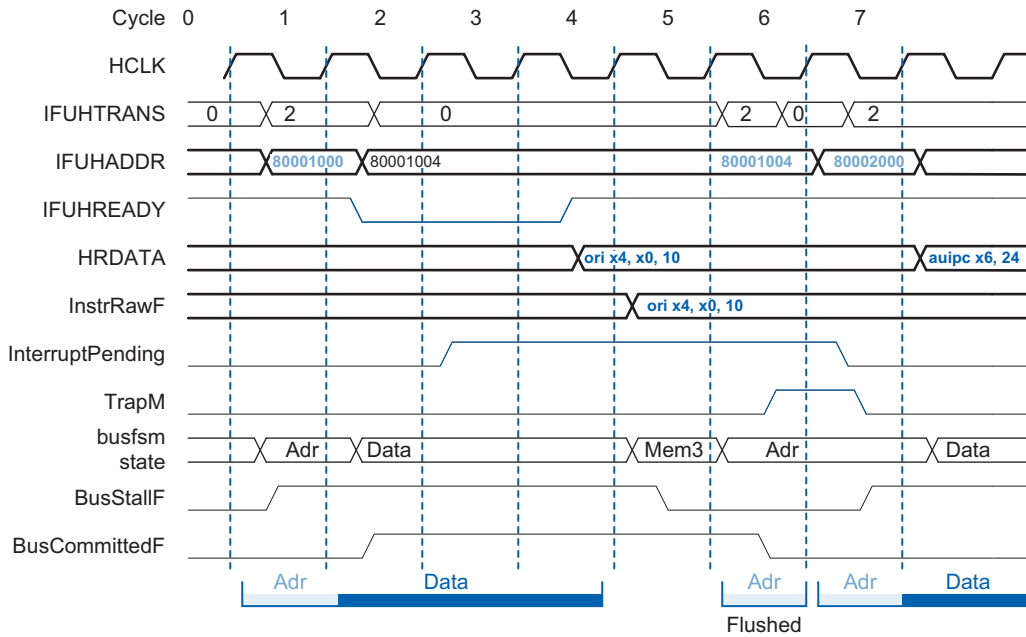


Fig. 9.34 Privileged unit interlock with the IFU's busfsm.

Fig. 9.34 shows how the IFU busfsm delays interrupts until the current bus transaction completes during the execution of Code Example 9.1:

- ▶ In cycle 1, the CPU begins fetching the `ori` instruction from 0x80001000 by entering the Adr phase of the bus.
- ▶ In cycle 2, the busfsm asserts `BusCommittedF` when entering the Data phase of the bus.
- ▶ In cycle 3, an interrupt becomes pending (`InterruptPending = 1`), but because `BusCommittedF = 1`, the interrupt is delayed.
- ▶ In cycle 5, the bus completes fetching `ori` (enters Mem3 phase).
- ▶ In cycle 6, the CPU begins fetching the `addi` instruction from 0x80001004 by entering the Adr phase of the bus. `BusCommittedF = 0` in the Adr phase, which allows the pending interrupt to be handled. `TrapM` flushes the pipeline, including the `addi` fetch that was in the Adr phase.
- ▶ In cycle 7, the trap handler begins fetching the `auipc` from 0x80002000, which enters the Adr phase of the bus.

The IFU could allow `TrapM` in cycle 5 to flush the completed fetch because the IFU only reads instructions. In contrast, when an instruction begins to write to memory, it must commit. Section 11.2.2.1 defines *nonidempotent* reads that have side effects and also must commit. For example, reading the Receive Buffer Register (RBR) in a 16550 UART also empties the buffer. Therefore, the RBR is nonidempotent because consecutive reads to the same address return different values.

The hazard unit could choose to stall just the Decode stage when IFUStallF asserts. However, this would mean that an exception could occur in the Memory stage while an instruction is being fetched. To ensure the exception is taken, the hazard unit would have to stall the Memory stage until the IFU lowers BusStall. Caches complicate this issue further. Thus, Wally instead stalls the entire pipeline during both IFU and LSU bus accesses.

Code Example 9.1 DELAY INTERRUPTS DURING BUS TRANSACTION

```
80001000:    ori  x4, x0, 10
80001004:    addi x5, x4, 10
. . .
TrapHandler:
80002000:    auipc x6, 24
```

In this chapter, CommittedF = BusCommittedF. Later chapters will add other cases when CommittedF asserts. Similar to the IFU, the LSU asserts CommittedM when its BusCommittedM asserts. The trap module in the privileged unit asserts Committed when either the LSU's CommittedM or the IFU's CommittedF asserts. Committed delays interrupts until the instruction completes. Fig. 9.35 shows the trap unit from Fig. 8.14 with the new Committed logic highlighted in blue.

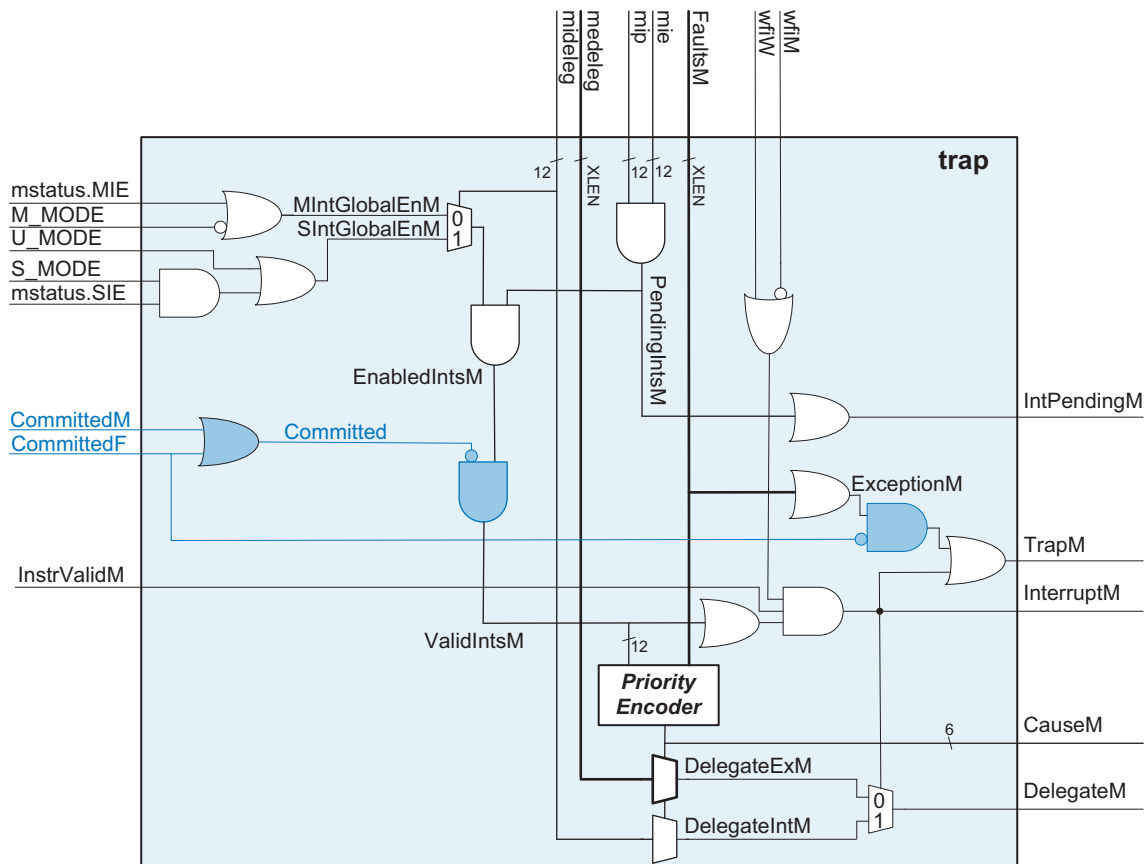


Fig. 9.35 trap unit delays pending interrupts while committed instructions complete.

9.6 SUMMARY

A bus interface provides a standardized way for a system to communicate with external memory or peripherals. Modern on-chip buses normally use synchronous point-to-point links between a controller and one or more peripherals. The controller generates a clock signal and sends an address and some control information to each peripheral. During a write operation, the controller also sends the write data. During a read operation, the peripheral responds with the read data. The peripheral may insert wait states if it needs longer to respond. Because of the point-to-point links, the controller uses a read multiplexer to choose the read data from the selected peripheral. High-performance buses are normally pipelined to overlap consecutive accesses. They support features such as burst and subword transfers and bus locking for atomic operations. Many bus protocols are used in industry, so a bridge is often needed to adapt between protocols.

Wally uses an EBU to connect the IFU and LSU to a single AHB bus so they can access peripherals and external memory, beyond the capacity of an IROM or DTIM. When both the IFU and LSU want to use the bus at the same time, the EBU grants priority to the LSU so the pipeline can make forward progress. Bus transfers take multiple cycles and must finish once they have started, so the EBU interlocks with the hazard and privileged units to stall the pipeline and prevent interrupts during in-flight transfers until the transfer completes.

Caches

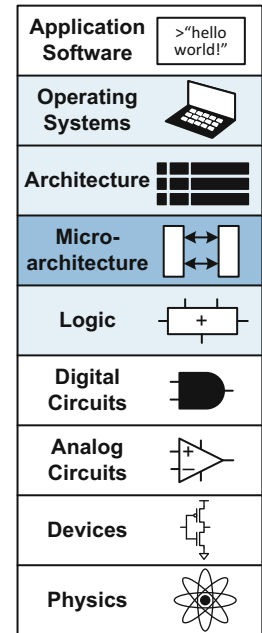
10

- 10.1 Principles
- 10.2 RISC-V Practices
- 10.3 Test Plan
- 10.4 Wally Implementation
- 10.5 Summary

An ideal processor would have a large, fast, cheap memory. Unfortunately, the memory available on-chip is usually static random access memory (SRAM), which is fast but expensive per bit and has relatively low density. Off-chip dynamic random access memory (DRAM) provides gigabytes of memory at low cost and high density but is significantly slower than a CPU (central processing unit). Processors provide the illusion of a large, fast, cheap memory by storing most data in the large, cheap DRAM but *caching* frequently accessed information in the smaller but fast on-chip SRAM. Pipelined CPU designs typically use separate first-level (L1) caches for instructions and data so that both can be accessed concurrently without stalling the pipeline. The *instruction cache* (I\$) is generally only readable, but the *data cache* (D\$) must support both reads and writes. In systems with multiple cache levels, these are referred to as *level 1* caches (L1I\$ and L1D\$). Larger systems usually also have a *second-level cache* (L2\$), with larger capacity but longer access times, that holds both instructions and data. Multicore systems often have an even larger *third-level cache* (L3\$) shared across cores. Some server SoCs, such as IBM Power 8 and 9, add an off chip L4\$ using embedded DRAM (eDRAM) that is still faster than going to the main system DRAM.

The levels of cache and memory in the system are called the *memory hierarchy*. Fig. 10.1 shows a memory hierarchy with three levels of cache, a main memory, and *virtual memory*. Virtual memory uses a slow but inexpensive hard drive to hold more information than fits in DRAM.

This chapter begins with a short review of cache principles. Then it examines the cache architecture used in the Wally instruction and data caches. Section 9.5 described how the processor accesses main memory using the system bus. Chapter 11 discusses virtual memory, and Chapters 12 and 13 discuss the rest of the load/store unit (LSU) and instruction fetch unit (IFU), which contain these caches.



The *lowest* level of the memory hierarchy is an L1\$; the *higher* levels are the L2\$ and L3\$ followed by main memory and then virtual memory.

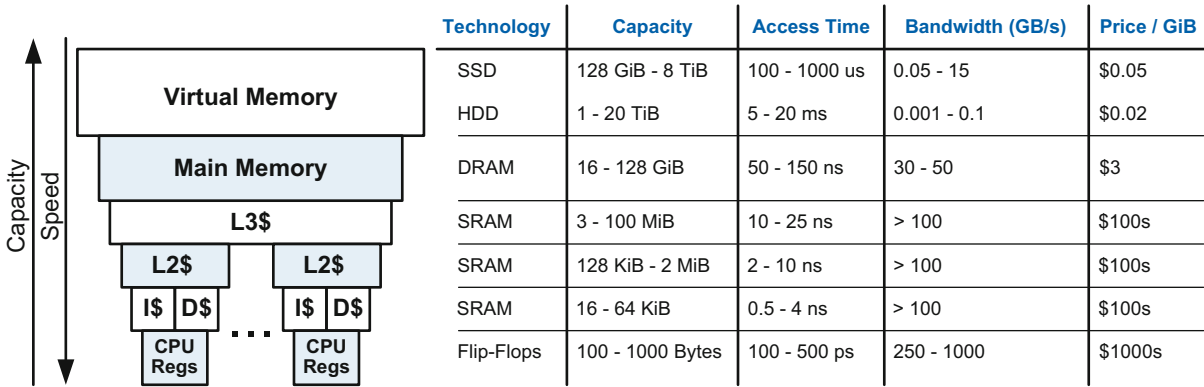


Fig. 10.1 Memory hierarchy.

See [Hennessy19] for a more detailed introduction to caches.

In March 2025 AMD released the Ryzen 9 9950X3D with a massive 128 MiB L3\$ using a 3D stacked memory. Two dice, the 9950X with a 64 MiB L3\$ and an extra 64 MiB L3\$ cache, are stacked and connected with through-silicon vias (TSV). Previous generations of X3D chips, the 5800X3D and 7800X3D stacked the cache die on top of the compute die to simplify power delivery. However, the 9950X3D moved the cache die below the compute die to improve device cooling achieving higher clock frequencies with both designs boosting to 5.7 GHz. The 9950X3D has 20 – 30 % higher frames rates than the 9950X in many video games, which highlights the benefits of a large cache.

10.1 PRINCIPLES

Caches improve the effective latency and bandwidth of the overall memory system by taking advantage of repetitive patterns in program behavior. For some data-intensive applications, large caches improve performance more than CPU clock speed does.

A memory access *hits* in the cache if it finds the information there. If it *misses* the cache, it must go to the next-level cache or to main memory to find the information, which increases the access latency.

This section first examines how caches affect performance, then explores how caches exploit *locality* to increase hit rates, and finally describes principles to build a cache.

10.1.1 Cache Performance

An ideal memory system would introduce no delays, but a practical system slows down when the cache misses and needs to wait for data from the next level of the memory hierarchy. The *miss rate* is the fraction of cache accesses that miss. The *miss penalty* is the average number of cycles required to handle the miss. In a simple pipelined processor, misses to the L1\$ stall the pipeline until it can fetch the information from a higher-level cache or main memory. The number of stall cycles is:

Memory stall cycles = Number of instructions $\times$

$$(\text{Memory accesses / instruction}) \times \text{Miss rate} \times \text{Miss penalty} \quad (10.1)$$

Example 10.1 EFFECT OF CACHES ON CPI AND PERFORMANCE

Suppose a pipelined processor has a CPI (cycles per instruction) of 1.2 when all memory accesses hit in the cache. 30% of instructions are loads or stores. The miss rate is 2%, and the miss penalty is 40 cycles. How much slower is the processor because of cache misses? How much slower would it be if there were no cache?

Solution: The program has 1.3 memory accesses/instruction (1 instruction fetch and 0.3 loads/stores). So, the number of memory stall CPI, which is Equation 10.1 divided by the number of instructions, is $1.3 \times 0.02 \times 40 = 1.04$. This adds to CPI, so overall CPI is $1.2 + 1.04 = 2.24$. This increases program execution time by $(2.24 - 1.2)/1.2 = 87\%$. If there were no cache, every memory access would incur a miss penalty, and CPI would be $1.2 + (1.3 \times 1 \times 40) = 53.2$, or about 24x ($53.2/2.24$) slower.

A measure of overall memory performance is the *average memory access time (AMAT)*, which is defined as the average time memory operations take to complete. AMAT is:

$$\text{AMAT} = A_1 + \text{MR}_1(A_2 + \text{MR}_2(A_3 + \text{MR}_3 \dots)) \quad (10.2)$$

A_i is access time, and MR_i is miss rate for level i of the memory hierarchy. The miss penalty for level i is the AMAT starting at level $i+1$.

Example 10.2 AMAT

If 95% of instructions hit in the L1\$, which has a 1-cycle access time and 15-cycle miss penalty, what is the AMAT?

Solution: $\text{AMAT} = 1 + 0.05 \times 15 = 1.75$ cycles.

Cache misses are classified as *compulsory*, *capacity*, or *conflict*. The first time data is accessed, a compulsory miss occurs because the data must be loaded into the cache. Capacity misses occur when the cache is too small to hold the concurrently used data, which is called the *working set*. Conflict misses occur when too many addresses map to the same set (location in the cache), causing active cache lines to be *evicted*. Section 10.1.2 addresses design methods to reduce each class of miss.

The miss penalty can be reduced by using a faster, more expensive memory or by adding intermediate levels of cache. The miss penalty increases as the processor cycle time gets faster relative to the main memory.

Example 10.3 AMAT AND CPI

A 5-stage pipelined CPU has 2 levels of caches and DRAM main memory. The L1I\$ has a hit rate of 98% and a 1-cycle access time. The L1D\$ behaves the same except the hit rate is 96% and The L2\$ has a hit rate of 80% (for accesses that missed the L1\$) and a 3-cycle access time. DRAM has a 30-cycle access time. What is the AMAT for loads/stores and for instructions? Suppose the running application has 40% loads/stores, and all other instructions take a single cycle to execute if the L1I\$ hits. Use AMAT to estimate overall CPI.

Solution: AMAT for fetching instructions is $0.98 \times 1 + 0.02(3 + 0.2(30)) = 1.16$. AMAT for accessing data is $0.96 \times 1 + 0.04(3 + 0.2(30)) = 1.32$. The CPI of a non-load/store is the instruction fetch AMAT, 1.16. To compute the load/store CPI, add the two AMATs and subtract 1 to account for the overlap of one instruction fetched while 1 load or store is read or written. $\text{CPI}_{\text{load/store}} = 1.16 + 1.32 - 1 = 1.48$. So, $\text{CPI} = 0.6 \times 1.16 + 0.4 \times 1.48 = 1.29$ cycles.

A mainstream PC in 2022 typically has 16–64 GiB of DDR4 DRAM costing \$3–\$6/GiB with a bandwidth of about 19.2–35.2 GB/s and a latency of nearly 100 ns to access a random word.

A Core i7 12900K Alder Lake processor costs about \$600 and has 30 MiB of SRAM in the L3\$. The L3\$ has a 17.6 ns latency at the 3.2-GHz base core frequency, and the 48 KiB L1\$ has a latency of under 1 ns [RyzenVi9]. The L1 caches deliver about 1 TB/s (1000 GB/s) of bandwidth. Fig. 1.24 shows a Raptor Lake die with the L2\$ and L3\$ highlighted.

Out-of-order processors can hide part of the miss penalty by executing other instructions while waiting, though data dependencies in the program limit this parallelism. Such caches are called *nonblocking* because the CPU is not stalled on a miss. Multithreaded processors hide the miss penalty by switching to another thread when a cache misses.

Memory access time is an increasing concern as processor clock rate increases. If main memory takes 50 ns to return data, a processor running at 20 MHz ($= 1/50$ ns) takes 1 processor clock cycle to access main memory, whereas a processor running at 2 GHz takes 100 clock cycles.

The information held in a cache is generically referred to as data, even though for an I\$ the information held is instructions.

10.1.2 Locality

Caches reduce the AMAT for programs that reuse the same or adjacent data. [Code Example 10.1](#) shows a `for` loop that adds the loop index to each element in an array. This code exhibits both *temporal* and *spatial* locality, which are defined as follows:

- ▶ **Temporal locality:** Recently accessed memory is likely to be used again.
- ▶ **Spatial locality:** Nearby memory addresses are likely to be used.

In [Code Example 10.1](#), the I\$ sees temporal locality because the same instructions (in the loop) are executed repeatedly and spatial locality because consecutive instructions are executed. The D\$ sees temporal locality because each array element is read and then written. It also sees spatial locality because consecutive elements are accessed.

Code Example 10.1 LOOP DEMONSTRATING LOCALITY

```
for (int index = 0; index < 100; index++)
    array[index] = array[index] + index;
```

Some texts refer to a cache line as a cache *block*.

Caches exploit both types of localities by holding a copy of recently accessed instructions and data. When the CPU reads or writes an address, it first checks the cache; if the instruction or data is present, the CPU accesses it immediately. However, if the item is missing, the cache fetches this item and nearby consecutive items into a *cache line*. Spatial locality tells us the nearby addresses have a high probability of being used soon, so it is beneficial to fetch data from these nearby addresses into the cache. Each access to the next-level cache incurs some latency independent of data size plus additional time to transfer each word in the line, depending on line size. So, making the line too long may hurt performance.

10.1.3 Direct-Mapped Cache

The simplest kind of cache is a *direct-mapped* cache, in which the least significant bits of the address specify the location of an item in the cache. [Fig. 10.2](#) shows a 4-KiB direct-mapped cache organized as 128 *sets*, each containing a single 256-bit (32-byte) cache line and an associated *tag* holding the upper bits of the line's address. Suppose addresses are 64 bits. Bits 11:5 of the address specify the set and bits 4:0 specify the bytes within the set. If the cache reads 8-byte *double words* aligned to 8-byte boundaries, bits 2:0 are always 000, and bits 4:3 specify one of the four 8-byte words in the line, called the *word offset*, indicated by “ww” in the address of [Fig. 10.2](#).

The word offset is also referred to as the *block offset*.

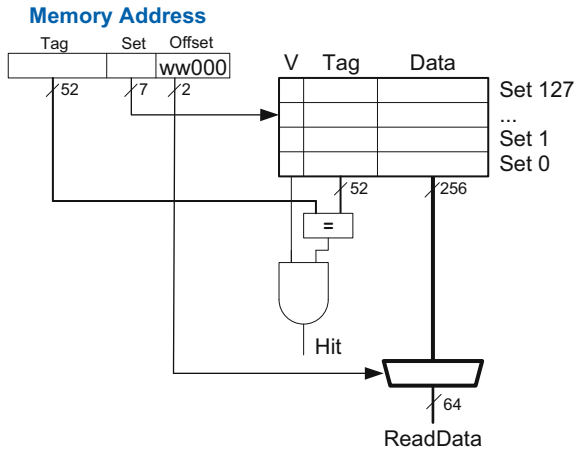


Fig. 10.2 4-KiB direct-mapped cache with 128 256-bit cache lines.

Any memory address with the same 11:5 bits will map to the same set. Therefore, the cache must store a *tag* that indicates which address is in the set, as well as a valid (*V*) bit indicating that the set contains an item at all. During a memory read, the cache compares the tag (upper bits of the requested address) with the tag in the set. Address bits 11:5 determine the set. If the line is valid ($V = 1$) and the tags match, the cache *hits* and returns the selected data word from the cache line. The word offset, address bits 4:3, determines the word within the line. Subword reads (1b, 1bu, 1h, 1hu, 1w, 1wu) would then use bits 2:0, called the *byte offset*, to select the byte(s) within the word.

Because multiple addresses map to the same set, the cache must hold both the data and its address. However, the set and word offset bits already encode the lower bits of the address, so only the remaining upper bits, the tag, must be held in the cache, not the entire address.

Example 10.4 DETERMINE SET AND TAG

In which set would the cache from Fig. 10.2 hold an 8-byte double-word located at memory address 0x2138? What would the tag be?

Solution: Fig. 10.3 shows the cache organization. With 128 lines, this cache needs $\log_2 128 = 7$ bits of the address to indicate the set. The least significant 3 bits are 000 for each 8-byte word, and the next 2 bits indicate which of the four words (00–11) is requested within the 32-byte line. The next 7 bits indicate the set. The most significant bits are the tag. In binary, the memory address 0x2138 is 0010 **0001 001** 11 000, with the set bits highlighted in blue. Hence, the 8-byte word is stored in set 9 with a tag of 2, indicated by the bits to the left of the set bits. The stored word is the most-significant 8-byte word of the four in the set, as indicated by the bolded “11” in the address.

The size or capacity of a cache refers to the number of data bits and does not count tag or valid bits.

Multiple addresses map to the same cache set. Using the cache from Example 10.4, an 8-byte word at address 0x4120 (0100 **0001 001** 00 000) would also be held in set 9 but with a tag of 4. A *conflict* occurs when the cache accesses two different addresses with the same set but different tags. The direct-mapped cache must evict one line to make room for the other. This increases the cache miss rate.

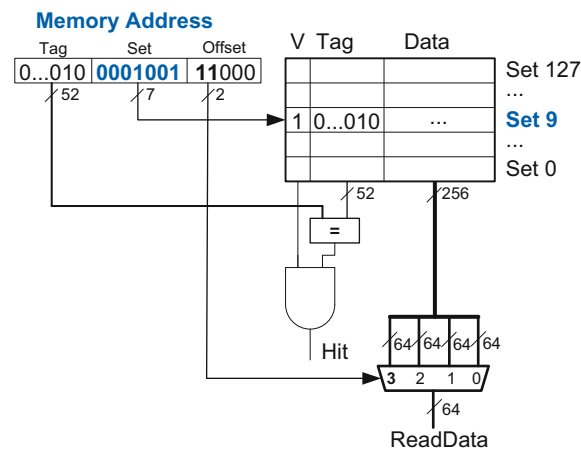


Fig. 10.3 Mapping of address 0x2138 to a 4-KiB direct-mapped cache with 32-byte lines.

If the tag does not match or the set is invalid, the cache *misses*. The cache controller must send a request to the next level of the memory hierarchy to fetch the line. The cache then writes the new data along with the tag of the fetched line and sets the valid bit to 1. If the set previously contained a valid line, the cache must first *evict* the old line to make room for the new one. Read-only caches simply replace the old line with the new one. Writable caches may first have to write a line that has been changed (called a *dirty line*) to the higher level of the memory hierarchy as discussed in Section 10.1.4. Each cache line adds a *dirty bit* (*D*) to indicate a dirty line. The valid and dirty bits are collectively called *status bits*.

10.1.4 Cache Writes

Instruction caches only need to read data, but data caches also need to write data from the CPU. This involves adding a write port, supporting partial line writes to a multiword cache line, and updating the written data in higher levels of the memory hierarchy.

Fig. 10.4 shows the interface of a generic read/write cache. The write port includes WriteData and read/write enable (RW) inputs as well as a ByteMask input that indicates which bytes in the word should be written.

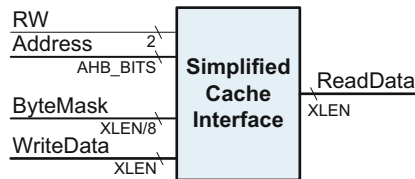


Fig. 10.4 Read/write cache interface.

For example, `ByteMask` is `0b11111111` for an `sd` instruction that writes all 8 bytes of a double-word, `0b00000001` for a `sb` instruction writing the least significant byte, or `0b11000000` for a `sh` instruction writing the most significant half-word. Read and write enables are grouped together in the `RW[1:0]` signal; read enable is the msb, `RW[1]`, and write enable is the lsb, `RW[0]`.

A store begins the same way as a cache read: the cache reads the tag and valid fields of the selected set to detect a hit. On a hit, `WriteData` is written to the correct word offset within the line, and the dirty bit (`D`) for that line is set so the cache knows that the line will need to be written back to the next level of the memory hierarchy when it is evicted. The valid and dirty bits are collectively called *status bits*.

Fig. 10.5 shows an example of writing `0xAABBCCDD` to address `0x14`. It assumes a 16-bit address, `XLEN = 32` bits, 16-byte (4-word) cache lines, and 64 sets. First, `WriteData` is replicated four times in the line-length (4-word) signal `LineWriteData`. A demultiplexer shifts `ByteMask` (`0xF`) left by the word offset (`0b01`) into the correct position within `LineByteMask` (`0x00F0`) and zeros the other bits. `LineByteMask` controls which bytes are written within the line, so only bytes 7:4 are updated in the cache line. The dirty bit is also set.

If a write misses, the cache must first fetch the line, as is done for a read miss. After the data arrives from the next level of the memory hierarchy, the cache then merges the new write data into the line at the correct word offset before writing the line into the cache memory. On read or write misses, `LineByteMask` is `0xFFFF` because all bytes need to be written.

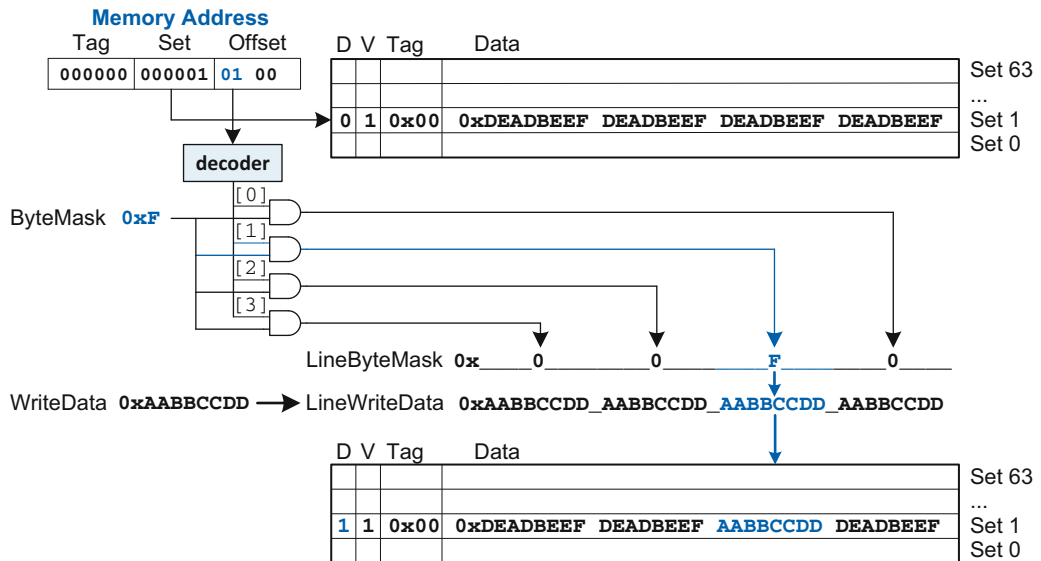


Fig. 10.5 Direct-mapped cache write.

The memory hierarchy becomes inconsistent when data is written to a cache but not to the higher levels of the memory hierarchy. The two strategies for handling this inconsistency are called *write-through* and *write-back*.

In a write-through cache, each write is also sent to the next level(s) of the memory hierarchy, so all memories remain consistent. This seriously limits write bandwidth and costs substantial power, so it is only viable for simple low-performance systems.

Practical processors now use write-back caches, which use the dirty bit to indicate that the cache line has changed and, thus, is inconsistent with the next level of the memory hierarchy. When a line is initially fetched on a read miss, the dirty bit is cleared. Cache writes set the dirty bit. When a dirty line is *evicted*, it must be written back to the next level of the hierarchy.

Multicore systems add another degree of complexity because different cores may have different dirty lines in their local caches. The programmer expects memory *coherency*, meaning that all cores will eventually see the value most recently stored to an address. Thus, the memory system uses a *coherency protocol* such as snooping or directories to maintain coherency between the caches [Hennessy19].

10.1.5 Associative Caches

With direct-mapped caches, conflicting addresses compete for the same set. This can lead to conflict misses, as shown in [Code Example 10.2](#), which alternately accesses two 8-KiB arrays. If run on a direct-mapped 8-KiB cache, every access will miss in the cache because each `a[i]` and `b[i]` map to the same set. Thus, the cache has a 100% miss rate and does not benefit from spatial locality. In fact, the system will run more slowly than it would without a cache because it wastes time fetching each cache line and then immediately evicting it. This is called *thrashing*.

Code Example 10.2 LOOP CAUSING THRASHING

```
int64_t a[1024], b[1024];
int i;
for (i = 0; i < 1024; i++)
    b[i] = a[i];
```

An *N-way set associative cache* (or *associative cache*, for brevity) reduces conflicts by providing *N* separate *ways* in each set. A line can be placed in any of the ways. Each way has its own tag, valid, and dirty bits. [Fig. 10.6](#) shows an 8-KiB 2-way associate cache with 128 sets, each containing 256 bits (32 bytes = 4 8-byte words) of data in each way.

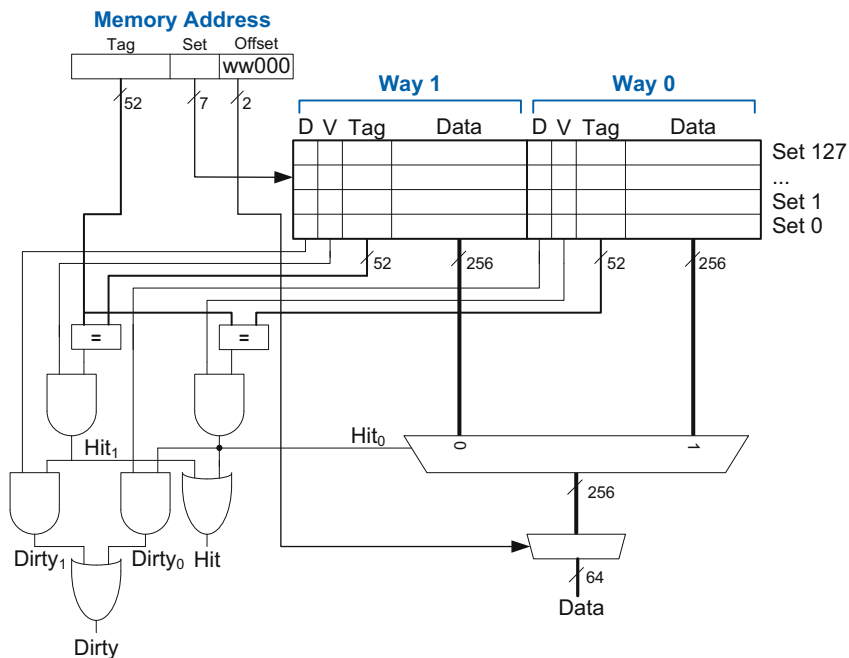


Fig. 10.6 Associative cache read access.

Like the direct-mapped cache discussed earlier, this cache reads 8-byte words, so the three least significant bits of the address are always 0. The next two bits of the address (ww) specify one of the four words in the line. The next seven bits indicate the set, and the upper bits are compared against the tag in both ways. If either way hits, the matching way reads the data.

When an associative cache misses and all ways are valid in the set, the cache must select a *victim* way to evict to make room for the new line. The cache *replacement policy*, discussed in [Section 10.1.7](#), describes how to pick the victim way.

Considering [Code Example 10.2](#) again with the 2-way associative cache, no conflict misses occur anymore because `a[i]` and `b[i]` each reside in a different way. The cache misses on the first word in each line but then hits on the next three words because of spatial locality. Even though the program has no temporal locality (repeated access to the same data), it still has a 75% hit rate because of spatial locality.

A direct-mapped cache can be viewed as a 1-way associative cache. The opposite extreme is a *fully associative* cache, which has a single set containing all the lines in different ways. Fully associative caches never have conflict misses, although they still have capacity misses. The fully associative hit comparators and hit multiplexer are expensive and grow with the size of the cache, so full associativity is only practical for

modest-sized caches. Translation lookaside buffers (TLBs), described in Section 11.4.1.3, are often fully associative to achieve the best hit rate for a limited capacity.

10.1.6 Cache Design Tradeoffs

Cache designers must pick the line size, associativity, and total capacity to achieve the best performance given a limited area and power budget.

Section 10.1.2 describes how longer cache lines improve hit rate due to spatial locality but take longer to fill or evict. For a given cache size, very long lines also increase the miss rate because fewer lines are available altogether, increasing conflict misses. Fig. 10.7 shows these tradeoffs for various cache sizes (4–256 KiB) when running the SPEC92 benchmark [SPEC92].

Fig. 10.8 plots cache miss rate versus size and associativity when running the SPEC2000 benchmark [SPEC2000]. Capacity misses dominate, especially for typical 16- to 32-KiB L1 caches. Capacity misses can be alleviated by providing a larger L2\$. Conflict misses can be significant in direct-mapped caches but become rare in caches with at least 4-way associativity. As we will see in Section 11.1.1.7, in systems with virtual memory, such associativity is also used to ensure that each way is no larger than the page size, which moves address translation off the critical path. In nontrivial programs, compulsory misses are negligible because they only occur at the start of a program.

Higher-level caches may be *inclusive* (containing all contents of the lower-level cache), *exclusive* (containing none), or *non-inclusive non-exclusive* (NINE, possibly containing some). Exclusive caches make better use of limited cache space, but inclusive top-level caches are needed to maintain cache coherence in multiprocessor systems. To provide significant miss rate reduction, the next-level cache should be significantly (at least 4-8

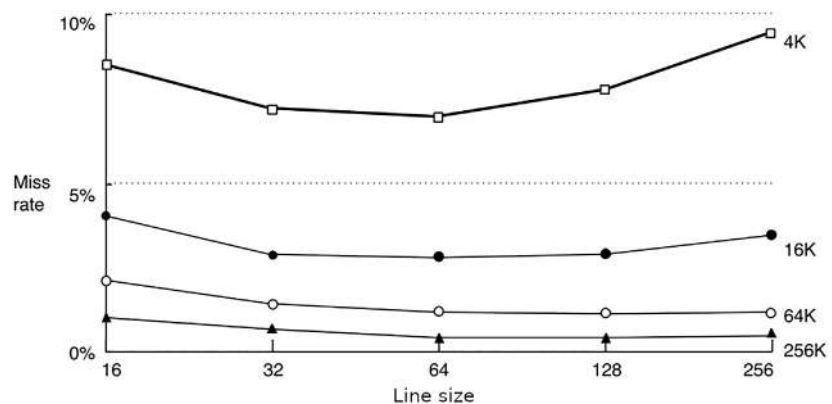


Fig. 10.7 Cache miss rate vs. line size for various cache sizes. (Figure courtesy of Elsevier [Hennessy19, Figure B.10]. Adapted and reprinted with permission.)

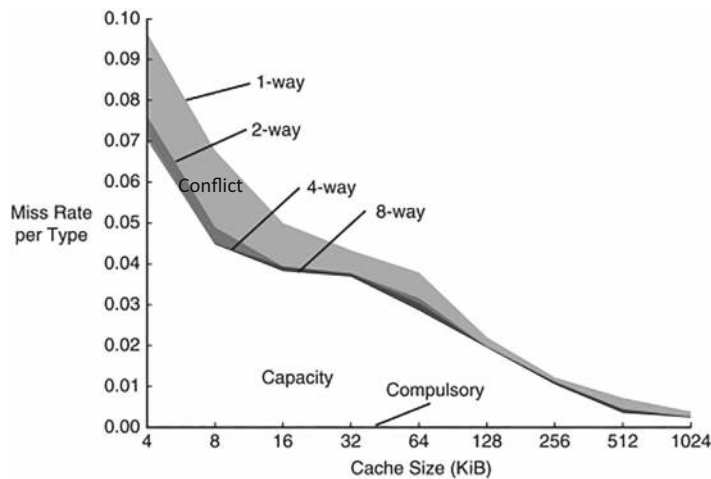


Fig. 10.8 Cache miss vs. size and associativity. (Figure courtesy of Elsevier [Hennessy19, Figure B.9]. Adapted and reprinted with permission.)

times) bigger than the one below it. So, in a system with 16-KiB L1I\$ and L1D\$, the combined L2\$ should be at least $8(16 + 16) = 256$ KiB. L2 caches with up to several MiB are sometimes used, but they can have long access times when they get too big. Hence, the system may add an L3\$, commonly with 4- to 64-MiB of capacity. The highest-level cache is shared across all cores of a multicore system to support shared-memory communication between cores.

10.1.7 Replacement Policy

As mentioned in [Section 10.1.5](#), an associative cache needs a replacement policy to select a victim way to evict during a cache miss when all ways are in use (all valid bits in the set are 1). An ideal replacement policy selects the way with the lowest probability of being used in the near future. However, program behavior is not known ahead of time, so we use past behavior to predict the future. The *least recently used* (LRU) replacement policy records the order in which the ways are accessed (read or written) and selects the oldest accessed way as the victim.

In a 2-way associative cache, LRU uses 1 extra bit in each set to indicate which of the two ways was used least recently. So, each set's LRU bit indicates the *victim way*, which way to replace on a miss, and is updated any time that set is accessed. For example, suppose a load misses and that set's LRU bit is 0, indicating way 0 is the victim. The cache writes back the set's way 0 if dirty and then fetches the missing line into way 0 and updates the LRU bit to 1.

LRU does not scale efficiently beyond two ways because recording the order of accesses and selecting the oldest becomes expensive. For example,

L1 caches are typically 8-32 KiB to achieve 1-2 cycle latency, depending on clock speeds and quality of circuit design. Each way is usually 4 KiB to work nicely with 4-KiB virtual memory pages, so they have 2-8 ways. L1 caches often have 64-byte lines, which offer a good tradeoff between hit rate and miss penalty.

Despite pLRU being an approximation of LRU replacement, the way to replace is still called the LRU way, and the bits to keep track of that way are still called the LRU bits.

an N -way associative cache can keep a list of N entries for each set indicating the relative order of accesses. When the cache accesses a way, it also moves the entry for that way to the head of the list. This requires $N \log_2 N$ LRU bits per set and hardware to search and update the list.

Tree pseudo-LRU, also referred to simply as *pseudo-LRU (pLRU)*, approximates LRU with a less-costly implementation that uses only $N-1$ LRU bits and still achieves a hit rate that is nearly as good as LRU. The bits are organized as a binary tree, with $\log_2 N$ levels, to approximate which side of the tree was least recently accessed. pLRU works by grouping ways into powers of two: 2 groups, then 4 groups, and so on until only one way is in each group.

Fig. 10.9(a) shows an example of a tree for an 8-way associative cache in which way 4 was least recently used (LRU). The LRU bits $N0$ – $N6$ in the tree are shown in circles and the ways are squares. A zero (0) in the tree indicates the right branch was more recently accessed and the left branch is older, a one (1) indicates the opposite. Traversing the tree along the arrows to the leaf box finds the pLRU (labeled LRU) victim way. In Fig. 10.9(a), the root LRU bit $N6$ is 1, so the right half (ways 4–7) were not used in the last access. Following the arrow to the right half leads to a 0 bit at $N4$, so the left half (ways 4–5) were not used the last time an access occurred to ways 4–7. Following the next arrow left to $N1$ reveals another 0, so way 4 was not used the last time an access occurred to ways 4–5. When the set is full and a new entry is needed, pLRU traverses the binary tree by following the arrows from $N6$ to $N4$ to $N1$ to select way 4 as the victim way to replace.

After a cache access uses a particular way, pLRU updates the tree to reflect this way's recent use. Every node between the root and the accessed way is updated to point away from the accessed way. Fig. 10.9(b) illustrates this pLRU update after accessing way 6, with the affected nodes labeled in blue. The path to traverse the tree from the root to the accessed

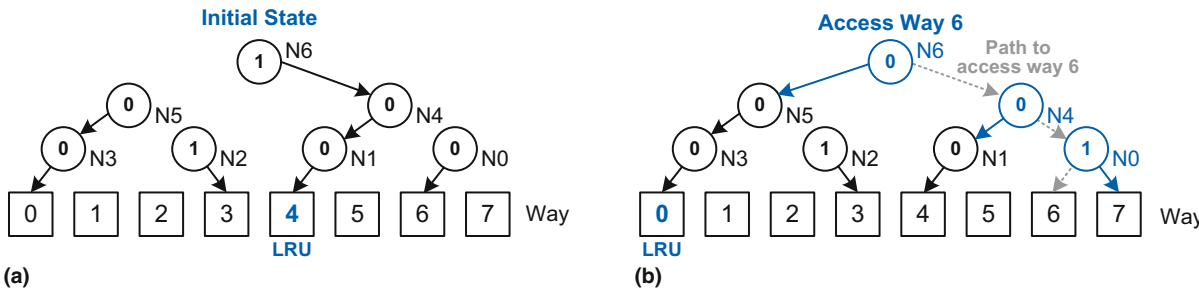


Fig. 10.9 Pseudo-LRU examples: (a) initial and (b) after accessing way 6.

way (6) is N6 to N4 to N0, shown by the gray dashed arrows. Each traversed node's LRU bit is updated to point away from the accessed way, as shown by the blue arrows: The root node N6 changes to 0 to point away from ways 4–7 (toward ways 0–3). The next node N4 is already 0, pointing away from ways 6–7 (toward ways 4–5). The final node N0 becomes 1 to point away from way 6 (toward way 7). Now the arrows indicate way 0 is least recently used.

Example 10.5 LRU TREE UPDATE

How would the tree of Fig. 10.9(b) be updated after accessing way 3? What way is now least recently used?

Solution: The path from the root to way 3 traverses nodes N6, N5, and N2, as shown by the gray dashed arrows in Fig. 10.10. Each node along this path is updated to point away from way 3, as shown by the blue arrows. Now the arrows indicate way 4 is least recently used.

Both LRU and pLRU can behave poorly under pathological access patterns. Thrashing occurs if the working set is larger than the number of ways. For example, if a program's working set maps five addresses to the same set and then accesses them in the same order repeatedly, a 4-way associative cache will conflict miss on every access.

Tree pLRU was first used in 1989 in the Intel 486 CPU [Intel89].

10.1.8 Cache Flush and Invalidation

Sometimes the cache may need to be *flushed* (have dirty lines written back to the next level of the memory hierarchy) or *invalidated* (valid bit $v = 0$ for all cache lines). For example, on reset, the cache must be invalidated. Also, because an instruction cache is read-only, anytime a program needs to write instructions, the writes go to the data cache instead. Hence, for the new instructions to be visible in the instruction cache, the processor must flush the dirty lines from the data cache back to the next level of the memory hierarchy and invalidate the instruction cache so that it reads the new instructions from the next level of the memory hierarchy.

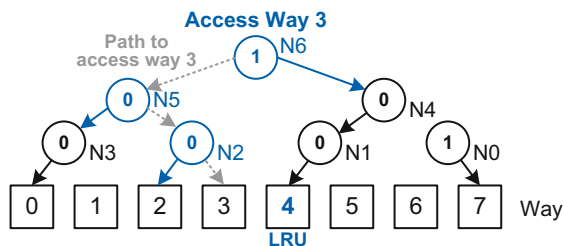


Fig. 10.10 Pseudo-LRU example after accessing way 3.

Beware that the term flush has multiple meanings. A *cache flush* writes back dirty lines. A *pipeline flush* clears an instruction from a pipeline stage. In this chapter, *flush* refers to a cache flush except when discussing pipeline hazards.

A flush operation walks through all data cache lines and writes the dirty lines to memory, then clears the dirty bits. An invalidate operation walks through cache lines and clears the valid bits.

Many processors support a sleep mode in which the cache array is powered off to save energy. Before entering sleep, write-back caches should be flushed. All caches should be invalidated when waking up.

10.2 RISC-V PRACTICES

In addition to handling fetches, loads, and stores, RISC-V caches have several design issues specific to the architecture:

- ▶ Atomic memory operations (AMOs) read a memory location, modify it, and write back to the same location in one indivisible, uninterruptable operation such that no other operation can access the memory between the read and write. This is discussed in [Chapter 17](#).
- ▶ Coherency between the I\$ and D\$ is maintained by flushing the D\$ and invalidating the I\$ using a `fence.i` instruction. Cache invalidation and flushes are discussed in [Sections 10.4.2.6](#) and [10.4.5.4](#).
- ▶ A physical memory attributes (PMA) checker indicates whether an address is cacheable. For example, memory-mapped peripherals cannot be cached because their value may change because of external events, such as a voltage changing on a general-purpose input/output (GPIO) pin.
- ▶ [Section 11.4.1.1](#) describes PMAs, and [Section 12.3.1](#) describes the LSU logic that handles uncacheable addresses.
- ▶ When compressed instructions are supported, instructions may wrap across cache lines. Such spills will be handled in [Section 14.3.3](#).
- ▶ The operating system (OS) likes to know the size of a cache line so it can allocate memory sensibly aligned to the cache. The Zi64b RISC-V extension indicates that cache lines are 64 bytes. This size has become an industry standard because it is a good tradeoff between spatial locality and time to fetch a missing cache line. Wally defaults to 64-byte (512-bit) cache lines and thus implements Zi64b by default.

10.3 TEST PLAN

The Wally caches are currently exercised through the ordinary instruction tests and Linux boot. [Section 21.3.1](#) describes cache performance verification.

10.4 WALLY IMPLEMENTATION

Wally has optional configurable level 1 instruction and data caches (L1I\$/L1D\$). Currently, it does not have any higher-level caches. Fig. 10.11 shows a simplified top-level block diagram of the `cache` module. A more detailed version is shown in Fig. 10.24. The cache interfaces to the core, bus, privileged unit, and hazard unit. It contains three main submodules: `cacheway`, `cachefsm`, and `cacheLRU`. An N -way associative cache stores lines in N `cacheway` modules, each of which contains data, tag, and status bits for each line. The `{I/D}CACHE_NUMWAYS` configuration parameter specifies N . Each `cacheway` handles load, store, invalidate, and flush operations. The `cachefsm` sequences multicycle operations such as line fetches caused by cache misses. An associative cache also contains a `cacheLRU` unit to select a victim way for replacement. The shaded boxes show other top-level cache

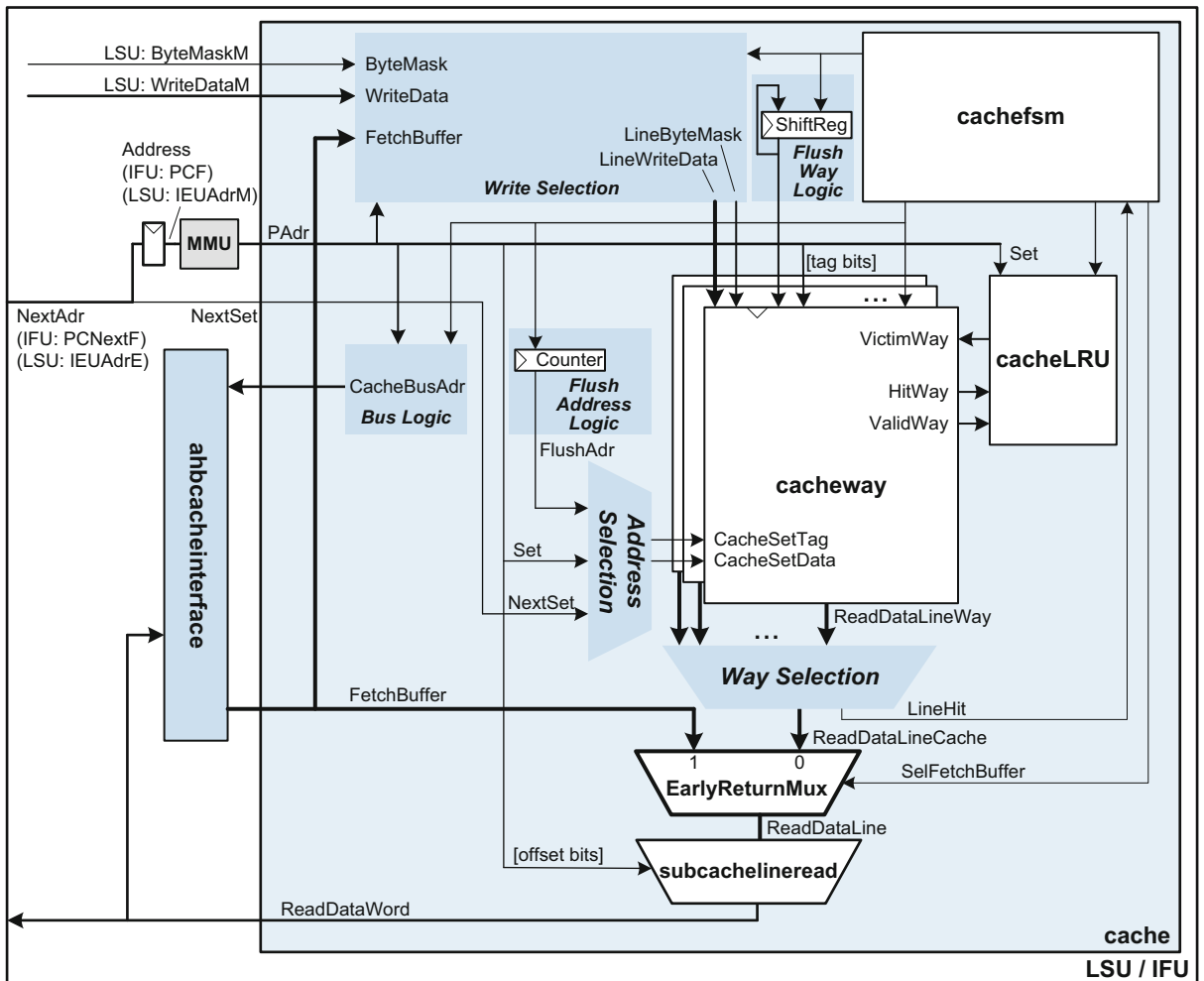


Fig. 10.11 Cache top-level block diagram.

logic. The cache and hazard unit interact to stall the pipeline during cache misses, and the cache interlocks with the privileged unit when traps occur. Table 10.1 lists the cache parameters and common values. This section explains the cache implementation common to instruction and data caches, and Chapters 12 and 13 explore the design issues specific to integrating the caches into the LSU and IFU.

The I\$ and D\$ both use the same configurable SystemVerilog `cache` module. The module has a `READ_ONLY_CACHE` parameter, which is 0 for the D\$ and 1 for the I\$. Both types of caches support reads, but only data caches support writes, writebacks, and flushing dirty lines (`READ_ONLY_CACHE = 0`). Both instruction and data caches support invalidation.

The caches use synchronous memories that expect the address, control, and write data to set up before the rising edge of the clock. They read or write on the rising edge. Unlike the rest of the CPU core, the caches do not use suffixes F, D, E, M and W to denote the pipeline stage because the same SystemVerilog code (RTL) is used for both caches. Caches operate in two pipeline stages, Address and Data.

The primary inputs and outputs of a read-only cache used during a cache hit are address (`NextAdr` and `PAdr`) and `ReadDataWord`. `PAdr` is delayed by one cycle relative to `NextAdr`; if virtual memory is enabled, it is also translated from virtual to physical by the memory management unit (MMU) described in Chapter 11. A writable cache adds the `WriteData` and `ByteMask`. The cache only needs a few of the address bits, called

Table 10.1 Cache configuration for figures in this section, when not generic.

Cache Parameter	Parameter Name	Value	Description
Line size	<code>LINELEN</code>	64 bytes	512 bits = 2^6 bytes = 8 doublewords
Number of ways	<code>NUMWAYS</code>	4	
Number of sets	<code>NUMSETS</code>	64	2^6 sets
Way size		4 KiB	$\text{NUMSETS} \times \text{LINELEN} = 2^6 \text{ sets} \times 2^6 \text{ bytes/set} = 2^{12} \text{ bytes}$
Cache size		16 KiB	$\text{NUMWAYS} \times \text{NUMSETS} \times \text{LINELEN}$
Word size	<code>WORDLEN</code>	64 bits	
Physical address size	<code>PA_BITS</code>	56 bits	Sv39 or Sv48
Tag size		44 bits	<code>PAdr[55:12]</code> $\text{PA_BITS} - \log_2(\text{Way size})$
Set bits		6 bits	<code>PAdr[11:6]</code> $\log_2(\text{NUMSETS})$
Word offset		3 bits	<code>PAdr[5:3]</code> $\log_2(\text{LINELEN}/\text{WORDLEN})$
Byte offset		3 bits	<code>PAdr[2:0]</code> $\log_2(\text{WORDLEN})$

NextSet or Set, to specify which set to access. These bits are the same in the virtual and physical addresses, so accessing the cache does not have to wait for translation.

During the Address stage, the least significant bits of NextAdr (NextSet) set up before the clock edge, indicating which set to read. During the Data stage, the cache reads the ways. It checks the valid bits and compares the tags to the upper bits of PAdr to determine if each way hits. For a read hit, the cache returns a word of data from that way on ReadDataWord. For a write hit, the cache uses PAdr (Set), WriteData, and ByteMask to write part of a line to the way on the next clock edge.

10.4.1 Interfaces

Fig. 10.12 shows the Wally cache interfaces to the core (lsu and ifu), privileged, and hazard units.

The previous section described the primary interface signals used during a cache hit: address (NextAdr and PAdr), ReadDataWord, WriteData, and ByteMask. The cache also takes the CacheRW[1:0] control signal, with bit 1 indicating a read and bit 0 indicating a write.

On a cache miss or eviction, the cache fetches or writes back a line from memory over the EBU via the AHB cache interface (ahbcacheinterface).

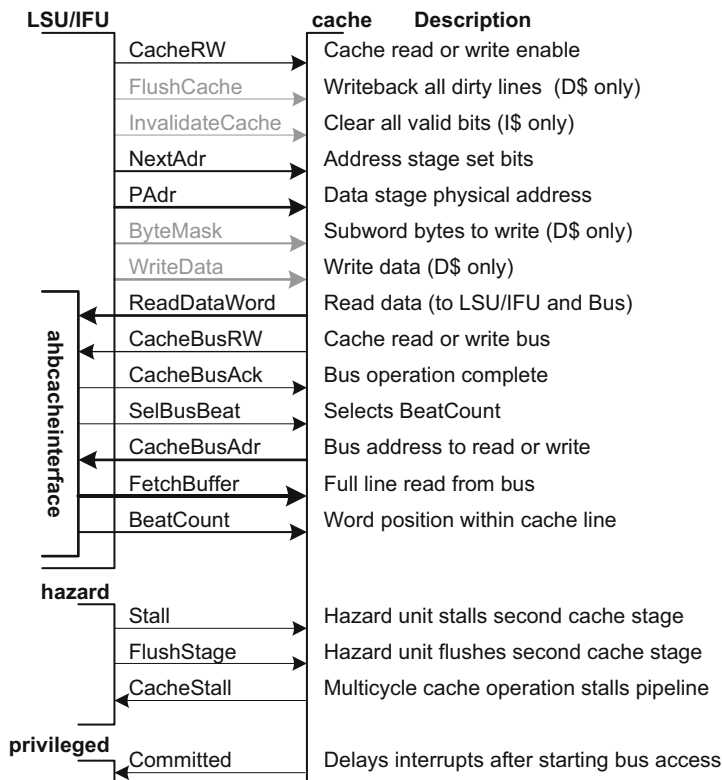


Fig. 10.12 Cache interfaces to CPU and memory bus.

Although Fig. 10.12 shows the cache outside the LSU/IFU for ease of drawing, the cache is actually located inside the LSU/IFU and communicates with other components within these units. The black signals are major busses or control signals, and the gray signals only exist for specific types of caches (I\$ or D\$). Chapters 12 and 13 delve into the LSU and IFU interfaces.

`FlushStage` is a pipeline flush, not a cache flush. It resets the cache FSM and cancels pending D\$ writes.

The overall way size (4 KiB) matches the virtual memory page size, which is described in [Section 11.1.1.7](#). The line length of 64 bytes is a typical length in commercial processors and offers a tradeoff between spatial locality and miss penalty. The number of lines in each cache way is $\text{NUMSETS} = \text{CACHE_WAYSIZEINBYTES} / (\text{CACHE_LINELENINBITS} / 8) = (4,096 \text{ bytes/way}) / ([512/8] \text{ bytes/line}) = 64 \text{ lines/way}$.

The physical address is 56 bits in `rv64gc`. As shown in [Table 10.1](#), the 12 least significant bits (the set, word offset, and byte offset bits) are used to select a byte within a 4-KiB way,... so the tag is the remaining 44 bits. Thus, the tag array size is 64 lines $\times$ 44 bits/line.

[Table 6.5](#) shows how SRAM is roughly four times denser and has two times lower energy than flash-based memories.

The `CacheBusAdr` and `CacheBusRW[1:0]` signals indicate the address to access and whether to read or write. The EBU uses burst transfers for performance, with `BeatCount` and `SelBusBeat` tracking and selecting the word in the burst and `CacheBusAck` signaling completion. Words read from the bus accumulate in the fetch buffer until the full cache line is ready to write to the cache. Words are written to the bus one at a time over `ReadDataWord`.

The caches may stall the pipeline or postpone an interrupt (using their `CacheStall` output), and the pipeline may stall the I\$ or D\$ or flush a D\$ operation using a cache's `Stall` input. During a cache miss or other multi-cycle operation, the cache asserts `CacheStall`, which causes the hazard unit to assert `StallW` and stall the entire pipeline. Conversely, when the hazard unit asserts `StallD` or `StallW`, which are connected to the `Stall` input of an I\$ or D\$, respectively, the given cache stalls. When the cache's Data stage needs to be flushed, the hazard unit asserts `FlushStage`. For example, `TrapM` flushes the pipeline's Memory stage so a D\$ operation should be flushed. The cache sends `Committed` to the privileged unit to postpone interrupts while a bus operation is in progress.

10.4.2 Cache Way Module

The `cacheway` module implements a single way of an *N*-way associative cache. [Fig. 10.13](#) shows a block diagram of `cacheway` configured according to [Table 10.1](#). `cacheway` includes storage arrays to hold data, tags, and status bits for each line in a way. It uses the following:

- ▶ Separate SRAM arrays to hold the data and tags
- ▶ Arrays of flip-flops to hold the status (valid and dirty) bits for each cache line so these bits can be easily written or reset

The cache way reads the tag specified by `CacheSetTag`. If `ReadTag` matches the upper bits of the physical address `PAdr` and the line is valid, the cache hits and returns the data line as `ReadDataLineWay`. The control logic in [Fig. 10.13](#) handles pipeline flushes, cache invalidation, and updating status bits. This section details how `cacheway` reads, writes, fetches, writes back, and invalidates cache lines. It also describes a structural hazard caused by synchronous SRAM timing.

`cacheway` is one of the largest modules in Wally, so it is designed for area efficiency. The large data and tag arrays are built from SRAM for density and energy efficiency. SRAM generators have a maximum width, typically around 128 bits. Therefore, the 64-line $\times$ 512-bit wide data array is partitioned into four 64 $\times$ 128-bit banks of SRAM. The arrays have byte enables to support writing words or bytes within the line. They also have chip enable pins to save power and retain their old data output when disabled. When the hazard unit stalls the pipeline, the chip enable is turned off, so the cache retains its output. The status bits may be modified

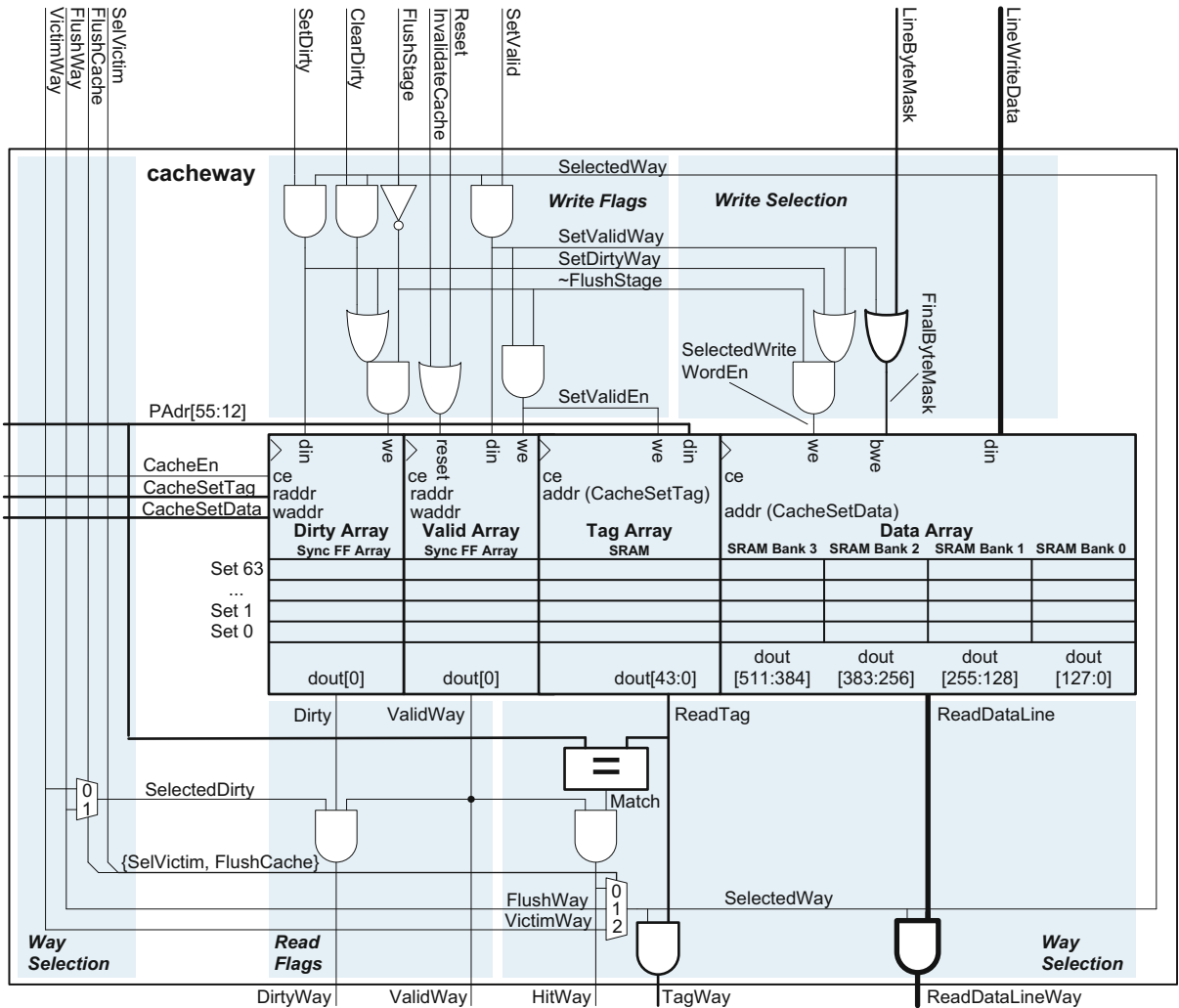


Fig. 10.13 cacheway block diagram.

independent of writing the data and tag arrays, so they are most conveniently built from flip-flops. The valid bits are reset on startup or invalidation. The data, tag, and dirty bits are ignored if the valid bit is 0, so they do not require the overhead of reset circuitry.

The synchronous memories require an address to setup before the rising clock edge and read or write on the clock edge. So, the cache operates in two stages: Address and Data stages. For cache reads that hit, this operates as follows:

- ▶ **Address stage:** CacheSetTag sets up to read the tag and status arrays. CacheSetData sets up to read the data array. CacheSetTag = CacheSetData = NextSet.

- ▶ **Data stage:** The SRAM arrays output the tag and data bits, `ReadTag` and `ReadDataLine`, and the valid array outputs the valid bit, `ValidWay`. A hit occurs when the requesting address tag and the line's tag match and that line is valid.

Separate set signals, `CacheSet{Tag/Data}`, are needed to access the tag/status arrays and data arrays to avoid structural hazards during back-to-back writes. Writes set up in the Address stage to read the tag and status arrays (but, unlike reads, not the data array) and then set up in the Data stage to write the data array. Specifically, upon a write hit:

- ▶ **Address stage:** `CacheSetTag = NextSet` sets up to read the tag and status arrays.
- ▶ **Data stage:** The tag and valid arrays output `ReadTag` and `ValidWay`. On a hit, `CacheSetData = PAddr[11:6]` sets up to write the data array. [Section 10.4.2.2](#) explains this further.

The `SelectedWay` mux chooses whether a given cache way is selected to read or write a line. `SelectedWay` equals:

- ▶ `HitWay` for regular loads and stores
- ▶ `FlushWay` for flushes (when `FlushCache = 1`)
- ▶ `VictimWay` for writing back a dirty line (when `SelVictim = 1`)

The cache uses a distributed AND-OR multiplexer to read out the selected way. Each `cacheway` ANDs its data (`ReadDataLine`) with `SelectedWay`, so only the selected way has a nonzero `ReadDataLineWay`. The top-level cache module ORs these signals from each way to deliver `ReadDataLineCache`, as illustrated in [Section 10.4.5.1](#) and [Fig. 10.25](#).

The remainder of this section describes read hits, write hits, structural hazards, line fetches, line writebacks, and invalidation in detail.

10.4.2.1 Cache Way Read Hit

A cache way read (load or instruction fetch) begins in the Address stage with the set bits (`CacheSetTag = CacheSetData = NextSet`) specifying which set to read, as shown in [Fig. 10.14\(a\)](#). On the rising edge of the clock (start of the Data stage), `cacheway` reads `ReadDataLine`, `ReadTag`, and `ValidWay`. If `ReadTag` matches the address's tag (`ReadTag == PAddr[55:12]`) and the line is valid (`ValidWay = 1`), then `Match = 1` and, thus, `HitWay = 1`, which indicates a hit in that way. `SelectedWay = HitWay = 1`, so the `ReadDataLineWay` AND gate outputs `ReadDataLine`.

Example 10.6 CACHE WAY HITS

A processor loads 4 unsigned bytes from address `0x00_0000_8000_0384`. Sketch waveforms for a cache way hit using the `rv64gc` configuration of [Fig. 10.13](#) and [Table 10.1](#) and no virtual memory. The cache line contains `0x0...0_AAAA_5555_1111_2222`.

Solution: Fig. 10.14(b) shows a timing diagram.

- In cycle 0 (Address stage), set up to read data and tags from line 14: $\text{CacheSet}\{\text{Tag}/\text{Data}\} = \text{NextAdr}[11:6] = 001110_2$.
- In cycle 1 (Data stage), the cache way reads ReadDataLine , ReadTag , and Valid from line 14. The line is valid and the tag matches PAdr , so $\text{HitWay} = 1$, $\text{SelectedWay} = 1$, and ReadDataLineWay outputs the whole line, $0x0\dots0_AAAA_5555_1111_2222$. The subcachelineread module (see Fig. 10.11) then selects the word from that line using the word offset bits $\text{PAdr}[5:2] = 0001$ to select $\text{ReadDataWord} = 0xAAAA_5555$.

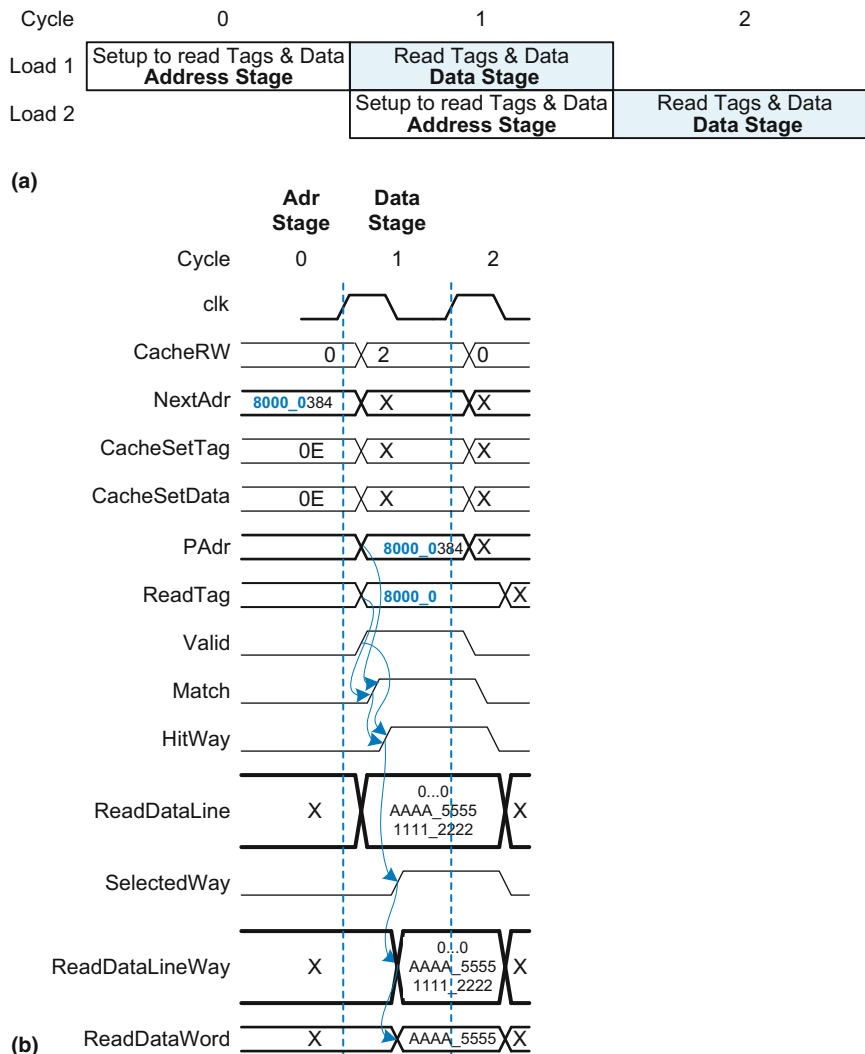


Fig. 10.14 Cache read: (a) back-to-back reads and (b) detailed cacheway timing diagram for a single read.

10.4.2.2 Cache Way Write Hit

A cache way write (store) begins in the Address stage by reading the tag and status arrays to check the tag and valid bits using `CacheSetTag`. However, the data array is not read. During the Data stage, `SetDirty = 1`, so upon a hit (`SelectedWay = HitWay = 1`), `cacheway` asserts `SetDirtyWay`. The cache drives `CacheSetData` with the set to write, `LineWriteData` with `WriteData`, and `LineByteMask` to specify the bytes within the line to write. On the rising clock edge at the end of the Data stage, the line is written and the dirty bit is set.

By not reading the data array in the Address stage (which is unnecessary for stores), back-to-back stores do not have to stall because only the tag array is used in the Address stage, and only the data array is used in the Data stage. For example, Fig. 10.15 shows three consecutive store instructions that hit, where the white and blue boxes show the cache’s Address and Data stages, respectively. In cycle 0, store 1 sets up its address to read the tag and status arrays. In cycle 1, store 1 reads the tag and status bits, detects a hit, and sets up to write the status array and its word into the data array (at the end of the cycle). Also in cycle 1, store 2 sets up to read the tag and status arrays. In cycle 2, store 2 reads the tag and status bits, detects a hit, and sets up to write the status bits and its word into the cache’s data array, and so on. Accessing the tag and data arrays in this way requires separate addresses, `CacheSetTag` and `CacheSetData`. For loads, these addresses are the same and are both driven (set up) in the Address stage. However, for stores, only `CacheSetTag` is used in the Address stage, and only `CacheSetData` is used in the Data stage. So, for writes, `CacheSetData` lags `CacheSetTag` by one cycle.

The status arrays may be used in both stages: they are read in the Address stage and, for writes, written in the Data stage. However, the status arrays are dual-ported, so they may be read and written in the same cycle. In contrast, the tag and data arrays are single-ported, so they can only be read or written in a given cycle.

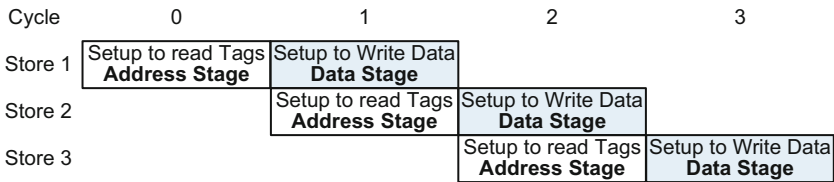


Fig. 10.15 Timing of back-to-back stores that hit.

Example 10.7 CACHE WRITE HITS

Sketch waveforms for a processor writing a 4-byte value 0x12345678 to address 0x00_0000_8000_0388 followed by a second 4-byte write of 0xAABBCCDD to address 0x00_000_8000_03C8. Assume the line is already in the cache.

Solution: Fig. 10.16 shows the waveforms:

- ▶ In cycle 0 (Address stage of store 1), `CacheSetTag = NextAdr[11:6] = 14`, as in Example 10.6.
- ▶ In cycle 1 (Data stage of store 1), the cache way reads the tag and valid bit from set 14. The cache way hits because the tag matches (`ReadTag = PAdr[55:12]`), and `ValidWay` is 1, so `Match = 1`, `HitWay = 1`, and `SelectedWay = 1`, and `SetDirtyWay = 1`. The cache drives `CacheSetData = PAdr[11:6] = 14` to write

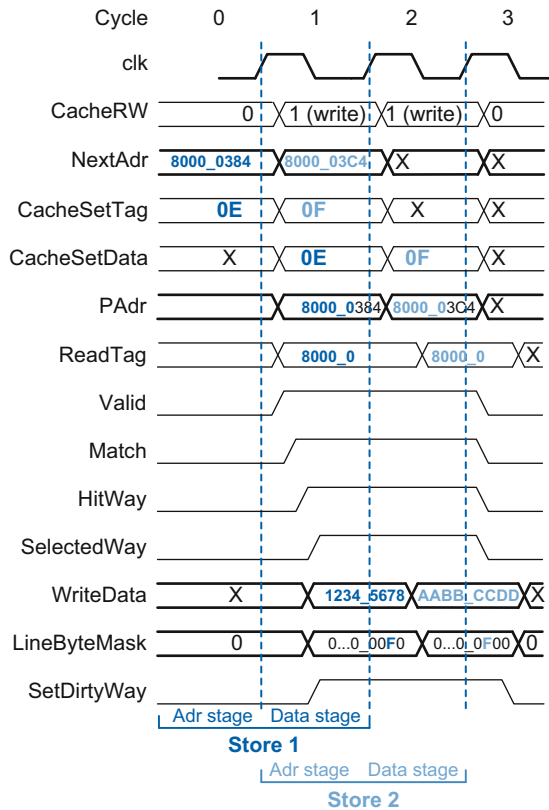


Fig. 10.16 cacheway timing diagram for back-to-back stores.

to the status and data arrays. The word offset is $\text{PAAdr}[5:2] = 0001_2 = 1$, so the write selection logic writes word 1 (bytes 7:4) in the 16-word line by computing $\text{LineByteMask} = 0x0000_0000_0000_00F0$. The cache replicates the write data into $\text{LineWriteData} = 0x12345678_..._12345678_12345678$. The write completes on the next rising clock edge at the end of the Data stage.

- ▶ **Also in cycle 1** (Address stage of store 2), $\text{CacheSetTag} = \text{NextAdr}[11:6] = 001111_2 = 15$.
- ▶ **In cycle 2** (Data stage of store 2), the word offset is $0010_2 = 2$, which selects $\text{LineByteMask} = 0x0000_0000_0000_0F00$ to write word 2 (bytes 11:8) in the line. $\text{LineWriteData} = 0xAABBCCDD_..._AABBCCDD$, which is the desired WriteData replicated in each word (group of 4 bytes).
- ▶ Notice that, for store hits, CacheSetData lags CacheSetTag by one cycle.

10.4.2.3 Structural Hazards

Whereas back-to-back stores do not cause hazards, a store followed by a load does. Fig. 10.17 shows this scenario, with the cache's Address stage

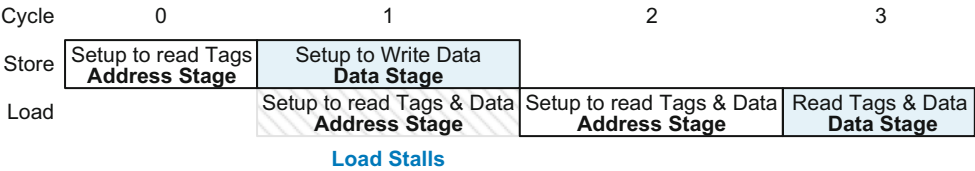


Fig. 10.17 Timing of a store followed by a load.

This hazard is identical to the store delay hazard in the DTIM discussed in [Section 7.2.2.3](#). The hazard logic in the integer execution unit (IEU) controller asserts LoadStallD when a load is followed by its use and StoreStallD when a store is followed by a load.

shown in white and its Data stage in blue, as before. The hazard occurs in cycle 1, where both the store and load want to access the data array: the store sets up to write the data array and the following load wants to read the data array, so the load must stall by one cycle.

10.4.2.4 Line Fetch

If no cache way hits during a memory access (HitWay for all cacheways is 0), the cache misses and must fetch the data from the next level of the memory hierarchy. This requires the following:

- ▶ Selecting a cache way for the new line, potentially needing to evict a line to make room (see [Section 10.4.4](#)).
- ▶ Writing back the evicted line if it is dirty (see [Section 10.4.2.5](#)).
- ▶ Fetching a new line using the AHB cache interface (ahbcacheinterface) word by word into a fetch buffer (see [Sections 10.4.5.3](#) and [12.2.1](#)).
- ▶ For a store miss, merging WriteData into the appropriate position in the fetch buffer (see [Section 10.4.5.2](#)).
- ▶ Writing the fetch buffer into the line of the selected way (see [Section 10.4.5.2](#)) while updating the tag from PAdr and setting the Valid bit. Dirty is set to 1 for a write or cleared to 0 for a read.

10.4.2.5 Line Writeback

If a cacheLRU selects a dirty line for eviction, the line must first be written back to the next level of the memory hierarchy. [Fig. 10.18](#) shows how the victim way outputs its tag and data. When a cache way evicts a line, FlushCache = 0 and VictimWay = 1. The cache’s controller drives SelVictim = 1, so SelectedDirty = 1. SelectedDirty is ANDed with the status array’s dirty bit to check if the victim set is dirty. The SelectedWay mux chooses VictimWay, which is 1, so the cache way outputs TagWay and ReadDataLineWay. The cache writes the line in ReadDataLineWay back to the address indicated by TagWay and CacheSetTag. [Section 12.2.2](#) describes writing the line over the bus interface one word at a time.

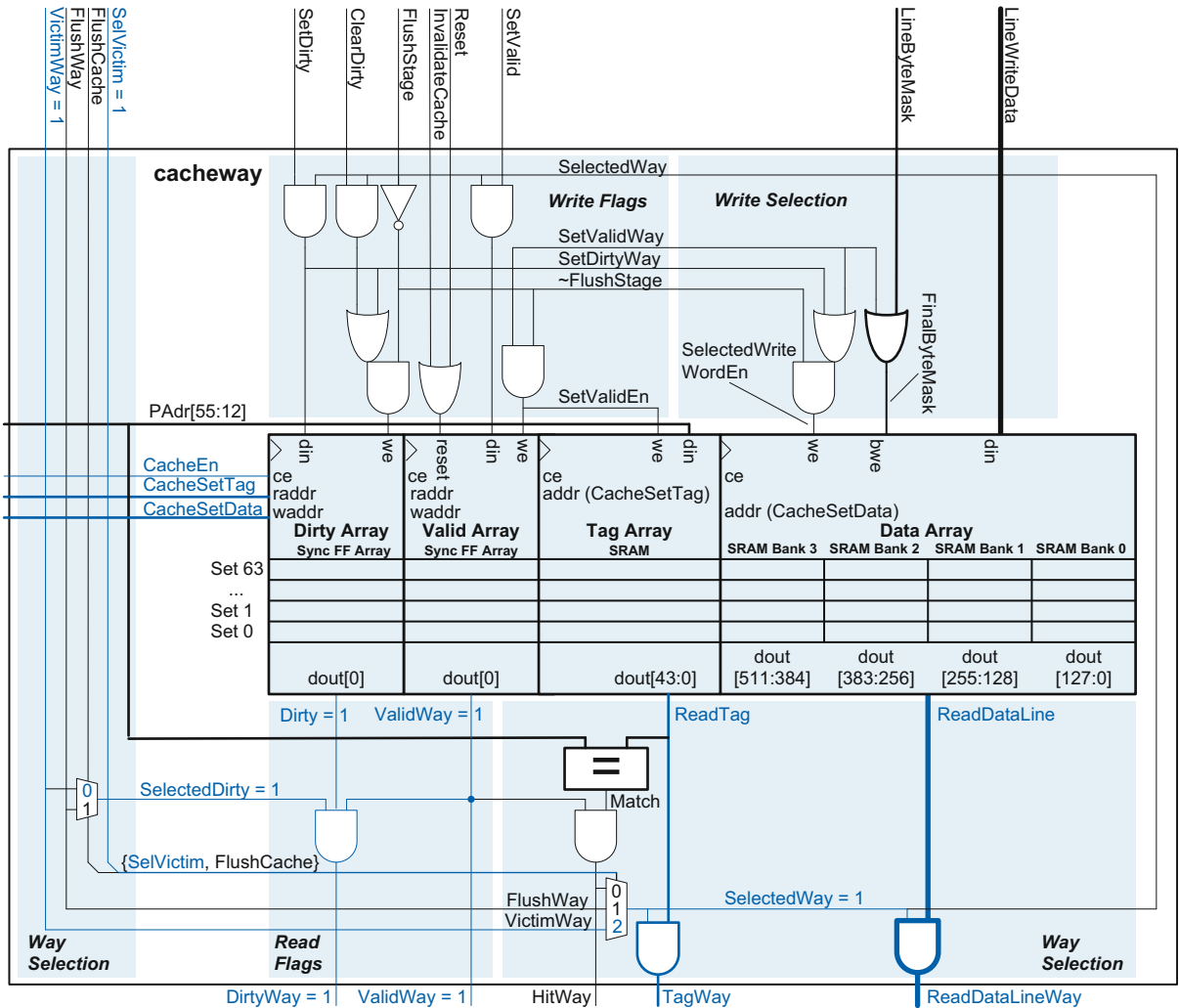


Fig. 10.18 cacheway during line writeback.

10.4.2.6 Way Invalidation

Recall from Section 10.1.8 that a cache is invalidated by clearing its valid bits. Both the D\$ and I\$ must be invalidated on reset. The Valid array contains resettable flip-flops. It receives a reset signal to clear all valid bits in parallel on the rising edge of the clock.

The Zifencei RISC-V extension defines `fence.i` to ensure coherence between the I\$ and D\$. For example, the OS may load a program or parts of a program into memory by copying them from other regions of memory. Because this is performed using loads and stores in the OS code, these writes go through the D\$ rather than the I\$, so the I\$ may contain incorrect stale instructions. The `fence.i` instruction flushes the D\$ to main memory and invalidates the I\$ so that the I\$ is forced to fetch the new instructions. `fence.i` is an expensive operation, so fences should be used sparingly.

10.4.3 Cache FSM

Fig. 10.19 shows the cache FSM (*cachefsm*), which handles cache misses and flushes. The FSM resets to the Access state, where it accepts read, write, invalidate, and flush requests from the CPU. If the cache hits on a read or write, it services the request immediately and remains in the Access state. Otherwise, *cachefsm* sequences the multicycle operations to writeback, fetch, or flush cache lines.

Read and write misses assert *AnyMiss*, select a victim line to evict using the *cacheiru* module, write back the victim if dirty, and fetch the missing line. If the victim is dirty, it must be evicted by writing the line back over the bus (in the Writeback state). Each bus operation waits for *CacheBusAck* to continue. Then the cache fetches the missing line over the bus (in the Fetch state). The cache writes the missing line and tag into the selected way and sets the valid bit in the WriteLine state. Also in the WriteLine state, writes merge the new word from *WriteData* into *LineWriteData* and set the dirty bit; reads clear the dirty bit. Misses pass through the AddressSetup state before returning to the Access state because the next access needs a cycle to set up the cache address (set bits) to the cache before accessing it, as discussed in Section 10.4.2.1. If the other cache is also stalling the pipeline, Stall is asserted to hold the read data (in the AddressSetup state), as discussed in Section 10.4.6.

Invalidation occurs in the Access state. It resets all valid bits simultaneously, as discussed in Section 10.4.2.6. Flushes iterate through all lines of all ways, writing back any dirty lines as will be discussed in Section 10.4.5.4.

Table 10.2 summarizes the *cachefsm* outputs. *SetValid*, *SetDirty*, *ClearDirty*, *FlushCache*, and *SelWriteback* control writes to *cacheway*, as shown in Fig. 10.13. *SelVictim* selects a way when writing back or fetching

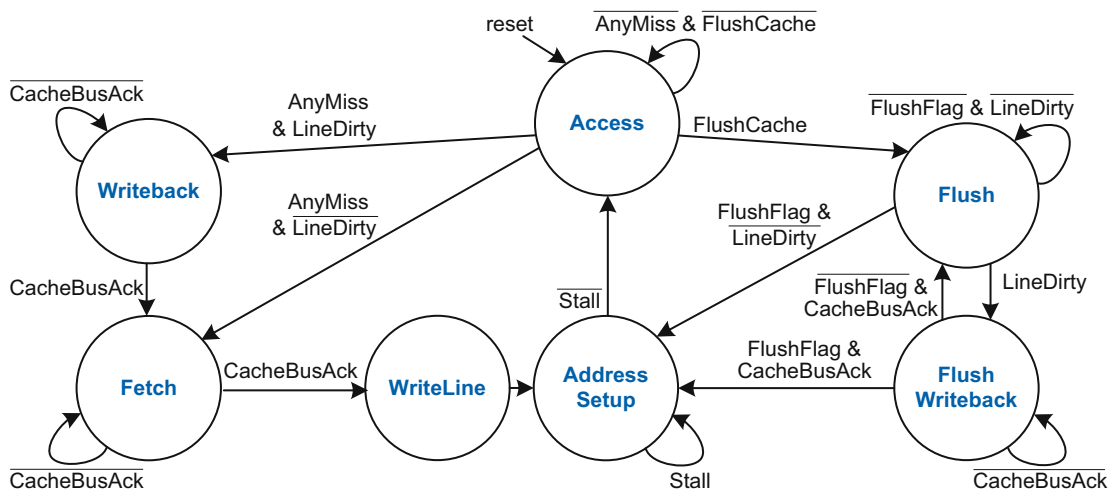


Fig. 10.19 Cache FSM (*cachefsm*).

Table 10.2 cache fsm outputs.

FSM Output	Destination	Meaning	Criteria to Assert
SetValid	cacheway, cachelru, cache	For a line in a way: set valid bit; write data and tag	WriteLine
SetDirty	cacheway, cache	For a line in a way: set dirty bit	Write & (Access & Hit WriteLine)
ClearDirty	cacheway	For a line in a way: clear dirty bit	~Write & WriteLine (Flush & ~LineDirty)
SelWriteback	cacheway	Writeback a cache line	NextState = Writeback
SelVictim	cacheway	A cache way outputs a victim line (used for writeback and fetch)	Access & Miss & LineDirty WriteLine WriteBack & ~CacheBusAck
LRUWriteEnable	cachelru	Update LRU after an access	Access & Hit WriteLine
SelAdrData	cache	Select CacheSetData from PAdr (instead of NextAdr) during multicycle operation or store	Access & (Miss Write) Writeback Fetch WriteLine
SelAdrTag	cache	Select CacheSetTag from PAdr (instead of NextAdr) during multicycle operation	Access & Miss Writeback Fetch WriteLine
SelFetchBuffer	cache	Pick word from FetchBuffer during line fetch	WriteLine AddressSetup
CacheEn	cacheway, cachelru	Enable reads or writes	~Stall FlushCache AnyMiss ~Access InvalidateCache
CacheBusRW[1:0]	ahbcacheinterface	Request to read or write bus	[1]: Access & Miss & ~LineDirty Writeback & CacheBusAck Fetch & ~CacheBusAck [0]: Access & Miss & LineDirty Writeback & ~CacheBusAck FlushCheck & LineDirty FlushWriteback & ~CachBusAck
FlushAdrCntEn	cache	Increment the flush line address counter	See Section 10.4.5.4
FlushWayCntEn	cache	Increment the flush way address counter	See Section 10.4.5.4
FlushCntRst	cache	Reset both set and way counters (that are used for flushes)	Reset Flush & FlushFlag & ~LineDirty FlushWriteback & FlushFlag & CacheBusAck
CacheStall	hazard	Cache stalls the pipeline	NextState != (Access AddressSetup)
CacheCommitted	trap	Cache delays interrupts	~Access
CacheAccess	Performance counter	Cache is accessed	Access & (AMO CacheRW)
CacheMiss	Performance counter	Cache misses	Access & ~Hit

a line. `LRUWriteEnable` enables updated LRU bits to be written. `SelAdrTag` and `SelAdrData` control the address muxes for `CacheSetTag` and `CacheSetData`, respectively, and select among the current address (`PAdr`), the next address (`NextAdr`), or the flush address (`FlushAdr`), as discussed in Section 10.4.5.

After fetching a cache line, the `cachefsm` asserts `SelfFetchBuffer` in the `Writeline` state to bypass the `cacheway` SRAMs and select the desired word from the fetch buffer (supplied by the AHB cache interface, see Fig. 10.12) instead. This saves a cycle as compared to waiting for the fetch buffer to be written to the cache and then reading the desired word from the cache. `CacheEn` asserts when reading or writing the cache, and `CacheRW[1:0]` is 1 when reading a cache line (line fetch) or 0 when writing a dirty line back, both using the AHB bus.

The flush counter controls are described in Section 10.4.5.4. `CacheStall` is described in Section 10.4.6, and `CacheCommitted` is described in Section 10.4.7.

The `cachefsm` pulses `CacheAccess` on each access, and `CacheMiss` on each miss. The performance counters use these signals to track cache accesses and misses.

10.4.4 pLRU Implementation

Section 10.1.7 introduced the theory of tree pseudo least recently used (pLRU) for selecting a victim way on a cache miss. This section maps the theory into an implementation suitable for a cache of any number of ways. An N -way associative cache puts the pLRU tree into an $(N-1)$ -bit LRU signal associated with each set. On a miss, the cache selects the victim way from these LRU bits. The cache also updates the LRU bits after each access. This section uses an 8-way associative pLRU tree as an example and generalizes the result at the end.

Fig. 10.20(a) shows a pLRU tree for an $N = 8$ -way associative cache. The node directions (left or right) are held in an $(N-1)$ -bit LRU signal,

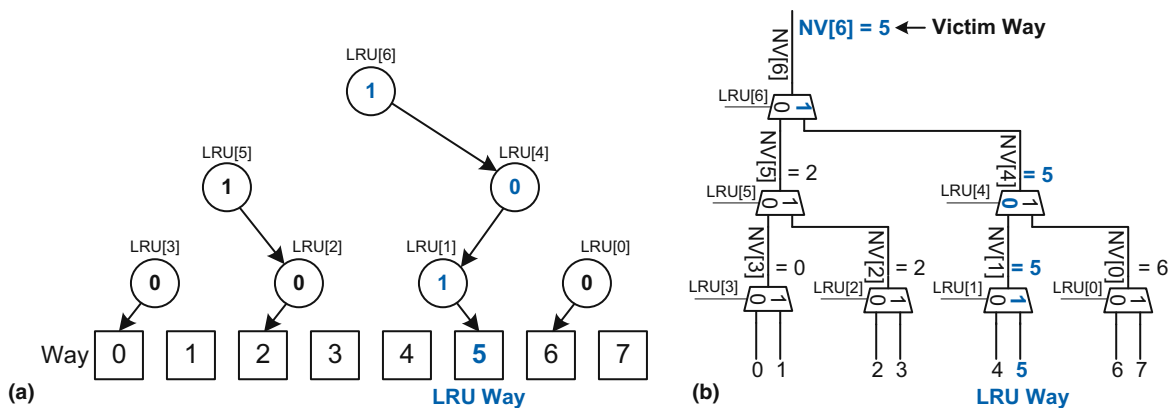


Fig. 10.20 8-way associative cache pLRU victim way implementation.

with 1 bit per node, that points to the least recently used (LRU) way. $LRU[N-2]$ is the direction of the top node, and the remaining nodes map to the lower bits, top to bottom and left to right in the tree. In Fig. 10.20(a), $LRU = 1100010$.

Fig. 10.20(b) shows the tree of multiplexers used to find the *new victim* way. Each node calculates its new victim (LRU) way in its $\log_2 N$ -bit NV (new victim) signal and then passes that signal to the node above it. The overall victim way is the top node's new victim, $NV[N-2]$, in this case $NV[6]$. In this example, $LRU[6] = 1$ chooses the victim from ways 4–7. $LRU[4] = 0$ chooses the victim from ways 4 and 5, and $LRU[1] = 1$ selects the victim as way 5 ($NV[1] = 5$). The signal ($NV[1] = 5$) is passed up the tree using the multiplexers to make the new victim way 5 ($NV[6] = 5$).

Fig. 10.20(b) uses $(N-1) \log_2 N$ -bit 2:1 muxes to select the victim way. However, the way numbers (0–7) are constant, so the hardware simplifies dramatically, as shown in Fig. 10.21. The top multiplexer selects ways 0–3 (000-011) when $LRU[6] = 0$ and ways 4–7 (100-111) when $LRU[6] = 1$, so this reduces to the msb of $NV[6][2] = LRU[6]$. The lower multiplexers follow a similar pattern. The next most significant bit of the victim ($NV[6][1]$) will be either $LRU[5]$ or $LRU[4]$, and the lsb will be $LRU[3]$, $LRU[2]$, $LRU[1]$, or $LRU[0]$. In the example figure, $NV[6][2:0] = \{LRU[6], LRU[4], LRU[1]\} = 101$.

Fig. 10.22 shows the `cacheLRU` module, which selects a victim way and updates the LRU bits held in the LRU Array. After a cache access, the hardware updates LRU to point nodes away from the accessed (i.e., most recently used) way. Thus, the LRU array is read at the end of the Address stage and written at the end of the Data stage.

The Victim Way logic in Fig. 10.22 selects either an invalid way, when available, or the victim calculated by the pLRU logic. `AllValid` is the bit-wise AND of `ValidWay[N-1:0]`, a signal showing the valid way, using one bit per way.

- ▶ When all ways are valid (`AllValid = 1`), the new victim is chosen as the pLRU output, (`VictimWayEnc = NV[6]`).

The LRU array is held in flip-flops rather than SRAM because it is small, requires reset, and is both read and written each cycle.

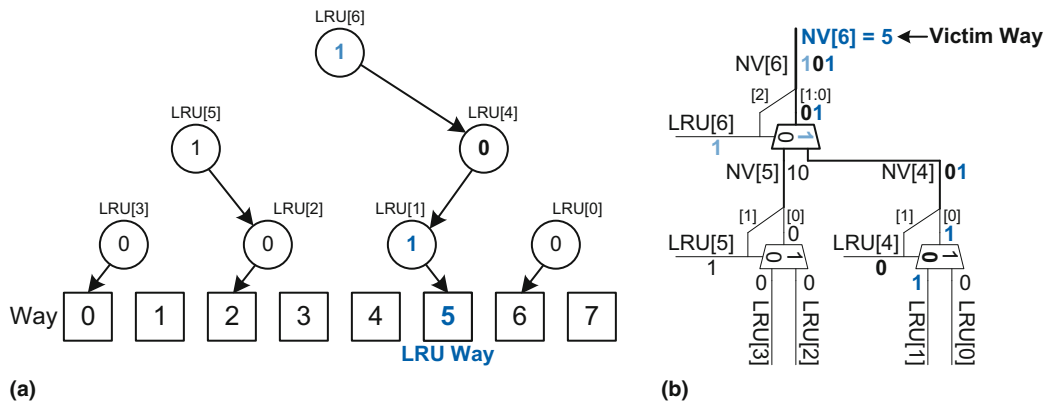


Fig. 10.21 Simplified way selection hardware.

- **WayExpand:** The direction each node points to reach the accessed way.
- **NextLRU:** Updated LRU bits, which are the original LRU bits for non-traversed nodes, and the inverse of WayExpand for traversed nodes, so that these nodes point away from the accessed way.

WayExpand replicates each bit of Way for each successive level of the pLRU tree. For example, if the accessed way (Way) is 5 (101), then WayExpand is 1001111. That is, to access way 5, the top-level node (LRU[6]) points right (1), the next-level nodes (LRU[5:4]) point left (00), and the lowest-level nodes (LRU[3:0]) point right (1111). Exactly one node at each pLRU tree level will be traversed to access a given way. Thus, because LRU[6] points right, only LRU[4]’s direction is relevant. And because LRU[4] points left, only LRU[1]’s direction is relevant.

Example 10.8 CALCULATE THE NEXT LRU BITS

Using Fig. 10.22, show the process for calculating NextLRU during an access to way 6 given LRU = 1100010.

Solution: Fig. 10.23 shows the pLRU tree values before and after an access to way 6. First, calculate LRUUpdate = 1010001, which has 1s for traversed nodes (nodes 6, 4, and 0) along the path to way 6, shown with a gray dashed arrow. The other nodes are not involved in the access to way 6, so they will not be updated. Way = 110 (6), so WayExpand = 1110000. The NextLRU mux in Fig. 10.22 chooses LRU or WayExpand for each bit depending on each LRUUpdate bit (1010001). So, NextLRU equals the original LRU bits (LRU) for bits 5, 3, 2, and 1 (-1-001-) and WayExpand for bits 6, 4, and 0 (0-0---1), so that those nodes point away from way 6. Altogether, NextLRU = 0100011. The updated LRU bits (6, 4, and 1) are highlighted in blue in Fig. 10.23(b). Node 4 was already pointing away from node 6, so its direction happens to stay the same.

10.4.5 Top-Level Cache Logic

Fig. 10.24 shows the complete top-level cache module, providing more detail than the block diagram of Fig. 10.11, including the cacheway,

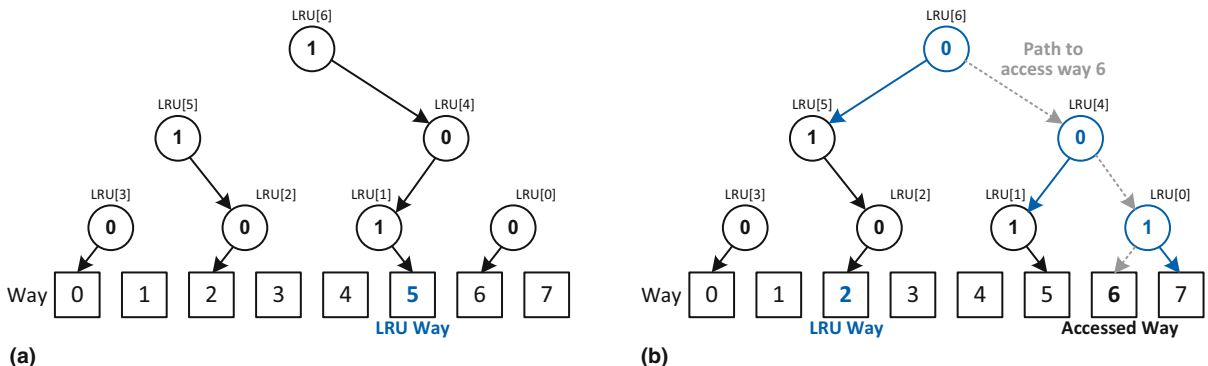


Fig. 10.23 pLRU tree (a) before and (b) after accessing way 6.

`cache fsm`, and `cache LRU` modules described in previous sections. This section describes the shaded logic in [Fig. 10.24](#).

- ▶ **Address selection:** Selects `CacheSet{Tag/Data}` from `NextAdr`, `PAdr`, or `FlushAdr`.
- ▶ **Way selection:** Delivers the tag, read cache line, dirty, hit, and valid bits from the selected way using a distributed AND-OR read mux.
- ▶ **Read selection:** Selects a word from the cache line for reads and writebacks.
- ▶ **Write selection:** Merges `WriteData` into `FetchBuffer` (using the bottom `LineWriteData` mux) during write misses and forms the byte mask for the line (`DemuxedByteMask`).
- ▶ **Bus logic:** Selects the address to send to the bus for fetches, writebacks, and flushes.
- ▶ **Flush logic:** Sequences through the lines and ways during a cache flush.

`FetchBuffer`, `BeatCount`, and `SelBeatCount` come from the AHB cache interface, which is described in [Section 12.2](#). [Section 12.5](#) gives examples of how the cache and bus transfer a burst of consecutive words during cache line fetches and writebacks.

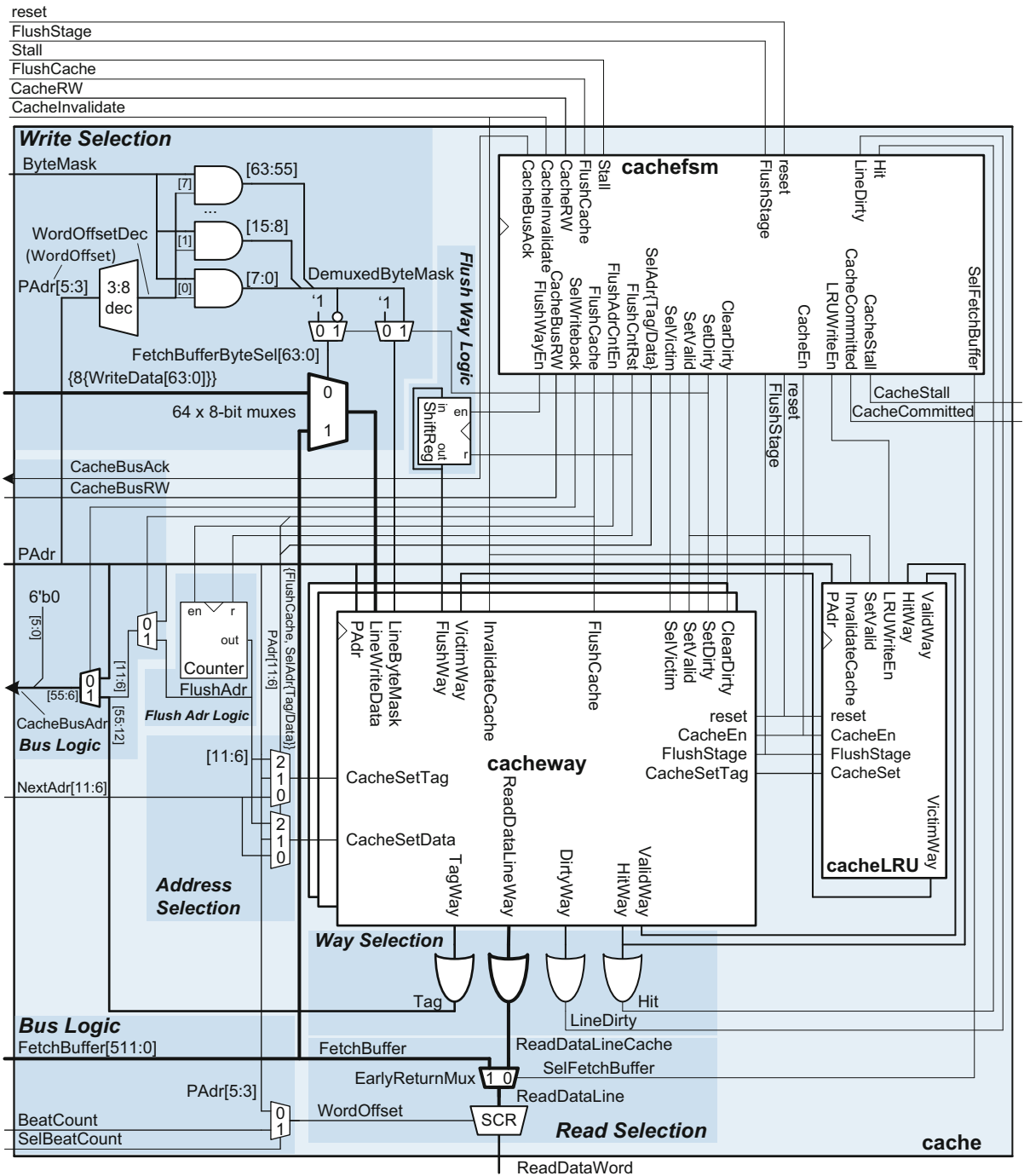
10.4.5.1 Address, Way, and Cache Line Read

`CacheSetData` addresses the data array and `CacheSetTag` addresses the tag and status arrays. Independent addressing of the tag/status and data arrays allows for back-to-back stores without stalling as described in [Section 10.4.2.2](#). During a flush, both are driven from `FlushAdr` to sequence through cache lines.

The read multiplexer that produces the cache line, `ReadDataLineCache`, is a distributed AND-OR multiplexer, as shown [Fig. 10.25](#). Recall from [Section 10.4.2](#) that, at most, one cache way outputs a nonzero cache line (`ReadDataLineWay`). These way outputs are ORED together to produce the cache's read line (`ReadDataLineCache`). The AND stage of the AND-OR mux is internal to each `cacheway` module (shown in [Fig. 10.13](#)), and the OR stage is part of the Way Selection logic.

Benefits of the distributed AND-OR mux over a traditional mux include its ability to directly use the one-hot `HitWay` signals (at most one `HitWay` signal is hot (1) across all ways) instead of having to encode that signal and its reduced power because at most one `ReadDataLineWay` bus is nonzero. The tag (`Tag`) and dirty bit (`LineDirty`) are similarly read using a distributed AND-OR multiplexer.

Distributed AND-OR muxes will appear again in the TLB (translation lookaside buffer) in [Sections 11.4.1.3.1-11.4.1.3.3](#).



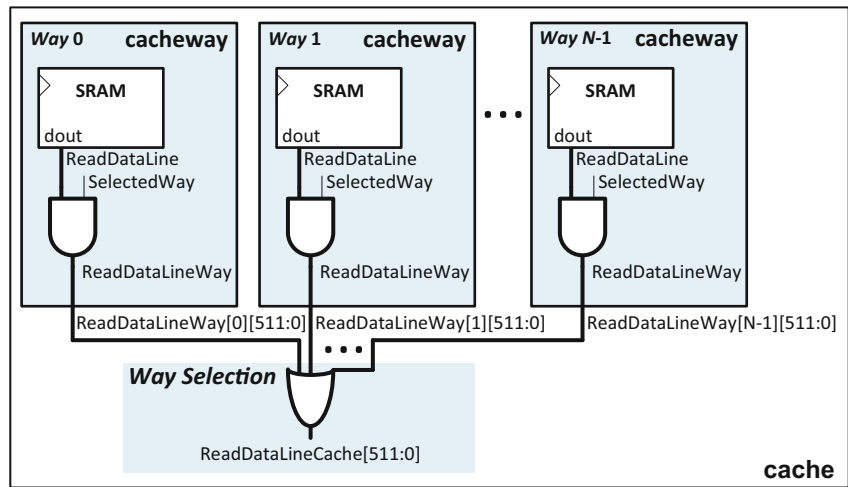


Fig. 10.25 Distributed AND-OR cache line selection multiplexer.

10.4.5.2 Write Selection Logic

The Write Selection logic computes `LineWriteData` and `LineByteMask` for the cache ways. During a store hit, the Write Selection logic replicates `WriteData` into each word of `LineWriteData`. Meanwhile, the logic uses the word offset bits of `PAdr` and the `ByteMask` to form `LineByteMask`, that indicates which bytes of the line should be written, as shown in Figs. 10.5 and 10.13.

During a cache line fetch, `LineWriteData` comes from `FetchBuffer`. On a write miss, `FetchBufferByteSelect` merges `WriteData` into the appropriate bytes of `LineWriteData`. The logic then asserts all bits of `LineByteMask` to write the entire (possibly merged) line into the cache.

10.4.5.3 Read Selection and Bus Interface Logic

The bus interface logic, labeled Bus Logic in Fig. 10.24, contains a `CacheBusAdr` address generation multiplexer (depicted as two separate multiplexers) to produce the `CacheBusAdr` that it sends to the `ahbcacheinterface` during a bus request. Table 10.3 shows how `CacheBusAdr` is selected using the multiplexer. All addresses are aligned to the cache line, so the bottom 6 bits are 0 in a 64-byte cache line. During a fetch, the upper bits come from the `PAdr` of the access causing the cache miss. During a writeback or flush, the evicted line's tag and set are used; the cache set is found in the line address bits of `PAdr` during writeback or from the `FlushAdr` counter during flush.

Table 10.3 CacheBusAdr mux operation.

{FlushCache, SelWriteback}	CacheBusAdr	Description
00	{PAdr[55:6], 000000}	Fetch
01	{Tag, PAdr[11:6], 000000}	Writeback
10	{Tag, FlushAdr, 000000}	Flush

The Read Selection logic selects the word within the cache line (see Fig. 10.24). This logic uses a sub-cache line read (SCR) mux and an early return mux to produce ReadDataWord for three cases:

- ▶ **Read hit:** The SCR mux selects a word within the cache line using the lower bits of the address: $\text{WordOffset} = \text{PAdr}[5:3]$.
- ▶ **Read miss:** The early return mux selects a word from the fetch buffer ($\text{SelFetchBuffer} = 1$) instead of the cache ways during a read miss. This saves a cycle at the end of the miss instead of waiting for the fetched cache line to be written to the cache before reading it.
- ▶ **Writeback:** The cache must writeback the cache line to external memory word by word. Thus, `ahbcacheinterface` asserts `SelBeatCount` so the SCR mux uses $\text{WordOffset} = \text{BeatCount}$ to output each word of the cache line in sequence.

This is discussed further in Sections 12.2.1 and 12.2.2.

10.4.5.4 Cache Flush Logic

As mentioned in Section 10.4.2.6, `fence.i` flushes the D\$. A cache flush writes all the dirty lines back to the next level of the memory hierarchy and clears the dirty bits but leaves the valid bits unchanged. This involves sequencing through each line of each way and reading the data, tag, valid, and dirty bits. If the line is valid and dirty, the data is written back to an address specified by the tag, and the dirty bit is cleared.

Fig. 10.26 highlights the flush logic, which contains two counters to sequence through the lines and ways. The `FlushAdr` counter counts through the lines in binary. The `FlushWay` counter uses a one-hot shift register to count through the ways. System reset initializes both counters, `FlushAdr` to 0 and `FlushWay` to 0...01. `FlushWayFlag` and `FlushAdrFlag` are inputs to the cache FSM. `FlushWayFlag` asserts when all ways in a given set have been flushed, and `FlushAdrFlag` asserts while flushing the last set in a cache. So, the entire cache has been flushed when $\text{FlushFlag} = \text{FlushWayFlag} \ \& \ \text{FlushAdrFlag}$ asserts.

During a flush, the cache FSM cycles through all ways of each successive set and writes back any dirty lines. A flush begins when the IEU

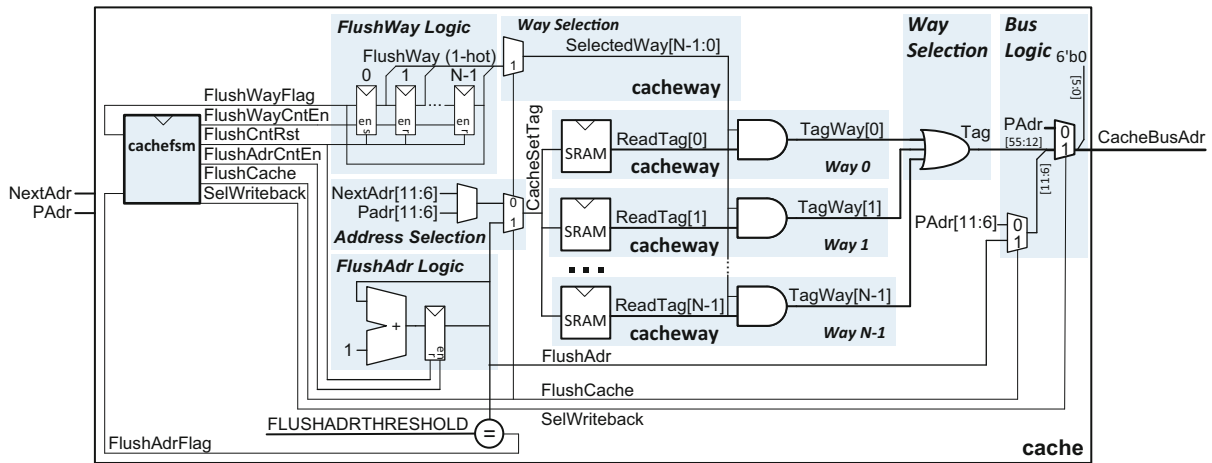


Fig. 10.26 Flush address generation.

asserts **FlushCache**, which causes the cache FSM (see Fig. 10.19) to transition from the Access to Flush state. During the transition, **FlushWay** = 1 and **FlushAdr** = 0, so the next cycle reads out way 0 set 0. The cache FSM asserts **FlushCache**, which chooses **FlushWay** (see Fig. 10.13) to select the 0th way. If that cache line is valid and dirty, **LineDirty** is asserted and **cachefsm** transitions to the FlushWriteback state. Concurrently, the cache computes the cache line's address by concatenating the line's **ReadTag** with the set bits from **FlushAdr** and appending zeros for the word and byte offset while asserting **CacheBusRW[0]** to initiate a bus write. The bus interface will increment the word offset to write back the dirty cache line word by word. When the bus completes the transaction by asserting **CacheBusAck**, **cachefsm** returns to the Flush state, where it increments **FlushWay** (by asserting **FlushWayCntEn**) to move onto flushing set 0 in way 1.

After the FlushWriteback state completes, that cache line will no longer be dirty, so **cachefsm** asserts **ClearDirty** on the transition from Flush to FlushWriteback. The dirty bit is cleared before data is written to the bus to minimize the number cycles spent on writeback. **ClearDirty** must be asserted for the current way and set (as indicated by **FlushWay** and **FlushAdr**).

If a line is clean (**LineDirty** = 0), **FlushWay** is incremented by asserting **FlushWayCntEn**. When **FlushWay** reaches the last way (10...0), it wraps back around to the first way (0...01), and **FlushAdr** is incremented to move to the next set by asserting **FlushAdrCntEn**.

The cache flush operation ends when the cache has iterated over all sets and ways, as indicated by **FlushAdr** and **FlushWay** hitting the maximum value. The check occurs in both the Flush and FlushWriteback states. **cachefsm** resets both counters by asserting **FlushCntRst** and then returns to the Access state.

For a single core, clearing the dirty bit before the writeback begins is safe. However, a multicore or multithreaded implementation may use a cache coherency mechanism, which requires the line to be cleared after writeback is completed.

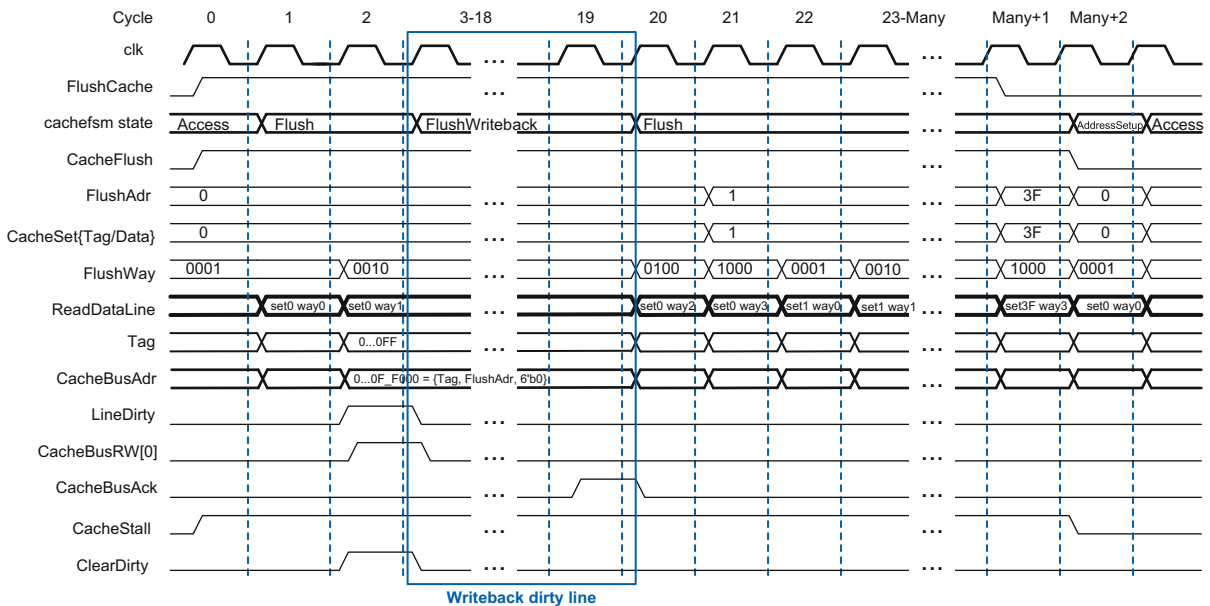


Fig. 10.27 Timing diagram for D\$ flush, showing one dirty line written back.

Flushing the cache is time-consuming because the cache must check each cache line's dirty bit and write back any dirty lines. In the best case, all lines are clean so run time takes (the number of sets $\times$ the number of ways) cycles. The rv64gc configuration has 64 sets and 4 ways for a total of 256 cycles. If all lines are dirty and each writeback takes 15 cycles (in addition to the 1 cycle to check the dirty bit), the D\$ flush will take $64 \times 4 \times (1+15)$ cycles = 4,096 cycles! Fortunately, D\$ flushes are rare events.

Fig. 10.27 shows the timing of a D\$ flush in a 4-way associative cache, showing the details of one dirty cache line written back over the bus.

- ▶ **In cycle 0**, the cache FSM receives `FlushCache`. It sets up `CacheSet{Tag/Data} = FlushAdr` in the `Access` state to read set 0.
- ▶ **In cycle 1**, set 0 way 0 is not dirty, so a writeback is not needed and `FlushWay`, which is a 1-hot signal, is incremented by shifting it to the left.
- ▶ **In cycle 2**, set 0 way 1 is dirty. The FSM asserts `ClearDirty` to reset set 0 way 1's dirty bit. The cache line's address, `CacheBusAdr = 0x00_0000_000F_F000` is computed by concatenating the line's `Tag` with `FlushAdr` and six 0s. `CacheBusRW[0] = 1` to begin a bus write.
- ▶ **In cycle 3**, `cachefsm` transitions to the `FlushWriteback` state where it holds `FlushAdr` and `FlushWay` constant until the bus transaction completes.

FlushAdr sets up which cache set to read (in the Address stage: `CacheSet{Tag/Data} = FlushAdr`), and `FlushWay` selects which way to read (in the Data stage). Recall that cache reads are pipelined, with one in the Data stage while the next is in the Address stage. As the last way is read (`FlushWay = 1000`), the set (`FlushAdr`) also increments, so that the cache sets up to read the new set (e.g., see cycle 21 of Fig. 10.27).

- ▶ In cycle 19, `CacheBusAck` rises to indicate the bus is finished and the `cachefsm` returns to the Flush state and enables `FlushWay` to increment, which occurs in the next cycle.
- ▶ In the remaining cycles, this process repeats until all cache lines are checked and written back if dirty. When `cachefsm` reaches the final set and way (set = $0x3F = 63$, way = 3), the flush is complete, so the address (set) and way counters are reset (to set 0 way 0) and `cachefsm` transitions back to the Access state via the `AddressSetup` state.

10.4.6 Pipeline Stalls

Any time either the I\$ or D\$ misses or performs a cache flush, it asserts `CacheStall` to stall the entire pipeline while fetching or flushing. `CacheStall` remains asserted until the `cachefsm` returns to the Access or `AddressSetup` state. The LSU and IFU, which will integrate the caches in Chapters 12 and 13, assert `LSUStallM` or `IFUStallF` when their `CacheStall` occurs. These in turn stall the pipeline, as shown in Fig. 9.33.

Conversely, the hazard unit stalls the LSU or IFU by asserting `StallW` or `StallD`, respectively. These stalls are renamed to `Stall` as an input to the cache and `ahbcacheinterface` because these modules are instantiated by both the LSU and IFU.

Both caches can simultaneously miss, stalling the entire pipeline. Both cache FSMs concurrently request the missing line or writeback a dirty cache line. However, as discussed in Section 9.5.5, the external bus unit (EBU) gives priority to the LSU (D\$ requests), so the D\$ fetches its missing line first. The D\$ then waits in the `AddressSetup` state, retaining `ReadDataWord`, until the I\$ also fetches its missing line and the pipeline resumes. In `AddressSetup`, the D\$ lowers `CacheStall` to signal the cache is no longer busy and reads from the `FetchBuffer` rather than `cacheway`. While the I\$ is busy, the hazard unit asserts `StallW`, which is connected to the cache's `Stall` input. In the `AddressSetup` state the `cachefsm` only reads the next memory request when `Stall` asserts. This allows `AddressSetup` to behave like the Access State to set up `CacheSetTag` and `CacheSetData` in the Address stage.

10.4.7 Exceptions and Interrupts

At the beginning of a memory operation, the MMU, which is introduced in Chapter 11, checks the validity of the operation and generates an exception if the access is bad. Such exceptions, as well as other exceptions and interrupts, assert `TrapM`, which flushes the entire pipeline including pending cache operations.

A pipeline flush raises `FlushStage` in the I\$ or D\$, which is connected to receive `FlushD` and `FlushW`, respectively. Note that these signals indicate

pipeline flushes, *not* cache flushes. `FlushStage` resets the given cache's `cachefsm` and aborts any ongoing writes to its `cacheway` and `cacheiru`.

Interrupts are asynchronous, arriving at any time, even when the pipeline is stalled for a bus access or cache miss. When a bus transaction is in flight, the bus cannot be interrupted without potentially corrupting memory. Just as the `busfsm` asserts `BusCommitted` when a bus transaction is in progress (see [Section 9.5.5](#)), the `cache` asserts `CacheCommitted` when `cachefsm` is busy (i.e., in the middle of a fetch, flush, or write back). The LSU or IFU raises their `CommittedM` or `CommittedF` signal when either the bus or cache is busy. Pending interrupts are delayed until `CacheCommitted` falls, using the hardware from [Fig. 9.35](#).

10.5 SUMMARY

Microprocessor designers approximate a large, fast, cheap memory with a large, slow, cheap main memory and a small, fast, cheap cache memory. If most memory accesses hit the cache, the AMAT is almost as fast as the cache, but the cost per total bit remains low. In a large system, this generalizes to a multilevel memory hierarchy with caches of varying size and speed.

Caches exploit temporal and spatial locality to obtain a high hit rate: memory locations that have been accessed recently are likely to be used again, and nearby memory locations are also likely to be used. To exploit spatial locality, caches are organized as a collection of lines, commonly 512 bits (64 bytes, or 8 double words) long.

The cache only holds a subset of memory. The middle bits of the requested address (the set bits) select the cache line, and the lower bits select the word within the line. The line also contains a tag as well as valid and dirty bits. If the tag bits match the upper bits of the address and the line is valid, the desired line is present, and the cache hits. Cache misses can be classified as compulsory (first time a line is accessed), capacity (cache is too small to hold the working set), or conflict (multiple addresses map to the same cache line). Long cache lines reduce compulsory misses at the expense of an increased miss penalty (time to transfer a cache line). Larger caches reduce capacity misses at the expense of cost and speed. Associative caches reduce conflict misses by providing multiple ways that can hold a given cache line, at the expense of complexity, cost, and access speed.

When a cache miss occurs, the entire line is transferred into the cache, so a subsequent access to another word in the line will hit. If the set is full, it must evict another line to make room. To exploit temporal locality by holding recently accessed lines, associative caches typically have an LRU or pLRU replacement policy to select the way to evict. If the evicted line is dirty, it must also be written back to memory.

A typical core makes most of its accesses to level 1 instruction and data caches (L1I\$ and L1D\$). The cache size is limited by the desired

access time and is typically 8–32 KiB. Because of interactions with the virtual memory system described in the next chapter, each cache way is no bigger than the virtual memory page size, usually 4 KiB. Therefore, the cache generally has 2, 4, or 8 ways, which is sufficient to eliminate most conflict misses. Experience has shown that a 512-bit line size (64 bytes, or 8 double words) is a good tradeoff between spatial locality and cache line transfer time. Hence, a common L1\$ organization is 16 KiB, built from four 4-KiB ways, each containing 64 64-byte lines, which achieves a miss rate of about 4% on Embench benchmarks. Wally rv32gc and rv64gc configurations have this cache topology. High-performance cores add an L2\$ with about 256-1024 KiB capacity. Multicore systems add a L3\$ with many megabytes of capacity that is also responsible for maintaining coherence when multiple cores are accessing the same data.

Wally uses a common parameterized module for both instruction and data caches. However, instruction caches are read-only, so they eliminate the write, dirty, and writeback logic. The heart of the cache is an array of cache ways. Each way is indexed by the set (middle bits of the address). The tags from each way are compared to the upper bits of the address. The hit logic checks for a valid way with a matching tag. If a hit occurs, a way selection mux picks the cache line from the matching way. For a read hit, the `subcacheline_read` mux picks the word from the line based on the lower bits of the address. For a write hit, the cache writes the line on the next cycle and sets the line's dirty bit. The cache uses a byte mask to specify which bytes in the line to write, depending on the size of the access and the word offset in the low bits of the address.

When a cache miss occurs, the `cachefsm` module sequences through a series of steps. The `cacheLRU` module selects the way to replace using a pseudo-LRU algorithm. If the line is valid and dirty, it must be written back to main memory via the AHB bus. The line transfer involves a sequence of consecutive word transfers organized as a burst. The cache then fetches the missing line into a fetch buffer with another AHB bus burst and finally writes the line into the cache. An early return mux saves a cycle from the miss latency of a read data by returning the read data directly from the fetch buffer rather than waiting for it to be written into the cache.

The cache also has hardware support for flushes and invalidates. Flushes iterate over every line in every way, writing dirty lines back to memory. Cache invalidates clear the cache's valid bits. The `fence.i` instruction enforces coherence between instruction and data caches by flushing the D\$ and invalidating the I\$.

Memory Management Unit

- 11.1 Principles
- 11.2 RISC-V Practices
- 11.3 Test Plan
- 11.4 Wally Implementation
- 11.5 Summary

The *memory management unit* (MMU) is responsible for handling virtual memory and memory protection. This allows the programmer to write a program as if it has a large amount of memory that cannot conflict with memory from other programs and makes it more difficult for malicious programs to steal information from other programs. The RISC-V privileged specification defines the programmer's view of virtual memory, along with two memory protection checkers: the physical memory attributes (PMA) checker and the physical memory protection (PMP) checker. Wally implements these features using two MMUs, one in the instruction fetch unit (IFU) and one in the load/store unit (LSU). Wally also includes a single hardware page table walker (HPTW) to efficiently fetch page table entries (PTEs).

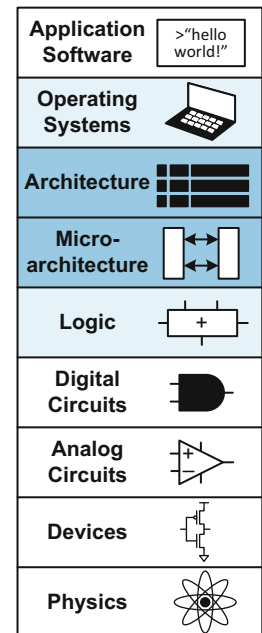
11.1 PRINCIPLES

This section summarizes the broad ideas of virtual memory and memory checking as they apply across computer architectures. [Section 11.2](#) elaborates on RISC-V-specific issues.

11.1.1 Virtual Memory

Virtual memory enables each process to use its entire address space without being concerned about other processes trying to use the same addresses.

In a simple computer system without virtual memory, a process (i.e., a running program) directly specifies the address of main memory that it wishes to access; this is called *physical addressing*. Physical addressing presents a variety of problems for multitasking and large processes, as occurred in early operating systems. Each process must be loaded into a different place in memory. To allow an arbitrary set of processes to run concurrently, each process must be placed in an empty place in memory



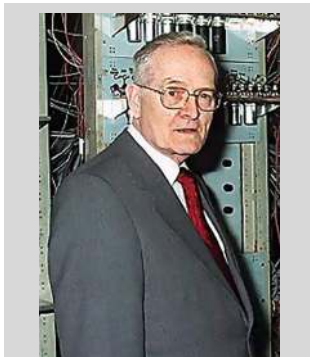
when it is loaded, and all the address references must be changed to reflect that location; this is known as *relocation*. Different processes need different amounts of memory, so when one process finishes, it may leave a chunk of free memory that is too small for another process; this is known as *fragmentation*. A process also might be too large to fit in memory at once. Early systems used burdensome *overlays*, in which the programmer might swap different parts of the program or data set between memory and disk as they were needed. Yet another challenge is that a process might accidentally write to another process's memory and cause both processes to crash, or a malicious process might read the password from another process's memory space. A better OS should provide *protection* to prevent processes from interfering with one another.

Virtual memory is an elegant solution to all these problems. With virtual memory, each process is given its own *virtual address (VA) space*, so the process has an illusion of a large, private, contiguous memory. Therefore, the process can be loaded at an address determined at compile/link time and does not require relocation. Because virtual addresses are private to the process, memory is protected from interference by other processes.

The virtual memory used by each process can be kept on the hard drive. Hard drives provide vast amounts of inexpensive storage, but are painfully slow, so computers use main memory (usually DRAM) to cache the commonly accessed virtual memory. This cache is called *physical memory*. Virtual memory is organized into fixed-sized blocks (conventionally 4 KiB) called *pages* that are analogous to cache lines. The MMU is a hardware block in the processor that performs *address translation*, converting virtual addresses used by the programmer into physical addresses (PAs) sent to the RAM. The OS manages moving pages between virtual (hard drive) and main (DRAM) memory as needed, so the programmer doesn't need to be concerned with overlays. Because pages are of fixed size, fragmentation does not occur. A process may have multiple threads that all share the same virtual address space, so communication between threads within a process is easier than communication between processes. Virtual memory is such a useful programming abstraction that all modern general-purpose operating systems use it.

Virtual memory was invented by Tom Kilburn and his team for the Atlas computer at the University of Manchester [Kilburn62]. Atlas had 16,000 50-bit words of fast magnetic core memory with a 0.4- μ s access time, 96,000 words of lower-cost drum memory on four rotating magnetic drums with a 4- μ s access time, and a large amount of inexpensive storage on magnetic tapes with an average 88- μ s transfer time per word. Now a typical computer has gigabytes of fast DRAM and terabytes of inexpensive hard drive storage.

In 1979, Ozalp Babaoglu, along with Sun Microsystems cofounder Bill Joy, added virtual memory support to Berkeley Unix on the VAX-11/780 computer that was distributed as 3BSD. The Intel 80286



Tom Kilburn at the Manchester Atlas console, 1971. (© 2014.

This image is openly licensed under CC BY-SA 3.0.)

introduced hardware support for virtual memory on x86 processors in 1982, but PC operating systems lagged nearly two decades. Operating systems such as Windows 3.11 (1992) and Mac MultiFinder (1987) used cooperative multitasking without protection where all programs shared the same address space. A bug in one program could crash the entire computer. This was not a rare occurrence. Windows NT supported virtual memory since 1993 but was incompatible with many popular programs. Windows 2000 and Mac OS X brought memory protection and preemptive multitasking to the mainstream personal computer world, making computing much more stable.

11.1.1.1 Pages

Pages of memory are kept in both physical (DRAM) and virtual (hard drive) memory. They are moved from virtual to physical memory as they are needed by a process. Let the *page size* be $P = 2^p$ bytes. Let a virtual address consist of n bits and a physical address consist of m bits, corresponding to $N = 2^n$ and $M = 2^m$ bytes of virtual and physical address space, respectively. n is typically 32 or 64 for 32- or 64-bit systems, but the upper bits may not all be used because, currently, few applications require $N = 2^{64} = 16$ exabytes of memory.

Hence, each process has up to 2^{n-p} virtual pages, and the system has 2^{m-p} physical pages. Like a larger cache line size, a larger page size takes advantage of spatial locality and has less overhead but greater penalties on a miss. Because disks have long latency but decent bandwidth, the page size should be large for highest performance. Most processors and operating systems have standardized on $P = 4\text{-KiB}$ page sizes ($p = 12$ bits). Thus, a computer with $M = 16\text{ GiB}$ of main memory (DRAM) would have $m = 34$ and would be able to hold $2^{34-12} = 2^{22}$ physical pages.

Each virtual page for a process is in one of three states:

- ▶ **Unallocated:** No data is associated with this page, so it does not use any disk space.
- ▶ **Cached:** An allocated page is currently cached in physical memory.
- ▶ **Uncached:** An allocated page is currently stored on disk.

Fig. 11.1 illustrates a mapping between virtual and physical pages. In this example, virtual page 0 is uncached and is only found somewhere on the hard drive. Virtual page 2 is cached in physical page 0 and so forth. Cached pages are found in physical memory and on disk, as indicated by the dashed blue lines, with the disk version potentially being out of date. Physical pages may also be empty, not holding any virtual page yet, as shown.

The MMU uses a *page table* to perform address translation between virtual and physical pages, which are uniquely identified by virtual and

Some systems use the terms *resident* and *nonresident* instead of cached and uncached pages.

The area on the hard drive reserved for uncached (nonresident) virtual pages is called *swap space*.

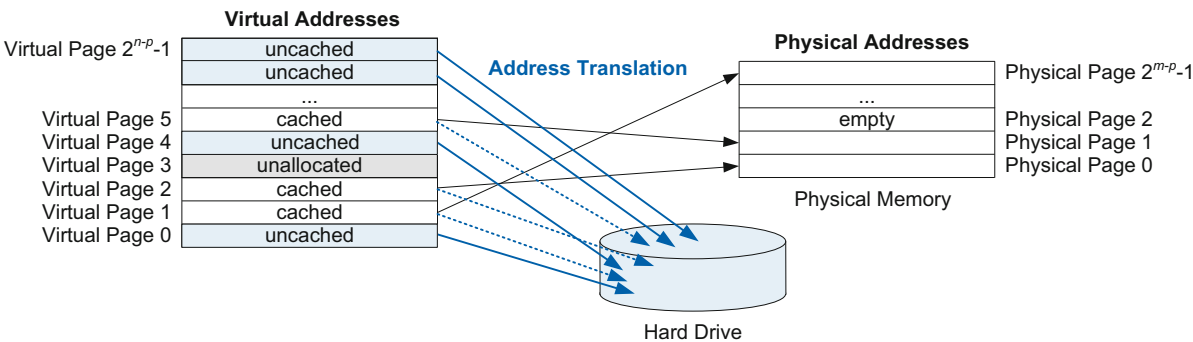


Fig. 11.1 Mapping between virtual and physical pages.

physical page numbers (VPNs and PPNs), respectively. The page table has one *page table entry* (PTE) for each virtual page. Each PTE contains status bits to indicate the state of the virtual page (unallocated, cached, or uncached) and address bits that specify the physical page number, if cached, or information for the OS to find the location on disk, if uncached. The upper $n-p$ bits of the virtual address specify the *virtual page number* (VPN) and hence are used to index the page table. If the PTE indicates the cached state, then a *page hit* occurs and the MMU reads the $(m-p)$ -bit *physical page number* (PPN). The bottom p bits, called the *page offset*, are the same for virtual and physical addresses and hence need no translation. The physical address is formed by concatenating the PPN and the page offset. Fig. 11.2 shows a page table for the mapping from Fig. 11.1, with 32-bit virtual addresses, 34-bit physical addresses, and 4-KiB pages.

Example 11.1 VIRTUAL TO PHYSICAL ADDRESS TRANSLATION

Suppose a program accesses virtual address 0x00005028. At what physical address will the data be found? Use the page table from Fig. 11.2, with $n = 32$, $m = 34$ and $p = 12$.

Solution: As shown in Fig. 11.3, the VPN is the upper $32 - 12 = 20$ bits of the virtual address: 0x00005. The page offset is the bottom 12 bits, 0x028. According to the page table, virtual page 5 maps to physical page 1. Hence, the 34-bit physical address is 0x000001028.

11.1.1.2 Page Faults

If a process accesses a virtual page that is in the uncached or unallocated state, then a *page fault* occurs, and the system must fetch the page from disk. The processor takes a *page fault exception*. The OS handles the exception by reading the desired virtual page from disk, placing it into an empty physical page in main memory, and updating the PTE to the cached state with the proper PPN. The exception then returns to the

VPN	PPN	Status
$2^{n-p} - 1$		Uncached
$2^{n-p} - 2$		Uncached
...	...	...
5	000001	Cached
4		Uncached
3		Unallocated
2	000000	Cached
1	3FFFFFF	Cached
0		Uncached

Fig. 11.2 Page table for mapping from Fig. 11.1.

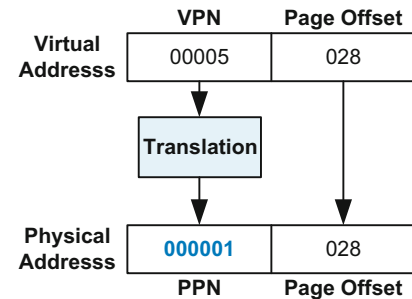


Fig. 11.3 Virtual to physical address translation.

interrupted instruction, which executes again and gets a page hit on the second try.

If the physical memory is completely full, the OS must choose a physical page to replace. If this *victim* page is *dirty*, the OS must write it back to disk. As in a regular cache, one might choose the least recently used (LRU) page. But determining true LRU is expensive, so the system may use a heuristic that approximates LRU. The PTEs typically contain additional state to keep track of whether the page is dirty or has been accessed recently.

11.1.1.3 Protection

Computers are constantly beset by malicious or buggy software. Virtual memory provides a convenient form of memory protection. Page table entries usually have additional bits to indicate whether a page is writable or executable and what privilege level(s) can access it. For example, code pages should not be writable, and data pages should not be executable so that a virus can't use a buffer overflow attack to inject bad instructions. The page tables must not be writable in user mode, lest the program rewrite its own page table to change this protection or to access another program's physical memory.

11.1.1.4 Multilevel Page Tables

The full page table described above can be unreasonably large. Because most processes use only a tiny fraction of the virtual address space, most PTEs contain unallocated pages and are unnecessary.

Example 11.2 1-LEVEL PAGE TABLE SIZE

Determine the size of the page table for $n = 32$ -bit and $n = 64$ -bit virtual addresses with 4-KiB pages.

Solution: The page table has 2^{n-p} entries, with $p = 12$. So, the number of entries is $2^{(32-12)} = 2^{20}$ for $n = 32$ and $2^{(64-12)} = 2^{52}$ for $n = 64$. Each PTE must be large enough to hold the PPN and some status bits and, as a practical matter, should be word aligned. Thus, each PTE is typically one word (4 bytes in a 32-bit system and 8 bytes in a 64-bit system). Hence, the page table size is $4 \text{ bytes} \times 2^{20} = 4 \text{ MiB}$ in the 32-bit system and $8 \text{ bytes} \times 2^{52} = 32 \text{ PiB}$ in the 64-bit system.

Note that the *root* (top-level) page table has the largest level number, whereas the lowest-level page table, which contains PPNs, is level 0.

Large page tables are inefficient or impossible to implement, so practical systems use multilevel page tables. In a 2-level page table, the upper bits of the virtual address are used to choose an entry from a level 1 page table, which points to a level 0 page table. The lower bits of the virtual address are used to index into the level 0 page table and choose a PTE, which contains the PPN and status bits. If only a small part of virtual memory is used, most of the level 0 page tables contain only unallocated pages. In such a case, the level 1 PTE is marked NULL (not valid), and the level 0 page table does not even exist.

For large virtual memory systems, this can be generalized to more levels. Each page table itself is typically chosen to fill one page. Only the top-level page table need be in main memory at all times. Some pages of the lower levels could themselves be uncached and loaded from secondary storage only upon demand.

Example 11.3 2-LEVEL PAGE TABLE SIZE

Consider the 2-level page table in Fig. 11.4 for $n = 32$ with 4-KiB pages. The virtual address is shown at the top: the most significant 10 bits index the level 1 page table, the next 10 bits index the level 0 page table, and the least significant 12 bits determine the page offset. Suppose the virtual memory has 2,048 (2^{11}) pages allocated at the bottom of the address space for code and an upward-growing heap, then a large gap, then one page for a downward-growing stack, and nothing above. Determine the size of the page table.

Solution: The level 1 page table occupies 1 page (4 KiB) because it contains 2^{10} 4-byte entries. Similarly, each level 0 page table occupies 1 page. With only 3 active level 0 page tables (2 for the 2^{11} virtual pages of code and heap and 1 for the single page allocated to the stack), the overall page table size is $4 \text{ pages} = 4 \times 4 \text{ KiB} = 16 \text{ KiB}$.

11.1.1.5 Translation Lookaside Buffer

A virtual memory system with a k -level page table adds k page table accesses to every memory access, which reduces performance dramatically. The MMU mostly avoids this penalty by using a small cache of recent virtual-to-physical page number translations, called a *translation lookaside buffer* (TLB). A small (e.g., 128-entry) fully associative TLB can achieve a

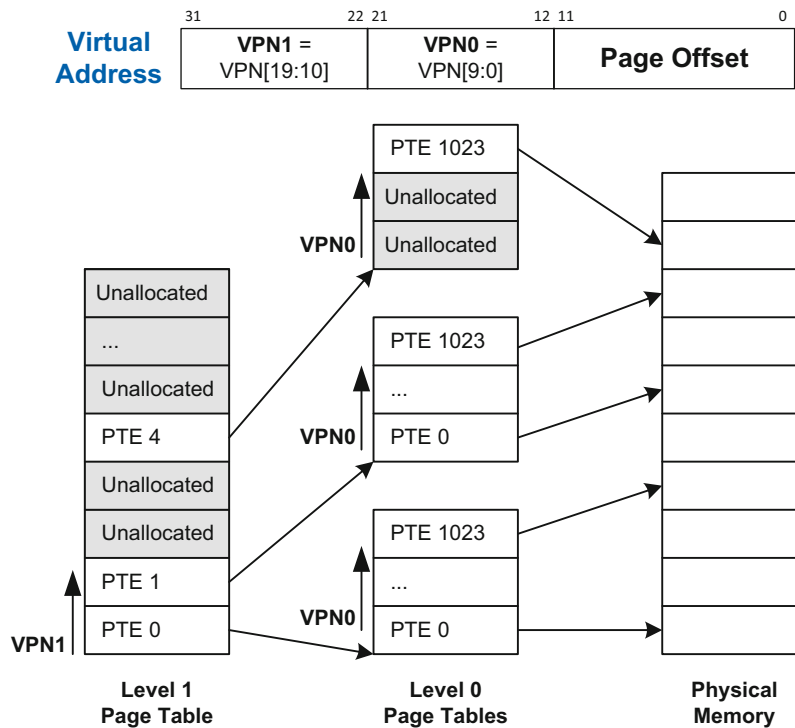


Fig. 11.4 2-level page table.

hit rate of better than 99% on many applications, and a larger TLB may achieve 99.99%. Some systems use a smaller first-level TLB with single-cycle access and a larger second-level TLB with high hit rate. For example, the AMD Opteron processor uses a 2-level TLB with 40 entries in the fast first level and 512 entries in the second level with 7-cycle latency [Hennessy19].

Historically, processors would take an exception on a TLB miss. The exception handler would then issue a sequence of page table reads to traverse the multilevel page table to reach the PTE and then put this entry in the TLB and return to the program. This is slow to do in software, so most modern processors have *hardware page table walkers* (HPTWs), which are state machines that issue the page table reads.

11.1.1.6 Variable Page Size

A program that accesses widely separated portions of a large data structure may touch many different pages, causing the TLB to run out of capacity and frequently miss. Thus, most architectures support more than one page size. For example, x86-64 supports 4-KiB, 2-MiB, and 1-GiB pages. Pages bigger than 4 KiB are called *large pages*, *huge pages*, or *superpages*.

These larger pages save space in the TLB and in the page tables. They make efficient use of disks or networks that achieve peak bandwidth when transferring large contiguous pages. However, the latency of paging these superpages in and out is high. Also, the OS must make a more complex decision of whether to allocate one superpage or many smaller pages.

11.1.1.7 Interaction of Processes, Virtual Memory, and Caches

Each process has its own virtual address space. Therefore, a particular virtual address in two different processes usually maps to a different physical address. The OS manages this by maintaining separate page tables for each process. The MMU keeps a pointer to the active process's page table.

The TLB must adjust translations for different processes as well. One solution is to flush the TLB on each context switch, but this incurs a performance penalty reloading the TLB. A higher-performance design adds a *process ID (PID)* field to each TLB entry so that translations for multiple processes can coexist. Some parts of memory, such as the OS or I/O peripherals, may belong to a *global address space* and thus be accessible by any process at the same virtual address. The TLB may include a global bit to indicate that these PTEs apply to all PIDs.

More than one process can access shared variables by configuring their page tables to point to a shared physical page. Thus, different virtual addresses can translate to the same physical address.

Programs use virtual addresses, so, ideally, cache accesses also use virtual addresses before they are translated to physical addresses. However, this is complicated because the same virtual address in different processes could correspond to different physical addresses, and different virtual addresses in different processes could *alias* to the same physical address. This is known as the *synonym problem*. The system could flush the cache on every context switch to prevent such aliasing, but this degrades performance. Although some solutions exist with various complications [Cekleov97], most systems avoid the problem entirely by using *virtually indexed, physically tagged* caches with a way size less than or equal to the page size (usually both 4 KiB). A processor accesses the L1 cache using the set bits from the page offset, which are not translated. The TLB is accessed concurrently with the L1 cache. After they are both read, the physical address from the TLB is checked against the physical address tag from the cache line. Page offsets are normally 12 bits, which limits each L1 cache way to 4 KiB. Larger L1 caches use associativity; for example, the Intel Haswell 32 KiB cache with 4-KiB ways is 8-way set associative. This limitation doesn't apply to higher-level caches because the physical address is available by the time an L1 cache miss is discovered.

Increasing associativity costs power (more RAMs to access concurrently) and time (a larger way mux). Some designs find 4-way associativity is a practical limit, restricting physically tagged virtually indexed L1

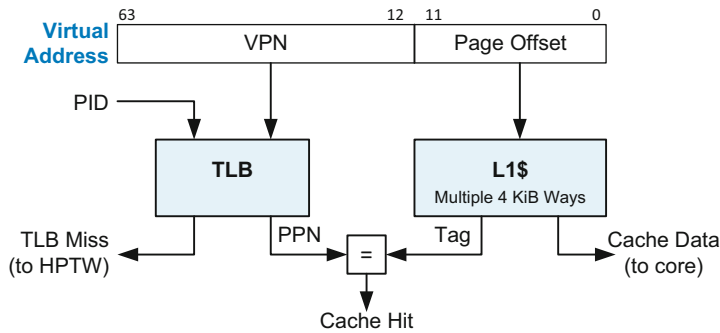


Fig. 11.5 Address translation and caching.

caches to 16 KiB. The AMD Opteron 64 KiB cache is 2-way associative with 32-KiB pages. It addresses the synonym problem with hardware *anti-aliasing* by checking for aliases in all eight possible locations during a cache miss, and invalidating an alias if it is in the cache. Alternatively, some operating systems use *page coloring* to allocate aliased pages to physical addresses with the same least significant bits (e.g., same 16 lsbs in a 2^{16} -byte cache).

Fig. 11.5 shows these interactions. The memory system receives a virtual address, which consists of a page offset and virtual page number. The fully associative TLB is accessed using the VPN and PID and produces the PPN and a TLB miss signal. If a TLB miss occurs, the hardware or software page table walker fetches the translation from the page tables in main memory and updates the TLB before continuing. Meanwhile, the level 1 cache is accessed using the page offset to produce the data and physical tag. The tag(s) are then compared against the PPN to determine whether the cache hit.

11.1.2 Physical Memory Checking

A processor with memory-mapped I/O needs to route physical memory accesses to the appropriate device. For example, a range of physical memory addresses correspond to main memory (DRAM), but other ranges could correspond to a UART (universal asynchronous receiver/transmitter), GPIO (general-purpose I/O), CLINT (Core-Local Interruptor), and PLIC (Platform-Level Interrupt Controller). Table A.2 in Appendix A lists the default memory map for Wally. Reserved addresses outside of mapped ranges should cause an illegal address exception. Moreover, the processor may need to know attributes about each of the devices, such as whether the values are cacheable (yes for memory, no for most peripherals) or whether only certain-sized memory accesses are supported. Some of these addresses or attributes may change when new devices are plugged into the

The DEC PDP-11 was a highly successful minicomputer in the 1970s with 16-bit addressing. Memories soon grew too large for 16 bits, and the PDP-11 architecture did not gracefully evolve to allow wider addresses, so the architecture rapidly became obsolete and was replaced by the even more successful VAX-11. The architects reflected on this error:

“There is only one mistake that can be made in computer design that is difficult to recover from — not having enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every known computer” [Bell76].

Extensions with the *Sv* prefix refer to supervisor-mode features related to virtual memory.

system. Moreover, certain physical addresses may only be accessed in a privileged mode for added security. Hence, most systems require some configurable checking of physical memory addresses in hardware. The details vary significantly between architectures.

11.2 RISC-V PRACTICES

RISC-V supports several flavors of virtual memory. It also supports memory checking through the PMA and PMP checkers. Tables I.14 to I.20 at the back of the book summarize CSRs and PTEs associated with these features. Bad memory accesses produce various types of faults. When a page table is modified, the OS uses `s_fence.vma` to synchronize the TLB with the new page table. When a PTE is not in the TLB, a HPTW traverses the multilevel page table to find the missing entry.

11.2.1 RISC-V Virtual Memory

In RISC-V, machine-level processes exclusively use physical addressing. Virtual memory is an optional RISC-V feature. If supported, supervisor- and user-level processes can enable virtual addressing through the *supervisor address translation and protection* (`satp`) register. RV32 supports 32-bit virtual addresses and 34-bit physical addresses with a 2-level page table. Hence, each process may have up to a 4-GiB address space, but the system may have up to 16 GiB of physical memory. RV64 supports 39-, 48-, and 57-bit virtual addresses with 3-, 4-, and 5-level page tables, respectively, with 56-bit physical addresses. Future expansion is reserved for 64-bit virtual addresses with 6 levels of page tables, should 128 PiB (petabytes) of virtual memory prove insufficient. RV32 and RV64 systems can run in modes that support virtual memory called Sv32, Sv39, Sv48, and Sv57, with numbers corresponding to the virtual address size. Sv`bare` mode, also called simply *bare mode*, indicates that virtual memory is off and address translation is disabled.

11.2.1.1 `satp` Register

The `satp` register enables the various virtual addressing modes and points to the root (level 1) page table for a process. It also contains the *address space identifier* (ASID), which acts as a short process ID (PID) assigned by the OS to differentiate multiple processes' virtual address spaces that might coexist in a TLB. Fig. 11.6 shows the RV32 and RV64 `satp` registers. PPN defines the PPN of the root page table. Shift it left by 12 to get the physical address (base address) of the root page table, which is always aligned to a page boundary. ASID is 9 or 16 bits. In RV32, `MODE` is 0 to disable virtual memory (Sv`bare` mode) or 1 to enable it (Sv32 mode with

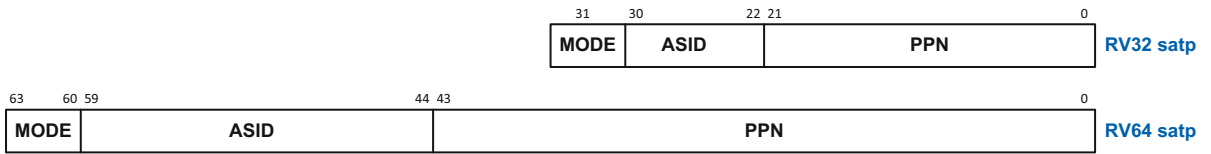


Fig. 11.6 Supervisor address translation and protection register (*satp*).

32-bit virtual addresses). In RV64, *MODE* is 0000 for Sv_{bare}, 1000 for Sv39, 1001 for Sv48, or 1010 for Sv57.

Each user or supervisor-mode process has its own virtual memory space and ASID. When the OS performs a context switch, it reloads *satp* with the page table address and ASID of the new process.

Older x86 processors without an ASID flushed the TLB on a context switch.

Example 11.4 2-LEVEL PAGE TABLE PHYSICAL ADDRESS ACCESSES

Consider again an Sv32 system with a 2-level page table organization shown in Fig. 11.4. Suppose *satp* = 0x80012345, and the virtual address being accessed is 0x013FF124. The PTE is not in the TLB. Determine all physical addresses that are accessed.

Solution: Fig. 11.7 depicts this access.

- ▶ The *satp* register's msb (*MODE* bit) is 1, so virtual memory is enabled. The next 9 bits, which are 0, give the ASID, which is unique to the process.
- ▶ The *satp*'s remaining 22 bits are the PPN of the root (level 1) page table: 0x012345. Append a page offset number of zeros (12 zeros) to obtain the 34-bit physical address (base address) of the level 1 page table: 0x012345000.
- ▶ From the virtual address, VPN1 = 4, which is the index into the level 1 page table. So, 0x012345010 is the address of the needed level 1 PTE. Specifically, 0x01234500 + 4 words (16 bytes) = 0x012345000 + 0x10 = 0x012345010.
- ▶ The level 1 PTE gives the PPN of the level 0 page table. In this case, it is 0x1AABBC. Append a page offset number of zeros (12 zeros) to get the physical address of the level 0 page table: 0x1AABBC000. VPN0 (0x3FF = 1023) is the index into the level 0 page table, so the level 0 PTE is at physical address 0x1AABBC000 + (0x3FF << 2) = 0x1AABBCFFC.
- ▶ That entry gives the PPN of the physical memory address: 0x377889.
- ▶ Append the page offset of 0x124 from the virtual address to get the final 34-bit physical address of 0x377889124.

This chapter (and the Wally implementation) focus on virtual memory systems up to Sv48. The December 2021 release of the RISC-V specification added Sv57 and the implementation is a straightforward extension but only relevant to systems with exceptional memory requirements.

11.2.1.2 Virtual Address and Page Table Structure

Fig. 11.8 shows the virtual address formats for Sv32, Sv39, and Sv48. In all formats, the bottom 12 bits, which are not translated, give the page offset for a 4-KiB page. Whereas in RV32 (Sv32), the PTEs are 32 bits = 4 bytes each, in RV64 (Sv39/48), the entries are 64 bits = 8 bytes each. Each page table fits on one page, so RV32 has $2^{12}/4 = 2^{10}$ entries in a page table, and RV64 has $2^{12}/8 = 2^9$ entries. Hence, in Sv32, the upper 10 bits of the

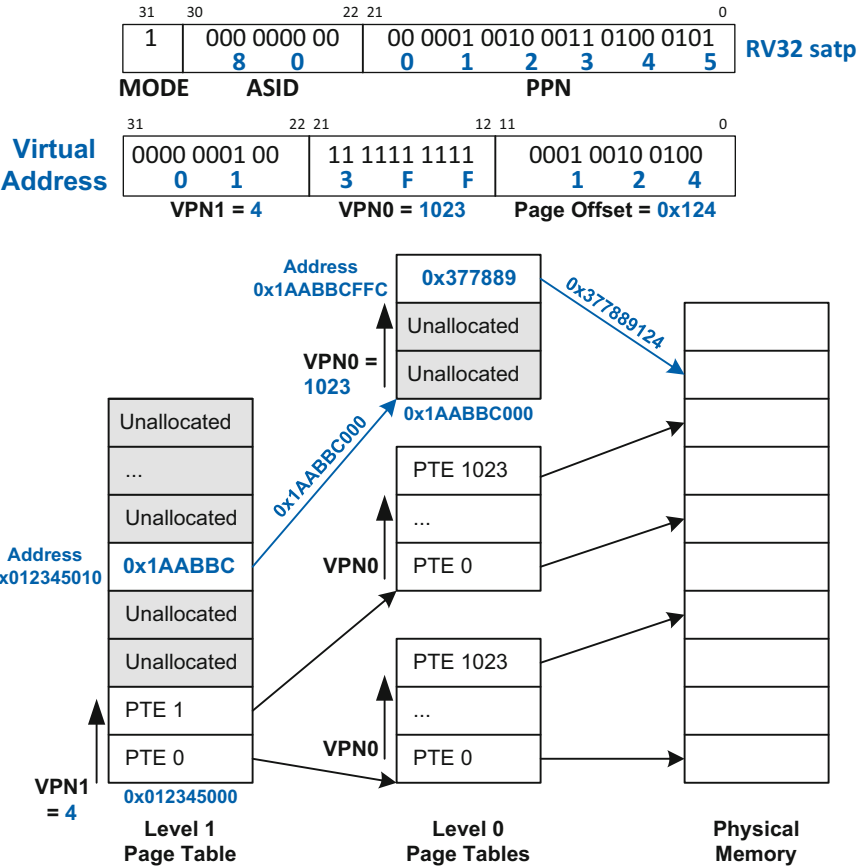


Fig. 11.7 Physical addresses of 2-level page table.

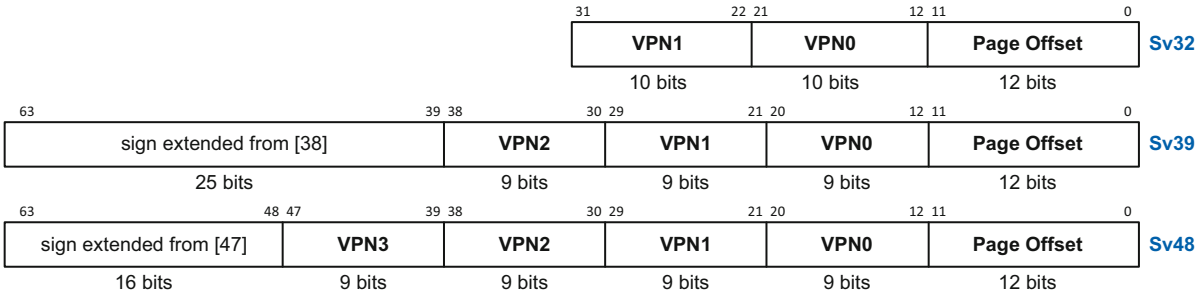


Fig. 11.8 RISC-V virtual address formats.

virtual address index into a level 1 page table, which then points to the level 0 page table. The remaining (middle 10) bits of the virtual address index into the level 0 page table, which holds the PPN. In Sv39, the upper 9 bits of the virtual address index into a level 2 page table, which points to the base address of the level 1 page table. The next 9 bits give the index into the level 1 page table, which then points to the level 0 page table. The next 9 bits index into the level 0 page table, which holds the PPN. Sv48 extends to a level 3 top-level page table. The virtual address space is $2^{32} = 4$ GiB for Sv32, $2^{39} = 512$ GiB for Sv39, and $2^{48} = 256$ TiB for Sv48. Sv39 is sufficient for many contemporary systems, uses less page table storage, and has faster page table walking, so it is preferred unless the larger address space is required. For RV64 (Sv39 and Sv48), the upper bits of the virtual address must all be the same (either all 1s or all 0s) so that the address space includes the bottom and top of the range but leaves non-existent addresses in the middle; if the bits are not the same, a page fault occurs. This matches conventional practices, where user-mode programs occupy low addresses, while the supervisor-mode OS occupies high addresses, and the OS can easily distinguish between the two by looking at the most significant bit of the virtual address.

Fig. 11.9 shows the PTE formats for RV32 and RV64. Both formats contain the same 10 flag bits. In RV32, the PPN is in the upper 22 bits. Hence up to $2^{22+12} = 16$ GiB of physical memory is supported. In RV64, the PPN is 44 bits, so up to $2^{44+12} = 64$ PiB of physical memory is supported.

The flags are described in Table 11.1. The valid (V) bit is set if the page is in main memory. Otherwise, the MMU triggers a page fault, which causes the OS's *page fault handler* to load the page from disk. Collectively, U, X, W, and R provide protection. X, W, and R indicate if the page is executable, writable, and readable. A page fault occurs if the privilege or permissions are incorrect for the access attempted. In user mode, a fault occurs for any access if U = 0. In supervisor mode, if U = 1, a fault always occurs when trying to execute code; reads and writes are allowed only when sstatus.SUM (supervisor user memory) = 1. This protects the system from certain security vulnerabilities. The G bit indicates that a page is global and applies to all processes; the ASID is ignored for global pages. OS pages are typically global because the OS is called from many different user-mode processes.

The Svnaptot (naturally aligned power of two) and Svpbmt (page-based memory types) RISC-V extensions for RV64 use bits 63 and 62:61, respectively, to allow for larger page sizes (see Section 11.2.1.3.1) and to apply attributes to memory pages (see Section 11.2.2.1.1).

The OS sets sstatus.SUM to read or write user-mode pages. But it normally keeps SUM low to avoid accidentally trampling user programs.

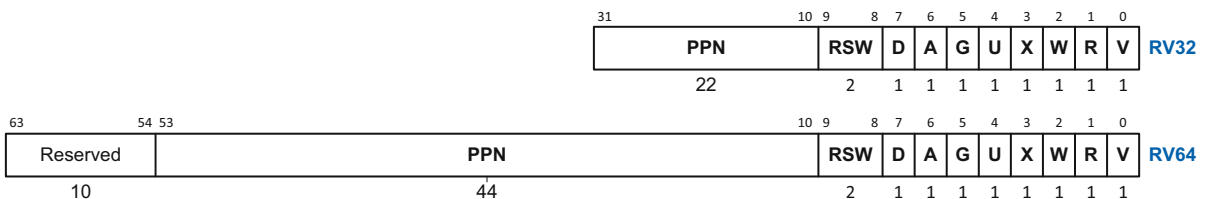


Fig. 11.9 RISC-V PTE formats.

Table 11.1 PTE flags.

Flag	Meaning
V	Valid: 1 if the page is in main memory, 0 if not
X, W, R	Execute, Write, Read Permissions: 000: Pointer to next level of page table 001: Read only 010: <illegal> 011: Read-write 100: Execute only 101: Read-execute 111: Read-write-execute
U	User mode
G	Global: page exists for all address spaces; ignore ASID in TLB matching
A	Accessed: page has been accessed at least once since all pages' A bits were last cleared
D	Dirty: at least one byte from the page has been written
RSW	Reserved for OS

The OS reserves part of the disk for *swap space* to hold pages that don't fit in physical memory. The OS uses the PPN field of the PTE to locate the page on disk. Paging from disk is slow, so if a computer doesn't have enough RAM to hold the working set, it will begin *thrashing*, frequently page faulting and reducing the system to a crawl. Mobile devices generally terminate an entire app that hasn't been used recently to reclaim its memory rather than enduring the delay and wear and tear of paging to flash memory.

When two processes wish to communicate through shared memory, they allocate a shared page through a system call such as `shmget()`. Each process has its own virtual addresses mapping to the shared physical addresses. Thus, each process also has its own PTE pointing to the shared PPN. The PTE flags could differ between processes; a producer-consumer buffer might be read-only for the consumer but read/write for the producer.

Table 11.2 RISC-V page sizes.

Format	Leaf Page Table Level	Page Offset Bits	Page Size	Page Type
Sv32	0	12	4 KiB	Kilopage (normal)
Sv32	1	$12 + 10 = 22$	4 MiB	Megapage
Sv39	0	12	4 KiB	Kilopage (normal)
Sv39	1	$12 + 9 = 21$	2 MiB	Megapage
Sv39	2	$12 + 9 \times 2 = 30$	1 GiB	Gigapage
Sv48	0	12	4 KiB	Kilopage (normal)
Sv48	1	$12 + 9 = 21$	2 MiB	Megapage
Sv48	2	$12 + 9 \times 2 = 30$	1 GiB	Gigapage
Sv48	3	$12 + 9 \times 3 = 39$	512 GiB	Terapage

11.2.1.3 Variable Page Sizes

RISC-V supports multiple page sizes. Usually, the level 0 page table is the leaf page table and points to normal 4-KiB pages (also called *kilopages*). The nonleaf PTEs use $\{X, W, R\} = 000$ to indicate that they point to the next level page table. (For these pointer PTEs, the D , A , and U bits are reserved and must be 0.) However, if XWR are nonzero for a higher-level PTE, then that entry points to a superpage, as shown in Table 11.2. The superpage must be aligned to its natural (page) boundary (with all the low-order bits being 0). The low-order untranslated bits of the virtual address form an offset into the superpage, which are also referred to by their more specific names of *megapage*, *gigapage*, or *terapage*. Megapages are 4 MiB in RV32 or 2 MiB in RV64. Gigapages are 1 GiB, and terapages are 512 GiB, and these are only available in RV64.

Recall that each page table fits into one page (4 KiB = 2^{12} bytes). In Sv32, each entry is 32 bits (4 bytes = 2^2 bytes), so each page table index is 10 bits ($2^{12}/2^2 = 2^{10}$). In Sv39 and Sv48, each entry is 64 bits (8 bytes = 2^3 bytes), so each page table index is 9 bits ($2^{12}/2^3 = 2^9$).

For this reason, megapages are larger in Sv32 because PTEs are 32 rather than 64 bits, giving room for twice as many PTEs in a fixed-size (4-KiB) page table.

Example 11.5 ADDRESS TRANSLATION WITH VARIOUS PAGE SIZES

Consider an Sv48 virtual memory system with the page table in Fig. 11.10(a). (Fig. 11.10(b) shows the page table for an Sv32 virtual memory system.) Find the physical addresses corresponding to virtual addresses (a) 0000_0000_8020_1340, (b) FFFF_FFFF_FFFF_F238, (c) 0000_0000_8000_FFF8, and (d) 0000_0100_C0A0_7BC0.

Solution: Fig. 11.11 shows how the Sv48 virtual addresses are split into fields according to Fig. 11.8.

- (a) $VPN3 = 0$, $VPN2 = 2$, $VPN1 = 1$, $VPN0 = 1$. Hence, entry 0 of the level 3 page table points to the bottom level 2 page table, whose entry 2 points to the bottom level 1 page table, whose entry 1 points to the bottom level 0 page table, whose entry 1 contains the 44-bit PPN 00_0000_8020_0. This is a normal kilopage because it is in the level 0 table. Thus, the page offset is the bottom 12 bits of the address (340), and the 56-bit PA is 00_0000_8020_0340.
- (b) $VPN3 = VPN2 = VPN1 = VPN0 = 511$. Hence, entry 511 of the level 3 page table points to the top level 2 page table, whose entry 511 points to the top level 1 page table, whose entry 511 points to the top level 0 page table, whose entry 511 contains the 44-bit PPN 00_0000_8020_1. This is, again, a normal kilopage, so the page offset is 238, and the 56-bit PA is 00_0000_8020_1238.
- (c) $VPN3 = 0$, $VPN2 = 2$, $VPN1 = 0$. Hence, entry 0 of the level 3 page table points to the bottom level 2 page table, whose entry 2 points to the bottom level 1 page table, whose entry 0 contains the 44-bit PPN 00_0000_9000_0. This is a megapage because XWR is nonzero; thus, the leaf entry is in level 1, so the page offset comes from the bottom 21 bits of the VA (00FFF8). The bottom 9 bits of the megapage PPN are discarded before appending the 21-bit page offset, resulting in a 56-bit PA of 00_0000_9000_FFF8.
- (d) $VPN3 = 2$. Hence, entry 2 of the level 3 page table contains the 44-bit PPN 00_0000_0000_0. This is a terapage because XWR is nonzero in this level 3 page table; thus, the leaf entry is on level 3 and the page offset comes from the bottom 39 bits of the VA (00_C0A0_7BC0). The bottom 27 bits of the PPN are discarded before appending the 39-bit page offset, resulting in a 56-bit PA of 00_0000_C0A0_7BC0.

In the page table examples in this chapter, white-shaded entries are good, gray entries are invalid ($V = 0$), and blue entries are malformed such that they should cause a page fault when used. For example, in Fig. 11.10, level 0 PTE 0 is blue because it is a pointer to the next level page table ($XWR = 000$), which is illegal for a level 0 PTE.

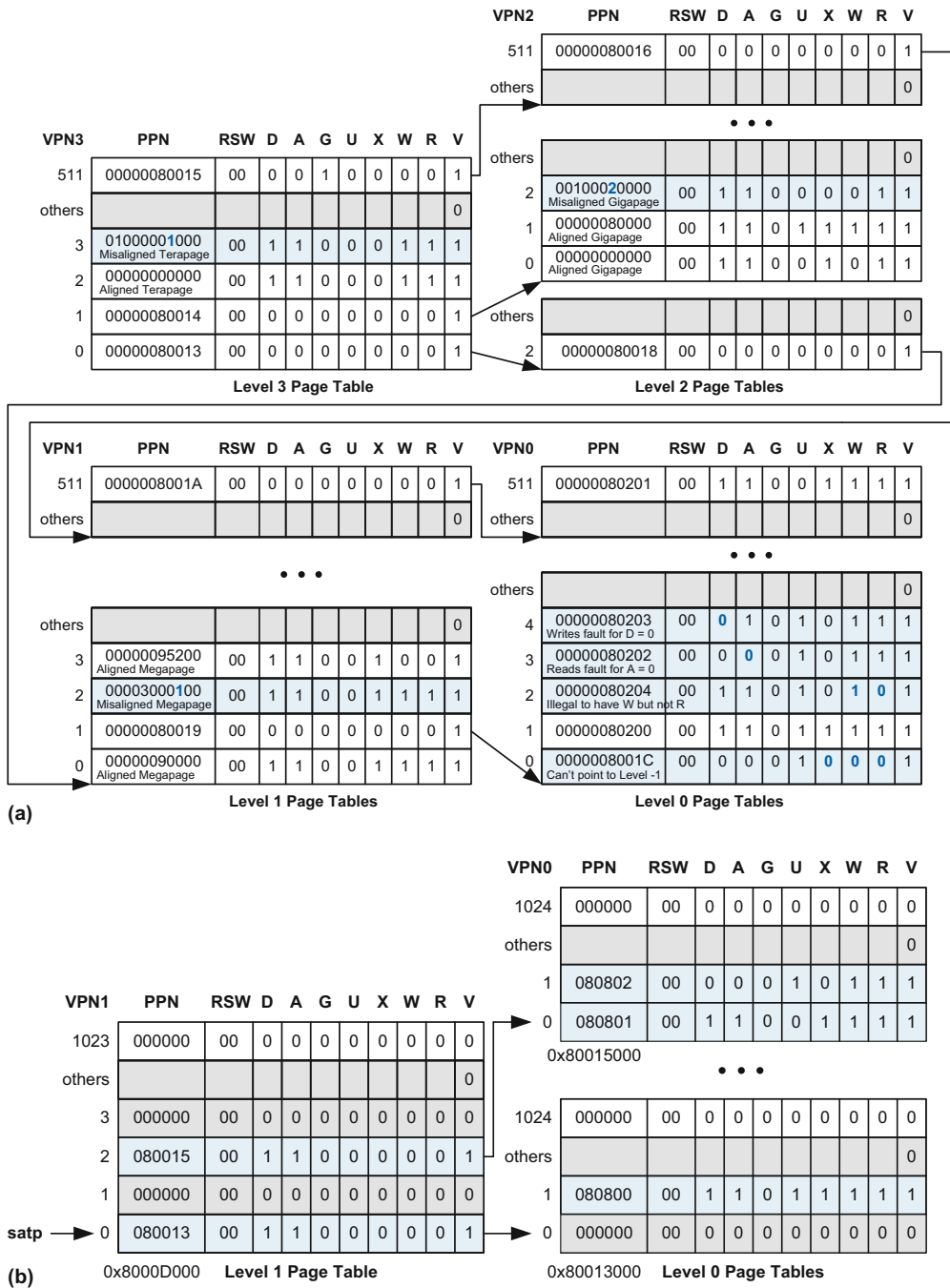


Fig. 11.10 Sample page tables: for an (a) Sv48 and (b) Sv32 system.

	Sign Extension	VPN3	VPN2	VPN1	VPN0	Page Offset
	63 48 47	39 38	30 29	21 20	12 11	0
00000000_80201340	0000 0000 0000 0000	0000 0000 0 = 0	000 0000 10 = 2	00 0000 001 = 1	0 0000 0001 = 1	0x340
FFFFFFFF_FFFFF238	1111 1111 1111 1111	1111 1111 1 = 511	111 1111 11 = 511	11 1111 111 = 511	1 1111 1111 = 511	0x238
00000000_8000FFF8	0000 0000 0000 0000	0000 0000 0 = 0	000 0000 10 = 2	00 0000 000 = 0	0 0000 1111 = 15	0xFF8
00000100_C0A07BC0	0000 0000 0000 0000	0000 0001 0 = 2	000 0000 11 = 3	00 0000 101 = 5	0 0000 0111 = 7	0xBC0

Fig. 11.11 Virtual address fields.

Example 11.6 VIRTUAL AND PHYSICAL ADDRESS MAPPING

Consider again the Sv48 virtual memory system with the page table in Fig. 11.10(a). Sketch the virtual and physical address space and the mapping between them for the valid PTEs.

Solution: Fig. 11.12 shows the virtual and physical address spaces. Recall that virtual addresses are 48 bits, and physical addresses are 56 bits. The VPN3, 2, 1, and 0 fields are populated by tracing the page table entries, and their mapping to the PPN is extracted from the page tables. Note that larger pages can alias with smaller pages; for example, virtual addresses 0x0100_9531_A00C (on a terapage), 0x0080_5531_A00C (on a gigapage), and 0x0000_8071_A00C (on a megapage) all alias to physical address 0x00_0000_9531_A00C, as shown in Table 11.3; the page offsets are in blue, the VPNs are in italics, and the PPNs are in bold black.

11.2.1.3.1 Huge Pages

Many important programs use tens to hundreds of kilobytes of code and data. They are too small to make efficient use of a megapage yet clutter the TLB with large numbers of 4-KiB pages. Competing architectures such as ARMv8 have observed that 64-KiB *huge pages* are valuable, and Linux takes advantage of such pages when they are supported.

The optional RV64 Svnop extension combines sixteen contiguous 4-KiB pages to create 64-KiB huge pages naturally aligned to 64-KiB address boundaries. In general, the extension anticipates any *naturally aligned power of two* (NAPOT) page size (i.e., 8, 16, 32, 64 KiB), but only 64-KiB pages are standardized in Version 1.0 of the extension.

When the Svnop extension is supported, bit 63 of the PTE is the N bit. If a PTE has $N = 1$ and $PPN[3:0] = 1000$, the entry is considered NAPOT. The four least significant bits of the VPN are ignored when accessing the page table, so the 16 contiguous 4-KiB virtual pages all can share the same PTE. After translation, the four bottom bits of the PPN are replaced by these bits of the VPN. If $PPN[3:0]$ has any

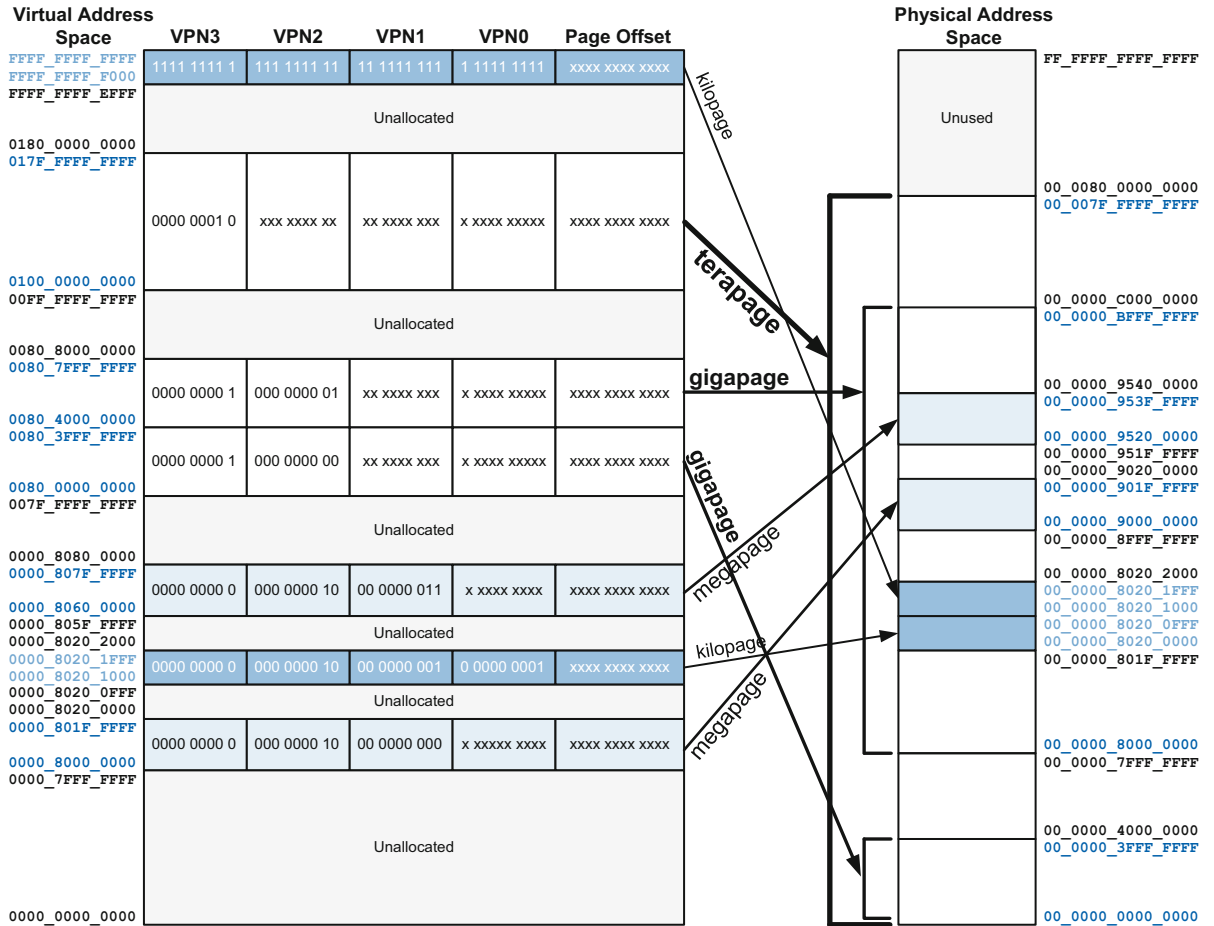


Fig. 11.12 Virtual and physical address spaces.

Table 11.3 Aliasing of virtual addresses to the same physical address.

Virtual and Physical Address Bits	
0000 0001 0000 0000 1001 0101 0011 0001 1010 0000 0000 1100	VA: 0x0100_9531_A00C
0000 0000 0000 0000 1001 0101 0011 0001 1010 0000 0000 1100	PA: 0x0000_9531_A00C
0000 0000 1000 0000 0101 0101 0011 0001 1010 0000 0000 1100	VA: 0x0080_5531_A00C
0000 0000 0000 0000 1001 0101 0011 0001 1010 0000 0000 1100	PA: 0x0000_9531_A00C
0000 0000 0000 0000 1000 0000 0111 0001 1010 0000 0000 1100	VA: 0x0000_8071_A00C
0000 0000 0000 0000 1001 0101 0011 0001 1010 0000 0000 1100	PA: 0x0000_9531_A00C

value other than 1000 in the PTE when PTE bit $N = 1$, a page fault occurs.

Example 11.7 ADDRESS TRANSLATION FOR A HUGE PAGE

Consider a PTE for VPN 0x80020 with $N = 1$ and PPN = 0x80308. Translate virtual address 0x80025678 to a physical address using this PTE.

Solution: This PPN's four least significant bits are 1000, so this is a NAPOT entry for a huge page. The address 0x80025678 has a VPN of 0x80025. The TLB matches against the VPN entry of 0x80020 (by ignoring the bottom four bits) and returns the PPN of 0x80308. It replaces the bottom 4 bits of the PPN (0x8) with those from the VPN (0x5), giving 80305, and then appends the page offset 0x678 to obtain a PA of 0x80305678. Observe that the bottom 16 bits of the virtual and PA are identical, as expected for a $2^{16} = 64\text{-KiB}$ huge page.

11.2.1.4 Dirty and Accessed Bits

The OS uses A/D (accessed/dirty) bits to manage pages. The A bit indicates that the virtual page has been accessed since that bit was last cleared. D indicates that the virtual page has been written since that bit was last cleared. The OS writes dirty pages (with $D = 1$) back to the disk when they are evicted and could use the A bit in a pseudo least recently used (pLRU) page replacement scheme. For example, one strategy is for the OS to periodically mark all pages as unused ($A = 0$). Accessed pages subsequently raise their accessed bits to $A = 1$. When a page needs to be evicted, the OS picks a page that is still marked as unused.

The RISC-V specification indicates that setting these bits can be done in hardware or by triggering a page fault exception to manage the bits in software. Linux supports software management of the A/D bits, so many cores lack hardware support.

The Svade (*supervisor virtual memory A/D exception*) extension indicates that the hart can take a page fault when accessing a page with $A = 0$ or when writing to a page with $D = 0$. The Svadu (*supervisor virtual memory A/D update*) extension indicates that the hart can update the A/D bits of the PTE in hardware without faulting. The Svadu extension also adds `menvcfg.ADUE` (bit 61). When `ADUE = 0`, the Svadu extension is disabled, and the hart takes a page fault to update the PTE in software rather than updating the PTE in hardware.

11.2.1.5 status Register

The `mstatus/sstatus` register (see [Section 8.2.3.1](#)) has three bits that affect address translation: `MPRV`, `MXR`, and `SUM`. These bits manage accesses across privilege levels.

The `MPRV` and `MXR` bits help write efficient trap handlers for misaligned loads and stores. Some RISC-V implementations perform misaligned loads and stores with a trap to a machine-mode handler rather than in hardware because the hardware is costly and rarely used. The handler needs to do address translation even though it is running in machine mode, which usually uses physical addressing only. When set, the `mstatus.MPRV` (**M**odify **P**riVilege) bit causes the system to do address translation and protection according to the privilege level specified in `MPP` (normally the user or supervisor mode that caused the trap). When set, `mstatus.MXR` (**M**ake **e**Xecutable **R**eadable) allows loads from executable nonreadable pages (with $X = 1$ but $R = 0$) so the trap handler can read instructions.

Supervisor code normally should not access information from user-mode pages, to reduce the risk of malicious user-mode programs affecting privileged operations. However, the supervisor occasionally needs to control user-mode data. The `mstatus/sstatus.SUM` (permit Supervisor User Memory access) bit allows these accesses when set; otherwise, supervisor-mode access to user memory causes a load or store page fault. The OS typically sets `SUM` only when necessary and then clears it for safety.

11.2.2 RISC-V Memory Protection

A key requirement for security is to control which processes have access to which parts of memory. RISC-V provides three, often-overlapping, memory protection mechanisms:

- ▶ The PMA checker defines the addresses and access limitations for memory ranges and memory-mapped I/O devices.
- ▶ The PMP checker restricts types and privilege of access to certain memory ranges.
- ▶ Virtual memory provides the `X`, `W`, `R`, and `U` access-permission flags and limits user-mode processes to only access pages explicitly allocated to them by the OS.

PMA and PMP checkers are also simply called the PMA and PMP.

11.2.2.1 PMA Checker

Most processors require an address decoder to access the appropriate bank of memory or I/O peripheral specified by a physical address. RISC-V uses a PMA checker to perform this address decoding. The PMA specifies and enforces the memory map and its attributes and is often hardwired at design time. However, it may be partially configurable (e.g., to allow for variable amounts of external memory to be connected to a processor).

RISC-V specifies that a core must contain a platform-dependent PMA that generates select signals for each memory bank or I/O peripheral and produces access faults if the memory access is unsuccessful. An unsuccessful access may be caused by a physical address located in a *vacant* portion

of the memory map or an unsupported operation (e.g., a byte access to a peripheral that only supports word accesses).

Main memory is required to support all forms of accesses, but peripheral memory operations that might be constrained by the PMA checker include:

- ▶ Reads, writes, executes
- ▶ Access width, from byte to multiword burst
- ▶ Hardware page table accesses
- ▶ Cacheability (whether the value can be stored in the cache):
 - Normally yes for main memory and no for memory-mapped I/O
- ▶ Idempotency (whether the device produces the same outcome when accessed more than once)
 - Normally yes for main memory: reading or writing the same value twice is harmless
 - Normally no for peripherals that could have side effects
 - Instruction fetches and speculative or redundant accesses must be avoided for non-idempotent devices
- ▶ Atomic memory operations (AMOs); this depends on whether `lr-sc` (load reserved — store conditional) and AMOs are supported in hardware (see [Chapter 17](#))
 - Often yes for all cacheable main memory

[Table 11.4](#) lists the PMA attributes for the various memory regions of the Wally memory map from [Fig. 2.8](#) and [Table A.2](#). The base address is the beginning of the memory region, and the (base address + range) is the top of the region. It is prudent to place no memory at address 0 so that dereferencing a NULL pointer will cause an access fault. Some peripherals have fixed-sized registers and only permit accesses of that size. The size is commonly 32 bits, although it is 8 bits for the UART to match its byte interface.

11.2.2.1.1 Page-Based Memory Types (Svpbmt)

Some peripherals perform *direct memory access* to read or write from main memory, as introduced in [Section 9.1.4](#). If the peripherals aren't specially designed to interact with the cache, they may face a coherence problem where the data cache contains valid data and the main memory contains different data. It is useful to mark certain portions of main memory as uncacheable to avoid this coherence problem. The portion of memory may be application dependent, so competing architectures such as x86 and ARMv8 allow the OS to mark certain PTEs as uncacheable pages. Linux relies on this capability, so RISC-V also offers it.

PMA ranges are non-overlapping because the memory map is non-overlapping.

Table 11.4 PMA for rv64gc Wally memory map.

Region	Base Address	Range	Access Width	ExecuteRead/ Write	Cacheable	Idempotent	Atomic Operations
RAM	80000000	7FFFFFFF	Any	XWR	Yes	Yes	Yes
GPIO	10060000	FF	32-bit	RW	No	No	No
SPI	10040000	FFF	8-bit	RW	No	No	No
UART	10000000	07	8-bit	RW	No	No	No
PLIC	0C000000	3FF0000	32-bit	RW	No	No	No
CLINT	02000000	FFFF	Any	RW	No	No	No
SDC	00013000	FFF	{32/64}-bit	RW	No	No	No
Boot ROM	00001000	FFF	Any	XR	Yes	Yes	No

The optional RV64 RISC-V Svpbmt (*supervisor virtual memory page-based memory types*) extension allows a PTE to override the PMA for certain pages. The `menvcfg.PBMT` bit must be set to enable page-based memory types (see [Section 11.4.1.1](#)). When enabled, bits 62:61 of the PTE are the `PBMT` field. If these bits are 00, the system uses the standard PMA checker. If the bits are 01 or 10, the system overrides the PMA checker to indicate uncacheable main memory and I/O, respectively. Whereas main memory is idempotent, I/O space is non-idempotent. If the `PBMT` field of the PTE is nonzero and Svpbmt is not supported or enabled, a page fault occurs. Linux automatically detects and uses Svpbmt.

11.2.2.2 PMP Checker

For security and to detect pointer errors in programs, it is helpful to limit which physical addresses software may access. An optional RISC-V PMP checker enforces protected address ranges concurrently with the PMA checker. By default, the PMP grants machine-mode access to all parts of physical memory and denies supervisor- and user-mode access to all of physical memory; the PMP is then configured in software to grant access to certain portions in S- or U-mode or to revoke access to certain portions in M-mode. For example, the OpenSBI M-mode firmware configures the PMP so that S- and U-mode software can access all memory except a region used by the firmware. Even if malicious software compromises the OS, the PMP prevents it from corrupting the firmware.

Physical memory protection is specified by 0, 16, or 64 PMP address registers named `pmpaddr0`-`pmpaddr63`. These indicate the addresses associated with a corresponding region. Every PMP address register is also

Ordinarily, M-mode code has access to the entire memory. However, in secure systems, the M-mode boot loader might need to read cryptographic keys only during startup. The boot loader could then revoke access to ROM addresses containing the keys so that they are inaccessible to an attack that compromises other M-mode firmware.

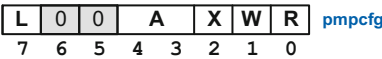


Fig. 11.13 pmpcfg fields.

Because the pmpcfg fields are only 8 bits, multiple sets of fields are packed into a single CSR. In RV32 or RV64, each pmpcfg register contains four (32/8 = 4) or eight (64/8 = 8) sets of 8-bit PMP configuration fields, respectively. Thus, each pmpcfg register supports 4 pmpaddr registers in RV32 and 8 in RV64. For example, in RV32, pmpcfg0 supports pmpaddr0-pmpaddr3, pmpcfg1 supports pmpaddr4-pmpaddr7, and so on. RV64 uses only even-numbered pmpcfg registers; so, pmpcfg0 supports pmpaddr0-pmpaddr7, pmpcfg2 supports pmpaddr8-pmpaddr15, and so on.

The PMP specification allows an implementation to require that PMP region sizes are a multiple of some *grain size* $g = 2^{(G+2)}$. If $G > 0$, then NA4 regions cannot exist.

associated with an 8-bit field in a pmpcfg (PMP configuration) register indicating the type of region and its access privileges (Fig. 11.13).

PMP addresses must be aligned to 4-byte boundaries, so the least significant bits of the addresses are 0 and are not stored in the pmpaddr registers. Thus, in RV32, the pmpaddr registers each specify bits [33:2] of a 34-bit physical address. In RV64, the registers each specify bits [55:2] of a 56-bit physical address.

pmpcfg registers hold multiple sets of the configuration fields shown in Fig. 11.13. The X, W, and R fields determine whether the region is executable, writable, and readable, respectively. The L field indicates if the region is locked. In a locked region ($L = 1$), the corresponding pmpaddr and pmpcfg registers cannot be written except when the system is reset. The XWR permissions always apply to S- and U-mode programs. When L is set, they also apply to M-mode programs. The two bits of A determine the type of PMP region, as described in Table 11.5.

The address of a PMP region is determined by its pmpaddr register and the type (A) field in its pmpcfg register. When the type is top of range (TOR), the region starts at the address in the previous pmpaddr and ends at the address in the current pmpaddr. When the type is NA4, the region is 4 bytes long starting at the address in pmpaddr. When the type is NAPOT, the region is a power of 2 long, whose base address and size are encoded in pmpaddr. Specifically, the string of 1s at the bottom of pmpaddr determines the size. If the least significant k bits of pmpaddr are 1s, the region size is 2^{k+3} . The base address is determined by treating these k bits as 0 and shifting the address left by 2.

The PMP checker examines each of the regions in sequential order to find the first region that matches the physical address. It applies the protection specified by the X, W, R, and L bits for that region.

Table 11.5 PMP region types.

A[1:0]	Name	Region Type
00	OFF	Disabled
01	TOR	Top of range
10	NA4	Naturally aligned 4-byte region
11	NAPOT	Naturally aligned power of two region (≥ 8 bytes)

At reset, all PMP regions are OFF; so all accesses are granted to M-mode and denied to S- and U-mode (if PMP is supported).

Example 11.8 PMP ADDRESS RANGES

Considering Table 11.6, what is the range of addresses in region 6?

Solution: The A field of `pmpcfg` is 3 (11_2), so this is a NAPOT region. The region's `pmpaddr` = 0x2004013F has $k = 6$ trailing 1s, so its size is $2^9 = 512$ bytes. Discarding these ones gives 0x20040100 and shifting left by 2 gives 0x80100400. Hence, this region spans 512 bytes from 0x80100400 to 0x801005FF.

Table 11.6 PMP regions example.

Region	pmpaddr	pmpcfg	L	A	X	W	R	Comments
0	0FFFFFFF	1B	0	NAPOT	0	1	1	I/O 00000000-7FFFFFFF RW
1	20040000	00	0	OFF	0	0	0	NULL
2	2004003F	09	0	TOR	0	0	1	80100000-801000FC R
3	20040080	00	0	OFF	0	0	0	NULL
4	20040084	0C	0	TOR	1	0	0	80100200-80100210 X
5	200400C0	90	1	NA4	0	0	0	80100300-80100303 locked out
6	2004013F	18	0	NAPOT	0	0	0	80100400-801005FF no access
7	00000000	00	0	OFF	0	0	0	NULL
8	00000000	00	0	OFF	0	0	0	NULL
9	00000000	00	0	OFF	0	0	0	NULL
10	00000000	00	0	OFF	0	0	0	NULL
11	00000000	00	0	OFF	0	0	0	NULL
12	00000000	00	0	OFF	0	0	0	NULL
13	00000000	00	0	OFF	0	0	0	NULL
14	00000000	00	0	OFF	0	0	0	NULL
15	2FFFFFFF	1F	0	NAPOT	1	1	1	Main mem 80000000-FFFFFFFF RWX

Example 11.9 PMP MEMORY REGION CHARACTERISTICS

Describe the physical memory regions defined in [Table 11.6](#).

Solution: The comments field describes the regions with their access permissions. For example, according to region 5 registers, the 4 bytes starting at $200400C0 \ll 2 = 80100300$ is completely inaccessible from any privilege mode ($L = 1$). Region 2 registers define a TOR region that is read-only ($R = 1$) for U- and S-mode programs. The range starts at the address in the previous register ($\text{pmpcfg1} \ll 2 = 20040000 \ll 2 = 80010000$) and ends at the address in its own register ($\text{pmpcfg2} \ll 2 = 2004003F \ll 2 = 801000FC$). Register 15 makes all memory from 80000000 - $FFFFFFFF$ completely accessible. However, permission is evaluated in sequential order, so the lower-numbered regions 2, 4, 5, and 6 take precedence and are more restrictive.

If physical memory protection (PMP) is implemented, at least one region must be configured before entering supervisor or user mode; otherwise, all instruction fetches and data accesses will trap. A simple configuration is to make all memory accessible by setting $\text{pmpcfg0} = 0F$ and $\text{pmpaddr0} =$ all 1s.

11.2.3 Memory Exceptions

RISC-V has nine types of memory exceptions: three flavors (access, page, misaligned) for each of three types of accesses (instruction, load, store/AMO).

Access-fault exceptions occur when a process attempts an illegal access, such as to nonexistent memory, writing a read-only memory, or making an access in violation of PMP. The exception handler generally responds by printing an error and killing the process.

Page-fault exceptions occur when the virtual memory system detects an issue that cannot be handled in hardware. These exceptions are usually delegated to the S-mode exception handler because the OS typically deals with virtual memory. The handler must determine the cause. If the PTE is invalid, the OS must allocate a page of physical memory and fetch the missing page from disk. The OS may also set the access or dirty bits in the PTE. If the access permissions are inappropriate, the OS may kill the process.

Misaligned exceptions differ for instruction fetches and for data loads/stores. Instruction-misaligned exceptions only occur when the address is an odd multiple of 2 and compressed instructions are not supported. The handler should kill the process. Load-misaligned and Store/AMO-misaligned exceptions occur when the data address is not a multiple of the access size and the core does not have hardware support for misaligned accesses. [Section 19.2](#) describes a hardware implementation of misaligned support for cacheable memory. When a misaligned access is not handled in hardware, the processor throws an exception, and the handler may issue a sequence of accesses to emulate the misaligned access. If the PMA Aligned attribute is 1, then a misaligned access can throw an

access-fault instead. For example, peripherals may only support aligned accesses even though main memory supports misaligned accesses.

11.2.4 Fences

The `sfence.vma` instruction invalidates all TLB entries. The OS should use `sfence.vma` after modifying the page tables to ensure that programs see the new page tables rather than stale entries in the TLB.

If both PMP and virtual memory are implemented, the TLB may cache PMP permissions, even in M-mode, when virtual memory is not active. In this situation, the firmware must also issue a `sfence.vma` after updating PMP CSRs.

11.2.5 HPTW

An OS expects a hardware page table walker (HPTW) to handle TLB misses. The RISC-V `Sscpr` (*supervisor cacheable coherent page table read*) extension indicates that the HPTW can access any main memory region. Wally's HPTW supports this feature. The HPTW is located in the LSU, but it handles misses from both the data and instruction TLBs (DTLB and ITLB). [Section 12.3.4](#) shows how the HPTW is integrated into the LSU.

11.3 TEST PLAN

At the time of this writing, `riscv-arch-test` does not yet have MMU tests. Handwritten Wally MMU tests that exercise each feature at least once are in `tests/wally-riscv-arch-test/riscv_test_suite/rv{32/64}i_m/privilege`. More comprehensive functional verification tests are under development at the time of this writing.

The `WALLY-pma.S` and `WALLY-pmp.S` test files check basic operation of the PMA and PMP, including each type of normal and exceptional access. The virtual memory tests are in `WALLY-mmu-sv32` for RV32 and `WALLY-mmu-sv{39/48}` for RV64. If Svadu, Svnepot, and Svbpmt are supported, use `WALLY-mmu-sv*-svadu-svnepot-svbpmt` instead. These tests require a page table rich enough to exercise all the virtual memory features and exception conditions. SV48 uses the 4-level table from [Fig. 11.10\(a\)](#), and the other flavors use similar tables. No tests are yet written for `sfence.vma`.

The detailed MMU test plans are in the `cvw` repository under `docs/testplans`.

11.4 WALLY IMPLEMENTATION

[Fig. 11.14](#) shows the Wally top-level block diagram, highlighting in blue the MMU and HPTW. Wally has two MMUs: an instruction MMU (IMMU) in the IFU and a data MMU (DMMU) in the LSU. Both have

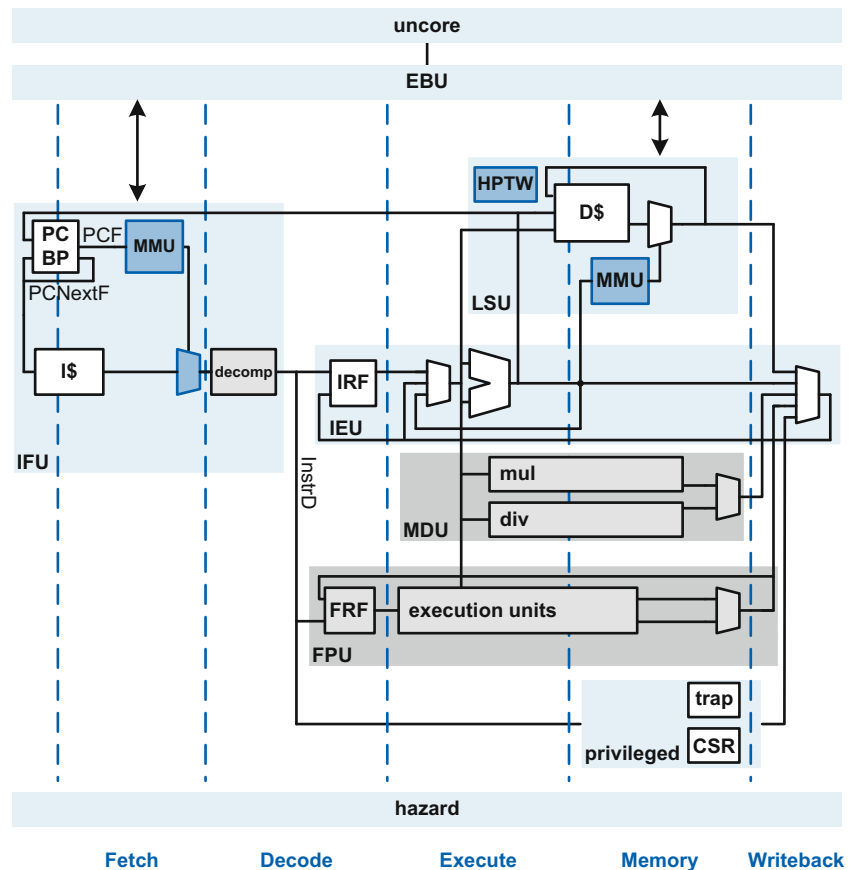


Fig. 11.14 Wally high-level block diagram, highlighting the instruction and data MMUs and HPTW.

identical SystemVerilog code, but the IMMU receives only execute requests (but no read or write requests), and the DMMU receives only read or write requests (but no execute requests), so logic synthesis optimizes out unnecessary access fault circuitry based on the parameters. The MMUs are configurable. At a minimum, the MMU must contain a small PMA checker to detect access and alignment faults for memory reads and writes. At a maximum, the MMU further contains a TLB for virtual memory, a PMP checker with 16 or 64 entries, and a PMA checker that supports many devices. The MMUs are present whenever `ZICSR_SUPPORTED = 1` because they generate the access, misaligned, and page faults.

If virtual memory is supported, Wally adds an HPTW in the LSU to support TLB misses. The HPTW services both instruction and data TLB misses; only one HPTW is needed because only one miss can be serviced at a time through the bus interface anyway. All page table accesses are

routed through the data cache (and on to external memory if the cache misses) and are checked by PMA and PMP checkers in the DMMU.

The MMUs are designed to be agnostic to the choice of bus interface; they send requests to the external bus unit (EBU), which directs the bus transactions. Figs. 12.2 and 13.12 show in more detail how the MMUs and HPTW are integrated into the LSU and IFU.

The configuration file (`configs/*/config.vh`) specifies the memory map for the PMA checker using the `BASE` and `RANGE` (size) parameters from Table A.4 corresponding to the typical memory map from Table A.2. Virtual addresses are always `XLEN` bits. Physical addresses are `PA_BITS = 34` for RV32 and `56` for RV64. For example, the PLIC has `PLIC_BASE = 0x0C000000` and `PLIC_RANGE = 0x03FFFFFF`, so the PLIC spans addresses `0x0C000000-0x0FFFFFFF`. Other key MMU configuration parameters include:

- ▶ `VIRTMEM_SUPPORTED` 1 to enable virtual memory
- ▶ `ITLB_ENTRIES/DTLB_ENTRIES` TLB size, typically 16 each
- ▶ `PMP_ENTRIES` 0, 16, or 64
- ▶ `SVNAPOT_SUPPORTED` 64-KiB huge pages
- ▶ `SVPBMT_SUPPORTED` Page-based memory types
- ▶ `SVADU_SUPPORTED` HPTW updates A and D bits in PTE

11.4.1 MMU

Fig. 11.15 shows a block diagram of the MMU, focused on the major busses. The MMU always contains a PMA checker. If `VIRTMEM_SUPPORTED = 1`, it also contains a TLB, and if `PMP_ENTRIES > 0`, it also contains a PMP checker. The MMU receives the address `VAdr` used by the program. Note that `VAdr` is a virtual address if virtual memory is enabled or a physical address if not. If virtual memory is enabled, the MMU first translates the VPN to a PPN using a TLB. The MMU performs physical address decoding and checking to select the appropriate memory region and generate access faults. If the PTE is not in the TLB, the MMU asserts `TLBMISS`, which causes the HPTW to fetch the missing PTE.

All memory accesses have their addresses routed through the MMU so that it can perform PMA and PMP checking. The accesses may come from ordinary program execution (instruction fetch, or load/store operations) or from the HPTW. Addresses generated by the HPTW are physical, so they do not require translation but are still routed through the LSU MMU for PMA and PMP checking.

The MMU is on the critical path in Wally's 5-stage pipeline. The path occurs in the Memory stage, starting with the TLB translating a virtual

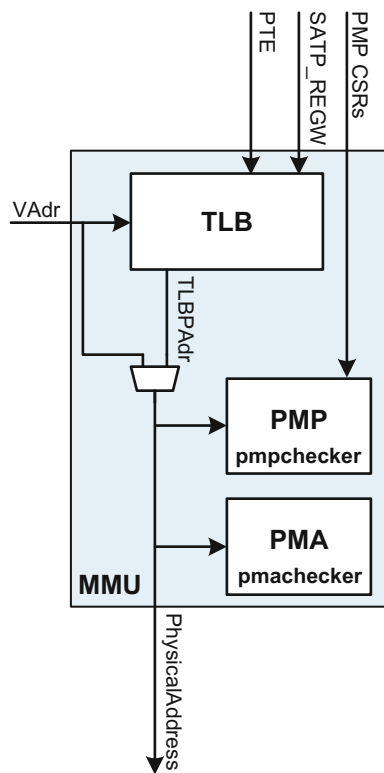


Fig. 11.15 Wally MMU (mmu).

address to a physical address. The PMP checker then compares this address to the PMP CSRs to determine if an access fault occurred. The privileged unit combines this information with other possible faults to cause a trap. The trap changes the program counter, which must set up in time to access the L1 I\$ on the rising edge at the end of the cycle.

11.4.1.1 PMA Checker

The PMA checker uses the physical address to determine which memory region is accessed and the properties of that region. The PMA triggers an access fault if the region is vacant or does not support the type or size of access. Fig. 11.16 shows a block diagram of the RV64 PMA checker (pmachecker). It contains an array of address decoders (adrdecs) to check addresses in each region of the memory map, such as the tightly integrated on-chip memories, external memory, and the I/O peripherals. It checks which region the physical address matches and whether the access size and type are legal for that region, according to Table 11.4. adrdecs

The RV32 PMA checker is nearly identical but uses shorter addresses and omits the leading 1 in the SizeMask to prohibit 64-bit accesses.

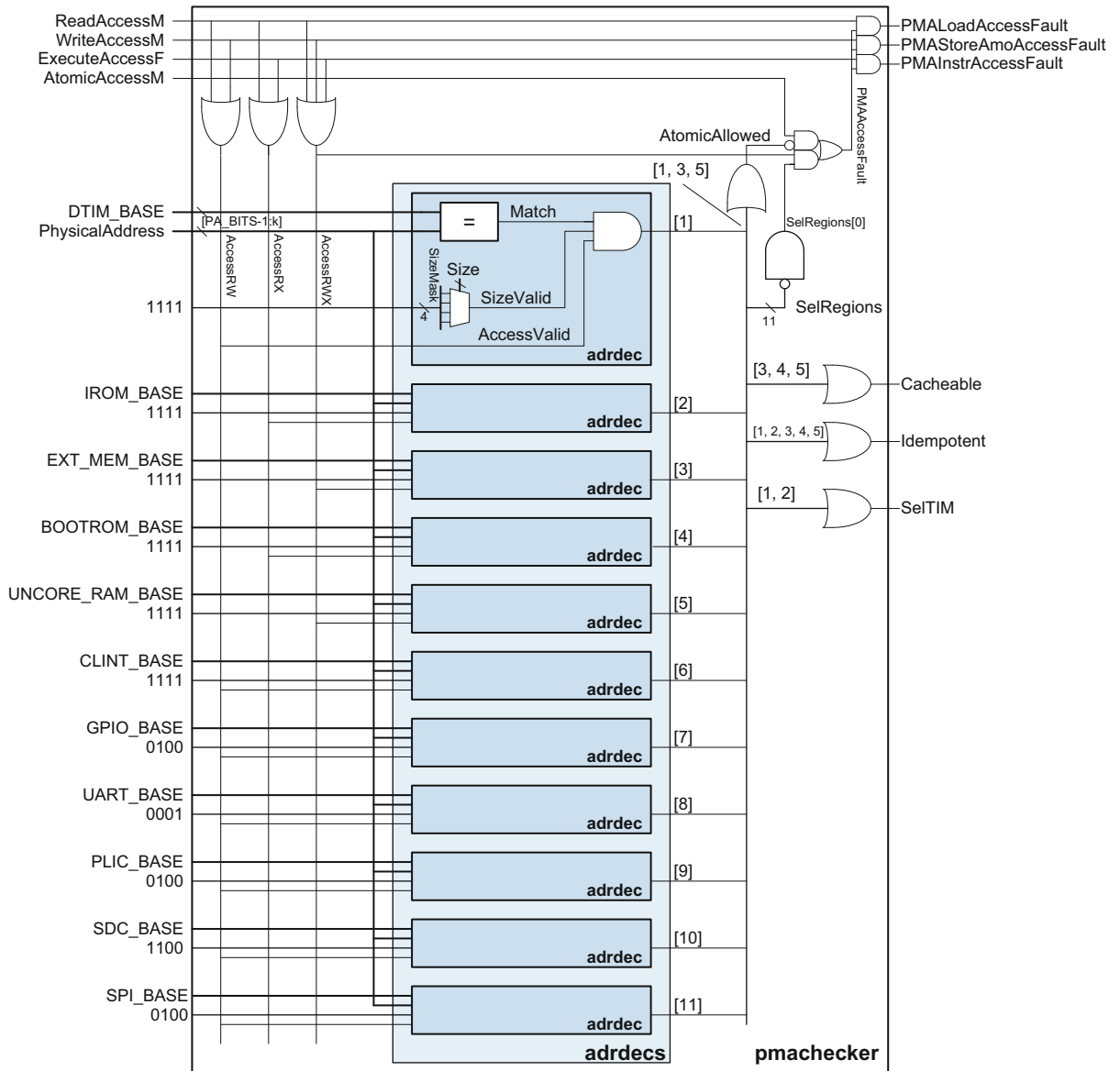


Fig. 11.16 Wally PMA checker (`pmachecker`) for RV64.

produces a one-hot select output (`SelfRegions`) that indicates the accessed region or raises `SelfRegions[0]` if the region is vacant. The PMA checker determines whether the access is to a cacheable, idempotent, and/or atomic region or to a tightly integrated memory in the core. For example, the uncore RAM, Boot ROM, and external memory (regions 3, 4, 5) are all

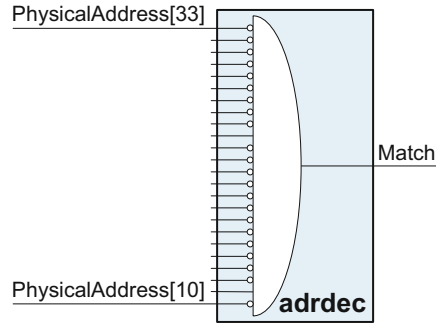


Fig. 11.17 Match logic example.

cacheable. If the access is not legal, the PMA checker produces an instruction-, load-, or store-access fault. For example, if an atomic memory operation (`AtomicAccessM = 1`) attempts to access a nonatomic region (`AtomicAllowed = 0`), the PMA produces a store/AMO-access fault (`PMAStoreAMOAccessFault = 1`).

Wally address decoding is made fast and cheap by imposing a natural alignment requirement on the memory map that each region covers all the possible combinations of the least significant bits while having the upper significant bits be the same for the region. For example, a region of `0x01000800-0x01000BFF` is legal because it spans all values of the bottom 10 address bits while keeping the upper 24 address bits the same. A region of `0x00001000-00003FFF` is not legal because not all combinations of the lower 13 bits are in the region. Because regions satisfy this requirement, address decoding simply involves an equality comparison on the upper k bits of the address, which involves k 2-input XNOR gates and a k -input AND gate but no costly subtraction. The base address is hardwired, so the XNORs further simplify to NOT gates or wires. For example, Fig. 11.17 shows the Match logic for the region spanning `0x001000800` to `0x001000BFF` with 34-bit physical addresses. The upper 24 bits (`0b00 0000 0001 0000 0000 0000 10`) must match for that region. Compared to the PMP checker, the PMA checker does not need a costly priority encoder to choose among multiple matching regions.

Although the PMA checker is written to handle both instruction and data accesses, the `ReadAccess` and `WriteAccess` signals are tied low for the IMMU, and the `ExecuteAccess` signal is tied low for the DMMU, so logic synthesis can optimize away the unneeded access types in each PMA checker. If a given peripheral is not supported, its address decoder (`addrdec`) is also optimized out of the circuit.

When `SVPBMT_SUPPORTED = 1` and the `PBMT` field of a PTE is nonzero, the PMA checker overrides the Cacheable and Idempotent attributes according to the page-based memory type.

The uncore contains the address decoders required to select one of the AHB devices, as discussed in Section 9.5.3.1. The `pmachecker` modules in both the instruction and data MMUs each contain another copy of the address decoders to check access faults. The EBU could choose `HSEL` signals coming from the instruction or data MMU instead of needing a third copy of the decoders. However, Wally supports configurations with a bus but no MMU, so it is simplest to have the extra decoders in the uncore.

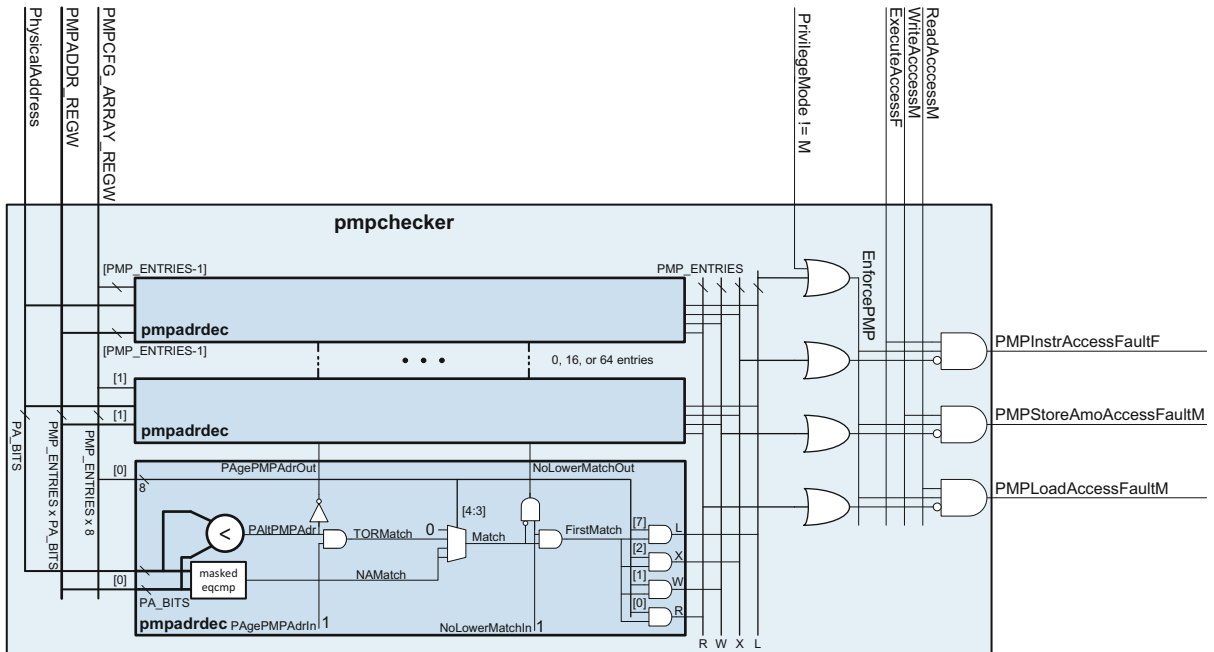


Fig. 11.18 Wally PMP checker (pmpchecker).

11.4.1.2 PMP Checker

The PMP checker causes access faults when a physical address violates PMP permission. Fig. 11.18 shows a block diagram of the PMP checker. It contains 16 or 64 PMP address decoder (**pmpadrdec**) blocks, each of which checks one PMP entry. Each **pmpadrdec** performs a less than comparison and a masked equality comparison to check for TOR and NAPOT/NA4 (naturally aligned) matches. The masked equality comparator tests only the more significant bits, disregarding trailing zeros in the PMP address. Based on the address mode (bits 4:3 of the **pmpcfg** field), the **pmpadrdec** selects the appropriate kind of match. More than one PMP entry could match a given address. A distributed priority encoder determines the first PMP entry that matches, **FirstMatch**. It uses daisy-chained AND gates to create a thermometer-encoded **NoLowerMatch** signal that is 1 for all of the entries up to the first match. The **R**, **W**, **X**, and **L** fields of **pmpcfg** for that entry are extracted using a distributed mux made of AND gates and a wide OR. **pmpchecker** enforces PMP rules when matching a locked region in M-mode, or anytime in S- or U-mode. Finally, the PMP checker generates an access fault based on the type of access occurring, privilege mode, and whether the PMP entry forbids that type of access.

The critical path through the PMP checker includes the magnitude comparator, priority encoder, wide OR, and a handful of other gates.

When Wally supports compressed instructions or misaligned loads and stores, it also requires a grain size (PMP region size) at least as large as a cache line. Misaligned accesses that wrap a grain boundary are decomposed into multiple smaller accesses. PMP checks run independently on each access and trap if any access fails.

The PMP check starts after the TLB delivers a physical address, and the check must complete in time to process a fault. This may limit the cycle time.

11.4.1.3 TLB

Recall that a fully associative cache has a single set and N ways, where N is the number of lines or, in the case of a TLB, entries.

A CAM compares its input with each of its entries. So, an N -entry CAM consists of N comparators. When a CAM's input matches an entry, it asserts a match for that entry.

A PTE is held in at most one TLB entry, so at most one entry matches.

The TLB quickly translates virtual to physical addresses using a cache of recent translations. It behaves as a fully associative cache with $\{I/D\}$ `TLB_ENTRIES`. Each entry consists of a VPN, ASID, PTE (containing the PPN and access bits), and status bits including Page Type and Valid bits. During a memory access, the TLB searches all entries in parallel. If it finds a PTE that matches the ASID in the `satp` register and the VPN in the upper bits of the address, then the TLB checks the access permissions and either outputs the PPN or triggers a page fault. If the TLB does not find a match, it invokes the HPTW to traverse the page tables in main memory and fetch the missing PTE. An entry with the new PTE replaces one of the TLB's least recently used entries.

Fig. 11.19 shows a top-level block diagram of the TLB. The core of the TLB consists of a CAM (content-addressable memory: for tag comparison), RAM (for storing PTEs), and LRU (least-recently used replacement logic). These core blocks contain one line for each TLB entry, with a typical configuration having 32 TLB entries. When the CAM detects that an incoming tag matches the VPN and ASID, it raises the corresponding bit in `Matches` and reads that PTE from the RAM. When a new PTE needs to be written to the TLB, the LRU logic selects the line to be replaced and raises that line's bit in `WriteEnables`. The `tlbmixer` combines the PPN and page offset to produce the physical address. In the case of superpages, it also takes some address bits from the VPN. The controller `tlbcontrol` determines if the TLB hit, missed, or caused a page fault and whether translation should occur. The following subsections describe each block.

Unlike a conventional memory that accepts an address in binary and decodes it, the TLB has 1-hot `Matches` and `WriteEnables` signals to access an entry. The `tlblru` generates one `WriteEnable` for each entry, and the `tlbcam` generates one `Match` for each entry.

11.4.1.3.1 `tlbcam`

Fig. 11.20 shows a schematic of the TLB CAM, which holds the tags. The CAM is shown for systems that support Sv48 with a 36-bit VPN; those that only handle Sv39 or Sv32 use fewer bits and comparators. The CAM uses flops to hold the VPN, ASID, Page Type, and Valid bit. Each entry receives a write enable. The Valid bit is set to 1 on a write and cleared for all lines on a CAM flush. `PageType` is a two-bit code that indicates the type of page, as given in Table 11.7. The CAM matches if the VPN of the current access and the ASID bits field in `satp` match the values stored in a CAM line. Lower-order bits of the VPN are ignored for superpages, and the most

Table 11.7 Page types.

PageType	Type
00	kilopage
01	megapage
10	gigapage
11	terapage

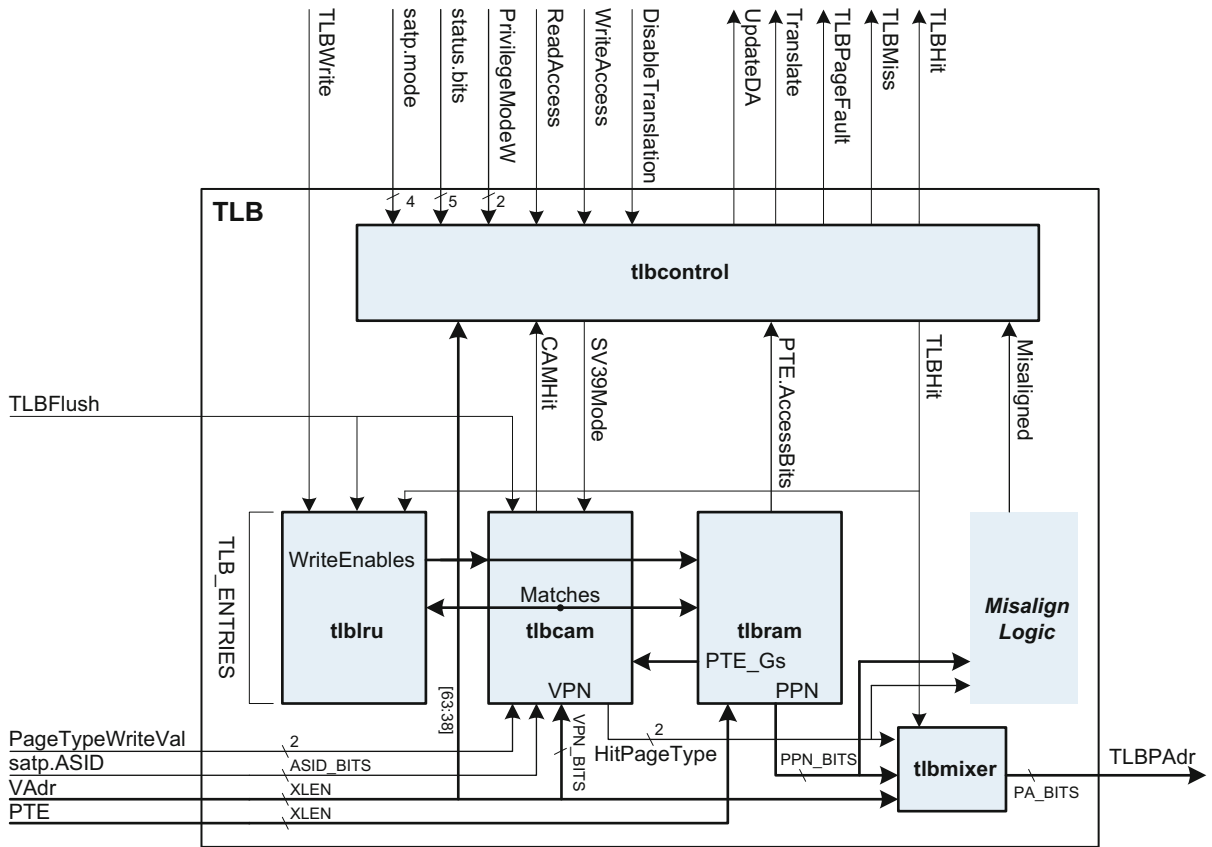


Fig. 11.19 Wally TLB (t1b).

significant bits are ignored when operating in Sv39 mode. The ASID is ignored if the PTE entry is marked global in the TLB RAM.

CAMHit asserts when any of the lines Match and TLBFlush is low. The CAM also outputs HitPageType, which indicates the type of the matched line. CAMHit and HitPageType are output using a distributed AND-OR multiplexer similar to Fig. 10.25 to combine the outputs of all the TLB lines (tlbcamlines) to determine if any of them hit and, if so, the page type. For example, each tlbcamline ANDs all the comparisons to create Match and then tlbcam ORs all the Match signals (and ANDs it with TLBFlush) to produce CAMHit. Similarly, each tlbcamline's PageType is ANDed with Match to produce PageTypeRead (so at most one PageTypeRead will be 1). Then tlbcam ORs PageTypeRead from all lines to produce HitPageType.

The Misalign Logic in Fig. 11.19 detects misalignment for superpages (mega-, giga-, or terapages) by detecting whether the correct PPN bits are 0s as determined by PageType.

A traditional custom CAM is built using a specialized cell that both holds a bit and also detects matches [Weste12]. Typical synthesized ASICs and FPGAs lack such CAM cells, so the TLB is implemented with flip-flops and gates to hold the tag and check the match.

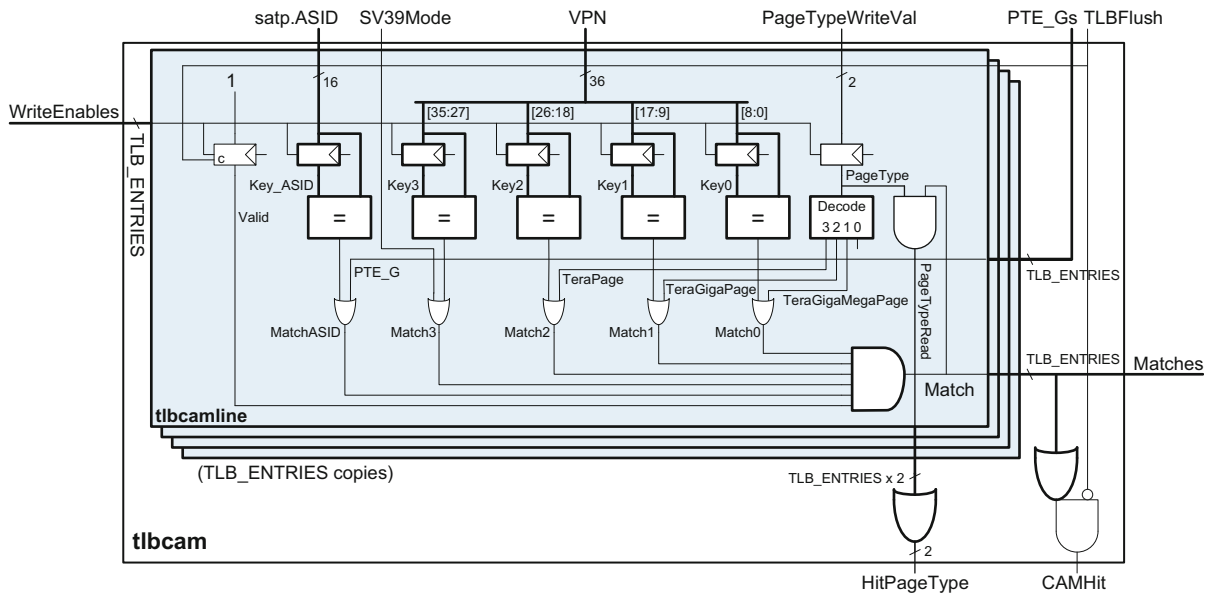


Fig. 11.20 Wally TLB CAM (tlbcam).

A PTE's G flag (PTE_G) is 1 when an entry is global (exists for all address spaces, see Table 11.1). The TLB RAM outputs PTE_G for all TLB entries (Fig. 11.21), and the TLB CAM then uses that signal to determine a match (Fig. 11.20).

11.4.1.3.2 tlbam

Fig. 11.21 shows a schematic of the TLB RAM (tlbam) that holds the PTEs. tlbam, like tlbcam is also built using flip-flops on an FPGA or ASIC. tlbam holds TLB_ENTRIES lines, each containing a PTE (PageTableEntry) which holds the PPN and 8 access bits (see Tables 11.1 and I.18). Like tlbcam, the WriteEnables bus determines which line to write, and a distributed AND-OR multiplexer reads out the line matched by the CAM. Essentially, the Matches signals act as read enables to select which entry, if any, to read out.

11.4.1.3.3 tlbmixer

Fig. 11.22 shows the RV64 tlbmixer. It forms the PPN from a combination of the VPN and PPN bits and the page offset. For kilopages, it concatenates the PPN with the page offset to form the physical address. For superpages, it selects some of the lower-order page number bits from the VPN rather than the PPN, as shown in Example 11.5. Because these low-order bits are always 0 in the PPN, the selection logic reduces from a multiplexer to an AND-OR gate. It then combines the PPNMixed signal with the page offset from the bottom 12 bits of the address. If the TLB misses, the physical address is forced to 0 (NULL); this isn't strictly necessary but will cause an access fault in the event that the TLB output is incorrectly used later. TLBHit is the CAMHit signal from tlbcam qualified by the access bits.

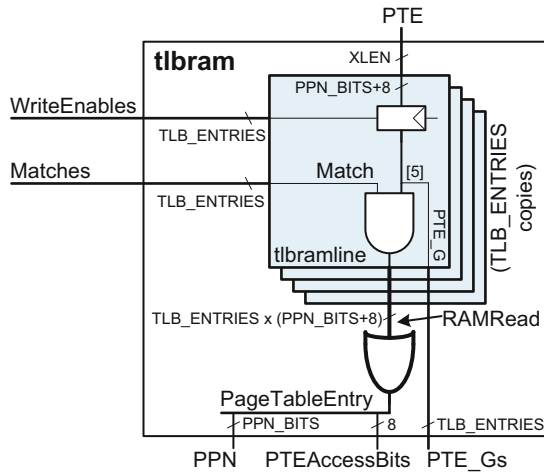


Fig. 11.21 Wally TLB RAM (**tlbram**).

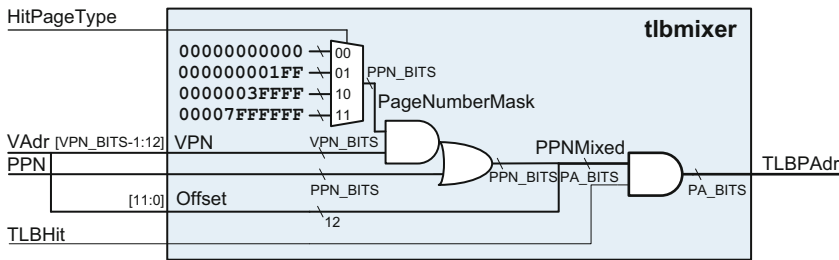


Fig. 11.22 Wally TLB mixer (**tlbmixer**).

11.4.1.3.4 **tlblru**

Fig. 11.23 shows a schematic of the TLB LRU unit that determines a least-recently used entry to replace. True LRU logic is too costly, so the block approximates it by keeping track of each entry that has been accessed and picking the first entry that hasn't been accessed. The recently used bits (**RUBits**) register keeps track of recently used entries. If all entries have been used recently, then all entries' **RUBits** are cleared. The priority unit **pri** picks the first entry that has not been recently used. On a write, the corresponding bit in the **WriteEnables** bus is raised to write the selected line of the CAM (tag) and RAM (PTE).

11.4.1.3.5 **tlbcontrol**

The **tlbcontrol** module determines the effective privilege mode; whether translation should occur; and whether the TLB hit, missed, or caused a page fault.

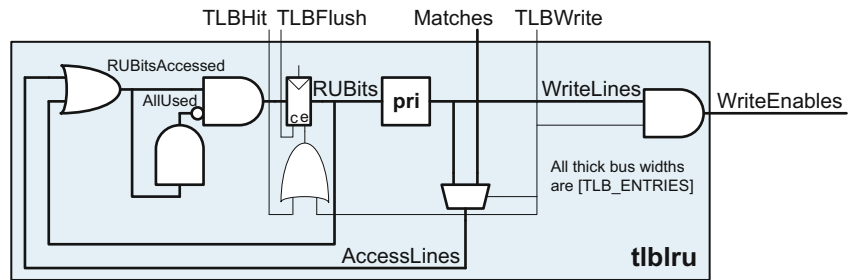


Fig. 11.23 Wally TLB LRU (`tlblru`).

Fig. 11.24 shows a schematic of `tlbcontrol` in the DTLB configuration. The TLB's `EffectivePrivilegeMode` is normally the core's `PrivilegeModeW`, but is `status.MPRV` for the DTLB when the `status.MPRV` bit is set. Virtual address translation is enabled for S- and U-mode accesses when `satp.MODE` $\neq$ BARE but disabled during page table walks and cache flushes.

The TLB causes the appropriate kind of instruction/load/store page fault (`TLBPageFault`) when any of the following conditions occur during an access that hits the TLB.

- ▶ **UpperBitsUnequalPageFault:** The upper bits of the VA do not match the most significant bit of the VPN.
- ▶ **ImproperPrivilege:** The effective privilege mode doesn't match the `PTE.U` bit (also considering `status.SUM`).
- ▶ **InvalidRead/InvalidWrite:** The operation (execute/read/write) is prohibited by the `PTE.X/R/W` bits (also considering `status.MXR`).
- ▶ **UpdatedA:** The `PTE.A` bit is clear or the `PTE.D` bit is clear on a write. If `SVADU_SUPPORTED = 0` or `menvcfg.ADUE = 0`, `TLBPageFault` is asserted to manage the A and D bits in software. Otherwise, `TLBPageFault` is not asserted, and the HPTW manages the bits.
- ▶ **PTE.V = 0:** The PTE is invalid.
- ▶ **Misaligned:** The PPN of a superpage is misaligned.

The `TLBHit` output is asserted when the CAM hits (`CAMhit`) during a read or write. The TLB misses when an access takes place, translation is enabled, no flush is occurring, and the CAM misses.

The ITLB is similar to the DTLB but is only used for instruction fetches. Hence, it checks `PTE.X` rather than `InvalidRead/InvalidWrite`.

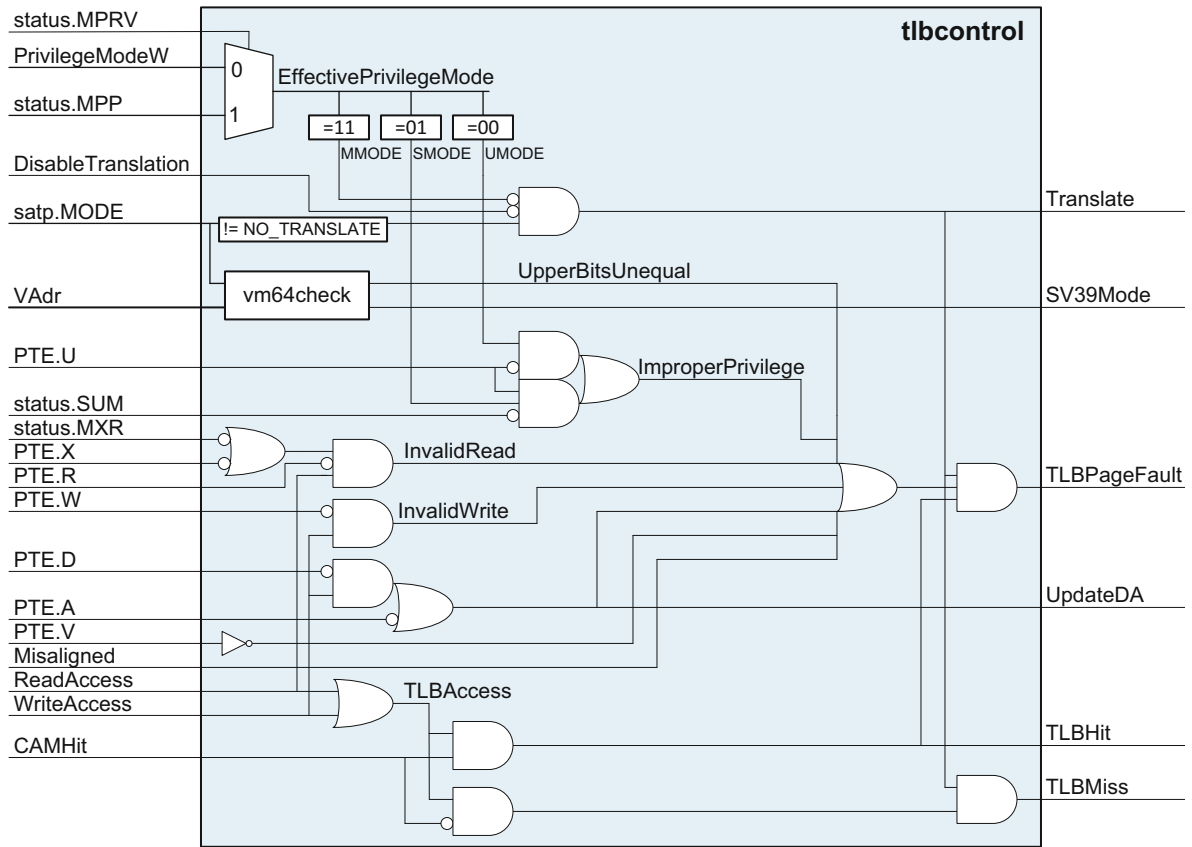


Fig. 11.24 Wally TLB control (tlbcontrol).

11.4.1.3.6 Svnapot Extension

When `SVNAPOT_SUPPORTED = 1`, the `tlbram` also asserts a `PTE_NAPOT` signal for each entry that is marked as a 64-KiB huge page by having the `PTE.N` bit set and `PPN[3:0] = 1000`. For huge pages, the `tlbcam` ignores the bottom four bits of the VPN, and the `tlbmixer` replaces the bottom four bits of the page number with the corresponding VPN bits. Hence, the cost of supporting huge pages is small. Only one TLB entry is used for the 64-KiB huge page compared to 16 entries to hold consecutive 4-KiB regular pages.

11.4.2 HPTW

The HPTW services a TLB miss by issuing a sequence of physical memory accesses to traverse the page table and fetch the missing PTE into the TLB. The HPTW consists of a datapath to insert page table accesses into the LSU, an FSM to sequence the walk through the multilevel page table, and control logic to detect page faults.

The HPTW is also simply called the *walker*.

The HPTW uninstalls the LSU (and thus also its D\$) while the pipeline is stalled so that the HPTW can make data memory requests to access PTEs.

In accordance with the RISC-V specification, if the HPTW triggers an access fault, it is reported back as the type of access fault that originally caused the page table walk. For example, if an instruction fetch triggers a page table walk and a bad PTE causes the HPTW to try reading from a nonexistent memory location, the MMU causes an instruction access fault rather than a load access fault. Thus, the exception handler will know that the root cause was related to an instruction fetch rather than a load instruction.

Fig. 11.25 shows a block diagram of the HPTW datapath. During a page table walk, the $IHAdrM$ mux overrides the normal $IEUAdrM$ to send a physical PTE address $HPTWAdr$ to the LSU. The walker receives the PTE on $ReadDataM$ and captures it in the PTE register. Whereas the most significant bits of the HPTW address ($HPTWAdr$) come from the PPN, the least significant bits are the page offset, which is selected from a field in the VPN based on the current level of the page table walk. The virtual address in turn comes from either $IEUAdrM$ or PCF for data or instruction TLB misses, respectively. On the initial step, the $satp$ register provides the PPN of the top-level page table. On subsequent steps, the PPN comes from the PTE of the previous step.

Section 12.3.4 describes the integration of the HPTW with the LSU. In brief, all HPTW physical memory accesses are issued through the LSU, which uses the DMMU PMA and PMP checkers to detect access faults, and the data cache to return the PTE or send the request up the memory hierarchy. Because the page table addresses are already physical, HPTW accesses don't require translation.

Fig. 11.26 shows the HPTW FSM state transition diagram. The walker sits in an idle state until a TLB miss occurs. Based on the virtual memory mode ($Sv\{32/39/48\}$), it then begins walking from the appropriate page table level. For example, $Sv32$ always begins walking at the level 1 page table. Each level of the page table walk involves at least two cycles: driving the physical address to the LSU and reading the PTE from the LSU. These steps cannot be pipelined because the next physical address depends on the previous level's PTE. Section 12.5.5 shows a timing diagram of the HPTW fetching a PTE through the LSU.

The optional gray portions to update the PTE's D (dirty) and A (accessed) bits apply when configured with $SVADU_SUPPORTED = 1$, as discussed in Section 11.4.2.2.

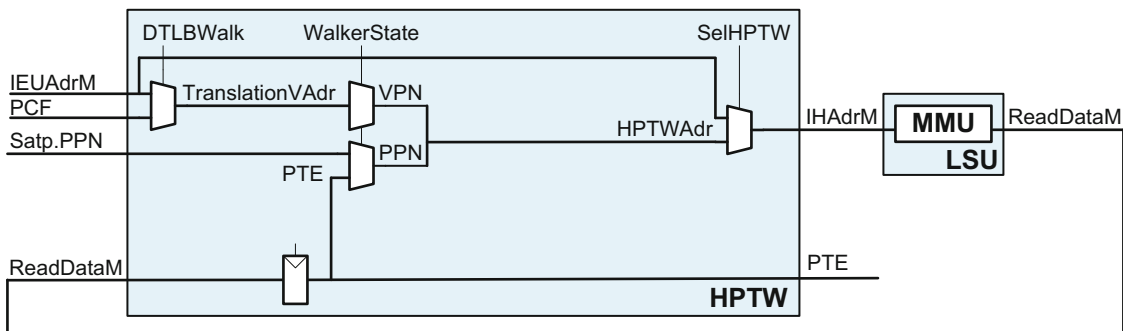
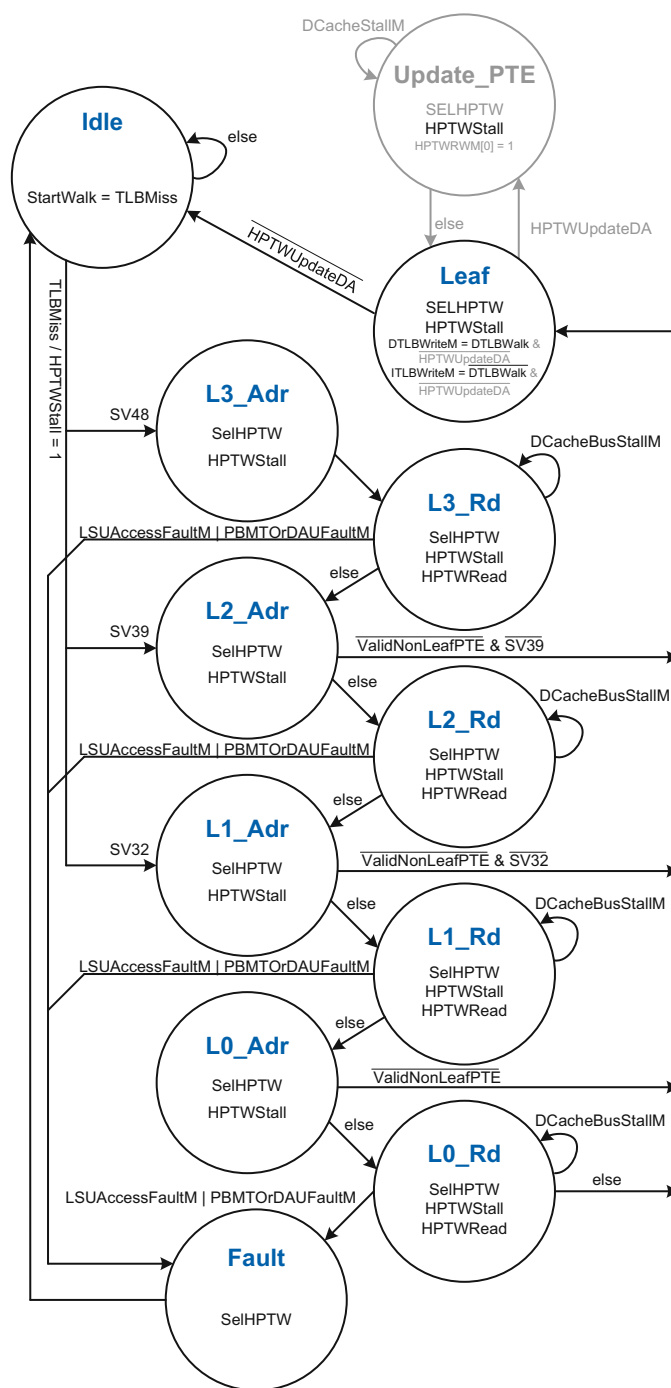


Fig. 11.25 HPTW datapath.



11.4.2.1 HPTW Stalls

When the HPTW is walking the page table, it asserts `SeIHPTW`. This drives the `IHAdrM` multiplexer from Fig. 11.25 to choose the address from the walker. Not shown in the figure, muxes also choose control signals (e.g., the read enable and access size) from the walker.

During a walk, `HPTWStall` asserts `LSUStallM`, which in turn asserts `StallW` to stall the pipeline. However, the LSU doesn't stall. The `GatedStallW = StallW & SeIHPTW` signal stalls the LSU when high, but it remains low during a walk so that the LSU can continue to serve the walker requests while the rest of the pipeline stalls.

11.4.2.2 PTE Status Bit Updates

The accessed (A) status bit of a PTE is set on the first access (read, write, or fetch) to that page, and the dirty (D) status bit is set on the first write. The RISC-V specification allows the HPTW to set these bits in hardware or to cause a page fault so that the OS can set them. When the system is out of memory and must replace a physical page, the OS uses the A bits to choose a page to evict that has not been accessed recently. The OS examines the D bit of the victim page to see if the page is dirty and must be written back to disk.

Because not all hardware implementations support writing the PTE status bits, the OS must have code to support this. However, the page fault handler takes a significant amount of time, so a hardware implementation is attractive.

Wally optionally supports hardware status bit updates when `SVADU_SUPPORTED = 1`. In this case, the HPTW state machine adds the `Update_PTE` state that was shown in gray in Fig. 11.26. Fig. 11.27 extends the HPTW datapath to support such updates, with the new hardware emphasized in blue. It adds an `IHWriteDataM` signal to write the page table with the updated PTE rather than the usual `WriteDataM`. It adds an `UpdatePTE` mux to modify the A and D bits when a page fault would

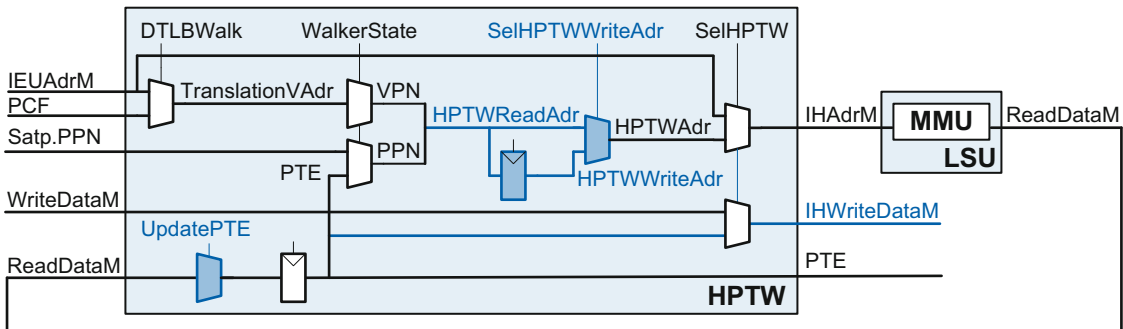


Fig. 11.27 HPTW datapath with PTE status bits update (Svadu) support.

otherwise be needed. Finally, it adds a register and mux to save the leaf PTE address for page table updates.

The HPTW asserts `HPTWUpdatedA` when the A or D bits need to be updated. When hardware status bit updates are disabled, this triggers a page fault. When hardware status bit updates are enabled and the FSM enters the Leaf state, `HPTWUpdatedA` causes the FSM to go to the `Update_PTE` state rather than writing the TLB. In `Update_PTE`, the FSM asserts `HPTWRWM[0]` to write the PTE back to memory, then returns to the Leaf state to write the updated PTE to the TLB. This behavior is shown in gray in Fig. 11.26. These two writes must be atomic so that the RAM and TLB remain coherent. This is guaranteed because interrupts are disabled during page table walking because `Se1HPTW` asserts `CommittedM`.

The memory write may cause a PMA or PMP access fault, but no such check occurs with the TLB write. If the PMA or PMP checkers fault, `LSUAccessFaultM = 1` and the HPTW aborts immediately by transitioning to the Fault state and then returning to Idle, thus not writing the TLB. The HPTW also exits to the Fault state if a nonleaf PTE contains nonzero dirty, access, user, or `PBMT` bits (if `Svbpmt` is supported and enabled) when `PBMTOrADUFaultM = 1`.

11.4.3 MMU Faults

The MMU is responsible for access, misaligned, and page faults for instructions, loads, and store/AMO operations, except that instruction-misaligned faults are created by the IFU based on the branch target address.

Fig. 11.28 shows a simplified view of the MMU's fault logic that combines PMA, PMP, TLB, and misalignment fault signals. The `Align Check` block verifies that `VAdr` is a multiple of the access size. The `privilegepiperegs` block in the privileged unit pipelines the Fetch stage instruction faults into the Memory stage where traps are taken. Atomic operations that read and write memory assert a store/AMO fault rather than a load fault.

11.4.4 Fence Instruction

The `sfence.vma` instruction flushes the TLB. Although the instruction supports flushing only particular address space IDs or virtual addresses, Wally's simple implementation *overfences* by flushing all entries.

If executed in supervisor mode when the `status.TVM` bit is set, `sfence.vma` raises an illegal-instruction exception instead, to support virtualization.

The `Svinval` extension (`SVINVAL_SUPPORTED = 1`) adds the `sinval.vma`, `sfence.w.inval`, and `sfence.inval.ir` instructions. These fine-grained TLB invalidation instructions provide further control of ordering for

The IMMU generates instruction faults, and the DMMU generates load and store faults. The logic for reporting the original cause of the fault (e.g., a load or store fault) when an HPTW access fault occurs is not shown in Fig. 11.28.

If the HPTW encounters an illegal PTE, such as one with `V = 0`, it writes the entry to the TLB but does not immediately generate a fault. When the instruction is replayed, the bad entry will be fetched and trigger a page fault at that time. This simplifies Wally's HPTW implementation by reusing the existing page fault mechanism.

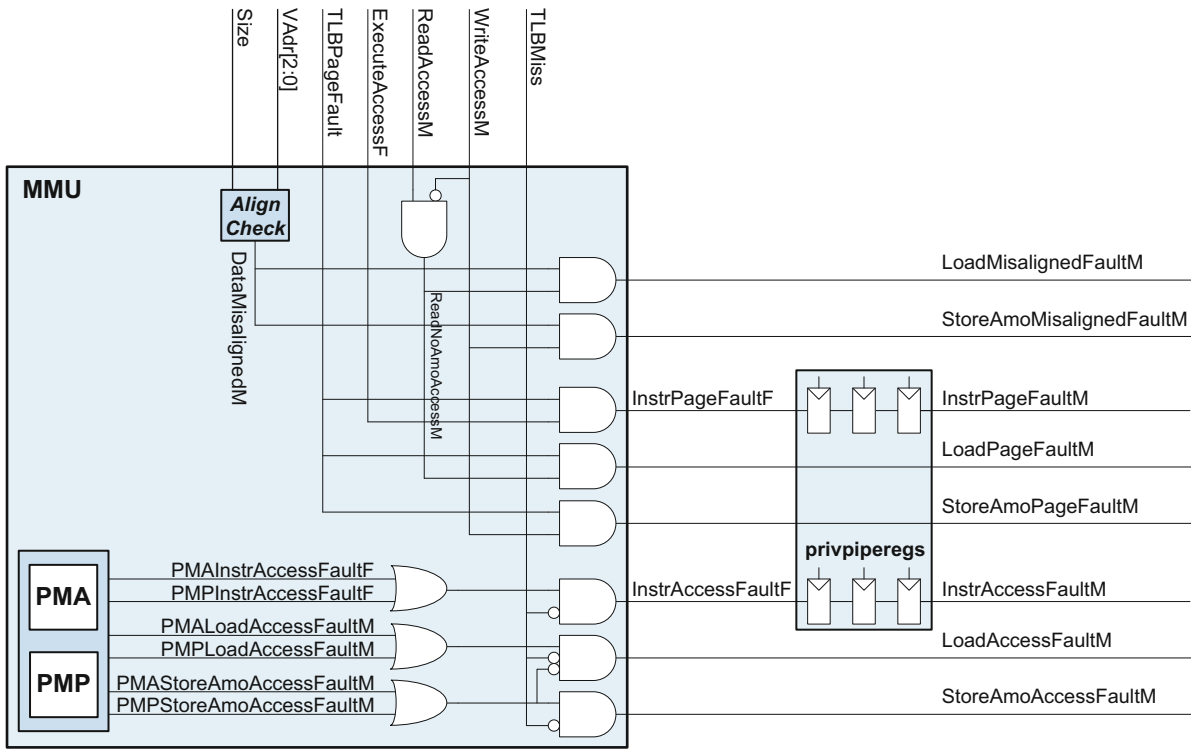


Fig. 11.28 MMU fault logic.

out-of-order and multicore machines. Wally simply treats them the same as `s fence.vma`.

11.5 SUMMARY

The MMU is responsible for virtual memory, PMA checking, and PMP checking.

Virtual memory elegantly addresses the problems of relocation, fragmentation, and protection for cores running multiple concurrent processes. It gives each process an independent virtual address space, so the process has the illusion of its own large and continuous memory. The memory is divided into pages, commonly 4 KiB, which can be placed anywhere in physical memory, or swapped out to disk when there is insufficient physical memory. Each process is protected from other processes because they can only access their own virtual addresses.

The MMU is responsible for translating the process’s virtual address into a physical address to access main memory. Each process has a page table that identifies the (PPN) for each virtual page. In a practical system,

the page table is too big, so it is organized hierarchically into 2-5 levels. The top-level page table is indexed with the most significant bits of the virtual address. It points to the next-level page table, which is indexed with the next most significant bits of the virtual address. This level in turn may point to another page table until the leaf page table entry (PTE) is reached. The leaf PTE contains the PPN and a set of flags such as valid, accessed, dirty, and access permission bits. If the PTE is invalid, has incorrect access permissions, or the page table is otherwise malformed, the MMU causes a page-fault exception, which fetches the missing page from disk into main memory, or kills a misbehaving process.

Traversing the page table on each memory access would greatly slow loads and stores, so the MMU contains a TLB, which is a cache of frequently used PTEs. Pages are large, so even a small TLB generally has a high hit rate because of spatial locality. If the TLB misses, the HPTW injects accesses into the LSU to walk the page table and find the missing PTE.

In RISC-V, machine-mode firmware uses physical addressing (except when `mstatus.MPRV` is enabled to facilitate address translation in an exception handler). The supervisor-mode OS enables virtual addressing with the `satp` CSR.

RISC-V also defines PMA and PMP checking. The PMA checker uses the physical address to determine which memory or peripheral is being accessed and the attributes of that memory. If the access goes to an unimplemented physical address or is of a type the peripheral cannot handle, the PMA checker causes an access-fault exception. The PMA is typically hardwired to reflect the system's memory map.

The PMP checker enhances security by limiting access to certain physical memory regions to machine or supervisor mode, triggering an access fault exception for disallowed accesses. For example, PMP is often used to restrict access to the firmware code to machine mode only, thus preventing a malicious user program from modifying the firmware. RISC-V has 0, 16, or 64 PMP CSRs to define PMP regions; 16 is typical for an application processor.

Wally's IFU and LSU each have their own MMU but share an HPTW in the LSU to handle TLB misses. Each MMU contains a TLB to translate virtual to physical addresses when virtual memory is enabled. The TLB is fully associative, using a CAM to identify which entry matches the VPN and then a RAM to read out the PPN and flags. The TLB operates concurrently with the cache access in the Fetch or Memory stage. Thus, the cache only knows the bottom 12 bits of the physical address when the access begins. By limiting cache ways to 2^{12} bytes = 4 KiB, the way is always indexed by a physical address. The rest of the physical address is available from the TLB by the time it is needed to compare against tags. Thus, the cache is physically addressed, avoiding tricky synonym problems. The `sfence.vma` instruction flushes the TLB by clearing its valid bits. The DMMU is on the critical path in Wally's 5-stage pipeline.

The `wally-riscv-arch-test` programs exercise the virtual memory system, PMA checkers, and PMP checkers.

The load/store unit (LSU) handles data memory access for load and store instructions. It also fetches PTEs (page table entries) on behalf of the hardware page table walker (HPTW). Chapter 17 adds LSU support for AMOs (atomic memory operations).

Past chapters described various LSU elements. Chapter 7 introduced a simple LSU with a DTIM (data tightly integrated memory). Chapter 9 added support for off-chip memory and peripherals via the external bus unit (EBU). Chapter 10 described cache design, and Chapter 11 described the MMU (memory management unit), including address translation, the TLB (translation lookaside buffer), the HPTW (hardware page table walker), and PMP (physical memory protection) checker. This chapter combines all these capabilities into a fully-featured LSU.

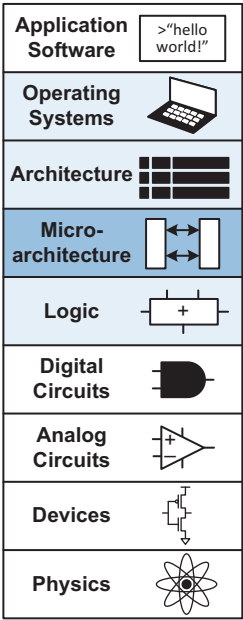
Fig. 12.1 shows the Wally block diagram that highlights in blue the LSU features integrated in this chapter. The LSU receives memory addresses from the IEU (integer execution unit). When virtual memory is enabled, the TLB translates the addresses from virtual to physical. The PMP checker within the LSU faults on illegal accesses to protected memory. The LSU accesses the core memories: either the DTIM (not shown in Fig. 12.1) or the data cache (D\$). When the LSU requests an address that is not in the core memories, it uses the external bus to access peripherals or memory in the uncore or off-chip. The HPTW is part of the LSU because it fetches PTEs through the D\$ for TLB misses. The HPTW services both data and instruction TLBs and writes status bits back to the page tables in memory during page faults.

This chapter presents no new principles but focuses on the implementation of the LSU, especially the integration of the D\$, MMU, and bus components previously described.

12.1 WALLY LOAD/STORE UNIT OVERVIEW

The Wally LSU can be configured to have a DTIM, a D\$, or no core memory. Table A.1 in Appendix A lists various standard configurations and

- 12.1 Wally Load/Store Unit Overview
- 12.2 Data Cache Integration
- 12.3 Memory Management and HPTW Integration
- 12.4 Stalls, Flushes, and Traps
- 12.5 Examples
- 12.6 Test Plan
- 12.7 Summary



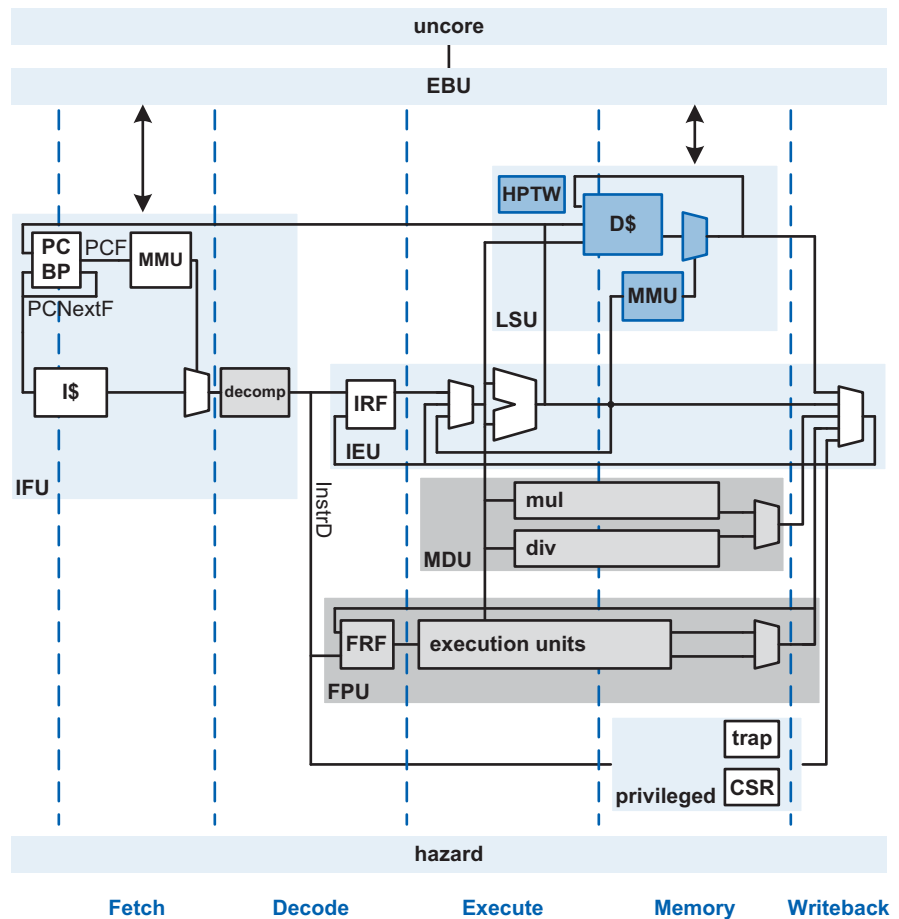


Fig. 12.1 Wally high-level block diagram, highlighting the LSU.

their LSU parameters. The embedded rv32e configuration has no core memory. The simple rv{32/64}i and microcontroller rv32imc configurations have tightly integrated memories. The apps processors rv{32/64}gc have caches. The microcontroller and apps processors have a privileged unit and MMU with a PMA (physical memory attributes) checker to catch illegal access and address misalignment faults; they also have a bus interface to uncore peripherals, external memory, or both. The apps processors also add virtual memory and a PMP checker to the MMU along with associated page faults.

Fig. 7.9 shows a basic LSU for the simple pipelined processor (rv{32/64}i). Fig. 8.19 adds optional endian swapping. The DTIM and IROM take up most of the area in the simple rv32i CPU, so for a small embedded core (rv32e), the DTIM is removed. Fig. 9.21 substitutes a bus interface to

external memory and peripherals, which results in a tiny rv32e core area with no RAM. Figs. 9.30 and 9.31 combine the DTIM with the bus interface for an embedded microcontroller (rv32imc). This chapter adds the remaining LSU features, including integrating the D\$, MMU, and HPTW for rv{32/64}gc.

Fig. 12.2 shows the fully featured (rv64gc) LSU for an application processor configuration capable of booting Linux. Control and byte write enables are omitted from top-level LSU diagrams for clarity. This configuration replaces the DTIM of Fig. 9.31 with a cache and AHB cache interface and adds an HPTW and a fully featured MMU. Details of the endianswap (ES), subwordread (SWR), and subwordwrite (SWW) modules are shown in Fig. 9.30 and are abstracted away here. Like the DTIM discussed in Section 7.1.6, the D\$ supports XLEN-sized *native* words. The floating-point unit increases native word size to LLEN = max(XLEN, FLEN) and adds write multiplexing as described in Section 16.6.1. The atomic block is discussed in Section 17.4.1. These blocks are shown in gray because they are only needed when floating-point (F/D) or atomic (A) extension are enabled. The other components are explored in this chapter.

Fig. 12.2 is too complex to fully explain at once. Instead, this chapter builds up to the full LSU step by step. Section 12.2 integrates the D\$ into

Recall from Section 10.4.5.3 that the cache's sub-cache line read (SCR) module selects the word within the cache line (ReadDataLine) to produce ReadDataWord.

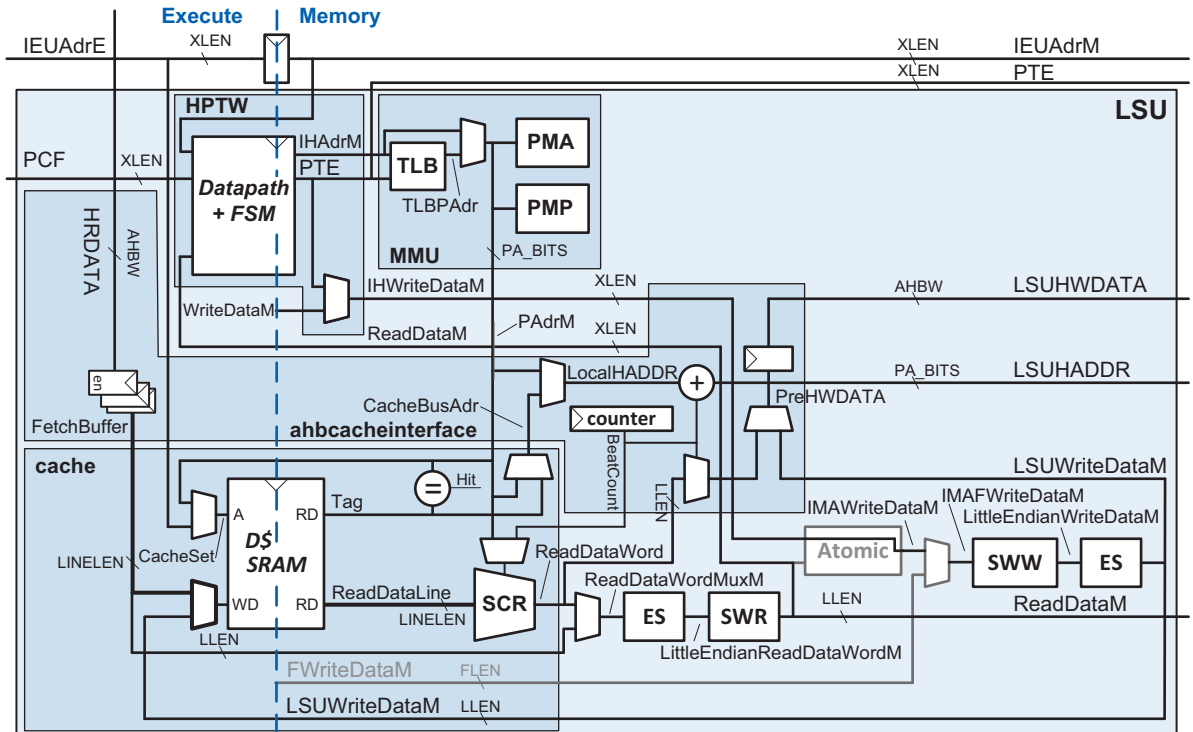


Fig. 12.2 LSU block diagram.

the LSU. Section 12.3 integrates the MMU. Section 12.4 describes LSU stalls, flushes, and traps. Section 12.5 presents timing diagrams explaining each of the use cases.

12.2 DATA CACHE INTEGRATION

This section integrates a D\$ into the LSU, as shown in Fig. 12.3, if `DCACHE_SUPPORTED = 1` in `config.vh`. Key features include servicing load and store hits, fetching lines from the external bus for cache misses, and writing back dirty evicted lines. Compared to Fig. 9.31, this LSU replaces the DTIM and `ahbinterface` modules with cache and `ahbcacheinterface` modules. It temporarily drops the PMA logic because this version of the LSU does not yet support uncached accesses; this capability will be restored in Section 12.3.1. The LSU retains the subword read/write (SWR/SWW) and endian swap (ES) modules.

The cache module was introduced in Fig. 10.24 in Section 10.4.5. All memory operations begin with the IEU calculating the address `IEUAdrE` in the Execute stage. On the rising edge starting the Memory stage, loads and stores both read the cache's ways using the lower (set) bits of `IEUAdrE`. Loads read both the tag and data using `CacheSetTag` and `CacheSetData`,

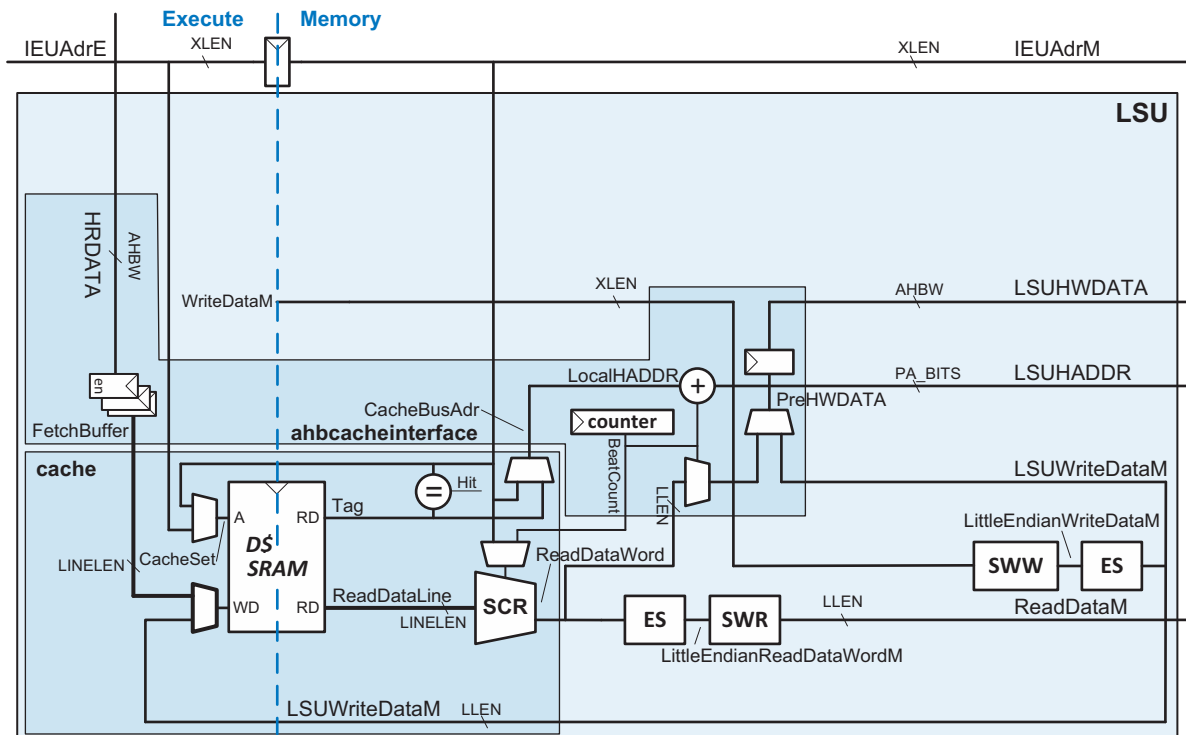


Fig. 12.3 Simplified LSU with data cache only.

but stores read only the tags, using `CacheSetTag`. If one of the tags is valid and matches the upper bits of `IEUAdrM`, the cache hits. Load hits output `ReadDataM` in the Memory stage, and store hits set up to write into the matching way (using `LSUWriteDataW` and `CacheSetData` = the lower (set) bits of `IEUAdrM`) on the next rising clock edge, at the start of the Writeback stage. Both loads and stores perform endian and subword muxing using the `endianswap` (ES) and `subwordread/write` (SWR/SWW) modules.

Cache misses stall the pipeline while fetching the missing line over the bus. Fig. 12.4 shows how the `ahbcacheinterface` manages communication between a cache and the EBU. At a high level, the `cachefsm` makes a bus request to writeback or fetch an entire cache line by placing the line address on `CacheBusAdr` and raising the write or read bit of `CacheBusRW`. For a writeback, the cache sends the evicted line `ReadDataLine` one word at a time over `ReadDataWord`, and the `ahbcacheinterface` generates an AHB write burst. For a line fetch, the `ahbcacheinterface` generates an AHB read burst and accumulates the words into `FetchBuffer`. `ahbcacheinterface` raises `CacheBusAck` when the burst completes.

The `CacheSet{Tag/Data}` inputs to the cache, which access the cache's tag and data arrays separately, are simplified to `CacheSet` in LSU figures and timing diagrams, such as Figs. 12.2, 12.3, etc.

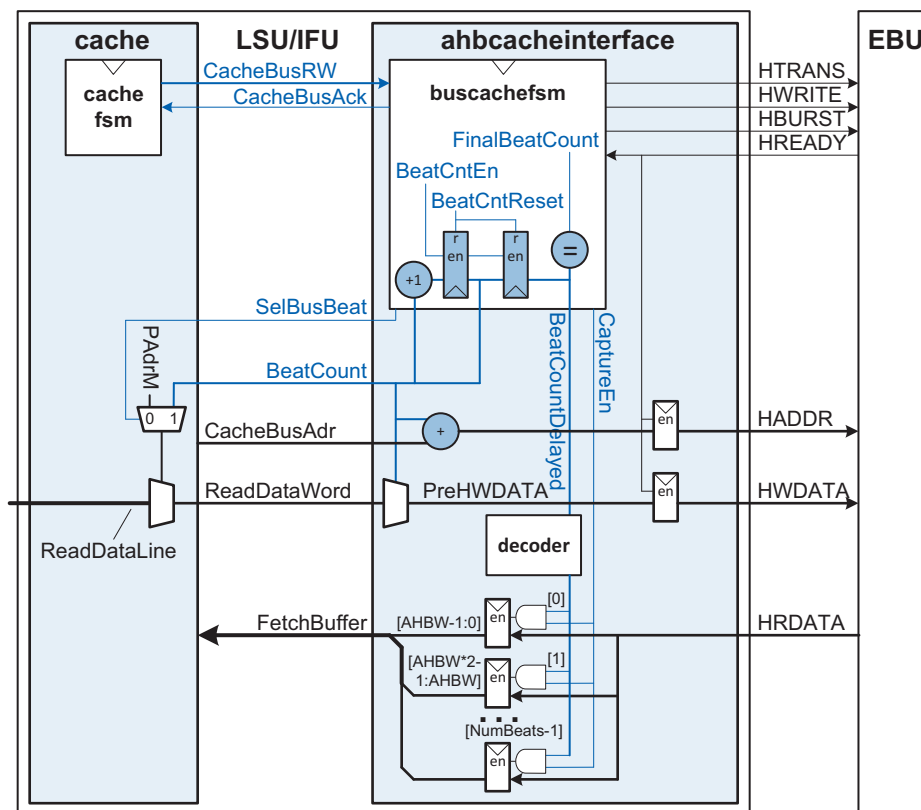


Fig. 12.4 Cache, AHB cache interface, and EBU connections.

The next subsections describe in more detail how the cache uses the `ahbcacheinterface` upon a miss to fetch and write back cache lines from noncore memory using burst mode.

12.2.1 Cache Line Fetches

During a cache miss, the cache selects a victim line to replace, as discussed in [Sections 10.1.7 and 10.4.4](#). If the victim line is dirty, it must be written back to the next level of the memory hierarchy. Then the cache fetches the new line from the next level of the memory hierarchy (e.g., over the AHB bus), writes the line into its data array, and updates its tag array. This section discusses how the LSU handles cache line fetches, and [Section 12.2.2](#) describes how it writes back dirty lines. Note that fetches of a new line occur on both load and store misses; store misses merge the stored word into the fetched line before writing the line into the cache.

To fetch a cache line, the cache places the starting address of the line on `CacheBusAdr`. This address is `IEUAdrM` with the offset bits zeroed. The `cachefsm` asserts `CacheBusRW[1]` to ask the `ahbcacheinterface` to perform the fetch. The `ahbcacheinterface` generates an AHB read burst to fetch all the words in the line into the `FetchBuffer`. `BeatCount` and `BeatCountDelayed` track the position within the burst to select which `FetchBuffer` word is updated by `HRDATA`. Once `FetchBuffer` receives the last word, `ahbcacheinterface` asserts `CacheBusAck`, and the cache writes `FetchBuffer` into the cache.

`ahbcacheinterface` implements burst transfers, introduced in [Section 9.2.2.5](#), to fetch or write consecutive addresses in the cache line. Each access within the sequence is called a *beat*. A burst transfer can maximize external memory throughput because large RAMs (e.g., off-chip DRAM) are faster at reading and writing consecutive addresses than random addresses. The `HBURST` and `HTRANS` signals tell the RAM controller to expect and optimize for such consecutive accesses. AHB interfaces pipeline their Address and Data phases, obtaining a burst throughput of one word per cycle after some initial RAM latency.

AHB burst transfers use a sequence of beats to access consecutive addresses. `HSIZE` defines the word width for a beat, and `HBURST` defines the number of beats. Wally has a bus width of `AHBW` (typically equal to `XLEN`). Each beat uses the full bus width, so `HSIZE` encodes `AHBW` (`HSIZE = 011` for 32 bits or `100` for 64 bits). The number of beats in the burst is the cache line length (`LINELEN`) divided by `AHBW`. `HADDR` is incremented by the bus width (in bytes) after each beat. [Table 12.1](#) summarizes some configurations of line length and bus width and the associated burst length. For example, `rv32gc` fetches a 512-bit line with a burst of 16 32-bit beats.

Because AHB pipelines burst requests, the module tracks the Address phase beat using `BeatCount` and the Data phase beat using

The AHB cache interface's bus cache FSM has a parameter `BURST_EN` that defaults to 1, which enables burst line transfers. It can be disabled by the testbench for testing nonburst transfers. [Section 23.2.2.5](#) compares burst and nonburst performance of an FPGA implementation of Wally that uses DDR3 memory.

Table 12.1 HBURST values.

Configuration	LINELEN (bits)	AHBW (bits)	HBURST	HSIZE
N/A	128	32	011 (4 beats)	011 (32 bits)
rv32gc	512	32	111 (16 beats)	011 (32 bits)
rv64gc	512	64	101 (8 beats)	100 (64 bits)

BeatCountDelayed. During the Address phase, ahbcacheinterface sends the memory address $HADDR = \text{CacheBusAdr} + \text{BeatCount} \ll (\text{AHBW}/8)$. During the Data phase, the AHB cache interface stores HRDATA into FetchBuffer at BeatCountDelayed.

Fig. 9.23 shows the busfsm for a basic AHB interface, sequencing through Adr, Data, and Mem3 phases to read or write a single word over the bus. By adding a cache to the system, the bus must now support both single-word accesses (for peripheral accesses) and multiword bursts (for cache line fetches and writebacks). Fig. 12.5 shows the buscachefsm, an enhanced version of busfsm. It adds the CacheFetch and CacheWriteback states. This section describes cache fetches, and Section 12.2.2 describes cache writebacks. Section 12.3.1 then describes how the PMA checker detects uncacheable devices (e.g., peripherals) and issues single-word accesses; these are introduced in Section 9.1.3.

Cache bus requests start in the Adr state. If the cache requests a line fetch by asserting CacheBusRW[1] and the bus is free ($HREADY = 1$), the FSM enters the CacheFetch state and starts an AHB read burst by setting $HTRANS = 2$ (NON_SEQ) to indicate the first beat in the request. HBURST is set to the number of beats in the burst as shown in Table 12.1. HWRITE is set to 0. If the bus is not ready, the FSM remains in the Adr state.

The ahbcacheinterface uses a BeatCount counter to step through the beats in the burst each cycle the bus is ready ($HREADY$). For example, a DDR memory may have many wait states to service the first beat but then have single-cycle throughput for each subsequent beat. A comparator raises FinalBeatCount at the end of the burst.

The FSM remains in the CacheFetch state until the burst completes. After the first beat completes, $HTRANS$ is set to 3 (SEQ) to signal sequential transfers in the burst. Each beat's HRDATA is written into the next word of FetchBuffer. When the burst completes its last beat, the FSM sends the cache CacheBusAck, and the FSM returns to the Adr state to wait for the next request.

If the memory operation is a load miss, the cache writes the cache line into its SRAM but also bypasses the SRAM to directly return

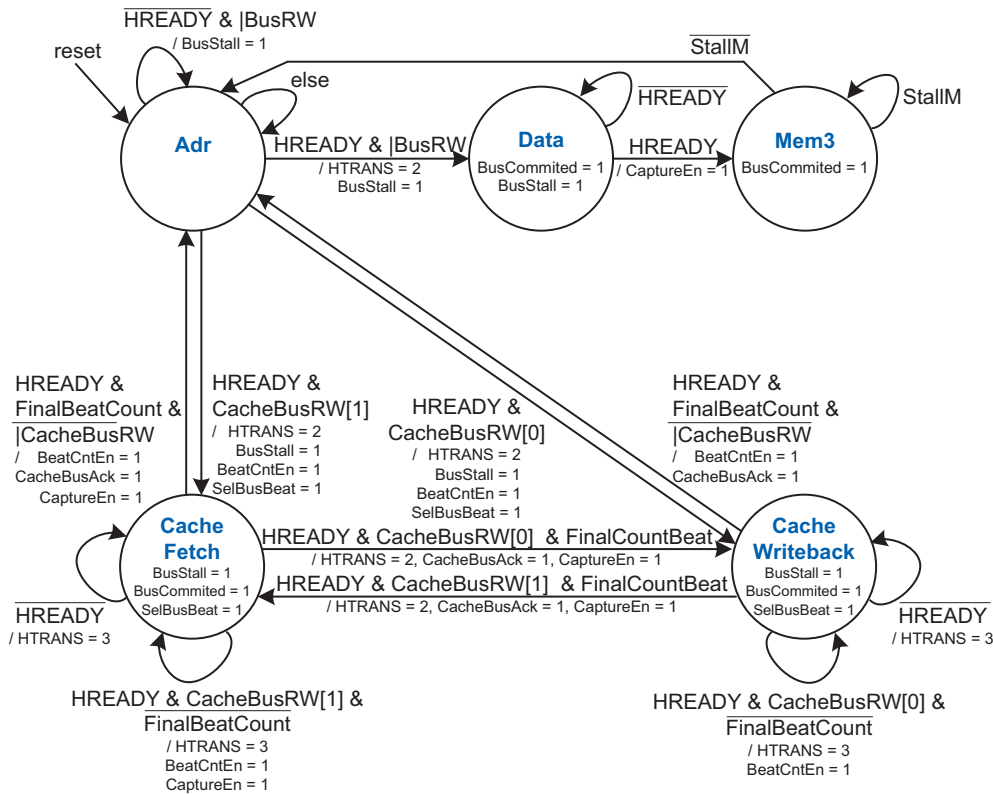


Fig. 12.5 The AHB cache interface's bus cache FSM (buscachefsm).

This path from the FetchBuffer to ReadDataWord is not shown in the LSU diagrams (Fig. 12.2, etc.).

Figs. 10.19 and 12.5 show the cache's FSM (cachefsm) and the bus cache FSM (buscachefsm), respectively.

ReadDataWord from the FetchBuffer to the core. Store misses merge the write data (FinalWriteDataM) with the FetchBuffer signal according to ByteMask before writing the line into the cache.

Code Example 12.1 illustrates cache misses and hits using a sequence of three memory instructions. Assume a word size (XLEN) of 64, a 32-byte (4-word) cache line, and a 64-bit (1-word) bus (AHBW = 64). Suppose the first load misses, and the cache fetches the line containing addresses 0x20-0x3F. Thus, the second load and store will hit. Fig. 12.6 shows the timing diagram for these instructions with reference to the CPU pipeline, an AHB bus read burst, and the cachefsm handshake with buscachefsm. The first load misses, which triggers a fetch burst of four 64-bit words. The first word has two wait states ($\text{HREADY} = 0$), and the next three words have single-cycle throughput. The next load and store hit the cache, so the bus is not involved.

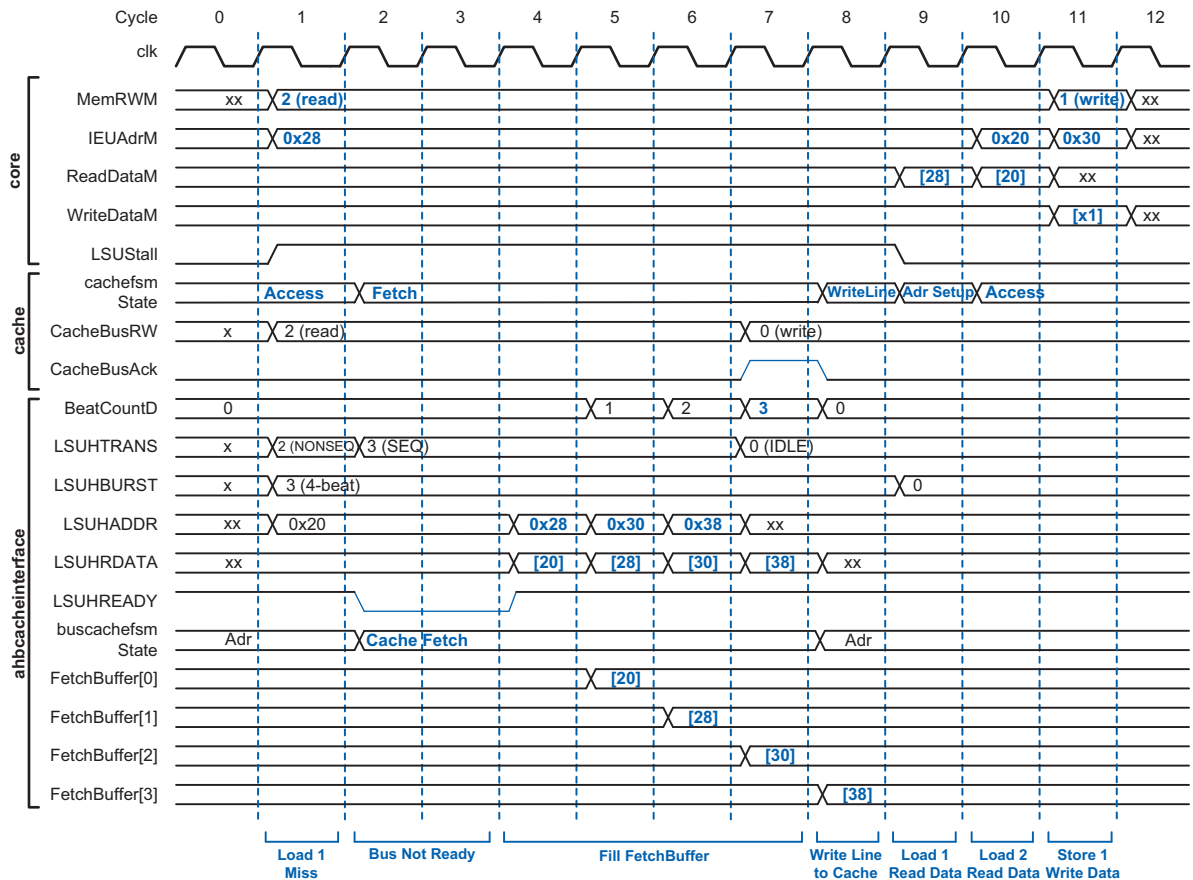


Fig. 12.6 Timing of a load miss followed by a load hit and store hit.

Code Example 12.1 LOAD MISS FOLLOWED BY LOAD AND STORE HITS

Sample code to demonstrate a load miss:

```
ld x1, 0x28(x0)
ld x2, 0x20(x0)
sd x1, 0x30(x0)
```

12.2.2 Cache Line Writeback

The `buscache fsm` handles cache line writebacks similar to how it handles fetches. The `CacheWriteback` state initiates a write burst similar to the fetch burst. Fig. 12.7 shows the timing for writing back an evicted line and then fetching the missing line. The data in each line is as follows:

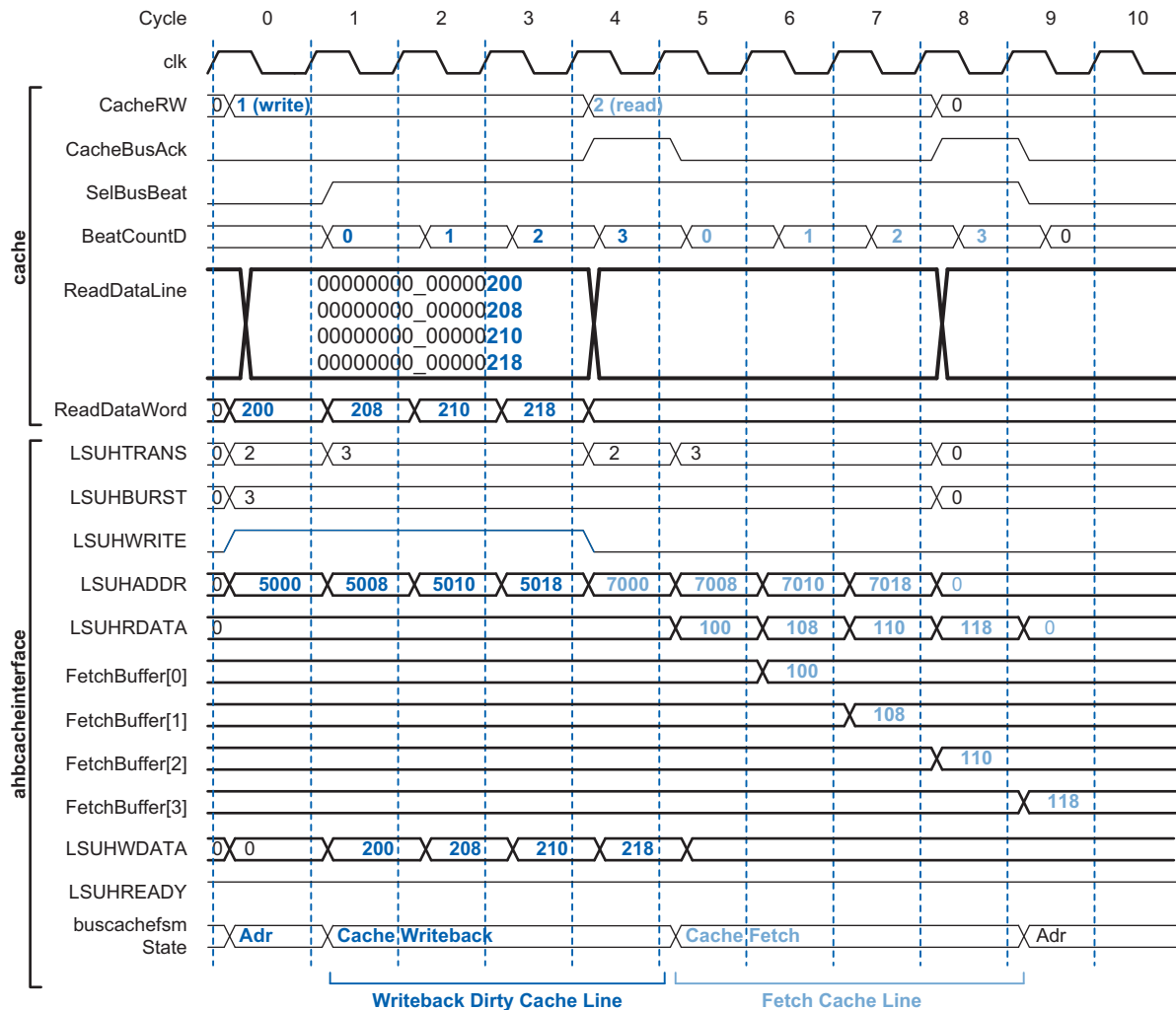


Fig. 12.7 Timing diagram of a fetch that requires a writeback.

- The evicted line's data (at address 0x5000):

0x00000000_00000218_00000000_00000210_00000000_00000208_00000000_00000200

- The missing line's data (at address 0x7000):

0x00000000_00000118_00000000_00000110_00000000_00000108_00000000_00000100

As with Fig. 12.6, the example assumes 8-byte words, a 32-byte (4-word) line, and a 64-bit (8-byte) bus, so each burst has four beats. Again, `HTRANS` is `NON_SEQ` on the first beat and `SEQ` on each of the subsequent beats. The `BeatCount` counter controls the `SubCacheLineRead (SCR)` multiplexer (Fig. 12.4) by asserting `SelBusBeat` during writebacks to step through the words of the cache line to write a 64-bit word over the bus in each beat. This example assumes no wait states, so `HREADY` is always high. `HWRITE` is asserted for the line writeback (it was 0 for the line fetch). The `CacheBusRW` and `CachBusAck` signals handshake between the `cache fsm` (Fig. 10.19) and `buscachefsm` (see Fig. 12.5). `CacheBusRW[1:0] = 1 (012)` to writeback and 2 (10₂) to fetch.

Observe from Fig. 12.5 that the `buscachefsm` transitions directly from the `CacheWriteback` state to the `CacheFetch` state without an idle `Adr` state, pipelining the last beat Data phase of the writeback with the first beat Address phase of the fetch and saving a cycle of latency. Similarly, the `buscachefsm` can write back multiple lines consecutively without intervening idle states; this occurs during a cache flush, as described in Section 10.4.5.4.

12.2.3 Burst EBU

The EBU from Section 9.5.2 arbitrates between the IFU and LSU AHB requests, with the LSU having priority. Fig. 12.8 enhances the EBU to support burst transactions, with the changes shown in blue. The IFU's `controllerinput` module adds a register and multiplexers to save and restore the IFU burst request, `IFUHBURST`, during concurrent IFU and LSU requests. Both the IFU's and LSU's `controllerinput` modules add an AND gate to force their `HREADY` signals (`IFUHREADY` or `LSUHREADY`) to 0 when the other module has a burst in progress, which delays new requests until the in-flight request completes. The `controllerinput` module ORs the `HTRANS` bits to signal a request (`IFUReq` or `LSUReq`). `HTRANS` is 2 (`NON_SEQ`) for the first word of a burst request and 3 (`SEQ`) during the remainder of the request.

The EBU's FSM arbiter (`ebufsmarb`) (originally shown in Fig. 9.26) resolves requests between the `buscachefsm`s of the IFU and LSU. It detects requests and produces control signals to grant AHB access to the LSU or IFU. `ebufsmarb` is enhanced in Fig. 12.9 to support burst mode. It adds a beat counter to track when the burst completes. The lookup table (LUT) decodes the number of beats from `HBURST`, according to Table 12.1. When an LSU request arrives while the IFU is in the midst of a burst, the `LSUDisable` signal asserts to delay the LSU request.

In its initial design, the EBU could disable IFU requests only, not LSU requests (see Fig. 9.25). In a system without virtual memory, this worked, because concurrent LSU and IFU requests only occurred in the same cycle, and the LSU always had precedence. However, in a system with virtual memory, LSU requests can occur after an IFU request is in flight because page table walks occur during a stalled pipeline.

For example, suppose a DTLB miss and I\$ miss occur in the same cycle. The DTLB miss triggers a page table walk, and the I\$ miss causes an IFU AHB request. While the IFU request is in flight, the HPTW traverses the page tables using the D\$. The HPTW's first PTE access hits in the D\$.

However, its second access misses, which generates an LSU AHB request to fetch the missing cache line. Because LSU requests were not disabled and the LSU has precedence, the LSU request preempts the existing IFU cache line fetch, leaving stale instructions in the I\$. This caused Wally to hang during Linux boot.

To fix this, the EBU added the IFUReq and LSUDisable signals. LSUDisable delays the LSU request in the example above until the IFU transaction completes. This was one of the most difficult bugs to figure out while designing Wally.

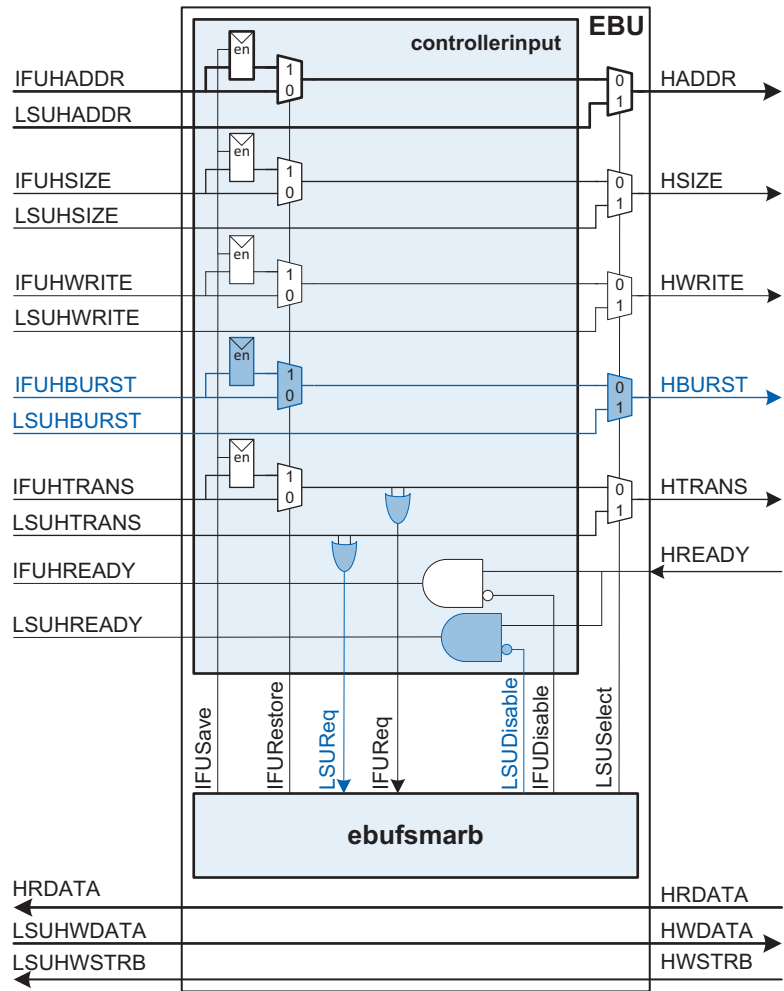


Fig. 12.8 EBU with burst support.

12.3 MEMORY MANAGEMENT AND HPTW INTEGRATION

This section integrates the main MMU components and the HPTW into the LSU, one by one. The three major MMU components are:

- ▶ PMA checker
- ▶ PMP checker
- ▶ TLB

These components rely on CSRs to handle memory exceptions and enable address translation, so the MMU is enabled when `ZICSR_SUPPORTED = 1`. The TLB also depends on `VIRTMEM_SUPPORTED = 1`,

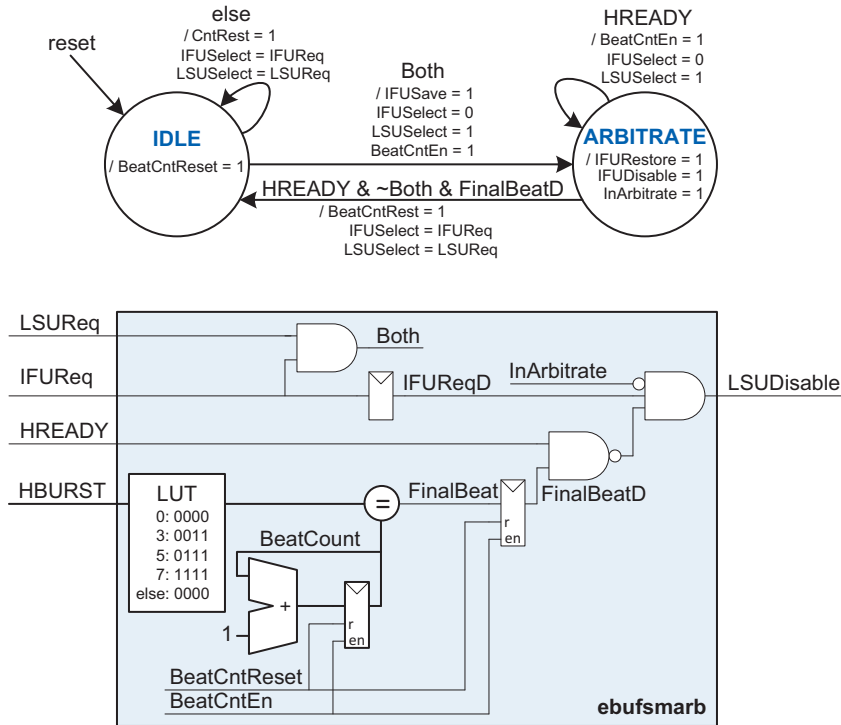


Fig. 12.9 Fully featured **ebufsmarb** with burst support.

and the PMP depends on `PMP_ENTRIES > 0`. With these MMU capabilities, Wally now has everything needed to boot Linux except atomic memory operations, which are described in [Chapter 17](#).

12.3.1 PMA Checker

The PMA checker provides address decoding according to the memory map to select which memory device is accessed for an address range and what attributes that device has, as described in [Section 11.2.2.1](#). Whereas main memory is cacheable, the peripherals are accessed directly over the bus. [Fig. 12.10](#) enhances the LSU of [Fig. 12.3](#) with PMA support to determine whether an access is cacheable. These new components are highlighted in blue. Cache flushes take priority and disregard the PMA by ORing `FlushDCacheM` with `CacheableM` to produce `CacheableOrFlushM`. Uncached accesses use AHB rather than the cache by driving `LocalHADDR = IEUAdrM` and `PreHWDATA = LSUWriteDataM` (for stores) or `ReadDataWordMuxM = FetchBuffer` (for loads).

The read and write control signals should go to either the cache or bus depending on whether the access is cacheable. [Fig. 12.11](#) shows a

demultiplexer added to the LSU to drive MemRWM from the IEU to either CacheRW or BusRW.

With the PMA integration, the LSU supports a fully featured set of seven types of memory operations, summarized in Table 12.2.

Table 12.2 Types of LSU memory operations.

Operation	Cacheable	Flush	Cache Access Type	LSUHADDR	ReadData WordMuxM	PreHWDATA	CacheWrite Data
Uncached load	0	0	BusRW = 10	cIEUAdrM	FetchBuffer [AHBW-1:0]	N/A	N/A
Uncached store	0	0	BusRW = 01	IEUAdrM	N/A	LSUWrite DataM	N/A
Cached load hit	1	0	CacheRW = 10	N/A	ReadData Word	N/A	N/A
Cached store hit	1	0	CacheRW = 01	N/A	N/A	N/A	LSUWrite DataM
Cache fetch	1	0	CacheBusRW = 10	CacheBusAdr + BeatCount	N/A	N/A	FetchBuffer
Cache writeback	1	0	CacheBusRW = 01	CacheBusAdr + BeatCount	N/A	ReadData Word	N/A
Cache flush	N/A	1	CacheBusRW = 01	CacheBusAdr + BeatCount	N/A	ReadData WordM	N/A

12.3.2 PMP Checker

The LSU's MMU contains an optional PMP checker described in Section 11.4.1.2. No LSU modifications are required. Together, the PMA and PMP checkers generate load- and store-access faults and misaligned faults, which are passed onto the privileged unit.

12.3.3 TLB and Cache Integration

Recall that when virtual memory is enabled, the program uses virtual addresses, but the memory system expects physical addresses. As discussed in Section 11.1.1, the processor translates virtual to physical addresses using a multilayer page table. A brute force implementation would slow down the processor with multiple page table memory accesses for each load or store, so a practical system adds a TLB to cache recently used translations.

The TLB is queried with the virtual address and returns a physical address and a hit signal. This section describes LSU accesses when the TLB hits, and [Section 12.3.4](#) describes how the HPTW handles TLB misses.

As mentioned in [Section 11.1.1.7](#), accessing a cache with a virtual address leads to synonym problems when multiple virtual addresses refer to the same physical address. These problems are avoided by indexing the cache with the least significant bits, which are the same for virtual and physical addresses. After translation, the tags (upper bits) are then checked against the physical tag. Therefore, the cache and TLB accesses occur in parallel, and the tag output of the cache is compared against the physical address output of the TLB. The size of a cache way is limited to the 4-KiB page size so that the cache set bits are in the physical page offset, which are available immediately without waiting for address translation.

[Fig. 12.12](#) shows the LSU with a full MMU, including a TLB. When translation is off, the physical address $PAddrM$ comes directly from $IEUAdrM$. When translation is on, the TLB looks up $PAddrM$ concurrently with the cache read. In either case, the cache hit logic compares the tags to $PAddrM$.

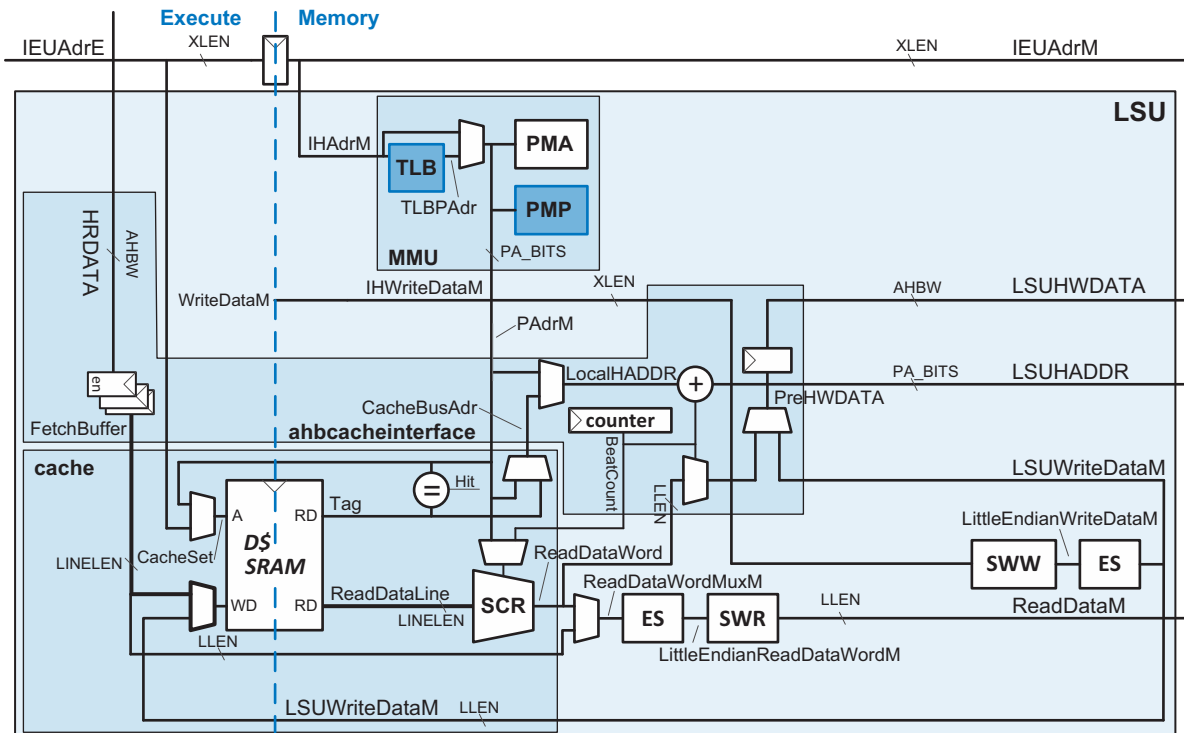


Fig. 12.12 LSU with full MMU and TLB address translation.

Like the cache, the `ahbcacheinterface` uses the physical address `PAdrM` in place of `IEUAdrM` for uncached memory operations. The cache and TLB are the boundary between virtual and physical addressing, which ensures that the bus always uses physical addresses.

With the inclusion of the full MMU, the LSU is nearly complete, but it lacks a mechanism to handle TLB misses. The next section integrates the HPTW from [Section 11.4.2](#), which completes this last major LSU feature.

12.3.4 HPTW Integration

When virtual memory is enabled and the TLB misses, the processor must walk through the multilevel page table to *refill* the TLB with the missing page table entry (PTE). On a TLB miss, RISC-V may trigger a page-fault exception to ask the operating system to refill the TLB, but this can add hundreds of cycles to a TLB miss. Most application processors, including Wally, provide a HPTW to refill the TLB, as described in [Section 11.4.2](#). TLB fill requests can come from either the ITLB (in the IFU) or DTLB (in the LSU), but the requests are all serviced by the single HPTW located in the LSU.

The HPTW issues a sequence of memory reads to each level of the multilevel page table, so it must have access to the memory system. The HPTW asserts `SelHPTW` to override other memory accesses and instead select the HPTW's address, control, and data during the page table walk.

The HPTW is located in the LSU and routes its memory reads through the LSU's D\$ because this:

- ▶ Ensures coherency (recently changed PTEs may be in the D\$ but not yet in main memory).
- ▶ Enables updating (writing) a PTE's accessed and dirty bits because the D\$, unlike the I\$, is writeable.
- ▶ Improves performance for page table walks because PTEs may be in the D\$ instead of main memory.

[Fig. 12.13](#) shows the LSU enhanced with the HPTW. [Fig. 11.25](#) shows the simplified internal logic of the HPTW and [Section 11.4.2](#) describes its operation. During a page table walk, the HPTW outputs physical addresses as `IHAdrM` so the PMA and PMP checker can verify the access is valid. The HPTW's `IHAdrM` mux (which is internal to the Datapath + FSM logic) chooses `IEUAdrM` (for regular loads and stores) or `HPTWAdr` (for a TLB miss and subsequent page table walk). Because the HPTW handles both ITLB and DTLB misses, the HPTW cobbles together the page table address (`HPTWAdr`) from either `PCF` or `IEUAdrM` (which are both virtual addresses), `satp`, and `ReadDataM`, which is the PTE read during the page table walk.

The MMU drives `IHAdrM` onto `PAdrM`, and the LSU makes an untranslated PTE read from this physical address. It outputs the PTE via `ReadDataM`

`IHAdrM` is a concatenation of names `IEUAdrM` and `HPTWAdr`.

StallD, StallE, StallM, StallW) while LSUS StallM is asserted. Specifically, LSUS StallM asserts during a D\$, HPTW, or bus stall, indicated by:

- ▶ DCacheStallM: asserted by the cache fsm during a cache miss or flush
- ▶ HPTWStall: asserted when the HPTW is processing a TLB miss
- ▶ BusStall: asserted by buscache fsm or bus fsm when the bus is in use

On the last cycle of a memory operation, the respective stalls are lowered, which indicates that ReadDataM is valid or that the store has committed to memory.

The hazard unit also stalls the Decode stage for data dependencies and structural hazards. The forwarding logic in the controller raises LoadStallD when a load result is used by the next instruction and raises StoreStallD when a load follows a store (see [Sections 7.2.2.3 and 10.4.2.3](#)).

12.4.2 LSU Trap Generation

The TLB, PMA, and PMP generate page, access, and misaligned faults by asserting TrapM at the beginning of a memory access, before a cache miss could occur. Uncached memory operations are flushed if the access would generate a fault.

12.4.3 Disabling Interrupts

The LSU raises CommittedM to disable interrupts during a multicycle memory operation, as indicated by any of the following signals:

- ▶ BusCommittedM: a memory operation is in flight on the bus
- ▶ DcacheCommittedM: the cache has partially updated its state
- ▶ SelHPTW: HPTW is walking the page table

When CommittedM is asserted, the privileged unit suppresses pending interrupts, as shown in [Fig. 8.14](#). Unlike LSUS Stall, CommittedM remains high during the last cycle of a memory operation.

12.4.4 LSU Response to Stalls and Flushes

The LSU's response to a stall depends on the current state of the D\$, the bus, and the HPTW. Stalls are first gated by the HPTW with $\text{GatedStallW} = \text{StallW} \ \& \ \sim \text{SelHPTW}$. This allows the HPTW to issue loads and stores through the D\$ while the rest of the pipeline is stalled. GatedStallW is renamed Stall in the cache. The cache fsm and buscache fsm only check Stall at the end of a multicycle memory operation, which allows the cache or bus to continue with the current operation while

the pipeline is stalled. On the last cycle of a memory operation, if `Stall` is high, the `cachefsm` remains in the final, `AddressSetup`, state. This prevents the cache from executing the same memory operation over and over. Similarly, when `Stall` is asserted, the `buscachefsm` waits in its last state (`Mem3`, see Fig. 12.5) for uncached operations. However, `buscachefsm` does return to the `Adr` state after a cache line fetch or writeback so that the bus no longer asserts `BusStall`. This lowers `LSUStallM`, which allows the hazard unit to unstick the pipeline.

The Memory stage may be flushed in three cases: on the first cycle of a cache miss, on an uncached bus access due to a memory fault, or when another exception or interrupt flushes the pipeline. However, when a memory operation is in flight, Wally will not generate faults until that memory operation completes. In-flight cache fetch and writeback addresses were already checked by the PMA and PMP checkers, so they will not generate a fault when they are in flight. Specifically, cache fetches access `PAdrM` with the byte offset bits zeroed and cache writeback addresses are always the Victim Way's tag + the set bits from `PAdrM`, with zeros for the byte offset. These addresses were checked by the PMA and PMP when the cache line was first fetched.

12.5 EXAMPLES

This section illustrates the LSU in action, showing the active paths and timing diagrams for D\$ hits and misses during loads and stores, uncached accesses, and TLB hits and misses.

12.5.1 Cache Load Hit

A load hit, highlighted in blue in Fig. 12.14, begins in the Execute stage using `IEUAdrE`, the address for accessing the D\$'s SRAM. The cache uses the page offset bits, which are never translated, to choose the set in the cache. It reads the data, tags, and status bits. In the Memory stage, the TLB translates the virtual address `IEUAdrM` to the physical address `PAdrM` if address translation is enabled; otherwise, $PAdrM = IEUAdrM$. The cache compares its tags to `PAdrM` to indicate a hit and outputs all bytes in the cache line on `ReadDataLine`. The sub-cache line read (SCR) module uses the offset bits of `PAdrM` to select the word within `ReadDataLine` to produce `ReadDataWord`. Finally, because it is a hit, the mux passes `ReadDataWord` to `ReadDataWordMuxM`, optionally performs an endianswap (ES), and selects the final `ReadDataM` using the subwordread (SWR) module. The `cachefsm` (Fig. 10.19) remains in the Access state throughout the load hit.

12.5.2 Cache Store Hit

Like loads, a store begins by reading the cache SRAM and optionally translating the virtual address `IEUAdrM` to the physical address `PAdrM`. The



Fig. 12.16 shows the timing diagram for consecutive load, store, load, and load hits to addresses 0x10, 0x18, 0x20, 0x08, which all hit in the same set. The first load and store have single-cycle throughput. The second load stalls for a cycle because of the store delay hazard caused by a store writing the cache data array during the Writeback stage and the following load attempting to read the cache data array in the same cycle in the Memory stage. In cycle 3, the controller detects the hazard in the Decode stage when a store is executing and a load is in the Decode stage. It asserts `StallD` to delay load 2 by a cycle and to insert a pipeline bubble between the two

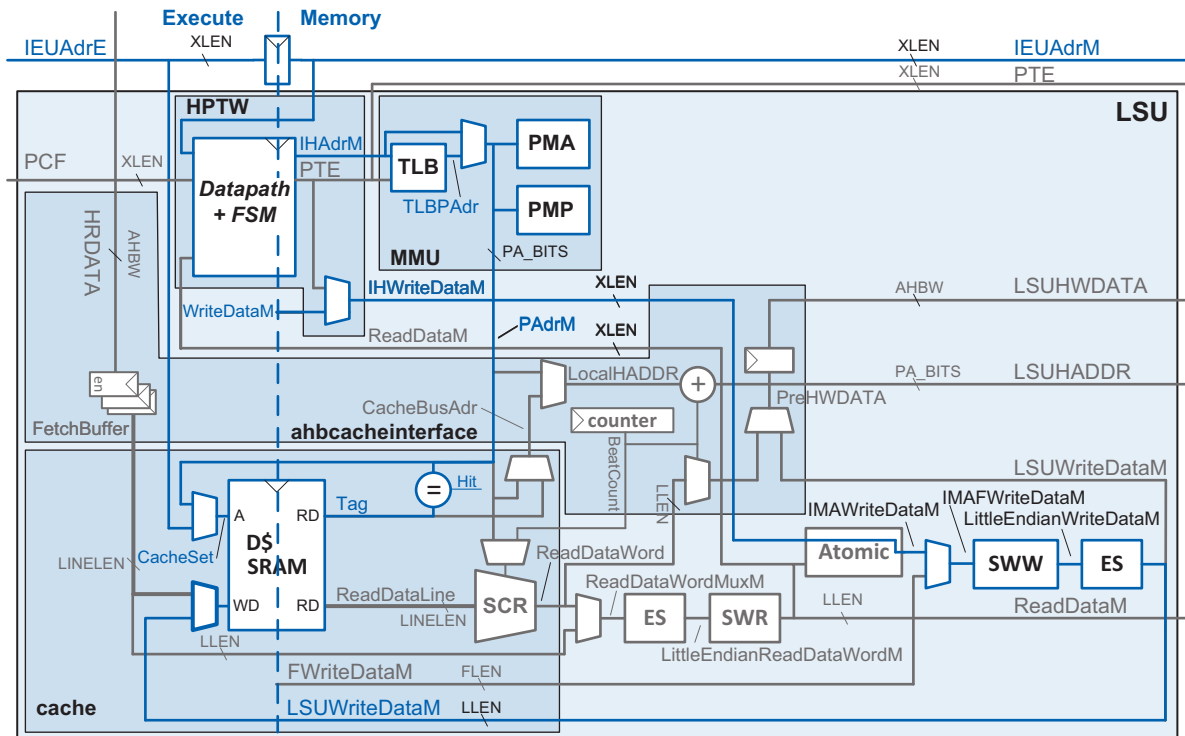


Fig. 12.15 Example store hit.

instructions. The last load has single-cycle throughput. All four accesses share the same cache line at set 0 (`CacheSet = 0`). The cache memory arrays are written on the rising clock edge at the beginning of cycle 5 (in the store's Writeback stage). On the rising clock edge at the beginning of cycle 6, the memory switches back to reading and `ReadDataLine` now contains the updated data from the store. An 8-byte word is written, so `ByteMask = 11111111 (0xFF)`, and the fourth 8-byte word of the 8-word line is written, so `LineByteMask` expands this to `0x00000000FF000000` to indicate which 8 of the 64 bytes in the line should be written.

12.5.3 Cache Miss

On a load or store miss, the cache fetches the missing line from the next level of memory over the bus, as described in [Section 12.2.1](#). [Fig. 12.17](#) highlights the LSU path for fetching a cache line from the bus, and [Fig. 12.6](#) shows the timing diagram. The cache outputs the cache line address, `CacheBusAdr`. `ahbcacheinterface` outputs `BeatCount` to sequence through the words of the cache line and to increment `LSUHADDR`, then raises `CacheBusAck` when `FetchBuffer` contains the cache line.

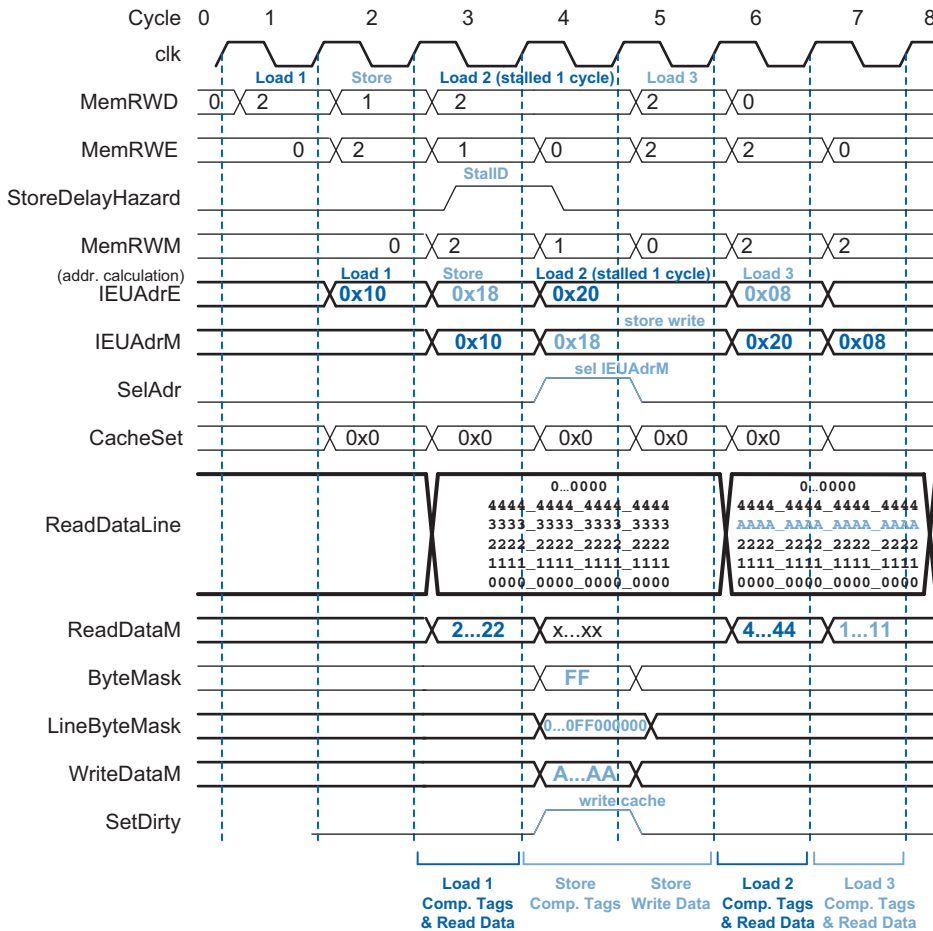


Fig. 12.16 Timing diagram of consecutive load, store, load, and load hits.

Section 12.2.2 describes a writeback of a dirty cache line. Fig. 12.18 highlights the cache line writeback paths in the LSU, and Fig. 12.7 shows the timing diagram.

12.5.4 Uncached Access

The LSU makes uncached accesses to uncacheable devices such as I/O peripherals and also in cacheless systems with external memory. Section 9.5.1 describes how the `ahbinterface` module makes single `NON_SEQ` bus requests. Section 12.3.1 reintroduces the PMA checker to select between cached and uncached bus access. Fig. 12.19 highlights the LSU uncached load path, and Fig. 12.20 highlights an uncached store.

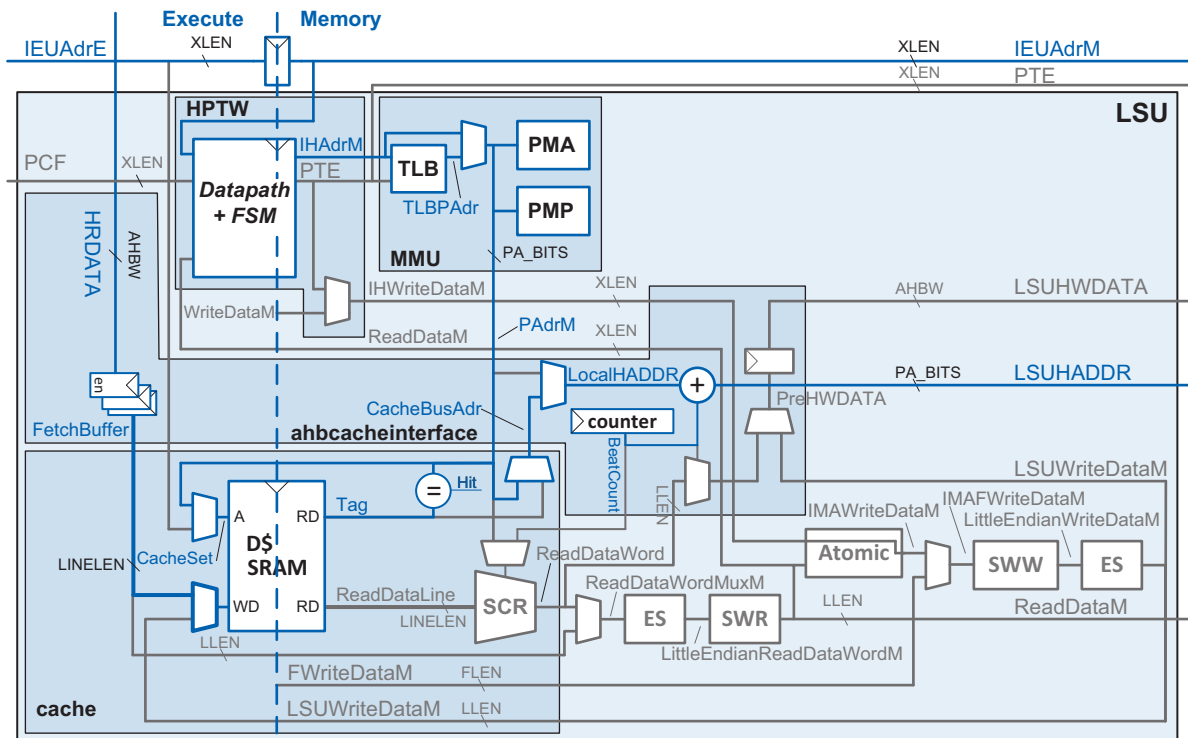


Fig. 12.17 Example cache line fetch.

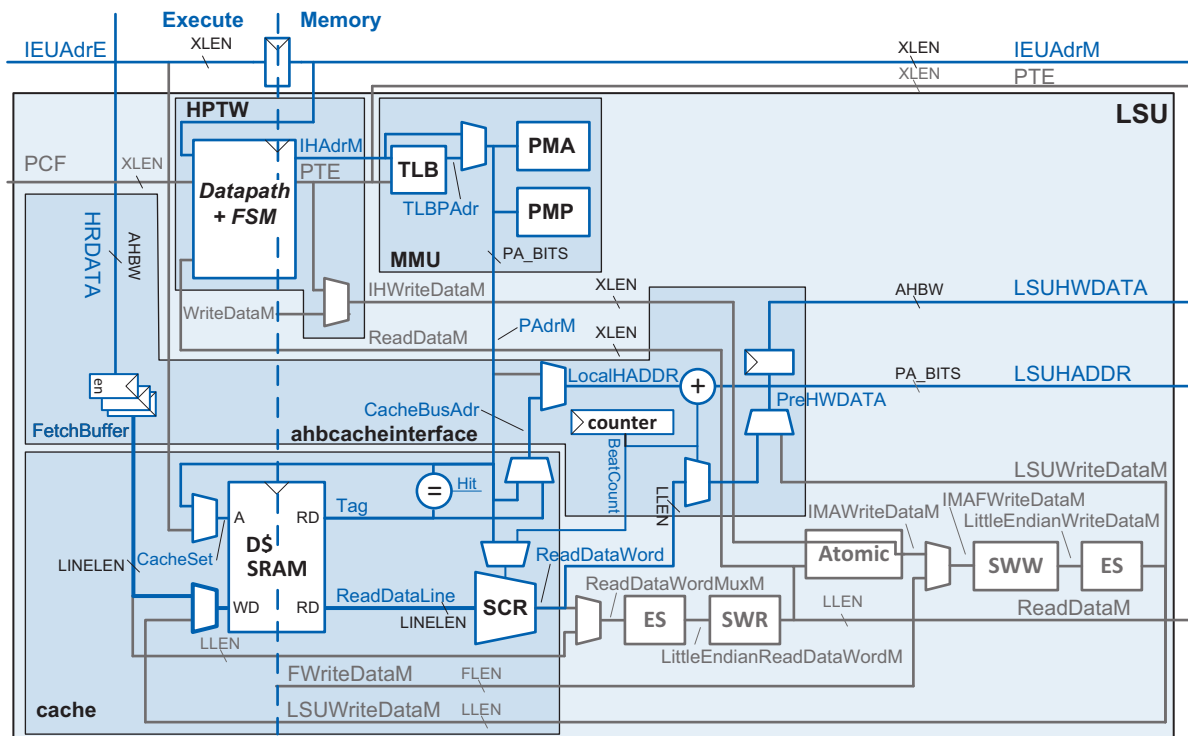


Fig. 12.18 Example cache line writeback.

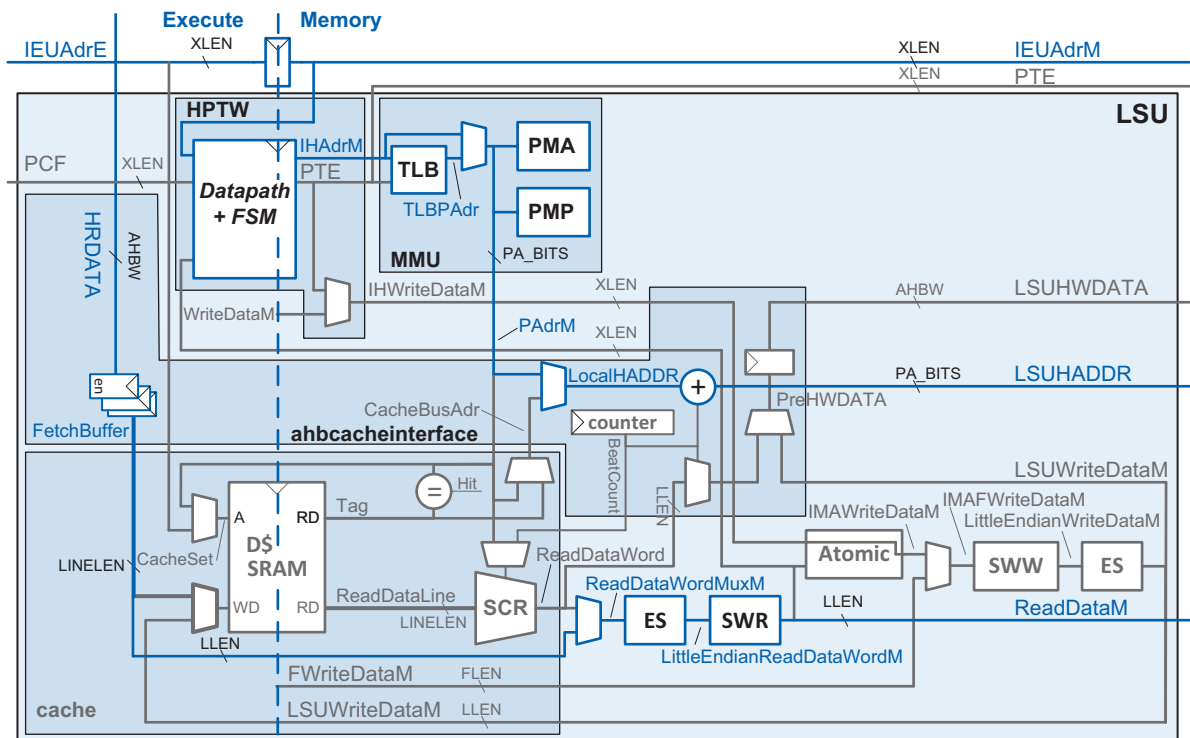


Fig. 12.19 Example uncached load.

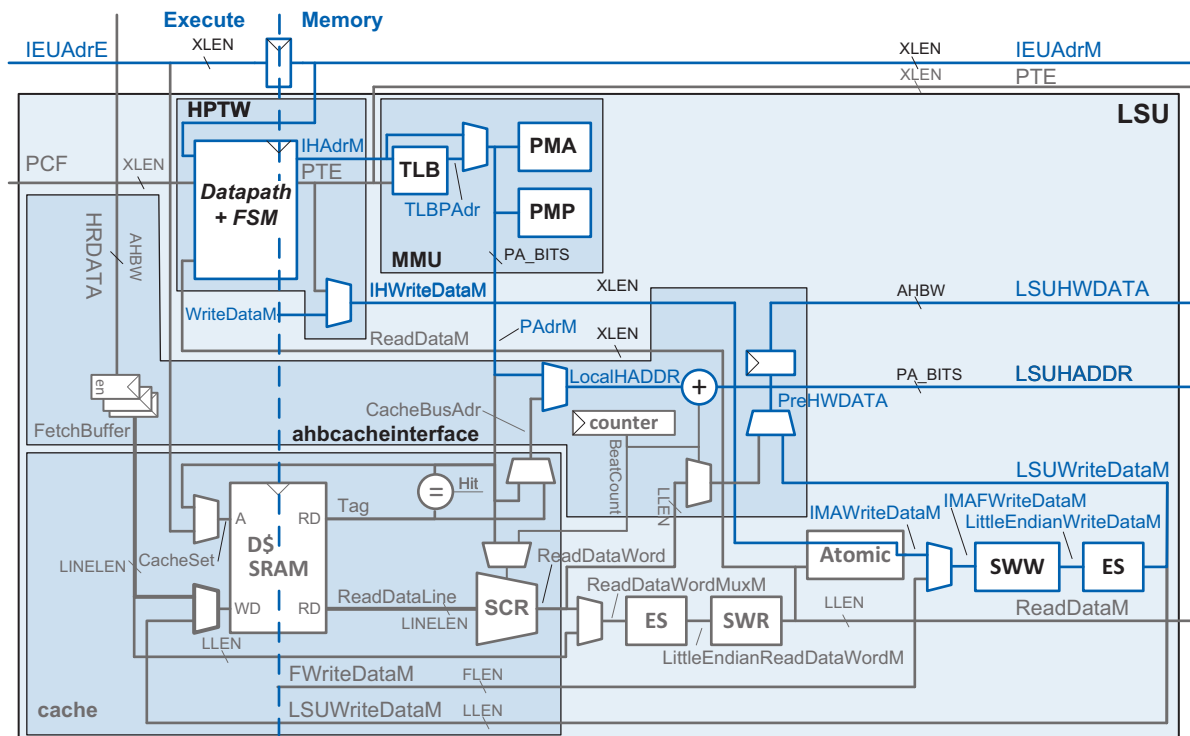


Fig. 12.20 Example uncached store.

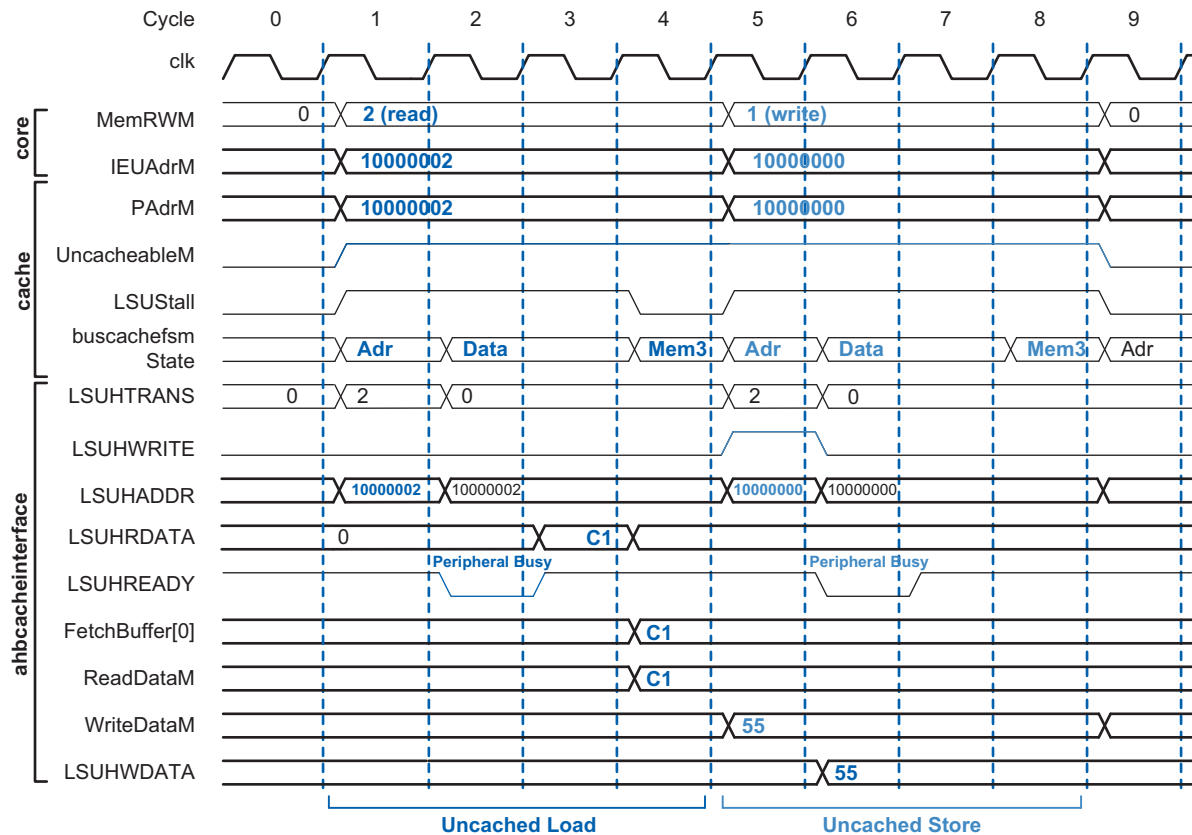


Fig. 12.21 Timing diagram of a load followed by store to an uncached address.

Fig. 12.21 shows the timing of an uncached load followed by an uncached store. The load begins in cycle 1 loading one byte from peripheral address 0x10000002 by setting `LSUTRANS` to 2. During cycle 2, the peripheral indicates it needs more time by lowering `LSUHREADY`. The peripheral outputs `HRDATA` = 0xC1 in cycle 3 and raises `LSUHREADY` to indicate it is ready. The `buncache fsm` captures the data into `FetchBuffer[0]` and outputs `ReadDataM` on cycle 4. The store follows a similar pattern, setting the address to 0x10000000 and `LSUTRANS` to `NON_SEQ` (2). This time, `LSUWRITE` asserts, and `WriteDataM` = 0x55. Just like the load, the peripheral is slow to write and inserts a wait state in cycle 6. During cycle 7, `LSUHREADY` rises to indicate that the peripheral completed the write.

12.5.5 TLB Miss

The LSU processes TLB misses by fetching the missing PTE in the same way it would process a usual memory read.



Suppose the level 1 page table read hits in the D\$, and the level 0 read misses the D\$ and must be fetched from main memory. Fig. 12.22 highlights the level 1 page table hit. The HPTW produces $\text{IHADRM} = 8000\text{D}000$, and the MMU selects this address as PAdrM . The cache uses the set bits $\text{CacheSet} = 0$ and reads out the PTE on ReadDataLine on the next cycle. The cache extracts $\text{ReadDataM} = 20004\text{CC}1$.

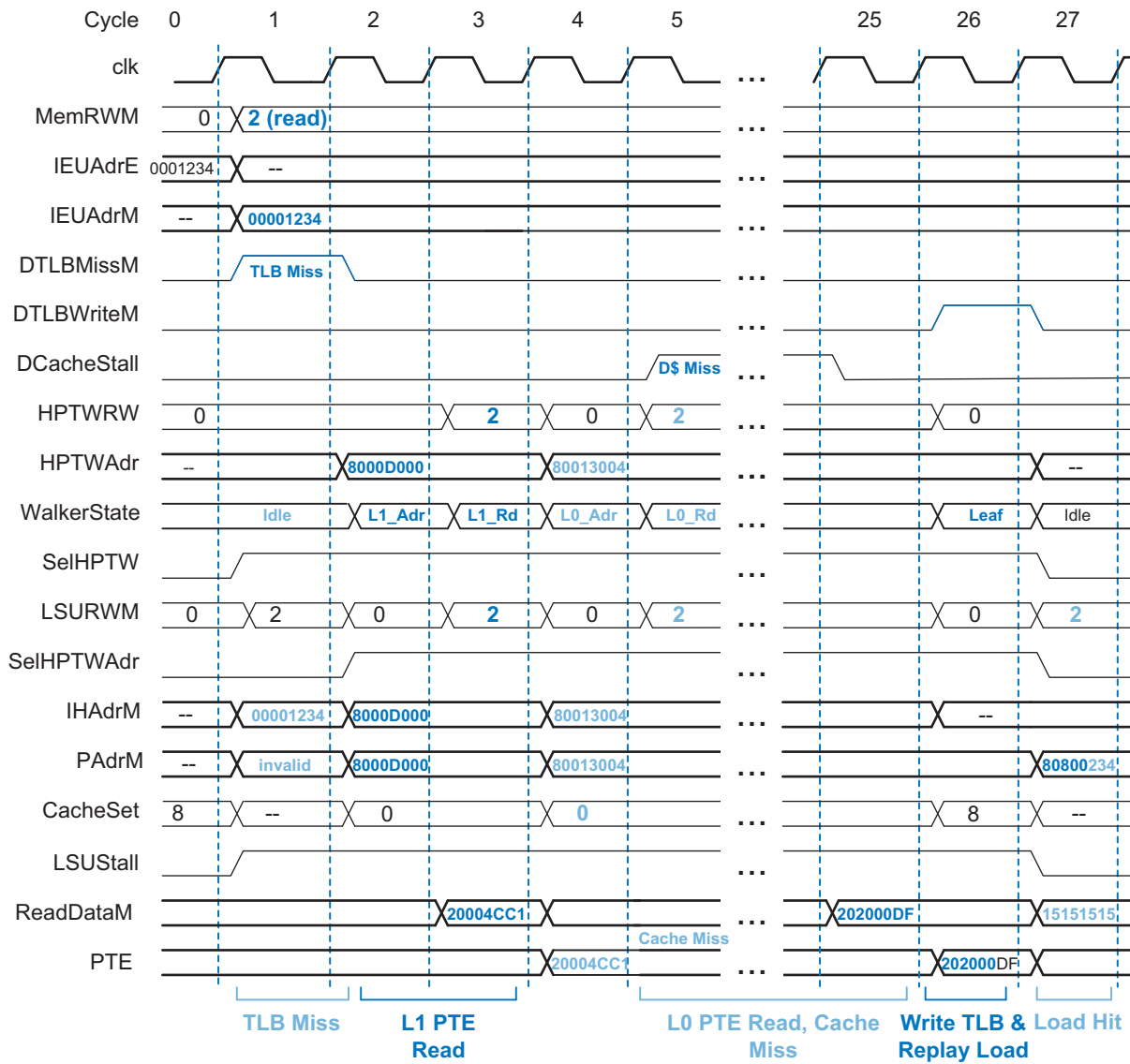


Fig. 12.23 TLB miss timing.

Fig. 12.23 shows a timing diagram of the TLB miss. In cycle 1, the TLB misses ($DTLBMissM = 1$). In cycles 2 and 3, the HPTW reads the level 1 PTE from the cache, as described earlier. On cycle 4, the HPTW produces address $IHAAdrM = 80013004$, and the MMU selects this address as $PAdrM$. The cache uses the set bits $CacheSet = 0$ and misses on cycle 5. The cache then fetches the missing line over the bus and eventually returns $ReadDataM = 202000DF$ to write into the DTLB. Now the LSU replays the original memory access. The DTLB hits, translating virtual address 00001234 into physical address

$P_{\text{Ad}rM} = 80800234$. This address could trigger another cache miss and fetch, but the following instruction is shown as a load hit to save space.

12.5.6 TLB Access Fault

When the processor accesses a page, it must update the access bit (A) if it is 0 and the dirty bit (D) if it is 0 on a write, as described in [Section 11.4.2.2](#). If the hardware supports page table updates ($SVADU_SUPPORTED = 1$), the HPTW issues a write to the PTE in memory while concurrently updating the TLB. Otherwise, the system generates a page fault, and the operating system performs the update in software.

Consider a store to virtual address 00801450 ($VPN = 00801$).

- ▶ Suppose VPN 00801 is in the TLB.
- ▶ But the PTE of 20200817 ($PPN = 080802$, bits $DAGU_XWRV = 0001_0111$) at physical address 80015004 from [Fig. 11.10\(b\)](#) has $A = D = 0$.
- ▶ If the hardware supports page table updates, it must set these A and D bits and write the new PTE of 202008D7 (which has $DAGU_XWRV = 1101_0111$) back to physical address 80015004.

[Fig. 12.24](#) highlights this PTE update. The HPTW sets $IH_{\text{Ad}rM}$ to 80015004, which is selected as $P_{\text{Ad}rM}$. The cache selects $CacheSet = 0$.

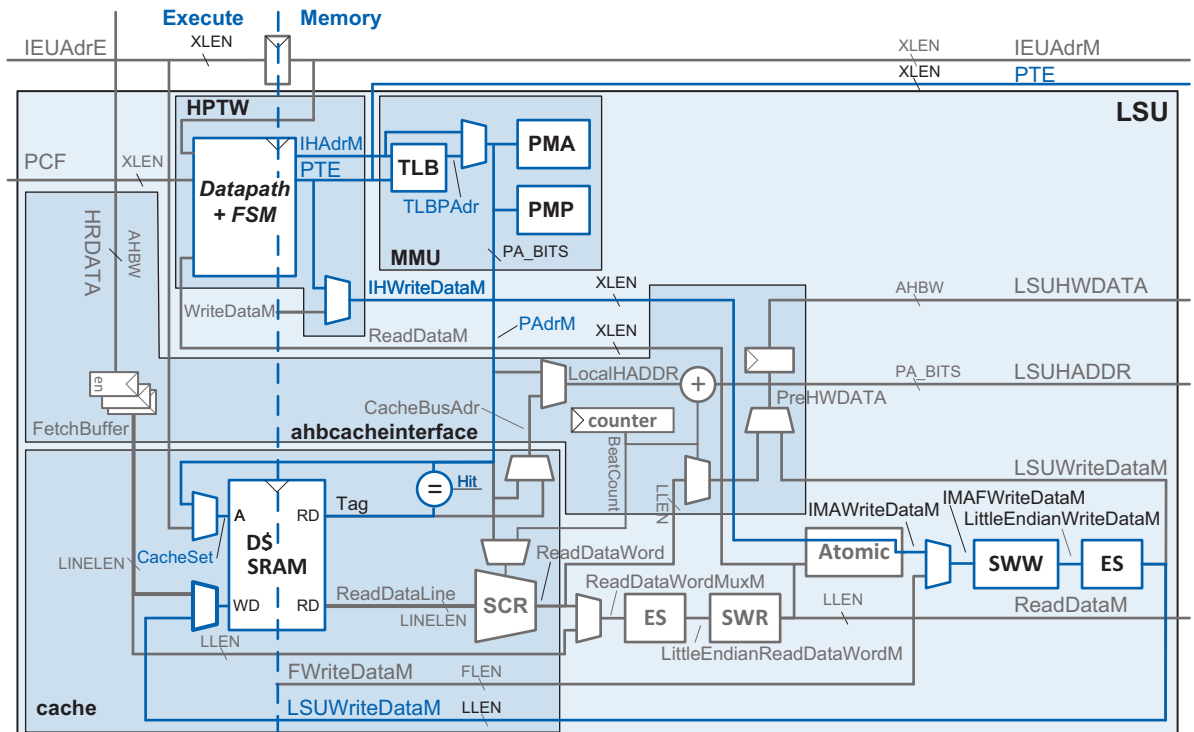


Fig. 12.24 HPTW writes access (A) and dirty (D) bits to PTE in memory.

On cycle 2, the cache detects a hit and sets up `LSUWriteDataM = 202008D7`. On cycle 3, the cache writes this value into its data array. Meanwhile, the HPTW updates the TLB with the new PTE.

12.6 TEST PLAN

The basic features of the LSU are tested with existing tests, including the load/store tests from [Section 7.4](#), the privileged memory access fault tests from [Section 8.3](#), the bus tests from [Section 9.4](#), and the MMU tests from [Section 11.3](#).

A more comprehensive test plan would stress the cache and its interaction with the HPTW.

12.7 SUMMARY

The LSU is a highly configurable module responsible for loads and stores. In the simplest configuration, it only has a DTIM. In the most sophisticated configuration, it interfaces to the EBU for main memory access, has a D\$ to service most accesses, contains an MMU with a TLB for virtual to physical address translation, a PMP for physical memory protection, and a PMA for physical memory address checking; it also has an HPTW to service a TLB miss by the HPTW taking over the memory system to walk the page table. All these components have been described in earlier chapters. Some LSU features needed to coordinate among the components include the following:

- ▶ The `buscachefsm` to sequence cache line fetches and writebacks
- ▶ Multiplexers so that the HPTW can take over the address and write data signals during page table walks
- ▶ Stall logic for multicycle LSU operations and data dependencies
- ▶ Trap logic for page, access, and misaligned faults
- ▶ Disabling interrupts when a multicycle memory operation is in flight

Instruction Fetch Unit

- 13.1 Branch Prediction Principles
- 13.2 Test Plan
- 13.3 Wally Implementation
- 13.4 Summary

The *instruction fetch unit* (IFU) is the *front end* of the processor responsible for delivering instructions to the pipeline. It contains the program counter logic, instruction memory, and IMMU (instruction memory management unit). Chapter 14 adds IFU decompression logic for compressed instructions.

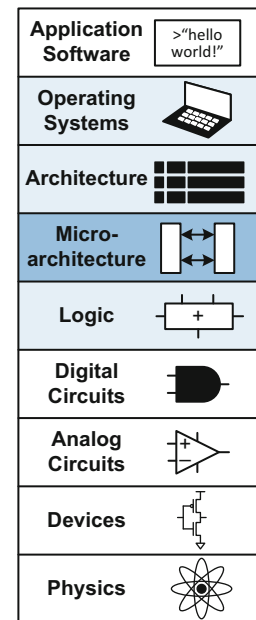
Like the LSU, the IFU also supports several memory options: a cache, tightly integrated memory (TIM), and bus interface to external memory. The memory system is somewhat simpler because the IFU holds instructions, which are read-only. The IMMU also resembles the LSU's MMU but relies on the LSU's hardware page table walker (HPTW). The major distinguishing feature of the IFU is the PC logic, which includes an optional branch predictor to improve performance.

Fig. 13.1 shows the Wally block diagram, with blue highlighting for the new IFU features integrated in this chapter. The PC logic generates the address of the next instruction. The IFU accesses the I\$ (instruction cache) or the ITIM (instruction TIM) or uses the external bus to access external memory. Meanwhile, within the IMMU, the ITLB (instruction translation lookaside buffer) translates the PC from virtual to physical and the PMA and PMP checkers detect illegal accesses. ITLB misses are serviced by the LSU's HPTW.

This chapter introduces the principles of branch prediction. It then explores the Wally IFU implementation, including the I\$, branch predictor, and HPTW interface.

13.1 BRANCH PREDICTION PRINCIPLES

Recall that the basic 5-stage pipeline from Section 7.2 predicts that branches are not taken and continues fetching subsequent instructions. In the Execute stage, it determines whether the branch should have been taken. If so, it flushes the next two instructions and updates the PC to the correct branch target address (BTA). So, the *misprediction penalty*, the number of cycles lost because of the misprediction, is 2 cycles. The flush increases the CPI (cycles per instruction) to 3 for taken branches. Branches



An ITIM is also called an IROM (instruction read-only memory) because it is read-only. When included in a system, the IROM replaces (or works alongside) the I\$ shown in Fig. 13.1.

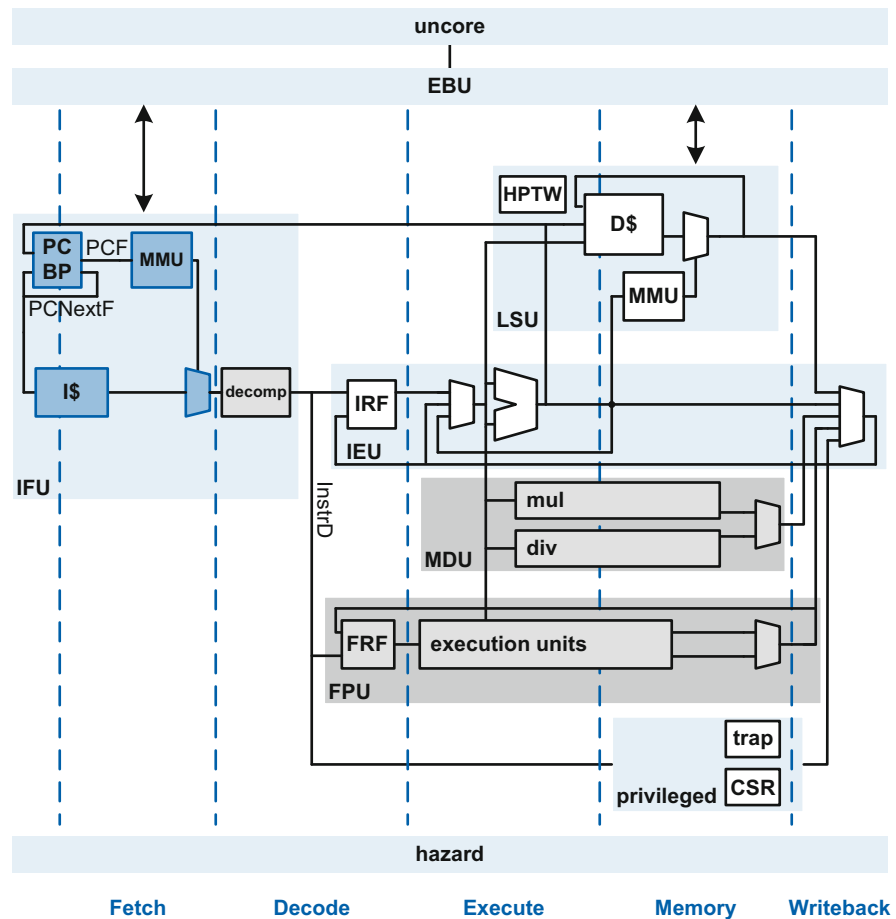


Fig. 13.1 Wally high-level block diagram emphasizing the IFU.

occur every 4-7 instructions in typical code, meaning that roughly 20% of executed instructions are branches.

Example 13.1 EFFECTS OF BRANCH MISPREDICTION ON CPI

In the CoreMark benchmark from [Section 7.2.2.5](#), 22.2% of executed instructions are branches, and 41.9% of them are taken. Optimistically, assume that nonbranch instructions have a CPI of 1. What is the average CPI?

Solution: A taken branch requires 3 cycles: 1 to execute and 2 to flush the Fetch and Decode stages. A not taken branch requires 1 cycle.

$$\text{CPI} = (1 - 0.168) \times 1 + 0.168 \times [3 \times 0.423 + 1 \times (1 - 0.423)] = 1.14$$
, which is 14% slower than the ideal CPI of 1.

Next, consider a 2-way superscalar processor (ideal CPI of 0.5), which predicts all branches as not taken and runs the same CoreMark code. A not-taken branch and other instructions require 0.5 cycles. A taken branch takes 3 cycles to execute, and it

flushes 5 instructions: 2 from the Fetch stage, 2 from Decode, and 1 from Execute. $CPI = (1 - 0.168) \times 0.5 + 0.168 \times [3 \times 0.423 + 0.5 \times (1 - 0.423)] = 0.68$, which is 36% slower than the ideal CPI of 0.5.

Branch misprediction becomes much more significant in wide superscalar processors with deeper pipelines. For example, the Pentium 4 has a 20-cycle branch misprediction penalty and dispatches up to 4 instructions per cycle, meaning a misprediction could flush up to 80 instructions [Hinton01]. Even a misprediction rate of 1% would have a significant impact on CPI.

Branch prediction is an essential feature for high-performance processors. The simplest form is *static branch prediction*, such as predicting that all backward branches will be taken. *Dynamic branch prediction* improves accuracy by using recent branch history. Simple dynamic branch prediction uses 1 or 2 bits to remember the history of specific branches. *Gshare*, a widely used predictor, incorporates the history of multiple recent branches [McFarling93]. Branch predictors guess whether a branch is taken and work in conjunction with a *branch target buffer (BTB)* that holds the BTA. Other control instructions such as function returns and other indirect jumps use special predictors tuned to their behavior. Deeply pipelined superscalar processors devote even more resources to advanced branch predictors to minimize misprediction.

Branch prediction takes place in the Fetch stage to predict which instruction the IFU should fetch next. A dynamic branch predictor performs three main operations: it determines if the instruction is a branch or jump, predicts whether the branch is taken, and keeps track of target addresses for branches and jumps. To do this, the dynamic branch predictor uses:

- ▶ A *branch history table (BHT)* to track recent branch outcomes and use them to predict whether the branch will be taken.
- ▶ A BTB to record recent branches or jumps and their target addresses. The target buffer is essentially a direct-mapped cache indexed by the PC that produces the target address for taken branches and jumps.
- ▶ A *return address stack (RAS)* to track recent return addresses because function returns are poorly tracked with an ordinary BTB.

Each of these components is discussed further in the remainder of this section.

Although these principles apply to any microarchitecture, this section illustrates branch predictors using a 5-stage pipeline with branch conditions computed in the Execute stage, as is the case with Wally. This section shows abstract diagrams to illustrate the principles, and the detailed Wally diagrams in 13.3 expand on them. Signal and unit names in this section are the same as in Wally for consistency. When discussing branch predictors, all control flow instructions, including jumps, are referred to, generally, as branches for brevity, except when the difference matters.

Direct jumps, such as branches and `jal`, use the immediate field to specify the target address relative to the PC. Indirect jumps, such as `jalr`, specify the *location* where the target address is held, for example, in a register. Direct jumps compute the same target address each time they are executed, but indirect jumps may not. For example, the `ret` instruction, which returns from a function call and is a pseudoinstruction for `jr ra`, is *indirect* because it specifies the target address in a register. So, the target address could be different for each execution of the same `jr`.

13.1.1 Static Branch Prediction

A static branch predictor relies on information known at compile time. The simplest static branch predictor guesses that a branch is not taken, as in Section 7.2. However, backward branches, such as those used in `for` and `while` loops, are taken about 90% of the time. Forward branches are taken about 50% of the time. A better static branch predictor guesses that backward branches are taken. The ARM 11 core used static branch prediction [ARM11].

13.1.2 Simple Dynamic Branch Prediction

Dynamic branch prediction captures past program behavior to guess the *direction* of the current branch, that is, whether it is taken or not. A *1-bit dynamic predictor* tracks whether a branch was taken last time and predicts to do the same next time.

Consider the nested loops in Code Example 13.1. `branch_j` at the end of the `j` loop (address 0x134: `blt s1, t1, forj`) is executed 10 times for each iteration of the `i` loop. That `blt` is taken on the first nine times and not taken on the 10th. A 1-bit predictor mostly guesses taken. On the tenth time, it is wrong, and it now records not taken. However, on the next iteration of the `i` loop, `branch_j` should be taken, but it is predicted not taken. Thus, the branch is mispredicted on the first and last iteration of the `j` loop. In this example, the prediction accuracy is 80% for `branch_j`. Similarly, the `i` loop’s first and last branches are also mispredicted, so that loop has a $(5 - 2)/5 = 60\%$ prediction accuracy.

When discussing branch prediction, direction refers to whether a branch is taken or not; it does not refer to the direction of the branch (forward or backward).

Code Example 13.1 NESTED LOOPS

C Code	Assembly Language
<pre>int i, j; for (i=0; i<5; i++) for (j=0; j<10; j++) dostuff</pre>	<pre># i in s0, j in s1 0x100: li t0, 5 # Bound on i loop 0x104: li t1, 10 # Bound on j loop 0x108: li s0, 0 # i = 0 0x10C: fori: li s1, 0 # j = 0 (start of i loop body) 0x110: forj: # dostuff (start of j loop body) ... 0x130: addi s1, s1, 1 # j++ 0x134: blt s1, t1, forj # branch_j: Repeat if j < 10 0x138: donej: addi s0, s0, 1 # i++ 0x13C: blt s0, t0, fori # branch_i: Repeat if i < 5 0x140: donei:</pre>

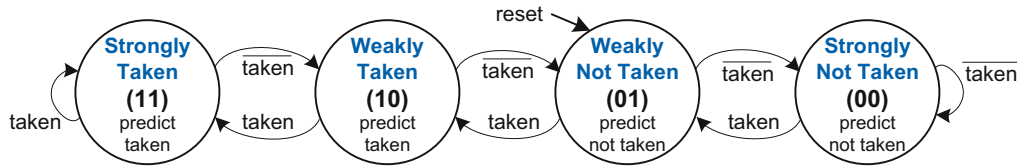


Fig. 13.2 2-bit branch predictor state transition diagram.

The 2-bit dynamic branch predictor (or 2-bit predictor) in Fig. 13.2 improves the accuracy by remembering more history and only changing the prediction after multiple incorrect guesses. The states are encoded such that the most significant bit indicates the prediction (1 for taken, 0 for not taken). During the j loop, the predictor for `branch_j` enters the Strongly Taken state, with an output of 11. When it mispredicts on the last time through the j loop, it moves to the Weakly Taken state (10). The next time it starts the j loop, it still predicts `branch_j` as taken. Thus, `branch_j` is only mispredicted the last time through the j loop. In this example, the prediction accuracy is 90% for `branch_j`.

The FSM's state bits are referred to as a *saturating counter*, with the counter saturating at 11 when a given branch is Strongly Taken and 00 when it is Strongly Not Taken.

The branch predictor stores the history information in a BHT (branch history table). The table has a finite number of entries and is typically indexed by the less significant b bits of the PC, typically excluding the two lsbs, which are 0 if all instructions are 32 bits. Fig. 13.3 shows a 2^b -entry BHT built from a dual-port synchronous RAM with one read and one write port for reading the predicted direction in the Fetch stage and writing the updated prediction in the Execute stage. Each 2-bit entry indicates the predictor state for any branch with the corresponding less significant address bits. Just before the Fetch stage, `PCNextF` indexes the BHT so that it outputs the predicted direction `BPDireF` just after the next clock edge, at the beginning of the Fetch stage. The predicted direction is pipelined to the Execute stage, where the true branch outcome is actually calculated. An XOR gate compares the predicted and actual direction, `PCSrcE`, which is 1 when the branch is taken. If the prediction is wrong, `BPDireWrongE` = 1, and the PC logic must flush the previous two instructions from the pipeline and fetch the correct instruction, as will be described in Section 13.1.4.

The branch predictor updates the BHT with the new predicted direction (`NewBPDire`) in the Execute stage, according to Fig. 13.2, after the IEU determines the true branch direction. Even though the branch predictor of Fig. 13.2 is drawn as an FSM, only its state is stored in the BHT. The 2-bit saturating counter logic updates the BHT state based on the current state (`BPDire`) and the direction of the branch (`PCSrcE`).

The 2-bit branch predictor is also called a *bimodal* predictor.

The BHT behaves similarly to a direct-mapped cache with no tags, which means conflicts could exist between branches at different program addresses. These conflicts reduce prediction accuracy but don't cause program failure because the consequence of a mispredicted branch is just to flush the incorrectly fetched instructions.

Throughout the branch predictor, predicted signals are prefixed with BP.

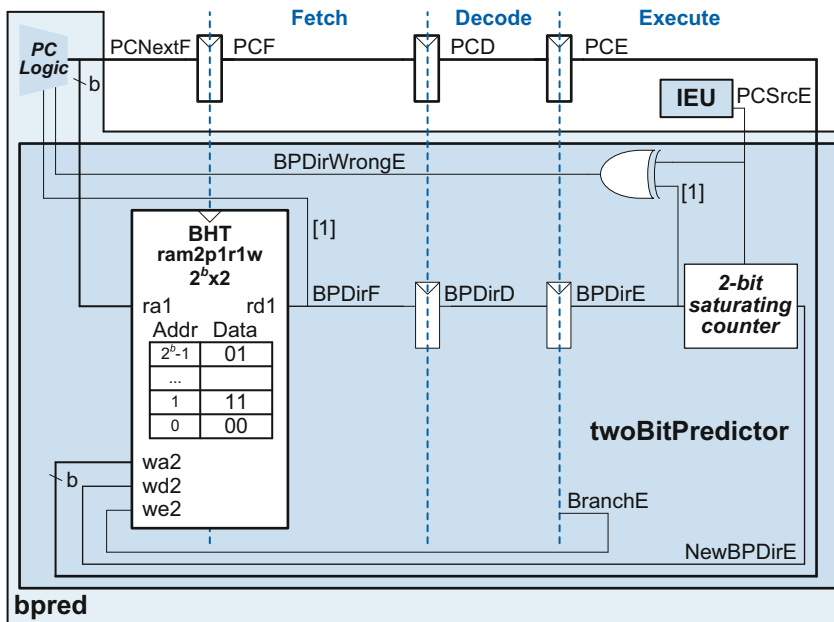


Fig. 13.3 BHT for 2-bit branch predictor.

When the prediction is wrong, $BPDirWrongE = 1$ triggers the hazard unit to flush the Fetch and Decode stages. The PC logic updates PC_{NextF} to the correct PC, as described in Section 13.3.3.5.

Example 13.2 BHT ENTRIES

Suppose Code Example 13.1 is run on the 2-bit branch predictor from Fig. 13.3 with $b = 4$. The lower two bits of the PC are always 0, so $PC[5:2]$ are used to index the BHT. At reset, the BHT entries are initialized to 01 (Weakly Not Taken). What are the values in the BHT after the program completes?

Solution: This first branch is at address $0x134 = 0b000100110100$. Bits 5:2 of this address are **1101**, so the branch maps to BHT entry 13. The next branch is at address $0x13C (0b000100111100)$ and maps to BHT entry 15. The other BHT entries are unused. Each branch is repeatedly taken, moving to the Strongly Taken state. However, on the last iteration of each loop, the branch is not taken, moving back to the Weakly Taken (10) state. Thus, entries 13 and 15 both contain 10, and the other entries remain at their reset value 01.

Fig. 13.4 shows the 2-bit branch predictor misprediction rate on SPEC89 benchmarks for a 4,096-entry BHT. The accuracy (misprediction rate) is highly dependent on the application, ranging from 82% to 99%.

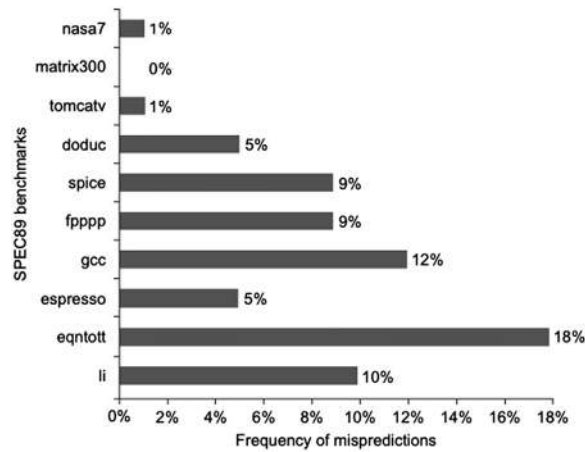


Fig. 13.4 2-bit branch predictor accuracy with 4,096 entries. (Figure courtesy of Elsevier [Hennessy19]. Reprinted with permission.)

Some complex programs, such as gcc, have relatively low accuracy, but scientific computing programs with highly repetitive loops (e.g., *nasa7*, *matrix300*, and *tomcatv*) are largely predictable and, thus, have higher accuracy. Increasing the BHT size to infinity improves results by less than 1%, so 4,096 entries is sufficient for many programs.

13.1.3 Gshare

The 2-bit predictor considers each branch independently. Careful program analysis shows that some branch outcomes are correlated with others. To improve accuracy, a *global history dynamic branch predictor* considers the history of the last several branches. The widely used *Gshare* predictor incorporates global history to achieve prediction accuracy greater than 96% with 4,096 entries [McFarling93].

Code Example 13.2 shows a snippet of the *eqntott* SPEC89 benchmark with a heavily used and difficult-to-predict branch. If *branch_aa* and *branch_bb* are both not taken (and hence make *aa* and *bb* = 0), then *branch_aabb* will be taken. Hence, *branch_aabb* partially depends on the outcome of previous branches. By separating the prediction of *branch_aabb* into four scenarios that depend on the outcomes of the previous two branches, the predictor can reduce the interference between the different scenarios and improve the prediction accuracy [Yeh91, Pan92].

A $4,096 \times 2$ -bit SRAM has an extreme aspect ratio, making it impractical to build on a chip. It is normally folded into a more square-like shape. For example, a 32x fold gives a 128×64 -bit SRAM, with the lower bits of the original address used as mux selects and bit write enables to read or write 2 of the 64 bits.

Code Example 13.2 `eqntott` CORRELATED BRANCHES

C Code	Assembly Language
	<code># aa in s0, bb in s1</code>
<code>if (aa == 2)</code>	<code>0x1000: li t0, 2 # 2 for comparison</code>
<code>aa = 0;</code>	<code>0x1004: bne s0, t0, skipaa # branch_aa: Skip if aa != 2</code>
<code>if (bb == 2)</code>	<code>0x1008: li s0, 0 # aa = 0</code>
<code>bb = 0;</code>	<code>0x100C skipaa: bne s1, t0, skipbb # branch_bb: Skip if bb != 2</code>
<code>if (aa != bb) {</code>	<code>0x1010: li s1, 0 # bb = 0</code>
<code>dostuff</code>	<code>0x1014 skipbb: beq s0, s1, skipaabb # branch_aabb: Skip if aa != bb</code>
<code>}</code>	<code>0x1018: # dostuff</code>
	<code>0x101C skipaabb:</code>

Gshare and global history predictors are often referred to as *correlating predictors* because they consider how branch outcomes are interrelated with other branches. They are also frequently called *2-level adaptive predictors* because they record branch history using two levels: the GHR and PHT.

Fig. 13.5 shows a block diagram of a Gshare branch predictor. Like a 2-bit dynamic predictor, Gshare uses a history table, a dual-port synchronous SRAM with 2-bit entries that indicate the prediction direction (whether it is taken or not) and the prediction strength as given in Fig. 13.2. Gshare calls this a *pattern history table* (PHT) instead of a BHT because it reflects patterns of branch history.

Gshare adds a *k*-bit *global history shift register* (GHR) that tracks the outcome of the last *k* branches. Instead of using the PC directly to index the PHT, Gshare indexes it using a *hash*, which is $PC_{NextF} \oplus GHR$. Therefore, sequences of branches with different histories each map to different locations and are separately predicted. Empirically, prediction accuracy is best when the number of table index bits *b* (extracted from the PC) equals the GHR length *k*.

Gshare updates the GHR and PHT after each branch outcome is known in the Execute stage. A given branch affects many entries of the PHT based on the GHR, so Gshare needs time to train all these entries before producing accurate predictions.

Like the 2-bit predictor, Gshare flushes the pipeline on misprediction and updates PC_{NextF} . The PC logic is described in Sections 13.1.4, 13.3.2, and 13.3.3.5.

Fig. 13.6 shows the improved branch prediction accuracy of Gshare over the simple 2-bit saturating counter predictor for Embench. Gshare can reasonably achieve branch prediction accuracy approaching 97% after sufficient warmup. Gshare generally outperforms 2-bit branch prediction because it captures the correlation among branches. However, for tiny history tables, 2-bit outperforms Gshare because it has fewer conflicts. Fig. 13.6 also shows theoretical accuracy from a branch predictor simulator (CModel) described in Section 21.3.2.1. Wally matches the theoretical results well, with discrepancies caused by the pipeline delay between making predictions and updating the tables.

Several processors use Gshare such as the Shakti RISC-V core (configurable size) and the Sun UltraSPARC-III (with 16K entries [Horel99]).



13.1.4 Branch Target Buffer

The branch predictor must guess not only whether a branch is taken but also the branch target address (BTA) where the branch would go if taken. Both guesses should be ready in time to fetch the next instruction. A BTB caches BTAs. Like the BHT or PHT, a BTB is indexed with the lower bits of PC. The BTB is accessed in parallel with the BHT/PHT. If the branch predictor guesses taken, the processor uses the BTA as the next PC; otherwise, it uses $PC + 4$. Like the BHT/PHT and unlike a conventional cache, the BTB omits tags, so multiple branches could alias to the same BTB entry. If the BTB incorrectly guesses the target address, the pipeline flushes the bad fetch after the correct target address is computed and updates the BTB with the corrected BTA. The BTB also caches the target address for jumps.

Fig. 13.7 shows a branch predictor with a 2-bit direction predictor and a BTB. For now, let us assume the hardware knows if the Fetch stage instruction is a branch or jump. Sections 13.3.3.2 and 13.3.3.3 explain how the processor predicts the *instruction class* (IC), which tells the branch

Because multiple instruction alias to the same BTB entry, the BTB may mispredict both the target address and instruction class. The data shows that this usually has a small effect on BTB accuracy, although certain benchmarks such as Embench's *nsichneu* experience notable performance loss because of this aliasing.

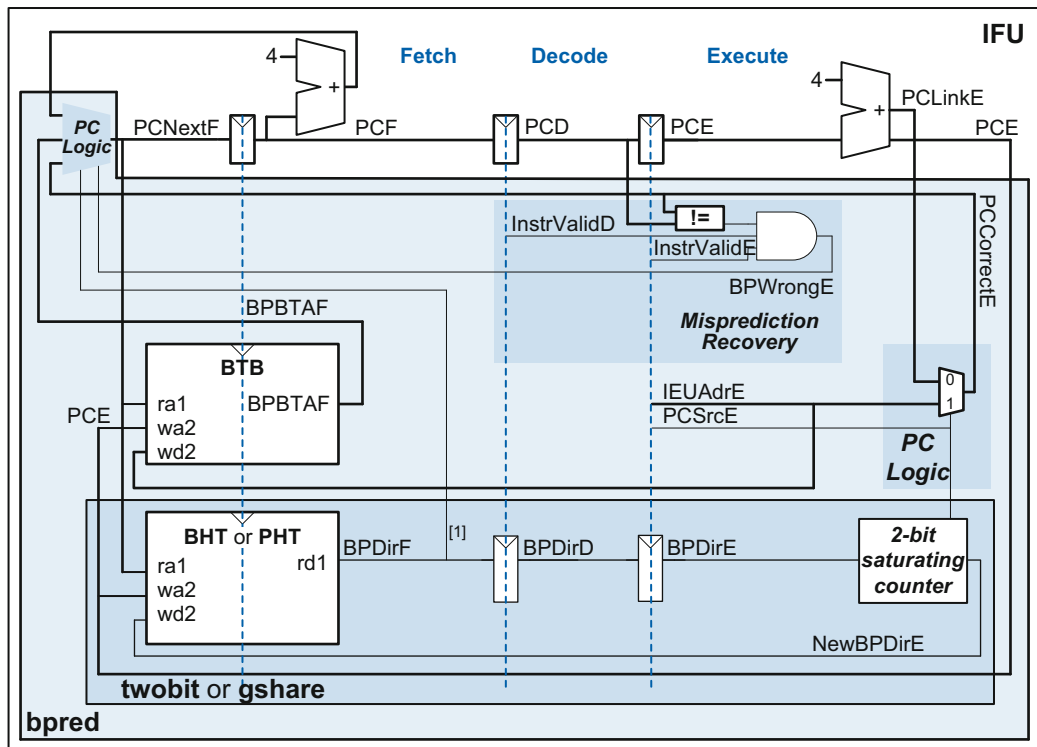


Fig. 13.7 Branch predictor with BTB.

predictor if the instruction is a branch, jump, function return, or jump and link. If the instruction is a branch and the direction predictor predicts taken or if the instruction is a jump, then the PC logic selects $PC_{NextF} = BPBTAF$ (branch predictor BTA from the BTB). However, if a branch is predicted not taken, then the PC logic selects the fall-through address, $PC_{NextF} = PCF + 4$. Like the BTH/PHT, the BTB's memory array is synchronous SRAM so PC_{NextF} sets up the read address before the rising clock edge at the start of the Fetch stage, and $BPBTAF$ is valid at the beginning of the Fetch stage.

If the Execute stage determines the next PC was mispredicted, it asserts $BPWrongE$ and PC logic selects $PC_{NextF} = PC_{CorrectE}$, the PC of the correct next instruction after the branch. If the Execute stage instruction is a taken branch or a jump, then $PC_{CorrectE}$ is the BTA, $IEUAdrE$. Otherwise, $PC_{CorrectE}$ is the fall through address $PCE + 4$. Unlike the branch direction predictors of Fig. 13.3 and Fig. 13.5, which checks if the branch predictor's direction matches, now the predictor needs to check if the PC of the next instruction (PCD) matches what the IEU computed ($PC_{CorrectE}$). If they mismatch and both instructions are valid ($InstrValidD = InstrValidE = 1$), the branch predictor fetched the wrong instruction and must assert $BPWrongE$ to flush the Decode and Execute stages.

Fig. 13.8 shows the BTB accuracy versus the number of BTB entries for Embench. Again, Wally's implementation matches the theoretical CModel closely. The working sets of Embench programs are typically small and comfortably fit inside both the I\$ and branch predictor. Even with just 256 entries, the BTB hits 99% of all jump and branch instructions after

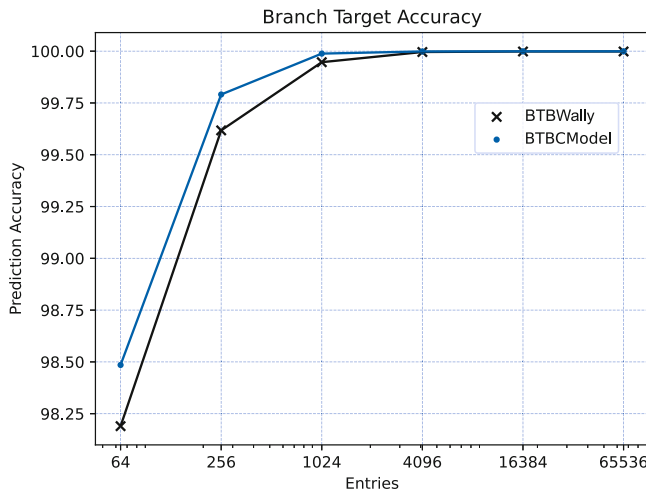


Fig. 13.8 BTB accuracy for Embench vs. number of entries.

a sufficient warmup time. All RISC-V branches and most jumps have *direct* branch/jump targets because they compute their target address relative to PC. A direct target never changes; therefore, the BTB readily achieves near 100% accuracy. The next section describes the most common type of *indirect* jumps, function returns.

Section 2.9 lists BHT and BTB sizes for several RISC-V processors. BTB sizes range from 4 to 256 entries in low-end processors. The smaller BTBs are typically associative and don't contain instruction class prediction.

13.1.5 Return Address Stack

The BTB works well for branches and `jal` because the destination is part of the instruction. However, function returns (`ret`) are difficult to predict because they have different return addresses depending on where the function was called. Because function calls occur in pairs with returns, a *return address stack* (RAS) reliably predicts the return address. A RAS is a small hardware stack that pushes the return address when calling a function and pops the address when returning.

Fig. 13.9 shows the block diagram of a RAS, which consists of a small memory and a pointer (`Ptr`) to the top of the stack. When the processor executes a function call, `jal ra, <dest>`, it computes the return address ($PCLinkE = PCE + 4$). It *pushes* this return address onto the stack by incrementing `Ptr` and then writing the return address to the RAS memory at `Ptr`. When the processor fetches a return instruction, it *pops* the return address off the stack by reading the RAS memory at `Ptr` and decrementing `Ptr`.

A RAS typically has 3 to 16 entries because the majority of programs have a limited call depth. However, when the RAS overflows, `Ptr` wraps

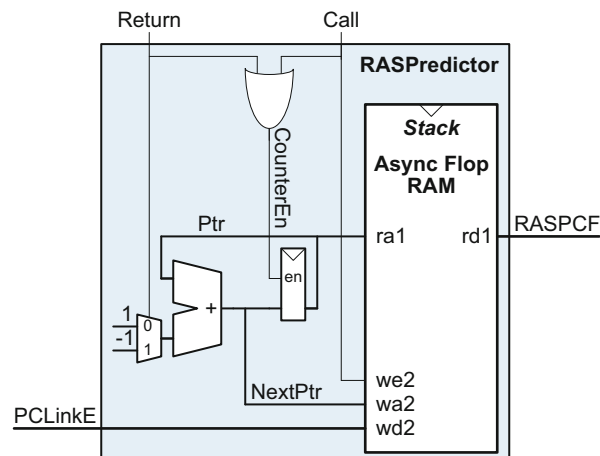


Fig. 13.9 RAS.

around, overwriting the oldest valid return address. In practice, this has a small effect on prediction accuracy because the recent pushes are still valid.

Fig. 13.10 shows a sequence of two function calls and returns using a RAS with 4 entries. In Fig. 13.10(a), *Ptr* is initialized to the bottom entry. In Fig. 13.10(b), a *jal* at 0x105c pushes its return address, 0x1060. In Fig. 13.10(c), the next *jal* also pushes its return address, 0x1144. In Fig. 13.10(d), a *ret* pops the stack and predicts $PC_{NextF} = 0x1144$. In Fig. 13.10(e), the last *ret* pops $PC_{NextF} = 0x1060$.

Fig. 13.11 integrates the RAS into the branch prediction hardware from Fig. 13.7. If the Fetch stage instruction is *ret*, the RAS pops the return address, and the branch predictor selects $BPPCF = RASPCF$. The IFU computes and pushes the return address $PCLinkE = PCE + 4$ onto the RAS during the Execute stage.

jal and *ret* instructions are usually paired together; however, buggy programs could execute *ret* without a preceding *jal*. Without a *jal* the RAS pops the stack as usual but will contain an invalid return address. The misprediction recovery logic of Fig. 13.11 detects the incorrect address and flushes the pipeline just like a branch misprediction. Additionally, program code could change the return address in *ra*, but the RAS won't be updated; this would cause *ret* to jump to (fetch) the wrong instruction. Again, the recovery logic detects the mistake and flushes the pipeline. In general, the recovery logic handles all control flow mispredictions regardless of the source of the misprediction: BTB, RAS, and so on. If the calculated BTA ($PC_{CorrectE}$) matches what was fetched (PCD), then the branch predictor fetched the correct instruction.

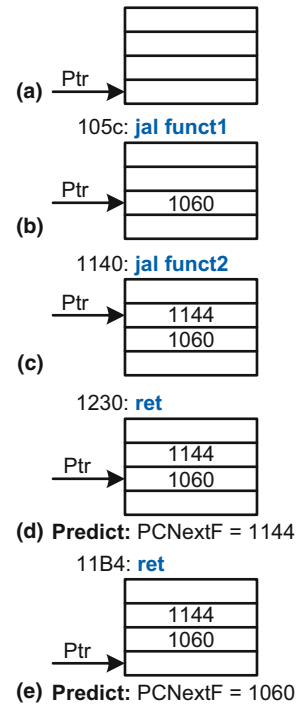


Fig. 13.10 RAS example.

13.1.6 TAGE

High-performance processors need even better prediction accuracy than Gshare can provide. A *tournament branch predictor* includes multiple sub-predictors and chooses among them based on which proves most accurate for a given branch.

The *partially TAgged GEometric global history* tournament branch predictor (TAGE) achieves better than 99% accuracy with sufficiently large memories [Seznec06, Hennessy19]. TAGE may be too complex to complete in one cycle. Therefore, it is sometimes used as a second-level predictor to correct a faster but less accurate first-level predictor. For example, recent AMD Ryzen CPUs use TAGE as a second-level predictor [Suggs20].

TAGE uses multiple Gshare-like subpredictors, each with a different-length GHR. The lengths are typically 0, 2, 4, 8, 16, and so on, forming a geometric sequence. Shorter subpredictors warm up faster and avoid irrelevant branch history, while the longer subpredictors capture the behavior of complex branch sequences.



The IFU test plan stresses the branch predictor and instruction cache to verify both correctness and performance.

Wally presently uses the riscv-arch-test regression tests to verify instruction operation, including branches and jumps. When the IFU and LSU both include caches, the Zifence riscv-arch-test checks the correctness of `fence.i`, which synchronizes LSU updates to instruction memory; the test does this by first storing a new instruction to instruction memory, which will be held in the data cache, then performing a `fence.i`, and finally fetching the new instruction.

The CoreMark and Embench benchmarks are small enough to fit entirely in the cache, so they do not meaningfully stress the memory system. [Section 21.3.1](#) describes instruction cache performance verification.

Section 21.3.2 describes branch predictor performance verification. Figs. 13.6 and 13.8 show good agreement between Wally and a theoretical branch predictor.

13.3 WALLY IMPLEMENTATION

Fig. 13.12 shows the fully featured (rv64gc) Wally IFU with the PC logic and optional I\$, IMMU, AHB cache interface (*ahbcacheinterface*), and branch predictor (*bpred*). The grayed-out spill logic for compressed instructions is discussed in Chapter 14. The IFU and LSU serve similar functions and thus have similar structure. Compared to the LSU in Fig. 12.2, the IFU drops the subword and endian swapping logic because instructions are always 32 bits and little-endian. The read-only I\$ omits write circuitry. The IFU adds PC logic, including an optional branch predictor. The IMMU relies on the LSU's HPTW to service ITLB misses. Exceptions generated during the Fetch stage are pipelined to generate *TrapM* during the Memory stage.

The IFU fetches an instruction every cycle unless *StallF* is asserted. The PMA checker decodes whether the instruction comes from cacheable memory or the IROM inside the IFU. Table A.1 summarizes typical IFU memory configurations.

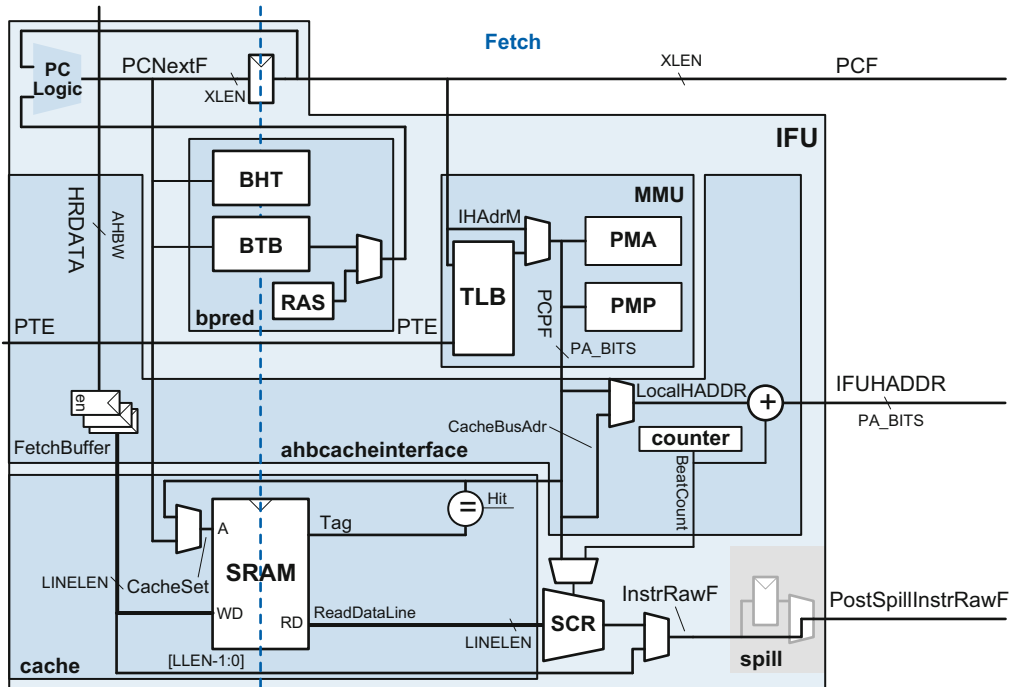


Fig. 13.12 IFU block diagram.

- ▶ Embedded configuration (rv32e): external memory only
- ▶ Simple CPU configurations (rv{32/64}i): IROM only
- ▶ Microcontroller configuration (rv32imc): IROM and external memory
- ▶ Application processor (rv{32/64}gc): I\$ and external memory

13.3.1 Instruction Cache

Chapter 10 described the generic level 1 cache used by both the LSU and IFU. The IFU has a cache when `ICACHE_SUPPORTED = 1`, and it instantiates the `cache` module with the parameter `READ_ONLY_CACHE = 1`. Fig. 13.13 shows a block diagram of the I\$. Compared to the generic cache of Fig. 10.24, it omits the write logic, dirty bits, and writeback and flush operations. Eliminating the writeback and flush logic also simplifies the bus and address selection logic. The cache FSM (`cachefsm`) also eliminates the Writeback, Flush, and FlushWriteback states.

13.3.1.1 Invalidation

During a `fence.i` instruction, the IFU invalidates the I\$ to synchronize the instruction and data caches. To do so, it asserts `InvalidateICacheM`, which drives `InvalidateCache` in the I\$ to clear the valid bits in all cache lines.

13.3.2 PC Logic

Fig. 13.14 shows the PC logic. Compared to Fig. 8.17, the new branch predictor muxes are highlighted in darker blue. When the branch predictor guesses a taken branch, `BPPCSrcF = 1` and PC takes its value from the BTA `BPPCF`, which is generated by the BTB or RAS (Fig. 13.15). Otherwise, the PC is incremented by 4.

In the Execute stage, the IEU determines whether the branch is taken and where it goes. If the branch predictor was correct, `BPWrongE = 0` and the IFU continues fetching instructions normally. Otherwise, `BPWrongE = 1` to choose the correct address from `PCCorrectE` and flush the misfetched instructions. `PCSrcE` selects the correct address from either the calculated BTA (`IEUAdre`), for taken branches, or `PCE + 4` (`PCLinkE`).

Fences and CSR writes change the architectural state, so when these instructions reach the Memory stage, they flush and reissue the subsequent instructions. The PC must be reloaded with the address of the first instruction to reissue, `PCM + 4`. This next PC is normally `PCE`. However, the instructions in the Execute and Decode stages may be flushed from a previous branch misprediction. When `BPWrongM = 1`, the PC logic selects `PCF` because it is the address of the next valid instruction.

`fence.i` was introduced in Section 10.4.2.6.

Fences and CSR writes can change the behavior of subsequent instructions, so any in-flight instructions must be restarted. For example, when `satp` or the page table changes and `sfence.vma` occurs, the subsequent instructions must be reissued using the new addressing.

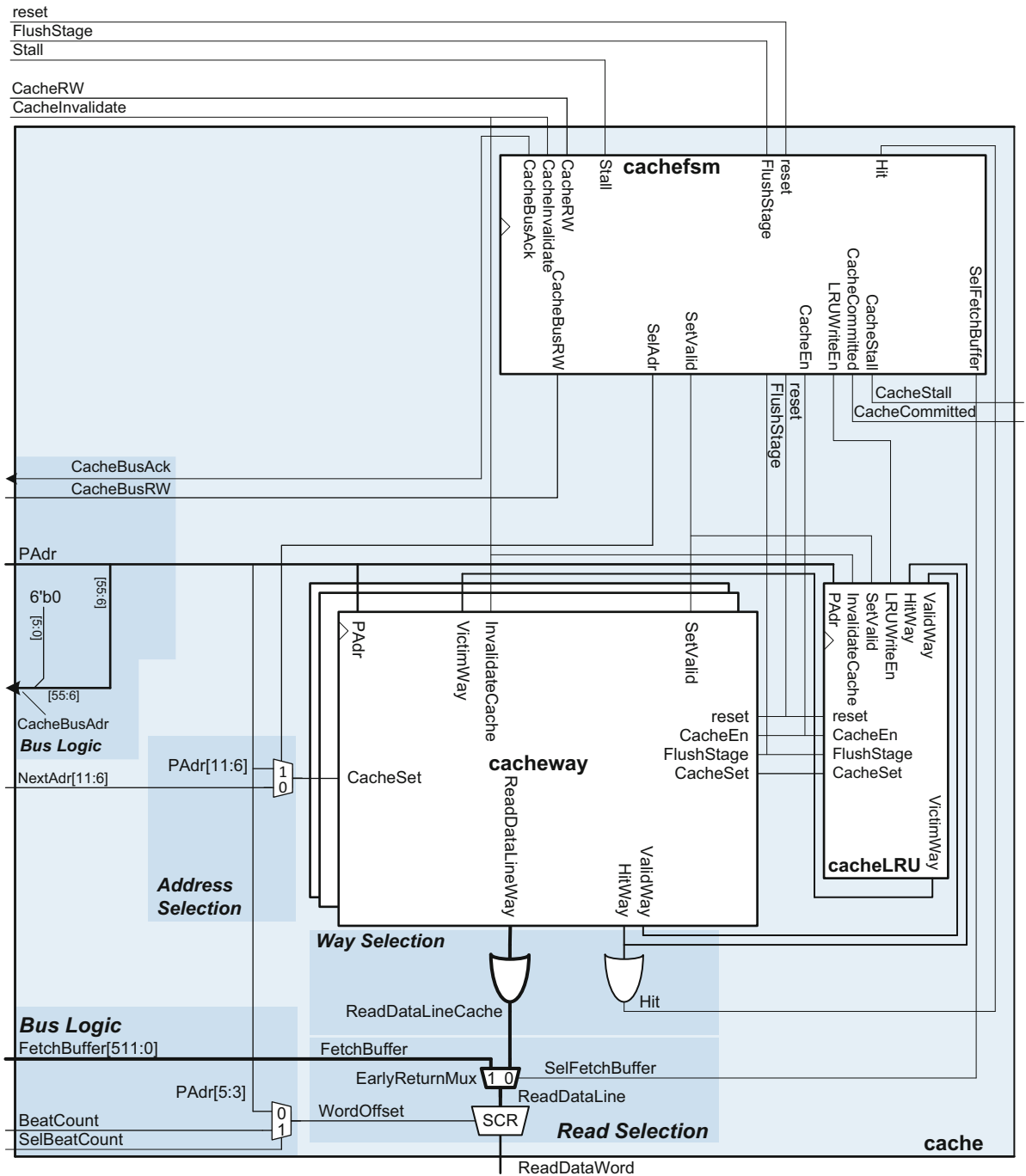


Fig. 13.13 I\$ block diagram.



At the time of writing, Wally supports additional branch predictors not discussed in the text. For more information see [Yeh91].

Fig. 13.15 shows a top-level diagram of `bpred`. It is similar to Fig. 13.11 but includes more details of the top level while abstracting

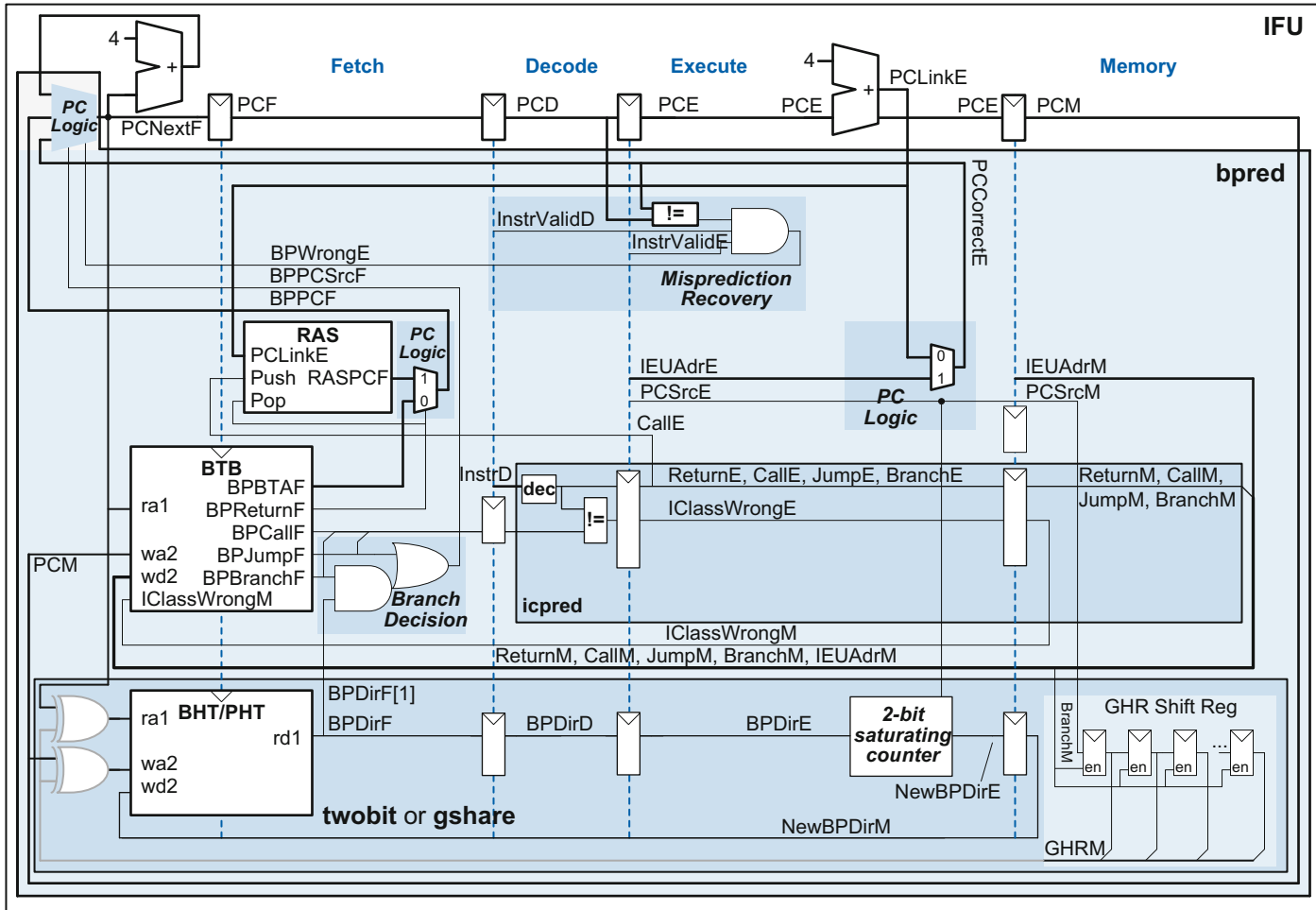


Fig. 13.15 Branch predictor top-level diagram (bpred).

some of the details of the branch direction and class predictors and PC logic. In addition to predicting the BTA, the BTB now also predicts the IC with the aid of the IC prediction module `icpred`, which determines the type of instruction: branch, jump, return, call, or noncontrol flow instruction (non-CFI). By predicting IC, the next PC (`PCNextF`) can be calculated ($PCF + 4$ for non-CFI instructions) or predicted (for branch/jump instructions) before the instruction is actually decoded in the Decode stage. The IC prediction module (`icpred`) keeps track of the actual and predicted IC and, in the Decode stage, detects if they don't match (`ICWrongD`). Mispredicted IC typically causes `bpred` to predict the wrong `PCNextF`, which leads to the same 2-cycle penalty as other branch mispredictions. However, even if the IC was predicted incorrectly, the behavior could be the same, for example a nontaken branch and an `addi` both fetch the next instruction from $PC + 4$. So, there is no need to flush the pipeline. The only side effect is that the performance counters tracking IC and branch direction mispredictions increment.

`bpred` updates the predictors (RAS, BTB, and direction predictor: BHT/PHT) at the end of the Memory stage rather than Execute. The PC logic details are shown in Fig. 13.14; note that only part of the PC logic is in `bpred`, and the rest is elsewhere in the IFU and privileged unit. The branch predictor may use the 2-bit or Gshare algorithm; Gshare is shown in gray and requires a GHR and XOR gates. `bpred`'s submodules, discussed in the next sections, are shown in Figs. 13.17 (Gshare PHT), 13.18 (BTB), 13.19 (`icpred`), and 13.20 (RAS). `PCSrc` and `IEUAdr` come from the IEU.

At the beginning of the Fetch stage, the BTB outputs the predicted target address `BPBTAF` and the IC: `BPReturnF`, `BPCallF`, `BPJumpF`, and `BPBranchF`. None of these being high indicates a non-CFI. If a return instruction is predicted (`BPReturnF = 1`), the PC logic chooses `RASPCF` as the predicted PC (`BPPCF`). Otherwise, `BPPCF = BPBTAF`, the predicted BTA. If the Branch Decision logic determines that the predicted IC is a jump or a taken branch, $PCNextF = BPPCF$.

Branch predictors are tricky because many types of bugs only result in lower prediction accuracy rather than incorrect processor behavior. In particular, a good predictor must take care to only update its predictions when valid branches commit. For example, flushed branches should not update predictions.

13.3.3.1 Gshare Implementation

Section 13.1.3 introduced the principles of Gshare. The ideal Gshare algorithm predicts, executes, and updates the predictor one instruction at a time. However, a pipelined implementation may have multiple branches in-flight between prediction, execution, and when the predictor updates. Additionally, pipeline flushes should roll back updates to the GHR. Wally handles these complications in a similar manner as *read after write* (RAW)

IEU hazards; it delays GHR and PHT updates until the Memory stage and uses forwarding to send GHR updates (from later stages) back to the Fetch stage.

A simple implementation (`BPRED_TYPE = `BP_GSHARE_BASIC`) avoids flushes by updating the GHR and PHT during the Memory stage but forgoes forwarding. A full implementation (`BPRED_TYPE = `BP_GSHARE`) adds logic to forward the GHR from a later pipeline stage to the Fetch stage. This section begins by explaining the simple implementation and then adds forwarding.

Flushed branch instructions should not update the GHR and PHT. Consider a CSR write followed by a branch. When the CSR write gets to the Memory stage, the branch will be in the Execute stage. The CSR write flushes the branch so it should not modify the GHR or PHT. By delaying GHR shifts until the Memory stage, the GHR does not require any repair.

Fig. 13.16 shows this simple implementation. It resembles Fig. 13.5 but moves the GHR and PHT updates to the end of the Memory stage.

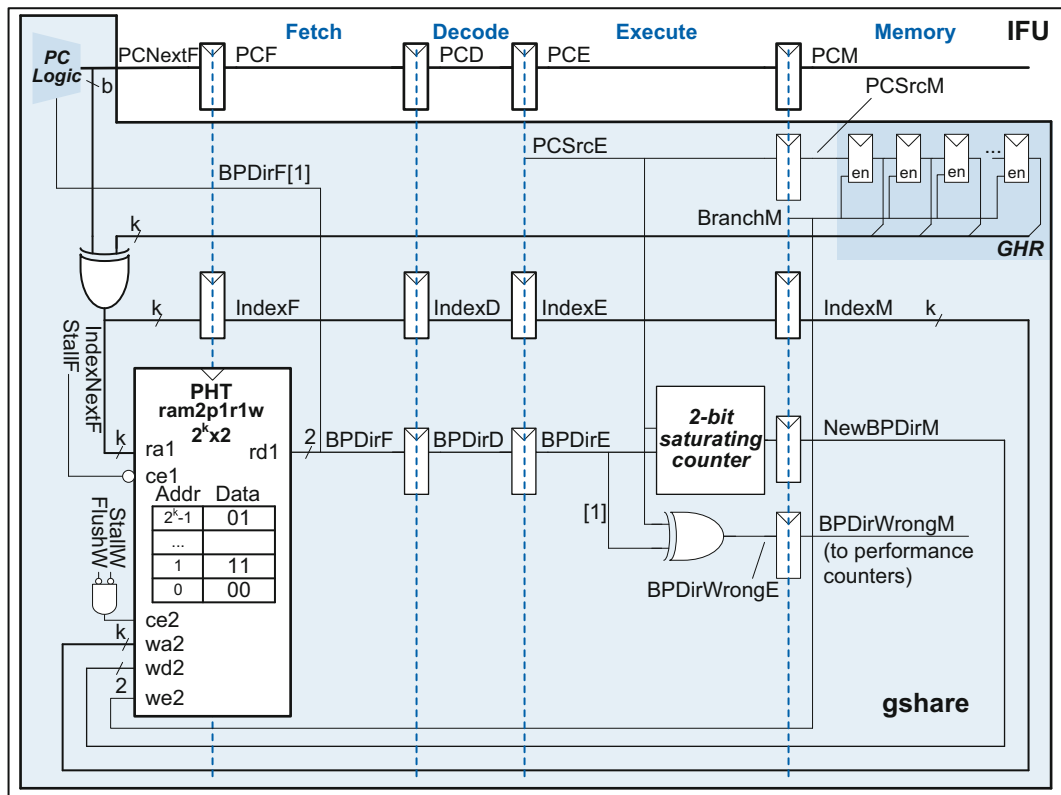


Fig. 13.16 Simple Gshare implementation (`BPRED_TYPE = BP_GSHARE_SIMPLE`).

Recall that the GHR is a shift register that tracks the outcomes of the last k branches. It is only updated on valid branches ($\text{BranchM} = 1$, $\text{FlushW} = 0$, and $\text{StallW} = 0$). The update shifts in PCSrcM , the outcome of the current branch.

The PHT is accessed in the Fetch stage using IndexNextF , which is an XOR hash of PCNextF and the GHR. The same entry of the PHT is subsequently updated at the end of the Memory stage using IndexM , a pipelined version of the entry. The new direction entry NewBPDire is calculated by the 2-bit saturating counter.

The branch direction misprediction rate is measured by the performance counters from [Section 8.4.3.1](#) as $(\text{branch prediction wrong direction})/(\text{total branches})$.

A consequence of updating the GHR at the end of the Memory stage is that the GHR lags four cycles behind the PC. So, the GHR fails to account for in-flight branches, thus degrading prediction accuracy. The full Gshare implementation of [Fig. 13.17](#) adds forwarding logic, shown in blue, to incorporate in-flight branches into the prediction by forwarding updated versions of both the GHR and the predicted direction to the Fetch stage. We focus on GHR forwarding first, then predicted direction.

The chain of $\text{GHR}\{\text{E/D/F/NextF}\}$ muxes at the top of the forwarding logic incorporates each stage's branch outcomes into the GHR seen by the Fetch stage. If a stage has a branch instruction in it, the GHR is shifted right by 1, and that stage's branch direction is shifted in. All stages except Fetch use the calculated branch ($\text{Branch}\{\text{D/E/M}\}$) to determine whether to shift the GHR. Fetch must use the predicted value (BPBranchF). Execute and Memory use calculated branch directions ($\text{PCSrc}\{\text{E/M}\}$) to shift into the GHR; Fetch and Decode have to use the predicted values ($\text{BPDire}\{\text{F/D}\}[\text{1}]$). As a result of forwarding, GHRNextF includes the outcomes of all recent branches, including in-flight branches. Gshare XORs PCFNextF with GHRNextF , instead of the stale GHRM , to form PHTIndexNextF for reading the PHT in the Fetch stage.

The remaining Gshare forwarding logic passes along the predicted direction of a branch in a later stage to the Fetch stage if their indexes match. If an index match occurs with an in-flight branch, the BPDireF mux chooses the forwarded predicted direction (FwdNewBPDireF) instead of the stale prediction read from the PHT. The muxes forward the predicted direction from the earliest stage that matches because it has the most up-to-date prediction.

In summary, the forwarding logic ensures that the Fetch stage sees all uncommitted branch outcomes not yet reflected in the GHR and PHT.

This forwarding logic is analogous to the RAW data forwarding in [Section 7.2.2.3](#) when an instruction's source operand depends on the destination of another but the destination is not yet written to the register file.

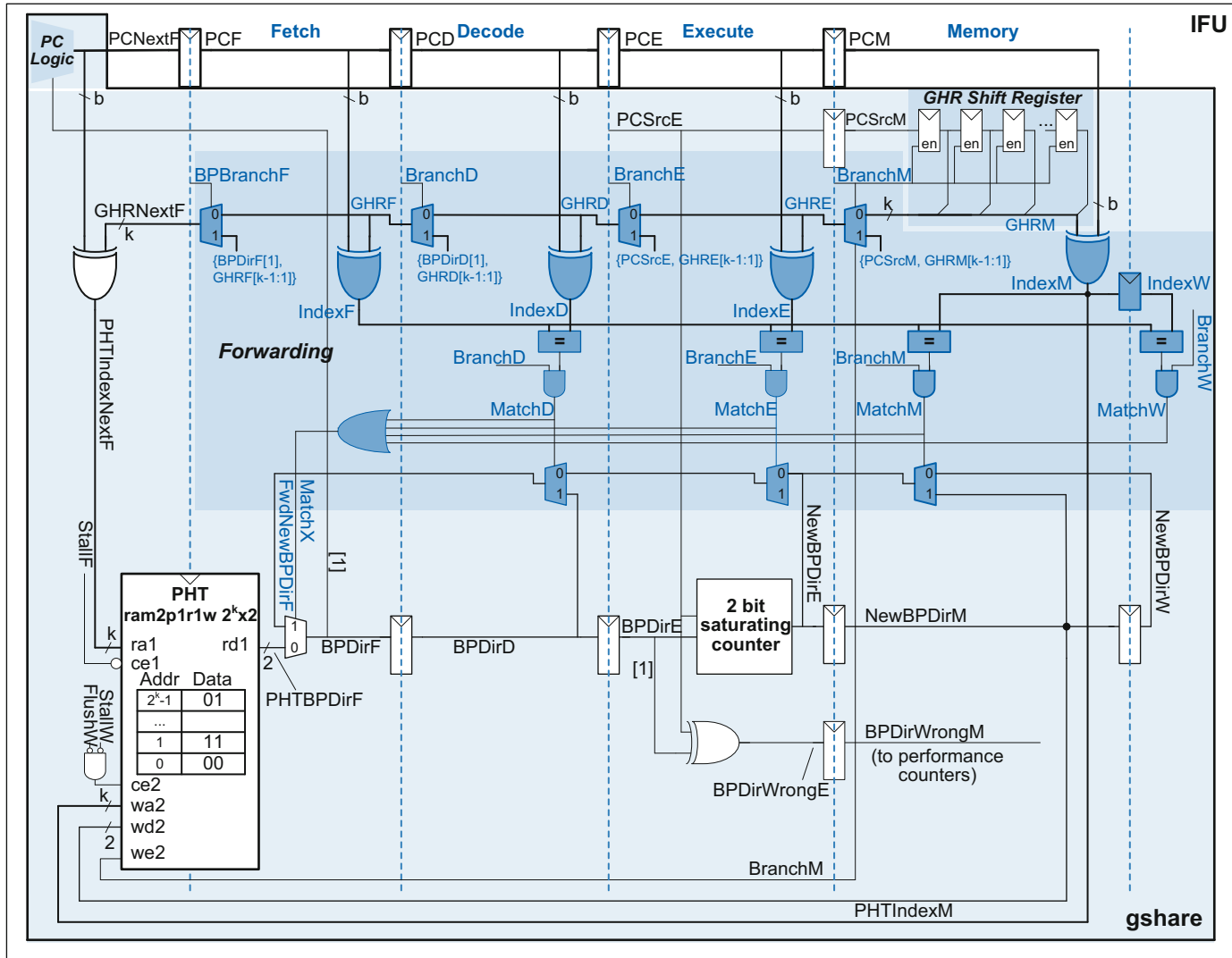


Fig. 13.17 gshare with forwarding.

Example 13.3 GHR CALCULATION

State the value of GHRNextF in terms of GHRM for the following three cases: when no stages contain branches, when only the Memory stage holds a branch and it is taken, and when all stages hold branches and only Execute isn't taken (or predicted taken).

Solution: If no stages hold branches, $\text{GHRNextF} = \text{GHRM}$, the same as without forwarding. For the other two cases, $\text{GHRNextF} = \{1, \text{GHRM}[k-1:1]\}$ and $\text{GHRNextF} = \{1, 1, 0, 1, \text{GHRM}[k-4:4]\}$.

Example 13.4 BRANCH PREDICTION USING FORWARDING

What is the predicted direction (BPDirF) in the Fetch stage if a branch is in the Execute stage ($\text{BranchF} = 1$) and matches the Fetch stage index ($\text{IndexE} = \text{IndexF}$), with no other stage's index matching?

Solution: $\text{MatchE} = (\text{IndexE} == \text{IndexF}) \ \& \ \text{BranchE} = 1$. So, $\text{MatchX} = 1$ and the predicted direction from the Execute stage is forwarded to the Fetch stage: $\text{BPDirF} = \text{FwdNewBPDirF} = \text{NewBPDirE}$.

When compressed instructions are not supported, instructions are word-aligned so $\text{PC}[1:0] = 00$. Hence, the BTB is indexed by $\text{PC}[b+1:2]$. When compressed instructions are supported, PC might be a multiple of 2, so the BTB could be indexed by $\text{PC}[b:1]$. However, this trades an often useless bit (bit 1 is often 0) for a useful bit (bit $b+1$). Using $\{\text{PC}[b+1] \oplus \text{PC}[1], \text{PC}[b:2]\}$ to index the BTB empirically performs slightly better because $\text{PC}[1]$ is disproportionately 0 (i.e., more uncompressed instructions exist), which enables use of $\text{PC}[b+1]$ as a significant bit.

The BTB contains both the BTA and IC because a single large memory is denser than two smaller memories.

13.3.3.2 BTB Implementation

The Wally BTB uses the same 2-port memory as in Fig. 13.7. Like the PHT and GHR, the BTB is updated in the Memory stage to avoid writing flushed branches to it, so it also uses forwarding to account for in-flight branches and jumps.

Fig. 13.18 shows the detailed BTB implementation. The BTB has 2^b entries, each comprising an XLEN-sized BTA and 4 IC bits that guess if the instruction at the current address is a branch, jump, function call, or return. The next section explains how the BTB learns the IC. The BTB thus uses a $2^b \times (\text{XLEN} + 4)$ -bit dual-port RAM. Read addresses come from the lower bits of PCNextF and write addresses come from the lower bits of PCM .

On the clock edge at the start of the Fetch stage, the RAM reads TableBTBPredF , which gives the BTA and IC prediction. The RAM updates on the rising clock edge at the end of the Memory stage, when the instruction commits, and the true target and class are known.

When an in-flight instruction's address matches PCF , the address of the fetched instruction, the BTB forwards the in-flight instruction's class and BTA to the Fetch stage. In any later stage, the true IC can be forwarded ($\text{ICClass}\{\text{D/E/M/W}\}$) because it is determined in the Decode stage by the icpred module. However, the BTA is not computed until the Execute stage; therefore, the Decode stage must forward BPBTAD , its predicted BTA. Forwards from the Execute, Memory, and Writeback stages use the true BTA, $\text{IEUAdr}\{\text{E/M/W}\}$. When multiple instructions' PCs match PCF , the earliest pipeline stage has priority as it contains the most recent BTA.

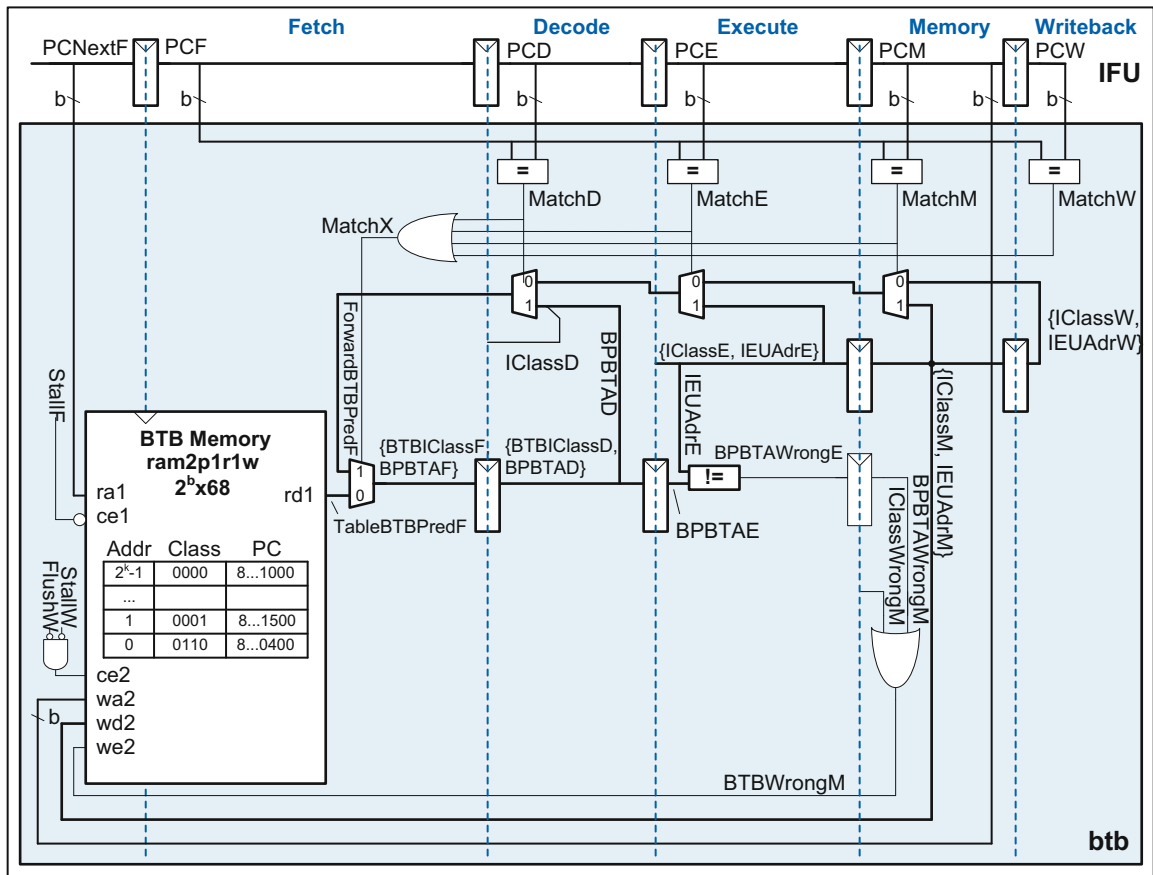


Fig. 13.18 BTB implementation with pipeline forwarding.

The BTB target misprediction rate is measured with the performance counters from [Section 8.4.3.1](#) as the ratio of (branch/jump prediction wrong target address)/(total branches + total jumps).

13.3.3.3 Instruction Class Prediction

Fig. 13.19 shows the IC prediction module, `icpred`, which tracks the predicted and actual IC and, in the Decode stage, determines if they match. Because the actual IC is not available until the Decode stage, the BTB provides an IC prediction `BTBIClassF` in addition to the predicted BTA. `BTBIClassF` is separated into individual signals and becomes `BPBranchF`, `BPJumpF`, `BPCallF`, and `BPReturnF`. In the Decode stage, `icpred` decodes the actual IC and asserts `IClassWrongD` if it doesn't match the predicted class.

The BTB may guess the wrong IC because it is smaller than the working set of the program or because multiple instructions conflict for the same location in the BTB.

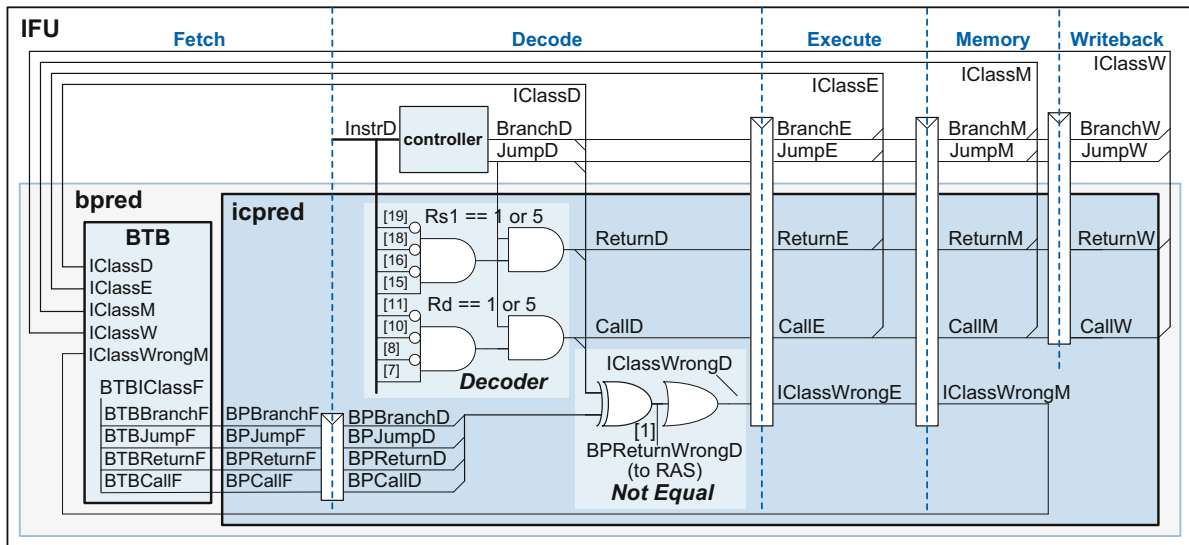


Fig. 13.19 IC prediction (icpred).

Saving and restoring registers on the stack is common for most functions. Waterman envisions reducing code size by sharing this common register save/restore code, which he refers to as *millicode*, across function calls [Waterman16, p. 66]. He suggests invoking this shared millicode with a `jal t0, <save/restore>`, to avoid overwriting `ra`. For this reason, `CallD` asserts for both `jal ra` and `jal t0`.

`JumpD` asserts for both `jalr` (including function returns) and `jal` instructions.

The rest of `icpred` pipelines the correct IC to the Memory stage, where it updates the BTB if IC was wrong. `icpred` also lets the RAS know if the instruction is a return (`ReturnD`) and if it was incorrectly predicted (`BPReturnWrongD`) so the RAS can undo popping its stack.

`icpred` uses `BranchD` and `JumpD` from the IEU. It recognizes certain `jal` instructions as function calls (`CallD`) and certain `jalr` instructions as function returns (`ReturnD`). Most function calls and returns are recognized because they use `x1 (ra)` as the destination (for `jal`) or source (for `jalr`). However, `icpred` also recognizes calls and returns that use `x5 (t0)`. `icpred` checks the `rd` and `rs1` fields in the Decode stage to detect function calls and returns. `CallD` asserts for `jal {ra/t0}`. `ReturnD` asserts for `jalr x0, {ra/t0}`.

The BTB only updates when `IClassWrongM` is asserted or the BTA was wrong, saving power when the prediction is correct. The performance counters also use `IClassWrongM` to measure the number of IC mispredictions. Similarly, the performance counters use `BranchM`, `JumpM`, and `ReturnM` to track the total number of branches, jumps, and returns. As with all predictors in `bpred`, any misprediction only impacts performance by requiring a pipeline flush of the Fetch and Decode stages.

The IC misprediction rate is measured with the performance counters from Section 8.4.3.1 as: (IC predictor wrong)/(instructions retired).

13.3.3.4 RAS Implementation

Wally's RAS implementation consists of two main parts: a small asynchronous flip-flop memory to record return addresses and a `Ptr` register

that indicates the top of stack. The memory is organized as a circular buffer with `RAS_SIZE` entries. Each time a call executes, the RAS pushes the return address by writing it to the memory at the top of its stack and then incrementing its stack pointer, `Ptr`. When the branch predictor predicts a return instruction, the RAS provides the return address by popping the stack (reading the memory at the top of the stack and decrementing `Ptr`). Pipeline flushes and IC mispredictions repair the RAS by undoing pushes or pops.

Fig. 13.20 shows the complete RAS implementation. The stack memory and up/down counter to push or pop `Ptr` are on the bottom right. The combinational logic converts calls, returns, pipeline flushes, and IC mispredictions into pushes and pops. When a call executes (`CallE`) and is not flushed or stalled, it pushes the return address, `PCLinKE`, onto the stack and increments `Ptr`. When a return is predicted in the Fetch stage (`BPReturnF`), it pops the return address, `RASPCF`, off the stack and decrements `Ptr`.

If the hazard unit flushes a return from either the Decode or Execute stage (`FlushedRetDE`), the RAS must undo the pop by incrementing `Ptr`. If the class predictor incorrectly predicted a return (`WrongPredRetD` & `ReturnD`), the RAS also increments `Ptr` to undo the faulty pop. Conversely, if the class predictor failed to predict a return (`BPReturnWrongD` & `ReturnD`), the RAS decrements `Ptr` to catch up.

The `bpred` module disables IC prediction when `INSTR_CLASS_PRED = 0`. In this configuration, the IC is determined by partially decoding `Instr` in the Fetch stage. Although this lengthens the Fetch stage, it currently doesn't affect cycle time because the Fetch stage is not in the critical path. Testing with class prediction enabled and disabled shows that the class predictor has negligible impact on branch prediction accuracy and thus does not have major performance bugs.

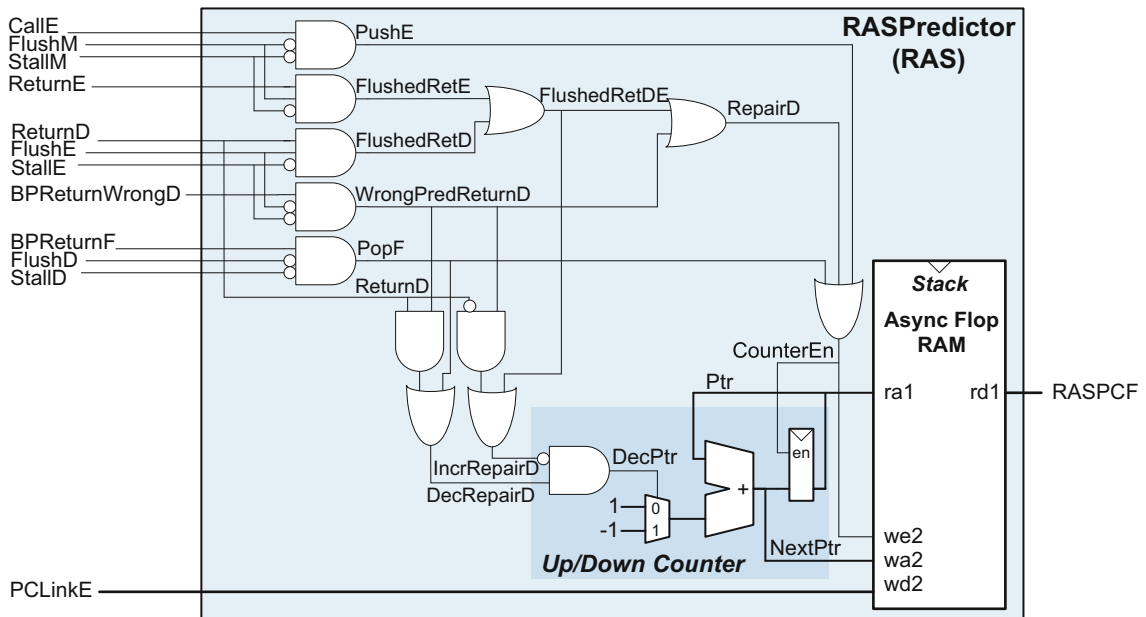


Fig. 13.20 RAS with repair.

Because the RAS uses asynchronous flip-flop memory, reads take place combinationally rather than requiring the address to set up before the clock edge. The memory has separate read and write ports so that the RAS can simultaneously push (write) and pop (read).

The RAS performance counter tracks the total number of return instructions committed and the number of RAS mispredictions. The “Return address stack wrong address” performance counter increments based on $RASPredPCWrongE = (RASPC \neq IEUAdrE) \& ReturnE$. The RAS misprediction rate is measured as $(RAS \text{ wrong address}) / (\text{total returns})$ using the performance counters from [Section 8.4.3.1](#).

13.3.3.5 Branch Misprediction Recovery

bpred can mispredict direction, target address, and IC, which leads to a pipeline flush if the PC of the predicted instruction is wrong, regardless of the type of misprediction. The branch predictor reports the misprediction to the hazard unit, which flushes the Decode and Execute stages. If the predicted PC is correct but the other predictors are wrong, the branch predictor still updates its state with the correct target and IC and records the misprediction in the optional performance counters.

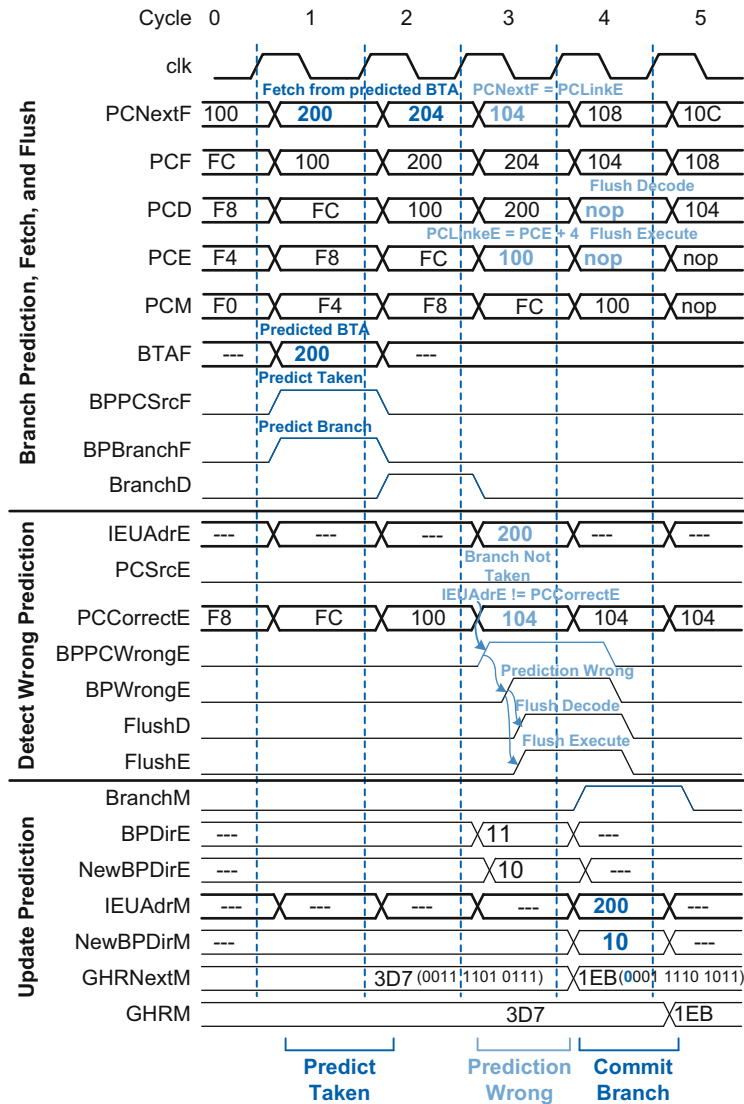
[Table 13.1](#) summarizes the various branch predictor outcomes. In the Execute stage, the PC logic computes $PCCorrectE$, which is $PCE + 4$ if the branch is not taken or the BTA, $IEUAdrE$, if the branch is taken. If PCD matches $PCCorrectE$, the correct instruction was fetched, and the program counter continues incrementing normally. Otherwise, the IFU fetched the wrong instructions. It recovers by flushing the next two instructions and setting $PCNextF = PCCorrectE$.

[Fig. 13.21](#) shows a timing diagram for a branch at 0x100 incorrectly predicted as taken with a BTA of 0x200. The processor fetches two instructions, from addresses 0x200 and 0x204, before branch misprediction is discovered in the Execute stage.

- ▶ **In cycle 1**, the BTB predicts the instruction is a branch ($BPBranchF = 1$) with $BTA = 0x200$. gshare predicts the branch is taken ($BPPCSrcF = 1$) and sets $PCNextF = BTA = 0x200$.
- ▶ **In cycle 2**, the IEU decodes the instruction and detects a branch ($BranchD = 1$).
- ▶ **In cycle 3**, the IEU executes the branch comparison and finds it is not taken ($PCSrcE = 0$). Therefore, the next instruction should be the fall through address $PCLinkE = PCE + 4 = 0x104$. bpred compares $PCD = 0x200$ to the expected address $PCCorrectE$. Because they do not match, $BPPCWrongE$ and $BPWrongE$ are asserted. The hazard unit asserts $FlushD$ and $FlushE$ to flush the two instructions at addresses 0x200 and 0x204. bpred updates $PCNextF = PCCorrectE = 0x104$, and the processor resumes fetching at the correct target address.
- ▶ **In cycle 4**, the branch commits by writing the BTB entry: $PCM = 0x100$ with $IEUAdrM = 0x200$ and an IC of $BranchM = 1$. gshare updates by moving to the Weakly Taken state. The GHR shifts in a 0 for the untaken branch.

Table 13.1 Branch outcomes and recovery.

Correct Predicted PC? $PCD == PCCorrectE$	Branch Direction $PCSrcE$	Flush Pipeline?	$PCNextF$
Yes	either	No	$PCF + 4$
No	0: Not Taken	Yes	$PCCorrectE = PCE + 4$
No	1: Taken	Yes	$PCCorrectE = IEUAdrE$

**Fig. 13.21** Timing diagram of branch misprediction recovery.

If both the ITLB and DTLB miss in the same cycle, the HPTW gives the DTLB priority and handles the DTLB miss before handling the ITLB miss.

13.3.4 ITLB Misses

As introduced in [Section 12.3.4](#), the LSU HPTW services both LSU and IFU TLB misses. When the IFU's ITLB misses, it asks the LSU's HPTW to find the missing PTE. [Fig. 13.22](#) shows the interface between these modules, with optional signals in gray. Specifically, the ITLB asserts `ITLBMissF` and sends `PCF` to the HPTW. The page table walker handles the ITLB miss, returns `PTE` and `PageType`, and pulses `ITLBWriteF` to write PTE into the ITLB. If `SVADU_SUPPORTED = 1`, the HPTW atomically updates access and dirty bits in memory and the ITLB.

13.3.5 Reset, Stalls, Flushes, and Traps

On reset, the cache clears the cache line valid bits. When a line is invalid, the other bits don't matter. The valid bits are implemented with flip-flops, so they are easy to reset.

Resetting the BTB and BHT/PHT is more problematic because these prediction memories are large and, hence, far more efficient to implement with SRAM, yet SRAM lacks a reset input. Thus, the branch memories are normally not reset but power up with unknown values. As a consequence, branch prediction is nondeterministic and may vary from one hardware boot to another depending on the random values in the prediction memories. In simulation, these memories begin with 0 values because they are implemented with the System Verilog `bit` type rather than `logic`. Recall from [Section 4.2.2.3](#) that `bit` has two values (0 and 1) in contrast to `logic`'s four: 0, 1, X, and Z.

The IFU's stalls, flushes, and traps closely resemble those of the LSU, as described in [Section 12.4](#). The IFU asserts `IFUStallF` to stall the pipeline

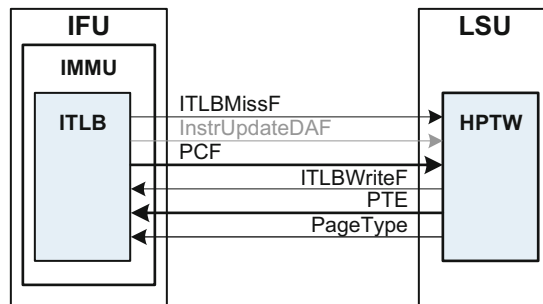


Fig. 13.22 ITLB/HPTW interface.

during multicycle cache operations (`ICacheStallF`), bus accesses, or ITLB misses. The IFU generates page, access, and misaligned instruction address faults during the Fetch stage as described in [Sections 8.2.5.2 and 11.4.3](#). These faults are pipelined as shown in Fig. 11.28 to generate traps in the Memory stage. During a multicycle operation, such as a bus access, the IFU disables interrupts by asserting `CommittedF` to ensure bus operations complete once in flight.

The hazard unit sends `StallID` and `FlushD` to the IFU. Like the LSU, when the IFU is stalled, it completes multicycle operations and then waits

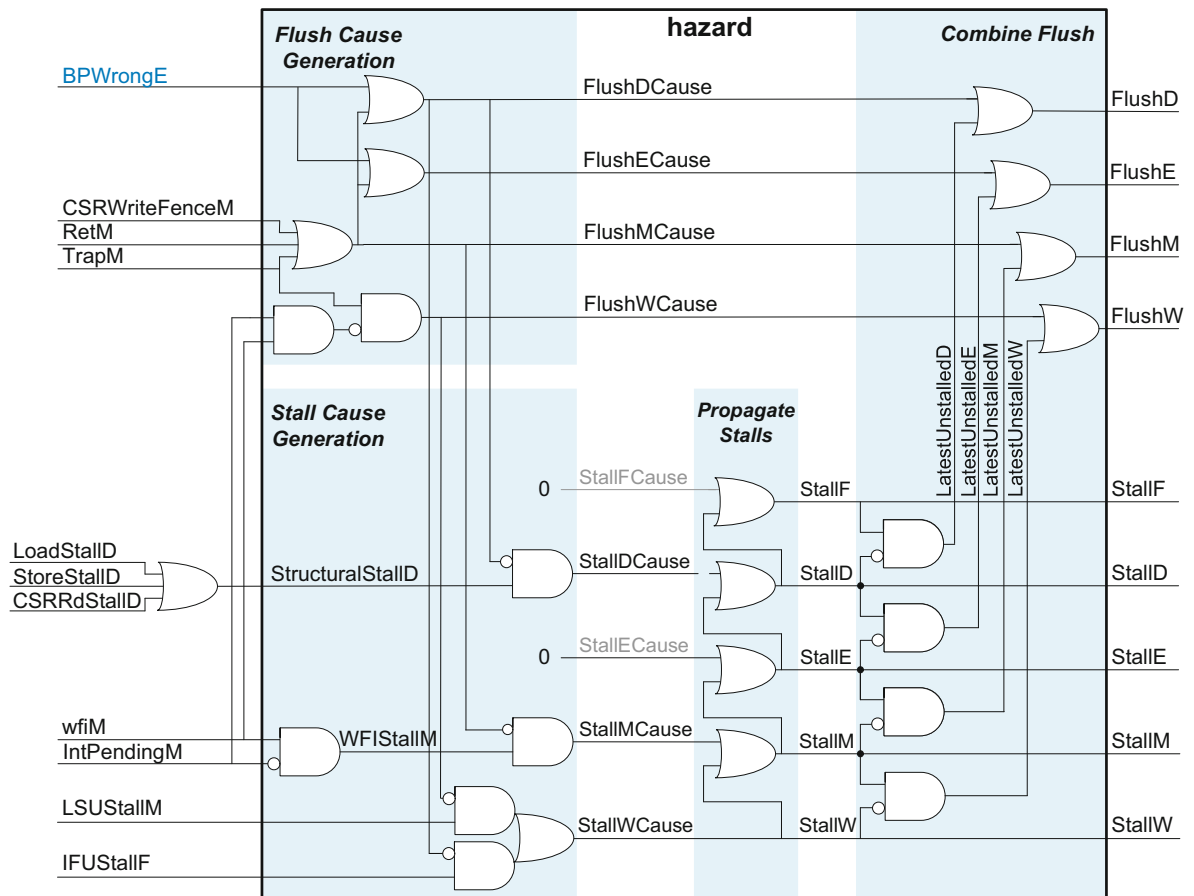


Fig. 13.23 Hazard unit with branch predictor changes highlighted in blue.

for the stall to be released. Flushes clear the instruction in the Decode stage to be a nop. They also cancel cache or bus operations before they begin but do not interfere with in-flight multicycle bus or cache operations.

Fig. 13.23 shows the updated hazard unit with changes to support branch prediction in blue. `BPWrongE` replaces `PCSrcE` from Fig. 9.33. In Chapter 9, the processor assumed not taken on a branch and simply fetched `PC + 4`. If the branch was actually taken (`PCSrcE = 1`), the processor flushed the Decode and Execute stages. Now those stages are only flushed when the branch predictor fetched the wrong instruction (`BPWrongE = 1`).

13.4 SUMMARY

Like the LSU, the IFU is highly configurable. In the simplest configuration, it has only an IROM, `PC + 4` logic, and a branch multiplexer. In the most sophisticated configuration, it interfaces to the EBU for main memory access, has an instruction cache to service most accesses, and contains an MMU with PMA and PMP physical address checkers and a TLB for virtual to physical address translation. Moreover, the IFU adds a branch predictor. The full PC logic chooses the next program counter from one of `PC + 4`, the predicted BTA, the trap vector, or the trap return address.

Gshare is a widely-used and reasonably effective branch predictor. It maintains a PHT of 2-bit branch predictions. It also adds a GHR that tracks the direction of recent branches. The PHT is indexed with an XOR of the least significant bits of the PC and the GHR, so it can make predictions that depend on both the branch being executed and the history of recent branch directions. The branch predictor takes some time to warm up the PHT. After warming up, a PHT with 4-Ki-entries achieves around 96% accuracy.

The branch predictor needs to provide the predicted BTA in the Fetch stage to update the PC even though the Wally pipeline does not determine the actual target address until the Execute stage. Therefore, the branch predictor adds a BTB, which caches the target address of recent branches. It can be viewed as a direct-mapped cache indexed by the less significant bits of the PC.

When a function is called from multiple places in a program, a standard BTB is not good at predicting where to return. However, a RAS that tracks the return addresses of recent function calls can easily determine the return address.

The complete branch predictor includes a Gshare direction predictor, BTB, and RAS. The true branch outcome is calculated in the Execute stage. If the prediction was wrong, the pipeline must flush the incorrectly fetched instructions and restart at the true address. The predictor also updates the BHT/PHT, GHR, BTB, and RAS based on the branch outcome. Subtle logic is needed to handle stalls and flushes and to repair incorrect predictions.

Extensions: C (Compressed)

- 14.1 **RISC-V Practices**
- 14.2 **Test Plan**
- 14.3 **Wally Implementation**
- 14.4 **Summary**

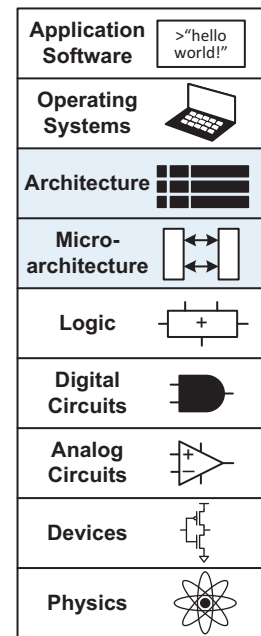
Most reduced instruction set computer (RISC) architectures were developed with 32-bit instruction encodings. As they began to compete with 16-bit microcontrollers in cost-sensitive markets, RISC was at a disadvantage because compiled programs typically required more nonvolatile memory, which is one of the costliest parts of the system. Moreover, fewer 32-bit instructions fit in a cache, resulting in higher miss rates. Therefore, many RISC architectures have introduced compressed instruction sets that provide 16-bit representations of commonly used instructions. These compressed instructions are mostly a subset of the full instruction set and can be “inflated” back to 32 bits to be executed by the normal execution units.

A processor that supports compressed instructions may execute both compressed and regular 32-bit instructions in the same program. So, a 32-bit instruction might be fetched from a 16-bit boundary, crossing cache lines or IROM words. This is called a *spill* and requires two memory accesses to fetch both halves of the instruction.

14.1 RISC-V PRACTICES

The optional RISC-V C extension defines compressed instructions. This compressed instruction set is summarized on the last pages of the book in Tables I.6 and I.22, and compressed instruction formats are in Fig. I.2. All compressed instructions are 16 bits long. To achieve this, the first source register is generally also used as the destination, and immediates are shorter than in ordinary instructions. Compressed instructions use many formats, trading regularity for density. The formats have different numbers of function (*funct*) bits. Some instructions only allocate 3 bits to specify a register, so they are restricted to using only eight registers (x8-x15).

The compressed instruction format is cleverly chosen to play gracefully with uncompressed instructions. Whereas the two least significant bits of uncompressed instructions are always 11, these bits have a different value for compressed instructions. Thus, the instruction encodings do not



Other RISC compressed instruction sets include the ARM Thumb instructions, MIPS16e, PowerPC VLE, and Hitachi SHcompact.

See the RISC-V specification for complete details about compressed instruction formats.

A few compressed instructions decompress into two or more standard RISC-V instructions.

overlap for compressed and uncompressed instructions, and a program can freely use both. The register fields, when present, are in the same place in all compressed formats to reduce the cost of decompression, and the most significant bit of the immediate is always in instruction bit 12 to simplify sign extension.

Every compressed instruction maps onto an equivalent 32-bit instruction. Therefore, instead of adding a whole new datapath, a RISC-V processor can decompress compressed instructions and then execute that decompressed instruction normally on the existing datapath.

Example 14.1 COMPRESSED INSTRUCTION ENCODING

Give the machine language encoding for `c.add x5, x6` and show how it decompresses to a 32-bit instruction.

Solution: According to Table I.6 (on the last pages of the book), `c.add` is a CR-type instruction with `op = 10` and `funct4 = instr15:12 = 1001`. The `rd/rs1` field is 5 and the `rs2` field is 6. Hence, the machine language is:

```
funct4 rd/rs1 rs2  op
1001   00101  00110 10 = 0x929A
```

The instruction inflates to `add x5, x5, x6`, with machine encoding:

```
funct7 rs2 rs1 funct3 rd  op
0000000 00110 00101 000 00101 0110011 = 0x006282B3
```

According to the RISC-V Unprivileged Specification 20240411, a compiler that uses compressed instructions when possible reduces code size by 25%–30% for typical programs [RISCV-Unprivileged].

To make best use of the limited encoding space, certain compressed machine language opcodes expand into different instructions in different ISAs. For example, the instruction with `op = 01`, `instr15:13 = 001` is `c.jal` in RV32IC but is `c.addiw` in RV64IC. The instruction with `op = 00`, `instr15:13 = 011` is `c.flw` in RV32ICF but is `c.ld` in RV64IFC. Therefore, a program compiled for RV32IC will not run correctly on RV64IC.

14.1.1 Zc* Extension

The Zc* family of extensions provides additional compressed instructions to improve code density even more than the original compressed (C) extension, especially for embedded processors where the IROM is a major part of system cost.

The C extension defines various compressed floating-point loads and stores: `c.flw`, `c.flwsp`, `c.fsw`, `c.fswsp`, `c.fld`, `c.fldsp`, `c.fsd`, and `c.fsdwp`. The Zca extension is a subset of the C extension that omits these compressed floating-point load/store instructions to save hardware in embedded processors with no floating-point unit (FPU). Zcf and Zcd define these missing compressed single- and double-precision loads and stores, respectively.

Zcb defines more compressed instructions overlooked in C. When code density is important, data density is also important, so the Zcb extension adds instructions to load and store bytes and half-words. Subword arithmetic can overflow into the upper part of the word, so it is often followed by sign- or zero-extension to discard the overflow. Zcb includes these subword zero-/sign-extension instructions. Zcb also provides compressed multiply and NOT instructions, which are common in integer programs. Certain Zcb instructions rely on the Zbb basic bit manipulation extension or the M multiplication extension described in [Chapters 15](#) and [18](#).

Saving and restoring registers on the stack takes many instructions in RISC-V. Other architectures have discovered dedicated PUSH and POP instructions improve code density by saving or restoring multiple registers with a single instruction. `c.popret` pops registers and also returns from a function call. These operations decompress into two or more standard RISC-V instructions and are included in the Zcmp extension. Similarly, the Zcmt extension defines table jumps, which improve code density for jumps and function calls. They use a new *jump vector table (JVT)* CSR that points to a table of jump addresses; table jumps do not correspond to standard RISC-V instructions. Zcmp and Zcmt redefine compressed double-precision load and store opcodes, so they are incompatible with the full C extension and Zcd.

In summary, the Zc* family of extensions are Zca, Zcd, Zcf, Zcb, Zcmp, and Zcmt:

- ▶ **Zca:** All C extension instructions except Zcf and Zcd F/D loads and stores. Excludes:

`c.flw, c.flwsp, c.fsw, c.fswsp, c.fld, c.fldsp, c.fsd, c.fsdwp`

- ▶ **Zcf:** Compressed single-precision loads & stores (requires F extension, RV32 only):

`c.flw, c.flwsp, c.fsw, c.fswsp`

- ▶ **Zcd:** Compressed double-precision loads and stores (requires D extension):

`c.fld, c.fldsp, c.fsd, c.fsdsp`

- ▶ **Zcb:** Subword and other basic integer instructions:

`c.lbu, c.lh, c.lhu, c.sb, c.sh`

`c.zext.b, c.not`

Zbb only: `c.sext.b, c.zext.h, c.sext.h`

M only: `c.mul`

RV64 only: `c.zext.w` (requires Zba extension)

- ▶ **Zcmp:** Push/pop compressed instructions (opcodes conflict with Zcd):

`cm.push, cm.pop, cm.popret, cm.popretz, cm.mva01s, cm.mvsa01`

- ▶ **Zcmt:** Table jump instructions (opcodes conflict with Zcd):

`cm.jt, cm.jalt`

The RV32GC/RV32IFDC extension implies $Zca + Zcf + Zcd$. RV64GC/RV64IFDC implies $Zca + Zcd$; they do not support Zcf because these instruction encodings are reused for 64-bit integer loads and stores. An RV64GC_Zcb_Zbb processor implements $Zca + Zcb + Zcd$, but not $Zcmp$ or $Zcmt$.

Zce is shorthand for $Zca + Zcb + Zcmp + Zcmt$. In RV32F, Zce also includes Zcf . It excludes Zcd , whose opcodes are reused for $Zcmp$ and $Zcmt$. Benchmark results show that Zce provides an average 11% further code size reduction over the C extension [codesize]. Zce is a natural replacement for the C extension in embedded processors.

Table I.22 lists the Zcb instructions and Fig. I.2 lists new formats used by these instructions (see the last pages of the book).

14.2 TEST PLAN

The C-extension instructions are tested with the riscv-arch-test suite as discussed in Section 7.4. Table 14.1 lists the number of test cases for each Zca and Zcb instruction.

14.3 WALLY IMPLEMENTATION

Wally uses the `ZCA_SUPPORTED`, `ZCB_SUPPORTED`, `ZCF_SUPPORTED`, and `ZCD_SUPPORTED` configuration parameters to enable compressed instructions. There is no `C_SUPPORTED` parameter. Instead, the C bit of the `misa` register is set if Zca is supported and either Zcd (and D) or Zcf (and F) are supported. Wally does not presently implement $Zcmp$ or $Zcmt$ and, therefore, does not support Zce .

Fig. 14.1 shows the Wally IFU changes required to support compressed instructions. Wally uses an optional decompression unit (`decompress`, abbreviated to `decomp` in Fig. 14.1), which receives 16-bit compressed instructions and inflates them to their equivalent 32-bit instructions. The decompression takes place at the start of the Decode stage and must complete in time to access the integer register file (`regfile`, shown as `IRF` in Fig. 14.1). The PC logic adds 2 instead of 4 for compressed instructions. The IFU takes multiple cycles to handle spills, when an instruction wraps a cache line or IROM word. The following sections describe decompression, PC logic, and spills in detail.

14.3.1 Instruction Decompression

As shown in Fig. 14.2, `decompress` is combinational logic with a 32-bit input `InstrRawD` and a 32-bit output `InstrD`. If `InstrRawD` is uncompressed (detected by bits `[1:0] = 11`), it is passed through to `InstrD`. Otherwise, the combinational logic examines at least bits `[15:13]` and `[1:0]` to determine the type of compressed instruction and to inflate it to a 32-bit output, `InstrD`.

Table 14.1 riscv-arch-test C cases.

Instruction	RV32Zca Tests	RV64Zca Tests	RV32Zcb Tests	RV64Zcb Tests
c.add	584	668		
c.addi	375	440		
c.addi16sp	69	132		
c.addi4spn	20	19		
c.addiw		438		
c.addw		678		
c.and	585	670		
c.andi	376	439		
c.beqz	85	149		
c.bnez	85	149		
c.ebreak	1	1		
c.j	18	18		
c.jal	18			
c.jalr	31	31		
c.jr	31	31		
c.lbu			8	8
c.ld		13		
c.lh			8	8
c.lhu			8	8
c.ldsp		31		
c.li	32	32		
c.lui	31	31		
c.lw	14	13		
c.lwsp	31	31		
c.mul			589	677
c.mv	84	148		
c.nop	14	14		
c.not			66	130
c.or	590	679		
c.sb			8	9
c.sd		132		
c.sdsp		131		
c.sext.b			65	129
c.sext.h			66	130
c.sh			8	8
c.slli	86	148		
c.srai	87	148		
c.srli	85	149		
c.sub	580	670		
c.subw		670		
c.sw	70	131		
c.swsp	69	133		
c.xor	581	662		
c.zext.b			65	129
c.zext.h			66	130
c.zext.w				130
Total	4,632	7,829	957	1,496

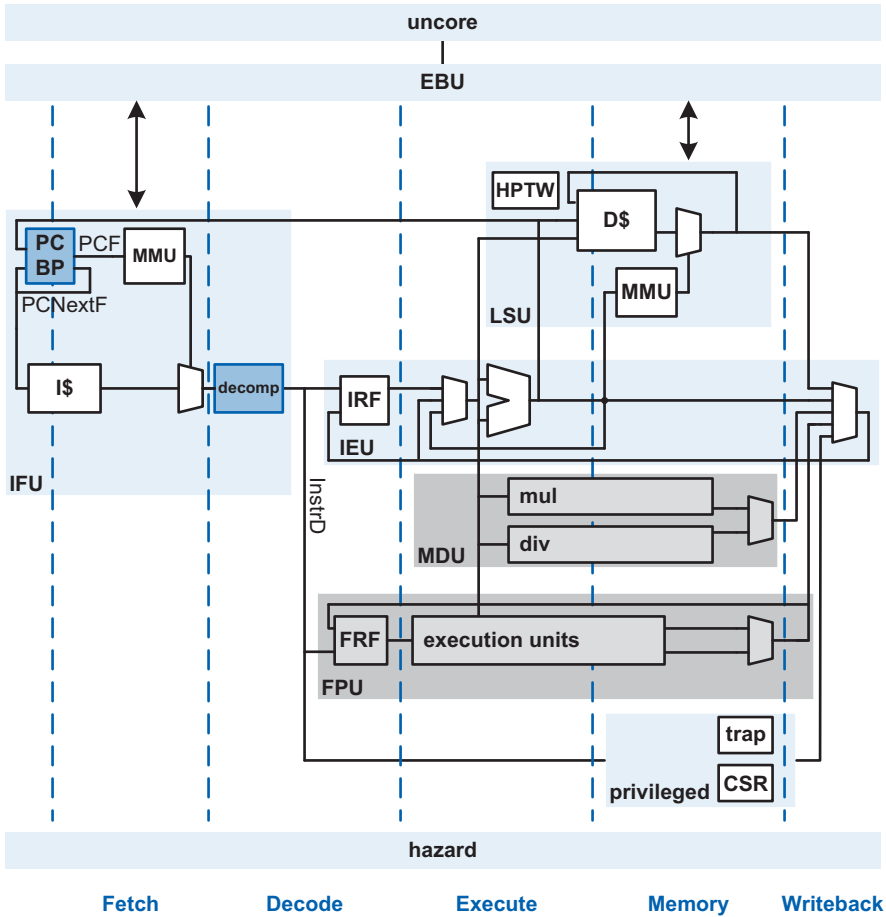


Fig. 14.1 Wally RV32IC/64IC high-level block diagram emphasizing the decompress unit.

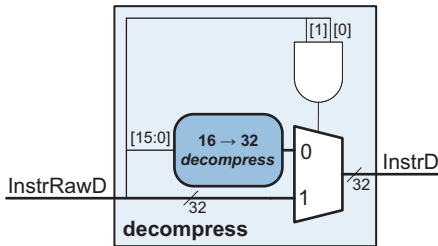


Fig. 14.2 decompress block diagram.

Code Example 14.1 shows a snippet of the `decompress` module's Verilog code. Based on `InstrRawD`'s `op` field (bits [1:0]) and `funct3` field (bits [15:13]), the module identifies the type of compressed instruction. It then identifies the registers and immediate from the other `InstrRawD` bits and cobbles them together into an uncompressed instruction. Thus, the decompressor can be viewed as a giant multiplexer that selects fields from the compressed instruction to emit an uncompressed instruction.

Code Example 14.1 EXCERPT OF `decompress` MODULE

```
assign op = instr16[1:0];
assign rslp = {2'b01, instr16[9:7]};
assign rdp = {2'b01, instr16[4:2]};
assign rs2p = {2'b01, instr16[4:2]};
assign rds1 = instr16[11:7];

assign immCL = {5'b0, instr16[5], instr16[12:10], instr16[6], 2'b00};
assign immCS = {5'b0, instr16[5], instr16[12:10], instr16[6], 2'b00};
assign immCI = {{7{instr16[12]}}, instr16[6:2]};

always_comb
  if (op == 2'b11) InstrD = InstrRawD; // Noncompressed instruction
  else case ({op, instr16[15:13]}) // Inflate compressed instruction
    5'b00010: InstrD = {immCL, rslp, 3'b010, rdp, 7'b0000011}; // c.lw
    5'b00110: InstrD = {immCS[11:5], rs2p, rslp, 3'b010, immCS[4:0], 7'b0100011}; // c.sw
    5'b01000: InstrD = {immCI, rds1, 3'b000, rds1, 7'b0010011}; // c.addi
    ...
```

Adding Zcb support simply involves extending the `decompress` module with more opcodes. The `ZCB_SUPPORTED` configuration flag indicates that these instructions are available.

14.3.2 PC Logic

When the IFU fetches an instruction, it asserts `CompressedF` if the instruction is compressed. Fig. 14.3 shows how the PC logic is modified to add 2 to PCF instead of 4 for compressed instruction, with the mux that selects 2 or 4 highlighted in bright blue.

As described in Section 8.2.5.2, instruction address-misaligned faults ordinarily occur if the branch or jump target address is not a multiple of 4. When compressed instructions are supported, that fault is not permitted because target addresses may be multiples of 2.

The branch predictor of Section 13.3.3 aligned the BHT (branch history table) to a 4-byte boundary, such that a 2^k -entry table used `PC[k+1:2]` to index the BHT. Because compressed instructions align to 2 bytes, the branch predictor should include `PC[1]`. A typical program contains a

IFU adds 2 to PCF to read 0x03D0 from address 0x320. The spill logic (spill) concatenates the two halves and then outputs the full 32-bit instruction.

When an instruction fetch spills, the spill module pipelines the fetch over 3 cycles:

- ▶ **Step 1:** The I\$/IROM sets up to read the instruction at PCNextF.
- ▶ **Step 2:** The I\$/IROM outputs the first 2 instruction bytes, and the spill module saves them to a local register. The I\$/IROM also sets up the next read with PCF+2.
- ▶ **Step 3:** The I\$/IROM outputs the next instruction bytes, and the spill module merges the two half instructions to form the full 4-byte instruction. The PC logic also sets PCNextF to PCF+4 to set up the read for the next instruction.

The spill adds one stall cycle of latency when both halves hit the I\$/IROM. A cache miss in either half behaves like a conventional I\$ miss.

Fig. 14.5 shows the complete IFU. Compared to Fig. 13.12, the new spill circuitry is highlighted in blue. The spill module adds a +2 adder and muxes to generate PCSpillNextF and PCSpillF. PCSpillNextF drives the

Unlike the I\$, the branch predictors continue to read using PCNextF because they do not need a separate prediction for the second instruction half.

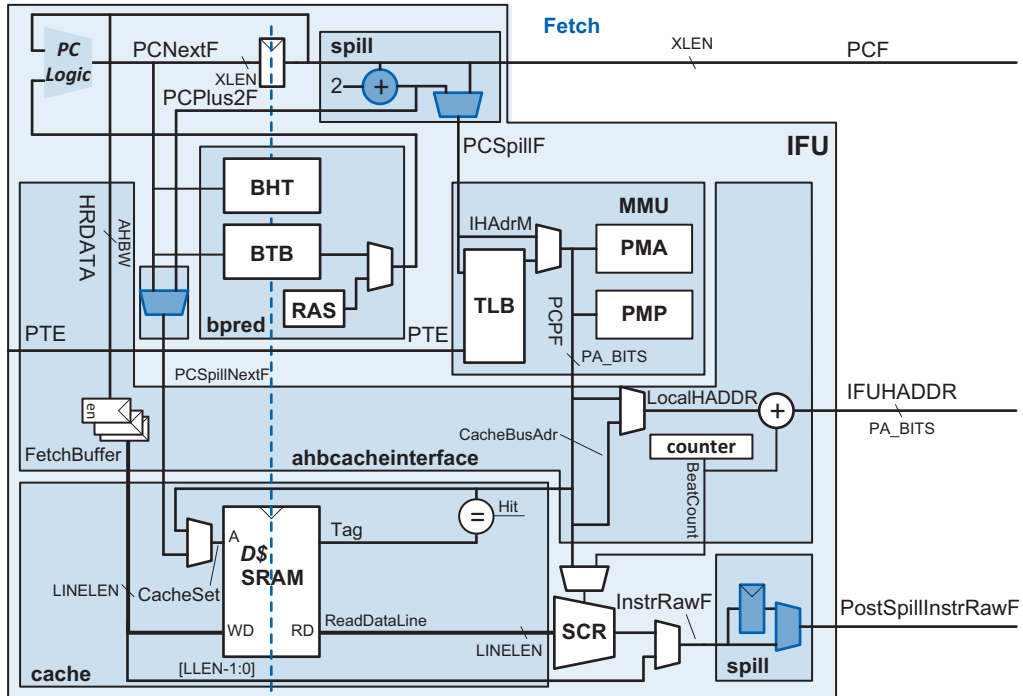


Fig. 14.5 IFU with spill highlighted in blue.

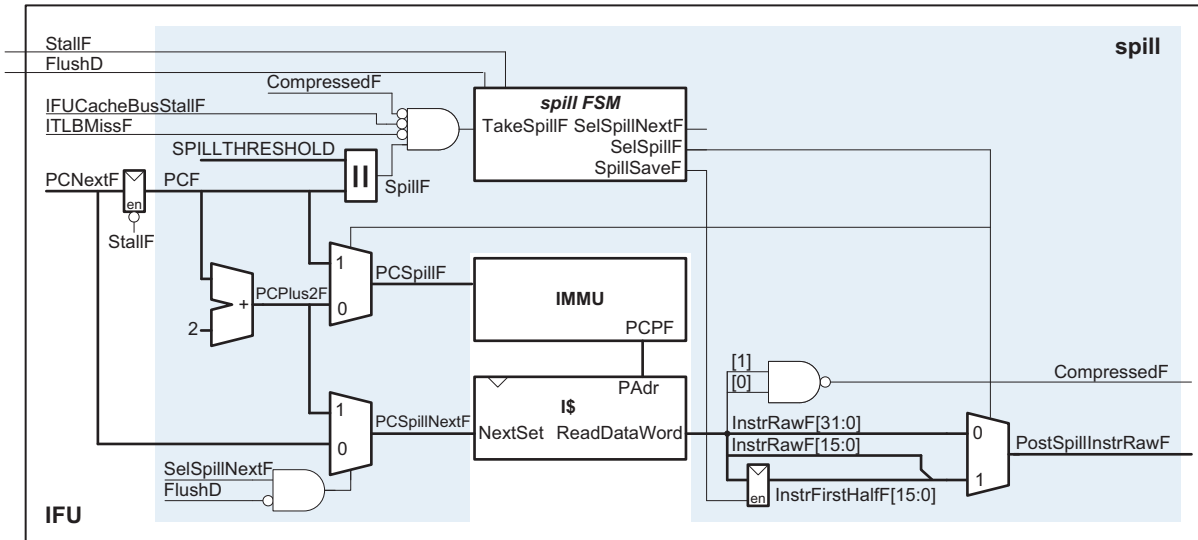


Fig. 14.6 spill module.

IROM address or the set bits of the I\$. PCSpillF goes through the MMU for address translation to become the physical program counter PCPF. Finally, the spill logic adds a register and mux to merge together the two halves of the instruction.

Fig. 14.6 shows a detailed view of the spill module. When handling a spill, the module asserts SelSpillNextF that drives IFUStallF to stall the pipeline. The SPILLTHRESHOLD comparator asserts SpillF when a 4-byte instruction read crosses two cache lines. For example, with 32-byte cache lines, SpillF is asserted when PCF[4:1] = 1111.

Upon a spill, the first access is a usual I\$ read: PCSpillNextF = PCNextF. After the clock edge, a spill is detected (TakeSpillF = 1 because PCF == SPILLTHRESHOLD), and all the spill FSM outputs rise. As a result, PCSpillNextF = PCPlus2F (SelSpillNextF = 1), and the first half of the instruction is saved in InstrFirstHalfF (because SpillSaveF = 1). Just after the next clock edge, the I\$ outputs the upper half of the instruction in InstrRaw[15:0], and SelSpillF = 1 selects the merged instruction to send to the IFU: PostSpillInstrRawF = {InstrRaw[15:0], InstrFirstHalf[15:0]}. Cache misses make this process take longer.

The spill FSM in Fig. 14.7 stays in the Ready state on a regular instruction read and to read the lower half of an instruction. When the FSM detects a spill (TakeSpillF = 1), it transitions to the Spill state where it waits for the upper half of the instruction read to complete.

Fig. 14.6 shows an adder producing PCPlus2F, but the RTL actually uses a multiplexer to save hardware. When PCF[1] = 0, adding 2 just sets PCF[1]: PCPlus2F = {PCF[XLEN-1:2], 2'b10}. Otherwise, PCPlus2F = {PCPlus4F[XLEN-1:2], 2'b00}. For example, if PCF = 1100 = 12, PCPlus2F = 1110 = 14 (set PCF[1]). If PCF = 0010 = 2, PCPlus4F = 0110 = 6 and PCPlus2F = 0100 = 4 (clear PCPlus4F[1]).

Fig. 14.8 shows a timing diagram of an IFU fetching the load immediate (li) instruction (0x03D00C13) from address 0x031E (see Fig. 14.4) with cache misses on both lines.

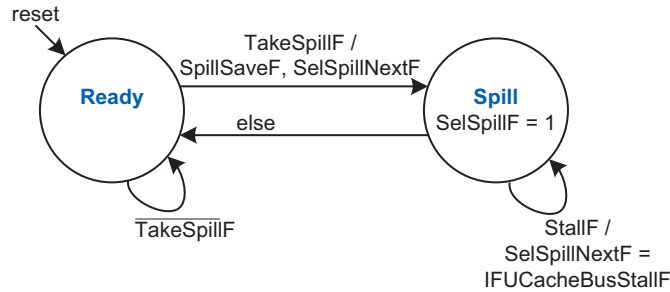


Fig. 14.7 spillfsm.

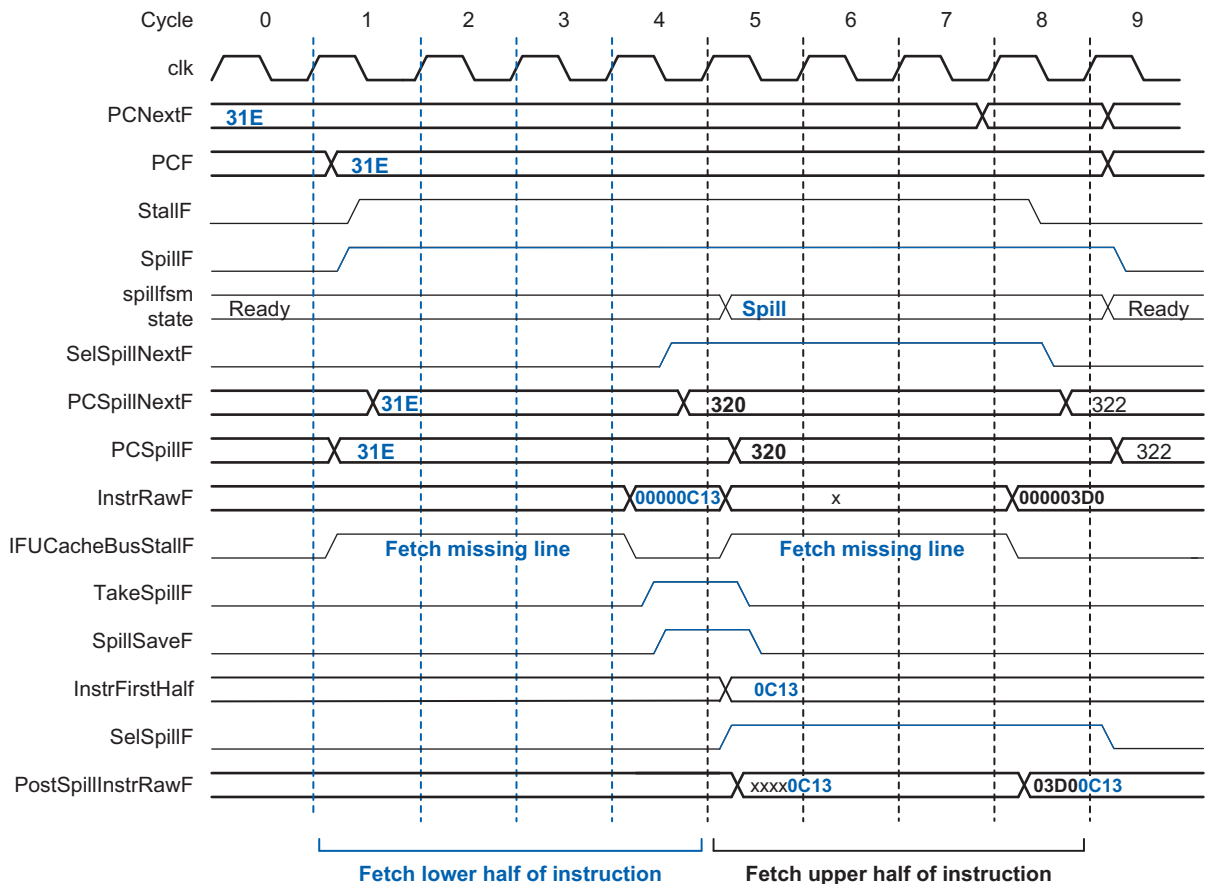


Fig. 14.8 Spill timing diagram.

- ▶ **Cycle 1: Step 1:** `SpillF` asserts when the IFU attempts to fetch from `PCF = 0x31E`.
- ▶ **Cycles 2 and 3:** The IFU stalls, waiting for the I\$ to fetch the missing cache line.
- ▶ **Cycle 4: Step 2:** The I\$ returns `InstrRawF = 0x00000C13`. The `spill` module detects a spill (the address is split across two cache lines) and raises `TakeSpillF`, which makes the `spillfsm` raise `SpillSaveF` (to capture the lower half of the instruction into the `InstrFirstHalf` flop) and `SelNextSpillF` (to select `PCSpillNextF = 0x320` as the input to the I\$). The spill FSM also transitions to the Spill state.
- ▶ **Cycles 5-7:** The IFU stalls, waiting for the I\$ to fetch the missing line from address `0x320`.
- ▶ **Cycle 8: Step 3:** The I\$ returns `InstrRawF = 0x000003D0`. The `spill` module picks off the lower bits, `0x03D0`, and the `PostSpillInstrRawF` mux merges the two halves (`0x03D00C13`).

An alternative solution is to place a small *fetch buffer* between the Fetch and Decode stages. The three-entry circular buffer fetches ahead of the PC, and the decoder can pull 2 or 4 bytes of instruction as necessary, avoiding a stall if the next entry is in the buffer. A fetch buffer also decouples the Fetch stage from later-stage stalls, potentially improving the critical path. Wally does not yet support fetch buffers.

14.4 SUMMARY

The RISC-V C/Zc* extension defines a set of 16-bit encodings for common instructions. By using these instructions, the compiler can reduce code size by 25%–30%, which effectively reduces the cost of IROM or increases the number of instructions that can fit in the IROM or I\$.

To support compressed instructions, the Wally IFU adds a `decompress` module in the Decode stage that detects compressed instructions and replaces them with their 32-bit equivalent. The remainder of the pipeline executes the uncompressed instructions.

The PC logic also changes to add 2 instead of 4 after fetching a compressed instruction. As a consequence, a 32-bit instruction may spill over from 16 bits of one cache line or IROM word into 16 bits of the next. The IFU must detect such spills, fetch both lines or words, and reassemble the instruction.

The `riscv-arch-test` suite contains test cases for compressed instructions.

Extensions: M (Multiply and Divide)

- 15.1 Principles
- 15.2 RISC-V Practices
- 15.3 Test Plan
- 15.4 Wally Implementation
- 15.5 Summary

Integer multiplication and division can be performed in software with a series of shifts and additions or subtractions. However, these operations (especially multiplication) are common enough that most microprocessors implement them in hardware for speed. RISC-V has an optional M extension so that these operations are not required in a minimal core but can be added in a standardized way.

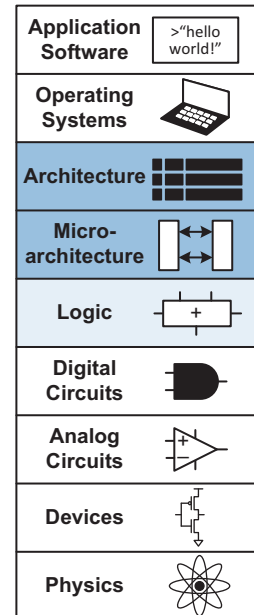
15.1 PRINCIPLES

This section begins by reviewing why multiplication and division are different for signed and unsigned numbers. It then presents algorithms and implementations for integer multiplication and division.

15.1.1 Signed and Unsigned Arithmetic

Recall that addition, subtraction, and logical operations produce the same answer whether a number is treated as unsigned or signed (two's complement). Examples are easiest to express with a short word size, $XLEN = 4$. Consider the binary addition $1110 + 0001 = 1111$. Interpreted in decimal as unsigned, this is $14 + 1 = 15$. Interpreted as signed, this is $-2 + 1 = -1$. In either interpretation (signed or unsigned), the answer has a bit pattern of 1111.

However, multiplication and division produce different answers depending on whether numbers are signed or unsigned. For example, consider multiplying two 4-bit numbers to get an 8-bit product: 1111×1111 . 1111 is 15 as an unsigned number but -1 as a signed number. Interpreting both numbers as unsigned (unsigned $\times$ unsigned multiplication), the product is $15 \times 15 = 225 = 11100001$. Interpreted as signed $\times$ unsigned, the product is $-1 \times 15 = -15 = 11110001$. Interpreted as signed $\times$ signed, the product is $-1 \times -1 = 1 = 00000001$. In each case, the four least significant bits are 0001. However, the four most significant bits depend on the flavor of multiplication. RISC-V includes the `mul` instruction that calculates the lower 32 bits of 32×32 -bit



multiplication, which are the same for all flavors. `mulh`, `mulhu`, and `mulhsu` produce the upper 32 bits of the multiplication, interpreting the operands as signed, unsigned, and signed/unsigned, respectively.

15.1.2 Carry-Save Addition

In computer arithmetic, we often need to add multiple operands—for example, when adding a multiplier’s partial products. The most obvious way to add k N -bit operands is with $k-1$ consecutive carry-propagate adders (CPAs) as shown in Fig. 15.1(a), but this approach is slow and costly. A better technique is called *carry-save addition*. Recall that a full adder sums three 1-bit inputs of unit weight (i.e., not shifted) and produces a sum output with unit weight and a carry output with double weight (i.e., shifted left by 1 bit): $\{C, S\} = X + Y + Z$. In other words, the carry output (C) belongs in the column to the left of the sum bit, S . If we place N full adders in parallel, they can accept three N -bit operands $X_{N-1:0}$, $Y_{N-1:0}$, $Z_{N-1:0}$ and produce two N -bit outputs $S_{N-1:0}$ and $C_{N-1:0}$ that satisfy $X + Y + Z = 2C + S$, as shown in Fig. 15.1(b). This is called *carry-save redundant format* because the carry output of each column is preserved rather

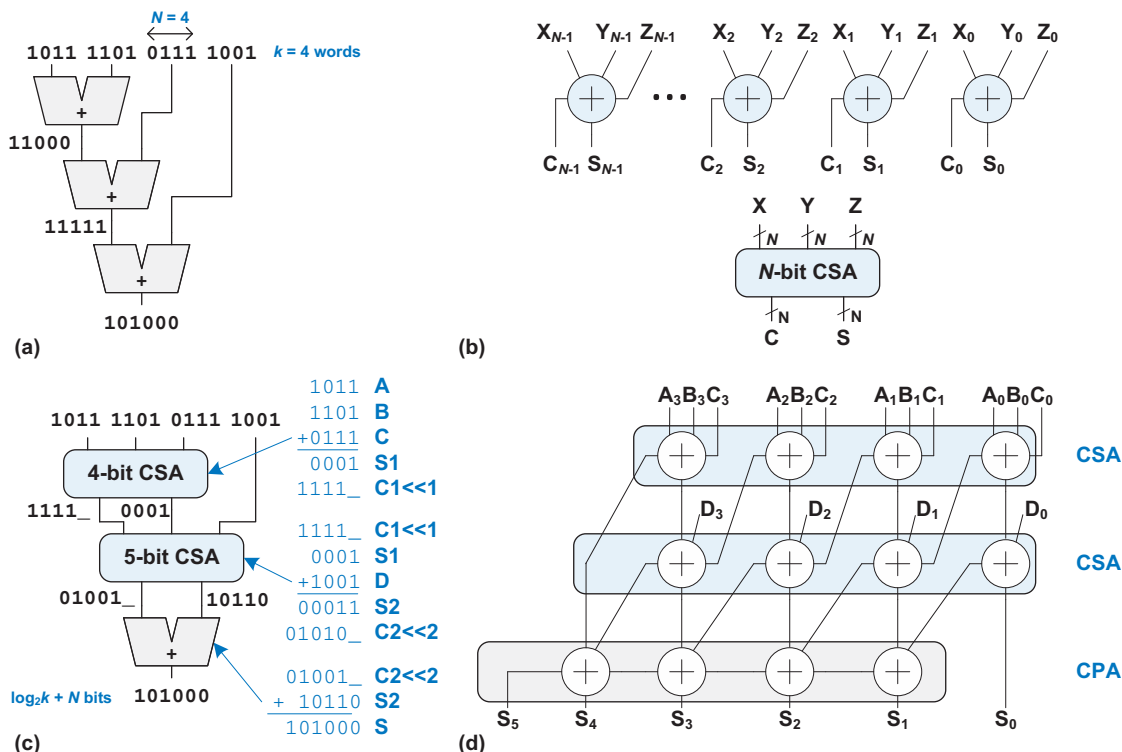


Fig. 15.1 Multiple-input adders: (a) CPAs, (b) N -bit CSA, (c) CSAs and 1 CPA, (d) CSAs and 1 CPA details.

than propagated along the adder. In this application, the full adders are called 3:2 *carry-save adders* (CSAs) because they accept 3 inputs and produce two outputs in carry-save (redundant) form; CSAs are also referred to as 3-to-2 adders. To obtain the sum $X + Y + Z$ in nonredundant binary form (i.e., as a single binary number), the carry word C is shifted left by one column (because it has double weight) and added to the sum word S with an ordinary CPA. Alternatively, a fourth word could be added to the carry-save redundant result with another row of CSAs, again giving a carry-save redundant result. This result could again be summed with a CPA as shown in Fig. 15.1(c and d).

We use carry-save addition in Section 15.1.3 to quickly add partial products in a multiplier and in Section 16.2.3 to quickly compute the partial remainder for division and square root.

15.1.3 Multiplication

Consider multiplying an M -bit multiplicand $Y = \{y_{M-1}, y_{M-2}, \dots, y_1, y_0\}$ by an N -bit multiplier $X = \{x_{N-1}, x_{N-2}, \dots, x_1, x_0\}$ to produce an $M+N$ -bit product P . If X and Y are both unsigned, then the product can be written as

$$P = \left(\sum_{j=0}^{M-1} y_j 2^j \right) \left(\sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j} \quad (15.1)$$

This can be viewed as forming N appropriately shifted partial products of M bits each, then summing the columns to compute the product. 1×1 -bit binary multiplication is equivalent to a logical AND operation, so generating partial products consists of taking the logical AND of the appropriate bits of the multiplier and multiplicand. Fig. 15.2(a) shows these partial products for 4×4 -bit multiplication of unsigned numbers, and Fig. 15.2(b) shows the details of multiplying $1011 \times 0101 = 0110111$.

Large multiplication is easier to illustrate using dot diagrams. Fig. 15.3 shows a Dadda dot diagram for a 16×16 unsigned multiplier. The diagrams in this text are adapted from [Dadda65, Bewick94, Weste12]. Each dot represents a bit. The blue dots on the right are the multiplier bits, and horizontal boxed rows represent partial products, shifted according to their weight. Each partial product is either the multiplicand Y or 0 depending on whether the multiplier bit for that row is 1 or 0.

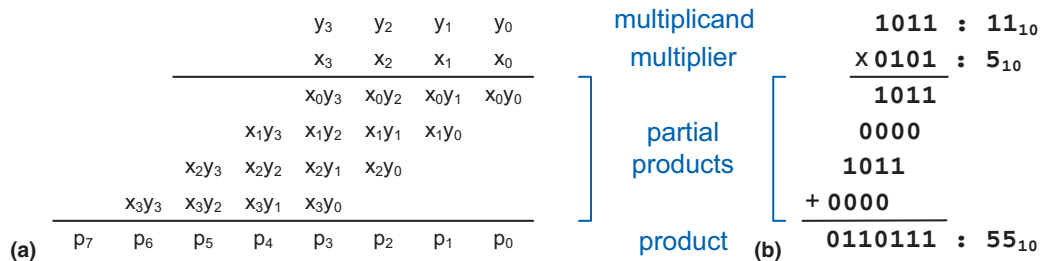


Fig. 15.2 Unsigned multiplication: (a) operations and (b) example.

A CPA takes N full adder delays to add two N -bit numbers. In contrast, a CSA takes only 1 full adder delay to add three N -bit numbers; however, the result is in redundant format.

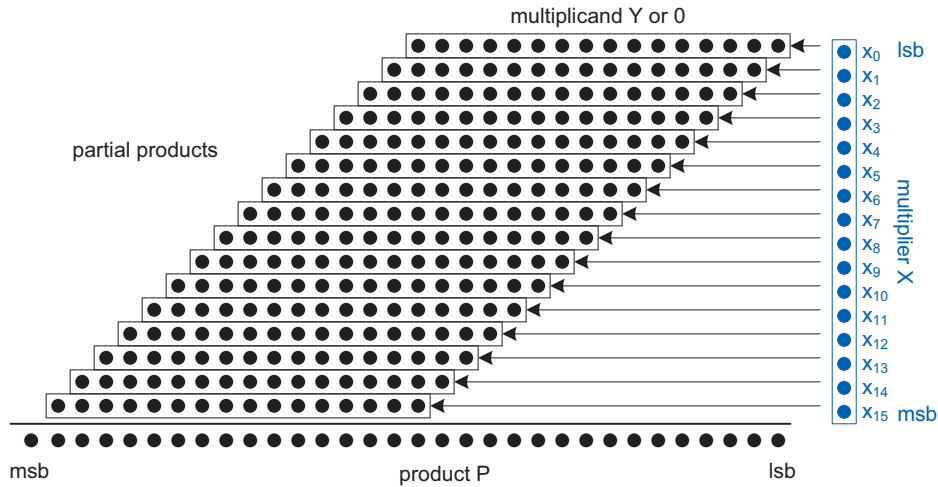


Fig. 15.3 Multiplier dot diagram.

The speed, cost, and power consumption of multipliers are largely determined by how the partial products are summed. A simple approach is to add the first two partial products with a CPA, then add the third partial product to the running sum, and so forth. This requires $N-1$ CPAs and is slow even when using a fast CPA. However, various algorithms make adding partial products faster or more efficient.

The remainder of this section addresses multiplication implementation techniques, focusing on array multipliers, Booth encoding, and tree multipliers. An array multiplier uses an array of CSAs that is simple and reasonably fast. The array can also be easily modified to perform signed multiplication. Booth encoding is an important optimization to reduce the number of partial products. Tree multipliers can sum the partial products faster than array multipliers. Although other multiplier optimizations are known, logic synthesis tools are quite mature and generally produce excellent multipliers from behavioral code that uses the $*$ operator.

15.1.3.1 Array Multiplier

Practical multipliers use CSAs to add the partial products to reduce delay and size compared to CPAs. Fig. 15.4(a) shows a 4×4 array multiplier for unsigned numbers. Each square in the diagram contains a 2-input AND gate to compute one bit of a partial product, corresponding to a black dot in Fig. 15.3, and a full adder to add that bit to a running sum in carry-save redundant format. Note that the S_{in} and C_{in} inputs to the top row are 0 because there is no running sum yet, so the full adders could be omitted from that row. The least significant N output bits come directly from the sum outputs of the CSAs. The most significant N bits arrive in carry-save redundant format, so the multiplier needs a final CPA to add them together. In this implementation, the CPA is a ripple-carry adder made from one more row of full adders. Fig. 15.4(b) shows the same adder squashed into a rectangular form that fits a chip datapath more easily.

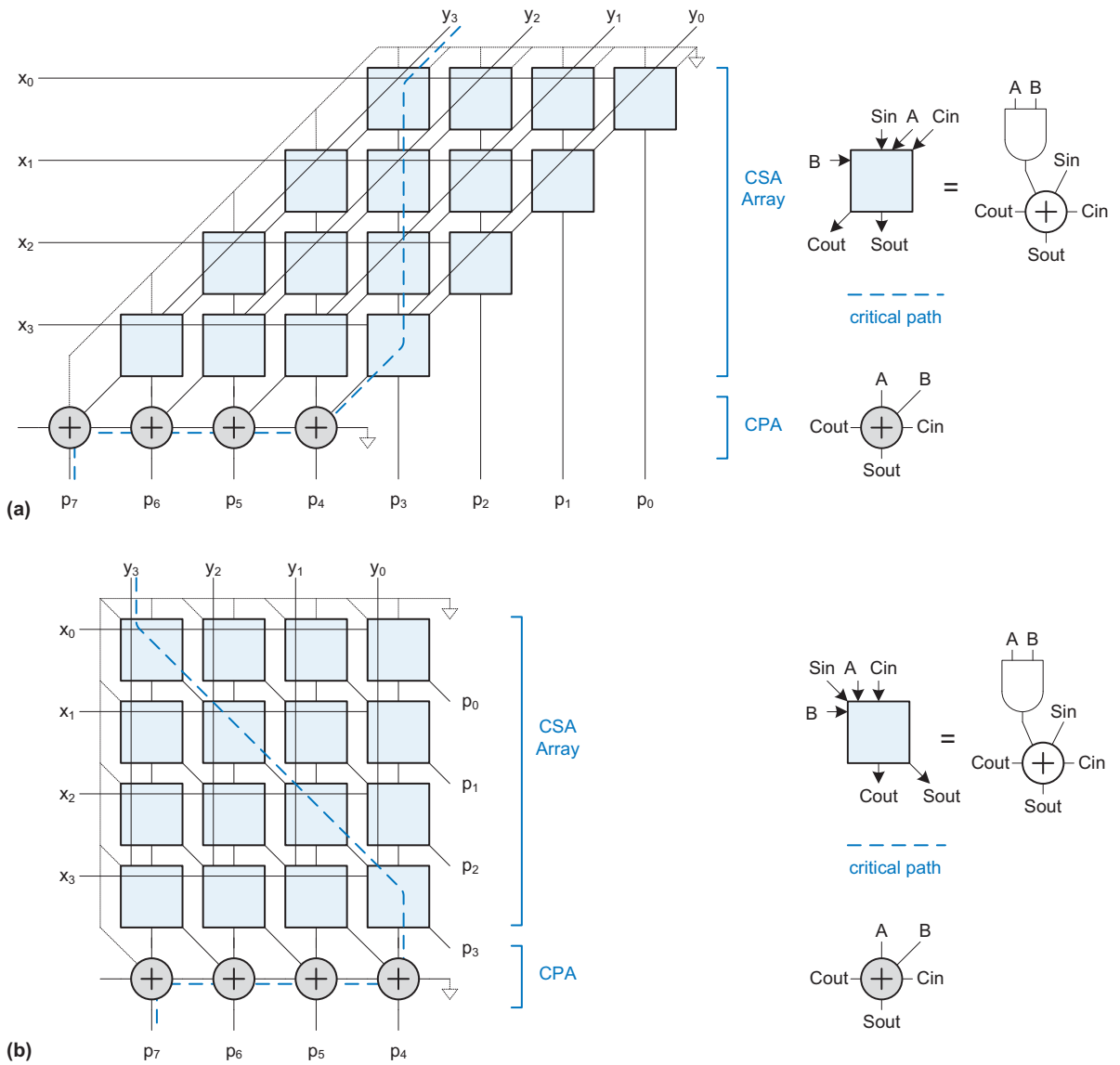


Fig. 15.4 Array multiplier: (a) concept and (b) rectangular form.

15.1.3.2 Signed Multiplication

When multiplying signed two's complement numbers, two of the partial products may be negative, as shown in Equation 15.2. Fig. 15.5 shows how these partial products, shown in blue, involve inverting the bits and adding 1. Observe how even the implicit zero (less significant) bits must be inverted.

Recall that the weight of the most significant bit of an N -bit two's complement number is -2^{N-1} .

All of the 1s in the bottom right of Fig. 15.5 add together to create the 1 in the top row of Fig. 15.6. The two 1s in column p_6 of Fig. 15.5 add together to produce the 1 in column p_7 in Fig. 15.6. The two 1s in column p_7 of Fig. 15.5 would only produce a carry out, which is discarded, so they are not represented in Fig. 15.6.

$$\begin{aligned}
 P &= \left(-y_{M-1}2^{M-1} + \sum_{j=0}^{M-2} y_j 2^j \right) \left(-x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i \right) \\
 &= \sum_{i=0}^{N-2} \sum_{j=0}^{M-2} x_i y_j 2^{i+j} + x_{N-1} y_{M-1} 2^{M+N-2} \\
 &\quad - \left(\sum_{i=0}^{N-2} x_i y_{M-1} 2^{i+M-1} + \sum_{j=0}^{M-2} x_{N-1} y_j 2^{j+N-1} \right) \quad (15.2)
 \end{aligned}$$

The *modified Baugh-Wooley multiplier* [Hatamian86] reduces the hardware cost by precomputing the sum of the constant ones and pushing some of the terms upward, as shown in Fig. 15.6. Fig. 15.7 shows a rectangular array implementation, which is almost identical to the unsigned multiplier in Fig. 15.4(b) except that the blue cells contain NAND instead of AND gates to compute the negative partial products, and two cells receive carries in of 0. By conditionally inverting (using XOR gates), the same hardware can perform both signed and unsigned multiplication.

$$\begin{aligned}
 &\sum_{i=0}^{N-2} \sum_{j=0}^{M-2} x_i y_j 2^{i+j} \left[\begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \end{array} \right] \\
 &+ x_{N-1} y_{M-1} 2^{M+N-2} \quad x_3 y_3 \\
 &- \sum_{i=0}^{N-2} x_i y_{M-1} 2^{i+M-1} \quad \begin{array}{ccccccc} 1 & 1 & \overline{x_2 y_3} & \overline{x_1 y_3} & \overline{x_0 y_3} & 1 & 1 & 1 \end{array} \\
 &- \sum_{j=0}^{M-2} x_{N-1} y_j 2^{j+N-1} \quad \begin{array}{ccccccc} 1 & 1 & \overline{x_3 y_2} & \overline{x_3 y_1} & \overline{x_3 y_0} & 1 & 1 & 1 \end{array} \\
 &\quad \begin{array}{cccccccc} p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \end{array}
 \end{aligned}$$

Fig. 15.5 Partial products for a signed multiplier.

$$\begin{aligned}
 &\begin{array}{ccccccc} & y_3 & y_2 & y_1 & y_0 \\ x_3 & x_2 & x_1 & x_0 \\ \hline & x_0 y_2 & x_0 y_1 & x_0 y_0 \\ x_1 y_2 & x_1 y_1 & x_1 y_0 \\ x_2 y_2 & x_2 y_1 & x_2 y_0 \end{array} \\
 &\quad \begin{array}{ccccccc} 1 & \overline{x_3 y_0} & x_0 y_2 & x_0 y_1 & x_0 y_0 \\ \overline{x_3 y_1} & x_1 y_2 & x_1 y_1 & x_1 y_0 \\ \overline{x_3 y_2} & x_2 y_2 & x_2 y_1 & x_2 y_0 \\ 1 & x_3 y_3 & \overline{x_2 y_3} & \overline{x_1 y_3} & \overline{x_0 y_3} \end{array} \\
 &\quad \begin{array}{cccccccc} p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \end{array}
 \end{aligned}$$

Fig. 15.6 Simplified partial products for a signed multiplier.

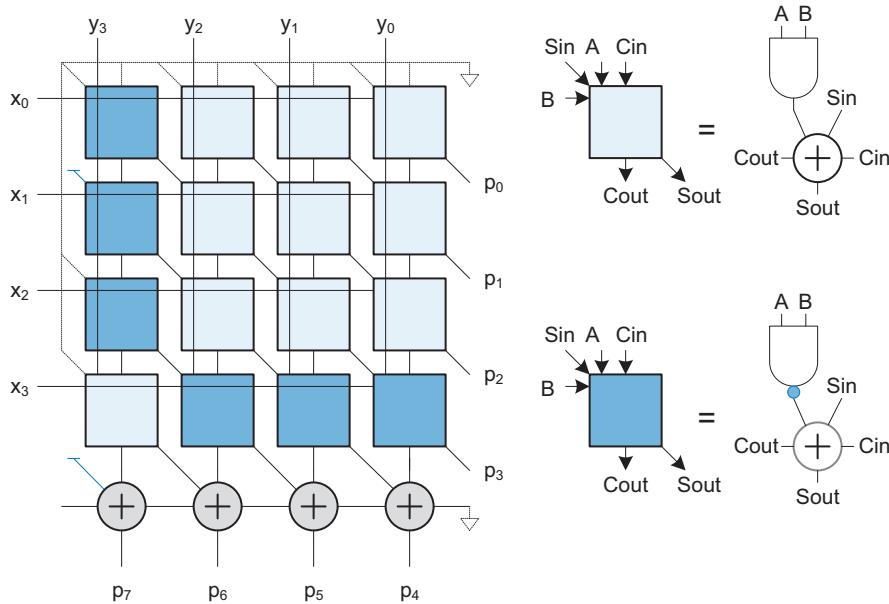


Fig. 15.7 Modified Baugh-Wooley signed multiplier.

15.1.3.3 Booth Encoding

The array multipliers considered so far are radix-2, producing one partial product (either multiplicand Y or 0) for each bit of the multiplier X . A radix-4 multiplier produces one partial product for every two bits of X , saving area and delay in the CSA array. The partial product is 0 , Y , $2Y$, or $3Y$. $2Y$ is a simple 1-bit shift of Y , but $3Y$ involves adding $Y + 2Y$ with an expensive and slow CPA.

Modified Booth encoding [MacSorley61] provides the cost and speed benefits of higher radix multipliers without generating the hard $3Y$ multiple. Instead, radix-4 modified Booth encoding generates multiples $-2Y$, $-Y$, 0 , Y , and $2Y$. It then exploits the property that $3Y = 4Y + (-Y)$ and $2Y = 4Y + (-2Y)$. Because each partial product has 4 times the weight of the previous partial product, adding $4Y$ is equivalent to incrementing the next partial product by 1. Hence, partial product i is selected by considering a pair of bits $\{x_{2i+1}, x_{2i}\}$, along with the most significant bit of the previous pair x_{2i-1} . When the most significant bit in a pair is 1, choose a negative partial product; the next partial product will need to increment to compensate. Thus, when the most significant bit of the *previous* pair is 1, add 1. For inputs with an even number of bits, one extra partial product of 0 or Y is needed to handle the increment from the most significant bit of X .

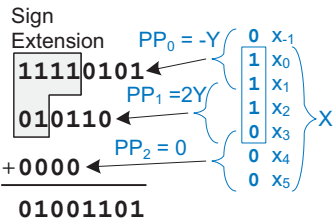


Fig. 15.8 Booth encoded example.

For simple radix-4 multiplication, $\{x_{2i+1}, x_{2i}\}$ of 00, 01, 10, and 11 produce partial products of 0, Y , $2Y$, and $3Y$. But with modified Booth encoding, the 10 and 11 combinations produce $-2Y$, and $-Y$ instead; then, the next partial product must add $4Y$ to produce a combined result of $4Y + (-2Y) = 2Y$ and $4Y + (-Y) = 3Y$, as desired. In other words, when the msb of a pair is 1 ($x_{2i+1} = 1$), the next partial product must add $4Y$, which is just Y in that 2-bit left-shifted position. Thus, when the previous bit is 1, $\{x_{2i+1}, x_{2i}, x_{2i-1}\} = 001, 011, 101$, and 111 , the partial products are Y , $2Y$, $-Y$, and 0 (i.e., $Y + 0 = Y$, $Y + Y = 2Y$, $Y + (-2Y) = -Y$, $Y + (-Y) = 0$).

Table 15.1 shows the partial product logic, and Fig. 15.8 shows an example of 4×4 -bit Booth-encoded multiplication with four partial products: $1011 \times 0111 = 01001101$ ($11 \times 7 = 77$).

Fig. 15.9 shows a radix-4 Booth encoder hardware implementation. Negative partial products involve inverting Y or $2Y$ and adding 1. Again, they must be sign-extended to fill the array. Fig. 15.10 shows these partial products, with s indicating the sign of partial product i . Observe how a negative partial product ($s = 1$) has 1s in all the upper bits and also a 1 on the next row. All of the sign extension bits are either 0 or 1. They can be simplified as all 1 plus the complement of the sign bit in the first position, as shown in Fig. 15.11(a). Now the partial product array has many constant ones, whose sum can be precomputed as shown in Fig. 15.11(b). In summary, Booth-encoding an $N \times N$ -bit unsigned multiplier results in $\lfloor N/2 + 1 \rfloor$ partial products, each up to $N + 4$ bits in length. These partial products can then be summed with CSAs as described earlier.

Booth encoding for signed two's complement multiplication is similar, but the multiplicand Y could be negative, so sign extension is performed on the sign bit of the partial product $e_i = PP_{iM}$ [Bewick94]. e_i is the exclusive-OR of the sign of Y and the sign of the multiple being selected. Only $N/2$ partial products are needed because $PP_{N/2} = 0$.

Table 15.1 Radix-4 Booth Encoding

Inputs			Partial Products	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	$Single_i$	$Double_i$	Neg_i
0	0	0	0	0	0	0
0	1	0	Y	1	0	0
1	0	0	$-2Y$	0	1	1
1	1	0	$-Y$	1	0	1
0	0	1	$Y (= Y + 0)$	1	0	0
0	1	1	$2Y (= Y + Y)$	0	1	0
1	0	1	$-Y (= Y - 2Y)$	1	0	1
1	1	1	$0 (= Y - Y = 0)$	0	0	1

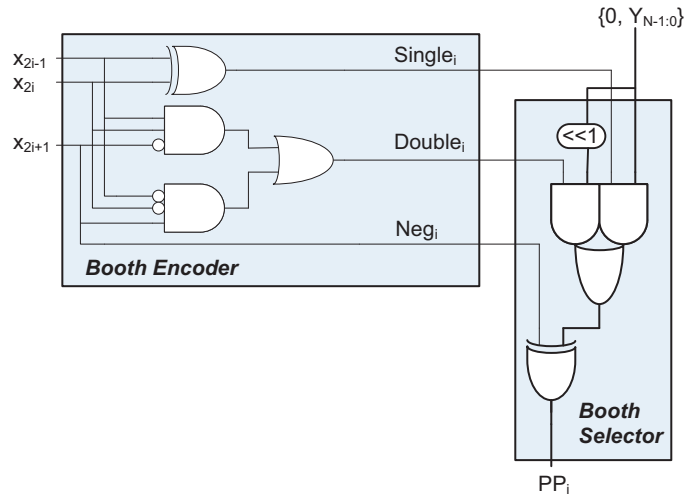


Fig. 15.9 Radix-4 Booth encoder.

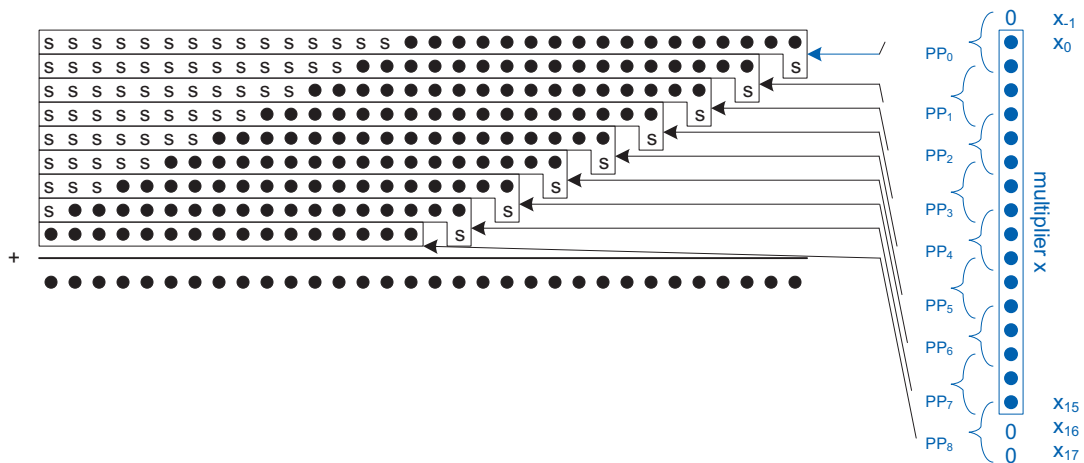
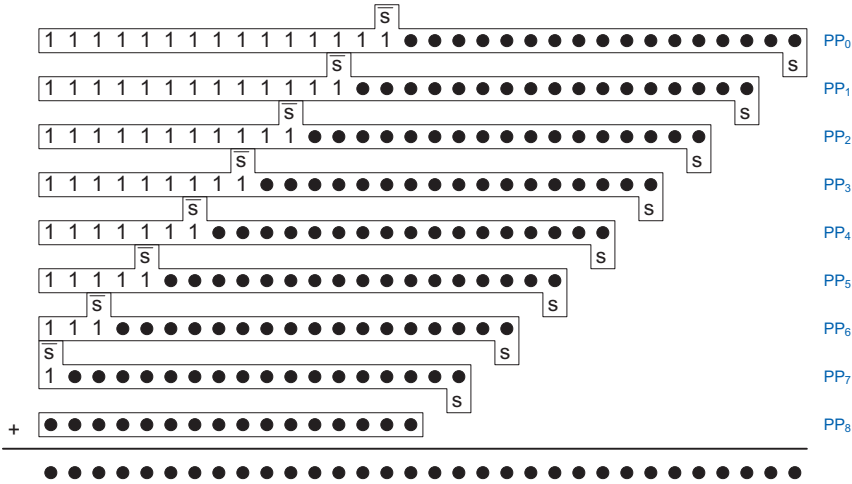


Fig. 15.10 Radix-4 Booth encoded partial products with sign extension.

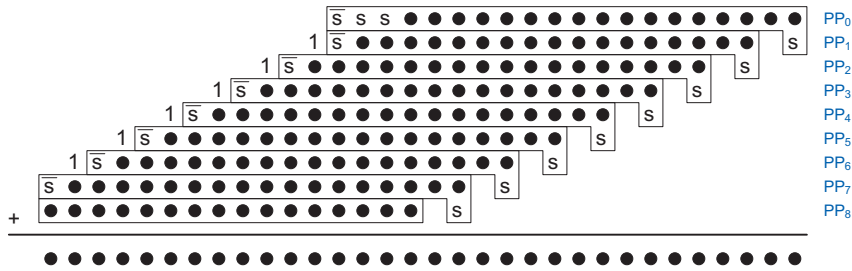
15.1.3.4 Tree Multipliers

A multiplier sums the partial products in carry-save redundant format using CSAs. It produces a final pair of values that are summed by a carry-propagate adder to give the final result. The array multipliers described so far sum k partial products with a linear array of $k-2$ CSAs, as shown in Fig. 15.13(a). A faster approach is to use a CSA tree to sum the partial products. For example, Fig. 15.13(b) shows a Wallace tree with $\log_{3/2}(k-2)$ levels of CSAs. The Wallace tree's irregular and sometimes lengthy wiring cuts into the benefit of fewer logic levels.

In Figs. 15.10 - 15.12, the s_i and e_i subscripts are dropped for readability. But remember that s and e are specific to a given partial product.



(a)



(b)

Fig. 15.11 Simplified sign extension: (a) partial products with sign extension, (b) precomputed.

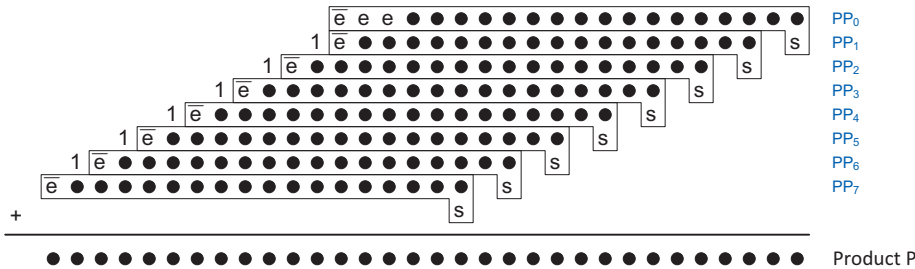


Fig. 15.12 Sign extension for Booth-encoded signed multipliers.

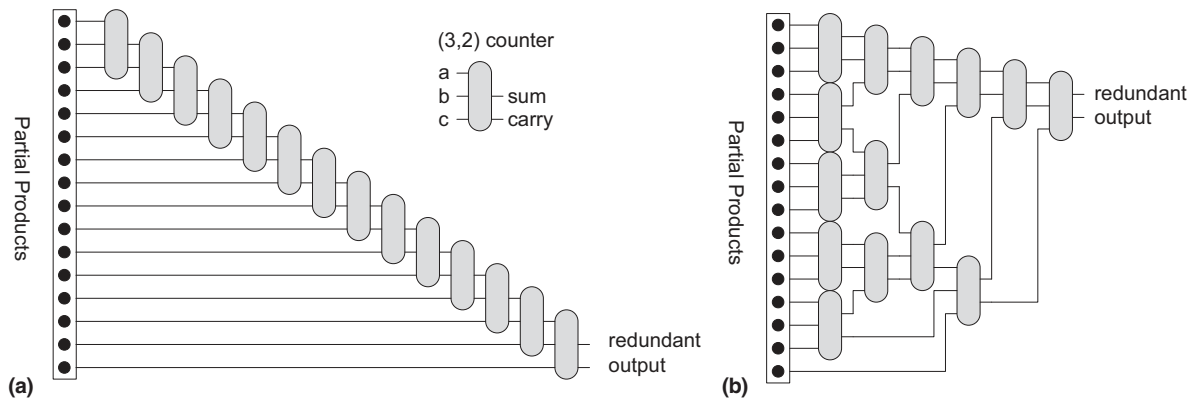


Fig. 15.13 Summing partial products: (a) linear array and (b) Wallace tree.

15.1.3.5 Other Multiplication Techniques

Radix-4 Booth encoding reduces the number of partial products to nearly half. Radix-8 Booth encoding extends this idea to reduce the partial products to nearly a third. However, it requires selecting from partial products of $\{4Y, 3Y, 2Y, Y, 0, -Y, -2Y, -3Y, -4Y\}$. The $3Y$ multiple is not trivial to obtain; it requires a CPA to compute $Y + 2Y$ before Booth selection begins. Thus Radix-8 Booth encoding only becomes beneficial for large multiplications.

Wallace tree multipliers suffer from irregular layouts with long, slow wires. Many other structures are used, including *[4:2] compressor trees* [Weinberger81, Santoro89], the *3-Dimensional Method* [Oklobdzija96, Stelling98], and hybrids between arrays and trees such as [Dhanesha95].

All multipliers involve a CPA that can be optimized with the usual techniques summarized in [Weste12] and also by exploiting the fact that the upper and lower bits arrive earlier than the middle bits [Zimmerman96]. When a multiplication is followed by one or more additions, the additions may be folded into the CSA tree.

15.1.3.6 Multiplier Synthesis

Logic synthesis of multipliers is a mature field. Synthesis tools use the techniques described earlier and generally produce multipliers that are as good as an engineer could handcraft with extensive effort. Therefore, the most practical technique for designing multipliers is to write

```
assign p = x * y;
```

and allow synthesis to map onto your target technology, automatically taking advantage of Booth encoding and CSA trees.

In C and related programming languages, the `/` and `%` operators are defined to compute $X / D = Q$ and $X \% D = REM$. Hence, for signed numbers, `%` may produce a negative result and is really the remainder operator, not a modulus operator. For example, $8 / 5 = 1$ and $8 \% 5 = 3$. $-8 / 5 = -1$ and $-8 \% 5 = -2$. However, $-8 \text{ modulus } 3 = 1$.

The *modulus operator* finds $M \in [0, D)$ satisfying $X = Q' \times D + M$. In other word, the modulus is nonnegative. If the remainder REM is nonnegative, $M = REM$. If REM is negative, $M = REM + D$.

The remainder W is also referred to as the *residual* or *residue*.

15.1.4 Division

Division of N -bit integers takes a *dividend* X and a *divisor* D . It computes a *quotient* Q and *remainder* REM satisfying $X = Q \times D + REM$. If X and D are signed; then REM has the same sign as X .

15.1.4.1 Radix-2 Restoring Unsigned Integer Division

In a fashion similar to elementary school long division, the quotient can be computed one bit at a time over N steps, as given in Fig. 15.14. In each step, the divisor is compared to a *partial remainder* W . If it fits, the quotient bit is 1, and the divisor is subtracted from the partial remainder; otherwise, the quotient bit is 0. Then the partial remainder is shifted left, and the process is repeated to obtain the next less significant quotient bit.

Using this simple algorithm, the quotient is computed one bit at a time starting with the most significant bit. For convenience and consistency with floating-point division covered in Section 16.2.3, we use big-endian notation, where bit 0 is the most significant bit and bit $N-1$ is the least significant. In other words, the quotient and dividend, Q and X , are written as follows:

$$Q = \sum_{i=0}^{N-1} q_i 2^{N-1-i} \quad X = \sum_{i=0}^{N-1} x_i 2^{N-1-i} \tag{15.3}$$

At the end of this process, Q contains the N -bit quotient, and W contains the N -bit remainder. This algorithm is known as *radix-2 restoring unsigned integer division* because it produces 1 bit per cycle and discards W' when negative, restoring W to its value before the subtraction. Fig. 15.15 shows long division compared to the radix-2 restoring division algorithm for unsigned numbers using the example: $1101/0010 = 0110 \text{ R } 0001$ ($13/2 = 6 \text{ R } 1$).

Fig. 15.16 shows a clever hardware implementation using only a $2N$ -bit shift register, an N -bit subtractor, and some multiplexers. The left shift units are just wires and thus are free. The left path is initialized with a partial remainder of 0, and the right side is initialized with X at the start of a

$W = 0$
for $i = 0$ to $N-1$
 $W = \{W \ll 1, x_i\}$
 $W' = W - D$
 if $(W' \geq 0)$ $q_i = 1, W = W'$
 else $q_i = 0$

Fig. 15.14 Radix-2 restoring unsigned integer division algorithm.

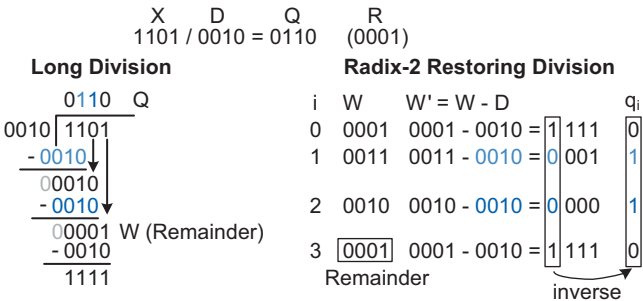


Fig. 15.15 Long division compared to radix-2 restoring unsigned integer division.

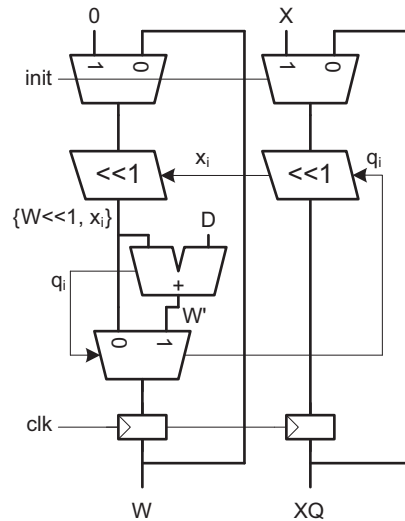


Fig. 15.16 Radix-2 restoring integer divider.

division. On each step, the partial remainder is shifted left by 1, and the most significant remaining bit of X is shifted into the bottom. The subtractor computes W' and its carry out indicates the comparison, which determines the quotient bit q_i . q_i is shifted into the vacancy at the least significant bit of X . After N steps, all the bits of X have been shifted out of the right register and replaced with the bits of Q . The left register holds the final remainder $REM = W$.

Example 15.1 RADIX-2 RESTORING DIVISION

Compute the quotient and remainder for $X/D = 1101/0010 = 13/2$ using radix-2 restoring division.

Solution: Table 15.2 shows the division steps. The black and blue XQ register shows how the black bits of X are gradually shifted out and replaced by blue bits of Q . At the end of the division, we see the W register holds the remainder 0001 and the XQ register holds the quotient 0110.

Arithmetic functional unit block diagrams traditionally flow from top to bottom, in contrast with our pipeline diagrams that flow from left to right.

The slowest step in Fig. 15.16 is the subtraction. Recall that subtraction involves inverting an input and adding, so the delay of the subtractor is essentially that of a CPA.

Table 15.2 Radix-2 restoring unsigned integer division example.

i	$W = \{W \ll 1, x_i\}$	$W' = W - D$	q_i	W Output	XQ Output
init				0000	1101
0	0001	1111	0	0001	1010
1	0011	0001	1	0001	0101
2	0010	0000	1	0000	1011
3	0001	1111	0	0001	0110

Table 15.3 Signed division.

X	D	Q	REM
13	2	6	1
13	-2	-6	1
-13	2	-6	-1
-13	-2	6	-1

15.1.4.2 Signed Integer Division

Recalling that the definition of Q and REM are $X = Q \times D + REM$ and that REM has the same sign as X , Table 15.3 shows examples of signed division. We see that Q is negative when X and D have different signs, and REM is negative when X is negative.

Signed integer division can be performed by taking the absolute value of the inputs and then performing unsigned integer division to compute unsigned Q and REM . Afterward, if the signs of X and D differ, negate Q . If the sign of X is negative, negate REM .

15.1.4.3 Other Integer Division Optimization

If the cycle time permits, the divider may duplicate the shifter, subtractor, and restoring mux to produce more than one bit per cycle.

The quotient of integer division is often a small integer. Some *early termination* dividers, commonly used in ARM and RISC-V microcontrollers, are optimized to detect this case and finish after a variable number of cycles [Kim24].

Recurrence techniques for floating-point dividers can also be used to speed up integer dividers and potentially produce more bits per cycle, as described in Section F.2.

15.2 RISC-V PRACTICES

As shown in Table I.5 (see the last pages of the book), RV32M has multiply, divide, and remainder instructions that operate on the full XLEN-bit words, treated as signed or unsigned values.

RV64M adds word operations `mulw`, `divw`, `divuw`, `remw`, `remuw` that act on the lower 32-bit word of each operand, which is sign-extended to fill 64-bit registers.

15.2.1 Signed and Unsigned Operations

As discussed in Section 15.1.3, multiplication of two N -bit numbers produces a $2N$ -bit result, with the least significant N bits being the same independent of the operands' sign. These XLEN least significant bits are produced by the `mul` instruction. `mulhu`, `mulhsu`, and `mulh` compute the most significant XLEN bits for each combination of unsigned $\times$ unsigned, signed $\times$ unsigned, and signed $\times$ signed.

Similarly, division and remainder instructions have different flavors for signed and unsigned.

15.2.2 Division Special Cases

RISC-V defines results for integer division by zero and overflow. In RISC-V, division by 0 ($rs2 = 0$) produces a result of $rd =$ all 1s. Remainder by 0 ($rs2 = 0$) produces a result of $rd = rs1$. For example, `0x12345678/0x00000000` produces a quotient of `0xFFFFFFFF` and a remainder of `0x12345678`.

C and related languages prohibit mixing signed and unsigned variables in arithmetic statements; the signed input is automatically converted to unsigned. Hence, `div` and `divu` are sufficient; there is no `divsu`. However, `mulsu` is used to write multiword signed multiplication code to multiply the most significant word of one input (containing the sign bit) by the other unsigned words of the other input.

Signed division of the largest negative number (e.g., 0x80000000) by -1 overflows because the result is larger than the largest positive number (e.g., 0x7FFFFFFF). RISC-V defines the result to be the largest negative number. For example, signed division of 0x80000000/0xFFFFFFFF produces a quotient of 0x80000000 and remainder 0x00000000.

RISC-V does not trigger hardware exceptions for either of these cases because no arithmetic instructions cause exceptions. The programmer or compiler can test for division by 0 with `beq rs2, zero, div0_error_handler` before performing the divide.

15.2.3 Software Arithmetic Implementations

When the M extension is not supported, RISC-V performs multiplication and division in software. For example, [Code Example 15.1](#) shows how `a * b` for RV32I compiles into a call to the `libgcc __mulsi3` function. The function may take up to 195 instructions, so it is far slower than hardware multiplication.

Code Example 15.1 libgcc RISC-V INTEGER MULTIPLICATION LIBRARY FUNCTION

C Code

```
int mul(int a, int b) {
    return a * b;
}

/* Equivalent C code
int mul(int a, int b) {
    int b0, result = 0;
    do {
        b0 = b & 0x1;
        if (b0) result += a;

        b = b >> 1; // Logical shift
        a = a << 1;
    } while (b);
    return result;
}
*/
```

Assembly Language

```
mul(int, int):
    addi    sp,sp,-16
    sw      ra,12(sp) # Save return address
    call    __mulsi3  # Call mulsi3 library
    lw      ra,12(sp) # Restore return address
    addi    sp,sp,16
    ret

__mulsi3:
    mv      a2, a0     # Save a in a2
    li      a0, 0      # Init result = 0 in a0
.L1:
    andi    a3, a1, 1   # Get lsb of b
    beqz    a3, .L2     # If lsb = 0, skip
    add     a0, a0, a2   # Else result += a
.L2:
    srli    a1, a1, 1   # b = b >> 1
    slli    a2, a2, 1   # a = a << 1
    bnez    a1, .L1     # Repeat while b != 0
    ret
```


See also the libgcc division library at github.com/gcc-mirror/gcc/blob/master/libgcc/config/riscv/div.S

15.2.4 Zmmul

RISC-V defines an optional extension Zmmul, which contains all of the multiply instructions but none of the divide or remainder instructions. It is intended for embedded processors that need fast multiplication but rarely use division. Zmmul saves hardware and simplifies the pipeline control by eliminating the multicycle divider. The M extension implies Zmmul.

15.3 TEST PLAN

The M-extension instructions are tested with the riscv-arch-test suite. The configuration files to generate these random tests are similar to the one described in [Section 7.4](#). [Table 15.4](#) lists the number of test cases for each instruction.

Table 15.4 riscv-arch-test for M-extension.

Instruction	RV32M Tests	RV64M Tests
div	587	675
divu	722	821
divuw		820
divw		667
mul	583	680
mulh	583	673
mulhsu	645	740
mulhu	719	811
mulw		670
rem	582	662
remu	720	819
remuw		815
remw		672
Total	5,141	9,525

15.4 WALLY IMPLEMENTATION

Fig. 15.17 shows how Wally adds an optional multiply and divide unit (MDU). Multiplication takes place in the Execute and Memory stages with a two-cycle latency. Division has a long latency, which stalls the pipeline and delivers a result in the Memory stage.

When the M extension is disabled but `ZMMUL_SUPPORTED = 1` in the Wally configuration, Wally provides the multiplier but not the divider.

15.4.1 Multiplication

RISC-V must support all flavors of signed and unsigned multiplication. SystemVerilog lacks a graceful mechanism to express this. It would be possible to code the entire multiplier structurally, adding multiplexers to the partial product generators from Fig. 15.7 to produce either signed or unsigned partial products. However, this would prevent logic synthesis

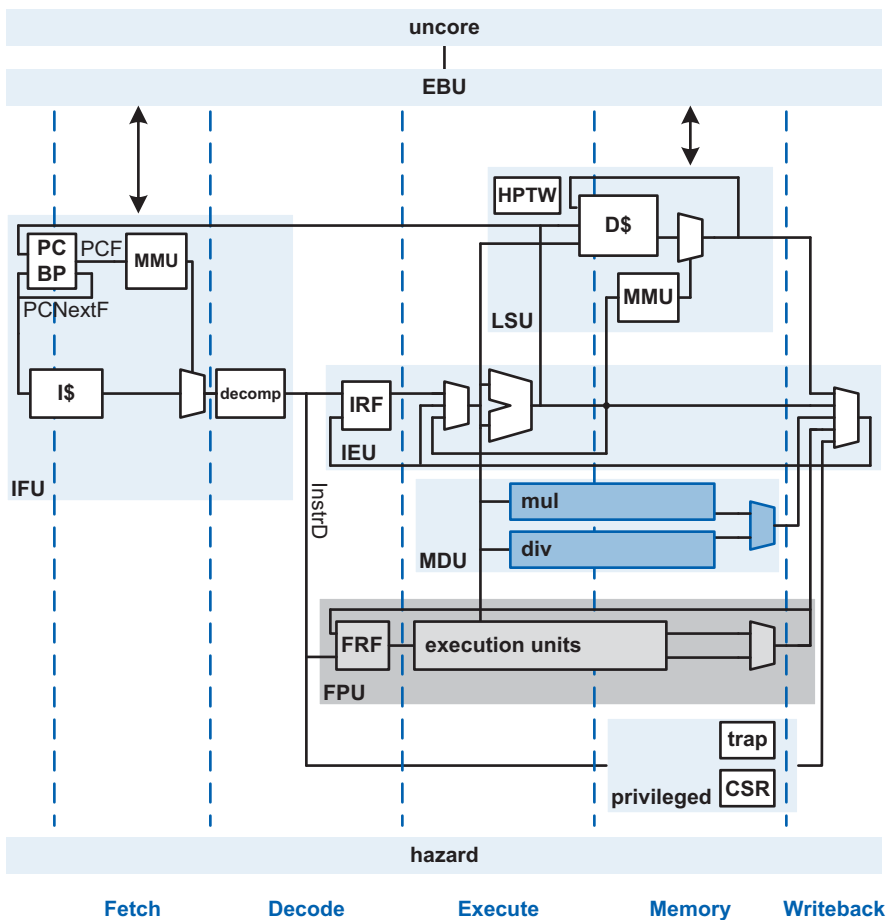


Fig. 15.17 RV{32/64}IMC high-level block diagram emphasizing the MDU.

tools from selecting their well-optimized adders or from mapping onto hardware multiplier blocks on an FPGA. Instead, our design observes that an $N \times N$ multiplier can be built from an $N-1 \times N-1$ multiplier and three other terms to handle the most significant bits. The $N-1 \times N-1$ multiplier can be expressed behaviorally to allow the logic synthesizer freedom to optimize.

Consider computing $P = X \times Y$, with X and Y being N -bit numbers and P being the $2N$ -bit product. We can write $X = \{X_{n-1}, X'\}$, with X_{n-1} being the most significant bit and $X' = X_{n-2:0}$ being the remaining $N-1$ bits. If X is treated as unsigned, its value is $X' + (X_{n-1} \times 2^{N-1})$. If X is treated as a signed two's complement number, its value is $X' - (X_{n-1} \times 2^{N-1})$. The same applies to Y .

Table 15.5 defines partial products of these terms. Table 15.6 expands the multiplication in terms of these partial products for the flavors of signed and unsigned multiplications.

The signed flavors involve some negative partial products. As in Fig. 15.5, these are produced by inverting the bits and adding 1. This results in six 1s in the dot diagram, which can be simplified by pre-adding them. Fig. 15.18 shows this for a 4×4 -bit signed $\times$ signed case. Part (a) shows the four partial products, in which the negative products are inverted, sign-extended and have a 1 added. Part (b) shows the four simplified partial products. The signed $\times$ unsigned case is similar.

In summary, each flavor of multiplication can be written as the sum of four partial products $P = P1 + P2 + P3 + P4$, as given in Table 15.7. $P1$ involves a large $N-1 \times N-1$ bit multiplication, and the other partial

Table 15.5 Partial products.

Term	Definition	Comments
P'	$X' \times Y'$	$N-1 \times N-1$ bit multiplication
PX	$Y_{n-1} \times X'$	$N-1$ bit partial product
PY	$X_{n-1} \times Y'$	$N-1$ bit partial product
P_{msb}	$X_{n-1} \times Y_{n-1}$	1-bit partial product

Table 15.6 Expansion of signed and unsigned multiplications.

Flavor	X	Y	P
mulhu	Unsigned	Unsigned	$P' + (PX + PY)2^{N-1} + P_{\text{msb}} 2^{2N-2}$
mulh	Signed	Signed	$P' + (-PX + -PY)2^{N-1} + P_{\text{msb}} 2^{2N-2}$
mulhsu	Signed	Unsigned	$P' + (PX + -PY)2^{N-1} - P_{\text{msb}} 2^{2N-2}$

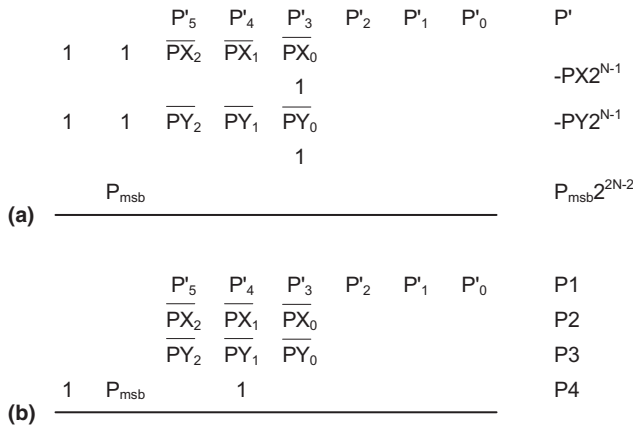


Fig. 15.18 Pre-adding 1s in dot diagram.

Table 15.7 Simplified partial products.

Flavor	$P1$	$P2$	$P3$	$P4$
mul_{hu}	P'	$(PX)2^{N-1}$	$(PY)2^{N-1}$	$P_{msb} 2^{2N-2}$
mul_h	P'	$(\overline{PX})2^{N-1}$	$(\overline{PY})2^{N-1}$	$P_{msb} 2^{2N-2} + 2^{2N-1} + 2^N$
mul_{hsu}	P'	$(PX)2^{N-1}$	$(\overline{PY})2^{N-1}$	$\overline{P_{msb}} 2^{2N-2} + 2^{2N-1} + 2^N$

products are relatively small. $P4$ accounts for the 1s from taking the two's complement, as well as the product of the sign bits. The N least significant bits are the same for all flavors, so mul can be performed by doing any of these flavors and then selecting the bottom half. Logic synthesis is good at optimizing large multiplications and the additions.

Example 15.2 SIGNED MULTIPLICATION

Use the partial product method to perform signed multiplication of $X \times Y = 1101 \times 0110$.

Solution:

$$\begin{aligned}
 X_{n-1} &= 1 & X' &= 101 \\
 Y_{n-1} &= 0 & Y' &= 110 \\
 P' &= X' \times Y' & &= 101 \times 110 = 011110 \\
 PX &= Y_{n-1} \times X' & &= 0 \times 101 = 000 \\
 PY &= X_{n-1} \times Y' & &= 1 \times 110 = 110 \\
 P_{msb} &= X_{n-1} \times Y_{n-1} & &= 1 \times 0 = 0 \\
 P1 &= P' & &= 00011110 \\
 P2 &= (PY) 2^{N-1} & &= 00111000 \\
 P3 &= (PY) 2^{N-1} & &= 00001000 \\
 P4 &= P_{msb} 2^{2N-2} + 2^{2N-1} + 2^N & &= 10010000 \\
 P &= P1 + P2 + P3 + P4 & &= 11001000 + 00011110 = 11101110 = -18
 \end{aligned}$$

In decimal, $X = -3$, $Y = 6$, $P = -18$.

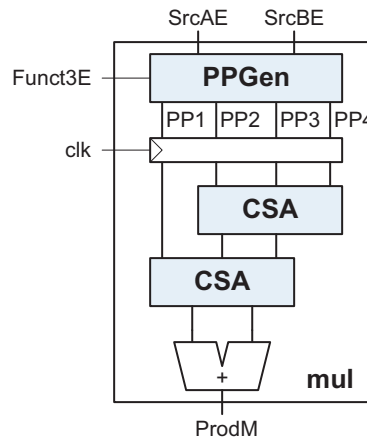


Fig. 15.19 Wally integer multiplier unit (mul).

Fig. 15.19 shows the integer multiplication unit (mul), which has a two-cycle latency. The Execute stage generates partial products P1–P4 according to Table 15.7, and the Memory stage adds these products. The slowest step is the $N-1 \times N-1$ multiplier for P1.

15.4.2 Division and Remainder

Fig. 15.20 shows the Wally integer divide/remainder unit that computes $Q = X/D$ and $REM = X \% D$. It is based on the design from Fig. 15.16. It is configurable to produce $k = \text{IDIV_BITSPERCYCLE} = 1, 2$, or 4 bits per cycle by duplicating the core logic that many times (taking care that the divider doesn't become the critical path of the processor.) The latency is given in Table 15.8; division by 0 is a special case that completes in only 2 cycles.

The inputs arrive at the top (Fig. 15.20) during the Execute stage, with X in SrcA and D in SrcB. RV64 handles W-type operations in the shaded blue box by treating the inputs as 32-bit values: D is sign-extended to fill the 64-bit DinE, and the 32-bit value of X is placed in the top half of XinE so that only 32 division steps are required to process. Negators and multiplexers take the absolute value of D and X during signed division. On the first cycle, indicated by DivStartE, W is initialized to 0, XQ is initialized to X , and the absolute value of D is registered.

The divider then iterates, with k copies of the restoring division hardware producing k bits of result on each cycle. The pipeline stalls in the Execute stage while the iterations take place. After enough iterations, W and XQ hold the absolute value of the remainder and quotient, respectively.

Even though the Execute stage registers are stalled during division, $-|D|$ is held in a register because the divisor comes from SrcBE, which comes from forwarding muxes that may change value as the Memory and Writeback stages' results change.

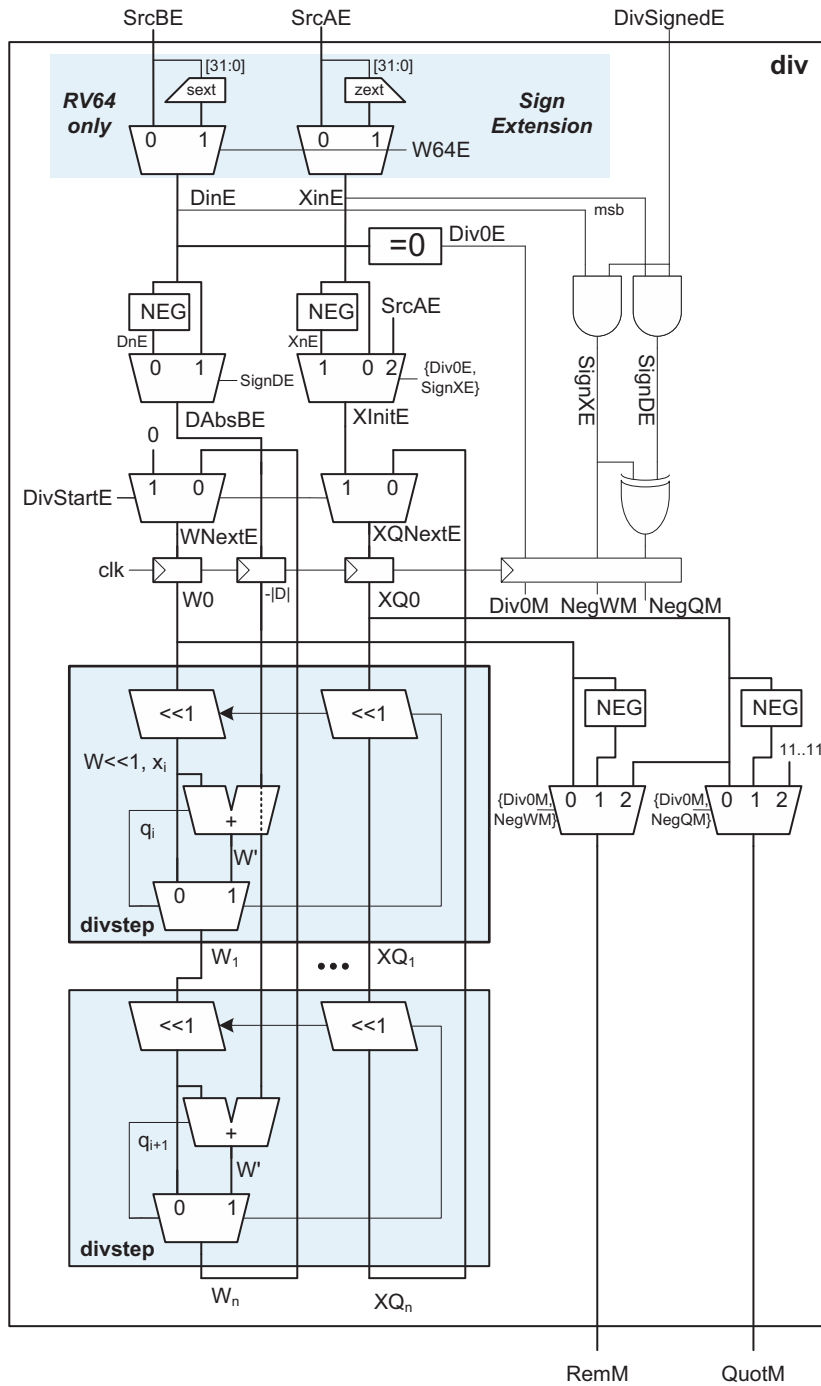


Fig. 15.20 Wally integer divide/remainder unit (div).

Table 15.8 Division and remainder latencies.

Architecture	<code>div</code> / <code>divu</code> / <code>rem</code> / <code>remu</code>	<code>divw</code> / <code>divuw</code> / <code>remw</code> / <code>remuw</code>
RV32M	$32/k + 2$	n/a
RV64M	$64/k + 2$	$32/k + 2$

Finally, the stall is released, and the divider continues to the Memory stage, in which it flips the sign of the quotient and/or remainder when necessary for signed operations. Output multiplexers also choose the special case values of X and all 1s for remainder/division by 0.

Fig. 15.21 shows the control logic and timing. A simple finite state machine in Fig. 15.21(a) manages the sequencing through Idle, Busy, and Done states to count the number of iterations needed and stall the pipeline appropriately. It begins in the Idle state. When a division operation enters the Execute stage, `DivE` asserts, which causes `DivStartE` to pulse high for a cycle, as shown in Fig. 15.21(b). Normally, the divider then enters the Busy state, which starts a step counter. `DivBusyE` is raised on the starting cycle and while the divider is Busy, causing a `StallE` Execute stage stall. After N steps, the FSM enters the Done state, and the division advances to the Memory stage. For division by 0, the FSM skips over Busy to complete in 2 cycles. The divider registers are disabled to save power (and simulation time) when idle.

Fig. 15.21(c) shows the timing for an $N = 6$ -bit divider computing $33/6 = 5$ remainder 3. In cycle 1, `XInitE` and `DabsBE` are 33 and 6, and `DivStartE` is pulsed. In cycles 2–8, the Execute stage stalls while the division occurs. `XQ` is initialized with 33, and `W` is initialized to 0. On each step, `W` is shifted left by 1, and the next bit of `X` is shifted in. `D` is subtracted and compared to find the blue quotient bit q_i and the next residual `W`. Meanwhile, `XQ` is shifted left to discard the most significant bit of `X` and prepend the blue quotient bit. In cycle 8, the results are in `QuotM` and `RemM`, the FSM enters the Done state, and `StallE` is released.

The floating-point divider uses a more sophisticated recurrence algorithm that can produce more bits in a shorter cycle time and handles early termination. Wally has an `IDIV_ON_FPU` configuration option to route integer division and remainder to the floating-point divider, as discussed in Sections 16.6.4 and F.2.9. This unified approach can have a lower latency and smaller area than separate integer and floating-point dividers, especially for configurations such as `rv32gc` where the floating-point datapath is already wider than `XLEN`.

Some dividers do early termination to halt when the residual is less than the divisor. Our formulation of division has a low hardware cost. Adding early termination would require an expensive shifter to extract the residual and quotient from a variable location in the `W` and `XQ` registers.

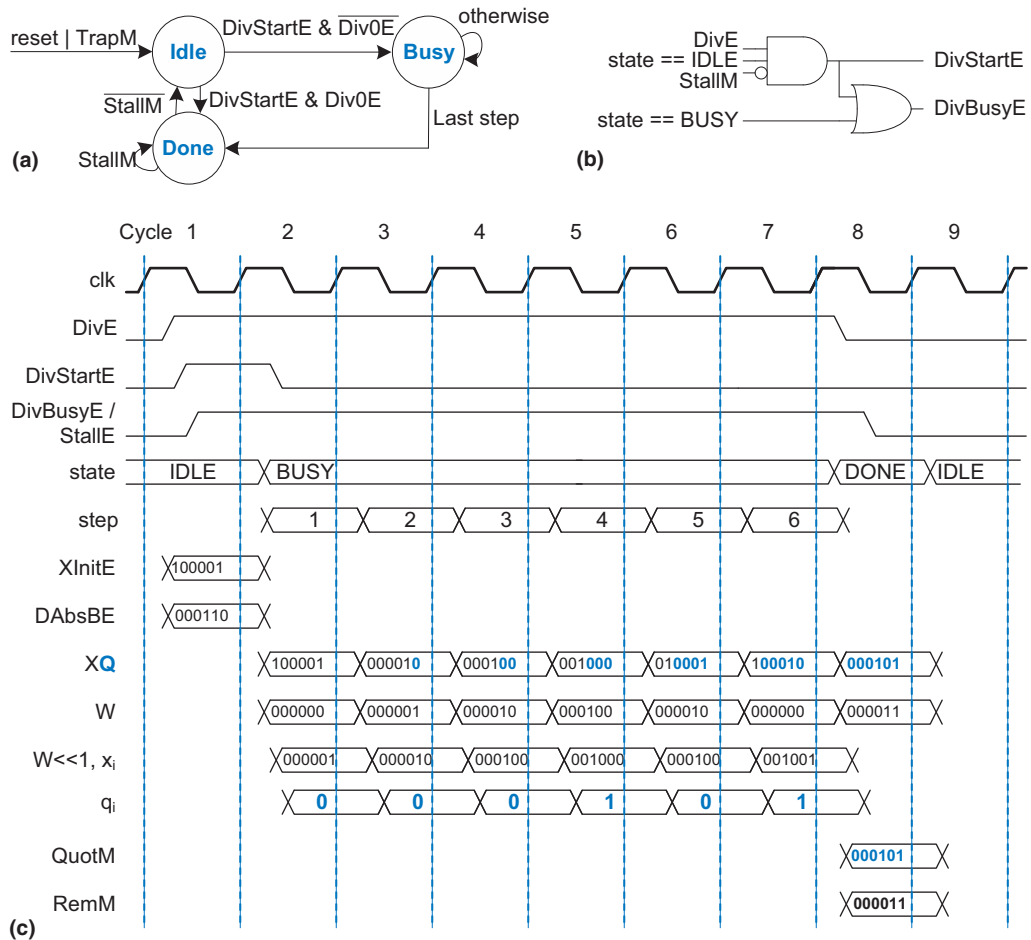


Fig. 15.21 Integer division control: (a) FSM, (b) start and busy logic, and (c) timing.

15.4.3 Multiply and Divide Unit

Fig. 15.22 shows a block diagram of the entire multiply and divide unit (MDU), omitting some control signals. The unit contains the multiplier and divider that operate in the Execute and Memory stages. These are followed by a result multiplexer that picks the quotient, remainder, or upper or lower half of the product based on the `funct3` field of the instruction. Finally, in the RV64 configurations, to handle W-type instructions (`mulw`, `divw`, `divuw`, `remw`, `remuw`), the bottom 32 bits of the result are sign-extended into the top 32. The MDU's result is captured in a Writeback stage flip-flop, which connects to the IEU and forwarding logic.

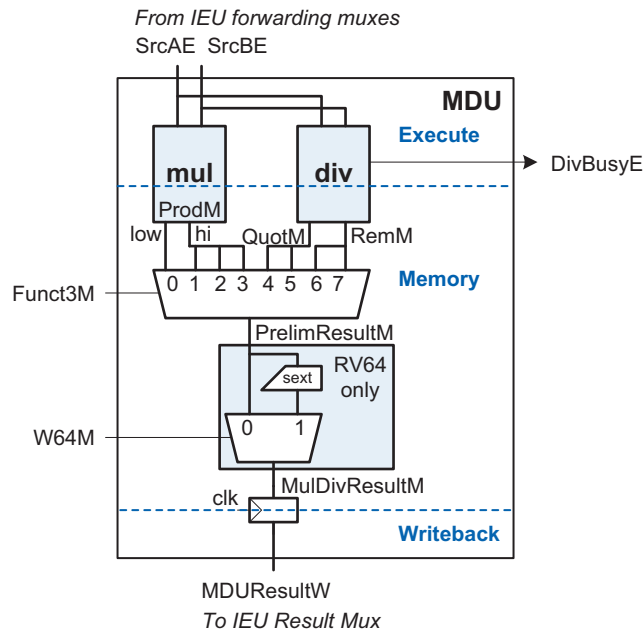


Fig. 15.22 MDU block diagram.

15.4.4 Hazard Unit

The multiplier is pipelined with a two-cycle latency (Execute + Memory) but a throughput of one instruction per cycle. The divider also has a minimum two-cycle latency in special cases (division by zero), but it usually stalls in the Execute stage for many cycles while completing the division. The multicycle division is not pipelined, so only one divide can operate at a time. However, the next MDU instruction can issue when the division enters the Memory stage.

Fig. 15.23 shows the revised hazard unit supporting the MDU, with MDU changes highlighted in blue. The MDU returns results in the Memory stage, so like other structural hazards, it stalls the Decode stage when the next instruction depends on the result of the multiply or divide ($MDUStallD = 1$).

The pipeline stalls at the Execute stage while the divider is busy. The divider FSM is reset to idle if the Execute stage is flushed ($FlushE$). However, when the IFU mispredicts the instruction class ($BPPredWrongE$) and needs to flush the instructions currently in the Fetch and Decode stages by asserting $FlushD$ and $FlushE$, it must wait for the division to complete ($\overline{DivBusyE}$).

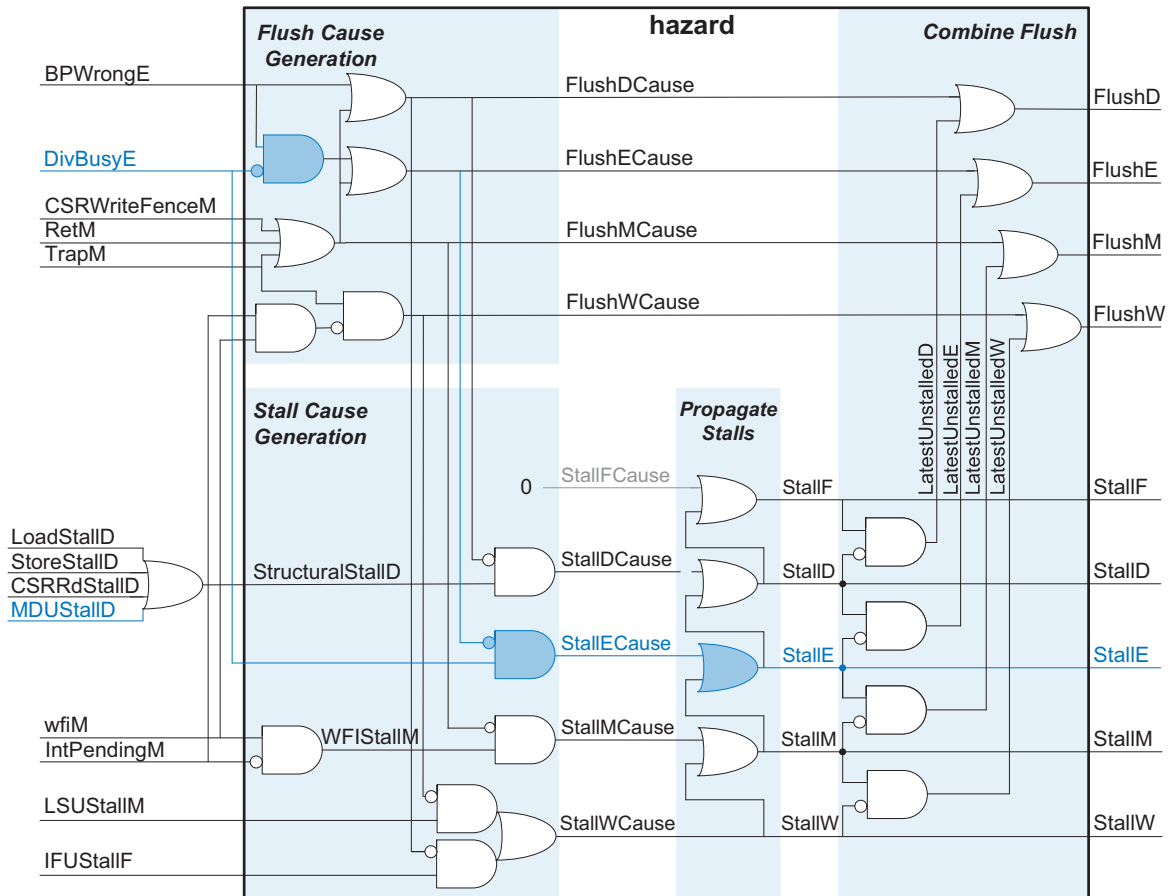


Fig. 15.23 Hazard unit, with MDU changes highlighted in blue.

15.5 SUMMARY

The RISC-V base instruction set omits multiply and divide instructions because they require substantially more hardware than other instructions and are not necessary in all applications. The RISC-V M extension defines multiply, divide, and remainder instructions. Different flavors are used for signed and unsigned operands and to obtain the upper or lower half of a 2XLEN-sized product produced by an $XLEN \times XLEN$ multiplication.

Multiplication involves forming partial products and adding them together. The field has been studied extensively. Booth encoding is used to reduce the number of partial products. Tree multipliers sum the partial products in parallel for greater speed. Logic synthesis understands these optimizations, so the designer can simply write $x * y$ and allow the

synthesis tool to produce the smallest multiplier that meets delay constraints. RISC-V defines both signed and unsigned multiplication, which differ in the handling of the most significant bit. Wally handles this by synthesizing an $\text{XLEN}-1 \times \text{XLEN}-1$ multiplier and adding three partial products that depend on whether the most significant bit is signed.

Unfortunately, combinational division is both large and slow, so synthesizing X/D generally produces unsatisfactory results. Instead, division is usually implemented sequentially, producing one or a few bits per cycle. A radix-2 restoring integer divider resembles long division in base 2. It initializes the partial remainder to 0. At each step, it shifts 1 bit from the dividend X into the partial remainder. If the divisor D is less than or equal to the partial remainder, the quotient digit is 1, and the divisor is subtracted from the partial remainder. Otherwise, the quotient digit is 0. After XLEN steps, the divider has produced an XLEN -bit quotient, and the partial remainder is the true remainder.

Wally adds the MDU to support the M extension. Multiplication is slower than other integer operations, so it is pipelined to use both the Execute and Memory stages. Hence, a hazard exists when a multiply result is used by the subsequent instruction. The hazard unit adds an `MDUStall` signal to resolve this hazard by waiting for a cycle. It also adds a `DivBusy` signal to stall the pipeline while multicycle division is taking place.

The `riscv-arch-test` suite contains test cases for the M extension instructions.

Extensions: F/D/Q/Zfh/Zfa (Floating-Point)

Floating-point numbers are similar to scientific notation in that they use a sign, fraction, and exponent to represent a number. They represent real numbers with a wide dynamic range but limited precision. Early mainframe and minicomputers used a chaotic soup of idiosyncratic vendor-specific floating-point formats [Beebe17]. The Institute for Electrical and Electronic Engineers (IEEE) standardized floating-point arithmetic in 1985, facilitating vendor-independent numerical programming [Severance98].

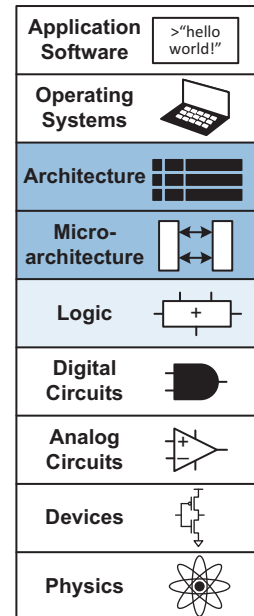
The original IEEE 754-1985 standard defines single- and double-precision floating-point number formats, including special representations for zero, infinity, not-a-number (NaN), and very small *subnormal* numbers, as well as rounding modes and result flags. Although floating-point arithmetic can be performed in software with a lengthy sequence of integer operations, most processors that run numerically intensive code support instructions that execute floating-point arithmetic in hardware.

This chapter reviews the floating-point standard, then discusses issues related to the RISC-V floating-point extensions. Although floating-point units (FPUs) have been included in commercial processors for more than 40 years, this book is the first to show a complete FPU design. Floating-point hardware has many tricky corner cases and is notoriously difficult to get right, but fortunately, there is a mature University of California, Berkeley (UCB) TestFloat suite of test cases. This chapter concludes with a detailed description of the Wally FPU, including all the computational blocks. Many implementation details are deferred to Appendix F so that this chapter flows more smoothly. Readers with deep interest in floating-point arithmetic will enjoy [Muller18] and [Ercegovac03].

16.1 PRINCIPLES

The IEEE Standard for Binary Floating-Point Arithmetic released in 1985 defines the industry standard for single- and double-precision (32- and 64-bit) floating-point representations and operations [IEEE754-1985]. The

- 16.1 [Principles](#)
- 16.2 [Floating-Point Unit Design](#)
- 16.3 [SoftFloat](#)
- 16.4 [RISC-V Practices](#)
- 16.5 [Test Plan](#)
- 16.6 [Wally Implementation](#)
- 16.7 [Summary](#)



The IEEE 754 standard is unusually difficult to read. This section summarizes the general concepts. When tracing an obscure floating-point bug, you may need to turn to the standard to sort it out.

Half-, single-, double-, quad-, and octuple-precision floating-point formats that use 16, 32, 64, 128, and 256 bits to represent a number are also called simply halves, floats, doubles, quads, and octs.

Note that some architectures, notably x86, support an 80-bit double extended precision format.

draft standard was first implemented in 1980 on the Intel 8087 math coprocessor and has been used in nearly every commercial system since the 1980s. The 2008 version of the IEEE 754 standard added fused multiply-add (FMA), which is faster and more accurate by avoiding rounding between the two steps (multiply and add), as well as half- and quad-precision (16- and 128-bit), and decimal floating-point. The 2019 version [IEEE754-2019] makes minor extensions such as octuple-precision. Most commercial processors, including the RISC-V instruction set architecture (ISA), presently support half-, single-, and double-precision IEEE 754 arithmetic. Some IBM POWER and Z-series main-frame processors support quad-precision in hardware [Lichtenau16], but quads and octs have not yet been widely needed or adopted.

16.1.1 Normalized Floating-Point Formats

Binary floating-point numbers are normally written in the following form:

$$(-1)^s \times (1.f) \times 2^{(e-bias)} \tag{16.1}$$

It may seem odd to use biased exponents rather than two’s complement representation for exponents. This approach has been widespread since the 1960s because it allows programs to compare floating-point numbers using ordinary integer comparison [Beebe17]. With a biased exponent representation, larger floating-point numbers have larger values when their bit patterns are treated as signed integers.

The *sign bit* s is 0 for positive and 1 for negative numbers. The *fraction* f and *biased exponent* e have a limited number of bits of precision, N_f and N_e . The biased exponent is treated as an unsigned number, and a bias of approximately half the maximum exponent value is subtracted to give the true exponent so that positive and negative values can be represented. Because normalized numbers always have a leading 1 before the fraction, this 1 is not stored in the binary representation. This leading 1 is sometimes called the “hidden 1.” The $(1.f)$ term is called the *normal significand*, or simply the *significand* and will be abbreviated m . m has 1 integer bit and N_f fraction bits. m is often called the *mantissa*, but this usage is slightly improper [Muller18, p. 17].

Table 16.1 summarizes the number of bits and the exponent bias in each IEEE floating-point format, along with the approximate range and decimal digits of precision. As shown in Fig. 16.1, the sign bit is in the most significant position, followed by the biased exponent, and then the fraction.

Table 16.1 IEEE 754 floating-point format.

Name	Common Name	Fraction bits N_f	Exponent bits N_e	$bias$	e_{min}	e_{max}	Range	Decimal Digits of Precision
binary16	half	10	5	15	−14	15	$\pm 10^4$	3
binary32	float/single	23	8	127	−126	127	$\pm 10^{38}$	7
binary64	double	52	11	1,023	−1,022	1,023	$\pm 10^{308}$	16
binary128	quad	112	15	16,383	−16,382	16,383	$\pm 10^{4932}$	34
binary256	octuple/oct	236	19	262,143	−262,142	262,143	$\pm 10^{78913}$	71

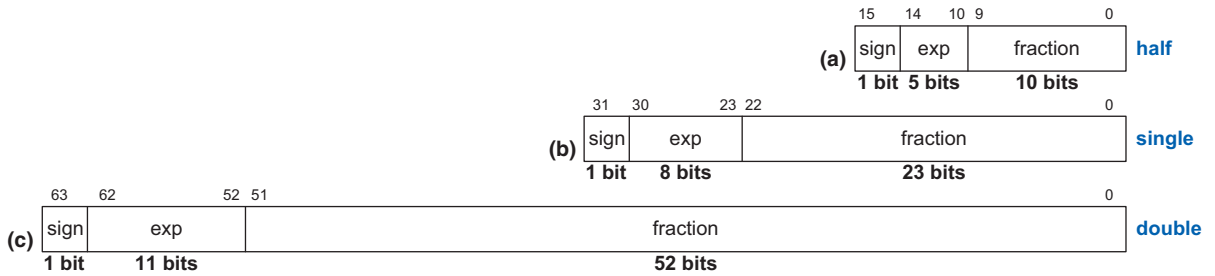


Fig. 16.1 Floating-point fields: (a) half-, (b) single-, and (c) double-precision.

Biased exponent fields of all 0s and all 1s are reserved for special cases discussed in the next section. The largest and smallest allowable true exponents for normalized numbers are called $e_{\max}$ and $e_{\min}$. $e_{\max} = \text{bias}$ and $e_{\min} = 1 - \text{bias}$. For example, $e_{\max} = 127$ and $e_{\min} = -126$, respectively, for single-precision numbers.

Example 16.1 SINGLE-PRECISION FLOATING-POINT

Write -9.625 as a single-precision floating-point number. Express your answer as an 8-digit hexadecimal value.

Solution: The sign bit is 1 because the number is negative. The magnitude, written in binary, is $1001.101 = 1.001101 \times 2^3$. Hence, the biased exponent is $3 + 127 = 130 = 10000010_2$. Dropping the implicit leading 1 yields a fraction of 001101. Fig. 16.2 shows how these fields are packed into a 32-bit number. For brevity, the number is usually expressed in hexadecimal: $0xC11A0000$.

Example 16.2 DOUBLE-PRECISION FLOATING-POINT

Determine the decimal value of the double-precision floating-point number $0x3FDB000000000000$.

Solution: Fig. 16.3 shows converting the number to binary and unpacking it into fields. The sign bit is 0, so the number is positive. The biased exponent is 1021. Subtracting a bias of 1023 gives the true exponent of -2 . The fraction is 1011. Hence, the number is $1.1011 \times 2^{-2} = 11011 \times 2^{-6} = 27/64 = 0.421875$.

Double-precision has 53 bits or $\log_{10}(2^{53}) \approx 16$ decimal digits of precision. This is sufficient to count the number of virus particles that could be lined up from the earth to the moon and back. The largest number that can be represented is approximately $2^{1024} \approx 10^{308}$.

The IEEE standard also defines 32-, 64-, and 128-bit *decimal floating-point* representations and operations that avoid rounding errors when operating on decimal values such as in financial calculations or on a pocket calculator. This chapter does not address decimal floating-point.

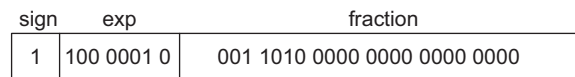


Fig. 16.2 Fields for single-precision representation of -9.625 ($0xC11A0000$).

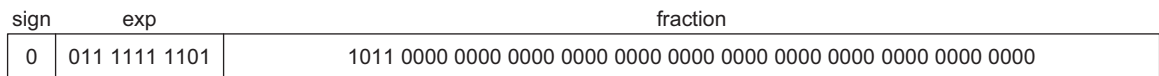


Fig. 16.3 Fields for double-precision number $0x3FDB000000000000$ (1.1011×2^{-2}).

Table 16.2 IEEE 754 special cases.

Sign	Biased Exponent	Fraction	Special Case
0	All zeros	All zeros	+0
1	All zeros	All zeros	−0
any	All zeros	Nonzero	Subnormal
0	All ones	All zeros	+∞
1	All ones	All zeros	−∞
any	All ones	Nonzero	NaN

16.1.2 Special Cases

The IEEE floating-point standard defines some special cases, encoded with the largest or smallest value in the biased exponent field, as shown in Table 16.2. All the other floating-point numbers are referred to as *normalized* and have a biased exponent that is not all zeros or ones.

16.1.2.1 Zero

Because IEEE 754 floating-point numbers technically use a sign-magnitude format, representations of both positive and negative zero exist. −0 occurs when a very small negative number is rounded to zero. Comparisons ignore the sign of zero, so +0 and −0 are considered equal. Note that +0 is conveniently written with all zeros.

16.1.2.2 Infinity

Infinity (∞ or INF) represents a number larger than any normalized number or the limiting cases of real arithmetic with operations of arbitrarily large magnitudes. Infinities can be generated by the following operations:

- ▶ Overflow: operations on finite operands producing a large result
- ▶ Division by zero
- ▶ log(0) or the exponent of a large positive number
- ▶ Most operations with infinity as an input

Positive infinity is expressed in half-, single-, and double-precision as 0x7C00, 0x7F800000, and 0x7FF0000000000000, respectively.

16.1.2.3 NaN

Not a Number (NaN) indicates a result that has no other sensible answer. For example, $0/0$ or $\sqrt{-1}$ produces NaN. NaNs are returned by the following types of operations:

- ▶ Indeterminate operations
 - Multiplication $0 \times \infty$
 - Division $\frac{0}{0}$ or $\frac{\infty}{\infty}$
 - Remainder $x \% y$ if x is ∞ or y is 0
 - Subtraction $\infty - \infty$
- ▶ Real operations with complex results, such as
 - Square root of a negative number
 - Logarithm of a negative number
 - Inverse sine or cosine of numbers outside the range $[-1, 1]$
- ▶ Most operations with at least one NaN input

The nonzero fraction in the NaN is called its *payload* and can carry implementation-specific information about the cause of the NaN. NaNs are also classified as *quiet* (*qNaN*) or *signaling* (*sNaN*). Operations on sNaNs trigger an invalid operation exception even if the result is not NaN. Thus, a program may set uninitialized variables to a sNaN value to detect if they are accidentally used. Unfortunately, the encoding of sNaNs is also implementation-specific, making them hard to use in portable code. Operations producing a NaN result, such as those listed above, deliver a qNaN.

The most significant bit of the fraction field is 1 for quiet NaNs and 0 for signaling NaNs. RISC-V instructions that generate NaN results produce the *canonical NaN*, which has the most significant bit of the fraction set and all the other fraction bits zeroed.

16.1.2.4 Subnormal

Subnormal numbers, also called *subnorms*, are very small positive or negative numbers. They fill the *underflow gap* below the smallest positive and smallest negative normalized numbers. Their true exponent is always $e_{\min}$, but they have no implicit leading 1 before the fraction. For example, the single-precision floating-point number 0x00300000 is subnormal because it has an exponent field of 0. The fraction field is 01100...00, so it represents the value $0.0110...00 \times 2^{-126} = 3 \times 2^{-129}$.

Numerical analysts find that subnormal numbers make it easier to write robust, stable numerical software [Coonen81]. For example, consider the following code to avoid division by zero:

```
if (a ≠ b) y = 1/(a - b);
```

If $(a - b)$ is smaller than the smallest normalized number and subnorms don't exist, $(a - b)$ will be truncated to 0 and cause a division by zero error despite the check. Subnormal numbers provide *gradual underflow*; loss of precision occurs slowly for tiny numbers. IEEE floating-point arithmetic will give a reasonable answer even when a and b are nearly equal.

Subnormal numbers were the most controversial part of the IEEE 754 specification because they are the most difficult to implement and add cost and delay to the FPU. Thus, they are sometimes implemented in software, which degrades performance when they occur.

Like subnormal numbers, *denormalized* numbers have a leading 0 in the significand, but unlike subnormal numbers, they have an arbitrary exponent. Subnormal numbers are denormalized numbers with a true exponent of $e_{\min}$, so their magnitudes are always less than the smallest normalized number. The original IEEE 754 standard used *denormal* to refer to such numbers, but the 2008 revision of the specification switched to the more proper term *subnormal*. The literature still has a mix of the two terms.

16.1.3 Rounding

The result of floating-point arithmetic may not be exactly represented with the precision available and must be rounded to a representable value. IEEE 754 defines five rounding modes, as listed below:

- ▶ **RNE:** Round to nearest even
- ▶ **RNTA:** Round to nearest ties away
- ▶ **RZ:** Round toward zero
- ▶ **RN:** Round toward $-\infty$
- ▶ **RP:** Round toward $+\infty$

Round to nearest even is preferred for most numerical computation and is the default rounding mode in the IEEE 754 standard. It rounds to the closest representable value. If two values are equally close (a “tie”), it rounds to the even one. This is also called *unbiased rounding* because it avoids introducing systematic errors by rounding up or down on a tie. Thus, it maintains an average zero error in computation.

Round to nearest ties away rounds to the number with the larger magnitude in the event of a tie. This is required by law for certain financial calculations, such as converting Euros. The mode is also known as *round to nearest ties to maximum magnitude* (RNM).

Round toward zero truncates the fraction, rounding positive numbers down and negative numbers up. This mode is also known as truncation.

Rounding toward positive or *negative infinity* (sometimes called *directed* rounding) rounds a number up or down, respectively. They can be used to perform *interval arithmetic* [IEEE1788.1-2017], where the true result is within a bounded interval. Specifically, a computation is repeated with both rounding modes; thus, the true answer is guaranteed to lie in an interval between an upper and lower bound. If the interval is small, the program knows that the answer is close to exact and was not corrupted by the rounding error. If the interval is large, the program can detect numerical instability, and either flag the answer as unreliable or repeat the calculation with greater precision.

Proper rounding is costly in hardware but critical for writing well-behaved numerical software.

MAXNUM is defined as the largest normalized number ($1.1111\dots111 \times 2^{e_{\max}}$). Results that overflow the range of finite numbers are rounded to either $\pm\infty$ or $\pm\text{MAXNUM}$ depending on the rounding mode. For example, round toward zero (RZ) rounds a huge positive number to **MAXNUM**, whereas RP rounds it to ∞ .

Rounding is performed by computing the exact result and then picking the appropriate representable result. The rounded result may be one of the following:

- ▶ **TRUNC**: The result with the fraction truncated to the number of bits in the floating-point representation.
- ▶ **RND**: **TRUNC** plus 1 in the least significant bit of the representable fraction. If the fraction had been all 1s, adding 1 will produce a fraction of all 0s and increment the exponent. If the exponent was $e_{\max}$ before incrementing, the result will be ∞ .
- ▶ **MAXNUM or ∞** : For results that overflow before rounding.

The 1 in the least significant bit is known as an *ulp* (*unit in the last place*, rhymes with *gulp*). Informally, for any real number x , $\text{ulp}(x)$ is the gap between the two closest finite floating-point numbers straddling x . Given two consecutive finite floating-point numbers a and b such that $|x| \in (a, b]$, $\text{ulp}(x) = b - a$. *ulp* is helpful to reason about rounding error without getting bogged down in exponent or precision. Whereas rounding to nearest produces an error no more than $\frac{1}{2}$ *ulp*, the other rounding modes produce an error of no more than 1 *ulp*.

[Muller05] explores corner cases in the definition of *ulp* related to infinities, NaN, and numbers very close to an exponent boundary.

Example 16.3 FINDING $\text{ulp}(x)$

For the half-precision floating-point number $x = 2.125$, find $\text{ulp}(x)$.

Solution: $x = 2.125 \times 2^0 = 1.0625 \times 2^1 = 1.0001000000 \times 2^1$. x is an exactly representable floating-point number, so $b = x$. Half-precision numbers have 10 fraction bits, so the previous number is $a = 1.0000111111 \times 2^1$. Hence, $\text{ulp}(2.125) = b - a = 0.0000000001 \times 2^1 = 2^{-9}$.

Computing the exact result is troublesome, especially because division and square root can have an infinite number of bits in the exact result (e.g., for $1/3$ or $\sqrt{2}$). Fortunately, rounding can be performed using only the N_f fraction bits and three additional bits: the *guard* (G), *rounding* (R), and *sticky* (T) bits. L is the least significant bit (lsb) of the N_f fraction bits of the exact result (**TRUNC**). G is the next bit of the exact result followed by the *rounding* bit R and the *sticky* bit T , which is 1 if any of the remaining bits are nonzero (i.e., T is the OR of all the remaining bits of the exact result). Together, G , R , and T are called the *rounding bits*.

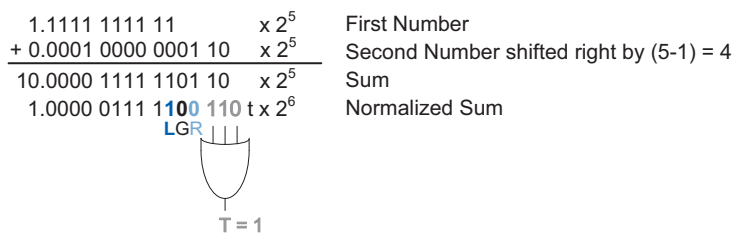


Fig. 16.4 Addition and rounding bits.

Example 16.4 HALF-PRECISION ADDITION AND ROUNDING

Compute the exact half-precision sum $1.1111\ 1111\ 11 \times 2^5 + 1.0000\ 0001\ 10 \times 2^1$. Find the L , G , R , and T bits.

Solution: The sum is computed by shifting the smaller (in this case, second) number right by $(5 - 1) = 4$ bits to make the exponents equal and then adding. Fig. 16.4 shows the exact sum is $1.0000\ 0111\ 1100\ 110 \times 2^6$. The half-precision result includes only the first 10 fraction bits. $L = 1$, $G = 0$, $R = 0$, and $T = \text{OR}(1, 1, 0) = 1$.

Dividing two numbers in the range $[1, 2)$ produces a quotient in the range $(1/2, 2)$. If this quotient is less than 1, it requires a left normalization shift by one bit to bring it back into the range $[1, 2)$. This moves G into the least significant fraction bit. The guard bit hence guards against a loss of precision. Let L' , R' , and T' be the effective lsb, rounding bit, and sticky bit accounting for this possible shift. Table 16.3 summarizes their selection. R' refers to the bit used for rounding, which may or may not be R , and T' is the OR of all bits to the right of R' . If there is no shift, L' is L , and rounding (R') is done using G . The effective sticky bit T' is the OR of all the lower bits, $R \mid T$. If a shift occurs, $L' = G$ because the guard bit moves into the lsb. This leaves $R' = R$ and $T' = T$.

The rounding decision is made based on the rounding mode, the sign, the overflow flag, and the L' , R' , and T' bits, according to Table 16.4. For example, RZ always truncates and rounds overflowing values down to MAXNUM. RP truncates negative numbers and rounds up positive numbers whenever the truncated result is not exact (R' or T' bits are 1);

Table 16.3 Rounding bits.

Normalization Shift	L'	R'	T'
No	L	G	$R \mid T$
Yes	G	R	T

Floating-point subtraction also uses a guard bit to maintain precision while normalizing with a left shift of 1 after producing a significand with a 0 integer bit. However, this chapter focuses on a more general fused multiply-add, which involves a large normalization shift before L , G , R , and T are determined. The fused multiply-add (FMA) module uses the top row of Table 16.3 (the no normalization shift entry), but L , G , R , and T remain important for underflow detection as will be described in Section F.4.7.1.

Table 16.4 Rounding.

Sign	Overflow	L'	R'	T'	RNE	RNTA	RZ	RP	RN
0	0	x	0	0	TRUNC	TRUNC	TRUNC	TRUNC	TRUNC
0	0	x	0	1	TRUNC	TRUNC	TRUNC	RND	TRUNC
0	0	0	1	0	TRUNC	RND	TRUNC	RND	TRUNC
0	0	1	1	0	RND	RND	TRUNC	RND	TRUNC
0	0	x	1	1	RND	RND	TRUNC	RND	TRUNC
0	1	x	x	x	$+\infty$	$+\infty$	+MAXNUM	$+\infty$	+MAXNUM
1	0	x	0	0	TRUNC	TRUNC	TRUNC	TRUNC	TRUNC
1	0	x	0	1	TRUNC	TRUNC	TRUNC	TRUNC	RND
1	0	0	1	0	TRUNC	RND	TRUNC	TRUNC	RND
1	0	1	1	0	RND	RND	TRUNC	TRUNC	RND
1	0	x	1	1	RND	RND	TRUNC	TRUNC	RND
1	1	x	x	x	$-\infty$	$-\infty$	-MAXNUM	-MAXNUM	$-\infty$

RN does the opposite. RNE rounds down when the R' bit is 0 and rounds up when the R' and T' bits are 1. When only the R' bit is 1, the number is exactly halfway between two representable numbers, and RNE rounds to an even value by rounding up if $L' = 1$ and rounding down if $L' = 0$. Hence, RNE rounds up for $R' \& (L' | T')$.

Example 16.5 ROUNDING MODES

Given the exact sum of $1.0000\ 0111\ 1100\ 110 \times 2^6$ from Example 16.4, what is the rounded result in the RNE and RP rounding modes?

Solution: The rounding bits are $L = 1$, $G = 0$, $R = 0$, and $T = 1$. No further normalization shift is needed, so according to Table 16.3, $L' = L = 1$, $R' = G = 0$, $T' = R | T = 1$. As shown in Table 16.4, with RNE, the result rounds down to $TRUNC = 1.0000\ 0111\ 11$. In RP, the result rounds up to $RND = 1.0000\ 1000\ 00$.

Most numerical algorithms, including matrix multiplication and filtering, involve multiplication followed by an addition: $R = X \times Y + Z$. It is faster and more accurate to compute this in one step and round at the end rather than multiply, round, add, and round again. This combined operation is called *fused multiply-add (FMA)* or *multiply-accumulate*.

Example 16.6 MULTIPLICATION AND ROUNDING MODES

Compute the exact product of two half-precision floating-point numbers $0x4010 \times 0x4420$. What is the rounded result in each of the rounding modes?

Solution: $0x4010 = 0\ 10000\ 0000010000$, with a positive sign, biased exponent of 16 (unbiased exponent = 1), and fraction of .000001, making 1.000001×2^1 . $0x4420 =$

For trigonometric functions, logarithms, and related functions, it is difficult to find the exact result required for rounding. For example, if the exact result is very close to halfway between two representable numbers, an unbounded amount of computation might be needed to determine whether it is slightly higher or lower than the boundary. This is called the *Table Maker's Dilemma* [Muller18]. If exact rounding is impractical, the function may arbitrarily pick *TRUNC* or *RND*, one of the two closest representable numbers. This is called a *faithful result* and differs from the correct result by no more than one ulp (unit in the last place).

0 10001 0000100000, with a positive sign, biased exponent of 17, and fraction of .00001, making 1.00001×2^2 . The exact product is $(1+2^{-6})(1+2^{-5})(2^1)(2^2) = (1 + 2^{-5} + 2^{-6} + 2^{-11})2^3 = 1.0000\ 1100\ 001 \times 2^3$. $L = 0$, $G = 1$, $R = 0$, $T = 0$. There is no further shift, so $L' = L = 0$, $R' = G = 1$, $T' = R \mid T = 0$. Half-precision numbers have only 10 fraction bits, so the right-most bit is not representable, and we must round down to $TRUNC = 1.0000\ 1100\ 00 \times 2^3 = 0x4830$ or round up to $RND = 1.0000\ 1100\ 01 \times 2^3 = 0x4831$. The exact result is a tie half-way between *TRUNC* and *RND*. RNE rounds to the even answer, *TRUNC*. RNTA rounds the tie up to *RND*. RZ and RN round down to *TRUNC*. RP rounds up to *RND*.

16.1.4 Flags

The IEEE 754 standard defines five types of exceptions that must be signaled by setting a status flag or taking a trap whenever they arise: Invalid, Division by Zero, Overflow, Inexact, and Underflow. Inexact, Overflow, and Underflow are never raised when Invalid or Division by Zero are raised or when any of the inputs are *NaN* or *INF*. The Overflow and Underflow flags depend on the *intermediate result* obtained by rounding the exact answer to the specified precision but with an unbounded exponent. These flags are raised in the circumstances described next, with I_e being the exponent of the intermediate result.

Invalid

- ▶ Operations on an *sNaN*
- ▶ $0 \times \infty$ or $\infty \times 0$
 - IEEE specifically mentions a case where $\infty \times 0 + qNaN$ may or may not set the Invalid flag. But for RISC-V, this operation causes the Invalid flag to be set.
- ▶ $\frac{0}{0}$ or $\frac{\infty}{\infty}$
- ▶ Remainder (i.e., $rem(x, y)$) where y is 0 or x is ∞ but neither is a *NaN*
- ▶ Square root of a negative input operand.
- ▶ Effective subtraction of two infinities (e.g., $\infty - \infty$)
- ▶ Conversion of a floating-point number to an integer format where one of the input operands is a *NaN*, ∞ , or a value that would convert to an integer outside the range of the result format under the applicable rounding attribute.
- ▶ Comparison ($<$, $\leq$) of *NaN*.
- ▶ The operation $\log(x)$ where x is *NaN*, ∞ , or 0.

- ▶ Inverse sine or cosine of a value outside the domain of $[-1, 1]$.

Division by zero

- ▶ Dividing a nonzero finite number by zero
- ▶ Computing $\log(0)$

Overflow

- ▶ If the intermediate result is larger than the largest finite number

$$Overflow = I_e > e_{\max} \quad (16.2)$$

Inexact

- ▶ If the rounded result differs from what would have occurred with unbounded precision. This may occur during overflow and underflow as well.
- ▶ $R' | T'$ indicates whether there are any nonzero results beyond the truncated result.

$$Inexact = R' | T' | Overflow \quad (16.3)$$

Underflow

- ▶ If the result is *tiny* and a loss of accuracy occurs. A result is considered tiny when it is subnormal ($|result| < 2^{e_{\min}}$). A loss of accuracy occurs when the Inexact flag is set, which indicates that the answer differs from what would have occurred with unbounded precision. The IEEE 754 specification has an ambiguity that tininess may be detected either before rounding or after rounding to an intermediate result; not all implementations set this flag in the same way. RISC-V detects tininess after rounding the exact answer to the *intermediate result* with bounded precision but unlimited exponent range. Note that underflow is signaled when a very small result is rounded to zero but not when the exact result is zero.

$$Underflow = (I_e < e_{\min}) \& Inexact \quad (16.4)$$

Note that Overflow can be set even when the answer is not infinite. For example, in half-precision, $0x7800 + 0x7800 = 1.0 \times 2^{15} + 1.0 \times 2^{15}$, which has an intermediate result of 1.0×2^{16} . This is rounded to either ∞ or $MAXNUM$, depending on the rounding mode, but in either case, the Overflow flag is set.

Example 16.7 INTERMEDIATE AND FINAL RESULTS

Consider an exact result of $1.1111\ 1111\ 110 \times 2^{-15}$ being expressed in half-precision format with 10 fraction bits and $e_{\min} = -14$, using RP rounding mode. Find the intermediate and final results.

Solution: The intermediate result is $1.1111\ 1111\ 11 \times 2^{-15}$, rounded to 10 fraction bits but with an exponent less than $e_{\min}$. The rounding is down because the blue R' bit is 0, and there is no sticky bit, T' . On the other hand, to obtain the final result, the exact result is first shifted into subnormal form with the exponent bounded to $-e_{\min}$: $0.1111\ 1111$

Some algorithms aim to avoid under/overflow. For example, [Beebe17, p. 113] shows a function that computes the length of a vector $\sqrt{x^2 + y^2}$. A direct implementation will overflow or underflow if either input is very large or very small. The function checks if either flag is set and, if so, recomputes the calculation as

$$x \sqrt{1 + \left(\frac{y}{x}\right)^2} \text{ for } x > y.$$

This alternate computation avoids unnecessary under/overflow but is more time consuming and requires additional code.

1110×2^{-14} . It is then rounded up because the blue R' bit is 1, giving $1.0000\ 0000\ 00 \times 2^{-14}$. Thus, the intermediate and final result differ.

Underflow is the most subtle of the flags. Whereas underflow is intuitively associated with subnormal results, the Underflow flag is not raised if the subnormal result is exact. Moreover, it is raised even when the final answer is zero or the smallest normalized number $2^{e_{\min}}$ if the intermediate result is tiny. Underflow warns the programmer of risk of inaccuracy related to tiny, inexact results. Some examples of underflow include:

- ▶ Some half-precision examples with $e_{\min} = -14$ illustrate corner cases of tininess and underflow.
- ▶ $0x0000 + 0x0400 = 0.0 + 0.1 \times 2^{-14} = 0.1 \times 2^{-14}$. This result is tiny but exact, so the Underflow flag is not set.
- ▶ $0x0800 \times 0x0800 = 1.0 \times 2^{-13} \times 1.0 \times 2^{-13} = 1.0 \times 2^{-26}$. If the exponent range were unbounded, this would be the properly rounded result, which is tiny but nonzero. Define *MINNUM* to be the smallest subnormal number, $0.0000\ 0000\ 01 \times 2^{e_{\min}} = 2^{-24}$. Because the exponent range is bounded and the result is less than *MINNUM*, the result must be rounded to either 0.0 or *MINNUM* depending on the rounding mode. Hence, the result is inexact. The Underflow flag is set even when the result is rounded to 0.0 because the result with an unbounded exponent is tiny and nonzero.
- ▶ $0x07FE \times 0x3400 + 0x0200 = 1.1111\ 1111\ 10 \times 2^{-14} \times 1.0 \times 2^{-2} + 0.1 \times 2^{-14} = 1.1111\ 1111\ 110 \times 2^{-15}$ (see Example 16.7). The intermediate result can be exactly represented with 10 fraction bits because the exponent range is unbounded. However, because the exponent range is actually bounded, the number cannot be represented and (in RNE) is rounded up to 1.0×2^{-14} . This result is inexact and tiny so the Underflow flag is set even though the final result is normalized.
- ▶ $0x07FF \times 0x3400 + 0x0200 = 1.1111\ 1111\ 11 \times 2^{-14} \times 1.0 \times 2^{-2} + 0.1 \times 2^{-14} = 1.1111\ 1111\ 111 \times 2^{-15}$, which cannot be exactly represented with 10 fraction bits even if the exponent range is unbounded. In RNE, the number is rounded up to 1.0×2^{-14} , so it is not tiny. This result is inexact but not tiny, so the underflow flag is not set. In RZ, the number is rounded down to $0.1111\ 1111\ 11 \times 2^{-14}$, which is tiny and inexact, so the Underflow flag is set.

16.1.5 Fixed-Point Notation

Floating-point hardware uses fixed-point arithmetic internally to operate on the significand. To keep track of the number of integer and fraction bits and whether the number is signed or unsigned, we use the Q and U

formats. $Ua.b$ indicates an unsigned fixed-point number with a integer bits and b fraction bits. $Qa.b$ indicates a two's complement signed number with a integer and b fraction bits. In this chapter, we also use Ua , Qa , and B^b_a to indicate unsigned, two's complement signed, and biased integers with a bits and a bias of b .

Example 16.8 FIXED-POINT REPRESENTATIONS

Interpret the bit pattern 10111 as U3.2, U2.3, Q2.3, U5, Q5, and B^{15}_5 .

Solution: Table 16.5 interprets the bit patterns. Remember that the most significant bit of a two's complement number has a negative weight (e.g., for an n -bit two's complement number a):

$$-a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \quad (16.5)$$

A $Ua.b$ number represents a value bounded by $[0, 2^a)$, and a $Qa.b$ number represents a value bounded by $[-2^{a-1}, 2^{a-1})$. Performing operations increases the range and the number of bits required. When adding two fixed-point numbers with a and c integer bits, the result may require $\max(a, c) + 1$ integer bits. Exactly multiplying $Ua.b \times Uc.d$ numbers produces a $U(a+c).(b+d)$ result. Exactly multiplying $Qa.b \times Qc.d$ numbers produces a $Q(a+c-1).(b+d)$ result. Observe that signed products have one fewer integer bit than unsigned products because the range of signed numbers is smaller, although the signed product may be sign-extended to $a + c$ integer bits for consistency.

Fixed-point numbers are commonly used in application-specific computing such as signal processing and machine learning where low power consumption and inexpensive massively parallel hardware are more important than precision. For example, Q1.15 format is used to represent normalized numbers between ± 1 , which is convenient because such numbers remain in the same range when multiplied. However, the product of two Q1.15 numbers is often represented as a Q1.31 number to preserve accuracy.

Table 16.5 Fixed-point interpretation examples.

Format	Interpretation	Value
U3.2	$2^2 + 2^0 + 2^{-1} + 2^{-2}$	5.750
U2.3	$2^1 + 2^{-1} + 2^{-2} + 2^{-3}$	2.875
Q2.3	$-2^1 + 2^{-1} + 2^{-2} + 2^{-3}$	-1.125
U5	$2^4 + 2^2 + 2^1 + 2^0$	23
Q5	$-2^4 + 2^2 + 2^1 + 2^0$	-9
B^{15}_5	$2^4 + 2^2 + 2^1 + 2^0 - 15$	8

Signal names in this chapter and in Appendix F are italicized instead of in code font to be consistent with the theoretical development. Unit/module names are still in code font.

The unpacking unit is also referred to as the *unpack unit* or *unpacker*. Likewise, the postprocessing unit is also referred to as the *postprocess unit* or *postprocessor*.

Handling subnormal numbers in hardware was once considered difficult, so some FPUs would trap on subnormal values and handle them in software instead. Designers have discovered hardware subnormal support is not too expensive [Schwartz05], and RISC-V doesn't support such traps, so this section focuses on FPUs that handle subnormal numbers and all special cases in hardware.

16.2 FLOATING-POINT UNIT DESIGN

A floating-point unit (FPU), as shown in Fig. 16.5, typically contains several execution units:

- ▶ **fma:** Fused multiply-add (FMA) unit for add, subtract, multiply, and FMA
- ▶ **fdivsqrt:** Divide and square root
- ▶ **fcvt:** Conversions between integer and floating-point formats
- ▶ **fcmp:** Comparisons and min/max

The FMA unit computes $W = X \times Y + Z$, where W is the result, X and Y are the *multiplicands*, and Z is the *addend*. The FMA can perform floating-point multiplies $X \times Y$ or adds $X + Z$ by tying Z to 0.0 or Y to 1.0, respectively. It can also invert input signs to compute $X \times Y - Z$, $-(X \times Y) + Z$, $-(X \times Y) - Z$, or $X - Z$.

The `fdivsqrt` unit computes $W = X \div Y$ or $W = \sqrt{X}$.

All of these execution units need to extract the significand, exponent, and sign, and identify the special cases. Thus, that logic is shared in an *unpacking* unit (`unpack`). The logic for normalization, rounding, special-case handling, and flag generation is similar across the `fma`, `fdivsqrt`, and `fcvt` modules, so it is also shared in a *postprocessing* unit (`postprocess`). The FMA requires three inputs (X , Y , and Z), but other units need only one or two inputs. The FPU produces the floating-point result and flags. It may also output an integer result (when converting from `float` to `int`) and some comparison bits.

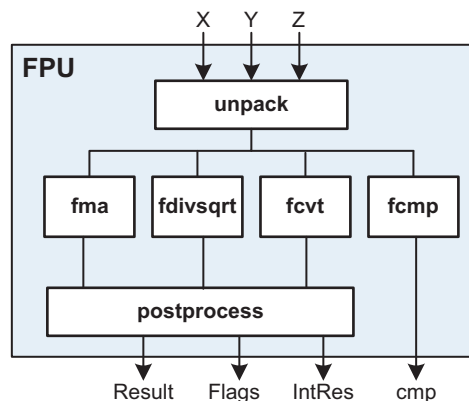


Fig. 16.5 Generic FPU block diagram.

This section describes the unpacker, postprocessing unit, and various floating-point execution units. All of these use parameterized signals to support arbitrary precision (FLEN) operands. The units have many corner cases, especially related to rounding and special case handling. [Section 16.4](#) discusses the RISC-V floating-point instruction set, and [Section 16.6](#) describes the Wally implementations in more detail, including configurable units capable of supporting multiple precisions on shared hardware.

16.2.1 Unpack Unit

The unpack unit unpacks lower-precision inputs, determines special cases, and extracts the sign, exponent, and significand fields of each input so that this bookkeeping work can be shared across the other functional units. [Fig. 16.6\(a\)](#) shows a schematic of the unpack unit comprising three channels to unpack each of the three inputs. [Fig. 16.6\(b\)](#) shows how the unit unpacks a single channel.

The Wally FPU is located in the Execute stage and uses a two-stage pipeline to unpack inputs, execute the operations, and postprocess, including normalization, rounding, and special cases. The FPU then writes the result to the register file in the Memory stage. The FPU can limit the Wally cycle time. Most commercial FPUs use a 3-7 stage floating-point pipeline to improve cycle time [Muller18].

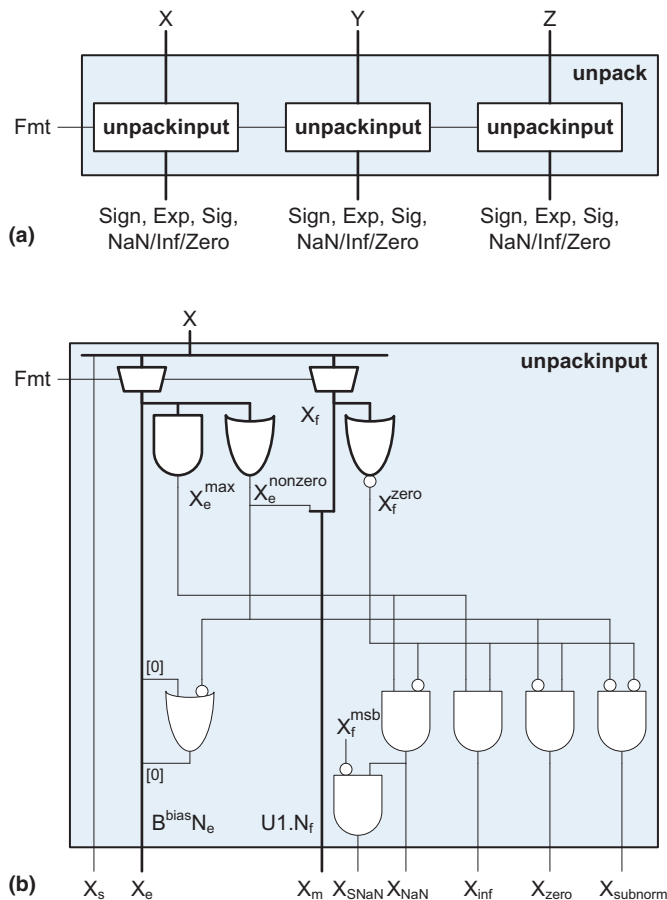


Fig. 16.6 The unpack unit: (a) top level and (b) unpack hardware for a single channel.

Table 16.6 Unpacking signals.

Signal	Format	Quad	Double	Single	Half	Comments
Inputs						
X	FLEN	X _{127:0}	X _{63:0}	X _{31:0}	X _{15:0}	Input
Intermediate Signals						
X _e ^{max}	Boolean	AND(X _e)				Exponent is max(INF, NaN)
X _e ^{nonzero}	Boolean	OR(X _e)				Exponent is nonzero
X _f	U0.N _f	X _{112:0}	X _{51:0}	X _{22:0}	X _{9:0}	Fraction part of X
X _f ^{zero}	Boolean	X _f == 0				Fraction part is zero
Outputs						
X _s	Q1	X ₁₂₇	X ₆₃	X ₃₁	X ₁₅	Sign
X _e	B ^{bias} N _e	X _{126:112}	X _{62:52}	X _{30:23}	X _{14:10}	Biased Exponent
		+ X _e ^{nonzero}				
X _m	U1.N _f	X _e ^{nonzero} , X _f				Significand
X _{subnorm}	Boolean	X _e ^{nonzero} X _f ^{zero}				Subnormal
X _{zero}	Boolean	X _e ^{nonzero} X _f ^{zero}				Zero
X _{inf}	Boolean	X _e ^{max} X _f ^{zero}				Infinity
X _{NaN}	Boolean	X _e ^{max} X _f ^{zero}				NaN
X _{sNaN}	Boolean	X _{NaN} X _f ^{msb}				Signaling NaN

Recall that sNaNs are recognized by a 0 in the most significant bit of the fraction.

Table 16.6 defines the unpacker inputs, intermediate signals, and outputs for the four supported precisions of X . Y and Z have similar signals. The sign bit X_s is always found in the most significant bit. The biased exponent X_e is from the next group of bits, and the significand (also called mantissa) X_m is from the lower bits. If the exponent is nonzero (i.e., the number is not subnormal or zero), then the unpacker prepends a leading 1 to the fraction. Otherwise, it prepends a leading 0 and then either handles it as a zero or increments the biased exponent to one to represent a subnorm. The unpacker also checks whether the exponent and fraction are all zeros or all ones to determine whether the number is normalized, denormalized, zero, infinity, or a NaN (signaling or quiet NaN). Section 16.6.2 discusses unpacking lower-precision inputs for Wally.

Example 16.9 UNPACKING

Unpack the half-precision input $X = 0xC900$.

Solution: The number is 1 10010 0100000000 in binary. The sign $X_s = X_{15} = 1$. The biased exponent $X_e = X_{14:10} = 10010 = 18$, so $X_e^{\max} = 0$, and $X_e^{\text{nonzero}} = 1$. $X_f = 0100000000$, so $X_f^{\text{zero}} = 0$. The significand $X_m = 1.01$. All the other conditions are 0. Thus, we have the normalized floating-point number -1.01×2^3 .

16.2.2 Fused Multiply-Add

Many numerical algorithms such as matrix multiplication, finite impulse response filtering, or the fast Fourier transform (FFT) involve multiplications followed by adds, such as multiplying data by coefficients and adding the result to a running total. Designers discovered that fusing the multiply and add into a single instruction runs nearly as fast as either one alone and doubles the number of FLOPs (*FLoating-point OPerations per second*) that a system can perform. A fused multiply-add (FMA) unit computes $W = X \times Y + Z$, where W is the result, X and Y are the *multiplicands*, and Z is the *addend*. It can perform floating-point multiplies or adds by tying Z to 0.0 or Y to 1.0 and can also invert input signs to compute $X \times Y - Z$, $-(X \times Y) + Z$, $-(X \times Y) - Z$, or $X - Z$. Moreover, FMA is more precise than separate multiplies and adds because it avoids rounding the intermediate result. Thus, the FMA is the workhorse of an FPU.

16.2.2.1 FMA Algorithm

FMA is easiest to understand by first considering floating-point multiply (fmul) and addition (fadd). To multiply two floating-point numbers $W = X \times Y$, multiply their significands and add their exponents. The product P of two normalized floating-point significands in the range $[1, 2)$ is in the range $[1, 4)$, so it may be necessary to right-shift the product by 1 and increment the exponent to get a normalized result M . If the result cannot be exactly expressed as a floating-point number, it is rounded according to the specified rounding mode. Finally, special cases of zero, infinity, and NaNs are handled, and the flags are set. Fig. 16.7 summarizes these steps, neglecting subnormal number handling. The slowest step is the large significand multiplication.

- | | |
|--|---|
| 1. Multiply the significands of X and Y : | $P_m = X_m \times Y_m$ |
| 2. Add the exponents of X and Y : | $P_e = X_e + Y_e - \text{bias}$ |
| 3. Find the leading 1 for normalization shift: | if $(P_m \geq 2.0)$ $M_{\text{cnt}} = -1$ else $M_{\text{cnt}} = 0$ |
| 4. Shift the result to renormalize: | $M_m = P_m \ll M_{\text{cnt}}$; $M_e = P_e - M_{\text{cnt}}$ |
| 5. Round the result: | $R = \text{round}(M)$ |
| 6. Handle flags and special cases: | $W = \text{specialcase}(R, X, Y)$ |

Fig. 16.7 Simplified FMUL algorithm.

The original IEEE 754-1985 specification did not define FMA. By the mid-1990s, high-performance processors widely supported the nonstandard operation for speed and accuracy. The IEEE standard was finally revised in 2008 [IEEE 754-2008] to add FMA. Graphics processing units (GPUs) also began widely adopting IEEE 754 fused FMA operations starting in the early 2010s. The adoption of FMA operations in GPUs marked a significant advancement in their computational capabilities, aligning them more closely with the needs of scientific computing and other fields requiring high precision.

Adding two floating-point numbers $W = X + Z$ is trickier because their binary points will not line up if their exponents differ. To align the binary points, shift the significand of Z by the exponent difference, which might be positive or negative. Then add to get the sum S . The result could be huge if Z is much bigger than X . It could also be nearly 0 if X and Z are nearly equal but had opposite signs. Normalize by shifting the sum into the range $[1, 2)$ and adjusting the exponent by the shift amount. Finally, round and handle special cases and flags. Fig. 16.8 summarizes these steps. The slowest steps are the alignment and normalization shifts.

1. Determine the alignment shift count: $A_{cnt} = (X_e - Z_e)$
2. Shift the significand of Z into alignment: $A_m = Z_m \gg A_{cnt}$
3. Add the aligned significands: $S_m = X_m + A_m$
4. Find the leading 1 for normalization shift: $M_{cnt} = \# \text{ of bits to shift}$
5. Shift the result to renormalize: $M_m = S_m \ll M_{cnt}; M_e = X_e - M_{cnt}$
6. Round the result: $R = \text{round}(M)$
7. Handle flags and special cases: $W = \text{specialcase}(R, X, Z)$

Fig. 16.8 Simplified fadd algorithm.

The FMA combines the multiply and add steps, as given in Fig. 16.9. The slow significand multiplication P and alignment shift A steps take place concurrently. S is the sum $P + A$. M is the normalized result after S is shifted to have a single 1 to the left of the binary point. R is the result after rounding, and W is the final result, accounting for special cases. This section describes computing P , A , and S . M , R and W are computed by the postprocessing unit described in Section 16.2.6.

1. Multiply the significands of X and Y : $P_m = X_m \times Y_m$
2. Add the exponents of X and Y : $P_e = X_e + Y_e - \text{bias}$
3. Determine the alignment shift count: $A_{cnt} = (P_e - Z_e)$
4. Shift the significand of Z into alignment: $A_m = Z_m \gg A_{cnt}$
5. Add the aligned significands: $S_m = P_m + A_m$
6. Find the leading 1 for normalization shift: $M_{cnt} = \# \text{ of bits to shift}$
7. Shift the result to renormalize: $M_m = S_m \ll M_{cnt}; M_e = P_e - M_{cnt}$
8. Round the result: $R = \text{round}(M)$
9. Handle flags and special cases: $W = \text{specialcase}(R, X, Y, Z)$

Fig. 16.9 Simplified FMA algorithm.

FMA is faster than multiply followed by add because the hard step of multiplying the significands of A and B is performed in parallel with the slow step of aligning the significand of Z . Wally's FMA uses the same

hardware for multiplication alone or addition alone. Some FMAs are optimized to skip steps, delivering multiplication or addition results in fewer cycles than a full multiply-add.

The next three examples illustrate these steps, including the possibility of massive cancellation. All significands are written in binary.

Example 16.10 FMA: FUSED MULTIPLY ADD

Compute $(48)(5/16) + 28 = (1.1 \times 2^5)(1.01 \times 2^{-2}) + 1.11 \times 2^4$.

Solution:

- ▶ First multiply the significands $P_m = X_m \times Y_m = 1.1 \times 1.01 = 1.111$.
 - ▶ Add the exponents $P_e = 5 + (-2) = 3$.
 - ▶ Now add this product to Z: $1.111 \times 2^3 + 1.11 \times 2^4$. To add the significands, the exponents must be equal. Thus, align Z to be 11.1×2^3 . The alignment shift is by $A_{cnt} = (P_e - Z_e) = 3 - 4 = -1$. The aligned significand is $A_m = Z_m \gg A_{cnt} = 1.11 \gg -1 = 11.1$. Add these aligned significands: $S_m = P_m + A_m = 1.111 + 11.1 = 101.011$.
 - ▶ So, the sum is $S = 101.011 \times 2^3$. Floating-point numbers are written in normalized form (M) with a single 1 to the left of the binary point, so normalize by shifting S right by $-M_{cnt} = 2$ and increasing its exponent by 2: $M = 1.01011 \times 2^5$. The result is exact and does not require rounding. In decimal, this is $W = 43$, as expected.
-

Example 16.11 FMA WITH MASSIVE CANCELLATION

Compute $(48)(5/16) - 13 = (1.1 \times 2^5)(1.01 \times 2^{-2}) - 1.101 \times 2^3$.

Solution:

- ▶ X and Y are the same as in the last example, so $P_m = 1.111$, and $P_e = 3$.
 - ▶ The exponents are already equal, so no alignment shift is required.
 - ▶ Add these aligned significands: $S_m = 1.111 + (-1.101) = 0.010$. So, the sum is $S = 0.010 \times 2^3$.
 - ▶ Normalize by shifting S left by 2 and decreasing the exponent by 2: $M = 1.0 \times 2^1$. Again, the result is exact and does not require rounding. In decimal, this is $W = 2$, as expected. This is an example of *massive cancellation* where many leading digits become 0 when nearly equal values are subtracted.
-

Example 16.12 FMA WITH ROUNDING

Compute $(48)(5/16) + 2^{35} = (1.1 \times 2^5)(1.01 \times 2^{-2}) + 1.0 \times 2^{35}$ and round the result toward positive infinity to produce a single-precision floating-point result with 23 fraction bits. Express your answer in hexadecimal.

Solution:

- ▶ X and Y are the same as in the last example, so $P_m = 1.111$ and $P_e = 3$. $Z_e = 35$.
- ▶ The alignment shift brings Z left by $A_{cnt} = (3 - 35) = -32$ bits. This difference is greater than the 23 bits of precision of the result, so P is insignificant compared to Z and can be discarded except that the sticky bit (T') becomes 1 for rounding.
- ▶ The normalized result $M = S = Z$ is inexact and requires rounding. Because the number is positive and the sticky bit is set, we round up by 1 ulp, as given in

Table 16.4. The rounded answer is $1.00000000000000000000000000000001 \times 2^{35}$. Thus, the sign is 0, the biased exponent is $127 + 35 = 162 = 10100010$, and the fraction is $00000000000000000000000000000001$. Fig. 16.10 shows this number in floating-point format as 0x51000001.

sign	exp	fraction
0	101 0001 0	000 0000 0000 0000 0000 0001

Fig. 16.10 Fields for single-precision representation of $1.00000000000000000000000000000001 \times 2^{35}$.

16.2.2.2 FMA Implementation

Fig. 16.11 shows a hardware implementation of the FMA algorithm from Fig. 16.9. This section provides a mathematical description of each block and explains the fixed-point formats used in each signal. Whereas this

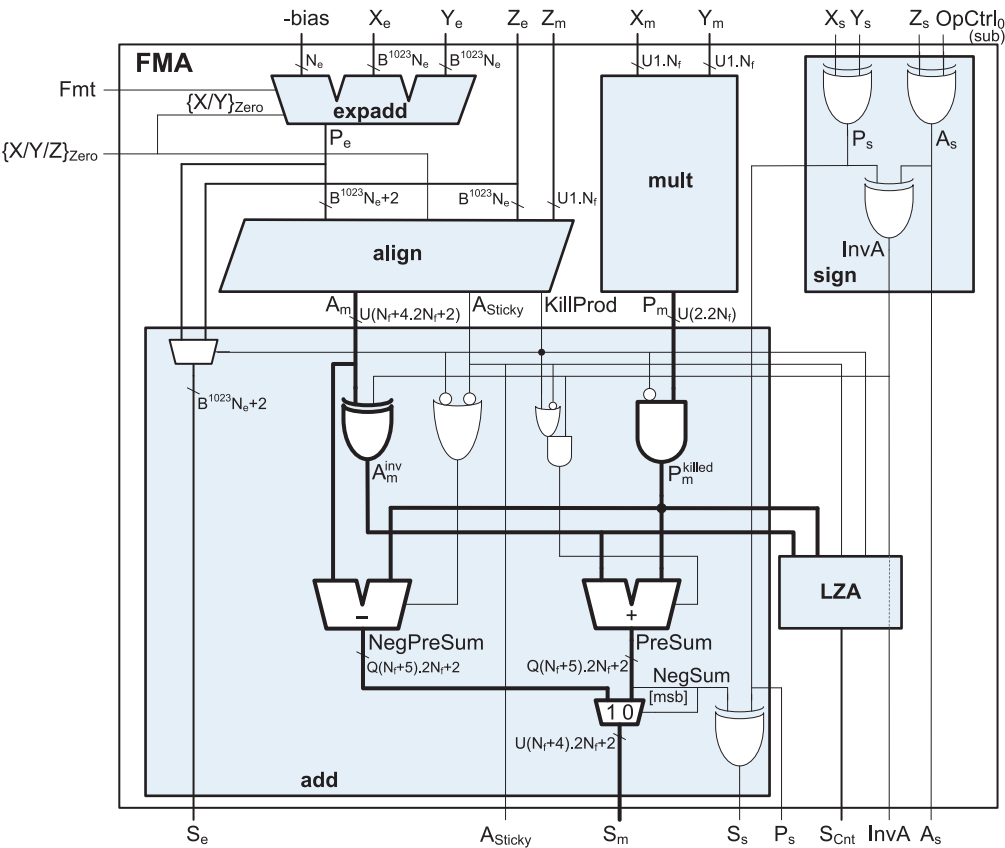


Fig. 16.11 FMA datapath.

design focuses on a simple single-path approach like the classic IBM RS/6000 design from [Montoye90], Section F.1.7 discusses multipath optimizations. See `src/fpu/fma.sv` for Wally implementation details.

The details of each block are given in Appendix F.1.

16.2.3 Division and Square Root

Recall from Section 15.1.4 that division of N -bit numbers takes a *dividend* X and a *divisor* D . It computes a *quotient* Q and *remainder* REM satisfying $X = Q \times D + REM$. Or stated another way, $X/D = Q + REM/D$. If X and D are signed, then REM has the same sign as X .

A full hardware implementation of RV64G requires computing integer division and remainder operations on 32- and 64-bit operands and computing properly rounded floating-point division and square root on 24- and 52-bit single- and double-precision operands, including subnormal numbers.

The simple division algorithm from Section 15.1.4 involves a slow carry propagate addition in each step to compute the partial remainder. A faster *digit recurrence* algorithm keeps the partial remainder in *carry-save redundant form* (see Section 15.1.2) so that each step's addition reduces to one full adder delay. However, it is impossible to exactly determine the correct quotient digit from a redundant partial remainder without converting back to nonredundant form, which eliminates the performance benefit. Instead, we guess the quotient digit based on the most significant bits of the partial remainder. The guess might be incorrect, but close. We then introduce a *redundant digit set* including negative quotient digits. If the quotient digit proves to be too high, we later correct by selecting a negative quotient digit. Radix-2 division uses a redundant digit set of $\{1, 0, -1\}$ to compute 1-bit digits. Radix-4 division is faster, generating 2-bit digits on each step, typically using a *minimally redundant digit set* of $\{2, 1, 0, -1, -2\}$. Some designers have explored higher radices, but this section focuses on radix-2 and radix-4. The redundant quotient digits are converted back to nonredundant form in an *on-the-fly* fashion.

Remarkably, square root can be computed digit by digit in much the same way as division, also keeping the intermediate results in carry-save redundant format and selecting digits of the answer from a redundant digit set. The similarities are so strong that the two operations can be performed on a shared hardware unit.

The recurrence algorithm assumes a normalized divisor in the range $[1, 2)$. To support subnormal floating-point numbers, the inputs must be preshifted into normalized form. Recurrence division is sometimes avoided for integers because of this normalization issue, but because the preshifters are needed for floating-point subnorms, they can also be used for integer division. Moreover, a variable latency design supports different operand sizes as well as early termination detection. Shorter operands are kept in the most significant bits so that the least significant register bits can be clock-gated for power

Wally's FMA operates in the Execute stage. The stage suffixes are dropped from signal names within the FMA for brevity.

Digit recurrence gets its name from a *recurrence relation*, which is an equation that defines the next term of a sequence as a function of the previous term. In digit recurrence division, each digit depends on the partial remainder, which depends on the previous digits.

This idea for fast division was invented almost concurrently by [Sweeney57], [Robertson58], and [Tocher58] and thus became known as *SRT division*. Ercegovac and Lang have studied the topic extensively [Ercegovac94], [Ercegovac03], and their more generalized techniques are now known as *digit recurrence* algorithms. Readers interested in a proof of the technique should consult their books.

savings. Hence, RISC-V is well-suited to a configurable unified divide/square root unit (`fdivsqrt`) capable of supporting all of these floating-point and integer operations. The system has the following parameters:

XLEN:	Integer register length (32, 64, or 128)
N_s :	Floating-point significand length = $N_f + 1$ (11, 24, 53, or 113, or 0 for no FP)
$N = \max(\text{XLEN}, N_s)$:	Size of largest divider operands

Fig. 16.12 outlines the basic radix- 2^r (radix-R) recurrence division algorithm. The algorithm accepts a dividend X and a divisor D . It produces the quotient Q one digit q_j at a time. It keeps the partial remainder W in carry-save redundant form $W = WS + WC$ so that addition can be performed quickly. Comparing the full divisor to the full partial remainder would be slow, so the quotient selection logic (QSLC) estimates the quotient digit based on the most significant bits of the partial remainder and divisor. It uses enough bits so that the estimate will either be correct or just slightly wrong and correctable with the digit calculation in the next step. It then chooses $-q_j D$ from a set of precomputed divisor multiples, adds it to the partial remainder with a carry-save adder (CSA), and shifts left by one digit. It repeats until all the quotient digits are known (or terminates early when the partial remainder becomes 0 and the result is exact). Fig. 16.13 shows a block diagram of the divider.

$WC_{-1} = 0, WS_{-1} = X$
 for $j = 0$ to n
 $q_j = \text{QSLC}(WC_{\text{msbs } j-1}, WS_{\text{msbs } j-1}, D_{\text{msbs}})$
 $\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} - q_j D\} \ll r$

Fig. 16.12 Recurrence division algorithm.

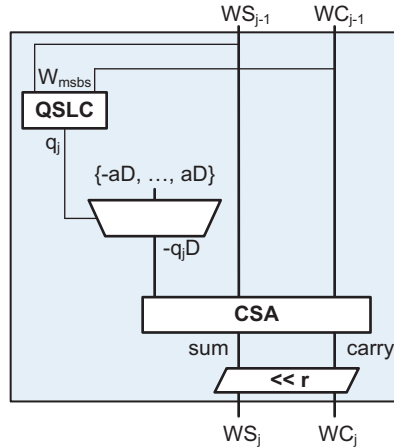
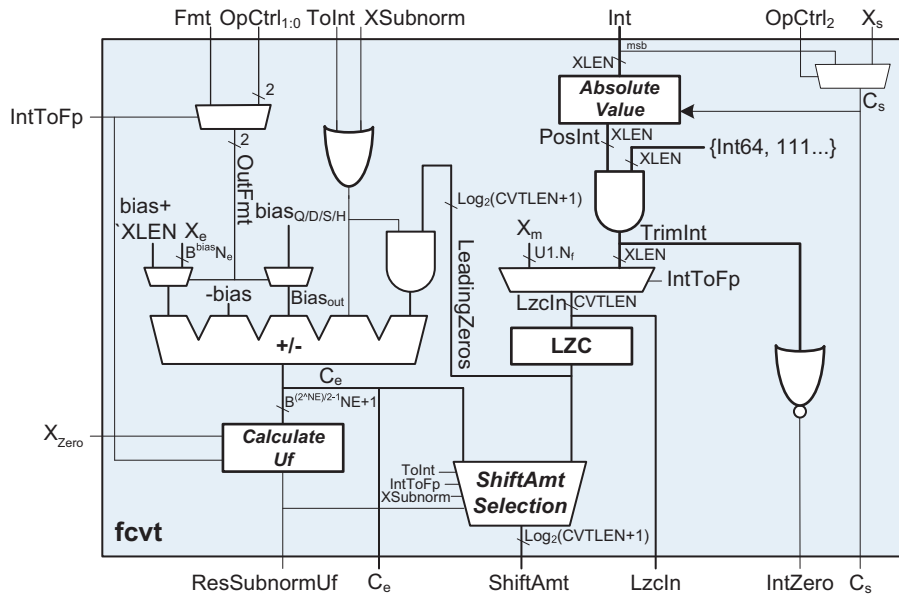


Fig. 16.13 Recurrence division hardware.

Fig. 16.14 **fcvf** unit.

Appendix F.2 describes floating-point division and square root in further detail.

16.2.4 Conversion

The floating-point convert unit (**fcvf**) handles conversion from int to floating-point, floating-point to int, and floating-point to different-precision floating-point.

Conversions from shorter to longer floating-point format involve adding the difference in biases to the exponent and placing zeros in the least significant bits of the fraction. Conversions from longer to shorter format involve subtracting the difference in biases from the exponent and rounding the fraction to the new precision. The result may overflow or underflow to create an infinite or zero answer. Both types may shift the significand to handle subnorms.

Conversions from floating-point to int involve an alignment shift, rounding, and taking the two's complement. Conversion from int to floating-point involves taking the absolute value, counting the leading zeros, normalizing the magnitude and adjusting the exponent, rounding, and setting the sign bit.

Fig. 16.14 shows the main components, which are mostly shared across types of conversions. These components include taking an absolute value to handle negative integers, counting leading zeros for integers and

subnormal floating-point numbers, calculating the normalization shift amount, and adjusting the exponent accordingly. As with the FMA, normalization, rounding, special cases, and flags are handled by the post-processing unit. Appendix F.3 describes each block in detail.

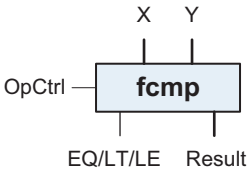


Fig. 16.15 fcmp unit.

16.2.5 Comparison

Fig. 16.15 shows a floating-point comparison unit (fcmp) that computes =, <, and ≤ comparisons. The unit also performs fmin and fmax operations. It produces Boolean outputs for comparisons and an FLEN Result output for min/max.

The unit computes EQ and LT (equal and less than) flags, then selects the appropriate output based on these flags, Zero and NaN signals, and the type of comparison.

Comparisons of NaNs are always 0 except for the inequality case (≠) which returns 1. For non-NaN cases, we compare the absolute values of the floating-point numbers by discarding the sign bit and doing unsigned integer comparison on the remaining bits. An interesting feature of floating-point numbers with biased exponents is that larger floating-point numbers are also larger when treated as integers. This even works for subnormal numbers and zeros. The exponent is in the upper bits, and numbers with larger biased exponents are strictly larger. If the exponents are the same, numbers with larger fractions are also larger.

Hence, <, ≤, and = comparison instructions (flt, fle, feq) first compute flags EQ, and LT_{abs} according to Equations 16.6 and 16.7). EQ is true if all of the bits are equal. LT_{abs} is true if the absolute value of X is less than the absolute value of Y. Next, compute the less-than flag LT according to Table 16.7 to correct for signs. Select the comparison result according to Table 16.8. Note that +0 and −0 are considered equal. Finally, for NaNs, the comparison result is always 0.

Table 16.7 Floating-point < (LT).

X_s	Y_s	$LT(X < Y)$
0	0	LT_{abs}
0	1	0
1	0	1
1	1	$\sim(LT_{abs} \mid EQ)$

Table 16.8 Floating-point comparisons.

Type	X_{zero} & Y_{zero}	Result
f _{eq}	0	EQ
f _{eq}	1	1
f _{lt}	0	LT
f _{lt}	1	0
f _{le}	0	$LT \mid EQ$
f _{le}	1	1

Table 16.9 Floating-point min/max.

Type	LT	Result
f _{min}	0	Y
f _{min}	1	X
f _{max}	0	X
f _{max}	1	Y

$$EQ = \{X_s, X_e, X_f\} == \{Y_s, Y_e, Y_f\} \quad (16.6)$$

$$LT_{abs} = \{X_e, X_f\} < \{Y_e, Y_f\} \quad (16.7)$$

`fmin` and `fmax` instructions are handled similarly based on the `LT` flag as shown in Table 16.9. Note that `-0` is considered less than `+0` for these operations. When only one input is NaN, the output is the non-NaN input. When both inputs are NaN, the output is a quiet NaN.

16.2.6 Postprocessing Unit

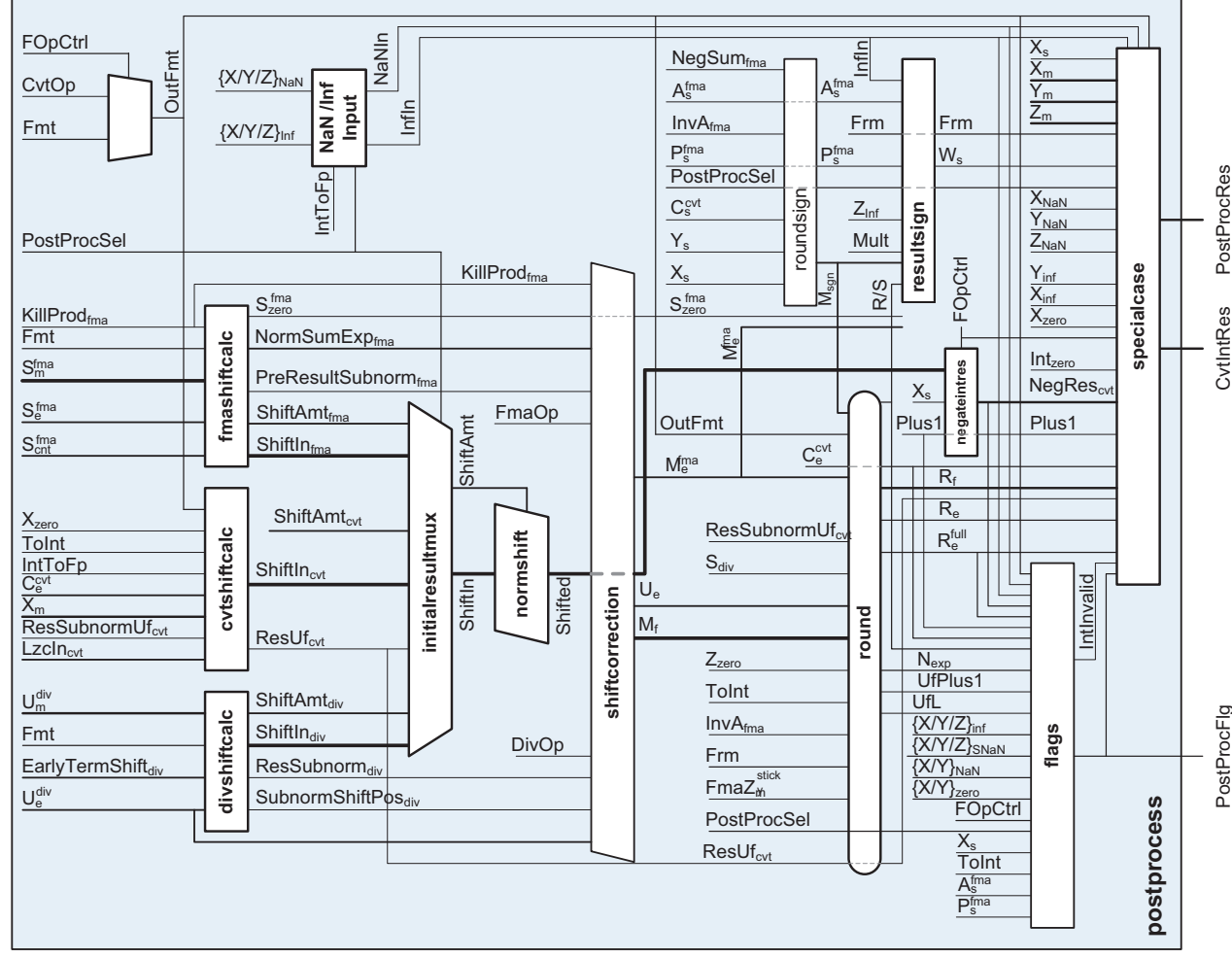
The postprocessing unit (`postprocess`) performs normalization, rounding, special-case handling, and flag generation common to all floating-point execution units including `fma`, `fdivsqrt`, and `fcvt`.

Fig. 16.16 shows a block diagram of the `postprocess` unit. The heavy lines show the primary flow of wide buses. The unit receives raw results from the execution units that need to be normalized. Like the alignment shifter, the normalization shifter only shifts in one direction for simplicity. Thus, the inputs are prealigned to the right and then left-shifted for normalization. For each execution unit, the postprocessor performs a shift calculation involving prealigning the input for the normalization shifter and computing how much to shift. The postprocessor selects the input from the active execution unit and performs a left shift to normalize, then adjusts the exponent. The normalization shifter also produces subnormal results when the exponent is small. Example 16.10 shows normalization for an FMA. An additional 1-bit left shift may be needed to compensate for an error in the LZA (leading zero anticipator) and for quotients less than 1. The rounding unit uses the normalized result, the round and sticky bits, the signs, and the rounding mode to produce a correctly rounded result. Finally, the postprocessing unit handles the floating-point special cases and flag generation. The postprocessing unit relies on various control signals, input attributes (e.g., `Inf`, `NaN`, and `Zero`), and minor intermediate results. Appendix F.4 describes each block.

16.3 SOFTFLOAT

Checking floating-point arithmetic is notoriously difficult, so we need a reliable reference model. John Hauser has developed the open-source UCB SoftFloat library [Hauser18] that defines `float_16t`, `float_32t`, `float_64t`, and `float_128t` and operations on these types, including `add`, `sub`, `mul`, `div`, `sqrt`, `mulAdd` (FMA), `remainder`, `comparisons`, and `type conversions`. For example, `f64_mulAdd` takes three `float64_t` arguments and produces a `float64_t` result. The `softfloat_roundingMode` global variable specifies the rounding mode. RISC-V uses SoftFloat as a reference model, and we use the associated TestFloat tool to generate corner cases for testing.

SoftFloat is also in the `cvw/addins/berkeley-softfloat-3` directory. See Section 16.5.2 about compiling TestFloat, which uses the SoftFloat library.



In the next section, we describe `softfloat_demo`, which shows how to call SoftFloat to convert between hexadecimal and floating-point representations, perform a floating-point operation with a specified rounding mode, and print the result and flags. In the following section, we describe `fpcalc`, a simple utility to perform floating-point calculations and visualize the numbers.

16.3.1 SoftFloat Demo

Code Example 16.1 shows the `softfloat_demo.c` program from `examples/fp/softfloat_demo`. The program uses SoftFloat to compute $1.5 \times 1.5 +$ the smallest subnormal number ($1.4013\text{e-}45$). It prints the results (rounded to 2.25) and the flags (Inexact is set because of the rounding). Compile and run it using:

```
$ make
$ ./softfloat_demo
X: 0x3fc0_0000 = 1.5
Y: 0x3fc0_0000 = 1.5
Z: 0x0000_0001 = 1.4013e-45
result = X*Y+Z: 0x4010_0000 = 2.25
Flags: Inexact 1 Underflow 0 Overflow 0 DivideZero 0 Invalid 0
```

The program initializes SoftFloat by setting the rounding mode to RNE, clearing the flags, and setting the underflow logic to detect tininess before rounding (the default for RISC-V). It then assigns values to single-precision `float32_t` variables `x`, `y`, `z`. SoftFloat declares `float32_t` to be a 32-bit number with a value field (or member) `.v` that can be written with an ordinary unsigned number, so `x.v = 0x3fc00000` gives that hexadecimal pattern to `x`, equivalent to 1.5. The program performs the FMA operation, then prints the result and the flags. It uses a union of type single-precision (`sp`) to convert the 32-bit unsigned value into a floating-point value for printing. It also uses the `softfloat_exceptionFlags` global variable to report the flags.

16.3.2 fpcalc

`examples/fp/fpcalc` contains a simple floating-point calculator that performs computations on half-, single-, or double-precision floating-point numbers in hexadecimal form and reports the results in hexadecimal, decimal, and scientific notation. This is a handy way to visualize an operation when debugging floating-point test cases. See the following examples of using the program.

This calculator is also conveniently available online as WebFloat:
s-y-wu.github.io/webfloat/

Code Example 16.1 `softfloat_demo.c`

```
// softfloat_demo.c
// David_Harris@hmc.edu 27 February 2022
//
// Demonstrate using SoftFloat for a floating-point computation, then print results.

#include <stdio.h>
#include <stdint.h>
#include "softfloat.h"
#include "softfloat_types.h"
typedef union sp {
    uint32_t v;
    float f;
} sp;

void printF32 (char *msg, float32_t f) {
    sp conv;
    int i, j;
    conv.v = f.v; // Use union to convert between hexadecimal and floating-point views
    printf("%s: ", msg); // Print out nicely
    printf("0x%04x_%04x = %g\n", (conv.v >> 16), (conv.v & 0xFFFF), conv.f);
}

void printFlags(void) {
    int NX = softfloat_exceptionFlags % 2;
    int UF = (softfloat_exceptionFlags >> 1) % 2;
    int OF = (softfloat_exceptionFlags >> 2) % 2;
    int DZ = (softfloat_exceptionFlags >> 3) % 2;
    int NV = (softfloat_exceptionFlags >> 4) % 2;
    printf ("Flags: Inexact %d Underflow %d Overflow %d DivideZero %d Invalid %d\n",
           NX, UF, OF, DZ, NV);
}

void softfloatInit(void) {
    // Rounding modes: RNE: softfloat_round_near_even
    //                  RZ:  softfloat_round_minMag
    //                  RP:  softfloat_round_max
    //                  RM:  softfloat_round_min
    softfloat_roundingMode = softfloat_round_near_even;
    softfloat_exceptionFlags = 0; // Clear exceptions
    softfloat_detectTininess = softfloat_tininess_afterRounding;
}

int main() {
    float32_t x, y, z, r;
    x.v = 0x3fc00000;
    y.v = 0x3fc00000;
    z.v = 0x00000001;

    softfloatInit();
    r = f32_mulAdd(x, y, z);
    printF32("X", x); printF32("Y", y); printF32("Z", z);
    printF32("result = X*Y+Z", r); printFlags();
}
```

Add simple floats: $2 + 1 = 3$.

Solution:

```
$ ./fpcalc 0x40000000 + 0x3f800000
X: 0x4000_0000 = 2 = +1.0 × 21: Biased Exp 128 Fract 0x0
Y: 0x3f80_0000 = 1 = +1.0 × 20: Biased Exp 127 Fract 0x0
0x40000000 + 0x3f800000: 0x4040_0000 = 3 = +1.1 × 21: Biased Exp 128 Fract 0x400000
exceptions: Inexact 0 Underflow 0 Overflow 0 DivideZero 0 Invalid 0
```

Add floats with rounding to nearest (RNE), the default rounding mode, which sets the Inexact flag: $1.0 + 1.4 \times 10^{-45} = 1.0$.

Solution:

```
$ ./fpcalc 0x3f800000 + 0x00000001  
X: 0x3f80_0000 = 1 = +1.0 × 2^0: Biased Exp 127 Fract 0x0  
Y: 0x0000_0001 = 1.4013e-45 = Subnorm: +0.000000000000000000000001 × 2^-126: Biased Exp 0 Fract 0x1  
0x3f800000 + 0x00000001: 0x3f80_0000 = 1 = +1.0 × 2^0: Biased Exp 127 Fract 0x0  
exceptions: Inexact 1 Underflow 0 Overflow 0 DivideZero 0 Invalid 0
```

Add floats with rounding to positive infinity (RP): $1.0 + 1.4 \times 10^{-45}$ is slightly more than 1.0.

Solution:[illegible]

Multiply half-precision numbers that overflow to ∞ and set the Overflow flag.

Solution:

```
$ ./fpcalc 0x7200 x 0x4600
X: 0x7200 = 12288 = +1.1 x 2^13: Biased Exp 28 Fract 0x200
Y: 0x4600 = 6 = +1.1 x 2^2: Biased Exp 17 Fract 0x200
0x7200 x 0x4600: 0x7c00 = inf = +inf: Biased Exp 31 Fract 0x0
exceptions: Inexact 1 Underflow 0 Overflow 1 DivideZero 0 Invalid 0
```


Example 16.17 FMA USING FPCALC

FMA of floats: $1.5 \times 1.625 + 1.0 = 3.4375$.

Solution:

```
$ ./fpcalc 0x3fc00000 × 0x3fd00000 + 0x3f800000
X: 0x3fc0_0000 = 1.5 = +1.1 × 2^0: Biased Exp 127 Fract 0x400000
Y: 0x3fd0_0000 = 1.625 = +1.101 × 2^0: Biased Exp 127 Fract 0x500000
Z: 0x3f80_0000 = 1 = +1.0 × 2^0: Biased Exp 127 Fract 0x0
result = X*Y+Z: 0x405c_0000 = 3.4375 = +1.10111 × 2^1: Biased Exp 128 Fract 0x5c0000
exceptions: Inexact 0 Underflow 0 Overflow 0 DivideZero 0 Invalid 0
```

Example 16.18 DIVISION USING FPCALC

Divide doubles: $\infty / 1.5 = \infty$.

Solution:

```
$ ./fpcalc 0x7ff0000000000000 / 0x3ff8000000000000
X: 0x7ff0_0000_0000_0000 = inf = +inf: Biased Exp 2047 Fract 0x0
Y: 0x3ff8_0000_0000_0000 = 1.5 = +1.1 × 2^0: Biased Exp 1023 Fract 0x80000000000000
0x7ff0000000000000 / 0x3ff8000000000000: 0x7ff0_0000_0000_0000 = inf = +inf: Biased Exp 2047
Fract 0x0
exceptions: Inexact 0 Underflow 0 Overflow 0 DivideZero 0 Invalid 0
```

16.4 RISC-V PRACTICES

As described in [Section 2.8](#), RISC-V has optional Zfh, F, D, and Q extensions for half-, single-, double-, and quad-precision floating-point instructions that are compliant with the IEEE standard. It also has optional Zfa additional floating-point instructions applicable to all precisions. These instructions are summarized on the last pages of the book. RISC-V also has a proposed L extension for decimal floating-point that is presently under development.

Recall from [Section 8.2.3.1](#) that the `status.FS` bits enable floating-point. They are typically 00 at reset, which disables the FPU and makes floating-point instructions cause illegal-instruction exceptions. Write 01 to `status.FS` before executing floating-point code.

The floating-point control and status register (CSR) specifies the rounding mode and stores flags. RISC-V has 32 floating-point registers wide enough to hold the highest precision operands supported; smaller operands are packed in the bottom bits. RISC-V also defines canonical NaNs so that NaN behavior is predictable.

16.4.1 Floating-Point Control and Status Register

RISC-V has a floating-point control and status register (`fcsr`) shown in [Fig. 16.17\(a\)](#) that defines the dynamic rounding mode and flags. Each

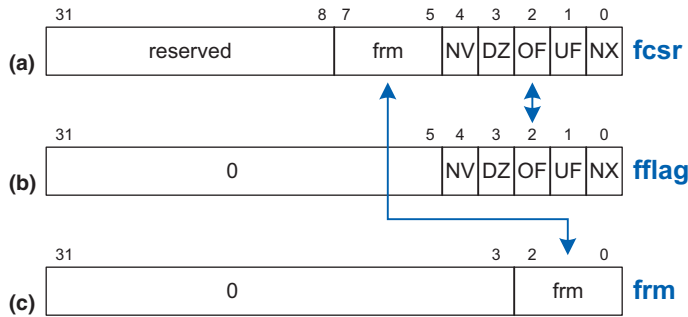


Fig. 16.17 (a) Floating-point control and status register (**fcsr**), (b) floating-point flags (**fflag**), and (c) floating-point rounding mode (**frm**).

floating-point instruction has an **rm** field to either specify the rounding mode or dynamically select the mode indicated by the **fcsr**. When the rounding mode isn't listed in the instruction (e.g., `fadd.s f0, f1, f2`), programs apply the dynamic rounding mode. Programs use the instruction field only when it is specified for a particular instruction (e.g., `fadd.s f0, f1, f2, rtz`). The rounding mode encodings are given in Table 16.10.

RISC-V has a flag register containing the five standard IEEE flags: **NV** (iNValid operation), **DZ** (Divide by Zero), **OF** (OverFlow), **UF** (UnderFlow), and **NX** (iNeXact). If an instruction has an exception condition such as divide by zero or underflow, that bit in the accrued exception flag register is set and remains set until cleared by software. Thus, a program can run efficiently with no exception checking, then test if any of the

The C11 language defines the `fesetround()` function to change rounding modes on the fly while a program is running. This effectively mandates that instruction sets support dynamic rounding modes.

Table 16.10 Floating-point instruction rounding modes.

frm	Mnemonic	Rounding
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round Toward Zero
010	RDN	Round DowN (toward $-\infty$)
011	RUP	Round UP (toward $+\infty$)
100	RMM	Round to nearest, ties to Max Magnitude
101	Reserved	
110	Reserved	
111	DYN	In instruction rm field: select based on fcsr.frm In fcsr.frm : invalid

flags are set at the end of the calculation. RISC-V does not have any floating-point traps.

Programs often need to access the rounding mode or flags separately, so RISC-V also has `frm` and `fflag` CSRs that allow accessing these bits of the `fcsr` directly without needing to mask the other fields, as shown in Fig. 16.17(b) and (c).

For the underflow flag, RISC-V detects tininess after rounding (see Section 16.1.4).

16.4.2 Registers and NaN Boxing

RISC-V has 32 floating-point registers. If the D extension is supported, the registers are all 64 bits wide, and single-precision values are held in the least significant half of the register with all 1s in the upper half. This is known as *NaN boxing* because the double-precision value is a NaN with a payload of the single-precision value in the bottom half. NaN boxing generalizes to any supported precision; when lower-precision values are represented in a register capable of higher precision, the higher-precision values are NaNs.

Example 16.19 NAN BOXING

Show how to store the half-precision number $1.0 = 0x3C00$ as a NaN-boxed quad-precision number.

Solution: The half-precision number goes in the bottom 16 bits and the upper 112 bits are filled with 1s: `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF3C00`.

16.4.3 NaNs in RISC-V

The IEEE 754 specification and default SoftFloat implementation recommend propagating the payload from one of the NaN inputs to a NaN output. In contrast, RISC-V produces a *canonical quiet NaN* (or, simply, *canonical NaN*) as the output of any floating-point operation resulting in NaN. This canonical NaN has a positive sign and all fraction bits of 0 except the most significant bit. Table 16.11 lists the canonical NaNs for various precisions.

The RISC-V designers concluded that propagating NaN payloads is not used in portable code and thus was not worth the hardware cost of a mux to choose the payload.

Table 16.11 Canonical NaNs.

Precision	Canonical NaN
Half	7E00
Single	7FC00000
Double	7FF8000000000000
Quad	7FFFF800000000000000000000000000

Table 16.12 RISC-V floating-point moves.

Instruction	Operation
<code>fmv.x.w</code>	Move from single-precision floating-point to integer register
<code>fmv.w.x</code>	Move from integer to single-precision floating-point register
<code>fmv.x.d</code>	Move from double-precision floating-point to integer register (RV64D only)
<code>fmv.d.x</code>	Move from integer to double-precision floating-point register (RV64D only)

Floating-point minimum and maximum instructions `FMIN` and `FMAX` select the non-NaN operand if one operand is a NaN.

16.4.4 Floating-Point Move Instructions

The `fmv` family of instructions moves raw bits between integer and floating-point registers without conversion, as summarized in Table 16.12. If the integer register is wider than the floating-point format, moves to floating-point take the lower bits of the integer register, and moves from floating-point sign-extend the bits to the width of the integer register.

16.4.5 Zfa Additional Floating-Point Instructions

As the RISC-V ecosystem of compilers and libraries matured, developers discovered a variety of handy floating-point instructions were absent from the initial F, D, Q, and Zfh specifications. The Zfa extension instructions are summarized in Table I.26 at the back of the book. This section describes each group of instructions.

16.4.5.1 `fli`

Loading a floating-point constant is surprisingly expensive. Loading it from memory may take many cycles to fetch from DRAM. It can take many instructions to assemble an immediate integer with adds and shifts, then convert to floating-point. The `fli` instruction loads one of 32 constants (commonly used in numerical libraries) into a floating-point register, as given in Table 16.13.

16.4.5.2 `fround` and `froundnx`

`fround` rounds a floating-point number to an integer according to the rounding mode and represents the result in floating-point. If the input is a signaling NaN, it sets the Invalid flag. `froundnx` is similar but also sets the Inexact flag if the rounded value differs from the input value. These instructions correspond to the IEEE754-2019 standard *roundToInteger*

Table 16.13 Floating-point load-immediate values and hexadecimal patterns.

rs1	Value	Decimal	f1i.h	f1i.s	f1i.d	f1i.q
0	-1.0	-1.0	BC00	BF800000	BFF00...0	BFFF00...0
1	$2^{e_{\min}}$	Minimum positive normal	0400	00800000	00100...0	000100...0
2	2^{-16}	1 / 65,536	0100	37800000	3EF00...0	3FEF00...0
3	2^{-15}	1 / 32,768	0200	38000000	3F000...0	3FF000...0
4	2^{-8}	1 / 256	1C00	3B800000	3F700...0	3FF700...0
5	2^{-7}	1 / 128	2000	3C000000	3F800...0	3FF800...0
6	2^{-4}	0.0625	2C00	3D800000	3FB00...0	3FFB00...0
7	2^{-3}	0.125	3000	3E000000	3FC00...0	3FFC00...0
8	1.00×2^{-2}	0.25	3400	3E800000	3FD00...0	3FFD00...0
9	1.01×2^{-2}	0.3125	3500	3EA00000	3FD40...0	3FFD40...0
10	1.10×2^{-2}	0.375	3600	3EC00000	3FD80...0	3FFD80...0
11	1.11×2^{-2}	0.4375	3700	3EE00000	3FDC0...0	3FFDC0...0
12	1.00×2^{-1}	0.5	3800	3F000000	3FE00...0	3FFE00...0
13	1.01×2^{-1}	0.625	3900	3F200000	3FE40...0	3FFE40...0
14	1.10×2^{-1}	0.75	3A00	3F400000	3FE80...0	3FFE80...0
15	1.11×2^{-1}	0.875	3B00	3F600000	3FEC0...0	3FEFC0...0
16	1.00×2^0	1.0	3C00	3F800000	3FF00...0	3FFF00...0
17	1.01×2^0	1.25	3D00	3FA00000	3FF40...0	3FFF40...0
18	1.10×2^0	1.5	3E00	3FC00000	3FF80...0	3FFF80...0
19	1.11×2^0	1.75	3F00	3FE00000	3FFC0...0	3FFFC0...0
20	1.00×2^1	2.0	4000	40000000	40000...0	400000...0
21	1.01×2^1	2.5	4100	40200000	40040...0	400040...0
22	1.10×2^1	3	4200	40400000	40080...0	400080...0
23	2^2	4	4400	40800000	40100...0	400100...0
24	2^3	8	4800	41000000	40200...0	400200...0
25	2^4	16	4C00	41800000	40300...0	400300...0
26	2^7	128	5800	43000000	40600...0	400600...0
27	2^8	256	5C00	43800000	40700...0	400700...0
28	2^{15}	32,768	7800	47000000	40E00...0	400E00...0
29	2^{16}	65,536	7C00	47800000-	40F00...0-	400F00...0-
30	∞		7C00	7F800000	7FF00...0	7FFF00...0
31	Canonical NaN		7E00	7FC00000	7FF80...0	7FFF80...0

and *roundToIntegralExact* operations. With suitable selection of rounding mode, it can perform the standard C library functions `trunc()`, `ceil()`, `floor()`, and `round()`.

16.4.5.3 *fmvh* and *fmvp*

These instructions are used in RV32D and RV64Q to transfer information between a floating-point register and a pair of integer registers. *fmvh* moves the upper half of a 2XLEN-sized floating-point register into an XLEN-sized integer register. It is used in conjunction with *fmv*, which moves the lower half. *fmvp* moves a pair of XLEN-sized integer registers into the lower and upper halves of a 2XLEN-sized floating-point register.

16.4.5.4 *fminm* and *fmaxm*

The *fminm* / *fmaxm* instructions are similar to *fmin* / *fmax* but produce a canonical NaN if either input is NaN (rather than if both are NaN). They correspond to the IEEE754-2019 standard *minimum/maximum* operations, while *fmin* / *fmax* corresponds to *minimumNumber/maximumNumber*.

16.4.5.5 *fltq* and *fleq*

fltq / *fleq* are similar to *flt* / *fle* comparison instructions but do not set the invalid flag when an input is a quiet NaN. They correspond to the IEEE754-2019 *compareQuiet* operations, and *flt* / *fle* correspond to *compareSignaling* operations.

16.4.5.6 *fcvtmod.w.d*

fcvtmod.w.d is an unusual instruction that accelerates handling of JavaScript Numbers. *Number* is a double-precision floating-point data type, but it is sometimes treated as a signed integer modulo 2^{32} .

fcvtmod.w.d is similar to *fcvt.w.d* but always rounds to zero. The bottom 32 bits are taken from the unbounded result and are sign-extended to XLEN and written to an integer register. Infinities and NaNs are converted to 0. The instruction sets the same flags as *fcvt.w.d*.

16.5 TEST PLAN

Floating-point verification is notoriously tricky. Wally tests the FPU using the *riscv-arch-test* suite to prove basic functionality of the pipeline and the units and also tests each unit using the robust UCB *TestFloat* suite to verify corner cases.

16.5.1 *riscv-arch-test*

The *riscv-arch-test* suite contains *rv32i_m/F*, *rv32i_m/D*, *rv64i_m/F*, and *rv64i_m/D* test suites for single- and double-precision floating-point

instructions. It is derived from the IBM Floating-Point Test-Suite for IEEE [Aharoni08]. The IBM test suite defines categories of tests identified in [Table 16.14](#). IBM uses the FPgen test generator to create inputs covering all the situations. The number of cases is approximate and sometimes varies across instructions or FLEN.

Approximately 91% of the tests use the RNE rounding mode; the remainder test the other rounding modes.

The floating-point tests are the slowest part of the regression, and the FMA tests are the biggest part of the floating-point tests. To speed up regression, the FMA tests are separated out to run in parallel with the other tests. Thus, four FMA and other floating-point test suites cover rv{32/64} and f/d, such as rv32gc_arch32d_fma and rv64gc_arch64f.

Note that TestFloat does not produce vectors for RISC-V instructions fclass, fsgnj, fsgnj{n/x}, fmax or fmin.

TestFloat has `-level 1` and the more comprehensive `-level 2` options of test generation. Level 2 generates tens or hundreds of millions of tests for 2-input operations, so we use level 2 for 1-input square root and convert operations and level 1 for all others.

The full set of TestFloat testvectors is 8 GB, which is why it is generated rather than kept in the repository.

16.5.2 TestFloat

As mentioned in [Section 16.3](#), TestFloat produces test cases to check compliance of floating-point implementations using the UCB SoftFloat library to compute expected results. Wally has a testbench to run the TestFloat tests directly on Verilog FPUs rather than by creating RISC-V programs.

TestFloat covers the instructions in [Table 16.15](#). It generates separate test vectors files for every rounding mode $rm \in \{rd, rne, ru, rz, rnm\}$ and precision $fp \in \{f16, f32, f64, f128\}$. Integer conversions also support integer precisions $ip \in \{i32, u32, i64, u64\}$.

TestFloat and SoftFloat are submodules in `$WALLY/addins`. Softfloat has a compiler flag called `SPECIALIZE_TYPE=RISCV` to support RISC-V behavior. Wally's Makefile uses this flag for RISC-V compatibility. A Makefile builds SoftFloat and TestFloat and then uses TestFloat to create floating-point testvectors. Invoke with:

```
$ cd $WALLY/tests/fp
$ make -jobs
```

It runs TestFloat for each operation, precision, and rounding mode, then converts the vectors into a format that Verilog simulators can read using `$readmemh`. It places the vectors in `$WALLY/tests/fp/vectors`.

For example, `f32_mulAdd_rne.tv` is shown in [Fig. 16.18](#). The first three 32-bit hexadecimal numbers are the three source operands. The fourth 32-bit number is the result. The next two digits are a 5-bit hexadecimal number representing flags: {Invalid, Divide by Zero, Overflow, Underflow, Inexact}.

`make` also creates test vectors for the combined integer divide/remainder and floating-point divide/square root unit in `tests/fp/combined_IF_vectors/IF_vectors`, as shown in [Fig. 16.19](#). It requires the

Table 16.14 IBM test categories.

Category	Description	Applicable Instructions	Number of Cases
B1	Floating-point basic types: all combinations of all signs of: 0, 1, min subnorm, subnorm, max subnorm, min norm, norm, max norm, INF, NaNs	fadd/fsub/fmul/fdiv/fmadd/fsqrt fclass; fsgnj/fsgnfn/fsgnjx feq/fle/flt; fmin/fmax CvtFP2Int; CvtInt2FP; fmv	33 for 1 input 580 for 2 input 13,835 for 3 input
B2	Near FP basic values	fadd/fsub/fmul/fdiv/fmadd/fsqrt	280
B3	Rounding: sign, LSB, guard, sticky	fadd/fsub/fmul/fdiv/fmadd/fsqrt	1124
B4	Overflow and near overflow	fadd/fsub/fmul/fdiv/fmadd/fsqrt	144
B5	Underflow and near underflow	fadd/fsub/fmul/fdiv/fmadd/fsqrt	224
B6	Tiny rounding intermediate tasks	fmul/fdiv/fmadd	144
B7	Sticky bit calculation	fadd/fsub/fmul/fdiv/fmadd/fsqrt	164
B8	Vicinity of rounding boundaries	fadd/fsub/fmul/fdiv/fmadd/fsqrt	1,684
B9	Special significands on inputs	fmul/fdiv/fsqrt	1,324
B10	Add: shift	fadd/fsub	78
B11	Add: shift combined with special significands	fadd/fsub	7,016
B12	Add: cancellation	fadd/fsub	54
B13	Add: cancellation and subnormal result	fadd/fsub	204
B14	FMA: shift	fma	84
B15	FMA: shift combined with special significands	fma	10,096 for half 38,129 for float 102,005 for double
B16	FMA: cancellation	fma	211
B17	FMA: cancellation and subnormal result	fma	211
B18	FMA: special events	fma	281
B19	Compare: different input field relations	feq/fle/flt; fmin/fmax	1,072
B20	Divide: trailing zeros	fdiv/fsqrt	106
B21	Divide: divide by zero exception	fdiv	680
B22	CvtFP2Int: special input exponents	CvtFP2Int	41
B23	CvtFP2Int: overflow	CvtFP2Int	46
B24	CvtFP2Int: Underflow	CvtFP2Int	106
B25	CvtInt2FP: special inputs	CvtInt2FP; fmv	33
B26	CvtInt2FP: number of significant bits	CvtInt2FP; fmv	33
B27	CvtInt2FP: NaN	CvtFP2Int; fmv	33
B28	Convert: Special Values	CvtFP2Int; fmv	33
B29	Convert: Rounding	CvtFP2Int; fmv	81

Source: riscv-isac.readthedocs.io/en/stable/code.html

Table 16.15 TestFloat test cases.

RISC-V Instruction	TestFloat Case Name	Number of Tests/Rounding Mode/Precision
fadd	<fp>_add_<rm>.tv	46,464
fsub	<fp>_sub_<rm>.tv	46,464
fmul	<fp>_mul_<rm>.tv	46,464
fdiv	<fp>_div_<rm>.tv	46,464
fsqrt	<fp>_sqrt_<rm>.tv	2,448/8,800/26,112/69,108
fmadd ^a	<fp>_mulAdd_<rm>.tv	6,133,248
feq/flt/fle ^b	<fp>_eq/le/lt_<rm>.tv	46,464
convert float to float fcvt.h/s/d/q.h/s/d/q	<fp>_to_<fp>_<rm>.tv	2,448/8,800/26,112/69,108
convert float to int fcvt.w/wu/l/lu.h/s/d/q	<fp>_to_<ip>_<rm>.tv	2,448/8,800/26,112/69,108
convert int to float fcvt.h/s/d/q.w/wu/l/lu	<ip>_to_<fp>_<rm>.tv	15,500/63,756

^afmsub, fnmadd, and fnmsub are the same as fmadd except sign handling is not tested via TestFloat.

^bComparisons do not round, so the results are the same in all rounding modes. RISC-V deviates from the default IEEE behavior of SoftFloat in four ways:

1. RISC-V NaN boxes lower-precision floating-point numbers stored in wider registers.
2. Floating-point to unsigned integer conversions of negative numbers produce all 1s in IEEE but all 0s in RISC-V.
3. Floating-point to integer conversions that overflow the positive range produce the most positive integer (0x8000...) in IEEE, but the largest positive integer (0x7FFF... signed or 0xFFFF... unsigned) in RISC-V.
4. Operations with a NaN input preferably propagate the payload to the output in IEEE but produce a canonical NaN in RISC-V.

```
8683F7FF_C07F3FFF_00000000_07839504_01
00000000_00000000_3C072C85_3C072C85_00
9EDE38F7_3E7F7F7F_DF7EFFFF_DF7EFFFF_01
...
```

Fig. 16.18 f32_mulAdd_rne.tv.

[illegible]

...

Fig. 16.19 cvw_64_div-01.tv.

compiled RISCOF test cases from `tests/riscov/work`. The format of these vectors is `operation_A_B_Y_flags_roundingMode`, with the operation code defined in Table 16.16.

Table 16.16 Divider test vector operations.

Code	Instruction	Operation
0	<code>div</code>	Integer divide
1	<code>fdiv</code>	Floating-point divide
2	<code>fsqrt</code>	Floating-point square root
3	<code>rem</code>	Integer remainder
4	<code>divu</code>	Integer unsigned division
5	<code>remu</code>	Integer unsigned remainder
6	<code>divw</code>	Integer 32-bit division
7	<code>divuw</code>	Integer unsigned 32-bit division
8	<code>remw</code>	Integer 32-bit remainder
9	<code>remuw</code>	Integer unsigned 32-bit remainder

To run the tests on an RV64 configuration that supports all four floating-point precisions, use one of the following commands, with choices of tests being `add`, `sub`, `mul`, `div`, `sqr`, `fma`, `cmp`, `cvtint`, or `cvtfp`:

```
$ wsim fdqh_rv64gc fma -t testbench_fp
$ wsim fdqh_rv64gc div -t testbench_fp -gui
$ wsim fdqh_rv64gc cvtint -t testbench_fp -sim verilator
```

These commands invoke Questa or Verilator on `$WALLY/testbench/testbench-fp.sv`, optionally with the Questa GUI. This testbench reads the TestFloat vectors from `$WALLY/tests/fp/vectors` and runs them through the unpacker and each of the units. The testbench then chooses the result and flags of the unit being tested and compares them to the expected values.

16.6 WALLY IMPLEMENTATION

Fig. 16.20 shows the changes to the `wallypipelinedcore` to support floating-point. The FPU operates in the Execute and Memory stages with

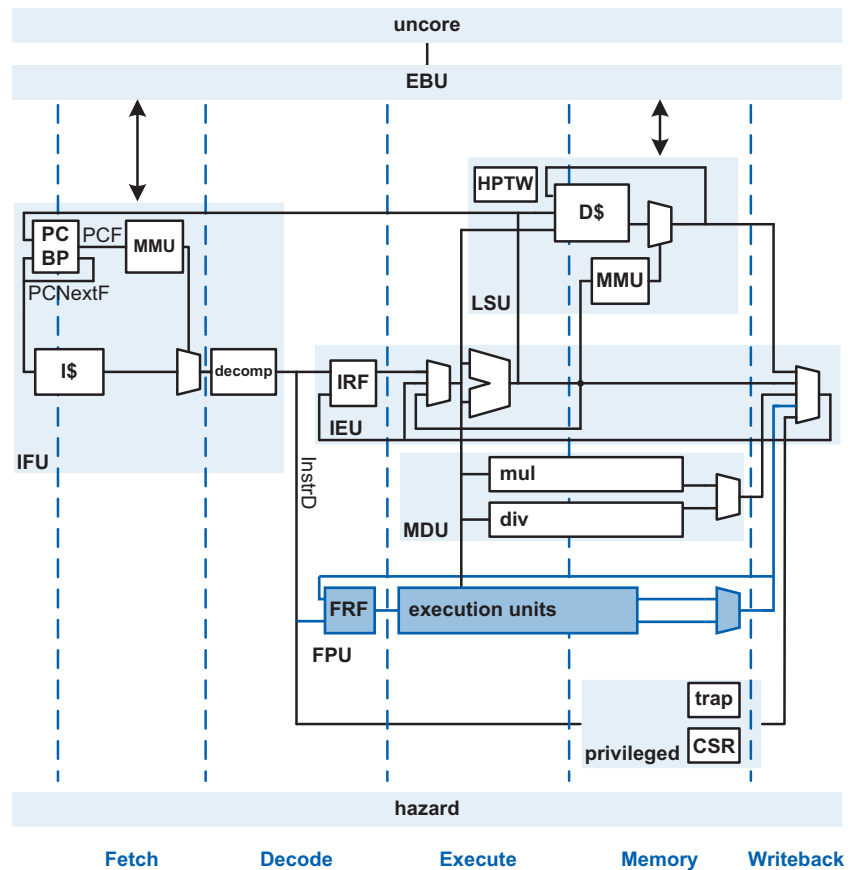


Fig. 16.20 Wally R{32/64}IFDMC high-level block diagram, emphasizing the FPU.

a two-cycle latency and single-cycle throughput, except that division and square root have longer latencies and stall the pipeline. The privileged unit includes a module to support the `fcsr` and keep track of whether the floating-point registers are dirty. The FPU connects to the LSU for loads and stores, to the IEU for moves and converts, to the privileged unit for the `fcsr`, and to the hazard unit for stalls.

16.6.1 Floating-Point Unit

Fig. 16.21 shows a block diagram of the Wally FPU. The FPU is parameterized to support half-, single-, double-, and quad-precision or any subset thereof. The highest precision supported by the FPU is called `FLEN`. For example, `FLEN = 64` for an RV32IFD configuration because it supports single- and double-precision.

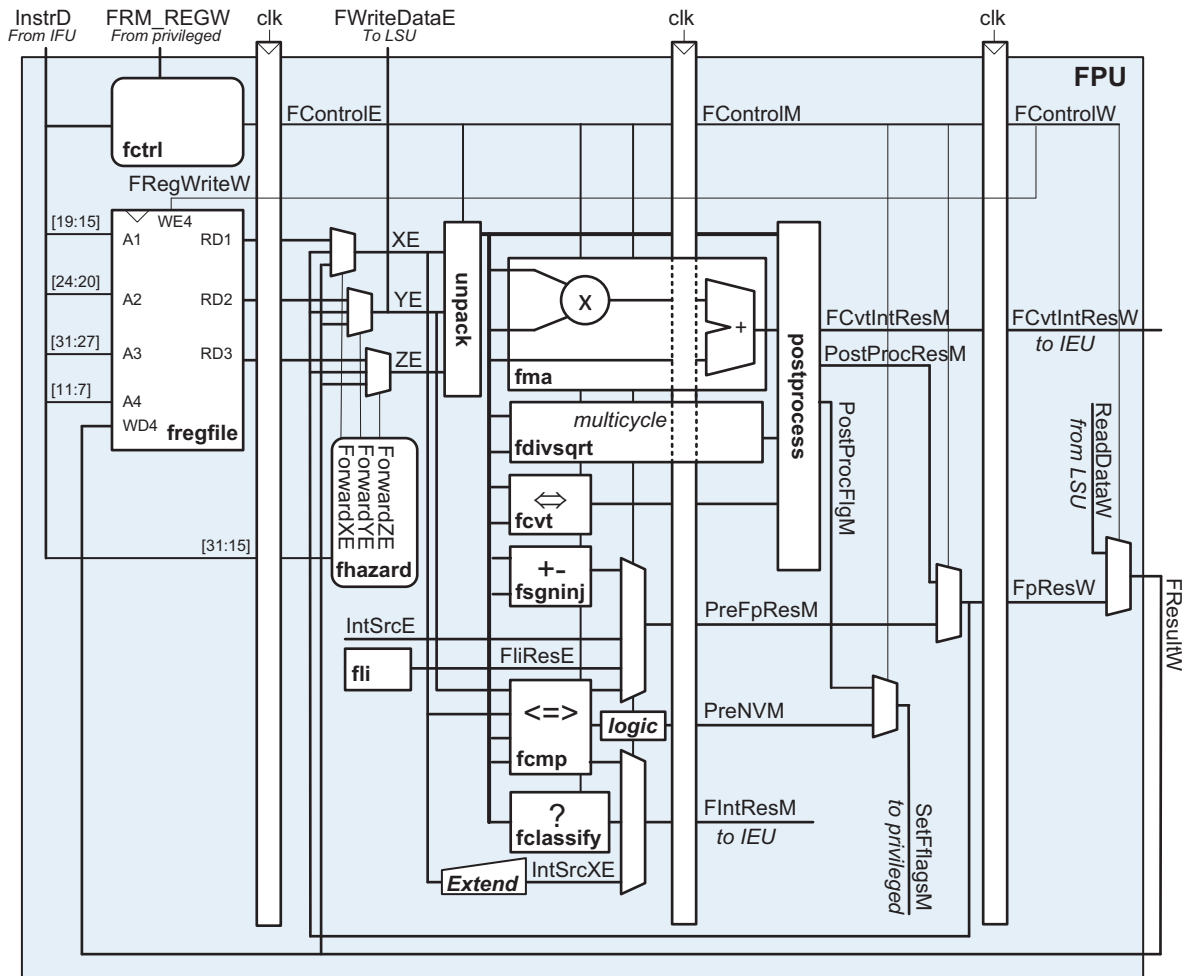


Fig. 16.21 Wally FPU.

The supported precisions depend on the F (32-bit), D (64-bit), and Q (128-bit) extensions in the *misa* (M-mode ISA register), as well as the *ZFH_SUPPORTED* (16-bit) configuration. Legal combinations are: F, FD, FDQ, HF, HFD, and HFDQ.

The *fregfile* register file holds the highest precision FLEN, with lower precisions NaN-boxed. The execution units, such as the FMA, are also sized for the highest supported precision but can handle lower precisions. The unpacking unit (*unpack*) extends lower-precision operands to FLEN. The postprocessing unit (*postprocess*) rounds to the target precision.

The optional RISC-V *Zfhmin* extension provides only the following half-precision instructions: *flh*, *fsh*, *fmv.x.h*, *fmv.h.x*, *fcvt.*.h*, and *fcvt.h.**. It is intended for systems that use half-precision as a storage format and perform computations at higher precision. Wally presently optionally supports *Zfh* but not *Zfhmin*.

The FMA handles `fadd`, `fsub`, and `fmul` as well as `fmadd`, `fmsub`, `fnmadd`, and `fnmsub`.

The `fdivsqrt` unit stalls the pipeline for many cycles. It uses the Execute stage for preprocessing, then performs many recurrence cycles in the Execute stage before delivering the result for postprocessing, which occurs in the Memory stage. For simplicity, only the Execute portion is shown in Fig. 16.21.

The FPU also receives `SrcAE` from the IEU for integer-to-floating-point conversion and delivers `FIntResM` to the IEU for floating-point-to-integer moves and conversions, comparisons, and classifications. The FPU sends `FWriteDataE` to the LSU and receives back `ReadDataW` for floating-point stores and loads.

`fma` and `fcvt` instructions have a 2-cycle latency, so subsequent dependent instructions incur a one-cycle stall. `fcmp`, `fsgninfj`, `fclassify`, and `fmv` instructions have a 1-cycle latency. Division latency depends on the size of the operands and the radix and number of recurrence steps per cycle.

16.6.1.1 Floating-Point Register File

The floating-point register file has 4 ports: 3 read ports and 1 write port. It has 32 entries, each `FLEN`-bits wide, to support the 32 maximum-precision floating-point registers. If the system supports multiple precisions, lower-precision numbers are NaN-boxed in the maximum-precision word. Because multiport memories are not available in many ASIC processes, the register file can be implemented with standard cells using `FLEN*32` enabled flip-flops, three 32:1 `FLEN`-bit read multiplexers, and a 5:32 write decoder.

16.6.1.2 Result and Bypass Multiplexers

The floating-point result multiplexers choose a floating-point result and an integer result. The floating-point result comes from the postprocessor for `fma`, `fdivsqrt`, or `fcvt` instructions or from `fcmp`, `fsgninfj`, or the IEU (for int-to-float moves). The postprocessor result arrives late, so the other results are premultiplexed in the execute stage to save time and registers. The floating-point result is written to the floating-point register file and may be selected by the bypass (also called forwarding) muxes for a dependent operation.

The integer result comes from the postprocessor (for float-to-int conversions), `fcmp`, `fclassify`, or `SrcX` (for float-to-int moves). The IEU Result multiplexer sends that integer result to the IEU register file, as described in Section 16.6.13. The multiplexing occurs in the Execute stage and the

results are sent to the IEU at the start of the Memory stage in time to forward to the next instruction.

The three bypass multiplexers at the start of the Execute stage select `SrcX`, `SrcY`, and `SrcZ` from the floating-point register file or from previous instructions in the Memory or Writeback stages.

16.6.2 Unpacking Unit

The unpacking unit (`unpack`) extracts fields from the inputs as described in Section 16.2.1. Because Wally supports multiple precisions on the same configurable hardware, it also extends lower precisions to `FLEN` by appending trailing 0s to the fractions and by adding a suitable offset to the exponents. The `fmt` field of the instruction represents the desired precision according to Table 16.17. `PLEN` ($\leq$ `FLEN`) is the length of the operand. For example, in RV32IFD, `XLEN` = 32 because the base architecture is 32 bits and `FLEN` = 64 because the register file and datapath accommodate double precision. In an `fadd.s f1, f2, f3` instruction, the precision is single, so `PLEN` = 32.

Unpacking also verifies that lower-precision numbers are properly NaN-boxed by having all 1s in the leading bits. It asserts the NaN output if they are not boxed correctly. Thus, if a program inadvertently attempts to perform a double-precision operation on a single-precision number in a double-precision register, the number is treated as NaN to warn about the mistake.

The unpacker then pulls out the sign X_s from the most significant bit of the specified precision (X_{PLEN-1}). The raw fraction comes from the bottom bits and the raw exponent from the middle bits. The unpacker checks for special cases of all 0s and all 1s in the exponents. The fraction is extended to the full length by appending 0s to the least significant bits.

Extending the exponent to the desired length and bias initially seems difficult but simplifies gracefully. For example, consider extending an 8-bit single-precision exponent with a bias of 127 to an 11-bit double-precision exponent with a bias of 1,023. We add 1,023 and subtract 127 to get the new exponent. In binary, we add $0111111111 - 01111111 = 01110000000$ to the raw 8-bit exponent. If the raw exponent were $e_7e_6e_5e_4e_3e_2$, adding 01110000000 gives $e_7\bar{e}_7\bar{e}_7\bar{e}_7e_6e_5e_4e_3e_2e_1e_0$. In other words, the lower 7 bits remain the same, and if e_7 is 0, the upper bits are 0111; otherwise, they are 1000. Thus, extending an exponent reduces to inserting complemented versions of the raw exponent's most significant bit. If the exponent is subnormal (all original exponent bits are 0), replace e_0 with 1 to bump up the exponent.

Table 16.17 Floating-point format codes.

fmt	Precision	PLEN
00	Single	32
01	Double	64
10	Half	16
11	Quad	128

Example 16.20 UNPACK A HALF-PRECISION NUMBER

Consider an FPU that supports single- and half-precision operands. The half-precision number $0x8300 = -0.11 \times 2^{-14} = -1.1 \times 2^{-15}$ is stored in NaN-boxed form as $0xFFFF0300$. Unpack it to the fields of a single-precision number.

Solution: $0x8300 = 1\ 00000\ 1100000000$, so the sign is $X_s = 1$, the raw fraction is $X_f = 1100000000$, and the raw exponent is $X_e = 00000$. Extend this exponent from 5 to 8 bits by adding $127 - 15 = 112$ ($01111111 - 01111 = 01110000$), then add 1 (because the exponent was originally subnormal) to get: $01110001 = 113$, or an unbiased exponent of -14 . Because the original exponent was zero, the number was subnormal and we don't prepend a 1 to the fraction; so, the significand is $0.110\dots0$. This is normalized to a significand of $1.10\dots0$ (giving a fraction of $10\dots0$), and the exponent is decremented by 1 to 01110000 to account for normalization. Hence, the single-precision number is $1\ 01110000\ 100\dots0 = 0xB8400000$.

The remainder of the special case detection is the same as in the fixed-size unpacker from [Table 16.6](#).

16.6.3 Fused Multiply-Add

Compared to the basic FMA from [Section 16.2.2](#), the RISC-V FMA provides the option of inverting the addend, the result, or both: $W = \pm (X \times Y \pm Z)$. Thus, it supports all four flavors of RISC-V instructions: `fmadd`, `fmsub`, `fnmadd`, `fnmsub`. The FMA is also used to compute `fadd`, `fsub`, and `fmul` by setting one of the sources to 1.0 or 0.0.

16.6.4 Divide and Square Root

The Wally divide and square root unit (`fdivsqrt`) uses minimally-redundant radix-2 or radix-4 steps with hardware for each step replicated by a configurable amount $k = 1$ to 4 to perform k steps each cycle.

Division and square root perform a variable number of steps to produce $b = N_f + 1$ fraction bits, with N_f depending on the target precision. Radix- 2^r operations produces r -bit digits so requires b/r steps. Division needs an additional step to produce the integer digit. The unit contains k copies of the recurrence hardware to perform k steps per cycle. Preprocessing and postprocessing each involve an extra cycle. Hence, the number of cycles is:

$$\text{Cycles} = \left\lceil 1 + \frac{b}{r} \right\rceil + 2 \quad (16.8)$$

This latency is given in [Table 16.18](#).

Wally optionally uses the floating-point divider for integer division. When `IDIV_ON_FPU` is 1 in the configuration files, the integer division logic

k is `FDIVSQRT_STEPS_PER_CYCLE` in the configuration files.

Table 16.18 Wally's divide and square root unit (`fdivsqrt`) latency.

Precision	Latency				
	Radix-2			Radix-4	
	$k = 1$	$k = 2$	$k = 4$	$k = 1$	$k = 2$
Half	15	8	5	9	5
Single	28	15	8	15	8
Double	57	29	15	30	16
Quad	117	59	30	60	31

from Section 15.4.3 is removed, and the FPU divider is enhanced to support integer division. These enhancements include the integer preprocessing and postprocessing from Section F.2.9, an FSM modification to select the number of cycles for integer division, and a multiplexer in the IEU datapath to choose the division result from the FPU rather than the MDU. `DivBusyE` is disabled; instead, `FDivBusyE` is used to stall both integer and floating-point division.

`rv64gc` is configured with Radix-4, $k = 4$, and `IDIV_ON_FPU = 1`. `rv32gc` is configured with Radix-4, $k = 2$, `IDIV_ON_FPU = 0`, `IDIV_BITSPERCYCLE = 2`. `rv32gc` has smaller operands, so it can get away with less division hardware and still have reasonable latency. Also, having different configurations makes the regression suite test both floating-point and integer division hardware and different k .

16.6.5 Convert

Wally's floating-point convert unit (`fcvt`) handles the RISC-V conversion instructions defined in Table 16.19, depending on which floating-point sizes are supported. Note that the first letter specifies the destination format (half/single/double/quad) and the second specifies the source format, so `fcvt.s.d` converts a double to a single. `w` and `wu` are signed and unsigned 32-bit integers, and `l` and `lu` are signed and unsigned 64-bit integers. For example, `fcvt.s.wu` converts an unsigned 32-bit number to a 32-bit single-precision floating-point number.

RV64 doubles the number of supported integer conversions by adding `l` and `lu` everywhere `w` and `wu` appear for RV32.

In RISC-V, float-to-int conversions set the `Inexact` flag when the result is inexact. In the IEEE standard, this operation is called *ConvertToIntegerExact*.

Table 16.19 Floating-point conversions supported in RV32.

Modes Supported	Float-Float Conversions	RV32 Float-Int Conversions	RV32 Int-Float Conversions
F	None	fcvt.w.s, fcvt.wu.s	fcvt.s.w, fcvt.s.wu
FD	fcvt.s.d, fcvt.d.s	F + fcvt.w.d, fcvt.wu.d	F + fcvt.d.w, fcvt.d.wu
FDQ	FD + fcvt.s.q, fcvt.d.q, fcvt.q.s, fcvt.q.d	FD + fcvt.w.q, fcvt.wu.q	FD + fcvt.q.w, fcvt.q.wu
HF	fcvt.h.s, fcvt.s.h	F + fcvt.w.h, fcvt.wu.h	F + fcvt.h.w, fcvt.h.wu
HFD	HF + FD + fcvt.h.d, fcvt.d.h	HF + FD	HF + FD
HFDQ	HFD + FDQ + fcvt.h.q, fcvt.q.h	HF + FDQ	HF + FDQ

16.6.6 Move

The FPU module handles moves from floating-point to integer registers by selecting `FIntResE` as a sign-extended version of the floating-point bits. Similarly, moves from integer to floating-point registers place the integer bits in the lsbs of the floating-point register and 1s in the upper bits for the NaN box, creating the move result `AlignedSrcAE`.

16.6.7 Compare

In RISC-V, when both inputs are NaN, `fmin` and `fmax` produce a canonical NaN result.

16.6.8 Sign Injection

The `fsgn` sign injection block handles the `fsgnj`, `fsgnjn`, and `fsgnjx` instructions that modify the sign of `X` based on the sign of `Y` according to [Table 16.20](#). The hardware involves only a few 1-bit multiplexers and gates.

Sign injection supports pseudoinstructions for more common operations of floating-point moves, negation, and absolute value.

Table 16.20 Sign injection.

Instruction	Funct3	Result
<code>fsgnj</code>	000	$\{Y_s, X_e, X_f\}$
<code>fsgnjn</code>	001	$\{\overline{Y_s}, X_e, X_f\}$
<code>fsgnjx</code>	010	$\{X_s \oplus Y_s, X_e, X_f\}$

<code>fmv</code>	<code>ft1, ft0</code>	<code>fsgnj</code>	<code>ft1, ft0, ft0</code>
<code>fneg</code>	<code>ft1, ft0</code>	<code>fsgnjn</code>	<code>ft1, ft0, ft0</code>
<code>fabs</code>	<code>ft1, ft0</code>	<code>fsgnjx</code>	<code>ft1, ft0, ft0</code>

16.6.9 Classify

The `fclassify` classification module handles the `fclass` instruction that writes a 10-bit code to the integer register file based on the characteristics

of X according to Table 16.21. All of the characteristics are determined during unpacking, and the module contains only a handful of gates.

16.6.10 Zfa Additional Floating-Point Instructions

When `ZFA_SUPPORTED = 1`, the FPU is enhanced to support the additional instructions from Section 16.4.5. The new and modified modules include:

- ▶ **f1i:** New module with tables of immediates, indexed with `rs1` and the destination size
- ▶ **fround:** New rounding module, described next
- ▶ **fpu:** Two new move multiplexers to select a pair of registers in `fmvp` (`IntSrcE`) and the upper half of the floating-point register for `fmvh` (`IntSrcXE`)
- ▶ **fcmp:** Modifications supporting the NaN handling and flags for `fminm`/`fmaxm`/`fltq`/`fleq`
- ▶ **specialcase:** `fcvtmod.w.d` result and flag multiplexing

16.6.10.1 fround

`fround` resembles `fcvt` in that it rounds a floating-point number to an integer but differs in that it expresses the result in floating-point form. Reusing `fcvt` logic and adding the conversion back to floating-point would significantly increase the FPU postprocessing critical path or add an extra cycle that the pipeline does not presently support. Therefore, Wally adds a dedicated `fround` module that is surprisingly simple.

The `fround` module receives the exponent X_e , significand X_m , and sign bit X_s from the unpacker, expanded to the full datapath width `FLEN`. The basic idea is to examine the unbiased exponent. If it is negative, the input is less than 1 and will round to either 0 or 1 based on the significand bits. Otherwise, the unit will round to an integer by discarding the least significant fraction bits. The unit computes L' , R' , and T' to round according to Table 16.4. A few special cases exist: If the input exponent is greater than or equal to N_f , then the input is a large integer value that does not require rounding; it is passed to the output. If the input is NaN or 0, the output will also be the same.

Fig. 16.22 summarizes the `fround` algorithm, and Fig. 16.23 shows a block diagram of the unit. The unit generates a thermometer-encoded *TMask* signal with 1s in the least significant columns indicating which bits contribute to the sticky bit T' . If the input is 1.0 or larger and hence has an integer portion to round, it also creates an *IMask* signal with 1s in the most significant columns indicating which bits contribute to the truncated integer result *TRUNC*. Finally, it generates one hot *HotE* and *HotEP1* signals that indicate the positions of the L' and R' bits (the least

Table 16.21 Classification.

Bit	Meaning
0	$-\infty$
1	Negative normalized
2	Negative subnormal
3	-0
4	$+0$
5	Positive subnormal
6	Positive normalized
7	$+\infty$
8	Signaling NaN
9	Quiet NaN

```

W = fround(X)
E = Xe - bias                                // Unbiased exponent
// Calculate L', R', and T'
if (E < 0)                                     // Negative exponents round to 0 or 1.
    L' = 0                                     // Lsb L' = 0
    if (E = -1) R' = 1; TMask = 0.1111...111 // If (E = -1) 0.5 ≤ X < 1. Round bit is 1
    else      R' = 0; TMask = 1.1111...111 // If (E < -1), X < 0.5. Round bit is 0
else                                              // Positive exponents truncate fraction and may add 1
    IMask = 1.0000...000 >>> E                // (in U1.Nf form); implies thermometer code (1.1...100...0)
    TMask = IMask >>> 1                        // 1.1 ... 100 ... 0 >>> 1 = 0.0...001...1
    HotE = IMask & (IMask << 1)                // A 1 in column E, where bit 0 is the integer bit,
                                              // bit 1 is the most significant fraction bit, etc.
    HotEP1 = HotE >> 1                        // A 1 in column E + 1
    L' = OR(Xm & HotE)                        // Xm[E], where Xm[0] is the integer bit,
                                              // Xm[1] is the most significant fraction bit, etc.
    R' = OR(Xm & HotEP1)                    // Xm[E+1]
    TRUNC = Xm & IMask                      // Truncated fraction, corresponds to truncated integer value
    RND = TRUNC + HotE                       // TRUNC + (1 >> E), corresponds to next integer
    T' = OR(Xm & TMask)                     // T' = OR(Xm[E+2:Nf]) if E ≥ 0, OR(Xf) if E = -1, 1 if E < -1
// Round and set Flags
Flags = 0                                     // Initialize Flags (might be updated in a later step)
if (X is NaN)
    W = Canonical NaN
    Invalid = (X is signaling NaN)
else if (E ≥ Nf)
    W = X                                     // Is exact; this also handles infinity
else
    RoundUp = RoundingLogic(Xs, L', R', T', rm) // Table 16.4
    if (E < 0)                                // 0 ≤ X < 1 rounds to 0 or 1
        if (RoundUp) {Ws, We, Wf} = {Xs, bias, 0} // ± 1.0
        else        {Ws, We, Wf} = {Xs, 0, 0} // ± 0
    else                                          // X ≥ 1 rounds to an integer or overflows to infinity
        if (RoundUp) Rm = RND else Rm = TRUNC // Round up to RND or down to TRUNC
        if (Rm = 2.0) {Ws, We, Wf} = {Xs, Xe+1, 0} // Round up to 1.0 × 2E+1 incrementing exponent
        else        {Ws, We, Wf} = {Xs, Xe, Rf} // Rounded fraction, retain sign and exponent
    if (is an froundx instruction) Inexact = R' | T'
// Finally, pack W into specified precision (including NaN boxing when needed)

```

Fig. 16.22 fround algorithm.

significant bit of the integer result and the first fraction bit afterward). It extracts L' , R' , and T' with AND-OR masking and reduction logic. Rounding logic determines whether to round up, which is performed by adding 1 to the significand in the $HotE$ column. If it rounds up to 2.0, the exponent is incremented, and the fraction is 0. Otherwise, the fraction R_f is the significand R_m without the leading 1. If the input is 0 or has an

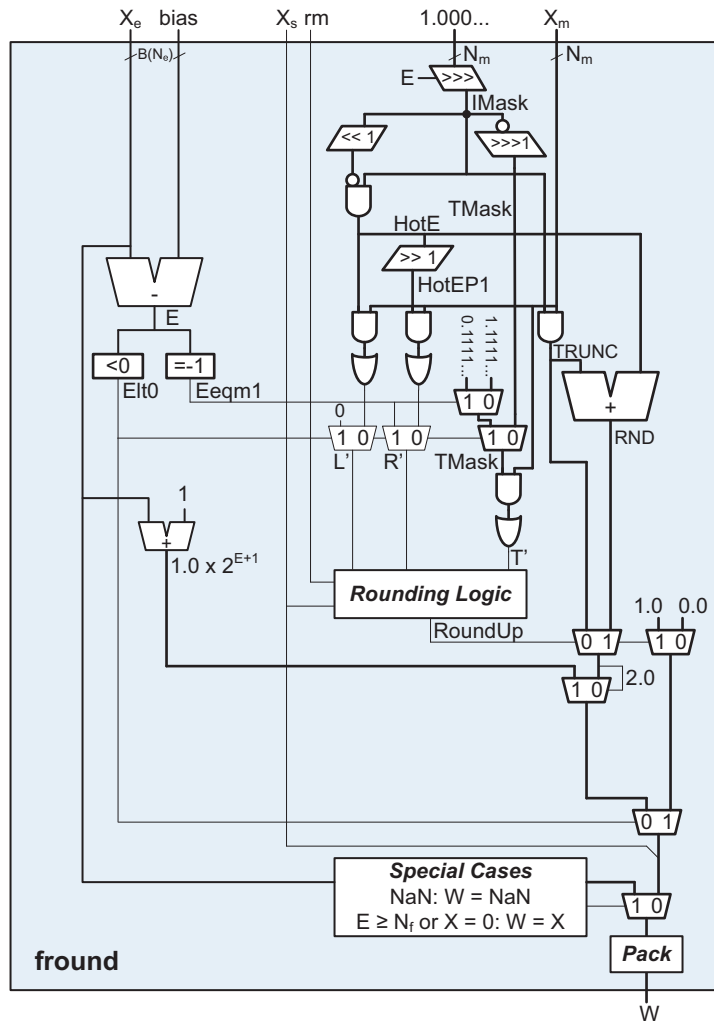


Fig. 16.23 fround unit.

exponent of at least N_f , it already represents an exact integer and does not require rounding. The unit also handles flag generation and special case of NaN. Finally, the output is packed back into the desired format.

Example 16.21 fround ALGORITHM

Express 6.75 as a half-precision floating-point number. Use the algorithm of Fig. 16.22 to round to an integer value (RNE mode) in half-precision floating-point format.

Solution: $X = 6.75 = 1.1011000000 \times 2^2$, with $X_s = 0$, $X_e = 15 + 2 = 17 = 10001_2$, and $X_f = 1011000000$. This exponent $E = 2$ is between 0 and 10, so we need to round. $IMask = 1.0000000000 \gg \gg 2 = 1.1100000000$. $TMask = IMask \gg \gg 1 =$

0.0001111111. $HotE = 0.0100000000$, with a 1 in column 2 of the fraction, and $HotEP1 = 0.0010000000$, with a 1 in column 3. $L' = 0$, $R' = 1$, and $T' = 1$. $TRUNC = 1.1000000000$, which is X truncated to an integer (6). $RND = TRUNC + HotE = 1.1100000000$, which is the significand for the next integer greater than $TRUNC$ (7). According to Table 16.4, based on L' , R' , and T' , round to $R_m = RND = 1.1100000000$. R_f is the fractional part of R_m , .1100000000. Hence, $W = \{X_s, X_e, R_f\} = 0\ 10001\ 1100000000 = 0x4700 = 7.0$.

16.6.11 Postprocessing

The rounding mode is set in the `funct3` field of floating-point instruction. This field is also referred to as the `rm` field when it indicates rounding. A mode of 7 indicates *dynamic rounding* that finds the rounding mode in the `fcsr` register instead. Dynamic rounding is effectively a requirement of the C99 programming language standard. However, changing the `fcsr` flushes the pipeline, so performance is better when specifying the rounding mode in the instruction if modes change often.

By default, the Wally FPU generates canonical NaNs according to the RISC-V specification. When `IEEE754 = 1` in `config_wally.vh`, the FPU will propagate NaN payloads instead, as recommended in the IEEE standard. This costs additional muxes in the `postproc` and `fcmp` units.

Wally supports multiple-precision operations on the same hardware, so some computations, such as those in the `fma`'s `shiftcalc` unit, depend on the precision of the current operation.

16.6.12 Privileged Unit Changes

When floating-point extensions are supported, the user-mode CSR module (`csru`) contains the `fcsr`, which is the only RISC-V user-mode CSR state. It includes 5 bits of flags and 3 bits of rounding mode; these fields can be written independently or together by software. The flags can also be set by operations and are sticky, remaining set until cleared by writing the `fcsr`. Fig. 16.24 shows a schematic of the `csru`.

The FPU also updates the `mstatus.FS` field, as described in Section 8.2.3.1. Specifically, `mstatus.FS` is set to 11 (dirty) whenever `FloatRegWriteW` asserts.

16.6.13 Integer Execution Unit Changes

When floating-point is supported, the IEU receives `FIntResM` and `FCvtIntResW` from the FPU. The FPU produces `FIntResM` for `fclass`, `fcmp`, and `fmv` instructions, which take a single cycle, and `FCvtIntResW` for `fcvt` (to integer) instructions, which take two cycles. The IEU datapath adds two muxes to choose these results, as shown in Fig. 16.25.

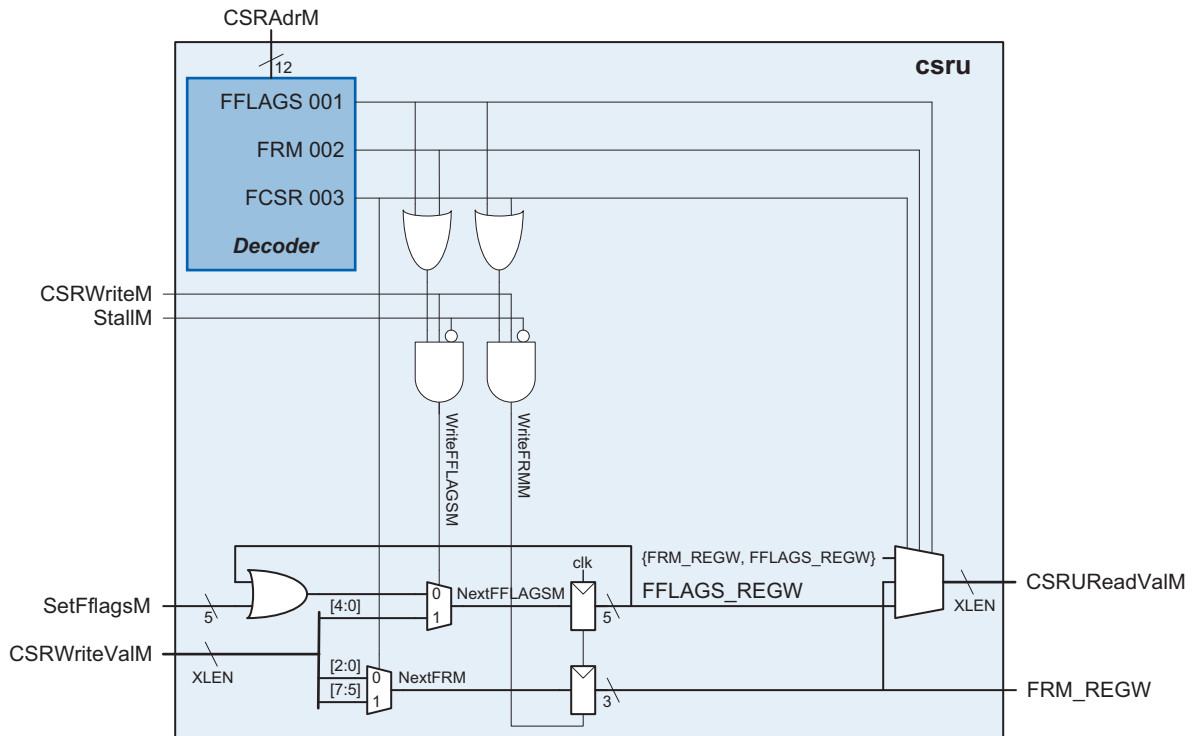
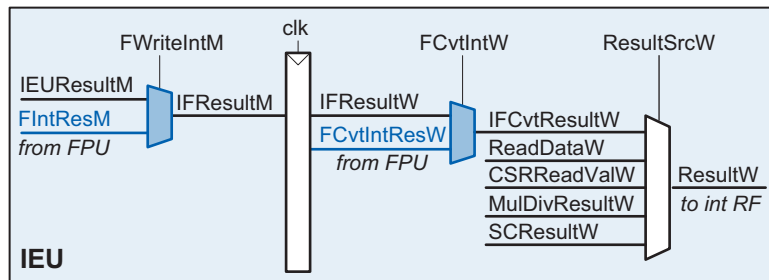
Fig. 16.24 User-mode CSR module (*csru*) with *fcsr*.

Fig. 16.25 IEU, highlighting the FPU signals.

16.6.14 Hazard Unit Changes

The hazard unit adds stalls for instructions with two-cycle latency and for divides with long and variable latency. Fig. 16.26 shows the complete Wally hazard unit, with FPU changes highlighted in blue.

Like the integer multiplier and the LSU, the FPU's *fma* and *fcvt* modules have a two-cycle latency, introducing a structural hazard. If the sources of these floating-point instructions are dependent on the result of the

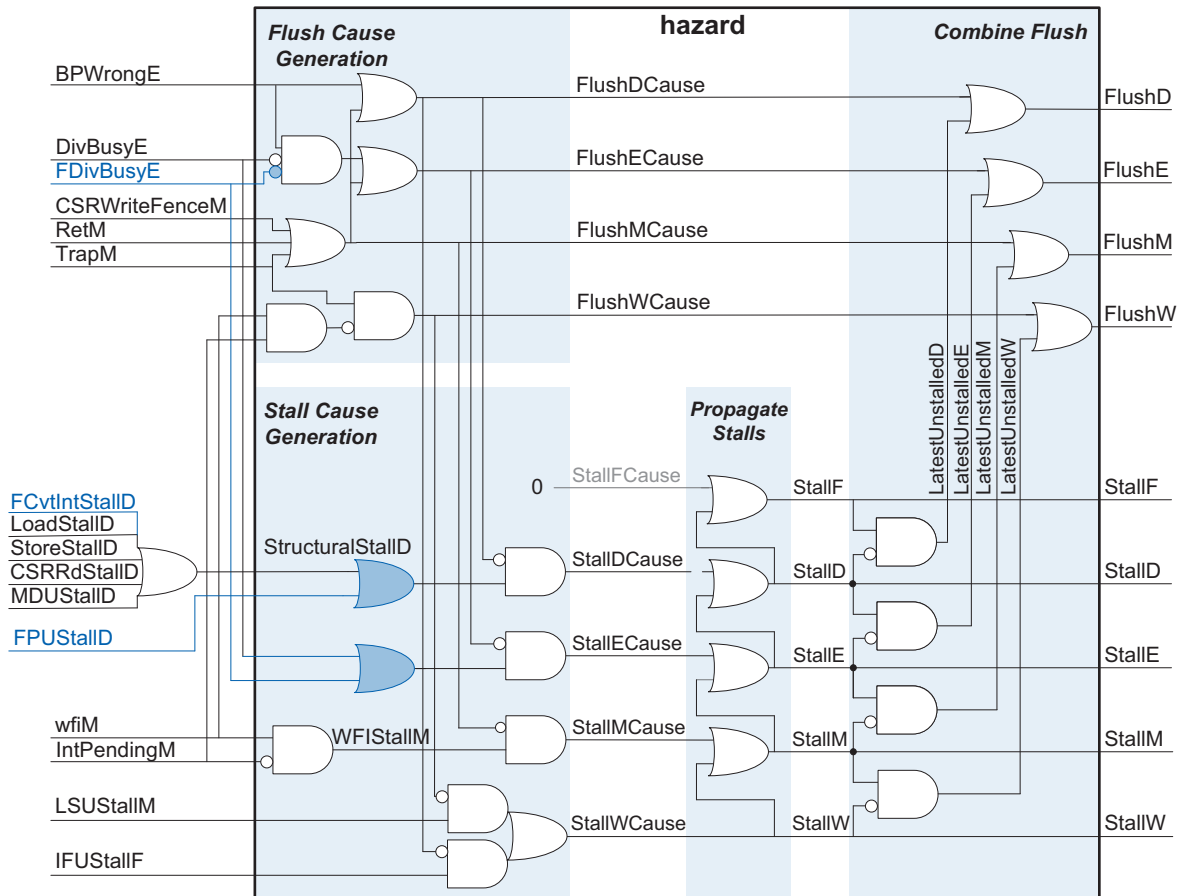


Fig. 16.26 Hazard unit, with FPU changes highlighted in blue.

preceding instruction, the processor must stall for a cycle to allow the preceding instruction to complete. The FPU's `fhazard` unit generates an `FPUS StallD` signal when this is detected during Decode. Floating-point to integer conversion must stall when an IEU instruction is dependent on the result of `fcvt`. The IEU's controller generates `FCvtIntStallD` when this is detected during Decode. The hazard unit's `StallDCause` logic ORs `FPUS StallD` and `FCvtIntStallD` with the other structural stall causes.

The `fdivsqrt` unit asserts `FDivBusyE` while it is busy, and the hazard unit stalls the pipeline until it finishes. As with `DivBusyE` from [Section 15.4.4](#), a branch instruction class misprediction waits for the division to complete before flushing the Execute stage.

FPU to IEU operations such as `fcmp`, `fclass`, and `fmv` have single-cycle latency, so no stalls are needed. Integer to floating-point conversion doesn't require new stalls either; most integer instructions have single-cycle latency, and load/store/MDU stalls already exist.

16.6.15 LSU Changes

As shown in gray in Fig. 12.2, the LSU (load/store unit) adds an `FWriteDataM` port containing data from the FPU for floating-point stores. It also adds a multiplexer to select between IEU and FPU write data to implement floating-point stores. The mux width is $LLEN = \max(FLEN, XLEN)$. For example, on RV32GC, $FLEN = 64$ and $XLEN = 32$, so $LLEN = 64$ to accommodate the doubles.

The external bus remains AHBW (typically $XLEN$), so if $FLEN > AHBW$, the system uses a data cache to service the wide accesses rather than going out to external memory. Moreover, the `ReadDataWordAHB` mux is needed for cache writebacks to reduce the $FLEN$ -sized `ReadDataWordM` to an AHBW-sized `LSUHWDATA` word. Presently, an LSU without a cache is unable to support the wide accesses needed for RV32D.

16.7 SUMMARY

The IEEE 754 Floating-Point Standard revolutionized numerical computing by allowing programmers to write code that behaves identically on different computers and performs gracefully in exceptional circumstances. Floating-point numbers are represented in 16-, 32-, 64-, or 128-bit half, single, double, and quad formats with a sign bit, a biased exponent, and a fraction with an implicit leading 1. Normalized numbers are of the form $(-1)^s \times (1.f) \times (2^{e-bias})$, where s is the sign, f is the fraction bits, and e is the exponent. The largest exponent is reserved for infinity and NaN. The smallest exponent is used for zero and subnormal numbers, which provide gradual underflow for more stable numerical computing. By default, results are rounded to the nearest representable value but can also be rounded toward positive or negative infinity or toward zero. The standard also provides sticky flags to indicate if a result has overflowed, underflowed, or was inexact, invalid, or involved division by zero.

An FPU typically includes an FMA (fused multiply-add) unit to perform addition, subtraction, multiplication, and FMA operations; a `fdivsqrt` unit for division and square roots; a convert unit for int-float, float-int, and float-float conversions; and a compare unit. FPUs often support multiple-precision operands on shared hardware. This involves the following:

- ▶ Unpacking the specified-precision operands into the longest supported format

- ▶ Detecting special cases
- ▶ Performing the operation in the longest format
- ▶ Normalization shift
- ▶ Rounding to the specified precision
- ▶ Handling special cases
- ▶ Updating flags

The FMA block computes $W = X \times Y \pm Z$. Thus, it can also be used to perform addition, subtraction, and multiplication by setting Y to 1 or Z to 0. Wally uses the FMA for all four operations to save chip area. The FMA first multiplies the significands and adds the exponents of X and Y . It aligns the significand of Z by right-shifting by the difference between the exponents of the product and of Z . It then adds or subtracts the significands of the product and aligned Z . If the result significand is negative, take the absolute value and flip the sign of the result. To normalize the result significand, it finds the leading 1 and shifts it into the integer position, adjusting the exponent accordingly. Then it rounds the results and handles special cases and flags. High-performance FMAs have several logic optimizations, such as a dual path adder to remove absolute value logic from the critical path, and a leading zero anticipator to speed up normalization.

The divsqrt block (`fdivsqrt`) uses a recurrence algorithm to produce one digit of the quotient or square root on each step. The division algorithm requires normalized significand inputs in the range $[1, 2)$, so sub-normal significands are left shifted into this range. Square root requires inputs in the range $[1/4, 1)$ with an even exponent, so they are right-shifted by one or two bits. The key to recurrence divsqrt is to avoid slow carry-propagate addition in each step. To do this, the residual is maintained in carry-save redundant form, representing a weighted version of the partial remainder or error between the approximate and true square root. Furthermore, the recurrence algorithm uses a symmetric redundant digit set for the quotient or square root digits: $\{-1, 0, 1\}$ for radix 2, or $\{-2, -1, 0, 1, 2\}$ for radix 4. At each step, the algorithm selects a result digit based on the most significant bits of the residual and the divisor or square root approximation, then updates the residual with a carry-save adder. The algorithm uses an on-the-fly converter to form the nonredundant binary result without a CPA. If the cycle time is long enough, the unit can perform more than one step per cycle to reduce latency. If the result is exact, the recurrence can terminate early. Again, the result may need normalization, rounding, special case handling, and flags. Integer division can share the divsqrt recurrence unit by normalizing the integer inputs to have a leading 1.

The convert block handles int-to-float, float-to-int, and float-to-float conversions. The algorithm is easiest to understand by considering each

case separately, but the hardware has many similarities. A shared unit for all three types of conversions involves an exponent calculation, an LZC (leading zero counter), and a significand shifter, as well as rounding, special cases, and flags.

The compare block takes advantage of the fact that positive floating-point numbers can be ordered with simple integer comparison. Therefore, it involves an integer comparator and some logic to handle sign correction and special cases.

Floating-point has many tricky corner cases and is notoriously difficult to get right. We check it against the UCB SoftFloat reference model using TestFloat and riscv-arch-test (IBM) test vectors.

RISC-V supports IEEE floating-point using the F, D, Q, and Zh extensions. The `FS` bits of the `mstatus` register must be turned on to enable floating-point. The floating-point control and status register (`fcsr`) is used to set rounding modes and check flags. Lower-precision operands are NaN-boxed with 1s in the upper bits when stored in the full-precision floating-point registers. Operations that produce a NaN always generate a canonical NaN with a 1 in the most significant bit of the fraction and 0s in the other bits, rather than propagating a NaN from one of the inputs.

Wally implements a simple FPU operating in the Execute and Memory stages with a critical path comparable to the cache access paths in the `rv{32/64}gc` configurations. The `fdivsqrt` unit supports both floating-point and integer division and uses multiple copies of recurrence hardware to produce many result bits per cycle.

Extensions: A (Atomic)

- 17.1 Principles
- 17.2 RISC-V Practices
- 17.3 Test Plan
- 17.4 Wally Implementation
- 17.5 Summary

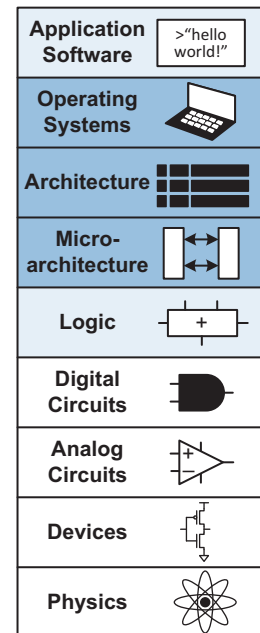
This chapter introduces principles of concurrency, consistency, coherence, synchronization, and atomic instructions. It describes the RISC-V atomic instructions and their implementation in Wally.

Recall that a *process* is a running program that comprises one or more *threads*. Sophisticated computer systems run multiple threads at the same time. This is known as *concurrency*. Users want concurrency because they want to do multiple things at once (e.g., listening to music while printing a term paper while simulating Verilog code while compressing a video file while monitoring for incoming emails and Slack messages). Each thread runs on a virtual user-level hart. The operating system (OS) may time-multiplex many virtual user-level harts onto a single physical core or may spread them across multiple cores to boost performance. For example, video compression runs faster when using multiple cores.

When concurrent threads access memory, the loads and stores might be interleaved in different ways. *Consistency* describes the order in which memory accesses are seen across the machine. *Coherence* is when all threads and cores eventually see the same values stored in memory. Often a specific order of memory accesses is necessary for correct operation; ensuring a specified order is called *synchronization*. Processors provide *atomic* instructions to facilitate synchronization. Such instructions reserve a memory address and then write to it or read, modify, and write it without interference from another thread. Readers interested in delving further into these topics may enjoy [Nagarajan20], [Scott13], and [Arpaci-Dusseau21].

17.1 PRINCIPLES

Concurrent systems have subtle and potentially nondeterministic behavior when multiple threads attempt to access the same data. This is clearly an issue in systems with multiple cores, but it may also occur when a single core context switches among multiple threads. Multicore systems face further challenges maintaining coherence among separate caches.



Atomic instructions derive their name from the ancient Greek word *atomos*, meaning indivisible.

Furthermore, concurrent systems raise questions of memory consistency: the allowable ordering of reads and writes to different memory locations. The most intuitive model of sequential consistency requires each thread to read and write in program order but allows different threads to interleave accesses in arbitrary ways. However, this model is overly restrictive for high-performance processors. The more relaxed *release consistency* model allows most memory accesses to be in any order and asks the programmer to explicitly acquire and release access to a shared variable when order matters. Acquiring and releasing access to a shared variable is called *synchronization*.

Threads must synchronize to avoid improper access to shared variables. Two common synchronization techniques are *locks* and *condition variables*. A thread places a lock on a shared variable, which guarantees that only that thread can access it until the thread releases the lock. Condition variables interact with the OS scheduler so that one thread can sleep until notified that another thread has produced or consumed data. Incorrect synchronization can also cause threads to *deadlock*, *livelock*, or *starve*, as discussed in [Section 17.1.5](#).

17.1.1 Concurrency

“In computer science, as in real life, concurrency makes it much more difficult to reason about events.”

— Michael L. Scott [Scott13]

Concurrency is essential to a good user experience and to high performance but introduces subtle issues that are easily mishandled. Concurrency arises for a single core running multiple threads, for multiple cores sharing memory, and for distributed systems that communicate through message passing. Although this book focuses on a single-core processor, we also briefly outline the issues faced by shared-memory and distributed systems.

In a single-core system running a multitasking OS such as Linux, the S-mode OS time-multiplexes among many user-level harts, each running a different thread. The SoC periodically produces a timer interrupt, which causes the OS scheduler to *context switch* to the next active thread. On an interrupt, the core enters S-mode, saves the PC, and jumps to an interrupt handler. The interrupt handler saves the state of the current thread’s user-level hart, loads the state for the next thread’s user-level hart, and returns to U-mode to run the next thread. When two threads are accessing a shared resource such as a memory location, file, or peripheral, the order of accesses among the threads may be important.

In a shared-memory multiprocessor, multiple cores have access to the same memory. For example, a typical phone, laptop, or server is a shared-memory multiprocessor. In 2025, a low-end processor might have only 4–8 cores, but a high-performance AMD or Intel laptop processor has

16–24 cores, and a high-performance server may have 2–8 sockets for chips, each containing up to 60–288 cores. In any of these configurations, each core typically has small (16–32 KiB) L1 instruction and data caches that can be accessed quickly and a medium (256 KiB–8 MiB) L2 cache. The chip also has a large (32–1,152 MiB) L3 cache that serves all the cores.

Fig. 17.1 shows an AMD Epyc 9004 server motherboard, which is a good example of a shared-memory multiprocessor. The Gigabyte MZ73-LM0 motherboard contains two sockets and a 12-channel DDR5 (double data rate 5) memory interface with 24 DIMM (dual inline memory module) sockets. Fully populated with 24 32-GiB modules, the motherboard holds 768 GiB of memory. Each 72×75 -mm socket contains an AMD processor. The AMD 9684X processor has 13 dies on a multichip module, including the I/O controller and a dozen CPU dies built on TSMC's 5-nm process. The processor can consume up to 400 watts and has 1,152 MB of L3\$ using cache chips stacked vertically on top of the CPU die. Each CPU die has 8 dual-threaded cores, for a total of 96 cores or 192 threads. Each Zen 4 core operates at a base frequency of 2.55 GHz and a maximum

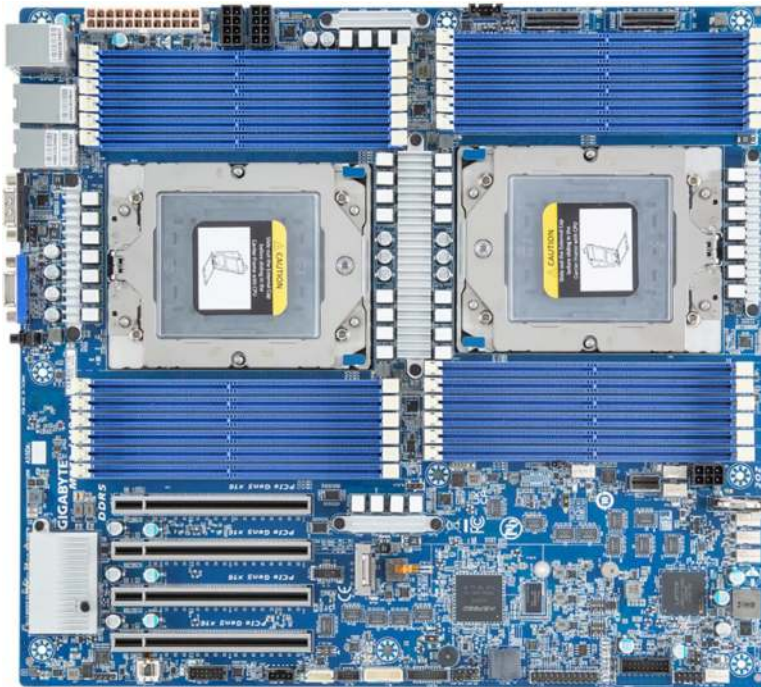


Fig. 17.1 AMD Epyc 9004 motherboard. (Figure courtesy of GigaByte. Reprinted with permission.)

frequency of 3.7 GHz. The cores each have 32 KiB L1 instruction and data caches and a 1-MiB 8-way set associative L2\$ cache. Their branch predictors have 1,536- and 7,168-entry L1 and L2 branch target buffers (BTBs) and a 32-entry return address stack (RAS). For instruction and data translation lookaside buffers (ITLBs and DTLBs), the L1 and L2 ITLBs have 64 and 512 entries, and the DTLBs have 72 and 3,072 entries. The core can decode 4 instructions per cycle, so the server has a peak throughput of $2 \text{ sockets} \times 96 \text{ cores} \times 3.7 \text{ GHz} \times 4 \text{ instructions/cycle} = 2.8 \text{ trillion instructions per second}$.

17.1.2 Coherence

Shared-memory multiprocessors face the challenge of *cache coherence*. The goal of cache coherence is to ensure that the memory system behaves as if the caches were not there. In particular, every store to memory must eventually be seen by subsequent loads from the same memory address. For example, imagine that a value at address 1000 is in the L1 cache of two different cores. One core writes a new value to address 1000, changing the value in its L1 cache. After sufficient time for the write to propagate through the memory system, the second core must get the new value when it reads from address 1000. If the second core simply kept the old value in its L1 cache indefinitely and never wrote it back to main memory, the memory system is called *incoherent*. Early shared-memory multiprocessors used *snooping* to maintain coherence. In such systems, when one core writes to its cache, the write address is also broadcast over a shared bus. Other cores snoop the bus and invalidate any cache entries at this address. Snooping does not scale gracefully because of contention for the shared bus. Larger multiprocessors use *directory*-based coherence, maintaining a shared or distributed directory of cache lines. They often use the *MESI* protocol, in which each line is marked as Modified, Exclusive, Shared, or Invalid [Hennessy19].

Although cache coherence is primarily a consideration for multicore processors, coherence between the instruction and data caches is also important for single-core processors. If the processor writes to a certain memory address and then executes an instruction from that same address, this *self-modifying code* should see the new instruction. As discussed in [Sections 10.4.2.6 and 10.4.5.4](#), the processor must perform a fence operation to flush any dirty lines from the data cache and invalidate them in the instruction cache to keep the caches coherent.

Very large systems would pay an unacceptable overhead keeping memory coherent across all the cores. Instead, such *clusters* or warehouse-scale computers are built from distributed processors with independent memories tied together on a high-speed network. Processors that wish to communicate pass *messages* over the network rather than using shared memories.

17.1.3 Consistency

Coherence describes the behavior of reads and writes to the same memory location, and *consistency* defines the relative ordering of reads and writes to different memory locations.

Suppose two concurrent threads T_1 and T_2 run [Code Example 17.1](#) with shared variables a , b , c , and d , which are all initialized to 0 in memory. What values could c and d have after the threads finish?

Code Example 17.1 CONCURRENT THREADS ILLUSTRATING CONSISTENCY

T_1	T_2
$a = 1$	$b = 1$
$c = b$	$d = a$

If T_1 runs first, $c = 0$, and $d = 1$. If T_2 runs first, $c = 1$, and $d = 0$. If T_1 and T_2 run concurrently, it is possible that $c = 1$ and $d = 1$. This could occur even on a single core. For example, T_1 could execute $a = 1$ and then context switch to T_2 . T_2 could complete and then context switch back to T_1 .

Surprisingly to many programmers, c and d could both be 0. For example, Intel and AMD processors have *write buffers* to improve performance. If T_1 and T_2 executed concurrently on different cores, the writes to a and b could both be placed into buffers on each core. The other thread on the other core would read the old value of these variables (0) because the writes are not yet visible to a cache coherence system. Moreover, most high-performance processors use out-of-order execution. T_1 could execute in reverse order ($c = b$, $a = 1$) because that would not violate any data dependencies within T_1 . Similarly, T_2 could execute in reverse order. In this case, c and d would both be 0 again, even if writes were instantly visible to all threads.

[Code Example 17.1](#) demonstrates that even very simple concurrent threads can have nondeterministic and counterintuitive behavior related to memory consistency. Programmers need to understand what behaviors are possible on a particular computer. Thus, architects define *memory consistency models* that specify the legal behaviors of concurrent programs. This section explores various models trading off simplicity and performance. [Section 17.1.4](#) describes synchronization required to guarantee correct behavior considering these consistency issues. Consistency is a subtle subject, and [Nagarajan20] is presently the best source if you wish to delve deeper.



Leslie Lamport [1941–] is a computer scientist and recipient of the 2013 Turing Award for contributions to distributed and concurrent systems, including the notion of sequential consistency. He is also the developer of the open-source LaTeX system for document preparation. Lamport received his bachelor's degree and PhD in mathematics from MIT and Brandeis and worked at SRI, DEC, Compaq, and Microsoft Research. (Image courtesy of Leslie Lamport. Reprinted with permission.)

17.1.3.1 Sequential Consistency

The most straightforward consistency model is called *sequential consistency* [Lamport79, Hennessy19]:

- ▶ Memory accesses by a particular thread occur in program order
- ▶ Memory accesses by multiple threads are arbitrarily interleaved

In the case of [Code Example 17.1](#), sequentially consistent orderings of memory accesses are given in [Table 17.1](#). Systems with sequential consistency do not reorder instructions and cannot use write buffers; thus, they never produce $c = d = 0$. An ordering such as $c = b \rightarrow d = a \rightarrow a = 1 \rightarrow b = 1$ also never occurs because sequential consistency requires that each thread's accesses occur in program order.

Table 17.1 Sequentially consistent orders of access.

Ordering	c	d
$a = 1 \rightarrow c = b \rightarrow b = 1 \rightarrow d = a$	0	1
$a = 1 \rightarrow b = 1 \rightarrow c = b \rightarrow d = a$	1	1
$a = 1 \rightarrow b = 1 \rightarrow d = a \rightarrow c = b$	1	1
$b = 1 \rightarrow d = a \rightarrow a = 1 \rightarrow c = b$	1	0
$b = 1 \rightarrow a = 1 \rightarrow d = a \rightarrow c = b$	1	1
$b = 1 \rightarrow a = 1 \rightarrow c = b \rightarrow d = a$	1	1

Just as coherence makes caches invisible, sequential consistency makes shared memory look like a single memory module [Nagarajan20]. Sequential consistency makes it easier to reason about concurrent programs and avoids some counterintuitive behaviors. However, it comes at a high cost to performance. It prohibits out-of-order memory accesses that are legal in a nonconcurrent program. More significantly, it makes shared-memory multiprocessors painfully slow at writing because a write to one cache cannot complete until it invalidates any lines containing the same address in different caches. These issues led architects to consider alternate consistency models that support write buffers and out-of-order issue of memory operations at the expense of a more subtle programming model.

17.1.3.2 Total Store Ordering

The SPARC architecture formalized the *total store ordering* (TSO) consistency model to support multicore processors with write buffers, and Intel/AMD x86 processors adopted TSO as well. It is similar to sequential

consistency except that it allows memory accesses by a particular thread to perform loads before a previous store completes, as long as the accesses are to distinct memory addresses. Stores, however, must occur in program order. TSO allows the counterintuitive result of $c = d = 0$ in the earlier example. However, most practical programs that properly synchronize access to shared variables see no difference between sequential consistency and TSO.

In situations where sequential consistency matters, the processor can use a *fence* operation. A fence ensures that all memory operations before the fence are seen by all threads to complete before any memory operation after the fence occurs. The fence can be implemented by stalling the core(s) until the write buffers finish updating the caches. A fence can be an expensive (time-consuming) operation, especially in large systems with long memory latencies.

Code Example 17.2 adds fences to Code Example 17.1. Now, even on a TSO machine, T1 finishes writing $a = 1$ before it may read b . Similarly, T2 finishes writing $b = 1$ before it may read a . The threads continue to inter-

Fences are sometimes called *memory barriers* (*membars*) or *syncs*.

Code Example 17.2 CONCURRENT THREADS WITH FENCES

T1	T2
$a = 1$	$b = 1$
FENCE	FENCE
$c = b$	$d = a$

leave in arbitrary ways but never produce a sequentially inconsistent result of $c = d = 0$.

17.1.3.3 Release Consistency

TSO facilitates multicore processors with write buffers, but TSO is not flexible enough to accommodate other important optimizations, including instruction reordering. Advanced compilers and speculative out-of-order processors seek to arbitrarily reorder loads and stores to distinct addresses. A *coalescing write buffer* can merge several writes to the same cache line into a single operation. A more relaxed consistency model permits these performance optimizations.

Define memory access ordering rules $X \rightarrow Y$ to mean operation X must complete before operation Y completes. Four orderings of reads and writes are possible: $R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$, and $W \rightarrow W$. Recall that sequential consistency requires all four to occur in program order for a particular thread. Total store order removes the $W \rightarrow R$ constraint. *Partial store order* removes both $W \rightarrow R$ and $W \rightarrow W$ to allow for reordering writes with coalescing write buffers. *Relaxed ordering* allows threads to

reorder independent memory accesses in any fashion [Adve95]. Of course, within a thread, accesses to a particular address take place in program order.

Relaxed ordering comes in several subtly different flavors, but the *release consistency* model, which requires the programmer to perform explicit synchronization on shared variables when order matters, is widely used; it is the model for RISC-V, ARMv8, and MIPS. The C/C++ language standard is based on release consistency or stronger. Power, Alpha, and SPARC architectures use other relaxed ordering variants. Release consistency avoids introducing unnecessary ordering in unshared variables. Release consistency uses fences or acquire/release operations (see [Section 17.1.3.4](#)) to synchronize between threads only when required for correct program operation. Release consistency gives maximum flexibility for hardware optimization but also requires a highly sophisticated programmer to write correct synchronization routines.

Under a release consistency model without fences, [Code Example 17.1](#) could produce the sequentially inconsistent result $c = d = 0$. Adding fences in [Code Example 17.2](#) is sufficient to prevent this result.

[Code Example 17.3](#) gives another example where, in the absence of fences, release consistency differs from sequential consistency and TSO. Suppose `a`, `b`, and `flag` are initially 0. The producer thread `T1` writes a nonzero value to `a` and `b`, then sets the `flag`. The consumer thread `T2` waits until the `flag` is set, then reads `a` and `b` into `c` and `d`. Consider behavior with and without the `FENCE`. Under sequential consistency, $c = d = 1$. Under TSO, the program also produces $c = d = 1$ even when there are no fences, because `T2` will never see `flag = 1` until after `T1` stores 1 into `a` and `b`. However, under release consistency, without fences, the stores could be reordered arbitrarily. `T2` might observe `flag = 1` before `a` or `b` have become 1 and, hence, might set `c` and/or `d` to 0. Less intuitively, `T2` could reorder the loads such that it reads `a` or `b` = 0 (and thus sets `c` or `d` to 0) before finding `flag = 1`. When the fences are added, the `T1` fence ensures all other threads see $a = b = 1$ before they see `flag = 1`. Similarly, the `T2` fence ensures that `T2` reads `flag = 1` before reading `a` or `b`.

Code Example 17.3 PRODUCER-CONSUMER EXAMPLE SHOWING RELEASE CONSISTENCY ORDERING

<code>T1: Producer</code> <code>a = 1</code> <code>b = 1</code> <code>FENCE</code> <code>flag = 1</code>	<code>T2: Consumer</code> <code>While (!flag); // Wait for flag</code> <code>FENCE</code> <code>c = a;</code> <code>d = b</code>
--	--

Even with the fences, release consistency does not impose strict ordering. For example, T2 might observe `b` becoming 1 before `a` becomes 1. This is harmless because T2 doesn't care about `a` and `b` until after `flag` is raised, and it gives the hardware flexibility to reorder operations for performance.

Notice that the fence is conservative. For example, if T1 performed other subsequent loads or stores to independent variables, they could not be reordered before the fence even if that wasn't important to program operation.

17.1.3.4 Acquire and Release Constraints for Release Consistency

Programs in a release consistency model must specify ordering more often than in TSO or sequentially consistent models because release consistency lacks the implicit ordering of the other models. Fences are sufficient but can be overkill because they order all operations, not just the essential ones. Moreover, each fence is another instruction to fetch and execute, slowing program performance. To address these issues, release consistency models provide *acquire* and *release* constraints to specify ordering of certain memory operations. These constraints are typically specified by bits in the memory operation, so no separate fence instruction is needed.

A memory operation is called an *acquire operation* if it has an acquire constraint, a *release operation* if it has a release constraint, and a *sequentially consistent operation* if it has both acquire and release constraints. Acquire and release operations in some thread T enforce how other threads observe ordering of other memory operations from thread T. All threads will observe an acquire operation in T before any later memory operations in T. All threads will observe a release operation in T after all earlier memory operations in T. All threads will observe a sequentially consistent operation in T after all earlier memory operations in T and before any later operations in T.

Code Example 17.4 adapts Code Example 17.3 to use acquire/release constraints instead of fences. In T1, the store to `flag` has a release constraint, ensuring that T2 observes `flag` becoming 1 after `a` and `b` became 1. In T2, the load of `flag` has an acquire constraint, ensuring that T2 reads `a` and `b` after `flag` becomes 1.

Code Example 17.4 PRODUCER-CONSUMER EXAMPLE WITH ACQUIRE/RELEASE CONSTRAINTS

```
T1: Producer
a = 1
b = 1
flag = 1; <release>
```

```
T2: Consumer
while (!flag); <acquire>
c = a;
d = b
```

17.1.3.5 Consistency Summary

A memory consistency model describes how reads and writes to different memory locations are ordered. The programmer must understand the consistency model to understand the possible behaviors of a multithreaded program; they must also understand fences or other synchronization required to achieve the desired behavior. Microarchitects design the out-of-order issue logic and memory system to obtain the desired level of consistency. Three common models are sequential consistency, TSO, and release consistency.

Sequential consistency is the most natural to a programmer. In this model, each thread issues memory accesses in program order, but multiple threads may interleave in arbitrary order. Hence, shared memory behaves to the programmer as one single large memory, even if it is physically implemented with per-core caches. Unfortunately, sequential consistency comes at a high performance cost. It prevents out-of-order issue of memory accesses that would be legal ways to increase performance in a single-threaded system. It also costs power and performance because a write to an address in one core's cache must invalidate any line in another core's cache that contains the same address. It prevents the use of store buffers that hide the latency of stores and coalesce multiple stores to consecutive addresses into a single cache line update.

TSO is similar to sequential consistency except that a thread can reorder a load to occur before a previous store to a different address completes. This allows microarchitects to use store buffers and out-of-order issue. Most properly synchronized programs behave the same under TSO as under sequential consistency. When stronger ordering is necessary, the programmer inserts fence operations that ensure all memory accesses before the fence are seen by all threads to have completed before any memory access after the fence occurs.

Release consistency is an even more relaxed ordering that allows maximum flexibility for hardware optimization, including out-of-order execution. It comes at the cost of a thread having to acquire a shared variable before using it and then release the variable before another thread can use it. Fences or acquire/release constraints are inserted to ensure ordering where it matters. [Section 17.2](#) shows how RISC-V supports these operations for release consistency.

Getting these constraints right takes careful thought by a sophisticated developer. Programmers normally rely on carefully designed synchronization libraries to avoid introducing subtle consistency bugs.

17.1.4 Synchronization

Concurrent threads can misbehave when they interleave in undesirable ways. *Synchronization* is the art of avoiding incorrect interleaving. Most

synchronization problems can be classified as *mutual exclusion* or *producer-consumer*. These problems are usually solved with *locks* and *condition variables*, respectively. This section describes the problems, along with synchronization mechanisms and hardware support to solve the problems.

17.1.4.1 Mutual Exclusion: Locks

Imagine that you and your mother share access to your college bank account. You have \$500 in the account and need to withdraw \$400 to buy textbooks at the campus bookstore. Your mother promises to deposit \$300 in your account to help out.

You and your mother each visit an ATM at the same time. Your ATM invokes the Withdrawal function with thread T_1 , and your mother's invokes the Deposit function with thread T_2 , as shown in [Code Example 17.5](#). The functions run concurrently and could interleave in arbitrary ways even if the system is sequentially consistent. Suppose they run in the order shown in [Table 17.2](#). At the end of the day, your mother's deposit vanished, you have only \$100 in your account, and you are poor and angry.

Code Example 17.5 MUTUAL EXCLUSION BANKING PROBLEM

```
T1: Withdrawal
b1 = getBalance()
b1 = b1 - withdrawalAmount
setBalance(b1)
```

```
T2: Deposit
b2 = getBalance()
b2 = b2 + depositAmount
setBalance(b2)
```

Table 17.2 Mutual exclusion ordering example.

Thread	Statement	Notes
T1	<code>b1 = getBalance()</code>	<code>b1 = 500</code>
T2	<code>b2 = getBalance()</code>	<code>b2 = 500</code>
T2	<code>b2 = b2 + depositAmount</code>	<code>b2 = 800</code>
T2	<code>setBalance(b2)</code>	<code>balance = 800</code>
T1	<code>b1 = b1 - withdrawalAmount</code>	<code>b1 = 100</code>
T1	<code>setBalance(b1)</code>	<code>balance = 100</code>

Mutual exclusion locks are also referred to as *mutex locks* or simply *mutexes*.

This is an example of a *race condition* in which the result depends on the order of operations in the interleaved threads. To solve this problem, the program must use *mutual exclusion*, where code within a *critical section* cannot interleave. T1 and T2 each are executing *critical sections* (from `getBalance` to `setBalance`) that should not be allowed to interleave. They should be synchronized so that either the withdrawal or deposit occurs first and runs to completion before the other thread runs at all. If T1 ran before T2, your balance would drop to \$100 and then rise to \$400. If T2 ran before T1, your balance would rise to \$800 and then drop to \$400. Although the exact sequence is nondeterministic, the final answer is correct.

Race conditions can occur even on systems with a single core because the OS may context switch between threads at an inopportune time. For example, even if T1 and T2 both executed on a single computer core, the computer could switch from T1 to T2 and back to T1 as in [Table 17.2](#).

Locks

Threads use locks around critical sections to achieve mutual exclusion. A *lock* is a shared variable used to ensure only one thread can be in its critical section at a time. [Code Example 17.6](#) fixes the banking program using locks. The `lock(L)` function uses a lock variable `L`. It waits until `L` is 0, then “locks” it: sets `L` to 1. `unlock(L)` clears `L` to 0. Only one of the threads can hold the lock at a time. Therefore, only one thread at a time can change the balance, and the transactions occur properly, although their order is unpredictable.

Code Example 17.6 MUTUAL EXCLUSION BANKING PROBLEM WITH LOCKS

T1: Withdrawal lock(L) // Critical section start b1 = getBalance() b1 = b1 - withdrawalAmount setBalance(b1) unlock(L) // Critical section done	T2: Deposit lock(L) // Critical section start b2 = getBalance() b2 = b2 + depositAmount setBalance(b2) unlock(L) // Critical section done
---	---

[Code Example 17.7](#) shows three methods of implementing lock and unlock. The lock variable `L` is a pointer to a memory location that holds 0 when the lock is available or 1 when the lock is claimed.

Code Example 17.7 LOCK FUNCTION IMPLEMENTATIONS

(a) Incorrect software	(b) Interrupt-based	(c) Test and set
<pre>void lock(int *L) { while (*L); // Wait *L = 1; } void unlock(int *L) { *L = 0; }</pre>	<pre>void lock() { DisableInterrupts(); } void unlock() { EnableInterrupts(); }</pre>	<pre>void lock(int *L) { while (TestAndSet(L)); FENCE } void unlock(int *L) { FENCE *L = 0; }</pre>

In method (a), the `lock` function waits until `L` is 0, then sets `L` to 1. This is called *spinning*. `unlock` restores `L` to 0. This approach has a subtle bug. Imagine that `T1` and `T2` execute `lock()` concurrently. Both detect that `L` is 0 and exit the while loop. Both advance to `*L = 1` and think they have claimed the lock and can run their critical section. This may occur even on a single-core processor if a context switch occurs from `T1` and `T2` between when `T1` detected `L = 0` and when it sets `L = 1`.

Method (b) works on a single-core processor. The lock function simply disables interrupts. In particular, it turns off the timer interrupt that could allow the OS scheduler to switch threads. Because interrupts are disabled, when a thread has the lock, it is guaranteed to continue executing until it releases the lock. This method has many drawbacks. If it holds the lock for a long time, other threads will be starved from executing, so the system may feel unresponsive. A buggy or malicious thread may never release the lock, hanging the system. For these reasons, an OS typically does not let user code disable interrupts. Moreover, on a multicore system, the threads could be executing on different cores so the race condition could occur even if interrupts are disabled on both cores.

Method (c) relies on the hardware support of an *atomic* `TestAndSet` instruction. The instruction reads the value at an address, writes a 1 to that address, and returns the value read. *Atomic instructions* cannot be interrupted in the middle; the read and write happen back to back such that no other thread may access the memory system in between the read and write. Either `T1` or `T2` will call `TestAndSet` first, detecting `L` was 0 and setting it to 1. That thread now has the lock and can execute the critical section. When the other thread calls `TestAndSet`, it reads `L = 1` and waits in the while loop until the lock is released. Even if both `T1` and `T2` attempt `TestAndSet` at the same time, the memory system will arbitrate between the two requests and only issue one at a time. In a system with release consistency, fences are needed after acquiring the lock and before releasing

The `TestAndSet` instruction dates back to the earliest multiprocessors, such as the Burroughs B5000 in the 1960s [Mayer82].

it to ensure the critical section runs while the lock is held. Sequential consistency or TSO guarantees such ordering so the fence could be omitted.

Theoreticians have devised clever algorithms to reliably lock without atomic instructions [Dijkstra65]. These algorithms are tricky, and they are even trickier when they must support more than two threads. Moreover, these algorithms require sequential consistency, which most computers don't provide. Atomic support for locks has become a standard instruction set feature for concurrent systems.

Better Locking

A good lock should not just protect the critical section but also be *fair* and *efficient*. When multiple threads are waiting on the same lock, each thread should have a fair chance of getting the lock when it becomes available, and a thread should never *starve* waiting indefinitely while other threads always get the lock. An efficient lock doesn't waste large amounts of CPU time waiting.

The lock from [Code Example 17.7\(c\)](#) is called a *spin lock* and is simple but neither efficient nor fair. It spins (waits) in the while loop until the lock is available. Spinning for a long time is a waste of processor resources. Moreover, when the lock becomes free, there are no guarantees which thread will get it next, so an unlucky thread might starve.

A spin lock is analogous to trying to buy hard-to-get concert passes from a website. Users keep refreshing the website hoping a pass becomes available, then trying to purchase the pass before somebody else gets it. This is a waste of time, and you might never get your pass. In contrast, a *ticket lock* resembles taking a ticket at the deli counter. The butcher calls customers in the order of tickets. You still have to wait, but you are guaranteed eventual service. Waiting is inefficient, so it is even better to put the thread to sleep if the lock is unavailable and to have it wake up when its ticket is ready. However, context switching a sleeping process out and in is also expensive, so it is best to spin for a little while in the hopes of acquiring the lock quickly. This *sleep lock* is analogous to a customer service phone line, where the agent will help you if they are available soon or take your phone number and call you back in the order your call was received. Ticket and sleep locks take advantage of more capable atomic instructions and support from the OS, as discussed next.

Atomic Instructions

Typical atomic instructions include `TestAndSet`, `Swap`, `FetchAndIncrement`, `CompareAndSwap`, and the `LoadReserved-StoreConditional` pair. The atomic `TestAndSet` instruction is sufficient to build a spin lock, which in turn can be used to build other synchronization operations. The `Swap` instruction is similar; it atomically reads a memory location and writes a new value back to that location; it differs from `TestAndSet` in that the value written is

configurable rather than always 1. Either of these instructions is sufficient for building synchronization mechanisms, but more powerful atomic instructions make synchronization more efficient.

The atomic `FetchAndIncrement` instruction reads a value from memory, adds one, and writes it back. `FetchAndIncrement` can be used in place of `TestAndSet`, but it is more powerful because it facilitates ticket locks and semaphores, as will be discussed soon.

The even more powerful atomic `CompareAndSwap` instruction takes three inputs: a memory address, a comparison value, and a swap value. It reads a value from memory. If the read value matches the comparison value, it then writes the swap value back to the same memory location. In either case, it returns the read value. `TestAndSet`, `FetchAndIncrement`, and other lock-free and wait-free synchronization algorithms can be implemented with `CompareAndSwap`, as shown in [Code Example 17.8](#) for `TestAndSet`. `CompareAndSwap` has been part of the IBM 370 architecture since 1970 and is supported by x86, ARM, and most but not all other architectures. `CompareAndSwap` does not play as gracefully with RISC architectures because it requires three source operands.

`CompareAndSwap` is susceptible to the *ABA problem*, in which the value in memory changes from A to B and back to A without being detected in a comparison. To avoid this problem, `CompareAndSwap` often uses double-wide operands, storing the value in one word and a counter in the second word so that the counter can be atomically incremented at the same time the value is swapped.

Code Example 17.8 IMPLEMENTING `TestAndSet` WITH `CompareAndSwap`

```
int TestAndSet(int *adr) {
    while (CompareAndSwap(adr, 0, 1) == 1); // Spin until adr contains 0, then set *adr = 1
}
```

The `LoadReserved-StoreConditional` pair of instructions (abbreviated as `lr-sc`) supports the same synchronization algorithms as `CompareAndSwap`; avoids the implementation difficulties of reading more than two registers; and avoids the need to read, modify, and write memory in a single complex instruction. `LoadReserved` (`lr`) acts like a normal load, but it also flags that the memory address is reserved. `StoreConditional` (`sc`) attempts to write a value to a memory address. If the address is still reserved, the write succeeds, and `sc` returns a success code (e.g., 0). If the reservation is no longer valid, the store is rejected, and `sc` returns a failure code (e.g., nonzero). The reservation is invalidated by another `lr` to a different address, any write to the reserved address, or an `sc` to any address. In other words, the `lr-sc` pair only completes the store and returns success if the pair operates in an atomic-like fashion, but across two separate instructions. If the pair fails, it can be repeated until it succeeds. This raises a potential of livelock, addressed in [Section 17.1.5.2](#). [Code Example 17.9](#) shows how to implement other atomic operations and a spin lock using `lr-sc`.

RISC-V defines the *Zacas* (*atomic compare and swap*) extension that defines a `CompareAndSwap` instruction. `amocas.w rd, rs1, rs2` performs the following:

```
temp = mem[rs1]
if (mem[rs1] == rd)
    mem[rs1] = rs2
rd = temp
```

This instruction requires pipeline modifications because it reads `rd` as well as `rs1` and `rs2`. There is also an `amocas.d` double-wide version that operates on consecutive pairs of registers for RV32.

MIPS introduced `lr-sc` instructions. RISC-V, ARM, and other RISC architectures provide similar instructions.

Code Example 17.9 IMPLEMENTING ATOMIC OPERATIONS WITH **lr-sc**

```
int TestAndSet(int *adr) {
    int val;
    while (1) {          // Repeat until lr-sc succeeds atomically
        val = LoadReserved(adr);
        if (StoreConditional(adr, 1) == 0) return val;          // Return if successful
    }
}

int Swap(int *adr, int swap) {
    int val;
    while (1) {          // Repeat until lr-sc succeeds atomically
        val = LoadReserved(adr);
        if (StoreConditional(adr, swap) == 0) return val;      // Return if successful
    }
}

int FetchAndIncrement(int *adr) {
    int val;
    while (1) {          // Repeat until lr-sc succeeds atomically
        val = LoadReserved(adr);
        if (StoreConditional(adr, val+1) == 0) return val;      // Return if successful
    }
}

int CompareAndSwap(int *adr, int comp, int swap) {
    int val;
    while (1) {          // Repeat until lr-sc succeeds atomically
        val = LoadReserved(adr);
        // Return value read from memory if it doesn't compare or if swap succeeds
        if (val != comp | StoreConditional(adr, swap) == 0) return val;
    }
}

void lock(int *L) { // Spin lock using lr-sc
    int val;
    while (1) {          // Repeat until lr-sc succeeds atomically and finds lock was zero
        while (LoadReserved(L)); // Spin until lock is free
        if (StoreConditional(adr, 1) == 0) return;              // Return if successful
    }
}
```

Ticket Locks

A *ticket lock* aims for fairness by ensuring each thread gets the lock in the order it was requested. The lock consists of two parts: turn and ticket. The ticket counts up, like a ticket at a deli counter. Each thread that requests the lock takes the next value of the ticket. The turn also counts up, indicating which ticket is presently being served, like the “Now Serving” sign at the deli counter. Each thread that releases the lock increments the turn to make the lock available to the next ticketholder. [Code Example 17.10](#)

shows an implementation of a ticket lock using `FetchAndIncrement` or `lr-sc` [Arpaci-Dusseau19]. As with the spin lock, taking a lock requires an atomic operation, but releasing it does not.

Code Example 17.10 TICKET LOCK

```
typedef struct lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *L) {
    L->ticket = 0; L->turn = 0;
}

void lockWithFetchAndIncrement(lock_t *L) {
    int myTurn = FetchAndIncrement(&L->ticket);
    while (L->turn != myTurn); // Spin until it is my turn
}

void lockWithLRSC(lock_t *L)
    int myTurn;
    do { // Repeat until lr-sc succeeds atomically
        myTurn = LoadReserved(&L->ticket);
    } while (StoreConditional(&L->ticket, myTurn+1));
    while (L->turn != myTurn); // Spin until it is my turn
}

void unlock(lock_t *L) {
    L->turn++;
}
```

Sleep Locks

Ticket locks are fair but still inefficient because threads spin while waiting. A better strategy would be for a thread to go to sleep rather than spin when the lock is busy, then wake up when its turn has arrived. This requires coordination with the OS, which manages thread scheduling. It also involves maintaining a queue of sleeping threads to reawaken in the order they requested the busy lock. [Sections 17.1.4.3 and 17.1.4.4](#) examine implementing a *sleep lock* with a semaphore, which keeps track of the number of resources available, or a *futex*, which is similar to a semaphore but with reduced overhead. Both are described further in this section.

17.1.4.2 Producer-Consumer: Condition Variables

Locks solve the problem of mutual exclusion, allowing only one thread at a time to enter a critical section of code and manipulate a shared variable. The other common synchronization problem is known as *producer-consumer* or *bounded buffer*, in which *producer* threads *put* results into

a finite-sized buffer and *consumer* threads *get* the results from the buffer. For example, the Apache web server receives and processes many requests simultaneously. Producer threads receive the HTTP requests and put them in a queue. Consumer threads get the queued requests and return HTML web pages.

When a consumer thread finds the buffer empty, it should *wait* by going to sleep until it gets a signal that a new result is in the buffer. Likewise, if a producer thread finds the buffer full, it should wait until it gets a signal that the buffer has room. This coordination is called *handshaking*. *Condition variables* are used to synchronize producer and consumer threads.

Code Example 17.11 shows producer-consumer code without synchronization. One or more producer threads and one or more consumer threads share a *circular buffer* of `SIZE` entries. Each producer puts n integers (0 through $n-1$) into the buffer, and each consumer gets and prints those n integers.

Code Example 17.11 PRODUCER-CONSUMER PROBLEM

Shared Buffer

```
int buf[SIZE];
int head = 0, tail = 0, entries = 0;
```

T1: Producer

```
void put(int val) {
    buf[head] = val;
    head = (head + 1) % SIZE;
    entries = entries + 1;
}

void producer(int n) {
    for (int i = 0; i < n; i++) {
        // Wait until buffer has space
        while (entries == SIZE);
        // Insert result in buffer
        put(i);
    }
}
```

T2: Consumer

```
void get(void) {
    int val = buf[tail];
    tail = (tail+1) % SIZE;
    entries = entries - 1;
    return val;
}

void consumer(int n) {
    for (int i = 0; i < n; i++) {
        // Wait until buffer is nonempty
        while (entries == 0);
        // Get result from buffer
        printf("%d\n", get());
    }
}
```

If the system has a single producer thread and a single consumer thread, it may print the values from 0 to $n-1$ in order. But this code can fail in several ways. Suppose the producer and consumer are running on a single core and the buffer has $k > 0$ entries. The producer thread calls `put` and reads `entries = k`. Before updating `entries` to $k+1$, the scheduler

unfortunately context switches to the consumer thread, which calls `get` and decrements `entries` to `k-1`. Then the scheduler context switches back to the producer, which finishes `entries = entries + 1` using the old value of `entries = k`, updating `entries` to `k+1`. Even though the producer put one value in the buffer and the consumer took one value out, the system now believes it has `k+1` entries rather than `k` entries. A *synchronization failure* has occurred. The consumer could now pull one too many values from the buffer.

Another issue could occur when the system has multiple producer threads. While the buffer is full, all producers must wait. When a consumer gets an entry, two or more producers might concurrently detect that `entries < SIZE` and stop waiting, and both put values into the buffer, causing the buffer to overflow. This is another synchronization failure. A similar problem exists for consumers checking that the buffer is empty.

Yet another drawback of [Code Example 17.11](#) is that the producers and consumers spin while waiting on a full or empty buffer, wasting CPU time.

The producer-consumer problem can be synchronized with condition variables, as shown in [Code Example 17.12](#) [Arpaci-Dusseau19]. We keep the same shared buffer and `put` and `get` functions from [Code Example 17.11](#) but rewrite the producer and consumer functions to add synchronization. We add a lock `L` and condition variables `putEntry` and `getEntry`.

Synchronization failures depend on the relative times events occur in different threads and can lead to erratic failures that are difficult to reproduce. It can be helpful to reason about possible failures by assuming a *malicious scheduler* that context switches at the worst possible time.

Code Example 17.12 PRODUCER-CONSUMER WITH CONDITION VARIABLES

Synchronization Variables

```
int putEntry, getEntry;
int L = 0;
```

```
void producer(int n) {
    for (int i = 0; i < n; i++) {
        // Wait until buffer has space
        lock(&L);
        while (entries == SIZE)
            cond_wait(&getEntry, &L);
        // Insert result in buffer
        put(i);
        cond_signal(&putEntry);
        unlock(L);
    }
}
```

```
void consumer(int n) {
    for (int i = 0; i < n; i++) {
        lock(&L);
        // Wait until buffer is nonempty
        while (entries == 0)
            cond_wait(&putEntry, &L);
        // Get result from buffer
        printf("%d\n", get());
        cond_signal(&getEntry);
        unlock(&L);
    }
}
```

The producer(s) and consumer(s) use the lock `L` to prevent more than one producer or consumer from being in the critical section at a time. Therefore, only one at a time will modify the bounded buffer, preventing synchronization problems.

If the producer finds that the buffer is full, rather than spinning, it calls `cond_wait(&getEntry, &L)`. `cond_wait` is an OS call whose arguments are the addresses of a condition variable and a lock. It places the condition variable and lock addresses in a wait queue, releases the lock, and schedules the thread to sleep. When the OS receives a `cond_signal` call for a condition variable, it reacquires the lock and wakes up the next thread in the wait queue.

When the producer wakes back up, it checks the `while` loop condition once more to ensure the buffer really has space. (Another thread might have swooped in and refilled the buffer before this producer woke up.) Upon success, it puts the next element into the buffer and signals the `putEntry` condition variable to wake up any consumer that might be sleeping. Finally, it releases the lock.

17.1.4.3 Semaphores

A *semaphore* is another synchronization mechanism that can implement both locks and condition variables [Dijkstra68]. A semaphore is an integer that is initialized to the number of shared resources available. The semaphore is atomically decremented or incremented using the OS calls `sem_wait` and `sem_post` to take or release resources, respectively. If the semaphore becomes negative because no resource is available, `sem_wait` puts the thread to sleep. If any threads are sleeping, `sem_post` wakes one.

Semaphore-Based Mutex

Code Example 17.13 shows how to build a semaphore-based mutual exclusion lock (mutex), also called a sleep lock with a semaphore. The semaphore `S` is initialized to 1 because a single thread can hold the lock. The lock is acquired by calling `sem_wait`, which decrements `S` to 0. If another thread tries to acquire the lock that is already held, it will decrement `S` to a negative value and go to sleep until the lock becomes available. Unlocking with `sem_post` increments `S` and wakes up a sleeping thread that was waiting for the lock.

Code Example 17.13 SLEEP LOCK USING A SEMAPHORE

```
int S = 1; // Lock is initially available

void lock(int *S) {
    sem_wait(S);
}

void unlock(int *S) {
    sem_post(S);
}
```

Semaphore-Based Producer-Consumer

Code Example 17.14 modifies Code Example 17.12 to solve the producer-consumer synchronization problem with semaphores instead of condition variables [Arpaci-Dusseau19]. It replaces the `entries` variable, the lock `L`, and the `putEntry` and `getEntry` condition variables with two semaphores: `empty` and `full`. It initializes `empty` to `SIZE` and `full` to 0 because all the buffer entries are empty, and none are full. The producer waits on `empty` to ensure an empty entry is available. Initially, `empty` is positive, so the producer can proceed. When the buffer becomes full, `empty` becomes negative and the producer sleeps until a consumer posts. `put` and `get` are critical sections of code, so they must be surrounded by `lock` and `unlock` so only one thread can manipulate the shared buffer variables at a time.

Code Example 17.14 SEMAPHORE-BASED PRODUCER-CONSUMER

Shared Buffer

```
int buf[SIZE];
int head = 0, tail = 0;
int empty = SIZE, full = 0; // Initialize semaphores
int L = 0; // Lock
```

T1: Producer

```
void put(int val) {
    buf[head] = val;
    head = (head + 1) % SIZE;
}

void producer(int n) {
    for (int i = 0; i < n; i++) {
        // Wait until buffer has space
        sem_wait(&empty);
        lock(&L);
        put(i);
        unlock(&L);
        sem_post(&full);
    }
}
```

T2: Consumer

```
void get(void) {
    int val = buf[tail];
    tail = (tail+1) % SIZE;
    return val;
}

void consumer(int n) {
    for (int i = 0; i < n; i++) {
        // Wait until buffer is nonempty
        sem_wait(&full);
        lock(&L);
        printf("%d\n", get());
        unlock(&L);
        sem_post(&empty);
    }
}
```

17.1.4.4 Futex

Semaphores use OS calls, which have a high overhead of context switching and saving/restoring registers. For applications such as multithreaded

Linux and Windows provide `futex` capabilities, which are currently the most efficient synchronization primitives.

databases that need extensive locks but rarely face contention, the OS call overhead is enormous. A *futex* [Franke02] solves this problem by attempting to take a lock in U-mode and only calling the OS if the lock is contended. Futex initially stood for *fast user mutex* but can also be used to address producer-consumer synchronization like a condition variable or semaphore.

A futex library provides `futex_wait` and `futex_wake` system calls, similar to `sem_wait` and `sem_post`. `futex_wait` puts a thread to sleep until it is notified. `futex_wake` notifies a sleeping thread to wake up. Futex operations attempt synchronization with atomic operations in U-mode and only call the OS if the synchronization spins too long. This is much faster than an OS call in the uncontended case; it also avoids inefficient lengthy spinning. In comparison, a semaphore always involves an OS call even in the uncontended case.

`futex_wait(int *F, int expected)` takes the address `F` of an integer futex variable and an expected value. If the futex variable does not match the expected value, it returns immediately. Otherwise, the calling thread goes to sleep until woken by `futex_wake`.

`futex_wake(int *F, int num)` wakes `num` threads waiting for the futex `F`. `num` is usually 1 to wake up one waiter but sometimes is set to `INT_MAX` to wake up all waiting threads.

Code Example 17.15 shows how to implement sleep locks with futex [Lockless23]. A lock is still an integer at some address `L` in memory. It is initialized to 0 and then set to 1 when a single thread has the lock or 2 when one thread has the lock and another wants it. The `lock()` function uses `CompareAndSwap` to atomically set the lock to 1 if it was 0. If it doesn't succeed immediately, it spins for a little while in the hopes that the lock becomes available soon. If that fails, it sets the lock to 2 to indicate contention, then calls `futex_wait` to sleep until the lock is available. When `futex_wait` returns, the `lock()` function checks that the lock really was available (`c == 0`) and waits again if not. The `unlock()` function checks if the lock was contended. If not contended, it tries to change the lock from 1 to 0 and returns if successful. Otherwise, the code calls `futex_wake` to wake up one of the sleeping threads that is waiting for the lock.

Code Example 17.15 LOCKS USING FUTEX

```

void lock_init(int *L) {
    *L = 0;
}

void lock(int *L) {
    int c;

    // Spin a little while and try to take lock
    for (int i = 0; i < 100; i++)
        if ((c = CompareAndSwap(L, 0, 1)) == 0) return;

    // The lock is now contended
    if (c == 1) c = Swap(L, 2);

    while (c) { // Wait in the kernel while futex is 2
        futex_wait(L, 2);
        c = Swap(L, 2);
    }
}

void unlock(int *L) {
    // Unlock, and if not contended then exit
    if (*L == 2) *L = 0;                // Contended
    else if (Swap(L, 0) == 1) return;   // Uncontended
    futex_wake(L, 1);                  // Wake up 1 waiting thread
}

```

17.1.4.5 Synchronization Libraries

Although this chapter has explored building synchronization operations from low-level atomic instructions and fences, real programs should use standard synchronization libraries. Standard libraries have many advantages. It is easy to identify the synchronization taking place. The code is portable across machines. They include the necessary fences for a particular CPU's memory consistency model. Most importantly, standard libraries have been thoroughly debugged. A study of 12 major programs such as Apache Web Server, Mozilla Firefox, and MySQL discovered that every one of the programs had at least some ad-hoc synchronization and that 22%–67% of the ad-hoc synchronization contained undetected bugs [Xiong10].

The POSIX thread library is a widely-used standard for creating and synchronizing threads in C/C++. It defines locks, condition variables, and semaphores:

- ▶ pthread_mutex_lock
- ▶ pthread_mutex_unlock
- ▶ pthread_cond_wait

- ▶ `pthread_cond_signal`
- ▶ `sem_wait`
- ▶ `sem_post`

Linux defines the `SYS_futex` call to perform the scheduling portions of futex operations.

There is growing interest in lock-free synchronization algorithms, but these are outside the scope of this chapter.

17.1.5 Deadlock, Livelock, and Starvation

Concurrent threads can get stuck for several reasons: deadlock, livelock, and starvation. *Deadlock* occurs when two or more threads are waiting for each other and are unable to proceed. *Livelock* is closely related to deadlock; the threads continue taking action but never make forward progress. *Starvation* occurs when a lower priority thread never gets resources because it is outcompeted by higher priority threads. These problems lead to the familiar annoyance of computers locking up and requiring a reboot.

17.1.5.1 Deadlock

Deadlock is illustrated by the classic “dining philosophers problem” [Dijkstra71]. Five philosophers sit around a circular table. Each has a plate containing a difficult kind of spaghetti that must be eaten with two forks. There is a fork between each plate, as shown in Fig. 17.2. When a philosopher gets hungry, she picks up the fork on the left, then the fork on the right, then eats. Once satisfied, she puts down the fork on the left and then the fork on the right. She faces a dire problem that if all her colleagues become hungry simultaneously, they will all pick up their left forks, leaving no right fork. They are now deadlocked, unable to get a right fork and unwilling to put down the left fork.

Deadlock requires four conditions, referred to as *Coffman Conditions*, to occur [Coffman71]:

- ▶ **Mutual exclusion:** Threads take exclusive control of resources they require.
- ▶ **Hold and wait:** Threads hold allocated resources while waiting for additional resources.
- ▶ **No preemption:** Resources cannot be forcibly removed from a waiting thread.
- ▶ **Circular wait:** A circular chain of threads exists such that each holds a resource requested by the next thread.

In the dining philosophers problem, philosophers are threads, and forks are resources. The problem has mutual exclusion because only

In a later retelling, the philosophers are eating rice with pairs of chopsticks. In a post-COVID retelling, no sane philosopher would agree to share utensils.

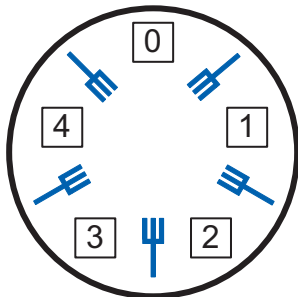


Fig. 17.2 Dijkstra dining philosophers problem.
[Dijkstra71]

one philosopher can hold a fork at a time. It has hold and wait because a philosopher won't set down a fork until done eating. It has no preemption because a hungry philosopher won't wrestle a fork out of her colleague's hand. And it has circular wait because each philosopher could be holding a left fork and waiting for a right fork held by the next philosopher.

Concurrent programs face the same issue. Locks are a common example of mutual exclusion. Suppose two threads in a simple system both want to print documents. They each want a lock on the printer and on a spooling file on the disk. If one thread gets the lock on the printer and the other gets the lock on the spool file and neither can preempt the other, they will deadlock. The threads become unresponsive until the user kills them or reboots the computer.

A better system can break deadlocks by preventing one of the Coffman Conditions using the following strategies:

- ▶ **Mutual exclusion:** Threads use lock-free data structures, when possible, to minimize the need for exclusive control.
- ▶ **Hold and wait:** Each thread requests all its resources at once and does not proceed until all resources have been granted.
- ▶ **No preemption:** If a thread is denied access to a resource, it releases all the resources it already holds (so that another thread can proceed), then tries again later to obtain all resources.
- ▶ **Circular wait:** Threads are required to acquire resources in a certain order so that circular chains can't exist.

For example, the dining philosophers could set two forks at each plate to avoid the need for mutual exclusion caused by sharing forks. Or the philosophers could pick up both forks at a time, rather than the left and then right. If they pick up a left fork and find the right already in use, they could set down the left fork and try again after pondering deep thoughts for a random amount of time. Or the forks could be labeled 1–5, and a philosopher could pick up the lower-numbered fork before taking the higher-numbered one, avoiding the circular wait.

Other approaches involve:

- ▶ **Avoidance:** When a thread requests a resource, the OS denies the request if granting the resource would put the system in an unsafe state.
- ▶ **Detection and recovery:** The OS detects deadlock and removes resources until the deadlock is solved.
- ▶ **Ignorance:** Assume deadlock will not occur. Let the user reboot the computer when it occurs. This may be acceptable if deadlocks are less frequent than other system failures. The Windows OS is notorious for using this strategy, and even Linux is susceptible to this for certain deadlocks.

*No, you can't always get
what you want*

*You can't always get what
you want*

*You can't always get what
you want*

*But if you try sometime
you'll find*

You get what you need.

—The Rolling Stones

Section 17.2.3.2 defines RISC-V constrained `lr-sc` loops that are guaranteed to eventually succeed and hence avoid livelock.

17.1.5.2 Livelock

Livelock occurs when the threads repeat the same sequence of operations without making real progress. Suppose the dining philosophers break the no-preemption problem by setting down their left fork if the right fork is already in use. However, they are ravenous and immediately pick up the left fork again before their neighbor can take it. They enter livelock, repeatedly picking up and setting down the left fork without ever getting both forks.

A disadvantage of `lr-sc` over `CompareAndSwap` is that the former can enter livelock. Two threads might spin to acquire the same lock. If they exactly interleave, each one might invalidate the other thread's reservation such that neither ever successfully completes the `StoreConditional`.

17.1.5.3 Starvation

Starvation occurs when one or more threads never get the resources they need. For example, if threads have priority and a condition variable or semaphore wakes up the highest priority waiting thread, a lower priority thread might never be woken when the system is busy.

Starvation is avoided by fair synchronization algorithms such as ticket locks (see Section 17.1.4.1) or by raising the priority of a thread that has been waiting too long.

17.2 RISC-V PRACTICES

RISC-V uses the release consistency model to allow maximum flexibility in hardware and software optimizations while ensuring that synchronized programs behave correctly. The base integer instruction set includes fence instructions to synchronize memory operations when needed. The *atomic* (A) extension provides atomic instructions to support synchronization across threads; this extension is necessary for single-core processors with multiple threads as well as for multicore processors.

17.2.1 Consistency Models

RISC-V uses a release consistency model called *RISC-V Weak Memory Ordering* (RVWMO) [RISCV-Unprivileged]. Each thread runs on its own hart. The hart may be a physical core or may be a virtual hart time-multiplexed by the OS onto the physical core. The thread appears to execute in order from the perspective of other memory instructions in the same hart, but memory instructions from another hart may observe the memory instructions from the first hart being executed in a different order. For example, stores can be sent to a write buffer rather than completing immediately. A speculative out-of-order hart may reorder instructions as long as the result is indistinguishable by that hart from executing in program order.

If a system needs a critical section of a given hart's memory accesses to appear in order from another hart's perspective, the given hart must add explicit synchronization through fences or acquire and release. To do so, a hart must use an *acquire* operation before a critical section of code that manipulates shared variables and follow that section with a *release* operation. The acquire operation acquires the lock and ensures that all memory operations in the subsequent critical section are seen by other threads as happening after the lock is acquired. The release operation releases the lock and ensures that all memory operations in the critical section are seen by other threads as happening before the lock is released. Therefore, all threads have a consistent view of the shared variables. The acquire and release operations can be enforced by either a fence instruction or by a more selective acquire or release annotation on an atomic operation, as described in the following subsections.

Recall from [Section 11.2.2.1](#) that a RISC-V core can use physical memory attributes (PMAs) to designate certain address ranges as main memory and others as peripheral I/O. Peripheral reads are called *input*, and peripheral writes are called *output*. The RVWMO can enforce consistency separately for memory and I/O accesses; for example, using a lock to control access to a shared data structure in main memory does not have to wait for an access to a peripheral to complete. [Section 17.2.2](#) defines fences that order memory or I/O accesses relative to one another.

RISC-V optionally supports the *RISC-V Total Store Ordering* (RVTSO) consistency model, which is called the Ztso extension. RVTSO makes it easier to port programs written for other TSO architectures such as x86 or SPARC. It may come for free on simple in-order processors because they never access memory out of program order. The compiled Executable and Linkable Format (ELF) file sets bit 4 of the `e_flags` field in the ELF header if a program requires Ztso to run correctly.

17.2.2 Fences

The `fence` instruction is an I-type instruction listed in Table I.8 (on the last pages of the book). It is part of the RV{32/64}I base instruction set and is supported on all RISC-V processors. Fences can enforce ordering for memory reads (r), memory writes (w), peripheral input (i), and peripheral output (o). A fence takes two arguments that specify which types of accesses before the fence must complete before which types of accesses after the fence may occur.

The 12-bit immediate field specifies which types of accesses are ordered, as shown in [Fig. 17.3](#). The `fm` field is normally 0000. The `PI`, `P0`, `PR`, and `PW` bits define the predecessor set, and the `SI`, `S0`, `SR`, and `SW` bits define the successor set. The fence orders accesses so other peripherals or RISC-V harts cannot see any access in the successor set before it sees all the accesses in the predecessor set. For example, `fence rw, rw` has an

fm	PI	PO	PR	PW	SI	SO	SR	SW
31:28	27	26	25	24	23	22	21	20

Fig. 17.3 fence immediate field format.

Simple in-order pipelined cores with no store buffers, such as Wally, are inherently sequentially consistent and thus don't require fences between memory accesses. `fence` can be treated as a nop on these cores.

immediate field of 0000_0011_0011 (`PR`, `PW`, `SR`, and `SW` are 1), and ensures that all reads and writes to memory before the fence complete before any read or write to memory after the fence completes. For example, placing a fence `rw, rw` between each memory operation ensures that they take place in a sequentially consistent fashion.

The fence pseudoinstruction is a shorthand for fence `iorw, iorw`, which enforces sequential consistency across both memory and I/O peripheral accesses. Weaker fences are also available, as described in Table 17.3. The `fence.tso` instruction is encoded as a fence `rw, rw` with `fm` = 1000. It implies total store order, which is weaker than sequential consistency. Other fences with nonzero `fm` are reserved. Code Example 17.18 gives an example of using a fence in a lock. Similarly, in Code Example 17.3, the producer thread T1 can use a fence `w, w` to ensure other cores see the variables written before `flag` is written to 1, and T2 can use a fence `r, r` to ensure that the flag is read as 1 before the variables are read.

However, peripherals with direct memory access (DMA) capability that can read or write main memory independently of the core introduce memory consistency issues and can require synchronization. Section 9.1.4 describes such synchronization.

Recall that RISC-V also supports `fence.i` (synchronize instruction memory) and `sfence.vma` (synchronize page table). `fence.i` requires the Zifencei extension. It synchronizes the instruction and data caches, as discussed in Sections 10.4.2.4 and 13.3.1.1. `sfence.vma` requires virtual memory support and synchronizes the TLB as described in Section 11.2.4.

When the `menvcfg.FIOM` bit is set, a fence involving the `PI`, `PO`, `SI`, or `SO` bits at a privilege level below M-mode behaves as if the corresponding `PR`, `PW`, `SR`, or `SW` bits are also set. Similarly, `menvcfg.FIOM` modifies I/O fences to also imply memory fences in U-mode. These bits help the OS to emulate virtual I/O devices in software, as described in the RISC-V Privileged Specification Release 20211203, Section 4.1.10.

17.2.3 Atomic Instructions

The RISC-V atomic (A) extension defines two forms of atomic instructions: atomic memory operations (AMOs) and `LoadReserved-StoreConditional` (`lr-sc`), as listed in Table L.24 in the last pages of the book. AMO instructions fetch data, operate on it, and store it back into the same memory address from which it was fetched.

Both forms of atomic instructions, AMOs and `lr-sc`, have acquire and release bits (`aq` and `rl`) that can optionally be set to order the atomic operations relative to other memory accesses within the same hart, without

The `menvcfg.FIOM` bit stands for Fence of I/O implies Memory.

Table 17.3 Common RISC-V fences.

Fence	imm	Meaning
fence iorw, iorw	0xFF	Other cores see all input, output, reads, and writes before the fence complete before any input, output, read, or write after the fence completes. This gives sequential consistency across both memory and I/O accesses.
fence rw, rw	0x33	Other cores see all reads and writes before the fence complete before any read or write after the fence completes. This gives sequential consistency for memory accesses.
fence r, rw	0x23	Other cores see all reads before the fence complete before any read or write after the fence. This is an <i>acquire</i> operation.
fence rw, w	0x31	Other cores see all reads and writes before the fence complete before any write after the fence. This is a <i>release</i> operation.
fence r, r	0x22	Other cores see all reads before the fence complete before any read after the fence. Linux calls this a <i>read memory barrier</i> : <code>smp_rmb()</code> .
fence w, w	0x11	Other cores see all writes before the fence complete before any write after the fence. Linux calls this a <i>write memory barrier</i> : <code>smp_wmb()</code> .
fence.tso	0x833	Other cores see all reads before the fence complete before any read or write after the fence completes and all writes before the fence complete before any write after the fence completes. This gives total store order. A hart can optionally implement <code>fence.tso</code> as <code>fence rw, rw</code> for stronger sequential consistency.

the need for an additional fence. When `aq` is set, no later memory operation in the hart can be observed to take place before this atomic access. When `rl` is set, all previous memory operations in the hart will be observed to take place before this atomic access. When both `aq` and `rl` are set, the atomic access is sequentially consistent and will be observed to occur after all previous accesses and before any subsequent accesses. If neither are set, the atomic operation may be seen in any order relative to other memory operations in the hart. The acquire and release bits do not enforce ordering of memory relative to I/O operations; `fence` should be used if this is necessary.

17.2.3.1 Atomic Memory Operations

The atomic extension provides a family of AMOs that read a value from memory, modify it, and write it back to memory in one uninterruptible step. AMOs are also referred to as `FetchAndOp` because they fetch data from memory and perform an operation. They are not as flexible as `lr-sc` but are simpler, do not livelock, and perform better on highly parallel systems because they involve a single instruction and always succeed.

AMOs are of the form `amo<op>.w rd, rs2, (rs1)`. The operation `<op>` is one of `swap`, `add`, `and`, `or`, `xor`, `max`, `maxu`, `min`, or `minu`. These instructions read a value from memory address `rs1` into `rd`. They also compute

RV64 also supports `.d` double-word AMOs.

(rs1) op rs2 and put the value back into the same address. For example, if $a0 = 1000$, $t0 = 2$ and memory address 1000 contains 5, then `amoadd.w s0, t0, (a0)` reads 5 from address 1000, puts 5 into `s0`, and writes the value $2 + 5 = 7$ back to memory address 1000. `amoswap` writes the value from `rs2` to memory and puts the original memory value in `rd`.

AMOs can implement the `TestAndSet`, `Swap`, and `FetchAndIncrement` atomic operations from [Section 17.1.4.1](#). Assuming $t1 = 1$, $a0 = \text{adr}$, $a1 = \text{swap}$, and val is in `s0`:

- ▶ `val = TestAndSet(adr)` `amoswap.w s0, t1, 0(a0)`
- ▶ `val = Swap(adr, swap)` `amoswap.w s0, a1, 0(a0)`
- ▶ `val = FetchAndIncrement(adr)` `amoadd.w s0, t1, 0(a0)`

`amoor`, `amoand`, and `amoxor` can be used to atomically set, clear, or toggle bits in memory-mapped device registers.

[Code Example 17.16](#) shows a spin lock using `amoswap` to perform `TestAndSet`. The lock is at address `a0`. It first spins using `lw` until it detects that the lock is free (0). It uses `amoswap` to test if the lock is still free and set it busy. Spinning on `lw` before `amoswap`, called `TestAndTestAndSet`, is more efficient than spinning on `amoswap` because `lw` doesn't stress the shared memory system as much as an atomic operation. The code then enters a critical section and finally releases the lock with another `amoswap` storing 0. The `amoswap` that acquires the lock is marked `aq` to ensure no later memory operations in the critical section are seen to take place before the lock is acquired. Similarly, the `amoswap` releasing the lock is marked `rl` to ensure that all memory operations in the critical section are seen to take place before the lock is released. If the `aq` and `rl` were not specified, accesses to shared variables in the critical section might appear to happen when the lock is not held, resulting in improper synchronization.

Code Example 17.16 SPIN LOCK USING `amoswap`

```
# Acquire lock
li t0, 1                                # Initialize swap
lock:
lw t1, (a0)                             # Check if lock is held
bnez t1, lock                           # Retry if lock is held
amoswap.w.aq t1, t0, (a0)               # Attempt to acquire lock
                                         # No critical section memory accesses appear before here
bnez t1, lock                           # Retry, if held

#
# Critical Section Code: do stuff with shared variable. . .
#

# Release lock
amoswap.w.rl x0, x0, (a0)               # Release lock by storing 0
                                         # No critical section memory accesses appear after here
```

When an AMO instruction causes an exception, it is reported as a store instead of a load. Thus, the exceptions are formally called store/AMO. The possible exceptions are store/AMO access fault, store/AMO address misaligned, and store/AMO page fault.

17.2.3.2 LoadReserved-StoreConditional

As described in [Section 17.1.4.1](#), an `lr-sc` pair is used to perform complex AMOs. `lr.w rd, (rs1)` loads a word from the memory address specified by `rs1` into `rd` and marks the address as reserved. `sc.w rd, rs2, (rs1)` attempts to store the value in `rs2` into the memory address specified by `rs1`. If the address is still reserved, it succeeds and clears `rd` to 0. If the address is not reserved, `sc` fails (doesn't change memory) and sets `rd` to a nonzero failure code. The reservation is canceled by a write to the reserved address from any hart or by the reserving hart's `sc` to any address.

Strictly speaking, `lr` reserves a block of contiguous addresses rather than a single word. This block is called the *reservation set*. A write to any address within the reservation set cancels the reservation. The reservation set size is often one full cache line, so the reservation is tracked with an extra status bit on the cache line. Cache lines are often 512 bits (64 bytes), so this is a common reservation set size.

[Code Example 17.17](#) shows a RISC-V assembly language implementation of `CompareAndSwap` using `lr` and `sc` based on the C version from [Code Example 17.9](#).

RV64 also supports the `lr.d` and `sc.d` double-word forms.

During a context switch or change of relevant page table entries, the OS invalidates any pending reservations by performing a `StoreConditional` to a scratch address.

Code Example 17.17 COMPAREANDSWAP USING `lr-sc`

```
# int CompareAndSwap(int *adr, int comp, int swap);
# adr is in a0, comp in a1, swap in a2, val in t0, return value in a0
CompareAndSwap:
    lr.w t0, (a0)           # val = LoadReserved(adr)
    bne t0, a1, done        # if (val != comp) return val
    sc.w t1, a2, (a0)       # t1 = StoreConditional(adr, swap)
    bnez t1, CompareAndSwap # Repeat if StoreConditional fails
done:
    mv a0, t0               # Return val
    ret
```

As described in [Section 17.1.5.2](#), `lr-sc` is at risk of livelock if two concurrent threads constantly cancel each other's reservations. RISC-V prevents livelock with an *eventual success guarantee* for *constrained lr-sc loops* [RISCV-Unprivileged]. In simple terms, the execution environment guarantees that one of the threads will eventually successfully store. This

17.2.4 Atomic Subsets

Although the RISC-V atomic (A) extension supports both AMOs and `lr-sc` instructions, AMOs may be preferred because they scale better to highly parallel systems and also are easy to implement on cacheless microcontrollers. The Zaamo extension supports only the AMO instructions of the A extension. The Zalrsc extension only supports the `lr-sc` instructions of the A extension. The A extension implies Zaamo + Zalrsc.

17.2.5 Profile Extensions

The atomic extension imposes few restrictions on the implementation, but operating systems prefer to rely on some sensible expectations. The following extensions are expected in the RVA20U64 and later profiles, as given in Table A.6.

17.2.5.1 Ziccif

An OS normally expects that main memory is suitable for holding programs and that interrupts cannot occur midway through fetching an instruction. The RVA20U64 and later profiles formalize this understanding with the Ziccif extension, which requires that main memory (i.e., memory with cacheable and coherence PMAs) must support instruction fetch. Moreover, the instruction fetch must be atomic (i.e., an interrupt never takes place while an instruction is only partially fetched).

Wally's IFU (instruction fetch unit) can fetch instructions from any memory address. The fetch is atomic; even if a spill occurs, the IFU stalls the rest of the pipeline until the full instruction is fetched. Therefore, Wally inherently supports the Ziccif extension.

17.2.5.2 Ziccrse

As discussed in [Section 17.2.3.2](#), `lr-sc` could livelock in certain multicore systems unless precautions are taken. The *RsvEventual* property indicates that the memory system is designed such that `lr-sc` will eventually succeed. The Ziccrse extension formalizes this by requiring that all main memory has the *RsvEventual* property.

Wally has a single core and meets this property with a reasonable OS scheduler. Therefore, it inherently supports Ziccrse.

17.2.5.3 Ziccamao

As with Ziccrse, the PMA checker can prohibit some or all AMO instructions from accessing a region of memory. The *AMOArithmetic* property indicates that a memory region supports all AMO instructions. The Ziccamao extension requires that all main memory has the *AMOArith* property. Wally's PMA checker defaults to supporting all atomic instructions in main memory, so it supports Ziccamao.

17.2.5.4 Za128rs and Za64rs

RISC-V allows a reservation set of arbitrary size, such that a store to any address within the reservation set cancels a `LoadReserved` to any other address in the set. When the OS allocates memory for a lock, it prefers to have no other variables within the same reservation set so other accesses do not interfere with acquiring the lock. If the reservation set is large, this wastes memory, and if the OS doesn't know the size of the reservation set, it cannot allocate memory effectively.

The `Za128rs` extension tells the OS that the reservation set is no larger than 128 bytes. Similarly, the `Za64rs` extension indicates that the reservation set is no larger than 64 bytes. The `RVA22` profiles expect `Za64rs` to limit wasted memory. 64 bytes is a natural implementation choice because it is a typical cache line size.

17.3 TEST PLAN

At the time of this writing, `riscv-arch-test` has AMO tests listed in [Table 17.4](#), but lacks `lr-sc` tests.

The `tests/wally-riscv-arch-test` directory contains simple `WALLY-lrsc-01` tests. They are located in the `privileged` subdirectory because the test infrastructure does not yet look for an `A` subdirectory. `WALLY-lrsc` performs a successful `LoadReserved` and `StoreConditional` word and double-word operation followed by an unsuccessful operation to check basic functionality.

17.4 WALLY IMPLEMENTATION

Wally is an in-order single-core processor, so it is inherently sequentially consistent. It also has a single instruction cache and single data cache, so caches are inherently coherent except that store operations are not reflected in the instruction cache. Wally needs the `fence.i` instruction for coherence between the instruction and data caches (see [Sections 10.4.2.6 and 10.4.5.4](#)) and `sfence.vma` for TLB coherence (see [Section 11.2.4](#)). In Wally, all other fences are treated as nops, and the acquire/release bits of atomic instructions are ignored.

When Atomic extensions are supported (`ZAAMO_SUPPORTED = 1` and/or `ZALRSC_SUPPORTED = 1`), the LSU (load/store unit) adds an `atomic` module, as shown in [Fig. 17.4](#). This module handles `lr-sc` instructions and AMOs. It is relatively simple because it does not need to handle interactions with other cores. The LSU does not have an FSM to sequence atomic read-modify-write operations over multiple cycles, so the atomic module relies on the LSU having a data cache. The Wally PMA checker only allows atomic operations on cachable main memory. The cache read (`ReadDataWordM`), modify (`IMAWriteDataM`), and write (`LSUWriteDataM`) take place in the same cycle.

Table 17.4 riscv-arch-test for A extension

Instruction	RV32IA Tests	RV64IA Tests
amoadd.w	69	133
amoand.w	69	133
amomax.w	69	133
amomax.w	69	133
amomin.w	69	133
amominu.w	69	133
amoor.w	69	133
amoswap.w	69	133
amoxor.w		133
amoadd.d		133
amoand.d		133
amomax.d		133
amomax.d		133
amomin.d		133
amominu.d		133
amoor.d		133
amoswap.d		133
amoxor.d		133
Total	552	2,394

The `atomic` module takes advantage of the existing LSU hardware to add atomic support with minimal change. `LoadReserved` resembles an ordinary load but adds state to keep track of which address is reserved. `StoreConditional` resembles an ordinary store but is squashed if the address is no longer reserved. The LSU already can perform a read at the start of the Memory stage and a write at the end of the Memory stage. AMO operations take the read result, perform an ALU operation on it in the `atomic` module, and mux this ALU result back into the write path. Specifically, the LSU reads `ReadDataM` at the start of the Memory stage.

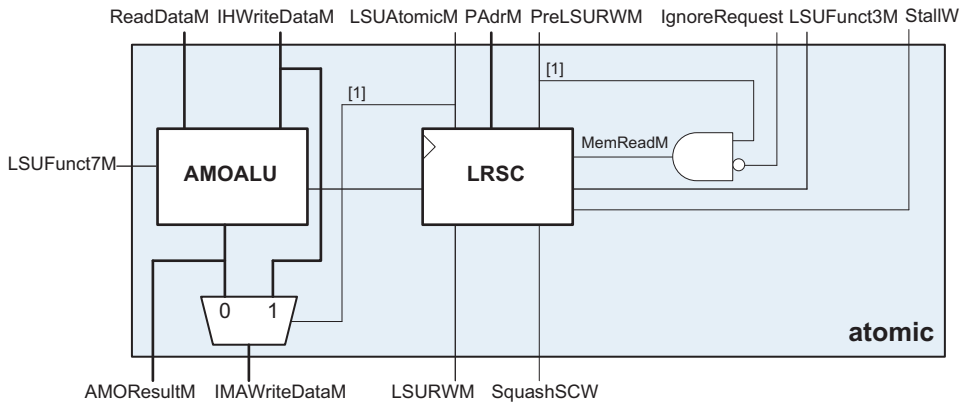


Fig. 17.5 atomic module.

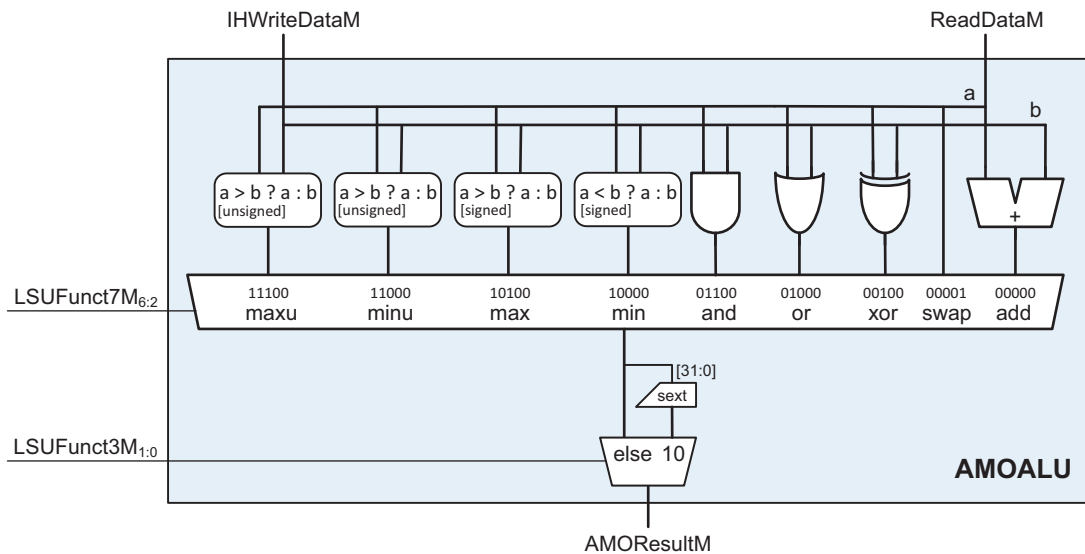


Fig. 17.6 amoa1u module.

17.4.1.1 AMO ALU

Fig. 17.6 shows the AMO ALU (*amoalu*). It performs an operation on *ReadDataM* and *IHWriteDataM* to produce *AMOResultM*. RV64 provides sign-extension multiplexers for 32-bit word operations. Although *amoalu* has many similarities with the integer ALU, it operates in the Memory stage rather than the Execute stage, because it needs the read memory data as an operand. The AMO ALU also provides max/min operations. Note that the four comparators are closely related and can use a shared carry chain. The

AMO ALU uses dedicated hardware rather than sharing the integer ALU because sharing would add busses and multiplexers between the IEU and LSU and would introduce a dependency that would stall the Execute stage when an AMO instruction needed the shared ALU. The extra hardware is modest in the context of a full-featured application core.

17.4.1.2 LRSC module

Fig. 17.7 shows the LRSC module. During a `LoadReserved` instruction, the module records the physical address of the reservation and marks the reservation valid. Any `StoreConditional` instruction invalidates the reservation. If the `StoreConditional` does not match the physical address of a valid reservation, the LRSC module asserts `SquashSCM` to squash the store. This squash turns off the `LSURWM` write enable and writes a 1 to the destination register to indicate failure. Fig. 17.8 shows how the IEU `Result` multiplexer (from Fig. 7.13) takes an additional input for the `StoreConditional` result, highlighted in blue.

17.4.1.3 Atomic Hazards

As described in Section 10.4.2.3, the cache is unable to perform a load immediately after a store because both instructions would need to use the data SRAM's address in the same cycle. For the same reason, the cache cannot perform an AMO after a store or AMO or before a load. The `StoreStallD` logic that detects load after store also detects these AMO cases because they use the same `MemRW` signals, so no extra hardware is needed.

Observe that this implementation tracks the reservation with an address register rather than a bit on each cache line. Therefore, the size of the reservation set is the native word size (4 bytes on RV32 and 8 bytes on RV64).

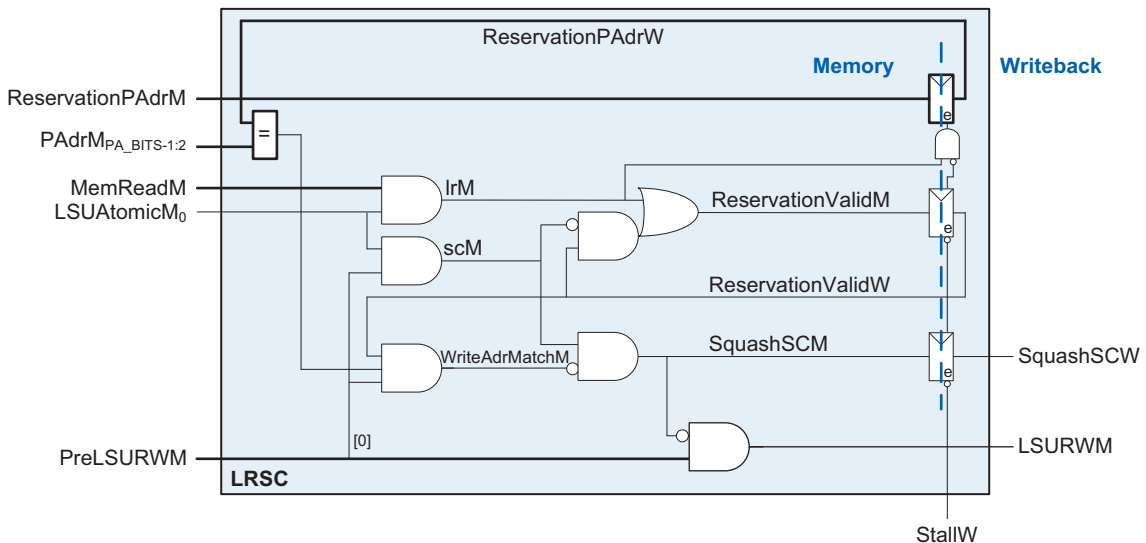


Fig. 17.7 lrsc module.

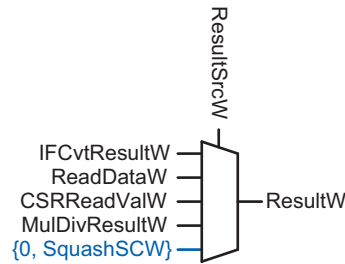


Fig. 17.8 IEU Result multiplexer supporting `StoreConditional`.

17.4.2 Timing Diagrams

The timing for atomic operations is similar to other LSU operations. They rely on a data cache in the LSU. Loads, including `LoadReserved`, read the cache at the start of the Memory stage. Stores, including `StoreConditional`, write the cache at the end of the Memory stage. AMOs read memory at the start of the Memory stage, perform an operation, and write the result back to the cache at the end of the Memory stage.

Code Example 17.19 exercises `LoadReserved`, successful and unsuccessful `StoreConditional`, and an AMO. Assume the program runs on an RV32IA machine in little-endian mode with virtual memory disabled, so the virtual and physical addresses are the same, and the subword and endian swap blocks do not modify the data. Assume memory location `0x80001000` initially contains 42.

The program performs a `LoadReserved` to address `0x80001000`, reading the value 42 and reserving the address. It then performs a `StoreConditional` back to the same address, writing the value 43 and clearing the reservation. The second `StoreConditional` fails, returning a success code of 0. It attempts to repeat the `StoreConditional`, now squashing the store and getting a failure code of 1 because the reservation is no longer valid. Finally, it performs an `amoadd` at the same address, reading 43 and writing 45.

Code Example 17.19 ATOMIC PROGRAM

```

li a0, 0x80001000      # a0 = 80001000
li a1, 2               # a1 = 2
lr.w t0, (a0)          # t0 = mem[80001000] = 42; reserve 80001000
addi t0, t0, 1         # t0 = 43
sc.w t1, t0, (a0)      # Successful StoreConditional to 80001000; t1 = 0
sc.w t2, t0, (a0)      # Unsuccessful StoreConditional to 80001000; t2 = 1
amoadd.w s0, a1, 0(a0) # s0 = 43; mem[80001000] = 45

```

Fig. 17.9 shows the waveforms while running this example. `MemRWM` is 10 for loads, 01 for stores, and 11 for AMO. `LSUAtomicM` is 00 for

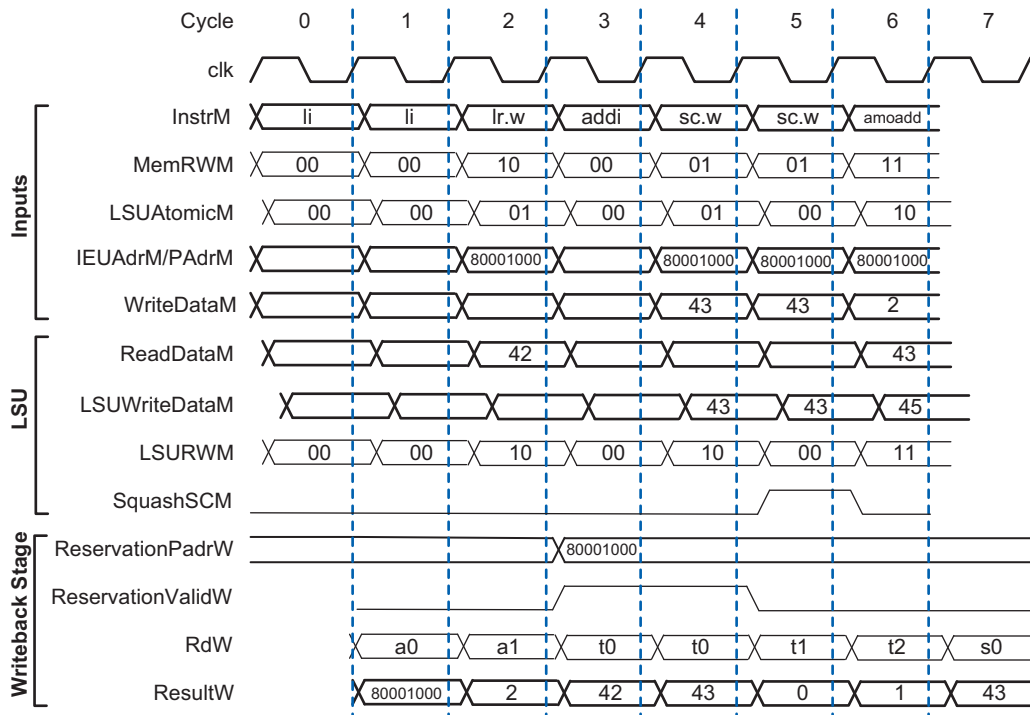


Fig. 17.9 Atomic waveforms.

nonatomic operations, 01 for `lr-sc`, and 10 for AMOs. The first `StoreConditional` succeeds in cycle 4, writing 0 to `t0` and clearing `ReservationValidW` in cycle 5. The second `StoreConditional` in cycle 5 finds the reservation is no longer valid and fails. It raises `SquashSCM`, clears `LSURWM` to prevent storing, then writes 1 to `t0` in cycle 6. The `amoadi` in cycle 6 reads 43 from memory and 2 from `a1`. It adds these operands and writes 45 back to memory, then returns 43 to `s0`.

17.5 SUMMARY

Modern computers run processes concurrently by time-multiplexing processes and threads on a single core and/or using multiple cores. Programmers want coherence, consistency, and synchronization in concurrent systems. Coherence means that all processes eventually see the same value stored to memory. Consistency describes the ordering of reads and writes with respect to different locations. Programmers intuitively expect a sequential consistency model, which means that each thread performs reads and writes in program order but that different threads may interleave in arbitrary ways. Unfortunately, guaranteeing sequential consistency precludes important hardware optimizations such as write buffers,

so high-performance systems use more relaxed total store order or release consistency models. Under release consistency, the programmer must explicitly insert fences or acquire and release operations when a certain ordering is necessary to synchronize multiple threads. Common synchronization requirements and techniques are as follows:

- ▶ **Mutual exclusion (locks):** Only one thread can access a shared variable at a time.
- ▶ **Producer-consumer (condition variables):** Threads communicate by putting and getting information in a bounded buffer and waiting when the buffer is empty or full.

Processors provide atomic instructions to facilitate synchronization. The RISC-V atomic (A) extension defines `LoadReserved-StoreConditional` (`lr-sc`) and atomic memory operations (AMOs). RISC-V has a release consistency model, so it also defines a set of fences to enforce memory consistency. Wally is an in-order single-core processor, so it is inherently sequentially consistent. However, it still needs atomic instructions to synchronize multiple threads. It also uses `fence.i` to maintain coherence between the instruction and data caches and `sfence.vma` for TLB coherence.

Extensions: Zb* & Zk* (Bit Manipulation and Cryptography)

- 18.1 **Cryptography Principles**
- 18.2 **RISC-V Practices**
- 18.3 **Test Plan**
- 18.4 **Wally Implementation**
- 18.5 **Summary**

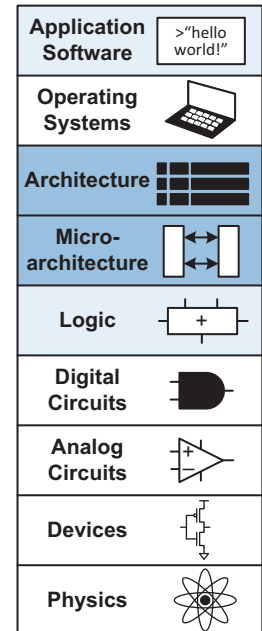
RISC-V defines bit manipulation (Zb*) and cryptography (Zk*) extensions. Bit manipulation instructions are an extension to ordinary integer instructions. They include rotation, setting or clearing bits, and counting the number of 1s in a register. Although programs can be written without bit manipulation instructions, competing computer architectures provide them to improve code speed and size at the expense of a modest amount of new hardware. For example, [Section 21.2.3](#) shows these extensions make Embench 5% faster and 4% denser than an rv32imc baseline. In [Section 21.1.3](#), CoreMark is 8% faster with bit manipulation instructions.

Cryptography is critical for maintaining security. Consider Alicia (A) and Ben (B) attempting to communicate. The key tasks of cryptography are:

- ▶ **Secrecy:** Only A and B can see what is sent.
- ▶ **Integrity:** What A sends is what B receives.
- ▶ **Authenticity:** Nobody can impersonate A or B.

The most widely used cryptographic operations are *encryption* and *decryption* for secrecy, *hashing* for integrity, and *signatures* for authenticity. Pure software implementations are slow, power hungry, and vulnerable to timing attacks that can be exploited remotely. Modern instruction sets provide dedicated cryptography instructions that are faster, simpler, and provide better performance than pure software implementations. Moreover, having cryptographic instructions promotes standardized software and reduced code size, which helps reduce the risk of inadvertent security flaws. RISC-V provides the Zk* family of cryptographic extensions to support existing standardized cryptographic algorithms and practices for encryption, decryption, and hashing. Cryptography extensions are discussed in the same chapter as bit manipulation because they share the same ALU and have some instructions in common.

This chapter discusses cryptography principles, including finite field arithmetic and its applications for Advanced Encryption Standard (AES)



Cryptography is also referred to simply as crypto, not to be confused with cryptocurrency.

Claude Shannon (1916–2001) originally thought of cryptography in terms of their *confusion* and *diffusion* [Shannon45] and later expanded this into his communication theory of secrecy systems [Shannon49]. Confusion involves obscuring the input (plaintext) from the output (ciphertext) by applying the key to the data. Diffusion deals with spreading information across the elements of the cipher. Shannon's papers are associated with the birth of cryptography for computing. Shannon also developed the theory of digital logic design in his MIT master's thesis, built the first machine learning device, and founded the field of information theory.



(Photo © 2012. This image is openly licensed under CC BY 2.0.)

encryption and Secure Hash Algorithm (SHA) hashing. It then details the RISC-V bit manipulation and cryptography extension instructions. Finally, it summarizes the test plan and shows how Wally extends the ALU to support these extensions.

18.1 CRYPTOGRAPHY PRINCIPLES

Cryptography is the science of hiding the meaning of messages. Although it has gained interest in recent decades for computer security, it has been around at least since Julius Caesar wrote B in place of A to prevent the Gauls from reading his messages to his generals. Cryptography uses many primitives, including symmetric ciphers, asymmetric (also called public key) ciphers, hash functions, and cryptographic protocols. This text mainly focuses on symmetric ciphers and hash functions as they pertain to the RISC-V Zk* extension. [Ferguson10] offers a good general introduction to cryptography.

Symmetric ciphers use the same *secret key* for both encryption and decryption. It is an efficient way to encrypt bulk data. A substitution cipher is an example of a simple symmetric cipher, where each letter of the alphabet is substituted with another one k letters later in the alphabet. The key k is needed for encryption and decryption. For example, encrypting WALLY with the key 5 gives $W \rightarrow B, A \rightarrow F, L \rightarrow Q, L \rightarrow Q, Y \rightarrow D$ or BFQQD. More formally, the substitution cipher with key k encrypts the input (*plaintext*) message one character x at a time by outputting a *ciphertext* message $(x + k) \bmod 26$ one character at a time. This cipher is easy to crack by trying all 26 possible keys.

Fig. 18.1 shows a symmetric-key encryption system that encrypts a plaintext message with a key to produce the ciphertext. In a good system, deducing the plaintext from the ciphertext without the key is impractically difficult. Many such systems are constructed using finite field arithmetic.

18.1.1 Finite Fields

In mathematics, a *field* is a set that supports the operations of multiplication and addition. A field has additive and multiplicative *identity elements*, 0 and 1, respectively. Table 18.1 describes properties that apply to any elements a , b , and c of the field. Each element a has an *additive inverse* $-a$,

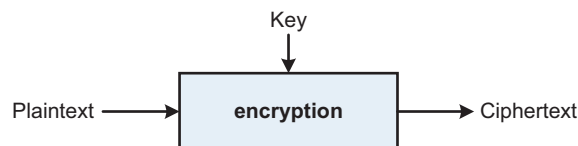


Fig. 18.1 Symmetric-key encryption.

Table 18.1 Properties of fields.

Property	Addition	Multiplication
Closure	$a + b$ is in the field	$a \times b$ is in the field
Identity	$a + 0 = a$	$a \times 1 = a$
Inverse	$a + (-a) = 0$	if $a \neq 0$, $a \times a^{-1} = 1$
Commutativity	$a + b = b + a$	$a \times b = b \times a$
Associativity	$(a + b) + c = a + (b + c)$	$(a \times b) \times c = a \times (b \times c)$
Distributivity	$(a + b) \times c = (a \times c) + (b \times c)$	

and each element other than 0 has a *multiplicative inverse* a^{-1} . For example, the set of real numbers $\mathbb{R}$ forms a field because it satisfies these properties. The sum or product of any two real numbers is another real number, and every real number has an additive and a multiplicative inverse. The additive inverse of 42 is -42 , and the multiplicative inverse of π is $1/\pi$. Similarly, rational numbers (numbers that can be written as a fraction x/y , where x and y are integers) are also a field. On the other hand, integers are not a field because 3 has no multiplicative inverse (i.e., 3 cannot be multiplied by another integer to obtain 1).

Fields are the building block of many cryptographic algorithms that involve multiplication and addition. Computers use data types with finite precision, so, unsurprisingly, cryptography does too. A *finite field* or *Galois field* is a set with a finite number of elements.

The two Galois fields most used in cryptography are:

- ▶ $\text{GF}(p)$: integer arithmetic modulo some prime number p
- ▶ $\text{GF}(2^n)$: arithmetic on n -bit binary numbers where, as described later in this section, addition is performed as bitwise XOR and multiplication overflow is handled with XOR

18.1.1.1 $\text{GF}(p)$

$\text{GF}(p)$ contains the integers $0, 1, \dots, p - 1$, where p is prime. Operations of this field are addition and multiplication modulo p . The modulo operation causes results to wrap around and, thus, remain in the field, providing closure. For example, in $\text{GF}(7)$, $5 + 6 = (11 \bmod 7) = 4$. The additive inverse of 2 (referred to as “ -2 ”) is 5 because $2 + 5 = (7 \bmod 7) = 0$. The multiplicative inverse of 2 (referred to as “ 2^{-1} ”) is 4 because $2 \times 4 = (8 \bmod 7) = 1$.

Galois fields are named in honor of Évariste Galois (1811–1832). A brilliant French mathematician, he made seminal contributions to abstract algebra in his teens. He was expelled from college for criticizing the director, then imprisoned for six months for political protest. Soon after, at 20 years old, he died in a duel over a woman.



(This image is Public Domain.)

+	0	1	x	0	1
0	0	1	0	0	0
1	1	0	1	0	1

Fig. 18.2 Addition and multiplication over GF(2).

An important field for cryptography is $GF(2)$, which is the smallest finite field that can exist. It is useful for computing because its set is the binary values $\{0, 1\}$. Interestingly, $GF(2)$ addition is equivalent to an XOR gate, and $GF(2)$ multiplication is equivalent to an AND gate. The truth tables for addition and multiplication are shown in Fig. 18.2. Note that the additive inverse of 1 is 1 because $(1 + 1) \bmod 2 = 0$. The additive inverse of 0 is 0 because $(0 + 0) \bmod 2 = 0$. The multiplicative inverse of 1 is also 1 because $(1 \times 1) \bmod 2 = 1$.

18.1.1.2 GF(2^n)

Another important Galois field for cryptography is $GF(2^n)$, whose elements are polynomials of degree up to $n - 1$ with coefficients $\{0, 1\}$ in $GF(2)$. $GF(2^n)$ is also called a *binary extension field* because the coefficients are binary numbers. Multiplication is performed modulo some irreducible polynomial $m(x)$ of degree n . Wow, that's a mouthful, and enough to send mild-mannered hardware engineers running for cover, but it turns out to be easier than it sounds and especially easy to implement in hardware. It is best explained by example.

Consider $GF(2^3)$ with the irreducible polynomial $m(x) = x^3 + x + 1$. This field has 8 elements listed in Table 18.2. The field's elements are the polynomials of degree up to $n - 1 = 2$ with coefficients of $\{0, 1\}$ from $GF(2)$. $m(x)$ is of degree n . x is a dummy variable; its value is not relevant, but rather powers of x serve to separate the coefficients into n columns, as shown in the Full Polynomial column. A digital designer views the elements as n -bit binary numbers defined by their coefficients.

First, we consider addition. Because the coefficients are from $GF(2)$, addition is performed modulo 2 on each column. For example, if $a = x^2 + 1$ and $b = x + 1$, then addition is performed as:

$$\begin{aligned} a + b &= (1x^2 + 0x^1 + 1x^0) + (0x^2 + 1x^1 + 1x^0) \\ &= [(1+0) \bmod 2]x^2 + [(0+1) \bmod 2]x^1 + [(1+1) \bmod 2]x^0 \\ &= (1x^2 + 1x^1 + 0x^0) \\ &= x^2 + x \end{aligned}$$

Substitute a and b

Distributive property

Modular addition

Simplify

Table 18.2 $\text{GF}(2^3)$ elements and inverses for $m(x) = x^3 + x + 1$.

Element	Full Polynomial	Binary Representation	Additive Inverse	Multiplicative Inverse
0	$0x^2 + 0x^1 + 0x^0$	000	000	N/A
1	$0x^2 + 0x^1 + 1x^0$	001	001	001
x	$0x^2 + 1x^1 + 0x^0$	010	010	101
$x + 1$	$0x^2 + 1x^1 + 1x^0$	011	011	110
x^2	$1x^2 + 0x^1 + 0x^0$	100	100	111
$x^2 + 1$	$1x^2 + 0x^1 + 1x^0$	101	101	010
$x^2 + x$	$1x^2 + 1x^1 + 0x^0$	110	110	011
$x^2 + x + 1$	$1x^2 + 1x^1 + 1x^0$	111	111	100

There is a one-to-one correspondence between polynomials in $\text{GF}(2^n)$ and n -bit binary numbers: the polynomial $a = a_2x^2 + a_1x^1 + a_0x^0$ corresponds to the binary number $a_2a_1a_0$. Thus, in binary representation, $a = 101$, $b = 011$, $a + b = 101 + 011 = 101 \oplus 011 = 110$. In other words, addition in $\text{GF}(2^n)$ is simply bitwise XOR on the two n -bit binary representations, which is even easier than integer addition. The esoteric mathematics of Galois binary extension fields is cheap and easy to implement in hardware.

Addition in $\text{GF}(2^n)$ is closed because the XOR of any pair of n -bit numbers is another n -bit number. Moreover, for any element a , $-a = a$ because $a + -a = a \oplus a = 0$. In other words, each element is its own additive inverse.

Multiplication is more interesting because multiplying a polynomial by x increases the degree of the polynomial. To provide closure, the multiplication is done modulo $m(x)$. You might be rusty at computing the modulus of a polynomial, especially in a finite field. Write the irreducible polynomial as $m(x) = x^n + f(x)$, where $f(x)$ is of degree less than n . For any polynomial $p(x)$ of degree less than n , $p(x) \bmod m(x) = p(x)$, because when $p(x)$ is divided by $m(x)$, the quotient is 0 and the remainder is $p(x)$. The more interesting case is x^n . $x^n = m(x) - f(x)$, so when x^n is divided by $m(x)$, the quotient is 1, and the remainder is $-f(x)$. In $\text{GF}(2^n)$, $-f(x) = f(x)$ because it is its own additive inverse. Hence, $x^n \bmod m(x) = f(x)$. This is called *reduction* because it reduces the polynomial to degree less than n .

Now we consider multiplication by x . Again, consider a polynomial $a = x^2 + 1$ in $\text{GF}(2^3)$ with the irreducible polynomial $m(x) = x^3 + x + 1$. $f(x) = m(x) - x^n = x + 1$. Compute the product ax as:

$ax = (x^2+1)x \bmod (x^3 + x+1)$	Definition of multiplication
$= (x^3 + x) \bmod (x^3 + x+1)$	Polynomial multiplication
$= x^3 \bmod (x^3 + x+1) + x \bmod (x^3 + x+1)$	Distributive property
$= (x+1) + x \bmod (x^3 + x+1)$	$x^n \bmod m(x) = f(x); x^n = x^3, f(x) = x + 1$
$= (x+1) + (x)$	$p(x) \bmod m(x) = p(x); p(x) = x$
$= (x+x) + (1)$	Associativity
$= 1$	$x + x = 0$ in a binary extension field

Multiplication by higher powers of x involves repeated multiplication by x . ax^2 is computed from ax above as:

$$\begin{aligned}
 ax^2 &= (ax)x \\
 &= (1)x \\
 &= x
 \end{aligned}$$

Multiplication by an arbitrary polynomial can be expressed as the sum of multiplications by powers of x . For example, with $a = x^2 + 1$ and $b = x + 1$, the product ab is computed as:

$ab = a(x+1)$	Substitute b
$= ax + a$	Distributive property
$= (1) + (x^2+1)$	Substitute precomputed ax
$= x^2 + (1 + 1)$	Commutativity
$= x^2 + (0)$	$1 + 1 = 1 \oplus 1 = 0$
$= x^2$	Addition

In binary representation, multiplying a by x gives $ax = a \times 10 = (a \ll 1) \bmod m$. Thus, multiplication by x is a left shift by 1. If the result has a 0 shifted into column n , it is less than m , and the modulus does nothing. Otherwise, as with polynomials, the modulus reduces the result by dropping the 1 from column n and XORing the other bits with f , the binary representation of $f(x)$. Formally, for an arbitrary element a of $\text{GF}(2^n)$ written as an n -bit binary number $a_{n-1:0}$

$$ax = \begin{cases} (a_{n-2:0} \ll 1) & \text{when } a_{n-1} = 0 \\ (a_{n-2:0} \ll 1) \oplus f & \text{when } a_{n-1} = 1 \end{cases} \quad (18.1)$$

For example, in $\text{GF}(2^3)$, we can write a as a 3-bit binary number $a = a_2a_1a_0$. Thus, $(a \ll 1) \bmod m = (a_2a_1a_00) \bmod m$. When

$a_2 = 0$, this is just $(0a_1a_00) \bmod m = a_1a_00$. When $a_2 = 1$, this is $(1a_1a_00) \bmod m = (a_1a_00 \bmod m) + (1000 \bmod m) = a_1a_00 + f = a_1a_00 \oplus f$.

Let us repeat the multiplication examples above using binary representation. In binary, $a_{2:0} = 101$, $m = 1011$, so $f = 011$.

$$ax = (101 \ll 1) = \mathbf{1}010, \text{ which reduces to } 010 \oplus 011 = 001$$

$$ax^2 = (001 \ll 1) = \mathbf{0}010, \text{ which needs no reduction} = 010$$

In binary, multiplication by higher powers of x involves repeated shifts and XOR. Multiplying arbitrary polynomials in binary representation is the addition of selected multiplications by powers of x . For example, $a(x^2 + x) = ax^2 + ax = 001 + 010 = 001 \oplus 010 = 011$.

Example 18.1 AES MULTIPLICATION

AES treats bytes as elements of $\text{GF}(2^8)$ with $m(x) = x^8 + x^4 + x^3 + x + 1$. Thus, $f(x) = x^4 + x^3 + x + 1$; $f = 00011011$. Let $a = 11000101 = x^7 + x^6 + x^2 + 1$ and $b = 10100001 = x^7 + x^5 + 1$. Compute the product ab .

Solution: First, find the products ax^k for $k \in [0, 7]$, as shown in Table 18.3. For each k , use $ax^k = ax^{k-1} \times x$, left-shifting the previous result and XORing to reduce if necessary. The products are represented in binary for easy and compact computation.

Table 18.3 ax^k for $k \in [0, 7]$

Product	Expression	Binary Computation	Result
ax^0	A	11000101	11000101
ax^1	$(ax^0 \times x) \bmod m(x)$	$1000101 \ll 1 \oplus 00011011$	10010001
ax^2	$(ax^1 \times x) \bmod m(x)$	$0010001 \ll 1 \oplus 00011011$	00111001
ax^3	$(ax^2 \times x) \bmod m(x)$	$0111001 \ll 1$	01110010
ax^4	$(ax^3 \times x) \bmod m(x)$	$1110010 \ll 1$	11100100
ax^5	$(ax^4 \times x) \bmod m(x)$	$1100100 \ll 1 \oplus 00011011$	11010011
ax^6	$(ax^5 \times x) \bmod m(x)$	$1010011 \ll 1 \oplus 00011011$	10111101
ax^7	$(ax^6 \times x) \bmod m(x)$	$0111101 \ll 1 \oplus 00011011$	01100001

Next, sum the terms based on b :

$$\begin{aligned}
 ab &= a(x^7 + x^5 + 1) \\
 &= ax^7 + ax^5 + ax^0 \\
 &= 01100001 \oplus 11010011 \oplus 11000101 \\
 &= 01110111 \\
 &= x^6 + x^5 + x^4 + x^2 + x^1 + 1
 \end{aligned}$$

This code can be found in `examples/crypto/gfmul` if you wish to experiment.

Code Example 18.1 shows a C program to perform $GF(2^8)$ multiplication with a sequence of shifts and XORs [StDenis07]. The `a` variable is recycled on each step to hold ax^k . This code performs reduction as: $a \wedge m(x)$, which discards the most significant 1 of a and XORs with $f(x)$ in a single operation. In accordance with Example 18.1, $a \times b = 0xC5 \times 0xA1 = 0x77$.

Code Example 18.1 EXAMPLE C CODE FOR $GF(2^8)$ MULTIPLICATION FOR AES

```
// Return ab mod m(x): GF(2^8) multiplication
int gfmul(int a, int b) {
    int result = 0;
    while (b) {
        if (b & 1) result = result ^ a;    // Cycle through bits in b starting with lsb
        a = a << 1;                        // If bit of b is set, add a to result
        if (a & 0b100000000)               // Multiply a by x
            a = a ^ 0b100011011;           // If msb of a is 1
        b = b >> 1;                        // Reduce: a = a ^ m(x); m(x) = 100011011
    }
    return result;
}

int main() {
    int a = 0xC5;
    int b = 0xA1;
    printf("The GF(2^8) result is %x\n", gfmul(a,b));
}
```

Finding the multiplicative inverse of an element is complicated. It could be done by guess and check but is more efficiently performed by the *extended Euclid algorithm*. According to a number theory property called Bézout's identity, the inverse is unique because the $m(x)$ is irreducible. For a small finite field such as $GF(2^8)$, it is easiest to precompute and tabulate the inverses, as shown in Fig. 18.3. 0 has no multiplicative inverse, so the entry is 0.

For example, according to Fig. 18.3, the multiplicative inverse of $0xC1$ is $0x28$ because $11000001_2 \times 00101000_2 = 00000001_2$ in $GF(2^8)$.

18.1.2 AES: Advanced Encryption Standard

The most widely used encryption method is the Advanced Encryption Standard (AES), adopted in 2001 by the US National Institute of Standards and Technology (NIST) [NIST-FIPS-197-upd1]. AES is a symmetric key encryption algorithm because it uses a single key for both encryption and decryption. It encrypts 128-bit blocks using a 128-, 192-, or 256-bit

Federal Information Processing Standards (FIPS) publications are issued by NIST after they are approved by the U.S. Secretary of Commerce.

		Y															
A ⁻¹		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
		00	01	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
X	1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
	2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2
	3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
	4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09
	5	ED	5C	05	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
	6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
	7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	06	A1	FA	81	82
	8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	02	B9	A4
	9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
	A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
	B	0C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
	C	0B	28	2F	A3	DA	D4	E4	0F	A9	27	53	04	1B	FC	AC	E6
	D	7A	07	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
	E	B1	0D	D6	EB	C6	0E	CF	AD	08	4E	D7	E3	5D	00	1E	B3
	F	5B	23	38	34	68	46	03	8C	DD	9C	7D	A0	CD	1A	41	1C

Fig. 18.3 Multiplicative inverse table for $GF(2^8)$ for $A = \{X, Y\}$ (in hexadecimal format).

key. AES is used worldwide and is the basis of most encryption on the internet because it is secure and fast.

AES operates on 128-bit blocks of plaintext to produce 128-bit blocks of ciphertext. The 128 bits at each intermediate step are called *state* and are organized as a 4×4 matrix of bytes $s_{i,j}$ representing elements in $GF(2^8)$, shown in Fig. 18.4.

The AES encryption algorithm is summarized in Fig. 18.5. It comprises $N_r = 10, 12$, or 14 rounds of operation for 128-, 192-, or 256-bit keys, respectively. Each round performs a nonlinear substitution (for confusion), applies linear transformations to the rows and columns (for diffusion), and then adds that round's key, referred to as the *round key*. The first $N_r - 1$ rounds use the SubBytes, ShiftRows, MixColumns, and AddRoundKey units. The last round omits the MixColumns step because it does not contribute to security. The key is expanded to form $(N_r + 1)$ 128-bit round keys $K_{N_r:0}$ so each round uses a different version of the key. Each of these units is described further in the following subsections.

18.1.2.1 AddRoundKey

The *AddRoundKey* unit adds the current round key K_j to the current state so that the ciphertext depends on both the plaintext and the key. Recall

$$\text{state} = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

Fig. 18.4 Array structure of AES data or state matrix.

AES is based on the Rijndael cipher developed by Belgian cryptographers Joan Daemen and Vincent Rijmen. It replaced the previous Data Encryption Standard (DES) after the Electronic Frontier Foundation demonstrated a custom chip in 1988 that could crack the 56-bit key in about a day using brute force search. AES is too hard to crack in a reasonable amount of time by guessing keys or by any other known attack. With 256-bit keys, it is expected to be resistant to hypothetical quantum computers as well.

The AES standard is an exceptionally well-written specification, only 38 pages long, including 10 pages of worked examples showing every intermediate value expected at every step of an encryption. Anyone implementing AES will want to read the standard.

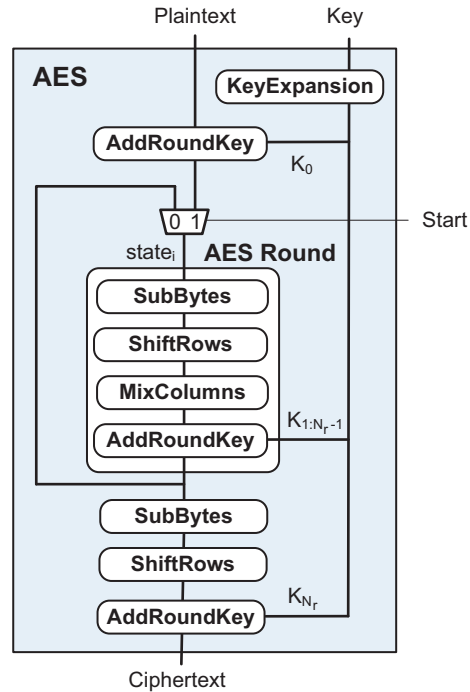


Fig. 18.5 Overall AES encryption architecture.

that addition in $GF(2^8)$ is XOR, so the `AddRoundKey` block is simply 128 2-input XOR gates to XOR the 128-bit state and 128-bit round key.

18.1.2.2 SubBytes

The *SubBytes* unit replaces each of the 16 bytes in the state matrix with another byte using a 2^8 -input lookup table called the *S-box* (*substitution-box*), as shown in Fig. 18.6. The S-box performs substitution to introduce non-linearity into the cipher. The contents of the S-box are listed in Fig. 18.7. For example, if $s_{0,0} = 0x42$, the S-box will replace it with $s'_{0,0} = 0x2c$.

S-boxes are common in symmetric key ciphers because they obscure the relationship between the plaintext and the ciphertext. The AES S-box is conceptually a $256\text{-entry} \times 8\text{-bit}$ lookup table. It follows a mathematical structure, so it can also be synthesized onto logic gates, potentially in a more compact way [Swann23, github.com/mjosaarinen/lwaes_isa/blob/master/hdl/sboxes.v].

18.1.2.3 ShiftRows

The *ShiftRows* unit diffuses information across the block by rotating rows 1, 2, and 3 of the state matrix left by 1, 2, and 3 elements respectively, as

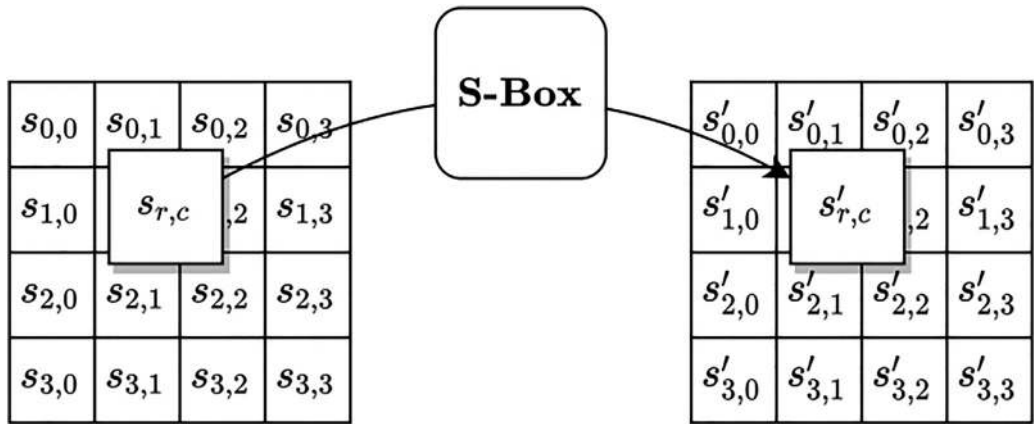


Fig. 18.6 SubBytes operation. [NIST-FIPS-197-upd1]

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Fig. 18.7 S-box contents. [NIST-FIPS-197-upd1]

shown in Fig. 18.8. Constant rotations simply rename the bits and require no hardware.

18.1.2.4 MixColumns

The *MixColumns* unit also diffuses information by multiplying the state matrix by a simple linear transformation matrix containing values 0x01, 0x02, and 0x03, as shown in Fig. 18.9. This transformation is called *maximum distance separable* (MDS). If two plaintexts differ only in a

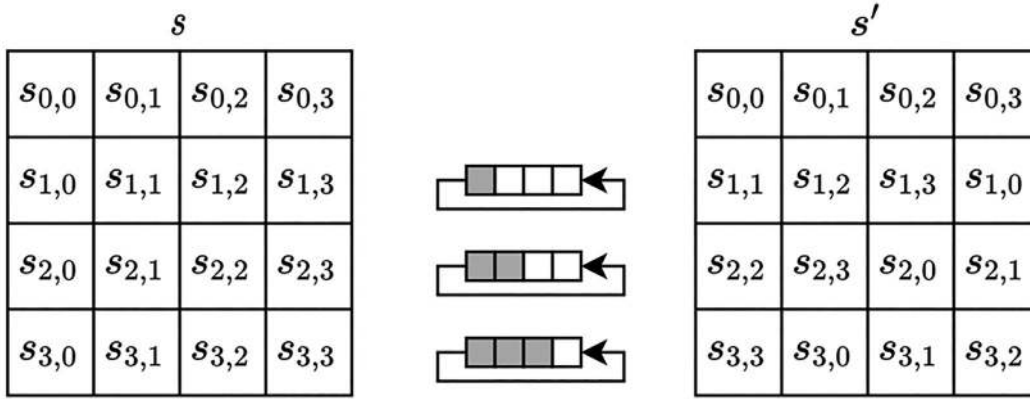


Fig. 18.8 ShiftRows operation. [NIST-FIPS-197-upd1]

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

Fig. 18.9 MixColumns operation.

single byte, the MDS transformation spreads that change to other bytes of the state in as few steps as possible [St Denis07].

The elements 01, 02, and 03 (binary 01, 10, 11) correspond to polynomials 1, x , and $x+1$ in $\text{GF}(2^8)$. Multiplication by 1 has no effect. Recall that multiplication by x is a left shift by 1; if the result overflows, it is reduced by XOR with $f = 00011011$. Multiplication by $x+1$ is a multiplication by x followed by adding the original number with an XOR.

For example, the upper left entry of the result is computed as:

$$\begin{aligned} s'_{0,0} &= (02 \times s_{0,0}) \oplus (03 \times s_{1,0}) \oplus (01 \times s_{2,0}) \oplus (01 \times s_{3,0}) \\ &= (10 \times s_{0,0}) \oplus (11 \times s_{1,0}) \oplus (01 \times s_{2,0}) \oplus (01 \times s_{3,0}) \\ &= (x s_{0,0}) \oplus ((x+1) s_{1,0}) \oplus s_{2,0} \oplus s_{3,0} \\ &= (x s_{0,0}) \oplus (x s_{1,0}) \oplus s_{1,0} \oplus s_{2,0} \oplus s_{3,0} \end{aligned}$$

18.1.2.5 KeyExpansion

The *KeyExpansion* unit performs the *key schedule*, which is the process of deriving the *round keys* K from the *key*. The key schedule distributes the entropy of the key across rounds, making it resistant to certain types of attacks with related keys. Table 18.4 highlights the variables used in the KeyExpansion unit.

Table 18.4 KeyExpansion unit variables.

Variable	Size			Description
key	128 bits	192 bits	256 bits	Input key used to generate round keys
N_k	4	6	8	Number of 32-bit words in the key
N_r	10	12	14	Number of rounds
K_{N_r+1}	11×128	13×128	15×128	Round keys: ciphers used to encrypt the message during each round ($N_r + 1$) round keys, 128 bits each
$w_{4N_r+3:0}$	44×32	52×32	60×32	Round keys organized as 32-bit words, generated by the key schedule $4(N_r + 1)$ 32-bit words

```

for  $i = 0$  to  $4(N_r + 1) - 1$            // Generate all  $N_r + 1$  round keys, which are 4 words each
  if ( $i < N_k$ )  $w_i = key_i$            // Initialize first  $N_k$  words of the round keys with the key
  else                                   // Generate the rest of the words of the round keys
    if ( $i$  is a multiple of  $N_k$ )
       $t = \text{SubWord}(\text{RotWord}(w_{i-1})) \oplus \text{Rcon}_{i/N_k}$ 
    else if ( $N_k = 8$  and ( $i + 4$  is a multiple of 8))
       $t = \text{SubWord}(w_{i-1})$ 
    else  $t = w_{i-1}$ 
     $w_i = w_{i-N_k} \oplus t$ 
for  $j = 0$  to  $(N_r + 1) K_j = w_{4j+3:4j}$  // Assign round keys

```

Fig. 18.10 KeyExpansion key schedule.

N_k is the number of 32-bit words in the key. The key is indexed by word, $key_{N_k-1:0}$. Using the key, the key schedule generates w , which contains all words of all round keys. So, w contains $4(N_r + 1)$ words, indexed by word. Each round key K_j , indexed by round key number, is then formed from four consecutive words in w : $K_j = w_{4j+3:4j}$ for $j = 0$ to N_r . So, $K_0 = w_{3:0}$, $K_1 = w_{7:4}$, and so on. The key schedule that produces the round key words, $w_{4N_r+3:0}$, is given in Fig. 18.10.

The SubWord operation applies four S-box substitutions to the four bytes in a word. RotWord rotates a word left by one byte. The round constants $\text{Rcon}_j = (x^{j-1} \bmod m(x))$ are given in hexadecimal in Table 18.5.

Example 18.2 ROUND KEY CALCULATIONS

Compute the first three round keys for the 128-bit key = 0x2b7e151628aed2a6abf7158809cf4f3c. Note that AES uses big-endian notation with 0x2b being the lowest-numbered byte of the key.

Solution: Table 18.6 shows the key expansion [NIST-FIPS-197-upd1, Appendix A.1]. $N_k = 4$ and $N_r = 10$ for 128-bit keys. $K_0 = key = 0x2b7e...3c$. $K_1 = w_{7:4} = a0fafe1788542cb123a339392a6c7605$. $K_2 = w_{11:8} = f2c295f27a96b9435935807a7359f67f$.

Table 18.5 Round constants for key schedule.

j	Rcon_j
1	0x01
2	0x02
3	0x04
4	0x08
5	0x10
6	0x20
7	0x40
8	0x80
9	0x1B
10	0x36

Table 18.6 Key expansion example.

i	w_{i-1}	After RotWord	After SubWord	$Rcon_{i/4}$	t	w_{i-4}	$w_i = \text{key}_i \quad i < 4$ $= t \oplus w_{i-4} \quad i \geq 4$
0							2b7e1516
1							28aed2a6
2							abf71588
3							09cf4f3c
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafa17
5	a0fafa17				a0fafa17	28aed2z6	88542cb1
6	88542cb1				88542cb1	abf71588	23a33939
7	23a33939				23a33939	09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafa17	f2c295f2
9	f2c295f2				f2c295f2	88542cb1	7a96b943
10	7a96b943				7a96b943	23a33939	5935807a
11	5935807a				5935807a	2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d				3d80477d	7a96b943	4716fe3e
14	4716fe3e				4716fe3e	5935807a	1e237e44
15	1e237e44				1e237e44	7359f67f	6d7a883b
...							
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8				d014f9a8	19fadc21	c9ee2589
42	c9ee2589				c9ee2589	28d12941	e13f0cc8
43	e13f0cc8				e13f0cc8	575c006e	b6630ca6

Example 18.3 PLAINTEXT ENCRYPTION USING ROUND KEYS

Encrypt the plaintext 0x3243f6a8_885a308d_313198a2_e0370734 with the 128-bit round keys from Example 18.2.

Solution: Fig. 18.11 (on page 734) shows the encryption steps. The input is first XORed with the round key. In each round, AddRoundKey, SubBytes, ShiftRows, and MixColumns are applied. On the last round, MixColumns is omitted. Specifically, consider the upper left entry $s_{0,0}$ in Round 1. The entry at the start of the round comes from the initial AddRoundKey: $32 \oplus 2b = 19$. Referring to Fig. 18.7, $\text{SubBytes}(19) = d4$. ShiftRows does not modify the top row, so the element remains 19. MixColumns computes $[02\ 03\ 01\ 01] [d4\ bf\ 5d\ 30]^T = (d4)x \oplus (bf)(x+1) \oplus (5d)1 \oplus (30)1 = ((d4 \ll 1) \oplus 11b) \oplus ((bf \ll 1) \oplus 11b) \oplus bf \oplus 5d \oplus 30 = b3 \oplus 65 \oplus bf \oplus 5d \oplus 30 = 04$. The upper left entry of the Round Key is a0 from the row 4 w_i entry of Table 18.6.

18.1.2.6 Decryption

Because the AES algorithm is a symmetric cryptographic algorithm, decryption (also called the *inverse cipher*) consists of the same steps as encryption but with operations replaced by their inverses, as shown in Fig. 18.12. The key schedule also changes. AddRoundKey is its own inverse because $a = -a$ in $\text{GF}(2^n)$. Therefore, the hardware XORs with the

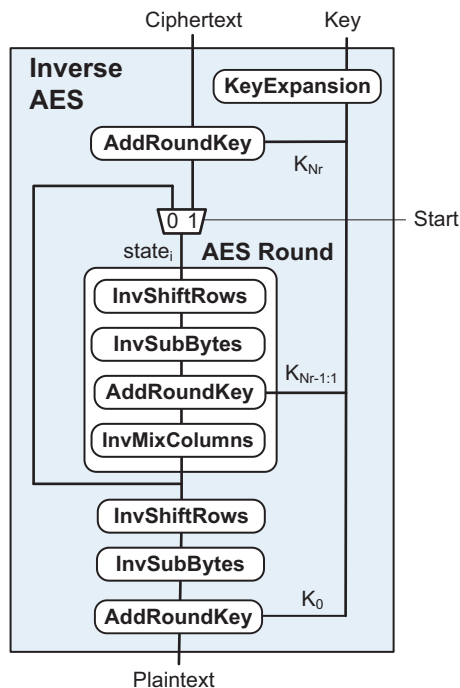


Fig. 18.12 Overall AES decryption architecture.

Round Number	Start of Round	After SubBytes	After ShiftRows	After MixColumns	Round Key Value
input	32 88 31 e0				2b 28 ab 09
	43 5a 31 37				7e ae f7 cf
	f6 30 98 07				15 d2 15 4f
	a8 8d a2 34				16 a6 88 3c
1	19 a0 9a e9	d4 e0 b8 1e	d4 e0 b8 1e	04 e0 48 28	a0 88 23 2a
	3d f4 c6 f8	27 bf b4 41	bf b4 41 27	66 cb f8 06	fa 54 a3 6c
	e3 e2 8d 48	11 98 5d 52	5d 52 11 98	81 19 d3 26	fe 2c 39 76
	be 2b 2a 08	ae f1 e5 30	30 ae f1 e5	e5 9a 7a 4c	17 b1 39 05
2	a4 68 6b 02	49 45 7f 77	49 45 7f 77	58 1b db 1b	f2 7a 59 73
	9c 9f 5b 6a	de db 39 02	db 39 02 de	4d 4b e7 6b	c2 96 35 59
	7f 35 ea 50	d2 96 87 53	87 53 d2 96	ca 5a ca b0	95 b9 80 f6
	f2 2b 43 49	89 f1 1a 3b	3b 89 f1 1a	f1 ac a8 e5	f2 43 7a 7f
3	aa 61 82 68	ac ef 13 45	ac ef 13 45	75 20 53 bb	3d 47 1e 6d
	8f dd d2 32	73 c1 b5 23	c1 b5 23 73	ec 0b c0 25	80 16 23 7a
	5f e3 4a 46	cf 11 d6 5a	d6 5a cf 11	09 63 cf d0	47 fe 7e 88
	03 ef d2 9a	7b df b5 b8	b8 7b df b5	93 33 7c dc	7d 3e 44 3b
10	eb 59 8b 1b	e9 cb 3d af	e9 cb 3d af		d0 c9 e1 b6
	40 2e a1 c3	09 31 32 2e	31 32 2e 09		14 ee 3f 63
	f2 38 13 42	89 07 7d 2c	7d 2c 89 07		f9 25 0c 0c
	1e 84 e7 d2	72 5f 94 b5	b5 72 5f 94		a8 89 c8 a6
output	39 02 dc 19				
	25 dc 11 6a				
	84 09 85 0b				
	1d fb 97 32				

Fig. 18.11 Encryption example. [NIST-FIPS-197-upd1, Appendix B]

same round keys for both encryption and decryption. InvSubBytes uses an inverse S-box that flips inputs and outputs. InvShiftRows shifts right instead of left. InvMixColumns uses the transformation matrix from Fig. 18.13, which involves a more complicated sequence of shifts and XORs than the MixColumns matrix.

0e	0b	0d	09
09	0e	0b	0d
0d	09	0e	0b
0b	0d	09	0e

Fig. 18.13 InvMixColumns matrix.

18.1.3 Hash Functions

Cryptographic hash functions are important elements of cryptography. They transform a variable-length message into a short numerical fingerprint called a *message digest*. Hashes are used to verify data integrity and as a building block for digital signatures. Any alteration to a message will corrupt the message digest. Passwords are usually stored in hashed form instead of plaintext, so stealing the password file will not reveal the password itself. A good hash function has several properties:

- ▶ **Avalanche effect:** Any change in the message will, with very high probability, change many bits of the digest.
- ▶ **Pre-image resistant:** Given a message digest, it is computationally infeasible to find the original message.
- ▶ **Collision resistant:** Given a message digest, it is computationally infeasible to find another message that produces the same digest.
- ▶ **Fast and easy to compute**

SHA-2 is the hash function used in most internet protocols, such as TLS, SSL, and PGP, well as to verify transactions in Bitcoin and other cryptocurrencies. It was designed by the US National Security Agency (NSA) and was first published in 2001. It is now a standard maintained by the National Institute of Standards and Technology [NIST-FIPS-180-4]. SHA-2 generates 224-, 256-, 384-, or 512-bit message digests, replacing the SHA-1, MD4, and MD5 algorithms that produced shorter digests that are no longer considered secure. This section describes the SHA-256 and SHA-512 flavors of SHA-2 that produce 256- or 512-bit digests. Other flavors of SHA-2 are similar but truncate the digest to fewer bits after it is computed, trading compactness for reduced security.

Table 18.7 summarizes the structure of SHA-256 and SHA-512. The hash operates on a message M comprising N m -bit blocks; it is padded if necessary to be an integral number of blocks. Each block is formed from 16 w -bit words. Word j of block i is denoted $M_j^{(i)}$. The message digest (also called the hash) H is formed from 8 w -bit words. Each block goes through r rounds of hashing, which involve applying some left shifts, right rotates, and XOR operations. The hashing steps involve sigma (Σ/σ) functions that are expressed in terms of right rotations (ror) and right shifts

This chapter refers to cryptographic hash functions as hash functions for brevity. Other kinds of noncryptographic hash functions exist that are optimized for different goals.

Table 18.7 SHA-2 structure and sigma operations.

SHA-2 Algorithm	Variable	SHA-256	SHA-512
Message digest size (bits)	d	256	512
Block size (bits)	m	512	1,024
Word size (bits)	w	32	64
Rounds	r	64	80
$\Sigma_0^{(d)}(x)$		$(x \text{ ror } 2) \oplus (x \text{ ror } 13) \oplus (x \text{ ror } 22)$	$(x \text{ ror } 28) \oplus (x \text{ ror } 34) \oplus (x \text{ ror } 39)$
$\Sigma_1^{(d)}(x)$		$(x \text{ ror } 6) \oplus (x \text{ ror } 11) \oplus (x \text{ ror } 25)$	$(x \text{ ror } 14) \oplus (x \text{ ror } 18) \oplus (x \text{ ror } 41)$
$\sigma_0^{(d)}(x)$		$(x \text{ ror } 7) \oplus (x \text{ ror } 18) \oplus (x \gg 3)$	$(x \text{ ror } 1) \oplus (x \text{ ror } 8) \oplus (x \gg 7)$
$\sigma_1^{(d)}(x)$		$(x \text{ ror } 17) \oplus (x \text{ ror } 19) \oplus (x \gg 10)$	$(x \text{ ror } 19) \oplus (x \text{ ror } 61) \oplus (x \gg 6)$

The random-looking initialization and round constants in Table 18.7 correspond to the most significant fractional parts of the cube and square roots of the first 8 and r prime numbers, respectively. They are called *nothing-up-my-sleeve numbers* and give confidence that they were not specially chosen by the NSA to provide a backdoor.

($\gg$) of the words. The hash is initialized with 8 w -bit constants $H_i^{(0)}$ and uses r w -bit round constants K_t tabulated in the SHA-2 specification.

Fig. 18.14 describes the SHA-2 algorithm, which is the same for SHA-256 and SHA-512 except for the parameter values, constants, and definition of sigma functions, as given in Table 18.7. Addition is defined to be modulo 2^w , so overflow is discarded. The algorithm also uses *Choose* (Ch) and *Majority* (Maj) functions which, respectively, choose y or z based on x and output 1 if a majority (at least 2 of the 3) inputs is TRUE. These functions are defined as:

$$\begin{aligned}\text{Ch}(x, y, z) &= xy \oplus \bar{x}z = xy \mid \bar{x}z = x ? y : z \text{ (bitwise)} \\ \text{Maj}(x, y, z) &= xy \oplus xz \oplus yz = xy \mid xz \mid yz = 1\end{aligned}$$

SHA-2 initializes the 8-word hash $H^{(0)}$ to known constants. It then processes each of the N blocks. It begins by expanding the block into an R -word *message schedule* $W_{S-1:0}$. The first 16 words of the message schedule are the 16 words of the block, and the remaining words of the schedule depend on rotates and shifts of some of the earlier words. Next, the algorithm initializes 8 *working variables* a_j based on the hash of the previous block. It then introduces diffusion and confusion into the hash by applying r rounds. In each round, the working variables are shifted by one word, and a hashing function is applied to certain working variables. At round t , the nonlinear hashing function depends on the round constant K_t and message schedule word W_t . The arbitrary-seeming choices of the hashing function draw from the *Merkle-Damgård method* of building

```

for j = 0 to 7
     $H_j^{(0)} = \text{initial value}(d, j)$  // Initialize hash
for i = 1 to N // Process each block in order
    for j = 0 to 7 // Initialize working variables
         $a_j = H_j^{(i-1)}$ 
    for t = 0 to r-1 // Hashing rounds
        if (t < 16)  $W_t = M_t^{(i)}$  // Message schedule
        else  $W_t = \sigma_1^{(d)}(W_{t-2}) + W_{t-7} + \sigma_0^{(d)}(W_{t-15}) + W_{t-16}$ 
         $T = a_7 + \Sigma_1^{(d)}(a_4) + \text{Ch}(a_0, a_5, a_6) + K_t + W_t$ 
        for j = 7 downto 1 // Shift words
             $a_j = a_j - 1$ 
             $a_0 = \Sigma_0^{(d)}(a_0) + \text{Maj}(a_0, a_1, a_2) + T$  // Hash word 0
             $a_4 = a_4 + T$  // Hash word 4
        for j = 0 to 7
             $H_j^{(i)} = H_j^{(i-1)} + a_j$  // Update hash after block i
     $H = \{H_0^{(N)} H_1^{(N)} H_2^{(N)} H_3^{(N)} H_4^{(N)} H_5^{(N)} H_6^{(N)} H_7^{(N)}\}$  // Final D-bit hash

```

Fig. 18.14 SHA-2 algorithm.

For the sake of brevity and to emphasize the similarities between SHA-256 and SHA-512, the SHA-2 algorithm in Fig. 18.14 is described with notation that is simpler than but equivalent to that of [NIST-FIPS-180-4]. The working variables are called a_7 , a_6 , ..., a_0 rather than $\{a, \dots, b\}$. T_1 is called T and T_2 is folded into the a_4 equation. The rounds are also called iterations.

collision-resistant hashes. After all the rounds are complete, the 8 words of the hash for block i are computed as the sum of the working variables and the words of the hash for block $i-1$. After all blocks are processed, the message digest is the hash for the final block. This diffusion and compression block for a_j is sometimes called a compression function and is visualized in Fig. 18.15.

Most Linux distributions come with programs to compute the SHA-256 hash function to verify data integrity. For example, the command at the top of page 738 produces the hash for “Hello World!” using SHA-256. If somebody changed the exclamation point to a question mark, `sha256sum` would give a different hash, revealing that the message had been corrupted.

You can interactively watch SHA-256 run at sha256algorithm.com.

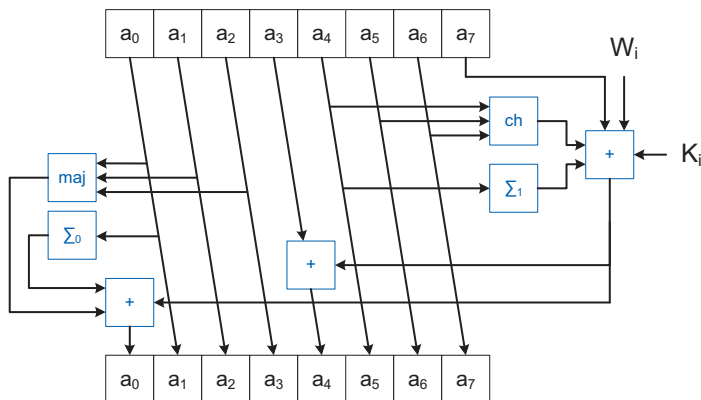


Fig. 18.15 SHA256 compression function.

```
$ echo -n "Hello World!" | sha256sum
7f83b1657ff1fc53b92dc18148a1d65d6c2d4b1fa3d677284
add200126d9069 -
```

Scalar cryptography instructions operate on the ordinary register file. They provide lightweight cryptography acceleration practical even for low-cost microcontrollers. Vector cryptography instructions are faster because they can process multiple elements concurrently from the wider vector register file. The RISC-V vector extensions are outside the scope of this book, but the crypto instructions are similar.

RISC-V also defines the Zksed and Zksh extensions for the Shāngmì SM4 block cipher and SM3 hash functions. These are primarily of interest to markets where the Chinese government requires these algorithms; they are not described in this chapter. The Chinese Cybersecurity Law established a legal obligation for internet operators to turn over data to the government [Laskai21].

18.2 RISC-V PRACTICES

The RISC-V base integer instruction set (RV{32/64}I) has a minimal set of instructions sufficient to write any program, but bit manipulation instructions make programs faster and smaller. RISC-V has ratified the four separate bit manipulation extensions listed in Table 18.8. A configuration may support any or all of these extensions. For example, a microcontroller might support Zba, Zbb, and Zbs but omit Zbc to save cost if it rarely performs cryptographic operations. The B extension is shorthand for Zba + Zbb + Zbs. RV64 adds W64-type operations for Zba and Zbb, which act on only the lower 32 bits of each operand. The instructions are listed in Table I.23 (on the last pages of the book).

RISC-V has also ratified various scalar cryptography extensions listed in Table 18.9. The Zbk* family of instructions performs bit manipulations for cryptography and partially overlaps with the Zb* extensions. The Zkn* family of instructions performs key steps in AES decryption and encryption and SHA-2 hashing. The Zkn extension is shorthand for all these instructions, collectively called NIST cryptography. If instructions have data-dependent delay, hackers can use *timing attacks* to infer information about a cryptographic key. Therefore, the Zkt extension specifies that a list of simple and common instructions execute in data-independent time to prevent such attacks; this list excludes division and remainder and is automatically satisfied by most microarchitectures, including Wally. The Zkr extension defines a seed CSR that provides entropy bits to initialize a cryptographic random number generator. The Zk extension is shorthand for Zkn + Zkt + Zkr, which provides all the features that cryptographic library code would need.

18.2.1 Zba: Address Generation

To access an array, a program calculates an element’s address by adding the array’s base address to the element index multiplied by the element size. The Zba extension makes accessing halfwords, words, and double words more efficient by incorporating multiplications by 2, 4, and 8

Table 18.8 Bit manipulation extensions.

Extension	Name	Summary
Zba	Address generation	Shifts and add to access 2/4/8-byte elements
Zbb	Basic bit manipulation	Min/max, rotations, counting, sign/zero extension, etc.
Zbc	Carry-less multiplication	For Galois-field operations such as cryptography
Zbs	Single-bit instructions	Set, clear, invert, or extract single bits
B	Bit manipulation	Zba + Zbb + Zbs

Table 18.9 Scalar cryptography extensions.

Extension	Name	Summary
Zbkb	Bit manipulation	Bit manipulation instructions for cryptography
Zbkc	Carry-less multiplication	Subset of Zbc
Zbkx	Crossbar permutations	Useful for implementing S-boxes
Zknd	Decryption/key scheduling	Functions of the AES block cipher for decryption
Zkne	Encryption/key scheduling	Functions of the AES block cipher for encryption
Zknh	Hash functions	For SHA-2 family of cryptographic hash
Zkn	NIST crypto instructions	Zbkb + Zbkc + Zbkx + Zknd + Zkne + Zknh
Zkt	Data-independent latency	Instructions execute in data-independent time
Zkr	Entropy source	For random number generation
Zk	Standard scalar crypto	Zkn + Zkt + Zkr

into an add instruction. [Code Example 18.2](#) shows how the `sh2add` (shift left by 2 and add) instruction saves an instruction in the common case of accessing an array element.

Code Example 18.2 ADDRESS GENERATION

C Program	Assembly Without Zba	Assembly With Zba
<pre>int i, b; # s0, s1 int a[10]; # Base address in s2 b = a[i];</pre>	<pre>slli t0, s0, 2 # i * 4 addi t0, s2, t0 # &a[i] lw s1, 0(t0) # b = a[i]</pre>	<pre>sh2add t0, s0, s2 # &a[i] # t0=s2+s0<<2 lw s1, 0(t0) # b = a[i]</pre>

The `.uw` instructions for RV64 are helpful for generating code for the 32-bit unsigned int data type. In [Code Example 18.2](#), if `i` is an unsigned int, then the RV64 assembly language code becomes `sh2add.uw t0, s0, s2`. [Code Example 18.3](#) illustrates how `add.uw` simplifies zero extension of a 32-bit unsigned int.

Code Example 18.3 ZERO EXTENSION OF UNSIGNED INT

C Program	Assembly Without Zba	Assembly With Zba
<pre>long a, y; // s0, s2 unsigned int b; // s1 y = a + b;</pre>	<pre>slli s1, s1, 32 srli s1, s1, 32 # Zero ext. add s2, s0, s1</pre>	<pre>add.uw s2, s1, s0</pre>

18.2.2 Zbb: Basic Bit Manipulation

Zbb provides a miscellaneous set of widely used bit manipulation operations.

18.2.2.1 Rotates

Rotate operations (rol, ror, rori) circularly shift an input left or right and fill the empty positions with the bits shifted off the end. For example:

rori 0000_1111_0000_0000_0000_0000_1000_0100, 4
= 0100_0000_1111_0000_0000_0000_0000_1000

C does not have a rotate operator, but rotations can be performed by merging a pair of logical shifts. Code Example 18.4 shows C code and its assembly equivalent without and with the rotate instruction.

rol is unnecessary because it is equivalent to ror with a complementary version of the immediate. For example, rol t0, s0, 1 can be performed as ror t0, s0, 31.

Code Example 18.4 LEFT ROTATE (RV32)

C Program	Assembly Without Zbb	Assembly With Zbb
<pre># Rotate left by n int rol(unsigned int a, int n) { return (a << n) (a >> (32-n)); }</pre>	<pre>sll t0, a0, a1 # a << n sub t1, zero, t1 # -n srl t1, a0, t1 # a >> 32-n or a0, t0, t1 # rol ret</pre>	<pre>rol a0, a0, a1 ret</pre>

The assembly language code in Code Example 18.4 optimizes the shift of 32−n to a shift of just −n because RV32 shift amounts are interpreted modulo 32.

Rotations show up in many algorithms from discrete mathematics, including hashing, cryptography, and cyclic redundancy checks (CRCs) used in error correction.

18.2.2.2 Inverted Operations

Inverted operations (andn, orn, xnor) perform the base operation (and, or, xor) with the second operand inverted. This is essentially free in hardware because subtraction already requires the ability to invert the second operand.

For example, andn masks off the asserted bits, clearing them to zero.

andn 1010_1111_1101_0000, 0000_0000_1110_0000 = 1010_1111_0001_0000

18.2.2.3 Max/Min

Maximum and minimum operations have signed and unsigned flavors. For example,

max 0xFE000000, 0x12345678 = 0x12345678 # Signed
maxu 0xFE000000, 0x12345678 = 0xFE000000 # Unsigned

Code Example 18.4 shows C code and its assembly equivalent without and with the max instruction.

Code Example 18.5 MAX

C Program	Assembly Without Zbb	Assembly With Zbb
<pre>int max(int m, int n) { return m > n ? m : n; }</pre>	<pre> blt a1, a0, skip mv a0, a1 skip: ret</pre>	<pre>max a0, a0, a1 ret</pre>

18.2.2.4 Extension

The extension operations (`zext.h`, `sext.h`, and `sext.b`) zero- or sign-extend half-words or bytes (16- or 8-bit values). For example, the following code sign-extends the least significant byte.

```
sext.b  0000_1111_0000_0000_0000_0000_1000_0100
        =1111_1111_1111_1111_1111_1111_1000_0100
```

Code Example 18.6 shows C code to add two signed 16-bit short integers. In RV32, the upper 16 bits of the 32-bit register are copies of the sign bit. Ordinary 32-bit addition can overflow into the 17th bit, producing the wrong answer. Hence, the addition is followed by a sign extension operation. 16-bit sign extension can be done with a pair of shifts or the `sext.h` instruction.

Other sign or zero extensions can be performed using pseudoinstructions:

```
# Zero-extend the LSB
zext.b rd, rs1 ->
andi  rd, rs1, 255
# Zero-extend the lower
#32 bits
zext.w rd, rs1 ->
add.uw rd, rs1, 0
# Sign-extend the lower
# 32 bits
sext.w rd, rs1 ->
addi.w rd, rs1, 0
```

Code Example 18.6 EXTENSION

C Program	Assembly Without Zbb	Assembly With Zbb
<pre>short sum(short m, short n) { return m + n; }</pre>	<pre>add a0, a0, a1 slli a0, a0, 16 srai a0, a0, 16 ret</pre>	<pre>add a0, a0, a1 sext.h a0, a0 ret</pre>

18.2.2.5 Counting, Combining, Reversing

`cpop` (population count) counts the number of ones in a register. For example,

```
cpop 0000_1111_0000_0000_0000_0000_1000_0100 = 6
```

`clz` (count leading zeros) and `ctz` (count trailing zeros) count the number of zeros at the most significant or least significant end of a register. For example,

cpop has many applications, such as counting 1s in a bitstream, computing Hamming distance, and implementing associative arrays with an efficient data structure called *hashed array mapped trie* (HAMT) [Bagwell00]. Computing Hamming distance is important for cryptanalysis, and the National Security Agency (NSA) reportedly asked for the cpop instruction in early supercomputers.

```
clz 0000_1111_0000_0000_0000_0000_1000_0100 = 4
ctz 0000_1111_0000_0000_0000_0000_1000_0100 = 2
```

orc.b (OR combine bytes) checks each byte in a register. If the byte is nonzero, it sets the corresponding result byte to all ones. Otherwise, the result byte remains all zeros. For example,

```
orc.b 00001111_00000000_00000000_10000100
      = 11111111_00000000_00000000_11111111
```

rev8 (reverse bytes) reverses the bytes in a register. For example,

```
rev8 00001111_00000000_00000000_10000100
      = 10000100_00000000_00000000_00001111
```

Cryptography uses the rev8 byte reversal instruction described in Section 18.2.2.5 to convert between big and little-endian formats for input and output data.

orc.b is helpful to find a NULL (0) byte for string manipulation functions. rev8 is helpful to reverse the endianness. See the RISC-V unprivileged specification for example code using these instructions to find the length of a string and to compare two strings (using functions `strlen` and `strcmp`).

C does not have any of these operators. GCC currently does not infer them from C code either, but it does produce them using built-in counting functions:

```
int __builtin_popcount(unsigned int x);
int __builtin_clz(unsigned int x);
int __builtin_ctz(unsigned int x);
```

```

      0 1 1 1
x    1 0 1 0
-----
      0 0 0 0
      0 1 1 1
      0 0 0 0
+   0 1 1 1
-----
    0 1 1 0 1 1 0
```

Fig. 18.16 Carry-less multiplication example.

18.2.3 Zbc: Carry-less Multiplication

Carry-less multiplication produces partial products like regular multiplication but sums each column using XOR, discarding carries out of each column. Fig. 18.16 shows an example of 4×4 -bit carry-less multiplication. In the blue column, the answer is $0 \oplus 1 \oplus 0 \oplus 1 = 0$, with no carry out to the next column.

Carry-less multiplication is a building block for multiplication in the Galois field $\text{GF}(2^n)$. It is used in a variety of algorithms, including CRC, Galois/Counter Mode symmetric key encryption, and fast universal hash functions.

The @ symbol indicates carry-less multiplication in Table I.23. `clmul` and `clmulh` produce the low and high half of the 2XLEN -bit carry-less product. `clmulr` (carry-less multiply reversed) is used in CRC calculations.

18.2.4 Zbs: Single-Bit Operations

Zbs provides instructions to set, clear, invert, and extract a bit within a register. `bset` sets the bit to 1. `bclr` clears the bit to 0. `binv` inverts the bit. `bext` produces a result that is either 0 or 1, matching the specified bit in the source register. These instructions are often used to modify bitfields within a word. In memory-mapped I/O, they are used to turn on and off control bits or to check the value of a status bit. For example:

```
bset 0000_1111_0000_0000_0000_0000_1000_0100, 5
    = 0000_1111_0000_0000_0000_0000_1010_0100
bclr 0000_1111_0000_0000_0000_0000_1000_0100, 2
    = 0000_1111_0000_0000_0000_0000_1000_0000
binv 0000_1111_0000_0000_0000_0000_1000_0100, 1
    = 0000_1111_0000_0000_0000_0000_1000_0110
bext 0000_1111_0000_0000_0000_0000_1000_0100, 3
    = 0
```

Code Examples 18.7 and 18.8 show that the assembly language code for bit manipulation is shorter with Zbs.

Code Example 18.7 SINGLE-BIT OPERATIONS

C Program	Assembly Without Zbs	Assembly With Zbs
<pre>// Clear, invert, set, extract // a, n in a0, a1 // b, c, d, e in s0, s1, s2, s3 b = a & ~(1 << n); // Clear c = a ^ (1 << n); // Invert d = a (1 << n); // Set e = (a >> n) & 1; // Extract</pre>	<pre>li t0, 1 # t0 = 1 sll t0, t0, a1 # t0 = 1 << n not t1, t0 # t1 = ~(1 << n) and s0, a0, t1 # b = a & ~(1 << n) xor s1, a0, t0 # c = a ^ (1 << n) or s2, a0, t0 # d = a (1 << n) srl s3, a0, a1 # e = a >> n andi s3, s3, 1 # e = (a >> n) & 1</pre>	<pre>bclr s0, a0, a1 binv s1, a0, a1 bset s2, a0, a1 bext s3, a0, a1</pre>

Code Example 18.8 BIT CLEAR OPERATION

C Program	Assembly Without Zbs	Assembly With Zbs
<pre>int bitclear(int a, int n) { return a & ~(1 << n); }</pre>	<pre>li t0, 1 # t0 = 1 sll t0, t0, a1 # t0 = 1 << n not t0, t0 # t0 = ~(1 << n) and a0, a0, t0 # b = a & ~(1 << n) ret</pre>	<pre>bclr a0, a0, a1 ret</pre>

18.2.5 Zbkb: Bit Manipulation for Cryptography

Zbkb provides bit manipulation instructions specifically for cryptography. In addition to crypto-related instructions also available in Zbb, it provides `pack`, `packh`, `packw`, `brev8`, `zip`, and `unzip`.

18.2.5.1 Packing

Packing instructions place the bottom portions of two source register into a destination register. This is useful for rearranging small data elements within wider registers. For example, AES uses packing when loading keys that are not word aligned. [Code Example 18.9](#) shows the assembly language code for the `pack` function that stores low halves of registers into a destination. The code is shorter with Zbkb.

- ▶ `pack` places the bottom half of `rs1` and `rs2` into the bottom and top half of `rd`, respectively.
- ▶ `packh` places the bottom byte of `rs1` and `rs2` into a halfword in `rd`.
- ▶ `packw` (RV64Zbkb only) places the bottom halfword of `rs1` and `rs2` into the lower word of `rd`.

```
pack 0x01234567, 0x89ABCDEF = 0xCDEF4567 (RV32)
packh 0x01234567, 0x89ABCDEF = 0x0000EF67
```

Code Example 18.9 PACK OPERATION

C Program	Assembly Without Zbkb	Assembly With Zbkb
<pre>unsigned int pack(unsigned int a, unsigned int b) { return ((a << 16) >> 16) (b << 16); }</pre>	<pre>slli a0, a0, 16 # a << 16 srli a0, a0, 16 # () >> 16 slli a1, a1, 16 # b << 16 or a0, a1, a0 # lo hi ret</pre>	<pre>pack a0, a0, a1 ret</pre>

18.2.5.2 Bit Reversal

The `brev8` instruction reverses the bits within each byte of a register. This is a step in the GHASH (Galois Hash) authentication function for AES encryption and, without `brev8`, this step is painful to implement in software.

```
brev8 00001111_10101010_00110011_01111111
      = 11110000_01010101_11001100_11111110
```

18.2.5.3 Bit Interleaving

64-bit rotations are difficult to implement with 32-bit arithmetic. Various algorithms including SHA-3 require such 64-bit rotations. Hence, RV32Zbkb provides `zip` and `unzip` instructions to facilitate longer rotations. `zip` interleaves bits (like a zipper interleaves its two sides) by placing the bits in the low half of a word into the even bits of the destination and

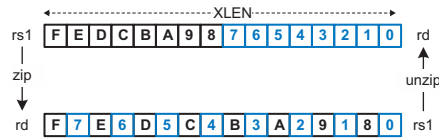


Fig. 18.17 `zip` and `unzip` instructions showing interleaving of bit positions for $XLEN = 16$.

places the bits in the high half into the odd bits. `unzip` does the inverse, as shown in Fig. 18.17.

```
zip   00001111_10101010_00110011_01111111
      = 00000101_10101111_10011101_11011101
unzip 00001111_10101010_00110011_01111111
      = 00111111_01010111_00110000_01011111
```

For example, an RV32 SHA-3 hash implementation uses `unzip` and `zip` to transform 64-bit rotations into pairs of 32-bit rotations on the even and odd halves [github.com/rvkrypto/rvkrypto-fips/blob/main/sha3/sha3_f1600_rvb32.c].

18.2.6 Zbkx: Crossbar Permutations

Crossbar permutations are used to select nibbles or bytes from one register based on indices stored in another register. For example, `xperm4` uses each nibble of `rs2` as an index to select a nibble from `rs1`. The `xperm4` example below shows `rs2` selecting the 5th nibble of `rs1` (2) as the least significant nibble in `rd`, as highlighted in blue. If the index is out of bounds, the permutation returns 0. For example, the `A` index in the `xperm4` example below is out of bounds, so the corresponding result nibble is 0, as highlighted in gray. `xperm8` is similar but selects bytes.

```
xperm4 0xE3217DB3, 0x71023A45 = 0xEB3D7012
xperm8 0xAA_BB_CC_DD, 0x01_00_72_03 = 0xCC_DD_00_AA
```

Crossbar permutations are useful for implementing S-boxes that create nonlinearities in many symmetric key cryptosystems, including AES. The substitution could be done with a table lookup from memory, but cache systems introduce data-dependent delays that can leak information about the key.

Code Example 18.10 shows an example using `xperm8` to implement an arbitrary S-box in software with a data-independent delay. It processes eight byte-wide inputs in parallel packed into a 64-bit word and produces eight outputs in the same format. It uses `sbox`, a 32-element array of 64-bit words that contains the 256-entry $\times$ 1-byte S-box lookup table. Whereas the `mask` is used to step through the upper 5 bits of each byte, `xperm8` uses the lower 3 bits if the upper 5 bits match the `mask`. None of the memory accesses depend on the input, so the delay is independent of the input.

Code Example 18.10 is adapted from github.com/riscv/riscv-crypto/blob/main/benchmarks/permutation/permutation.c.

Code Example 18.10 S-BOX SUBSTITUTION

```
uint64_t sbox_8bit(uint64_t sbox[32], uint64_t in) {
    uint64_t rd = 0;
    uint64_t mask = 0;

    for (int i = 0; i < 32; i++) {
        uint64_t sb_i = sbox[i];
        rd |= xperm8(sb_i, in ^ mask);
        mask += 0x0808080808080808LL;
    }
    return rd;
}
```

18.2.7 Zknd and Zkne: AES Decryption and Encryption

AES decryption and encryption are such common and computationally intensive operations that it is valuable to have specialized instructions to accelerate them. Recall from [Section 18.1.2](#) that AES operates on 128-bit blocks organized as a 4×4 state matrix of bytes. Each encryption round involves the following steps (except that the final round omits MixColumns). Note that the order of ShiftRows and SubBytes does not matter because these operations are commutative.

- ▶ **AddRoundKey:** XOR with key
- ▶ **SubBytes:** Replace each byte based on an S-box lookup
- ▶ **ShiftRows:** Shift each row by 0, 1, 2, or 3 bytes
- ▶ **MixColumns:** Galois field multiplication: 1-bit shifts and XORs

AddRoundKey is straightforward to perform with the XOR instruction. It is not immediately obvious that the bytes could be packed into registers in such a way that RISC-V instructions with two sources and one destination could accelerate the other steps. We will see that RV64 can pack the state matrix into two 64-bit registers in such a way that SubBytes, ShiftRows, and MixColumns can all be performed on the whole matrix with just two identical instructions called `aes64esm`. RV64Zknd and Zkne also provide key scheduling instructions `aes64ks1i` and `aes64ks2` to compute two 32-bit words of the round key at a time [Saarinen20]. On the other hand, RV32 cannot provide the whole state matrix at once and instead operates one byte of one column at a time using `aes32esm`. RV32 computes one word of the round key at a time without specialized key scheduling instructions. The decryption instructions in each architecture are similar to encryption. Compilers cannot generally infer these instructions, so crypto libraries are normally written by domain experts in assembly language. The remainder of this section describes each approach.

18.2.7.1 RV64Zkne: 64-Bit AES Encryption

Let a and b be two 64-bit registers, each organized as eight bytes, $a_{7:0}$ and $b_{7:0}$. Fig. 18.18(a) shows the 16-byte state matrix s , which is packed into a and b in *column-major order*, that is, starting in the upper left and stepping through each row before continuing to the next column. The bytes are in little-endian order with the least-significant byte at the right. Written in vector form,

// a holds columns 0 and 1 of s in column-major order

$$a = [s_{3,1} \ s_{2,1} \ s_{1,1} \ s_{0,1} \ s_{3,0} \ s_{2,0} \ s_{1,0} \ s_{0,0}]$$

// b holds columns 2 and 3 of s in column-major order

$$b = [s_{3,3} \ s_{2,3} \ s_{1,3} \ s_{0,3} \ s_{3,2} \ s_{2,2} \ s_{1,2} \ s_{0,2}]$$

Define a *ShiftRows* operation that takes two 8-byte inputs and produces an 8-byte output:

$$\text{ShiftRows}(x, y) = [x_3 \ y_6 \ y_1 \ x_4 \ y_7 \ y_2 \ x_5 \ x_0] \quad (18.2)$$

Applying *ShiftRows* twice while exchanging the order of the inputs gives the columns of the matrix after the AES *ShiftRows* step, again in column-major order, as shown in Fig. 18.18(b).

// c holds columns 0 and 1 of s after *ShiftRows*

$$c = \text{ShiftRows}(a, b) = [s_{3,0} \ s_{2,3} \ s_{1,2} \ s_{0,1} \ s_{3,3} \ s_{2,2} \ s_{1,1} \ s_{0,0}]$$

// d holds columns 2 and 3 of s after *ShiftRows*

$$d = \text{ShiftRows}(b, a) = [s_{3,2} \ s_{2,1} \ s_{1,0} \ s_{0,3} \ s_{3,1} \ s_{2,0} \ s_{1,3} \ s_{0,2}]$$

a_0 $s_{0,0}$	a_4 $s_{0,1}$	b_0 $s_{0,2}$	b_4 $s_{0,3}$
a_1 $s_{1,0}$	a_5 $s_{1,1}$	b_1 $s_{1,2}$	b_5 $s_{1,3}$
a_2 $s_{2,0}$	a_6 $s_{2,1}$	b_2 $s_{2,2}$	b_6 $s_{2,3}$
a_3 $s_{3,0}$	a_7 $s_{3,1}$	b_3 $s_{3,2}$	b_7 $s_{3,3}$

(a)

c_0 $s_{0,0}$	c_4 $s_{0,1}$	d_0 $s_{0,2}$	d_4 $s_{0,3}$
c_1 $s_{1,1}$	c_5 $s_{1,2}$	d_1 $s_{1,3}$	d_5 $s_{1,0}$
c_2 $s_{2,2}$	c_6 $s_{2,3}$	d_2 $s_{2,0}$	d_6 $s_{2,1}$
c_3 $s_{3,3}$	c_7 $s_{3,0}$	d_3 $s_{3,1}$	d_7 $s_{3,2}$

(b)

Fig. 18.18 Packing the state matrix in column-major order into two 64-bit registers: (a) input and (b) after *ShiftRows*.

Define an *Sbox* operation that takes an 8-byte input and produces an 8-byte output that contains 8 AES substitution-boxes (S-boxes) that operate on the individual bytes.

Define a *MixColumns* operation that takes a 4-byte input column and produces the corresponding 4-byte output column by applying the MixColumns transformation matrix. Recall that MixColumns multiplies elements by 01, 02, or 03 using $\text{GF}(2^8)$. Recall from [Section 18.1.1.2](#) that multiplying by 01 has no effect. Multiplying by 02 (0b10) is a multiply by x (which is a left shift, followed by an XOR for overflow). Multiplying by 03 (0b11) is a multiply by $(x + 1)$, which is the XOR of the original value and the value multiplied by x . Addition is XOR in this field. If the column vector is $z = [z_3 \ z_2 \ z_1 \ z_0]^T$, MixColumns produces:

$$\begin{aligned} \text{MixColumns}(z) &= \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} \\ &= \begin{bmatrix} 02z_0 + 03z_1 + 01z_2 + 01z_3 \\ 01z_0 + 02z_1 + 03z_2 + 01z_3 \\ 01z_0 + 01z_1 + 02z_2 + 03z_3 \\ 03z_0 + 01z_1 + 01z_2 + 02z_3 \end{bmatrix} = \begin{bmatrix} xz_0 \oplus xz_1 \oplus z_1 \oplus z_2 \oplus z_3 \\ z_0 \oplus xz_1 \oplus xz_2 \oplus z_2 \oplus z_3 \\ z_0 \oplus z_1 \oplus xz_2 \oplus xz_3 \oplus z_3 \\ xz_0 \oplus z_0 \oplus z_1 \oplus z_2 \oplus xz_3 \end{bmatrix} \\ &\quad (18.3) \end{aligned}$$

Finally define the `aes64esm` instruction, which takes two 64-bit inputs and produces a 64-bit output:

```

aes64esm(x, y):
    w = ShiftRows(x, y) // Wires permute inputs
    s = Sbox(w)          // S-box operations on all 8 bytes in parallel
    // GF(2^8) multiplications and XORs on two columns in parallel
    return t = {MixColumns(s63:32), MixColumns(s31:0)}
```

If x and y contain the first and second pair of columns, respectively, w contains the first pair of columns after ShiftRows. s contains this pair of columns after SubBytes. t contains the pair of columns after MixColumns is applied to each.

Code Example 18.11 shows code for an AES encryption round using `aes64esm`. The state matrix s is stored in $\{a1, a0\}$ in column-major form. The RoundKey is in $\{a3, a2\}$. `AddRoundKey` uses two XOR instructions to update the state registers into $\{t1, t0\}$. Then two `aes64esm` instructions perform the ShiftRows, SubBytes, and MixColumns operations, putting the new state back into $\{a1, a0\}$. Observe how the `aes64esm` source registers, $t0$ and $t1$, are simply reversed to select the pair of columns.

Code Example 18.11 RV64Zkne AES ENCRYPTION ROUND

```
AES_ENCRYPT_ROUND:
    xor      t0, a0, a2    // AddRoundKey columns 0, 1
    xor      t1, a1, a3    // AddRoundKey columns 2, 3
    aes64esm a0, t0, t1    // Rest of round for columns 0, 1
    aes64esm a1, t1, t0    // Rest of round for columns 2, 3
```

Code Example 18.11 is adapted from a RISC-V AES encryption library at github.com/riscv/riscv-crypto/blob/main/benchmarks/aes/zscrypto_rv64/aes_enc.S.

18.2.7.2 RV32Zkne: 32-bit AES Encryption

RV32 holds the 128-bit state matrix s in four 32-bit registers, as shown in Fig. 18.19 and as expressed in vector form next. Encryption is performed byte by byte because RV32Zkne instructions with two source registers are not able to provide the entire state matrix at once as was done in RV64Zkne.

$$\begin{aligned} a &= [s_{3,0} \ s_{2,0} \ s_{1,0} \ s_{0,0}] && // a \text{ holds column 0 of } s \\ b &= [s_{3,0} \ s_{2,0} \ s_{1,0} \ s_{0,0}] && // b \text{ holds column 1 of } s \\ c &= [s_{3,2} \ s_{2,2} \ s_{1,2} \ s_{0,2}] && // c \text{ holds column 2 of } s \\ d &= [s_{3,3} \ s_{2,3} \ s_{1,3} \ s_{0,3}] && // d \text{ holds column 3 of } s \end{aligned}$$

Define the `aes32esmi` instruction, which takes two 32-bit inputs and a 2-bit byte-select bs . The first input x is initialized as the key and transformed into a new column of the state matrix 1 byte at a time using four `aes32esmi` instructions. The second input y is an old column of the state

a_0 $s_{0,0}$	b_0 $s_{0,1}$	c_0 $s_{0,2}$	d_0 $s_{0,3}$
a_1 $s_{1,0}$	b_1 $s_{1,1}$	c_1 $s_{1,2}$	d_1 $s_{1,3}$
a_2 $s_{2,0}$	b_2 $s_{2,1}$	c_2 $s_{2,2}$	d_2 $s_{2,3}$
a_3 $s_{3,0}$	b_3 $s_{3,1}$	c_3 $s_{3,2}$	d_3 $s_{3,3}$

Fig. 18.19 Packing the state matrix into four 32-bit registers.

matrix. The instruction selects one of the four byte-wide elements of the column and applies the S-box for SubBytes. It then multiplies by $\begin{bmatrix} 3 & 1 & 1 & 2 \end{bmatrix}$ in $GF(2^8)$, and rotates the result left by bs to implement the selected row of the MixColumns matrix. Finally, it XORs with x to accumulate the rows of the matrix and complete the AddRoundKey step.

```

aes64esm(x, y, bs):
    si = y[bs]           // Select one of the four elements of y
    s = sbox(si)         // S-box on one element
    col = {xs ⊕ s, s, s, xs} // One row of MixColumns matrix
    mixed = col rol bs    // Rotate col into proper place
    return t = x ⊕ mixed

```

Code Example 18.12 shows code for an AES encryption round using `aes32esmi`. The state matrix is stored in $\{a3, a2, a1, a0\}$. The RoundKey is in $\{t3, t2, t1, t0\}$. Then 16 `aes32esmi` instructions perform the AES round AddRoundKey, ShiftRows, SubBytes and MixColumns steps, putting the results back into $\{t3, t2, t1, t0\}$. Observe how the second argument varies to perform ShiftRows, and the byte select iterates from 0 to 3 to handle each byte in a column. The final round, not shown, uses the `aes32esi` instruction instead, which omits the MixColumns step.

Code Example 18.12 RV32Zkne AES ENCRYPTION ROUND

```

AES_ENCRYPT_ROUND:
    aes32esmi    t0, t0, a0, 0
    aes32esmi    t0, t0, a1, 1
    aes32esmi    t0, t0, a2, 2
    aes32esmi    t0, t0, a3, 3
    aes32esmi    t1, t1, a1, 0
    aes32esmi    t1, t1, a2, 1
    aes32esmi    t1, t1, a3, 2
    aes32esmi    t1, t1, a0, 3
    aes32esmi    t2, t2, a2, 0
    aes32esmi    t2, t2, a3, 1
    aes32esmi    t2, t2, a0, 2
    aes32esmi    t2, t2, a1, 3
    aes32esmi    t3, t3, a3, 0
    aes32esmi    t3, t3, a0, 1
    aes32esmi    t3, t3, a1, 2
    aes32esmi    t3, t3, a2, 3

```

18.2.7.3 RV{32/64}Zknd: AES Decryption

The inverse of AES encryption is decryption, which uses `aes32dsmi`, `aes32dsi`, `aes64dsm`, and `aes64dsi` instructions. These instructions apply inverse operations, in the same order as encryption. They use an inverse S-box and the InvMixColumns matrix from Fig. 18.13 (or the $\begin{bmatrix} b & d & 9 & e \end{bmatrix}$

row for `aes32dsml`). `ShiftRows` is inverted by shifting in the opposite direction.

18.2.7.4 RV64Zkn{e/d} AES Key Scheduling

Recall from Fig. 18.10 that `RoundKeys` are generated using a sequence of rotations, S-box substitutions, and XORs. RV32 does this in software using RISC-V `rori` and `xor` instructions along with `aes32esi` for an S-box.

RV64Zkn{e/d} introduce `aes64ksli`, `aesks2`, and `aes64im` instructions to accelerate key scheduling by processing two 32-bit words at a time in a 64-bit register and applying four S-boxes in parallel. See the ISA specification and [\[github.com/riscv/riscv-crypto/blob/main/benchmarks/aes/zscrypto_rv64/aes_128_ks.S\]](https://github.com/riscv/riscv-crypto/blob/main/benchmarks/aes/zscrypto_rv64/aes_128_ks.S) for details.

18.2.8 Zknh: SHA-2 Instructions

SHA-2 is another common and computationally intensive operation. Recall from Section 18.1.3 that each round applies the σ_0 and σ_1 functions to create the message schedule and the Σ_0 and Σ_1 functions to perform the hash. These functions are defined using rotations, shifts, and XOR in Table 18.7. Recall that whereas SHA-256 uses 32-bit words, SHA-512 uses 64-bit words, and that the four functions also differ accordingly. The RISC-V Zknh extension provides instructions to accelerate these functions.

For SHA-256, the instructions are called `sha256sig0`, `sha256sig1`, `sha256sum0`, and `sha256sum1`, with `sig` corresponding to σ and `sum` corresponding to Σ . Each instruction takes a register containing a 32-bit value and applies the rotations, shifts, and XORs to produce a 32-bit output. RV64Zknh sign-extends the results to fill the upper 32 bits, but these bits are ignored during hashing.

```
sha256sig0: rd = SignExt( $\sigma_0^{(256)}(rs1_{31:0})$ ) = (rs1 ror 7)   $\oplus$  (rs1 ror 18)  $\oplus$  (rs1 >> 3)
sha256sig1: rd = SignExt( $\sigma_1^{(256)}(rs1_{31:0})$ ) = (rs1 ror 17)  $\oplus$  (rs1 ror 19)  $\oplus$  (rs1 >> 10)
sha256sum0: rd = SignExt( $\Sigma_0^{(256)}(rs1_{31:0})$ ) = (rs1 ror 2)   $\oplus$  (rs1 ror 13)  $\oplus$  (rs1 ror 22)
sha256sum1: rd = SignExt( $\Sigma_1^{(256)}(rs1_{31:0})$ ) = (rs1 ror 6)   $\oplus$  (rs1 ror 11)  $\oplus$  (rs1 ror 25)
```

In RV64Zknh, SHA-512 uses a similar set of instructions: `sha512sig0`, `sha512sig1`, `sha512sum0`, and `sha512sum1`. They operate on 64-bit words and use the SHA-512 rotates and shifts.

```
sha512sig0: rd =  $\sigma_0^{(512)}(rs1)$  = (rs1 ror 1)   $\oplus$  (rs1 ror 8)   $\oplus$  (rs1 >> 7)
sha512sig1: rd =  $\sigma_1^{(512)}(rs1)$  = (rs1 ror 19)  $\oplus$  (rs1 ror 61)  $\oplus$  (rs1 >> 6)
sha512sum0: rd =  $\Sigma_0^{(512)}(rs1)$  = (rs1 ror 28)  $\oplus$  (rs1 ror 34)  $\oplus$  (rs1 ror 39)
sha512sum1: rd =  $\Sigma_1^{(512)}(rs1)$  = (rs1 ror 14)  $\oplus$  (rs1 ror 18)  $\oplus$  (rs1 ror 41)
```

In RV32Zknh, SHA-512 is somewhat more difficult because the 64-bit words do not fit in a single register. Instead, the words are kept in a pair of registers, and a pair of instructions are needed to compute the upper and lower half of each function.

Code Example 18.13 shows how a SHA-512 library implements the round and schedule operations with the RV64Zknh instructions. Note how C invokes the assembly language instructions.

Code Example 18.13 is adapted from: github.com/riscv/riscv-crypto/blob/main/benchmarks/sha512/zscrypto_rv64/sha512.c.

Code Example 18.13 SHA-512 ROUND AND SCHEDULE OPERATIONS

```
static inline uint_xlen_t _sha512sig0 (uint_xlen_t rs1) {uint_xlen_t rd; __asm__
("sha512sig0 %0, %1" : "=r"(rd) : "r"(rs1)); return rd;}
...

#define ROR64(X,Y) ((X>>Y) | (X << (64-Y)))
#define SHR64(X,Y) ((X>>Y) )
#define CH(X,Y,Z) ((X&Y)^(~X&Z))
#define MAJ(X,Y,Z) ((X&Y)^(X&Z)^(Y&Z))
#define SUM_0(X) (_sha512sum0(X))
#define SUM_1(X) (_sha512sum1(X))
#define SIGMA_0(X) (_sha512sig0(X))
#define SIGMA_1(X) (_sha512sig1(X))

#define ROUND(A, B, C, D, E, F, G, H, K, W) { \
    H = H + SUM_1(E) + CH(E,F,G) + K + W ; \
    D = D + H ; \
    H = H + SUM_0(A) + MAJ(A, B, C) ; \
}

#define SCHEDULE(M0, M1, M9, ME) { \
    M0 = SIGMA_1(ME) + M9 + SIGMA_0(M1) + M0; \
}
```

18.3 TEST PLAN

The bit manipulation and cryptographic instructions are tested with the riscv-arch-test suite. The configuration files to generate these random tests are similar to the one described in Section 7.4. Tables 18.10 and 18.11 list the number of test cases for each instruction.

18.4 WALLY IMPLEMENTATION

Wally extends the integer ALU to support cryptography and bit manipulation operations, as highlighted in Fig. 18.20.

18.4.1 ALU Modifications for Bit Manipulation

The ZBA_SUPPORTED, ZBB_SUPPORTED, ZBC_SUPPORTED, and ZBS_SUPPORTED configuration options determine which bit manipulation instructions are

Table 18.10 riscv-arch-test for Zb* extensions.

Instruction	RV32B Tests	RV64B Tests
add.uw	-	701
andn	158	705
bclr	292	421
bclri	291	158
bext	291	419
bexti	158	158
binv	291	421
binvi	158	158
bset	291	419
bseti	158	158
clmul	162	288
clmulh	162	288
clmulr	162	288
clz	68	132
clzw	-	132
cpop	68	132
cpopw	-	132
ctz	68	132
ctzw	-	132
max	572	700
maxu	572	700
min	572	700
minu	572	700
orc.b	72	136
orn	592	720
rev8	154	155
rol	269	525
rolw	-	525
ror	269	525
rori	158	158
roriw	-	158
rorw	-	525
sext.b	87	151
sext.h	88	152
shladd	572	700
shladd.uw	-	700
sh2add	572	700
sh2add.uw	-	700
sh3add	572	700
sh3add.uw	-	700
slli.uw	-	161
xnor	592	720
zext.h	88	152
Total	9,151	17,137

Table 18.11 riscv-arch-test for Zk* extensions.

Instruction	RV32Zk Tests	RV64Zk Tests
aes32dsi	281	-
aes32dsmi	281	-
aes32esi	281	-
aes64ds	-	57
aes64dsm	-	57
aes64es	-	57
aes64esm	-	57
aes64im	-	150
aes64ks1i	-	86
aes64ks2	-	280
brev8	154	136
pack	269	525
packh	269	525
packw	-	525
sha256sig0	86	150
sha256sig1	86	150
sha256sum0	86	150
sha256sum1	86	150
sha512sig0	-	150
sha512sig0h	153	-
sha512sig0l	153	-
sha512sig1	-	150
sha512sig1h	153	-
sha512sig1l	153	-
sha512sum0	-	150
sha512sum0r	153	-
sha512sum1	-	150
sha512sum1r	153	-
unzip	132	-
xperm4	289	545
xperm8	289	395
zip	132	-
Total	3,639	4,595

supported, if any. The Wally rv32gc and rv64gc configurations support all four flavors, and the other configurations support none.

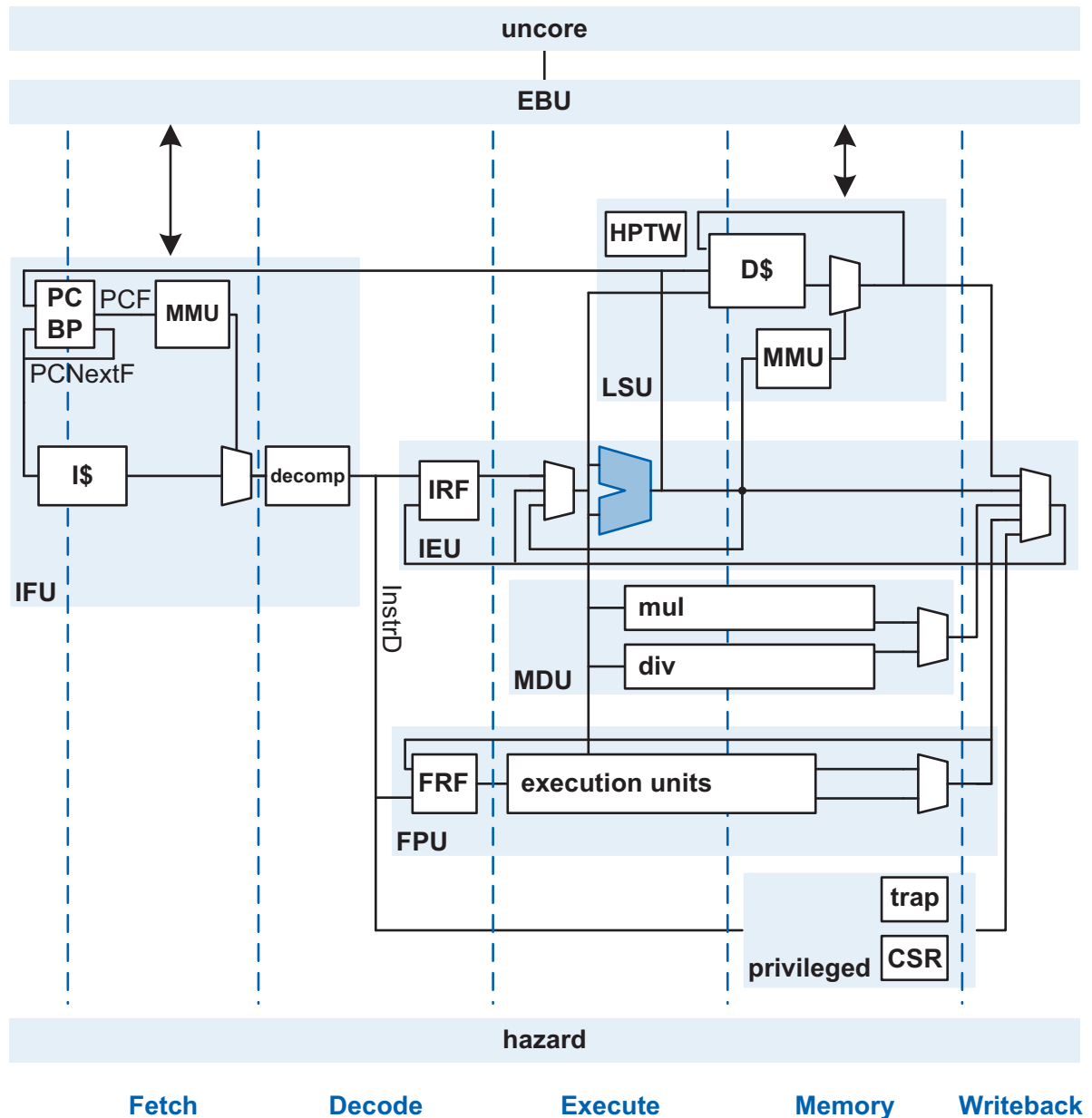


Fig. 18.20 RV{32/64}GC high-level block diagram emphasizing the ALU with bit manipulation extensions.

Fig. 18.21 shows a diagram of the ALU with optional enhancements for bit manipulation in the `bitmanipalu` module, highlighted in blue. The major changes are:

- ▶ **Zba:** Preshifter to shift A left by 0, 1, 2, or 3 bits. For RV64 (cross-hatched), conditionally zero-extend A for `.uw` instructions.
- ▶ **Zbb:** Block for min/max, counting, extension, combining, and byte reversal. The shifter is also enhanced to support rotations.
- ▶ **Zbc:** Block for carry-less multiplication.
- ▶ **Zbs:** Decoder to produce one-hot mask from B. Reduction OR block for bit extract operations.

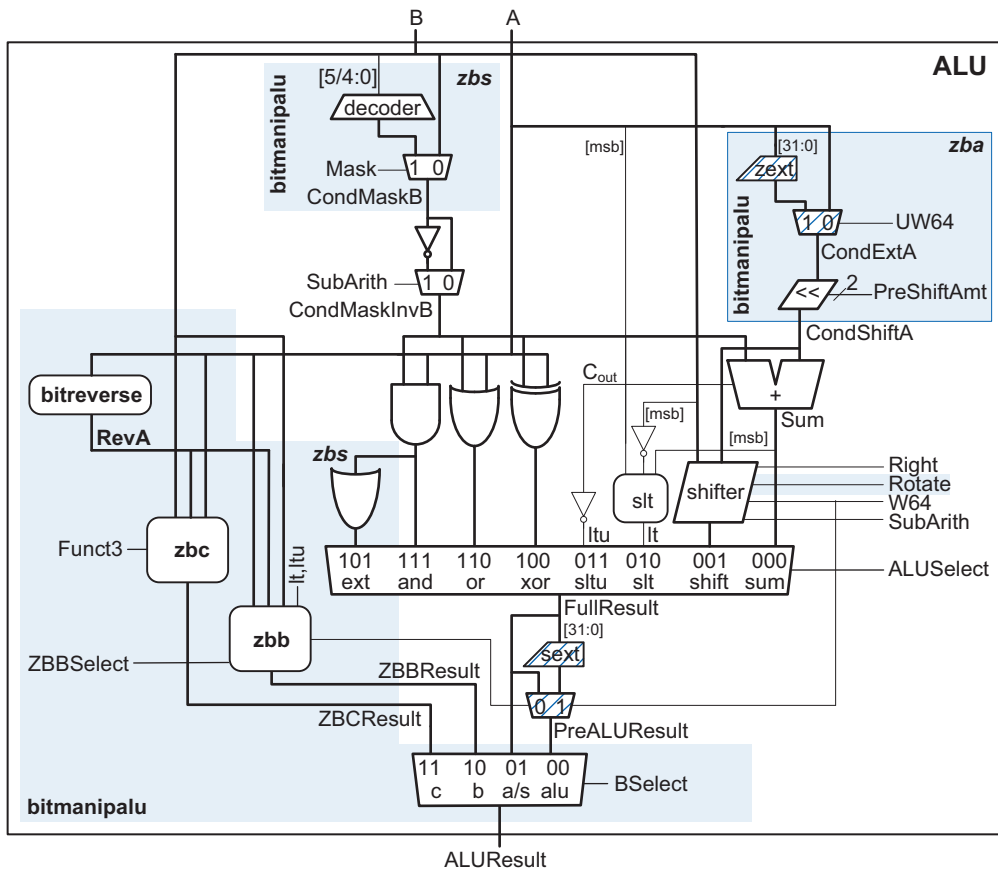


Fig. 18.21 ALU, with optional enhancements for bit manipulation in blue boxes.

Verilog lacks a graceful idiom to reverse bits in a bus. The least bad way is with a generate loop.

```
for (i=0; i<WIDTH; i++)
    assign RevA[WIDTH-i-1]
        = A[i];
```

When any of the bit manipulation extensions are supported, the ALU uses a BSelect multiplexer to choose ALUResult from the regular ALU (PreALUResult) or from the bit manipulation modules. The Rotate input to the shifter (shift) is only used by the zbb unit. The bitreverse module reverses the bits in A to produce RevA by using wires only, not gates. Bit reversal is used for counting trailing zeros in Zbb and for carry-less multiplication in Zbc.

18.4.1.1 Zba

The Zba extension adds a preshifter and zero-extender on A to support shift-and-add and .uw instructions.

18.4.1.2 Zbb

The Zbb extension adds a zbb block to perform min/max (min, minu, max, maxu), extension (sext.b, sext.h, zext.h), counting (cpop, clz, ctz), combining (orc.b), and byte reversal (rev8). It also enhances the shifter to support rotates (rol, ror) and uses the existing inverter on the B input to support inverted logical operations (orn, andn, xnor).

Rotates

The zbb module extends the funnel shifter source generator from [Section 7.1.3.3](#) to support rotates. [Table 18.12](#) describes the source generator for RV64, which produces the source to shift, Z. In the table, RotA = {A_{31:0}, A_{31:0}} for W64 rotates (rolw, rorw), and RotA = A_{63:0} for ordinary rotates (rol, ror). [Table 18.13](#) describes the source generator for RV32, which is simpler without the W64 operations.

Table 18.12 64-bit shifter source generator and shift amount that supports rotates.

Shift Type	z	Shift Amount
srl	{63'b0, A _{63:0} }	k
sra	{{63{A ₆₃ }}, A _{63:0} }	k
sll	{A _{63:0} , 63'b0}	$\bar{k}$
srlw	{95'b0, A _{31:0} }	k
sraw	{64'b0, {31{A ₃₁ }}, A _{31:0} }	k
sllw	{32'b0, A _{31:0} , 63'b0}	$\bar{k}$
ror / rorw	{RotA _{62:0} , RotA _{63:0} }	k
rol / rolw	{RotA _{63:0} , RotA _{63:1} }	$\bar{k}$

Table 18.13 32-bit shifter source generator and shift amount that supports rotates.

Shift Type	z	Shift Amount
srl	{31'b0, A _{31:0} }	<i>k</i>
sra	{{31{A ₃₁ }}, A _{31:0} }	<i>k</i>
sll	{A _{31:0} , 31'b0}	$\overline{k}$
ror	{A _{30:0} , A _{31:0} }	<i>k</i>
rol	{A _{31:0} , A _{31:1} }	$\overline{k}$

Example 18.4 FUNNEL SHIFTER

Show how the 64-bit funnel shifter performs `rorw 0x01234567_C9ABCDEF, 8`.

Solution: RotA = {A_{31:0}, A_{31:0}} = C9ABCDEF_C9ABCDEF. Z = {RotA_{62:0}, RotA_{63:0}} = 49ABCDEF_C9ABCDEF_C9ABCDEF_C9ABCDEF; note that Z is 127 rather than 128 bits. *k* = 8. The funnel shifter shifts right by 8 bits and selects the bottom 64 bits of the result FullResult = EFC9ABCD_EFC9ABCD. Finally, the W64 mux truncates to 32 bits and sign extends: ALUResult = FFFFFFFF_EFC9ABCD. Thus, the shifter performed an 8-bit right rotation on the 32-bit word.

Inverted Operations

For inverted operations (`orn`, `andn`, `xnor`), the ALU performs $A \text{ op } \overline{B}$ using the inverter already available from subtraction, so no new datapath hardware is required. The ALU controller drives `SubArith` = 1 to select $\overline{B}$ as the second ALU operand.

zbb Module

Fig. 18.22 shows the `zbb` module. Each of the min/max, extension, counting, and byte operations are described below. For I-type instructions, B contains the immediate, and certain lsbs of B are used for control.

The min/max (`min`, `minu`, `max`, `maxu`) operations select either A or B based on either the *less than* (`lt`) or *less than unsigned* (`ltu`) flag from the ALU.

Sign and zero extension select from among the three possible values for `sxt.b`, `sxt.h`, and `zext.h` using the `ext` module.

Fig. 18.23 shows the `cnt` module that handles `cpop`, `clz`, and `ctz`. It contains `popcnt` and `lzc` modules for counting either population (total number of ones) or leading zeros. It counts trailing zeros by reversing the input bits and counting leading zeros. Recall that `RevA` is the bit-reversed version of A. Fig. 18.24 shows a simplified 8-bit version of the

The Verilog idiom for a configurable-width leading zero counter and priority encoder involves an unusual loop syntax.

```
int i;
always_comb begin
    i = 0;
    while ((i < WIDTH) &
           ~num[WIDTH-1-i])
        i = i + 1;
    ZeroCnt =
        i[$clog2(WIDTH+1)-1:0];
end
```

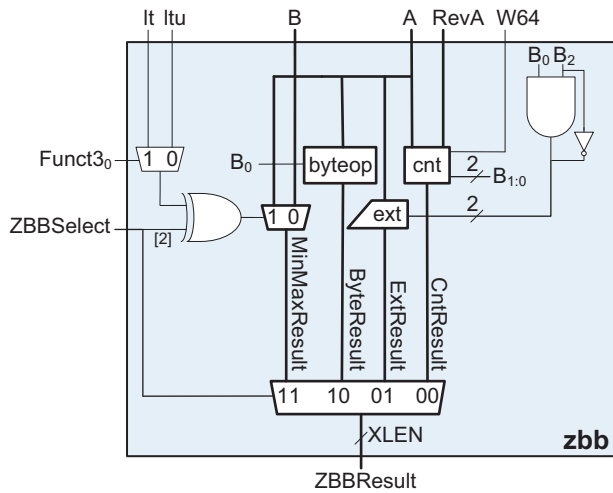


Fig. 18.22 zbb module.

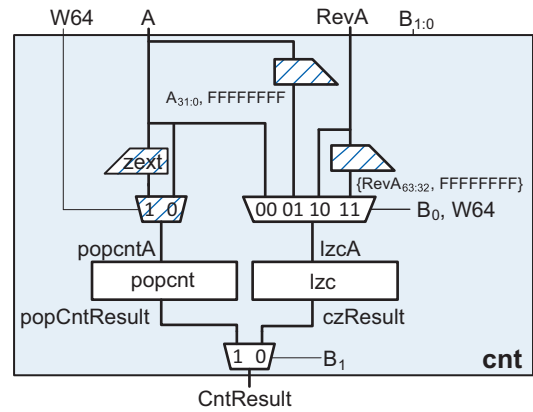


Fig. 18.23 cnt module.

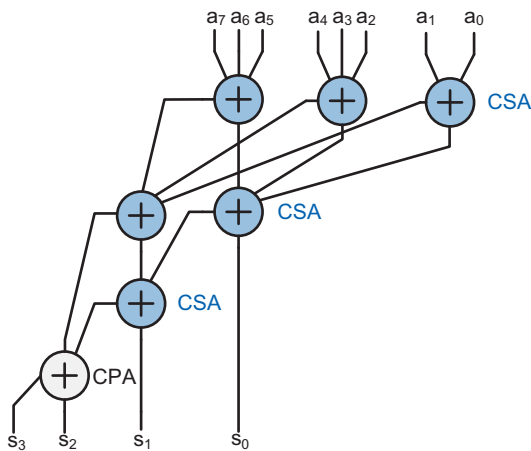


Fig. 18.24 8-bit popcnt module.

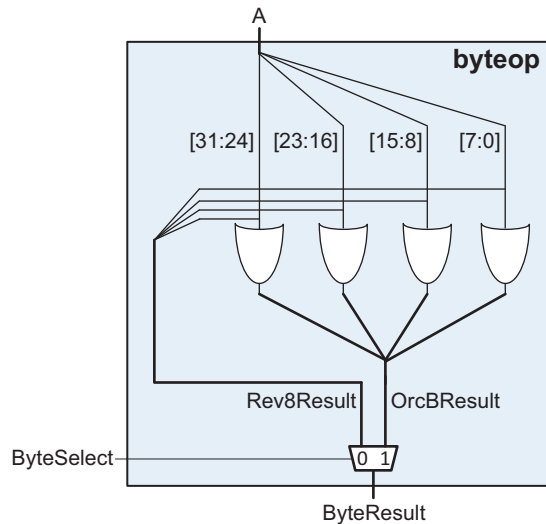


Fig. 18.25 32-bit byteop module.

popcnt module needed for the cpop instruction. In general, the module uses an XLEN:2 tree of carry-save adders (CSAs), followed by a carry-propagate adder (CPA). The leading zero counter is simply a priority encoder with a binary output.

Fig. 18.25 shows the byteop module that handles orc.b and rev8. It contains one 8:1 OR gate for each byte to do combining and a network of wires to reverse bytes.

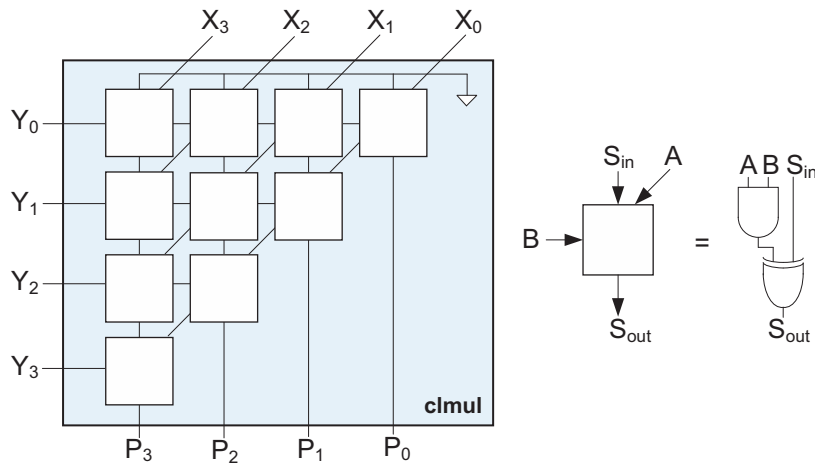


Fig. 18.26 4-bit `c1mul` module.

18.4.1.3 Zbc

The `zbc` unit performs three flavors of carry-less multiplication: `clmul`, `clmulh`, and `clmulr`. Multiplying two XLEN-bit inputs X and Y gives a $2XLEN$ -bit product P . `clmul` returns the lower half and `clmulh` returns the upper half. `clmulr` is similar to `clmulh` and is discussed later in this section.

Carry-less multiplication is similar to ordinary multiplication from Fig. 15.4(a) but does not propagate carries between columns. Hence, the full adders reduce to simple XOR gates to sum the carries within a column modulo 2. Only the least significant half of the multiplier is necessary to obtain an XLEN-sized result. Fig. 18.26 shows a 4-bit version of a the `cmul` (carry-less multiplication) module. In this section, the `@` operator refers to this carry-less multiplication.

The `clmulh` instruction computes the upper XLEN bits of a carry-less multiplication. Fig. 18.27(a) shows a complete 2XLEN-sized carry-less multiplier, where XLEN = 4, with the upper half highlighted in blue. If the blue bits are flipped horizontally and vertically as shown in Fig. 18.27(b), we see they have the same structure as Fig. 18.26 except that the inputs are reversed, Y is shifted left by 1, and then the outputs are reversed: `clmulh = rev(rev(X) @ (rev(Y) << 1))`. Hence, `clmulh` is executed on the same `clmul` module by bit-reversing and shifting the inputs.

`clmulr` is similar to `clmulh` but omits the shift on `Y`: `clmulr = rev(rev(X) @ rev(Y))`. Hence, `clmulh = clmulr >> 1`.

Fig. 18.28 shows a `zbc` module that performs all three flavors of carry-less multiplication. It uses a single `clmul` carry-less multiplication array and

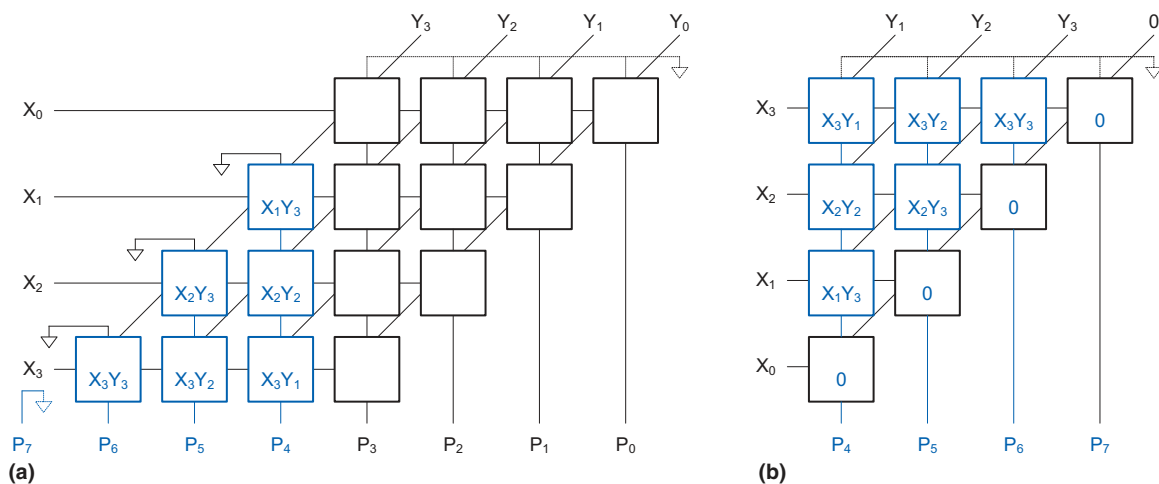


Fig. 18.27 Carry-less multiplication: (a) 8-bit and (b) 4-bit `c1mulh` using a 4-bit `c1mul` array.

conditionally reverses the inputs and outputs using two `bitreverse` modules and the already bit-reversed input `RevA` from the `bitmanipalu` module.

[Code Example 18.14](#) shows the Verilog for the carry-less multiplier. SystemVerilog lacks a built-in idiom, so the module uses two nested loops to create the $N^2/2$ AND and XOR gates. Unfortunately, for $N = 64$, the inner loop is evaluated nearly 2,048 times. `Zbc` slows down `rv64gc` simulations by at least 20% because of these loops.

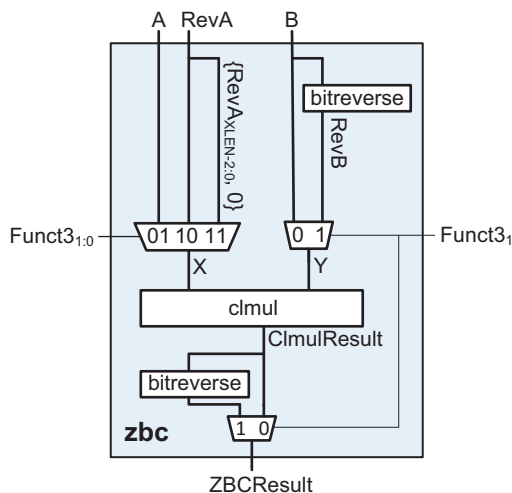


Fig. 18.28 `zbc` module.

Code Example 18.14 CARRY-LESS MULTIPLY MODULE (`clmul`)

```

module clmul #(parameter N=32) (
  input  logic [N-1:0] X, Y,           // Operands
  output logic [N-1:0] ClmulResult); // Zbc result

  logic [(N*N)-1:0] S;                // Intermediate signals for carry-less multiply

  integer i, j;

  always_comb begin
    for (i=0; i<N; i++) begin: outer
      S[N*i] = X[0] & Y[i];
      for (j = 1; j <= i; j++) begin: inner
        S[N*i+j] = (X[j] & Y[i-j]) ^ S[N*i+j-1];
      end
      ClmulResult[i] = S[N*i+j-1];
    end
  end
endmodule

```

18.4.1.4 Zbs

As shown in Fig. 18.21, the Zbs extension uses a decoder to create a mask with a single one in the column specified by B: $\text{CondMaskB} = 1 \ll B$. The ALU then performs logical operations:

- ▶ bset: $\text{rd} = A \mid \text{CondMaskB}$
- ▶ bclr: $\text{rd} = A \ \& \ \sim \text{CondMaskB}$
- ▶ binv: $\text{rd} = A \ ^ \text{CondMaskB}$
- ▶ bext: $\text{rd} = \mid (A \ \& \ \text{CondMaskB})$

The ALU also adds an XLEN:1 OR gate for `bext` to determine if the masked bit is true.

18.4.2 ALU Modifications for Cryptography

Fig. 18.29 shows a diagram of the ALU with optional enhancements for cryptography highlighted in different shades of blue and grey. The major changes are:

- ▶ **Zbkb**: Extension to bit manipulations for cryptography beyond Zbb.
- ▶ **Zbkx**: Block for permutations instructions.
- ▶ **Zknde**: Block for AES decryption and/or encryption.
- ▶ **Zknh**: Block for SHA-2 operations.

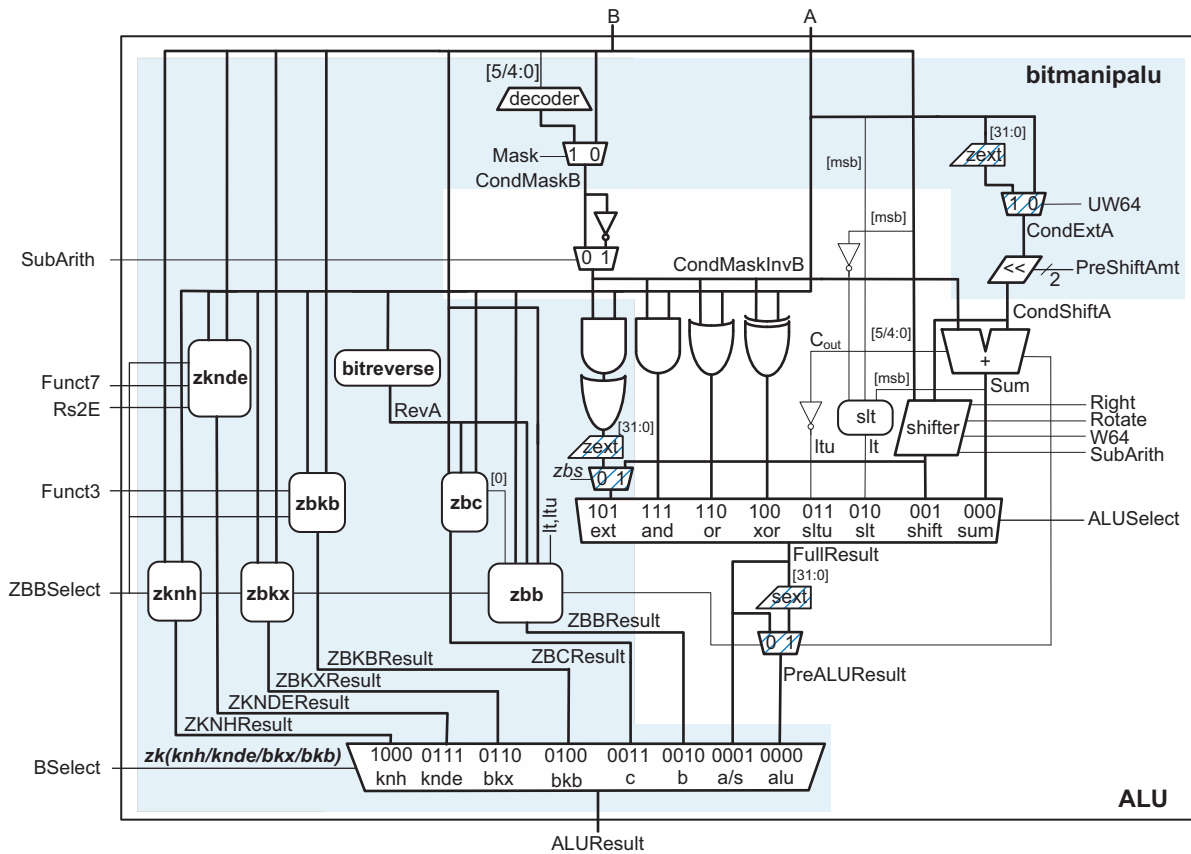


Fig. 18.29 alu enhancements for cryptography.

When any of cryptography extensions are supported, the ALU adds the supported extensions to the bitmanipalu and extends the ALUResult multiplexer to choose the new results.

18.4.2.1 Zbkb

The Zbkb extension includes a subset of the Zbb instructions, along with new zip/unzip, pack/packh/packw, and brev8 instructions. The Zbkb unit, shown in Fig. 18.30, executes these new crypto bit manipulation instructions. Note that the packw instruction is only implemented in RV64, and zip/unzip are only implemented on RV32.

The zipper module contains a rat's nest of wiring to interleave bits and a 2-input mux to choose zip or unzip. The packer also contains wiring to rearrange the bits and a 3-input mux to choose among the flavors of pack. brev8 simply contains wiring to reverse bits within the bytes.

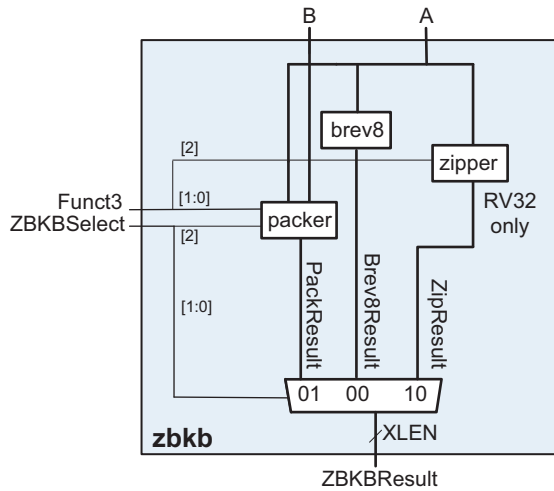


Fig. 18.30 zbkB module.

18.4.2.2 Zbkx

The `zbx` module performs `xperm4` and `xperm8` permutations, as shown in Fig. 18.31.

A `xperm $k$` module of size k (4 or 8) produces $n = \text{XLEN}/k$ chunks of k -bit outputs by selecting from one of n k -bit chunks of the input A based on an index in the corresponding k -bit chunk of the input B . If the index exceeds n (i.e., its upper bits are nonzero), the index is illegal and the chunk is set to 0. This is done with n k -bit $n:1$ multiplexers, and some OR-AND logic for illegal indices. Fig. 18.32 shows an `xperm8` module for `rv32zbx` ($k = 8$, $n = 4$).

The logic for an `xperm $k$` block can be described succinctly with System-Verilog's part select notation (see Section 4.2.3.3.3) choosing k -bit chunks $B[i+:k]$, as shown in Code Example 18.15 for $k=8$. The right shift by a

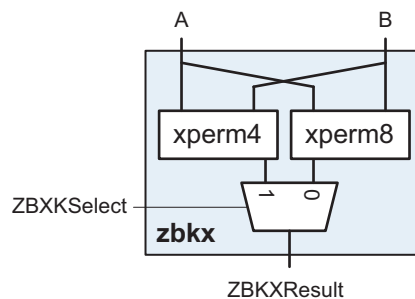


Fig. 18.31 zbkx module.

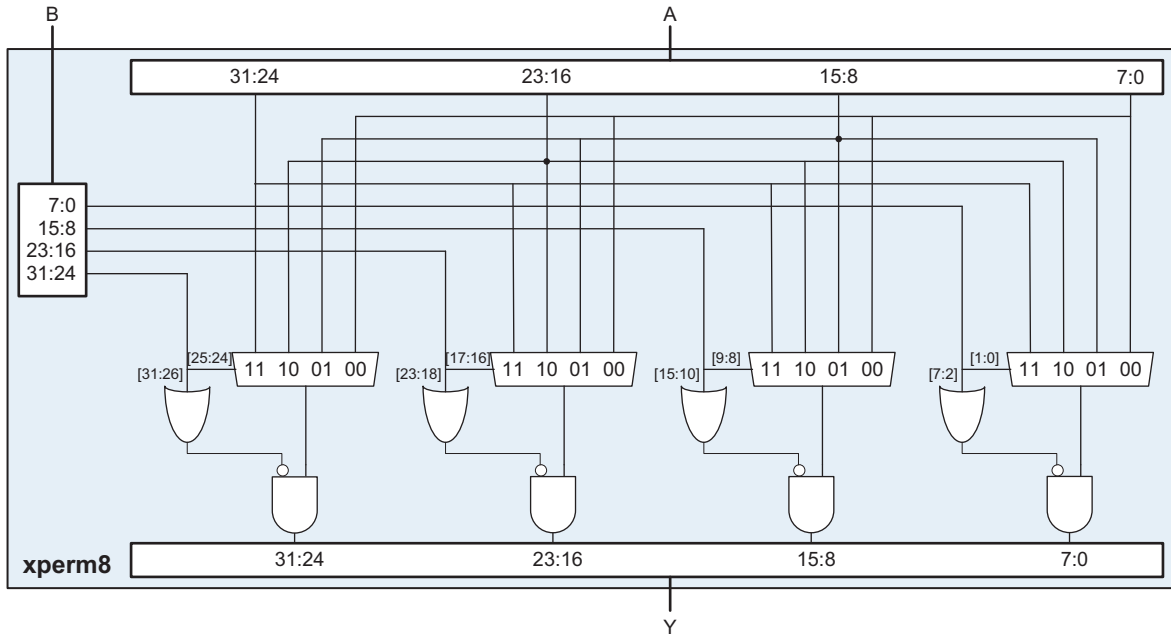


Fig. 18.32 `xperm8` for `rv32zbbkx`.

multiple of 8 implies the $m:1$ mux and the OR-AND logic for handling out-of-range indices.

Code Example 18.15 `xperm8` USING PART-SELECT NOTATION

```
for (i = 0; i < WIDTH; i = i + 8) begin: xperm8calc
  xperm8lookup = A >> {B[i+:8], 3'b0};
  xperm8[i+:8] = xperm8lookup[7:0];
end
```

18.4.2.3 `Zknd` and `Zkne`

`Zknd` and `Zkne` are described together because they share common hardware. Recall that they accelerate the AES SubBytes, ShiftRows, and MixColumns operations with a single instruction. However, as described in [Section 18.2.7](#), AES encryption and decryption instructions are fairly different for RV32 versus RV64 because RV32 cannot hold the entire state matrix in a pair of registers. Hence, whereas RV32 only processes one byte at a time, RV64 processes 8 bytes at a time. The largest portion of the modules is the S-boxes, which are $256 \text{ entry} \times 8\text{-bit}$ lookup tables. RV64 uses 8

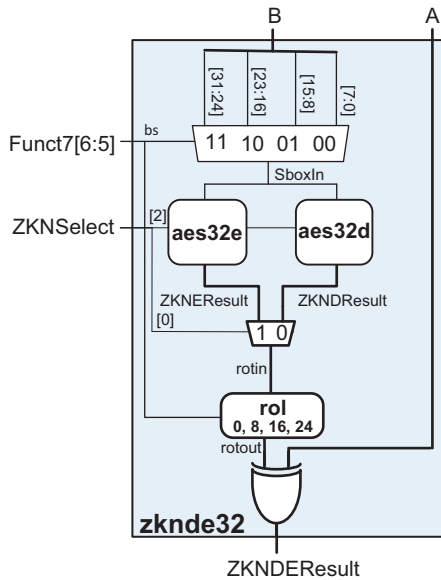


Fig. 18.33 zknde32 encrypt/decrypt module.

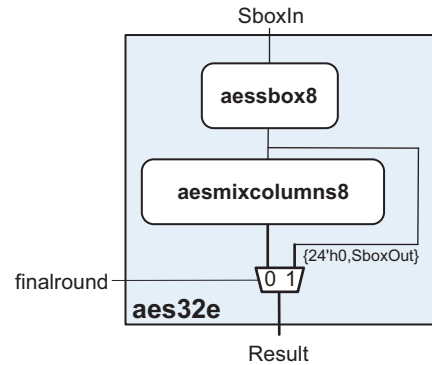


Fig. 18.34 32-bit aes32e encrypt module.

S-box and 8 inverse S-box tables, while RV32 only uses one of each and only delivers $1/8^{\text{th}}$ of the throughput. This section begins with the 32-bit implementation, then describes the 64-bit implementation.

zknde32

RV32 instructions `aes32esm`, `aes32es`, `aes32dsmi`, and `aes32dsi` accelerate AES one byte at a time selected from a 32-bit register containing 4 bytes of the state matrix. The `zknde32` module shown in Fig. 18.33 selects the byte from the 32-bit input. The `aes32e` and `aes32d` modules handle encryption and decryption steps, respectively. Either could be omitted if only `Zkne` or `Zknd` is supported. `zknde32` then selects the desired operation, rotates the result into place, and XORs it into a running result.

Fig. 18.34 shows the `aes32e` module. It applies `aessbox8` to the selected entry, then `aesmixcolumns8` to the output word to perform one step of the SubBytes and MixColumns steps, respectively. `aessbox8` performs one step of SubBytes according to the lookup table from Fig. 18.7. `aesmixcolumns8` computes 32 bits of MixColumns based on the 1-byte input a according to Equation 18.4, which involves one $\text{GF}(2^8)$ multiplication by x and one XOR. A multiplexer optionally skips the MixColumns step for the final round (`aes32es`). The `aes32d` module is similar except that it uses the `aesinvsubox8` and `aesinvmixcolumns8` modules.

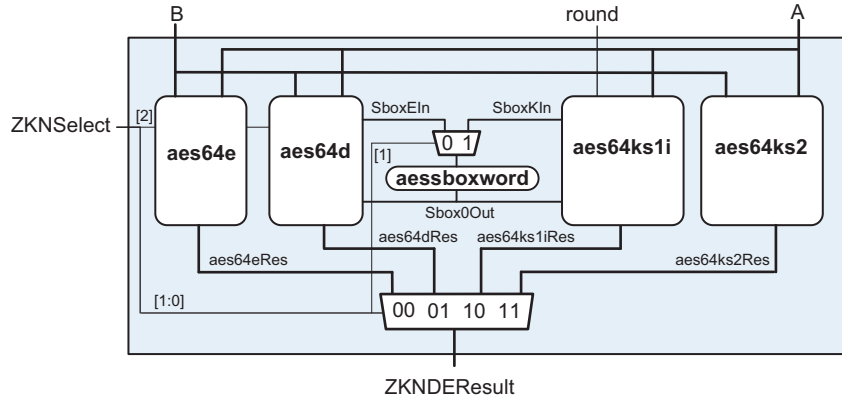


Fig. 18.35 `zknde64` encrypt/decrypt module.

$$\begin{aligned} \text{aesmixcolumns8} &= \{3 \ 1 \ 1 \ 2\}a = \{x + 1, 1, 1, x\}a \\ &= \{xa \oplus a, a, a, xa\} \end{aligned} \quad (18.4)$$

zknde64

RV64 instructions accelerate AES by processing eight bytes at a time. The `zknde64` module shown in Fig. 18.35 supports key scheduling instructions `aes64ks1i`, `aes64ks2`, and `aes64im`; encryption instructions `aes64esm` and `aes64es`; and decryption instructions `aes64dsm` and `aes64ds`. Like `zknde32`, it contains `aes64e` and/or `aes64d` hardware for encryption and/or decryption depending on whether `Zkne` and `Zknd` are supported. However, the hardware works on 8 bytes at a time rather than 1 byte, achieving eight times the throughput but requiring 8 `sbox` and `invsbox` modules rather than one of each. Because it works on the entire word, it does not need the logic from Fig. 18.33 to select a byte or to roll the output into place and XOR it with a running total. However, `aes64ks1i` requires 4 S-boxes in `aessboxword`, which are shared with `aes64e` if `Zkne` is enabled.

Fig. 18.36 shows the `aes64e` module. The `aesshifrow64` module simply contains wires to rearrange bits according to Equation 18.2. The `aes64es`/`aes64esm` instructions apply 8 S-boxes in parallel; 4 are shared with `aes64ks1i` in `zknde64`, and the other 4 are in `aessboxword` here. `aesmixcolumns32` computes one column of the output using four $\text{GF}(2^8)$ multiplications according to Equation 18.3. This MixColumns step is skipped on the last round by `aes64es`.

`aes64d` is similar to `aes64e`, but it uses inverse operations. It contains all 8 inverse S-boxes because none are shared with key scheduling. Inverse-MixColumns contains 12 Galois field multipliers because the inverse matrix coefficients from Fig. 18.13 are larger. The block also performs

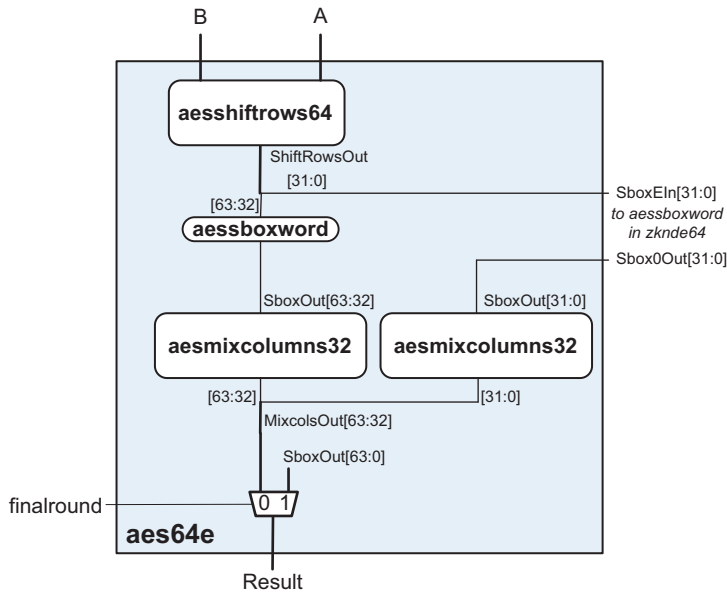


Fig. 18.36 aes64e encrypt module.

aes64im by skipping the first steps and only applying InverseMixColumns to two 32-bit words.

aes64ks1i looks up a round constant based on the round. It rotates 32 bits of the incoming key, sends them back to aessboxword to apply an S-box to each byte, and XORs the results with the round constant. aes64ks2 performs two 32-bit XORs. Both modules are relatively small.

18.4.2.4 Zknh

Fig. 18.37 shows a block diagram of the RV64 zknh64 module, which provides separate modules performing the various SHA-2 σ and Σ instructions for each of SHA-256 and SHA-512. The RV32 zknh32 module is similar but sign-extends SHA-256 outputs to 64 bits and has a different sha512_32 module to operate on half of a 64-bit word at a time.

Fig. 18.38 shows the sha512_64 module. It supports four instructions, each of which need three 64-bit rotations or shifts. Thus, it performs all 12 rotations and shifts, chooses the three needed for the given instruction, and XORs them together. sha256 is similar, but it performs different 32-bit shifts and rotations. sha512_32 is also similar but assembles the upper or lower 32 bits of a 64-bit result from shifts operating on two 32-bit registers.

18.5 SUMMARY

In summary, the Zb^* bit manipulation extensions are a family of four optional extensions that provide modest speedups for certain common code sequences:

- ▶ **Zba:** Shifts and adds to facilitate address generation.
- ▶ **Zbb:** Basic bit operations: min/max, rotate, zero/sign extension, count, combine, reverse.
- ▶ **Zbc:** Carry-less multiplication for cryptography and error checking codes.
- ▶ **Zbs:** Single-bit set/clear/invert/extract.
- ▶ **B:** $Zba + Zbb + Zbs$

Cryptographic extensions are added for Zb^* and Zk^* that provide instructions to accelerate cryptography:

- ▶ **Zbkb:** Bit manipulations generally useful for cryptographic algorithms.
- ▶ **Zbkc:** Subset of **Zbc** needed for cryptography.
- ▶ **Zbkx:** Permutations useful for implementing S-boxes in data-independent time.
- ▶ **Zknd:** Instructions for accelerating the decryption and key schedule functions of the AES block cipher.
- ▶ **Zkne:** Instructions for accelerating the encryption and key schedule functions of the AES block cipher.
- ▶ **Zknh:** Instructions for accelerating the σ and Σ functions for the SHA-2 family of hash functions.
- ▶ **Zkn:** NIST cryptography:
 $Zbkb + Zbkc + Zbkx + Zkne + Zknd + Zknh$
- ▶ **Zkt:** Common data processing instructions must have data-independent latency.
- ▶ **Zkr:** Seed CSR to provide entropy for a random number generator.
- ▶ **Zk:** $Zkn + Zkt + Zkr$

Wally modifies the integer IEU to support the Zb^* and **Zkn** extensions but lacks **Zkr**. The additional hardware is modest except that **Zbc** implies a large array of $XLN^2/2$ AND and XOR gates for the carry-less multiplier array, and **RV64Zknd/Zkne** implies sixteen 256×8 -bit lookup tables for AES S-boxes.

Other Extensions

RISC-V is evolving rapidly. Table A.2 lists various extensions that had been ratified by the time of this writing, the profiles that use them, and the chapters that describe them. Most of the extensions are described in earlier chapters with their required hardware. This chapter describes extensions (particularly those needed for the RVA22S64 profile) that do not naturally fit elsewhere in the book:

- ▶ **Zicbom, Zicboz, Zicbop:** Cache management operations (CMOs)
- ▶ **Zicclsm:** Misaligned loads and stores
- ▶ **Svinval:** TLB management
- ▶ **Zihintpause:** Pause hint
- ▶ **Zicond:** Conditional instructions

Some extensions define *hints*. Hints are encoded as ordinary instructions that have no effect on architectural state but that suggest the core change nonarchitectural state to improve performance or power. For example, the `prefetch.r` instruction suggests that the cache prefetch a cache line that the program anticipates will be needed soon.

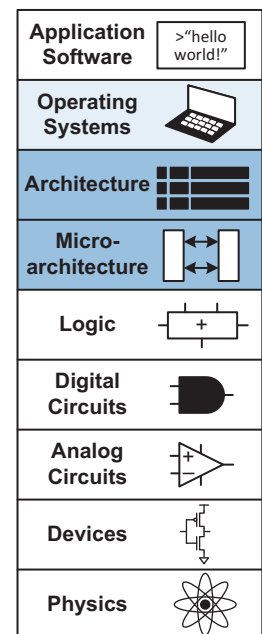
19.1 CACHE MANAGEMENT: Zicbom, Zicboz, Zicbop

The cache block management instructions give a program fine-grained control over cache lines to improve performance and maintain coherence with certain I/O devices. These extensions provide instructions to zero, clean, flush, invalidate, and prefetch cache lines.

19.1.1 RISC-V CMO Instructions

The cache management operation (CMO) extensions are defined in Table I.21 at the back of the book. They are divided into three independent extensions (Zicbom, Zicboz, and Zicbop), but most cores implement all or none.

- 19.1 **Cache Management: Zicbom, Zicboz, Zicbop**
- 19.2 **Misaligned Loads and Stores: Zicclsm**
- 19.3 **TLB Management: Svinval**
- 19.4 **Pause Hint: Zihintpause**
- 19.5 **Conditional Instructions: Zicond**
- 19.6 **Custom Extensions**
- 19.7 **Summary**



`cbo.clean`, `cbo.inval`, `cbo.flush`, and `cbo.zero` are encoded as I-type instructions except that the immediate field is used to distinguish the various operations, and is not added to the base address in `rs1`. Prefetch hints are encoded as `ori x0, rs1, imm`, which is a `nop` because writes to `x0` do nothing.

19.1.1.1 Zicbom Cache Block Management

The Zicbom *cache block management* instructions, `cbo.clean`, `cbo.inval`, and `cbo.flush`, are used to maintain coherence between the data cache and main memory when an *incoherent agent* (e.g., a peripheral with direct memory access) might write data to the main memory without updating the cache. The device driver may clean the cache to write back dirty lines to main memory before the incoherent agent reads this memory and may invalidate stale cache lines after the incoherent agent writes to memory.

These instructions all operate on a virtual address specified by `rs1`. `cbo.clean` writes the contents of a dirty cache block (line) to main memory. `cbo.inval` invalidates a cache block. `cbo.flush` performs a clean followed by invalidate as one atomic operation.

19.1.1.2 Zicboz Cache Block Zero

The Zicboz *cache block zero* instruction `cbo.zero` writes zeros to an entire cache block (line) or to the underlying physical memory if the addresses are not cacheable. Programs sometimes need to initialize regions of memory to zero; for example, the operating system (OS) clears the `.bss` section of memory when starting a process. Using `cbo.zero` to clear one cache line at a time is faster than storing zeros one word at a time and also avoids the need to fetch a cache line from main memory that is about to be zeroed anyway.

19.1.1.3 Zicbop Cache Block Prefetch

When a program knows it is likely to access a memory address soon, it can prefetch the corresponding cache line to hide the latency of waiting for the cache line. Zicbop defines three prefetch hint instructions (`prefetch.i/r/w`), which suggest that the processor prefetch instructions, read data, or write data, if beneficial. Prefetching for data reads and writes is often identical. The hints are encoded as `ori x0, rs1, imm`, which is technically a `nop` because `x0` is hardwired to 0. However, the hint indicates that the line containing a certain address should be prefetched into the cache. The bottom 5 bits of `imm` are treated as 0s, so the virtual address is computed as `Address = [rs1] + {imm[11:5], 00000}`.

19.1.1.4 Environment Configuration Registers

As described in [Section 8.2.3.6](#), the `menvcfg` and `senvcfg` registers enable the use of CMO instructions outside of M-mode. Each of these registers has `CBZE`, `CBCFE`, and `CBIE[1:0]` enable fields. `menvcfg`'s enable must be set to use a CMO instruction in S- or U-mode. If S-mode is supported, `senvcfg`'s enable must also be set to use the instruction in U-mode.

When `CBZE = 0` and the hart is in a lower privilege mode, the `cbo.zero` instruction causes an illegal-instruction fault. Similarly, when `CBCFE = 0` and

the hart is in a lower privilege mode, `cbo.clean` and `cbo.flush` are illegal. `CBIE = 00` disables `cbo.inval` and `CBIE = 11` enables it. `CBIE = 01` causes `cbo.inval` to operate as a flush, which invalidates the line but also ensures dirty data is first written back to the memory hierarchy, making the instruction safer to use.

19.1.2 Test Plan

The CBO instructions are tested with `tests/wally-riscv-arch-test/riscv-test-suite/rv{32/64}i_m/privilege/src/WALY-cbo{m/z}-01.S`. These tests assume a 64-byte cache line.

The `WALLY-cbom-01` test copies a reference region of memory to another region aligned to the size of a cache way. Then it invalidates, cleans, or flushes the copied region and checks main memory for the correct data.

The `WALLY-cboz-01` test copies a region of memory with known data, then overwrites it with `cbo.zero`, and then verifies that the region contains zeros. `riscv-arch-test/riscv-test-suite/rv{32/64}i_m/CM0/src/cbo.zero-01.S` also recently added `cbo.zero` tests.

19.1.3 Wally Implementation

The Wally `ZICBOM_SUPPORTED` and `ZICBOZ_SUPPORTED` configuration parameters indicate whether Wally should add hardware support for CBO instructions. The CBO extension modifies the `controller`, `privileged`, `lsu`, `pmachecker`, and `cache` units. The IEU controller detects CBO instructions, and gates them with the appropriate environment configuration registers, `envcfg`. It sends `CBOpM[3:0]`, which specifies `{zero, clean, flush, inval}`, to the LSU. The MMU then checks for exceptions, and the LSU sends the request to either the cache or the AHB bus.

19.1.3.1 CMO LSU Modifications

[Fig. 19.1](#) updates the LSU with the required CMO changes shown in blue. CBO instructions add additional access types to the PMA checker, mux zeros into `WriteDataM` for `cbo.zero`, and update the cache and AHB cache interface.

This section describes the cache modifications for the `cbo.*` instructions. `cbo.zero` operates on both cacheable and uncachable memory, so it is more efficient to zero `IHWriteDataM` in the LSU rather than in both the cache and the AHB cache interface. The LSU uses the MMU's `CacheableM` attribute to select if the cache directly zeros the data or if the operation should be transmitted to the bus as an AHB burst request, as described in [Section 19.1.3.3](#).

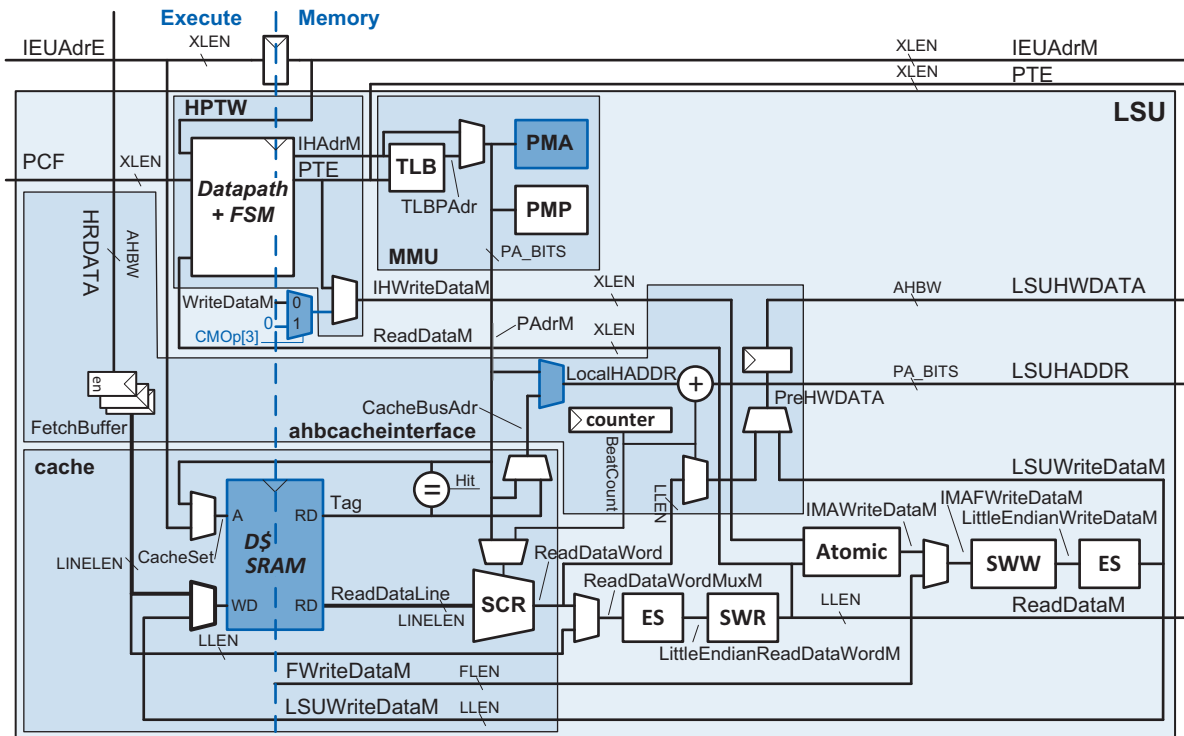


Fig. 19.1 LSU with CMO support additions in darker blue.

As with a regular load or store, during a `cbo.*` operation, the LSU receives the virtual address for the CMO instruction. If virtual memory is enabled, the TLB translates it to a physical address, throws a store page fault if the page is not accessible, and sets the access/dirty bits if necessary. The PMA and PMP check that the physical address is accessible and throw a store access fault if not. The PMA adds CMO to the types of valid accesses for the RAMs, as shown in Fig. 19.2.

19.1.3.2 CMO Cache Modifications

Fig. 19.3 shows the top-level cache with the changes highlighted in blue. `CMOp` is sent to the cache FSM, which controls the various operations. The FSM produces a new `ClearValid` output to clear the valid bit for CMO invalidate and flush operations. For `cbo.zero`, the cache always selects `LineWriteData = 0`.

The cacheway tag and valid logic determines if this address matches a line in the way. `cbo.inval` clears the valid bit for the matching line and updates the LRU (least recently used) bits, such that the invalidated way

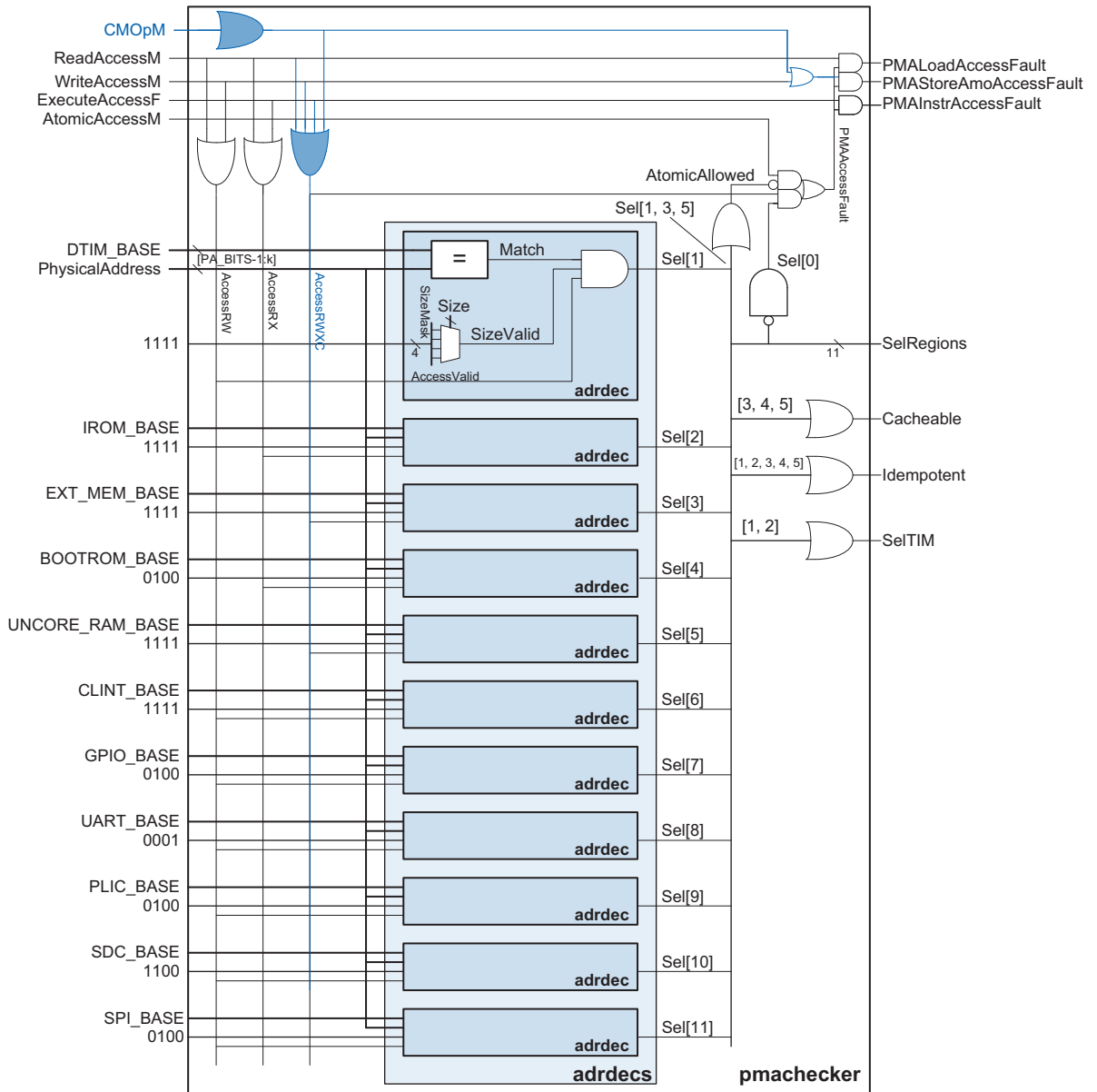


Fig. 19.2 PMA checker with CMO support additions in darker blue.

becomes the next victim. `cbo.clean` writes the matching line back to main memory if it is dirty, and clears the dirty bit. `cbo.flush` cleans and invalidates the line. `cbo.zero` writes all zeros to the matching line and marks it dirty and valid.



Fig. 19.4 shows the cache FSM updates, highlighted in blue. Table 19.1 shows the FSM inputs and outputs that have been modified from Table 10.2.

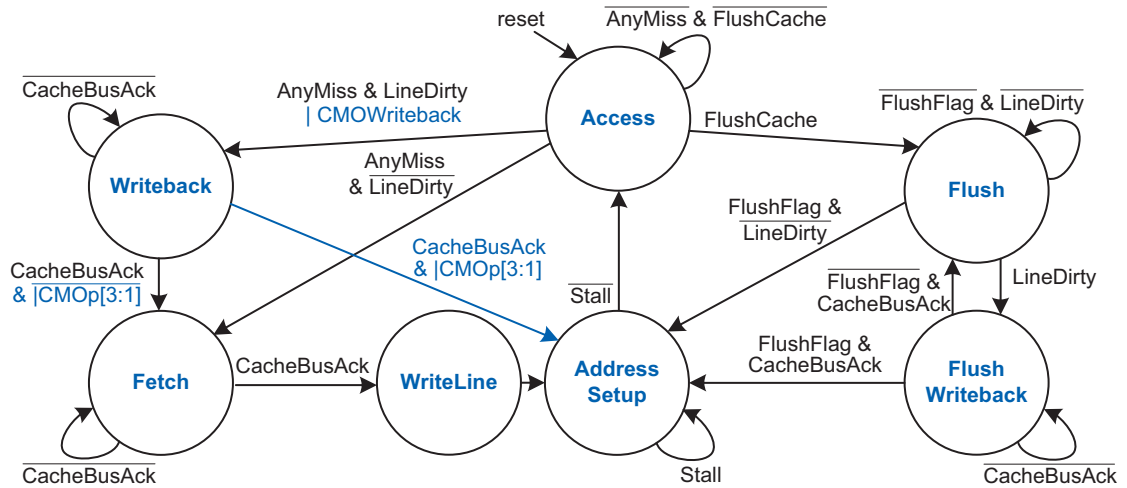


Fig. 19.4 cacheFSM with CMO support additions in blue.

Table 19.1 cacheFSM inputs and outputs with CMO support additions in blue.

FSM Input	Criteria to Assert
CMOp[3:0]	cmo.{zero/clean/flush/invl}
HitLineDirty	HitDirtyWay
FSM Internal	Criteria to Assert
CMOZeroNoEviction	CMOp[3] & ~LineDirty
CMOWriteback	((CMOp[1] CMOp[2]) & Hit & HitLineDirty) (CMOp[3] & LineDirty)
FSM Output	Criteria to Assert
SetValid	WriteLine Access & CMOZeroNoEviction Writeback & CacheBusAck & CMOp[3]
ClearValid	Access & (CMOp[0] CMOp[2] & ~HitLineDirty) Writeback & CMOp[2] & CacheBusAck
SetDirty	(Access & (AnyUpdateHit CMOZeroNoEviction)) (WriteLine & CacheRW[0]) (WriteBack & CMOp[3] & CacheBusAck)
ClearDirty	(WriteLine & ~CacheRW[0] (Flush & LineDirty)) (Writeback & ((CMOp[1] CMOp[2]) & CacheBusAck))
SelVictim	Access & (AnyMiss & LineDirty ~Hit & CMOZeroNoEviction) WriteLine Writeback & ((~(CMOp[1] CMOp[2]) & ~CacheBusAck) (CMOp[3] & CacheBusAck))
LRUWriteEn	Access & (AnyHit CMOZeroNoEviction) WriteLine Writeback & CMOp[3] & CacheBusAck
SelAdrTag	Access & (AnyMiss (CMOp)) Fetch Writeback WriteLine
SelAdrData	Access & (AnyMiss (CMOp) CacheRW[0]) Fetch Writeback WriteLine
CacheBusRW[1:0]	[1]: Access & AnyMiss & ~LineDirty Writeback & CacheBusAck & ~ CMOp Fetch & ~CacheBusAck [0]: Access & LoadMiss & LineDirty (Writeback FlushWriteback) & ~CacheBusAck

An alternate implementation for `cbo.*` instructions could transition from Writeback to Access but would require lowering `CacheStall` during the transition, thus increasing the critical path. By first transitioning to AddressSetup, `CacheStall` only depends on the current state.

`cbo.inval` remains in the Access state. It asserts `ClearValid` to invalidate the line if there is a hit.

`cbo.clean` and `cbo.flush` both transition to the Writeback state if the line hits and is dirty. Once in Writeback, the cache FSM waits for `CacheBusAck` as it did for bus operations in [Section 10.4.3](#). However, for `cbo.*` instructions, it continues directly to AddressSetup rather than Fetch because it does not need to fetch a missing line. In the last cycle of Writeback, the cache FSM also asserts `ClearValid` for `cbo.flush` to invalidate the line now that it has been cleaned. During Writeback, `CacheBusRW[0]` is asserted to initiate writing the dirty cache line back to main memory via the AHB bus. Both `cbo.clean` and `cbo.flush` clear the dirty bit in the last cycle of Writeback.

`cbo.zero` writes zeros to a cache line. If `cbo.zero` hits the cache or misses but the victim line is clean, it writes zeros to the entire line and sets the valid and dirty bits; it also updates the tag on a miss. This takes a single cycle from the Access state. If `cbo.zero` misses the cache and the victim line is dirty, it transitions to the Writeback state to evict the dirty victim. It writes zeros to the line, sets valid and dirty, and updates the tag on the last cycle of Writeback. Like the other `cbo.*` instructions, it transitions to the AddressSetup state to finish.

[Fig. 19.5](#) shows how `cacheway` writes a 0 to the valid bit of the selected cache line when `ClearValid` = 1. Similarly, but not diagrammed, `cacheLRU` updates the `LRUMemory` for that line when `ClearValid` is asserted.

`SelVictim` controls whether `cacheway` outputs the cache line corresponding to a hit or ignores the hit logic to select the `VictimWay` cache line for writeback or `FlushWay` cache line for an entire cache flush resulting from `fence.i`. `SelVictim` cannot control `DirtyWay` because the `cachefsm` drives `SelVictim` based on the value of `DirtyWay`. Unlike other cache operations, `cbo.clean` and `cbo.flush` need to know if the hit cache line is dirty, indicated by the new `HitDirtyWay` output.

19.1.3.3 Uncached `cbo.zero`

The MMU can mark certain main memory pages as uncacheable using page-based memory types (PBMT), as described in [Section 11.2.2.1.1](#). `cbo.zero` must still write a cache line worth of zeros to such memory, but it must directly do so through the AHB bus. The `BusCMOZero` = `CMOp[3] & ~Cacheable` signal indicates this situation.

The LSU leverages the `ahbcacheinterface`'s existing `CacheWriteback` state (see [Fig. 12.5](#)) to iterate over the beats of a burst request with only two changes to the hardware. The transition from `Adr` to `CacheWriteback` becomes `HREADY & (CacheRW[0] | BusCMOZero)`. The `CacheBusAck` output is not asserted when `BusCMOZero` is high because the `cachefsm` did not make this memory request and should not receive this acknowledgment.

19.1.3.4 Prefetch

Wally is an in-order processor that stalls during cache fills. Therefore, prefetch would stall the processor and not improve performance. Hence, the `prefetch.*` hints are ignored.

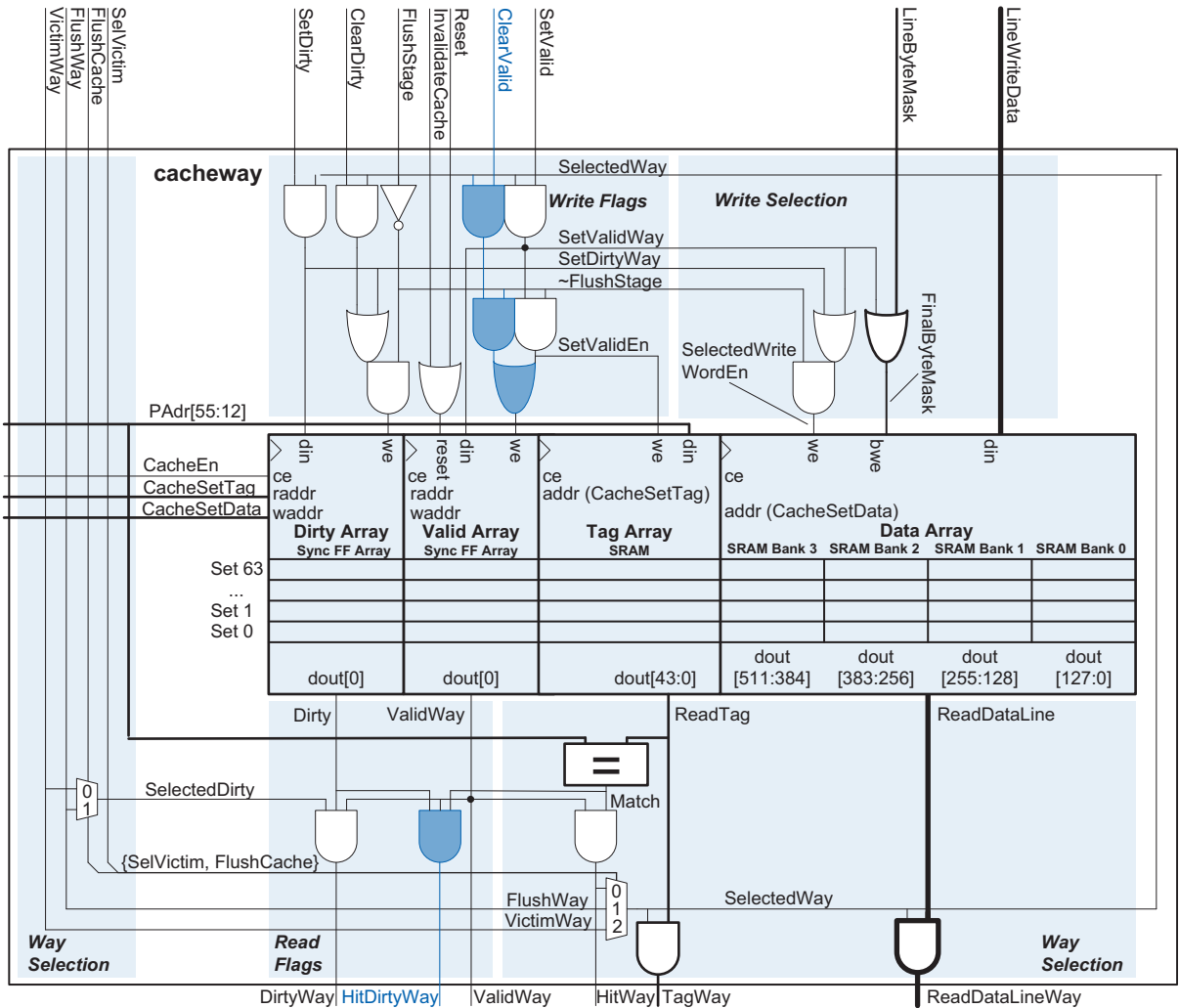


Fig. 19.5 cacheway with CMO support additions in darker blue.

19.2 MISALIGNED LOADS AND STORES: Zicclsm

By default, RISC-V produces a misaligned-load or misaligned-store exception when the address is not a multiple of the size of the element being accessed. For example, `lw t0, 6(x0)` produces a misaligned-load exception because address 6 is not a multiple of the 4-byte word size. The trap handler may emulate the misaligned access through a sequence of smaller aligned accesses, such as:

```

lh t0, 6(x0)           # Load lower half word
lh t1, 8(x0)           # Load upper half word
slli t1, t1, 16        # Move upper half word into upper bytes
or t0, t0, t1          # Combine lower and upper half words

```

RISC-V allows misaligned accesses to be handled by hardware rather than trapping. The *Zicclsm* (*load/store misaligned*) extension indicates that a hart handles misaligned loads and stores to cacheable coherent memory in hardware. They may be handled slowly, but they will not trap. Even with *Zicclsm*, misaligned atomic memory operations (AMOs) might still trap. The RVA20 and later profiles require support for *Zicclsm*. This is valuable for porting legacy code, which may contain misaligned structures, and which becomes unacceptably slow if each misaligned access traps.

19.2.1 Test Plan

No load or store can produce `{Load/Store}MisalignedFaultM` when *Zicclsm* is enabled, so the misaligned `riscv-arch-tests` will not function correctly. To test both *Zicclsm* and misaligned faults, Wally `rv64gc` enables *Zicclsm*, but `rv32gc` does not. Misaligned access hardware handling is tested with `tests/wally-riscv-arch-test/riscv-test-suite/rv64i_m/privilege/src/WALLY-misaligned-access-01.S`, which sequences over all possible combinations of byte offsets, access sizes, load and store operations, and enough memory accesses to cross 64-byte cache line boundaries.

19.2.2 Wally Implementation

Wally adds support for misaligned loads and stores to cacheable memory when `ZICCLSM_SUPPORTED = 1`. [Fig. 19.6](#) shows the modified LSU, with the new `align` block highlighted in blue. The `align` block affects three parts of the LSU: reads, writes, and cache line spills. The implementation prioritizes fast misaligned access while minimizing changes to the rest of the LSU. It doubles the width of the cache read and write buses and adds the `align` module to shift misaligned data into the correct position. If the address spills across two cache lines, it issues requests to both lines and merges the data for loads. Misaligned atomic operations still throw a `store/AMO-misaligned` fault.

The data cache from [Section 10.4](#) reads and writes a native word size `WORDLEN = LLEN = max(FLEN, XLEN)`. To support *Zicclsm*, Wally extends `WORDLEN` to `LLEN*2`, which allows the cache to output the entire access in one request for misaligned addresses within a single cache line.

[Fig. 19.7](#) shows the details of the `align` module, with the read, write, and spill logic all together. It computes `ByteOffsetM` as `IEUAdrM` modulo the access size. If this offset is nonzero for a read or write, it asserts `MisalignedM` and shifts read or write data by `ByteOffsetM` bytes to align the data.

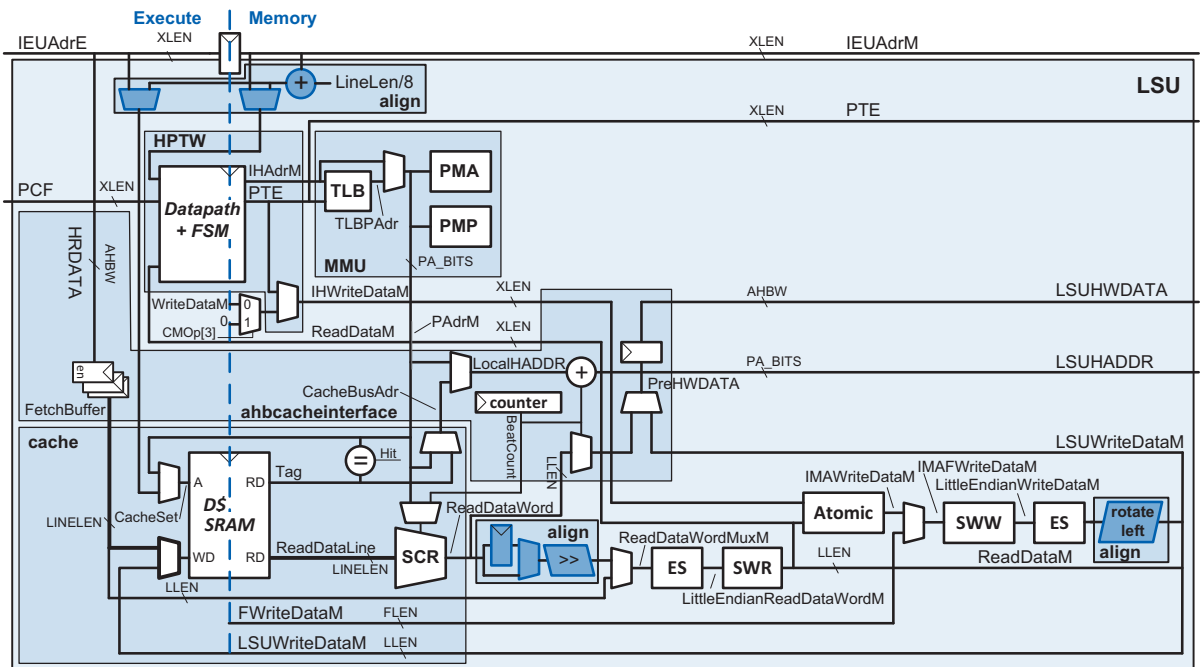


Fig. 19.6 LSU with Zicclsm support.

A spill (`ValidSpillM`) occurs when a misaligned address (`MisalignedM`) is in the last native-sized access of a cache line (`PotentialSpillM`). During a load spill, the align FSM asserts `SpillSaveM` to capture the `DCacheDataWordM` into register `ReadDataWordFirstHalfM`. `SelfSpillE` is also asserted to increment `IEUAdrSpillE` by the size of the cache line in bytes. If the request is a store, the data cache writes the lower part of the word with the lower bits of `LSUWriteDataM`.

Fig. 19.8 shows a simplified diagram of the align FSM. See the code for full details. During a load spill, the FSM transitions to `Spill` and asserts `SelfSpillM`, which increments `IEUAdrSpillM` by the size of the cache line in bytes. The MMU translates the address if address translation is enabled to the physical address `PAAdrM`, and the cache outputs the second half of the load. `SelfSpillM` also selects the `DCacheReadDataWordM` concatenated with `ReadDataWordFirstHalfM` which is then shifted by `ShiftAmount`.

19.3 TLB MANAGEMENT: Svinval

The `Svinval` TLB management extension defines `sinal.vma`, `sfence.w.inval`, and `sfence.inval.ir` instructions. They are closely

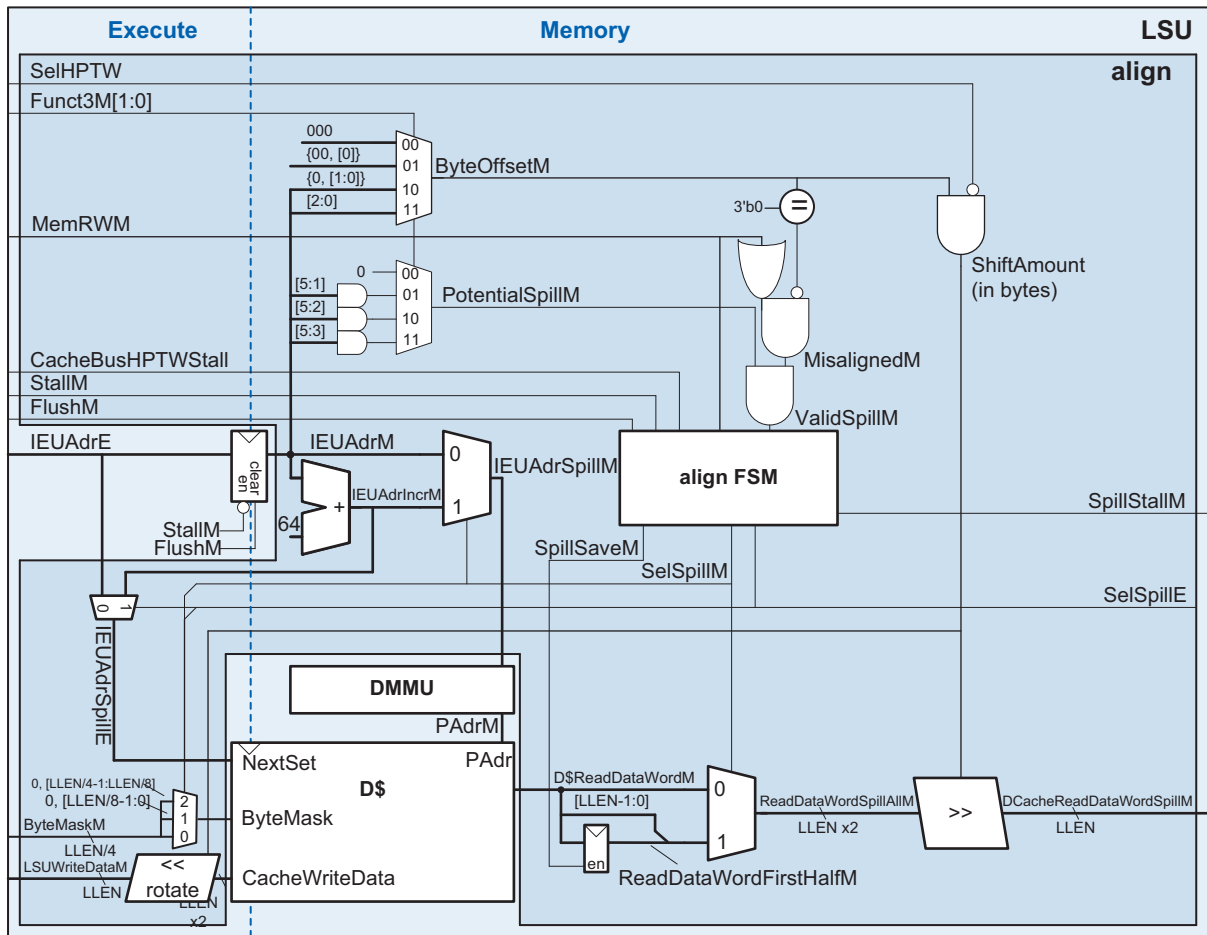


Fig. 19.7 align module (assumes 64-byte cache line).

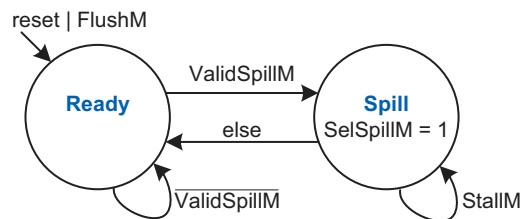


Fig. 19.8 align FSM.

related to `sfence.vma` but relax the ordering relative to other memory instructions, potentially improving performance on out-of-order or multi-core systems. Their encoding matches `sfence.vma` except in bits 29:27.

Wally is an in-order single-core processor with inherent total store order (TSO) behavior, so these fine-grained TLB management instructions are unnecessary. Thus, the `Svinval` instructions are conservatively implemented as `sfence.vma` when `SVINVAL_SUPPORTED = 1`.

19.4 PAUSE HINT: Zihintpause

Section 17.1.4.1 described spin locks, which can be used to wait until a resource is available. The processor wastes power while spinning and, in a multithreaded system, it wastes execution resources potentially available to another thread. The Zihintpause extension introduces a `pause` instruction, which is a hint suggesting that the hart temporarily reduce the instruction issue rate while spinning. A spin lock loop should contain exactly one `pause` instruction. Code Example 19.1 illustrates a spin lock, based on Code Example 17.7, that waits until the lock is free (0), then takes the lock. If not immediately successful, it uses a `pause` hint before trying again.

`pause` is a hint encoded as a `fence w` (machine code: 0x0100000F), which does nothing.

Code Example 19.1 SPIN LOCK WITH PAUSE

```
spinlock:                                # Address of lock is in a0
    lr.w t0, (a0)                        # Read the lock
    bnez t0, retry                       # Spin until free
    li t1, 1
    sc.w t0, t1, (a0)                   # Try to write a 1 to take lock
    bnez t0, retry                       # Spin until successful
    ret                                  # Got the lock!
retry:                                   # No lock yet
    pause                                # Pause hint to reduce spin power
    j spinlock                           # Try again
```

Wally presently has no power management features to control instruction issue rate. It accepts the `pause` hint but ignores it.

19.5 CONDITIONAL INSTRUCTIONS: Zicond

Branches are expensive. When they are mispredicted, they flush the pipeline and reduce performance. Even when branches are predicted correctly, they take up space in the branch predictor memories. They also impact extracting instruction-level parallelism for superscalar and out-of-order microarchitectures.

Many architectures define *conditional* instructions to avoid branches by performing an operation only if a condition is true. For example, MIPS, Alpha, PowerPC, SPARC, x86, and ARM all have conditional move instructions to copy one register to another depending on the state

of a third register or condition code. ARM and IA-64 provide more extensive *predication* in which any instruction can be conditionally executed based on condition codes. However, RISC-V lacks space in the instruction encodings to specify a third source register or condition code for full predication. Furthermore, conditional move instructions are difficult to implement in microarchitectures that use register renaming because they require reading three registers: the condition, the value to move, and the old value of the destination register that must be written to the renamed register if the condition is not satisfied.

RISC-V introduces a Zicond extension for conditional operations. This extension defines `czero.eqz` and `czero.nez`, listed in Table I.25 (on the last pages of the book). `czero.eqz rd, rs1, rs2` sets `rd = 0` if `rs2 = 0`, or `rd = rs1` otherwise. `czero.nez` is similar but checks if `rs2 ≠ 0`.

Conditional zero instructions, `czero.*`, are easy to implement, even on a microarchitecture with register renaming and two register read ports per instruction issued. A programmer can use these instructions to write short sequences of code that operate conditionally. Table 19.2 lists some sequences. `cond` is `eq` or `ne` (equal or not equal zero). A conditional operation `add.eq rd, rs1, rs2, rc` computes `rd = (rc == 0) ? rs1 + rs2 : rs1`. In other words, it conditionally makes `rd` equal to `rs1 + rs2` if `rc` equals 0, and just `rs1` otherwise. `add.ne` adds if `rc` does not equal 0. The conditional

Table 19.2 Conditional operation sequences.

Conditional Operation	Cond: <code>eq</code>	Cond: <code>ne</code>
<code>add.cond rd, rs1, rs2, rc</code> # <code>rd = (rc cond 0) ? rs1+rs2 : rs1</code>	<code>czero.nez rd, rs2, rc</code> <code>add rd, rs1, rd</code>	<code>czero.eqz rd, rs2, rc</code> <code>add rd, rs1, rd</code>
<code>sub.cond rd, rs1, rs2, rc</code> # <code>rd = (rc cond 0) ? rs1-rs2 : rs1</code>	<code>czero.nez rd, rs2, rc</code> <code>sub rd, rs1, rd</code>	<code>czero.eqz rd, rs2, rc</code> <code>sub rd, rs1, rd</code>
<code>or.cond rd, rs1, rs2, rc</code> # <code>rd = (rc cond 0) ? rs1 rs2 : rs1</code>	<code>czero.nez rd, rs2, rc</code> <code>or rd, rs1, rd</code>	<code>czero.eqz rd, rs2, rc</code> <code>or rd, rs1, rd</code>
<code>xor.cond rd, rs1, rs2, rc</code> # <code>rd = (rc cond 0) ? rs1rs2 : rs1</code>	<code>czero.nez rd, rs2, rc</code> <code>xor rd, rs1, rd</code>	<code>czero.eqz rd, rs2, rc</code> <code>xor rd, rs1, rd</code>
<code>and.cond rd, rs1, rs2, rc</code> # <code>rd = (rc cond 0) ? rs1&rs2 : rs1</code>	<code>and rd, rs1, rs2</code> <code>czero.eqz rtmp, rs1, rc</code> <code>or rd, rd, rtmp</code>	<code>and rd, rs1, rs2</code> <code>czero.nez rtmp, rs1, rc</code> <code>or rd, rd, rtmp</code>
<code>sel.cond rd, rs1, rs2, rc</code> # <code>rd = (rc cond 0) ? rs1 : rs2</code>	<code>czero.nez rd, rs1, rc</code> <code>czero.eqz rtmp, rs2, rc</code> <code>or rd, rd, rtmp</code>	<code>czero.eqz rd, rs1, rc</code> <code>czero.nez rtmp, rs2, rc</code> <code>or rd, rd, rtmp</code>
<code>mov.cond rd, rs1, rc</code> # <code>rd = (rc cond 0) ? rs1 : rd</code>	<code>czero.nez rd, rd, rc</code> <code>czero.eqz rtmp, rs1, rc</code> <code>or rd, rd, rtmp</code>	<code>czero.eqz rd, rd, rc</code> <code>czero.nez rtmp, rs1, rc</code> <code>or rd, rd, rtmp</code>

Table 19.3 Code examples with branches, conditional operations, and conditional move.

Code	Branches	Conditional Zero	Conditional Move
if (cond) x = 0;	beq s0, zero, skip mv s1, zero skip:	czero.nez s1, s1, s0	cmov.nez s1, zero, s0
if (cond) x = 1;	beq s0, zero, skip li s1, 1 skip:	li t0, 1 czero.nez t0, t0, s0 czero.eqz s1, s1, s0 or s1, s1, t0	li t0, 1 cmov.nez s1, t1, s0
if (cond == 0) x = x + y;	bne s0, zero, skip add s1, s1, s2 skip:	czero.eqz t0, s2, s0 add s1, t0, s1	add t0, s1, s2 cmov.eqz s1, t0, s0
if (y >= 7) x = 2;	li t0, 7 blt t1, t0, skip li s1, 2 skip:	li t0, 7 slt t0, s2, t0 czero.eqz s1, s1, t0 li t1, 2 czero.nez t1, t1, t0 or s1, s1, t1	li t0, 7 slt t0, s2, t0 li t1, 2 cmov.eqz s0, t1, t0

operations in the left column of [Table 19.2](#) are purely explanatory; they are not pseudoinstructions supported by RISC-V assemblers and must be replaced by the actual sequence of instructions in the columns on the right. Note that some of the conditional operations involve a three-instruction sequence and a temporary register.

[Table 19.3](#) compares how some common code sequences can be written with branches, conditional zero, or conditional moves. Assume `cond`, `x`, and `y` are in `s0`, `s1`, and `s2`, respectively. Conditional zero can sometimes replace a branch without increasing program length, but other times they take more instructions than a branch solution. Conditional moves (`cmov`) are more powerful than `czero.*` but are not part of the RISC-V instruction set because of the implementation difficulty.

19.5.1 Test Plan

The Zicond-extension instructions are tested with the `riscv-arch-test` suite. [Table 19.4](#) lists the number of tests for each instruction.

Table 19.4 `riscv-arch-test` for the Zicond extension.

Instruction	RV32Zicond Tests	RV64Zicond Tests
<code>czero.eqz</code>	613	741
<code>czero.nez</code>	613	741
Total	1,226	1,482

19.5.2 Wally Implementation

The `czero.*` instructions are executed on the ALU using the existing AND gate. The AND gate's second source is set to all 1s or all 0s to either select the first source `rs1` or 0, respectively, based on `rs2`. Fig. 19.9 shows how the ALU from Fig. 18.21 is enhanced with Zicnd support, highlighted in darker blue. A mux produces the `ZeroCondMaskInvB` of all 0 or all 1 for `czero` instructions based on `rs2 == 0` (`BZero`). The controller is also modified to drive `ALUSelect = 111` to choose the AND output when it decodes `czero` instructions.

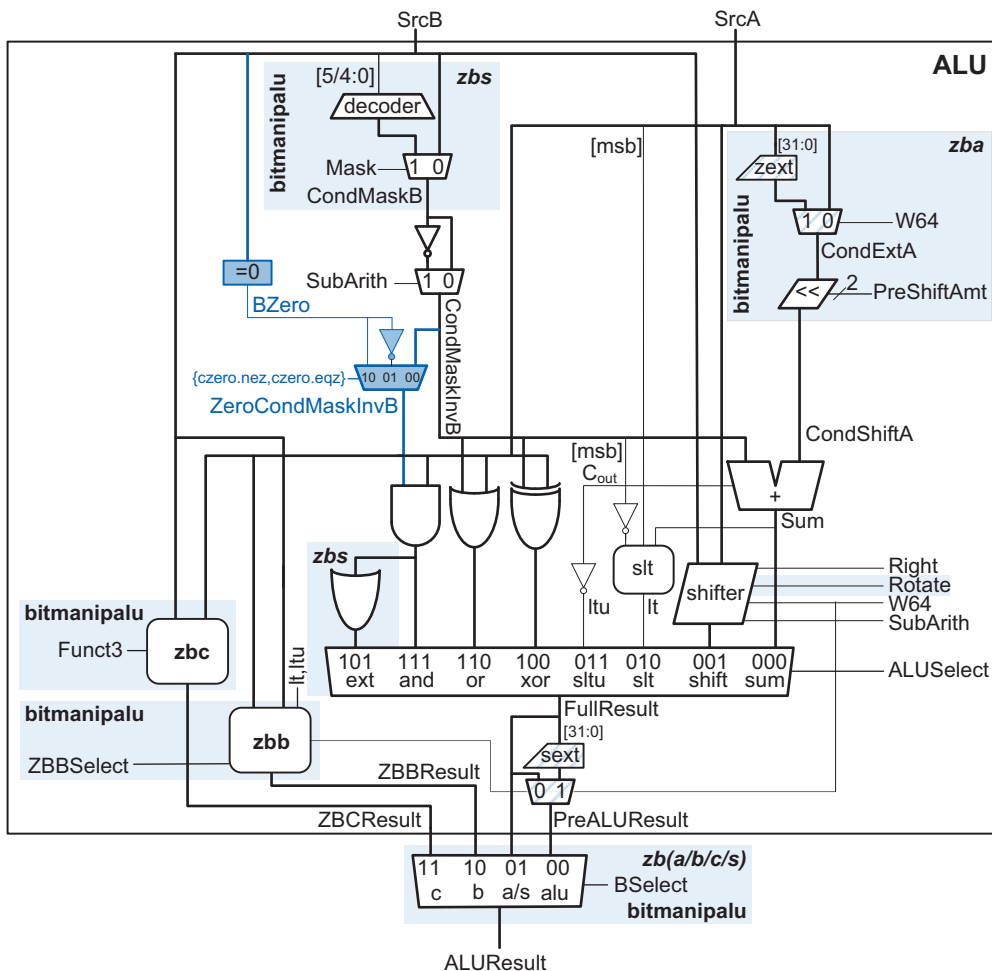


Fig. 19.9 ALU with Zicnd support (crypto support not shown).

19.6 CUSTOM EXTENSIONS

RISC-V facilitates custom extensions for specific applications. The general process of creating a custom extension is to start with a precisely written specification, critically reviewed by an experienced team. In a test-driven design methodology, create a test plan and a set of tests to fully exercise the custom extension. Then update the HDL implementation. If the extension adds new instructions, the controller must decode the instructions and send appropriate commands to functional units. If the extension adds new CSRs, the privileged unit must add support. A custom extension usually requires a simulator and patches to the compiler and possibly the OS to run the extension.

Some companies create proprietary extensions to differentiate their products in the market. However, when a custom extension proves useful, there are many incentives to publish and standardize it:

- ▶ More software and hardware developers will use the extension.
- ▶ The extension become accessible on the main branch of the compiler, simulator, and OS.
- ▶ The ecosystem support burden is shared among many developers.
- ▶ The extension will not conflict with subsequent standard extension encodings and CSRs.
- ▶ The inventor gains credibility in the open-source RISC-V community and in the marketplace.

Relatively simple or noncontroversial extensions are standardized through a RISC-V International fast track process:

- ▶ A proposer documents the extension's name, motivation, use cases, and high-level description.
- ▶ The proposer requests that the RISC-V Architecture Review Committee open a fast-track review.
- ▶ Appropriate tech chairs review the plan milestones.
- ▶ The proposer provides a draft specification for internal review by an appropriate RISC-V committee.
- ▶ The proposer provides a final draft specification and checklist of deliverables.
- ▶ The final draft specification goes for a 2-week public review period.
- ▶ If no objections are raised, the RISC-V Technical Steering Committee votes to approve the extension.

More complex extensions, such as vector, matrix, CHERI, and crypto, are usually developed by a task group over a longer period of time with collaboration from many stakeholders.

19.7 SUMMARY

RISC-V is rapidly ratifying more extensions. This chapter covers extensions required for the RVA22S64 profile that don't fit naturally elsewhere in the book. They include cache management, misaligned loads and stores, TLB management, a pause hint, and conditional instructions.

- 20.1 GPIO
- 20.2 CLINT
- 20.3 PLIC
- 20.4 UART
- 20.5 SPI
- 20.6 Summary

An SoC has a variety of peripherals and memories outside the core. This chapter presents a variety of basic peripherals that are common for embedded systems or necessary to boot Linux. Fully-featured RISC-V systems require peripherals to generate the timer, software, and external interrupts. Embedded systems usually need the ability to control individual I/O pins with general-purpose I/O (GPIO). Many systems need a serial port to print to a console or control serial peripherals. Most processors access peripherals using memory-mapped I/O using loads and stores to memory locations associated with the peripherals.

In RISC-V, the SiFive Core Local Interruptor (CLINT) has become a de facto standard timer and software interrupt controller [FE310]. The Platform-Level Interrupt Controller (PLIC) is the standard RISC-V external interrupt controller that arbitrates among external interrupts and routes them to the appropriate harts [PLIC22]. The SiFive GPIO peripheral [FE310] is a de facto standard for general-purpose I/O. Many Linux system serial ports are compatible with the classic Texas Instruments PC16550D Universal Asynchronous Receiver/Transmitter (UART) [PC16550]. Serial Peripheral Interface (SPI) is a faster synchronous serial port that supports a wide range of peripheral such as SD cards and NOR flash memory. On Wally, the peripherals are in the uncore module and connect to the bus via an AHB-to-APB bridge, as shown in Figs. 9.14 and 9.28. The memory-mapped peripherals are tested by a sequence of reads and writes that exercise their functions.

Wally contains the peripherals described in this chapter. Fig. 2.8 showed the default Wally memory map, including the base addresses of all these peripherals. The RAM and ROM are discussed in Sections 9.2.2.3 and 9.5, and the other peripherals are discussed here.

When testing peripherals, it can be helpful to connect their outputs back to their inputs so a test program can drive known values and read them back. This is known as *loopback testing*.

The uncore uses the following configuration parameters to specify whether a peripheral exists and where it goes in the memory map:

Application Software	>"hello world!"
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

The memory locations associated with memory-mapped peripherals are often referred to as control, status, and data registers.

- ▶ CLINT_SUPPORTED / BASE / RANGE
- ▶ GPIO_SUPPORTED / BASE / RANGE
 - GPIO_LOOPBACK_TEST Connect GPIO outputs back to inputs
- ▶ UART_SUPPORTED / BASE / RANGE
 - UART_PRESCALE Reduce HCLK by $2^{\text{UART_PRESCALE}}$
- ▶ SPI_SUPPORTED / BASE / RANGE
 - SPI_LOOPBACK_TEST Connect SDO back to SDI
- ▶ PLIC_SUPPORTED / BASE / RANGE
 - PLIC_NUM_SRC Number of interrupt sources
 - PLIC_GPIO_ID Interrupt ID for GPIO
 - PLIC_UART_ID Interrupt ID for UART
 - PLIC_SPI_ID Interrupt ID for SPI
 - PLIC_SDC_ID Interrupt ID for SD Card
- ▶ SDC_SUPPORTED / BASE / RANGE

The test plan for each peripheral is described in the given peripheral's section below. The test programs are in `tests/wally-riscv-arch-test`. Each is an assembly language program that performs memory-mapped I/O accesses and checks for expected results. They use the same `WALLY-TEST-LIB-32/64.h` file as in [Chapter 8](#), so that test cases can be expressed as a series of memory accesses. The library logs the results to the signature. Memory-mapped I/O addresses are defined by a peripheral base address and register offset so that a peripheral could be relocated by changing only the base address in `config.vh` and the peripheral's test program. Interrupt testing leaves the global interrupts disabled and just checks the MIP interrupt pending bits; interrupt handlers are tested in [Chapter 8](#).

Some of these peripherals are too complex to describe here in full detail. They are worth reading about if you wish to learn about peripheral design and bus interfacing. See the SystemVerilog code in `src/uncore` for more details.

20.1 GPIO

The general-purpose input/output (GPIO) peripheral controls up to 32 GPIO pins on a chip. It can configure each pin individually to be an input or output and can generate interrupts when a particular pin is 0, 1, rises, or falls.

[Fig. 20.1](#) shows a block diagram of the GPIO peripheral top-level interface. The module connects to the APB bus, PLIC, and GPIO pins on the chip. Each pin driver receives `GPIO PinsIn`, `GPIO PinsOut`, and `GPIO PinsEn` signals to read or write (drive) the GPIO pin and control

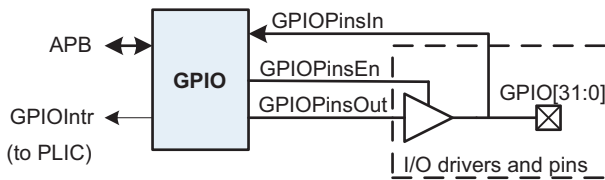


Fig. 20.1 GPIO top-level interface.

whether the pin is an input or output (i.e., disable/enable the tristate output driver). The GPIO can send interrupts to the PLIC when specified events occur.

20.1.1 Memory Map

Table 20.1 lists the memory-mapped GPIO registers and their address offsets. For example, if the GPIO base address is 0x10060000, then a write to 0x1006000C puts a value in the `output_val` register. All GPIO registers are 32 bits, even on RV64. The registers are cleared on reset, making all pins inputs.

Each register has 32 1-bit fields corresponding to the 32 GPIO pins. The `output_en` (output enable) register enables the tristate drivers when a bit is 1 to make the corresponding pin an output. The `output_val` register contains the values to drive onto the output pins. The `input_val` register contains values read from the input pins. The `input_en` register makes the corresponding bits in `input_val` 0 when not enabled. The `out_xor` register conditionally inverts the output value going to the pin.

The inputs are generally asynchronous to the peripheral (APB) clock, so they are synchronized to the peripheral's clock domain with a 2-stage synchronizer (see Fig. 20.4) to prevent metastability. The inputs control the interrupt pending register. When an input is 1, the corresponding high interrupt pending `high_ip` bit is set. Interrupt pending bits are sticky so that the bit will remain 1 even if the input becomes 0. Writing a 1 to an interrupt pending bit clears it. This may seem counterintuitive but allows a programmer to easily clear a particular pending interrupt without disturbing other pending interrupts. Similarly, the `low_ip`, `rise_ip`, and `fall_ip` interrupt pending bits are set by an input being 0, rising, or falling, and will remain set until a 1 is written to that bit of the interrupt pending (`*_ip`) register.

The interrupt enable registers (`high_ie`, `low_ie`, `rise_ie`, `fall_ie`) control whether particular GPIO pins generate an interrupt when the interrupt pending bit is set. If any of the high, low, rise, or fall interrupts are both pending and enabled, the `GPIOIntr` signal raises to inform the interrupt controller.

Some chips have configurable drive strength for output pads and configurable internal pullup resistors to prevent input pads from floating. These features are controlled by the `pue` and `ds` registers.

Table 20.1 GPIO register map.

Offset	Name	Description
0x00	input_val	Pin value
0x04	input_en	Pin input enable
0x08	output_en	Pin output enable
0x0C	output_val	Output value
0x10	pue	Internal pullup enable
0x14	ds	Pin drive strength
0x18	rise_ie	Rise interrupt enable
0x1C	rise_ip	Rise interrupt pending
0x20	fall_ie	Fall interrupt enable
0x24	fall_ip	Fall interrupt pending
0x28	high_ie	High interrupt enable
0x2C	high_ip	High interrupt pending
0x30	low_ie	Low interrupt enable
0x34	low_ip	Low interrupt pending
0x38	iof_en	I/O function enable
0x3C	iof_sel	I/O function select
0x40	out_xor	Output XOR

Some embedded processors with limited numbers of pins multiplex the GPIO pins with other peripherals such as serial ports. Optional `iof_en` and `iof_sel` (I/O function enable and select) GPIO registers control whether the pins are driven by an alternate I/O function and select which function.

Example 20.1 GPIO PROGRAM

Consider the system in Fig. 20.2 in which two pushbutton switches and a light-emitting diode (LED) are connected to GPIO pins. The pulldown resistors set a default value of 0 when the switches are not pressed. The series current-limiting resistor sets the brightness of the LED and prevents overheating and damage to the pin by preventing it from driving too much current. Write a program to turn on the LED when either switch is pressed.

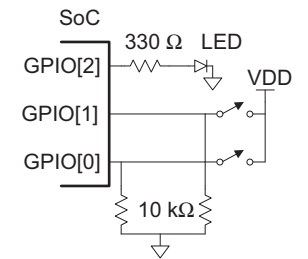


Fig. 20.2 GPIO pins connected to switches and LED.

Solution:

```

        la t0, 0x10060000      # GPIO base address
        li t1, 3               # Enable input pins 0 and 1 for switches
        sw t1, 4(t0)           # Using input_en register
        li t1, 4               # Enable output pin 2 for LED
        sw t1, 8(t0)           # Using output_en register
loop:    lw t2, 0(t0)           # Read input_val (switches, other bits = 0)
        beq t2, zero, led_off  # Are both switches 0?
        sw t1, 0xC(t0)        # No: turn on LED
        j loop                 # Repeat
led_off: sw t0, 0xC(t0)        # Yes: turn off LED
        j loop                 # Repeat

```

The same program is written in C below:

```

volatile uint32_t *GPIO_INPUT_VAL = (uint32_t*)0x10060000;
volatile uint32_t *GPIO_INPUT_EN  = (uint32_t*)0x10060004;
volatile uint32_t *GPIO_OUTPUT_EN = (uint32_t*)0x10060008;
volatile uint32_t *GPIO_OUTPUT_VAL = (uint32_t*)0x1006000C;

void gpio_switch_led(void) {
    int switches;

    // Activate two switch inputs and one LED output
    *GPIO_INPUT_EN = (1 << 1) | (1 << 0); // Switches/inputs on pins 1 and 0
    *GPIO_OUTPUT_EN = (1 << 2);           // LED/output on pin 2

    while (1) {                          // Loop forever
        switches = *GPIO_INPUT_VAL;       // Read switches
        if (switches) *GPIO_OUTPUT_VAL = 4; // Turn on LED
        else          *GPIO_OUTPUT_VAL = 0; // Turn off LED
    }
}

```

None of the code examples in this chapter are tested. *Caveat emptor.*

20.1.2 Loopback Testing

On the GPIO peripheral, `GPIO_PinsIn` and `GPIO_PinsOut` are normally connected at the top level of the chip to the I/O pin drivers. When the Wally `GPIO_LOOPBACK_TEST` configuration is set, the GPIO module connects the signals internally, so `GPIO_PinsIn` gets its value from `GPIO_PinsOut` when `GPIO_PinsEn` is set. This facilitates testing at lower levels of the SoC hierarchy when there are no pin drivers.

20.1.3 Test Plan

GPIO is tested by checking the reset state, then driving outputs and checking inputs in loopback mode. Each interrupt enable and pending bit is also exercised. The cvw repository enumerates all these tests.

The *volatile* keyword tells a C compiler that the variable might change between accesses, even though it is not modified by the program. It forces the compiler to fetch the variable from memory rather than use the last value stored in a register. Memory-mapped I/O registers should be declared volatile because the hardware might change them. For example, the `GPIO_INPUT_VAL` register will change when the user flips a switch.

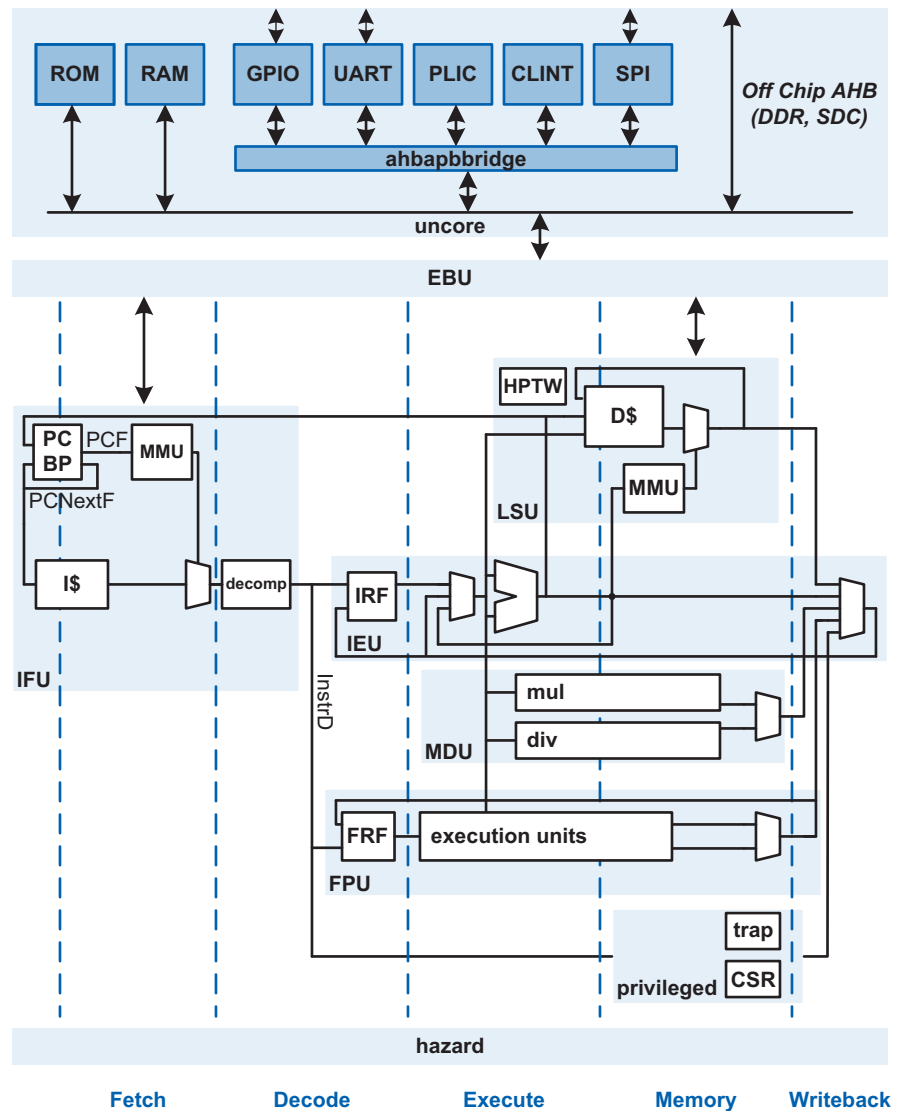


Fig. 20.3 Wally high-level block diagram, highlighting the peripherals.

20.1.4 Wally Implementation

Fig. 20.3 shows the Wally top-level diagram, highlighting the peripherals. Fig. 9.28 shows a more detailed schematic of the peripherals and their interface with the EBU (external bus unit). The on-chip ROM and RAM use the AHB bus provided by the EBU, and the other peripherals (GPIO, UART, PLIC, CLINT, and SPI) use an AHB-to-APB bridge to interface

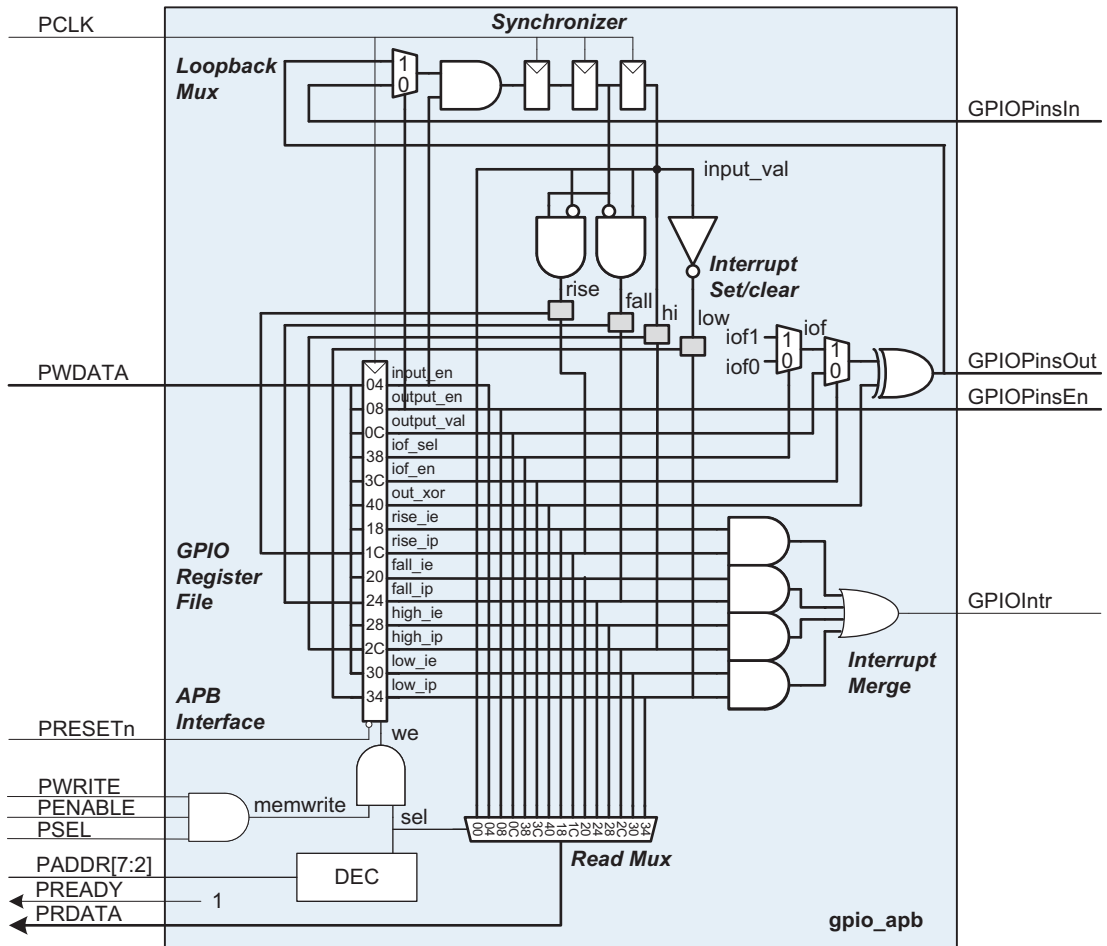


Fig. 20.4 GPIO implementation.

with the core through the EBU's AHB bus. Fig. 9.14 shows the detailed ahbabbridge module.

Fig. 20.4 shows a schematic of the GPIO module (gpio_apb) in the RV32 configuration of Wally. It is an extension of the simplified APB GPIO module from Section 9.2.1. The module contains a 14-register x 32-bit register file that holds the GPIO registers (excluding the input_val, pue, and ds registers from Table 20.1). input_val is a synchronized register at the top of the schematic, and pue and ds are not implemented in Wally. An address decoder examines the bottom bits of the address to determine which register to read or write. On the access phase of a write when the GPIO is selected, the memwrite signal asserts to enable the appropriate entry. The read multiplexer also reads the entry selected by the address decoder. The loopback mux takes the input from GPIOPinIn when the output is disabled and from GPIOPinOut when the output is enabled. If the input is enabled, the input goes through a two-stage

The RV64 design is identical except that it adds multiplexers on PRDATA and PWDATA to use the upper or lower 32 bits of a 64-bit word.

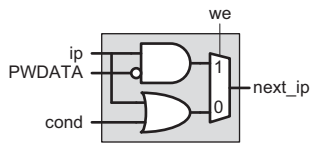


Fig. 20.5 Interrupt pending set/clear logic.

synchronizer and one more delay register. Output logic selects from `output_val` or optional alternative I/O functions (`iof`) and optionally inverts the polarity with an XOR. Interrupt set/clear logic detects rise/fall/high/low input conditions and writes to the interrupt pending registers and clears or sets the sticky registers. If any interrupt is both pending and enabled, `GPIOIntr` asserts.

Fig. 20.5 shows a schematic of the gray box in Fig. 20.4 that sets and clears pending interrupts. Recall that writing 1s to bits in an interrupt pending register clears those interrupts. `PWDATA` indicates which interrupts to clear. Pending interrupt bits are set when their condition (rise/fall/high/low) occurs, indicated by signal `cond`.

Wally does not rely on I/O pads having internal pull-up resistors or adjustable drive strength, so it does not implement the `pue` and `ds` registers.

20.2 CLINT

The Core Local Interruptor (CLINT) contains memory-mapped registers to generate software and timer interrupts and read the current time. Specifically, it has a 64-bit time counter, a 64-bit time compare register that produces a timer interrupt when the timer exceeds the compare value, and a 1-bit register to create a software interrupt. In a multicore system, the CLINT gives cores a common time reference and provides a timer compare and software interrupt for each core.

Fig. 20.6 shows a block diagram of the CLINT peripheral top-level interface. The module connects to the APB bus and sends M-mode timer and software interrupts to the core(s).

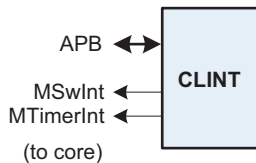


Fig. 20.6 CLINT top-level interface.

20.2.1 Memory Map

Table 20.2 lists the memory-mapped CLINT registers and their address offsets. For example, if the CLINT base address is `0x02000000` then a 64-bit read from `0x0200BFF8` reads the `time` register. `mtime` and `msip` are zeroed on reset.

The `time` memory-mapped register is a system timer that typically runs continuously at a constant frequency even if some or all of the cores are sleeping with their clocks stopped. It may use a different frequency clock than the core, especially if the core frequency is variable. If the timer is asynchronous to the cores, the CLINT requires clock domain crossing circuitry.

Recall from Section 8.2.3 that a hart can read (but not write) the read-only `time` memory-mapped register from the CLINT.

Table 20.2 CLINT register map. [FE310]

Offset	Name	Description
0x00	msip hart 0	1-bit software interrupt pending
$i * 4$	msip hart i	Additional bits if multiple harts exist
0x4000	mtimecmp hart 0	64-bit timer compare value
$0x4000 + i * 8$	mtimecmp hart i	Additional bits if multiple harts exist
0xBFF8	mtime	Shared 64-bit timer

Example 20.2 INTERRUPTS AND DELAYS USING THE CLINT

Write functions to set the CLINT software interrupt to 0 or 1 and to create a delay of a specified number of timer ticks.

Solution: The following program defines the CLINT register addresses. `setSWInt()` turns the software interrupt on or off. `delay()` causes the timer interrupt to rise after the specified number of ticks.

```
volatile uint64_t *CLINT_MSIP      = (uint64_t*)0x20000000;
volatile uint64_t *CLINT_MTIMECMP = (uint64_t*)0x20004000;
volatile uint64_t *CLINT_MTIME    = (uint64_t*)0x2000BFF8;

void setSWInt(int val) {
    *CLINT_MSIP = val;
}

void delay(uint64_t ticks) {
    // Create a delay of the specified number of timer ticks
    uint64_t doneTime = *CLINT_MTIME + ticks;

    while (*CLINT_MTIME < doneTime); // Spin until time has elapsed
}
```

20.2.2 Test Plan

The CLINT is tested by toggling software interrupts, `mtime` and `mtimecmp`, and checking that the `mip.MSIP` and `mip.MTIP` (software and timer interrupt pending) bits respond accordingly. The `cvw` repository enumerates all these tests.

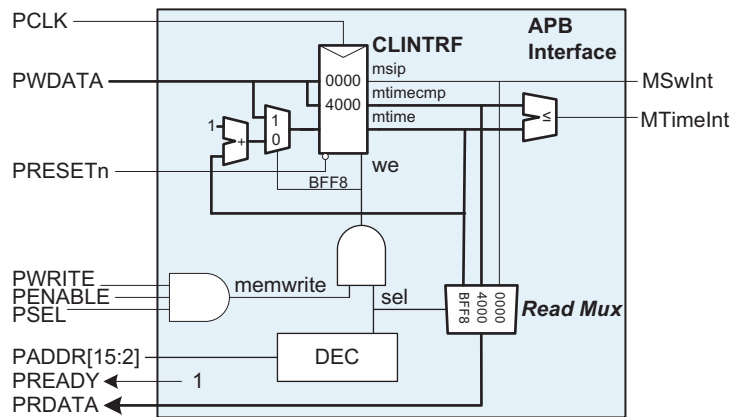


Fig. 20.7 CLINT implementation.

20.2.3 Wally Implementation

Fig. 20.7 shows a schematic of the CLINT module on RV64 with an APB interface.

The CLINT has a 64-bit `mtime` counter, a 64-bit `mtimecmp` register, 1-bit `msip` register, and a 64-bit timer interrupt comparator along with circuitry to read and write these registers. The `mtime` counter increments on each cycle unless it is being written.

20.3 PLIC

The Platform-Level Interrupt Controller (PLIC) takes interrupts from multiple peripherals, prioritizes them, and routes them to the appropriate hart's M- or S-mode external interrupt pin [PLIC22]. In general, the PLIC standard supports up to 1,024 *interrupt sources* and 15,872 *hart context* destinations, although a system normally has only a few of each.

The PLIC has an *interrupt pending* bit and a programmable *interrupt priority* for each of the interrupt sources. Source 0 does not exist. A priority of 0 means never interrupt.

The *hart context* is a specific privilege mode on a specific hart to which an interrupt could be routed. The PLIC drives one external interrupt pin for each hart context. M-mode has a hart context, and S-mode, when implemented, has a second hart context. The PLIC has a programmable *priority threshold* and *claim/complete* register for each context. It also has an *interrupt enable* bit for each source/context pair, so each context can receive interrupts from specific sources.

The PLIC identifies the highest priority pending interrupt. If multiple interrupts with the same priority are pending, it selects the lowest source number. The PLIC broadcasts that interrupt to all hart contexts whose

Early SiFive processors have 54 interrupt sources, of which only a fraction connect to peripherals. Wally does the same.

For example, a 4-core system with 2-way multithreading has 8 harts. Each hart has M- and S-mode, so the system has 16 hart contexts. Wally has 1- and 2-hart contexts.

interrupts are enabled for that source and whose priority threshold is less than the interrupt priority.

A hart context claims an interrupt by reading its claim/complete register. The register returns the source number of the highest priority pending interrupt or 0 if no interrupt is pending. Reading the register also clears the pending bit within the PLIC. The hart context should respond to the claimed interrupt and cause the peripheral to stop issuing the interrupt. For example, if the GPIO signals an interrupt on a rising input, the hart should clear the `rise_ip` bit. Then the hart context writes the interrupt source number back to the claim/complete register to indicate that the interrupt has been processed and that the PLIC can raise the pending bit again the next time that interrupt source is active.

Section 8.2.5.3 described external interrupts from a hart's perspective.

20.3.1 Memory Map

Table 20.3 lists the memory-mapped PLIC registers and their address offsets, assuming 54 sources and 2 contexts. All PLIC registers are 32 bits wide.

Wally's PLIC receives interrupts from two sources (UART and GPIO) and two hart contexts (M- and S-mode). Like the SiFive FE310, Wally supports up to 54 interrupt sources with 3-bit priorities (1–7). `PLIC_NUM_SRC` configures the actual number of sources. The interrupt

Table 20.3 PLIC register map.

Offset	Name	Description
0x000004	<code>intPriority1</code>	Priority for interrupt source 1
0x000008	<code>intPriority2</code>	Priority for interrupt source 2
...	...	...
0x0000DB	<code>intPriority53</code>	Priority for interrupt source 53
0x001000	<code>intPending0</code>	Pending bits for interrupts 31:0
0x001004	<code>intPending1</code>	Pending bits for interrupts 53:32
0x002000	<code>intEn00</code>	Interrupt enables for context 0 sources 31:1
0x002004	<code>intEn01</code>	Interrupt enables for context 0 sources 53:32
0x002080	<code>intEn10</code>	Interrupt enables for context 1 sources 31:1
0x002084	<code>intEn11</code>	Interrupt enables for context 1 sources 53:32
0x200000	<code>threshold0</code>	Priority threshold for context 0
0x200004	<code>claimComplete0</code>	Claim/complete register for context 0
0x201000	<code>threshold1</code>	Priority threshold for context 1
0x201004	<code>claimComplete1</code>	Claim/complete register for context 1

numbers are defined in `config.vh`, with `GPIOIntr` going to source `PLIC_GPIO_ID = 3` and `UARTIntr` going to source `PLIC_UART_ID = 10`. Because GPIO has a lower source number than the UART, GPIO takes precedence when the `intPriority` values are equal.

Example 20.3 EXTERNAL INTERRUPTS USING THE PLIC

Write a function to configure the PLIC to route UART and GPIO interrupts to Wally's M-mode external interrupt pin `MExtInt`, with the GPIO receiving higher priority. Write an external interrupt handler to determine which peripheral caused the interrupt and handle it accordingly.

Solution: The following program defines the PLIC register addresses. Context 0 is M-mode for Wally's only hart. `plicInit()` sets the GPIO to have an arbitrary priority higher than that of the UART and sets the threshold for context 0 to accept both sources. The program also enables interrupts from both sources. When an external interrupt occurs, the hart enters the M-mode interrupt handler and saves registers. It should then call `mextIntHandler()`, which reads the `claimComplete` register to determine which peripheral caused the external interrupt. It handles that interrupt (e.g., by reading data waiting on the UART) and ensures the peripheral is no longer asserting the interrupt. It then writes the peripheral source number back to `claimComplete` to inform the PLIC that the interrupt has been handled.

```
volatile uint32_t *PLIC_INTPRIORITY3 = (uint8_t*)0x0C00000C;
volatile uint32_t *PLIC_INTPRIORITY10 = (uint8_t*)0x0C000028;
volatile uint32_t *PLIC_INTEN00 = (uint8_t*)0x0C002000;
volatile uint32_t *PLIC_THRESHOLD0 = (uint8_t*)0x0C200000;
volatile uint32_t *PLIC_CLAIMCOMPLETE0 = (uint8_t*)0x0C200004;

void plicInit(void) {
    *PLIC_INTPRIORITY3 = 5; // GPIO has priority 5
    *PLIC_INTPRIORITY10 = 4; // UART has lower priority 4
    *PLIC_INTEN00 = (1 << 3) | (1 << 10); // Enable GPIO and UART
    *PLIC_THRESHOLD0 = 2; // Interrupt if source's priority > 2
}

void mexIntHandler(void) {
    uint32_t *source = *PLIC_CLAIMCOMPLETE0; // Interrupt cause?
    if (source == 10) handleUARTInt();
    else if (source == 3) handleGPIOInt();
    *PLIC_CLAIMCOMPLETE0 = source; // Signal completion
}
```

20.3.2 Test Plan

The general method of testing the PLIC is to go through various permutations of interrupts from the UART and GPIO that have different priorities and enables and that are routed to different contexts. The `cvw` repository enumerates all these tests.

20.3.3 Wally Implementation

See the Verilog code in `src/uncore/plic_apb.sv`. Major components include the following:

- ▶ The AHB interface
- ▶ The PLIC registers, including read and write logic
- ▶ Interrupt logic, which receives interrupt requests from each peripheral
- ▶ For each context {machine, supervisor}:
 - For each priority level [1...7]
 - Find which requests are pending and enabled at that context and priority
 - Determine whether that context and priority has at least one request
 - Find highest priority level with at least one pending enabled request
 - Find lowest-ID peripheral that has a pending enabled request at that priority level
 - Send interrupt to context if the priority is high enough for that context

20.4 UART

Computers historically communicated with teletype machines or VT100 text consoles over phone lines using a modem. A modem (short for modulator/demodulator) operates asynchronously because it is impractical to send a clock over a phone line. It communicates information with audio pulses at tens to thousands of bits per second traveling both directions along the line. The modem converts these audio pulses to electrical pulses and conveys them to a computer or device using a universal asynchronous receiver/transmitter (UART, pronounced “you-art”). Many devices still use the UART standard for backward compatibility or historical reasons, despite the existence of simpler and faster SPI and I²C serial links.



VT100 terminal. (© 2013 Jason Scott. This photo is openly licensed under CC BY 2.0.)

The Digital Equipment Corporation’s VT100 series of terminals, introduced in 1978, was a wildly successful way to communicate over a modem serial port with mainframe computers before the PC revolution. Over 6 million VT terminals were sold. Linux terminal windows still use serial communication, now called a COM port.

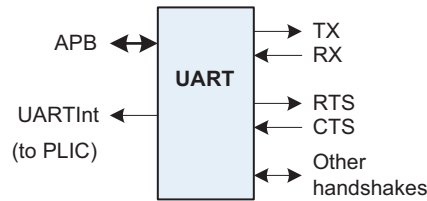


Fig. 20.8 UART top-level interface.

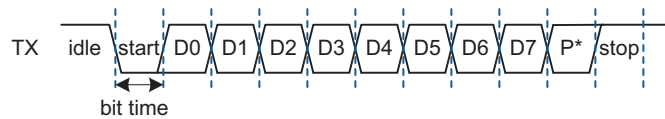


Fig. 20.9 UART timing.

Fig. 20.8 shows the top-level interface of a UART. Serial communication takes place over the TX and RX transmit and receive pins. The UART optionally has handshake signals (RTS, CTS, DSR, DTR, RI, and DCD) to control a modem.

Fig. 20.9 shows the waveforms to transmit a character over a UART. The TX line idles high. The start of the transmission is indicated by TX going low for one bit period; this is called the *start bit*. The UART then sends *data bits* (typically 8), optionally followed by a *parity bit* used to detect an error in the transmission. Finally, it raises TX for at least one bit period; this is called the *stop bit*. If the UART uses one start bit, one stop bit, no parity bits, and an 8-bit character, it takes 10 bit periods to send 8 data bits. Hence, the UART speed is called the *baud rate* rather than the bit rate. The bit period is the inverse of the baud rate. For example, 9600 baud means 9600 bit periods per second, or 960 characters per second ($9600 \text{ bit periods/second} \times 8 \text{ data bits/10 bit periods} \times 1 \text{ character/8 data bits}$) after accounting for the start and stop bits.

The UART is asynchronous, so the baud rate is set by mutual agreement between the transmitter and receiver rather than by a common clock. The existence and polarity of the parity bit and the number of data and stop bits is also set by agreement. Some systems need the optional handshaking signals, but others ignore them. TX of the UART is hooked up to RX of the device it is speaking to, and vice versa. In a normal cable, RTS of the UART connects to RTS of the device, and CTS connects to CTS, but in a null modem cable, RTS connects to CTS and vice versa. Sometimes the 0s and 1s are converted to $\pm 12 \text{ V}$ and sent over a bulky RS-232 serial cable. Alternatively, they may be sent across a circuit board at a standard digital logic level, such as 3.3 V. As you can see, UART communication has many fussy parameters that must be set exactly right

on each end, or the system will fail and be hard for ordinary users to troubleshoot.

A UART must generate signals to sample the received data at the right time. It typically begins with a high-speed peripheral clock PCLK operating at the system clock frequency (clk). The UART divides the peripheral clock down by a *baud rate divisor* $\times 2^{\text{UART_PRESCALE}}$ to 16 times the desired baud rate. The UART oversamples the RX line at this 16x clock. The first high-to-low transition indicates going from idle to the start bit. The UART can then wait 8 clock periods and sample every 16 clocks thereafter to measure the data bits in the middle of their window.

For example, suppose the system clock runs at 320 MHz, and $\text{UART_PRESCALE} = 5$. If the baud rate is 9600, the 16x clock is at $9600 \times 16 = 153.6$ kHz, and its period is $1/153600 = 6.51$ μs . The baud rate divisor should be the baud period/ $(2^{\text{UART_PRESCALE}} / f_{\text{clk}}) = 6.51 \times 10^{-6} \text{ s} / (2^5 / 320 \times 10^6 \text{ Hz}) = 65.1$, so the UART uses a baud rate divisor of 65 to generate a 16x clock with a 6.5- μs period. The error is small enough that the UART can sample all of the bits in the character accurately, and then synchronize again to the next start bit.

A UART classically has a *Transmit Holding Register* (THR) and a *Receiver Buffer Register* (RBR) along with control signals to indicate whether each of these registers is full. A host processor sends a character by waiting until the THR is empty and then writing a byte to the THR. The UART marks THR full, generates the appropriate start, stop, parity, and data bits at suitable times and sends them out the TX line, then marks THR empty again. Similarly, a host processor receives a character by waiting until RBR is full and reading it out. If a second character arrives before the host reads the first from the RBR, a *buffer overflow* occurs, and data is lost. The system must use RTS/CTS or software flow control to prevent buffer overflow. More advanced UARTs have a first-in, first-out (FIFO) buffer on the transmitter and receiver so that the system can send and receive many characters before having to wait for buffers to empty. UARTs may also generate interrupts when the buffers are empty so that the host can do other work and get notified when the next character is ready.

The original IBM PC used the 8250 UART in its COM port. At speeds above 9600 baud, the PC had trouble servicing interrupts fast enough to avoid buffer overflows. The 16550 UART [PC16550] added FIFOs while maintaining backward compatibility, and the register set for the 8250/16550 remains a widely used standard. National Semiconductor initially manufactured the 16550 used in IBM PS/2 computers. Texas Instruments and others soon developed compatible products. RISC-V Linux has device drivers for UARTs compatible with the classic 16550, as well as the simpler SiFive FU540 UART [FU540].

USB was developed to end these troublesome UART configurations as well as to increase the data rate, reduce the connector size, and reduce cable cost. Unfortunately, USB has inherent security holes. For example, the firmware on a USB drive can be configured to pose as a keyboard device instead and type malicious keystrokes into a computer.

20.4.1 Memory Map

The PC16550 has a memory-mapped interface with eight 8-bit registers at offsets of 0–7 from the base address. Fig. 20.10 shows the PC16550 register map. A read from offset 0 reads the RBR, while a write to offset 0 writes the THR. Offset 1 is the Interrupt Enable Register (IER), which controls what events generate interrupts. Reads from offset 2 return the Interrupt Identification Register (IIR) that provides information about the interrupt, and writes to offset 2 control the FIFO. The Line Control Register (LCR) configures the number of start bits, the parity, the number of bits in a character, and so on. The Modem Control Register (MCR) generates outputs such as DTR and RTS. Bit 4 is the Loop field, which configures the transmit and modem control outputs to feed back to the receive and modem status inputs for loopback testing. The Line Status Register (LSR) indicates error conditions such as parity or buffer overrun. The Modem Status Register (MSR) has more handshaking signals. The scratch register (SCR) holds one byte of information; it has little practical purpose now. Bit 3 of the LCR is the Divisor Latch Access Bit (DLAB). When DLAB is set, writes to address 0 and 1 set the low and high byte of the

Bit No.	REGISTER ADDRESS											
	0 DLAB=0	0 DLAB=0	1 DLAB=0	2	2	3	4	5	6	7	0 DLAB=1	1 DLAB=1
	Receiver Buffer Register (Read Only)	Transmitter Holding Register (Write Only)	Interrupt Enable Register	Interrupt Ident. Register (Read Only)	FIFO Control Register (Write Only)	Line Control Register	MODEM Control Register	Line Status Register	MODEM Status Register	Scratch Register	Divisor Latch (LS)	Divisor Latch (MS)
	RBR	THR	IER	IIR	FCR	LCR	MCR	LSR	MSR	SCR	DLL	DLM
0	Data Bit 0 ⁽¹⁾	Data Bit 0	Enable Received Data Available Interrupt (ERBFI)	"0" if Interrupt Pending	FIFO Enable	Word Length Select Bit 0 (WLS0)	Data Terminal Ready (DTR)	Data Ready (DR)	Delta Clear to Send (DCTS)	Bit 0	Bit 0	Bit 8
1	Data Bit 1	Data Bit 1	Enable Transmitter Holding Register Empty Interrupt (ETBEI)	Interrupt ID Bit (0)	RCVR FIFO Reset	Word Length Select Bit 1 (WLS1)	Request to Send (RTS)	Overrun Error (OE)	Delta Data Set Ready (DDSR)	Bit 1	Bit 1	Bit 9
2	Data Bit 2	Data Bit 2	Enable Receiver Line Status Interrupt (ELSI)	Interrupt ID Bit (1)	XMIT FIFO Reset	Number of Stop Bits (STB)	Out 1	Parity Error (PE)	Trailing Edge Ring Indicator (TERI)	Bit 2	Bit 2	Bit 10
3	Data Bit 3	Data Bit 3	Enable MODEM Status Interrupt (EDSSI)	Interrupt ID Bit (2) ⁽²⁾	DMA Mode Select	Parity Enable (PEN)	Out 2	Framing Error (FE)	Delta Data Carrier Detect (DDCD)	Bit 3	Bit 3	Bit 11
4	Data Bit 4	Data Bit 4	0	0	Reserved	Even Parity Select (EPS)	Loop	Break Interrupt (BI)	Clear to Send (CTS)	Bit 4	Bit 4	Bit 12
5	Data Bit 5	Data Bit 5	0	0	Reserved	Stick Parity	0	Transmitter Holding Register (THRE)	Data Set Ready (DSR)	Bit 5	Bit 5	Bit 13
6	Data Bit 6	Data Bit 6	0	FIFOs Enabled ⁽²⁾	RCVR Trigger (LSB)	Set Break	0	Transmitter Empty (TEMT)	Ring Indicator (RI)	Bit 6	Bit 6	Bit 14

Fig. 20.10 PC16550 register map. [PC16550]

baud rate divisor, respectively, rather than the THR and IER. This divisor is normally programmed during initialization, and then DLAB is cleared.

Fig. 20.11 shows the manufacturer's block diagram of the PC16550, including each register. The serial output comes from the transmitter shift register, which comes from either the THR or transmitter FIFO. Similarly, the serial input goes to a receiver shift register and on to the RBR or receiver FIFO.

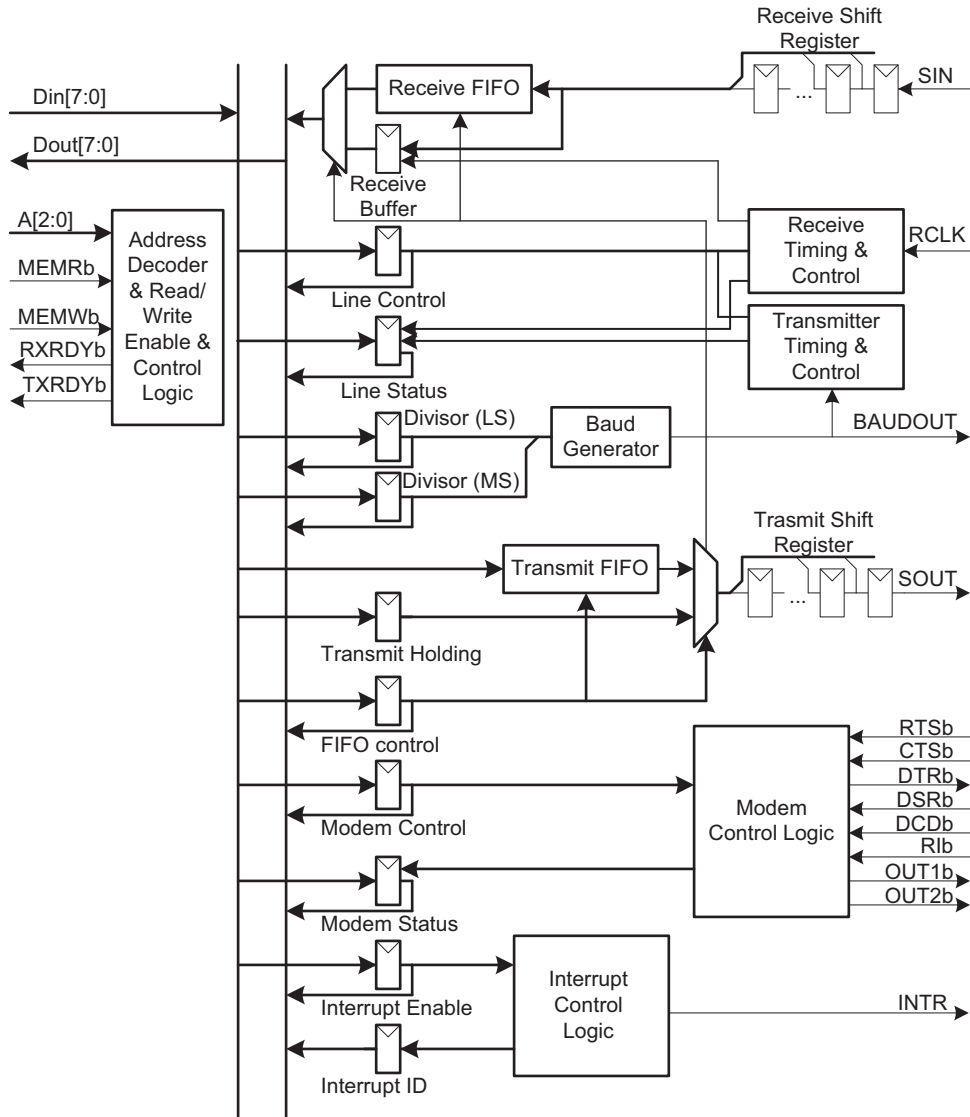


Fig. 20.11 PC16550 block diagram. [PC16550]

Example 20.4 UART PROGRAM

Write a function to configure the UART to operate at 9600 baud with eight data bits, one start bit, one stop bit, and no parity. Set the baud rate divisor to 65 as in the earlier example. Also write functions to send a character and receive a character.

Solution: The following program defines the UART register addresses. Note that the RBR, THR, and DLL are all at offset 0. When `DLAB = 1`, the UART accesses the DLL. Otherwise, writes go to the THR and reads come from the RBR. `uartInit()` sets the Line Control Register (LCR) to configure the data, stop, and parity bits. It sets `DLAB`, then writes the clock divisor to DLL, then clears `DLAB`. `uartSend()` and `uartReceive()` wait for the transmit holding register to be empty or the data ready signal to be asserted, then write or read the character.

```
volatile uint8_t *UART_RBR = (uint8_t*)0x10000000;
volatile uint8_t *UART_THR = (uint8_t*)0x10000000;
volatile uint8_t *UART_DLL = (uint8_t*)0x10000000;
volatile uint8_t *UART_LCR = (uint8_t*)0x10000003;
volatile uint8_t *UART_LSR = (uint8_t*)0x10000005;

void uartInit(void) {
    *UART_LCR = 0b1000011; // 8-bit data, 1 stop bit, no parity, DLAB
    *UART_DLL = 65;        // Write clock divisor with DLAB = 1
    *UART_LCR = 0b0000011; // Turn off DLAB
}

void uartSend(char c) {
    while (!(*UART_LSR & (1<<5))); // Wait for THRE (THR empty)
    *UART_THR = c;
}

char uartReceive(void) {
    while (!(*UART_LSR & (1<<0))); // Wait for DR (Data Ready)
    return *UART_RBR;
}
```

20.4.2 Test Plan

The general method of testing the UART is to transmit data in loopback mode under various configurations and check that the received data is correct. The tests check each configuration of the line control register. They verify that interrupts occur from the transmitter's line status register, the modem control register, and the FIFO. They also check that the receiver FIFO triggers thresholds and overflow. The `cvw` repository enumerates all these tests.

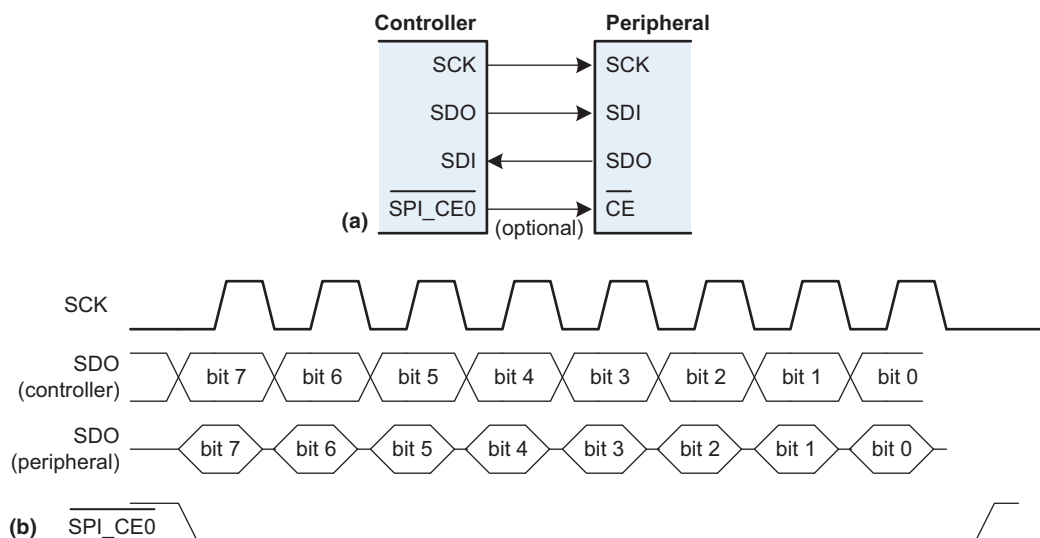


Fig. 20.12 SPI link: (a) SPI controller-peripheral interface and (b) SPI signaling. Reprinted from [Harris21].

20.4.3 Wally Implementation

Wally has a 16550-compatible UART. The logic is interesting to those who are curious how a serial port works, but it is too complex to describe in this chapter. Consult Wally’s Verilog (`src/uncore/uartPC16550D.sv`) for details.

20.5 SPI

The Serial Peripheral Interface (SPI, pronounced “S-P-I”) is a simple standard for synchronous serial communication between a controller and a peripheral. It is easier to implement and runs far faster than a UART. As shown in Fig. 20.12(a), the physical interface consists of three pins: serial clock (SCK), serial data out (SDO), and serial data in (SDI). The controller initiates communication by sending clock pulses on SCK. The controller sends data from its SDO pin to the peripheral’s SDI pin one bit per cycle, starting with the most significant bit. The peripheral may simultaneously respond with its SDO pin back to the controller’s SDI pin. Fig. 20.12(b) shows the SPI waveforms for an 8-bit data transmission. Bits change on the falling edge of SCK and are stable to sample on the rising edge. The SPI interface may also send an active-low chip enable ($\overline{CE}$) to select one peripheral from among several sharing an SPI bus. For high-performance operations such as loading code from a Flash memory chip, SPI can use bit widths of 1, 2, or 4 for SDO and SDI signals. The clock polarity can optionally be inverted.

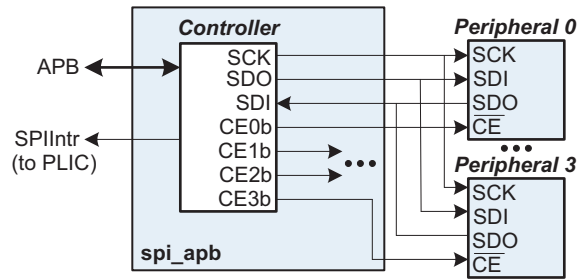


Fig. 20.13 SPI top-level interface.

Wally contains two SPI controllers compatible with the SiFive FU540 SPI [FU540] except that it does not implement memory-mapped SPI flash mode or 2/4-bit data width. The first SPI controller is instantiated as a generic SPI interface using `SPI_SUPPORTED = 1`, and the second is reserved for the SD flash memory card using `SDC_SUPPORTED = 1`. The SPI controller has the following features:

- ▶ Clock divider to produce a slower off-chip SCK from the high-speed on-chip APB PCLK
- ▶ Configurable serial clock phase and polarity
- ▶ 4 configurable chip select signals with adjustable delay
- ▶ Configurable data frame size (1-8 bits) and endianness
- ▶ 8-frame transmit and receive FIFOs with interrupts

Fig. 20.13 shows a block diagram of the SPI top-level interface. The SPI controller connects to the Wally core via the APB bus. It communicates with peripherals over the SCK, SDO, and SDI lines. It has four low-asserted chip enable (CEb) outputs to select from among up to four peripherals sharing the bus. Each peripheral has a tristate driver on its SDO output controlled by its chip enable, so at most one peripheral will send a response to the controller. The SPI controller sends SPIIntr back to the PLIC to raise an interrupt when the transmit FIFO is nearly empty or the receive FIFO is nearly full.

20.5.1 Memory Map

Table 20.4 lists the memory-mapped SPI registers and their address offsets. All SPI registers are 32 bits, even on RV64. See [FU540] for details. Wally's SPI is configured for a base address of 0x10040000 and connects to interrupt ID 6 of the PLIC.

Table 20.4 GPIO register map.

Offset	Name	Description	Details
0x00	sckdiv	Serial clock divisor	$f_{\text{SCK}} = f_{\text{PCLK}} / 2(\text{sckdiv}[11:0] + 1)$
0x04	sckmode	Pin input enable	Bit 0: phase, bit 1: polarity
0x10	csid	Chip select ID	Which chip select to assert
0x14	csdef	Chip select default	Typically 1111 (all CS off)
0x18	csmode	Chip select mode	Assert CS between frames
0x28	delay0	Delay control 0	Extra delay between CS and SCK
0x2C	delay1	Delay control 1	CS inactive time
0x40	fmt	Frame format	SPI width, endianness, bits/frame
0x48	txdata	Transmit FIFO data	[7:0] data; [31] full flag
0x4C	rxdata	Receive FIFO data	[7:0] data; [31] empty flag
0x50	txmark	Transmit FIFO watermark	Threshold for interrupt
0x54	rxmark	Receive FIFO watermark	Threshold for interrupt
0x70	ie	SPI interrupt enable	[1:0]: {Tx, Rx} watermark enable
0x74	ip	SPI interrupt pending	[1:0]: {Tx, RX} watermark pending

Example 20.5 SPI PROGRAM

Consider interfacing the MCP3002 analog-to-digital converter (ADC) to an SoC using our SPI controller as shown in Fig. 20.14. Assume the SoC is compatible with 3.3 V I/O levels. Read channel 0 of the MCP3002 by sending 0x60 over SPI to begin conversion, then sending 0x00 and receiving a byte back from the ADC. Assume the APB clock runs at 200 MHz and the SPI clock should run at 2 MHz.

Solution: The following program initializes the SPI clock divisor to 49 to divide by $2(49 + 1) = 100$ to reduce the 200 MHz clock to 2 MHz. It assumes the clock mode and chip select registers are in their default states and do not need initialization. It then waits until the SPI transmit FIFO is not full, sends 0x60 to begin conversion, waits until the receive FIFO is not full, and receives a byte back. It discards this dummy byte. Then it sends 0x00 and receives back the ADC result.

```

volatile uint32_t *SPI_SCKDIV = (uint32_t*)0x10040000;
volatile uint32_t *SPI_TXDATA = (uint32_t*)0x10040048;
volatile uint32_t *SPI_RXDATA = (uint32_t*)0x1004004C;

uint8_t spiSendReceive(uint8_t send) {
    uint32_t rxdata;

    while (*SPI_TXDATA >> 31); // Wait until TX FIFO is not full
    *SPI_TXDATA = send;       // Send a byte
    do { // Wait until RX FIFO is not empty, then return data
        rxdata = *SPI_RXDATA;
        if (!(rxdata >> 31) return (uint8_t)rxdata;
    }
}

void main(void) {
    uint8_t sample;

    *SPI_SCKDIV = 49; // Clock divisor of 49 to divide PCLK by 100

    while (1) {
        spiSendReceive(0x60); // Initiate conversion
        sample = spiSendReceive(0x00); // Receive sample back
        delay(200);
    }
}

```

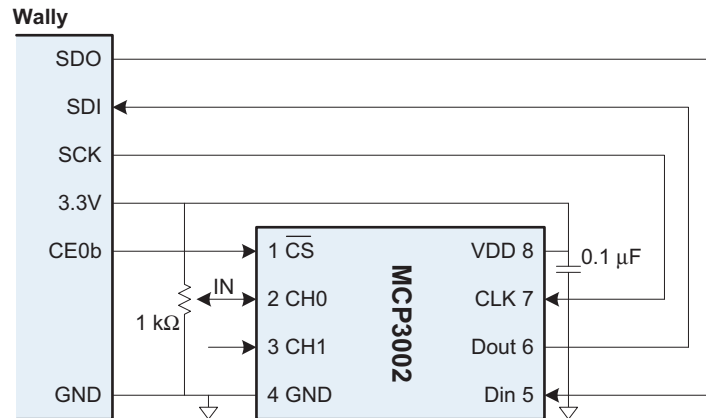


Fig. 20.14 A/D converter interfaced to Wally SoC.

20.5.2 Loopback Testing

On the SPI module, the SDI pin normally connects to an off-chip peripheral. When the Wally `SPI_LOOPBACK_TEST` configuration parameter is set, the SDI pin gets its value from SDO so the SPI can read back values it transmits.

With the default clock `sckmode = 00`, the SPI transmitter changes data on the falling edge of the clock, and the SPI receiver samples data on the rising edge. Thus, in loopback testing, SDI is stable when sampled.

20.5.3 Test Plan

The cvw repository contains `tests/wally-riscv-arch-test/riscv-test-suite/rv{32/64}_i/privilege/src/WALLY-spi-01.S` and corresponding `WALLY-spi-01.reference_output` that provide the SPI test cases.

The tests run in loopback mode. They check the reset state, then test each configuration register while sending single and 4-byte frame burst messages and checking that the same value was received. It then enables the PLIC and tests watermark interrupts.

The SPI has not yet been tested with external peripherals.

20.5.4 Wally Implementation

The SPI peripheral has an APB interface to the memory-mapped registers, transmit and receive FIFOs, and control logic driving the SPI pins. Consult the Verilog (`src/uncore/spi_apb.sv` and `src/uncore/spi_controller.sv`) for details.

20.6 SUMMARY

RISC-V systems commonly use a Core Local Interruptor (CLINT) to generate timer and software interrupts, and a Platform-Level Interrupt Controller (PLIC) to collect external interrupts from other peripherals and route them to machine or supervisor external interrupts of a core. They often have other peripherals such as a general-purpose I/O (GPIO) to read and write digital pins, a universal asynchronous receiver/transmitter (UART) for COM port communication, and a Serial Peripheral Interface (SPI) for higher-speed synchronous serial communication. The SiFive implementations of these peripherals are a de facto RISC-V standard with OS device driver support. This chapter described the function, memory map, testing, and implementation of each of the peripherals.

The peripherals use memory-mapped I/O such that processor loads or stores to certain addresses interact with the hardware and external world, instead of memory. The PMA checker detects accesses to addresses in the peripherals, memory map and generates the corresponding AHB HSEL. The AHB-to-APB bridge in the uncore provides a simple synchronous interface to each peripheral with a few bits of PADDR to select the peripheral and PRDATA/PWDATA to read and write data busses.

Benchmarking

21

The goal of benchmarking is to quantify the performance of a computer. This turns out to be a remarkably subtle problem [Hennesy19]. Different programs run better on different computers, and small programs are notoriously susceptible to clever optimizations. Section 2.8.5 summarized the Wally CoreMark and Embench benchmarks scores. This chapter describes each benchmark in more detail, including how to compile and run it. It also discusses performance validation tests run to check that the caches and branch predictor are performing as expected.

21.1 COREMARK

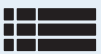
CoreMark from EEMBC is a widely used and freely available benchmark to measure embedded processor performance. It performs the following steps that exercise the execution units, memory system, and branch predictor:

- ▶ List processing: find and sort
- ▶ Matrix multiplication
- ▶ State machine: parsing an input stream
- ▶ Cyclic redundancy check (CRC)

CoreMark was developed in 2009 to address limitations of an earlier benchmark called Dhrystone, which was overly susceptible to compiler and library optimizations. Both are small synthetic benchmarks (as opposed to real applications) that attempt to reflect the types of operations commonly used in integer programs, while being easy to compile and fit in a small processor’s memory. Both avoid floating-point arithmetic, whose performance is highly dependent on whether the processor has a floating-point unit.

The main CoreMark source code is in the `addins/coremark` directory as a submodule linked to the master repository at `github.com/eembc/coremark`. The RISC-V port and Makefile for Wally are in `benchmarks/coremark`.

- 21.1 CoreMark
- 21.2 Embench
- 21.3 Performance Verification
- 21.4 Summary

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

“[Synthetic benchmarks] are discredited today, usually because the compiler writer and architect can conspire to make the computer appear faster on these stand-in programs than on real applications.”—David Patterson & John Hennessy [Hennesy19]

To compile the CoreMark benchmark to run on the bare-metal rv32gc Wally processor configuration with caches and branch prediction but no operating system, compiled for RV32IMB:

```
$ cd $WALLY/benchmarks/coremark
$ make
```

`make` builds the `coremark.bare.riscv` executable file, object dump, and `memfile` in the `work` subdirectory. Expect some spurious warnings about `ITERATIONS` being redefined. Then `make run` to simulate in Verilator.

```
$ make run
```

`make run` invokes Verilog simulation with the same `wsim` command and `testbench.sv` as used in verification tests. However, CoreMark is slightly

```
WALLY CoreMark Results (from get_time)
Elapsed MTIME: 2972203
Elapsed MINSTRET: 2589000
COREMARK/MHz Score: 10,000,000 / 2972203 = 3.36
CPI: 2972203 / 2589000 = 1.14
2K performance run parameters for coremark.
CoreMark Size      : 666
Total ticks        : 2972203
Total time (secs): 29
Iterations/Sec     : 0
Iterations         : 10
Compiler version   : GCC13.2.0
Compiler flags     : -mcmmodel=medany -fno-tree-loop-distribute-patterns-fno-common -lm -lgcc
-T /home/harris/cvw/benchmarks/coremark/riscv64-baremetal/link.ld -g -mabi=ilp32 -march=rv32im
_zicsr_zba_zbb_zbs -static -falign- functions=16 -mbranch-cost=1 -DSKIP_DEFAULT_MEMSET
-mtune=sifive -3 -series -03 -finline-functions -falign-jumps=4 -fno-delete-null- pointer-
-checks -fno-renameregisters --param=loop-max-datarefs-for-datadepts=0 -funroll-all-loops on-
--param=uninlined-functi insns=8 - fno-tree-vrp -fwrapv -fipa-pta -nostdlib -nostartfiles
-ffreestanding -mstrict-align-DTOTAL_DATA_SIZE=2000 -DMAIN_HAS_NOARGC=1
DPERFORMANCE_RUN=1 -DITERATIONS=10 -DXLEN=32 -static-libgcc -lgcc
Memory location   : Code and Data in external RAM
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xfcaf
Correct operation validated. See README.md for run and reporting rules.
Load Stalls 262774
Store Stalls 26087
D-Cache Accesses 698750
D-Cache Misses 76
I-Cache Accesses 2868917
I-Cache Misses 246
Branches 591846
Branches Miss Predictions 66966
BTB Misses 5570
Jump and JR 31170
RAS Wrong 45
Returns 15244
BP Class Wrong 19469
```

Fig. 21.1 CoreMark run output.

different in that it terminates with an `ecall` rather than with a `write_tohost`. The simulation takes about two minutes and prints the report shown in Fig. 21.1.

CoreMark runs a specified number of iterations (default 10) and measures the number of clock cycles required to execute. The CoreMark/MHz score is $1,000,000 * \text{ITERATIONS}/\text{cycles}$. In this run, it is $1,000,000 * 10/2,972,203 = 3.36$.

The simulation also reports results from performance counters. The cycles per instruction (CPI) is the number of clock cycles divided by the number of instructions retired; this is measured with `mtime/minstret` = $2,972,203/2,589,000 = 1.14$.

The difference `mtime - minstret` = 383,203 is the number of excess cycles caused by pipeline hazards. The performance counters show this includes 262,774 load stalls caused by the 2-cycle load-use latency, 26,087 store stalls, and 66,966 branch mispredictions causing pipeline flushes (11% branch misprediction rate). Other stalls come from multiply/divide stalls and cache misses. The working set fits into the cache, and the cache was already warmed up, so very few cache misses occur.

`mcycles` would be more proper than `mtime`, but the two are identical on Wally.

21.1.1 Porting CoreMark

Compiling CoreMark requires modifying files in the `benchmarks/coremark/riscv64-baremetal` directory, particularly `core_portme.c`, to access platform-specific timers and printing.

The `GETMYTIME` macro for Wally is defined as:

```
#define GETMYTIME(_t) (_t = *(volatile ee_ptr_int*)0x0200BFF8)
```

This macro reads a 64-bit word from `0x0200BFF8`, which is the `mtime` register in the CLINT.

The `_send_char` function for Wally is defined below. It declares pointers to the UART memory-mapped Transmit Holding Register (THR) and Line Status Register (LSR). It waits until bit 5 of the LSR is 1, which indicates that the Transmit Holding Register is Empty (THRE). Then it writes a character to the THR and waits again until THRE shows that the character was sent.

```
void _send_char(char c) {
    volatile unsigned char *THR=(unsigned char *)0x10000000;
    volatile unsigned char *LSR=(unsigned char *)0x10000005;

    while(!(*LSR&0b100000));
    *THR=c;
    while(!(*LSR&0b100000));
}
```

Various other printing changes in `syscalls.c` were made to display performance statistics.

A key to performance in RV64 is to redefine the `ee_u32` datatype from unsigned to signed in `core_portme.h`:

```
typedef signed int ee_u32;
```

This data type represents an unsigned 32-bit integer and is widely used in CoreMark. When compiling for RV64, unsigned 32-bit integers are expensive because bits 63:32 must be cleared to 0 after each operation using left and right 32-bit logical shifts or `zext.w`. The upper bits don't matter, so changing to signed integers allows RV64 to use its default sign extension, giving a 13% CoreMark speed boost. This change is in violation of CoreMark requirements [github.com/eembc/coremark] but is widely done in RISC-V 64-bit benchmarks.

21.1.2 CoreMark Compiler Flags

CoreMark scores are strongly dependent on the compiler and, to boost performance scores, some compilers have been tuned with optimizations only applicable to CoreMark. But vendors are required to post their compiler version and flags along with their scores. In this chapter, Wally's CoreMark scores were found using programs compiled with GCC 13.2.0 using the flags listed in [Table 21.1](#), adapted from the flags used by Western Digital's SweRV (now called VeeR) core.

21.1.3 Instruction Set Impact

[Table 21.2](#) shows the CoreMark performance for `rv{32/64}i`, `im`, `imc`, `imb`, `gc`, and `gcb` Wally configurations. The configurations use a software multiplication subroutine that cripples performance (i.e., `rv64i` achieves only 1.02 CoreMarks/MHz vs. 2.85 for `rv64im`). `imc` and `gc` performances are nearly identical because CoreMark does not use floating-point or atomic instructions. They are slightly slower than `im` because compressed instructions can cause cache line spills. Adding bit manipulation instructions speeds up CoreMark by 8% because it saves about that fraction of instructions. `rv64` configurations are slower than `rv32` configurations because extra code is sometimes needed to handle 32-bit variables. Generate the results from [Table 21.2\(a and b\)](#) using the following commands:

```
$ cd $WALLY/benchmarks/coremark
$ ./coremark_sweep.py
<look at coremark_results.csv>
```

The current scores for CoreMark (and other benchmarks described in this chapter) are likely to be different from what is listed here due to subsequent compiler changes or bug fixes in Wally.

21.2 EMBENCH

Embench is a more recent benchmark for embedded processors, hosted by the Free and Open Source Silicon Foundation (FOSSi) [Bennett19,

Table 21.1 CoreMark compiler flags.

Flag	Meaning
-march=rv64im	Compile for RV64IM instruction set
-mabi=lp64	Longs and pointers are 64 bits
-mcmmodel=medany	Generate position-independent code up to 2 GiB in size.
-O3	High optimization, trading extra code size for greater speed
-g	Generate debug symbols
-static	Prevents linking to dynamic shared libraries
-nostdlib	Do not use the standard startup files or libraries when linking
-lgcc	Link libgcc.a with some essential GCC capabilities
-nostartfiles	Do not link the standard system startup file; redundant with -nostdlib
-mstrict-align	Require memory accesses be aligned to multiples of the object size
-mbranch-cost=1	Set the cost of a branch to roughly 1 instruction
-mtune=sifive-3-series	Optimize for processors with a pipeline similar to the SiFive 3-series
-falign-functions=16	Align functions to 16-byte boundary to improve cache behavior
-funroll-all-loops	Unroll loops to improve speed at cost of size
-finline-functions	Consider compiling all functions inline to improve speed at cost of size
-falign-jumps=4	Align branch target to multiple of 4 bytes to improve instruction fetch
-fno-delete-null-pointer-checks	Disable optimizing NULL pointer behavior
-fno-rename-registers	Keep initial register allocation
-fno-tree-vrp	Keep range checking
-fwrapv	Two's complement arithmetic wraps normally
-fipa-pta	Perform interprocedural pointer analysis
-fno-tree-loop-distribute-patterns	
-fno-common	Place uninitialized global variables in BSS section
-ffreestanding	Don't assume standard functions are defined.
--param=uninlined-function-insns=8	Estimated function overhead for inline decisions
--param=loop-max-datarefs-for-datadeps=0	Loop size limit for analyzing data dependence
-D<Variable=Value>	Equivalent to #define Variable Value
-lm	Link math library
-T \$WALLY/benchmarks/coremark/riscv64-baremetal/link.ld	Linker script path

Table 21.2(a) CoreMark performance vs. ISA/configuration (RV32).

-march flag	rv32i_zicsr	rv32im_zicsr	rv32imc_zicsr	rv32imb_zicsr	rv32gc	rv32gcb
CM/MHz	1.20	3.28	3.26	3.36	3.26	3.30
CPI	1.17	1.14	1.15	1.14	1.15	1.16
mtime	8,271,096	3,044,823	3,060,565	2,972,203	3,060,566	3,022,206
minstret	7,060,812	2,654,652	2,654,292	2,589,000	2,654,292	2,589,000
Load stalls	260,731	262,883	261,746	262,774	261,469	263,526
Store stalls	23,135	26,003	25,971	26,087	25,970	27,308
D\$ accesses	722,801	698,738	698,739	698,750	698,732	698,744
D\$ Misses	73	74	74	76	74	73
I\$ accesses	7,797,035	2,935,727	2,962,051	2,868,917	2,961,352	2,917,432
I\$ misses	899	644	245	521	247	250
Branches	2,057,061	591,370	590,315	591,846	590,031	591,611
Branch mispredicts	474,489	67,668	67,185	66,966	67,106	69,168
BTB misses	6,059	6,389	7,921	5,570	7,943	7,526
Jump/jr	120,259	31,518	31,160	31,170	31,153	31,165
RAS wrong	6,772	222	112	45	122	81
Returns	116,880	15,232	15,232	15,244	15,225	15,237
BP class pred wrong	21,846	22,747	29,107	19,469	29,135	26,582

Patterson25]. It draws lessons from previous benchmarks about what is needed for a benchmark to be widely used and credible. Embench is:

- ▶ Free and easy to use.
- ▶ Based on a suite of actual programs that are really used.
- ▶ Supported by both academics and industry.
- ▶ Summarized by a single score based on the geometric mean of the program performances.

Table 21.3 summarizes the programs included in Embench Version 1.0. The programs span a range of common embedded applications, from sorting to data compression to mathematics. All are less than 3,000 lines of C code and require less than 16 KiB of program memory and 4 KiB of data

Table 21.2(b) CoreMark performance vs. ISA/configuration (RV64).

-march flag	rv64i_zicsr	rv64im_zicsr	rv64imc_zicsr	rv64imb_zicsr	rv64gc	rv64gcb
CM/MHz	1.02	2.85	2.83	3.06	2.83	2.98
CPI	1.14	1.13	1.13	1.13	1.13	1.16
mtime	9,725,033	3,504,674	3,525,092	3,262,173	3,525,094	3,345,692
minstret	8,499,801	3,101,161	3,101,161	2,874,629	3,101,161	2,874,629
Load stalls	272,724	270,290	272,097	272,986	271,784	280,263
Store stalls	22,035	22,019	21,984	24,519	21,984	24,478
D\$ accesses	725,556	688,278	688,279	688,130	688,272	688,124
D\$ misses	84	84	85	85	85	85
I\$ accesses	9,210,255	3,382,887	3,409,709	3,146,506	3,408,735	3,227,233
I\$ misses	352	477	231	503	233	219
Branches	2,387,629	601,917	602,012	602,415	601,692	601,273
Branch mispredicts	486,766	72,993	70,402	65,720	70,426	73,102
BTB misses	6,496	7,078	6,276	4,585	6,276	5,225
Jump/jr	131,111	36,818	36,820	36,831	36,813	36,824
RAS wrong	232	249	101	76	101	161
Returns	109,378	15,094	15,094	15,105	15,087	15,099
BP class pred wrong	19,564	22,488	19,904	15,049	19,904	17,622

memory, so they fit on the smallest of processors. Different programs stress the branch predictor, memory system, and ALU in different ways. Code size is measured as the number of instruction bytes when compiled for a Cortex-M4 with the `-Os` compiler flag. Each program is tuned to take about 4 seconds to run on a Cortex-M4 reference processor, so the entire benchmark can run in a few minutes on physical hardware.

The Embench repository is incorporated as a submodule in `addins/embench-iot`. The Wally Makefile to run Embench is in `benchmarks/embench`.

Embench depends on the Python `lief` library to parse ELF files. This library is already installed as part of the installation script, but if you need to install it manually, use:

```
$ pip3 install lief
```

Table 21.3 Embench 1.0 programs.

Name	Description	Original Source	C Lines of Code	Code Size (bytes)	Data Size (bytes)	Branch	Memory	Compute
aha-mont64	Montgomery mult	AHA	151	1,052	24	low	low	high
crc32	CRC error check	MiBench	101	230	1,024	high	med	low
cubic	Cube root solver	MiBench	121	2,466	192	low	med	med
edn	Filter	WCET	285	1,452	1,600	low	high	med
huffbench	Compress	Scott Ladd	309	1,628	1,004	med	med	med
matmult	Matrix multiply	WCET	175	420	1,600	med	med	med
minver	Matrix inversion	WCET	188	1,076	168	high	low	med
nbody	N body problem	CLBG	167	708	656	med	low	high
nettle-aes	AES encryption	Nettle	1,018	2,874	10,546	med	high	low
nettle-sha256	Secure hash	Nettle	349	5,558	520	low	med	med
nsichneu	Petri net	WCET	2,676	15,036	8	med	high	low
picojpeg	JPEG	MiBench2	2,182	8,022	1,196	med	med	high
qrduino	QR code	GitHub	936	6,056	1,540	low	med	med
sglib	Generic C library	SGLIB	1,844	2,316	800	high	high	low
slre	Regular expr	SLRE	506	2,422	126	high	med	med
st	Statistics	WCET	116	880	64	med	low	high
statemate	State machine	C-LAB	1,301	3,686	64	high	high	low
ud	Matrix decomp.	WCET	95	702	0	med	low	high
wikisort	Merge sort	GitHub	866	4,208	3,268	med	med	med

Wally Embench benchmarking is driven by `benchmarks/embench/Makefile`. Embench measures both the speed and code size. Moreover, it is compiled in flavors that optimize for speed and for size. Hence, Embench makes four measurements. The overall Embench results are the geometric mean of the size or speed of all the Embench benchmarks [Fleming86].

Invoke `make` to build the code and `make run` to run the speed benchmark in Verilator on `rv32imc_zicsr`.

```
$ cd $WALLY/benchmarks/embench
$ make run # Compile and run Embench for rv32imac_zicsr
```

`make` compiles the benchmarks for speed and for size into the `src` subdirectories of `$WALLY/addins/embench-iot/{bd_sizeopt_size, bd_sizeopt_speed, bd_speedopt_size, bd_speedopt_speed}`. [Section 21.2.2](#) describes the compiler flags used for each version. In brief, the `sizeopt` builds use `-Os` optimization for size, while the `speedopt` builds use `-O2` optimization for speed. The `_size` versions omit the startup files and libraries to measure only the main code of each benchmark, while the `_speed` versions include the necessary startup and library files as well as instrumentation to log and report cycles elapsed and instructions retired.

`make run` measures the size of the `_size` versions, then runs a Verilator simulation of the `_speed` versions on the `rv32imc_zicsr` configuration. The cycle count and instructions retired are logged to `$WALLY/addins/embench-iot/logs`. It collects the results into four JSON files in `benchmarks/embench/wally{Size/Speed}Opt_{size/speed}.json`. The results for each program are normalized to the baseline size and speed for a Cortex-M4 processor running the ARMv7-M instruction set using the GCC 9.2 compiler. Higher speed and lower size numbers are better. The simulation generally takes more than an hour to run.

The Embench `main()` function in `embench-iot/support/main.c` warms up the caches and branch predictor by running a benchmark once using the `warm_caches()` function. Warmup removes compulsory cache misses and initializes the branch history table, giving a more realistic estimate of long-term program behavior. The benchmarks are small enough to fit in the caches, so the cache miss rate is negligible. `main()` calls `start_trigger()` to log the performance counter initial values, then runs the benchmark, then calls `stop_trigger()` to log the final values. Finally, it invokes `verify_benchmark()` to check that the benchmark produced the expected output. Each benchmark writes five words to `embench-iot/bd<case>_speed/src/<benchmark>/benchmark.sim.output: start_cycle, stop_cycle, start_inst, stop_inst` and a success code of 1.

[Table 21.4](#) summarizes the scores running Embench 1.0 on Wally compiled with GCC 13.2.0 targeting `rv32imc_zicsr`. The left half shows the `sizeopt` results (compiled for smallest size with `-Os`), and the right half shows the `speedopt` results (compiled for fastest speed with `-O2`). Each half lists the size and speed (normalized to the Cortex-M4), the execution time (`mtime` = number of cycles), instructions retired (`instret`), and the cycles per instruction (`CPI` = `mcycles/minstret`).

The `sizeopt` size is 1.05, indicating that the RV32IMC ISA uses 5% more code storage than ARMv7-M (primarily because it lacks bit

The Cortex-M4 is a 3-stage pipelined processor for embedded applications. It has the Thumb compressed instruction set, multiplication instructions, and AHB interfaces to code and data memory. It is a reasonable competitor to a pipelined implementation of RV32IMC.

The JSON files also report results for `md5sum`, `primecount`, and `tarfind`, which are part of Embench 2.0. They are omitted from the results in this section.

On Wally, `mtime` increments on every clock cycle, and is thus interchangeable with `mcycles`. If `mtime` ticks at a different rate, the reporting must be more careful.

Table 21.4 Wally Embench 1.0 rv32imc_zicsr speed and size scores and CPI.

Benchmark	sizeopt					speedopt				
	Size	Speed	mcycles	minstret	CPI	Size	Speed	mcycles	minstret	CPI
aha-mont64	0.95	0.80	5,032,119	4,584,519	1.10	1.52	0.83	4,823,306	4,531,256	1.06
crc32	0.75	1.05	3,834,416	3,834,036	1.00	0.70	1.05	3,831,561	3,831,155	1.00
cubic	1.46	0.40	9,912,360	7,071,082	1.40	1.50	0.39	9,954,370	7,064,452	1.41
edn	1.05	0.88	4,553,905	3,867,541	1.18	1.11	1.06	3,778,469	3,499,291	1.08
huffbench	1.43	1.32	3,112,970	2,560,252	1.22	1.51	1.54	2,680,047	2,450,480	1.09
matmult-int	0.85	1.01	3,945,944	3,907,004	1.01	1.03	1.11	3,599,313	3,183,118	1.13
minver	0.89	0.63	6,346,683	5,192,060	1.22	1.11	0.66	6,026,585	4,925,144	1.22
nbody	0.86	0.69	4,060,066	3,475,852	1.17	0.86	0.69	4,074,924	3,471,712	1.17
nettle-aes	1.25	0.82	4,921,050	4,587,607	1.07	1.42	0.86	4,692,919	4,406,503	1.06
nettle-sha256	1.63	0.96	4,166,818	3,910,230	1.07	1.77	0.95	4,192,951	3,992,431	1.05
nsichneu	1.35	1.16	3,451,900	2,001,634	1.72	1.51	0.85	4,711,593	2,236,753	2.11
picojpeg	1.15	0.82	4,940,182	4,617,539	1.07	1.68	1.00	4,027,577	3,786,353	1.06
qrduino	1.03	1.23	3,457,964	2,968,715	1.16	1.46	1.33	3,196,265	2,791,232	1.15
sglib-combined	1.05	1.13	3,509,488	2,744,895	1.28	1.17	1.23	3,231,252	2,527,994	1.28
slre	1.08	1.25	3,219,250	2,937,917	1.10	1.29	1.50	2,682,180	2,468,879	1.09
st	0.84	0.88	4,643,364	3,956,854	1.17	1.00	0.89	4,573,429	3,911,612	1.17
statemate	0.82	2.16	1,850,183	1,258,951	1.47	0.88	2.96	1,351,367	1,182,355	1.14
ud	0.97	0.85	4,678,093	3,709,827	1.26	1.15	0.92	4,352,969	3,347,720	1.30
wikisort	0.95	1.77	1,567,725	1,247,457	1.26	1.14	1.81	1,532,449	1,251,511	1.22
Geometric mean	1.05	0.98			1.20	1.22	1.04			1.18

Low size and high speed scores are good.

manipulation instructions). The `speedopt` speed is 1.04, which indicates that the Wally rv32gc configuration takes a mean 4% fewer cycles to execute the benchmarks than the Cortex-M4. This speed reflects both the RV32IMC ISA used by the compiler and the pipeline, memory system, and branch predictor used by each processor microarchitecture. Compared to the mean `sizeopt` results, the mean `speedopt` results show a 6% speed increase (from 0.98 to 1.04) at a 17% increase in code size (from 1.05 to 1.22).

Most of these programs have similar speed on RISC-V and ARM. However, notable slow RISC-V programs include `cubic`, `minver`, `nbody`, and `st`. `cubic` uses `long double` variables, which GCC treats as 64-bit on ARM but 128-bit on RISC-V, so RISC-V is naturally much slower [Gallmann21]. The other three also use floating-point arithmetic, which is less efficient in the RISC-V libraries. `statemate` appears fast on Wally with GCC 13.2.0 compared to the M4 with GCC 9.2 because of GCC improvements that were not in the ARM code generator until version 11. It is not yet clear why `huffbench` or `wikisort` are so much faster on Wally.

`nettle-sha256` is much larger on `rv32imac` because of the lack of bit manipulation instructions such as `rotate` and `reverse` used for hashing. `cubic` is larger because of the 128-bit long doubles.

Embench 1.0 discards the libraries from the size calculations. Unfortunately, it is not perfectly consistent; in particular, `start_trigger` and `stop_trigger` comprise 36 bytes on ARM and 4 bytes on RISC-V. This causes short benchmarks to have a size score greater than 1 on RISC-V even though the density of the main code is comparable to ARM. Future versions of Embench plan to include library code in the size, giving more realistic results.

The mean CPI is 1.20 for `sizeopt` and 1.18 for `speedopt`. These are somewhat higher than for CoreMark because the Embench benchmarks push the pipeline and branch predictor harder. `nsichneu` has an unusually high CPI because of the large number of use-after-load stalls as well as a substantial number of branches and branch mispredictions.

21.2.1 Porting Embench

`$WALLY/addins/embench-iot` contains the following major directories:

- ▶ `src`: Benchmark C source code
- ▶ `support`: Main routine and libraries to warm up caches and run and check benchmark
- ▶ `pylib`: Python wrappers to extract results from benchmark runs
- ▶ `config`: Startup and support files for logging results on different platforms
- ▶ `logs`: Transcripts of benchmark results

`embench-iot` also contains `build_all.py` to compile the C program and `benchmark_size.py` and `benchmark_speed.py` to run the benchmarks and extract results.

Embench 2.0 came out while this book was in production [Patterson25]. It drops `cubic`, `minver`, `nbody`, and `st` and adds `depthconv`, `md5sum`, `targind`, and `xgboost`.

Running Embench in simulation on Wally involves the `config/riscv32/boards/rv32wallyverilog` support files specific to Wally. These include:

- ▶ `boardsupport.c`: Dummy functions for board initialization and start/stop trigger
- ▶ `board.cfg`: Compiler and linker flags common to both speed and size tests
- ▶ `link.ld`: Linker file
- ▶ `startup/crt0.S`: Startup file including stack initialization, `write_tohost`, and start/stop triggers
- ▶ `startup/dummy.S`: Dummy triggers used to minimize code size for size tests

The only Embench file in the main Wally repository is the Makefile found in `$WALLY/benchmarks/embench`. This Makefile contains the commands to launch the Embench Python scripts, along with others needed for simulation (i.e., generating `.objdump`, `.addr`, and `.lab` files and launching the simulation). All other files, including those created specifically for porting Embench to Wally, can be found in `$WALLY/addins/embench-iot`.

Files related to porting Embench are used by these Python wrappers for two main functions: building/linking and decoding the results. The directory containing the building and linking files is `embench-iot/config/riscv32/boards/rv32wallyverilog/`. `board.cfg` contains compiler and linking flags used by all Wally Embench runs. `boardsupport.c` and `boardsupport.h` contain C code used to boot Wally, and `startup/crt0.S` contains the assembly code used for timing and recording to the memory location `begin_signature` (to be accessed and saved by the testbench at the end of each run). The linker file `link.ld` specifies the memory locations of the compiled ELF files, with code starting at `0x8000000`.

The Embench porting file needed to decode the results is `.sim.output`. As the simulation runs complete, the testbench reads the contents of `begin_signature` and saves them to `.sim.output` files in the build directory. The `benchmarking_speed.py` file then reads these values and calculates the Embench score. `embench-iot/pylib/run_wally.py` specifies how to read and decode these values. The `build_benchmark_cmd` function specifies how to read in the values, and the `decode_results` function decodes them and converts them to milliseconds elapsed, which is then used to calculate the Embench score.

21.2.2 Embench Flags

Table 21.5 lists the compiler flags used for all RV32 Wally (`rv32wallyverilog`) builds, defined in `embench-iot/config/riscv32/`

Table 21.5 Embench compiler flags in `board.cfg`.

Flag	Meaning
-c	Compiles source code without linking
-fdata-sections	Places each data item in its own section in the output file
-ffunction-sections	Places each function in its own section in the output file
-Wl,-gc-sections	Configures linker to remove unused code (garbage collector)
-march=rv32imac_zicsr	Compiles for the RV32IMAC_Zicsr ISA; CSRs needed for <code>mcycle</code> and <code>minstret</code> measurement
-mabi=ilp32	Ints, longs, and pointers are 32 bits
-T../../../../config/riscv32/boards/rv32wallyverilog/link.ld	Linker script path
-lm	Links math library

`boards/rv32wallyverilog/board.cfg`. These include targeting RV32IMAC and eliminating unused functions and data. Additional flags defined in the Makefile are used for `speedopt` and `sizeopt`, as shown below.

```
speedopt: -O2 -nostartfiles
sizeopt:  -Os -nostdlib -msave-restore
```

21.2.3 Instruction Set Impact

Table 21.6 shows the Wally Embench `sizeopt` scores compiled with RV32I, IM, IMC, IMCB, and GCB with GCC 13.2.0. Table 21.7 shows the `speedopt` scores. As more extensions are added, the compiler can select more optimized instructions to reduce overall code size. Program size decreases or stays the same when a new extension is added. Generally, performance increases as extensions are added, with a few exceptions.

The tables are generated with a script called `embench_arch_sweep.py` that repeats the make process for the five different ISA variants. It tabulates the results in `$WALLY/benchmarks/embench/run_<date>`.

```
$ cd $WALLY/benchmarks/embench
$ ./embench_arch_sweep.py
# Run many configs (all day) & tabulate results
```

The base instruction set greatly underperforms the reference Cortex-M4 with only about half the speed. However, with just the addition of

Table 21.6 Embench code size (sizeopt_size with -Os).

-march	rv32i_zicsr	rv32im_zicsr	rv32imc_zicsr	rv32imc_zba_zbb _zbc_zbs_zicsr	rv32gcb or (rv32imafdc_zba_zbb _zbc_zbs_zicsr)
aha-mont64	1.53	1.37	0.95	0.95	0.95
crc32	1.23	1.15	0.75	0.74	0.74
cubic	2.54	2.54	1.46	1.45	1.31
edn	1.71	1.45	1.05	1.03	1.03
huffbench	2.09	2.09	1.43	1.39	1.39
matmult-int	1.25	1.13	0.85	0.85	0.85
minver	1.36	1.34	0.89	0.87	0.75
nbody	1.38	1.38	0.86	0.85	0.56
nettle-aes	1.64	1.63	1.25	1.21	1.21
nettle-sha256	1.93	1.93	1.63	1.16	1.16
nsichneu	1.6	1.6	1.35	1.35	1.35
picojpeg	1.54	1.47	1.15	1.12	1.12
qrduino	1.42	1.34	1.03	1.02	1.02
sglib-combined	1.55	1.52	1.05	1.04	1.04
slre	1.47	1.47	1.08	1.07	1.07
st	1.41	1.38	0.84	0.82	0.6
statemate	0.95	0.95	0.82	0.82	0.82
ud	1.52	1.36	0.97	0.94	0.91
wikisort	1.43	1.41	0.95	0.94	0.93
Geometric mean	1.52	1.46	1.05	1.01	0.96

the M extension, Wally outperforms Cortex M4 by 4% on average, clearly showing how important multiply is to the workloads in the Embench benchmark.

The C extension reduces code size by 28% for Embench. It improves the performance of some applications but reduces the performance of others. This highlights the inherent microarchitectural tradeoff Wally

Table 21.7 Embench speed (speedopt_speed with -O2).

-march	rv32i_zicsr	rv32im_zicsr	rv32imc_zicsr	rv32imc_zba_zbb _zbc_zbs_zicsr	rv32gcb (or rv32imafdc_zba_zbb_ zbc_zbs_zicsr)
aha-mont64	0.36	0.85	0.83	0.83	0.83
crc32	0.7	1.05	1.05	1.1	1.1
cubic	0.13	0.39	0.39	0.39	0.53
edn	0.05	1.06	1.06	1.07	1.07
huffbench	1.56	1.56	1.54	1.65	1.65
matmult-int	0.12	1.11	1.11	1.1	1.1
minver	0.28	0.67	0.66	0.67	7.24
nbody	0.18	0.69	0.69	0.69	1,480
nettle-aes	0.84	0.88	0.86	1.05	1.05
nettle-sha256	0.97	0.97	0.95	1.33	1.34
nsichneu	0.7	0.68	0.85	0.83	0.85
picojpeg	0.87	1.02	1	1.06	1.06
qrduino	0.78	1.36	1.33	1.39	1.39
sglib-combined	1.24	1.27	1.23	1.3	1.3
slre	1.53	1.53	1.5	1.5	1.49
st	0.22	0.89	0.89	0.89	23.7
statemate	2.99	2.99	2.96	2.98	2.98
ud	0.57	0.97	0.92	0.92	1.38
wikisort	0.8	1.84	1.81	1.83	4.53
Geometric mean	0.52	1.04	1.04	1.09	2.39

made when implementing the IFU spill logic (see [Section 14.3.3](#)). Because compressed instructions are smaller, more will fit within a single cache line and on average should improve the instruction cache hit rate as this is equivalent to increasing the cache line length ([Fig. 10.7](#)). However, the cache SRAM implementation has one read/write port, so if a 4-byte instruction spills across cache line boundaries, the spill logic breaks the

request into two sequential reads, thus slowing down instruction fetch. The net speed is approximately equal with and without the C extension.

The bit manipulation extensions (Zba, Zbb, Zbc, Zbs) have a profound impact on `nettle-sha256`, which uses rotations extensively, and a small improvement on a few others (e.g., `crc32`, `huffbench`, `nettle-aes`, `picojpeg`, `qrduino`, `sglib-combined`) that twiddle bits. The mean benefits are 4% decreased size and 5% increased speed.

The floating-point extensions F and D have some of the greatest impact on the performance. `minver` uses floats. `nbody` and `st` use doubles. Using floating-point instructions rather than floating-point libraries also reduces code size. `ud` and `wikisort` also contain multiplications by 1.0 or 2.0 that lead to floating-point code even though there are no float or double variables.

`nbody` in particular executes only 1,025 instructions with the D extension but more than 3.1 million without it. It is not credible that floating-point should bring a 3000x reduction in instructions. On close examination, we see that the `rv32imc` version calls `bodies_energy` 100 times in a loop. The `rv32gc` version detects that the call to `bodies_energy` produces an invariant result and then adds that result up 100 times. Discarding this bad benchmark brings the geometric mean for `speedopt_speed` down to a more plausible 1.67.

21.2.4 Troubleshooting

If Embench is not compiling successfully, check the build logs in `addins/embench-iot/logs/build-<version>.log`.

The logic simulators log the start and stop cycle and instruction counts (`mtime` and `minstret`) into the first four lines of the signature file in `$WALLY/addins/embench-iot/bd_speed/src/<testname>` followed by a code of 1 for success or 3 for failure. The cycle count and instructions retired are also collated into `addins/embench-iot/logs/speed-<version>.log`.

If you need to debug a new version of the benchmarks, it is faster to simulate in Spike first:

```
$ make clean
$ make spike
```

Spike gives a speed of 1.23 (again, normalized to the Cortex-M4 speed). Recall that Spike executes one instruction per cycle (IPC).

21.3 PERFORMANCE VERIFICATION

The Wally tests described in earlier chapters focus on *functional verification*, to show that Wally runs programs correctly. Caches and branch

predictors can have subtle errors that produce correct results but are slower than they should be. Consider a defective cache that always misses or a branch predictor that always predicts not taken. The processor recovers and eventually produces the correct result, but runs more slowly than intended. *Performance verification* is used to compare the processor against a reference model and detect performance bugs. This section describes performance verification for the caches and branch predictor. The branch predictor's performance verification unearthed numerous Wally implementation errors.

Recall that caches and branch predictors initially have *compulsory* misses when they are empty, then see a lower miss rate when they have loaded the working set. Performance validation typically runs a test once to *warm up* the system, then a second time to take measurements. Each benchmark executes a function to indicate the end of the warmup and another function to indicate the end of the benchmark. The performance counters are sampled at both events, and the difference is recorded into the transcript. For CoreMark, these functions are `start_time()` and `stop_time()`. For Embench, these functions are `start_trigger()` and `end_trigger()`.

21.3.1 Cache Performance Verification

Wally's instruction and data caches are tested against a Python reference model in `$WALLY/bin/CacheSim.py`. The Verilog testbench optionally records a trace of cache accesses.

Enable this trace with the `--params "I_CACHE_ADDR_LOGGER=1'b1 D_CACHE_ADDR_LOGGER=1'b1"` flag to `wsim`. This creates a log of memory accesses in `sim/verilator/ICache.log` and `sim/verilator/DCache.log`. The log contains one line for each memory access that indicates the address, the type of access (Write, Read, Flush, or Invalidate) and the outcome (Hit, Miss, or Evict).

Run the cache simulation on the various regression test cases. It is expedient to use the existing tests but is not a particularly thorough way of exercising the caches. `arch64i` is the most useful of the test cases because it exercises loads, stores, and branches.

The logs of memory accesses are created in the subdirectory of the simulator used, i.e., in `sim/verilator/ICache.log` or `sim/questa/ICache.log`, etc.

```
$ regression-wally --cache
```

For each test case, `regression-wally --cache` runs a simulation on the `rv64gc` configuration to produce the `ICache.log` and `DCache.log` files. It then runs `CacheSim.py` on these logs. `CacheSim.py` has a simple model of an associative cache with a configurable number of lines and ways and address size. It predicts whether each access from the logs should hit, miss, or evict (evict also implies miss) and compares against the Wally behavior. It should report no mismatches on any test case. It may take a few minutes to run.

21.3.2 Branch Predictor Performance Verification

Wally's branch predictor, `bpred`, consists of four major modules: a branch direction predictor (branch/pattern history table: BHT or PHT), branch target buffer (BTB), return address stack (RAS), and instruction class (IC) predictor. The accuracy of the first two are verified against a C reference model running Embench. The RAS and IC predictor trend reasonably as a function of size but have not been checked against a reference model.

21.3.2.1 Branch Direction and Target Predictor Reference Model

To check branch performance, we run the CoreMark and Embench benchmarks for various sizes and types of branch predictors and compare the results against a C-based branch predictor simulator, called `sim_bp` [github.com/rosethompson/branch-predictor-simulator], which is also referred to as the CModel. `sim_bp` takes a text file containing one line for each branch instruction encountered during program execution. Each line contains the branch instruction address and T for taken or N for not taken. `sim_bp` also receives parameters that specify the size of the direction and target predictors. It simulates the behavior of a 2-bit or Gshare branch predictor that encounters this instruction trace and reports the direction and target prediction accuracies. `sim_bp` also reports the BTB accuracy. Build the simulator with:

`sim_bp` is forked from github.com/synxlin/branch-predictor-simulator, which appears not to be presently accepting pull requests. It modifies the Gshare hashing function to use `PC[1]`, as described in [Section 13.3.3.2](#), which is necessary for good performance with compressed instructions.

```
$ cd $WALLY/addins/branch-predictor-simulator/src
$ make
```

Use the `--params "BPRED_LOGGER=1\b1"` flag of `wsim` to make the testbench record each branch instruction address and direction into `$WALLY/sim/verilator/branch.log`. With this flag, the testbench's branch logger also logs the address of all control flow instructions into `$WALLY/sim/verilator/cfi.log`, for simulating BTB accuracy. The logger automatically inserts text to indicate the start of a new benchmark and when the application is training. Generate a branch trace for all the programs in Embench, if you haven't already done so in [Section 21.2](#), using the following commands. (This may take an hour to run.)

```
$ cd $WALLY/benchmarks/embench
$ make clean
$ make run # This automatically passes the flag to enable branch logging
```

Then use `sim_bp` to compute the expected (theoretical) branch predictor accuracy for a specified predictor type and size based on the Embench branch trace in `branch.log`.

```
$ sim_bp bimodal 10 10 1 $WALLY/sim/verilator/branch.log # 2-bit,          b=10, btb size=10
...
$ sim_bp gshare 10 10 10 1 $WALLY/sim/verilator/branch.log # Gshare, k=10, b=10, btb size=10
...
Final Branch Predictor Statistics:
a. Number of branches: 132600
b. Number of predictions from the branch predictor: 132577
c. Number of mispredictions from the branch predictor: 6034
d. Number of mispredictions from the BTB: 11
e. Misprediction Rate: 4.56 percent
```

`sim_bp` reports the BTB accuracy using the instruction trace; however, `branch.log` does not contain jumps or jump registers, so it does not accurately reflect Wally's BTB implementation. The testbench's logger records all branches, jumps, and jump registers in `cfi.log`. To simulate true BTB performance, replace `branch.log` with `cfi.log` in the above commands.

To compute theoretical branch predictor accuracy as a function of predictor type and size for the benchmarks in the Embench suite, the provided scripts split the log by benchmark, measure each benchmark's theoretical accuracy, and then take the mean. `SeparateBranch.sh` breaks the Embench `branch.log` and `cfi.log` files into per-benchmark log files in `sim/verilator/branch` and `sim/verilator/cfi`, respectively.

```
$ cd $WALLY/sim/verilator
$ SeparateBranch.sh branch.log
$ SeparateBranch.sh cfi.log
```

Next, the `CModelBranchAccuracy.sh` script sweeps `sim_bp` for $k = b = 6$ to 16 in steps of 2 for both 2-bit and Gshare branch predictors on each Embench benchmark. The script prints the geometric mean of the branch direction misprediction rate (in percent) across the benchmarks for each predictor size and type.

```
$ CModelBranchAccuracy.sh branch
bimodal6 9.98882688699562
bimodal8 8.40254311359969
bimodal10 7.29787062603488
bimodal12 6.84715794645923
bimodal14 5.68432926870082
bimodal16 5.68432926870082
gshare6 11.4615251720426
gshare8 8.52700507523446
gshare10 6.32125365841062
gshare12 4.55623085851726
gshare14 3.54065071082266
gshare16 1.9021813477034
```

The 2-bit branch predictor results are listed as `bimodal*` in this script's output.

Similarly, the `CModelBTBAccuracy.sh` script sweeps `sim-bp` for BTB size = 6 to 16 in steps of 2 for all control flow instructions in each Embench benchmark. It prints the geometric mean of the branch target misprediction rate across benchmarks for each BTB size. The CModel results are shown in Fig. 13.8.

```
$ CModelBTBAccuracy.sh cfi
```

21.3.2.2 Embench Branch Accuracy Sweeps

Calculating Wally's branch predictor accuracy for the direction, target, RAS, and class predictors requires simulating each Embench benchmark for different predictor types and sizes, then extracting the accuracies from the hardware performance monitor (HPM) counters.

The `regression-wally --branch` script runs a sweep of Embench benchmarks across branch predictor types and sizes and logs the performance counters. It uses the `wsim` parameter flag to set `PrintHPMCounters = 1`, so that the testbench records the HPM counters in the simulation transcript. Embench is a long simulation, so `regression-wally` launches multiple simulations in parallel, each running a copy of Embench with a different branch predictor type and size. It saves the Verilator transcripts to the `sim/verilator/logs` directory with names `bpred_<type>_<BHT/PHT size>_<RAS entries>_<BTB size>_<Class prediction enabled>_rv32gc_embench.log`. The `parseHPMC.py` script postprocesses these logs and tabulates and plots the branch predictor accuracy. The per-benchmark plots are called `barSegment0...4.svg`. Run performance verification (which may take half a day to run all simulations) using:

The performance verification effort initially detected discrepancies from theory, helping uncover numerous performance bugs in early Wally implementations. For example, we found that the Gshare implementation worked for basic loops and spaghetti code but did not recover well from pipeline flushes. Ultimately, we settled on an implementation that only updated the global history register (GHR) in the Writeback stage but forwards in-flight branch results (see [Section 13.3.3.1](#)).

```
$ cd $WALLY/sim
# Sweep and plot Embench results across direction prediction (BHT/PHT) size and type, BTB size,
# RAS size, and class prediction BTB size
$ regression-wally --branch
$ parseHPMC.py -g -i -d -s bp-results/branch-list.txt verilator BrDirResult.svg # Direction Fig. 13.6
$ parseHPMC.py -d bp-results/branch-list.txt verilator # Per benchmark Fig. 21.4
$ parseHPMC.py -g -i -t -s bp-results/btb-list.txt verilator BTBResult.svg # BTB Fig. 13.8
$ parseHPMC.py -g -i -r -s bp-results/ras-list.txt verilator RASResult.svg # RAS Fig. 21.2
$ parseHPMC.py -g -i -c -s bp-results/class-list.txt verilator BrClassResult.svg # Class Fig. 21.3
```

Figs. 13.6 and 13.8 compared the theoretical and Wally branch predictor direction and target prediction accuracies. The results agree well enough that Wally appears to be operating correctly. Figs. 21.2 and 21.3 show Wally's RAS and class prediction accuracies. Although we have no reference model for comparison, the predictor accuracies increase with size as expected and are better than 99% for reasonable sizes, indicating no major performance bugs.

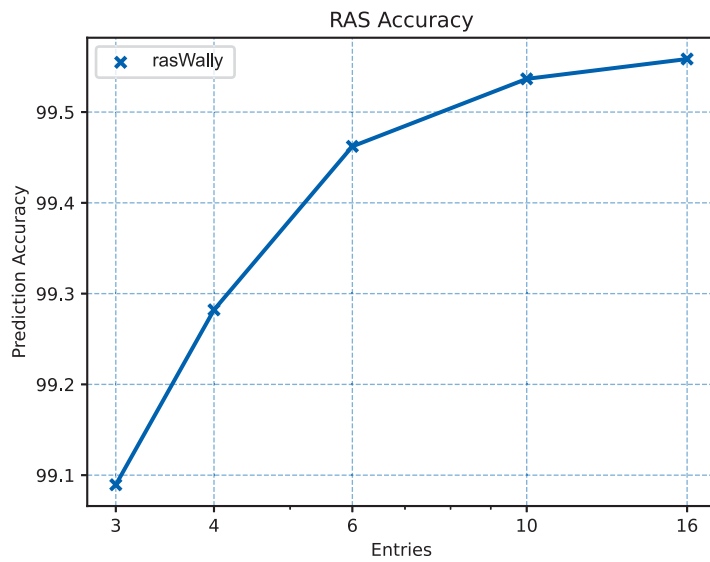


Fig. 21.2 RAS prediction accuracy sweep.

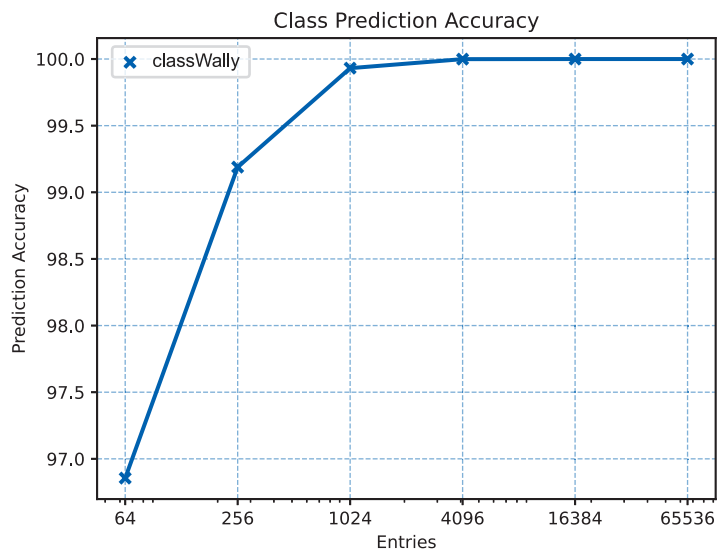


Fig. 21.3 Instruction class (IC) prediction accuracy sweep.

`parseHPMC.py` with the `-s` summary option generates a line graph of the branch direction prediction accuracy as shown in Fig. 13.6. It also includes the `sim_bp` accuracy for comparison. Without `-s`, it produces partially combined bar graphs. It groups speed optimized programs from the Embench suite as shown in Fig. 21.4. The `-i` option inverts misprediction to prediction accuracy. The `-g` option adds the `sim_bp` results to the plot or text output. With `-t`, it plots the branch target prediction accuracy as shown in Fig. 13.8. The `-r` option plots the RAS prediction accuracy in Fig. 21.2. The `-c` option plots the instruction class prediction accuracy in Fig. 21.3. The first positional argument (i.e., `bp-results/branch-list.txt`) reads a file with the list of transcripts and their branch predictor type and parameterizations. The first column is the transcript file, the second column is the name of the branch predictor, and the remaining columns are the predictor's parameters. The second positional argument indicates which simulator was used, and therefore which `sim` directory to search for transcripts. Finally, the third positional argument (i.e., `BrDirResult.svg`) indicates the output file.

Notice that `crc32` has very low misprediction rates (see Fig. 21.4).

Fig. 21.4 shows detailed branch predictor results for the different types and sizes of predictors for different Embench benchmarks compiled with `speedopt`. Some benchmarks such as `crc32` are highly predictable and have almost no mispredictions. The Gshare accuracy generally improves with the GHR length. Many applications such as `minver` and `ud` have correlated branches and benefit strongly from longer GHRs. For most benchmarks, the 2-bit predictor captures all branches with $b = 6$ (number of BHT entries = 2^b), so increasing predictor size only marginally improves accuracy. Larger applications benefit from a bigger predictor.

21.3.2.3 CoreMark Branch Accuracy

CoreMark running on Wally `rv64gc` reports a branch misprediction rate of $66,966/591,846 = 11\%$ for Gshare $k = b = 10$, as described in Section 21.1. The 16-entry RAS has a misprediction rate of $75/15,096 = 0.5\%$. The 1,024-entry BTB has a target misprediction rate of $4,525/(635,493 + 31,423) = 0.7\%$ and an instruction class misprediction rate of $14,604/3,448,364 = 0.4\%$

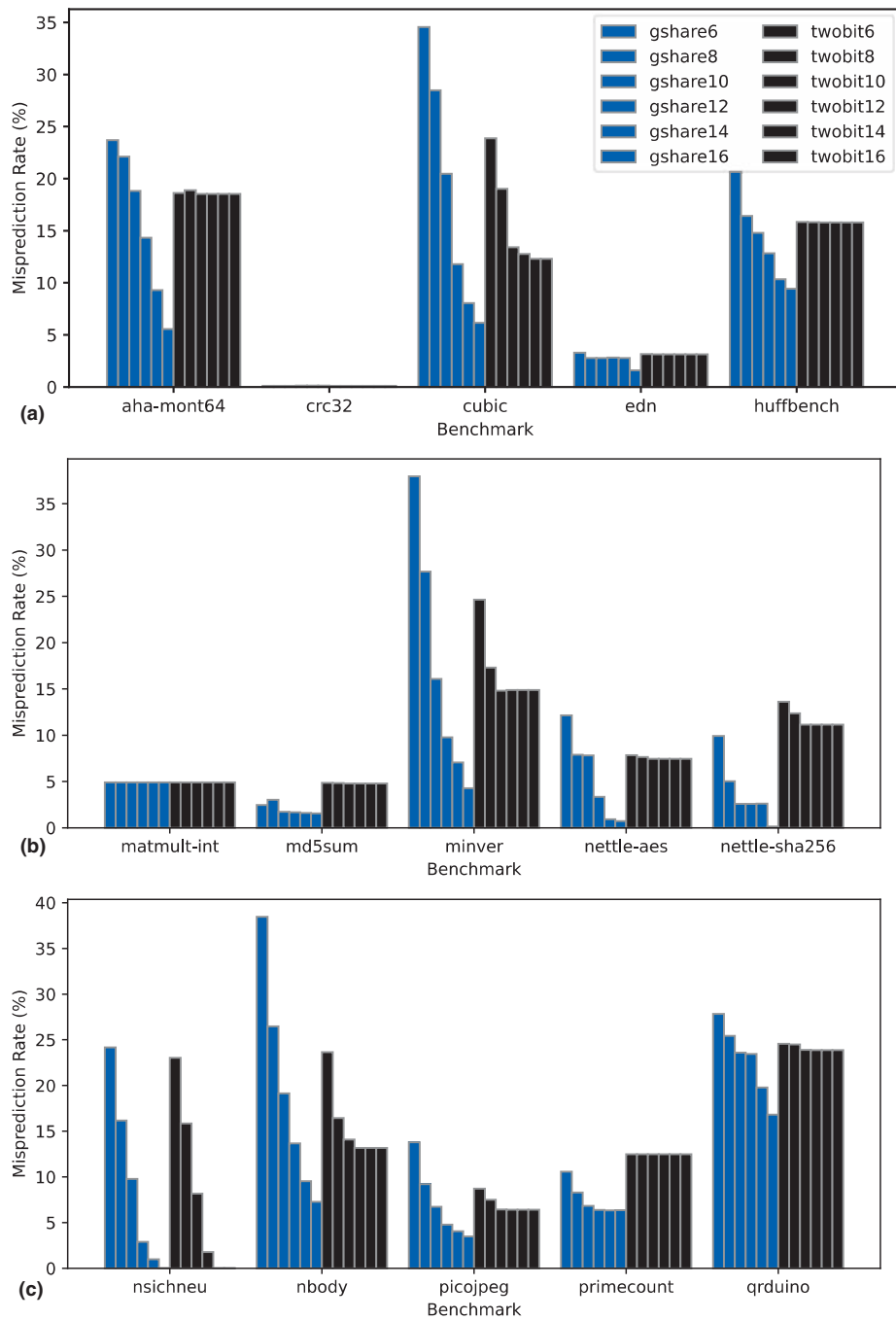


Fig. 21.4 Wally branch direction predictor accuracies for Embench.

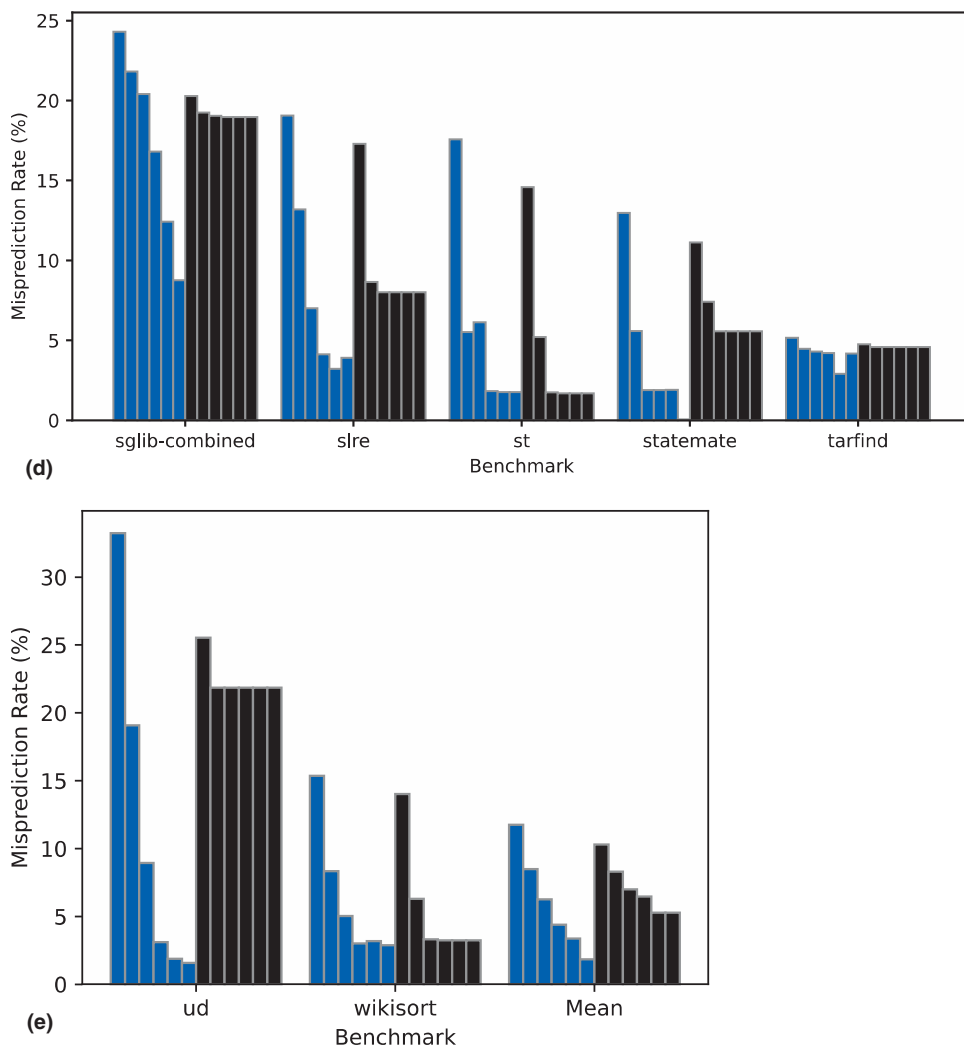


Fig. 21.4 Wally branch direction predictor accuracies for Embench. Cont'd

21.4 SUMMARY

This chapter has described how to compile and run the CoreMark and Embench benchmark suites on Wally. It also described performance verification tests showing that the cache and branch predictor perform as expected. The benchmarks show that Wally is comparable in IPC performance to other single-issue pipelined RISC-V processors and to the ARM Cortex-M4. They also show that RV32GC and ARMv7 code density are similar.

Linux

With contributions from Noah Boorstin, Ben Bracker, and Kaveh Pezeshki

- 22.1 Principles
- 22.2 Buildroot Linux on Wally
- 22.3 Booting Linux
- 22.4 Summary

Booting a Linux operating system (OS), in addition to being practical, is a good system test because it shows that all major subsystems of the processor work together, including the basic core, control and status registers (CSRs), exceptions, memory management, and peripherals. Linux comes in many distributions, and our application needs an embedded version that can start up from a small software image in memory, with only our basic peripherals and no disk, monitor, ethernet, or nonphysical “virtual peripherals” common in software simulations. Wally has been tested using the free and popular Buildroot Linux generator.

While developing Wally, we have run Buildroot Linux on three platforms:

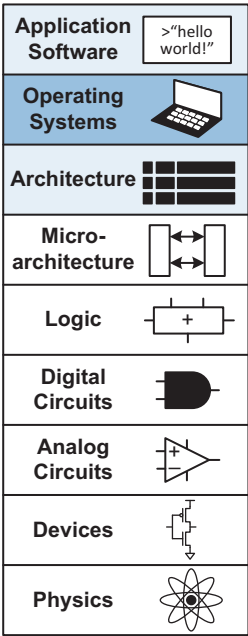
- ▶ The generic software RISC-V emulator: Quick Emulator (QEMU)
- ▶ Verilog simulation of the Wally SoC
- ▶ An FPGA implementation of the Wally SoC

This chapter demonstrates running Buildroot Linux in simulation on the QEMU and Verilog simulation platforms, and [Chapter 23](#) shows how to run it on an FPGA implementation of Wally.

22.1 PRINCIPLES

An OS has a variety of functions:

- ▶ Boot the computer when it turns on
- ▶ Protect hardware and other processes from faulty or malicious processes



- ▶ Provide resources to processes:
 - Process and thread management and scheduling
 - Memory management
 - File system management
 - Peripheral device drivers
 - Security
 - User interface

Contemporary operating systems for small computers can be classified as either a *real-time operating system* (RTOS) or a *general-purpose OS*. Options for embedded systems include no OS (“bare metal”), an RTOS, or embedded Linux.

Many RTOSes are on the market, such as FreeRTOS and VxWorks. They are generally invisible to the end user and are little known outside the developer community. They typically run on embedded systems, are designed to respond to inputs in a short and bounded amount of time, and are often optimized to run efficiently on inexpensive microcontrollers with small amounts of memory. Real-time operating systems typically do not support virtual memory because a page fault causes a long and unpredictable delay and because the system usually lacks secondary storage and enough memory to require complex management.

General-purpose operating systems, such as Linux, Windows, MacOS, and Android, are more widely known and visible to the user. Some run on personal computers (PCs) and mobile devices, but they are also used on embedded systems. Linux has become popular for complex embedded systems, such as televisions and printers, because it provides many powerful functions such as networking and scheduling, runs on most processors with memory management units (MMUs), has a wide set of device drivers, is free and open source, and has a large community of experts. Linux also has a real-time OS flavor called Preempt RT that guarantees a response within a bounded time. Linux has evolved to encompass more than 28 million lines of source code with contributions from more than 20,000 developers. As of 2025, about 46,000 lines of code are specific to supporting RISC-V.

Embedded Linux must be tailored to the particular system, including the processor architecture, peripherals, memory map, and software packages. *Open-source build systems*, also simply called *build systems*, automate this process. They produce a bootloader, kernel, and root file system. They often involve a cross-compiler, so they could build on an x86 PC or server but compile the OS to run on RISC-V or ARM. Contemporary build systems include Buildroot, Yocto, EmbToolkit, OpenEmbedded, OpnWrt, PTXdist, and Tizen. This section focuses on issues common to any Linux system, and [Section 22.2](#) examines a Buildroot Linux system running on Wally.

22.1.1 Embedded Linux Hardware Requirements

Embedded Linux runs on most 32- and 64-bit processor architectures with MMUs. It is easiest to configure with at least 16 MiB of RAM and 8 MiB of Flash memory storage, although it can be squeezed into smaller systems, and versions with many packages or lots of data need more storage. Embedded Linux usually needs a terminal interface, commonly RS-232 serial. It also needs a way to load software, such as an SD card or a USB, serial, or JTAG port.

22.1.2 Bootloader

When a processor first powers on or resets, it begins in the most privileged mode and sets the program counter to some address called the *reset vector*. The system must have a small ROM or Flash memory at this address containing *zero stage bootloader* (ZSBL) code to start the boot process. The DRAM controller is not yet initialized, so the boot loader initially has access only to a small on-chip SRAM and the boot ROM. Booting is typically divided into several stages, each of which brings up more capabilities for the system and transfers code for the next phase from nonvolatile storage into RAM until the boot loader can load the Linux kernel into DRAM and transfer control to it [Vasquez21]. The details are platform specific. Common bootloaders include GRUB for PCs and U-Boot for embedded systems.

The term *bootloader* gets its name from the expression “pulling yourself up by your bootstraps,” or humorously lifting yourself up into the air by pulling up on your own boots. The process of starting to run an OS without having the capabilities of an OS to run code is a similar challenge. The expression is now shortened to “booting” a computer.

22.1.3 Device Tree

During the boot process, the bootloader or OS must identify what hardware devices are in the system. For example, it needs to determine how much RAM is available and what nonvolatile storage device contains the kernel and filesystem. Later, it needs to determine what peripherals are available, such as a serial port to print to a console or a timer to generate interrupts. It needs to identify where these peripherals are in the memory map and what version of the hardware exists so the OS can use the corresponding device driver.

Historically, this information was compiled into the kernel, such that changing the hardware could require recompiling the OS. In the 1980s, Sun Microsystems defined the Open Firmware specification including a *device tree*, which is a data structure that describes the hardware configuration, including:

- ▶ Number and type of CPU cores
- ▶ Amount of DRAM
- ▶ Types and versions of peripherals, such as serial ports, timers, nonvolatile storage devices, and interrupt controllers

- ▶ Memory map with addresses for the memory and peripherals
- ▶ Mapping of peripherals to interrupt lines

Device trees are now used by almost all Linux platforms. They are described by human-readable text files with a `.dts` suffix. A device tree compiler converts them to a more compact binary format called a *device tree blob* or *flattened device tree (FDT)* with a `.dtb` suffix. Typically, the bootloader loads the FDT and passes it to the kernel.

Linus Torvalds maintains the *mainline* Linux kernel and releases new kernels every 9–10 weeks. Contributors develop new features that are integrated into release candidate builds and tested before becoming part of the mainline kernel. A *longterm maintenance* kernel is released about once a year and just receives critical bug fixes rather than constant new features. Linux distributions such as Ubuntu and Red Hat integrate the kernel with the rest of the OS (e.g., the U-mode shell and graphical user interface) and release periodic distributions and unscheduled bug fixes.

22.1.4 Kernel

The *kernel* is the portion of the OS software that operates in supervisor mode, providing an interface between the hardware and user processes. It is responsible for managing memory, processes, device drivers, and system calls.

One of the principles of Linux is that a common kernel image can run on a wide variety of hardware with different numbers and types of cores and peripherals, without requiring recompilation. The kernel calls an underlying firmware layer that provides a standardized interface to platform-dependent tasks such as starting and stopping cores. It uses the device tree to define the set of peripherals. It includes device drivers as kernel modules that can be loaded from the `/dev` directory in the file system and executed with the same privileges as the OS.

The kernel is often kept in nonvolatile storage in compressed format. After being loaded, it decompresses itself, enables virtual memory, and calls `start_kernel()`, which in turn initializes device drivers, enables interrupts, mounts a root file system, and then starts the idle process, scheduler, and init process.

The kernel may exist in various file formats in the `/boot` directory:

- ▶ `vmlinux`: The kernel is in a statically linked Executable and Linkable Format (ELF) file; suitable for disassembling or debugging
- ▶ `vmlinuz`: `vmlinux` compressed with `zlib`
- ▶ `Image`: `vmlinux` in binary format, with ELF headers and symbol table removed, suitable for booting directly
- ▶ `zImage`: `Image` compressed with `zlib`

22.1.5 Root Filesystem

The Linux kernel loads a *root filesystem* into RAM containing a minimal set of nonkernel programs necessary to boot. The filesystem is called `rootfs` or `initramfs`. It is kept in a `.cpio` (Copy In and Out) format archive. The archive may be compressed with `gzip` to save storage space and time to

read in from the storage device, so the kernel may uncompress the file system when loading it into memory.

Core Linux utilities such as `ls`, `cp`, and `init` collectively take a fair amount of disk space. To reduce the size of the root filesystem, small embedded Linux systems often use a program called BusyBox that performs most of the basic functionality of these utilities while being much smaller (e.g., ~6 MB) by leaving out unnecessary features and sharing code across utilities. Each of the regular utilities in `/bin` is actually a symbolic link to `/bin/busybox`, which interprets the command line and performs the functions.

A fully featured Linux system will use the drivers and utilities in `rootfs` to read a larger file system from disk later in the boot process, replacing `rootfs`.

22.1.6 Init

The first U-mode process is called `init` (short for initialization). It has a process ID of 1 and is the parent of all other Linux processes. It is called a *daemon* process, which runs in the background independent of the control of interactive users. The exact steps taken by `init` vary from system to system and continue to evolve. The Busybox `init` program reads a configuration file called `/etc/inittab` that contains a list of programs to run at startup time. These programs include daemons that provide services such as `nfsd` (Network File System Server), `sshd` (Secure Shell Server), `httpd` (Hypertext Transfer Protocol Server), and `lpd` (Printer Server). `systemd` is a more modern `init` daemon that starts processes in parallel to reduce boot time.

Daemons get their name from *Maxwell's daemon*, an imaginary being in thermodynamics that constantly sorts molecules in the background. *daemon* is pronounced “dee-mon” not “day-mon.”

22.1.7 Shell

`init` typically then launches a login *shell* with job control so that the user can log in, issue commands, and use `<Ctrl-c>` to terminate a running program. Linux offers various shells, with `bash` currently being the default on most distributions. `ash` is a lightweight version of `bash` provided by Busybox.

22.2 BUILDROOT LINUX ON WALLY

Although it is possible to build your own Linux system manually, it is faster and more maintainable to use an automated system to build these from source code in a configurable way. The Buildroot and Yocto build systems are among the most popular [Vasquez21]. We use Buildroot, which supports RISC-V and produces the OpenSBI bootloader/firmware, Linux kernel, and root filesystem for embedded Linux. This section applies the general principles from [Section 22.1](#) to the specific case of booting Buildroot Linux on RISC-V for Wally, including the bootloaders, device

tree, kernel, and root file system. [Section 22.2.1](#) gives directions for configuring, building, and disassembling Buildroot in the `$RISCV` directory using QEMU. The remainder of this section then examines the detailed process of booting Linux on Wally. [Section 22.3](#) gives detailed instructions for running Buildroot and booting Linux in QEMU and Verilog simulation. [Section 23.2.2.4](#) illustrates booting Linux on an FPGA.

This section first defines where all the source, binary, and disassembled files can be found and where they are loaded into physical and virtual memory. Then it summarizes key steps in the boot process. Finally, it explores the ZSBL, OpenSBI, kernel, file system, and device tree in more depth.

Most of the examples in this chapter were tested on Buildroot 2023.05.3-dirty with Linux 6.6 and OpenSBI 1.3. However, at the time of publication, Wally now uses Buildroot 2024.11 with Linux 6.12.8 and OpenSBI 1.6 and may continue to update.

22.2.1 Building Buildroot Linux

To download, cross-compile, disassemble, generate memory files for, and install Buildroot Linux for RISC-V into `$RISCV/buildroot`, use the `$WALLY/linux/Makefile`:

```
$ cd $WALLY/linux      # Change to your path to the repository
$ make                 # Clone, build, disassemble Linux and install in $RISCV
```

By default, `$RISCV = /opt/riscv`

Compilation takes approximately 10 minutes. If necessary, you will be prompted for your password for sudo permissions to install Buildroot in `$RISCV`. The `$WALLY/bin/wally-tool-chain-install.sh` script will automatically make Buildroot while installing everything else. [Section 22.2.2.4](#) describes how to generate a trace of the instructions executed during the Linux boot.

22.2.2 Source, Binary, and Disassembled Files

[Table 22.1](#) defines the directories where all key files are placed.

Table 22.1 Linux file locations.

Files	Path
Source code	<code>\$RISCV/buildroot/output/build</code>
Binary files and device tree	<code>\$RISCV/buildroot/output/images</code>
Disassembled files	<code>\$RISCV/buildroot/output/images/disassembly</code>
Memory files	<code>\$RISCV/linux-testvectors</code>

Table 22.2 Linux boot components.

Component	Source Code Subdirectory	Binary File	Disassembly
Zero stage bootloader	Code Example 22.2	<code>\$RISCV/linux-testvectors/ bootmem.bin</code>	N/A
OpenSBI firmware	<code>opensbi-1.6</code>	<code>fw_jump.bin</code>	<code>fw_jump.objdump</code>
Linux kernel	<code>linux-6.12.8</code>	Image	<code>vmlinux.objdump</code>
Root filesystem	N/A	<code>rootfs.cpio</code>	uncompressed into rootfs
Device tree	<code>\$WALLY/linux/devicetree/ wally-virt.dts</code>	<code>wally-virt.dtb</code>	N/A
Init	<code>busybox-1.36.1</code>	<code>\$ROOTFS/sbin/init -> /bin/busybox in rootfs</code>	<code>busybox.objdump</code>

[Table 22.2](#) summarizes the components of the boot process. Each is discussed briefly here and in more detail in subsequent subsections. The ZSBL is a few bytes in ROM at address 0x1000. In a real system, it would load more code from nonvolatile storage, but in this system, it just jumps to the OpenSBI firmware already in memory. This firmware contains the rest of the bootloader, which initializes the hardware (informed by the device tree) and M-mode traps, then jumps to the Linux kernel. The kernel enters S-mode, turns on virtual memory, and initializes services. The kernel then loads `init` from the root file system on the RAM disk and starts it as the first U-mode process.

22.2.2.1 Binary Files

The ZSBL is not stored in a file but is placed in memory by a ROM or QEMU. It is shown in [Section 22.2.5](#).

Building Buildroot produces the firmware and Linux kernel in both ELF format (`fw_jump.elf/vmlinux`) and raw binary format (`fw_jump.bin/Image`). Whereas the ELF versions include headers and debug information suitable for disassembly, the binary versions are loaded directly into memory by QEMU.

The root file system is stored as a compressed binary file, `rootfs.cpio`, which is also loaded directly into memory by QEMU. Compression speeds up reading the file from a slow flash memory device at the cost of the CPU operations needed to decompress the file.

Wally's device tree `wally-virt.dts` is compiled into a flattened device tree blob `wally-virt.dtb` and is also preloaded into memory.

22.2.2.2 Disassembled Files

After running `make disassemble`, `$RISCV/buildroot/output/images/disassembly` contains the following:

- ▶ `fw_jump.objdump`: OpenSBI firmware
- ▶ `vmlinux.objdump`: Linux kernel
- ▶ `busybox.objdump`: BusyBox
- ▶ `rootfs`: Uncompressed root file system (see [Section 22.2.8](#))

The flattened device tree blob is not disassembled because it is not an executable file and the raw `.dts` file is available.

22.2.2.3 Memory Files

When simulating Linux on Wally, `testbench.sv` preloads the boot ROM at `0x1000` and uncore RAM at `0x80000000` with binary files `bootmem.bin` and `ram.bin`, respectively. `bootmem.bin` is the trivial ZSBL described in [Section 22.2.5](#), and `ram.bin` is a 256-MiB memory image containing the firmware, kernel, and root file system.

Running `make dumptvs` uses QEMU to generate these memory images into `$RISCV/linux-testvectors`. The top-level Linux `make` command runs this automatically.

22.2.2.4 Boot Trace

To understand the behavior of the Linux boot, it is helpful to have a trace showing the sequence of instructions executed from power-on reset until Linux is waiting at the command prompt. Run a Questa simulation of the Linux boot in lock step with ImperasDV and record the trace into `$WALLY/sim/questa/logs/boottrace.log` using:

```
$ wsim buildroot buildroot --lockstep --define "+define+STD_LOG=1"
```

`boottrace.log` takes almost a day to generate and is about 51 GiB. Instructions causing exceptions are not listed because they don't retire.

[Code Example 22.1](#) shows the first instructions in the trace, starting with the bootloader at `0x1000`, then jumping into OpenSBI at `0x80000000`, and then to `main` at `0x80000578`.

Code Example 22.1 BEGINNING OF `boottrace.log`

#	PC	instr	rd	mcycle	minstret
00000000000001000,	00000297,	AUIPC	rf[5] = 0000000000001000		
00000000000001004,	02828613,	ADDI	rf[12] = 0000000000001028	csr[b00] = 000000000000000f	csr[b02] = 0000000000000002
00000000000001008,	f1402573,	CSRRS	rf[10] = 0000000000000000	csr[b00] = 000000000000001a	csr[b02] = 0000000000000003
0000000000000100c,	0202b583,	LD	rf[11] = 000000008fe00000	csr[b00] = 000000000000001b	csr[b02] = 0000000000000004
00000000000001010,	0182b283,	LD	rf[5] = 0000000080000000	csr[b00] = 000000000000001c	csr[b02] = 0000000000000005
00000000000001014,	00028067,	JALR	rf[0] = 0000000000000000	csr[b00] = 0000000000000028	csr[b02] = 0000000000000006
00000000080000000,	00050433,	ADD	rf[8] = 0000000000000000	csr[b00] = 000000000000002b	csr[b02] = 0000000000000007
00000000080000004,	000584b3,	ADD	rf[9] = 000000008fe00000	csr[b00] = 000000000000002c	csr[b02] = 0000000000000008
00000000080000008,	00060933,	ADD	rf[18] = 0000000000001028	csr[b00] = 0000000000000037	csr[b02] = 0000000000000009
0000000008000000c,	56c000ef,	JAL	rf[1] = 0000000080000010	csr[b00] = 0000000000000038	csr[b02] = 000000000000000a
000000000800000578,	8082557d,	C.LI	rf[10] = ffffffff	csr[b00] = 000000000000003b	csr[b02] = 000000000000000b

22.2.3 Linux Memory Map

Wally uses the memory map from Table A.2, including a small ROM at 0x1000 for the ZSBL, but expands the RAM to 256 MiB at 0x80000000-8FFFFFFF so the kernel, root filesystem, and running processes fit. Table 22.3 gives the Linux address space layout. OpenSBI expects the Linux kernel to begin at address 0x80200000. The root file system's location is determined by Buildroot at compile time. Finally, the device tree can be placed anywhere in memory as long as it does not overlap with other memory. It is typically placed 0x100000 to 0x1000000 bytes below the top of memory to avoid the call stack; the ZSBL loads the address of the device tree into register `a2`. In QEMU emulation or Verilog simulation, the emulator or testbench places the code, root file system, and device tree into memory before the boot process begins, and then a trivial ZSBL jumps to the firmware, as described in Section 22.2.5. On an FPGA, the ZSBL reads these files (code, root file system, and device tree) from a mass storage device such as an SD flash memory card, as described in Section 23.2.1.

Observe that the kernel is at physical address 0x80200000 even though it is compiled assuming a virtual address of 0xFFFFFFFF80000000.

Table 22.3 Linux address space layout.

Component	Physical Address	Binary Size
Zero stage bootloader (ZSBL)	0x1000	72 B
OpenSBI firmware	0x80000000	268 KiB
Linux kernel	0x80200000	83 MiB
Root filesystem	0x88200000	71 MiB
Device tree	0x8FE00000	2 KiB

Hence, the PC will be different running the kernel before and after virtual memory is turned on.

The kernel runs in S-mode and is placed in virtual memory at a high address with all leading 1s (0xFFFFFFFF80000000). When the kernel needs to access bare hardware, it calls the OpenSBI supervisor binary interface [github.com/riscv-non-isa/riscv-sbi-doc/blob/master/riscv-sbi.adoc], which runs in M-mode and gives a standardized supervisor execution environment for the OS to access the platform-dependent raw hardware, such as changing the timer; writing a character to a console; or starting, stopping, or synchronizing harts. The SBI is also called *firmware* and runs in M-mode starting at physical memory address 0x80000000. Linux programs run in U-mode and are sometimes loaded into virtual memory at low addresses.

22.2.4 Linux Boot Overview

Linux boot involves executing about half a billion instructions, although about 80% of them are for decompressing the root filesystem. [Table 22.4](#) summarizes the major events during the boot process in QEMU using Buildroot 2023.05 and Linux kernel 6.6. The disassembly column lists the file containing the disassembled code for context.

22.2.5 Zero Stage Bootloader

A RISC-V processor boots in M-mode. Although the RISC-V reset vector is nominally platform dependent, Spike, QEMU, the SiFive FU540, and Wally start the PC at 0x1000 and execute a short segment of code that jumps to another address (typically 0x80000000).

QEMU (`riscv_setup_rom_reset_vec` in `hw/riscv/boot.c`) places the ZSBL from [Code Example 22.2](#) in memory at 0x1000, and the Wally test-bench loads the same code from `bootmem.bin` for compatibility. This bootloader sets `a0` to the `mhartid`, `a1` to the address of the device tree, `a2` to a data structure that includes the Linux kernel address and then jumps to the OpenSBI firmware at 0x80000000.

[Chapter 23](#) describes a more realistic ZSBL, which is used in an FPGA implementation of Wally.

Code Example 22.2 ZERO STAGE BOOTLOADER (ZSBL)

```
00001000: 00000297      auipc t0, 0          # t0 = 0x00001000
00001004: 02828613      addi a2, t0, 0x28    # a2 = 0x00001028
00001008: f1402573      csrr a0, mhartid    # a0 = mhartid
0000100c: 0202b583      ld a1, 32(t0)        # a1 = 87000000 - device tree address
00001010: 0182b283      ld t0, 24(t0)        # t0 = 80000000 - start of firmware
00001014: 00028067      jr t0               # Jump to firmware
00001018: 00000000      # Firmware start address
00001020: 00000000      # Flattened device tree load address
00001028: 000000004942534f # a2 points to this 8 dword data structure
00001030: 0000000000000002
00001038: 0000000008020000
00001040: 0000000000000001
```

Table 22.4 Boot milestones from Linux 6.6.

Instructions Count	Address	Instruction	Source	Disassembly	Description
1	1000	auipc t0, 0		bootmem.txt	Reset vector: start of bootloader
6	1014	jr t0		bootmem.txt	Jump to SBI firmware
189,190	800003d8	csrw mtvec, a4	firmware/fw_base.S _start_warm	fw_jump.objdump	Enable machine trap handler called _trap_handler
875,039	800088a6	<mstatus_init>	sbi/sbihart.c mstatus_init()	vmlinux.objdump	Enable features: FPU, counters, CMO, Svpbmt
875,488	80008a44	csrw mideleg, 222	lib/sbi/sbi_hart.c delegate_traps()	fw_jump.objdump	Delegate all supervisor interrupts, page faults, U-mode ecall, and ebreak to supervisor
1,286,518	800008d0	jal <sbi_printf>	lib/sbi/sbi_init.c sbi_boot_print_banner()	fw_jump.objdump	Print SBI boot message
5,041,679	8000aa8e	mret	lib/spi/sbi_hart.c sbi_hart_switch_mode()	fw_jump.objdump	Enter S-mode and start kernel at 80200000
5,187,363	80201044	csrw satp, a0	arch/riscv/kernel/ head.S relocate_enable_mmu	vmlinux.objdump virtual address ffffffff80001044	Turn on virtual memory, causing page fault that enters kernel at ffffffff80001048
5,187,374	ffffffff8000106c	csrrw stvec, a0	arch/riscv/kernel/ head.S relocate_enable_mmu	vmlinux.objdump	Enable supervisor trap handler to handle_exception
12,344,404	ffffffff80400b0a	csrrsi sstatus, 2	init/main.c start_kernel()	vmlinux.objdump	Turn on S-mode interrupts
12,416,620	ffffffff80247f5c	<riscv_timer_interrupt>	drivers/clocksource/ timer-riscv.c riscv_timer_interrupt()	vmlinux.objdump	First timer interrupt

Continued

Table 22.4 Boot milestones from Linux 6.6. Cont'd

Instructions Count	Address	Instruction	Source	Disassembly	Description
20,500,509	ffffff804023c8	<populate_rootfs>	init/initramfs.c populate_rootfs()	vmlinux.objdump	Populate rootfs
424,172,631	ffffff80270f94	<run_init_process>	init/main.c run_init_process()	vmlinux.objdump	Start init
424,516,404	ffffff800031c8	sret		vmlinux.objdump	Enter U-mode program at 00007fff9ca04c3c
492,486,904	65254	<print_login_prompt>	libbb/login.c print_login_prompt()	busybox.objdump	Print login prompt
492,515,478	ffffff800033da	wfi	kernel/sched/idle.c do_idle()	vmlinux.objdump	Idle loop

22.2.6 OpenSBI Firmware

Like the ZSBL, the OpenSBI firmware also boots in M-mode. It is responsible for initializing the hardware.

This firmware supervisor binary interface (SBI) also provides a level of abstraction between the kernel (e.g., calls to `console_putc()` to write a character to the console) and the hardware (e.g., the underlying UART registers) so that the kernel can run on many different hardware *platforms* without modification. Thus, the firmware can be configured with the appropriate platform, such as the SiFive FU540 SoC or a CVA6 on an FPGA. Wally is configured to use a generic platform that examines the flattened device tree to determine which peripherals are available. See `$WALLY/linux/br2-external-tree/configs/wally_defconfig BR2_TARGET_OPENSBI_PLAT="generic"`.

By default, OpenSBI assigns all the harts and memory to a *root domain*. The firmware can optionally partition the system into multiple *domains* specified in the device tree such that the harts and memory associated with each domain are isolated from each other via physical memory protection (PMP). These domains are a hardware foundation for features such as trusted execution environments (TEEs), secure operating systems, and root-of-trust [static.sched.com/hosted_files/riscvsecurityforum2021/84/OpenSBI_domain_support_v4.pdf].

Each hart has scratch memory space that contains firmware data structures. The `mscratch` register points to this memory. The `sbi_scratch` structure includes pointers to the firmware, kernel, and `sbi_platform` structure. The `sbi_platform` structure contains the name and version number of the platform, the number of harts, and a pointer to the `sbi_platform_operations` structure. The `sbi_platform_operations` structure includes pointers to platform-specific functions for initializing and using the timer, console (UART), interprocessor communication software interrupts, and external interrupts from the IRQ (interrupt request) chip.

The initial assembly language bootloader code is at `_start` in `firmware/fw_base.S`. All RISC-V harts power up in random order, but the initial firmware boot occurs on only one hart. Thus, one of the harts is selected through a lottery (accessing shared memory) to be the boot hart, while the others sleep. Assembly language boot steps include the following:

- ▶ Initialize registers and BSS memory to zero.
- ▶ Set up temporary stack and trap handler to hang if boot fails.
- ▶ Read platform information from the flattened device tree.
- ▶ Point `mscratch` to initialized scratch space with per-hart firmware data.
- ▶ Wake up the other harts by writing a flag to a memory location.

- ▶ Disable and clear interrupts.
- ▶ Set up the stack pointer (`sp`) and `mtvec` to point to the trap handler.

The bootloader then calls `sbi_init` in `lib/sbi/sbi_init.c`. It reads the flattened device tree (FDT) to determine which types of peripherals are available and has simple drivers to initialize essential peripherals. It then calls a series of initialization functions:

- ▶ `sbi_scratch_init` Scratch space for firmware data structures
- ▶ `sbi_heap_init` Heap memory
- ▶ `sbi_domain_init` Root domain containing assigned memory and harts
- ▶ `sbi_hsm_init` Hart state management: start/stop/suspend
- ▶ `sbi_hart_init` Hart features, init `mstatus`, `fp`, trap delegation
- ▶ `sbi_console_init` Configure UART baud rate and control registers
- ▶ `sbi_boot_print_banner` Display boot messages
- ▶ `sbi_irqchip_init` Disable external interrupts
- ▶ `sbi_ipi_init` Configure software (inter-process) interrupts
- ▶ `sbi_tlb_init` Set up cross-hart TLB flush
- ▶ `sbi_timer_init` Configure timer
- ▶ `sbi_hart_pmp_configure` Set up PMP for domains
- ▶ `sbi_ecall_init` Register SBI extensions
- ▶ `sbi_boot_print_general` Display more boot messages
- ▶ `sbi_boot_print_domains` Domain-specific boot messages
- ▶ `sbi_boot_print_hart` Hart-specific boot messages
- ▶ `wake_coldboot_harts` Send IPI (inter-processor interrupt) to harts waiting for boot
- ▶ `sbi_hart_switch_mode` Enter the Linux kernel in S-mode

22.2.6.1 UART Initialization

To illustrate hardware initialization, consider the `sbi_console_init` UART initialization.

Wally uses the generic platform, which examines the flattened device tree to find the system has an ns16550a-compatible UART (see [Section 20.4](#)) at a clock frequency of 3.6864 MHz. The device tree does not specify the baud rate, so it defaults to 115200 in `lib/utils/fdt/fdt_helper.c`. `lib/utils/serial/fdt_serial_uart8250.c` assigns this UART to the UART8250 driver. `lib/utils/serial/uart8250.c` initializes the UART with a sequence of accesses given in [Table 22.5](#).

The UART is now ready to communicate. The firmware `console_putc` and `console_getc` routines invoke `uart8250_putc` and `uart8250_getc`. `putc` waits until the Line Status Register (LSR) indicates the Transmit Holding Register (THR) is empty and then writes the character to the THR. `getc` waits until the LSR indicates that data is ready and then reads the Receiver Buffer Register (RBR).

22.2.6.2 PLIC Initialization

As another example of hardware initialization, consider `sbi_platform_irqchip_init` that configures external interrupts on the Platform-Level Interrupt Controller (PLIC). The device tree indicates the PLIC has 53 sources and is compatible with `riscv`, `plic0`, so the firmware uses the driver in `lib/utils/irqchip/fdt_irqchip_plic.c` and `plic.c`.

Table 22.5 UART initialization accesses.

Address	Value	Register	Purpose
10000001	00	Interrupt Enable	Disable Interrupts
10000003	80	Line Control	Enable Divisor Latch Access Bit (DLAB)
10000000	02	Divisor Latch Low	Set baud rate to 115200
10000001	00	Divisor Latch High	Set baud rate to 115200
10000003	03	Line Control	8 data bits, 1 stop bit, no parity, disable DLAB
10000002	01	FIFO Control	Enable FIFO
10000004	00	Modem Control	Disable modem control signals such as CTS
10000005	Read	Line Status	Clear all line status errors
10000000	Read	Receiver Buffer	Clear any data sitting in receive buffer
10000007	00	Scratchpad	Clear UART scratchpad

The driver first calls `plic_cold_irqchip_init`, which sets a priority of 0 for all 53 interrupt sources, disabling them. The driver then calls `plic_warm_irqchip_init`, which clears the interrupt enable bits for all 53 sources in contexts 0 (machine) and 1 (supervisor) and also sets the priority threshold to 7 for both contexts to disregard all interrupts.

22.2.6.3 PMP Configuration

The `sbi_hart_pmp_configure` function sets up the PMP registers to isolate domains. Wally uses a single domain, so it has only two PMP regions, one to protect the firmware from lower-privileged access, and the other granting universal access to all other memory.

The PMP registers are configured with `pmpaddr0 = 0x2003fff`, `pmpaddr1 = 0xfffffffffffff`, and `pmpcfg0 = 0x1f18`. This sets regions 0 and 1 to naturally aligned power-of-two (NAPOT) sizes. Region 0 spans physical addresses 0x80000000-0x8001ffff, encompassing the SBI. The access bits are all 0, so this 128-KiB SBI memory space is secured against read, write, and execute access by S- and U-mode programs. Region 1 spans all of physical memory and has read, write, and execute access, so all programs can access any other part of physical memory excluding region 0. Notably, the S-mode kernel has direct access to I/O peripherals. The kernel prohibits mapping U-mode virtual addresses to peripheral physical addresses, so U-mode programs must access peripherals through OS calls rather than by direct memory-mapped I/O.

Recall from Section 11.2.2.2 how a region's size and starting address are calculated: `pmpaddr0` has 14 trailing 1s, so region 0's size is $2^{14+3} = 2^{17}$ bytes (128 KiB). Discarding these 1s and shifting left by 2 gives the region's starting address: `0x2000000 << 2 = 0x80000000`.

22.2.7 Kernel

The kernel runs in S-mode. It has some subtleties transitioning from firmware to kernel and from physical addressing to virtual addressing. The kernel is loaded at physical address 0x80200000, which will be mapped to virtual address 0xffffffff80000000. The firmware `sbi_hart_switch_mode` function enters the kernel by placing 0x80200000 in the `epc`, 01 (for S-mode) in `mstatus.MPP`, and executing an `mret` to jump to the start of the kernel and enter S-mode.

The assembly language kernel startup code is in `arch/riscv/kernel/head.S`. It begins at `_start` and immediately jumps to `_start_kernel`. Early steps include the following:

- ▶ Turn off interrupts
- ▶ Initialize the global pointer `gp` to a location above the Linux kernel
- ▶ Disable the FPU to detect the illegal use of floating-point operations in the kernel
- ▶ Hold a lottery to pick one hart to run the boot

`head.S` then calls `setup_vm` to create initial page tables mapping the virtual addresses in the kernel starting at `0xFFFFFFFF80000000` to the physical addresses where the kernel is loaded at `0x80200000`, using megapages for efficiency. It then calls `relocate_enable_mmu`, listed in [Code Example 22.3](#), which writes the `satp` to enable virtual memory. The instruction fetch immediately after the `satp` write causes a page fault because `PC = 0x80201054` is not a valid virtual address. The trap vector `stvec` is preset to the next instruction in the program but at the correct virtual address `0xffffffff80001054`, so the program continues properly. This switch from machine physical addressing to supervisor virtual addressing is called a *trampoline* because the trap handler immediately bounces the program into the kernel code. Now the kernel can load the rest of the kernel page tables, call `setup_trap_vector` to point `stvec` to the `handle_exception` supervisor trap handler, and eventually call `start_kernel` in `init/main.c`.

Code Example 22.3 LINUX KERNEL ENABLING VIRTUAL MEMORY (`head.S: relocate_enable_mmu`)

```
relocate_enable_mmu:
...
0000000080201020: 10561073 csrw stvec,a2          # stvec = ffffffff80001048 right after satp write
...
0000000080201040: 12000073 sfence.vma             # Ensure new page tables are in use
0000000080201044: 18051073 csrw satp,a0           # satp = 900000000084b2 (SV48 root)
                                # Use trampoline page table at 0x848b2000
                                # Cause page fault to stvec (next instruction)
0000000080201048: 00000517 auipc a0,0x0           # a0 = ffffffff80001048
000000008020104c: 03450513 add a0,a0,52           # a0 = ffffffff8000107c (a spin loop)
0000000080201050: 10551073 csrw stvec,a0         # Point stvec to ffffffff8000107c to spin for debug
0000000080201054: 046a8197 auipc gp,0x46a8        # Update global pointer to virtual addr
0000000080201058: ee418193 add gp,gp,-284        # gp = ffffffff846a8f38
000000008020105c: 18061073 csrw satp,a2         # satp = 900000000080805 (SV48 kernel page table)
0000000080201060: 12000073 sfence.vma           # Switch to the new page table
0000000080201064: 8082      ret                 # Return to _start_kernel
000000008020106e: f7bff0ef jal setup_trap_vector
setup_trap_vector:
0000000080201068: 00002517 auipc a0,0x2          # a0 = ffffffff80003068
000000008020106c: 02450513 add a0,a0,36         # a0 = ffffffff8000308c <handle_exception>
0000000080201070: 10551073 csrw stvec,a0         # stvec = ffffffff8000308c (handle_exception)
0000000080201074: 14001073 csrw sscratch,zero    # sscratch = 0 indicates kernel code
0000000080201078: 8082      ret                 # Return to _start_kernel
```

`start_kernel` invokes many more initializations including multiprocessing, memory management, scheduling, interrupts, timers, and the console. The details are beyond the scope of this book.

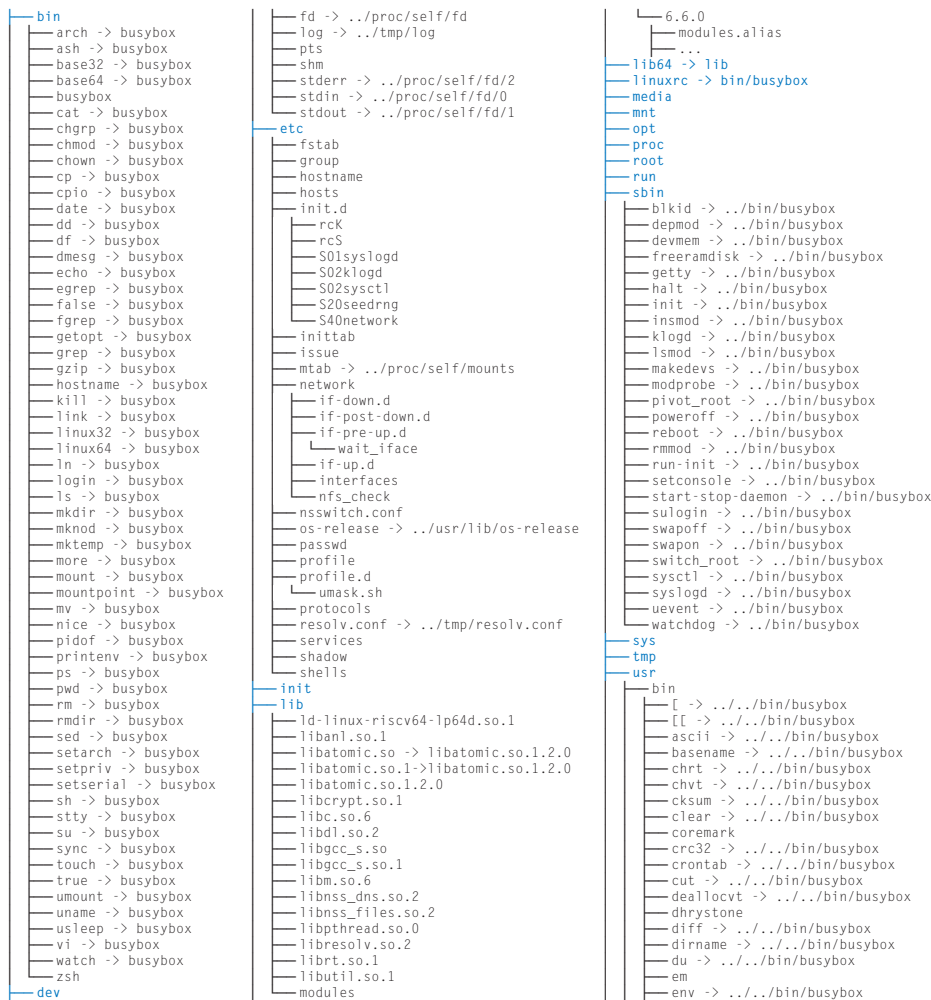


Fig. 22.1 Buildroot root filesystem using BusyBox.

22.2.8 Root File System (`rootfs`)

List the filesystem using
`tree $RISCV/buildroot/
 output/images/
 disassembly/rootfs`

The compressed root file system is placed in memory as `rootfs.cpio`. The file system is listed in Fig. 22.1. All the executable commands are symbolic links to the BusyBox root file system, which parses the command line and executes the appropriate command.

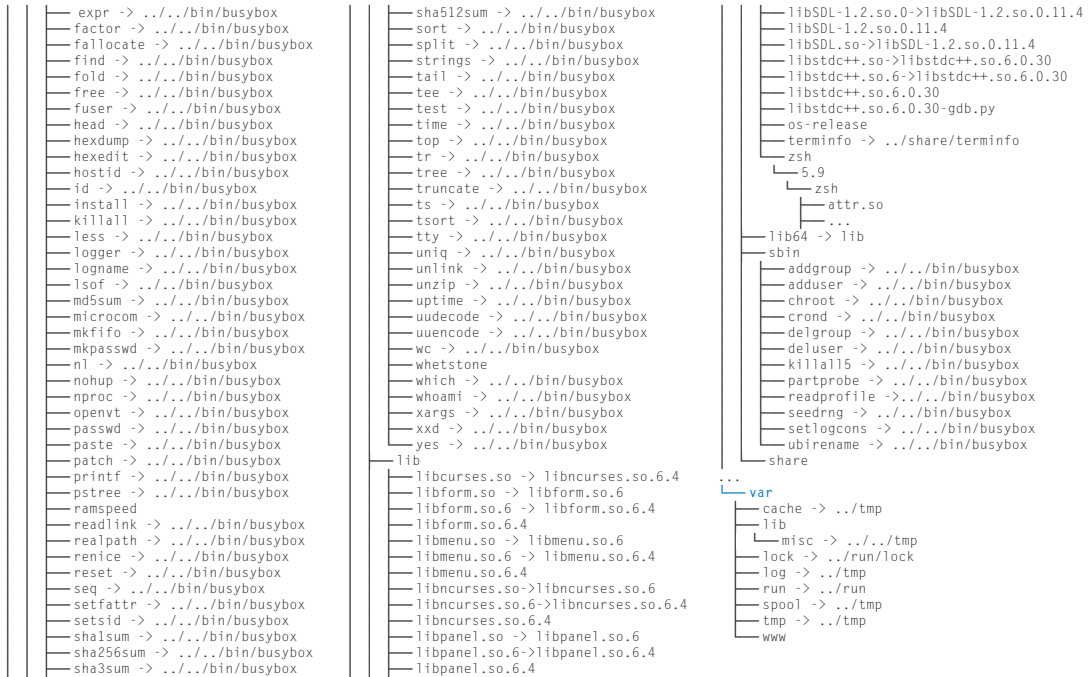


Fig. 22.1 Buildroot root filesystem using BusyBox. Cont'd

22.2.9 Device Tree

Fig. 22.2 lists the Wally device tree from \$WALLY/linux/devicetree/wally-virt.dts. It shows that physical memory begins at address 0x80000000 and has a size of 0x10000000 = 256 MiB. There is one CPU with a clock and timer timebase frequency of 0x1C9C380 = 30 MHz. The CPU supports all the standard extensions and SV48 virtual memory. The SoC has a UART, PLIC, and Core-Local Interruptor (CLINT) at specified addresses in the memory map. Linux uses the compatible fields to determine which device drivers to use.

22.2.10 Init Process

The first user process spawned by the kernel is /sbin/init. Like other user processes, it executes in U-mode. In Buildroot, init is part of BusyBox. The source code is in buildroot/output/build/busybox-1.36.1/init/init.c.

```

/dts-v1/;

/ {
    #address-cells = <0x02>;
    #size-cells = <0x02>;
    compatible = "wally-virt";
    model = "wally-virt,qemu";

    chosen {
        linux,initrd-end = <0x85c43a00>;
        linux,initrd-start = <0x84200000>;
        bootargs = "root=/dev/vda ro";
        stdout-path = "/soc/uart@10000000";
    };

    memory@80000000 {
        device_type = "memory";
        reg = <0x00 0x80000000 0x00 0x10000000>;
    };

    cpus {
        #address-cells = <0x01>;
        #size-cells = <0x00>;
        clock-frequency = <0x1C9C380>;
        timebase-frequency = <0x1C9C380>;

        cpu@0 {
            phandle = <0x01>;
            device_type = "cpu";
            reg = <0x00>;
            status = "okay";
            compatible = "riscv";
            riscv,isa-base = "rv64i";
            riscv,isa-extensions = "i", "m", "a", "f", "d", "c",
                "sstc", "svade", "svadu", "svinval", "svnapot", "svade", "svadu",
                "svinval", "svnapot", "svpbmt", "zba", "zbb", "zbc", "zbs", "zca",
                "zcb", "zcd", "zfa", "zfh", "zkn", "zkt", "zicbom",
                "zicboz", "zicntr", "zicond", "zicsr", "zifencei", "zihpm";

            riscv,cboz-block-size = <64>;
            riscv,cbom-block-size = <64>;
            mmu-type = "riscv,sv48";

            interrupt-controller {
                #interrupt-cells = <0x01>;
                interrupt-controller;
                compatible = "riscv,cpu-intc";
                phandle = <0x02>;
            };
        };
    };
};

```

Fig. 22.2 Wally device tree.

```

soc {
    #address-cells = <0x02>;
    #size-cells = <0x02>;
    compatible = "simple-bus";
    ranges;

    uart@10000000 {
        interrupts = <0x0a>;
        interrupt-parent = <0x03>;
        clock-frequency = <0x384000>;
        reg = <0x00 0x10000000 0x00 0x100>;
        compatible = "ns16550a";
    };

    plic@c000000 {
        phandle = <0x03>;
        riscv,ndev = <0x35>;
        reg = <0x00 0xc000000 0x00 0x210000>;
        interrupts-extended = <0x02 0x0b 0x02 0x09>;
        interrupt-controller;
        compatible = "sifive,plic-1.0.0\riscv,plic0";
        #interrupt-cells = <0x01>;
        #address-cells = <0x00>;
    };

    clint@2000000 {
        interrupts-extended = <0x02 0x03 0x02 0x07>;
        reg = <0x00 0x2000000 0x00 0x10000>;
        compatible = "sifive,clint0\riscv,clint0";
    };
};

```

Fig. 22.2 Wally device tree. Cont'd

`init` spawns a set of processes defined in `/etc/inittab`. These processes include setting the hostname, running the `/etc/init.d` scripts to start system logging, and starting a `getty` terminal with a prompt to log in.

22.2.11 Trap Handlers

The M-mode trap handler is part of OpenSBI and is named `_trap_handler` in `fw_base.S`. The handler fetches a thread pointer from `mscratch` and uses it to set up a stack pointer, then saves the `mepc`, `mstatus`, and general registers on the stack. It invokes the `sbi_trap_handler` in `sbi_trap.c` and then restores the registers and returns using `mret`.

The `sbi_trap_handler` processes M-mode interrupts. The handler performs SBI calls and emulates misaligned loads and stores (if they're

not already handled in hardware) as well as missing CSRs (see [Section 8.2.5.2](#)).

OpenSBI `delegate_traps` delegates S-mode interrupts (external, timer, and software) and breakpoints, misaligned instruction fetches, and U-mode `ecall` to the kernel's S-mode handler `handle_exception`, which is found in `linux-6.12.8/arch/riscv/kernel/entry.S`. `handle_exception` sets up a stack pointer from `sscratch` and saves `sstatus`, `sepc`, `stval`, `stcause`, `sscratch`, and general registers on the stack. It also disables the FPU and U-mode page accesses to protect against accidental illegal operations. Then it processes the exception, restores the registers, and returns using `sret`.

22.2.11.1 Software Interrupts

If one user process wants to send a message to another process, it can put the message in a shared memory buffer, then send a software interrupt to that process so it knows the message is waiting. Only the CLINT can set the machine software interrupt, but the OS needs to process this interrupt in S-mode, so it takes two traps to send a software interrupt and two more to receive.

The sending user-level processes use `ecall` to invoke the Linux `sbi_send_ipi(cpu)` system call. The S-mode Linux `sbi_send_ipi` function in turn makes another `ecall` to invoke the OpenSBI `sbi_send_ipi()` firmware system call. The M-mode OpenSBI `sbi_send_ipi` function sets the appropriate MSIP bit in the CLINT for the hart that should receive the software interrupt, then returns to Linux, which finally returns to the user process.

Setting MSIP in the CLINT triggers a machine software interrupt in the target hart, invoking `sbi_trap_handler`, which in turn calls `sbi_ipi_process()` to clear the MSIP bit and set `sip.SSIP`, then returns. This triggers a supervisor software interrupt, which is delegated to the S-mode exception handler, `handle_exception` routine, which invokes `do_irq`, which invokes `riscv_software_interrupt()`, which clears `sip.SSIP` and finally notifies the target process.

22.2.11.2 Timer Interrupts

The OS scheduler uses timer interrupts to determine when to switch between threads. Analogous to software interrupts, the CLINT can set `mip.MTIP`, triggering an M-mode timer interrupt, causing OpenSBI to set `sip.STIP`, causing a supervisor timer interrupt, invoking the Linux scheduler. Thus, the supervisor timer interrupt trap overhead saves and restores

registers twice. The timer interval is configurable but can be as often as every millisecond, so the overhead can be significant.

The Sstc extension (Section 8.2.5.4) reduces this overhead by sending an S-mode timer interrupt directly to the Linux kernel without involving M-mode firmware. If the `stimecmp` register exists, the OpenSBI firmware (`sbi_hart.c`) sets `menvcfg.STCE` to enable Sstc and `mcounteren.TM` to allow supervisor access to the counters. Then Linux programs the `stimecmp` register with the time for the next interrupt. The timer directly sets `sip.STIP`, which triggers the supervisor timer interrupt.

22.2.11.3 External Interrupts

External interrupts are routed through the PLIC, which collects pending interrupts from various sources, determines which ones are enabled, and delivers the highest priority interrupt to either the machine or supervisor context of a particular hart. Linux typically configures the PLIC to send the external interrupt directly to the OS by raising `sip.SEIP`.

22.3 BOOTING LINUX

Linux boots in seconds on the fast QEMU emulator. It takes hours or days to boot Linux on the Wally SystemVerilog code in simulation. Booting Wally on an FPGA takes only minutes, as described in Section 23.2.2.4 for the Arty A7 FPGA board.

22.3.1 Booting Linux on QEMU

Our Linux image boots on the QEMU *virt* platform, which provides software emulation of common hardware such as a UART. It requires the firmware, kernel, root file system, device tree, and CPU configuration passed on the command line. `qemuBoot.sh` sets all these options to match Wally. These images are loaded into memory before the boot starts so that virt does not have to emulate any mass storage devices.

```
$ cd $WALLY/linux
$ ./qemuBoot.sh
```

Fig. 22.3 shows the messages printed during boot. If all goes well, you will see a login prompt, highlighted in blue at the bottom of the figure. Type “root” to login and then feel free to play around with the system using the command prompt!

OpenSBI v1.3



```

Platform Name       : wally-virt.qemu
Platform Features   : medeleg
Platform HART Count : 1
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 30000000Hz
Platform Console Device : uart8250
Platform HSM Device : ---
Platform PMU Device : ---
Platform Reboot Device : ---
Platform Shutdown Device : ---
Platform Suspend Device : ---
Platform CPPC Device : ---
Firmware Base       : 0x80000000
Firmware Size        : 194 KB
Firmware RW Offset   : 0x20000
Firmware RW Size     : 66 KB
Firmware Heap Offset : 0x28000
Firmware Heap Size    : 34 KB (total), 2 KB (reserved), 9 KB (used), 22 KB (free)
Firmware Scratch Size : 4096 B (total), 760 B (used), 3336 B (free)
Runtime SBI Version  : 1.0

Domain0 Name        : root
Domain0 Boot HART    : 0
Domain0 HARTs        : 0*
Domain0 Region00     : 0x0000000002000000-0x000000000200ffff M: (I,R,W) S/U: ()
Domain0 Region01     : 0x0000000008000000-0x0000000008001ffff M: (R,X) S/U: ()
Domain0 Region02     : 0x00000000080020000-0x0000000008003ffff M: (R,W) S/U: ()
Domain0 Region03     : 0x0000000000000000-0xffffffffffffffff M: (R,W,X) S/U: (R,W,X)
Domain0 Next Address : 0x00000000080200000
Domain0 Next Arg1    : 0x00000000082200000
Domain0 Next Mode     : S-mode
Domain0 SysReset      : yes
Domain0 SysSuspend    : yes

Boot HART ID         : 0
Boot HART Domain     : root
Boot HART Priv Version : v1.12
Boot HART Base ISA    : rv64imafdc
Boot HART ISA Extensions : time,sstc
Boot HART PMP Count   : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits : 54
Boot HART MHPM Count  : 16
Boot HART MIDELEG      : 0x0000000000001666
Boot HART MEDELEG      : 0x0000000000f0b509
[ 0.000000] Linux version 6.6.0 (harris@chips) (riscv64-buildroot-linux-gnu-gcc.br_real (Buildroot 2023.05.3-dirty)
12.3.0, GNU ld (GNU Binutils) 2.38) #2 Tue Nov 21 19:16:18
PST 2023
[ 0.000000] Machine model: wally-virt.qemu
[ 0.000000] SBI specification v1.0 detected
[ 0.000000] SBI implementation ID=0x1 Version=0x10003
[ 0.000000] SBI TIME extension detected
[ 0.000000] SBI IPI extension detected
[ 0.000000] SBI RFENCE extension detected
[ 0.000000] efi: UEFI not found.
[ 0.000000] OF: reserved mem: 0x00000000080000000..0x0000000008001ffff (128 KiB) nomap non-reusable mmode_resv0@80000000
[ 0.000000] OF: reserved mem: 0x00000000080020000..0x0000000008003ffff (128 KiB) nomap non-reusable mmode_resv1@80020000
[ 0.000000] Zone ranges:
[ 0.000000] DMA32 [mem 0x00000000080000000-0x000000000800fffff]
[ 0.000000] Normal empty
[ 0.000000] Movable zone start for each node
[ 0.000000] Early memory node ranges
[ 0.000000] node 0: [mem 0x00000000080000000-0x0000000008003ffff]
[ 0.000000] node 0: [mem 0x00000000080040000-0x000000000800fffff]

```

Fig. 22.3 Linux boot messages.

```

[ 0.000000] Initmem setup node 0 [mem 0x0000000080000000-0x000000008fffffff]
[ 0.000000] riscv: base ISA extensions acdfim
[ 0.000000] riscv: ELF capabilities acdfim
[ 0.000000] Kernel command line: root=/dev/vda ro
[ 0.000000] Dentry cache hash table entries: 32768 (order: 6, 262144 bytes, linear)
[ 0.000000] Inode-cache hash table entries: 16384 (order: 5, 131072 bytes, linear)
[ 0.000000] Built 1 zonelists, mobility grouping on. Total pages: 64640
[ 0.000000] mem auto-init: stack:off, heap alloc:off, heap free:off
[ 0.000000] Virtual kernel memory layout:
[ 0.000000]   fixmap : 0xffffbffffea00000 - 0xffffbfffff000000 (6144 kB)
[ 0.000000]   pci io  : 0xffffbfffff000000 - 0xffff1c000000000000 ( 16 MB)
[ 0.000000]   vmemmap : 0xffff1c000000000000 - 0xffff20000000000000 (1024 TB)
[ 0.000000]   vmlalloc : 0xffff20000000000000 - 0xffff60000000000000 (16384 TB)
[ 0.000000]   modules : 0xffff600000000000 - 0xffff800000000000 (1977 MB)
[ 0.000000]   lowmem  : 0xffff800000000000 - 0xffff60000001000000 ( 256 MB)
[ 0.000000]   kernel  : 0xffff800000000000 - 0xffff800000000000 (2047 MB)
[ 0.000000] Memory: 123724K/262144K available (2542K kernel code, 3693K rwdata, 2048K rodata, 62517K reserved, 200K cma reserved, 0K bswap)
[ 0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
[ 0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irq: 0
[ 0.000000] riscv-intc: 64 local interrupts mapped
[ 0.000000] plic@c000000: mapped 53 interrupts with 1 handlers for 2 contexts.
[ 0.000000] clocksource: riscv_clocksource: mask: 0xffffffffffffffff max_cycles: 0xdd67c8a60, max_idle_ns: 881590406601 ns
[ 0.000016] sched_clock: 64 bits at 30MHz, resolution 33ns, wraps every 4398046511088ns
[ 0.000192] riscv-timer: Timer interrupt in S-mode is available via sstc extension
[ 0.000809] Console: colour dummy device 80x25
[ 0.000865] printk: console [tty0] enabled
[ 0.001425] Calibrating delay loop (skipped), value calculated using timer frequency.. 60.00 BogoMIPS (lpj=120000)
[ 0.001454] pid_max: default: 32768 minimum: 301
[ 0.001624] Mount-cache hash table entries: 512 (order: 0, 4096 bytes, linear)
[ 0.001634] Mountpoint-cache hash table entries: 512 (order: 0, 4096 bytes, linear)
[ 0.005643] riscv: ELF compat mode supported
[ 0.005708] ASID allocator using 16 bits (65536 entries)
[ 0.005894] EFI services will not be available.
[ 0.007213] devtmpfs: initialized
[ 0.008161] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 7645041785100000 ns
[ 0.008210] futex hash table entries: 256 (order: 1, 12288 bytes, linear)
[ 0.009045] DMA: preallocated 128 KiB GFP_KERNEL pool for atomic allocations
[ 0.009101] DMA: preallocated 128 KiB GFP_KERNEL|GFP_DMA32 pool for atomic allocations
[ 0.032655] cpu0: Ratio of byte access time to unaligned word access is 8.33, unaligned accesses are fast
[ 0.034975] clocksource: Switched to clocksource riscv_clocksource
[ 0.043042] workingset: timestamp_bits=62 max_order=15 bucket_order=0
[ 0.043545] io scheduler mq-deadline registered
[ 0.043562] io scheduler kyber registered
[ 0.043596] io scheduler bfq registered
[ 0.064218] Serial: 8250/16550 driver, 1 ports, IRQ sharing disabled
[ 0.067933] 100000000.uart: ttyS0 at MMIO 0x10000000 (irq = 2, base_baud = 230400) is a 16550A
[ 0.068242] printk: console [ttyS0] enabled
[ 0.082181] debug_vm_pgtable: [debug_vm_pgtable]: Validating architecture page table helpers
[ 0.087038] clk: Disabling unused clocks
[ 0.107776] Unpacking initramfs...
[ 0.163386] Freeing initrd memory: 60388K
[ 0.175992] Freeing unused kernel image (initmem) memory: 62516K
[ 0.176178] Run /init as init process
init started: BusyBox v1.36.1 (2023-11-21 19:10:20 PST)
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Saving 256 bits of non-creditable seed for next boot
Starting network: /etc/init.d/S40network: line 12: /sbin/ifup: not found
FAIL

BusyBox v1.36.1 (2023-11-21 19:10:20 PST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

Hello this ~/.profile is meant to demonstrate running some basic commands on Wally.
I am root
And I am on WallyHostname
Created myFile.txt and moved it to myDir. It contains:
Hello World!
And farewell!
Created myScript.sh. Running it yields:
Hello this is another example script
Created symLinkToMyScript.sh. Running it yields:
Hello this is another example script

```

Fig. 22.3 Linux boot messages. Cont'd


```

Now let's remove all these example files and scripts
Here is disk usage:
Filesystem                Size      Used Available Use% Mounted on
devtmpfs                   60.4M        0      60.4M   0% /dev
tmpfs                      120.4M        0      120.4M   0% /dev/shm
tmpfs                      120.4M        0      120.4M   0% /tmp
tmpfs                      120.4M      8.0K      120.4M   0% /run

And here are the current processes:
  PID USER      VSZ STAT COMMAND
    1 root         0 S    init
    2 root         0 SW   [kthreadd]
    3 root         0 SW   [pool_workqueue_]
    4 root         0 IW<  [kworker/R-slab_]
    5 root         0 IW   [kworker/0:0-eve]
    6 root         0 IW<  [kworker/0:0H]
    7 root         0 IW   [kworker/u2:0-ev]
    8 root         0 IW<  [kworker/R-mm_pe]
    9 root         0 SW   [ksoftirqd/0]
   10 root         0 SW   [kdevtmpfs]
   11 root         0 SW   [khungtaskd]
   12 root         0 SW   [oom_reaper]
   13 root         0 IW<  [kworker/R-write]
   14 root         0 SW   [kcompactd0]
   15 root         0 IW<  [kworker/R-kbloc]
   16 root         0 IW   [kworker/0:1-eve]
   17 root         0 SW   [kswapd0]
   18 root         0 IW   [kworker/u2:1-ev]
   19 root         0 IW   [kworker/u2:2-ev]
   40 root        2208 S    /sbin/klogd -n
   54 root        2208 S    -/bin/sh
   70 root        2208 R    ps

And finally a login prompt.
WallyHostname login:
WallyHostname login: root
login[71]: root login on 'console'

BusyBox v1.36.1 (2023-11-21 19:10:20 PST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# mkdir test
# ls
test

```

Fig. 22.3 Linux boot messages. Cont'd

To view the physical addresses of the boot files from [Table 22.3](#), press <Ctrl-a> c to enter the QEMU monitor. Then type `info roms`:

```

(qemu) info roms
addr=00000000000001000 size=0x000028 mem=rom name="mrom.reset"
addr=00000000000001028 size=0x000030 mem=rom name="mrom.finfo"
addr=00000000080000000 size=0x01f7d8 mem=ram name="fw_jump.elf ELF program header segment 1"
addr=00000000080020000 size=0x005058 mem=ram name="fw_jump.elf ELF program header segment 2"
addr=00000000080200000 size=0x46a8c00 mem=ram name="Image"
addr=00000000088200000 size=0x3af9c00 mem=ram name="rootfs.cpio"
addr=0000000008fe00000 size=0x0005ff mem=ram name="fdt"
(qemu) quit

```

22.3.2 Booting Linux on Wally using a Logic Simulator

Launch a simulation of Buildroot on Wally in Qesta using `wsim`:

```
$ wsim buildroot buildroot --args "+INSTR_LIMIT=600000000" --lockstep
```

Linux does not naturally terminate, so the `INSTR_LIMIT` argument specifies how many instructions to run; 1,600,000 is enough to reach the OpenSBI banner, and 600 million is more than enough to reach the login prompt. Use the `--lockstep` flag to optionally compare against the ImperasDV model using the lock-step-compare methodology described in [Sections 5.6 and 5.11.9](#).

The Linux boot presently executes about 581 million instructions and takes about 20 hours on a high-performance PC with ImperasDV lockstep in Questa or 2 hours in Verilator without lockstep. Linux continues to evolve, so the timing may vary.

```
# /etc/inittab
#
# Copyright (C) 2001 Erik Andersen <andersen@codepoet.org>
#
# Note: BusyBox init doesn't support runlevels. The runlevels field is
# completely ignored by BusyBox init. If you want runlevels, use
# sysvinit.
#
# Format for each entry: <id>:<runlevels>:<action>:<process>
#
# id          == tty to run on, or empty for /dev/console
# runlevels   == ignored
# action      == one of sysinit, respawn, askfirst, wait, and once
# process     == program to run

# Startup the system
::sysinit:/bin/mount -t proc proc /proc
::sysinit:/bin/mount -o remount,rw /
::sysinit:/bin/mkdir -p /dev/pts /dev/shm
::sysinit:/bin/mount -a
::sysinit:/sbin/swapon -a
null::sysinit:/bin/ln -sf /proc/self/fd /dev/fd
null::sysinit:/bin/ln -sf /proc/self/fd/0 /dev/stdin
null::sysinit:/bin/ln -sf /proc/self/fd/1 /dev/stdout
null::sysinit:/bin/ln -sf /proc/self/fd/2 /dev/stderr
::sysinit:/bin/hostname -F /etc/hostname
# now run any rc scripts
::sysinit:/etc/init.d/rcS

# (commented out) Put a getty on the serial port
#console::respawn:/sbin/getty -L console 0 vt100 # GENERIC_SERIAL
# Actually no, let's automatically login
console::respawn:-/bin/sh
# Stuff to do for the 3-finger salute
#::ctrlaltdel:/sbin/reboot

# Stuff to do before rebooting
::shutdown:/etc/init.d/rcK
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
```

Fig. 22.4 `/etc/inittab` modified to login automatically.

22.3.3 Modifying Buildroot

Advanced users may wish to modify Buildroot. The only source files manually given to Buildroot are located in `$WALLY/linux/br2-external-tree`. Some of these are configuration files, and others are meant to go in the root filesystem of the final machine.

22.3.3.1 Root File System Overlay

`$WALLY/linux/br2-external-tree/board/wally/rootfs_overlay` contains files that are meant to go in the root filesystem of the final machine. These files aren't strictly necessary for booting Linux but can be helpful for testing purposes. For example, `/etc/inittab` is modified from the default version to cause the system to login and start up a shell without any user input for username or password, as shown in blue in [Fig. 22.4](#). The `/.profile` file contains some test commands executed before initiating an actual login prompt, as shown in [Fig. 22.5](#). This way, shell commands can be tested without requiring user input (which may cause complications in QEMU record/replay mode).

```
echo "Hello this ~/.profile runs some basic commands on Wally."
echo "I am $(whoami)"
echo "And I am on $(hostname)"
touch myFile.txt
echo "Hello World!" > myFile.txt
echo "And farewell!" >> myFile.txt
mkdir myDir
mv myFile.txt myDir
echo "Created myFile.txt and moved it to myDir. It contains:"
cat myDir/myFile.txt
touch myScript.sh
echo "echo \"Hello this is another example script\"" > myScript.sh
chmod +x myScript.sh
echo "Created myScript.sh. Running it yields:"
./myScript.sh
cd myDir
ln -s ../myScript.sh symLinkToMyScript.sh
echo "Created symLinkToMyScript.sh. Running it yields:"
./symLinkToMyScript.sh
echo "Now let's remove all these example files and scripts"
cd /
rm -r myDir
rm myScript.sh
echo "Here is disk usage:"
df -h
echo "And here are the current processes:"
ps
echo "And finally a login prompt."
Login
```

Fig. 22.5 `/.profile`.

22.3.3.2 Configuration Files

Buildroot provides an interactive *menuconfig* interface for configuration, which consists of three files and configuration tools: `.config` (*menuconfig*), `linux.config` (*linux-menuconfig*), and `busybox.config` (*busybox-menuconfig*). For example, our configuration was edited to enable debug symbols. To edit `.config` using *menuconfig*:

```
# As the user with ownership of the files (likely root or cad)
$ cd $RISCV/buildroot
$ make menuconfig
```

Full details on how to configure Buildroot are beyond the scope of this book. See the Buildroot user manual (buildroot.org/downloads/manual/manual.html) for further information if you plan to modify Buildroot.

Running the Linux Makefile from [Section 22.2.1](#) automatically created `.config` from `$WALLY/linux/br2-external-tree/configs/wally_defconfig` and used `linux.config` and `busybox.config` from the `$WALLY/linux/br2-external-tree/board/wally/` directory.

`make menuconfig` automatically loads `$RISCV/buildroot/.config` and saves to the same file. Similarly, `make linux-menuconfig` and `make busybox-menuconfig` edit `linux.config` and `busybox.config`, respectively. All the configuration files may also be changed using a text editor. To save the changes to the main `.config` file, use `make savedefconfig`. For the Linux and BusyBox configurations, use `make linux-update-defconfig` and `make busybox-update-config` respectively.

22.3.3.3 Rebuilding the Image

After making changes to the root filesystem or configuration files, rebuild Buildroot with the command:

```
$ cd $WALLY/linux
$ make Image BUILDROOT=$(RISCV)/buildroot
$ make disassemble BUILDROOT=$(RISCV)/buildroot
$ make dumptvs
```

22.4 SUMMARY

Linux is an open-source industrial-strength OS widely used in everything from embedded systems and mobile devices to PCs and servers. This chapter described the hardware and software needed to boot Linux on Wally.

The OS components include a zero stage bootloader (ZSBL), firmware, device tree, kernel, and root file system. These may be read from an SD card into RAM on an FPGA board or simply initialized in RAM during logic simulation.

At reset, the processor executes the ZSBL from a small boot ROM. The ZSBL reads the firmware and device tree from the SD card if necessary, then jumps to the OpenSBI firmware bootloader.

The OpenSBI firmware bootloader runs in M-mode. It sets up an M-mode trap handler; reads platform-specific information from the device tree; initializes harts, peripherals, and CSRs; and prints initial boot messages to the COM port. The bootloader then loads the kernel and enters S-mode to begin executing the kernel.

The Linux kernel initializes the virtual memory page table and turns on virtual memory. Henceforth, Linux runs in S-mode with virtual addressing. It sets up an S-mode trap handler and then initializes many services including memory management, scheduling, timers, and console I/O. After this, the kernel reads the root file system and calls the `init` process, which is the first U-mode process and the parent of all other Linux processes. Finally, the user sees the login prompt on the console.

The cvw repository supports booting Buildroot Linux on three platforms. First, a generic RISC-V processor can be emulated using QEMU. Second, a logic simulator (Questa, VCS, or Verilator) simulates Wally booting OpenSBI and the Linux kernel. Finally, [Chapter 23](#) implements Wally physically on an FPGA and boots Linux into an interactive terminal shell.

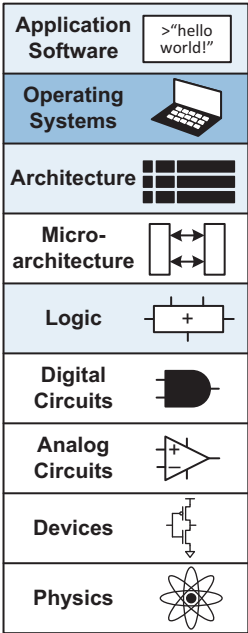
- 23.1 FPGA Principles
- 23.2 Wally FPGA Implementation
- 23.3 FPGA Results
- 23.4 Interesting Bugs
- 23.5 Summary

A field programmable gate array (FPGA) contains a configurable array of logic elements (LEs), memories, input/output (I/O) pins, and routing cells that can be programmed to perform arbitrary digital functions. To program an FPGA, design tools translate RTL (e.g., SystemVerilog code) into a *bitstream*, which configures the FPGA as particular logic functions, memory, and interfaces.

Semiconductor economics considers both *nonrecurring engineering* costs (*NREs*) and *per-unit* cost. The NRE to create a mask set for an application-specific integrated circuit (ASIC) runs in the millions of dollars, and fabrication takes 3–12 weeks. Each chip typically costs less than \$100 (perhaps less than \$1 for simple chips). In contrast, an FPGA evaluation board typically costs between \$100 and \$10,000, and a new bitstream can be created and downloaded in minutes to days depending on the size of the design. Therefore, FPGAs are suitable for rapid prototyping, as opposed to ASIC test chips, which are expensive and time-consuming to design and build. Running a system in hardware (on an FPGA) is faster than in simulation, and an FPGA can interact with external hardware in a way that is difficult to model in simulation. When a bug is found, the designer can generate and download a new bitstream to reprogram the FPGA.

Compared to ASICs, FPGAs tend to run at an order of magnitude lower clock frequency and cost two orders of magnitude more per gate. However, with parallelism and special-purpose logic, FPGAs can far exceed the performance and performance/watt of a general-purpose microprocessor. Therefore, FPGAs are suitable for simple or low-volume products when a general-purpose microprocessor is too slow. For example, for products selling thousands of units, the NRE of an ASIC can far exceed the per-unit cost. FPGAs are commonly used in communication, scientific, medical, military, and aerospace applications that need flexibility and special-purpose logic yet are produced in modest quantities.

FPGAs are also used in *emulation*, a logic verification technique where the design is mapped onto an FPGA so that large numbers of tests can be run on physical hardware at a much faster rate than is possible in software



simulation. FPGAs may include an integrated logic analyzer (ILA) to improve observability and controllability during testing. Industrial SoC design teams use commercial FPGA-based emulation systems to replace hundreds of servers running verification tests for months or years.

23.1 FPGA PRINCIPLES

FPGAs evolved from *programmable logic array* (PLA) chips, which were developed in the early 1980s to perform arbitrary sum-of-product functions. The hardware costs of PLAs grow quadratically with complexity, so FPGAs were developed in the mid-1980s as a more flexible way to implement larger systems.

23.1.1 FPGAs

As Moore's law continued to allow more transistors on a chip, the cost of logic arrays grew as the square of the number of I/Os, making it inefficient to scale to large designs [Trimberger15]. Moreover, large numbers of product terms loaded down individual connections causing slow propagation delays. Consequently, a new solution was needed to provide better scalability [DeHon99].

Altera (now part of Intel), Xilinx (now part of AMD), and Actel (now Microsemi) were founded between 1983 and 1985 to develop chips with reprogrammable logic. Actel coined the term *field programmable gate array* in 1988 [Trimberger15]. An FPGA has an array of programmable LEs connected by programmable interconnect. Each LE typically contains a *lookup table* (LUT) and a register. The LUT is a small memory that performs any combinational function of a small number of inputs (typically 4-6). An FPGA performs complex logic functions by connecting outputs of one LE to the inputs of another.

Fig. 23.1 shows a generic FPGA architecture. The configurable LEs are surrounded by input/output elements (IOEs) that allow connections to the outside world. Configurable switches that route signals between LEs are shown in the hexagon within each LE intersection. FPGAs are programmed by downloading the contents of the LEs and the configuration of the switches. Most modern FPGAs use static random access memory (SRAM) for configuration memory. Both the LUT contents and interconnect configurations are programmed by downloading a *bitstream file* into the configuration memory.

Large FPGAs now contain millions of LEs, along with other specialized blocks such as hundreds of megabits of SRAM, thousands of digital signal processing (DSP) units, high-speed serial I/O, multicore microprocessors, and clock generators. FPGAs also come with intellectual property (IP) libraries that include memory controllers, bus interfaces, and a variety of peripherals. FPGAs are sophisticated enough to implement complex SoCs. Vendor-specific electronic design automation tools synthesize HDL code

Altera/Intel refer to the configurable logic elements as *logic elements* (LEs), and Xilinx/AMD refer to them as *configurable logic blocks* (CLBs).

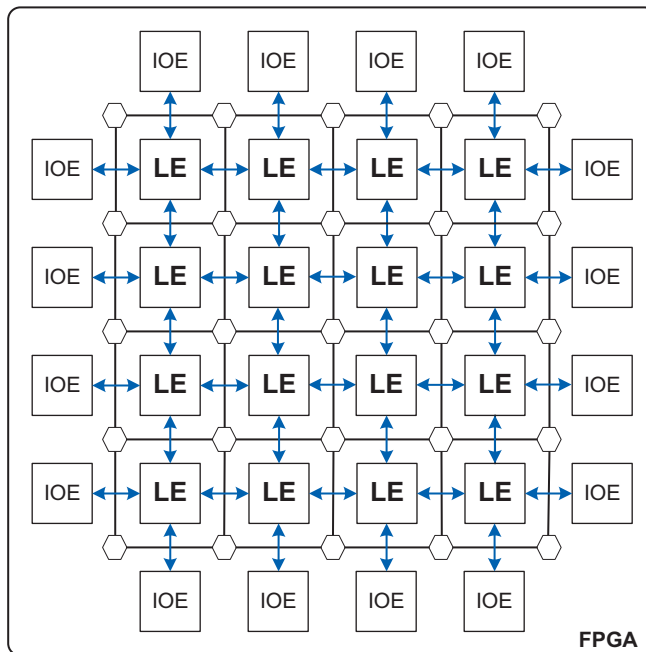


Fig. 23.1 Generic array structure of an FPGA.

and map it onto a specific FPGA. In addition to design files, users provide constraint files that specify I/O pin assignments and target clock rate.

23.2 WALLY FPGA IMPLEMENTATION

Wally boots Linux on an AMD (formerly Xilinx) FPGA. The FPGA implementation is useful for hands-on hardware designs and to diagnose logic bugs that are slow to reproduce in simulation. Specifically, the `fpga` configuration (a variation of `rv64gc`) of `wallypipelinedsoc` has been run on an Arty A7-100T board, an FPGA development board available from Digilent for \$299 in 2024. The board contains a Xilinx SC7A100TCSG324-1 FPGA, which is a midsized Artix-7 FPGA containing:

- ▶ 63,400 LUTs and 126,800 flops
- ▶ 4,860-KiB block RAM organized as 135 36-Kib dual-ported memories
- ▶ 240 DSP slices (25×18 -bit multiplier, adder, and accumulator)
- ▶ 6 clock managers with *phase-locked loops* (PLLs)
- ▶ Up to 300 I/O pins

Table 23.1 lists the hardware required to build this FPGA implementation. Any micro B USB cable or SD card (SDC) will work.

Table 23.1 Required FPGA hardware.

Item	Description	Vendor	Price
Arty A7	FPGA development board	digilent.com/shop/arty-a7-100t-artix-7-fpga-development-board/	\$299.00
Micro USB cable	Power, programming, and communication	digilent.com/shop/usb-a-to-micro-b-cable/	\$4.99
PmodSD board	Adapter board to connect to external flash card	digilent.com/reference/pmod/pmodsd/	\$9.99
SD card	Non-volatile storage for OS, programs, and data	digilent.com/shop/microsd-card-with-adapter/	\$12.99
Flash card reader	Read and write SD card from development PC	www.westerndigital.com/products/accessories/sandisk-image-mate-pro-usb-c-multi-card-reader-writer?sku=SDDR-A631-ANGNN	\$29.99

The `wallypipelinedsoc` module requires additional peripherals to implement a functional computer, including a dynamic random access memory (DRAM) memory controller, a PLL to generate arbitrary clock frequencies, and an interface to nonvolatile memory storage such as an SD flash card. The Wally FPGA implementation uses IP from AMD and open-source projects to fill these gaps.

Fig. 23.2 shows the block diagram of the Arty A7 board in light blue and the top-level Verilog module, `fpgatopArtyA7`, implemented on the Artix-7 FPGA in medium blue. The board connects the FPGA to a 256-MiB DDR3 memory, a 100-MHz clock, and various I/O devices, buttons, LEDs, and breakout ports for expansion boards.

`fpgatopArtyA7` instantiates the `wallypipelinedsoc` along with 5 IP blocks divided into three categories: clock and reset, bus converters, and memory controller. Table 23.2 describes the new AMD IP modules.

The *mixed-mode clock manager* (`mmcm`) is AMD IP that contains a PLL, which takes in the 100-MHz on-board clock and generates three system clocks: 167 MHz (for memory controller calibration), 200 MHz (for the DDR memory interface), and 20 MHz (`CPUCLK` for the Wally core). The system reset module (`sysrst`) takes in the asynchronous active low reset button (`resetn`) and produces `reset_periph` synchronized to the 20-MHz `CPUCLK`. The DDR3 DRAM controller (`memctrlddr3`) and the clock converter use `resetn` directly.

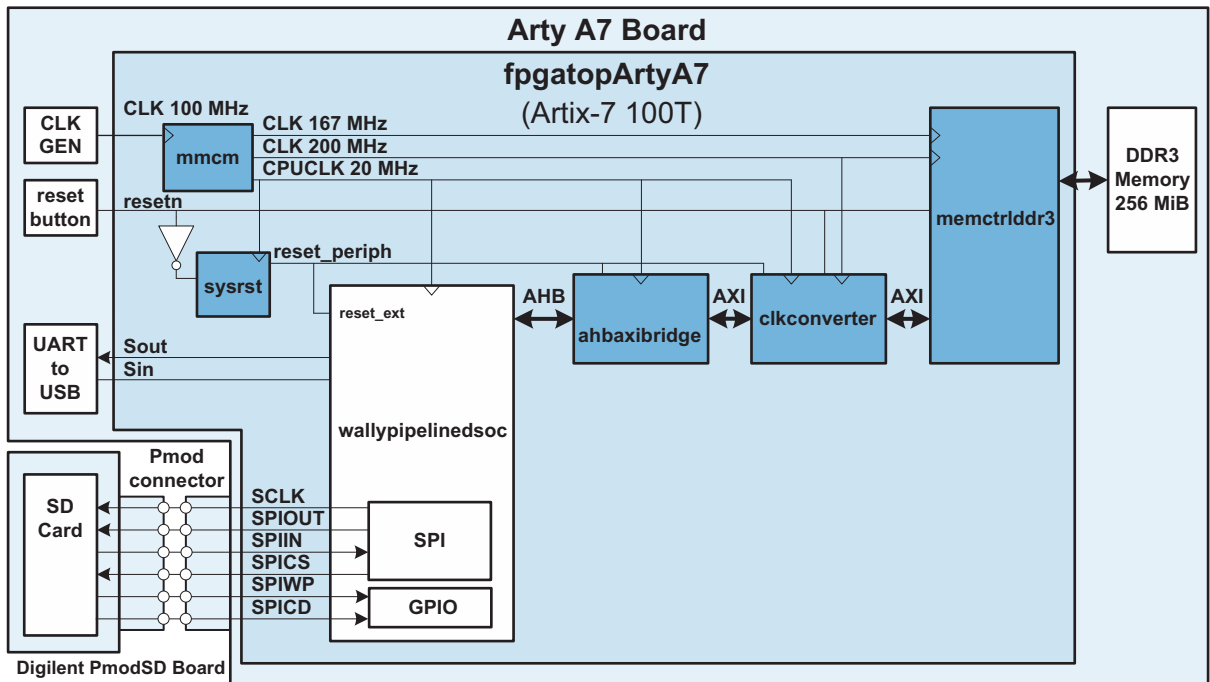


Fig. 23.2 Artix A7 FPGA block diagram.

Table 23.2 AMD IP modules.

Module	Description
sysrst	Synchronize reset to clock
mmcm	Clock generator
ahbaxibridge	Converts AHB to AXI4
clkconverter	Clock crossing domain between CPUCLK (20 MHz) and the 200-MHz memory clock
memctrlDDR3	DDR3 memory controller

The AMD DRAM controller (`memctrlDDR3`) implements the AXI4 subordinate protocol that converts the bus read and writes into DDR3 DRAM read and writes. `memctrlDDR3` also handles the nuances of DRAM management, including I/O timing, skew compensation, periodically refreshing data at each memory address, and opening and closing memory banks. The full scope of memory controller design is its own textbook.

Wally also supports higher performance FPGA boards such as the VCU108 and Gensys 2, which have 2-GiB DDR4 and 1-GiB DDR3 memory, respectively.

Whereas the `wallypipelinedsoc` external bus connection is AHB and runs at the 20-MHz core clock frequency, the DDR3 memory controller (`memctrlddr3`) uses AXI4 and runs at 200 MHz. The AHB to AXI bridge (`ahbaxibridge`) converts between the two bus protocols, and a first in, first out (FIFO) clock converter (`clkconverter`) buffers the two clock domains. AMD provides IP for both.

Wally uses a second instance of the SPI peripheral (see [Section 20.5](#)) to implement the SD card controller. [Section 23.2.1](#) describes additional details specific to the SPI-based SD card implementation.

[Fig. 23.3](#) highlights the important components of the Arty A7 board. The Artix-7 FPGA is located in the center with the DDR3 memory off to its right. The USB cable provides both power and communication. USB connects two universal asynchronous receiver/transmitter (UART) controllers: one for programming and debugging the FPGA (see [Section 23.2.2.3](#)) and the other for connecting to Wally's UART (see [Section 23.2.2.4](#)). The PmodSD board connects via the right-most Pmod port, port D. `resetn` resets all modules on the FPGA.

23.2.1 SD Card

The SD card communicates using the 1-bit SPI interface. With a clock frequency of f_c , the bandwidth is $(f_c \text{ transactions/s}) \times (1 \text{ bits/transactions}) / (8 \text{ bits/byte}) = f_c / 8 \text{ bytes/s}$. The maximum frequency of 25 MHz delivers 3.125 MB/s. Although relatively slow, SPI has several advantages: it

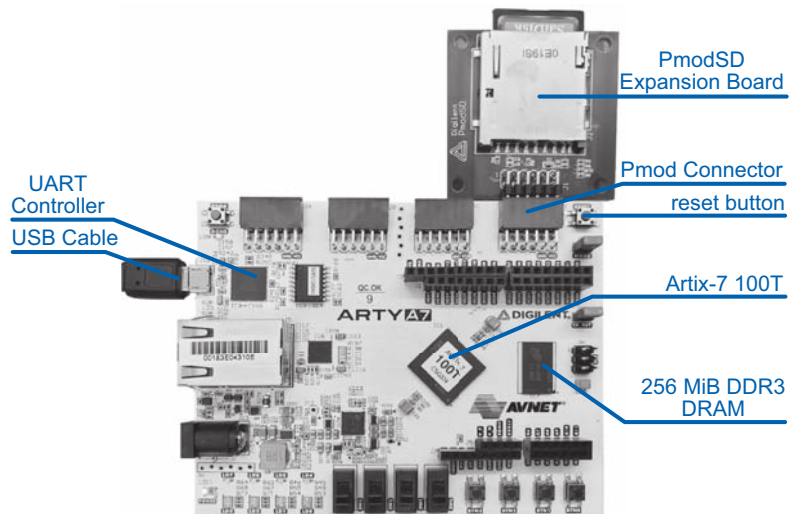


Fig. 23.3 Arty A7 FPGA and PmodSD DRAM expansion card.

does not require a costly license, and the Linux kernel contains built in drivers for it.

SPI does not have a way to indicate when a card is inserted or if the card is write protected. A pin from the card socket called *card detect* (CD) connects to general-purpose I/O (GPIO) pin 2 to interrupt the CPU when a card is inserted. Similarly, another pin called *write protect* (WP) drives GPIO pin 3 to prevent the CPU from writing the SD card if high.

SD cards have a second interface, *SDIO*, which is faster than the SPI interface and runs at 50 MHz with 4 bits/transaction, thus delivering 25 MB/s. However, the interface requires a costly license.

23.2.2 Arty A7 Setup

This section describes how to build a Wally SoC targeted to an Arty A7 FPGA development board, which is then capable of booting Buildroot Linux with an interactive command-line console. As described in detail in the remainder of this section, the four main steps are:

1. Build the SD flash card to hold a Buildroot Linux image.
2. Synthesize Wally and place and route it on the FPGA.
3. Program the FPGA to be configured as Wally.
4. Connect a UART terminal interface.

Buildroot, described in [Section 22.2](#), is an automated system for cross-compiling Linux to embedded platforms.

23.2.2.1 Build the SD Flash Card

The bootloader, operating system (OS), and user programs are stored on an SD flash card. The bootloader copies the OS from the flash card into main memory and then begins executing the OS. Specifically, Wally's zero stage bootloader (ZSBL) first reads the *globally unique identifier partition table* (GPT) information and copies the device tree and OpenSBI from the flash card into main memory. The GPT defines the location and size of partitions on the flash card. [Section 23.2.4](#) describes the bootloader in more depth. OpenSBI then uses the ZSBL to read the Linux kernel from the flash card into main memory and boot into Buildroot. To generate the flash card image, first create the flattened device tree, then partition the flash card and copy all the relevant files to it, as described next.

[Section 22.2.9](#) defines the device tree for virtual hardware emulation using QEMU. The Arty A7 uses the `cvw/linux/devicetree/wally-artya7.dts` device tree, which specifies a 20-MHz clock and 256-MiB main memory. This device tree uses the same addresses and interrupt IRQs from [Fig. 22.2](#) but adds the following sections:

A device tree is a human-readable description of a hardware configuration that defines clock speeds, memory mapped I/O addresses, main memory location and size, and many other features. See [Section 22.1.3](#) for more details.

```

refclk: refclk {
    #clock-cells = <0>;
    compatible = "fixed-clock";
    clock-frequency = <20000000>;
    clock-output-names = "xtal";
};

gpio0: gpio@10060000 {
    compatible = "sifive,gpio0";

    interrupt-parent = <0x03>;
    interrupts = <3>;
    reg = <0x00 0x10060000 0x00 0x1000>;
    reg-names = "control";
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
};
spi@13000 {
    compatible = "sifive,spi0";
    interrupt-parent = <0x03>;
    interrupts = <0x14>;
    reg = <0x0 0x13000 0x0 0x1000>;
    reg-names = "control";
    clocks = <&refclk>;

    #address-cells = <1>;
    #size-cells = <0>;
    mmc@0 {
        compatible = "mmc-spi-slot";
        reg = <0>;
        spi-max-frequency = <1000000>;
        voltage-ranges = <3300 3300>;
        disable-wp;
        // gpios = <&gpio0 6 1>;
    };
};

```

`refclk` defines a clock required by the SPI device. `gpio0` defines the two GPIO pins for CD and WP (GPIO pins 2 and 3, respectively), and `spi` defines the SPI device for the flash card. Finally, change `bootargs` to set the UART baud rate to 115,200.

```
bootargs = "root=/dev/vda ro console=ttyS0,115200";
```

If not already compiled, compile Buildroot with the options from [Section 22.2.1](#). Next, insert a flash card into the computer and copy the buildroot memory image, `$RISCV/buildroot/output/images/Image`, to the card using `flash_sd.sh`. The flash card will appear as an `sd{a, b, c, ...}` file in the `/dev` directory. To determine which one it is, type `ls /dev/sd*` before inserting the card and then `ls /dev/sd*` after inserting the card. The new device is the flash card.

Then run the `flash_sd.sh` script with the `/dev/sd<letter>` from above to write the flash card. This script creates a GPT and then 4 partitions. Partition 1 contains the device tree, partition 2 contains OpenSBI, partition 3 contains Buildroot, and partition 4 contains a mountable file ext4 file system.

```
$ cd $WALLY/linux/sdcard
$ ./flash-sd.sh -d wally-artya7.dtb /dev/sd<letter>
```

23.2.2.2 Synthesize Wally and Place and Route on FPGA

Vivado is an electronic design automation tool that translates RTL into a hardware configuration bitstream to program an AMD FPGA. To install Vivado, first download the latest version from Xilinx’s website (www.xilinx.com/support/download.html). At the time of writing, this was version 2023.1. Install on Linux using:

```
$ cd ~/Downloads
$ chmod +x Xilinx_Unified_2023.1_0507_1903_Lin64.bin
$ sudo ./Xilinx_Unified_2023.1_0507_1903_Lin64.bin
```

Then provide your AMD login and select install Vivado. For the free version, select Vivado ML Standard, or for the paid version, select Vivado ML Enterprise. Click next and agree to everything. Install it to your desired directory. After it is installed, click “OK” to finish.

The license manager will be active after installation. If you are using the free version, follow the instructions in the install wizard. If you have a site license, close the license manager and edit your `~/.bashrc` file to append a line pointing to the license:

```
$ export XILINXD_LICENSE_FILE=<port number>@<server name>
```

After the installation finishes, copy the board support files into the install directory.

```
$ sudo cp -r cvw/addins/vivado-boards/new/board_files <install path>/<version>/data/boards/
```

The next step is to synthesize the Wally RTL into a bitstream to program the FPGA using Vivado. On a fast computer, synthesis, place and route, and bitstream generation takes around 20 minutes. The process is automated using Makefiles to build Wally and the various AMD IP modules from [Section 23.2](#). The Makefile also optionally includes the *Integrated Logic Analyzer (ILA)*, which functions similar to a benchtop logic analyzer but probes internal FPGA wires. [Section 23.2.2.6](#) details how to use the ILA.

Vivado uses more than 90 GiB of disk space.

AMD does not require a paid license to use Vivado with the Artix-7 FPGA on the Arty A7 board. To access larger and faster FPGAs such as the Genesys 2’s Kintex 7 or the VCU’s Virtex Ultrascale, AMD requires a paid Vivado license.

The Makefile supports multiple development boards, which can be specified by name at the command line: `ArtyA7`, `vcu108`, or `genesys2`. The Arty A7 is the default and will be built if another target is not provided. To synthesize Wally to the Arty A7 FPGA development board:

```
$ cd $WALLY/fpga/generator
$ make // Or make <boardname> (i.e., make ArtyA7, make vcu108, etc.)
```

When synthesis completes, exit the Vivado prompt using:

```
$ exit
```

The Makefile begins by building the Vivado IP using the `*.tcl` scripts. For example, `_ddr3-ArtyA7.tcl` builds the memory controller IP for the Arty A7 board. Then it copies all the Wally source code from `cvw/src/` into `cvw/fpga/src/CopiedFiles_do_not_add_to_repo` and adds directives to preserve signal names during synthesis for debugging. The Makefile then calls the `wally.tcl` script to begin synthesis, place and route, and bitstream generation. `wally.tcl` uses constraint files defined in `cvw/fpga/constraints/`. The Arty A7 board uses `constraints-ArtyA7.xdc` and `small-debug.xdc`.

23.2.2.3 Program the FPGA

Now we are ready to configure the FPGA on the Arty A7 board as the Wally SoC. To do so, we first run Vivado in GUI mode using:

```
$ vivado &
```

Click on Open Project, select `WallyFPGA.xpr`, and click Ok (Fig. 23.4).

On the lefthand side of the screen is the Project Manager. At the bottom, click on Open Hardware Manager. Before programming the FPGA, remove the flash card from the computer and insert it into the Digilent PmodSD expansion board. Insert the expansion board into the Arty A7 board's right most Pmod socket (called port D), as shown in Fig. 23.3. Plug the USB cable into the FPGA and computer and click Open Target (Fig. 23.5) and then Auto connect.

Now program the FPGA by clicking Program Device (Fig. 23.6) and then clicking the popup FPGA part number. After a few seconds, the FPGA will be configured as the Wally SoC using the generated bitstream, and it will begin booting Linux.

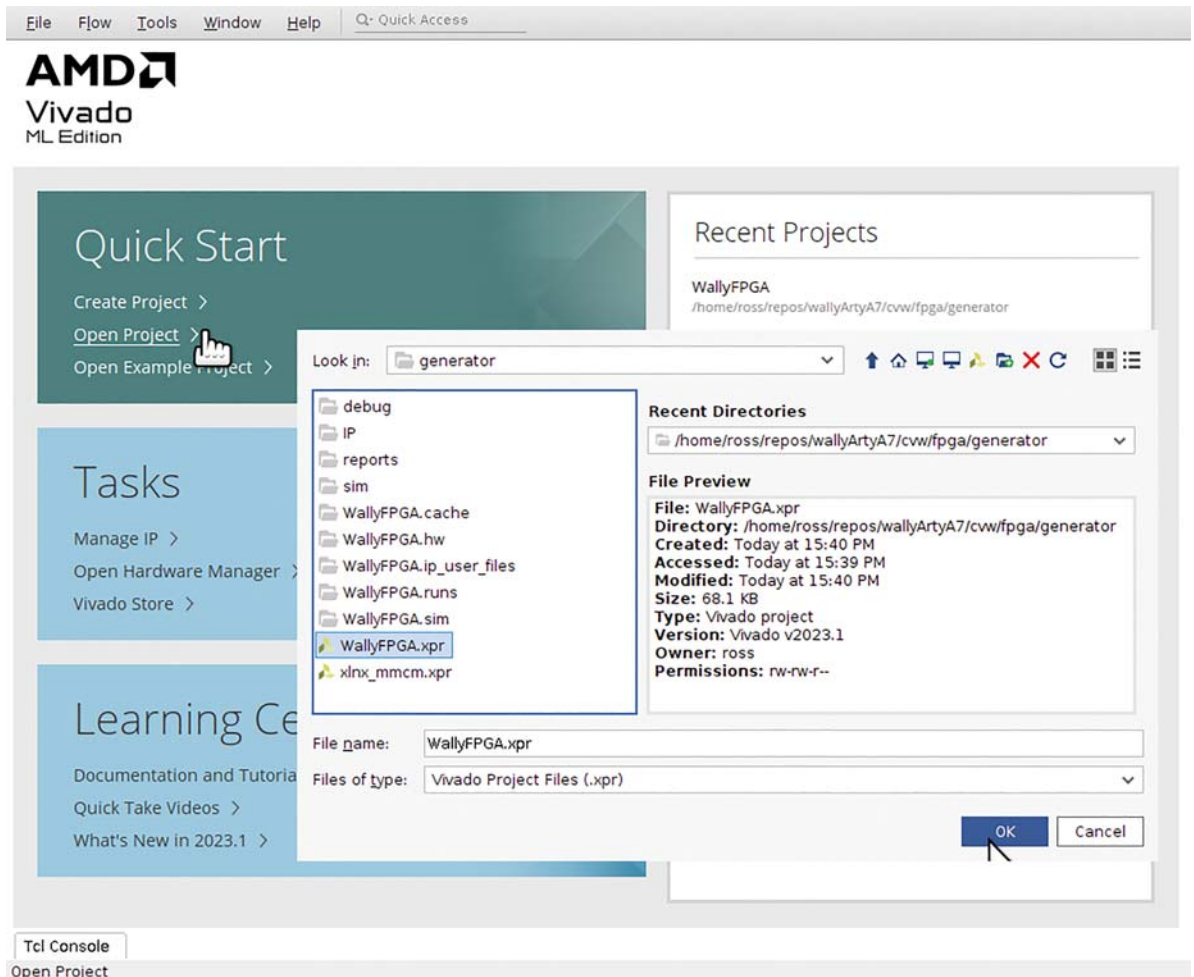


Fig. 23.4 Open Wally Vivado project. (Image courtesy of AMD. Reprinted with permission.)

23.2.2.4 Serial Terminal Over UART

As OpenSBI and Linux boot they display information over the UART serial console and eventually display a login prompt where the user can interactively issue commands in real time. This section describes how to set up the serial console, change baud rates and device permissions, and run programs on Wally.

The Arty A7 board's USB connection provides two devices: one for programming and debugging and the second as a USB-to-UART adaptor. A UART (see [Section 20.4](#)) is a full-duplex clockless serial communication standard that appears as a tty serial port to a Linux OS. Many different

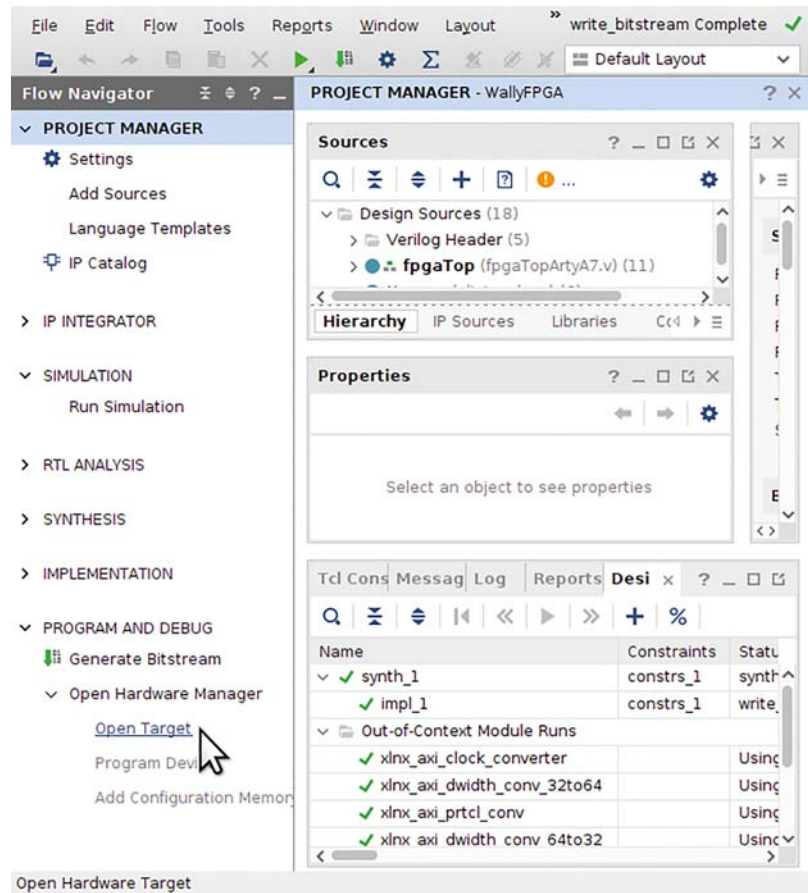


Fig. 23.5 Open Hardware Manager and Open Target. (Image courtesy of AMD. Reprinted with permission.)

A teletypewriter (tty) or teleprinter is a device that sends or receives typed characters. Originating as electromechanical devices in the 1800s, ttys allowed early computers to connect to remote systems through modems. Linux-based systems continue to use the tty for many serial connections including UART.

programs can interact with a tty serial port, but for this example, we use the program *screen*.

First, set the file permissions for the serial device. The UART device will appear as a `/dev/ttyUSB<number>`. Check the group permissions. The examples below assume the device number is `ttyUSB0`, but replace “0” with your device number.

```
$ ls /dev/ttyUSB0 -l
crw-rw---- 1 root dialout 188, 2 Jul 17 14:45 /dev/ttyUSB0
```

The fourth column is the group, and it should be `dialout`. If it is something else, change the group using:

```
$ sudo chgrp dialout /dev/ttyUSB0
```

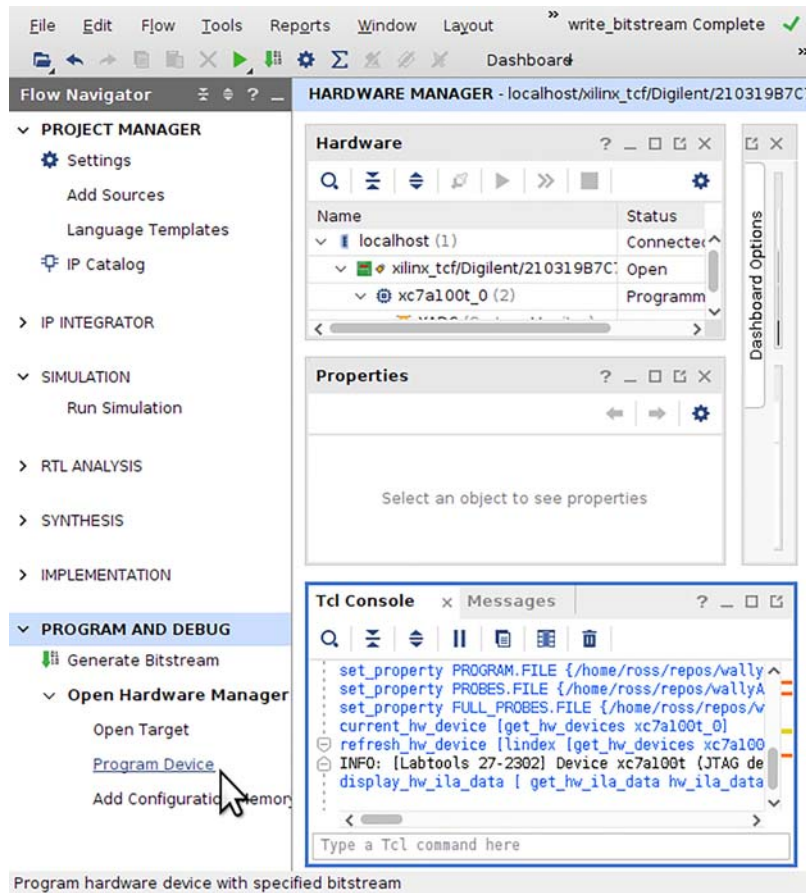


Fig. 23.6 Program Device. (Image courtesy of AMD. Reprinted with permission.)

Next, verify your user is part of the dialout group by typing (assumes username is wally):

```
$ groups
wally adm dialout cdrom sudo dip plugdev lpadmin lxd sambashare
```

If your user is not part of dialout, add it by typing:

```
$ usermod -a -G dialout <your username>
```

Now, use the following to connect to the Arty A7 over UART:

```
$ screen /dev/ttyUSB0 115200
```

Vivado uses JTAG over USB to program the FPGA's bitstream. Occasionally, Vivado incorrectly sets the JTAG clock frequency too high. The ILA specifies the JTAG clock should be no more than 2.5 times as fast as the debugger's clock frequency. Wally uses the same clock for the debugger as the main 20-MHz clock. If you cannot establish communication with the FPGA during programming, set the JTAG clock to 7.5 MHz, as shown in Fig. 23.7.

1. Select Program and Debug in the Project Navigator.
2. To the right of Flow Manager is Hardware. Expand localhost and select xilinx_tcf/Digilent/21031...
3. Go to the box below, Hardware Target Properties.
4. On the last line, select Properties.
5. Expand Param and change Frequency to 7500000.

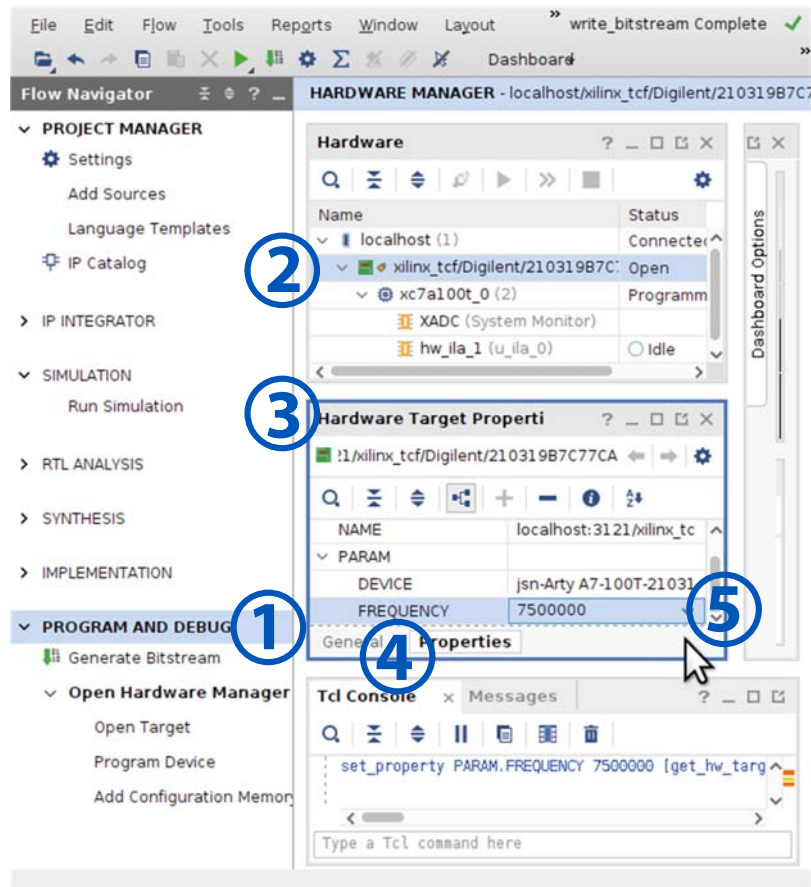


Fig. 23.7 JTAG settings. (Image courtesy of AMD. Reprinted with permission.)

115,200 is the baud rate, which was specified in the device tree. After about 40 seconds, OpenSBI prints a banner similar to Fig. 22.3. Over the next minute, Linux outputs boot messages and prompts the user for root login. The messages are similar to Fig. 22.3 except that the timestamps are different in hardware versus simulation.

The password is `root`. When logged in, the terminal behaves just like a conventional Linux system. It is possible to run text editors and benchmarks, write new programs, and save files to the flash card. Run the CoreMark benchmark with the following:

```

$ coremark
2K performance run parameters for coremark.
CoreMark Size      : 666
Total ticks        : 13298
Total time (secs)  : 13.298000
Iterations/Sec     : 45.119567Iterations      : 600
Compiler version   : GCC14.2.0Compiler flags : -D_LARGEFILE_SOURCE -
D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64 -Os -g3 -lrt
Memory location    : Please put data memory location here
                    (e.g. code in flash, data on heap etc)
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0xbd59
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 45.119567 / GCC14.2.0 -D_LARGEFILE_SOURCE -D_LARGEFILE64_SOURCE -
D_FILE_OFFSET_BITS=64 -Os -g3 -lrt / Heap

```

Wally's CoreMark performance in hardware (on an FPGA) is worse than in simulation for two reasons. First, CoreMark is not compiled with the same optimization described in [Sections 21.1.1](#) and [21.1.2](#). Second, the FPGA's clock is 20 MHz, and the Linux kernel takes a timer interrupt every 4 ms equal to $(20 \times 10^6 \text{ cycles/s}) \times (4 \times 10^{-3} \text{ s}) = 80 \times 10^3$ cycles. The average interrupt latency is 6,300 cycles, 7.9% of the CPU's time. From [Table 21.2\(b\)](#), CoreMark scores 2.83 in simulation on bare metal Wally rv64gc. When running on an FPGA, the interrupt alone reduces this to 2.61, and not using signed integers reduces it an additional 13% to 2.27, which is close to the measured 2.2.

Another interesting benchmark is `ramspeed`, which measures memory bandwidth for various size blocks. Notice how the bandwidth is higher for blocks that fit into the 16-KiB data cache. The theoretical maximum is $20 \text{ MHz} \times (8 \text{ bytes/cycle}) = 160 \text{ MB/s}$ because the D\$ supplies a full double word (8-bytes) on a hit.

```

$ ramspeed -b 2 -m 1 -g 1
RAMspeed (GENERIC) v2.6.0 by Rhett M. Hollander and Paul V.
Bolotoff, 2002-09

1Gb per pass mode

INTEGER & READING      1 Kb block: 138.23 MB/s
INTEGER & READING      2 Kb block: 141.25 MB/s
INTEGER & READING      4 Kb block: 140.70 MB/s
INTEGER & READING      8 Kb block: 138.67 MB/s
INTEGER & READING     16 Kb block: 123.22 MB/s
INTEGER & READING     32 Kb block: 30.74 MB/s
INTEGER & READING     64 Kb block: 30.74 MB/s
INTEGER & READING    128 Kb block: 30.49 MB/s
INTEGER & READING    256 Kb block: 30.47 MB/s
INTEGER & READING    512 Kb block: 30.47 MB/s
INTEGER & READING   1024 Kb block: 30.49 MB/s

```

`ramspeed` lists the block size in kibibytes (indicated as "Kb").

23.2.2.5 AHB Burst Performance

Section 12.2.1 introduced the load/store unit (LSU) with a D\$ that uses AHB burst cache line fetches. Burst mode dramatically improves memory bandwidth and, subsequently, Linux boot time. To test and measure Wally's performance without AHB burst, edit `$WALLY/src/ebu/buscachefsm.sv` by setting ``define BURST_EN 0`. Now run `ramspeed` again:

```
$ ramspeed -b 2 -m 1 -g 1
RAMspeed (GENERIC) v2.6.0 by Rhett M. Hollander and Paul V.
Bolotoff, 2002-09

1Gb per pass mode

INTEGER & READING      1 Kb block: 108.99 MB/s
INTEGER & READING      2 Kb block: 110.97 MB/s
INTEGER & READING      4 Kb block: 104.27 MB/s
INTEGER & READING      8 Kb block: 80.59 MB/s
INTEGER & READING     16 Kb block: 3.42 MB/s
INTEGER & READING     32 Kb block: 3.10 MB/s
INTEGER & READING     64 Kb block: 3.09 MB/s
INTEGER & READING    128 Kb block: 3.07 MB/s
INTEGER & READING    256 Kb block: 3.09 MB/s
INTEGER & READING    512 Kb block: 3.11 MB/s
INTEGER & READING   1024 Kb block: 3.13 MB/s
ramspeed -b 2 -m 1 -g 1
```

Data strides that don't fit into the cache (those ≥ 16 KiB, highlighted in blue font), show a dramatic drop in bandwidth, about 1/8th of the original. This configuration has 8 words per cache line, so one cache line fetch or writeback burst becomes 8 single transactions.

With burst, the Arty A7 boots Linux in 2 minutes and 17 seconds, but without burst, it takes 7 minutes and 43 seconds. The reason burst is so effective is that DRAM memories are optimized for large transfers but require many cycles of overhead to initiate a transaction. Burst amortizes that cost for all words in a cache line, thus reducing the average memory latency. It is worth revisiting Amdahl's law, which states: "Make the common case fast." Typically, we assume a program's working set will fit in the L1 cache. However, this assumption fails for the majority of `ramspeed`'s execution and, thus, results in overall poor performance. For comparison, CoreMark scores 1.875 CoreMarks/MHz without burst, just 22% slower than with burst (compared to `ramspeed`'s 7800% slower).

Table 23.3 ILA probes.

Signal Name	Width
PCM	XLEN
InstrValidM	1
InstrM	32
TrapM	1
IEUAdrM	XLEN
MemRWM	2

23.2.2.6 Integrated Logic Analyzer

After programming the FPGA with the Wally SoC, a new window to the right of the Hardware panel called `hw_ila_1` displays the ILA (Integrated Logic Analyzer) interface. Press the double arrow (`>>`) button (Fig. 23.8) to capture a sample of the processor state. Table 23.3 lists the signals the

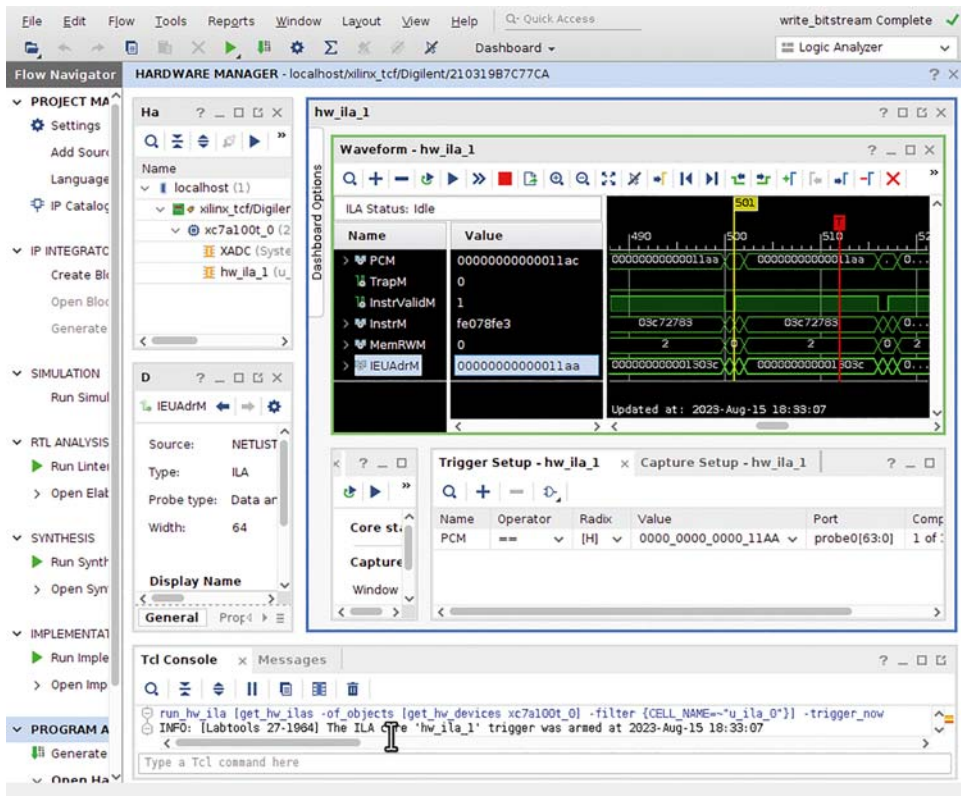


Fig. 23.8 ILA trigger menu and setup. (Image courtesy of AMD. Reprinted with permission.)

ILA captures in the Memory stage. Below the Hardware box is a trigger configuration. Click the plus icon to add a signal to the trigger configuration. For example, select PCM and set the hex value to 0x0000_0000_0000_1186, an instruction address in the ZSBL. Then press the triangle play button to wait until the trigger occurs.

The ZSBL reads the GPT partition header from the flash card to find the locations of the partitions containing the device tree, OpenSBI, and the kernel. It copies OpenSBI and Linux to DRAM using DMA transfer directly from the SD card's manager without using the CPU. However, the bootloader does not need to do other work during this time, so it implements a busy loop to continuously poll the SD card's interrupt flag. Cross-referencing the program counter with the ZSBL's object dump in `cvw/tests/custom/boot/bin/boot.objdump` indicates that 0x11AA is a load, and 0x11AC is a branch. The load reads the interrupt flag from the SD card's I/O register, and the branch checks if the flag says the

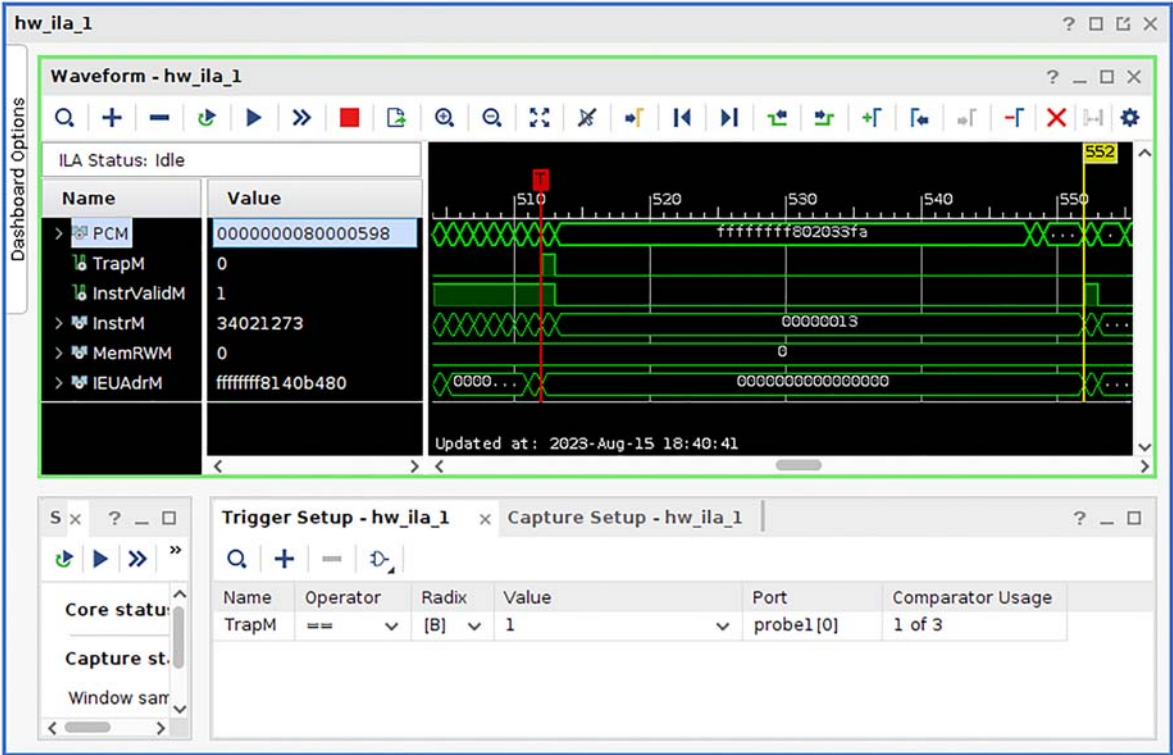


Fig. 23.9 Example OpenSBI TrapM. (Image courtesy of AMD. Reprinted with permission.)

DMA transfer has completed, looping back to read the register if the transfer is not finished.

After about 40 seconds, the transfer completes and the system begins executing OpenSBI and eventually Linux. Again, the ILA can be useful to see what instructions the CPU is running. The trigger setup can be changed to select any address in the program. Another interesting trigger option is `TrapM`. Any time this becomes 1, the CPU takes a trap and PCM changes to the M-mode trap handler at `0x80000598` several cycles later (Fig. 23.9). Once OpenSBI completes, it starts booting the Linux kernel, and the instruction address ranges change to `0xFFFF_FFFF_8xxx_xxxx`. The OpenSBI object dump is in `$RISCV/Buildroot/output/images/fw_jump.objdump`. The Linux object dump is in `$RISCV/Buildroot/output/images/vmlinux.objdump`.

The ILA uses block RAM (BRAM), LUTs, and flops. The Arty A7 supports a small number of useful signals in the ILA, but it is sometimes necessary to add other signals. This requires two steps. First, the logic must be annotated with `(* mark_debug = "true" *)` in the source code

so the optimizer knows not to remove the net from the netlist. Because adding `mark_debug` throughout the Wally RTL is distracting, instead copy the source code into `$WALLY/fpga/src/CopiedFiles_do_not_add_to_repo/`. Then the FPGA Makefile calls `insert_debug_comment.sh`, which uses `$WALLY/fpga/constraints/marked_debug.txt` to add the `mark_debug` directives to the copied files. Second, the ILA is configured to probe the net using the following Tcl commands.

```
create_debug_port u_ila_0 probe
set_property port_width 1 [get_debug_ports u_ila_0/probe5]
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe5]
connect_debug_port u_ila_0/probe5 [get_nets [list
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/LSUHREADY ]]
```

These Tcl commands are most easily added to a debugger ILA file in the `cvw/fpga/constraints` directory. The Arty A7 uses `small-debug.xdc`. The probe<number> needs to be incremented for each additional port, and the port width must exactly match the number of bits in the connected signal. A multiple-bit bus must have each bit explicitly enumerated. See `small-debug.xdc` for an example.

The `proberange` Python script in the `cvw/fpga/constraints` directory automates the laborious task of generating these debug scripts. `proberange` takes two arguments: a file with a list of signals and an output file. `proberange` generates the four ILA probe commands. The first file is formatted as `<signal_name width probe_number>`. The signal name must match the exact path in the instance hierarchy.

For example, to add the `LSUHADDR` bus as probe number 6:

1. Edit `marked_debug.txt` to include `lsu/lsu.sv: logic LSUHADDR`.
2. Create a new file in the constraints directory called `signals.txt`, which contains the following line:
`wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/LSUHADDR 64 6`
3. Run `proberange`.
`./proberange signals.txt new-probe.xdc`
4. Copy the contents of `new-probe.xdc` into `small-debug.xdc`.
5. Rerun synthesis.

23.2.3 Supporting Other Boards

The Arty A7 board is a relatively affordable platform compared to larger and more expensive FPGAs found on boards such as the VCU108 and VCU118. However, Wally uses nearly all the FPGA resources on the

A loss leader is a business strategy where a product is sold at a loss with the goal of enticing customers to buy other more expensive products. Milk, bread, and eggs are common loss leaders.

Arty A7 and is limited to a 20-MHz clock. Other FPGAs enable the possibility of multicore, larger caches, accelerators, and higher clock speeds. This section describes how to target Wally to UltraScale's VCU108 and VCU118. The steps can be generalized to support any new board.

The VCU108 contains a very large and fast AMD Vertex UltraScale XCVU095-2FFVA2104E FPGA. Unlike the Arty A7 board, the VCU boards are targeted to research and industry as loss leaders to get engineers comfortable using expensive FPGAs. The FPGA board costs an impressive \$21,000, but AMD subsidizes the board cost down to \$7,700 to attract new customers. Similarly, the VCU118 is based around the newer UltraScale+ XCVU9P-L2FLGA2104E chip, which costs an eye-dropping \$59,000 on its own, but because of AMD discounts, the board (including the FPGA) costs just \$9,099.

Both boards still include PMOD (peripheral module) connectors for the flash card and USB-to-UART converter chips, but they also add 5 GiB of DDR4 DRAM and many fast peripheral devices. Five main changes are required to support the new boards:

1. Board files
2. New top-level RTL
3. Memory controller
4. Clocking and timing constraints
5. I/O pin mappings
6. Updating the device tree and ZSBL

The majority of commercial FPGA boards come with templates that define the FPGA type and size, pin mappings for some I/O devices, and board clocks. The VCU boards' files come with Vivado. Many of the Digilent boards are found in `addins/vivado-boards` (from the github.com/Digilent/vivado-boards repository) and were installed as part of [Section 23.2.2.2](#). After the board files are installed, we can set a `partNumber` and `boardName` in the `cvw/fpga/Makefile`. For example, the VCU118's FPGA and `partNumber` is `xcvu9p-flga2104-2L-e` and its `boardName` is `xilinx.com:vcu118:part0:2.4`.

The VCU boards have a different enough architecture and naming convention for ports and other board connections that they require a new top-level Verilog module `fpgatopvcu.sv`. For example, whereas the Arty A7 board uses a 256-MiB DDR3, the VCU boards use DDR4. To use the full 5 GiB, two memory controllers are required. As of this time, only one memory controller is implemented, and it is configured to use only 2 GiB of memory. Make sure to update `cvw/config/fpga/config.vh` with the new memory range. Unlike the DDR3 MIG, the DDR4 MIG

for the UltraScale and UltraScale+ takes a single input clock and can generate multiple output clocks. Consequently, the MMCM IP is not needed. As with the Arty A7 board, the memory controller typically runs at a higher clock speed than the CPU, so the AXI clock converter is still required to support this multi-clock domain design. Depending on the IP and the specific FPGA, an extra MMCM may be required for some designs as the MIG may or may not contain additional clock output ports.

To change the clock speed on the Arty A7, edit `cvw/fpga/generator/xlnx_mmcmm.tcl` by replacing 20 with another frequency in MHz.

```
CONFIG.CLKOUT3_REQUESTED_OUT_FREQ {20}
```

Similarly, the VCU boards change their CPU clock speed using `cvw/fpga/generator/xlnx_ddr4.tcl`.

```
CONFIG.ADDN_UI_CLKOUT1_FREQ_HZ {71}
```

Each FPGA board has unique pin mappings between the FPGA I/O and peripherals. The `cvw/fpga/constraints/` directory contains `xdc` (Xilinx design constraint) files for each board that map the I/O pins. For example, in `constraints-vcu118.xdc` the following line maps FPGA pin L31 to the top-level Verilog module input `UARTSin`.

```
set_property PACKAGE_PIN L31 [get_ports UARTSin]
```

The board's datasheet specifies how the FPGA pins are connected on the board. The VCU108 and VCU118 user guides can be found on AMD's website.

Multiple bugs in the VCU118 board exist, which cause issues in both the on-board USB to UART adaptor and the Digilent PmodSD board. To work around the UART issue, SparkFun makes the DEV-09873 (USB to serial breakout) board, which happens to fit nicely into the male Pmod port. The 3.3-V version is required. It is inserted into the rightmost column of pins starting at the second from bottom. (see [Fig. 23.3](#)). The PmodSD board is a bigger problem. The root cause of the problem is that the SD card's level shifter is incompatible with the board. The VCU118 level shifter assumes the daughter boards provide strong pull-up resistors less than 300 Ω ; however, the PmodSD board has a different level shifter and uses 10-k Ω pullups. We built custom SD card boards with 300- Ω pullup resistors. Even with this change, the SD card controller cannot run faster than about 11 MHz without errors.

Finally, the device tree and ZSBL (see [Section 23.2.4](#)) need to be updated to reflect the changes in the size of memory, clock speeds, and any new peripherals. The clock needs to be updated for all devices in the

The design files for this custom SD board are located in `cvw/fpga/boards/pmodsd/`.

tree. The VCU108 and VCU118 have their own device trees located in the `cvw/linux/device` directory. The bootloader likewise needs to know the clock frequency, so it can set the correct flash card clock frequency. This is set in `cvw/tests/custom/boot/boot.c` as the macro `#FREQ` near the top of the file.

The bootloader must be recompiled with `make` and the memfile, `cvw/tests/custom/boot/bin/boot.memfile`, must be copied to `cvw/fpga/src/boot.mem`.

23.2.4 Wally Zero Stage Bootloader

The ZSBL, also referred to simply as the bootloader, is system dependent because it must manage peripheral-dependent features such as the SD card clock. Wally’s ZSBL is located in `cvw/fpga/zsbl/`. The ZSBL reads GPT partition 1 to find the location and size of the device tree, copying it into Wally’s RAM at `0x87000000`. Partition 2 contains the OpenSBI bootloader and firmware called `FW_JUMP`, described in [Section 22.2.4](#). The ZSBL copies `FW_JUMP` into RAM at `0x80000000`. Next, it copies the Linux kernel (see [Section 22.2.5](#)) from partition 3 to `0x80200000`. Finally, it flashes LED0 5 times by driving the `GPIO[0]` pin and jumps to `0x80000000` to start the OpenSBI bootloader.

23.3 FPGA RESULTS

Wally supports three development boards, the Arty A7 described in [Section 23.2.2](#), the UltraScale VCU108, and the Genesys 2. The VCU108 runs Wally at 50 MHz and has a 2-GiB DDR4 memory and plenty of additional CLBs, registers, and block RAM to support a larger ILA, multicore, more peripherals, and many extra features. The Genesys 2 runs Wally at 40 MHz and has a 1-GiB DDR3 and fewer resources than the VCU108. [Table 23.4](#) summarizes the system specifications of Wally compiled for these boards, and [Table 23.5](#) compares their resource usage. The Total Usage rows show the overall resource usage for the entire design including the `mmcm` clock generator, `memctrlddr3` memory controller, `ahbaxibridge`, and

Table 23.4 Wally FPGA specifications.

Board	Clock Speed	Memory	Linux Boot Time
Arty A7	20 MHz	256 MiB DDR3	3 min 54 sec
VCU108	50 MHz	2 GiB DDR4	1 min 36 sec
Genesys 2	40 MHz	1 GiB DDR3	1 min 58 sec

Table 23.5 Wally FPGA resource usage.

Board	LUTs	Registers	Block RAM	DSP
Arty A7 Max Available	63,400	126,800	135	240
Arty A7 Total Usage	52,122 (82%)	39,122 (30%)	92.5 (69%)	25 (10%)
Arty A7 wallypipelinedsoc	40,677 (64%)	23,661 (19%)	84 (62%)	25 (10%)
VCU108 Max Available	537,600	1,075,200	1,728	768
VCU108 Total Usage	64,962 (12%)	55,039 (5%)	78 (5%)	28 (4%)
VCU108 wallypipelinedsoc	43,522 (8%)	23,444 (2%)	44 (3%)	25 (3%)
Genesys 2 Max Available	203,800	407,600	445	840
Genesys 2 Total Usage	60,547 (30%)	47,245 (12%)	118 (27%)	25 (3%)
Genesys 2 wallypipelinedsoc	41,581 (20%)	23,757 (6%)	84 (19%)	25 (3%)

other hardware shown in Fig. 23.2. The Max Available rows show the total FPGA resources. The tweaked rv64gc configuration (wallypipelinedsoc) fits comfortably into the Arty A7 using about two thirds of the logic and BRAM and one fifth of the registers leaving enough room for the ancillary logic. This build includes an ILA with PCM, InstrM, InstrValidM, IEUAdm, ReadDataM, WriteDataM, MemRWM, and TrapM logging 4,096 clock samples. After wallypipelinedsoc, memctrllddr3 consumes the most resources.

The VCU108 has nearly 10 times as many resources as the Arty A7 and is based on the faster UltraScale FPGA architecture enabling more than double the clock speed. The board also has two DDR4 memories each supporting 2.5 GiB. However, recall that only one controller is implemented, which supports 2 GiB. The DDR4 memory controller uses more resources than the DDR3 memory controller. wallypipelinedsoc uses a tiny fraction of the VCU108's resources. This enables the use of a much larger ILA for faster debugging. Adding more signals to the ILA increases LUT, register, and BRAM usage. Increasing the sample length increases the BRAM. With the VCU108 it is possible to log hundreds of signals for 65,536 clock samples.

23.3 INTERESTING BUGS

FPGAs aid ASIC verification by emulating and validating RTL functionality in a much higher-performance environment compared to simulation. Wally's FPGA implementation found numerous functional and

performance bugs that the standard regression tests did not. This section highlights two of these bugs and ends with critical path analysis, showing how we improved clock speed by 33%.

23.3.1 Bug 1: AHB Bursts

The earliest version of Wally did not use AHB's burst mode to fill cache misses and instead issued separate XLEN-bit read or writes for each word in the cache line. Although it did work, the performance was poor, taking more than 5 minutes to boot Linux using the FPGA. When we added burst mode, Wally passed all regression tests, but the FPGA implementation failed. When booting Linux, it did not display the OpenSBI banner and then got stuck in an infinite loop. Unfortunately, nothing obvious, such as an illegal instruction, provided a good starting point for investigation.

We used the ILA to record a short 16,384 clock sample of Wally's internal state. After failing to boot, we noticed Wally was frequently in the Linux trap handler. Eventually, we found an instruction in the trap handler was causing another trap into OpenSBI M-mode. This instruction generated an unexpected load-access fault. We set up an ILA trigger on this instruction and `TrapM = 1` and restarted Wally to catch the first instance of this instruction faulting. Using an object dump of the Linux source code, we found the access fault's address. Because the ILA only captures 16,384 clock cycles, backtracking to the root cause may not be visible within a single ILA capture. This required reconfiguring the ILA multiple times and restarting the whole boot process repeatedly before finding the timeframe that showed a discrepancy between data written to memory and data read from memory, which led to the hardware bug described next.

The external bus unit (EBU) relied on three main assumptions about the instruction fetch unit (IFU) and LSU memory requests.

1. The IFU makes an AHB request and finishes before another request is made.
2. The LSU makes an AHB request and finishes before another request is made.
3. Or both make an AHB request on the same cycle, and the LSU is given precedence.

The FPGA challenged these assumptions by showing it was possible for the LSU to make an AHB request while the IFU already had an AHB request in flight. If the I\$ and data translation lookaside buffer (DTLB) miss in the same cycle, but the hardware page table walker's (HPTW's) first memory request to the D\$ hits, the I\$ will be granted access to the AHB. If later HPTW requests miss in the D\$, the LSU attempts

to make a request while the IFU already has access to the bus, thus violating the above assumptions and interrupting the IFU's burst request. The nonburst implementation worked because only one word was transferred at a time, so it was possible for the IFU to have transferred part of the cache line and then have the EBU give the LSU bus access. Then, after the LSU completed, the IFU was again given bus access to finish its requests. However, with burst mode enabled, the LSU could preempt the IFU's AHB request with only part of the I\$ line transferred.

The ILA has an advanced trigger mode called trigger state machine, which allows for sequenced triggers. This allowed us to check the above assumption by first having the IFU make a memory request and then having the LSU make a request. If the trigger occurred, then we knew the assumption failed, and it would also provide additional processor state at the time of the failure.

The `ahb_ram` tests use variations of the `rv64gc` configuration and run the floating-point `fadd_b11-01` test. These enable or disable burst mode and change latencies between the caches and main memory to be 0, 1 or 2 cycles, which latencies are designed to specifically trigger this scenario.

23.3.2 Bug 2: Concurrent Exceptions

The Linux boot involves an M-mode timer interrupt with a concurrent instruction page fault. Because the page fault is handled in S-mode, the M-mode timer interrupt has higher priority and should be handled first; however, some really strange behavior emerged. `mcause` oscillated between `0x800...007` (machine timer interrupt) and `0x7` (store access fault), settling on `0x7` after several cycles. Then, the IFU fetched an invalid address that was neither the `mepc` or `sepc`.

Just like the previous bug, this scenario never occurred in the regression test but happened during a Linux boot on the FPGA. The oscillating `mcause` pointed to a possible root cause. `Committed{F/M}` indicate the IFU/LSU are processing an in-flight bus transaction. When either is high, it prevents a `TrapM` and would also clear the msb of `mcause`. The LSU correctly disables bus access on `TrapM`; however, the IFU did not. In the first cycle of the trap, `mcause` was set correctly to `0x800...007`, but because the bus fetched data from an invalid region due to a branch misprediction, `CommittedF` raised, which forced `InterruptM` to go low and made the trap look like an exception (`0x7`). The solution was to disable bus accesses on `TrapM`.

23.3.3 Critical Path Analysis

Recall that the critical path is the longest combinational logic delay between two flip-flops and that it sets the maximum clock speed of a

synchronous design. As the critical path decreases, the clock speed increases, boosting performance. A design benefits from tuning to boost clock speed on a target FPGA.

The first Wally version to boot Linux on the Arty A7 FPGA only ran at 15 MHz or a critical path of $1/15 \text{ MHz} = 66.7 \text{ ns}$. By analyzing the timing paths, it was possible to increase the clock speed to 20 MHz with some simple RTL modifications.

Instructions spanning two virtual pages make two ITLB accesses. The $\text{PC} + 4$ adder that increments the virtual address is part of the unoptimized critical path. Fig. 23.10 shows a snippet of the Vivado timing report before optimization. As highlighted in blue, the critical path begins at the PCF register `pcreg/q_reg[3]/Q` and passes through the $\text{PC} + 4$ adder `PCPlus4F_inferred_i_4/0[0]`, through the spill logic (`PCSpillF`), and into the HPTW. The HPTW uses `PCPlus4F` to walk the page table on a TLB miss and produce `HPTWAdr`. This greatly increases the critical path by placing the $\text{PC} + 4$ adder at the start of the path. However, this incremented address is not required in the same cycle. This is called a *multicycle path*, which can be resolved in two ways. First, a register in the HPTW samples $\text{PC} + 4$ to prevent the path from existing. Second, we write a multipath timing constraint which specifies this path takes 2 cycles. The

```
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/pcreg/q_reg[3]/Q
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/PCF[3]
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/PCPlus4F_inferred_i_16/C0[3]
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/PCPlus4F_inferred_i_16_n_0
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/PCPlus4F_inferred_i_15/C0[3]
...
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/PCPlus4F_inferred_i_5/C0[3]
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/PCPlus4F_inferred_i_5_n_0
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/PCPlus4F_inferred_i_4/0[0]
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/Spill.spill/PCPlus4F[51]
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/Spill.spill/PCSpillF[51]_INST_0/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/vadrmux/d0[51]
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/vadrmux/y[51]_INST_0/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/y[39]
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[55]_INST_0_i_26/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[55]_INST_0_i_26_n_0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[55]_INST_0_i_10/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg_n_68
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/ITLBWriteF_inferred_i_4/0
...
```

Fig. 23.10 HPTW critical path before optimization.

```

wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/Spill.spill/CurrState_reg[0]/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/Spill.spill/pcplus2reg/out[0]
wallypipelinedsocwrapper/wallypipelinedsoc/core/ifu/Spill.spill/pcplus2reg/PCSpillF[46]_INST_0/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/vadrmux/d0[46]
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/vadrmux/y[46]_INST_0/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/y[34]
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[55]_INST_0_i_29/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[55]_INST_0_i_29_n_0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[55]_INST_0_i_12/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
q_reg[41]_0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[55]_INST_0_i_3/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
q_reg[47]_0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[55]_INST_0_i_1/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[55]_INST_0_i_1_n_0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/hptw.hptw/hptwwrites.HPTWAdrWriteReg/
IHAdrM[26]_INST_0/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/dmmu.dmmu/tlb.tlb/tlbcam/camlines[26]/
keyflop/0[14]
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/dmmu.dmmu/tlb.tlb/tlbcam/camlines[26]/
keyflop/PhysicalAddress[47]_INST_0_i_177/0
wallypipelinedsocwrapper/wallypipelinedsoc/core/lsu/dmmu.dmmu/tlb.tlb/tlbcam/camlines[26]/
keyflop/PhysicalAddress[47]_INST_0_i_177_n_0

```

Fig. 23.11 HPTW critical path after optimization.

first option is easier but consumes extra hardware. It also has the advantage of carrying over to the ASIC implementation. The second option uses less hardware but is hard to maintain if the hardware is updated.

Fig. 23.11 shows the impact of adding the extra register to the timing path. The starting point is no longer the PCF register, so producing PCSpillF is a shorter path and no longer requires going through the PC + 4 adder.

By repeatedly iterating over the timing report and updating the RTL, we increased the FPGA's clock speed from 15 to 20 MHz. The general strategy checks if late arriving combinational inputs can be moved later into the critical path. For example, the LSU cancels bus and cache requests by lowering BusRW and CacheRW, respectively, during a trap. An early RTL implementation first fed TrapM into the cache fsm which then output BusRW and CacheRW, resulting in very long paths. We refactored the code, so these late arriving signals do not go through the cache fsm logic, which then reduced the critical path.

23.4 SUMMARY

This chapter has shown how to implement Wally on AMD FPGAs ranging from a midrange Digilent Arty A7-100T board or a high-end AMD VCU108 FPGA evaluation board. The `fpgatopArtyA7` SystemVerilog module instantiates the `wallypipelinesoc`, along with clock and reset circuitry, an AXI bridge to a DDR DRAM memory controller, an SD card controller for an on-board flash memory, and a UART-to-USB interface to communicate with a terminal on a host computer.

The Wally ZSBL reads the OpenSBI bootloader, device tree, and Linux kernel from the SD card into DRAM and then jumps to the OpenSBI bootloader. The remainder of the OS boot is the same on an FPGA as in simulation.

FPGAs are invaluable tools in verification and performance analysis. Using the higher performance of FPGAs, Wally is able to execute hundreds of billions of instructions faster to find bugs the standard regression test failed to uncover in simulation. FPGAs tradeoff execution speed for internal signal visibility, making debugging harder when a bug is detected. Finally, the FPGA implementation aided in identifying key critical path issues without the effort of going through an ASIC synthesis and place and route step.

Wally Synopsis



This appendix provides a summary of the Wally implementation, including the following:

- ▶ Organization of the Wally repository
- ▶ Configurations, memory map, configuration parameters, and profiles
- ▶ Regression test suites
- ▶ Key microarchitectural diagrams

The microarchitectural diagrams in this appendix are found in the chapters, but they are also duplicated here for convenience.

A.1 WALLY REPOSITORY

Fig. A.1 lists the overall organization of the CORE-V Wally repository (cvw). The repository is described further in Section 3.2.

- A.1 **Wally Repository**
- A.2 **Wally Configurations**
- A.3 **Wally Regression and Test Suites**
- A.4 **Key Wally Commands**
- A.5 **Wally Diagrams**

The Wally GitHub repository is located at github.com/openhwgroup/cvw.

A.2 WALLY CONFIGURATIONS

Table A.1 lists the main Wally configurations introduced in Section 2.8.2, with their main features and figures of merit from TSMC 28nm synthesis. Table A.2 provides the default memory map for configurations that have peripherals. Table A.3 lists key Wally configuration parameters that vary from one configuration to another, and Table A.4 lists other microarchitectural parameters that have common values in the Wally configurations that need them but could be changed (including the base and range of the memory map). Table A.5 lists derivative configurations (from those listed in Tables A.1–A.4) that are needed for various platforms, synthesis, or simulation. Table A.6 lists the ratified RISC-V features (as of the time of this writing), and the ones that are required or optional in each profile.

The base and range parameters specify the memory map, with default values given in Tables A.2 and A.4. Note that the DTIM and IROM ranges of 0x7FFFFFFF in Table A.4 are chosen for testing only; they are large enough to accommodate the largest riscv-arch-test case for simulation but are far too large to synthesize as a realistic tightly integrated memory. Table A.1 gives the default IROM and DTIM sizes for the main Wally configurations and Table A.2 gives their maximum sizes for these configurations. Section 6.5 describes how memory and branch predictor configurations are downsized for logic synthesis experiments.

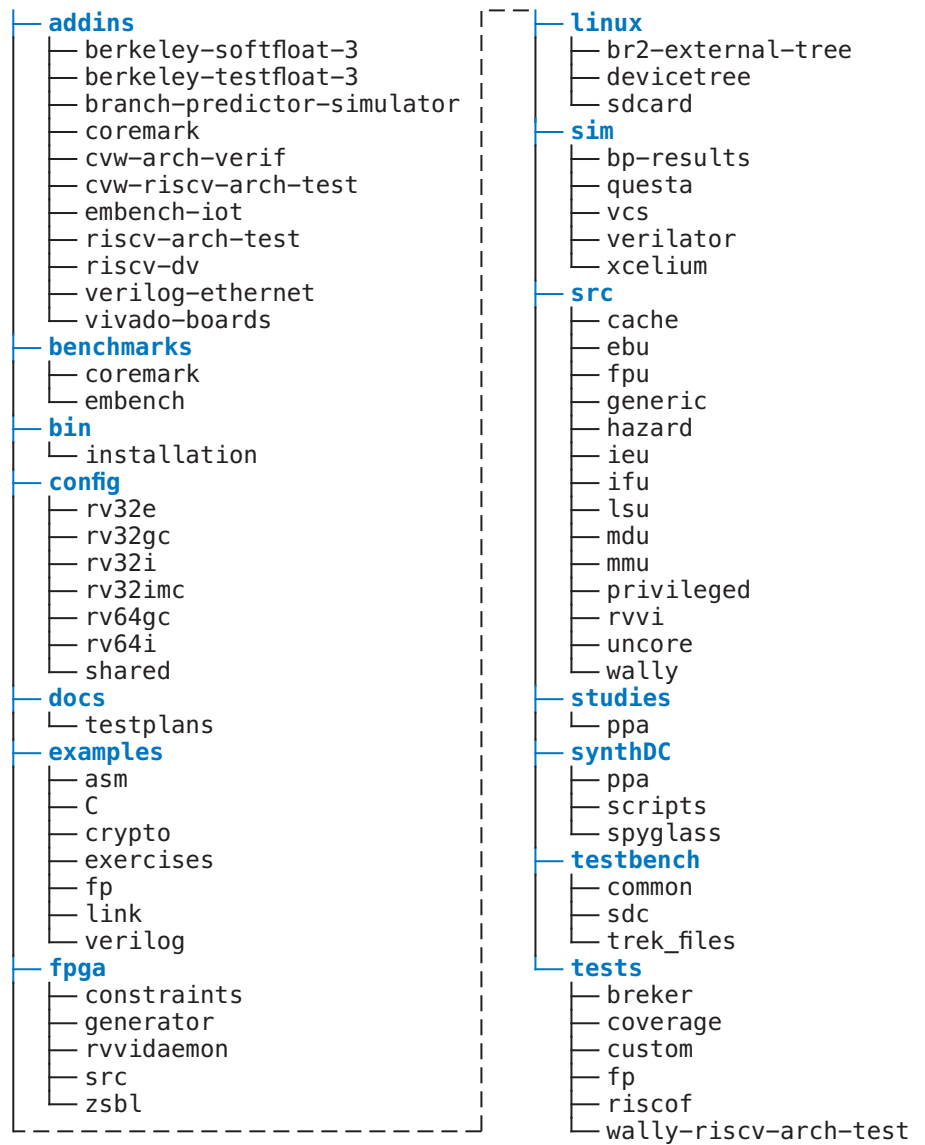


Fig. A.1 Wally (cvw) directory structure.

Table A.1 Wally configurations.

Configuration	Config	XLEN	Profile	DTIM / IROM Size	Bus	Periph	IS / D\$ Size	Privilege Modes	Virt Mem	Max Freq (GHz)	Synthesis Results (TSMC 28 nm)	
											Logic Area (mm ²)	Mem Area (mm ²)
Embedded	rv32e	32	N/A	N/A	YES	NO	N/A	none	NO	2.99	0.007	
Simple CPU	rv32i	32	RVI20U32	2 KiB / 2 KiB	NO	NO	N/A	none	NO	2.58	0.023	
Microcontroller	rv32imc	32	RVI20U32	4 KiB / 16 KiB	YES	YES	N/A	MU	NO	2.18	0.045	
Apps Processor	rv32gc	32	RVI20U32	N/A	YES	YES	16K	MSU	YES	0.89	0.119	0.314
Simple CPU	rv64i	64	RVI20U64	2 KiB / 2 KiB	NO	NO	N/A	none	NO	2.52	0.032	
Apps Processor	rv64gc	64	RVA22S64	N/A	YES	YES	16K	MSU	YES	0.80	0.176	0.314

Table A.2 Wally default memory map.

Memory/Device	Start Address	End Address	Size
RAM	80000000	Up to 87FFFFFF	Up to 128 MiB
DTIM	80000000	Up to 80003FFF	Up to 16 KiB
IROM	80000000	Up to 80000FFF	Up to 4 KiB
GPIO	10060000	100600FF	256 B
SPI	10040000	10040FFF	4 KiB
UART	10000000	10000007	8 B
PLIC	0C000000	0FFF0000	64 MiB
CLINT	02000000	0200FFFF	64 KiB
SDC	00013000	00013FFF	4 KiB
Boot ROM	00001000	00001FFF	4 KiB

In addition to the base configurations, Wally generates derivative configurations for verification and synthesis, as described in [Section 4.4](#), and for various platforms, such as an FPGA, as described in [Chapter 23](#). The derivatives are variations on base configurations created by changing

See wiki.riscv.org/display/HOME/Ratified+Extensions for the latest list of ratified extensions.

certain parameter values. Table A.5 lists each derivative and the associated base and changes.

Table A.6 summarizes the RISC-V extensions and profiles that were ratified at the time of this writing. If an extension is supported in at least one of the Wally configurations, the table indicates which section describes the support. If the extension is required (R) or optional (O) for a particular profile, it is marked in the corresponding column. In particular, the rv64gc profile supports all the required extensions for RVA22S64.

A.3 WALLY REGRESSION AND TEST SUITES

Section 5.11 describes running regression, applying each relevant test suite to each configuration. Table A.7 lists the regression test suites run on each configuration. The riscv-arch-test suite is steadily growing, and the latest list of regression tests can be found in `bin/wally-regression`.

The test suites used for the regression tests are listed in Table A.8. They are run using `testbench.sv` in the `$RISCV/testbench` folder.

Another testbench (`testbench_fp.sv`) exercises floating-point units in isolation for better performance, as listed in Table A.9.

A.4 KEY WALLY COMMANDS

Table A.10 lists commonly used commands. They assume you have created an account with a username `<yourgithubaccount>` at github.com and forked `openhwgroup/cvw` to `<yourgithubaccount>`.

To make a pull request to submit changes back to the main repository:

```
$ git pull upstream main
$ git add <filenames>
$ git commit -m "Describe edits"
$ git push
```

At github.com/openhwgroup/cvw, click New pull request and choose the fork.

A.5 WALLY DIAGRAMS

Table A.11 lists the high-level diagrams for Wally's main units, which are found throughout the book. It also gives their module names, brief descriptions, and their original figure number in the textbook. Those figures are then shown after Table A.11, for convenience.

Table A.3 Key Wally configuration parameters.

Parameter	Meaning	Section	rv32e (embedded)	rv32i (simple)	rv32imc (μcontroller)	rv32gc (apps proc)	rv64i (basic)	rv64gc (apps proc)
XLEN	Architecture width	2	32	32	32	32	64	64
E_SUPPORTED	16 registers instead of 32	2.5	1	0	0	0	0	0
ZIFENCEI_SUPPORTED	fence.i supported	8.4.5.5	0	0	0	1	0	1
ZICSR_SUPPORTED	CSRs supported	8.4.1	0	0	1	1	0	1
ZICCLSM_SUPPORTED	Misaligned load/store	19.2	0	0	0	0	0	1
ZICOND_SUPPORTED	Conditional instructions	19.5	0	0	0	1	0	1
M_SUPPORTED	Multiply and divide	15	0	0	1	1	0	1
ZMMUL_SUPPORTED	Multiply only	15.2.4	0	0	1	1	0	1
ZAAMO_SUPPORTED	AMO atomics	17.2.4	0	0	0	1	0	1
ZALRSC_SUPPORTED	Lr-sc atomics	17.2.4	0	0	0	1	0	1
ZB{A/B/C/S}_SUPPORTED	Bit manipulation	18.2	0	0	0	1	0	1
ZBK{B/C/X}_SUPPORTED	Crypto bit manipulation	18.2	0	0	0	1	0	1
ZKN{D/E/H}_SUPPORTED	Crypto NIST algorithms	18.2	0	0	0	1	0	1
ZCA_SUPPORTED	Basic compressed	14.1.1	0	0	1	1	0	1
ZCF_SUPPORTED	Float compressed	14.1.1	0	0	0	1	0	0
ZCD_SUPPORTED	Double compressed	14.1.1	0	0	0	1	0	1
ZCB_SUPPORTED	Compressed subword	14.1.1	0	0	0	1	0	1
F_SUPPORTED	Single-precision FP	16.6.1	0	0	0	1	0	1

(Continued)

Table A.3 Key Wally configuration parameters. Cont'd

Parameter	Meaning	Section	rv32e (embedded)	rv32i (simple)	rv32imc (μ controller)	rv32gc (apps proc)	rv64i (basic)	rv64gc (apps proc)
D_SUPPORTED	Double-precision FP	16.6.1	0	0	0	1	0	1
Q_SUPPORTED	Quad-precision FP	16.6.1	0	0	0	0	0	0
ZFH_SUPPORTED	Half-precision FP	16.6.1	0	0	0	1	0	1
ZFA_SUPPORTED	Additional FP instructions	16.4.5	0	0	0	1	0	1
S_SUPPORTED	Supervisor mode	8.2.1	0	0	0	1	0	1
U_SUPPORTED	User mode	8.2.1	0	0	1	1	0	1
SSTC_SUPPORTED	Supervisor timer compare	22.2.11.2	0	0	0	1	0	1
ZICNTR_SUPPORTED	Performance counters	8.4.3.1	0	0	1	1	0	1
ZIHPM_SUPPORTED	Hardware perf monitors	8.4.3.1	0	0	1	1	0	1
ZICBOM_SUPPORTED	Cache management ops	19.1.1.1	0	0	0	1	0	1
ZICBOZ_SUPPORTED	Cache block zero	19.1.1.2	0	0	0	1	0	1
ZICBOP_SUPPORTED	Cache block prefetch	19.1.1.3	0	0	0	1	0	1
SVPBMT_SUPPORTED	Page-based memory types	11.2.2.1.1	0	0	0	0	0	1
SVNAPOT_SUPPORTED	Huge pages	11.2.1.3.1	0	0	0	0	0	1
SVINVAL_SUPPORTED	TLB invalidation	11.4.4	0	0	0	1	0	1
SVADU_SUPPORTED	Hardware A/D bit update	11.2.1.4	0	0	0	1	0	1
BUS_SUPPORTED	AHB Interface	9.5	1	0	0	1	0	1
DCACHE_SUPPORTED	Data cache	12.2	0	0	0	1	0	1
ICACHE_SUPPORTED	Instruction cache	13.3.1	0	0	0	1	0	1

(Continued)

Table A.3 Key Wally configuration parameters. Cont'd

Parameter	Meaning	Section	rv32e (embedded)	rv32i (simple)	rv32imc (μ controller)	rv32gc (apps proc)	rv64i (basic)	rv64gc (apps proc)
VIRTMEM_SUPPORTED	Virtual memory	11.4.1	0	0	0	1	0	1
BIGENDIAN_SUPPORTED	Big endian	8.4.7	0	0	0	1	0	1
PMP_ENTRIES	PMP: 0, 16, or 64	11.4.1.2	0	0	0	16	0	16
PMP_G	PMP granularity	11.4.1.2	0	0	0	4	0	4
BPRED_SUPPORTED	Branch predictor	13.3.3	0	0	0	1	0	1
DIVCOPIES	Copies of divsqr hardware: 1, 2, or 4	F.2.11.6	0	0	0	2	0	4
IDIV_ON_FPU	Use FPU for int division	16.6.4	0	0	0	0	0	1
IDIV_BITSPERCYCLE	Int divider replication	15.4.2	N/A	N/A	2	N/A	N/A	N/A
DTIM_SUPPORTED	DTIM (in the LSU)	7.2.1	0	1	1	0	1	0
IROM_SUPPORTED	IROM (in the IFU)	7.2.1	0	1	1	0	1	0
BOOTROM_SUPPORTED	Boot ROM (uncore)	9.5	1	0	0	1	0	1
UNCORE_RAM_SUPPORTED	On-chip RAM (uncore)	9.5	1	0	0	1	0	1
EXT_MEM_SUPPORTED	External memory	9.5	0	0	0	0	0	0
CLINT_SUPPORTED	CLINT	20.2	0	0	1	1	0	1
GPIO_SUPPORTED	GPIO	20.1	0	0	1	1	0	1
UART_SUPPORTED	UART	20.4	0	0	1	1	0	1
PLIC_SUPPORTED	PLIC	20.3	0	0	1	1	0	1
SPI_SUPPORTED	SPI	20.5	0	0	1	1	0	1
SDC_SUPPORTED	SDC	23.2.1	0	0	0	0	0	0

0= Not supported, 1 = supported.

Table A.4 Additional Wally configuration parameters.

Parameter	Meaning	Section	Controlling Config	Typical Value
RESET_VECTOR	PC on reset	8.4.6	N/A	0x80000000
AHBW	Width of AHB interface	9.5.2	N/A	XLEN
BURST_EN	AHBW burst transfer enable	9.2.2.5	N/A	1
USE_SRAM	Target SRAM library memories	6.2.3	N/A	0
IEEE754	Use default IEEE SoftFloat NaN propagation and float-to-int behavior rather than RISC-V behavior	16.6.11	F_SUPPORTED	0
RADIX	FP divsqrt radix: 2 or 4	F.2.11.6	F_SUPPORTED	4
COUNTERS	Number of performance counters	8.4.3.1	ZIHPM_SUPPORTED	32
VECTORED_INTERRUPTS_SUPPORTED	Interrupt address depends on interrupt source	8.4.6	ZICSR_SUPPORTED	1
WFI_TIMEOUT_BIT	WFI times out after $2^{\text{WFI_TIMEOUT_BIT}}$ cycles	8.4.5.6	ZICSR_SUPPORTED	16
DCACHE_NUMWAYS	Number of D\$ ways	10.4	DCACHE_SUPPORTED	4
DCACHE_WAYSIZENBYTES	Size of D\$ way (bytes)	10.4.2	DCACHE_SUPPORTED	4,096
DCACHE_LINELENINBITS	D\$ line length (bits)	10.4.2	DCACHE_SUPPORTED	512
ICACHE_NUMWAYS	Number of I\$ ways	10.4	ICACHE_SUPPORTED	4
ICACHE_WAYSIZENBYTES	Size of I\$ way (bytes)	10.4.2	ICACHE_SUPPORTED	4,096
ICACHE_LINELENINBITS	I\$ line length (bits)	10.4.2	ICACHE_SUPPORTED	512
CACHE_SRAMLEN	Length of SRAM (in bits)	10.4.2	{D/I}CACHE_SUPPORTED	128
BPRED_TYPE	Branch predictor type	13.3.3	BPRED_SUPPORTED	BP_GSHARE
BPRED_SIZE	Branch predictor has $2^{\text{BPRED_SIZE}}$ entries	13.3.3	BPRED_SUPPORTED	10
BTB_SIZE	Branch target buffer has $2^{\text{BTB_SIZE}}$ entries	13.3.3	BPRED_SUPPORTED	10
RAS_SIZE	RAS entries	13.3.3	BPRED_SUPPORTED	16
INSTR_CLASS_PRED	Instruction class prediction	13.3.3.3	BPRED_SUPPORTED	1

(Continued)

Table A.4 Additional Wally configuration parameters. Cont'd

Parameter	Meaning	Section	Controlling Config	Typical Value
DTLB_ENTRIES	Data TLB size	11.4.1.3	VIRTMEM_SUPPORTED	32
ITLB_ENTRIES	Instruction TLB size	11.4.1.3	VIRTMEM_SUPPORTED	32
DTIM_BASE	DTIM base address	9.5.3.1	DTIM_SUPPORTED	0x80000000
DTIM_RANGE	DTIM size	9.5.3.1	DTIM_SUPPORTED	0x7FFFFFFF
IROM_BASE	IROM base address	9.5.3.1	IROM_SUPPORTED	0x80000000
IROM_RANGE	IROM size	9.5.3.1	IROM_SUPPORTED	0x7FFFFFFF
BOOTROM_BASE	Boot ROM base address	9.5.3.1	BOOTROM_SUPPORTED	0x00001000
BOOTROM_RANGE	Boot ROM size	9.5.3.1	BOOTROM_SUPPORTED	0xFFF
BOOTROM_PRELOAD	Initialize boot ROM	9.5.3.1	BOOTROM_SUPPORTED	0
UNCORE_RAM_BASE	Uncore RAM base address	9.5.3.1	UNCORE_RAM_SUPPORTED	0x80000000
UNCORE_RAM_RANGE	Uncore SRAM size	9.5.3.1	UNCORE_RAM_SUPPORTED	0x7FFFFFFF
UNCORE_RAM_PRELOAD	Initialize uncore RAM	9.5.3.1	UNCORE_RAM_SUPPORTED	0
RAM_LATENCY	Extra cycles of RAM latency	9.4	N/A	0
EXT_MEM_BASE	Off-chip memory base address	9.5.3.1	EXT_MEM_SUPPORTED	0x80000000
EXT_MEM_RANGE	Off-chip memory size	9.5.3.1	EXT_MEM_SUPPORTED	0x7FFFFFFF
CLINT_BASE	CLINT base address	9.5.3.1	CLINT_SUPPORTED	0x02000000
CLINT_RANGE	CLINT size	9.5.3.1	CLINT_SUPPORTED	0xFFFF
GPIO_BASE	GPIO base address	9.5.3.1	GPIO_SUPPORTED	0x10060000
GPIO_RANGE	GPIO size	9.5.3.1	GPIO_SUPPORTED	0xFF
UART_BASE	UART base address	9.5.3.1	UART_SUPPORTED	0x10000000
UART_RANGE	UART size	9.5.3.1	UART_SUPPORTED	7
PLIC_BASE	PLIC base address	9.5.3.1	PLIC_SUPPORTED	0x0C000000
PLIC_RANGE	PLIC size	9.5.3.1	PLIC_SUPPORTED	0x03FFFFFF
SPI_BASE	SPI base address	9.5.3.1	SPI_SUPPORTED	0x10040000
SPI_RANGE	SPI size	9.5.3.1	SPI_SUPPORTED	0xFFF
SDC_BASE	SDC base address	9.5.3.1	SDC_SUPPORTED	0x00013000

(Continued)

Table A.4 Additional Wally configuration parameters. Cont'd

Parameter	Meaning	Section	Controlling Config	Typical Value
SDC_RANGE	SDC size	9.5.3.1	SDC_SUPPORTED	0xFFF
GPIO_LOOPBACK_TEST	Tie GPIO outputs to inputs	20.1.2	GPIO_SUPPORTED	1
UART_PRESCALE	Prescale system clock down by $2^{\text{UART_PRESCALE}}$ for UART	20.4	UART_SUPPORTED	1
PLIC_NUM_SRC	Number of interrupt sources	20.3.1	PLIC_SUPPORTED	10
PLIC_GPIO_ID	GPIO interrupt source ID	20.3.1	PLIC_SUPPORTED	3
PLIC_SPI_ID	SPI interrupt source ID	20.3.1	PLIC_SUPPORTED	6
PLIC_SDC_ID	SDC interrupt source ID	20.3.1	PLIC_SUPPORTED	9
PLIC_UART_ID	UART interrupt source ID	20.3.1	PLIC_SUPPORTED	10

Table A.5 Wally derivative configurations.

Derivative	Base Configuration and Purpose	Main Changes from Baseline
buildroot	rv64gc Running Linux boot in simulation	RESET_VECTOR = 0x1000 UNCORE_RAM_RANGE = 0xFFFFFFFF UNCORE_RAM_PRELOAD = 1 GPIO_LOOPBACK_TEST = 0 SPI_LOOPBACK_TEST = 0 UART_PRESCALE = 0 PLIC_NUM_SRC = 53
fpga	buildroot Running Linux boot on an FPGA	BOOTROM_PRELOAD = 1 UNCORE_RAM_BASE = 0x2000 UNCORE_RAM_RANGE = 0x1FFF BOOTROM_RANGE = 0xFFF EXT_MEM_SUPPORTED = 1 EXT_MEM_BASE = 0x80000000 EXT_MEM_RANGE = 0x7FFFFFFF SDC_SUPPORTED = 1 PLIC_SDC_ID = 20 BPRED_SIZE = 12

(Continued)

Table A.5 Wally derivative configurations. Cont'd

Derivative	Base Configuration and Purpose	Main Changes from Baseline
syn_rv32e syn_rv32i syn_rv32imc syn_rv32gc syn_rv64i syn_rv64gc	rv32e rv32i rv32imc rv32gc rv64i rv64gc Logic synthesis with reduced memory sizes for runtime	DTIM_RANGE = 0x1FF IROM_RANGE = 0x1FF BOOTROM_RANGE = 0x1FF UNCORE_RAM_RANGE = 0x1FF WAYSIZINBYTES = 512 NUMWAYS = 1 BPRED_SIZE = 5 BTB_SIZE = 5
syn_sram_rv32e syn_sram_rv32i syn_sram_rv32imc syn_sram_rv32gc syn_sram_rv64i syn_sram_rv64gc	rv32e rv32i rv32imc rv32gc rv64i rv64gc Logic synthesis with SRAM	DTIM_RANGE = 0x1FF IROM_RANGE = 0x1FF USE_SRAM = 1
syn_rv64gc_pmp0 syn_sram_rv64gc_pmp0	syn_rv64gc syn_sram_rv64gc	PMP_ENTRIES = 0
syn_rv64gc_noPriv syn_sram_rv64gc_noPriv	syn_rv64gc_pmp0 syn_sram_rv64gc_pmp0	ZICSR_SUPPORTED = 0
syn_rv64gc_noFPU syn_sram_rv64gc_noFPU	syn_rv64gc_noPriv syn_sram_rv64gc_noPriv	F_SUPPORTED = 0 D_SUPPORTED = 0 ZCF_SUPPORTED = 0 ZCD_SUPPORTED = 0
syn_rv64gc_noMulDiv syn_sram_rv64gc_noMulDiv	syn_rv64gc_noFPU syn_sram_rv64gc_noFPU	M_SUPPORTED = 0 ZMMUL_SUPPORTED = 0
syn_rv64gc_noAtomic syn_sram_rv64gc_noAtomic	syn_rv64gc_noMulDiv syn_sram_rv64gc_noMulDiv	ZAAMO_SUPPORTED = 0 ZALRSC_SUPPORTED = 0
div_{2/4}_{1/1i/2/2i/4/4i}_rv {32/64}gc	rv{32/64}gc Divsqr architectures: • Radix • Bits per cycle • Integer division on FPU	RADIX = 2/4 DIVCOPIES = 1/2/4 IDIV_ON_FPU = 0/1 (based on i)
ram_0_0_rv64gc ram_1_0_rv64gc ram_2_0_rv64gc ram_1_1_rv64gc ram_2_1_rv64gc	rv64gc Twiddle RAM latency (first number) and burst mode (second number) to verify bus is robust	RAM_LATENCY = 0/1/2 BURST_EN = 0/1

(Continued)

Table A.5 Wally derivative configurations. Cont'd

Derivative	Base Configuration and Purpose	Main Changes from Baseline
bpred_GSHARE_{6/8/10/12/ 14/16}_16_10_{0/1}_rv32gc bpred_TWOBIT_{6/8/10/12/ 14/16}_16_10_{0/1}_rv32gc bpred_GSHARE_10_{2/3/4/6/ 10}_10_{0/1}_rv32gc bpred_GSHARE_10_16_{6/8/ 12/14/16}_{0/1}_rv32gc	rv32gc Twiddle branch predictor parameters for performance studies: • Type • Size • RAS • BTB • Instruction class (IC) prediction	Instruction class prediction BPRED_TYPE = {BP_GSHARE/ BP_TWOBIT} BPRED_SIZE = 6/8/10/12/14/ 16 RAS_SIZE = 2/3/4/6/10/16 BTB_SIZE = 6/8/10/12/14/16 ICCLASSPRED = 0/1
noicache_rv{32/64}gc nodcache_rv{32/64}gc nocache_rv{32/64}gc	rv{32/64}gc Cacheless operations	ICACHE_SUPPORTED = 0/1 DCACHE_SUPPORTED = 0/1
way_{1/2/8}_4096_512_rv{32/ 64}gc way_4_2048_512_rv{32/64}gc way_4_4096_{256/1024}_rv {32/64}gc	rv{32/64}gc Twiddle cache parameters: • Number of ways • Way size • Line size	DCACHE_NUMWAYS = 1/2/4/8 DCACHE_WAYSIZEINBYTES = 2048/4096 DCACHE_LINELENINBITS = 256/512/1024 Same for ICACHE
tlb2_rv{32/64}gc tlb16_rv{32/64}gc	rv{32/64}gc Twiddle TLB size	ITLB_ENTRIES = 2/16/32 DTLB_ENTRIES = 2/16/32
misaligned_rv32gc nomisaligned_rv64gc	rv{32/64}gc	ZICCLSM_SUPPORTED = 1 ZICCLSM_SUPPORTED = 0
nobigendian_rv{32/64}gc	rv{32/64}gc	BIGENDIAN_SUPPORTED = 0
zaamo_rv{32/64}gc zalrsc_rv{32/64}gc	rv{32/64}gc Test only Zaamo or Zalrsc	ZAAMO_SUPPORTED = 1/0 ZALRSC_SUPPORTED = 0/1
f_rv{32/64}gc fh_rv{32/64}gc fdh_rv{32/64}gc fdq_rv{32/64}gc fdqh_rv{32/64}gc	rv{32/64}gc Test FP representations	ZFH_SUPPORTED = 0/1 Q_SUPPORTED = 0/1 D_SUPPORTED = 0/1
zba_rv{32/64}gc zbb_rv{32/64}gc zbc_rv{32/64}gc zbs_rv{32/64}gc zbk_b_rv{32/64}gc zbk_c_rv{32/64}gc	rv{32/64}gc Bit manipulation and cryptography (Only one parameter in the right column is 1 at a time; the others are 0.)	ZBA_SUPPORTED ZBB_SUPPORTED ZBC_SUPPORTED ZBS_SUPPORTED ZBK_B_SUPPORTED ZBK_C_SUPPORTED

(Continued)

Table A.5 Wally derivative configurations. Cont'd

Derivative	Base Configuration and Purpose	Main Changes from Baseline
zbx_rv{32/64}gc zkd_rv{32/64}gc zke_rv{32/64}gc zkh_rv{32/64}gc	Bit manipulation and cryptography (continued from previous page)	ZBKX_SUPPORTED ZKD_SUPPORTED ZKE_SUPPORTED ZKH_SUPPORTED
noS_rv32gc	rv32gc	S_SUPPORTED = 0 SSTC_SUPPORTED = 0 VIRTMEM_SUPPORTED = 0 SVINVAL_SUPPORTED = 0 SVADU_SUPPORTED = 0
noS_rv64gc	rv64gc	S_SUPPORTED = 0 SSTC_SUPPORTED = 0 VIRTMEM_SUPPORTED = 0 SVINVAL_SUPPORTED = 0 SVADU_SUPPORTED = 0 SVPBMT_SUPPORTED = 0 SVNAPOT_SUPPORTED = 0
noU_rv{32/64}gc	noS_rv{32/64}gc	U_SUPPORTED = 0

Table A.6 RISC-V extensions and profiles.

Extension	Description	Date Ratified	Section	RV120U32	RV120U64	RV120U64	RV120S64	RV122U64	RV122S64	RV123U64	RV123S64	RV123U64	RV123S64
RV32I	Base ISA, 32-bit	12/2019	7	R									
RV64I	Base ISA, 64-bit	12/2019	7		R	R	R	R	R	R	R	R	R
A	Atomic = Zaamo + Zalrsc	12/2019	17		O	O	R	R	R	R	R	R	R
B	Bit manipulation = Zba + Zbb + Zbs	4/25/24	18.1					R	R	R	R	R	R
C	Compressed = Zca + some others	12/2019	14		O	O	R	R	R	R	R	R	R
D	Double-precision floating-point	12/2019	16		O	O	R	R	R	R	R	R	R
E	Base ISA RV{32/64}E (16 registers)	1/2023	2.5										

(Continued)

Table A.6 RISC-V extensions and profiles. Cont'd

Extension	Description	Date Ratified	Section	RV120U32	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64
F	Single-precision floating-point	12/2019	16	O	O	R	R	R	R	R	R	R	R
M	Integer multiply and divide	12/2019	15	O	O	R	R	R	R	R	R	R	R
V	Vector	11/2021	Not supported										
U	User mode	12/2021	8.2			R	R	R	R	R	R	R	R
S	Supervisor mode	12/2021	8.2			R	R	R	R	R	R	R	R
H	Hypervisor	11/2021	Not supported										
Zifencei	Instruction fetch fence (fence.i)	12/2019	10.4.2.5	O	O	R	R	R	R	R	R	R	R
Zicsr	Control and status registers (CSRs)	12/2021	8.2	O	O	R	R	R	R	R	R	R	R
Zicntr	Basic counters	3/2023	8.2.3.4	O	O	R	R	R	R	R	R	R	R
Zihpm	Hardware performance counters	3/2023	8.2.3.4	O	O	O	O	R	R	R	R	R	R
Zicclsm	Misaligned loads/stores	3/2023	19.2	O	O	R	R	R	R	R	R	R	R
Ziccif	Atomic instruction fetch	3/2023	17.2.4.1			R	R	R	R	R	R	R	R
Ziccrse	RsrvEventual	3/2023	17.4.4.2			R	R	R	R	R	R	R	R
Ziccamoa	AMO arithmetic	3/2023	17.4.4.3			R	R	R	R	R	R	R	R
Za128rs	Max 128B reservation set	3/2023	17.2.4.4			R	R	R	R	R	R	R	R
Za64rs	Max 64B reservation set	3/2023	17.2.4.4					R	R	R	R	R	R
Zihintpause	Pause instruction	2/2021	19.5					R	R	R	R	R	R
Zba	Address computation	11/2021	18.1.1					R	R	R	R	R	R
Zbb	Basic bit manipulation	11/2021	18.1.2					R	R	R	R	R	R
Zbs	Single-bit instructions	11/2021	18.1.4					R	R	R	R	R	R
Zca	Basic compressed instructions	4/2023	14			O	O	R	R	R	R	R	R
Zcb	Other compressed integer instructions	4/2023	14.1.1							R	R	R	R

(Continued)

Table A.6 RISC-V extensions and profiles. Cont'd

Extension	Description	Date Ratified	Section	RV120U32	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64
Zcf	Compressed float load/store	4/2023	14		O						
Zcd	Compressed double load/store	4/2023	14		O	O		R	R	R	R
Zic64b	64B cache blocks	3/2023	10.2					R	R	R	R
Zicbom	Cache block management	11/2021	19.1					R	R	R	R
Zicbop	Cache block prefetch	11/2021	19.1					R	R	R	R
Zicboz	Cache block zero	11/2021	19.1					R	R	R	R
Zfh	Half-precision floating-point	11/2021	16.6.1					O	O	O	O
Zfhmin	Half-precision fp transfer and cvt	11/2021	Subset of Zfh					R	R	R	O
Zbkb	Bit manipulation for crypto	11/2021	18.2.5					O	O		
Zbkc	Carry-less multiply (subset of Zbc)	11/2021	18.2.3					O	O		
Zbkx	Crossbar permutation	11/2021	18.2.6					O	O		
Zknd	NIST AES Decryption	11/2021	18.2.7					O	O		
Zkne	NIST AES Encryption	11/2021	18.2.7					O	O		
Zknh	NIST SHA2 Hash	11/2021	18.2.8					O	O		
Zkn	NIST crypto (Zkne + Zknd + Zknh + Zbk*)	11/2021	18.2					O	O	O	O
Zksed	ShangMi SM4 encrypt/decrypt	11/2021	Not supported					O	O		
Zksh	ShangMi SM3 hash	11/2021	Not supported					O	O		
Zks	ShangMi crypto (Zksed + Zksh + Zbk*)	11/2021	Not supported					O	O	O	O
Zkr	Entropy CSR	11/2021	Not supported					O		O	O
Zkt	Data-independent execution time	11/2021	18.2					R	R	R	R
Zk	Scalar crypto: Zkn + Zkr + Zkt	11/2021	Not supported								

(Continued)

Table A.6 RISC-V extensions and profiles. Cont'd

Extension	Description	Date Ratified	Section	RV120U32	RV120U64	RV120U64	RV120S64	RV122U64	RV122S64	RV123U64	RV123S64	RV123U64	RV123S64
Sm1p*, Ss1p*	Privileged version spec	11/2021	8.2			1.11		1.12		1.13			
Svbare	Disable virtual memory	3/2023	12.2.1			R		R		R		R	
Sv39	39-bit virtual addresses	11/2021	12.2.1			R		R		R		R	
Sv48	48-bit virtual addresses	11/2021	12.2.1			O		O		O		O	
Sv57	57-bit virtual addresses	11/2021	Not supported					O		O		O	
Svade	Exceptions on A and D bits	3/2023	11.2.1.4			R		R		R		R	
Svadu	Hardware update of A and D bits	11/2023	11.2.1.4							O		O	
Ssccptr	Hardware page table reads	3/2023	11 2.4			R		R		R		R	
Sstvecd	Unvectored interrupts	3/2023	8.4.1			R		R		R		R	
Sstvala	stval register	3/2023	8.2.5.2			R		R		R		R	
Ssu64xl	64-bit user mode	3/2023	8 4.3.3			O		O		R		R	
Sscounterenw	Writable scounteren	3/2023	8.2.3.4					R		R		R	
Svpbmt	Page-based memory types	11/2021	11.2.2.1.1					R		R		R	
Svinval	TLB invalidation	11/2021	19.3					R		R		R	
Svnapot	Translation of contiguous pages	11/2021	11.2.1.3.1					O		R		R	
Sstc	Supervisor timer interrupt	11/2021	8.2.5.2					O		R		R	
Sscofpmf	Counter overflow & filtering	11/2021	Not supported					O		R		R	
Ssstateen	Supervisor state-enable	11/2021	Not supported					O		R			
Sha	Augmented hypervisor	10/2024*	Not supported					O		O		O	
Shcounterenw	Writable hcounteren	10/2024*	Not supported					O		R			
Shvstvala	Writable vstval	10/2024*	Not supported					O		R			

(Continued)

Table A.6 RISC-V extensions and profiles. Cont'd

Extension	Description	Date Ratified	Section	RV120U32	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64	RV120U64
Shtvala	Writable htval	10/2024*	Not supported						O			R	
Shvstvecd	Unvectored hypervisor interrupts	10/2024*	Not supported						O			R	
Shvsatpa	Hypervisor satp modes	10/2024*	Not supported						O			R	
Shgatpa	Hypervisor virtual memory	10/2024*	Not supported						O			R	
Zicond	Conditional operations	11/2023	19.6									R R R R	
Zfa	Additional floating-point	9/2023	16.4.5									R R R R	
Zawrs	Wait on reservation set	11/2022	Not supported									R R R R	
Zvfhmin	Vector half-precision convert	6/2023	Not supported									R R O O	
Zvbb	Vector bit manipulation	9/2023	Not supported									R R O O	
Zvkt	Vector data-independent execution latency	9/2023	Not supported									R R O O	
Zihintntnl	Non-temporal locality hints	5/2023	Not supported									R R R R	
Zimop	Maybe operations	3/2024	Not supported									R R R R	
Zcmop	Compressed maybe operations	3/2024	Not supported									R R R R	
Zbc	Carry-less multiplication	11/2021	18.1.3									O O O O	
Zabha	Byte/halfword atomics	4/2024	Not supported									O O O O	
Zacas	Atomic compare and swap	11/2023	Not supported									O O O O	

(Continued)

Table A.6 RISC-V extensions and profiles. Cont'd

Extension	Description	Date Ratified	Section	RV120U32	RV120U64	RV120U64	RV120S64	RV122U64	RV122S64	RV123U64	RV123S64	RV123U64	RV123S64
Zvb*, Zvk*	Vector cryptography	9/2023	Not supported							O	O	O	O
Zvkng/Zvksg	Vector NIST/ShangMi crypto	9/2023	Not supported							O	O	O	O
Zvfh	Vector half-precision float	6/2023	Not supported							O	O	O	O
Zfbfmin	Scalar block float conversion	6/2024	Not supported							O	O	O	O
Zvfbfmin	Vector block float conversion	6/2024	Not supported							O	O	O	O
Zvfbfwma	Vector block float widening FMA	6/2024	Not supported							O	O	O	O
Ziccamoc	AMOCASQ support	10/2024*	Not supported							O	O	O	O
Zicfiss/Zicflp	Shadow-stack/landing pads	6/2024	Not supported							O	O	O	O
Zama16b	Misaligned accesses	10/2024*	Not supported							O	O	O	O
Sdtrig	Debug triggers	2/2025	Not supported							O		O	
Ssnpm	Pointer masking	10/2024	Not supported							R		O	
Supm/Sspm	Pointer masking	10/2024	Not supported							R	R/O	/O	O/O
Svptc	Visible page table transitions	6/2024	Not supported							O		O	
Sstrict	No non-conforming extensions	10/2024*	Not supported							O		O	
Sdext	Debug triggers	2/2025	Not supported										
Z*inx	FP in integer registers	11/2021	Not supported										

(Continued)

Table A.6 RISC-V extensions and profiles. Cont'd

Extension	Description	Date Ratified	Section	RV120U32 RV120U64 RVA20U64 RVA20S64	RV122U64 RVA22S64	RVA23U64 RVA23S64 RVB23U64 RVB23S64
Smstateen	State enable	11/2021	Not supported			
Smepmp	PMP enhancements	11/2021	Not supported			
Zmmul	Multiply but not divide	6/2022	15.2.3			
Ztso	Total store order (TSO)	1/2023	17.2.1			
Zcmp	Compressed push/pop	4/2023	Not supported			
Zcmt	Compressed table jumps	4/2023	Not supported			
Smaia, Ssaia	Advanced interrupt architecture	6/2023	Not supported			
Smcntrpmf	Counter privilege mode filtering	11/2023	Not supported			
Zaamo	AMO ops (A without lr-sc)	4/2024	17.2.4			
Zalrsc	lr-sc ops (A without AMO)	4/2024	17.2.4			
Zve*	Embedded vector computation	11/2021	Not supported			

*Ratified with the RVA23 profile

O = optional, R = required

Table A.7 Regression tests.

Configuration	Test Suites
rv32e	arch32e
rv32i	arch32i
rv32imc	arch32i, arch32c, arch32m, wally32periph
rv32gc	arch32i, arch32c, arch32m, arch32f, arch32f_fma, arch32f_divsqr, arch32d, arch32d_fma, arch32d_divsqr, arch32zfh, arch32zfh_fma, arch32zfh_divsqr, arch32priv, arch32a_amo, arch32zifencei, arch32zb*, arch32zk*, arch32zfa*, arch32zcb, arch32zicnd, arch32vm_sv32, arch32pmp, wally32a_lrsc, wally32priv, wally32periph
rv64i	arch64i
rv64gc	arch64i, arch64c, arch64m, arch64f, arch64d, arch64zfh, arch64f_fma, arch64d_fma, arch64zfh_fma, arch64f_divsqr, arch64d_divsqr, arch64zfh_divsqr, arch64zfa*, arch64zcb, arch64priv, arch64a_amo, arch64zb*, arch64zk*, arch64zifencei, arch64zicnd, arch64pmp, ahh64_ram_{0/1/2}_{0/1}_rv64gc, wally64a_lrsc, wally64priv, wally64periph, coverage64gc
buildroot	buildroot

Table A.8 Test suites.

Instructions	Chapter	RV32		RV64	
		riscv-arch-test	wally-riscv-arch-test	riscv-arch-test	wally-riscv-arch-test
I	7	arch32i		arch64i	
E	7	arch32e			
C	14	arch32c		arch64c	
M	15	arch32m		arch64m	
F	16	arch32f arch32f_fma arch32f_divsqr		arch64f arch64f_fma arch64f_divsqr	
D	16	arch32d arch32d_fma arch32d_divsqr		arch64d arch64d_fma arch64d_divsqr	
A	17	arch32a_amo	wally32a_lrsc	arch64a_amo	wally64a_lrsc
B	18	arch32zb{a/b/c/s}		arch64zb{a/b/c/s}	

(Continued)

Table A.8 Test suites. Cont'd

Instructions	Chapter	RV32		RV64	
		riscv-arch-test	wally-riscv-arch-test	riscv-arch-test	wally-riscv-arch-test
Zk	18	arch32zbk{b/c/x} arch32zk{d/e/h}		arch64zbk{b/c/x} arch64zk{d/e/h}	
Zfh	16	arch32zfh arch32zfh_fma arch32zfh_divsqr		arch64zfh arch64zfh_fma arch64zfh_divsqr	
Zfa	16	arch32zfaf arch32zfad		arch64zfaf arch64zfad	
Misalign	8	arch32priv arch32cpriv		arch64priv arch64cpriv	
Zcb	14	arch32zcb		arch64zcb	
Zicboz	19	arch32zicboz		arch64zicboz	
Zifencei	8	arch32zifencei		arch64zifencei	
Privileged	8		wally32priv		wally64priv
Peripherals	20		wally32periph		wally64periph
CoreMark	21		coremark		coremark
Embench	21		embench		embench

Table A.9 Additional tests.

TESTBENCH NAME	Section	Description
testbench-fp.sv	16.5.3	Applies TestFloat vectors to FPU

Table A.10 Key Wally commands.

Command	Directory	Notes
<code>git clone --recurse-submodules https://github.com/ <yourgithubaccount>/cvw.git</code>	Your home directory or subdirectory	Clone forked repository to your Linux directory
<code>make --jobs</code>	cvw	Build all tests: RISCOF, TestFloat, coverage
<code>make --jobs</code>	cvw/sim	Only rebuild RISCOF tests
<code>make --jobs wally-riscv-arch- test</code>	cvw/sim	Only rebuild wally-riscv-arch-test custom tests (this is much faster)
<code>lint-wally</code>	any	Run lint on SystemVerilog code
<code>regression-wally</code>	any	Run all tests cases; see <code>sim/verilator/logs/ <case>.log</code> for error messages if there are failures
<code>regression-wally --ccov</code>	any	Code coverage simulation; see <code>sim/questa/cov/ rv64gc_uncovered_hierarchical.rpt</code> for results
<code>regression-wally --fcov</code>	any	Functional coverage; see <code>addins/cvw-arch-verif/ work</code> for results
<code>regression-wally --testfloat</code>	any	Simulate TestFloat tests
<code>regression-wally --nightly</code>	any	Run a comprehensive set of simulations taking many hours
<code>wsim rv32i arch32i --gui</code>	any	Simulate a single configuration and test suite in the Questa GUI
<code>wsim rv32i arch32i</code>	any	Simulate a single configuration and test suite in batch mode with Questa (faster)
<code>wsim rv32i arch32i --sim verilator</code>	any	Simulate a single configuration and test suite in batch mode with Verilator
<code>wsim rv64gc <path>.elf --lockstep</code>	any	Run ImperasDV on a single ELF file in lock step with Wally
<code>wsim rv64gc <path>.elf --fcov --lockstep</code>	any	Run ImperasDV on a single ELF file in lock step with Wally while collecting functional coverage
<code>./coremark_sweep.py</code>	cvw/ benchmarks/ coremark	Run Coremark benchmark on many configurations
<code>./embench_arch_sweep.py</code>	cvw/ benchmarks/ embench	Run Embench benchmark on many configurations
<code>make synth DESIGN=wallypipelinedcore TECH=sky130 CONFIG=synth_rv32e FREQ=330</code>	cvw/synthDC	Synthesize DESIGN with TECH=sky130/sky90/ tsmc28psyn for a configuration at target frequency FREQ
<code>wsim buildroot buildroot --args +INSTR_LIMIT=600000000 --lockstep</code>	any	Run Linux boot simulation in lock step between Wally and ImperasDV

Table A.11 Wally main diagrams.

Figure in Appendix A	Figure in Chapters	Description	Module Name
Fig. A.2	Fig. 20.3	Wally top-level SoC diagram	wallypipelinedsoc
Fig. A.3	Fig. 7.13	Simplified Wally core	wallypipelinedcore
Fig. A.4	Fig. 13.12	IFU : Instruction fetch unit	ifu
Fig. A.5	Fig. 14.3	PC logic throughout IEU	in ifu, bpred, privileged
Fig. A.6	Fig. 13.15	Branch prediction unit	ifu/bpred
Fig. A.7	Fig. 16.26	Hazard unit	hazard
Fig. A.8	Fig. 12.13	LSU : Load/store unit	lsu
Fig. A.9	Fig. 10.11	Cache	cache
Fig. A.10	Fig. 11.15	MMU : Memory management unit	mmu
Fig. A.11	Fig. 11.19	TLB : Translation lookaside buffer	mmu/tlb
Fig. A.12	Fig. 11.16	PMA : Physical memory attributes	mmu/pmachecker
Fig. A.13	Fig. 11.18	PMP : Physical memory protection	mmu/pmpchecker
Fig. A.14	Fig. 17.5	AMO : Atomic memory operations	lsu/atomic
Fig. A.15	Fig. 9.25	EBU : External bus unit	ebu
Fig. A.16	Fig. 9.20	AHB and APB interfaces	wallypipelinedsoc
Fig. A.17	Fig. 8.13	Privileged unit	privileged
Fig. A.18	Fig. 8.14	Trap logic	privileged/trap
Fig. A.19	Fig. 9.28	Uncore : Memory and peripherals	uncore
Fig. A.20	Fig. 18.29	ALU with bit manip. and crypto	ieu/alu, ieu/bitmanipalu
Fig. A.21	Fig. 18.22	Zbb : Basic bit manipulation	ieu/bmu/zbb
Fig. A.22	Fig. 18.28	Zbc : Carry-less multiplication	ieu/bmu/zbb
Fig. A.23	Fig. 15.22	MDU : Integer multiply/divide unit	mdu
Fig. A.24	Fig. 15.19	Multiplier : Integer multiplier	mdu/mul
Fig. A.25	Fig. 15.20	Divider : Integer divider	mdu/div
Fig. A.26	Fig. 16.21	FPU : Floating-point (FP) unit	fpu
Fig. A.27	Fig. 16.6	Unpack : Unpack FP fields	fpu/unpack
Fig. A.28	Fig. 16.11	FMA : fused multiply-add unit	fpu/fma
Fig. A.29	Fig. F.35	FP/int divide and square root	fdivsqr
Fig. A.30	Fig. 16.14	FP/int convert unit	fcvt
Fig. A.31	Fig. 16.15	FP comparisons and min/max	fcmp
Fig. A.32	Fig. 16.16	FP postprocessing unit	postprocess

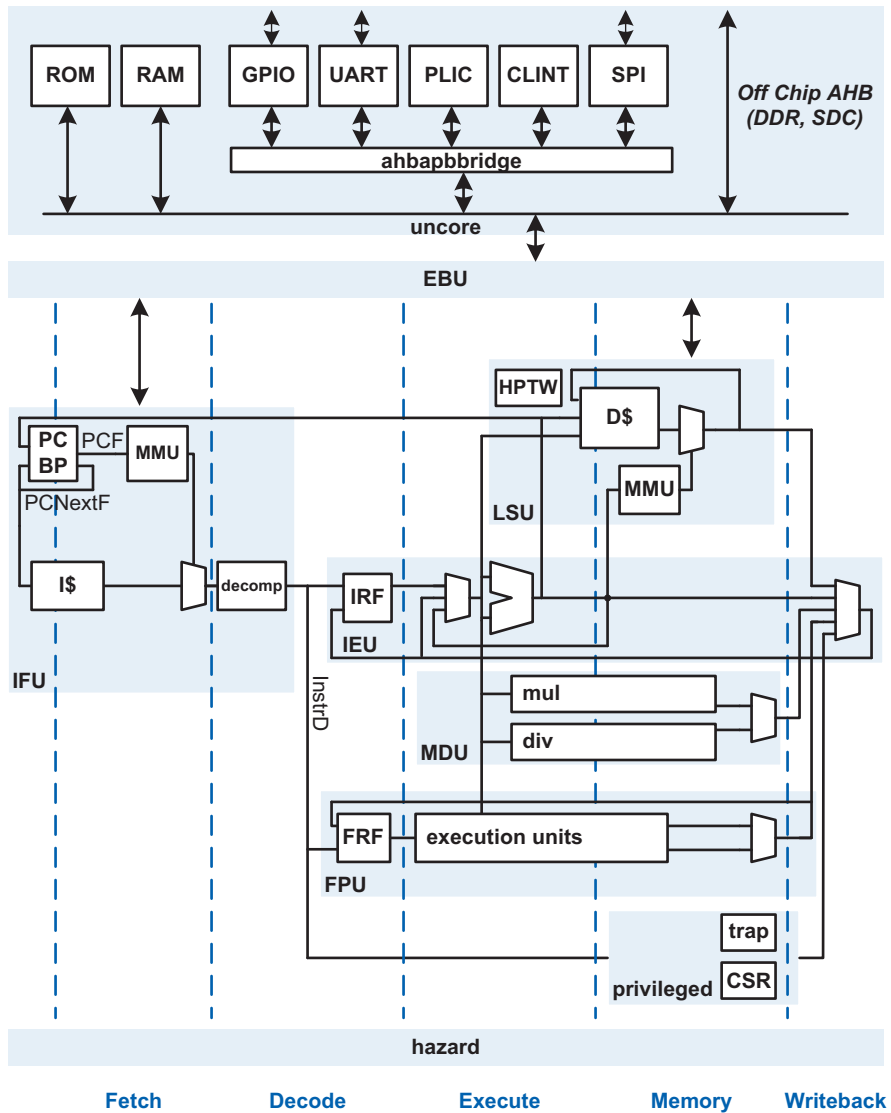


Fig. A.2 Wally top-level schematic.

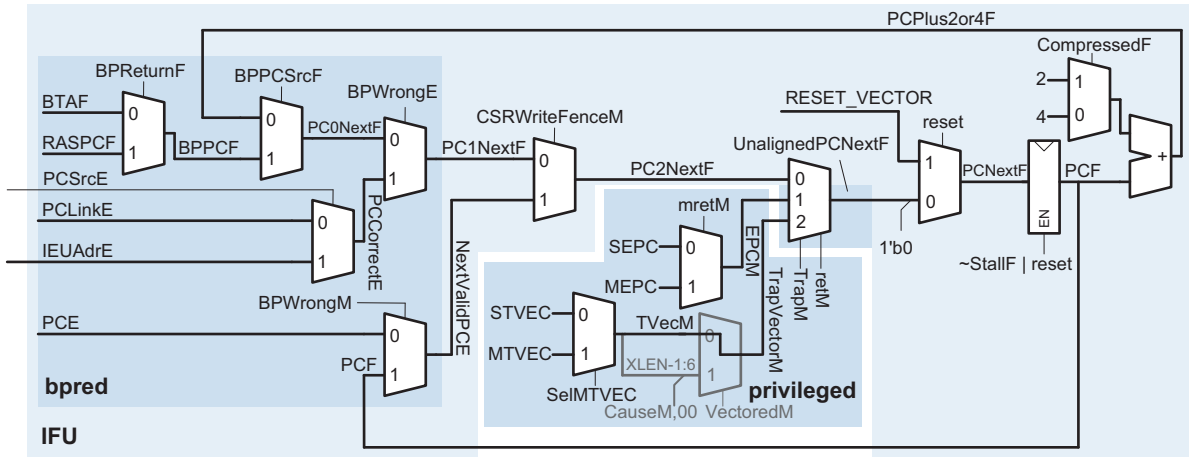


Fig. A.5 PC logic (spread out in IFU, branch prediction, and privileged units).

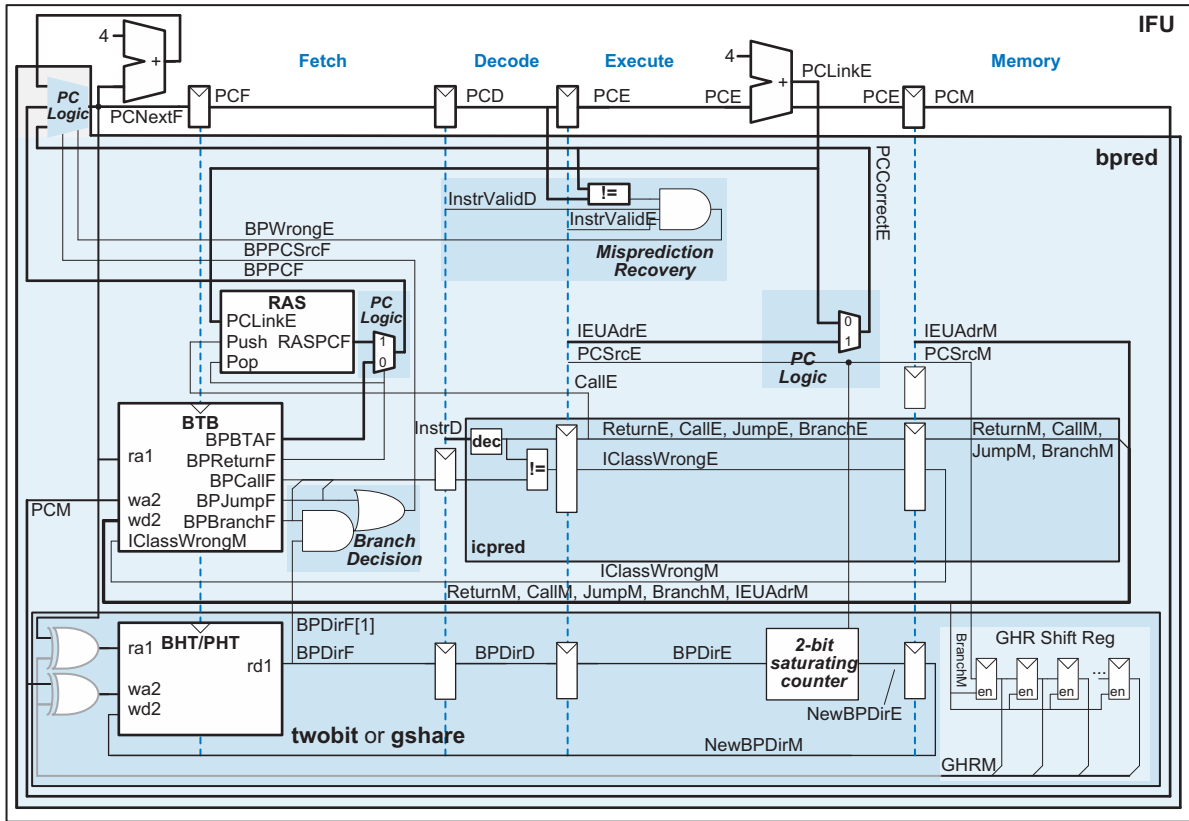


Fig. A.6 Branch prediction unit (bpred), shown as BP in Fig. A.2.

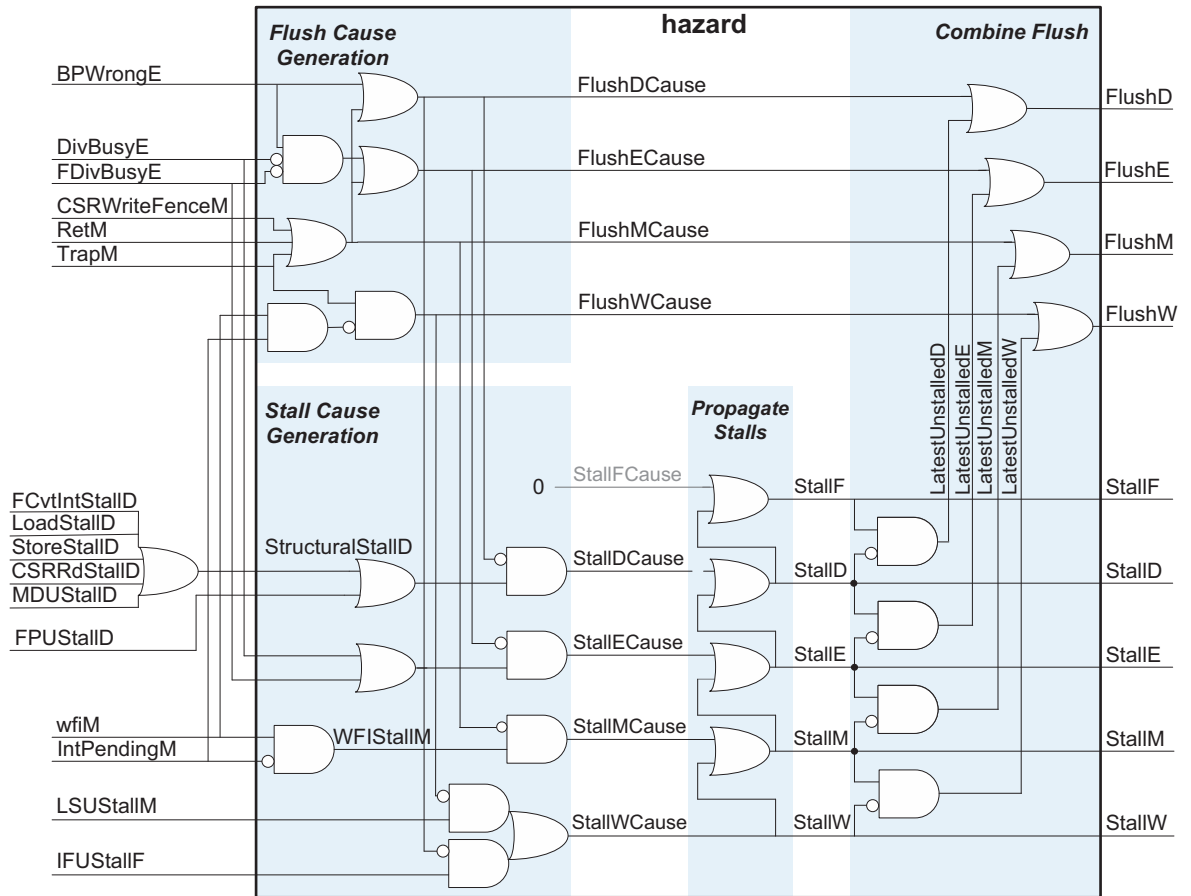
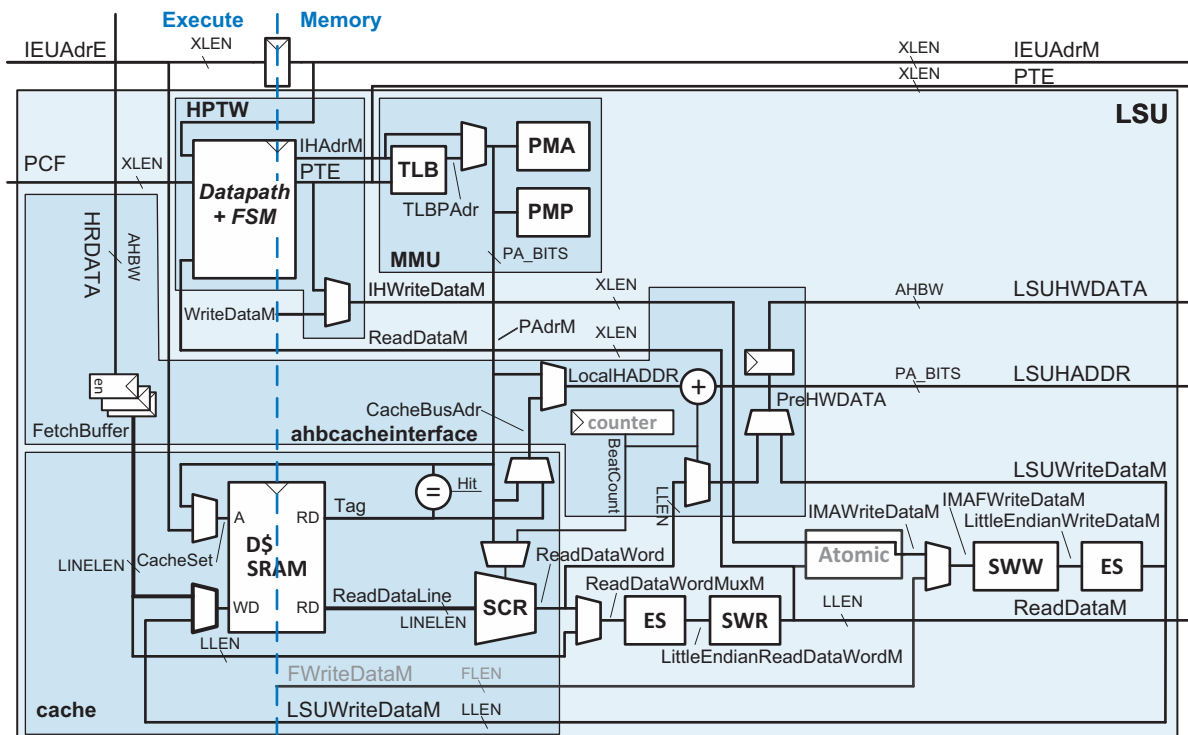


Fig. A.7 Hazard unit (hazard).



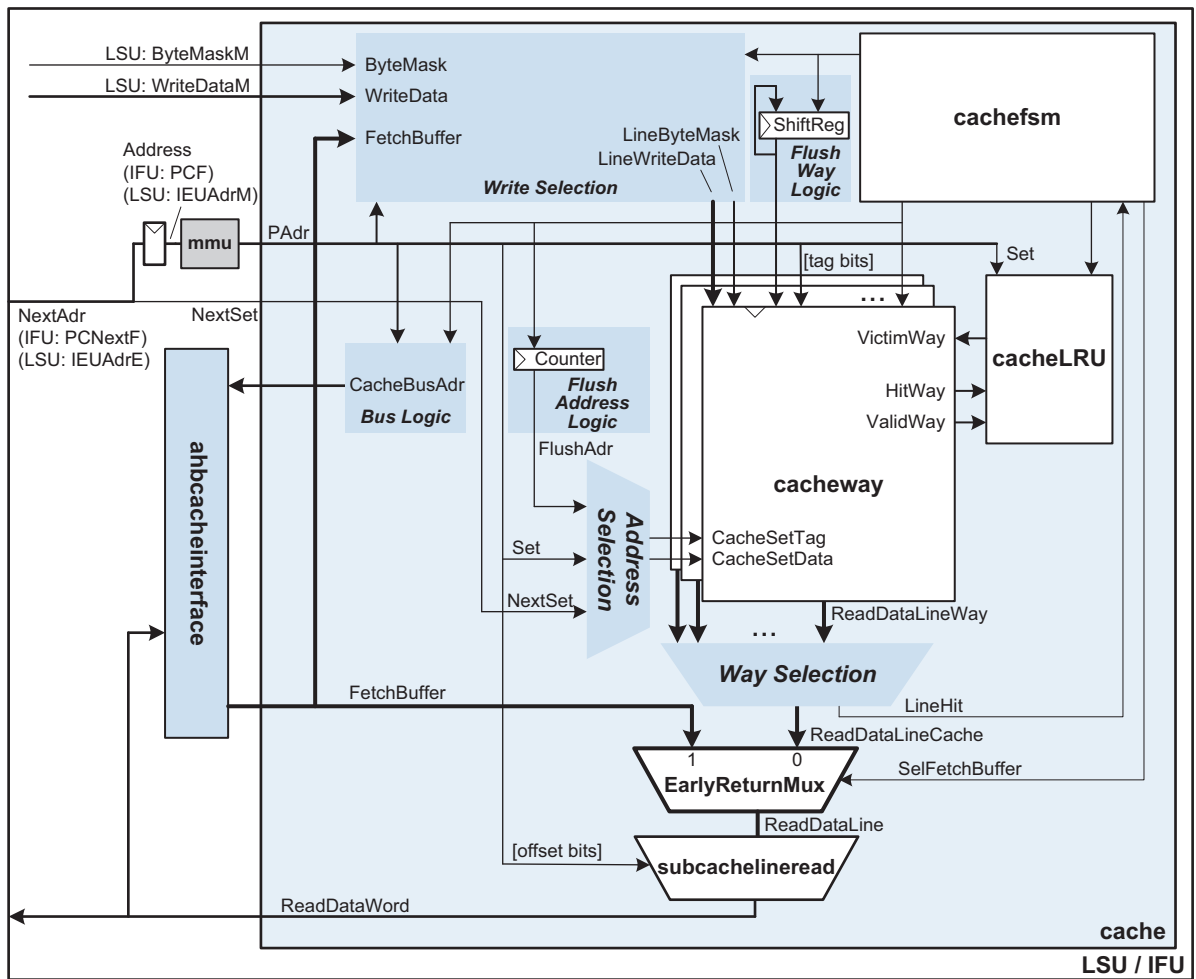


Fig. A.9 Cache (cache).

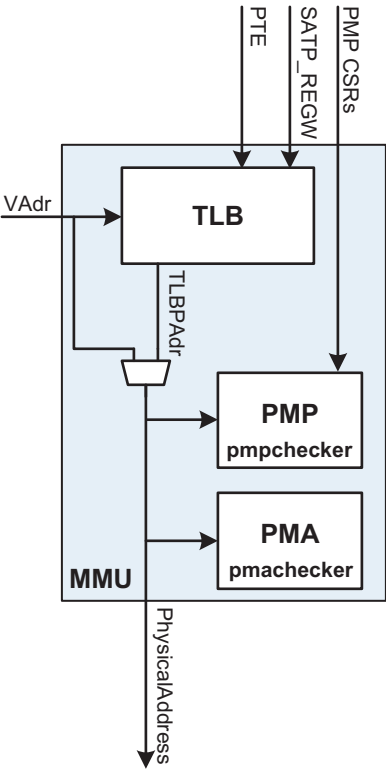


Fig. A.10 Memory management unit (mmu).





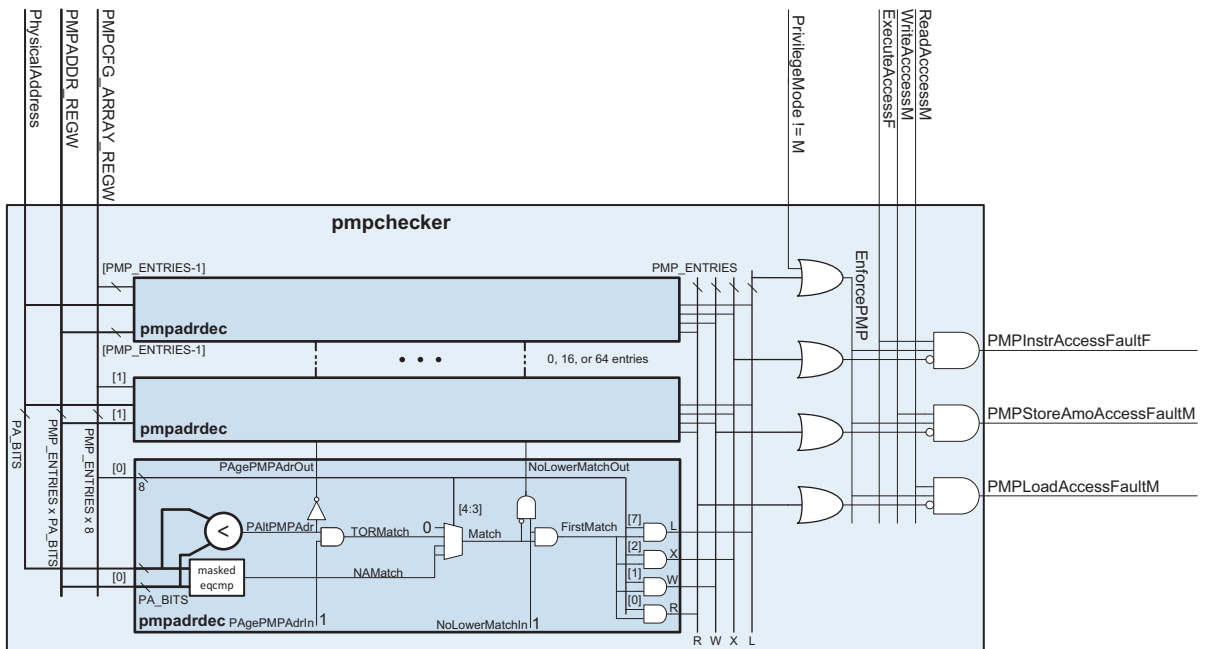


Fig. A.13 Physical memory protection checker (**pmpchecker**).

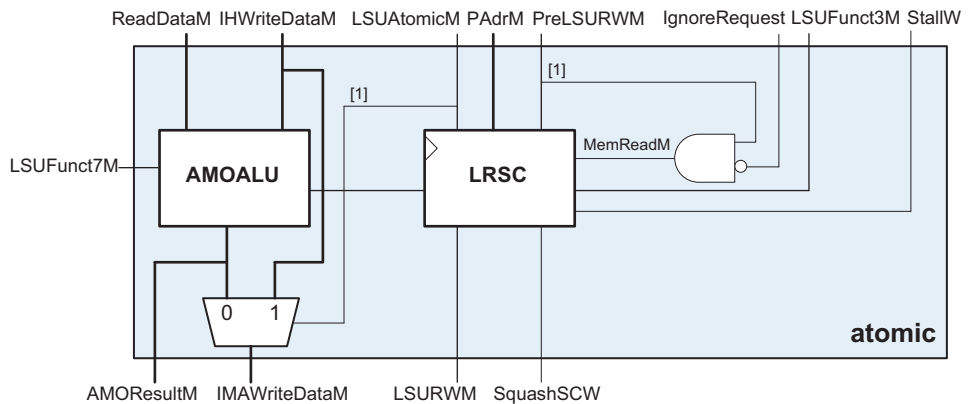


Fig. A.14 Atomic memory operations (AMO) unit (**atomic**).

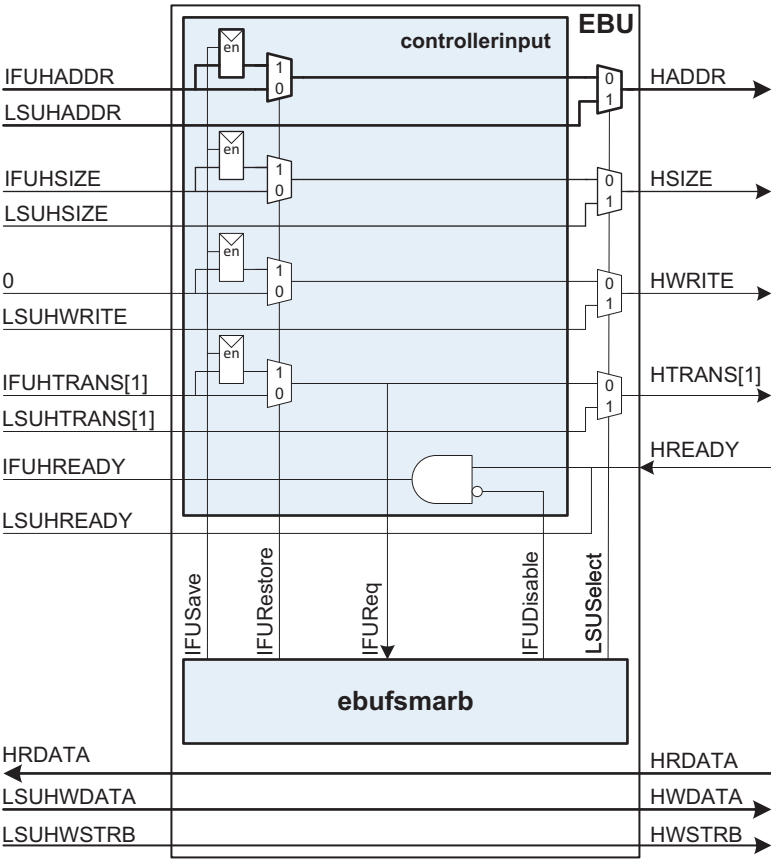


Fig. A.15 External bus unit (ebu).

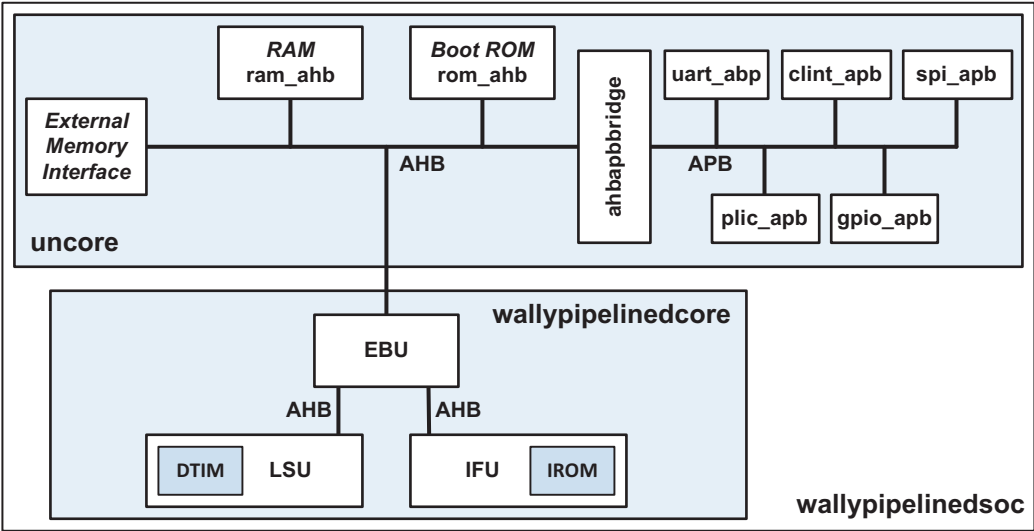


Fig. A.16 Wally AHB and APB interfaces.

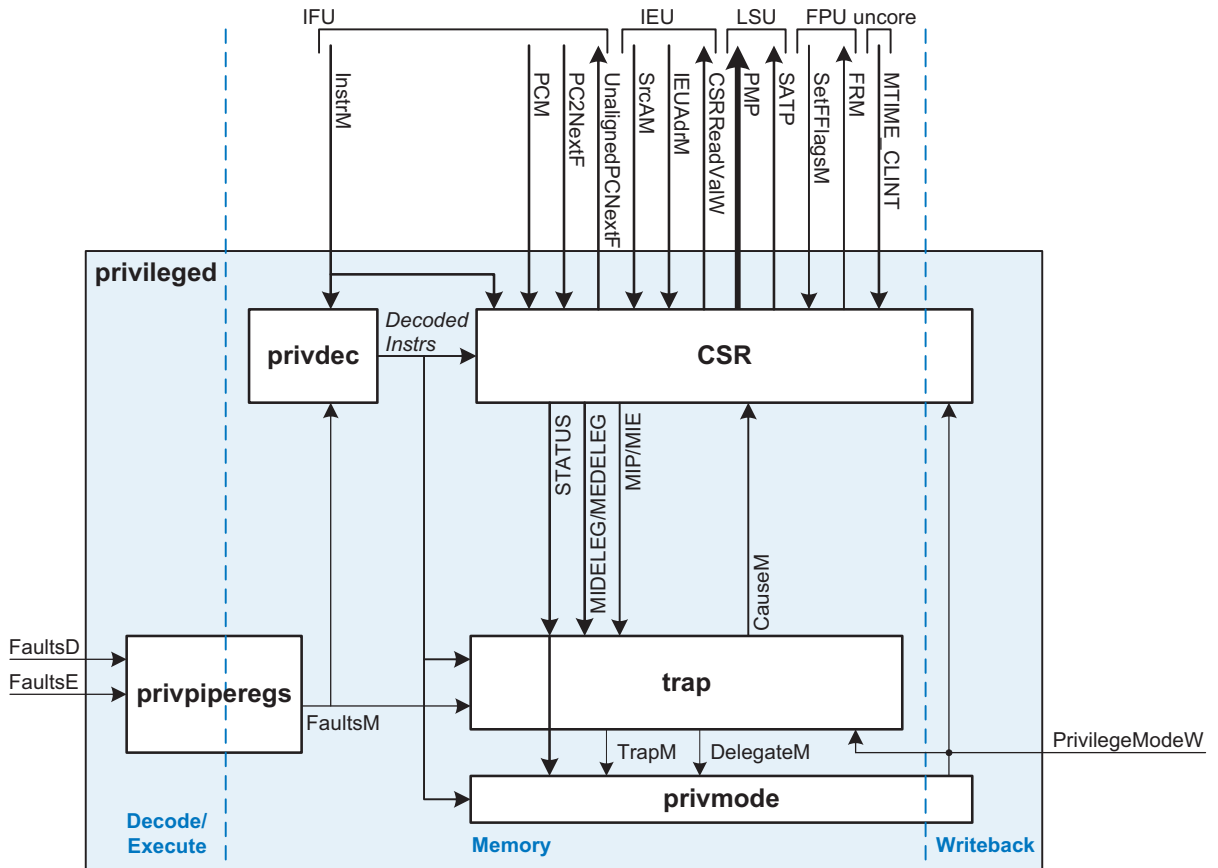


Fig. A.17 Privileged unit (privileged).

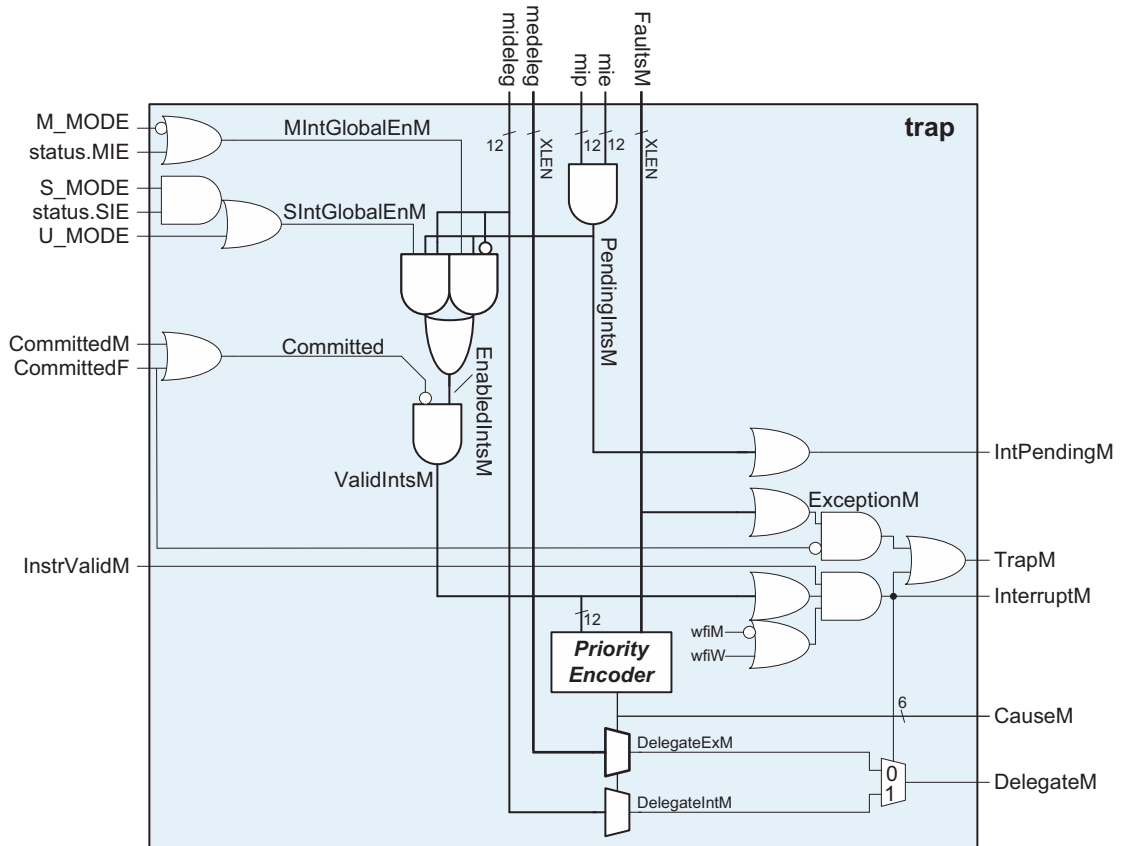


Fig. A.18 Trap unit (trap).

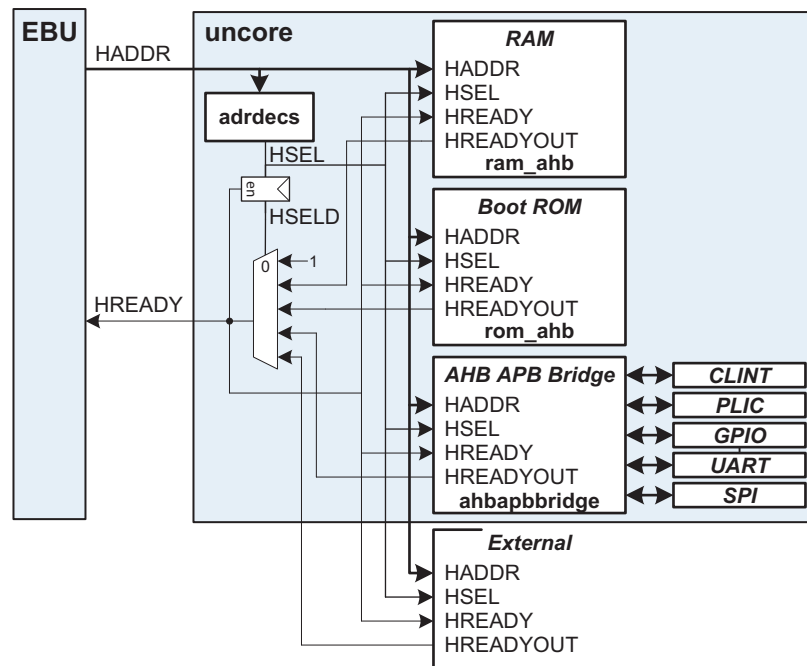


Fig. A.19 Uncore unit (uncore).

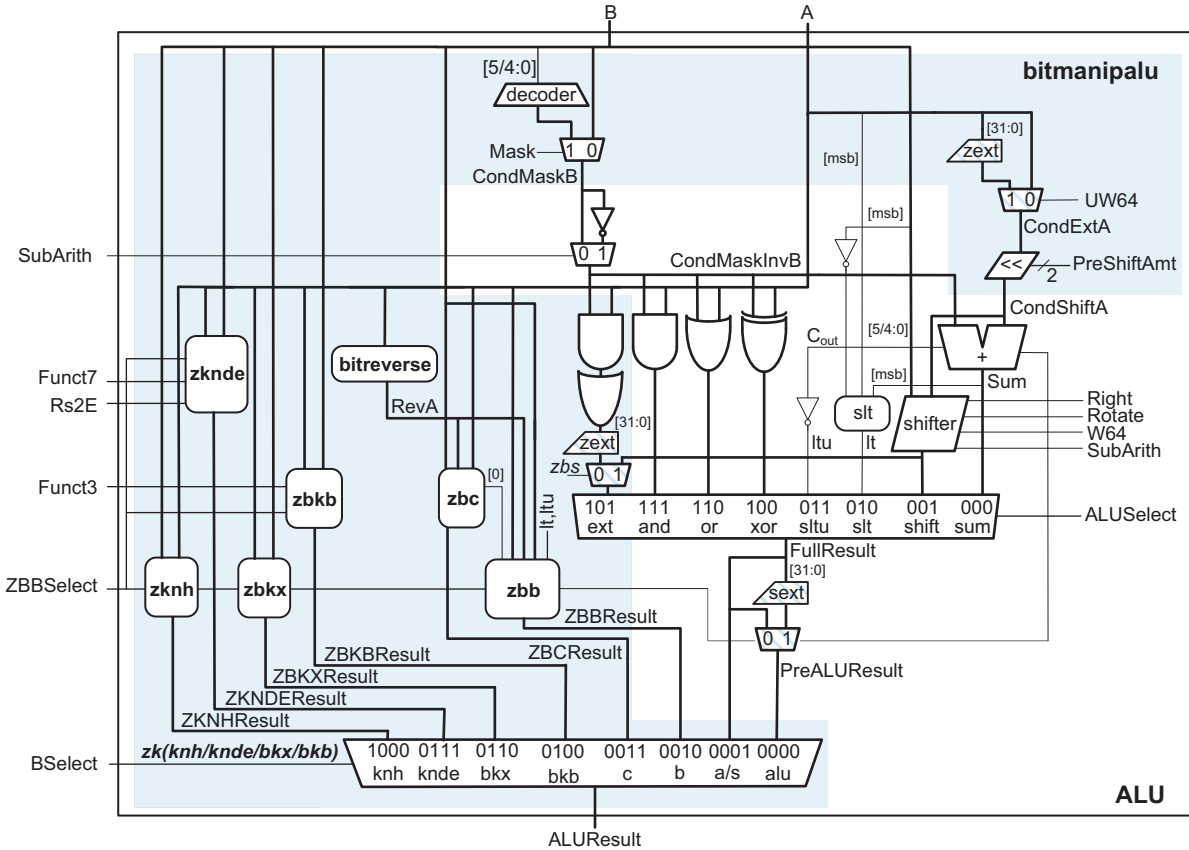


Fig. A.20 ALU with bit manipulation and cryptograph enhancements (alu, alubitmanip).

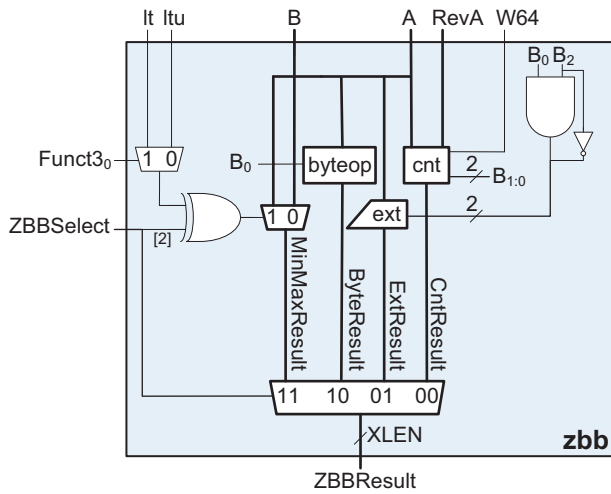


Fig. A.21 Zbb: Basic bit manipulation (zbb).

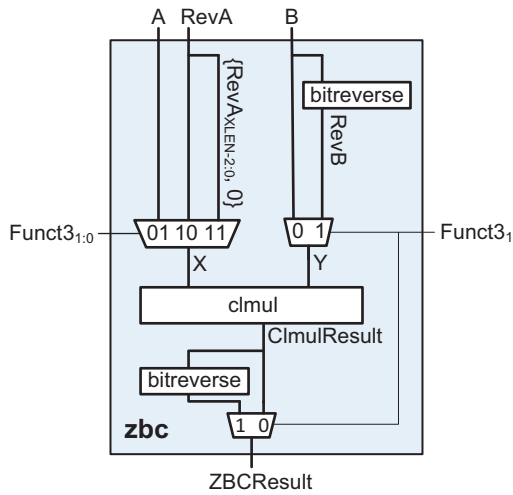


Fig. A.22 Zbc: Carry-less multiplication (zbc).

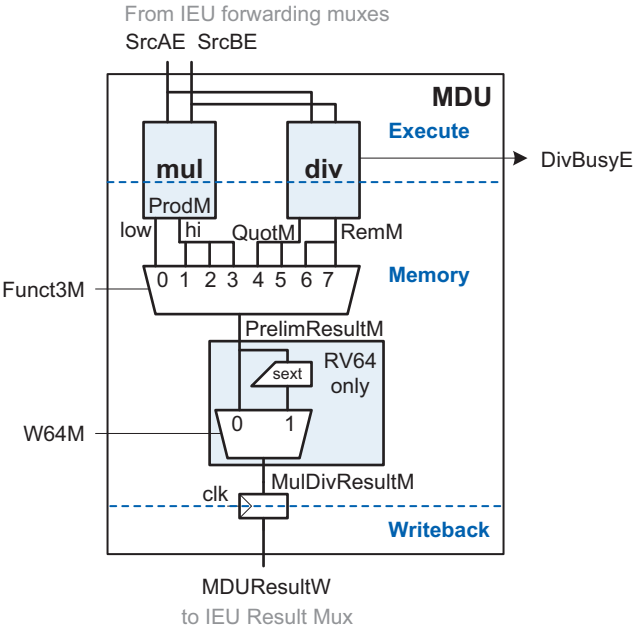


Fig. A.23 Integer multiply/divide unit (mdu).

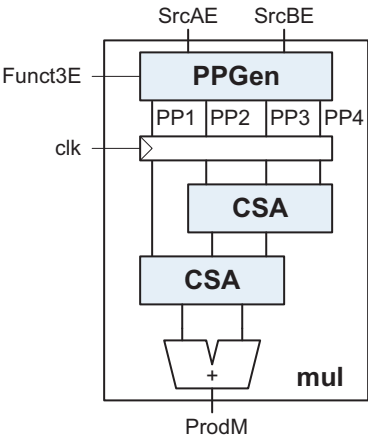


Fig. A.24 Integer multiplier (mul).



Fig. A.25 Integer divider (`div`).

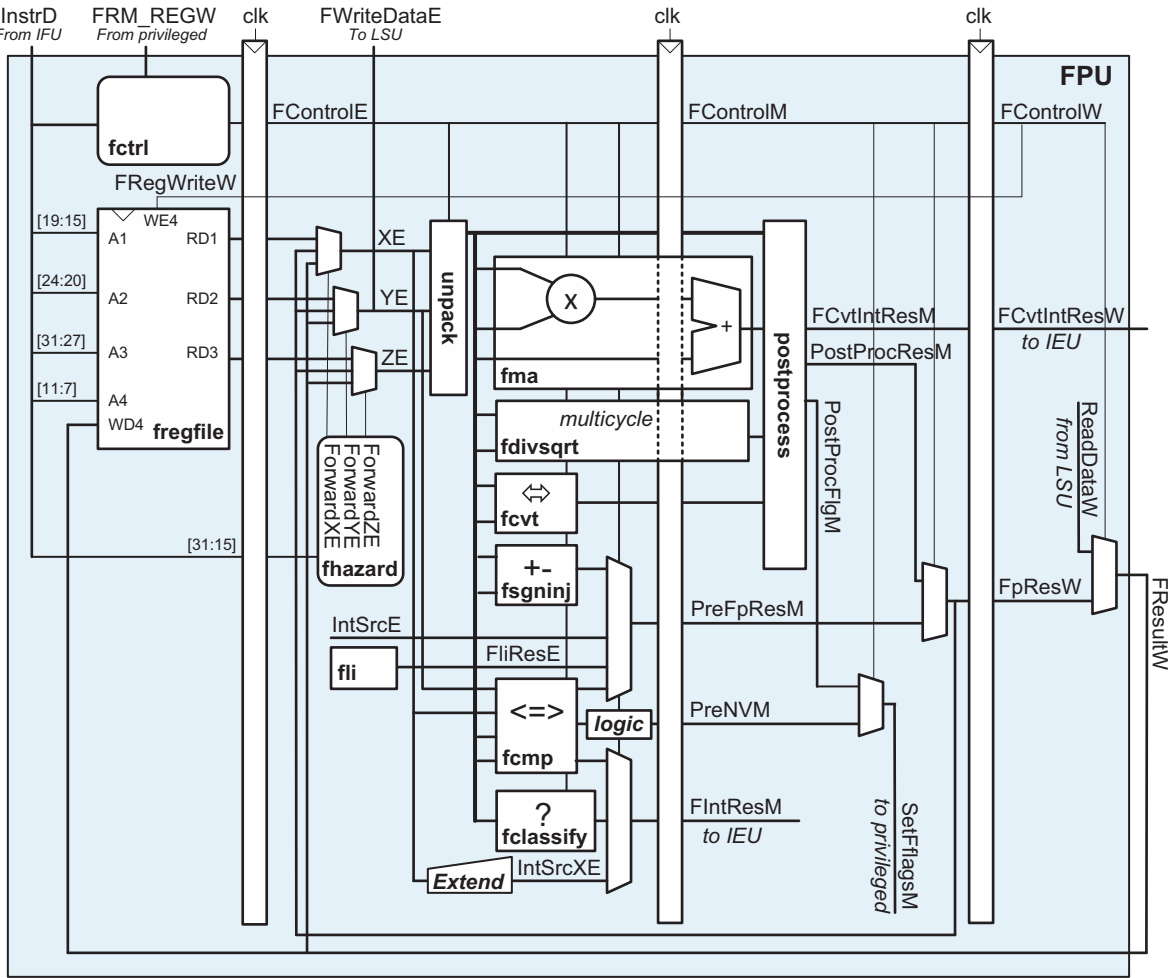


Fig. A.26 Floating-point unit (FPU).

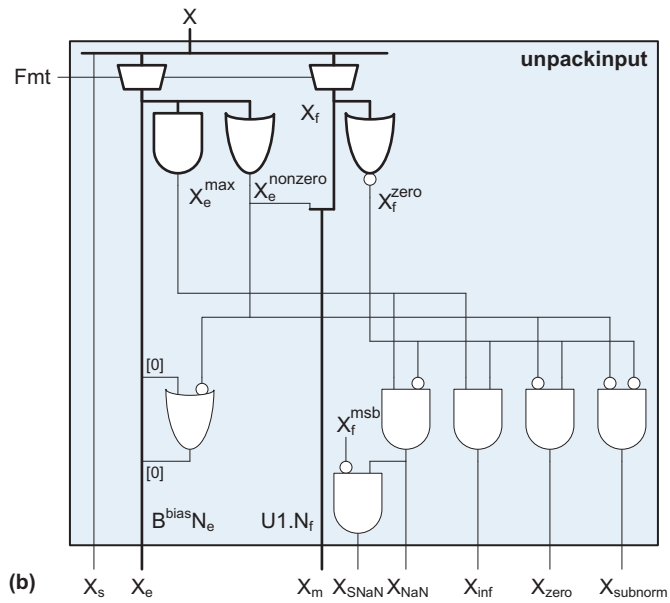
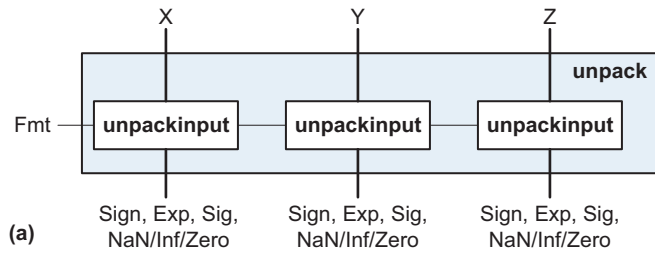


Fig. A.27 Unpack unit (unpack): (a) top level and (b) unpack hardware for a single channel.

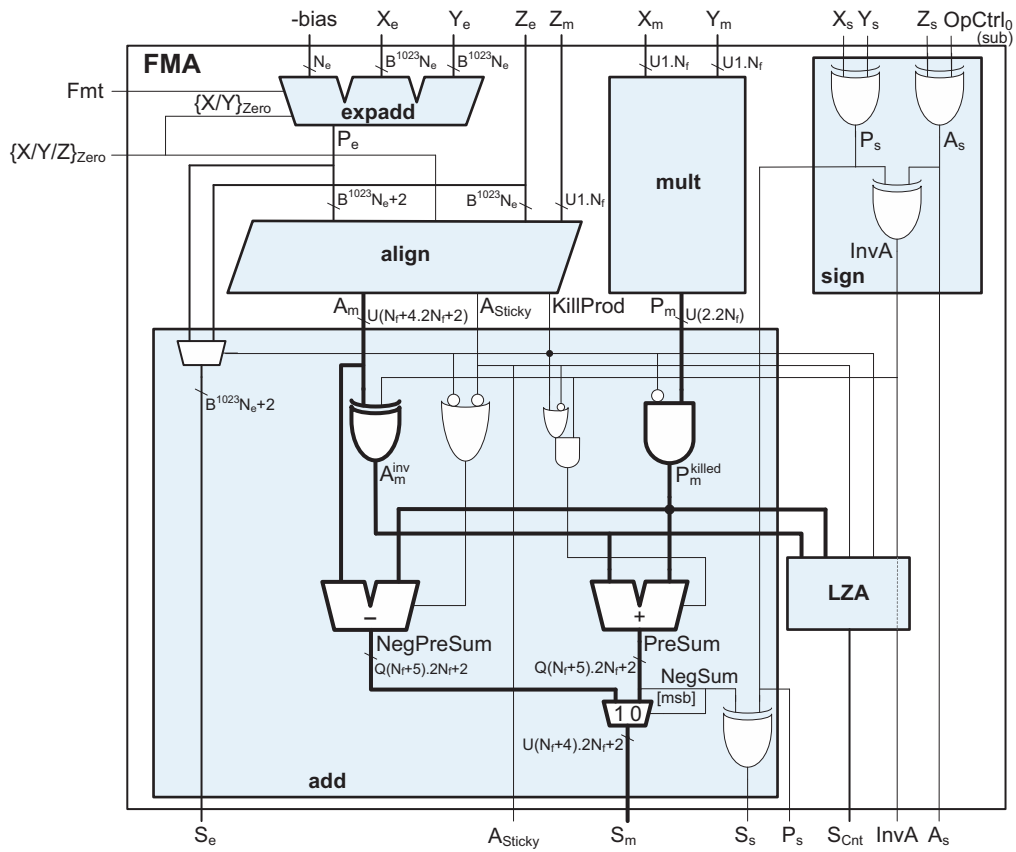


Fig. A.28 Floating-point fused multiply-add unit (FMA).

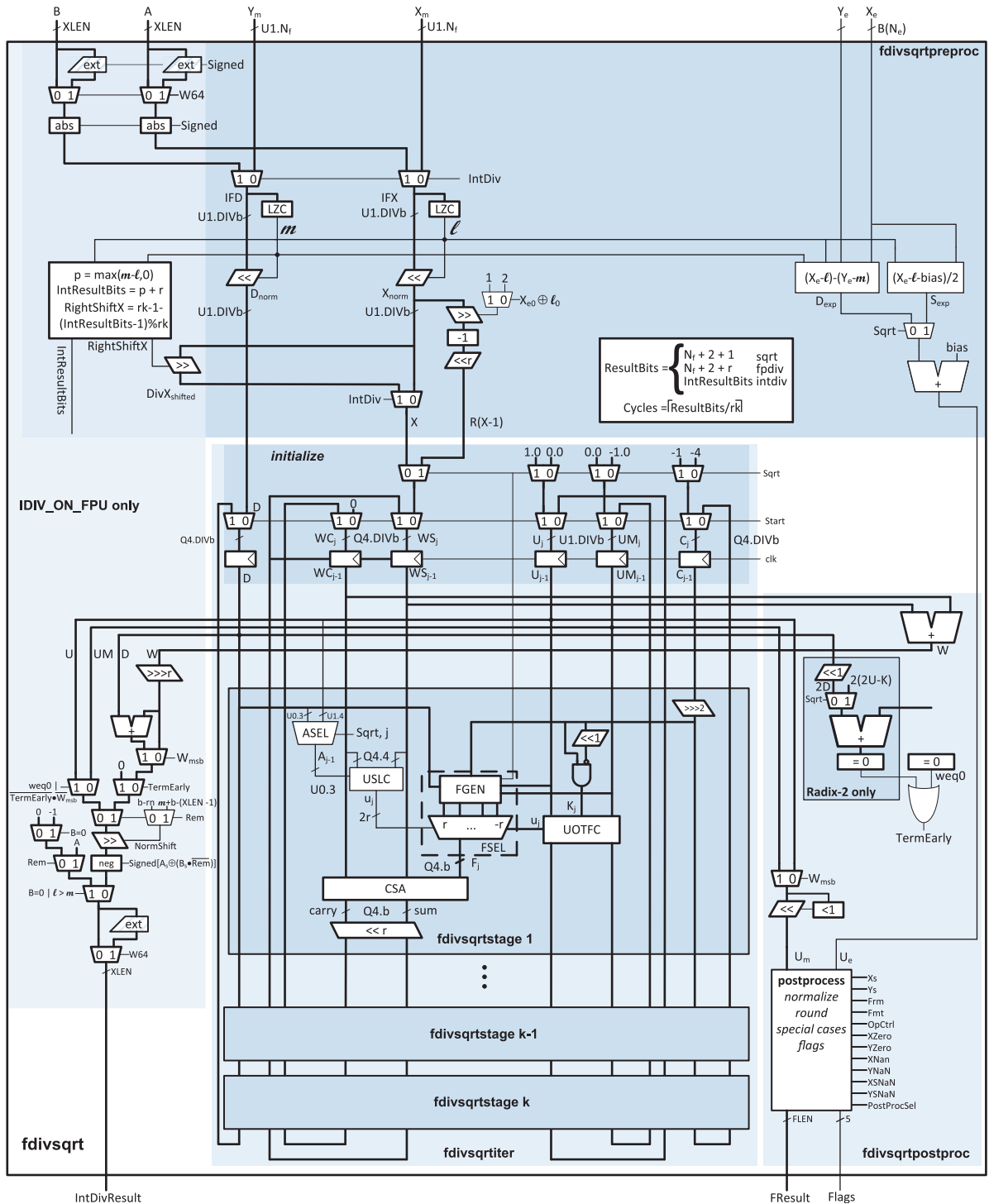


Fig. A.29 Floating-point divide and square root unit (fdivsqrt).

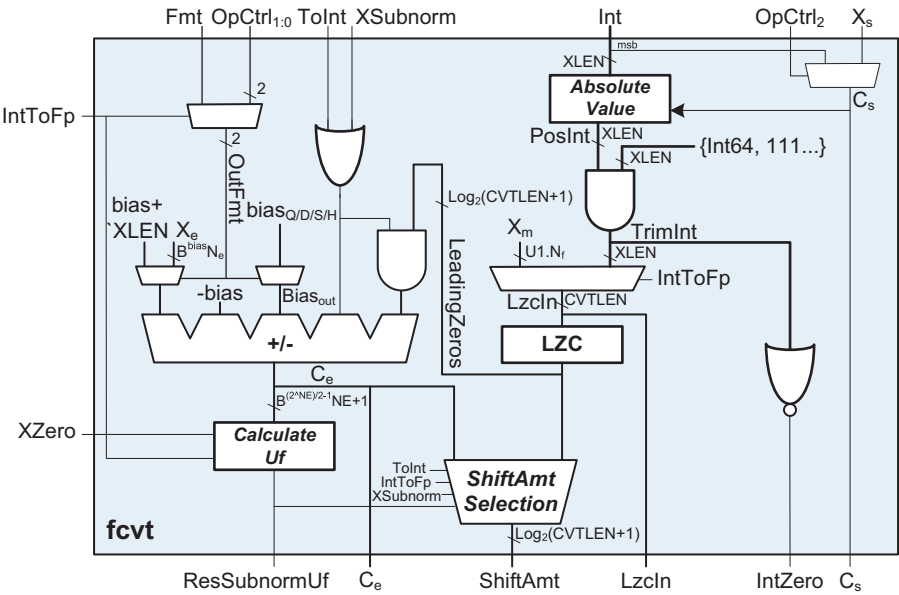


Fig. A.30 Floating-point convert unit (**fcvt**).

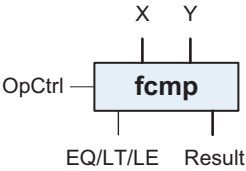
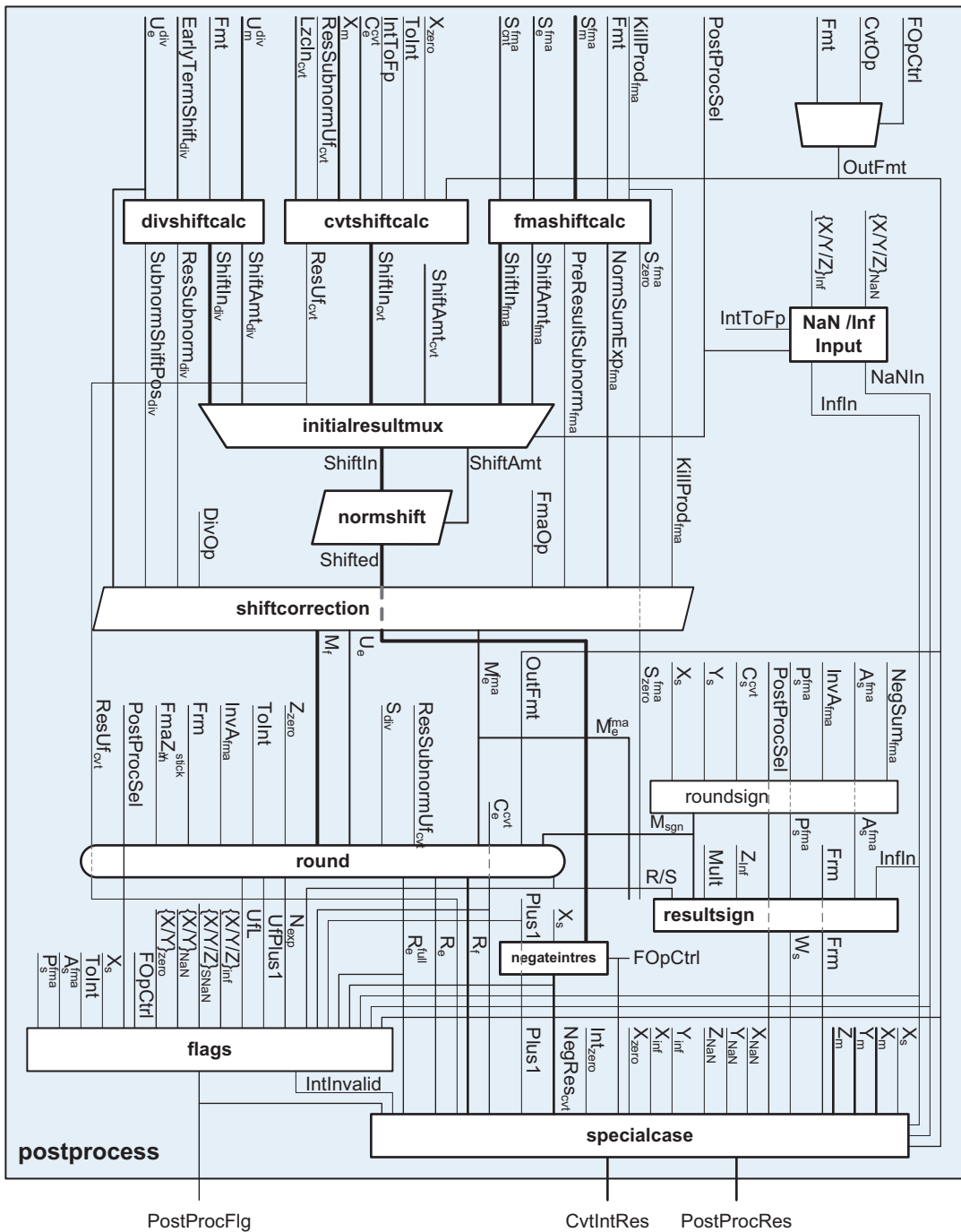


Fig. A.31 Floating-point compare unit (**fcmp**).



Hitchhiker's Guide to Linux

B

Linux is a powerful, free, and open development environment and is the only environment supported by commercial logic synthesis tools. This guide gives a brief introduction to Linux for engineers accustomed to Windows or Mac.

B.1 CONNECTING TO A LINUX SERVER

In a typical design environment, engineers work on a Linux *server*. The server has all the *computer-aided design* (CAD) tools installed so that users don't have to install them locally. It also has plenty of RAM (random access memory), hard drive space, and processors to support many users and to facilitate sharing files between users. Users access the server from a *client* computer, such as a desktop or laptop running Windows, MacOS, or Linux. Most users prefer accessing the server from a laptop computer because they can thus work from their office, dorm room, airport, or favorite Wi-Fi-enabled mountaintop. A system administrator gives users a username and initial password to access the server.

The Linux graphical user interface (GUI) windowing system is called X11, or simply X. To run X applications on the server and have them appear on a laptop, the user needs a program such as VNC (Virtual Network Computing) or X2Go that transmits pixels from the server to the client. For casual users, your best choice is the one supported by your company or university. RealVNC's VNC Connect is used widely but has a monthly subscription fee for commercial use. X2Go is not as fast but is free and fairly easy to use.

B.1.1 X2Go

To use X2Go, the system administrator needs to confirm that the `x2goserver` is running on the Linux server. The user then downloads the client (`x2goclient`) from x2go.org. Mac users must also download and install the X windows server XQuartz from xquartz.org. They may also have to go to System Preferences → Security and Privacy → General and under “Allow apps downloaded from” choose Open Anyway. Start `x2goclient` on your laptop. Choose Session → New Session. Give the session a name. Enter the Linux server host name in Host and your username in Login. Set Session type to XFCE, which gives a lightweight desktop environment. Then click on this session, enter your password, and choose OK.

- B.1 [Connecting to a Linux Server](#)
- B.2 [Working With Files](#)
- B.3 [More Handy Commands](#)
- B.4 [Linux Productivity](#)
- B.5 [Scripting and Programming](#)

Linus Torvalds (1969–) is the primary developer of the Linux kernel. He started the project for fun as a student, and it became his MSc thesis at the University of Helsinki. Linux grew into a free Unix clone and eventually surpassed and largely replaced Unix. Torvalds released version 1.0 in 1994, and he remains the guiding force for the kernel development.



(This image is openly licensed under CC BY-SA 3.0.)

Many new Linux distributions are adopting Wayland instead of X11, which requires a connection method other than X2Go, such as RDP (Remote Desktop Protocol).

Countless web pages with information about specific Linux commands exist, but none that we've yet found has a succinct introduction to the Linux features needed by hardware developers. [Schotts19] provides information beyond this chapter and is available as a free PDF or as a paid printed book. Many online Linux courses are also available from MOOC providers such as EdX, Coursera, and Udemy.

It is also possible to connect to a server with X-forwarding to your laptop using: `ssh -Y hostname`, but this is too sluggish for serious graphical use. `ssh` is sufficient and simple for terminal-based work.

The session on the server will open. You can suspend your session and reopen it later and come back to the same windows and programs you had been using. Long jobs will continue to run on the server even when your session is suspended.

B.1.2 Changing Your Password

After you've logged into a new system with a password provided by the system administrator, always change your password by opening a terminal and typing `passwd` at the command line. The system will then prompt you to verify your current password and then enter and retype your new password.

```
$ passwd
Changing password for lynn
(current) UNIX password:
New password:
Retype new password:
passwd: all authentication tokens updated successfully
```

In this appendix, the command prompt is represented as `$`. Your system may have a longer prompt. Only type the blue command after the prompt (in the example above `passwd`). The next lines show what will appear on the screen.

B.2 WORKING WITH FILES

Although Linux GUIs have a file manager, developers are most productive getting around the file system at the command line. [Table B.1](#) introduces commands for manipulating files in a terminal window. [Table B.1](#) assumes the user is `lynn` and the home directory is `/home/lynn`.

For best results, stick with filenames consisting of letters, numbers, and underscores. Avoid special characters, particularly spaces. Linux is case sensitive, so `readme` and `README` are two different files.

B.3 MORE HANDY COMMANDS

Now that you can get around, this section describes some of the most frequently used Linux commands.

Table B.1 Commands for manipulating files.

Command	Example
<code>mkdir</code> creates a new directory (i.e., folder)	\$ <code>mkdir tutorial</code>
<code>ls</code> lists files in the specified directory. If no directory is given (i.e., <code>ls</code>), it lists files in the current directory. <code>ls</code> current directory	\$ <code>ls tutorial</code>
<code>cat</code> (“concatenate”) to put text into a file. Press Ctrl-d to end the file.	\$ <code>cat > README</code> Hi! <Ctrl-d>
<code>less</code> displays the contents of a file. If it is a long file, it gives one screenful at a time. (<code>less</code> is an improved version of the older <code>more</code> command; remember “less is more.”)	\$ <code>less README</code> Hi! Press q to quit (i.e., stop displaying the file contents).
List the files again and now you’ll see both <code>README</code> and <code>tutorial</code> .	\$ <code>ls</code> <code>README tutorial</code>
<code>cp</code> copies a file. For example, copy <code>README</code> to <code>README2</code> .	\$ <code>cp README README2</code> \$ <code>ls</code> <code>README README2 tutorial</code>
<code>mv</code> moves or renames a file. For example, let’s rename <code>README2</code> to <code>README3</code> and move <code>README</code> into the <code>tutorial</code> directory.	\$ <code>mv README2 README3</code> \$ <code>mv README tutorial</code> \$ <code>ls</code> <code>README3 tutorial</code> \$ <code>ls tutorial</code> <code>README</code>
<code>cd</code> changes directories. <code>cd</code> by itself changes to your home directory.	\$ <code>cd tutorial</code> \$ <code>ls</code> <code>README</code>
<code>pwd</code> prints your current directory. The current directory is also called the working directory.	\$ <code>pwd</code> <code>/home/lynn/tutorial</code> \$ <code>cd</code> \$ <code>pwd</code> <code>/home/lynn</code>
<code>..</code> indicates the directory above your current directory. So <code>cd ..</code> moves up a level.	\$ <code>cd ..</code> \$ <code>pwd</code> <code>/home</code> \$ <code>ls</code> <code>ben alyssa lynn</code>
<code>~</code> indicates your home directory. <code>~lynn</code> is the home directory of user <code>lynn</code> .	\$ <code>cd ~/tutorial</code> \$ <code>pwd</code> <code>/home/lynn/tutorial</code>

(Continued)

Table B.1 Commands for manipulating files. Cont'd

Command	Example
You can also specify a full path name. Of course, replace /home/lynn with your home directory path that pwd reported.	<pre>\$ cp /home/lynn/README3 /home/lynn/tutorial/f \$ less /home/lynn/tutorial/f Hi!</pre>
ls -l gives a long listing with file permissions, owner, group, size, and modification date for each file. The permissions are 10 characters. The first character is a d for a directory or - for a regular file. The next characters indicate if a file is readable, writable, and executable for the user, group, and world (all users). The second column indicates the number of files in a directory. The third and fourth columns indicate the user and group that own the file. In most systems, the group has the same name as the user, and that user can add other users to their private group using: /usr/sbin/groupadd. The fifth column indicates the file size, and the sixth column lists the date and time the file was last modified.	<pre>\$ ls -l -rw-rw-r-- 1 lynn users 4 Nov 26 06:24 README3 drwxrwxr-x 2 lynn users 3 Nov 26 06:29 tutorial \$ ls -l tutorial -rw-rw-r-- 1 lynn users 4 Nov 26 06:17 README -rw-rw-r-- 1 lynn users 4 Nov 26 06:29 f</pre> For example, README3 can be read and written (but not executed) by lynn (-rw-rw-r--) and any member of the “users” group (-rw-rw-r-) but only read by others who are not in the group (-rw-rw-r--). README3 is 4 bytes long (-rw-rw-r-- lynn users 4) README3’s content is: Hi!, which is 3 characters (bytes) and a 1-byte NULL character to terminate the string. It was last modified on November 26th of this year. A user must have execute permission (x) on a directory to cd into it and execute permission on a program to run it.
chmod changes the mode, aka permissions. It takes 3 octal digits to indicate the permission for user, group, and world. Each digit has three bits indicating read (4), write (2), and execute (1), listed as rwx.	<pre>\$ chmod 640 README3 \$ ls -l README3 -rw-r----- 1 lynn users 4 Nov 26 06:24 README3</pre> The command shown makes README3 read and writable by lynn (6), readable by users (4), and inaccessible by others (0).
rm removes a file.	<pre>\$ rm README3 \$ ls tutorial</pre>
rmdir removes a directory, but the directory must be empty.	<pre>\$ rmdir tutorial rmdir: failed to remove ‘tutorial’: Directory not empty</pre>
Adding the -r option to rm makes it run recursively on all the contents of a directory. Adding -f forces rm to remove files without prompting. Join these together to remove a directory and all its contents. Now the working directory is empty. Use rm -rf cautiously; Linux does not have an undelete command (unless you are using version control as described in Appendix C).	<pre>\$ rm -rf tutorial \$ ls (empty)</pre>

B.3.1 Cast of Characters

The characters `*`, period (`.`), and `&` have special meanings, as listed in Table B.2.

Table B.2 Special characters: `*`, period (`.`), &.

Character	Example
<code>*</code> is a wildcard that matches any set of 0 or more characters.	<pre>\$ ls *.sv</pre> Lists all the files ending in <code>.sv</code> <pre>\$ rm -rf *</pre> Removes all files in the current directory and any subdirectories. Use with caution!
<code>.</code> is the current directory.	<pre>\$ mv tutorial/README .</pre> Moves <code>README</code> from the <code>tutorial</code> subdirectory to the current directory.
<code>&</code> runs a command in the background so that the user can continue using the terminal.	<pre>\$ vsim &</pre> Runs the ModelSim simulator, opening the window in the background.

Pressing the TAB key will autocomplete what you are typing, which is helpful if you are typing long filenames.

Pressing the up arrow followed by return will replay a previous command. Pressing the up and down arrows will move you back and forth through the history of previous commands. Pressing the left and right arrows allows you to move within a command to edit it. Pressing Ctrl-a or Ctrl-e will bring you to the beginning or end of the current command.

Filenames starting with a `.` (e.g., `.bashrc`) are called *dotfiles*. They are usually used for application configuration and are not displayed by `ls`. Use `ls -a` (list all) to show dotfiles. Combining the command options after the dash does them all. For example, `ls -al` shows the long listing for all files.

B.3.2 Standard Input and Output

Linux commands communicate with three *streams* called *standard input* (*stdin*), *standard output* (*stdout*), and *standard error* (*stderr*). For example, the `ls` command prints the contents of a directory to *stdout*. When you invoke a command from a terminal, these streams default to the terminal, so *stdout* and *stderr* print to the terminal and *stdin* receives keystrokes from the user.

Linux allows you to *redirect* streams or *pipe* them from one process to another as illustrated in Table B.3.

Table B.3 Commands to redirect streams.

Command	Example
> redirects stdout to a file.	<pre>\$ ls *.sv > svfilenames</pre> Puts a list of all the files ending in <code>.sv</code> into a file named <code>svfilenames</code> rather than displaying the list on the screen. Note that error messages are still printed to the screen.
pipes the one command's stdout to another command's stdin	<pre>\$ ls -l less</pre> Lists all files in a directory, one page at a time.
tee pipes its stdin to both stdout and one or more files. This is helpful to see the output of a tool while also generating a log file to review later.	<pre>\$ ls *.sv tee svfilenames</pre> Lists all files ending in <code>.sv</code>, printing them on the screen and also writing them to the <code>svfilenames</code> file.

B.3.3 Common Commands

Table B.4 lists other common Linux commands.

Table B.4 Common Linux commands.

Command	Example
man prints a manual page (information page) for a Linux command, including all the options.	<pre>\$ man ls</pre> LS(1) User Commands LS(1) NAME ls - list directory contents SYNOPSIS ls [OPTION]... [FILE]... DESCRIPTION List information about the FILEs (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor --sort is specified. Mandatory arguments to long options are mandatory for short options too. -a, --all do not ignore entries starting with .
which prints the path to where a command is found.	<pre>\$ which ls</pre> /usr/bin/ls

(Continued)

Table B.4 Common Linux commands. Cont'd

Command	Example
grep (global regular expression print) searches files for a string and prints where it is found in the specified files. The string may be simple text, or may be a <i>regular expression</i> (see The Linux Command Line for examples).	<pre>\$ grep XLEN *.sv *.vh</pre> Prints all the places XLEN appears in any file ending with .sv or .vh. <pre>\$ grep XLEN *</pre> Prints all the places XLEN appears in any file in the current directory <pre>\$ grep -r XLEN</pre> Prints all the places XLEN appears in any file in the current directory or recursive subdirectories
diff compares two files and prints the differences.	<pre>\$ diff file1 file2</pre> Compares file1 and file2 and prints the differences.
sort prints a file with the lines in sorted order.	<pre>\$ sort file > file1_sorted</pre>
wc prints the number of lines, words, and bytes in a file.	<pre>\$ wc file1.txt</pre>
find finds all the files in the specified directory (including its recursive subdirectories) matching a particular criterion.	<pre>\$ find . -name *.sv</pre> Prints the paths to all the files in the current directory and below, whose names end with .sv.
du reports the disk usage (size of files). Use the -h option to print in human-readable units of KiB, MiB, or GiB.	<pre>\$ du -h</pre> <pre>1.0K ./tutorial 2.0K .</pre>
ln -s creates a symbolic link to a file. A symbolic link is like a shortcut in Windows or an alias on MacOS. Accessing the symbolic link is equivalent to accessing the file it points to.	<pre>\$ ln -s tutorial/README README4</pre> <pre>-rw-r----- 1 lynn users 7 Nov 26 08:26 README3 lrwxrwxrwx 1 lynn users 15 Nov 28 10:15 README4 -> tutorial/README drwxrwxr-x 2 lynn users 3 Nov 26 06:29 tutorial</pre>

B.3.4 Working With Processes

Table B.5 lists commands for working with processes, including using arrow keys to run recent commands, displaying the history of the last commands, terminating processes, etc.

Typing Ctrl-c will usually terminate a process that is running. If that doesn't work, open another terminal, use ps to find the PID of the process, and then use kill <PID> to kill it.

Table B.5 Commands for working with processes.

Command	Example
Pressing up arrow on the keyboard brings back the previous commands to the cursor.	\$ <code>ln -s tutorial/README README4</code> Recalls previous commands to save typing.
history prints the previous commands you entered.	\$ <code>history</code> ... 102 <code>ls *.sv > svfilenames</code> 103 <code>ls -l less</code>
!! repeats the previous command. ! <code><string></code> repeats the last command starting with string. ! <code><number></code> repeats a numbered command reported by history.	\$ <code>!!</code> runs <code>history</code> again (if it was the last command). \$ <code>!c</code> runs <code>chmod</code> again (if it was the last command starting with <code>c</code>). \$ <code>!103</code> runs <code>ls -l less</code> again (see history above).
top displays the processes with the greatest load on the computer. htop gives more information on multiprocessor systems.	\$ <code>top</code> Press 1 to view the load on each CPU. \$ <code>htop</code>
ps displays the processes currently running, including their process ID (PID) and fraction of CPU and memory they are consuming.	\$ <code>ps -u</code> Prints all the processes you are running.
kill terminates process specified by its PID. This is especially useful if you have a runaway process taking up CPU time. Add the -9 flag to forcefully kill a process if the regular kill command doesn't work.	\$ <code>kill <PID></code> \$ <code>kill -9 <PID></code>
killall terminates all processes running the specified command.	\$ <code>killall vsim</code> Kills all <code>vsim</code> jobs.

Typing Ctrl-z puts a process to sleep. Then type `bg` to restart the process in the background (running concurrently with the terminal) or `fg` to restart the process in the foreground (tying up the terminal until complete.) Multiple processes can run in the background. The `jobs` command will list an integer associated with each process. `bg` and `fg` followed by % and the integer restarts the specific process; for example, `bg %2` restarts process 2 in the background.

B.3.5 Compressing, Bundling, and Transferring Files

Table B.6 shows commands for compressing, bundling, and transferring files. Together, these are especially useful for large files.

Table B.6 Compressing, bundling, and transferring files.

Command	Example
gzip compresses files. It works best for text files with repetitive or structured content, achieving compression of around 50% for a plain text novel to 90+% for more repetitive files. gzip -d or gunzip decompresses.	<pre>\$ ls -l -rw-rw-r-- 1 lynn users 1232922 moby.txt \$ gzip moby.txt \$ ls -l -rw-rw-r-- 1 lynn users 505050 moby.txt.gz \$ gunzip moby.txt.gz</pre>
tar (“tape archive”) creates a <i>tarball</i> , a single file containing all the files and subdirectories in a directory, making it easier to move directories between systems. It optionally compresses the files. -c Create the archive -x eXtract the archive -v Verbose: display which files are archived -f Filename of archive -z gZip compress the archive	<pre>\$ tar -czvf archive.tar.gz tutorial Archives and compresses the tutorial directory and all its contents \$ tar -xzvf archive.tar.gz Uncompress and expand the archive</pre>
rsync is a flexible tool to copy/synchronize files or directories. Directories can be remote or local, and rsync can efficiently sync only modified files or files that do not exist at the destination.	<pre>\$ rsync lynn@vlsi.hmc.edu:archive.tar.gz Copies archive.tar.gz from lynn’s home directory on vlsi.hmc.edu to the current directory where you issued the command. If the files already exist, it only copies the modified ones. \$ rsync -auv --info=progress2 src/mmu lynn@vlsi.hmc.edu:src/new Verbosely copies the src/mmu directory and all its contents from the local machine into the src/new directory on the remote computer. It preserves permissions, copies only files which are newer on the local machine, and provides a progress bar.</pre>

B.4 LINUX PRODUCTIVITY

B.4.1 Multiple Terminal Windows

It’s usually helpful to have multiple terminal windows open when you are working on a complex project. Keep one terminal in each directory where you are working so you aren’t constantly changing directories back and forth. Desktop environments usually have a built-in menu or button to

open a terminal with nice features, including scroll bars, copy and paste, and multiple tabs.

When working over `ssh` without a graphical user interface (GUI) or over a slow connection, `tmux` provides a multi-window terminal environment in a single session. `tmux` also provides persistent terminal sessions which can continue executing commands after an `ssh` session has disconnected. You can start a `tmux` session with `tmux new -s <sessionname>` and can join an existing session with `tmux -2 attach -t <sessionname>`. You can kill a session with `exit` inside a session or `tmux kill-session -t <sessionname>` outside a session. While in a session, you can use `Ctrl-b d` to disconnect. `Ctrl-b c` creates a new window. `Ctrl-b n` or `p` moves to the next or previous window.

B.4.2 Text Editors

Linux has many text editors, with `vim` and `emacs` being the most widely used console-based editors. Each has a base of fanatical users, and each is rather cryptic for new users.

`vim` is an improved version of the older `vi` editor. If `vim` is not installed on your system, try `vi` instead.

The free `vim` user manual is at www.eandem.co.uk/mrw/vim/usr_doc/index.html.

`vim` is fast and almost universally available but particularly cryptic. To open the `mmu.sv` file in `vim`, type `vim mmu.sv`. `vim` opens in *command mode*. In this mode, you can use the arrow keys to move around. Typing keys will invoke commands rather than making text appear. For example, typing `dd` will delete the current line. Press `i` to enter *insert mode*. Now you can type text as expected. Press the Escape key to return to command mode. `u` undoes the last command. `/text` searches for `text` in the document. `p` pastes in the last line or multiple lines you deleted. `:w` writes (saves) the file. `ZZ` saves and quits `vim`. `:q!` quits without saving. Expect to use `vim` for several days before you become comfortable with it. The editor has many powerful features that experienced users love.

Linux also has a variety of GUI-based editors that are more familiar to Windows and Mac users, such as `code` (Visual Studio Code) and `subl` (Sublime). You may need to install these editors yourself. They are not particularly responsive over an `x2go` connection.

Our preferred editing method for large projects, as long as a good network connection is available, is to run Microsoft's Visual Studio Code (also called VSCode, pronounced "V S code") natively on your laptop with an SSH connection to the server. VSCode is free and runs on Windows, Mac, and Linux. It is presently used by about 70% of developers. The idiosyncratic interface takes a little while to get used to but works well after you learn it. Download and install it from code.visualstudio.com.

To install extensions, click on the gear icon in the lower left and select Extensions. Browse for and install Remote — SSH. We also recommend installing other extensions:

- ▶ Verilog-HDL/SystemVerilog/Bluespec (for syntax highlighting)
- ▶ GitLens (helps track modifications to lines of files in a Git repository)

Click on the blue status bar icon with > < symbols in the lower left to set up an SSH connection to your server. Choose Connect to Host... and enter your hostname (e.g., `tera.eng.hmc.edu`). You will be prompted for your password. Then choose File * Open Folder and enter the path to your repository (e.g., `/home/lynn/cvw`). Now you are editing the files on the server directly.

B.4.3 The Linux File System

It's helpful to get to know your way around the Linux file system to be able to find files outside your home directory. Most Linux distributions are organized according to the Filesystem Hierarchy Standard summarized in [Table B.7](https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html) [refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html].

Table B.7 Linux file system organization.

Directory	Contents
/	The root of the file system
/home	User home directories, such as <code>/home/lynn</code> and <code>/home/ben</code> . If your permissions are set appropriately, you may be able to share files between users from your home directories.
/bin /sbin /lib	Binary executable files and shared libraries needed to boot the system. <code>/bin</code> holds most common commands like <code>ls</code> and <code>cd</code> . <code>/sbin</code> holds system binaries primarily used by system administrators (sysadmins). <code>/lib</code> holds libraries such as the standard C library <code>libc.so</code> .
/boot	Bootloader files such as the Grand Unified Bootloader GRUB, the compressed Linux kernel <code>vmlinuz</code> , and the root filesystem image <code>initramfs</code> .
/etc	Configuration files, such as <code>/etc/passwd</code> that holds hashed versions of user passwords.
/usr /usr/bin, /usr/lib /usr/local/bin, /usr/local/lib /opt	Additional software not required for system boot. Most built-in Linux commands are in <code>/usr/bin</code> Manually-installed tools go in <code>/usr/local/bin</code> or <code>/opt</code> .
/dev	I/O devices can be accessed using Linux file commands located in <code>/dev</code> .
/mnt	A place for extra filesystems to be mounted, such as USB memory sticks, CD-ROMs, and external hard disks.
/tmp	A place to store temporary files that could be deleted when rebooting.

Linux systems may have other directories such as `/cad`, `/courses`, or `/proj` for sharing software and files across users.

B.4.4 Environment Variables

Linux uses *environment variables* to configure how the system operates. Environment variable names start with `$` and are usually in upper case. One of the most important variables is `$PATH`, which defines an ordered list of directories in which Linux searches for commands. Use the `echo` command to print your path, which may look like:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/sbin:/sbin
```

Use the `export` command to change an environment variable and make the new value visible to other programs. For example, to add a path for a newly installed CAD tool:

```
$ export PATH=/cad/verilator/bin:$PATH
$ echo $PATH
/cad/verilator/bin:/usr/local/bin:/bin:/usr/sbin:/sbin
```

Nobody wants to have to modify the path every time you log in, so paths are generally set in a configuration file. Depending on your local configuration, the `.profile` or `.bash_profile` file will be sourced (executed) when you open a terminal. You can add the necessary `export` statements. Better yet, if the configuration is needed by many users, add a line to `.profile` or `.bash_profile` to source a shared configuration file:

```
source /cad/scripts/setups/S21/riscv-setup
```

The `export` statement by itself lists all the environment variable settings. Other common environment variables you might need to configure include:

- ▶ `LD_LIBRARY_PATH`: like `PATH` but used when searching for shared library files
- ▶ `LM_LICENSE_FILE`: points to a license manager, commonly for paid CAD tools

B.4.5 Web Browsing

Web browsing is generally less responsive from a Linux server than from your local computer. If you need to open a browser to download a file,

`firefox` is commonly available. Or download the file with a local browser to your local machine and `rsync` it to the server.

B.4.6 Document Viewing and Printing

`evince` and `okular` are common Linux tools for viewing PDF, Postscript, and other image file formats.

Printing from Linux can be finicky. If your server has printers set up, follow your site's documentation. Otherwise, your best bet may be to generate plain text or print to PDF and then `rsync` the text/PDF file to your local computer for printing. Or do a screen capture with software from your local computer (Snipping Tool on Windows or Screenshot on MacOS).

B.4.7 Installing Packages

Different flavors of Linux have different tools for installing packages of new software. The primary package management tool for RedHat is `dnf` (formerly `yum`) and for Ubuntu is called `apt`. One software package often depends on others, so the package manager handles dependencies, but this can be imperfect in Linux. Not all packages have an installer. Installing new software under Linux is frequently an exercise in frustration.

Usually you need superuser (root) permission to install new packages (with the `sudo` command). Such permission also gives you the power to accidentally break your server. If you are learning Linux for the first time from this tutorial, you aren't ready for superuser privileges. Ask your sysadmin for help. Don't be surprised if a sysadmin is reluctant to install new software on a production server, however.

B.5 SCRIPTING AND PROGRAMMING

Linux is particularly powerful as a development environment because of the ease of scripting and programming.

B.5.1 Shell Scripts

A shell script is a text file with a series of commands that get executed as if you typed them at the command line. It conventionally may have a `.sh` suffix.

For example, create a file called `hello.sh` containing:

```
echo Hello World!
```

Then run it using:

```
$ source hello.sh
```

See [Schotts19] for much more on shell scripting, including if and for statements.

If you don't want to type `source` to invoke a script, make the script executable using `chmod` and run it directly:

The `chmod 755 <filename>` command makes the file readable, writable, and executable for the user but only readable and executable for everyone else (`rwxr-xr-x`). To make a script executable for everyone, you could also type:

```
$ chmod +x hello.sh
```

`+r` and `+w` options also exist to make a file readable or writable for everyone.

`{1}` is the first argument on the command line.

```
$ chmod 755 hello.sh
$ ./hello.sh
```

For security reasons, your current directory (`.`) shouldn't be in your `$PATH`. Therefore, to run executable programs in the local directory, prefix them with `./`.

This prevents a malicious user from creating a file in their home directory with the same name as a standard system command (e.g., `/home/lynn/ls`) and then tricking the system administrator into running `ls` in the user's directory and inadvertently running the malicious program.

Another good place to keep personal scripts is in `~/bin`, which you can add to your path, or `~/local/bin`, which may already be in your path.

Scripts are handy to automate tedious tasks. For example, suppose you frequently `rsync`'d files between two computers in directories that were long and tedious to type. Using `cat` or a text editor, create the following `~/bin/mysync` script and make it executable using `chmod`. Then invoke the script using the name of the file to move, such as `foo.sv`, as shown below.

```
$ mkdir ~/bin // Only if necessary
$ cat > ~/bin/mysync
#!/usr/bin/bash
rsync -auv --info=progress2 src/${1}
lynn@vlsi.hmc.edu/src
<Ctrl-d>
$ chmod 755 ~/bin/mysync
$ mysync foo.sv // Linux finds the script if ~/bin is in your PATH
```

B.5.2 Python

Many good books and classes on Python programming exist. *Python for Everyone* is available freely [Severance16].

Python is a widely-used and easy-to-learn scripting language that comes preinstalled on most Linux distributions. To write a simple Python script, create a file called `hello.py` containing:

```
print("Hello World!")
```

Then run it using:

```
$ python hello.py
```

Alternatively, you can make a script in any language executable by adding a special comment starting with `#!` on the first line telling Linux which program to use to run the script. Change the script to read

```
#!/usr/bin/env python
print("Hello World!")
```

Make it executable, and then run it:

```
$ chmod 755 hello.py
$ ./hello.py
```

`/usr/bin/env` finds the first executable that matches the specified name (e.g., `python`) in your `$PATH`. This makes scripts more portable because they will work even if the Python executable is in a different place on different servers.

B.5.3 C

C is a good programming language for system programming and working with hardware because pointers allow you to access memory explicitly. The GNU Compiler Collection (GCC) comes installed in most Linux distributions and has a C compiler. To write a simple C program, create a file called `hello.c` containing:

```
#include <stdio.h>

void main(void) {
    printf("Hello World!\n");
}
```

Compile and run it using:

```
$ gcc -o hello hello.c
$ ./hello
```

Many good books and classes on C are available. A good free online book is *C Programming* [C17]. *The C Programming Language*, 2nd edition, is the classic book by the language inventors Kernighan and Ritchie.

B.5.4 Makefiles

Large programs are composed of many files and often require specifying command line options for the compiler. If you need to recompile a program, it's a hassle to try to remember exactly what command is needed to compile. Thus, it would be convenient to put the compile command in a standardized script. Moreover, when only one of the files has been changed, it would be faster to just recompile and relink that one file rather than rebuilding the entire program. These issues are addressed with *Makefiles*.

The GNU make documentation [Stallman22] is helpful. For an authoritative and free book on Make, see *Managing Projects with GNU Make* [Mecklenburg04].

Be sure to indent the command by pressing tab. Make will fail with a confusing “missing separator” message if you indent with spaces.

The `make` command looks for a file called `Makefile` in the local directory. The `Makefile` consists of a series of rules in the form:

```
target : dependencies
        command
```

To build `target`, `make` checks if any of the dependencies have changed since `target` has last changed. If they have, it issues the command to rebuild `target`.

For example, consider building a C program called `example` that has three source files `a.c`, `b.c`, and `c.c`, and two header files `x.h` and `y.h`.

```
objects = a.o b.o c.o

example : $(objects)
        gcc -o example $(objects)

a.o: a.c x.h y.h
        gcc -c a.c
b.o: b.c x.h
        gcc -c b.c
c.o: c.c y.h
        gcc -c c.c

clean:
        rm example $(objects)
```

The first rule is the default goal. Hence, typing `make` has a goal of building `example`. `example` depends on object files `a.o`, `b.o`, and `c.o`, so these targets are checked first. If any of their source files have been changed, they get rebuilt. For example, if the *source* files `a.c`, `x.h`, or `y.h` have modification dates newer than the *object* file `a.o`, `a.o` will be rebuilt using the command `gcc -c a.c`. After all the object files are rebuilt, `example` gets built from the object files. The first time `make` is run, everything must get compiled.

```
$ make
gcc -c a.c
gcc -c b.c
gcc -c c.c
gcc -o example a.o b.o c.o
```

Running `make` again does nothing because `example` and all its dependencies are already up to date. If we changed `y.h` or run `touch y.h` (which changes the modification date to now) and then `make` again, `a.o`, `c.o`, and `example` are remade, but `b.o` is already up to date because it doesn't depend on `y.h`.


```
$ touch y.h  
$ make  
gcc -c a.c  
gcc -c c.c  
gcc -o example a.o b.o c.o
```

Running `make clean` executes the `clean` goal instead of the default goal. In this case, it removes all the compiled output files, leaving only the source files.

```
$ make clean  
rm example a.o b.o c.o
```

Version Control Using Git



Version control is a method of keeping track of changes to a set of files. For example, you might save a copy of your project report as `project_old.docx` before making revisions. The next night, you might save the final copy as `project_final.docx`. Then your lab partner might make some more edits and save it as `project_final_ah.doc`. Then you might fix a bug and resave it as `project_final_notkiddingthistime.docx`. If you ever had to revisit your work, you might have difficulty determining which version is most up to date or what changes were made. If you and your partner made changes at the same time, you might overwrite each other and lose part of your work in a way that is very difficult to tell which changes were entered and which were lost. If your disk crashes, you'll lose all the versions. The problems are obviously worse for large code development projects involving many people and a large number of files over an extended period of time. Software developers use version control software, notably Git, to solve these problems. This appendix introduces the features of Git that we regularly use while designing a chip in an open-source collaboration.

The general idea of Git is to maintain a *repository* of all the files in a project with *snapshots* every time a user *checks in* changes. A Git project has at least one *remote repository*, often hosted on a website such as github.com where collaborators can easily access it. Each user *clones* the entire repository to their own *working copy* (e.g., in their home directory on a Linux server), also called a *local repository*. Users can *modify* the local repository and test the changes. The user then *stages* modified files for inclusion in the next *snapshot* and then *commits* the staged files to create a new snapshot. Each snapshot is associated with a *hash* to identify the snapshot and ensure its integrity. When a connection to the internet is available, the user can *push* local repository changes back to the remote repository or *pull* changes made by other users. Git fetches the snapshot and attempts to *merge* the changes. If more than one user edited the same line of code, a *merge conflict* occurs, and the user must review the conflict and decide which change(s) to accept. Git also allows you to *branch* a repository so that part of the team can experiment on a portion of the code without interfering with the main branch and then merge the branch back into the main branch after it is completed. Sometimes this leads to tricky merge conflicts if substantial and different changes were made to the same files in both branches. Git keeps an archive of all snapshots, so it is always possible

- C.1 [Gitting Started](#)
- C.2 [Setting Up a Repository](#)
- C.3 [Basic Git Flow](#)
- C.4 [Snapshots and HEAD](#)
- C.5 [Merge Conflicts](#)
- C.6 [Branching and Merging](#)
- C.7 [Tags](#)
- C.8 [Staging and Undo](#)
- C.9 [Submodules](#)
- C.10 [Other Git Capabilities](#)
- C.11 [Summary](#)

Git has far more capability than this short tutorial can cover. Learn more from the freely available *Pro Git* book [Chacon14].

Git was developed by Linus Torvalds in 2005. (And, yes, he was also the main developer of Linux. In fact, he developed Git to support the development of Linux.) Git has largely supplanted older version control tools such as Concurrent Versions System (CVS) and Subversion because it is more flexible, is a distributed tool resistant to failures of a centralized server, and offers easy worldwide collaboration through sites such as GitHub.

to revert if a change proves problematic. Git is designed and optimized for managing text-based source code files of small to medium size. It is not good for huge files (10s of megabytes or more) or binary files. Don't place executable files in a Git repository; instead, keep the source code and the Makefile necessary to regenerate the executable.

Although Git remote repositories can be hosted anywhere, including on a company's internal server or a home computer, many open-source repositories are hosted on GitHub, which is now owned by Microsoft. GitHub provides free repository disk space and is a convenient way for multiple people to collaborate on a shared project.

In a larger project with many contributors, a central *upstream* repository is maintained by *committers*, who review and approve contributions. A user who simply wants the code can clone the upstream repository. A user who wants to contribute code makes a *fork* of the upstream repository into her own GitHub account or private server. She then makes changes and tests them in her own fork. When they are complete, she makes a *pull request*, asking the committers to incorporate the new or modified code from the fork into the upstream repository. She should periodically sync her fork with the upstream repository to see the latest contributions from others and to avoid merge conflict surprises. In this workflow, the fork and upstream repositories are both remote repositories, while the clone on your computer is a local repository.

This section uses the terms *repository*, *repo*, and *project* synonymously. GitHub also has *project boards*, which are used to organize tracking issues, pull requests, and other notes. Projects and project boards are entirely different concepts.

C.1 GITTING STARTED

Before you do anything else with Git, you must configure your username and email, which are included when you commit snapshots. Also set Git to merge when pulling so you don't get warnings later on. Be sure to replace your email with the exact one you've used for your GitHub account.

```
$ git config --global user.name "Ben Bitdiddle"
$ git config --global user.email "ben_bitdiddle@wally.edu"
$ git config --global pull.rebase false
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

In all these examples, of course replace Ben's name and email with your own.

To use GitHub as your remote repository, register for an account at github.com if you don't already have one. Then set up a personal access token to access your repositories via HTTPS. Go to <https://github.com> and log in. Click on your profile icon in the upper right corner of the window and choose Settings. At the very bottom of the left sidebar, choose Developer settings, then click Personal access tokens → Tokens (classic) and then Generate a personal access token. Set the expiration date to Custom and choose a date a year away. Check the repo box under Select

scopes, then scroll to the bottom and click Generate token. Copy your token and save it some place secure on your computer because you will never be able to see it again on GitHub. It should be of the form `ghp_<long string of random letters and numbers>`.

The first time you perform a `git push` or other command that requires authentication, you will be prompted for your username (from your GitHub account) and password (enter the token that you recorded earlier). Git's *credential helper* will store your authentication information, so you do not need to reenter it in the future. Unfortunately, it is presently stored in unencrypted format.

The GitHub authentication process has changed over time. If this process no longer works by the time you read it, look at docs.github.com or search the internet for the latest procedure.

C.2 SETTING UP A REPOSITORY

To start an entirely new project, create a new directory, initialize a repository in that directory, and optionally associate it with a remote repository.

```
$ mkdir ~/my_project
$ cd ~/my_project
$ git init
$ git remote add origin https://github.com/<yourgithubid>/my_project
```

To begin working with an existing GitHub repository, such as Wally's `cvw`, fork the upstream repository to your own GitHub account and then clone your fork. This will create a working copy in `cvw` within your current working directory. In a web browser, visit Wally's GitHub repository: github.com/openhwgroup/cvw. Click the Fork button and then the Create Fork button to make a forked repository in your own GitHub account (e.g., github.com/<yourgithubid>/cvw). Then clone the fork into a directory where you want to work (often a Linux server, but it could be a PC or Mac where you want to review files) and point Git to the upstream repository.

In these commands, replace `<yourgithubid>` with your GitHub username, with no brackets.

```
$ git clone --recurse-submodules https://github.com/<yourgithubid>/cvw
$ cd cvw
$ git remote add upstream https://github.com/openhwgroup/cvw
```

A GitHub repository can be *private* or *public*. Typically, a team starts with a private repository and adds collaborators manually. If the team wants to release the project, the owner changes the repo to public.

C.3 BASIC GIT FLOW

Before starting a work session, it's a good idea to make sure you have the latest version of the repository:

```
$ git pull
```

The `--recurse-submodules` flag recursively checks out any other GitHub repositories that are submodules within the main repo.

If you are working on a forked repository, you can pull in changes from the upstream repo using:

```
$ git pull upstream main
```

Check that you don't have any modified files from a previous session:

```
$ git status
```

Now modify existing files and/or create new ones. Be sure to test your changes before you put them back in the repository.

To move or rename a file, use:

```
$ git mv <old name> <new name>
```

However, when you ask Git for the history of this file using `git log` or `git diff`, it will only show the history up to the last move. To get the full history, use:

```
$ git log -follow <new name>
```

When you are satisfied with your changes, check the status again:

```
$ git status
```

Your modified files will be listed under “Changes not staged for commit” and your new files will be listed under “Untracked files.” You can list the changes in a file using:

```
$ git diff <file>
```

If any of the modifications are bad, you can roll back to the version in the repository using:

```
$ git restore <file>
```

When you are satisfied, stage the files to be committed:

```
$ git add <file>
```

If you’ve deleted any files from the project, remove them from the repository as well. This only removes them from future checkpoints; you can always roll back to an earlier version where the file existed.

```
$ git rm <file>
```

If you run `git status` again, you should see all your files listed under “Changes to be committed.”

If you change your mind after staging a file, you can take it out using:

```
$ git restore --staged <file>
```

Then commit using the `-m` flag to add a meaningful message:

```
$ git commit -m "Fixed undeclared mmu/PhysAdr signal causing X in simulation"
```

The commit creates a *snapshot* of the repository at that instant. The commit is identified by a *hash*, which is a unique 40-digit hexadecimal number. You can see a log of recent commits along with the author, date, and message.

```
$ git log
commit 14d3059433e212205ebf30b64ffe71d467dabb94
Author: David Harris <david_harris@hmc.edu>
Date:   Fri Jan 21 00:12:14 2022 +0000

    Fixed path to riscv0VpsimPlus

commit 55e4d09084caa95cebfc36b778de16f5b8e051b3
Author: Rose Thompson <rose@rosethompson.net>
Date:   Thu Jan 20 16:39:54 2022 -0600

    Factored out InstrValidNotFlushedM from each csr*.sv to csr.sv
...
```

Always test your code before returning it to the remote repository so you don’t break things for anyone else. Then push it:

```
$ git push
```

If anyone else has made changes, you’ll have to pull their changes again before you can push. If you’re working from your own fork, nobody else will be making changes to your fork, but it is prudent to do a `git pull upstream main` before pushing to collect any changes from the upstream repository. Git will automatically merge the changes if possible.

Sometimes it is tedious to `git add` files one-by-one. `git commit -a -m "message"` will add all the tracked files and then commit in one fell swoop.

Git can identify a snapshot based on the first 7 digits of the hash (called the *short hash*) except in the unlikely event another snapshot shared the same digits. Fig. C.1 shows just the first 3 digits.

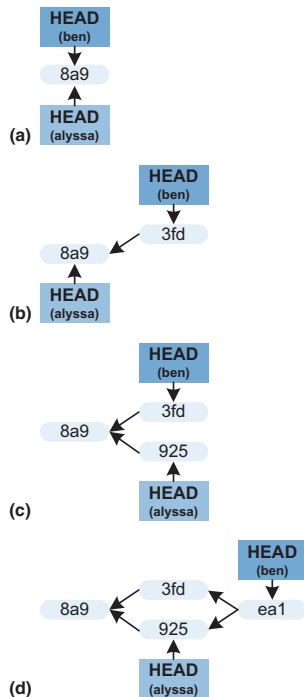


Fig. C.1 Snapshots and merging.

To get your commits back into the upstream repository, make a pull request. Only make pull requests for changes that are important for all users to receive in the upstream repository.

In a web browser, go to your forked repository (e.g., github.com/<yourgithubid>/cvw). You may see a message that you had recent pushes next to a Compare & pull request button. If that is not showing, you can click on Pull Requests, then New pull request, and then set the base repository to the upstream one ([openhwgroup/cvw](https://github.com/openhwgroup/cvw)) and the HEAD repository to your fork ([<yourgithubid>/cvw](https://github.com/<yourgithubid>/cvw)). Add a descriptive title, review the changes you've made, and click Create pull request. The pull request will appear in the upstream repository. A committer will review your request and approve it, request changes, or deny and close the request.

C.4 SNAPSHOTS AND HEAD

Each commit creates a new snapshot of the entire state of the repository linked back to all previous snapshots. Every user has a HEAD, which points to the snapshot that user is presently viewing. Most commonly, HEAD points to the tip of the main branch, but later we will discuss how to check out an old snapshot or make a different branch for exploratory work.

For example, suppose Ben and Alyssa are both working on a project. They both pull a snapshot of the main branch with the hash 8a9. Now both of their HEADs are pointing to that 8a9 snapshot, as shown in Fig. C.1(a). Next Ben makes a change and commits it, creating a new snapshot with hash 3fd, and advancing his HEAD to 3fd (Fig. C.1(b)). Meanwhile, Alyssa makes a change and commits it, creating a new snapshot with hash 925 (Fig. C.1(c)). Both of their snapshots are based on the original 8a9 snapshot. Now Alyssa pushes her change to the remote repository. Next, Ben attempts to push but gets an error because “the remote contains work that you do not have locally.” He must pull again, causing his changes to be merged with Alyssa’s, creating a new snapshot ea1 (Fig. C.1(d)). He can then push ea1 to the remote repository.

C.5 MERGE CONFLICTS

Suppose Ben and Alyssa both modified `hello.c` to add more enthusiasm. Ben capitalized `WORLD` in snapshot 3fd, and Alyssa added a bunch of exclamation points in snapshot 925. When Alyssa pushes her changes and then Ben attempts to pull, a merge conflict occurs because Git does not know whose changes to accept. Git will report:

```
CONFLICT (content): Merge conflict in hello.c
Automatic merge failed; fix conflicts and then commit the result.
```

Looking at `hello.c` will show the merge conflict between Ben’s HEAD and the 925 snapshot in the remote repository:

```
<<<<<< HEAD
printf("Hello WORLD!");
=====
printf("Hello World!!!!");
>>>>>> 0b2d6c97f4265819281bf1a8d77698eb9ff30925
```

To fix the merge conflict, Ben must edit these five lines and pick one or the other or a hybrid of the two `printf` statements and delete the other four lines of the message about the conflict, leaving a single line such as:

```
printf("Hello Alyssa!!!!");
```

Then he can commit the new `hello.c` and push it to the remote repo.

C.6 BRANCHING AND MERGING

By default, Git keeps your work in the `main` branch. When multiple developers are working concurrently, it is sometimes helpful to create another branch to write and debug code before integrating back into `main` so that instabilities in the development branch don't interfere with progress in the `main` branch.

Suppose Ben creates a new branch called `muldiv`:

```
$ git branch muldiv
```

Now Ben's `main` and `muldiv` both point to snapshot `eal` in Fig. C.2(a), and Ben's `HEAD` still points to his `main`. To switch Ben's `HEAD` to the branch:

```
$ git checkout muldiv
```

Now Ben can make changes and commit them and push them. They will affect the `muldiv` branch but not `main` (Fig. C.2(b)). The first time pushing to a new branch, Ben must also set the origin:

```
$ git commit -a -m "New muldiv branch changes"
$ git push -u origin muldiv
```

If he wishes to return to work on `main`, he does:

```
$ git checkout main
```

Now Ben's `HEAD` points to `main` (Fig. C.2(c)). He can make more changes and commit them (Fig. C.2(d)). To merge the changes from the `muldiv` branch back into `main` (Fig. C.2(e)):

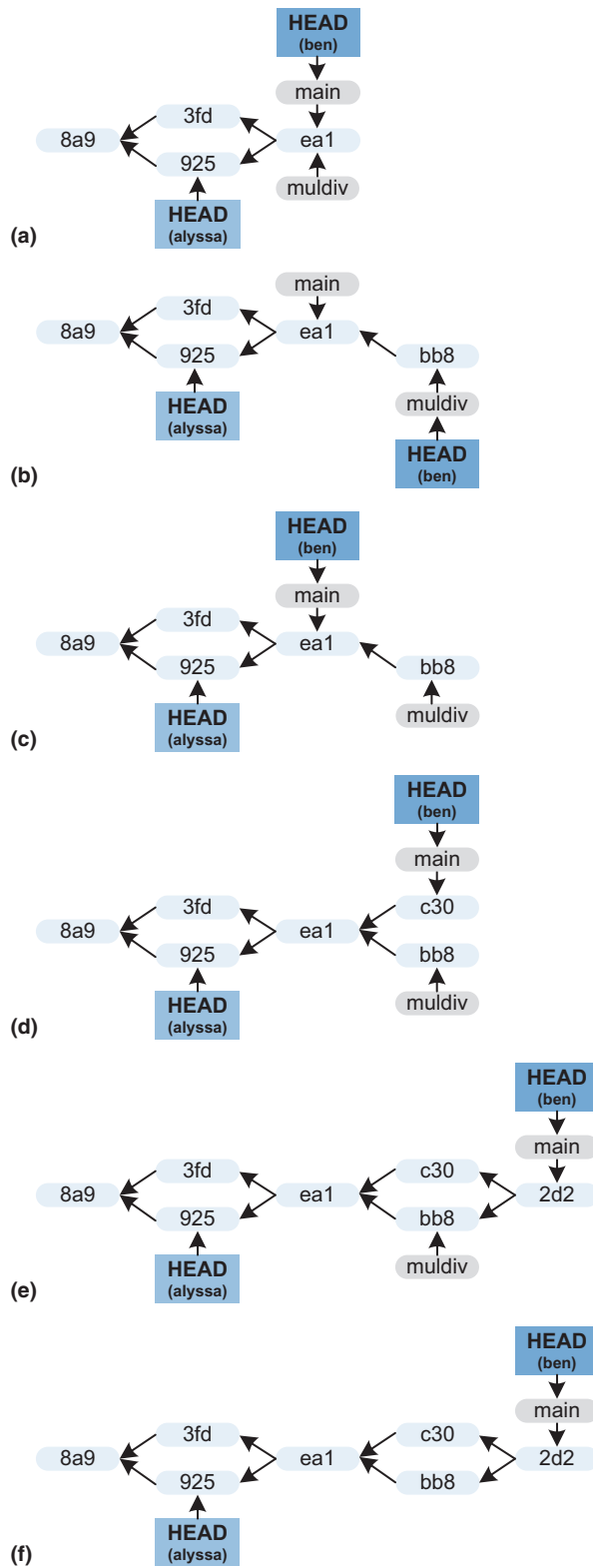


Fig. C.2 Git branches.

```
$ git merge muldiv
```

If the same piece of code received different modifications in different branches, he may have to resolve the merge conflicts manually.

Ben can list the branches that exist with:

```
$ git branch
```

Finally, he can delete the `muldiv` branch because it is no longer needed (Fig. C.2(f)):

```
$ git branch -d muldiv
```

If the commits in the branch have not been merged, Git will print a warning and not delete anything. To force the delete use `-D`. Remote branches are deleted using:

```
$ git push -d origin <branch name>
```

C.7 TAGS

Sometimes it is important to name snapshots for easy access in the future. For example, to name the current snapshot as `v2.0`:

```
$ git tag -a v2.0 -m "Version 2.0 Released by Ben Bitdiddle 30 November 2025"
```

You can get a list of all the available tags using:

```
$ git tag
v0.1
v1.0
v2.0
```

By default, tags are private to the user and aren't pushed into the remote repository. To push a tag, list its name:

```
$ git push v2.0
```

A user can check out a tagged snapshot with:

```
$ git checkout v2.0
```

C.8 STAGING AND UNDO

Each file in a Git repository exists in one of three states: committed, modified, or staged. Understanding the meaning of these states and how to move between these states is essential to using Git effectively. When a repo is first cloned, all files in the working directory will be in the committed state. If you modify a file either by changing the contents of the file or its meta data such as permissions, the file moves into the modified state. `git add` moves modified files into the staging area (also called the *index*). `git commit` creates a new commit from only the files in the staging area, and now the files are in the committed state. If multiple files are modified, it is possible to move only a subset of the files to the staging area and make a commit of just some files.

Git has several mechanisms for undoing changes.

If you have modified code in a file but have not yet staged (`git add`) or committed (`git commit`) it, you can throw it away and go back to the last committed version using:

```
$ git checkout <file>
```

If you have staged a file but not yet committed it, you can remove it from the staging area (but not discard the changes), leaving the file in the modified state:

```
$ git reset <file>
```

If you have committed a set of staged changes but not yet pushed it to the repo, reset with `--hard`:

```
$ git reset --hard hash#
```

IMPORTANT: Never perform `git reset` if commits are already pushed. Rewriting published history creates problems for other users.

`--hard` specifically tells Git to remove changes to both the staging area and the working directory. Any files tracked by Git will be set to the committed state of the hash. A common hash is `origin/main`, as it reverts the file back to the state of the main branch. `--hard` can only be used for the whole working directory. To revert a single file use `git checkout <file>`. [Fig. C.3](#) summarizes how to move between the various states.

If you have committed some changes any time in the history of a project and wish to roll back those changes and create a new snapshot without those changes but with everything else the same:

```
$ git revert <hash>
```

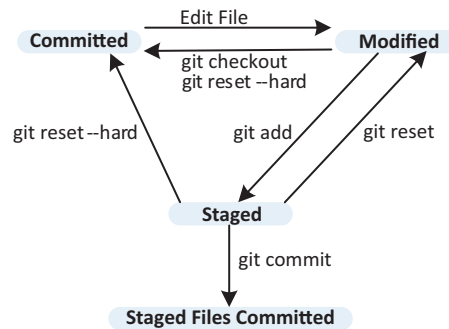


Fig. C.3 Git file states.

`<hash>` is the snapshot to undo. This is effectively the opposite of applying a patch. Typically, the resulting files will be in a committed state; however, it is possible to have a conflict. You will need to resolve the conflict, stage the changes, and then commit.

C.9 SUBMODULES

Large projects often need to incorporate other projects. For example, the `cvw` project includes test cases from the `riscv-arch-test` repository. If one simply copied the code from `riscv-arch-test` into `cvw`, it would be difficult to apply any improvements that were later made to `riscv-arch-test`. Instead, Git supports submodules to incorporate one repository within another:

```
$ git submodule add <URL>
```

To update to the latest version of a submodule:

```
$ git submodule update --remote
```

Or to pull your main repository and automatically update any submodules:

```
$ git pull --recurse-submodules
```

To clone a repository with submodules, be sure to include the `--recurse-submodules` flag so you don't have to initialize and update each submodule manually.

```
$ git clone --recurse-submodules <URL>
```

When submodules are added to a repository, `git pull` does not fetch the new submodules. Force the fetch by adding `--recurse-submodules`.

```
$ git pull --recurse-submodules
```

To update all submodules to the versions checked into the main project, use:

```
$ git submodule update --init --recurse-submodules
```

Often a submodule is owned by somebody else, and you should not modify it or attempt to push commits back into it. If you want to propose changes to the submodule, fork it and make a pull request.

C.10 OTHER GIT CAPABILITIES

Sometimes you are working on a branch in a messy state and need to switch to another branch for a while but don't want to commit your incomplete work. You can use the `git stash` command to stash your changes, then change branches, do your other work, and then `git stash apply` to restore your in-progress work.

Often, local repositories include files you don't want added to the repo, such as object files. To have Git ignore such files, add them and their path to the `.gitignore` file. Usually, `.gitignore` is in the root directory of the repo, but additional `.gitignore` files may be in subdirectories. Wildcards using “*” can be used to match many different files or even whole directories. The `.gitignore` file in [Code Example C.1](#) ignores all C object files and object dumps in any directory, as well as the `sum` and `fir` compiled binary files in certain examples directories.

Code Example C.1 `.gitignore`

```
*.o
*.objdump
examples/C/sum/sum
examples/C/fir/fir
```

To show the changes between two snapshots, use `git diff`. If you only specify one snapshot, it will be compared to your HEAD. For example, if the current HEAD is `2d2` (see [Fig. C.1](#)), to see what changed relative to snapshot `c30`, use:

```
$ git diff c30
```

C.11 SUMMARY

Git is a powerful version control tool that helps developers collaborate and systematically keep track of code versions. To use Git effectively, you need a mental model of its operations. Fig. C.4 shows the various layers of data storage in a project and the commands that move data between layers, as will be described in this summary.

The project has a remote upstream repository containing the authoritative code. For example, CORE-V Wally is curated by the OpenHW Foundation as a project hosted on GitHub at github.com/openhwgroup/cvw. Nobody directly edits the upstream repository. Instead, each developer who wants to make contributions first forks the upstream repository. For example, Ben Bitdiddle uses the GitHub site to create a remote fork at github.com/benbitdiddle/cvw. Then Ben uses `git clone` to clone the fork into a working directory on his personal computer or server, such as `/home/ben/cvw`. Now the working directory contains a local `cvw` repository that is a clone of the fork, and a workspace exactly matching the local repository. He uses `git remote add upstream` to tell the local repository that it has an upstream remote.

Ben now edits, adds, or removes files. These changes are part of Ben's workspace but are not yet in the local repository. As Ben becomes satisfied with changes, he stages them using `git add`. When he has completed a group of changes, he commits the staged changes to the local repository using `git commit`. Alternatively, he can add and commit all changes in one fell swoop with `git commit -a`. When he has an internet connection, he uploads them back to his remote fork with `git push`. Now other people can see his changes in the cloud. If he wants his changes to go into the remote upstream repository, he makes a pull request (through GitHub). An OpenHW Foundation committer reviews the pull request and may ask for changes or may merge it into the remote upstream repo.

If Ben is unsatisfied with a group of unstaged changes, he could discard them all using `git checkout`. If the changes have been staged, he can still discard them using `git checkout HEAD` to throw away the staged changes and roll the workspace back to the HEAD of the local repository. If Ben is only unsatisfied with changes to a specific unstaged file, he can discard those with `git restore`. If the file has already been staged, he can unstage it with `git restore --staged`.

Ben can examine the difference in his unstaged files using `git diff`. `git diff HEAD` compares against the local repository to also find differences in staged files.

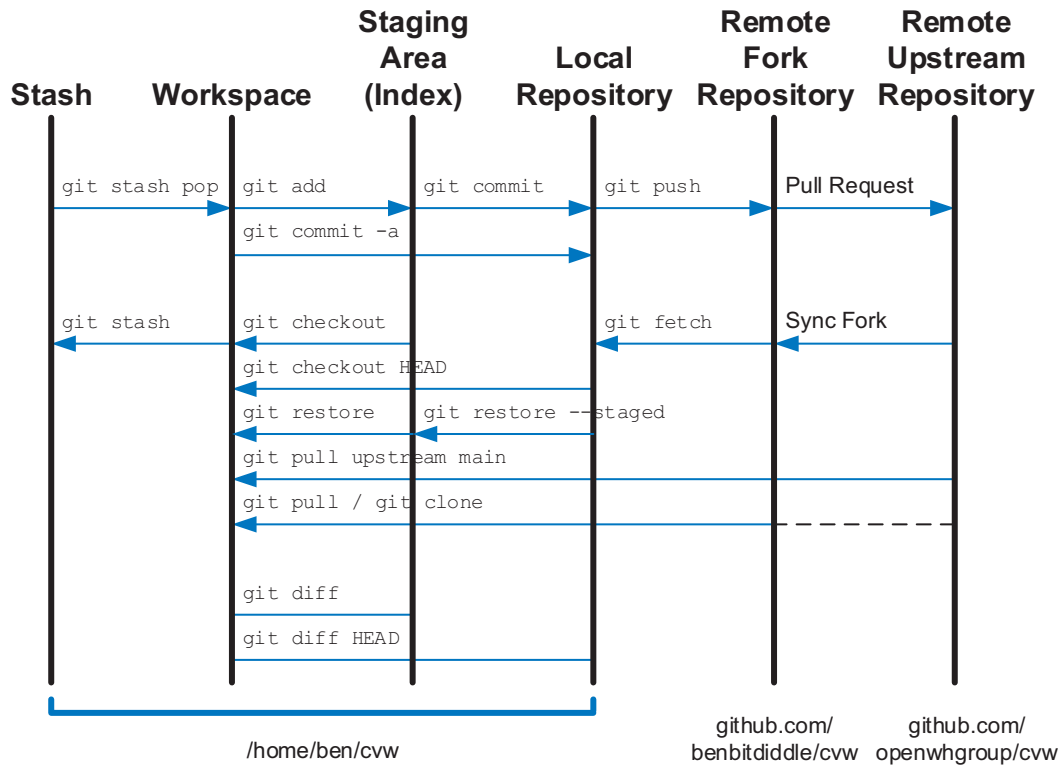


Fig. C.4 Git layers.

When others make changes to the remote upstream repository, Ben has several ways to get them. The simplest is to `git pull upstream main`, which fetches these changes from the upstream repository into the local repository and the workspace. If the remote upstream repository and the workspace both have changes to a particular file, Git will attempt to merge the changes. However, if a particular line is changed in both places, a merge conflict occurs. Ben must review the conflicting file(s) and pick which version to accept. Alternatively, Ben can sync his fork to the upstream remote on GitHub, then `git fetch` the changes into his local repository without applying them to the workspace yet.

`git stash` restores the workspace to match the local repo (like `git checkout HEAD`) but keeps a copy of the staged and unstaged changes. This is helpful to avoid merge conflicts or having to commit incomplete code before temporarily changing to a branch. `git stash pop` reapplies the stashed changes.

Branching is not shown in this diagram. `git branch` creates a new branch. `git checkout <branchname>` checks out the branch out into the workspace. Be sure to commit or stash any changes in the workspace before checking out a different branch. `git checkout main` returns to the main branch. `git merge` merges changes from another branch into the current branch.

If Ben just wants to look at the remote upstream repository but not make contributions, he can simply clone the remote upstream repo instead of making a fork. However, he will not have permission to push commits from the clone directly to the remote upstream repository.

Tcl Book of Armaments



Tcl (Tool Command Language) is an embeddable scripting language, i.e., an interpreted language implemented as a library package that can be easily incorporated into a variety of applications. Tcl is extensible, so additional functions (e.g., those provided by the enclosing application) can easily be added to those already provided by the Tcl interpreter. In addition, Tk is a graphical user interface (GUI) toolkit that is implemented as a Tcl extension.

Together, Tcl and Tk (pronounced “tickle” or Tcl/Tk) were widely adopted because they solved two problems [Ousterhout09]. First, the combination of Tcl and Tk made it much easier to create GUIs than previous frameworks such as Motif (and Tcl/Tk was eventually ported from Unix to Mac and Windows, making Tcl/Tk applications highly portable). Second, Tcl made it easy to include powerful command languages in a variety of applications ranging from design tools to embedded systems by just including the Tcl/Tk library. Furthermore, Tcl/Tk is free. Many Electronic Design Automation (EDA) tools use it for scripting.

System-on-chip (SoC) designs heavily use Tcl scripts to run synthesis and place & route. So, it is advantageous to learn these scripts for SoC runs and maximizing EDA optimization. Many books on Tcl/Tk exist, including a nice one from John Ousterhout [Ousterhout09], but we have yet to find a document that summarizes the main elements for EDA. This chapter is meant to address this by providing an overview, including some useful advanced features in Tcl/Tk.

D.1 TCL USAGE

Tcl has a stand-alone interface, but EDA tools usually embed Tcl within their toolsets. Although Tcl has an interface, it can also be used as a sequence of commands through a Tcl script. Tcl commands can be found inside a tool or read from a file to help optimize programs for creating SoC implementations. Most operating systems have Tcl/Tk installed by default and a user can typically run the interpreter by typing:

```
$ tclsh
% puts "I am using tclsh"
I am using tclsh
```

D.1 Tcl Usage

D.2 Advanced Commands

John Ousterhout (1954–) is a professor of computer science at Stanford University. He was formerly a professor of computer science at the University of California, Berkeley, where he created the Tcl scripting language and the Tk platform-independent widget toolkit. In 1994, Ousterhout left Berkeley to join Sun Microsystems, which hired a team to join him in Tcl development. He continues development on Tcl/Tk and other software-related projects.



(Image courtesy of John Ousterhout. Reprinted with permission.)

Tcl commands are written in words separated by a white space. The first word is the *command*, and all others are the *arguments*. The result is a string. Here is an example of running Tcl commands and arguments, which makes $a = 22$ and $b = 33$:

```
% set a 22
% set b 33
```

Comments begin with the `#` character. However, if you want a comment at the end of a Tcl command, you must also use a semicolon to indicate the end of the arguments.

```
% set a 42 ;# a = 42, the meaning of the universe
```

D.1.1 Arguments and Variables in Tcl

One of the challenging elements in Tcl is that the result is returned as a string. If you wish to use that result in some way (e.g., an expression), you must combine it with other commands. That is, different commands assign different meaning to their arguments. Specific “type checking” is done by the commands themselves. Here are some examples showing a variety of commands together.

```
% set a 122
122
% expr 24/3.2
7.5
% string length Hello
5
% expr 10 + 5
15
% expr 10.0 + 5
15.0
% expr 12 / 7
1
% expr 12 / 7 + 3.0
4.0
% expr 12.0 / 7 + 3
4.71428571429
```

The number representation in the argument indicates whether it is an integer or floating-point number. Therefore, care must be made to make sure these numbers are interpreted correctly. For example, one incident by the authors involved a frequency divider (dividing 500 MHz by 3) and using integers caused the result to be 0, which is incorrect. If a user wants to use a floating-point value, one of the numbers should be expressed as a floating-point value, i.e., 500.0/3

Variables are not declared separately, and their names can include letters, digits, or underscores. Variables are referred to by preceding the variable name with a `$`, and substitution may occur anywhere within a word as shown by the following examples:

```
% set b 77
66
% set a b
b
% set a $b
66
% set a $b+$b+$b
77+77+77
% set a $b.3
77.3
% set a $b4
no such variable
```

D.1.2 Nested Commands

One advantage of Tcl is that commands can be nested or embedded within other commands. This can occur anywhere within a word. The square brackets nest the command properly (i.e., []).

```
% set b 8
8
% set a [expr $b + 9]
17
% set a "b-5 is [expr $b-5]"
b-5 is 3
% set my_clk_freq_MHz 500
500
% set my_period [expr 1000.0 / $my_clk_freq]
2.0
```

D.1.3 Summary of the Basic Tcl Command Syntax

In Tcl commands, the first word is the operation, and others are arguments, each separated by white space. [Table D.1](#) gives a summary of special characters used in Tcl commands

Table D.1 Summary of Tcl command syntax.

Special Characters	Operation
\$	Variable substitution
[]	Command substitution
" "	Prevent breaks
{ }	Prevents all substitutions and word breaks
\	Escape special characters

Words will end or break at white spaces and semicolons, but there are some exceptions. Some of the more common ones are shown here:

- Double quotes prevent breaks:

```
% set x 5
% set y 5
% set a "x is $x; y is $y"
x is 5; y is 5
```

- Curly braces prevent breaks and substitutions:

```
% set a {[expr $y*$y]}
[expr $y*$y]
```

- Backslashes escape special characters:

```
% set a You're\ using\
coconuts!
You're using coconuts!
```

- Backslashes can also escape newlines (i.e., line continuation):

```
% report_constraints \
-all_violators
```

D.2 ADDITIONAL COMMANDS

Additional Tcl features include lists, outputting information to the screen, working with files, control flow statements, procedures, saving scripts to a file, and exiting the Tcl shell.

D.2.1 Lists

Tcl includes lists, which are an ordered group of elements where each element can be a string or another list. Lists group items such as a set of cells or report file names. To create a list, enclose the list elements in double quotation marks (or curly braces) as shown here:

```
% set a 5
5
% set D_pins "I1/FF3/D I1/FF4/D I1/FF5/D"
I1/FF3/D I1/FF4/D I1/FF5/D
% set b [list a 1 $a [list $a z]]
a 1 5 {5 z}
```

To access a specific element in a simple or compound list, use the `lindex` command. Each list numbers elements starting at 0. For example, the following commands print out the second element of the `D_pins` list:

```
% lindex $D_pins 1
I1/FF4/D
```

Table D.2 shows commands for working with lists.

Table D.2 List commands.

Command	Task
concat	Concatenate lists and returns a new list
join	Join elements of a list into a string
lappend	Append elements to a list
lindex	Retrieve an element from a list
linsert	Insert elements into a list
list	Return a list from its arguments
lsort	Sort a list
split	Split a string into a list

D.2.2 Outputting Data to the Screen With Tcl

The `echo` and `puts` commands output data to the screen. `echo` prints its argument to the console window. `puts` prints its argument to the standard output, which may or may not be the same as the screen.

```
% echo "Just a flesh wound."  
Just a flesh wound.
```

D.2.3 File, Glob, and Other File-Related Commands

Tcl provides several commands for working with directories, retrieving information about files, and writing to files. To retrieve information about a file, use the `file` command, which has the following syntax: `file option argument ...`. For example, `file exists filename` returns 1 when that file exists; otherwise it returns 0. The `glob` command generates a list of file names that match one or more patterns. For example, you could put all SystemVerilog files into a list called `mySVfiles` as follow:

```
% set mySVfiles [glob hdl/*.sv]
```

You can also use `open`, `close`, or `flush` to set up file access. The `open` command has an `access_mode` argument that specifies whether the file is in read, write, or append mode, or a mixture of these. The default access mode is read only. For example, the following commands open a file in read and write mode (`w+`) and then closes the file.

```
% set f [open fopenr.sv w+]  
% close $f
```

D.2.4 Control Flow

Tcl includes the following control flow statements: `if`, `while`, `for`, `foreach`, `break`, `continue`, and `switch`. Control flow statements are often used to check whether a previous command executed successfully. Any command can be used within a control flow statement, including nested control flow statements.

All control flow statements evaluate a *conditional expression*, which is enclosed in curly braces `{}` and is implicitly evaluated using the `expr` command. The evaluation result is treated as Boolean TRUE or FALSE or as an integer/float-point number depending on the value. For example, the `while` loop below repeatedly executes a set of commands while `p` is less than 5.

```
% set p 0
% while {$p < 5} {
%   puts "$p squared is : [expr $p * $p]"
%   incr p; # could also be set p [expr $p + 1]
% }
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
```

The Tcl `for` loop has slightly different syntax than other programming languages: each part of the `for` loop is separated by curly braces, as shown below:

```
% for {init} {test} {reinit} {
%   body
% }
```

A `for` loop runs `init` as a Tcl script, then evaluates `test` as an expression. If `test` evaluates to TRUE (a nonzero value), `body` runs followed by `reinit`; the `for` loop then repeats, starting by evaluating `test`. The following example uses a `for` loop to perform a similar operation as the `while` loop above:

```
% for {set p 0} {$p < 5} {incr p} {
%   puts "$p squared is: [expr $p * $p]"
% }
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
```

`continue` and `break` statements terminate a control loop. `continue` makes the script skip to the next loop iteration, whereas the `break` statement breaks out of the loop. In the following example, the `continue` statement causes only the squares of the even numbers between 0 and 8 to print:

```
% set p 0
% while {$p < 8} {
%   if {$p % 2} {
%     incr p
%     continue
%   }
%   puts "$p squared is: [expr $p * $p]"
%   incr p
% }
0 squared is: 0
2 squared is: 4
4 squared is: 16
6 squared is: 36
```

The following `break` statement exits the loop when `a` is greater than 4:

```
% set a 1
% while {$a < 42} {
%     puts "value of a: $a"
%     incr a
%     if { $a > 4} {
%         break
%     }
% }
value of a: 1
value of a: 2
value of a: 3
value of a: 4
```

D.2.5 Procedures

Some Tcl scripts reuse existing Tcl code by using procedures. A procedure (`proc`) is a named block of commands. The syntax of the `proc` command is `proc name args body`, where `name` is the procedure's name, `args` are the arguments, and the procedure's commands are in `body`. Procedures don't require arguments, but any argument to a procedure must be a scalar value. So, arrays cannot be arguments to a procedure.

The following procedure called `max` computes the maximum of two numbers. The example invokes `max` twice: first to print the result to the screen (`max 42 0`) and second to save `max`'s return value to the variable `meaning`.

```
% proc max {a b} {
%     if {$a > $b} {
%         return $a
%     }
%     return $b
% }
% max 42 0
42
% set meaning [max 42 0]
```

D.2.6 Running Tcl Scripts

A group of Tcl commands may be saved to a file, called a *script*. It is best practice to make Tcl scripts executable in Unix/Linux by invoking the Tcl/Tk interpreter (`tclsh`) in the first line of the script, as shown below for script `test.tcl`:

```
#!/usr/bin/tclsh
set p 0
while {$p < 5} {
    puts "$p squared is : [expr $p * $p]"
    set p [expr $p + 1]
}
```

The `test.tcl` script could also be run within a command shell using:

```
source test.tcl
```

Some Linux commands also work within Tcl. The `cd` and `pwd` commands are equivalent to the Linux commands with the same name. So, within a Tcl script, `cd` changes the current working directory and `pwd` prints the path of the current working directory. Other system commands can be executed using the `exec` command. For example, to print the date, use:

```
% exec date
Fri 18 Feb 2022 00:01:10 AM CST
```

Tcl can also access or create environment variables. For example, the following command makes environment variable `TECH = tech`.

```
% set tech $::env(TECH)
```

D.2.7 Aliases

The `alias` command creates short forms (aliases) of common commands. Guidelines for creating aliases include:

- ▶ Alias names can include letters, digits, underscores, and punctuation marks, but they cannot begin with a digit.
- ▶ Alias names are case sensitive.
- ▶ Alias names cannot be an existing command.
- ▶ An alias definition takes effect immediately but lasts only until you exit the Tcl session. To save commonly used alias definitions, store them in a setup file as shown in the `synthDC/.synopsys_dc.setup` file used for Wally synthesis.

For example, the following defines `h` as an alias for the history of commands in the Synopsys synthesis tool:

```
% alias h history
% h
1 report_area
2 report_timing
3 h
```

An alias for a multiword command must be formatted as a Tcl list—that is, enclosed in curly braces or quotation marks.

D.2.8 Exiting Tcl

You can exit the tool at any time and return to the OS. To exit the tool, use the `exit` command or type Ctrl-c.

Floating-Point Implementation



This appendix contains floating-point implementation details moved out of [Chapter 16](#) for brevity; it includes further details about the following units:

- F.1 FMA: Fused multiply-add
- F.2 `fddivsqrt`: Floating-point division and square root
- F.3 `fcvt`: Floating-point convert
- F.4 `postprocess`: Floating-point postprocessing

- F.1 **Fused Multiply-Add**
- F.2 **Floating-Point Division and Square Root**
- F.3 **Conversion**
- F.4 **Postprocessing**

F.1 FUSED MULTIPLY-ADD

[Section 16.2.2](#) introduces the fused multiply-add (FMA) module, and [Fig. 16.11](#) shows a block diagram. This section describes each of the blocks.

F.1.1 Multiply Significands

The significands of X and Y are multiplied using an unsigned integer multiplier to produce P_m in $U2.2N_f$ form, as given in [Table F.1](#).

F.1.2 Product and Addend Signs

[Table F.2](#) shows how the signs of the product and addend are calculated. The product's sign P_s is negative if X or Y but not both are negative. For subtraction, the sign of Z is flipped to produce the addend sign A_s . If P_s and A_s have different signs, *InvA* indicates that an effective subtraction is taking place.

F.1.3 Add Exponents

Concurrently with multiplication of the significands, the FMA adds the exponents of X and Y to obtain the product's exponent.

The FMA computes $X \times Y + Z$. The mantissas (significands) are X_m, Y_m, Z_m , the exponents are X_e, Y_e, Z_e , and the signs are X_s, Y_s, Z_s . The intermediate product is $P = X \times Y$. N_f and N_e are the number of fraction and exponent bits, respectively.

Multiplying two $U1.N_f$ numbers produces a $U2.2N_f$ product.

Signal names in this appendix are italicized instead of in code font to be consistent with the theoretical development.

Table F.1 Product significand output.

Signal	Format	Logic	Description
P_m	$U2.2N_f$	$X_m \times Y_m$	Unsigned integer multiply

Table F.2 Sign outputs.

Signal	Format	Logic	Description
P_s	Q1	$X_s \oplus Y_s$	Product is negative when X or Y but not both are negative
A_s	Q1	$Z_s \oplus \text{sub}$	Addend is effectively negative when Z is negative or upon subtraction (but not both)
$InvA$	Q1	$P_s \oplus A_s$	Effective subtraction occurs when the product and addend have opposite signs

Table F.3 The product's and sum's exponents.

Signal	Format	Logic	Description
P_e	$B(N_e + 2)$ two's complement	0 When $X_{\text{zero}} \mid Y_{\text{zero}}$ $X_e + Y_e - \text{bias}$ Otherwise	The product's exponent is 0 if either input is 0. Otherwise, it is the sum of the exponents minus the bias.
S_e	$B(N_e + 2)$ two's complement	Z_e When $A_e - P_e > N_f$ P_e Otherwise	If the product is negligibly small, the sum's exponent is the addend's exponent. Otherwise, it is the product's exponent.

$B(N_e + 2)$ in Table F.3 indicates a biased exponent that has $N_e + 2$ bits.

Table F.3 summarizes the calculations of the product's and sum's exponents. When adding biased numbers, the bias must be subtracted from the sum. If either X or Y is 0, the product and, thus, its exponent are 0. The unpacker already took care of adjusting the exponent for subnormals and converting between precisions. The product's biased exponent is in the range $[-\text{bias} + 2, 3\text{bias}]$, which requires $N_e + 2$ bits to represent it in two's complement form. The addend is typically aligned to the product, so the exponent of the sum S_e is simply the product's exponent P_e . However, if the product is zero or negligibly small such that it will be discarded, S_e is the addend's exponent Z_e .

F.1.4 Alignment Shift

Also concurrent with multiplication, Z_m is shifted right by $A_{\text{cnt}} = (P_e - Z_e)$ to align with P_m . Recall that P_m is in $U2.2N_f$ and Z_m is in $U1.N_f$. Z might be larger than, comparable to, or smaller than P . If Z is much larger, the unrounded sum is Z , and P reduces to a sticky bit for rounding. If Z is much smaller, the sum is P , and Z reduces to a sticky bit. After accounting for rounding, it is sufficient to keep Z_m^{shifted} in $U(N_f + 3).(3N_f + 1)$ form because any further bits can be captured with the sticky bit.

Z may need shifting to the left or right, but it is more difficult to build a bidirectional shifter than a unidirectional shifter. Hence, we preshift Z left by

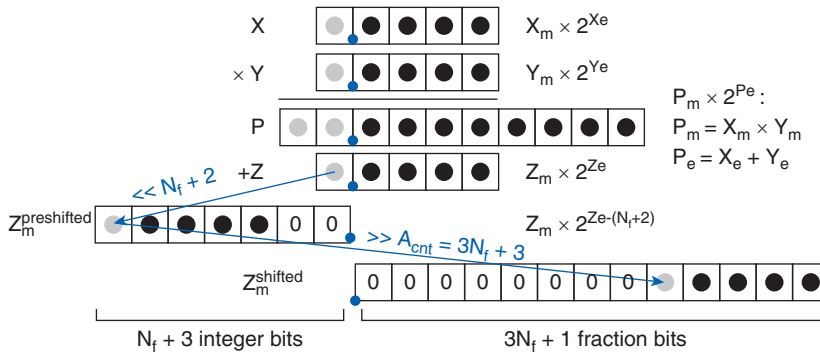


Fig. F.1 Alignment shifts ($N_f = 4$).

the maximum amount that might be needed, then shift right by the desired amount plus the preshift amount. The maximum interesting preshift is $N_f + 2$, which would put the most significant bit of the product in the guard bit position of the result. This preshift is a trivial renaming with no hardware required. Redefine $A_{\text{cnt}} = (P_e - Z_e) + N_f + 2$ to account for this preshift. Then the alignment shifter shifts right by A_{cnt} . If A_{cnt} is negative, P is insignificant and no shift is necessary. If A_{cnt} is too large, Z is insignificant and set to zero. The maximum interesting right shift is $A_{\text{cnt}} = 3N_f + 3$, which would put the most significant bit of Z into the guard position below the product. Any shift larger than this sets Z to zero and sets the sticky bit.

Fig. F.1 shows the alignment shift steps, with $N_f = 4$. The blue dot is the binary point. The gray circle is the integer bit, and the black dots are the fraction bits. The operand mantissas X_m , Y_m , and Z_m are in U1.4, and the product mantissa P_m is in U2.8. Z_m is initially preshifted left by 6, then right shifted by A_{cnt} . A_{cnt} is in the range of $[0, 3N_f + 3] = [0, 15]$ if the shift is relevant; the diagram shows Z_m^{shifted} by this extreme, 15. Z_m^{shifted} may have up to $N_f + 3$ integer bits and $3N_f + 1$ fraction bits.

Table F.4 describes the steps of the alignment shift. First, A_{cnt} is computed as the difference of the exponents, adjusted by the preshift amount. If the shift is negative or X or Y are zero, the unit kills the product (sets it to 0) instead. If the shift is too large, the unit kills Z (sets it to 0) instead. Otherwise, the unit preshifts Z left by a constant amount and then shifts Z right by A_{cnt} . The bottom N_f bits only affect the sticky bit and are dropped before forming A_m .

F.1.5 Add Aligned Significands

The adder computes $S = P + A$, where S , P , and A are all in sign-magnitude form. This is trickier than ordinary two's complement addition. P and A are represented by their magnitudes P_m and A_m and their sign bits P_s and A_s . If the signs of the inputs are the same, the magnitudes are added,

After this section was written, TestFloat revealed an FMA bug (cvv issue #578) related to subtraction of a small value. Z_m^{shifted} had to be widened to $N_f + 4$ integer bits and $3N_f + 2$ fraction bits to obtain a correct guard bit. The extra integer and fraction bit propagate through the rest of the FMA and postprocessing. The root cause has not yet been fully understood.

Table F.4 Alignment shift.

Signal	Format	Logic	Description
Internal Signals			
A_{cnt}	$Q(N_e + 2).0$	$(X_e + Y_e - \textit{bias}) - Z_e + N_f + 2$	Alignment shift amount. Note that the portion for $P_e = X_e + Y_e - \textit{bias}$ is replicated to reduce the critical path.
$\textit{KillZ}$	Boolean	$A_{\text{cnt}} > (3N_f + 3)$	$Z = 0$ because it is too small to affect anything but the sticky bit.
$Z_m^{\text{preshifted}}$	$U(N_f + 3).0$	$Z_m \ll (N_f + 2)$	Z is placed in the upper bits (no hardware required).
Z_m^{shifted}	$U(N_f + 3).(3N_f + 1)$	Z_m	$\textit{KillProd}$ Right-shift by A_{cnt} if not killing P or Z .
		0	$\textit{KillZ}$
		$Z_m^{\text{preshifted}} \gg A_{\text{cnt}}$	Otherwise
Outputs			
$\textit{KillProd}$	Boolean	$(A_{\text{cnt}} < 0) \mid X_{\text{zero}} \mid Y_{\text{zero}}$	$P = 0$ because it is too small to affect anything but the sticky bit.
A_{sticky}	Boolean	$\sim (X_{\text{zero}} \mid Y_{\text{zero}})$	$\textit{KillProd}$ Sticky bit set if killing a nonzero product or addend. Otherwise, the sticky bit is the OR of the bits to the right of the fraction bits.
		$\sim Z_{\text{zero}}$	$\textit{KillZ}$
		$\mid (Z_m^{\text{shifted}}[N_f - 1 : 0])$	Otherwise
A_m	$U(N_f + 3).(2N_f + 1)$	$Z_m^{\text{shifted}} \gg N_f$	Aligned Z (drop sticky bits).

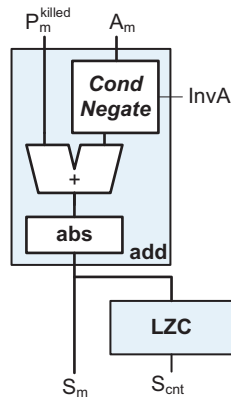


Fig. F.2 Simple significand addition path.

and the sign of the result is $S_s = P_s$. If the signs of the inputs differ, the magnitudes are subtracted. The unit then takes the absolute value of the result to obtain the sum's significand S_m . S is not normalized, so the FMA must compute the normalization shift amount M_{cnt} , which depends on S_{cnt} , the number of leading zeros in S_m . Fig. F.2 shows a simple implementation involving conditional negation, significand addition, absolute value, and a priority encoder (leading zero counter [LZC]) to count leading zeros. Negation and absolute value each involve inverting and adding 1, so the path has three adders and a priority encoder.

Fig. 16.11 shows a faster design used in Wally and most FMAs. Negation is replaced by an XOR to conditionally invert A , and the extra 1 is provided as a carry input to the significand adder. Instead of taking the absolute value, the circuit computes $P \pm A$ and $A - P$ in parallel and selects the positive one. Subtraction is performed using two's complement arithmetic. The most significant bit indicates a negative result. In the case of effective addition ($\textit{InvA} = 0$), the answer is always positive and comes from $\textit{PreSum}$. In the case of effective subtraction ($\textit{InvA} = 1$),

Table F.5 Significand addition.

Signal	Format	Logic	Comments	
Internal Signals				
A_m^{inv}	$Q(N_f + 3).(2N_f + 1)$	$\frac{A_m}{A_m}$	$\overline{InvA}$ $InvA$	Invert A_m during effective subtraction
p_m^{killed}	$U2.2N_f$	$P_m \overline{KillProd}$		Zero out P_m when $KillProd$ is asserted Append 2 lsb 0s to align with A
$PreSum$	$Q(N_f + 3).(2N_f + 1)$	$p_m^{killed} + A_m^{inv} + \left(\overline{A_{sticky}} \mid KillProd\right) InvA$		$P \pm A$, with correction for negative sticky bit of A
$NegPreSum$	$Q(N_f + 3).(2N_f + 1)$	$A_m + \overline{p_m^{killed}} + \left(\overline{A_{sticky}} \mid \overline{KillProd}\right)$		$A - P$, with correction for negative sticky bit of P
$NegSum$	Boolean	Sign of $PreSum$ (extra bit)		$P - A$ is negative
Outputs				
S_s	Q1	$P_s \oplus NegSum$		Sign of sum
S_e	$B(N_e + 2)$	P_e Z_e	$\overline{KillProd}$ $KillProd$	Exponent of sum before normalization
S_m	$Q(N_f + 3).(2N_f + 1)$	$PreSum$ $NegPreSum$	$\overline{NegSum}$ $NegSum$	Magnitude of sum

$PreSum = P - A$ and $NegPreSum = A - P$, and the answer is selected as the positive of these two results.

If an effective subtraction occurs and the sticky bit of the subtrahend is 1, the result must be decremented by 1. A_{sticky} comes from P when $KillProd = 1$, or from Z when $KillProd = 0$. This *negative sticky bit* can be handled now in the two's complement adders by setting the carry in to 0 for $P - A - 1$ and $A - P - 1$. A_{sticky} is effectively negative for $P - A$ when $KillProd = 0$, or for $A - P$ when $KillProd = 1$. Table F.5 describes the internal signals and outputs for significand addition.

Example F.1 FMA

Compute $(1.0)(1.0) - 2^{-13}$ using half-precision floating-point numbers ($N_f = 10$).

Solution: This is an effective subtraction with a negligible subtrahend, so $InvA = 1$ and $KillProd = 0$. The product is 1.0×2^0 , so $P_m = p_m^{killed} = 01.0000\ 0000\ 0000\ 0000$. The subtrahend of 1.0×2^{-13} is too small to directly affect the fraction bits of the answer, so it becomes a sticky bit of 1: $A_m = 0.0$ and $A_{sticky} = 1$. $A_m^{inv} = 1\ 1111\ 1111\ 1111.1111\ 1111\ 1111\ 1111\ 1111\ 1$. $S_m = PreSum = 0\ 0000\ 0000\ 0000.1111\ 1111\ 1111\ 1111\ 1111\ 1$, which is 1 ulp less than 1.0, with the round and sticky bits set. The result will round down to $TRUNC = 1.1111\ 1111\ 11 \times 2^{-1}$ or up to $RND = 1.0 \times 2^0$.

A leading zero anticipator (LZA) estimates the number of leading zeros concurrently with the addition, as described next in [Section F.1.6](#).

F.1.6 LZA: Leading Zero Anticipator

The postprocessing normalization logic needs to find the number of leading zeros in S_m to compute the normalization shift amount S_{cnt} . The most direct approach is to compute S_m and then use a priority encoder to count the leading zeros (LZC: leading zero counter). This is on the critical path, so a faster FMA uses an LZA to estimate the number of leading zeros in parallel with the addition. The estimate can sometimes be too high by 1 bit, so the postprocessor may have to use a subsequent 1-bit correction shift to compensate, as described in [Section F.4.1.1](#).

A further complication is that during effective subtractions, the result might be negative. When the result is negative, we must count the leading ones instead of the leading zeros. The LZA handles both cases but is nevertheless usually called an LZA. [Schmookler01] surveys LZA designs.

Consider N -bit addition $S = A + B + C_{in}$. Let P , G , and K indicate whether a column i ($N - 1 \geq i \geq 0$) of the addition will propagate, generate, or kill a carry [Weste12]. Let column $i = -1$ be the carry in column.

$$P_i = A_i \oplus B_i \quad G_i = A_i B_i \quad K_i = \overline{A_i} \overline{B_i} \quad (\text{F.1})$$

$$P_{-1} = 0 \quad G_{-1} = C_{in} \quad K_{-1} = \overline{C_{in}} \quad (\text{F.2})$$

First, consider addition with a positive sum; that is, the sum has $k \geq 1$ leading zeros. S is of the form $0^k 1 x^*$, which is a *regular expression* indicating k leading zeros followed by a 1 and then any number of arbitrary bits. Any pattern leading with the k -bit string K^k or $P^m G K^{k-m-1}$ will have k or $k - 1$ leading zeros. First, suppose the carry in to the k most significant bits is 0. If the k most significant columns all kill (K^k), then those columns' A and B values are all 0, so the k leading bits of the sum are 0. k leading 0s may also occur as follows. If the most significant m columns propagate followed by one column that generates and $k - m - 1$ columns that kill ($P^m G K^{k-m-1}$), the most significant m columns each have a single 1 in A_i or B_i , and the next column has two ones. The sum in all these $(m + 1)$ columns is zero because $1 + 1 = 0$ with a carry-out of 1 (in the generate column), and that carry ripples through the most significant m columns to produce sums of 0 and a carry-out of the msb of the adder that is discarded. Next, suppose the carry in to the k most significant bits is 1, but either of the regular expressions above still applies. Then there will be only $k - 1$ leading zeros because the least significant column will produce a sum of 1.

An LZA produces another N -bit signal F using Equation F.3 that detects the end of a regular expression of the form K^k or $P^m GK^{k-m-1}$. The LZA looks for two consecutive columns that terminate either of these regular expressions. Specifically:

- ▶ K_i followed by something other than K_{i-1} terminates both K^k and $P^m GK^{k-m-1}$.
- ▶ G_i followed by something other than K_{i-1} terminates $P^m GK^{k-m-1}$.
- ▶ P_i followed by K_{i-1} also terminates $P^m GK^{k-m-1}$.

Equation F.3 further shows how these three terms reduce to an exclusive NOR. In other words, F indicates the end of the regular expression, where P is followed by a K , or where a G or K is followed by something other than a K . The most significant nonzero column of F indicates the left-most termination. In summary, the LZA estimates the number of leading zeros using simple bitwise logic to compute F and then counting the number of leading zeros in F with a priority encoder (LZC).

$$\begin{aligned} F_i &= K_i \overline{K_{i-1}} + G_i \overline{K_{i-1}} + P_i K_{i-1} = (K_i + G_i) \overline{K_{i-1}} + P_i K_{i-1} \\ &= \overline{P_i \overline{K_{i-1}}} + P_i K_{i-1} = \overline{P_i \oplus K_{i-1}} \end{aligned} \quad (\text{F.3})$$

Example F.2 LZA: PREDICT NUMBER OF LEADING ZEROS

Use an LZA to predict the number of leading 0s in $S = 000101 + 111101 + 0$ without adding.

Solution: To check the answer, $S = 000010$, which has 4 leading zeros. Fig. F.3 shows the LZA calculation for this $N = 6$ -bit addition using Equation F.3. The Propagate (P) signals are true when either A or B but not both are true. Generate (G) is true when both A and B are true, and Kill (K) is true when neither are true. Column -1 , shown in blue, reflects the carry in of 0. In this example, the $k = 5$ most significant bits of F are 0, so the LZA estimates that S has 5 or 4 leading zeros.

i	5	4	3	2	1	0	-1
A	0	0	0	1	0	1	
B	1	1	1	1	0	1	
P	1	1	1	0	0	0	0
G	0	0	0	1	0	1	0
K	0	0	0	0	1	0	1
F	0	0	0	0	0	1	

Fig. F.3 LZA example calculation.

Similarly, addition that produces a negative sum has $k \geq 1$ leading ones, so S is of the form $1^k 0x^*$. Any pattern leading with the k -bit string G^k or $P^m KG^{k-m-1}$ will have k or $k - 1$ leading ones. The left-most column terminating this pattern is detected as the most significant nonzero column of F using Equation F.4.

$$F_i = \overline{P_i \oplus G_{i-1}} \quad (\text{F.4})$$

One could perform two separate LZA (F) calculations for positive and negative sums, then pick the estimate based on the sign S . However, this requires two priority encoders. A more compact LZA generates a combined F that applies to both leading zeros and leading ones using Equation F.5 and counts the number of leading zeros in this combined F

In Equation F.5, the most significant column ($i = N$) detects an addition by a propagate in the most significant bit (bit $N - 1$). This propagate means the most significant bit will be 1 or the calculation produces a carry out, an extra leading 1. In either case, there are no leading zeros, indicated by $F_N = 1$.

i	6	5	4	3	2	1	0	-1
A	0	0	0	1	0	0		
B	1	1	1	0	0	0		
P	1	1	1	1	1	0	0	
G	0	0	0	0	0	0	1	
K	0	0	0	0	1	1	0	
F	0	0	0	0	0	1	1	

Fig. F.4 LZA subtraction example calculation.

[Schmookler01]. The unified LZA requires examining three consecutive columns. This approach also requires defining $P_N = 1$ for effective subtraction and 0 for addition and adds a special case for F_N .

$$F_i = \begin{cases} \overline{P_N} P_{N-1} & i = N \\ P_{i+1} (G_i \overline{K_{i-1}} + K_i \overline{G_{i-1}}) + \overline{P_{i+1}} (G_i \overline{G_{i-1}} + K_i \overline{K_{i-1}}) & i < N \end{cases} \quad (\text{F.5})$$

The second row detects when a sequence of three columns terminates any of the regular expressions K^k , $P^m G K^{k-m-1}$, G^k , $P^m K G^{k-m-1}$ given earlier. Specifically, $P_{i+1} G_i \overline{K_{i-1}}$ detects the end of a string of the form $P^m G K$. $P_{i+1} K_i \overline{G_{i-1}}$ detects the end of a string of form $P^m K G$. $\overline{P_{i+1}} G_i \overline{G_{i-1}}$ detects the end of a string of the form G^k or $P^m K G^{k-m-1}$. $\overline{P_{i+1}} K_i \overline{K_{i-1}}$ detects the end of a string of the form K^k or $P^m G K^{k-m-1}$.

Example F.3 LZA TO PREDICT LEADING ONES

Use an LZA to predict the number of leading 1s in $000100 - 000111$ without subtracting.

Solution: For reference, $S = 111101$, which has 4 leading ones. A subtractor inverts the second input and adds a carry of 1, so the LZA inputs are $000100 + 111000 + 1$. Fig. F.4 shows the LZA calculation for this $N = 6$ -bit addition. The blue bits indicate $G_{-1} = 1$ to account for the carry in and $P_6 = 1$ for subtraction. Again, the 5 most significant bits of F are 0, so the LZA estimates that S has 5 or 4 leading 1s or 0s.

Fig. F.5 shows a schematic of the LZA. The LZA runs concurrently with a carry-propagate adder (CPA), so the PGK logic can be shared for both. A priority encoder counts the number of leading zeros in F .

The S_{cnt} output of the LZA is the estimated number of leading zeros, not the number of bits to shift from Fig. 16.9. The normalization shift calculation in Section F.4.1 shifts the result as needed.

F.1.7 Multipath FMA Optimization

The FMA never encounters the worst case of a large alignment shift and a large normalization shift on the same operation. In the *product-anchored* case, the product is larger than the addend, so the addend is right shifted, and only a small normalization shift is needed. In the *addend-anchored* case, the addend is larger than the product, so the product is right shifted, and only a small normalization shift is needed. In the *massive cancellation* case, the addend is subtracted from a product of comparable magnitude. Little alignment shifting is needed, but the result might have many leading zeros and require a large normalization shift. A *multipath* design uses separate logic for the product-anchored, addend-anchored, and massive cancellation cases to gain speed at the expense of more hardware [Seidel03, Quinnell07]. Wally does not implement a multipath FMA.

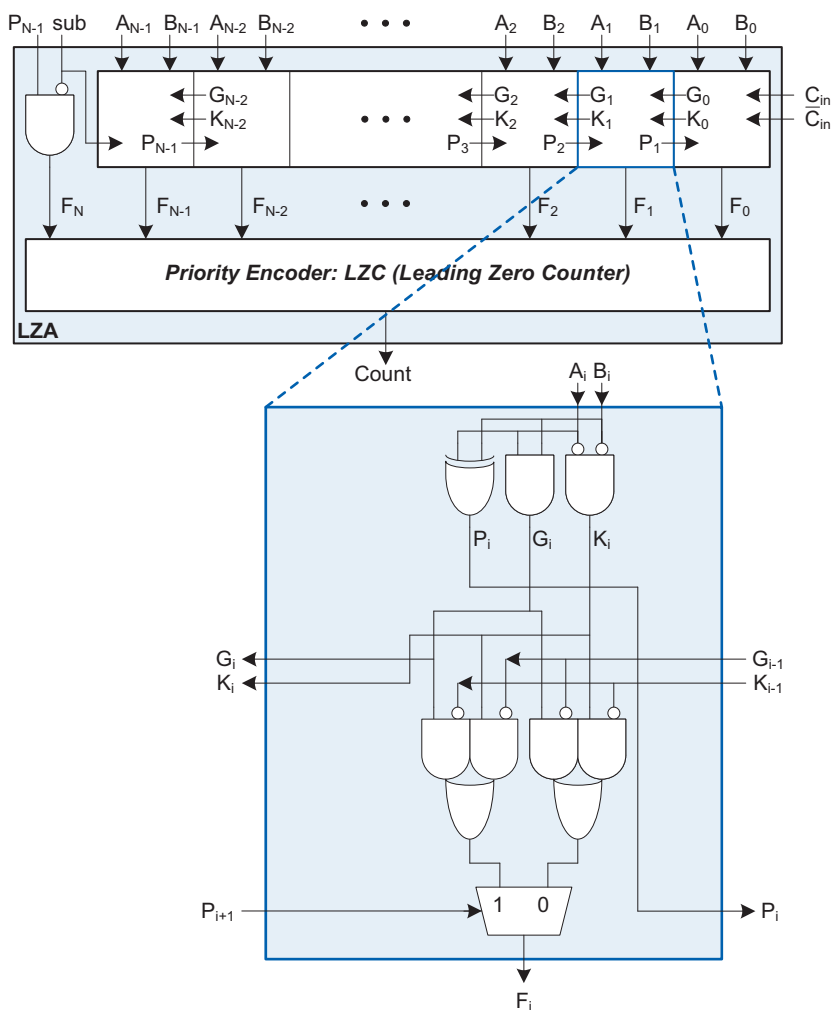


Fig. F.5 LZA schematic.

F.2 FLOATING-POINT DIVISION AND SQUARE ROOT

Section 16.2.3 introduced floating-point division and square root. This section shows the implementation details, including many optimizations.

Recall from Section 15.1.4 that division of N -bit numbers takes a *dividend* X and a *divisor* D . It computes a *quotient* Q and *remainder* REM satisfying $X = Q \times D + REM$. Or stated another way: $X/D = Q + REM/D$. If X and D are signed, then REM has the same sign as X .

Fig. 15.14 gives an algorithm for unsigned integer division, which is a binary form of long division that you learned in elementary school. It initializes the fraction part of a *partial remainder* W to X . The *radix*-2

algorithm produces 1 bit of quotient on each step. At each step, it compares the divisor to W . If $D \leq W$, then the quotient bit is 1, and D is subtracted from W . In any case, W is left shifted by one bit, and the process repeats until all of the quotient bits are produced. The value left in W is the remainder REM .

This algorithm involves N steps to produce an N -bit quotient. Each step involves an N -bit comparison of W and D performed by computing $W - D$ and looking at the sign. Thus, the comparison also can act as the subtraction. Recall that subtraction can be performed by taking the two's complement and adding with a CPA. Ignoring the inverter delay in taking a two's complement, define the time to compare/subtract to be t_{CPA-N} . The result of the comparison must fan out to all N bits, requiring $t_{buf} \approx \log_4 N$ FO4 inverter delays. Selecting W or $W - D$ takes t_{mux} . Left shifting is free (just a wire). A register clock-to-Q and setup time contributes t_{reg} . If each step takes one clock cycle, then division takes N cycles, with a cycle time of at least:

$$T_c \geq t_{CPA-N} + t_{buf} + t_{mux} + t_{reg} \quad (F.6)$$

This time is dominated by the CPA delay. The number of cycles could be reduced to $n = N/k$ by performing k steps in each clock cycle. However, this increases the required cycle time to:

$$T_c \geq k(t_{CPA-N} + t_{buf} + t_{mux}) + t_{reg} \quad (F.7)$$

Typical processors with a 25–50 FO4 cycle time often choose $k = 2$ –4 to speed up integer division without it becoming the critical path.

To eliminate the slow CPA, SRT dividers keep the partial remainder in carry-save redundant form. Specifically, an N -bit partial remainder W is represented using two redundant N -bit numbers WS and WC (sum and carry) such that $W = WS + WC$.

Radix-2 division produces one quotient bit per step and thus requires approximately N steps (perhaps plus a few more for rounding). To reduce the latency, some dividers use a *higher radix* to select a quotient digit representing more than one bit of information. Radix- $R = 2^r$ division produces r quotient bits per step and requires $n = N/r$ steps.

Experience has shown that the most practical redundant quotient digit sets are radix-2 $\{+1, 0, -1\}$ and *minimally redundant* radix-4 $\{+2, +1, 0, -1, -2\}$. Higher-radix digits are expensive to select and to generate divisor multiples, so a design producing more than 2 bits of quotient per cycle uses multiple stages. A faster division algorithm is recurrence division, which is discussed next.

F.2.1 Recurrence Division Algorithm

Section 16.2.3 introduced radix- R recurrence division, which computes partial remainders in carry-save redundant form, guesses quotient digits

Radix- R recurrence division is also referred to simply as *recurrence division*.

(quotient selection: QSLC) using the most significant bits of these partial remainders, corrects the guesses with subsequent digits if they're wrong, and finally converts the quotient digits to their final binary form using on-the-fly conversion (OTFC). Recurrence division is most straightforward with unsigned numbers, so this section assumes inputs are unsigned. Recurrence division expects the inputs to fit in a bounded range; we assume they have been normalized to be N -bit fixed-point fractions. This is straightforward for ordinary floating-point numbers (excluding subnormals, 0, NaN, and infinity), so we start with that case. [Section F.2.7](#) addresses subnormal floating-point numbers, and [Section F.2.9](#) discusses integer division.

Consider dividing normalized unsigned inputs: X/D . X and D are ordinary N -bit floating-point significands in the range $[1, 2)$ and format U1.N−1. The algorithm produces $\{Q, REM\}$, satisfying $X = DQ + REM$. Q is in the range $(1/2, 2)$ in Q1.b format, with $b = N_f + 2$ to provide the fraction bits plus guard and round bits. Q is expressed in terms of radix- R quotient digits q_i , with one integer digit and $n = \lceil b/r \rceil$ fraction digits. Each digit is in the range $[-a, a]$, where a is 1 for radix-2 and 2 or 3 for radix-4. The first $j + 1$ digits of the quotient are:

$$Q_j = \sum_{i=0}^j q_i R^{-i} \quad (\text{F.8})$$

The final quotient is:

$$Q = Q_n = \sum_{i=0}^n q_i R^{-i} \quad (\text{F.9})$$

In radix- R recurrence division, the *partial remainder* (also called the *residual*) W_j on step j is given in [Equation F.10](#). As in long division by hand, the partial remainder describes the shifted remainder when representing the quotient up to $j + 1$ digits.

$$W_j = R^{j+1} [X - DQ_j] \quad (\text{F.10})$$

At the beginning of recurrence division, at step $j = -1$, the partial remainder W_{-1} is initialized to $R^0[X - D \times 0] = X$. At each step j , a multiple q_j of the divisor is subtracted from the partial remainder. Then the partial remainder is shifted left by one digit, and the process repeats. Using [Equations F.8](#) and [F.10](#), this is expressed mathematically with the *recurrence relationship*, where the current residual depends on the value of the prior residual, in [Equation F.11](#).

$$\begin{aligned}
W_j &= R^{i+1} [X - DQ_j] \\
&= R^{i+1} \left[X - D \sum_{i=0}^j q_i R^{-i} \right] \\
&= R \left[(R^j) \left(X - D \sum_{i=0}^{j-1} q_i R^{-i} \right) - Dq_j \right]
\end{aligned}$$

Substituting W_{j-1} gives

$$W_j = R [W_{j-1} - Dq_j] \quad (\text{F.11})$$

F.2.1.1 Examples

Compare the integer division algorithm from Fig. 15.14 to Fig. F.6. The integer divider uses an unnormalized divisor D in the range $[0, 2^N - 1]$, initializes the partial remainder to 0, and shifts in one bit of X at a time. The recurrence divider uses a normalized divisor D in the range $[1, 2)$ and initializes the partial remainder to X .

The integer divider is simpler because it doesn't require normalization. The recurrence divider is faster because QSLC can be performed approximately using only the most significant bits in carry-save redundant form, allowing for fast carry-save addition and QSLC.

Section F.2.9 will also show that integer recurrence division may use fewer steps than floating-point division (depending on the number of leading 0s in the inputs) because it produces an integer quotient and an integer remainder.

Example F.4 RADIX-4 RECURRENCE DIVISION DIGITS

Consider radix-4 recurrence division of half-precision floating-point numbers. How many fraction digits are required? What are the digits in the quotient $= 287/7 = 1.00011111 \times 2^8 / 1.11 \times 2^2 = 0.101001000000 \times 2^6 = 41$?

Solution: Half-precision numbers have $N_f = 10$ fraction bits, so $b = 12$ fractional result bits (10 fraction bits + 2 rounding bits: G and R) are required. Radix- $R = 4$ recurrence division produces $r = 2$ bits per digit, so $n = 6$ fraction digits are needed (see Section 15.1.3.3). By inspection of the quotient significand from left to right starting with the integer bit and then considering each pair of fraction bits, the quotient digits are $q_0 = 0, q_1 = 2, q_2 = 2, q_3 = 1, q_4 = 0, q_5 = 0$, and $q_6 = 0$. In other words, $Q = 0 \times 4^0 + 2 \times 4^{-1} + 2 \times 4^{-2} + 1 \times 4^{-3} = 0.101001$.

Example F.5 RADIX-4 RECURRENCE DIVISION

Given $X = 1.00011111$, $D = 1.11$, and quotient digits from Example F.4, tabulate the quotient digit q_j , approximate quotient Q_j , and partial remainder W_j at each step.

Solution: Table F.6 lists the variables at each step using Equation F.10. Note that the quotient is exact from step 3 onward, and hence the partial remainder is 0.

F.2.1.2 Redundant Partial Remainder and Quotient Digit Set

The recurrence steps of Fig. 15.14 are now recast in Fig. F.6 with a redundant partial remainder and quotient digit set. This is now considered *non-restoring* recurrence division because the algorithm always subtracts a multiple of D ; however, the multiple may be 0. The partial remainder is initialized to X . In step j , the divider guesses a quotient digit q_j based on

Table F.6 Radix-4 recurrence division example.

Step j	Quotient Digit q_j	Approximate Quotient Q_j	Partial Remainder W_j
-1			$4^0[1.00011111 - 1.11 \times 0] = 1.00011111$
0	0	$0 \times 4^0 = 0$	$4^1[1.00011111 - 1.11 \times 0] = 100.011111$
1	2	$0 \times 4^0 + 2 \times 4^{-1} = 0.10$	$4^2[1.00011111 - 1.11 \times 0.10] = 11.1111$
2	2	$0 \times 4^0 + 2 \times 4^{-1} + 2 \times 4^{-2} = 0.1010$	$4^3[1.00011111 - 1.11 \times 0.1010] = 1.11$
3	1	$0 \times 4^0 + 2 \times 4^{-1} + 2 \times 4^{-2} + 1 \times 4^{-3} = 0.101001$	$4^4[1.00011111 - 1.11 \times 0.101001] = 0$
4	0	0.101001	$4^5[1.00011111 - 1.11 \times 0.101001] = 0$
5	0	0.101001	$4^6[1.00011111 - 1.11 \times 0.101001] = 0$
6	0	0.101001	$4^7[1.00011111 - 1.11 \times 0.101001] = 0$

$$WC_{-1} = 0, WS_{-1} = X$$

for $j = 0$ to n

$$q_j = \text{QSLC}(WC_{\text{msbs } j-1}, WS_{\text{msbs } j-1}, D_{\text{msbs}})$$

$$\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} - q_j D\} \ll r$$

Fig. F.6 Recurrence division algorithm.

the most significant bits of the partial remainder (WC_{msbs} , WS_{msbs}) and the divisor (D_{msbs}) using a quotient selection (QSLC) block. It subtracts a divisor multiple $q_j D$ from the redundant partial remainder to produce a new partial remainder. Finally, it shifts the partial remainder left by r to multiply by $R = 2^r$ to bring the next part of the partial remainder into play. Figs. F.9 and F.13 show worked examples for radix 2 and 4 using QSLC that will be developed in subsequent sections. Fig. F.7 shows a simple hardware implementation of the loop body. Table F.7 summarizes the formats and bit widths of the signals.

F.2.1.3 Containment of Partial Remainder

At each step, the divisor is multiplied by the quotient digit and subtracted to reduce the absolute value of the partial remainder, and then the partial remainder is multiplied by R . Each quotient digit must be chosen to keep the partial remainder bounded: $|W_j| \leq M$. To find the upper bound M , suppose $W_{j-1} \leq M$. If W_{j-1} has the largest value (M); then the divider must choose the largest quotient digit $q_j = a$; hence $W_j = (M - aD)R$ according

Our notation differs from [Ercegovac03] in that, for simplicity of notation, we initialize W to X rather than to a scaled version of X . Consequently, our bounds on W in (Equation F.12) differ from Ercegovac's bounds, and the digit selection (e.g., Fig. F.8) is similarly unscaled. Otherwise, the divider works in the same way.

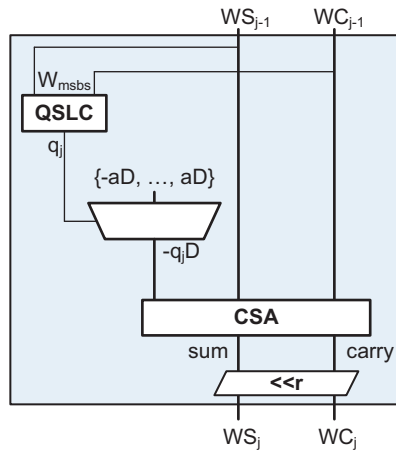


Fig. F.7 Recurrence division hardware.

Table F.7 Ordinary floating-point divider operand sizes.

Term	Format	Description
Design Parameter		
R		Radix = 2 or 4
Dependent Parameters		
r		$\log_2 R = 1$ or 2
a		Quotient digit range $\{-a, -a+1, \dots, 0, \dots, a-1, a\}$ $a = 1$ for radix-2, $a = 2$ for minimally redundant radix-4
ρ		Redundancy factor $a/(R-1)$ $\rho = 1$ for radix-2, $\rho = 2/3$ for minimally redundant radix-4
N		Input bit count = $N_f + 1$ for floating-point division (1 integer + N_f fractional input bits)
b		Fraction result bits (at least $N_f + 2$) Includes guard (G) and round (R) bits
n		Number of fraction quotient digits $\left\lceil \frac{b}{r} \right\rceil$ (see Section F.2.4)
Inputs		
X	U1.N-1	Dividend $\in [1, 2)$
D	U1.N-1	Divisor $\in [1, 2)$
Internal Signals		
WC_j, WS_j	Q4.b	Partial remainder in carry-save redundant form WS is initialized to X and WC to 0 $W = WC + WS$ always remains in range $(-\rho RD, \rho RD)$
q_j		Quotient digit in range $\{-a, a\}$
Output		
Q	U1.b	Quotient significand (before normalization and rounding) In range $(1/2, 2)$ for floating-point division

to Equation F.11. Equating these two bounds ($W_j \leq M$ and $W_j = (M - aD)R$) gives $M = (M - aD)R$, or $M = (RD a)/(R - 1) = \rho RD$. Hence, $M = \rho RD$. By the same reasoning, the lower bound is $-M = -\rho RD$. This containment is expressed mathematically as:

$$|W_j| = \left| R(W_{j-1} - q_j D) \right| \leq \rho RD \quad (\text{F.12})$$

Solve this inequality for bounds on the range of W and D for a given choice of quotient digit q_j :

$$\begin{aligned} \left| R(W_{j-1} - q_j D) \right| &\leq \rho RD \\ \left| (W_{j-1} - q_j D) \right| &\leq \rho D \\ -\rho D &\leq W_{j-1} - q_j D \leq \rho D \\ (q_j - \rho)D &\leq W_{j-1} \leq (q_j + \rho)D \end{aligned} \quad (\text{F.13})$$

Equation F.13 can be rewritten in terms of lower and upper bounds L_k and U_k for selecting $q_j = k$:

$$\begin{aligned} L_k &\leq W_{j-1} \leq U_k \\ L_k &= (k - \rho)D \\ U_k &= (k + \rho)D \end{aligned} \quad (\text{F.14})$$

The QSLC function depends on the radix and digit set. It will guess a quotient digit that is either correct or almost correct such that the divider unit can fix the answer later using a quotient digit in a less significant position.

F.2.1.4 Quotient Representation

The positive and negative quotient digits can be saved in *signed-digit form* in two shift registers, Q^+ and Q^- , and finally converted to unsigned $Q = Q^+ - Q^-$ with a carry propagate addition at the end of the recurrence division calculation. Better yet, they can be converted digit-by-digit using an *on-the-fly conversion* (OTFC) described in Section F.2.5, so only shift registers but no adder is required.

F.2.2 Radix-2 Recurrence Division

In radix-2 recurrence division, the quotient digit set is $\{-1, 0, +1\}$ with $a = 1$ and $\rho = 1$. According to the containment conditions of Equation F.14, the bounds on W and D for each choice of radix-2 quotient digit are:

$$q_j = \begin{cases} +1 & L_1 \leq W_{j-1} \leq U_1 & 0 \leq W_{j-1} \leq 2D \\ 0 & L_0 \leq W_{j-1} \leq U_0 & -D \leq W_{j-1} \leq D \\ -1 & L_{-1} \leq W_{j-1} \leq U_{-1} & -2D \leq W_{j-1} \leq 0 \end{cases} \quad (\text{F.15})$$

Because $|W| < \rho RD = 2D$, W_{j-1} is never greater than U_1 or less than L_{-1} .

Recall from Fig. F.6 that at each step j , the recurrence divider chooses a quotient based on the most significant bits of the partial remainder and divisor:

$$q_j = \text{QSLC}(WC_{\text{msbs } j-1}, WS_{\text{msbs } j-1}, D_{\text{msbs}}) \quad (\text{F.16})$$

We aim to design the simplest QSLC logic sufficient to ensure containment. We will soon show that for radix-2, it is sufficient to use only the four msbs of W in Q4.0 format and to disregard D .

For radix-2, $\rho = 1$, $R = 2$, and $1 \leq D < 2$. Hence $|W| \leq \rho RD < 4$. We estimate W based on the most significant bits WC_{msbs} , WS_{msbs} . This truncation drops the fraction bits, so the error between the true and truncated partial remainders is in the range $[0, 1)$. Hence, the truncated sum $W_{\text{msbs}} = WC_{\text{msbs } j-1} + WS_{\text{msbs } j-1}$ is in the range $(-6, 4)$ after accounting for truncation error. W_{msbs} is an integer in Q4.0 format, so it must be in the range $[-5, 3]$ because these are the only integers in the open set $(-6, 4)$.

[Atkins68] developed the graphical method called P-D diagrams to show how to choose a quotient digit based on the Partial remainder and Divisor to ensure containment. Fig. F.8 shows a P-D diagram for radix-2. The divisor D is on the x-axis, and the truncated partial remainder W is on the y-axis. The lines from Equation F.15 are drawn to separate the plane into three overlapping regions in which each quotient digit is a legal

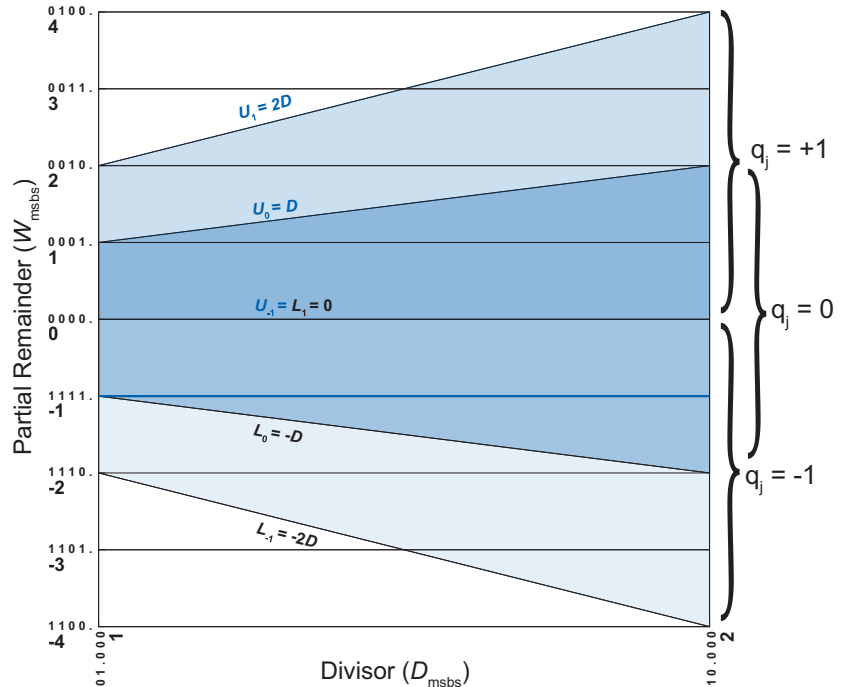


Fig. F.8 Radix-2 recurrence division P-D diagram.

choice. Observe that the region for $q = 0$ overlaps the bottom part of the region for $q = 1$ and the top part of the region for $q = -1$.

Because both WC and WS are truncated, the true value of W is the range $W_{\text{msbs}} \leq W < W_{\text{msbs}} + 2$. Equation F.17 shows a simpler criteria to select a legal quotient digit considering only the most significant bits of W and no bits of D . As the P-D diagram shows, for any true value of W in the range $[W_{\text{msbs}}, W_{\text{msbs}} + 2)$, the equation yields an acceptable choice of quotient digit, in other words, quotient selection (QSLC).

$$q_j = \begin{cases} +1 & W_{\text{msbs}} > -1 \\ 0 & W_{\text{msbs}} = -1 \\ -1 & W_{\text{msbs}} < -1 \end{cases} \quad (\text{F.17})$$

For example, when $W_{\text{msbs}} = -1$, the true value of W is in the range $[-1, 1)$. Fig. F.8 shows that we can select $q = 0$ for any value of W in that range and any value of D . Similarly, when $W_{\text{msbs}} = -2$, the true value of W is in the range $[-2, 0)$. Fig. F.8 shows that we can select $q = -1$ for any value of W in that range and any value of D .

Other QSLC logic could have been used, such as that in Equation F.18. However, this logic would have had to bound the error on W_{msbs} to under 1 instead of 2 to remain within the containment bounds when $W_{\text{msbs}} = 0$. Hence, we would have needed 1 extra bit, keeping W_{msbs} in Q4.1 form instead, which is slower and uses more hardware.

$$q_j = \begin{cases} +1 & W_{\text{msbs}} > 0 \\ 0 & W_{\text{msbs}} = 0 \\ -1 & W_{\text{msbs}} < 0 \end{cases} \quad (\text{F.18})$$

The divisor multiple is $qD \in \{-D, 0, D\}$ and is represented in twos complement form. Recall that $-D = \overline{D} + 1$. A carry-propagate addition to add 1 is expensive, so instead, we simply select $\overline{D}$ and then add the 1 into the vacant spot in the least significant column of WC created when WC is left shifted or initialized to 0.

F.2.2.1 Examples

Example F.6 QSLC

Suppose $WS = 1110.0001000 = -1 \frac{15}{16}$ and $WC = 0010.1100001 = 2 \frac{97}{128}$. Use Equation F.17 to determine the quotient digit.

Solution: QSLC uses the most significant bits of the residual. Truncate WS and WC to Q4.0 forms: $WC_{\text{msbs}} = 1110 = -2$, and $WS_{\text{msbs}} = 0010 = 2$. Their sum W_{msbs} is $0000 = 0$. According to Equation F.17, select $q = 1$. The exact sum is $W = 89/128$, which is in the $q = 1$ region according to Equation F.15.

Example F.7 RADIX-2 RECURRENCE DIVISION

Using radix-2 recurrence division, compute $Q = X/D = 1.0000010/1.1010000 = 0.10100000$ (with a remainder of 0). Viewed as $N - 1 = 7$ -bit fractions, $X = 130/128$, $D = 208/128$, so $Q = 130/208 = 80/128$.

Solution: Fig. F.9 shows the division using the algorithm from Fig. F.6. We start with the partial remainder of X by initializing $WS_{-1} = X$ and $WC_{-1} = 0$.

On step 0, we add the four most significant bits to find $W_{\text{msbs}} = WS_{\text{msbs}} + WC_{\text{msbs}} = 0001 + 0000 = 1$. This is greater than -1 , so according to Equation F.17, we select a quotient bit $q_0 = 1$ and divisor multiple $-q_0D = -D$. According to Equation F.1, before the R multiplication, the CSA adds $\{carry, sum\} = WC_{-1} + WS_{-1} - q_0D = WC_{-1} + WS_{-1} + \overline{q_0D} + 1$. We add $\overline{D} + 1$ without carry propagation by selecting $\overline{D}$ and then placing the 1 into the least significant column of WC , as shown in bold. Observe that the digits in the *carry* output are shifted left by one because they have twice the weight of the *sum* digits. *sum* and *carry* are then shifted left by one (r) to produce WS_0 and WC_0 , shown at the start of step 1.

On step 1, $W_{\text{msbs}} = 1110 = -2.0$ so we select a quotient bit $q_1 = -1$ and divisor multiple $-q_1D = D$. Steps 2 and 3 produce quotient bits of 1 and -1 , respectively. After step 3, $sum + carry = 0$, so the partial remainder is 0, the result is exact, and we can terminate early. According to Equation F.9, the quotient is $Q = q_02^0 + q_12^{-1} + q_22^{-2} + q_32^{-3} = 1 - 1/2 + 1/4 - 1/8 = 5/8 = 80/128$, as expected.

F.2.2.2 Radix-2 Implementation

Fig. F.10 shows a radix-2 divider implementation. Each signal is annotated with its format. When *Start* is asserted, the redundant partial remainder $\{WS, WC\}$ is initialized to $\{X, 0\}$. On each step, the quotient digit of $\{-1, 0, \text{ or } +1\}$ is selected based on the 4 most significant bits of the partial remainder. The divisor multiple $\{+D, 0, -D\}$ is chosen with a mux and added to the partial remainder with a CSA to form a new partial remainder, which is shifted right one for the next iteration. Meanwhile, the quotient digits are accumulated in an on-the-fly converter (OTFC) to form Q , as described in Section F.2.5.

F.2.2.3 Quotient Selection Optimization

A brute force QSLC implementation would compute $W_{\text{msbs}} = WC_{\text{msbs}} + WS_{\text{msbs}}$ on the top 4 bits of the partial remainder and then compare to -1 according to Equation F.17. A faster and smaller implementation selects the quotient digit directly from WC_{msbs} and WS_{msbs} using carry chain logic to determine whether the sum is -1 , larger, or smaller, without having to compute the exact sum first.

Equations:	$Q = Q_n = \sum_{i=0}^n q_i R^{-i}$ (E.35)	$q_j = \{1, 0, -1\}$ for $W_{\text{msbs } j} \{>, =, <\} -1$ (E.17)
Operation:	$X/D = Q + W/D$	$1.015625 / 1.625 = 0.625 + 0 / 1.015625$
	X	0001.0000 010
	D	0001.1010 000
		1.625
Initialization:	$-D = \bar{D}$ (+1)	1110.0101 111 (+1 ulp occurs in later steps, when used)
	$WS_{-1} = X$	0001.0000 010
	$WC_{-1} = 0$	0000.0000 000
Step 0:	WS_{-1}	0001.0000 010
	$WC_{-1} + 1$	0000.0000 001 (+1 ulp because of adding $-D$ below)
	$+ -q_0 D = -D$	1110.0101 111 $W_{\text{msbs } -1} = 0001 + 0000 = 0001$, so $q_0 = 1$
	sum_0	1111.0101 100
	$carry_0$	0000.0000 110
Step 1:	WS_0	1110.1011 000 $sum_0 \ll 1$
	WC_0	0000.0001 100 $carry_0 \ll 1$
	$+ -q_1 D = -D$	0001.1010 000 $W_{\text{msbs } 0} = 1110 + 0000 = 1110$, so $q_1 = -1$
	sum_1	1111.0000 100
	$carry_1$	0001.0110 000
Step 2:	WS_1	1110.0001 000 $sum_1 \ll 1$
	$WC_1 + 1$	0010.1100 001 $carry_1 \ll 1$ (+1 ulp because of adding $-D$ below)
	$+ -q_2 D = -D$	1110.0101 111 $W_{\text{msbs } 1} = 1110 + 0010 = 0000$, so $q_2 = 1$
	sum_2	0010.1000 110
	$carry_2$	1100.1010 010
Step 3:	WS_2	0101.0001 100 $sum_2 \ll 1$
	WC_2	1001.0100 100 $carry_2 \ll 1$
	$+ -q_3 D = -D$	0001.1010 000 $W_{\text{msbs } 2} = 0101 + 1001 = 1110$, so $q_3 = -1$
	sum_3	1101.1111 000
	$carry_3$	0010.0001 000
		Terminate Early: $sum_3 + carry_3 = 0$
Result:	Q	0000.1010 000 $0.625 = q_0 \times 2^0 + q_1 \times 2^{-1} + q_2 \times 2^{-2} + q_3 \times 2^{-3}$ $= (1) \times 2^0 + (-1) \times 2^{-1} + (1) \times 2^{-2} + (-1) \times 2^{-3}$ $= 1 - 0.5 + 0.25 - 0.125$
	W	0000.0000 000 $W = (sum_3 + carry_3) \times 2^{-b} = (sum_5 + carry_5) \gg b$
Check:	$X = QD + W$	$1.015625 = (0.625 \times 1.625) + 0$

Fig. F.9 Radix-2 recurrence division example.

Define bitwise propagate and generate logic:

$$p_{3:0} = WC_{\text{msbs } 3:0} \oplus WS_{\text{msbs } 3:0} \quad (\text{F.19})$$

$$g_{2:0} = WC_{\text{msbs } 2:0} \cdot WS_{\text{msbs } 2:0} \quad (\text{F.20})$$

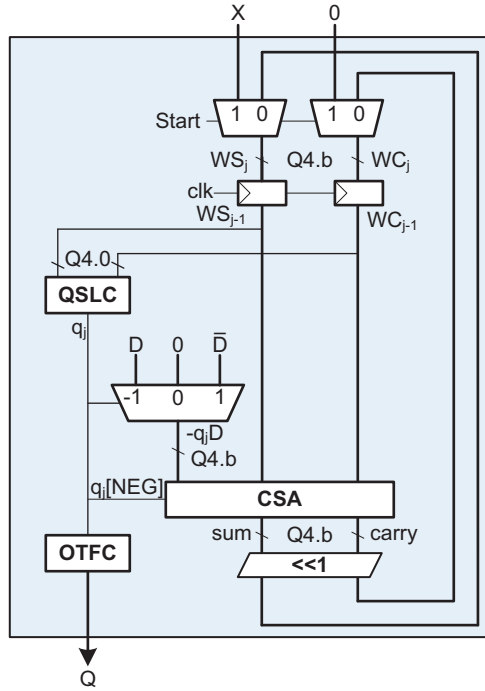


Fig. F.10 Radix-2 recurrence division implementation.

QSLC chooses $q = 0$ when the sum W_{msbs} is $1111 = -1$. A sum of 0111 can never occur because it is out of the containment bounds, so we only need to look at the bottom 3 bits. These bits are 111 when $p_2 = p_1 = p_0 = 1$ because there is no carry-in. Hence, define qz to indicate $q = 0$:

$$qz = p_2 \cdot p_1 \cdot p_0 \quad (\text{F.21})$$

The sign of the result is the most significant bit of W_{msbs} , which depends on p_3 and the carry-out of bits $2:0$. (g_3 is not considered; it would always be 0 because WC and WS are offset by 1 bit.)

$$\text{cout} = g_{2:0} = g_2 + p_2(g_1 + p_1 g_0) \quad (\text{F.22})$$

$$\text{sign} = p_3 \oplus \text{cout} \quad (\text{F.23})$$

If the quotient is nonzero, it must be $+1$ (qp) or -1 (qn) depending on the sign:

$$qp = \overline{\text{sign}} \cdot \overline{qz} \quad (\text{F.24})$$

$$qn = \text{sign} \cdot \overline{qz} \quad (\text{F.25})$$

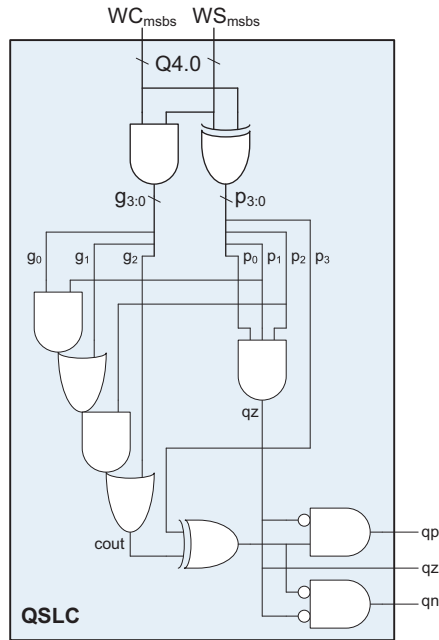


Fig. F.11 Simplified radix-2 QSLC implementation.

Thus, as described in Equation F.17, if W_{msbs} is positive ($\text{sign} = 0$) and not -1 ($qz = 0$), then qp should be 1 (i.e., $W_{\text{msbs}} > -1$). If W_{msbs} is negative ($\text{sign} = 1$) and not -1 ($qz = 0$), then qp should be -1 (i.e., $W_{\text{msbs}} < -1$).

Fig. F.11 shows this logic for radix-2 QSLC.

F.2.3 Radix-4 Recurrence Division

Radix-4 recurrence division uses the same algorithm from Fig. F.6 but with $R = 4$, $r = 2$. The digit set for the radix-4 quotient digit q_i may be *maximally redundant* ($a = 3$, $\rho = 1$) $\{-3, -2, -1, 0, +1, +2, +3\}$ or *minimally redundant* ($a = 2$, $\rho = 2/3$) $\{-2, -1, 0, +1, +2\}$. Minimally redundant divisor multiples are easier to form because each choice comes from shifting and/or negating D ; maximally redundant multiples include $3D$, which requires a CPA to compute $3D = D + 2D$. Therefore, most radix-4 implementations prefer the minimally redundant digit set, even though the QSLC logic is harder. The rest of this section considers minimally redundant radix-4.

According to the containment conditions of Equation F.13, the bounds on W and D for each choice of minimally redundant radix-4 quotient digit are:

$$q_j = \begin{cases} +2 & L_2 \leq W_{j-1} \leq U_2 & W_{j-1} \geq \frac{4}{3}D \\ +1 & L_1 \leq W_{j-1} \leq U_1 & \frac{1}{3}D \leq W_{j-1} \leq \frac{5}{3}D \\ 0 & L_0 \leq W_{j-1} \leq U_0 & -\frac{2}{3}D \leq W_{j-1} \leq \frac{2}{3}D \\ -1 & L_{-1} \leq W_{j-1} \leq U_{-1} & -\frac{5}{3}D \leq W_{j-1} \leq -\frac{1}{3}D \\ -2 & L_{-2} \leq W_{j-1} \leq U_{-2} & W_{j-1} \leq -\frac{4}{3}D \end{cases} \quad (\text{F.26})$$

When adding two numbers truncated to t fraction bits and truncating the sum to $t - 1$ bits, each addend could have a truncation error up to 2^{-t} , and the truncated sum bit also has a weight of 2^{-t} , so the final result is up to 3×2^{-t} less than the exact result. Specifically, for $t = 4$, adding WC_{msbs} and WS_{msbs} in Q4.4 format and truncating the sum W_{msbs} to Q4.3 format introduces a truncation error of up to $3/16$: $0 \leq (W - W_{\text{msbs}}) < 3/16$. This is a tighter bound than $1/4$ obtained if WC_{msbs} and WS_{msbs} were in Q4.3, so the extra bit in the adder buys some accuracy that will prove valuable in digit selection.

Fig. F.12 shows a P-D diagram for minimally redundant radix-4 recurrence division. Careful inspection of the P-D diagram shows that we need the 7 most significant partial remainder bits W_{msbs} in Q4.3 format and the 3 most significant fractional divisor bits in U1.3 format to distinguish quotient digits. An 8-bit adder sums WC_{msbs} and WS_{msbs} in Q4.4 format and then discards the least significant bit to produce a Q4.3 result.

For example, if $WC = 0000.0001111111\dots$ and $WS = 0000.0000111111\dots$ then $WC_{\text{msbs}} + WS_{\text{msbs}} = 0000.0001 + 0000.0000 = 0000.0001$, and $W_{\text{msbs}} = 0000.000$. W_{msbs} has a truncation error of almost $1/8 + 1/16 = 3/16$. This is because the bits not included in the sum are maximal (all 1s) and would have added to $\sim 1/8$ (i.e., $1/16 +$ just less than $1/16$), and the truncated least significant bit of the Q4.4 sum has a weight of $1/16$.

Each darker blue region in Fig. F.12 represents a *selection interval* $M_k = [L_k, U_{k-1} - 1/16]$ in which the quotient digit could either be k or $k - 1$. Let m_k be a *selection constant* with the same number of fraction bits (3) as W_{msbs} within the interval M_k . If $W_{\text{msbs}} \geq m_k$, then $W \geq L_k$, and the hardware can choose quotient digit $q_j = k$. On the other hand, if $W_{\text{msbs}} < m_k$, then $W_{\text{msbs}} \leq m_k - 1/8$ because W_{msbs} has only 3 fraction bits. $W < W_{\text{msbs}} + 3/16$, so $W < m_k + 1/16$. Because $m_k < U_{k-1} - 1/16$, $W < U_{k-1}$. Hence, the hardware can choose quotient digit $q_j = k - 1$.

The divisor is also truncated to 3 fraction bits, so there is some uncertainty in the exact value of the divisor. The P-D diagram is partitioned into small enough boxes so that it is possible to pick a viable m_k (fits between the lower and upper bounds) over an entire interval of divisor values with a given truncated fraction. The overlap region gets larger toward the right, so there is sometimes more than one valid choice of m_k .

To be viable, a selection constant must be in the dark blue selection interval across the full D_{msbs} interval. For example, when $D_{\text{msbs}} = 1.000$, the only viable m_2 is $12/8$. $11/8$ is too small because when $D = 1.000111$, the divider must select $q = 1$. Similarly, $13/8$ is too

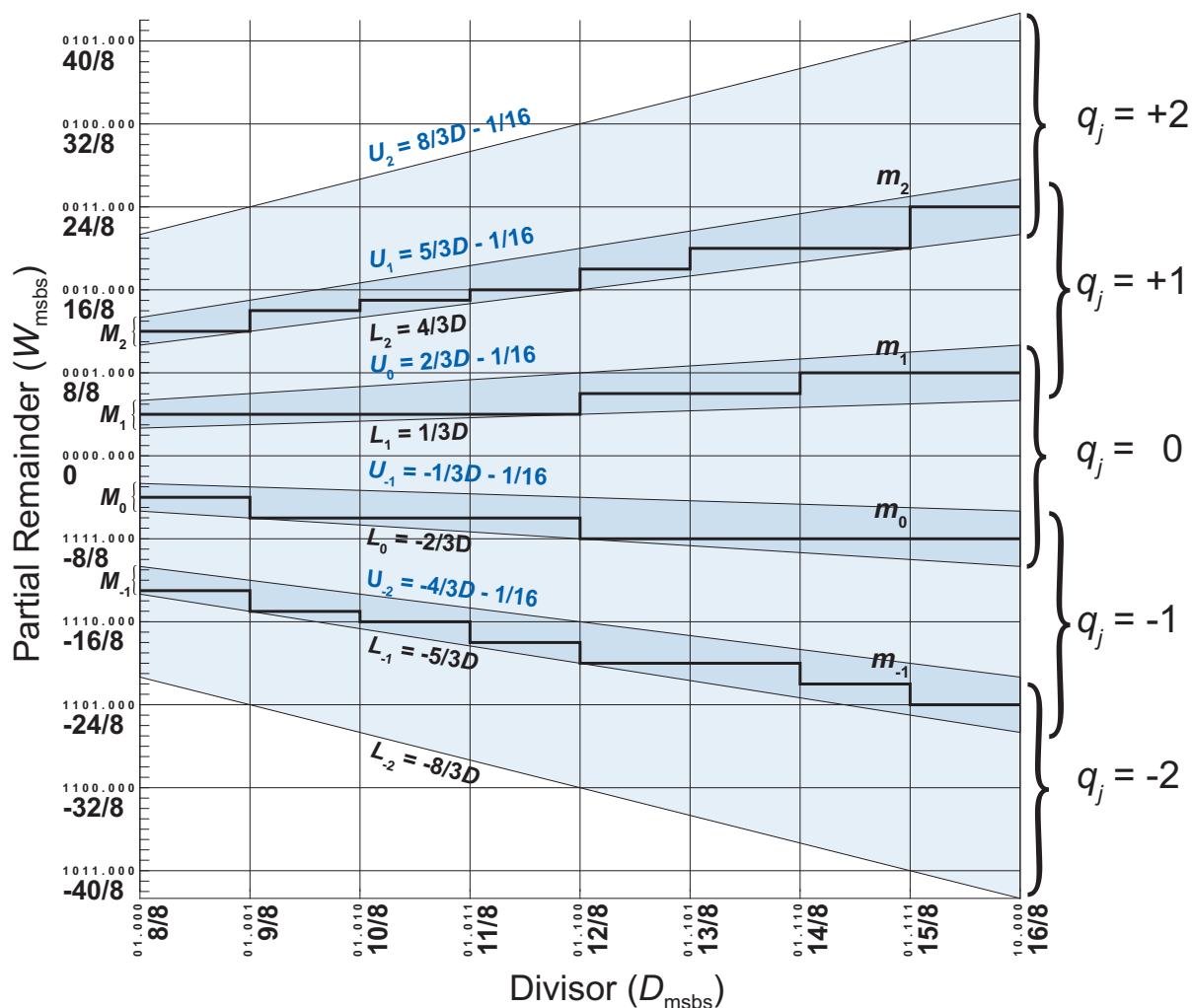


Fig. F.12 Minimally redundant radix-4 division P-D diagram.

large because when $D = 1.000$, $13/8$ is just slightly above the M_2 selection interval, so when W is slightly less than $13/8$, it is not safe to pick $q = 1$. In contrast, when $D_{\text{msbs}} = 1.010$, both $15/8$ and $16/8$ are viable m_2 selection constants.

In summary, the quotient digit is selected by comparing W_{msbs} to the selection constants m_k :

If the P-D diagram used fewer bits of W or D , it would have been impossible to find selection constants that don't cross boundaries between quotient digits.

$$q_j = \begin{cases} +2 & W_{\text{msbs } j-1} \geq m_2 \\ +1 & m_1 \leq W_{\text{msbs } j-1} < m_2 \\ 0 & m_0 \leq W_{\text{msbs } j-1} < m_1 \\ -1 & m_{-1} \leq W_{\text{msbs } j-1} < m_0 \\ -2 & W_{\text{msbs } j-1} < m_{-1} \end{cases} \quad (\text{F.27})$$

Table F.8 shows the viable QSLC constants. All constants are multiples of 1/8, so the 1/8th is dropped for brevity. For example, with $D_{\text{msbs}} = 1.000$, m_2 is listed as 12 (1100), but it is really $12/8 = 1.100$. The columns are indexed by the most significant divisor bits D_{msbs} . Each row indicates a selection constant m_k . Each cell contains one or more viable values of the selection constant for a given divisor. For example, if $D_{\text{msbs}} = 1.100$ and $W_{\text{msbs}} = 1.110 = 14/8$, then the table shows $q_j = +1$ because W_{msbs} is greater than or equal to the choices of m_1 and less than the choices of m_2 . If $W_{\text{msbs}} = 10.010 = 18/8$, then q_j could be chosen as either $+1$ or $+2$.

Table F.9 lists preferred selection constants (a subset of the choices from Table F.8), again in units of 1/8. When multiple viable constants exist for a cell, pick constants to minimize the selection complexity (and area and delay). A straightforward option, as shown in Table F.9, is to pick constants that are a large power of two or that are equal in adjacent cells when possible.

Remarkably, the selection constants in Table F.9 also work for square root, as shown in Section F.2.10.

Table F.8 Viable radix-4 division quotient selection logic constants (QSLC), in units of 1/8th.

D_{msbs}	1.000	1.001	1.010	1.011	1.100	1.101	1.110	1.111
m_2	12	14	15, 16	16, 17	18, 19	19, 20, 21	20, 21, 22	22, 23, 24
m_1	3, 4	4, 5	4, 5, 6	4, 5, 6	5, 6, 7	5, 6, 7, 8	5, 6, 7, 8	6, 7, 8, 9
m_0	-5, -4	-6, -5, -4	-6, -5	-7, -6, -5	-8, -7, -6, -5	-8, -7, -6	-9, -8, -7, -6	-10, -9, -8, -7, -6
m_{-1}	-13	-15, -14	-16	-18, -17	-20, -19, -18	-21, -20	-23, -22, -21	-25, -24, -23, -22

Table F.9 Preferred radix-4 division quotient selection logic constants (QSLC), in units of 1/8th.

D_{msbs}	1.000	1.001	1.010	1.011	1.100	1.101	1.110	1.111
m_2	12	14	16	16	18	20	20	24
m_1	4	4	4	4	6	6	8	8
m_0	-4	-4	-6	-6	-6	-8	-8	-8
m_{-1}	-13	-14	-16	-17	-18	-20	-22	-22

F.2.3.1 Example

Example F.8 RADIX-4 DIVISION

Using radix-4 recurrence division, compute $Q = X / D = 1.00001101/1.00110110 = 00.11011110$, with a round bit of 0 and sticky bit of 1. Viewed as $N = 8$ -bit fractions, $X = 269/256$, $D = 310/256$, so $Q = 269/310 \approx 222/256$.

For quotient selection (QSLC), the divisor has D_{msbs} of 1.001, so we use the second column of Table F.9 for the selection constants. Converting the quotient bits using on-the-fly conversion (OTFC) is explained more in Example F.9.

Solution: Fig. F.13 (on page 1018) shows the division. We start with the partial remainder of X by initializing $WS_{-1} = X$ and $WC_{-1} = 0$. We also compute the multiples of D to use them in later steps. The radix-4 recurrence algorithm operates nearly identically to that of the radix-2 algorithm, but rather than shifting WS and WC by 1 bit, it shifts by 2 bits. Because the inputs have 8 fraction bits, it must compute 2 additional bits for rounding; thus, the values in the algorithm must use 10 fraction bits. Furthermore, this means it must complete six steps total because each step generates 2 bits. After step 5, the algorithm terminates.

For quotient selection (QSLC), the divisor has D_{msbs} of 1.001, so we use the second column of Table F.9 for the selection constants. Converting the quotient bits using on-the-fly conversion (OTFC) is explained more in Example F.9.

F.2.3.2 Radix-4 Implementation

Fig. F.14 (on page 1019) shows a radix-4 divider implementation. It is similar to the radix-2 implementation but has a more complex QSLC and a larger divisor multiple (m_k) mux. It produces 2 bits per iteration instead of 1, so it completes in half as many cycles.

Fig. F.15 shows an implementation of radix-4 QSLC logic. The upper redundant partial remainder is converted to nonredundant form W_{msbs} with an 8-bit CPA, and the bottom bit is discarded, leaving a 7-bit nonredundant partial remainder W in Q4.3 form. Then the QSLC uses a giant 10-input logic function to select the quotient. The quotient is expressed with 4 bits $q[3:0]$ corresponding to +2, +1, -1, and -2, respectively; if none of the bits are hot, the quotient digit is 0.

The 10-input logic function may be implemented in various ways. The simplest is to give a SystemVerilog table of Q as a function of W_{msbs} and D_{msbs} and allow the logic optimization tool to map it onto the best implementation for a given technology. That implementation could be a minimal sum-of-products equation, a multilevel network of NANDs and NORs on an ASIC (application-specific integrated circuit), or a block RAM on an FPGA (field programmable gate array).

Because D is constant during division, another option is to pick four selection constants based on D_{msbs} during preprocessing. Then the QSLC simply compares W_{msbs} to the four constants. Furthermore, the comparison can be performed as $WC_{\text{msbs}} + WS_{\text{msbs}} - m_k \geq 0$ using a CSA followed by an $A + B \geq 0$ comparator (closely related to Fig. 7.6). This is faster than the design from Fig. F.15 but takes more area [Harris23].

Radix-2 QSLC uses only the 4 most significant bits of the residual W to guess a suitable quotient digit. Radix-4 QSLC needs 8 msbs of W and 3 bits of the divisor D to distinguish accurately enough among the larger number of quotient digit choices.

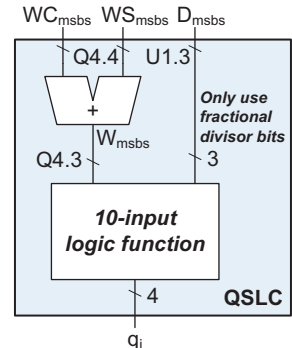


Fig. F.15 Radix-4 QSLC implementation.

Wally's `cvw/src/fpu/fdivsqr/fdivsqrtsqsel14.sv` describes a QSLC lookup table based on Table F.8 and Fig. F.15. `fdivsqrtsqsel14cmp.sv` describes a comparator approach. Wally presently uses the comparator in `fdivsqrtsqstage4`.

Equations:	$Q = Q_n = \sum_{i=0}^n q_i R^{-i}$ (E.35)	$q_j = \{2, 1, 0, -1, -2\}$ (see Equation E.26)
Operation:	$X/D = Q + W/D$ Q X D	$1.05078125/1.2109375 = 0.8681640625 + 0.00051169433\dots$ $= 889/1024 \approx 269/310$ 1.05078125 (269/256) in Q4.10 (Q4.b) format 1.2109375 (310/256) in Q4.10 (Q4.b) format
Initialization:	$2D = D \ll 1$ $-D = \overline{D}$ (+1) $-2D = 2\overline{D}$ (+1) $WS_{-1} = X$ $WC_{-1} = 0$	$0010.0110\ 1100\ 00$ $1110.1100\ 1001\ 11$ (+1 ulp occurs in later steps, when used) $1101.1001\ 0011\ 11$ (+1 ulp occurs in later steps, when used) $0001.0000\ 1101\ 00$ $0000.0000\ 0000\ 00$
Step 0:	WS_{-1} $WC_{-1} + 1$ $+ -q_0 D = -D$ sum_0 $carry_0$	$0001.0000\ 1101\ 11$ $0000.0000\ 0000\ 01$ (+1 ulp because of adding $-D$ below) $1110.1100\ 1001\ 11$ $W_{\text{msbs } -1} = 0001.000 + 0000.000 = 0001.000$, so $q_0 = -1$ $1111.1100\ 0100\ 10$ $0000.0001\ 0010\ 10$
Step 1:	WS_0 WC_0 $+ -q_1 D = D$ sum_1 $carry_1$	$1111.0001\ 0010\ 00$ $sum_0 \ll 2$ $0000.0100\ 1010\ 00$ $carry_0 \ll 2$ $0001.0011\ 0110\ 00$ $W_{\text{msbs } 0} = 1111.000 + 0000.010 = 1111.010$, so $q_1 = -1$ $1110.0110\ 1110\ 00$ $0010.0010\ 0100\ 00$
Step 2:	WS_1 $WC_1 + 1$ $+ -q_2 D = -2D$ sum_2 $carry_2$	$1001.1011\ 1000\ 00$ $sum_1 \ll 2$ $1000.1001\ 0000\ 01$ $carry_1 \ll 2$ (+1 ulp because of adding $-2D$ below) $1101.1001\ 0011\ 11$ $W_{\text{msbs } 1} = 1001.1011 + 1000.1001 = 0010.010$, so $q_2 = 2$ $1100.1011\ 1011\ 10$ $0011.0010\ 0000\ 10$
Step 3:	WS_2 WC_2 $+ -q_3 D = D$ sum_3 $carry_3$	$0010.1110\ 1110\ 00$ $sum_2 \ll 2$ $1100.1000\ 0010\ 00$ $carry_2 \ll 2$ $0001.0011\ 0110\ 00$ $W_{\text{msbs } 2} = 0010.111 + 1100.100 = 1111.011$, so $q_3 = -1$ $1111.0101\ 1010\ 00$ $0001.0100\ 1100\ 00$
Step 4:	WS_3 WC_3 $+ -q_4 D = -2D$ sum_4 $carry_4$	$1101.0110\ 1000\ 00$ $sum_3 \ll 2$ $0101.0011\ 0000\ 01$ $carry_3 \ll 2$ (+1 ulp because of adding $-2D$ below) $1101.1001\ 0011\ 11$ $W_{\text{msbs } 3} = 1101.011 + 0101.000 = 0010.011$, so $q_4 = -2$ $0101.1100\ 1011\ 10$ $1010.0110\ 0000\ 10$
Step 5:	WS_4 WC_4 $+ -q_5 D = -D$ sum_5 $carry_5$	$0111.0010\ 1110\ 00$ $sum_4 \ll 2$ $1001.1000\ 0010\ 01$ $carry_4 \ll 2$ (+1 ulp because of adding $-D$ below) $1110.1100\ 1001\ 11$ $W_{\text{msbs } 4} = 0111.001 + 1001.100 = 0000.101$, so $q_5 = -1$ $0000.0110\ 0101\ 10$ $1111.0001\ 0100\ 10$
Result:	Q W	$Q = q_0 \times 4^0 + q_1 \times 4^{-1} + q_2 \times 4^{-2} + q_3 \times 4^{-3} + q_4 \times 4^{-4} + q_5 \times 4^{-5}$ $= (1) \times 4^0 + (-1) \times 4^{-1} + (2) \times 4^{-2} + (-1) \times 4^{-3} + (2) \times 4^{-4} + (1) \times 4^{-5}$ $= 0.8681640625$ (889/1024) $W = (sum_5 + carry_5) \times 2^{-b} = (sum_5 + carry_5) \gg b$ $= -(0.1000\ 011 \gg 10) = -(0.0000\ 0000\ 00\ 1000\ 011)$ $= -(67/128 \times 2^{-10}) = -(67/2^{17}) = -0.0005116943359375$
Check:	$X = QD + W$	$1.05078125 = (0.8681640625 \times 1.2109375) - 0.000511\dots$

Fig. F.13 Radix-4 division example.

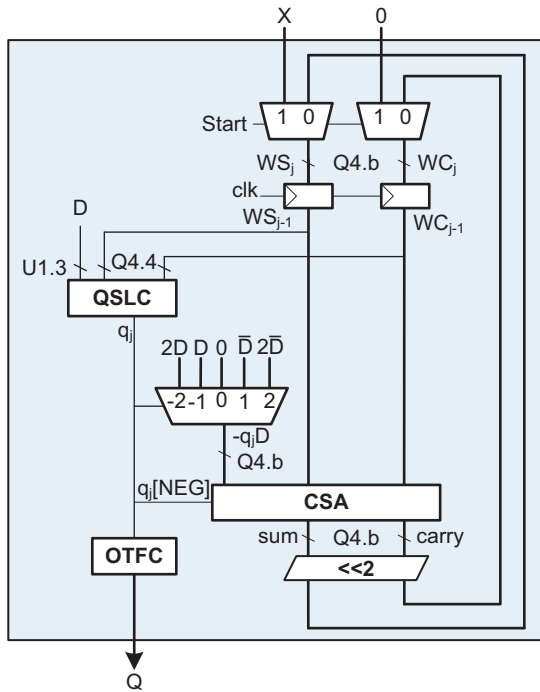


Fig. F.14 Radix-4 division implementation.

F.2.4 Divider Latency

Recurrence division has a long latency to produce all quotient digits. The first step of the floating-point divider produces the integer digit, so b more fraction bits are required, corresponding to $n = \lceil b/r \rceil$ fraction digits in radix- R . Hence, the total number of quotient digits and steps is $n + 1$.

Implementations shown so far perform one step per cycle and thus have an $(n + 1)$ -cycle recurrence latency. However, the logic in a step is fairly simple, so the logic can be replicated k times to compute k steps per cycle and reduce the recurrence latency to $(n + 1)/k$ cycles. Division generally requires at least one cycle for preprocessing and one for postprocessing beyond the recurrence latency.

$(n + 1)$ may not be divisible by k . It is harmless to produce more than the minimal number of quotient bits; the least significant bits are OR'd into the sticky bit. Thus, the quotient is extended to an integer multiple of k digits, or kr bits:

$$n + 1 = k \left\lceil \frac{r + b}{rk} \right\rceil \quad (\text{F.28})$$

The total number of quotient digits and steps is $n + 1$. The total number of bits is at least r (for the integer digit) + b (fraction bits). $n + 1$ must

be a multiple of k and greater than or equal to $(r + b)/r$. Rearranging the previous equation, the number of fractional quotient digits is:

$$n = k \left\lceil \frac{r+b}{rk} \right\rceil - 1 \quad (\text{F.29})$$

F.2.5 Quotient Conversion

According to Equation F.9, the final quotient is the sum of the weighted quotient digits. A simple implementation is to subtract the weighted negative quotient digits from the positive digits. In Example F.7, the quotient digits were $q_0 = 1$, $q_1 = -1$, $q_2 = 1$, $q_3 = -1$, and $q_{4:7} = 0000$. Recalling that digit 0 is the most significant in this notation, we could form $QP = 10100000$ and $QN = 01010000$, then compute $Q = QP - QN = 01010000$. Recall that these quotients are interpreted as U1.7 numbers.

On-the-fly conversion (OTFC) saves the cost and delay of this subtractor by accumulating the quotient digits as they appear on each step [Ercegovac03]. The quotient is initialized to 0. At each step, it is shifted left by r (multiplied by R), and the new quotient digit is added. This is trivial for positive quotient digits because they are less than R , so they are just concatenated. Unfortunately, negative quotient digits still require two's complement carry propagate addition.

To get around this addition, we keep track of two versions of the running quotient, Q and QM . At any given step, $QM = Q - 1$. Fig. F.16 shows the radix- R OTFC algorithm. At each step, we shift the running quotients (QR and QMR) left to obtain $QR = Q \times R$ and

$Q_{-1} = 0; QM_{-1} = -1$	# Initialization
for $j = 0$ to n	
$QR_j = Q_{j-1} \ll r$	# $QR = Q_{j-1} \times R$
$QMR_j = QM_{j-1} \ll r$	# $QMR = QM_{j-1} \times R$
if ($q_j > 0$)	
$Q_j = QR_j + q_j$	# $Q_j = Q_{j-1} \times R + q_j$
$QM_j = QR_j + (q_j - 1)$	# $QM_j = Q_{j-1} - 1$
else if ($q_j = 0$)	
$Q_j = QR_j$	# $Q_j = Q_{j-1} \times R + q_j$
$QM_j = QMR_j + (R - 1)$	# $QM_j = QM_{j-1} \times R + (R - 1)$
	# $= (Q_{j-1} - 1) \times R + (R - 1)$
	# $= Q_{j-1} \times R - 1 = Q_j - 1$
else if ($q_j < 0$)	
$Q_j = QMR_j + (R + q_j)$	# $Q_j = QM_{j-1} \times R + (R + q_j)$
	# $= (Q_{j-1} - 1) \times R + (R + q_j)$
	# $= Q_{j-1} \times R + q_j$
$QM_j = QMR_j + (R - 1 + q_j)$	# $QM_j = QM_{j-1} \times R + (R - 1 + q_j)$
	# $= (Q_{j-1} - 1) \times R + (R - 1 + q_j)$
	# $= Q_{j-1} \times R - 1 + q_j$
	# $= Q_j - 1$

Fig. F.16 On-the-fly conversion (OTFC).

```

 $Q_{-1} = 0; QM_{-1} = -1$ 
for  $j = 0$  to  $n$ 
  if ( $q_j = 1$ )  $Q_j = \{Q_{j-1}, 1\}$   $QM_j = \{Q_{j-1}, 0\}$ 
  else if ( $q_j = 0$ )  $Q_j = \{Q_{j-1}, 0\}$   $QM_j = \{QM_{j-1}, 1\}$ 
  else if ( $q_j = -1$ )  $Q_j = \{QM_{j-1}, 1\}$   $QM_j = \{QM_{j-1}, 0\}$ 

```

Fig. F.17 Radix-2 OTFC.

```

 $Q_{-1} = 0; QM_{-1} = -1$ 
for  $j = 0$  to  $n$ 
  if ( $q_j = 2$ )  $Q_j = \{Q_{j-1}, 10\}$   $QM_j = \{Q_{j-1}, 01\}$ 
  else if ( $q_j = 1$ )  $Q_j = \{Q_{j-1}, 01\}$   $QM_j = \{Q_{j-1}, 00\}$ 
  else if ( $q_j = 0$ )  $Q_j = \{Q_{j-1}, 00\}$   $QM_j = \{QM_{j-1}, 11\}$ 
  else if ( $q_j = -1$ )  $Q_j = \{QM_{j-1}, 11\}$   $QM_j = \{QM_{j-1}, 10\}$ 
  else if ( $q_j = -2$ )  $Q_j = \{QM_{j-1}, 10\}$   $QM_j = \{QM_{j-1}, 01\}$ 

```

Fig. F.18 Radix-4 OTFC.

$QMR = QM \times R$. Then we concatenate a non-negative number to the bottom r bits to obtain the new Q and QM . For example, when $q_i < 0$, we concatenate the positive value $(R + q_i)$ to QMR to find Q . At most, an r -bit subtraction is needed to form the concatenated term; this is inexpensive for $r = 1$ or $r = 2$. After n steps, the quotient is in Q and QM can be discarded or used in rounding.

For radix-2, the OTFC algorithm simplifies to two shift registers loading 0 or 1 into the least significant bit on each step, as shown in Fig. F.17. Note that when $q_j = -1$, Q_j should be $(Q_{j-1} \times 2) - 1 = 2Q_{j-1} - 2 + 1 = 2(Q_{j-1} - 1) + 1 = 2QM_{j-1} + 1 = QM_{j-1} \ll 1 + 1$. For radix-4, the shift registers load 2 bits on each step, as shown in Fig. F.18. Similar to the radix-2 version, but this time with 2-bit shifts (4x multiplications), when $q_j = -1$, Q_i should be $(Q_{j-1} \times 4) - 1 = 4Q_{j-1} - 4 + 3 = 4(Q_{j-1} - 1) + 3 = QM_{j-1} \ll 2 + 3$ and when $q_j = -2$, $Q_j = QM_{j-1} \ll 2 + 2$.

Example F.9 OTFC

Use OTFC to compute the quotient for a minimally redundant radix-4 division from Example F.8 with $q_0 = 1, q_1 = 0, q_2 = -2, q_3 = 0, q_4 = -2, q_5 = 1$. Recall that q_0 is the most significant and q_5 is least significant.

Solution: Table F.10 shows the steps of OTFC, giving a final quotient of $00.1101111001 = 1 \times 2^0 + 0 \times 2^{-2} + -2 \times 2^{-4} + 0 \times 2^{-6} + -2 \times 2^{-8} + 1 \times 2^{-10}$.

The revised recurrence division algorithm with OTFC is given in Fig. F.19, with the new lines highlighted in blue. The original division algorithm was shown in Fig. F.6.

Fig. F.20 shows a hardware implementation of radix-4 OTFC.

Table F.10 OTFC example.

Step j	q_j	Q_j	Q_j Equation	$QM_j = Q_{j-1}$	QM_j Equation
-1	—	000000000000	N/A	111111111111	N/A
0	1	000000000001	$(Q_{j-1} < 2) + 1$	000000000000	$(Q_{j-1} < 2) + 0$
1	0	000000000100	$(Q_{j-1} < 2) + 0$	000000000011	$(Q_{j-1} < 2) - 1 =$ $(QM_{j-1} < 2) + 3$
2	-2	000000001110	$(Q_{j-1} < 2) - 2 =$ $(QM_{j-1} < 2) + 2$	000000001101	$(Q_{j-1} < 2) - 3 =$ $(QM_{j-1} < 2) + 1$
3	0	000000111000	$(Q_{j-1} < 2) + 0$	000000110111	$(Q_{j-1} < 2) - 1 =$ $(QM_{j-1} < 2) + 3$
4	-2	000011011110	$(Q_{j-1} < 2) - 2 =$ $(QM_{j-1} < 2) + 2$	000011011101	$(Q_{j-1} < 2) - 3 =$ $(QM_{j-1} < 2) + 1$
5	1	001101111001	$(Q_{j-1} < 2) + 1$	001101111000	$(Q_{j-1} < 2) + 0$

$WC_{-1} = 0, WS_{-1} = X$

$Q_{-1} = 0, QM_{-1} = \text{all 1s}$

for $j = 0$ to n

$q_j = \text{QSLC}(WC_{\text{msbs } j-1}, WS_{\text{msbs } j-1}, D_{\text{msbs}})$

$\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} - q_j D\} << r$

$\{Q_j, QM_j\} = \text{OTFC}(Q_{j-1}, QM_{j-1}, q_j)$

Fig. F.19 Recurrence division algorithm with OTFC.

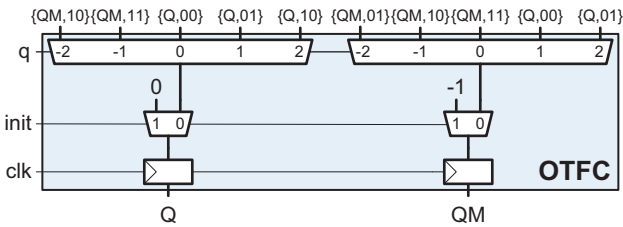


Fig. F.20 Radix-4 OTFC implementation.

F.2.6 Early Termination

If the partial remainder at the start of step j is 0, the quotient is exact, and no more bits need to be computed. This is called *early termination*.

The partial remainder is kept in redundant form, so the divider must check if $W = WS + WC = 0$. The straightforward way of checking completion is with a CPA and equality comparator, but this method is slow and large. A more efficient approach [Cortadella92, Prabhu95, Weste12 p. 463] is to check whether a column's carry-in required to

produce a sum of 0 is equal to the carry-out of the previous column when its sum is 0. If this is true for all columns, the result must be 0.

For column i , if $sum_i = 0$, $WC_i + WS_i + c_{i-1} = 0$, and the carry-in c_{i-1} must be $WC_i \wedge WS_i$. Moreover, the carry-out of a column c_i must be $WC_i \mid WS_i$ (i.e., if either WC_i or WS_i is 1, the carry-out $c_i = 1$; otherwise $c_i = 0$). Hence, we can compute an *eq0* signal indicating $WS + WC = 0$ using N 2-bit XOR and OR gates to compute the necessary carry ins and carry-outs followed by an N -bit equality comparison.

$$eq0 = ((WC \wedge WS) == \{WC[N-2:0] \mid WS[N-2:0], 1'b0\}) \quad (F.30)$$

The comparison requires a left shift by 1 bit for the carry-outs because the carry-outs for the previous columns ($i - 1$) are compared with the carry-ins of a column (i).

For early termination at the end of step j , the quotient must be left-shifted by $n - j$ digits (i.e., $(n - j)r$ bits). A recurrence divider that replicates the logic k times may only check early termination once per cycle after performing k steps to generate kr bits.

Radix-2 digit selection introduces another problem that when the truncated partial remainder is 0, the next digit selected will be 1 rather than 0, according to Equation F.17. This produces a partial remainder of $W_j = (W_{j-1} - q_j D) \ll r = (0 - D) \ll r = -2D$. Each subsequent digit will be -1 , and the partial remainder will remain $-2D$. This is a problem because it will set the Inexact flag and round improperly even though the division is exact [Prabhu95]. It will also waste time not terminating early. For example, a radix-2 divider with $k = 2$ copies will have this problem if the first stage produces an exact result with a partial remainder of 0, but termination is only checked after the second stage, which now has a partial remainder of $-2D$. The problem can be solved by modifying the early termination check to look for either $WS + WC = 0$ or $WS + WC + 2D = 0$. The second condition can be checked with a 3:2 CSA followed by the check for $(A + B) == 0$ described earlier. When the second condition holds, the quotient should be decremented by 2^{-j} to compensate.

The divider should also terminate immediately on special cases if X and/or D are 0, *INF*, or *NaN*.

F.2.7 Subnormal Floating-Point Preprocessing

Suppose the divisor D is subnormal with a leading 1 in the 2^{-m} position, and the dividend X has a leading 1 in the $2^{-\ell}$ position. Subnormal floating-point numbers D and/or X must be left shifted by ℓ and m to the range $[1, 2)$ to apply the recurrence algorithm. The divisor must be in this range to consider only the most significant bits of the partial remainder. The dividend must also be in this range to produce a quotient with N bits of precision.

Therefore, subnormal numbers are normalized before the recurrence using an LZC and left shifter. Because both X and D could be subnormal, the divider can either provide two copies of this expensive hardware or use

With early termination, the latency of division depends on the operands. A malicious program could potentially deduce information about the operands by measuring this latency. Some processors, especially those running cryptographic programs, avoid early termination to prevent leaking this *side-channel* information.

two cycles and one copy of the hardware. The exponent must be adjusted by adding $\ell - m$, as described in [Section F.2.8.2](#).

Example F.10 DIVISION WITH SUBNORMAL NUMBERS

Consider radix-2 floating-point division of subnormal $X/D = 0.0000\ 100/0.0000\ 011$. Find the prenormalized X and D .

Solution: Shift X left by $m = 5$ and D left by $\ell = 6$ to obtain $1.0000\ 000/1.1000\ 000$.

F.2.8 Floating-Point Postprocessing

The recurrence divider produces a raw quotient $Q \in (0.5, 2)$. The postprocessing (`postprocess`) unit generates the final floating-point result $\{Q_s, Q_e, Q_m\}$. Floating-point division shares the same postprocessing steps as other floating-point operations described in [Section 16.2.6](#) and can share the same hardware:

- ▶ Sign and exponent calculation
- ▶ Normalization
- ▶ Rounding
- ▶ Handling special cases of 0, infinity, NaN, and subnormals
- ▶ Flags

Division produces the quotient Q and partial remainder W . If the partial remainder is negative, the quotient is slightly less than Q . Decrement Q by 1 in the last digit by selecting QM instead. This requires a $(WS + WC) < 0$ comparison to determine the sign of W . The sticky bit T is 0 if $W = 0$ (or $-2D$) and 1 otherwise; more simply, the sticky bit is 1 unless the divider terminates with an exact result. Q and QM have at least $b = N_f + 2$ bits, with the G and R bits following the least significant bit L of the fraction. Any extra bits produced by putting the number into subnormal form or when $k > 1$ are ORed into the sticky bit T .

F.2.8.1 Sign

The sign of the quotient is negative if X or D but not both are negative: $Q_s = X_s \oplus D_s$.

F.2.8.2 Exponent

The exponent of the quotient (Q_e) for normalized inputs is the exponent of X minus the exponent of D . Because the exponents are in biased form, Q_e is:

$$Q_e = X_e - D_e + \text{bias} \quad (\text{F.31})$$

If the inputs may be subnormal, then the exponent of the quotient is:

$$Q_e = X_e - D_e + \text{bias} + \ell - m \quad (\text{F.32})$$

Example F.11 ANOTHER DIVISION WITH SUBNORMAL NUMBERS

Consider radix-2 floating-point division of single precision floating-point numbers $0xBBC00000/0x00100000 = -1.5 \times 2^{-8}/0.001 \times 2^{-126}$.

Solution: $X_e = 119$, $D_e = 1$. Remember that subnormals have an effective biased exponent of 1, not 0. The dividend X need not be shifted ($m = 0$) because it is normalized. The divisor D is shifted left by 3 ($\ell = 3$). The quotient exponent is $Q_e = 119 - 1 + 127 + 3 - 0 = 248$ using Equation F.32. The quotient significand is $Q_m = 1.1 / 1.0 = 1.1$. The quotient sign Q_s is negative. Hence, the quotient is $1\ 111\ 1100\ 0\ 1000\dots000 = 0xFC400000 = -1.5 \times 2^{121}$.

F.2.8.3 Normalization

The raw significand Q may be in the range $(0.5, 2)$. If the integer (most significant) bit is 0, the postprocessor must left shift by 1 to normalize the significand and decrement the exponent by 1. The postprocessor contains a variable normalization shift described in Table F.26, which can also be used to handle subnormal outputs and left shifts for early termination.

F.2.8.4 Rounding

The postprocessor determines the effective L' , R' , G' , and T' rounding bits based on whether a normalization shift occurred, as described in Table 16.3. Rounding is performed according to Table 16.4.

F.2.8.5 Special Cases and Flags

Special cases and flags are described in Section F.4.7, especially Table F.34. Overflow or underflow may occur if the exponent is out of range.

F.2.9 Integer Division

Recall that integer division from Section 15.1.4 operates directly on integers without normalization. It uses a full N -bit comparison to produce one quotient bit at each step. The step is relatively slow because of the long comparison. Recurrence division aims for more speed by looking at the most significant bits of a redundant partial remainder. Moreover, the core saves area by reusing the floating-point divider for integer division.

Define integer division to operate on dividend A and divisor B to produce the quotient $QUOT$ and remainder REM such that $A = QUOT \times B + REM$, where REM has the same sign as A . Stated another way, $A/B = QUOT + REM/B$.

To do this integer division on the floating-point recurrence hardware, the preprocessor transforms A and B into positive normalized X and D . The divisor must be kept in normalized form with a 1 in the most significant bit; otherwise, the comparison would need to consider all the bits. The divisor must also be positive to use the recurrence algorithm developed earlier. The preprocessor forms D as the absolute value of B , shifted left to

have a leading 1. Similarly, X is the absolute value of A , shifted left to have a leading 1. The recurrence algorithm then runs for a variable number of steps depending on the number of leading zeros in each original input and may terminate early with an exact answer when the partial remainder becomes 0. Finally, the integer divider must normalize the output, correct the sign, select either the quotient or remainder, and handle the special case of division by 0.

Let A and B be $N = \text{XLEN}$ -bit nonzero integers whose absolute values have ℓ and m leading zeros, respectively. They can be factored as $A = (-1)^{A_s} \times X \times 2^{N-1-\ell}$ and $B = (-1)^{B_s} \times D \times 2^{N-1-m}$ with X and D being normalized U1.($N-1$) numbers in the range $[1, 2)$. The quotient $QUOT = (-1)^{Q_s} \times Q \times 2^p$. Q is a U1. p number in the range $(1/2, 2)$ with $p = \max(m - \ell, 0)$ fraction bits. The sign of the quotient is $Q_s = A_s \oplus B_s$. The magnitude of the quotient $|QUOT| = \left\lfloor \frac{|A|}{|B|} \right\rfloor = Q \times 2^p$ is determined by recurrence division, truncated to n fraction bits. $|QUOT|$ is interpreted as an integer of length XLEN placed in the bottom $p + 1$ bits. The remainder is the residual after computing Q .

If $\ell > m$, $|A| < |B|$, so $QUOT = 0$ and $REM = A$. Recall from [Section 15.2.2](#) that the RISC-V standard defines two other special cases. For division by $B = 0$, $QUOT$ is all 1s and $REM = A$. In the special case of signed division of the largest negative number (e.g., $0x80000000 = -2^{31}$) by -1 , which causes overflow, the large positive result (2^{31}) cannot be represented, according to the standard, $QUOT$ becomes the largest negative number, and $REM = 0$.

F.2.9.1 Integer Preprocessing

Recurrence division operates on positive normalized fixed-point numbers. The preprocessor truncates inputs for W-flavor instructions, takes the absolute value of signed inputs and shifts the numbers into normalized form.

For RV64 W flavors (`divw/remw/divuw/remuw`), the inputs are truncated to 32 bits. The upper 32 bits of the word are filled with 0s (for unsigned flavors) or sign extended from bit 31 (for signed flavors).

Integer division has unsigned (`divu/remu`) and signed (`div/rem`) flavors. Unsigned integers are inherently positive. Two's complement signed integers require an absolute value operation (invert the bits and add 1) when negative. Hence, the preprocessor must take the absolute value of A and B for `div` and `rem`.

The preprocessor normalizes $|A|$ and $|B|$ by counting the number of leading zeros in each input (after taking their absolute values, if applicable) and shifting left by that amount. The N -bit results are interpreted as U1.($N-1$) normalized fixed-point numbers. Note that the number of fraction bits in the divider is b . If $b > N - 1$ (e.g., for RV32GC), then $b - (N - 1)$ trailing zeros must be appended to X and D to convert to U1. b form.

[Fig. F.21](#) shows the preprocessing steps, involving an absolute value circuit, a LZC, and a left shifter. The LZC is simply a priority encoder that produces a $\log_2 N$ -bit count. *signedDiv* is true for `div` and `rem` instructions. *W64* is true for W-flavor instructions.

The floating-point convert unit (see [Section 16.2.4](#)) has an integer absolute value module that could be shared with the divider for X .

```

if (W64) truncate  $A$  and  $B$  to 32 bits
if (signedDiv)  $A_s = \text{sign}(A)$   $B_s = \text{sign}(B)$ 
else  $A_s = 0$   $B_s = 0$ 
if  $A_s$   $|A| = -A$  else  $|A| = A$ 
if  $B_s$   $|B| = -B$  else  $|B| = B$ 
 $\ell = \text{LZC}(|A|)$   $m = \text{LZC}(|B|)$ 
 $X = |A| \ll \ell$   $D = |B| \ll m$ 

```

Fig. F.21 Integer recurrence division preprocessing.

Example F.12 OPERANDS FOR INTEGER DIVISION

Consider 4-bit unsigned radix-2 integer division of $A/B = 6/2 = 0110/0010$. Find the normalized X and D .

Solution: The dividend A has $\ell = 1$ leading zero. Shift A left by 1 to obtain 1100 and interpret as a fixed point number $X = 1.100$. The divisor B has $m = 2$ leading zeros. Shift B left by 2 to obtain 1000 and interpret as a fixed point number $D = 1.000$.

F.2.9.2 Integer Latency

Similar to floating-point division, integer division produces $1 + n$ quotient digits, comprising the integer and $n \geq \lceil p/r \rceil$ fraction digits of Q . The divider can terminate early during exact divisions when the residual becomes 0 because all subsequent fraction digits will be 0. Unlike floating-point division, the residual represents the integer remainder.

A divider with k copies of radix- R hardware produces k r -bit digits, or rk bits of quotient per cycle. In radix- R , the integer digit of Q is r bits (even though it must be 0 or 1), so the total number of bits in Q is at least $r + p$ (one r -bit integer digit and p fraction bits). As with floating-point division, the divider operates for an integral number of cycles $\left(\left\lceil \frac{r+p}{rk} \right\rceil\right)$ and thus may produce extra digits. Similar to Equation F.29, the actual number of fraction digits is:

$$n = k \left\lceil \frac{r+p}{rk} \right\rceil - 1 \quad (\text{F.33})$$

because each cycle produces k digits, and one of the digits is an integer rather than fractional.

The number of quotient bits $p + r$ may not be an exact multiple of rk . Unlike floating-point division, it is problematic for integer division to produce extra trailing quotient digits because the extra steps also affect the residual, changing the remainder. To prevent this problem, integer division right shifts X to introduce leading zeros in X and Q such that the number of digits is exactly right: $p + r + \text{rightshift}$ must be a multiple of rk . Thus, the amount to shift right is:

$$\text{rightshift} = rk - 1 - ((r + p - 1) \bmod rk) \quad (\text{F.34})$$

Example F.13 RIGHT SHIFTING FOR INTEGER DIVISION

Consider a radix-4 divider ($r = 2$) with $k = 4$ copies of the recurrence hardware, producing $rk = 8$ bits per cycle. If the divider must produce $p = 11$ fraction bits, how much right shifting is necessary?

Solution: $rightshiftx = 2 \times 4 - 1 - ((2 + 11 - 1) \bmod 2 \times 4) = 3$.

In summary, integer division preprocessing computes the number of fraction result bits p , the number of result digits n , and the right shift amount $rightshiftx$. It initializes the residual with the shifted value of X . k is preferably a power of two (e.g., 1, 2, or 4) so that the division and modulo operations reduce to shifts and truncations. Preprocessing is summarized here.

$$\begin{aligned} p &= \max(m - \ell, 0) && \# \text{ Number of fraction bits} \\ n &= k \left\lfloor \frac{r+p}{rk} \right\rfloor - 1 && \# n + 1 \text{ digits (0 to } n) \\ rightshiftx &= rk - 1 - ((r + p - 1) \bmod rk) \\ WS &= X \gg rightshiftx \\ WC &= 0 \end{aligned}$$

Integer division then uses the same recurrence hardware as floating-point division, performing $n + 1$ radix- R steps in $(n + 1)/k$ cycles.

F.2.9.3 Integer Postprocessing

Finally, we must convert the quotient and remainder to integers and convert the remainder to have the same sign as the dividend.

Integer division and remainder postprocessing steps include:

- ▶ Residual computation
- ▶ Early termination for exact results
- ▶ Handling special cases
- ▶ Sign correction
- ▶ Selecting quotient or remainder
- ▶ W-flavor sign extension

The residual is computed in $Q4.b$ form as $W = WS + WC$. Sign correction needs to know whether W is 0, positive, or negative.

Early termination for integer division works as described for floating-point division in [Section F.2.6](#). The postprocessor records the step j at which division terminates. j must be a multiple of k . The quotient is interpreted as an integer. If it is formed using OTFC as described in [Section F.2.5](#), it is already placed in the least significant bits of the result. In the case of early termination at the end of step $j < n$, the quotient must be

Even though the integer division result spans the range $(1/2, 2)$ when interpreted in $U1.p$ form, no normalization step is needed because the result will be interpreted as an integer instead.

If the quotient is formed using unified OTFC (UOTFC) as described in [Section F.2.11.6](#), it resides in the most significant bits and must be shifted right into the least significant bits. Because the U register has b fraction bits and division produces n r -bit fractional digits, $|QUOT|$ must be shifted right by $b - rn$. This also accounts for early termination.

left shifted by $n - j$ digits ($(n - j)r$ bits) as with floating-point division to account for the zero digits that would have been appended if the iterations did not terminate.

Unsigned division should have a positive remainder. If the residual is negative, the remainder is corrected by adding D to the residual and decrementing the quotient. $QM = Q - 1$ is already available from the OFTC, so this requires no extra quotient hardware. Moreover, D had been left-shifted by m and then had $b - (N - 1)$ trailing zeros appended at the start of the operation. Hence, the residual must be arithmetic right shifted by $m + b - (N - 1)$ to put the answer into the XLEN least significant bits of $|REM|$. Table F.11 describes sign correction.

Now the integer divider has produced the magnitudes of the quotient and remainder $|QUOT|$ and $|REM|$. Signed division conditionally negates the sign of the result with a two's complementer. Recall that the quotient is negative if A and B have opposite signs, and the remainder is negative if A is negative.

Special cases given in Table F.12 can terminate immediately with the specified result. However, the divider logic computes the correct output for the signed overflow case without explicitly detecting it.

The divider output is selected as $QUOT$ for `div/divu` and REM for `rem/remu`. For W flavors of RV64 division and remainder, the bottom 32 bits of the output is sign extended to 64 bits.

The special case of the most negative number divided by -1 produces the necessary $QUOT$ and REM with the regular division path, so the hardware doesn't have to check for it.

Table F.11 Residual sign correction and shift.

W	$ QUOT $	$ REM $
Non-negative	$Q \gg (b - rn)$	$W \gg (m + b - (N - 1))$
Negative	$(Q - 1 = QM) \gg (b - rn)$	$(W + D) \gg (m + b - (N - 1))$

Table F.12 Integer division special cases.

Case	Condition	$QUOT$	REM
$ A $ significantly less than $ B $	$\ell > m$	0	A
Divide by zero	$B = 0$	All 1s	A
Signed overflow	Signed DIV/REM $A = \text{most negative number}$ $B = -1$	Most negative number	0

F.2.9.4 Summary

Fig. F.22 lists the complete integer division algorithm, involving preprocessing, standard division iterations, and postprocessing.

$$\{QUOT, REM\} = \text{IntDiv}(A, B) \quad A / B = QUOT + REM / B; \quad A = B \times QUOT + REM$$

Preprocess:

if (*W64*) truncate A and B to 32 bits

if (*Signed*) $A_s = \text{sign}(A)$ $B_s = \text{sign}(B)$ $Q_s = A_s \oplus (B_s \ \& \ \overline{\text{Remainder}})$

else $A_s = 0$ $B_s = 0$ $Q_s = 0$

if A_s $|A| = -A$ else $|A| = A$

if B_s $|B| = -B$ else $|B| = B$

$\ell = \text{LZC}(|A|)$ $m = \text{LZC}(|B|)$

$X = |A| \ll \ell$ $D = |B| \ll m$

$p = \max(m - \ell, 0)$

Number of fraction bits

$n = k \left\lceil \frac{r+p}{rk} \right\rceil - 1$

$n + 1$ digits: 0 to n

$\text{rightshiftx} = rk - 1 - ((r + p - 1) \bmod rk)$

$X = X \gg \text{rightshiftx}$

Iterate:

$WS_{-1} = X$

$WC_{-1} = 0$

for $j = 0$ to n

Each cycle handles k iterations

$q_j = \text{QSLC}(WC_{msbsj}, WS_{msbsj}, D_{msbs})$

$\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} - q_j D\} \ll r$

$\{Q_j, QM_j\} = \text{UOTFC}\{Q_j, QM_j, q_j \times (-1)^{Q_s}\}$

if $(j \bmod k = 0 \text{ and } ((WC_j + WS_j == 0) \text{ or}$

$(R = 2 \text{ and } WS + WC_j + 2D = 0))$ terminate now

Postprocess: # Check special cases concurrently with first cycle iteration

if $(B = 0)$ if (*RemOp*) $\text{Result} = A$ else $\text{Result} = \text{all 1s}$ **# Divide by 0**

else if $(\ell > m)$ if (*RemOp*) $\text{Result} = A$ else $\text{Result} = 0$ **# $A < B$**

else **# Normal case**

if $(j < n)$ **# Terminate early exact**

if $(WC + WS = 0)$ $QUOT = Q$ $REM = 0$

else $QUOT = QM$ $REM = 0$

else **# Normal termination**

$W = (WC + WS) \gg \gg r$

if $(W < 0)$ $QUOT = QM$ $REM = W + D$

else $QUOT = Q$ $REM = W$

if (*RemOp*)

$\text{PreResult} = REM$ $\text{NormShift} = m + b - (N - 1)$

else

$\text{PreResult} = QUOT$ $\text{NormShift} = b - rn$

$\text{Result} = (\text{PreResult} \gg \gg \text{NormShift}) \times (-1)^{Q_s}$

if (*W64*) truncate Result to 32 bits; sign-extend if *signedDiv*

Fig. F.22 Integer recurrence division algorithm.

F.2.9.5 Examples

Example F.14 INTEGER DIVISION

Consider dividing 32-bit numbers $A/B = 2^{18}/1$ using a radix-4 integer divider with $k = 4$ copies of the recurrence in each cycle. How many cycles are required? What right pre-shift is required?

Solution: The divider has $r = 2$, $k = 4$, so it produces 8 bits = 4 digits per cycle. A has $\ell = 13$ ($31 - 18$) leading zeros, and B has $m = 31$ ($32 - 1$) leading zeros. The divider needs to compute one integer digit (2 integer bits) and at least $p = m - \ell = 18$ fraction bits. It actually computes $n = k \left\lceil \frac{r+p}{rk} \right\rceil - 1 = 11$ fraction digits. This takes $(1 + 11)/4 = 3$ cycles. The divider produces $3 \times 2 \times 4 = 24$ bits all together, so X must be preshifted right by $rightshiftx = 24 - (2 + 18) = 4$ bits to obtain the desired 2 integer and 18 fraction bits.

Note that Fig. F.22 uses UOTFC, which places the quotient digits in the most significant positions to be compatible with square root. If it used ordinary OTFC, it instead must left shift the quotient by $r(n - j)$ to handle early termination. For normal termination, $j = n$, the quotient digits are in the proper place, and no shift is required.

Example F.15 RADIX-2 INTEGER DIVISION

Using radix-2 unsigned integer division with $k = 1$ copy, compute $\{Q, REM\} = A/B = 11001101/00001011$. The quotient and remainder should be 00010010 and 00000111, respectively. Viewed as 8-bit unsigned integers, $A = 205$, $B = 11$, $Q = 18$, and $REM = 7$, with $205 = 11 \times 18 + 7$.

Solution: The divisor B has $m = 4$ leading zeros. Shift B left by m to normalize $D = 10110000$, interpret as a fixed-point number 1.0110000, and then extend to Q4.7 form to obtain 0001.0110000. The dividend A has $\ell = 0$ leading zeros, so it does not need shifting and can be interpreted as a fixed-point number (1.1001101) and then extended to Q4.7 form: $X = 0001.1001101$. The divider produces $n = 4$ fraction digits. Fig. F.23 shows the division steps.

After the final step in Fig. F.23, the quotient is $Q = 10011$, and the remainder is $W = WS + WC = 1111.0000\ 000$. The remainder is negative, so to match the signs of X and W , we must add D to W and subtract one from the quotient. This yields $Q = 10010 = 18$ and $W = 0000.1110\ 000$. Finally, right shifting W by $m + b - (N - 1) = (4 + 7 - (8 - 1)) = 4$ and interpreting as an 8-bit integer gives the correct remainder of 00000111, or 7.

Example F.16 RADIX-4 INTEGER DIVISION WITH EARLY TERMINATION

Using radix-4 signed integer division with 16-bit precision and $k = 4$ copies, compute $\{Q, REM\} = A/B = -0010101000000000/000000000000110$. The quotient is -0000011100000000 , and the remainder is 0000000000000000. Viewed as signed integers, $A = -10,752$, $B = 6$, $Q = -1,792$, and $REM = 0$, with $-10,752 = -1,792 \times 6 + 0$.

Solution: The divisor magnitude $|B|$ has $m = 13$ leading zeros. Shift it left by m to normalize $D = 1100000000000000$, interpret as a fixed-point number 1.1000000000000000, and then extend to Q4.15 form to obtain 0001.1000000000000000. The dividend $|A|$ has $\ell = 2$ leading zeros, so it is left-shifted and extended to $X = 0001.0101000000000000$. The result has $p = m - \ell = 11$ fraction bits. The number of fraction digits is $n = k \left\lceil \frac{r+p}{rk} \right\rceil - 1 = 4 \left\lceil \frac{2+11}{2 \times 4} \right\rceil - 1 = 7$. X is aligned by right shifting by $rightshiftx = rk - 1 - ((r + p - 1) \bmod rk) = 2 \times 4 - 1 - (2 + 11 - 1) \bmod (2 \times 4) = 3$. Fig. F.24 shows the division steps.

Equations:	$Q = Q_n = \sum_{i=0}^n q_i R^{-i}$ (E.35)	$q_j = \{1, 0, -1\}$ for $W_{\text{msbs } j} \{>, =, <\} -1$ (E.17)
Operation:	$X/D = Q + W/D$ A 1100 1101 B 0000 1011 X 0001.1001 101 D 0001.0110 000	$A/B = \text{QUOT} + \text{REM}/B$ (205/11 = 18 + 7/11) 205 ($\ell = 0$ leading zeros) 11 ($m = 4$ leading zeros) $X = A \ll \ell = 0$ (then prepend 3 0s for Q4.7 format) $D = B \ll m = 4$ (then prepend 3 0s for Q4.7 format)
Initialization:	$-D = \overline{D}$ (+1) $WS_1 = X$ $WC_1 = 0$	1110.1001 111 0001.1001 101 0000.0000 000 (+1 ulp occurs in later steps, when used)
Step 0:	WS_1 $WC_1 + 1$ $+ -q_0 D = -D$ sum_0 $carry_0$	0001.1001 010 0000.0000 001 1110.1001 111 (+1 ulp because of adding $-D$ below) $W_{\text{msbs } -1} = 0001 + 0000 = 0001$, so $q_0 = 1$ 1111.0000 011 0001.0011 010
Step 1:	WS_0 $WC_0 + 1$ $+ -q_1 D = -D$ sum_1 $carry_1$	1110.0000 110 0010.0110 101 1110.1001 111 (+1 ulp because of adding $-D$ below) $W_{\text{msbs } 0} = 1110 + 0010 = 0000$, so $q_1 = 1$ 0010.1111 100 1100.0001 110
Step 2:	WS_1 WC_1 $+ -q_2 D = -D$ sum_2 $carry_2$	0101.1111 000 1000.0011 100 0001.0110 000 $sum_1 \ll 1$ $carry_1 \ll 1$ $W_{\text{msbs } 1} = 0101 + 1000 = 1101$, so $q_2 = -1$ 1100.1010 100 0010.1110 000
Step 3:	WS_2 WC_2 $+ -q_3 D = -D$ sum_3 $carry_3$	1001.0101 000 0101.1100 000 0001.0110 000 $sum_2 \ll 1$ $carry_2 \ll 1$ $W_{\text{msbs } 2} = 0101 + 1001 = 1110$, so $q_3 = -1$ 1101.1111 000 0010.1000 000
Step 4:	WS_3 $WC_3 + 1$ $+ -q_4 D = -D$ sum_4 $carry_4$	1011.1110 000 0101.0000 001 1110.1001 111 $sum_3 \ll 1$ $carry_3 \ll 1$ (+1 ulp because of adding $-D$ below) $W_{\text{msbs } 3} = 1011 + 0010 = 0000$, so $q_4 = 1$ 0000.0111 110 1111.0000 010 Terminate: $n = p = m - \ell = 4 - 0 = 4$; # steps = $n + 1 = 5$
Result:	Q 0001.0011 000 W 0000.1110 000	$0.625 = q_0 \times 2^0 + q_1 \times 2^{-1} + q_2 \times 2^{-2} + q_3 \times 2^{-3} + q_4 \times 2^{-4}$ $= 1 \times 2^0 + (1) \times 2^{-1} + (-1) \times 2^{-2} + (-1) \times 2^{-3} + (1) \times 2^{-4}$ $= 1 + 0.5 - 0.25 - 0.125 + 0.0625$ $= 1.1875 = 1.0011$ $W = sum_4 + carry_4 = 0000.0111 110 + 1111.0000 010$ $= 1111.1000 000$ W is negative, so: $W = W + D$, $QUOT = QUOT - 1$ $W = W + D = 1111.1000 000 + 0001.0110 000$ $= 0000.1110 000$
Integer Result:	$QUOT$ 0001 0010 REM 0000 0111	$QUOT = [Q \gg (b - rn)] - 1$ (b : Q4.b, n : # fraction digits) $= [Q \gg (7 - (1 \times 4))] - 1 = [Q \gg 3] - 1$ $= [0001 0011 000 \gg 3] - 1 = [(000)0 0010 011] - 1$ $= 0001 0011 - 1 = 0001 0010 = 18$ $REM = W \gg [(m + b) - (N - 1)]$ $= W \gg [(4 + 7) - (8 - 1)] = W \gg 4$ $= 0000 1110 000 \gg 4 = (000)0 0000 111 = 7$
Check:	$A = QUOT \times B + REM$	$205 = (18 \times 11) + 7 = 198 + 7$

Fig. F.23 Radix-2 integer division from Example F.15.

Equations:	$Q = Q_n = \sum_{i=0}^n q_i R^{-i}$ (E.35)	$q_j = \{2, 1, 0, -1, -2\}$ (see Equation E.26)
Operation:	$X/D = Q + W/D$	$A/B = QUOT + REM/B$ $(-10,752/6 = -1,792 + 0/6)$
	A 0010 1010 0000 0000	$-10,752$ ($\ell = 2$ leading zeros)
	B 0000 0000 0000 0110	6 ($m = 13$ leading zeros) (Note: $p = m - \ell = 11$)
	X 0001.0101 0000 0000 000	$X = A \ll \ell = 2 = 1.0101$ (then put into Q4.15 format)
	D 0001.1000 0000 0000 000	$D = B \ll m = 13 = 1.1$ (then put into Q4.15 format)
Initialization:	$2D = D \ll 1$ 0011.0000 0000 0000 000	
	$-D = \overline{D}$ (+1) 1110.0111 1111 1111 111	(+1 ulp occurs in later steps, when used)
	$-2D = 2\overline{D}$ (+1) 1100.1111 1111 1111 111	(+1 ulp occurs in later steps, when used)
	$WS_{-1} = X \gg 3$ 0000.0010 1010 0000 000	$rightshftx = rk - 1 - [(r + p - 1) \bmod rk] = 3$
	$WC_{-1} = 0$ 0000.0000 0000 0000 000	
Step 0:	WS_{-1} 0000.0010 1010 0000 000	
	WC_{-1} 0000.0000 0000 0000 000	
	$+ -q_0 D = 0$ 0000.0000 0000 0000 000	$W_{msbs - 1} = 0000.001 + 0000.000 = 0000.001$, so $q_0 = 0$
	sum_0 0000.0010 1010 0000 000	
	$carry_0$ 0000.0000 0000 0000 000	
Step 1:	WS_0 0000.1010 1000 0000 000	$sum_0 \ll 2$
	WC_0 0000.0000 0000 0000 000	$carry_0 \ll 2$
	$+ -q_1 D = 0$ 0000.0000 0000 0000 000	$W_{msbs 0} = 0000.101 + 0000.000 = 0000.001$, so $q_1 = 0$
	sum_1 0000.1010 1000 0000 000	
	$carry_1$ 0000.0000 0000 0000 000	
Step 2:	WS_1 0010.1010 0000 0000 000	$sum_1 \ll 1$
	$WC_1 + 1$ 0000.0000 0000 0000 001	$carry_1 \ll 1$ (+1 ulp because of adding $-2D$ below)
	$+ -q_2 D = -2D$ 1100.1111 1111 1111 111	$W_{msbs 1} = 0010.101 + 0000.000 = 0010.101$, so $q_2 = 2$
	sum_2 1110.0101 1111 1111 110	
	$carry_2$ 0001.0100 0000 0000 010	
Step 3:	WS_2 1001.0111 1111 1111 000	$sum_2 \ll 1$
	WC_2 0101.0000 0000 0001 000	$carry_2 \ll 1$
	$+ -q_3 D = D$ 0001.1000 0000 0000 000	$W_{msbs 2} = 1001.011 + 0101.000 = 1110.011$, so $q_3 = -1$
	sum_3 1101.1111 1111 1110 000	
	$carry_3$ 0010.0000 0000 0010 000	Terminate Early: $sum_3 + carry_3 = 0$
Result:	Q 0000.0001 1100 0000 000	$Q = q_0 \times 4^0 + q_1 \times 4^{-1} + q_2 \times 4^{-2} + q_3 \times 4^{-3}$ $= 0 \times 4^0 + 0 \times 4^{-1} + 2 \times 4^{-2} + (-1) \times 4^{-3}$ $= 7/64 = 0.000111$
	W 0000.0000 0000 0000 000	$W = sum_3 + carry_3 = 0$
Integer Result:	$ QUOT $ 0000 0111 0000 0000	$ QUOT = Q \gg (b - m)$ (b : Q4.b, n : # fraction digits) $= Q \gg (15 - 2 \times 7) = Q \gg 1$ $= 0000 0001 1100 0000 000 \gg 1$ $= (000)0 0000 1110 0000 000 = 1,792$ $= -(0000 0111 0000 0000)$ $= 1111 1001 0000 0000 = -1,792$
	$QUOT$ 1111 1001 0000 0000	
	REM 0000 0000 0000 0000	$REM = W \gg [(m + b) - (N - 1)] = 0$
Check:	$A = QUOT \times B + REM$	$-10,752 = (-1,792 \times 6) + 0$

Fig. F.24 Radix-4 integer division from Example F.16.

The divider terminates early after step 3 because $sum + carry = 0$, so the result is exact, and $REM = 0$. At this point, $PreResult = QUOT = Q = 2 \times 4^{-2} + -1 \times 4^{-3} = 0.000111$, which is expressed in Q4.15 form as 0000 0001 1100 0000 000. $NormShift = b - rn = 15 - 2 \times 7 = 1$. $Result = -(Q >> 1) = -0000 0111 0000 0000 = 1111 1001 0000 0000$. Interpreted as a 16-bit integer gives the expected quotient of -1,792.

F.2.10 Square Root

Floating-point square root can be performed in a fashion remarkably similar to floating-point division, using almost the same hardware. The goal is to compute $S = \sqrt{X}$ for nonnegative *radicand* X .

Square root involves dividing the exponent by 2 and taking the square root of the significand. The exponent must be even to divide by 2. Plausible choices of significand ranges are $[1, 4)$, $[1/2, 2)$, or $[1/4, 1)$. $[1/2, 2)$ complicates digit selection because the square root may or may not have a leading 1 in the integer digit. $[1, 4)$ is also troublesome because the square root will be in the range $[1, 2)$, so the integer digit may need to be either 1 or 2 given that fraction digits may be negative. Hence, square root generally preshifts the significand of X right by 1 or 2 bits to the range $[1/4, 1)$ and increments the exponent of X by 1 or 2 so that the exponent is even. For ease of notation, this section drops further discussion of the exponent because it will be even, and dividing it by 2 is a simple right-shift by 1 bit. X will be in the range $[1/4, 1)$, and S in the range $[1/2, 1)$.

X is in $U0.N_f + 2$ format after the shift. S must be computed in $U1.b$ form with at least $b = N_f + 2$ fraction bits to produce the guard and round bits so that it can be left shifted by 1 during postprocessing into the range $[1, 2)$ and still have 1 extra bit for rounding.

Following the same treatment as division but replacing the quotient Q and quotient digits q_i with square root significand S and digits s_i , we express the j most significant digits of the square root as:

$$S_j = \sum_{i=0}^j s_i R^{-i} \quad (F.35)$$

Hence, S_j is in $U1.jr$ form (j radix- R fraction bits). The final square root after $n = \left\lceil \frac{b}{r} \right\rceil$ steps is:

$$S = S_n = \sum_{i=0}^n s_i R^{-i} \quad (F.36)$$

The *residual* W_j after j steps is given in (Equation F.37). The residual relates to the error when representing the square root to j digits. Before step 0, the residual W_{-1} is initialized to X . After each step, S_j becomes a better approximation to $\sqrt{X}$, and the residual is multiplied by R to keep it in a similar range.

$$W_j = R^{j+1}(X - S_j^2) \quad (F.37)$$

The final answer for S is in the range $[1/2, 1)$. However, intermediate estimates of S_j are in the range $[1/2, 1]$.

Square root refers to W as the *residual* rather than the *partial remainder*. W plays a similar role in both division and square root operations.

We choose the first square root digit $s_0 = 1$ to give an integer part of 1. According to (Equation F.37), we thus must initialize W_0 to $R(X - 1)$. Subtracting 1 from a fraction is easy in two's complement: W_0 has an integer part of -1 , which is all 1s, and a fraction part of X . Multiplying by R is a left shift by r . As with division, we will find from containment analysis in Section F.2.10.2 that W should be in Q4.b format to accommodate the range of the residual.

Example F.17 SQUARE ROOT

Preparing to compute the square root of $1,849 = 1.1100111001 \times 2^{10}$, find the normalized X and W_0 .

Solution: Rewrite the radicand with an even exponent and a fraction in the range $[1/4, 1)$ as $0.011100111001 \times 2^{12}$. X is the fraction part 0.011100111001 . $W_0 = X - 1 = 1111.011100111001$ (in Q4.b format).

In radix-2, s_0 could be chosen as either 0 or 1. In minimally redundant radix-4, s_0 must be 1 because otherwise S could never exceed $\rho = 2/3$. We choose $s_0 = 1$ to write the algorithm identically for both radices.

F.2.10.1 Recurrence Algorithm

W_j , the residual at step j , can be expanded in terms of the previous square root approximation S_{j-1} and the current digit s_j :

$$\begin{aligned}
 W_j &= R^{j+1}(X - S_j^2) \\
 &= R^{j+1}\left(X - (S_{j-1} + s_j R^j)^2\right) \\
 &= R^{j+1}X - R^{j+1}\left(S_{j-1}^2 + 2S_{j-1}s_j R^j + s_j^2 R^{2j}\right) \\
 &= R^{j+1}(X - S_{j-1}^2) - \left(2RS_{j-1}s_j + s_j^2 R^{j+1}\right) \\
 &= R\left[W_{j-1} - \left(2S_{j-1}s_j + s_j^2 R^j\right)\right]
 \end{aligned} \tag{F.38}$$

Defining

$$F_j = -\left(2S_{j-1}s_j + s_j^2 R^j\right) \tag{F.39}$$

gives a recurrence relationship similar to division (Equation F.11).

$$W_j = R[W_{j-1} + F_j] \tag{F.40}$$

The first part of F in Equation F.39, $(2S_{j-1}s_j)$, represents a multiple of the previous square root approximation, and the second part, $(s_j^2 R^j)$, represents the square of the radicand digit shifted right, related to completing the square. To obtain the first term, S_{j-1} must be in binary form. As in division, an OTFC produces S (and SM) on each step. However, S must be placed in the most significant bits instead of the least significant bits as was done for division. To obtain the second term, the algorithm places s_j^2 in a column that changes on each step. Using a shift register that slides right by one digit on each step, the algorithm tracks this column as:

$$K_j = R^{-j} \tag{F.41}$$

```

 $s_0 = 1$ 
 $S_0 = 1.0, SM_0 = 0.0$ 
 $K_0 = 1.0$ 
for  $j = 1$  to  $n$ 
   $s_j = \text{SSLC}(WC_{msbs\ j-1}, WS_{msbs\ j-1}, S_{j-1}, j-1)$ 
   $K_j = K_{j-1} \gg r$ 
   $F_j = \text{FSEL}(S_{j-1}, SM_{j-1}, s_j, K_j)$ 
   $\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} + F_j\} \ll r$ 
   $\{S_j, SM_j\} = \text{SOTFC}(S_{j-1}, SM_{j-1}, s_j, K_j)$ 

```

Fig. F.25 Recurrence square root algorithm with OTFC.

In terms of K , F_j becomes:

$$F_j = -\left(2S_{j-1}s_j + s_j^2K_j\right) \quad (\text{F.42})$$

Fig. F.25 shows a square root recurrence algorithm similar to the division algorithm. The figure drops subscripts on W , F , and S to match the hardware diagram and reduce clutter. When comparing with the division algorithm, the main differences are that this algorithm initializes the first digit $s_0 = 1$, uses a square root digit selection function (SSLC) instead of a quotient digit selection function (QSLC) and selects an addend F instead of a divisor multiple q_jD . The addend F depends on the sliding 1 in the K register. The OTFC also places results starting in the most significant bits to have an accurate estimate of the square root for digit selection.

$WC_0 = 0$, $WS_0 = R(X - 1)$: set WS to all 1s in the integer part and X in the fraction part, then shift left by r .

The next subsections describe the containment of the residual, then uses that to derive the square root digit selection logic. The addend selection chooses F with some clever logic to avoid carry propagation. OTFC is enhanced from the divider to place digits in the correct columns. This section concludes with radix-2 and radix-4 examples.

F.2.10.2 Containment of Residual

At each step, the recurrence algorithm must choose a radicand digit s_j to reduce the error between the square root approximation and the true square root. At step j , the square root approximation is S_j from Equation F.35. The largest value of the remaining weighted digits is:

$$M = \sum_{i=j+1}^n aR^{-i} \quad (\text{F.43})$$

We find an upper bound on M by letting n go to infinity and summing the infinite series:

$$M = \sum_{i=j+1}^{\infty} aR^{-i} = R^{-(j+1)} \frac{a}{1 - \frac{1}{R}} = \frac{a}{R-1} R^{-j} = \rho R^{-j} \quad (\text{F.44})$$

M is a conservative upper bound because the square root has a finite number of digits.

At each step, the difference between the true square root and the approximation must be bounded by M .

$$\begin{aligned}
 -\rho R^j &\leq \sqrt{X} - S_j && \leq \rho R^j && \left| \sqrt{X} - S_j \right| \leq M \\
 -\rho R^j + S_j &\leq \sqrt{X} && \leq \rho R^j + S_j && \text{Add } S_j \\
 (-\rho R^j + S_j)^2 &\leq X && \leq (\rho R^j + S_j)^2 && \text{Square} \\
 \rho^2 R^{2j} - 2\rho R^j S_j + S_j^2 &\leq X && \leq \rho^2 R^{2j} + 2\rho R^j S_j + S_j^2 && \text{Expand squares} \\
 \rho^2 R^{2j} - 2\rho R^j S_j &\leq X - S_j^2 && \leq \rho^2 R^{2j} + 2\rho R^j S_j && \text{Subtract } S_j^2 \\
 \rho^2 R^{j+1} - 2\rho R S_j &\leq R^{j+1} (X - S_j^2) && \leq \rho^2 R^{j+1} + 2\rho R S_j && \text{Multiply by } R^{j+1} \\
 \rho^2 R^{j+1} - 2\rho R S_j &\leq W_j && \leq \rho^2 R^{j+1} + 2\rho R S_j && \text{Substitute } W_j
 \end{aligned}$$

In summary, the residual W_j at step j must be bounded by:

$$\rho^2 R^{j+1} - 2\rho R S_j \leq W_j \leq \rho^2 R^{j+1} + 2\rho R S_j \quad (\text{F.45})$$

S_j is in the range $[1/2, 1]$. According to [Equation F.45](#), for radix-2 with $R = 2, \rho = 1$, W_j is bounded by $(-4, 6)$. For radix-4 with $R = 4, \rho = 2/3$, W_j is bounded by $(-16/3, 64/9)$. Hence, the residual can be represented in Q4.b form as in division.

Next, we need to find bounds on when a particular square root digit s_j may be selected based on the previous residual W_{j-1} and square root approximation S_{j-1} . Express W_j and S_j in terms of W_{j-1} , S_{j-1} , and s_j using [Equations F.35](#) and [F.37](#):

$$W_j = R \left[W_{j-1} - \left(2S_{j-1}s_j + s_j^2 K_j \right) \right] \quad (\text{F.46})$$

$$S_j = S_{j-1} + s_j R^j \quad (\text{F.47})$$

Then substitute W_j and S_j into [Equation F.45](#) and simplify.

$$\begin{aligned}
 \rho^2 R^{j+1} - 2\rho R (S_{j-1} + s_j R^j) &\leq R \left[W_{j-1} - \left(2S_{j-1}s_j + s_j^2 K_j \right) \right] \leq \rho^2 R^{j+1} + 2\rho R (S_{j-1} + s_j R^j) \\
 2(s_j - \rho)S_{j-1} + (s_j - \rho)^2 K_j &\leq W_{j-1} \leq 2(s_j + \rho)S_{j-1} + (s_j + \rho)^2 K_j
 \end{aligned} \quad (\text{F.48})$$

Similar to division, this analysis shows that the system can choose a square root digit $s_j = k$ when the residual is bounded between L_k and U_k :

$$L_k \leq W_{j-1} \leq U_k \quad (\text{F.49})$$

$$L_k = 2(k - \rho)S_{j-1} + (k - \rho)^2 R^j$$

$$U_k = 2(k + \rho)S_{j-1} + (k + \rho)^2 R^j$$

F.2.10.3 Radix-2 Square Root Digit Selection

Radix-2 digit selection bounds are found by substituting $R = 2$, $\rho = 1$ into Equation F.49.

$$s_j = \begin{cases} +1 & L_1 \leq W_{j-1} \leq U_1 & 0 \leq W_{j-1} \leq 4S_{j-1} + 4 \times 2^{-j} \\ 0 & L_0 \leq W_{j-1} \leq U_0 & -2S_{j-1} + 2^{-j} \leq W_{j-1} \leq 2S_{j-1} + 2^{-j} \\ -1 & L_{-1} \leq W_{j-1} \leq U_{-1} & -4S_{j-1} + 4 \times 2^{-j} \leq W_{j-1} \leq 0 \end{cases} \quad (\text{F.50})$$

This resembles the QSLC bounds from Equation F.15, substituting $2S_{j-1}$ for D , but has an additional 2^{-j} term. [Ercegovac03] performs case analysis showing that the same digit selection function can be used as in division, with W_{msbs} again being the 4 most significant bits of the residual in Q4.0 form.

$$s_j = \begin{cases} +1 & W_{\text{msbs}} > -1 \\ 0 & W_{\text{msbs}} = -1 \\ -1 & W_{\text{msbs}} < -1 \end{cases} \quad (\text{F.51})$$

F.2.10.4 Radix-4 Square Root Digit Selection

Minimally redundant radix-4 digit selection bounds are found by substituting $R = 4$, $\rho = 2/3$ into Equation F.49.

$$s_j = \begin{cases} +2 & W_{j-1} \geq \frac{4}{3}2S_{j-1} + \frac{16}{9}4^{-j} \\ +1 & \frac{1}{3}2S_{j-1} + \frac{1}{9}4^{-j} \leq W_{j-1} \leq \frac{5}{3}2S_{j-1} + \frac{25}{9}4^{-j} \\ 0 & -\frac{2}{3}2S_{j-1} + \frac{4}{9}4^{-j} \leq W_{j-1} \leq \frac{2}{3}2S_{j-1} + \frac{4}{9}4^{-j} \\ -1 & -\frac{5}{3}2S_{j-1} + \frac{25}{9}4^{-j} \leq W_{j-1} \leq -\frac{1}{3}2S_{j-1} + \frac{1}{9}4^{-j} \\ -2 & W_{j-1} \leq -\frac{4}{3}2S_{j-1} + \frac{16}{9}4^{-j} \end{cases} \quad (\text{F.52})$$

As with radix-2, these bounds resemble the QSLC bounds from Equation F.26, substituting $2S_{j-1}$ for D but with an additional term related to 4^{-j} . The additional term is greatest for small j . Remarkably, a careful case analysis [Harris23] shows that the square root digit can be selected according to Table F.9. In brief, in the small j cases, the possibilities for S_{j-1} or s_j are restricted, making digit selection easier even though the additional term is larger. For $j > 3$, the selection equation is almost identical to Equation F.26 because the 4^{-j} term is tiny. Unified radix-4 square root and division digit selection are performed as follows:

- Add the top 8 bits of WS and WC and discard the bottom bit to get W_{msbs} in Q4.3 form.

Table F.13 Square root ASEL.

$j - 1$	$S_{\text{msbs } j-1}$	A_{j-1}	Description
0	Any	0.101	Always pick 0.101 on first step
> 0	1.0000	0.111	Use 0.111 entry for 1.000
Otherwise		3 msbs of fraction part of $2S_{j-1}$	Normal case

- Index the QSLC table with W_{msbs} in Q4.3 form and A in U0.3 form.
 - o For division, A is the 3 msbs of the fraction part of D .
 - o For square root, the ASEL (A select) block chooses A_{j-1} according to [Table F.13](#).

F.2.10.5 Addend Input Selection

The FSEL (F select) block generates the various addend input (F_j) choices for step j and selects one based on the square root digit s_j . Recall F_j is defined by [Equation F.42](#), repeated here for convenience:

$$F_j = -(2S_{j-1}s_j + s_j^2K_j) \quad (\text{F.42})$$

Unlike division, F_j involves a term $2S_{j-1}s_j$ proportional to the square root digit s_j that represents a multiple of the previous square root approximation. Furthermore, it has a K_j term that slides right according to the step j , used to complete the square.

To obtain the first term, we need S_{j-1} in binary form. As in division, an OTFC produces S on each step. However, S must be placed in the most significant bits instead of the least significant bits as was done for division.

[Tables F.14](#) and [F.15](#) show how to compute F for radix-2 and radix-4 given the square root digit s_j and the square root approximation S . The F_j column of the table comes from [Equation F.42](#) for the value of s_j on that row. $K_j = R^{-j}$. The square root approximation S_{j-1} at step j has one integer digit and $j - 1$ fraction digits, or $r(j - 1)$ fraction bits. Adding K_j to a term with fewer bits simply involves setting the j th fraction bit with an OR gate. Unfortunately, subtracting K_j would seem to involve carry propagate addition. FSEL cleverly avoids this carry propagate addition by also using $SM_{j-1} = S_{j-1} - R^{-(j-1)}$, expressing S_{j-1} in binary as $a_0.a_1a_2a_3\dots a_{r(j-1)}$ and SM_{j-1} as $b_0.b_1b_2b_3\dots b_{r(j-1)}$. Using the identity for a fixed-point number S with $r(j - 1)$ fraction bits, $-S = \bar{S} + 2^{-r(j-1)}$.

Table F.14 Radix-2 adder input generation (FSEL).

F_j			
s_j	Equation F.32	In terms of S_{j-1} , SM_{j-1} , and j	Bit String
+1	$-2S_{j-1} - K_j$	$2\overline{S_{j-1}} + 2(2^{-(j-1)}) - 2^{-j} = 2\overline{S_{j-1}} + 3*2^{-j}$	$\overline{a_0}\overline{a_1}\overline{a_2}\dots\overline{a_{j-1}}11$
0	0	0	all zeros
-1	$2S_{j-1} - K_j$	$2(SM_{j-1} + 2^{-(j-1)}) - 2^{-j} = 2SM_{j-1} + 3*2^{-j}$	$b_0b_1.b_2\dots b_{j-1}11$

Table F.15 Minimally redundant radix-4 adder input generation (FSEL).

F_j			
s_j	Equation F.32	In terms of S_{j-1} , SM_{j-1} , and j	Bit String
+2	$-4S_{j-1} - 4K_j$	$4\left[\overline{S_{j-1}} + 4^{-(j-1)}\right] - 4(4^{-j}) = 4\overline{S_{j-1}} + 12*4^{-j}$	$\overline{a_0}\overline{a_1}\overline{a_2}\dots\overline{a_{r(j-1)}}1100$
+1	$-2S_{j-1} - K_j$	$2\left[\overline{S_{j-1}} + 4^{-(j-1)}\right] - 4^{-j} = 2\overline{S_{j-1}} + 7*4^{-j}$	$\overline{a_0}\overline{a_1}\overline{a_2}\dots\overline{a_{r(j-1)}}111$
0	0	0	all zeros
-1	$2S_{j-1} - K_j$	$2[SM_{j-1} + 4^{-(j-1)}] - 4^{-j} = 2SM_{j-1} + 7*4^{-j}$	$b_0b_1.b_2b_3\dots b_{r(j-1)}111$
-2	$4S_{j-1} - 4K_j$	$4[SM_{j-1} + 4^{-(j-1)}] - 4(4^{-j}) = 4SM_{j-1} + 12*4^{-j}$	$b_0b_1b_2.b_3\dots b_{r(j-1)}1100$

Example F.18 CALCULATE F_4

For radix-2 square root, compute F_4 given $S_3 = 0.101 = 5/8$ and $s_4 = -1$.

Solution: According to Equation F.42, $F_4 = 2S_3 - K_4 = 2(5/8) - 2^{-4} = 19/16 = 1.0011$ as a Q1.4 number. Using Table F.14, $F_4 = b_0b_1.b_2b_311$. $SM_3 = S_3 - 2^{-3} = 0.100 = b_0.b_1b_2b_3$. Hence, the bit string for $F_4 = 01.0011$. Observe that this bit string is computed using only shifts and appending without two's complementing or carry propagate subtraction.

Example F.19 CALCULATE F_3

For radix-4 square root, compute F_3 given $S_2 = 0.1101 = 13/16$ and $s_3 = 2$.

Solution: According to Equation F.42, $F_3 = -4S_2 - 4K_3 = -4(13/16) - 4(4^{-3}) = -53/16 = 100.101100$ as a Q3.6 number. Using Table F.15, $F_3 = \overline{a_0}\overline{a_1}\overline{a_2}\dots\overline{a_{r(j-1)}}1100$. $S_2 = a_0.a_1a_2a_3a_4 = 0.1101$. Hence, the bit string for $F_3 = 100.101100$. Again, this bit string is computed using only shifts and appending without two's complementing or carry propagate subtraction.

Table F.16 Radix-2 adder input generation (FSEL) using thermometer code.

F_j		
s_j	Bit String	Boolean Logic
+1	$\overline{a_0} \overline{a_1} \overline{a_2} \dots \overline{a_{j-1}} 11$	$(S_{j-1} < 1) \& C_j$
0	all zeros	0
-1	$b_0 b_1 b_2 \dots b_{j-1} 11$	$(SM_{j-1} < 1) \mid [C_j \& (\overline{C_j} < 2)]$

Table F.17 Minimally redundant radix-4 adder input generation (FSEL) using thermometer code.

F_j		
s_j	Bit String	Boolean Logic
+2	$\overline{a_0} \overline{a_1} \overline{a_2} \overline{a_3} \dots \overline{a_{r(j-1)}} 1100$	$(S_{j-1} < 2) \& (C_j < 2)$
+1	$\overline{a_0} \overline{a_1} \overline{a_2} \overline{a_3} \dots \overline{a_{r(j-1)}} 111$	$(S_{j-1} < 1) \& C_j$
0	all zeros	0
-1	$b_0 b_1 b_2 b_3 \dots b_{r(j-1)} 111$	$(SM_{j-1} < 1) \mid [C_j \& (\overline{C_j} < 3)]$
-2	$b_0 b_1 b_2 b_3 \dots b_{r(j-1)} 1100$	$(SM_{j-1} < 2) \mid [(C_j < 2) \& (\overline{C_j} < 4)]$

Unfortunately, FSEL still involves shifting strings of bits by a variable amount j , which is a costly operation. To reduce FSEL to simple Boolean logic, we recode the walking 1 K_j in thermometer format:

$$C_j = -K_j = -R^{-j} = 1111.11..11 \quad (\text{F.53})$$

Observe that C_j is a two's complement number with all integer 1s and rj fractional 1s. C is easy to generate in hardware with a shift register that shifts right by r bits on each step, bringing in 1s from the left. Now the trailing bits of the bit string can be expressed in terms of simple constant shift, AND, OR, and NOT operations on S_{j-1} , SM_{j-1} , and C_j , as given in [Tables F.16](#) and [F.17](#).

The recurrence units operate on Q4.b residuals, so C also has 4 leading 1s.

Example F.20 CALCULATE F_3 USING SHIFTS AND LOGIC OPERATIONS

For radix-4 square root, compute F_3 using the Boolean logic from [Table F.17](#) given $S_2 = 0.1101 = 13/16$ and $s_3 = 2$.

Solution: $C_3 = 1111.111111$. $F_j = (\overline{0.1101} \ll 2) \& (1111.111111 \ll 2) = \overline{11.01} \& 1111.111100 = \overline{0011.010000} \& 1111.111100 = 1100.101111 \& 1111.111100 = 1100.101100$, which matches the result from [Example F.19](#) with one extra leading 1 for Q4.6 format.

F.2.10.6 On-the-Fly Conversion

On-the-fly conversion (OTFC) is similar for square root as for division but must place the digits in the most significant positions to create the S_j approximation used for digit and addend selection rather than shifting them in from the least significant position. [Fig. F.26](#) shows the generic square root OTFC (SOTFC) algorithm. On step j , it concatenates the digit s_j to the next column (with a weight of K_j). The algorithm uses the same approach as division OTFC to handle negative quotient digits and $SM = S - K$.

Again, this formulation involves K_j , which implies a variable shift. This algorithm also uses the thermometer coded C_j to replace the variable shift with simple constant shifts and Boolean operations. [Fig. F.27](#) shows radix-2 SOTFC.

Example F.21 RADIX-2 SOTFC

Apply radix-2 SOTFC to convert the square root digits $s_0 = 1, s_1 = -1, s_2 = 1, s_3 = 0, s_4 = -1, s_5 = -1, s_6 = 1$ into binary form. These digits are produced in [Example F.23](#).

Solution: [Table F.18](#) shows C , S , and SM at each step. After six steps, the final answer is $1 \times 2^0 + -1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + -1 \times 2^{-4} + -1 \times 2^{-5} + 1 \times 2^{-6} = 0.101011$.

$S_0 = 1; SM_0 = 0$

for $j = 1$ to n

if $(s_j > 0)$	$S_j = S_{j-1} + s_j K_j$	$SM_j = S_{j-1} + (s_j - 1) K_j$
else if $(s_j = 0)$	$S_j = S_{j-1}$	$SM_j = SM_{j-1} + (R - 1) K_j$
else if $(s_j < 0)$	$S_j = SM_{j-1} + (R - s_j) K_j$	$SM_j = SM_{j-1} + (R - 1 - s_j) K_j$

Fig. F.26 Generic SOTFC.

$S_0 = 1; SM_0 = 0; C_0 = -1$

for $j = 1$ to n

$C_j = C_{j-1} \ggg 1$

$K_j = C_j \& (\overline{C_j} \ll 1)$

if $(s_j = 1)$	$S_j = S_{j-1} \mid K_j$	$SM_j = S_{j-1}$
else if $(s_j = 0)$	$S_j = S_{j-1}$	$SM_j = SM_{j-1} \mid K_j$
else if $(s_j = -1)$	$S_j = SM_{j-1} \mid K_j$	$SM_j = SM_{j-1}$

Fig. F.27 Radix-2 SOTFC.

$\ggg$ indicates an arithmetic shift right, which preserves the sign bit in the most significant position.

Table F.18 SOTFC radix-2 example.

Step	s_j	C_j	K_j	S_j	SM_j
0 (initialize)	1	1111.000000		0001.000000	0000.000000
1	-1	1111.100000	0000.100000	0000.100000	0000.000000
2	1	1111.110000	0000.010000	0000.110000	0000.100000
3	0	1111.111000	0000.001000	0000.110000	0000.101000
4	-1	1111.111100	0000.000100	0000.101100	0000.101000
5	-1	1111.111110	0000.000010	0000.101010	0000.101000
6	1	1111.111111	0000.000001	0000.101011	0000.101010

$S_0 = 1$; $SM_0 = 0$; $C_0 = -1$

for $j = 1$ to n

$C_j = C_{j-1} >>> 2$

$K_j = C_j \& (\overline{C_j} << 1)$

if ($s_j = 2$) $S_j = S_{j-1} \mid K_j << 1$

$SM_j = S_{j-1} \mid K_j$

else if ($s_j = 1$) $S_j = S_{j-1} \mid K_j$

$SM_j = S_{j-1}$

else if ($s_j = 0$) $S_j = S_{j-1}$

$SM_j = SM_{j-1} \mid (K_j << 1) \mid K_j$

else if ($s_j = -1$) $S_j = SM_{j-1} \mid (K_j << 1) \mid K_j$

$SM_j = SM_{j-1} \mid (K_j << 1)$

else if ($s_j = -2$) $S_j = SM_{j-1} \mid (K_j << 1)$

$SM_j = SM_{j-1} \mid K_j$

Fig. F.28 Radix-4 SOTFC.

Fig. F.28 shows radix-4 SOTFC. Each step computes $S_j = S_{j-1} + s_j K_j$ by ORing shifted flavors of K_j onto S_{j-1} or SM_{j-1} .

F.2.10.7 Early Termination

Like division, square root can terminate early when the residual is 0. This is detected in the same way by checking if $WS + WC = 0$.

Like division, if square root produces more than one digit per cycle and early termination is only checked once per cycle, radix-2 digit selection will select a 1 followed by a stream of -1s and produce a negative residual and inexact result. This condition can be detected when the square root's residual corresponds to being exactly high by 1 in the current digit. Substituting $W_{j-1} = 0$ and $s_j = 1$ or $W_{j-1} = 2(-2S_j + K_j)$ and $s_j = -1$ into the recurrence and grinding through algebra gives:

$$W_j = 2(-2S_j + K_j) \quad (\text{F.54})$$

Table F.19 Early termination example.

j	$K_j = 2^{-j}$	s_j	$S_j = S_{j-1} + s_j K_j$	$F_j = -(2S_{j-1}s_j + s_j^2 K_j)$	$W_j = 2(W_{j-1} + F_j)$	$2(2S_j - K_j)$
0	1	1	1		Init to $2(1-X) = -1.1$	2
1	0.1	-1	0.1	1.1	0	1
2	0.01	1	0.11	-1.01	-10.1	10.1
3	0.001	-1	0.101	1.011	-10.01	10.01
4	0.0001	-1	0.1001	1.0011	-10.001	10.001

Hence, radix-2 square root should also terminate after step j if $WS + WC + 2(2S_j - K_j) = 0$ and the quotient digit is -1 . When this occurs, the square root is exactly $S = S_j - 2^{-j}$. Note that the third term can be computed without addition using:

$$2(2S_j - K_j) = 2(2(SM_j + K_j) - K_j) = 4SM_j + 2K_j \quad (\text{F.55})$$

If the digits of SM_j are $b_0.b_1b_2\dots b_j$, then the term can be expressed as $b_0b_1b_2.b_3\dots b_j1$, which is formed as $(SM_j < 2) \mid (K_j < 1)$.

Example F.22 SQUARE ROOT WITH EARLY TERMINATION

Using radix-2 recurrence square root, compute $\sqrt{0.01} = 0.1$. In other words, $\sqrt{X} = S$, where $X = 1/4$ and $S = 1/2$. Check termination after $k = 4$ steps.

Solution: Table F.19 shows the results at each step. On step 1, W_j becomes 0 because the result is exact, and on subsequent steps, $W_j + 2(2S_j - K_j) = 0$ indicating that the recurrence can terminate exactly with an adjustment to S . Note that values are written in nonredundant sign/magnitude form for ease of reading even though hardware normally uses two's complement form, and W_j is normally stored in carry-save redundant form.

F.2.10.8 Radix-2 Example

Example F.23 RADIX-2 RECURRENCE SQUARE ROOT

Using radix-2 recurrence square root, compute a square root for the half precision number $X = 0x3F39 = 1.1100\ 1110\ 01 \times 2^0$. Viewed in fractional form, $X = 1849/1024$ and the square root $S = 43/32$.

Solution: The input has an exponent of 0, so right shift by 2 to get X in the range $[1/4, 1)$ while keeping an even exponent: $X = 0.0111\ 0011\ 1001$. Fig. F.29 shows the square root steps. The recurrence terminates early with a result of 0.101011. Shift left by 1 to get the normalized square root $S = 43/32 = 1.01011 = 0x3D60$.

Equations:	$s_j = \{1, 0, -1\}$ for $W_{\text{msbs } j} \{>, =, <\} -1$ (E.51) F_j (Tables E.14 and E.16)	$K_j = R^j$ (E.41) $C_j = -R^j = -K_j$ (E.53)	$S_j = \sum_{i=0}^j s_i R^{-i} = a_0.a_1\dots a_{j-1}$ (E.35) $SM_j = S_j - R^j = b_0.b_1\dots b_{j-1}$
Operation:	$S = \sqrt{X}$ X (original value) X	1.1100 1110 0100 0000.0111 0011 1001	$S = \sqrt{1.80566\dots} = 1.34375$ ($\sqrt{1849/1024} = 43/32$) 1849/1024, exp = 0 (X before shift to $[1/4, 1)$ and Q4.12) $X = X$ (original val) $>> 2$ to range $[1/4, 1)$, exp = 0 + 2 = 2
Initialization:	$WS_0 = 2(X - 1)$ $WC_0 = 0$	1110.1110 0111 0010 0000.0000 0000 0000	(0000.0111 0011 1001 + 1111.0000 0000 0000) $<< 1$ $s_0 = 1$
Step 1:	WS_0 WC_0 $+ F_1 = 2S_0 - K_1$ $= b_0 1.1$	1110.1110 0111 0010 0000.0000 0000 0000 0001.1000 0000 0000	$W_{\text{msbs } 0} = 1110 + 0000 = 1110$, so $s_1 = -1$ $K_1 = 0000.1000 0000 0000$, $S_0 = 1 = a_0$ $C_1 = 1111.1000 0000 0000$, $SM_0 = 0 = b_0$ $F_1 = 01.1$ (in Q4.12 = 0001.1000 0000 0000)
	sum_1 $carry_1$	1111.0110 0111 0010 0001.0000 0000 0000	
Step 2:	$WS_1 = sum_1 << 1$ $WC_1 = carry_1 << 1$ $+ F_2 = -2S_1 - K_2$ $= \overline{a_0} \overline{a_1} 11$	1110.1100 1110 0100 0010.0000 0000 0000 1110.1100 0000 0000	$W_{\text{msbs } 1} = 1110 + 0010 = 0000$, so $s_2 = 1$ $K_2 = 0000.0100 0000 0000$, $S_1 = 0.1 = a_0.a_1$ $C_2 = 1111.1100 0000 0000$, $SM_1 = 0.0 = b_0.b_1$ $F_2 = 10.11$ (in Q4.12 = 1110.1100 0000 0000)
	sum_2 $carry_2$	0010.0000 1110 0100 1101.1000 0000 0000	
Step 3:	$WS_2 = sum_2 << 1$ $WC_2 = carry_2 << 1$ $+ F_3 = 0$	0100.0001 1100 1000 1011.0000 0000 0000 0000.0000 0000 0000	$W_{\text{msbs } 2} = 0100 + 1011 = 1111$, so $s_3 = 0$ $K_3 = 0000.0010 0000 0000$, $S_2 = 0.11 = a_0.a_1a_2$ $C_3 = 1111.1110 0000 0000$, $SM_2 = 0.10 = b_0.b_1b_2$ $F_3 = 00.000$ (in Q4.12 = 0000.0000 0000 0000)
	sum_3 $carry_3$	1111.0001 1100 1000 0000.0000 0000 0000	
Step 4:	$WS_3 = sum_3 << 1$ $WC_3 = carry_3 << 1$ $+ F_4 = 2S_3 - K_4$ $= b_0b_1.b_2b_311$	1110.0011 1001 0000 0000.0000 0000 0000 0001.0111 0000 0000	$W_{\text{msbs } 3} = 1110 + 0000 = 1110$, so $s_4 = -1$ $K_4 = 0000.0001 0000 0000$, $S_3 = 0.110 = a_0.a_1a_2a_3$ $C_4 = 1111.1111 0000 0000$, $SM_3 = 0.101 = b_0.b_1b_2b_3$ $F_4 = 01.0111$ (in Q4.12 = 0001.0111 0000 0000)
	sum_4 $carry_4$	1111.0100 1001 0000 0000.0110 0000 0000	
Step 5:	$WS_4 = sum_4 << 1$ $WC_4 = carry_4 << 1$ $+ F_5 = 2S_4 - K_5$ $= b_0b_1.b_2b_3b_411$	1110.1001 0010 0000 0000.1100 0000 0000 0001.0101 1000 0000	$W_{\text{msbs } 4} = 1110 + 0000 = 1110$, so $s_5 = -1$ $K_5 = 0000.0000 1000 0000$, $S_4 = 0.1011 = a_0.a_1a_2a_3a_4$ $C_5 = 1111.1111 1000 0000$, $SM_4 = 0.1010 = b_0.b_1b_2b_3b_4$ $F_5 = 01.01011$ (in Q4.12 = 0001.0101 1000 0000)
	sum_5 $carry_5$	1111.0000 1010 0000 0001.1010 0000 0000	
Step 6:	$WS_5 = sum_5 << 1$ $WC_5 = carry_5 << 1$ $+ F_6 = -2S_5 - K_6$ $= \overline{a_0} \overline{a_1} \overline{a_2} \overline{a_3} \overline{a_4} 11$	1110.0001 0100 0000 0011.0100 0000 0000 1110.1010 1100 0000	$W_{\text{msbs } 5} = 1110 + 0011 = 0001$, so $s_6 = 1$ $K_6 = 0000.0000 0100 0000$, $S_5 = 0.10101 = a_0.a_1a_2a_3a_4a_5$ $C_6 = 1111.1111 1100 0000$, $SM_5 = 0.10100 = b_0.b_1b_2b_3b_4b_5$ $F_6 = 10.101011$ (in Q4.12 = 1110.1010 1100 0000)
	sum_6 $carry_6$	0011.1111 1000 0000 1100.0000 1000 0000	
Result:	S	0001.0101 100 0000	Terminate Early: $sum_6 + carry_6 = 0$ $= s_0 \times 2^0 + s_1 \times 2^{-1} + s_2 \times 2^{-2} + s_3 \times 2^{-3} + s_4 \times 2^{-4} + s_5 \times 2^{-5} + s_6 \times 2^{-6}$ $= (1)2^0 + (-1)2^{-1} + (1)2^{-2} + (0)2^{-3} + (-1)2^{-4} + (-1)2^{-5} + (1)2^{-6}$ $= 43/64$, exp = 1: $(0.1010 11 \times 2^1) = 43/32$ Then $<< 1$ to normalize: $1.0101 1 \times 2^0 = 1.34375 = 43/32$

Fig. F.29 Radix-2 square root example.

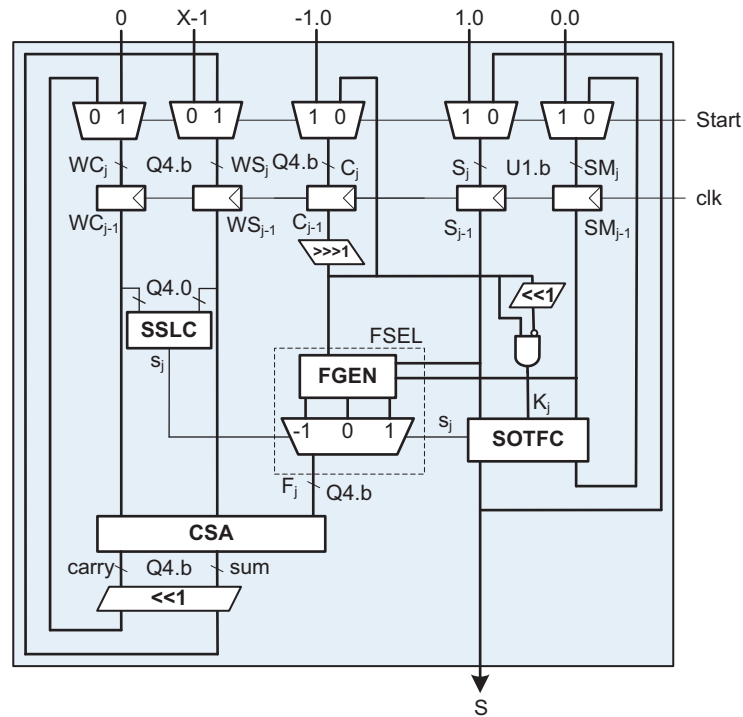


Fig. F.30 Radix-2 square root implementation.

F.2.10.9 Radix-2 Implementation

Fig. F.30 shows an implementation of a radix-2 square root unit. Like the divider, it stores the residual in redundant form. The QSLC is replaced with an SSLC unit to select the square root digit. The divisor multiple selection mux is replaced with an F selection mux. Like the divider, a CSA sums the redundant residual and the selected addend; then the redundant result is left shifted by one and recycled for the next step. A C register holds the sliding ones thermometer code used for F generation. The OTFC is replaced with a SOTFC that also receives C to place the square root digits in the appropriate column.

F.2.10.10 Radix-4 Example

Example F.24 RADIX-4 RECURRENCE SQUARE ROOT

Using radix-4 recurrence square root, compute a square root for the half-precision number $X = 0x51BC = 1.0110\ 1111\ 00 \times 2^5$.

Solution: Shift the input right by 1 to make the exponent even: $0.1011\ 0111\ 100 \times 2^6$ and the radicand in the range $[1/4, 1)$. Fig. F.31 shows the square root steps for $\sqrt{0.1011\ 0111\ 1000} \approx 0.1101\ 1000\ 1011$. The sticky bit is negative. Shifting back to have a leading 1 and restoring the exponent, the prerounded result is $1.1011\ 0001\ 01 \times 2^2 = 0x46C5$.

The leading four bits of C are always 1, so the register holding them is optimized away in Fig. F.30.

Equations:	$s_j = \{2, 1, 0, -1, -2\}$ (see Equation E.52) F_j (see Tables E.15 and E.17)	$K_j = R^j$ (E.41) $C_j = -R^j = -K_j$ (E.53)	$S_j = \sum_{i=0}^j s_i R^{-i} = a_0.a_1\dots a_{j-1}$ (E.35) $SM_j = S_j - R^j = b_0.b_1\dots b_{j-1}$
Operation:	$S = \sqrt{X}$ X (original value) X (normalized) X	101101.111 $\times 2^0$ 1.0110 1111 00 $\times 2^5$ 0000.1011 0111 1000	$S = \sqrt{45.875} = 6.773\dots$ ($\sqrt{367/8} \approx 867/128$) 45.875 = 367/8 (next: shift and into Q4.b format, $b = 12$) X before shift to [1/4, 1), even exp, and Q4.12 format $X = X$ (normalized) $\gg 1$ to range [1/4, 1), exp = 5 + 1 = 6
Initialization:	$WS_0 = 4(X - 1)$ $WC_0 = 0$	1110.1101 1110 0000 0000.0000 0000 0000	(0000.1011 0111 1000 + 1111.0000 0000 0000) $\ll 2$ $s_0 = 1$
Step 1:	WS_0 WC_0 $+ F_1 = 2S_0 - K_1$ $= b_01.11$	1110.1101 1110 0000 0000.0000 0000 0000 0001.1100 0000 0000	$W_{msbs\ 0} = 1110.110 + 0000.000 = 1110.110$, so $s_1 = -1$ $K_1 = 0000.0100 0000 0000$, $S_0 = 1$ $C_1 = 1111.1100 0000 0000$, $SM_0 = 0$ $F_1 = 01.11$ (in Q4.12 = 0001.1100 0000 0000)
	sum_1 $carry_1$	1111.0001 1110 0000 0001.1000 0000 0000	
Step 2:	$WS_1 = sum_1 \ll 2$ $WC_1 = carry_1 \ll 2$ $+ F_2 = -4S_1 - 4K_2$ $= \overline{a_0} \overline{a_1} \overline{a_2}.1100$	1100.0111 1000 0000 0110.0000 0000 0000 1100.1100 0000 0000	$W_{msbs\ 1} = 1100.011 + 0110.000 = 0010.011$, so $s_2 = 2$ $K_2 = 0000.0001 0000 0000$, $S_1 = 0.11$ $C_2 = 1111.1111 0000 0000$, $SM_1 = 0.10$ $F_2 = 100.1100$ (in Q4.12 = 1100.1100 0000 0000)
	sum_2 $carry_2$	0110.1011 1000 0000 1000.1000 0000 0000	
Step 3:	$WS_2 = sum_2 \ll 2$ $WC_2 = carry_2 \ll 2$ $+ F_3 = 4S_2 - 4K_3$ $= b_0b_1b_2.b_3b_41100$	1010.1110 0000 0000 0010.0000 0000 0000 0011.0111 0000 0000	$W_{msbs\ 2} = 1010.111 + 0010.000 = 1100.111$, so $s_3 = -2$ $K_3 = 0000.0000 0100 0000$, $S_2 = 0.1110$ $C_3 = 1111.1111 1100 0000$, $SM_2 = 0.1101$ $F_3 = 011.011100$ (in Q4.12 = 0011.0111 00 0000 0000)
	sum_3 $carry_3$	1011.1001 0000 0000 0100.1100 0000 0000	
Step 4:	$WS_3 = sum_3 \ll 2$ $WC_3 = carry_3 \ll 2$ $+ F_4 = -2S_3 - K_4$ $= \overline{a_0} \overline{a_1} \overline{a_2} \dots \overline{a_6}111$	1110.0100 0000 0000 0011.0000 0000 0000 1110.0100 1111 0000	$W_{msbs\ 3} = 1110.010 + 0011.000 = 0001.010$, so $s_4 = 1$ $K_4 = 0000.0000 0001 0000$, $S_3 = 0.1101\ 10$ $C_4 = 1111.1111 1111 0000$, $SM_3 = 0.1101\ 01$ $F_4 = 10.01001111$ (in Q4.12 = 1110.0100 1111 0000)
	sum_4 $carry_4$	0011.0000 1111 0000 1100.1000 0000 0000	
Step 5:	$WS_4 = sum_4 \ll 2$ $WC_4 = carry_4 \ll 2$ $+ F_5 = 2S_4 - K_5$ $= b_0b_1.b_2\dots b_8111$	1100.0011 1100 0000 0010.0000 0000 0000 0001.1011 0001 1100	$W_{msbs\ 4} = 1100.001 + 0010.000 = 1110.001$, so $s_5 = -1$ $K_5 = 0000.0000 0000 0100$, $S_4 = 0.1101\ 1001$ $C_5 = 1111.1111 1111 1100$, $SM_4 = 0.1101\ 1000$ $F_5 = 01.1011000111$ (in Q4.12 = 0001.1011 0001 1100)
	sum_5 $carry_5$	1111.1000 1101 1100 0000.0110 0000 0000	
Step 6:	$WS_5 = sum_5 \ll 2$ $WC_5 = carry_5 \ll 2$ $+ F_6 = 0$	1110.0011 0111 0000 0001.1000 0000 0000 0000.0000 0000 0000	$W_{msbs\ 5} = 1110.001 + 0001.100 = 1111.101$, so $s_6 = 0$ $K_6 = 0000.0000 0000 0001$, $S_5 = 0.1101\ 1000\ 11$ $C_6 = 1111.1111 1111 1111$, $SM_5 = 0.1101\ 1000\ 10$
	sum_6 $carry_6$	1111.1011 0111 0000 0000.0000 0000 0000	Terminate: $n = b/r = 12/2 = 6$; # steps = $n = 6$
Result:	S	0001.1011 0001 1000	$= s_0 \times 4^0 + s_1 \times 4^{-1} + s_2 \times 4^{-2} + s_3 \times 4^{-3} + s_4 \times 4^{-4} + s_5 \times 4^{-5} + s_6 \times 4^{-6}$ $= (1)4^0 + (-1)4^{-1} + (2)4^{-2} + (-2)4^{-3} + (1)4^{-4} + (-1)4^{-5} + (0)4^{-6}$ $= 867/1024$, exp = 3: $(0.1101\ 1000\ 11 \times 2^3) = 867/128$ Then $\ll 1$ to normalize: $1.1011\ 0001\ 1000 \times 2^2 = 6.773\dots$

Fig. F.31 Radix-4 square root example.

F.2.10.11 Radix-4 Implementation

The radix-4 square root implementation resembles Fig. F.30 with the following changes:

- ▶ SSLC operates on the 8 msbs of WS and WC in Q4.4 form as shown in Fig. F.15 using the selection ranges from Table F.9.
- ▶ FSEL generates and selects from 5 possibilities. These possibilities depend on S_j and K_j rather than being constant divisor multiples.
- ▶ The shifters shift by 2 bits instead of 1.

F.2.10.12 Floating-Point Square Root Postprocessing

Floating-point square root shares the same postprocessing hardware as other floating-point operations described in Section 16.2.6. It produces a properly rounded result $\{S_s, S_e, S_f\}$. The result is never subnormal. In brief:

Sign

The sign of the result S_s is negative if $X = -0$ and positive otherwise.

Exponent

To compute the exponent, first calculate the unbiased exponent adjusting for the bias and any leading zeros in subnormal numbers:

$$X_e^{\text{unbiased}} = X_e - \text{bias} - \text{LeadingZeros} \quad (\text{F.56})$$

Now we need to shift the significand of X right by one or two so that it is in the range $[1/4, 1)$ and has an even exponent. If X_e^{unbiased} is odd, shift right by one; otherwise shift right by two. Increment the exponent by the shift amount. Take the square root of the significand to find $S \in [1/2, 1)$. Divide the incremented exponent by 2. Now normalize the result by shifting left by 1 and decrementing the exponent by 1. Add the bias back to find the biased exponent of the result.

In summary, if the unbiased exponent is odd, write it as $X_e^{\text{unbiased}} = 2e + 1$. Add 1, divide by 2, subtract 1, and add the bias to get $S_e = e + \text{bias}$. If the unbiased exponent is even, write it as $X_e^{\text{unbiased}} = 2e$. Add 2, divide by 2, subtract 1, and add the bias to again get $S_e = e + \text{bias}$. In both cases, the exponent can be computed with a simple right shift that divides by two and discards the least significant bit.

$$S_e = (X_e^{\text{unbiased}} \gg 1) + \text{bias} \quad (\text{F.57})$$

Fraction

To produce the fraction of the result, drop the implicit leading 1 from the normalized significand. Because the digits were placed in the most significant positions, no extra shifting is needed in the event of early termination. Note that subtracting 1 just discards the integer bit and requires no hardware.

$$S_f = (S \ll 1) - 1 \quad (\text{F.58})$$

Example F.25 SQUARE ROOT

Compute the square root of the half-precision number $0x4739 = 1.1100\ 1110\ 01 \times 2^{-6} = 1849/65536$.

Solution: $X_s = 0$. $X_e = 9$. $X_f = 1100\ 1110\ 01$. The unbiased exponent is $X_e^{\text{unbiased}} = 9 - 15 - 0 = -6$. This is even, so X is the significand, $1.X_f$, shifted right by two: $0.0111\ 0011\ 1001 = 1849/4096$. According to [Example F.23](#), the square root is $S = 0.1010\ 1100\ 0000 = 43/64$. The result is exact, so no rounding is required. S is shifted left by 1 for normalization, and the leading 1 is discarded to obtain the fraction: $S_f = 0101\ 1000\ 00$. $S_e = X_e^{\text{unbiased}} >> 1 + \text{bias} = -3 + 15 = 12$. The sign S_s is positive. Hence, $\text{Result} = \{S_s, S_e, S_f\} = 0\ 01100\ 0101\ 1000\ 00 = 0x3160 = 43/256$.

Result is rounded, and special cases and flags are handled according to [Table F.35](#). Underflow and overflow never occur.

F.2.11 Unified Divide and Square Root

With small modifications, integer divide, floating-point divide, and square root can all share the same datapath. The preprocessing shifters are used both to prescale integer operands and to prenormalize denormalized floating-point operands. The digit selection table is shared by division and square root. The addend generation differs for divide ($q_j D$) and square root (F_j). UOTFC places the result digits in the most significant bits, as required for square root. [Fig. F.32](#) lists the unified algorithm.

This unified divide/square root algorithm described in this section glosses over some implementation details, especially of fixed-point representation. See the Verilog in `src/fpu/fdivsqrt` for a bit-accurate design.

F.2.11.1 Preprocessing

The unified `divsqrt` hardware expects positive inputs X and D normalized to $[1, 2)$. For signed integer operations, it takes the absolute value. It also truncates to 32 bits for W64 integer instructions. For floating-point operations, it prepends the implicit leading 1 for normal numbers or 0 for subnorms. It then uses an LZC and shifter to normalize integers and subnorms. Square root right shifts X right by 1 or 2 into the range $[1/4, 1)$ with an even exponent. The preprocessor also calculates the sign of the result U_s based on the signs of the inputs and the type of operation.

F.2.11.2 Digit Selection

Unified digit selection USLC uses the radix-2 or radix-4 selection logic from [Equation F.51](#) or [Equation F.52](#). Replace s_j with u_j in [Equation F.51](#).

F.2.11.3 Addend Generation

Addend generation uses either $q_j D$ for division or F_j for square root. The FSEL functions are given in [Table F.16](#) for radix-2 and [Table F.17](#) for radix-4.

Inputs:

X, Y Floating-point inputs
 X_e Biased exponent field of X , or 1 if subnorm
 X_f Fraction field of X
 X_s Sign field of X
 A, B Integer inputs
 Int 1: Integer div or rem; 0: FP div or square root
 $Sqrt$ FP square root operation
 REM Integer rem operation
 $QUOT$ Quotient
 $Signed$ Integer div/rem of signed two's complement numbers
 $W64$ RV64 W-type instruction operating on lower 32 bit-word

Outputs:

$Result$ Integer rem or quotient, or FP quotient or square root
 $Flags$ FP flags

Preprocess:**Select between Integer and Floating-Point**

```

if (Int)
  if (W64) truncate  $A$  and  $B$  to 32 bits; sign-extend if  $SignedDiv$ 
  if (Signed)
     $A_s$  = sign bit of  $A$             $B_s$  = sign bit of  $B$ 
    if  $A_s$   $IFX = -A$  else  $IFX = A$    if  $B_s$   $IFD = -B$  else  $IFD = B$ 
     $U_s = A_s \oplus (B_s \& \overline{REM})$ 
  else
     $IFX = A$             $IFD = B$             $U_s = 0$ 

else # Floating-point
  if ( $X$  normalized)    $IFX = 1.X_f$    else  $IFX = 0.X_f$ 
  if ( $Y$  normalized)    $IFD = 1.Y_f$    else  $IFD = 0.Y_f$ 
   $U_s = X_s \oplus (D_s \& \overline{Sqrt})$ 

```

Normalization shift for integers and subnorms

$\ell = \text{LZC}(IFX)$ $m = \text{LZC}(IFD)$
 $X = IFX \ll \ell$ $D = IFD \ll m$

Determine number of steps and preshift amount

```

if (Int)                                     # Integer operation
   $p = \max(m - \ell, 0)$                        # Number of fraction bits
   $n = k \left\lceil \frac{r+p}{rk} \right\rceil - 1$            #  $n + 1$  digits (0 to  $n$ ).
   $rightshiftx = rk - 1 - ((r + p - 1) \bmod rk)$  #  $rightshiftx = 0$  if  $rk = 1$ 
else                                         # FP operation
   $n = k \left\lceil \frac{N_f + 2 + r}{rk} \right\rceil - 1$ 
  if (Sqrt)
     $rightshiftx = (X_e \text{ normalized and biased } X_e \text{ is odd}) \oplus (\ell \text{ is odd}) ? 1 : 2$ 
     $U_e = (X_e - \ell - bias)/2 + bias$ 
  else
     $rightshiftx = 0$ 
     $U_e = (X_e - \ell) - (Y_e - m) + bias$ 

```

Fig. F.32 Unified divide/square root algorithm.

Special Cases (terminate if detected):

```

if (Int)                                     # Special int division
  if (B == 0)                                # Divide by 0
    if (REM)      Result = A                 # REM = A
    else          Result = all 1s            # QUOT = max
  else if ( $\ell > m$ )                          #  $A < B$ 
    if (REM)      Result = A                 # REM = A
    else          Result = 0                 # Q = 0
  else                                                    # FP div or sqrt
    Invalid = SignalingNaN | ((X ==  $\infty$  & Y ==  $\infty$ ) | (X == 0 & Y == 0)) &  $\overline{Sqrt}$  |
              ( $X_s$  & X != NaN & X != 0) & Sqrt
    DivZero = (Y == 0) &  $\overline{Sqrt}$ 
    if (X == NaN | (Y == NaN &  $\overline{Sqrt}$ ) | Invalid) Result = Canonical NaN
    else if (DivZero | X ==  $\infty$ )              Result =  $\infty$ 
    else if (X == 0 | (Y ==  $\infty$  &  $\overline{Sqrt}$ ))      Result = 0

```

Initialize:

```

 $U_{-1} = 0.0$ ,  $UM_{-1} = -1.0$ 
 $WC_{-1} = 0$ ,  $WS_{-1} = X \gg \text{rightshiftr}$ 
 $C_{-1} = -R$                                      # 00.000 for  $R = 4$ , 10.000 for  $R = 2$ 

```

Iterate:

```

for  $j = 0$  to  $n$ 
   $A_{j-1} = \text{ASEL}(D_{H\text{-fractional}}, U_{Hj-1}, Sqrt)$       # Table F.13
   $u_j = \text{USLC}(WC_{\text{msbs } j-1}, WS_{\text{msbs } j-1}, A_{j-1}, C_{Hj-1})$  # Equation E.51 or E.52
   $C_j = C_{j-1} \gg \gg r$ 
   $K_j = C_j \& (\overline{C_j} < < 1)$ 
  if (Sqrt)  $F_j = \text{FSEL}(U_{j-1}, UM_{j-1}, u_j, C_j)$       # Table F.17
  else       $F_j = -u_j D$ 
   $\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} + F_j\} < < r$ 
   $\{U_j, UM_j\} = \text{UOTFC}(U_{j-1}, UM_{j-1}, u_j, K_j)$       # Fig. F.34
  Terminate Early if  $WS + WC = 0$ ; for radix-2, also terminate early if  $WS + WC =$ 
   $-2D$  for division or  $2(-2S_j + K_j)$  for square root when  $s_j = -1$ 

```

Postprocess: # Check special cases concurrently with first cycle iteration

```

 $W = (WC + WS) \gg \gg r$ 
if (Int)
  if (Terminate Early)                          # Terminate early on exact result
    if ( $W = 0$ )      QUOT = U    REM = 0
    else            QUOT = UM   REM = 0
  else                                                    # Normal termination
    if ( $W < 0$ )      QUOT = UM   REM =  $W + D$ 
    else            QUOT = U    REM = W
  if (REM)      PreResult = REM    NormShift =  $m + b - (N - 1)$ 
  else          PreResult = QUOT   NormShift =  $b - rn$ 
  Result = (PreResult >> NormShift)  $\times (-1)^{U_s}$       # Shift into lower bits
  if ( $W64$ ) truncate Result to 32 bits; sign-extend if signedDiv
else # Floating-point
  if ( $W < 0$ )       $U_m = UM$ 

```

Fig. F.32 Unified divide/square root algorithm. Cont'd

```

else           $U_m = U$ 
if (Sqrt)
     $U_m = U_m \ll 1$                                 #  $U_m$  was in  $[1/2, 1)$ 
else          # Divide
    if ( $U_m < 1$ )     $U_m = U_m \ll 1$      $U_e = U_e - 1$     # Normalization shift
    if ( $U_e < 1$ )     $U_m = U_m \gg (1 - U_e)$      $U_e = 0$     # Subnormals
Round based on  $U_m$  (includes extra rounding bits). Sticky = Terminate Early
Compute Flags
Inexact = Terminate Early
Underflow / Overflow as usual
Invalid and Divide by Zero were computed in Special Cases
 $U_f$  = fractional part of  $U_m$  (drop integer bit)
Result =  $\{U_s, U_e, U_f\}$ 

```

Fig. F.32 Unified divide/square root algorithm. Cont'd

```

for  $j = \text{firststep}$  to  $n$ 
    if ( $u_j = 1$ )     $U_j = U_{j-1} \mid K_j$                  $UM_j = U_{j-1}$ 
    else if ( $u_j = 0$ )     $U_j = U_{j-1}$                  $UM_j = UM_{j-1} \mid K_j$ 
    else if ( $u_j = -1$ )     $U_j = UM_{j-1} \mid K_j$          $UM_j = UM_{j-1}$ 

```

Fig. F.33 Radix-2 UOTFC.

```

for  $j = \text{firststep}$  to  $n$ 
    if ( $u_j = 2$ )     $U_j = U_{j-1} \mid (K_j \ll 1)$          $UM_j = U_{j-1} \mid K_j$ 
    else if ( $u_j = 1$ )     $U_j = U_{j-1} \mid K_j$          $UM_j = U_{j-1}$ 
    else if ( $u_j = 0$ )     $U_j = U_{j-1}$                  $UM_j = UM_{j-1} \mid (K_j \ll 1) \mid K_j$ 
    else if ( $u_j = -1$ )     $U_j = UM_{j-1} \mid (K_i \ll 1) \mid K_i$      $UM_j = UM_{j-1} \mid (K_j \ll 1)$ 
    else if ( $u_j = -2$ )     $U_j = UM_{j-1} \mid (K_j \ll 1)$      $UM_j = UM_{j-1} \mid K_j$ 

```

Fig. F.34 Radix-4 UOTFC.

F.2.11.4 On-the-Fly Conversion

UOTFC is essentially the same as for square root and is described in Figs. F.33 and F.34.

F.2.11.5 Postprocessing

The postprocessor selects the significand U_m as U or UM based on the sign of the residual W . For integer division, it selects between the quotient and remainder. For floating-point division or square root, it computes the exponent. The result may be in the range $(0.5, 2)$, so a 1-bit normalization left shift may be required to bring the range to $[1, 2)$. The divider produces an extra guard bit to accommodate this shift.

If the floating-point quotient has an exponent of 0 or less, it is subnormal. A variable right shifter places the results in the appropriate bits of the fraction. Integer division differs from floating-point division in that it produces a variable number of result bits. The UOFTC places the bits in the most significant positions, but they must be shifted into least significant bits, possibly accounting for trailing zeros during early termination. This also involves a variable right shift, described in Fig. F.22. The unified postprocessor shares the right shifter for floating-point and integer operations. Sharing the three costly input and result variable shifters is a significant advantage of unifying integer and floating-point division.

Floating-point also rounds and generates flags according to Sections F.4.6 and F.4.7. Integer corrects the sign of the result and truncates for W64 operations.

Floating-point special cases are described in Table F.34 for division and Table F.35 for square root. Integer division and remainder special cases were described in Section 15.2.2.

F.2.11.6 Bit Widths

Table F.20 summarizes the design parameters, signals, and bitwidths used in the unified design.

The unit is characterized by its radix R (2 or 4) and number of copies of the recurrence hardware k (typically 1, 2, or 4), which are part of the config file. Each copy of the recurrence hardware produces $r = \log_2 R$ -bit digits, so the unit produces rk bits per cycle.

Floating-point division takes an input with $N = N_s = N_f + 1$ bits (1 integer bit and N_f fraction bits). The result has $b = N_f + 2$ fraction bits, including the guard and round bits. The result also has one r -bit integer digit, for a total of $ResultBits = r + b$ bits. Each cycle produces rk bits, so $\left\lceil \frac{ResultBits}{rk} \right\rceil$ cycles are needed to produce all the bits. Each cycle produces k result digits, and the entire result comprises 1 integer digit and n fraction digits, so $n = k \left\lceil \frac{ResultBits}{rk} \right\rceil - 1$. The operands are sized to accept all of these fraction digits, so they must have $DIVb = nr = rk \left\lceil \frac{N_f + 2 + r}{rk} \right\rceil - r$ fraction bits.

Floating-point square root is similar, but the input could have two extra fraction bits to right shift by 1 or 2 bits into the range $[1/4, 1)$. Like floating-point division, the result has $b = N_f + 2$ fraction bits, including the guard and round bits.

Integer division is also similar, but the input has $N = XLEN$ bits. It is treated as a fixed-point number with 1 integer bit and $XLEN - 1$ fraction bits. The quotient needs 1 guard bit because it might be in the range $(1/2, 2)$ and may need a left shift by 1. The quotient is always truncated, so no rounding bit is necessary. Hence, integer division produces at most $XLEN$ fraction bits.

Table F.20 Divide/square root signal summary.

Term	Verilog Name	Format	FP Div	FP Sqrt	Int Div	Unified	Notes
Design Parameters							
N							Operand size in bits
N_f							Operands' number of fraction bits
R	<i>RADIX</i>				2 or 4		Radix
k	<i>DIVCOPIES</i>				Usually 1 – 4 depending on cycle time goal		Copies of recurrence hardware
Dependent Parameters							
r	<i>LOGR</i>				$\log_2 R$ ($r = 1$ or 2)		Bits per digit
a					$\{-a, \dots, a\}$ ($a = 1$ for radix-2; $a = 2$ for minimally redundant radix-4)		Digit range
b	<i>DIVMINb</i>		$N_f + 2$ (+2 bits is for guard and rounding)	$N_f + 2$ (+2 bits is for guard and rounding)	XLEN (XLEN – 1 fraction bits + 1 guard bit)	$\max(N_f + 2, \text{XLEN})$	Minimum number of fraction bits in result
	<i>ResultBits</i>		$r + b$	$r + b$ (Code says $N_f + 2$)	$r + p$ ($p = \max(m - \ell + 1, 0)$; Max value = $r + \text{XLEN}$)	$r + \max(N_f + 2, \text{XLEN})$	Minimum number of result bits (r -bit integer digit + b or p fraction bits)
	<i>Cycles</i>			$\left\lceil \frac{\text{ResultBits}}{rk} \right\rceil$			Cycles to compute all result bits
n	nE / nM (preprocessor)			$k \times \text{Cycles} - 1$		Depends on operation	Number of fraction digits in result
	<i>DIVb</i>			$k \left\lceil \frac{r + b}{rk} \right\rceil - 1$	$k \left\lceil \frac{r + p}{rk} \right\rceil - 1$		
				$n \times r$		$rk \left\lceil \frac{r + \max(N_f + 2, \text{XLEN})}{rk} \right\rceil - r$	Smallest number of fraction bits $\geq b$ so that total number of bits, $r + \text{DIVb}$, is a multiple of (rk)
Inputs							
A	<i>PosA</i>		U1. $(N - 1)$ [1, 2] for ordinary FP [0, 1] for subnorms	U1.0 [0, $2^N - 1$]	N bits		Unnormalized dividend or radicand
B	<i>PosB</i>		Same as A	N/A	Same as A	N bits	Unnormalized divisor
X		U1. $N - 1$	Normalized to [1, 2)	Normalized to [1/4, 1)	$ A \ll \ell$ Viewed as [1, 2)	Depends on operation	Normalized dividend or radicand
D		U1. $N - 1$	Normalized to [1, 2)	N/A	$ B \ll m$ Viewed as [1, 2)	Depends on operation	Normalized divisor
Internal Signals							
ℓ		Integer	N/A	N/A	$\text{lzc}(A)$		Leading zeros in $ A $
m		Integer	N/A	N/A	$\text{lzc}(B)$		Leading zeros in $ B $
firststep		Integer	0	1	0	Depends on operation	Starting iteration j
WC_j, WS_j		Q4. <i>DIVb</i>	$WC_{-1} = 0$ $WS_{-1} = X$	$WC_0 = 0$ $WS_0 = R(X - 1)$	$WC_{-1} = 0$ $WS_{-1} = X \gg [rk - 1 - ((m - \ell) \bmod rk)]$ $C_0 = -R$	$\{WC_j, WS_j\} = \{WC_{j-1} + WS_{j-1} + F_j\} \ll r$	Residual in carry-save redundant
C_j		Q2. <i>DIVb</i>	$C_0 = -R$	$C_0 = -1.0$	$C_0 = -R$	$C_j = C_{j-1} \ggg r$	Sliding thermometer
K_j		U1. <i>DIVb</i>			$K_j = C_j \& (\overline{C_j} \lll 1)$		Sliding 1 = $-C_j$
A_{j-1}		U0.3	3 most significant fraction bits of D	$U_{\text{msbs } j-1}$	3 most significant fraction bits of D	Depends on operation	Addend
u_j		$[-a, a]$		$u_j = \text{USLC}(WC_{\text{msbs } j-1}, WS_{\text{msbs } j-1}, A_{j-1})$			Result digit
F_j		Q4. <i>DIVb</i>	$-u_j D$	$\text{FSEL}(S_{j-1}, SM_{j-1}, u_j, C_j)$	$-u_j D$	Depends on operation	Addend
Outputs (To Postprocessor)							
U, UM		U1. <i>DIVb</i> (n fraction digits, nr fraction bits)	$U_{-1} = 0.0$ $UM_{-1} = -1.0$	$U_0 = 1.0$ $UM_0 = 0.0$	$U_{-1} = 0.0$ $UM_{-1} = -1.0$	$\{U_j, UM_j\} = \text{UOTFC}(U_{j-1}, UM_{j-1}, u_j, K_j)$	$U = \text{Quotient or square root}$ $UM = U - 1 \text{ ulp}$

Example F.26 OPERAND SIZE FOR DIVSQRT

Find the size of the divsqrt unit's operands for the rv64gc configuration, which supports single and double-precision floating-point ($N_f = 23$ and 51, respectively) and 64-bit integers. Assume a radix-4 implementation with $k = 4$.

Solution: Single and double precision have $N_f = 23$ and 51, respectively. $XLEN = 64$. Hence, $b = \max(51+2, 64) = 64$. The unit produces at least a U2.64 result with one $r = 2$ -bit integer digit and $b = 64$ fraction bits. It thus needs at least 66 total bits. It produces 8 bits per cycle, so it requires 9 cycles ($\lceil 66/8 \rceil$), producing $8 \times 9 = 72$ bits in U2.70 form. Hence, $DIVb = 70$. The residual is in Q4.70 form.

Getting the number of bits right is tricky. With sufficient analysis, it may be possible to shorten signals further.

F.2.11.7 Implementation

Fig. F.35 shows an implementation of a radix-4 unified divide/square root unit with k copies of the recurrence hardware to produce $2k$ bits per cycle.

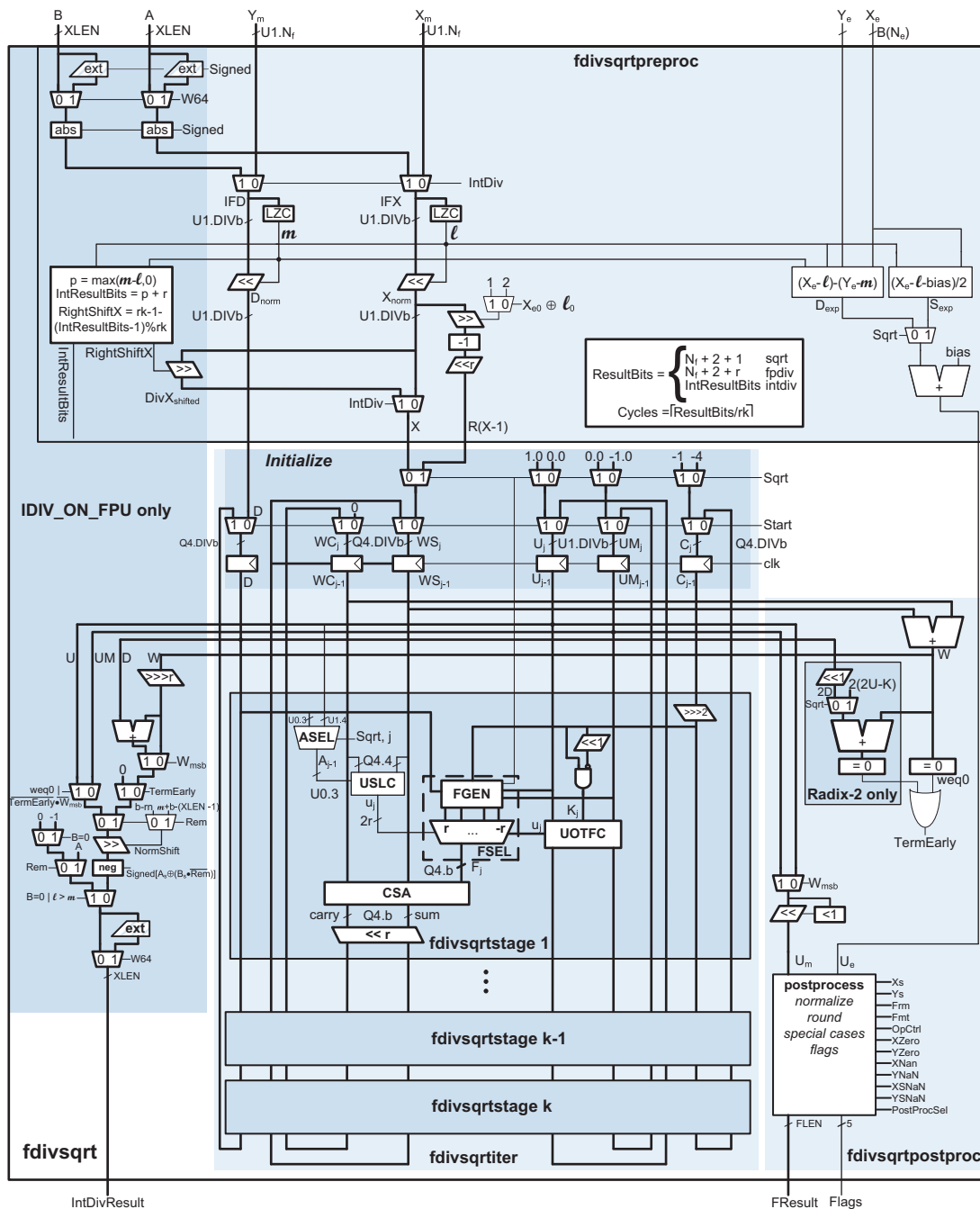
In Fig. F.35, the box at the top shows the preprocessing unit `fdivsqrtpreproc`. The column on the left in light blue shows the additional hardware needed for integer division (`IDIV_ON_FPU`). During preprocessing, integer operands may be sign extended for W64 operations and then are converted to absolute value if signed. The preprocessor chooses between integer and floating-point operations, counts the number of leading zeros (for integers and subnorms), and performs a normalization shift. Meanwhile, the unit computes the number of steps required, which also depends on the number of leading zeros in the inputs. Integer division and square root may require a small right shift. The preprocessor computes the exponent for floating-point operations.

The heart of the divsqrt unit (`fdivsqrt`) is the floating-point divide/square root iterator unit (`fdivsqrtiter`), comprising initialization muxes, registers, and k radix-R `fdivsqrtstage` blocks in the middle of Fig. F.35. Each stage performs digit selection, F generation, carry-save addition to update the residual, and a left shift. Meanwhile, the UOFTC accumulates the result digits. k may be chosen as large as possible to minimize the number of clock cycles without the divider becoming the core's critical path.

The `fdivsqrtpostproc` adds the redundant residual to compute W . If $W = 0$ (or other conditions in radix-2), the unit can terminate early. When the unit is done, it selects U or UM based on the sign of the residual and shifts a square root result back into the range $[1, 2)$. It then uses the FPU's postprocessor for normalization, rounding, special case handling, and flag generation.

The postprocessor also has an integer-specific column on the left responsible for selecting the appropriate quotient or remainder, normalizing the result, handling integer special cases, and W64 sign extension.

The unit requires $Cycles + 2$ cycles for the entire operation: one cycle of preprocessing, $Cycles$ to compute the result U , and one cycle for

Fig. F.35 Unified `fdivsqr` unit.

postprocessing. Like the integer divide FSM from Fig. 15.21, the `fdivsqrtfsm` stalls the divide in the Execute stage while the division is in progress and then continues to the Memory stage for postprocessing. The present FSM implementation adds an extra cycle in a DONE state that may be possible to optimize away.

F.2.12 Multiplicative Algorithms for Division and Square Root

A completely different way of performing division and square root is with the *Newton-Raphson method* of approximation. The general Newton-Raphson method finds a root of $f(x)$ starting with an initial guess x_0 . At each step, it refines the guess using:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (\text{F.59})$$

The method generally converges quadratically, meaning it doubles the number of accurate bits on each step.

To compute $Q = X/D$, we apply the Newton-Raphson method to find the reciprocal $R = 1/D$ and then multiply $Q = XR$. R is the root of $f(R) = 1/R - D = 0$. Start with a lookup table to find an initial approximation R_0 , typically to 8–14 bits of accuracy. Then apply the Newton-Raphson method:

$$R_{n+1} = R_n - \frac{1/R_n - D}{-(1/R_n)^2} = R_n(2 - DR_n) \quad (\text{F.60})$$

This approximation step can be performed with two FMA operations. The reciprocal converges to single-precision accuracy in one or two steps or double-precision accuracy in two to three steps. Hence, double-precision division can be performed with five to seven FMA operations.

Similarly, to compute the reciprocal square root $A = 1/\sqrt{X}$, use the Newton-Raphson method to compute the root of $f(A) = 1/A^2 - X = 0$. Start with a look-up table initial approximation A_0 and apply:

$$A_{n+1} = A_n - \frac{1/A_n^2 - X}{-2(1/A_n)^3} = \frac{A_n}{2} (3 - XA_n^2) \quad (\text{F.61})$$

This approximation step takes three FMA operations. After A is found, the square root is $S = \sqrt{X} = AX$.

These multiplicative algorithms are tempting in an architecture such as Itanium or POWER with a high-performance FMA with a few extra bits to facilitate correct rounding [Cornea04]. Each FMA operation depends on the previous one, and a fast FMA has a many-cycle latency, so Itanium requires 24–40 cycles for various precisions of divide and square root, though the throughput for concurrent operations can be

A key to using the Newton-Raphson method is to pick a function that leads to an easy-to-compute approximation step.

The Newton-Raphson method is named for Isaac Newton, who developed the approach in 1671, and Joseph Raphson, who simplified the method and first published it in 1690.

much higher. [Muller18] discusses parallelizing the approximation step with *Goldschmidt's algorithm*, as well as rounding. Nevertheless, FMA is power hungry, and the extra bits for rounding may not be available to software-based algorithms. Most processors presently use recurrence division [Liu12].

F.3 CONVERSION

Section 16.2.4 introduces a floating-point conversion unit. Table F.21 summarizes the conversion logic from Fig. 16.14. The following subsections discuss the exponent and shift amount logic further and give conversion examples.

F.3.1 Result Exponent

The exponent of the result depends on the type of conversion and the input values. Recall that the unpacker expresses all floating-point inputs in the largest supported format. *bias* is the bias of this format (e.g., 127 for single, 1,023 for double).

Let E be the unbounded exponent of the input. For normalized floating-point numbers, E is the unbiased exponent. For subnormals, E is the exponent if the subnormal value were normalized with an exponent less than the minimum representable value. For integers, E is the exponent of the integer represented in normalized form. For example, 1.101×2^3 has $E = 3$. $0.001 \times 2^{-15} = 1.0 \times 2^{-18}$ has $E = -18$. $101 = 1.01 \times 2^2$ has $E = 2$.

F.3.1.1 Float-to-Float

For float-to-float conversion, the exponent bias needs to be adjusted. Let $ExpIn$ be the biased exponent of the unpacked input. For a normalized input, the unbounded exponent is $E = ExpIn - bias$. For a subnormal input, the input fraction must be normalized by *LeadingZeros*, so $E = ExpIn - bias - LeadingZeros$. *LeadingZeros* = 0 for normalized inputs, so this expression works for all inputs.

Let $Bias_{out}$ be the bias of the desired output format. Thus, the output exponent is $C_e = E + Bias_{out}$.

For example, consider a system supporting up to double-precision (*bias* = 1,023). Tables F.22–F.24 list the biased exponent for the result of various conversions.

F.3.1.2 Float-to-Int

Integer results don't have an exponent, but left shift the floating-point significand by the floating-point unbounded exponent E if the exponent is non-negative.

Table F.21 Convert logic.

Signal	Format	Logic	Comments
Inputs			
A	$Q/U(XLEN)$		Integer input (signed or unsigned)
X_m	$U1.N_f$		Floating-point input significand
X_e	$B(N_e)$		Floating-point input exponent
X_s	$U1$		Floating-point input sign
Internal Signals			
A_s	$U1$	$A_{63/31}$ & Signed	Sign bit of A for signed int conversions
$PosA$	$U(XLEN)$	$ A $	Absolute value of A
$TrimA$	$U(XLEN)$	$PosA$ & (32 or 64 1s)	Trim off upper 32 bits for 32-bit integer conversions in RV64
$LzcInFull$	$\max(N_f + 1, XLEN)$	$TrimA$ X_m	int-to-float float-to-int/float Number being converted
$ExpIn$	$B(N_e)$	$bias + XLEN - 1$ X_e	int-to-float float-to-int/float Effective biased input exponent
$Bias_{out}$	$U(N_e)$	1 Bias of output format	float-to-int int/float-to-float $ Bias $ of output format (15 for half, 127 for single, etc.)
$LeadingZeros$	$\log_2(LzcInFull \text{ size})$	Count Leading Zeros($LzcInFull$)	Leading zeros in $LzcInFull$
Outputs			
E	$B(N_e + 1)$	$ExpIn - bias - LeadingZeros$	Unbounded exponent of input
C_e	$B(N_e + 1)$	$E + Bias_{out}$	Exponent of converted result (for cvt to float) or amount to shift (for cvt to int)
C_s	$U1$	X_s A_s	float-to-float/int int-to-float Sign of output
$LzcIn$	$\max(N_f, XLEN - 1)$	Drop msb from $LZCInFull$	Number being converted
$ShiftAmt$	$\log_2(ShiftIn \text{ size})$	$\max(C_e, 0)$ $C_e + N_f - 1$ $LeadingZeros$	float-to-int ResSubnormUf otherwise To normalization shifter
$IntZero$	1	$TrimA = 0$	Trimmed integer is 0
$ResSubnormUf$	1	$(C_e \leq 0) \& \overline{X_{zero}}$ 0	float-to-float/int int-to-float Result is subnormal or underflows

Table F.22 Float-to-float result exponent examples.

Instruction	Conversion Type	Input	Input Unbiased Exponent	$ExpIn$	$LeadingZeros$	E	$Bias_{out}$	C_e
fcvt.d.s	float-to-double	1.101×2^3	3	1,026	0	3	1,023	1,026
fcvt.h.s	float-to-half	1.010×2^{-2}	-2	1,021	0	-2	15	13
fcvt.s.h	half-to-float	0.001×2^{-14}	-14	1,008	3	-17	127	110

Table F.23 Float-to-int result exponent examples.

Instruction	Conversion Type	Input	Input Unbiased Exponent	$ExpIn$	$LeadingZeros$	E	$Bias_{out}$	C_e
fcvt.l.s	float-to-int64	1.101×2^3	3	1,026	0	3	1	4
fcvt.w.d	double-to-int32	1.010×2^{-2}	-2	1,021	0	-2	1	-1
fcvt.l.h	half-to-int64	0.001×2^{-14}	-14	1,008	3	-17	1	-16

Table F.24 Int-to-float result exponent examples.

Instruction	Conversion Type	Input	$LeadingZeros$	E	$Bias_{out}$	C_e
fcvt.s.l	int64-to-float	00...001101	60	3	127	130
fcvt.d.w	int32-to-double	00...0101	29 or 61	2	1,023	1,025

For reasons described in [Section F.4.1](#), the shifter first right shifts by 1 before left shifting by $E + 1$. Hence, for conversions to int, define $Bias_{out} = 1$ and use a shift amount $C_e = E + Bias_{out}$.

F.3.1.3 Int-to-Float

An integer has an effective exponent of $E = XLEN - LeadingZeros - 1$. Hence, when converting an integer to a floating-point representation, the exponent of the result is $C_e = E + Bias_{out}$.

By defining $ExpIn = bias + XLEN - 1$ for conversion from integer format, we can still use compute $E = ExpIn - bias - LeadingZeros$. Note that $ExpIn$ is a design-time constant for integers.

F.3.1.4 Summary

In all cases, $E = ExpIn - bias - LeadingZeros$ and $C_e = E + Bias_{out}$. $ExpIn = bias + XLEN - 1$ for integer inputs, or the unpacked biased exponent for floating-point inputs. $Bias_{out} = 1$ for integer results, or the floating-point bias for floating-point results.

If $C_e \leq 0$, the result is subnormal or underflowed. The *ResSubnormUF* flag indicates this condition, which is passed to the postprocessor to adjust the normalization shift.

F.3.2 Shift Amount Selection

The postprocessor `cvtshiftcalc` block computes *CvtShiftIn* by preshifting the integer or float-point significand right by padding with leading zeros as described in [Section F.4.1](#). Then `normshift` left-shifts *CvtShiftIn* by *ShiftAmt* as described in [Section F.4.3](#). This section analyzes how the floating-point convert unit (`fcvt`) selects *ShiftAmt*. As with the result exponent, we consider the three types of conversions.

F.3.2.1 Float-to-Float

Float-to-float conversion only requires shifting when the input or output is subnormal. If the input is subnormal, it must be left shifted by $ShiftAmt = LeadingZeros$. If the output is subnormal ($C_e < 0$), the normalized result is preshifted right by prepending $N_f - 1$ leading zeros and then is left shifted by $ShiftAmt = C_e + N_f - 1$. This gives a net left shift by C_e , but C_e is negative, so it is an effective right shift by $|C_e|$ to put the output in subnormal form. If both input and output are normalized, $ShiftAmt = 0$.

F.3.2.2 Float-to-Int

Float-to-int conversion effectively preshifts right by 1 by prepending leading zeros, then shifts left by $ShiftAmt = \max(C_e, 0)$. Thus C_e should be $E + 1$, as was described in [Section F.3.1.2](#). If $C_e \leq 0$, the integer part is 0, and the remaining bits are used for rounding.

F.3.2.3 Int-to-Float

Integers are shifted left by $ShiftAmt = LeadingZeros$ to produce a normalized significand with a leading 1.

F.3.2.4 Summary

In summary, different *ShiftAmt* values are used for different cases, as given in [Table F.21](#).

F.3.3 Examples

The following examples illustrate conversions. Assume they are executed on an FPU supporting half and single precision operands, so $bias = 127$.

Example F.27 SINGLE-TO-HALF

Convert the single-precision value $X = C17FF00 = -1.1111\ 1111\ 111 \times 2^3 = -15.9961$ to half-precision. Use round-to-nearest (RNE).

Solution: X has a biased exponent of $ExpIn = X_e = 130$. $LeadingZeros = 0$. $E = ExpIn - bias - LeadingZeros = 3$. $Bias_{out} = 15$ is the bias of the half-precision output format. The converted biased exponent is $C_e = E + Bias_{out} = 3 + 15 = 18$. The input and output are normalized so $ShiftAmt = 0$. X has significand $1.1111\ 1111\ 111$. This has no leading zeros but more than 10 fraction bits, so it is rounded to $10.0000\ 0000\ 0$ for half-precision. Renormalizing, the half-precision value has a negative sign, biased exponent of 19, and significand of $1.0000\ 0000\ 00$: $-1.0000\ 0000\ 00 \times 2^4 = 0xCC00 = -16$. This is the closest representable number in half-precision.

Example F.28 HALF-TO-SINGLE

Convert the half-precision subnormal value $X = 0060 = 0.00011 \times 2^{-14} = 5.72205 \times 10^{-6}$ to single-precision.

Solution: X has a biased exponent of $ExpIn = X_e = 113$ from the unpacker. $LeadingZeros = 4$. $E = ExpIn - bias - LeadingZeros = -18$. The converted biased exponent is $C_e = E + Bias_{out} = -18 + 127 = 109$. The input is subnormal so $ShiftAmt = LeadingZeros = 4$. The significand 0.00011 is normalized to 1.10 with a left shift by $ShiftAmt$. Hence, the single-precision value has a positive sign, biased exponent of 109, and significand of 1.10 : $1.10 \times 2^{-18} = 0x36C00000$. This is an exact representation of the half-precision value, and is now normalized.

Example F.29 HALF-TO-INT

Convert the half-precision value $X = C8C0 = -1.0011 \times 2^3 = -9.5$ to integer. Use round-to-zero.

Solution: X has a biased exponent of $ExpIn = X_e = 130$. $LeadingZeros = 0$. $E = ExpIn - bias - LeadingZeros = 3$. $C_e = E + 1 = 4$. X needs to be shifted to fixed-point form. The shift amount $ShiftAmt$ is $C_e = 4$. The significand of 1.0011 is shifted right by 1 and then left by $ShiftAmt$ to give 1001.1 . This is rounded down to $1001 = 9$. The input is negative, so the integer value is -9 . This is the closest suitably rounded integer.

Example F.30 INT-TO-HALF

Convert the 32-bit integer value $A = -5 = 111\dots1111011$ to half-precision floating-point.

Solution: A is negative, so take the two's complement to get the absolute value 0000 0000 0000 0000 0000 0101. A needs to be shifted left to have a leading 1. The shift amount $ShiftAmt$ is the number of leading zeros $LeadingZeros = 29$, giving a shifted value 1010 0000 0000 0000 0000 0000. The upper bits of this value form the significand 1.0100 0000 00. $ExpIn$ is $bias + XLEN - 1 = 127 + 32 - 1 = 158$ for integer inputs. Thus, $E = ExpIn - bias - LeadingZeros = 158 - 127 - 29 = 2$, and $C_e = E + bias_{out} = 2 + 15 = 17$. The sign is negative. Hence the half-precision value is $-1.01 \times 2^2 = -5.0 = 0xC500$, which is exact.

F.4 POSTPROCESSING

Section 16.2.6 introduced the postprocessing unit, and Fig. 16.16 shows a block diagram. This section describes each block.

F.4.1 Shift Calculation

The normalization shifter needs a fixed number of input bits, but each execution unit produces a different number of significand output bits. The shift calculation blocks place each execution unit output in suitable bitfields; then an initial result mux chooses the output for the active execution unit.

Meanwhile, the shift calculation block determines how much to shift. This depends on the number of leading zeros in the significand and on whether the result will be subnormal. If the biased exponent of the result is 0 or less, the result is subnormal, and the normalization shifter must shift it such that the biased exponent becomes one. When the datapath handles multiple precisions, the shift amount depends on the bias of the result precision.

The calculation is different for inputs from the `fma`, `divsqrt`, and `fcvt` modules. Each calculation is described in the next sections. The shift logic is complicated because it shares the same shifter for all three functional units and because the LZA has some uncertainty.

The shift calculation ultimately produces `ShiftIn` and `ShiftAmt` signals to drive the normalization shifter. Define the bit width of `ShiftIn` and other normalization shift buses to be $N_m = \max(3N_f + 6, b + N_f + 2)$, the greater of the widths from the FMA and divider. The normalization shifter places the answer in the most significant bits.

F.4.1.1 FMA

The FMA's shift calculation module (`fmashiftcalc`) receives the significand and exponent of the sum from the FMA adder along with the

Table F.25 FMA shift calculation in `fmashiftcalc`.

Signal	Format	Logic	Comments
Inputs			
S_m	$U(N_f+3).(2N_f+1)$		Sum significand from FMA
S_e	$B(N_e+2)$		Sum exponent from FMA
S_{cnt}	$U(N_e+2)$		Sum leading zeros from FMA
Outputs			
$NormSumExp$	$B(N_e+2)$	$S_e - (S_{\text{cnt}} + 1 - (N_f + 3)) + Bias_{\text{out}} - bias$	Normalized exponent before LZA correction
S_{zero}	Boolean	$S_m = 0$	FMA sum is zero
$PreResultSubnorm$	Boolean	$-(N_f + 1 \leq NormSumExp \leq 0 \ \& \ S_{\text{zero}})$	Result is subnormal (before LZA correction)
$ShiftIn_{\text{fma}}$	$U2.(N_m - 2)$	$\{00, S_m\}$	To normalization shifter; note that the total width is N_m
$ShiftAmt_{\text{fma}}$	$\log_2(ShiftIn \text{ size})$	$S_e + N_f + 2$ $S_{\text{cnt}} + 1$	$PreResultSubnorm$ To normalization shifter Otherwise

estimated number of leading digits (S_{cnt}) from the FMA's LZA. Recall that S_{cnt} is either accurate or too high by 1. It computes the normalization shift inputs, the normalized exponent, and some special cases according to Table F.25.

F.4.1.2 Divide/Square Root

The unified divide/square root (`divsqrt`) unit's shift calculation module (`divshiftcalc`) determines its shift amounts. Excluding integer division and special cases such as 0 or infinity, `divsqrt` produces a result $Q_m \in (1/2, 2)$. If the answer is normalized and less than 1, the result may need a normalization left shift by 1 to bring it into the range $[1, 2)$. If the answer is subnormal, it is preshifted right and then shifted back left into the appropriate position using the normalization shifter. These two cases can be combined by always preshifting right and then shifting left appropriately. Table F.26 summarizes this shift logic.

Integer division postprocessing is handled on separate hardware as described in Section F.2.11.5 and does not require normalization shifting or other floating-point postprocessing.

Table F.26 Divide/square root shift calculation in `divshiftcalc`.

Signal	Format	Logic	Comments
Inputs			
Q_m	U1.b		Significand from <code>divsqrt</code>
Q_e	$B(N_e + 2)$		Exponent from <code>divsqrt</code>
Outputs			
$ResSubnorm_{div}$	Boolean	$Q_e \leq 0$	Result is subnormal
$SubnormShiftPos_{div}$	Boolean	$Q_e + N_f \geq 0$	Subnormal result is nonzero
$ShiftIn_{div}$	$U2.(N_m - 2)$	$Q_m \gg (N_f + 1)$ with zero padding	Quotient preshifted right
$ShiftAmt_{div}$	$\log_2(\text{ShiftIn size})$	$\max(Q_e + N_f, 0)$ N_f	$ResSubnorm_{div}$ Otherwise Amount to shift left

F.4.1.3 Conversion

As discussed in [Section F.3.2](#), the floating-point convert (`fcvt`) unit's shift calculations, which occur in its `cvtshifcalc` module, arranges the shifter input appropriately for the different types of conversions. [Table F.27](#) describes how the convert unit's input is shifted.

F.4.2 Initial Result Mux

As shown in [Table F.28](#), the initial result mux chooses `ShiftIn`, `ShiftAmt`, and `NormExp` from one of the three execution units depending on the current operation.

F.4.3 Normalization Shift

The normalization shifter simply shifts `ShiftIn` left by `ShiftAmt`, as given in [Table F.29](#).

F.4.4 Correction Shift

The correction shifter (`shiftcorrection`) performs a possible 1-bit shift after normalization to compensate for LZA error.

The LZA estimated the number of leading digits S_{cnt} but may have overestimated by 1. The LZA error is detected by looking at the most significant bit in the 2-bit integer portion of the `Shifted` output. If it is

Table F.27 Convert shift calculations.

Signal	Format	Logic	Comments
Inputs			
X_m	$U1.N_f$		Input significand
C_e	$B(N_e + 2)$		Exponent from convert unit
$LzcIn$	$\max(N_f, \text{XLEN} - 1)$		Number being converted
$ResSubnormUf$	Boolean		$C_e \leq 0$; convert underflows or subnorm
Outputs			
$ResUf_{cvt}$	Boolean	$(C_e < -N_f) \overline{X_{zero}} \overline{IntToFp}$	Result underflows
$ShiftIn_{cvt}$	$U2.(N_m - 2)$	$\left\{ \begin{array}{l} X_{mN_f} \ \& \ \overline{Ce_{N_e}}, X_{mN_f-1} \mid Ce_{N_e} \ \& \ X_{mN_f}, X_{mN_f-2:0} \end{array} \right\}$ in lsbs X_m in lsbs $LzcIn$ in msbs	float-to-int $ResSubnormUf$ Otherwise Input suitably arranged for normalization shift

Table F.28 Initial result mux.

Signal	Format	FMA	divsqrt	cvt	Comments
$ShiftIn$	$U2.(N_m - 2)$	$ShiftIn_{fma}$	$ShiftIn_{div}$	$ShiftIn_{cvt}$	Significand to shift
$ShiftAmt$	$\log_2(N_m)$	$ShiftAmt_{fma}$	$ShiftAmt_{div}$	$ShiftAmt_{cvt}$	Amount to shift
$NormExp$	$B(N_e + 2)$	$NormSumExp$	$NormDivExp$	C_e	Exponent after shifting

Table F.29 Normalization shift logic.

Signal	Format	Logic	Comments
Inputs			
$ShiftIn$	$U(N_m)$		Result to shift
$ShiftAmt$	$\log_2(N_m)$		Amount to shift
Outputs			
$Shifted$	$U2.(N_m - 2)$	$ShiftIn \ll ShiftAmt$	Shifted result

Table F.30 Correction shift logic.

Signal	Format	Logic	Comments
Inputs			
<i>Shifted</i>	$U2.(N_m - 2)$		Significand from normalization shifter
<i>NormExp</i>	$B(N_e + 2)$		Exponent from initial result mux
Internal Signals			
<i>Shift1</i>	Boolean	<i>Shifted</i> _{msb}	LZA error = 1
Outputs			
M_f	$U0.CorrShiftSz$	Fractional part of <i>Shifted</i> >> 1 Fractional part of <i>Shifted</i>	<i>Shift1</i> Otherwise Fraction after normalization
M_e	$B(N_e + 2)$	0 <i>NormExp</i> + 1 <i>NormExp</i>	Result is 0 or denorm <i>Shift1</i> Otherwise Exponent after normalization

asserted, the correction shifter performs a 1-bit right shift. Subnormal results have no integer portion, but their normalization shift depends only on the exponent, not the LZA, so they have no LZA error. Table F.30 summarizes the correction shift logic.

When the most significant bit of the integer part of *Shifted* is 1, the LZA error is 1; the significand is right shifted by 1 and the exponent increased by 1. Otherwise, *Shifted* is already normalized, and no correction is required.

$CorrShiftSz = \max(3N_f + 4, b + N_f + 1)$ describes the number of bits in the normalized fraction M_f after the correction shift. It throws away the two integer bits from the FMA.

F.4.5 Sign Logic

The sign logic, shown in Table F.31, is performed by the `roundsign` and `resultsign` modules.

After this section was written, the correction shifter was modified to fix a `fcvt` bug in certain configurations (PR #784). The RTL (SystemVerilog code) contains the latest logic. There is likely still room for optimization.

Table F.31 Sign logic.

Signal Format Logic				Comments
Internal Signals				
Q_s	Q1	$X_s \oplus (Y_s \text{ \& Divide})$		Sign for division or square root
$Zero_s$	Q1	RM P_s	$(P_s \oplus A_s) \overline{Mult} \overline{(R' T')}$ Otherwise	Sign of zero result. • The operation is a multiplication (with no add/subtract) when $Mult = 1$. • The operation is an effective subtraction when the sign of the product differs from the sign of the addend and the operation is not a multiplication only: $(P_s \oplus A_s) \overline{Mult}$ • A zero result is exact when the round (R'), and sticky (T') bits are also both zero: $\overline{(R' T')}$ • For an effective subtraction with an exact zero result, the sign is positive (0) for rounding modes other than RM and negative (1) for RM. • Otherwise (multiplications and effective additions), the sign of the zero result is the sign of the product P_s.
Inf_s	Q1	A_s P_s	Z_{inf} Otherwise	Sign of infinity from FMA
Outputs				
M_s	Q1	S_s C_s Q_s	FMA Convert Divide/sqrt	Sign of normalized result
R_s	Q1	Inf_s $Zero_s$ M_s	$InfIn \text{ \& FmaOp}$ $S_{zero} \text{ \& FmaOp}$ Otherwise	Sign of final result

When rounding up a number with all 1s in the fraction, the fraction becomes all 0s, and the exponent increases by 1. The rounder handles this gracefully by adding an ulp to a bus comprising the exponent and fraction (see R_e^{full}). A carry-out of the fraction automatically bumps up the exponent.

F.4.6 Rounding

Recall from [Section 16.1.3](#) that rounding chooses between the two nearest values of the fraction. The truncated fraction is called *TRUNC*, and the next larger fraction is called *RND*. *RND* is *TRUNC* plus an ulp. Depending on the rounding mode and the rounding bits L' , R' , and T' , rounding chooses either *TRUNC* or *RND* as described in [Table 16.4](#). [Table F.32](#) describes internal signals and outputs for the rounding logic.

Table F.32 Rounding logic.

Signal	Format	Logic	Comments	
Internal Signals				
R	Boolean	$M_f[\text{CorrShiftSz} - \text{XLEN} - 2]$ $M_f[\text{CorrShiftSz} - N_f - 2]$	Convert FP to int Otherwise	Round bit
L	Boolean	$M_f[\text{CorrShiftSz} - \text{XLEN}]$ $M_f[\text{CorrShiftSz} - N_f]$	Convert FP to int Otherwise	lsb of result
$NormT$	Boolean	OR(lsb discarded by normalization)		Sticky bits discarded during normalization
OpT	Boolean	$FmaA_{sticky} \mid (S_e < 0)$ $CvtResUf$ $DivSticky$	$FmaOp$ $CvtOp$ $DivOp$	Sticky bits discarded during an operation
T	Boolean	$NormT \mid OpT$		Sticky bit
Outputs				
G	Boolean	$M_f[\text{CorrShiftSz} - \text{XLEN} - 1]$ $M_f[\text{CorrShiftSz} - N_f - 1]$	Convert FP to int Otherwise	Guard bit
$Plus1$	Boolean	Round to $RND = TRUNC + \text{ulp}$ based on Table 16.4 with $L' = L, R' = G, T' = T \mid R$		Round up?
$UfPlus1$	Boolean	Same calculation as $Plus1$ using $L' = G, R' = R, T' = T$		Round intermediate result up for underflow logic
R_e^{full} R_f	$B(N_e+2)$ $U0.N_f$	$M_e + Plus1$ $M_f[\text{CorrShiftSz} - 1 : \text{CorrShiftSz} - N_f] + Plus1$		Perform rounding; overflowing fraction bumps up to next exponent
R_e	$B(N_e)$	Bottom N_e bits of R_e^{full}		Rounded exponent

F.4.7 Special Cases and Flags

The special case and flags blocks adjust the result and set the Invalid (and possibly Divide by Zero) flags for cases involving NaNs, infinities, and zeros. Table F.33 shows special case handling for FMA operations. In the event of multiple rows being satisfied, the highest row has priority (e.g., NaN comes before infinities). Tables F.34–F.40 show special cases for other operations. *val* represents any normalized or subnormal value.

Integer to floating-point conversion has no NaN or infinity inputs and never sets the Invalid flag. If the input is large enough, it can set the

Table F.33 FMA special case handling.

X	Y	Z	Result	Invalid Flag	Notes
At least one sNaN			NaN	1	Signaling NaN sets invalid
At least one NaN			NaN	0	
At least one ∞	∞ of opposite sign		NaN	1	$\infty - \infty$ is invalid
One 0 and one ∞	val		NaN	1	$0 \times \infty$ is invalid
At least one ∞			∞	0	

Table F.34 Divide special case handling.

X	Y	Result	Invalid Flag	Divide by Zero Flag	Notes
At least one signaling NaN		NaN	1	0	Signaling NaN sets invalid
At least one NaN		NaN	0	0	
∞	∞	NaN	1	0	∞ / ∞
0	0	NaN	1	0	$0 / 0$
∞	val	∞	0	0	$\infty / 0$
val	0	∞	0	1	Finite nonzero / 0
val	∞	0	0	0	$0 / \text{nonzero}$
0	val	0	0	0	$0 / \text{nonzero}$

Table F.35 Square root special case handling.

X	Result	Invalid Flag	Notes
Signaling NaN	NaN	1	Signaling NaN sets invalid
NaN	NaN	0	
+0	+0	0	Square root of 0
−0	−0	0	Square root of −0
Negative	NaN	1	Square root of negative number
∞	∞	0	Square root of infinity

Table F.36 Compare special case handling.^a

X	Y	Op	Result	Invalid Flag	Notes
At least one signaling NaN	any		0	1	Signaling NaN sets invalid
At least one NaN	<, ≤		0	1	flt/fle set invalid on NaN input
At least one NaN	=		0	0	

^aNote that +0 and −0 are considered equal, and −0 is not less than +0.

Table F.37 Min/max special case handling.

X	Y	Result	Invalid Flag	Notes
At least one signaling NaN		See rows below	1	Same as NaN, except signaling NaN also sets Invalid
NaN	NaN	NaN	0	
NaN	non-NaN	Y	0	Pick non-NaN input
non-NaN	NaN	X	0	Pick non-NaN input

Table F.38 Maximum and minimum integer values.

Integer Format	Destination Code	MININT	MAXINT
Unsigned 32-bit integer	wu	0	2 ³² − 1
Unsigned 64-bit integer	lu	0	2 ⁶⁴ − 1
Signed 32-bit integer	w	−2 ³¹	2 ³¹ − 1
Signed 64-bit integer	l	−2 ⁶³	2 ⁶³ − 1

Table F.39 Convert float-to-int special case handling.

X (FP)	Result (Int)	Invalid Flag
Signaling or Quiet NaN	MAXINT	1
∞ or > MAXINT	MAXINT	1
−∞ or < MININT	MININT	1
0	0	0

Table F.40 Convert float-to-float special case handling.

X	Result	Invalid Flag	Notes
Signaling NaN	NaN	1	Signaling NaN sets invalid
NaN	NaN	0	
∞	∞	0	
0	0	0	

Overflow flag and produce infinity or *MAXNUM* as the result. For example, converting 1,000,000 to a half-precision number in RN mode produces ∞ . If the input cannot be represented exactly, it sets the Inexact flag.

Floating-point to integer conversion depends on the maximum or minimum representable integer values defined in Table F.38.

F.4.7.1 Other Flags

The `flags` module also raises Overflow, Underflow, and Inexact flags as described in Section 16.1.4. However, computing the intermediate result for Overflow and Underflow is costly because it would involve a variable normalization shift and a rounding addition. The `flags` block avoids this by deducing these flags from the results before and after rounding and from the *G*, *R*, and *T* bits. This section refers to the result before rounding as the *normalized result* and the result after rounding as the *rounded result*.

These results have exponents M_e and R_e^{full} , respectively.

Overflow is computed as $R_e^{\text{full}} > e_{\text{max}} + \text{bias}$. The bias is necessary because R_e^{full} is in $B(N_e + 2)$ biased form. Note that this check occurs before the result might be rounded to *MAXNUM* by the result selection logic.

Recall that underflow depends on tininess, which is computed from the intermediate result. A tiny intermediate result can be detected from the rounded result using two cases:

Case A: The intermediate result is tiny if the rounded result is tiny. The rounded result is tiny if the final exponent $R_e^{\text{full}} \leq 0$, meaning it is smaller than the smallest normalized number.

Case B: The intermediate result is also tiny if the rounded result is a normal number, the normalized result is subnormal, and the rounding does not push the intermediate result up to the smallest normalized value. This case only occurs when the result is rounded up from the largest sub-normal number $0.1111... \times 2^{e_{\text{min}}}$ to the smallest normalized number $1.0 \times 2^{e_{\text{min}}}$. To detect this, check the final exponent $R_e^{\text{full}} = 1$ and the exponent

0.1111 1111 11GRT $\times 2^{e_{\text{min}}}$ Subnormal result before rounding

1.1111 1111 1GRT $\times 2^{e_{\text{min}}-1}$ Intermediate result before rounding

Fig. F.36 Determining if the intermediate result is tiny.

Don't mix up the round bit *R* with the rounded result exponent R_e^{full} . Also beware that the normalized result is the output of the normalization shifter, but the result could actually be subnormal with $M_e = 0$.

Table F.41 Result selection logic.

Signal	Format	Logic	Comments
Internal Signals			
<i>NormRes</i>	FLEN	$\{R_s, R_e, R_f\}$	Normal case
<i>UfRes</i>	FLEN	0 or 1 depending on truncating or rounding up; Keep sign R_s	Round to 0 or 1 if the result is smaller than the smallest subnormal number
<i>OfRes</i>	FLEN	MAXNUM or INF depending on Table 16.4; Keep sign R_s	Round to INF or MAXNUM if the result is larger than MAXNUM
Outputs			
<i>Result</i>	FLEN	<i>OfResult</i> <i>Overflow</i> <i>UfResult</i> $R_e^{\text{full}} < 0$ <i>NormRes</i> Otherwise	Final result

before rounding $M_e = 0$. Fig. F.36 shows the subnormal result before rounding, which is left shifted by 1 to form an intermediate result before rounding. As shown in Table 16.3, the rounding is done with $L' = G, R' = R, T' = T$ and is performed according to Table 16.4. The rounding direction is computed in Table F.32 as *UfPlus1*. If *UfPlus1* is asserted and $G = 1$, the intermediate result will be the smallest normalized number. Otherwise, the intermediate result is tiny. In contrast, the rounded result was computed with $L' = L, R' = G, T' = R \mid T$, which explains why the intermediate and rounded results might differ.

The following equations express this logic. The intermediate results is tiny if either Case A or Case B applies. The Underflow flag is raised if the intermediate result is tiny, and the result is Inexact.

$$\text{IntermediateUp} = G \ \& \ \text{UfPlus1} \quad (\text{F.62})$$

$$\text{Tiny} = R_e^{\text{full}} \leq 0 \mid \left(R_e^{\text{full}} = 1 \ \& \ M_e = 0 \ \& \ \overline{\text{IntermediateUp}} \right) \quad (\text{F.63})$$

$$\text{Underflow} = \text{Tiny} \ \& \ \text{Inexact} \quad (\text{F.64})$$

F.4.7.2 Result Selection

The `specialcase` module picks *Result* as described for operation-specific special cases in the earlier tables. Otherwise, it uses Table F.41 to round very large or very small results or pick the normal result.

Bibliography

Descriptor	Reference
6SN7	“6H8C (6SN7 double triode) Vostok 1980, 2022 [6SN7],” 2022. [Online]. Available: https://commons.wikimedia.org/wiki/File:6%D0%9D8%D0%A1_(6SN7_double_triode)_Vostok_1980_03.jpg .
Adve95	S. V. Adve and K. Gharachorloo, “Shared memory consistency models: a tutorial,” <i>WRL Research Report 95/7</i> , 1995.
Aharoni08	M. Aharoni, “Floating-Point Test-Suite for IEEE,” IBM Labs in Haifa, Jan. 2008. [Online]. Available: https://research.ibm.com/haifa/projects/verification/fpgen/papers/ieee-test-suite-v2.pdf .
AHB21	AMBA AHB Protocol Specification IHI0033C, ARM Ltd., 2021.
Aiken46	H. H. Aiken and G. M. Hopper, “The Automatic Sequence Controlled Calculator – I,” <i>Electrical Engineering</i> , vol.65 no.8-9, pp. 384–391, Aug. 1946.
AMBA99	<i>AMBA Specification Rev 2.0 IHI0011A</i> , ARM Ltd., 1999.
APB21	<i>AMBA APB Protocol Specification IHI0024D</i> , ARM Ltd., 2021.
AppleIle	Pratyeka, "Front view of an Apple Ile system, including computer chassis, monitor, and external 5¼" floppy disk drive," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Apple_Ile#/media/File:Ile-system.jpg .
ARM11	<i>ARM11 Processor Technical Reference Manual</i> . Texas Instruments, 2005.
Arpaci-Dusseau19	R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, <i>Operating Systems: Three Easy Pieces</i> , Arpaci-Dusseau Books, 2019.
Atkins68	D. Atkins, “Higher radix division using estimates of the divisor and partial remainders,” <i>IEEE Trans. Computers</i> , vol. C-17, no. 10, pp. 925–934, Oct. 1968.
AXI21	<i>AMBA AXI and ACE Protocol Specification IHI0022H.c</i> , ARM Ltd., 2021.
[Babbage]	R. Roffe, Image of Charles Babbage, Wellcome Collection, 1833. [Online]. Available: https://wellcomecollection.org/works/krsuk329/images?id=d9hxrch7 .

- Bagwell00 P. Bagwell, “Ideal hash trees,” 2001. [Online]. Available: <https://lampwww.epfl.ch/papers/idealhashtrees.pdf>.
- Beck03 K. Beck, *Test-Driven Development by Example*, Pearson, 2003.
- Beebe17 N. Beebe, *The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library*, Springer, 2017.
- Bell75 G. Bell and W. Strecker, “Computer structures: What have we learned from the PDP-11?” *IEEE Computer Conference*, Florida, Nov. 1975.
- Bell76 G. Bell and W. Strecker, “Computer structures: what have we learned from the PDP-11?” *ACM SIGARCH Computer Architect. News*, pp. 1–14, Jan. 1976.
- Bennett19 D. Patterson, J. Bennett, P. Dabbelt, C. Garlati, G. S. Madhusudan, and T. Mudge, “Embench™: An evolving benchmark suite for embedded IoT computers from an academic-industrial cooperative: towards the long overdue and deserved demise of Dhrystone,” *RISC-V Workshop*, Zurich, Switzerland, June 2019. [Online]. Available: <https://github.com/embench/embench-iot>.
- Bewick94 G. W. Bewick, “Fast Multiplication: Algorithms and Implementation,” Ph.D. dissertation, Stanford University, 1994. [Online]. Available: <http://i.stanford.edu/pub/cstr/reports/csl/tr/94/617/CSL-TR-94-617.pdf>.
- Binkley09 D. Binkley, M. Davis, D. Lawrie, and C. Morrell, “To CamelCase or Under_score,” *Intl. Conf. on Program Comprehension*, Vancouver, British Columbia, Canada, pp. 158–167, 2009.
- Bohr07 M. Bohr, “A 30 year retrospective on Dennard’s MOSFET scaling paper,” *IEEE Solid-State Circuits Society Newslett.*, vol. 12, no. 1, pp. 11–13, 2007.
- bug The Naval Surface Warfare Center, Dahlgren, VA., U.S. Naval History and Heritage Command Photograph, “The First ‘Computer Bug’,” 1988. [Online]. Available: <https://www.history.navy.mil/content/history/nhhc/our-collections/photography/numerical-list-of-images/nhhc-series/nh-series/NH-96000/NH-96566-KN.html>.
- C17 “C programming,” *Wikimedia*. [Online]. Available: upload.wikimedia.org/wikipedia/commons/0/07/C_Programming.pdf.
- Cekleov97 M. Cekleov and M. Dubois, “Virtual-address caches part 1: problems and solutions in uniprocessors,” *IEEE Micro*, pp. 64–77, 1997.
- Chacon14 S. Chacon and B. Straub, *Pro Git*, Apress, 2014. [Online]. Available: <https://git-scm.com/book/en/v2>.
- codesize RISC-V Code Size Reduction Group, Archived 2024. [Online] Available: <https://github.com/riscvarchive/riscv-code-size-reduction?tab=readme-ov-file>.
- Coffman71 E. Coffman, M. Elphick, and A. Shoshani, “System deadlocks,” *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–78, June 1971.
- Colossus The National Archives (United Kingdom), catalogued under document record FO850/234, “A Colossus Mark 2 codebreaking computer being operated by Dorothy Du Boisson (left) and Elsie Booker (right),” 1943.

- [Online]. Available: https://en.wikipedia.org/wiki/Colossus_computer#/media/File:Colossus.jpg.
- Colwell02 B. Colwell, "If you didn't test it, it doesn't work," *Computer*, vol. 35, no. 5, pp. 11–13, May 2002.
- Colwell05 R. Colwell, *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips*, Wiley, 2005.
- Coonen81 J. Coonen, "Underflow and the denormalized numbers," *IEEE Computer*, vol. 4, no. 3, pp. 75–87, Mar. 1981.
- Cooper13 V. Cooper, *Getting Started with UVM: A Beginner's Guide*, Verilab Publishing, 2013.
- Corbato62 F. Corbato, M. Merwin-Daggett, and R. Daley, "An experimental time-sharing system," *Proc. Spring Joint Computer Conference*, vol. 21, pp. 335–344, 1962.
- corei9 F. Fritz and JmsDoug, "Intel Core i9-13900K die," 2023. [Online]. Available: https://commons.wikimedia.org/wiki/File:Intel_Core_i9-13900K_Labelled.
- Cornea04 M. Cornea, "Software implementations of division and square root operations for Intel Itanium processors," *Proc. Workshop on Computer Architecture Education*, ACM, 2004.
- Cortadella92 J. Cortadella and J. Llaberia, "Evaluation of $A + B = K$ conditions without carry propagation," *IEEE Trans. Computers*, vol. 41, no. 11, pp. 1484–1488, Nov. 1992.
- crtg "Sample constrained random test generator file," *Github*. [Online]. Available: https://github.com/riscv-software-src/riscv-ctg/blob/master/sample_cgfs/rv32i.cgf.
- Dadda65 L. Dadda, "Some schemes for Parallel Multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.
- Dally98 W. Dally and J. Poulton, *Digital Systems Engineering*, Cambridge Univ. Press, 1998.
- Davis64 E. M. Davis, W. E. Harding, R. S. Schwartz, and J. J. Corning, "Solid logic technology: Versatile, high-performance microelectronics," *IBM Journal of Research and Development*, vol. 8, no. 2, pp. 102–114, Apr. 1964.
- DeHon99 A. DeHon, "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)," *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays (FPGA '99)*, ACM, New York, NY, USA, pp. 69–78, 1999.
- Dennard74 R. H. Dennard, F. H. Gaensslen, H. -N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.
- Dhanesha95 H. Dhanesha, K. Falakshahi, and M. Horowitz, "Array-of-arrays architecture for parallel floating point multiplication," *Proceedings Sixteenth Conf. on Advanced Research in VLSI*, Chapel Hill, NC, USA, pp. 150–157, 1995.

- Dijkstra65 E. W. Dijkstra, “Cooperating sequential processes,” *Technical Report EWD-123*, 1965.
- Dijkstra68 E. W. Dijkstra, “The structure of the ‘THE’-multiprogramming system,” *Proceedings of the first ACM symposium on Operating System Principles (SOSP ’67)*, ACM, pp. 10.1–10.6, 1968.
- Dijkstra71 E. W. Dijkstra, “Hierarchical ordering of sequential processes,” *Acta Informatica* 1, pp. 115–138, 1971.
- Dobberpuhl92 D. W. Dobberpuhl et al., “A 200-MHz 64-b dual-issue CMOS microprocessor,” *IEEE Journal of Solid-State Circuits*, vol. 27, no. 11, pp. 1555–1567, Nov. 1992.
- Ellis21 L. Ellis, “It’s time for IEEE to retire ‘master/slave’,” *EE Times*, 2021. [Online]. Available: <https://www.eetimes.com/its-time-for-ieee-to-retire-master-slave/>.
- Ercegovac03 M. Ercegovac and T. Lang, *Digital Arithmetic*, Elsevier, 2003.
- Ercegovac94 M. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Springer, 1994.
- Eves69 H. Eves, *In Mathematical Circles*, Brookes/Cole Publishers, 1969.
- FE310 *SiFive FE310-G003 Manual v1p1*, SiFive, 2020. [Online]. Available: https://sifive.cdn.prismic.io/sifive/3af39c59-6498-471e-9dab-5355a0d539eb_fe310-g003-manual.pdf.
- Ferguson10 N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering: Design Principles and Practical Applications*, Wiley Publishing, 2010.
- Fleming86 P. Fleming and J. Wallace, “How not to lie with statistics: The correct way to summarize benchmark results,” *Communications of the ACM*, vol. 29, no. 2, pp. 218–221, Mar. 1986.
- Flynn97 D. Flynn, “AMBA: Enabling reusable on-chip designs,” *IEEE Micro*, pp. 20–27, July/Aug. 1997.
- Franke02 H. Franke, R. Russell, and M. Kirkwood, “Fuss, futexes and furwocks: Fast user-level locking in Linux,” *AUUG Conference Proceedings*, vol. 85. AUUG, Inc, 2002.
- FU540 *SiFive FU540-C000 Manual v1p0*, SiFive, 2018. [Online]. Available: <https://static.dev.sifive.com/FU540-C000-v1.0.pdf>.
- Gallmann21 N. Gallmann, P. Vogel, P. Schiavone, and L. Benini, “From swift to mighty: A cost-benefit analysis of Ibex and CV32E40P regarding application performance, power, and area,” *Workshop Computer Architecture Research with RISC-V*, 2021. [Online]. Available: https://carrv.github.io/2021/papers/CARRV2021_paper_8_Gallmann.pdf.
- Gelsinger01 P. P. Gelsinger, “Microprocessors for the new millennium: Challenges, opportunities, and new frontiers,” *2001 IEEE International Solid-State Circuits Conference Digest of Technical Papers ISSCC*, San Francisco, CA, USA, pp. 22–25, 2001.
- Gelsinger06 P. Gelsinger, “Into the Core . . . ,” 2006. [Online]. Available: <https://web.stanford.edu/class/ee380/Abstracts/060607-EE380-Gelsinger.pdf>.

- Golson16 S. Golson and L. Clark, “Language wars in the 21st century: Verilog versus VHDL—revisited,” *Synopsys Users Group Conference (SNUG)*, 2016.
- Greenblatt74 R. Greenblatt, “The LISP machine,” *Working Paper 79*, MIT Artificial Intelligence Laboratory, Cambridge, MA, USA, Nov. 1974.
- Halfhill95 T. Halfhill, “The truth behind the Pentium bug,” *BYTE.com*, 1995. [Online]. Available: <https://web.archive.org/web/20060209005434/http://www.byte.com/art/9503/sec13/art1.htm>.
- Harris21 S. Harris and D. Harris, *Digital Design and Computer Architecture: RISC-V Edition*, Elsevier, 2021.
- Harris23 D. Harris, J. Stine, M. Ercegovac, A. Nannarelli, K. Parry, and C. Turek, “Unified digit selection for radix-4 recurrence division and square root,” *IEEE Trans. Computers*, vol. 73, no. 1, pp. 292–300, Jan. 2024.
- Hatamian86 M. Hatamian and G. L. Cash, “A 70-MHz 8-bit/spl times/8-bit parallel pipelined multiplier in 2.5-/spl mu/m CMOS,” *IEEE Journal of Solid-State Circuits*, vol. 21, no. 4, pp. 505–513, Aug. 1986.
- Hauser18 J. Hauser, “Berkeley TestFloat release 3e general documentation,” 2018. [Online]. Available: <http://www.jhauser.us/arithmetic/TestFloat-3/doc/TestFloat-general.html>.
- Hennessy19 J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2019.
- Hennessy20 J. Hennessy and D. Patterson, *Computer organization and design RISC-V edition: The hardware software interface*, 2nd Edition, Morgan Kaufmann, 2020.
- Hennessy82 J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, “MIPS: A microprocessor architecture,” *Proceedings of the 15th Annual Workshop on Microprogramming (MICRO 15)*, IEEE Press, 17–22, 1982.
- Hennessy90 J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Ap Professional Publishers Inc., 1990.
- Hinton01 Hinton, G., Upton, M., Sager, D. J., Boggs, D., Carmean, D. M., Roussel, P., ... & Murray, R., “A 0.18-/spl mu/m CMOS IA-32 processor with a 4-GHz integer execution unit,” *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1617–1627, Nov. 30, 2001.
- Hopper J. S. Davis, “Commodore Grace M. Hopper, USN (covered),” 1984. [Online] Available: [https://commons.wikimedia.org/wiki/File:Commodore_Grace_M._Hopper,_USN_\(covered\).jpg](https://commons.wikimedia.org/wiki/File:Commodore_Grace_M._Hopper,_USN_(covered).jpg).
- Horel99 T. Horel and G. Lauterbach, “UltraSPARC-III: Designing third-generation 64-bit performance,” *IEEE Micro*, vol. 19, no. 3, pp. 73–85, May-June 1999.
- IBM64 IBM, “The IBM System/360.” [Online]. Available: <https://www.ibm.com/history/system-360>.
- IBMPc IBM, “The IBM PC.” [Online]. Available: <https://www.ibm.com/history/personal-computer>.

- IC3-21 FBI, *Internet Crime Report*, 2021. [Online]. Available: https://www.ic3.gov/Media/PDF/AnnualReport/2021_IC3Report.pdf.
- IDC23 IDC. [Online]. Available: <https://www.idc.com/getdoc.jsp?containerId=prAP50994023>.
- IEEE 1800.2-2017 IEEE, *Universal Verification Methodology*, IEEE Standard 1800.2-2017, 2017.
- IEEE1076-2019 IEEE, *VHDL Language Reference Manual*, IEEE Standard 1076-2019, 2019.
- IEEE1666-2011 IEEE, *Standard SystemC Language Reference Manual*, IEEE Standard 1666-2011, 2012.
- IEEE1788.1-2017 IEEE, *Interval Arithmetic*, IEEE Standard 1788.1-2017, 2017.
- IEEE1800-2017 IEEE, *SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE Standard 1800-201, 2018.
- IEEE1800-2023 IEEE, *SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE Standard 1800-2023, 2024.
- IEEE754-1985 IEEE, *Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985, 1985.
- IEEE754-2008 IEEE, *Floating-Point Arithmetic*, IEEE Standard 754-2008, 2008.
- IEEE754-2019 IEEE, *Floating-Point Arithmetic*, IEEE Standard 754-2019, 2019.
- Intel4004 IEEE Spectrum, “The surprising story of the first microprocessors.” [Online]. Available: <https://spectrum.ieee.org/the-surprising-story-of-the-first-microprocessors>.
- Intel69 Intel, “Intel’s first product—the 3101,” [Online]. Available: <https://www.intel.com/content/www/us/en/history/virtual-vault/articles/intels-first-product-3101.html>.
- Intel71 Intel, “Announcing a new era of integrated electronics: The Intel 4004.” [Online]. Available: <https://www.intel.com/content/www/us/en/history/virtual-vault/articles/the-intel-4004.html>.
- Intel89 “80486—Intel,” *WikiChip*. Intel Corporation, i486 Microprocessor, 1989. [Online]. Available: <https://en.wikichip.org/wiki/intel/80486#Datasheet>.
- iphone13 Tech Insights, “What’s inside the Apple iPhone 13 Pro teardown?” [Online]. Available: <http://www.techinsights.com/blog/teardown/apple-iphone-13-pro-teardown>.
- Isen09 C. Isen, L. K. John, and E. John, “A tale of two processors: Revisiting the RISC-CISC debate,” *2009 SPEC Benchmark Workshop*, Springer, vol. 5419, pp. 57–75, 2009.
- Keating02 M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*, Springer, 2002.
- Keating07 M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual*, Springer, 2007.
- Kidder81 T. Kidder, *The Soul of a New Machine*, Little, Brown, & Co., 1981.
- Kilburn62 T. Kilburn, D. Edwards, M. Lanigan, and F. Sumner, “One-level storage system,” *IRE Trans. Electronic Computers*, vol. EC-11, no. 2, pp. 223–235, Apr. 1962.

- Kim25 K. Kim, K. Parry, D. Harris, C. Turek, A. Maiuolo, R. Thompson, and J. Stine, “Shared Recurrence Floating-Point Divide/Sqrt and Integer Divide/Remainder With Early Termination” *IEEE Transactions on Computers*, 2024.
- Lamport79 L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Trans. Computers*, vol. C-28, no. 9, pp. 690–691, Sept. 1979.
- Larabel19 M. Larabel, “The Linux kernel enters 2020 at 27.8 million lines in Git but with less developers for 2019.” [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019.
- Laskai21 L. Laskai and A. Segal, “The Encryption Debate in China: 2021 Update,” 2021. [Online]. Available: <https://carnegieendowment.org/posts/2021/03/the-encryption-debate-in-china-2021-update>.
- Lichtenau16 C. Lichtenau, S. Carlough, and S. M. Mueller, “Quad precision floating point on the IBM z13,” *IEEE Symp. Computer Arithmetic*, pp. 87–94, 2016.
- Liu12 W. Liu and A. Nannarelli, “Power efficient division and square root unit,” *IEEE Trans. Computers*, vol. 61, no. 8, pp. 1059–1070, 2012.
- Lockless23 Lockless Inc., “Mutexes and condition variables using futexes,” 2023. [Online]. Available: http://locklessinc.com/articles/mutex_cv_futex.
- MacSorley61 O. L. Macsorley, “High-speed arithmetic in binary computers,” *Proc. IRE*, vol. 49, no. 1, pp. 67–91, Jan. 1961.
- Magnussen97 P. Magnussen, “Understanding stacks and registers in the Sparc architecture(s),” 1997. [Online]. Available: <https://cseweb.ucsd.edu/~gbournou/CSE131/sparcstack.html>.
- market25 T. Alsop, “Semiconductor market size worldwide from 1987 to 2025,” 2025. [Online]. Available: <http://www.statista.com/statistics/266973/global-semiconductor-sales-since-1988>.
- MarkI J. Kopplin, “An illustrated history of computers: Part 3,” 2002. [Online]. Available: <http://www.cs.kent.edu/~rothstei/10051/HistoryPt3.htm>.
- Mayer82 A. J. W. Mayer, “The architecture of the Burroughs B5000: 20 years later and still ahead of the times?” *SIGARCH Comput. Archit. News* 10, pp.3–10, 1982.
- McFarling93 S. McFarling, “Combining branch predictors,” *WRL Technical Note TN-36*, Digital Equipment Corporation, 1993.
- Mecklenburg04 R. Mecklenburg, *Managing Projects with GNU Make*, 3rd. ed., O’Reilly, 2004.
- Mehta20 A. Mehta, *SystemVerilog Assertions and Functional Coverage*, 3rd. ed., Springer, 2020.
- memory Ken Shirriff’s blog. [Online]. Available: <http://static.righto.com/images/ibm-360-mem/core-closeup2.jpg>.
- Met Museum of Art/Intel Intel, “October 17, 1986: Raising the bar with the 386.” [Online]. Available: <https://timeline.intel.com/1985/raising-the-bar-with-the-386>.

- Montoye90 R. Montoye, E. Hokonek, and S. Runyan, “Design of the IBM RISC System/6000 floating-point execution unit,” *IBM J. Research & Development*, vol. 34, no. 1, pp. 59–70, 1990.
- Moody01 G. Moody, “The Greatest OS that (N)ever Was,” *Wired*, Aug. 1, 1997.
- Moore03 G. E. Moore, “No exponential is forever: But “forever” can be delayed! [semiconductor industry],” *2003 IEEE International Solid-State Circuits Conference, 2003 Digest of Technical Papers ISSCC.*, San Francisco, CA, USA, vol. 1, pp. 20–23, 2003.
- Moore65 G. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.
- Muller05 J.-M. Muller, “On the definition of ulp,” *Research Report 2005-09*, École Normale Supérieure de Lyon, 2005.
- Muller18 J. Muller et al., *Handbook of Floating-Point Arithmetic*, 2nd ed., Springer, 2018.
- MultilayerAHB04 Multi-layer AHB V2.0 Technical Overview DVI0045B, ARM Ltd., 2004.
- Nagarajan20 V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, “A primer on memory consistency and cache coherence,” 2nd ed., *Synthesis Lectures on Computer Architecture*, vol. 15, no. 1, pp. 1–294, 2020.
- NameConv “Naming convention (programming),” *Wikipedia*. [Online]. Available: [https://en.wikipedia.org/wiki/Naming_convention_\(programming\)](https://en.wikipedia.org/wiki/Naming_convention_(programming)).
- NIST-FIPS-180-4 National Institute of Standards and Technology, *Secure Hash Standard*, Pub 180-4, Aug. 2015. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.
- NIST-FIPS-197-upd1 National Institute of Standards and Technology, *The Advanced Encryption Standard (AES)*, Pub 197. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf>.
- OBI22 Open Bus Interface 1.4, Silicon Labs, 2022. [Online]. <https://github.com/openhwgroup/core-v-docs/blob/master/cores/obi/OBI-v1.2.pdf> <https://github.com/openhwgroup/programs/blob/master/TGs/cores-task-group/obi/OBI-v1.4.pdf>.
- Oklobdzija96 V. G. Oklobdzija, D. Villeger, and S. S. Liu, “A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach,” *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–306, Mar. 1996.
- onsemi61 Fairchild/ON Semiconductor Corporation via The Chip History Center, “First Monolithic Silicon IC Chip. Invented by Robert Noyce, Fairchild Semiconductors,” 1961. [Online]. Available: https://www.chiphistory.org/chc_upload/content/jpg/7/1499079245/1499079245.jpg.
- opensource.org The Open-Source Initiative. [Online]. Available: <https://opensource.org>.
- OSHA22 Open Source Hardware Association, “A resolution to redefine SPI signal names,” 2022. [Online]. Available: <https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names/>.
- oshwa.org Open Source Hardware Association. [Online]. Available: <https://www.oshwa.org>.

- Ousterhout09 J. Ousterhout and K. Jones, *Tcl and the Tk Toolkit*, Upper Saddle River, NJ, USA: Addison-Wesley, 2009.
- Pan92 S.-T. Pan, I. So, and J. T. Rahmeh, “Improving the accuracy of dynamic branch prediction using branch correlation,” *Proc. Architecture Support for Programming Languages and Operating Systems*, pp. 76–84, 1992.
- Patterson25 D. Patterson et al., “Embench IOT 2.0 and DSP 1.0: Modern Embedded Computing Benchmarks,” *IEEE Computer*, vol. 58, no. 5, May 2025, pp. 37–47.
- Patterson80 D. R. Ditzel and D. A. Patterson, “Retrospective on high-level language computer architecture,” *Proceedings of the 7th Symposium Computer Architecture (ISCA '80)*, ACM, pp. 97–104, 1980.
- PC16550 Texas Instruments, “PC16550D universal asynchronous receiver/transmitter with FIFOs,” 1995.
- PDP-11 D. McGuire, “Later PDP-11/70 with disks and tape,” 2009. [Online]. Available: <https://en.wikipedia.org/wiki/PDP-11#/media/File:PDP-11-70-DDS570.jpg>.
- PLIC22 RISC-V International, “RISC-V Platform-Level Interrupt Controller Specification,” 2022. [Online]. Available: <http://github.com/riscv/riscv-pllic-spec/blob/master/riscv-pllic.adoc>.
- Prabhu95 J. A. Prabhu and G. B. Zyner, “167 MHz radix-8 divide and square root using overlapped radix-2 stages,” *IEEE Symposium on Computer Arithmetic*, pp. 155–162, 1995.
- Quinnell07 E. Quinnell, E. E. Ewertzlander, Jr., and C. Lemonds, “Floating-point fused multiply-add architectures,” *Asilomar Conf. Signals, Systems, and Computers*, pp. 331–337, 2007.
- RISCV23 “RISC-V Celebrates Momentum of 2022 and Accelerating Adoption in 2023,” RISC-V International, 2023. [Online]. Available: <https://riscv.org/blog/2023/01/risc-v-celebrates-momentum-of-2022-and-accelerating-adoption-in-2023/>.
- RISCV-ABI RISC-V International, “RISC-V ELF Processor-Specific ABI,” [Online]. Available: <http://github.com/riscv-non-isa/riscv-elf-psabi-doc>.
- riscv-arch-test RISC-V International, “RISC-V Foundation Architecture Test Special Interest Group (SIG),” [Online]. Available: <http://github.com/riscv-non-isa/riscv-arch-test>.
- RISCV-ASM RISC-V International, “RISC-V assembly programmer’s manual,” [Online]. Available: <http://github.com/riscv-non-isa/riscv-asm-manual>.
- RISCV-Privileged RISC-V International, “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.” [Online]. Available: <https://riscv.org/specifications/ratified/>.
- RISCV-Profiles RISC-V International, “RISC-V Profiles.” [Online]. Available: <https://riscv.org/specifications/ratified/>.
- RISCV-SAIL RISC-V International, “RISC-V Sail Model.” [Online]. Available: <https://github.com/riscv/sail-riscv>.
- RISCV-SBI RISC-V International, “RISC-V SBI Specification.” [Online]. Available: <https://github.com/riscv-non-isa/riscv-sbi-doc>.

- RISCV-Unprivileged RISC-V International, “The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture.” [Online]. Available: <https://riscv.org/specifications/ratified/>.
- Ritchie20 H. Ritchie and M. Roser, “Moore’s law,” 2020. [Online]. Available: https://commons.wikimedia.org/wiki/File:Moore%27s_Law_Transistor_Count_1970-2020.png.
- Ritchie84 D. Ritchie, “The evolution of the Unix Time-sharing System,” *Communications of the ACM*, 1984, pp. 365–375.
- Robertson58 J. Robertson, “A new class of digital division methods,” *IRE Trans. Electronic Computers*, vol. EC-7, pp. 218–222, Sept. 1958.
- RV22 M. Himmelstein, “RISC-V state of the union,” 2022. [Online]. Available: <https://open-src-soc.org/2022-05/media/slides/RISC-V-International-Day-2022-05-05-09h00-Mark-Himmelstein.pdf>.
- RVVI-TRACE “RISC-V verification interface,” *Github*. [Online]. Available: <https://github.com/riscv-verification/RVVI>.
- RyzenVi9 B. Justice, “Intel Core i9-12900K vs AMD Ryzen 9 5900X Performance Review,” Feb. 8, 2022. [Online]. Available: <https://www.thefpsreview.com/2022/02/08/intel-core-i9-12900k-vs-amd-ryzen-9-5900x-performance-review/6/>.
- Saarinen20 M.-J. O. Saarinen, “A lightweight ISA extension for AES and SM4,” *International Workshop on Secure RISC-V*, Apr. 2020.
- SAIL “The Sail ISA specification language,” *Github*. [Online]. Available: <https://github.com/remis-project/sail>.
- Santoro89 M. R. Santoro and M. A. Horowitz, “SPIM: A pipelined 64*64-bit iterative multiplier,” *IEEE Journal of Solid-State Circuits*, vol. 24, no. 2, pp. 487–493, Apr. 1989.
- Schmookler01 M. S. Schmookler and K. J. Nowka, “Leading zero anticipation and detection - a comparison of methods,” *IEEE Symp. Computer Arithmetic*, pp. 7–12, 2001.
- Schotts19 W. Schotts, “The Linux command line,” 2024. [Online]. Available: <http://linuxcommand.org/tlcl.php>.
- Schwartz05 E. M. Schwartz, M. Schmookler, and S. D. Trong, “FPU implementations with denormalized numbers,” *IEEE Trans. Computers*, vol. 54, no. 7, pp. 825–836, July 2005.
- Scott13 M. Scott, *Shared-Memory Synchronization*, Morgan & Claypool, 2013.
- Seidel03 P. Seidel, “Multiple path IEEE floating-point fused multiply-add,” *International Midwest Symp. Circuits and Systems*, pp. 1359–1362, 2003.
- Seligman23 E. Seligman, T. Schubert, and M. V. A. K. Kumar, *Formal verification: An essential toolkit for modern VLSI Design*, 2nd ed., Elsevier, 2023.
- Sequin82 C. H. Sequin and D. A. Patterson, “Design and Implementation of RISC I,” *UC Berkeley Technical Report*, 1982.
- Severance16 C. Severance, *Python for Everybody*, 2016. [Online]. Available: http://do1.dr-chuck.com/pythonlearn/EN_us/pythonlearn.pdf.

- Severance98 C. Severance, “IEEE 754: An interview with William Kahan,” *Computer*, vol. 31, no. 3, pp. 114–115, Mar. 1998.
- Seznec06 A. Seznec and P. Michaud, “A case for (partially) TAGged GEometric history length branch prediction,” *J. Instruction-Level Parallelism*, vol. 8, pp. 23, 2006.
- Shannon45 C. E. Shannon, “A mathematical theory of cryptography,” *Bell System Technical Memo*, MM 45-110-02, Sept. 1945.
- Shannon49 C. E. Shannon, “Communication theory of secrecy systems,” *Bell System Technical Journal*, vol. 28-4, pp. 656–715, 1949.
- SHD24 SHD Group, “RISC-V market report: Application forecasts in a heterogeneous world,” *SHD101A.b-24*, Jan. 2024.
- Shilov20 A. Shilov, “The future of leading-edge chips according to TSMC: 5nm, 4nm, 3nm and beyond,” TechRadar, 2020. [Online]. Available: <https://www.techradar.com/news/the-future-of-leading-edge-chips-according-to-tsmc-5nm-4nm-3nm-and-beyond>.
- SLT IEEE Spectrum, “Building the system/360 mainframe nearly destroyed IBM.” [Online]. Available: <https://spectrum.ieee.org/building-the-system360-mainframe-nearly-destroyed-ibm#toggle-gdpr>.
- SPEC2000 Standard Performance Evaluation Corporation, “SPEC CPU2000 V1.3 Benchmark,” 2000. [Online]. Available: <https://www.spec.org/cpu2000/>.
- SPEC92 Standard Performance Evaluation Corporation, “SPEC CPU92 Benchmark,” 1992. [Online]. Available: <https://www.spec.org/cpu92/>.
- Spike RISC-V International, “Spike RISC-V ISA simulator.” [Online]. Available: <http://github.com/riscv-software-src/riscv-isa-sim>.
- Stallman22 R. Stallman, R. McGrath, and P. Smith, “GNU make,” 2022. [Online]. Available: <http://www.gnu.org/software/make/manual/make.pdf>.
- Stallman23 R. Stallman, R. Pesch, S. Shebs, et al., “Debugging with GDB,” 2023. [Online]. Available: <http://sourceware.org/gdb/current/onlinedocs/gdb.pdf>.
- Stallman86 R. M. Stallman, “What is the Free Software Foundation?” *GNU’s Bulletin*, vol. 1, no. 1, Feb. 1986.
- Statista25 “Arm Holdings chip unit shipments worldwide from fiscal year 2017 to 2024, by quarter (in billions),” Statista, 2025. [Online]. Available: <https://www.statista.com/statistics/1131983/arm-based-chip-unit-shipments-as-reported-by-licensees-worldwide/>.
- StDenis07 T. St. Denis, *Cryptography for Developers*, Syngress Publishing, 2007.
- Stelling98 P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi, “Optimal circuits for parallel multipliers,” *IEEE Transactions on Computers*, vol. 47, no. 3, pp. 273–285, Mar. 1998.
- Suggs20 D. Suggs, M. Subramony, and D. Bouvier, “The AMD ‘Zen 2’ processor,” *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.
- Sutherland06 S. Sutherland and D. Mills, “Standard gotchas subtleties in the Verilog and SystemVerilog standards that every engineer should know,” *Synopsys User Group Conference Proc. (SNUG)*, 2006/7.

- Sutherland13 S. Sutherland, “I’m still in love with my X,” *Design and Verification Conference*, 2013.
- Sutherland99 I. Sutherland, R. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann, 1999.
- Swann23 R. Swann and J. E. Stine, “Evaluation of a modular approach to AES hardware architecture and optimization,” *J Sign Process Syst*, vol. 95, no. 7, pp. 797–813, 2023.
- Sweeny57 J. Cocke and D. Sweeney, “High speed arithmetic in a parallel device,” *IBM Technical Report*, 1957. [Online]. Available: <http://archive.computerhistory.org/resources/text/IBM/Stretch/pdfs/06-08/102632302.pdf>.
- Taylor12 M. B. Taylor, “Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse,” *DAC Design Automation Conference 2012*, San Francisco, CA, USA, pp. 1131–1136, 2012.
- Thompson23 R. Thompson, A. Sutton, D. Harris, L. Moore, and J. Stine, “Challenges in CPU DV: Why ‘It boots Linux’ is not verification,” *RISC-V Summit*, Nov. 2023. [Online]. Available: <https://www.youtube.com/watch?v=MM9tUC7rR88>.
- ThompsonRitchie “Ken Thompson (left) and Dennis Ritchie (right), creators of the Unix operating system,” 1973. [Online] Available: https://en.wikipedia.org/wiki/History_of_Linux#/media/File:Ken_Thompson_and_Dennis_Ritchie-1973.jpg.
- TileLink20 TileLink Specification 1.8.1, SiFive, 2020.
- Tocher58 K. Tocher, “Techniques of multiplication and division for automatic binary computers,” *Quart. J. Mech. Appl. Math.*, vol. 11, pt. 3, pp. 364–384, 1958.
- Trimberger15 S. M. Trimberger, “Three ages of FPGAs: a retrospective on the first thirty years of FPGA technology,” *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015.
- Turing “File:Alan Turing Aged 16.jpg,” *Wikimedia*. [Online]. Available: https://commons.wikimedia.org/wiki/File:Alan_Turing_Aged_16.jpg.
- Usage22 “Usage share of operating systems,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Usage_share_of_operating_systems.
- Vasquez21 F. Vasquez and C. Simmonds, *Mastering Embedded Linux Programming: Create Fast and Reliable Embedded Solutions With Linux 5.4 and the Yocto Project 3.1 (Dunfell)*, Packt Publishing, 2021.
- vonNeumann John von Neumann, Los Alamos National Laboratory, 2008. [Online]. Available: <https://commons.wikimedia.org/wiki/File:JohnvonNeumann-LosAlamos.gif>.
- vonNeumann45 J. von Neumann, “First draft of a report on the EDVAC,” June 30, 1945.
- Waterman16 A. Waterman, “Design of the RISC-V instruction set architecture,” *UC Berkeley Technical Report UCB/EECS-2016-1*, Jan. 3, 2016. [Online]. Available: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html>.
- Weinberger81 A. Weinberger, “4:2 Carry-Save Adder Module,” *IBM Technical Disclosure Bulletin*, vol. 23, Jan. 1981.

- Weste12
WikiChipProcTech
Wilson10
Wishbone10
Xiong10
Yeh91
Zafar19
Zimmerman96
- N. Weste and D. Harris, *CMOS VLSI Design*, 4th. ed., Pearson, 2012.
- “Process Technology History—Intel,” *WikiChip*. [Online]. Available: <https://en.wikichip.org/wiki/intel/process>.
- The 2010 Wilson Research Group Functional Verification Study. [Online]. Available: <https://blogs.sw.siemens.com/verificationhorizons/2011/03/30/part-1-the-2010-wilson-research-group-functional-verification-study/>.
- OpenCores, Wishbone B4 Specification, 2010. [Online]. Available: https://cdn.opencores.org/downloads/wbspec_b4.pdf.
- W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, “Ad hoc synchronization considered harmful,” *9th USENIX Symp. Operating Systems Design and Implementation*, pp. 163–176, Oct. 2010.
- T.-Y. Yeh and Y. Patt, “Two-level adaptive training branch prediction,” *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO 24)*, pp. 51–61, 1991.
- R. Zafar, “Apple A13 & beyond: How transistor count and costs will go up,” 2019. [Online]. Available: <https://wccfttech.com/apple-5nm-3nm-cost-transistors>.
- R. Zimmermann, “Non-heuristic optimization and synthesis of parallel-prefix adders,” *Proc.Intl. Workshop on Logic and Architecture Synthesis*, pp. 123–132, Dec. 1996.

Index

Note: Page numbers followed by f indicate figures, t indicate tables, and b indicate boxes.

A

Access faults, 353
Actel, 784.e79
Activity factor, 256
Addend generation, 784.e248
Address decoder and multiplexer,
 AHB, 397f
Address misaligned exception, 353
Address translation, 470
AddRoundKey, 723–724
Ad-hoc testing, 196
Advanced encryption standard (AES),
 715–716, 722–731
Advanced microcontroller bus
 architecture (AMBA) buses
 advanced eXtensible interface,
 404–410
 advanced high-performance bus,
 395–404
 additional signals, 397–400
 burst transfers, 402–403
 multilayer, 404
 peripherals, 400
 SRAM interface, 400–401
 -to-APB bridge, 401–402
 advanced peripheral bus, 392–394
AES decryption architecture, 729f
Agile project management, 143–144
Alignment shift, 784.e195f, 784.e196t
align module, 778f
Altera, 784.e79
AMD
 Epyc 9004 motherboard, 675f
 DRAM controller, 784.e82
 IP Modules, 784.e82t
 AMOALU atomic module, 709–710
 Android, 784
 Application binary interface (ABI),
 50, 95–99, 96t, 97f
 Application processors, 32
 Application-specific integrated circuit
 (ASIC), 141
 Architectural test pool, 134
 Area and delay *vs.* configurations,
 291–294, 292f, 294t–295t
 Area and delay *vs.* target clock period,
 287–291, 290f
 Arithmetic circuitry, 303–311, 304t
 Arithmetic/logic unit (ALU), 33
 decoder, 313, 314t
 operation, 58t
 Array multiplier, 594, 595f
 Arrays of modules, 176
 Arty A7 board, 784.e81, 784.e81
 AHB burst performance, 784.e93
 components, 784.e83
 FPGA block diagram, 784.e82f
 FPGA program, 784.e87
 in-circuit logic analyzer,
 784.e93–784.e96
 PmodSD DRAM expansion card,
 784.e83f
 SD Flash Card, 784.e84–784.e86
 setup, 784.e84–784.e96
 UART, 784.e88–784.e92
 Wally Vivado, 784.e88f
 Assembler directives, 103–104, 104t
 Assertions, 173, 195, 241
 Associative caches, 436–438

Atomic instructions, 673
 atomic memory operations, 701–703
 LoadReserved-StoreConditional,
 703–704
Atomic memory operations, 784.e135f
Atomic module
 amoalu, 709–710
 atomic hazards, 710
 LRSC module, 710
Atomic waveforms, 712f
AXI channel signals, 406t–407t

B

Bad synchronizer, 143–144
Basic bit manipulation, 784.e141f
Behavioral/register transfer level
 (RTL), 141
Benchmarking, 76–77, 77t
 CoreMark, 784.e24–784.e27
 Embench, 784.e27–784.e38
 goal, 784.e24
 performance verification
 for CoreMark, 784.e39
 for Embench, 784.e39
 Wally's branch predictor,
 784.e40–784.e44
 Wally's instruction and data
 caches, 784.e39
Better locking, 686
Bidirectional I/O, 154f
Bidirectional *vs.* unidirectional buses,
 385–386

- Binary files, 784
 - Bit manipulation (Zb*)
 - riscv-arch-test, 749f
 - Wally implementation, 748–764
 - ALU Modifications
 - inverted operations, 753
 - rotates, 752–753
 - Zba extension, 752
 - Zbb extension, 752
 - zbb module, 753–754
 - Zbc, 755–757
 - Zbs, 757
 - controller modifications, 764
 - Bitstream, 784.e78
 - Bit swizzling, 169
 - Bit widths, 784.e252–784.e254
 - Boolean equations, 141
 - Booth encoding, 594, 597–598
 - array multipliers, 597
 - example, 598f
 - modified, 597
 - radix-4, 598t, 599f
 - signed 2's complement
 - multiplication, 598
 - Booting Linux, 196
 - modifying buildroot
 - configuration files, 784
 - rebuilding the image, 784.e15
 - root file system overlay, 784
 - on QEMU, 784.e12–784
 - on Wally in Questa simulation, 784
 - Bootloader, 784.e3
 - Boot trace, 784.e3–784
 - Branch direction predictor, 784.e40
 - Branches, 784f
 - Branch logic, 312–313, 313f
 - Branch misprediction, 547
 - Branch offset, 35
 - Branch prediction principles
 - branch target buffer, 554–556
 - Gshare, 551–552
 - misprediction penalty, 545–547
 - return address stack, 556–557
 - simple dynamic branch prediction, 548–551
 - static branch prediction, 548
 - TAGE, 557
 - Branch predictor
 - branch misprediction recovery, 572
 - BTB implementation, 568–569
 - Gshare implementation, 564–568
 - instruction class prediction, 569–570
 - RAS implementation, 570–572
 - Branch predictor performance
 - verification
 - CoreMark branch accuracy, 784.e44
 - direction and target reference model, 784.e40–784.e42
 - embench branch accuracy sweeps, 784.e42–784.e44
 - Branch target address (BTA), 545–547, 554
 - Branch target buffer (BTB), 547, 554–556
 - BTA. *See* Branch target address (BTA)
 - BTB. *See* Branch target buffer (BTB)
 - B-type instructions, 35
 - Buildroot, 784
 - Buildroot Linux, 784
 - platforms, 784.e48
 - principles
 - Bootloader, 784.e3
 - device tree, 784.e50–784
 - embedded Linux, 784.e3
 - init, 784
 - kernel, 784
 - root filesystem, 784.e51–784
 - shell, 784
 - on Wally, 784.e52–784
 - boot overview, 784.e6
 - building Linux, 784
 - device tree, 784.e12
 - file locations, 784t
 - init process, 784.e12
 - kernel, 784.e63–784
 - memory map, 784
 - openSBI Firmware. *See* OpenSBI Firmware
 - root file system, 784.e12
 - trap handlers, 784.e69–784
 - zero stage bootloader, 784.e6
- Build systems, 784
- Burst EBU, 525
- Burst transfers, 402–403
- Bus cache FSM, 522f
- Bus hazards, 423–426
- Bus interface
 - other SOC buses, 410
- principles
 - bidirectional *vs.* unidirectional buses, 385–386
 - handshaking, 386
 - high-speed off-chip links, 390–391
 - multiple controllers, 387–389
 - multiple peripherals, 386–387
 - other bus features, 389–390
 - test plan, 410–411
- Bypassing, 321–322
- Byte write enables, 164–165, 164f
- ## C
- Cache coherence, 676
 - Cache design tradeoffs, 438–439
 - Cache flush and invalidation, 441–442
 - Cache FSM, 454–456
 - Cache line
 - fetches, 520–523
 - writeback, 523–525
 - Cache load hit, 534
 - Cache management operation (CMO)
 - extension, 389, 767
 - RISC-V instructions, 767–769
 - environment configuration registers, 768–769
 - Zicbom cache block management, 768
 - Zicbop cache block prefetch, 768
 - Zicboz cache block zero, 768
 - test plan, 769
 - Wally implementation
 - cachefsm Inputs and Outputs, 773t
 - cache modifications, 770–774, 771f, 773f
 - LSU modifications, 769–770, 770f
 - prefetch, 774
 - uncached cbo.zero, 774
- ## Caches
- modules, 241–242
 - performance, 430–431
 - RISC-V practices, 442
 - store hit, 534–536
 - test plan, 442
 - way module, 446–449
 - way read hit, 448–449
 - way write hit, 450–453
 - writes, 434–436
 - Capitalization conventions, 182t
 - Card detect (CD), 784.e84
 - Carry-less multiplication, 784.e141f
 - Carry-propagate adders (CPAs), 592–593

- Carry-save addition, 592–593
 - C extension, 580
 - Chip enable, 165–166
 - Clock/data recovery (CDR), 391
 - Clock gating, 257
 - Coalescing write buffer, 679
 - Code coverage, 195, 217–221
 - Code examples, 108b–114b
 - Coherence, 676
 - Combinational logic
 - using always_comb statements, 149–151, 150f
 - using assign statements, 148–149, 149t, 150f
 - Xs and Zs, Oh My, 151–153, 152f–153f
 - Combinational read, 161
 - Comparator, 308–311
 - Comparisons operations, 34
 - Compatible time sharing system (CTSS), 336
 - Compiler explorer, 123, 123f
 - Complex instruction set computer (CISC) architecture, 55
 - Computer-aided design (CAD), 250
 - Concurrency, 673–676
 - Concurrent assertions, 216–217
 - Condition variables, 682–683
 - Cone of influence, 227
 - Configurable hardware, 173–181, 189
 - arrays of modules, 176
 - conditional generates, 174–175
 - disabling signals, 173
 - generate loops, 175
 - with parameter structure, 178–181
 - values of configurable width, 174
 - Consistency, 673, 677–682
 - acquire and release constraints, 681
 - release consistency, 679–681
 - sequential, 678
 - summary, 682
 - total store ordering, 678–679
 - Constrained random tests, 226
 - Content-addressable memories (CAMs), 168
 - Control and status registers (CSRs), 31, 46, 212–213, 273, 299, 526–527
 - counters, 346–348
 - environment configuration registers, 348–349
 - interaction with peripherals, 348
 - machine information registers, 346, 371
 - misa and XLEN, 371
 - misa register, 346
 - performance counters, 371
 - status register, 343–346
 - Controller design, 59–60, 60f, 60t, 311–312, 312f
 - ALU decoder, 313, 314t
 - branch logic, 312–313, 313f
 - IFU, 314
 - load/store unit, 314, 315f
 - main decoder, 312, 313t
 - sample program, 316, 317f
 - Convert float-to-float special case handling, 784.e271t
 - Convert float-to-int special case handling, 784.e270t
 - Convert logic, 784.e258t
 - Core local interruptor (CLINT) peripheral, 99
 - memory-mapped registers, 784, 784.e1t
 - testing plan, 784.e1
 - top-level interface, 784, 784f
 - Wally implementation, 784
 - CoreMark
 - branch accuracy, 784.e44
 - clock cycles, 784.e26
 - compile, 784.e25
 - compiler flags, 784t
 - Dhrystone, 784.e24
 - instruction mix, 328t
 - instruction set impace, 784.e27
 - performance *vs.* ISA, 784.e28t, 784.e29t
 - porting, 784.e26–784.e27
 - run output, 784.e25f
 - scores, 77t
 - source code, 784.e24
 - steps, 784.e24
 - Verilog simulation, 784.e25–784.e26
 - Core running process, 47f
 - Cores, 47, 48f
 - CORE-V-Wally, 199
 - Cortex-M4 reference processor, 784.e28–784.e29
 - Counters, 346–348
 - Coverage-driven testing, 197
 - Coverage points, 197
 - C programming, 784.e167
 - Cryptographic hash functions, 731–734
 - Cryptography (Zk*), 715
 - principle
 - advanced encryption standard, 722–731
 - AddRoundKey, 723–724
 - architecture, 724f
 - array structure, 723f
 - decryption, 729–731
 - hash functions, 731–734
 - KeyExpansion, 726–729
 - MixColumns, 725–726
 - ShiftRows, 724–725
 - SubBytes, 724
 - finite fields, 716–722
 - GF(2_n), 718–722
 - GF(p), 717–718
 - riscv-arch-test, 749t
 - Wally implementation, 748–764
 - ALU Modifications, 757–764
 - Zbkb extension, 758
 - zbkx module, 759–760
 - Zknd and Zkne, 760–763
 - zknde32, 761–762
 - zknde64, 762–763
 - Zknh, 763–764
 - Zkt, 764
 - controller modifications, 764
 - Custom extensions, 783–784
 - Cycles per instruction (CPI), 784.e26
- ## D
- Data cache, 429
 - integration
 - burst EBU, 525
 - bus cache FSM, 522f
 - cache line fetches, 520–523
 - cache line writeback, 523–525
 - simplified LSU, 518f
 - Data tightly integrated memory (DTIM), 299
 - Deadlock, 696–697
 - Debugging, 203
 - with GDB, 128–132, 129t
 - hints, 235–239
 - Decryption, 729–731
 - Default subordinate, 396

Design Compiler, 281–282
 Design strategies, 142–144
 Design verification (DV) plan,
 194, 202–203, 223t, 227–228
 Device tree, 784.e50–784
 D flip-flop, 154
 Dhrystone, 784.e24
 Digital signal processing (DSP),
 784.e79–784.e80
 Digit selection, 784.e248
 Directed tests, 196, 222–226
 Direct-mapped cache, 432–434
 Direct memory access (DMA), 389
 Directory-based coherence, 676
 Direct ripple implementation, 176f
 Disassembled files, 784.e3
 Divider control schematic, 161f
 Divider FSM, 160f
 Divide special case handling,
 784.e269t
 Divide/Square root shift calculation,
 784.e264t
 Division
 N-bit integers, 602
 other integer division optimization,
 604
 radix-2 restoring unsigned integer,
 602–603
 signed integer, 604
 Divisor latch access bit (DLAB), 784–
 Double-data-rate (DDR), 144–145
 Double-precision,
 616.e3f, 616.e3b, 616.e4b
 dRead port, 161
 Dynamic RAM (DRAM), 168, 429
 Dynamic random access memory
 (DRAM), 784.e81
 Dynamic rounding, 616.e48

E

Edge-triggered read, 161
 Embedded processors, 32
 Embence
 benchmark, 784.e27–784.e28
 code size, 784.e36t
 Cortex-M4 processor, 784.e31
 flags, 784.e35t
 instruction set impact,
 784.e35–784.e38

JSON files, 784.e31
 porting, 784.e33–784.e34
 1.0 programs, 784.e30t
 Python lief library, 784
 repository, 784.e29
 size calculation, 784.e33
 speed, 784.e37t
 troubleshooting, 784.e38
 Wally benchmarking, 784.e30
 Wally Embench 1.0 rv32imc,
 784.e32t
 Embench branch accuracy sweeps,
 784.e43f
 Embench scores, 78t
 EmbToolkit, 784
 Emulation, 198–199
 Emulator, 51
 Enabled flip-flops, 156–157, 156f
 Encryption example, 730f
 Endianness, 37, 37f
 Endian swapper, 378–379
 Environment configuration registers,
 348–349
 Equality comparator, 150f
 Exceptions, 49, 466–467
 access faults, 353
 address misaligned, 353
 ecall and ebreak, 354
 generating instructions, 341–342
 illegal instructions, 352–353
 page faults, 354
 Executable and Linkable Format
 (ELF), 89
 Execution environment (EE),
 46, 49–51, 50f
 External bus unit (EBU),
 411, 415–417, 784.e136f
 External interrupts, 784

F

Factorial function, 42b
 optimized, 42b
 recursive, 43b
 Faster tree implementation, 176f
 fclassify classification, 662–663
 fcmp unit, 640
 Field programmable gate array
 (FPGA), 78, 141, 163, 194, 250
 Actel, 784.e79

vs. ASICs, 784.e78
 in emulation system,
 784.e78–784.e79
 generic architecture, 784.e80f
 interesting bugs, 784.e100–784.e104
 AHB's burst, 784.e101–784.e102
 concurrent exceptions, 784.e102
 critical path analysis,
 784.e103f, 784.e104f
 lookup table, 784.e79
 Moore's law, 784.e79
 principles, 784.e79–784.e80
 results, 784.e99–784.e100
 Wally
 implementation. *See* Wally
 implementation. *See* FPGA
 specifications, 784.e99t
 utilization, 784.e100t
 Finite fields, 716–722
 Finite state machines (FSMs),
 157, 158f
 First C program, 115–120, 120t
 First-in, first-out (FIFO), 214
 Flash memory controllers, 144–145
 Flattened device tree (FDT), 784.e51
 Flip-flops, 154–157, 154f–155f
 Floating-point, 617, 784.e193
 compare unit, 784.e148f
 comparisons, 640
 conversion. *See* Floating-point
 conversion unit
 conversions supported in RV32, 662t
 divider operand sizes, 784t
 fields, 619f
 format codes, 659
 fused multiply-add unit,
 784.e146f
 less than flag logic, 640
 Min/Max, 640t
 Postprocessing. *See* Floating-point
 postprocessing unit
 Floating-point conversion unit,
 784.e193
 examples, 784.e261–784.e262
 result exponent, 784.e257
 float-to-float, 784.e257
 float-to-int, 784.e257–784.e259
 int-to-float, 784.e259–784.e260
 ResSubnormUf, 784.e260
 shift amount selection, 784.e260
 float-to-float, 784.e260
 float-to-int, 784.e260
 int-to-float, 784.e260

- Floating-point division and square root, 784.e147f
 - algorithm for unsigned integer division, 784.e201–784.e202
 - CPA delay, 784.e202, 784.e202
 - divider latency, 784.e218–784.e219
 - early termination, 784.e221–784.e222
 - integer division, 784.e224–784.e225
 - examples, 784.e230–784.e233
 - integer latency, 784.e226–784.e227
 - integer postprocessing, 784.e227–784.e228
 - integer preprocessing, 784.e225–784.e226
 - minimally redundant radix-4, 784.e202
 - postprocessing, 784.e223
 - exponent, 784.e223–784.e224
 - normalization, 784.e224
 - rounding, 784.e224
 - sign, 784.e223
 - special cases and flags, 784.e224
 - quotient conversion, 784.e219–784.e220
 - radix-2 division
 - examples, 784.e208–784.e209
 - implementation, 784.e211f
 - quotient digit set, 784.e206
 - quotient selection optimization, 784.e209–784.e212
 - WC and WS, 784.e208
 - radix-4 division
 - example, 784.e216
 - implementation, 784.e216
 - minimally redundant divisor, 784.e212
 - P-D diagram, 784.e213, 784.e213
 - QSLC constants, 784.e215t
 - quotient digit, 784.e214–784.e215
 - recurrence division algorithm, 784.e202–784.e204
 - containment of partial remainder, 784.e205–784.e206
 - examples, 784.e204
 - quotient digit set, 784.e204–784.e205
 - quotient representation, 784.e206
 - redundant partial remainder, 784.e204–784.e205
 - square root. *See* Floating-point square root
 - subnormal floating-point preprocessing, 784.e222–784.e223
- Floating-point multiply-accumulate
 - add aligned significands, 784.e195–784.e198
 - addend signs, 784.e193
 - add exponents, 784.e193–784.e194
 - alignment shift, 784.e194–784.e195
 - leading zero anticipator, 784.e198–784.e200
 - multipath FMA optimization, 784.e200
 - multiply significands, 784.e193
 - product signs, 784.e193
- Floating-point postprocessing, 784.e149f
 - conversion, 784.e265t
 - correction shift, 784.e266t
 - initial result mux, 784.e265t
 - normalization shift, 784.e265t
 - rounding, 784.e267
 - shift calculation, 784.e262
 - conversion, 784.e264
 - FMA, 784.e263t
 - unified divide/square root, 784.e264t
 - sign logic, 784.e267t
 - special cases and flags, 784.e268–784.e271
 - other flags, 784.e271–784.e272
 - result selection, 784.e272
- Floating-point square root
 - addend input selection, 784.e238–784.e241
 - ASEL table, 784.e238t
 - containment analysis, 784.e235
 - containment of residual, 784.e235–784.e236
 - early termination, 784.e242–784.e243
 - on-the-fly conversion, 784.e241–784.e242
 - operation, 784.e233, 784.e233
 - postprocessing, 784.e247–784.e248
 - radix-2 example, 784.e243
 - radix-2 implementation, 784.e245
 - radix-2 square root digit selection, 784.e237
 - radix-4 example, 784.e245
 - radix-4 implementation, 784.e247
 - radix-4 square root digit selection, 784.e237–784.e238
 - recurrence algorithm, 784.e234–784.e235
- Floating-point units (FPUs), 198, 239–240, 580
 - design
 - comparison, 640–641
 - conversion, 639–640
 - division and square root, 637–639
 - fdivsqrt unit, 630
 - fused multiply-add, 633
 - postprocessing unit, 641
 - unpack unit, 631–633
 - generic block diagram, 630
 - IEEE 754
 - infinity, 620
 - Not a Number, 621
 - rounding, 625
 - special cases, 620t
 - subnormal numbers, 621–622
 - zero, 620
 - principles
 - fixed-point notation, 628–629
 - flags, 616.e10–616.e12
 - normalized floating-point formats, 618–619
 - RISC-V practices. *See* RISC-V practices. *See* FPUs
 - FMA special case handling, 784.e269t
- Formal verification (FV), 195
 - algorithmic improvements, 227
 - cone of influence, 227
 - definition, 226
 - OpenHW Group, 227
 - Forwarding logic, 324–325, 325f
 - fpcalc, 643
 - FPGA Hardware, 784.e81t
 - Free and Open Source Silicon Foundation (FOSSi), 784.e27–784.e28
 - FreeRTOS, 784
 - found algorithm, 664
 - found unit, 665
 - fsgn sign injection, 662
 - Fulladder simulation waveforms, 208f, 212f
 - Fully associative cache, 437–438
 - Functional coverage, 195, 245–246
 - architectural, 221
 - constrained random tests, 226

covergroup and coverpoint
statements, 222
directed tests, 222–226
DV plan, 221
measurement, 222
riscvISACOV, 214t, 222, 239f
Futex, 693–694

G

Galois fields, 717
General-purpose input/output (GPIO)
peripheral, 383–384
controls, 784
loopback testing, 784.e1
memory map, 784t
pins connected to switches and LED,
784f
test plan, 784.e1
top-level interface, 784–, 784.e1f
Wally implementation
peripherals highlighting,
784, 784f
RV32 configuration, 784.e1–
set/clear logic, 784, 784f
General-purpose operating systems, 784
GF(2_n), 718–722, 719t
GF(p), 717
Git
basic flow, 784.e172–784.e175
Ben’s workspace,
784.e181, 784.e183
branching and merging,
784.e176–784.e177
capabilities, 784.e181
features, 784.e170
HEADS, 784.e175
layers of data storage, 784.e182f
merge conflicts, 784.e175–784.e176
repository, 784.e170–784.e171
repository setting, 784.e172
snapshot,
784.e170–784.e171,
784.e175
staging and undo,
784.e178–784.e179
starting account, 784.e171–784.e172
submodules, 784.e179–784.e180
tags, 784.e177
version control tool, 784.e181

Git file states, 784.e179f
GitHub account, 784.e171
Git layers, 784.e200f
Global history shift register (GHR),
552
Globally unique identifier partition
table (GPT), 784.e84
GNU Compiler Collection (GCC)
ABI, 95–99, 96t, 97f
CLINT, 99
compiler explorer, 123, 123f
definition, 91
driving, 99
first C program, 115–120, 120t
inline assembly language, 120
linking and ELF, 93–94
mixed C and assembly language, 121
quick tutorial, 91–93, 92b
SBI, 99
standard library and startup code,
122–123
Good synchronizer, 144
Gshare, 551–552, 564–568
GTKWave, 212f

H

Handshaking, 386, 690
Hardware description language (HDL),
141,
201. *See also* Sequential logic
combinational logic. *See*
Combinational logic
design strategies, 142–144
reuse, 144–145
with systemverilog. *See* Systemverilog
version control, 144
Hardware emulation, 198–199
Hardware page table walkers
(HPTWs), 475, 515
integration, 531–532
Hardware thread, 45
Harts, 47–49, 48f
Hazard unit (hazard),
273f,
325–327,
326f–327f, 784.e129f
High-speed off-chip links, 390–391
Hitchhiker’s guide, Linux
commands

bundling, 784.e161
cast of characters, 784.e157
commonly used, 784.e158
compressing, 784.e161
standard input and output,
784.e157
transferring files, 784.e161
working with process,
784.e159–784.e160
productivity
document viewing and printing,
784.e165
environment variables, 784.e164
installing packages, 784.e165
Linux file system,
784.e163–784.e164
multiple terminal windows,
784.e161–784.e165
text editors, 784.e162–784.e163
web browsing,
784.e164–784.e165
scripting and programming
c programming, 784.e167
makefiles, 784.e167–784.e169
python, 784.e166–784.e167
shell scripts, 784.e165–784.e166
server connection,
784.e153–784.e154
password change, 784.e154
X2Go, 784.e153–784.e154
working with files, 784.e154

Hold time analysis, 254
Hold time constraint, 251
Host target interface (HTIF), 106
HPTW critical path
after optimization, 784.e104f
before optimization, 784.e103f
HPTWs. *See* Hardware page table
walkers (HPTWs)
HTRANS transfer types, 395t
Huge pages, 486–488

I

IEEE 754 Floating-Point Format,
617
IFU. *See* Instruction fetch unit (IFU)
Illegal-instruction exception, 352–353
Immediate assertions test, 215
ImperasDV, 213

lock-step-compare system, 213f
 In-circuit logic analyzer,
 784.e93–784.e96, 784.e86
 Init process, 784
 Inline assembly language, 120
 Input/output elements (IOEs), 784.e79
 Input/output (I/O) pins, 784.e78
 Institute for Electrical and Electronic
 Engineers (IEEE), 617
 Instruction address-misaligned
 exception, 353
 Instruction cache, 429
 Instruction class prediction, 784.e40
 Instruction fetch unit (IFU),
 56–57,
 198,
 241–242, 314, 377–378, 784f
 block diagram, 559f
 branch prediction. *See* Branch
 prediction principles
 signal declarations, 184b
 test plan, 558–559
 correctness, 558
 performance, 558–559
 Wally high-level block diagram, 546f
 Wally implementation
 branch predictor. *See* Branch
 predictor
 instruction cache, 560
 ITLB misses, 574
 PC logic, 560
 reset, stalls, flushes, and traps,
 574–576
 Instruction Read-Only Memory
 (IROM), 52, 56–57, 299
 Integer divider, 784.e143f
 Integer execution unit (IEU),
 56–57, 198, 784f
 Integer and floating-point registers,
 212–213
 Integer instructions, 300t
 Integer multiplier, 784.e142f
 Integer multiply/divide unit, 784.e142f
 Integer recurrence division
 preprocessing, 784.e226f
 Integrated circuit fabrication
 technologies
 chip manufacturers, 258
 Cypress Semiconductor, 259
 inverter characteristics,
 259–260, 259t
 NDA, 258–259

 28-nm node, 258–259
 planar processes, 257–258, 258f
 SKY130, 259
 Intellectual property (IP),
 784.e79–784.e80
 Interfaces, 445–446
 Interrupt enable register (IER),
 784.e16–784.e17
 Interrupts, 49, 354–355, 466–467
 triggering and generation, 355–357
 vectoring, 357
 Inverse cipher, 729–731
 InvMixColumns matrix, 731f
 I-type instructions, 34

J

J-type instructions, 35
 Jump and link register, 34
 Jump vector table (JVT), 581

K

Kernel, 784
 KeyExpansion
 example, 728t
 key schedule, 727f
 unit variables, 727t

L

Least recently used (LRU) replacement
 policy, 439
 Library considerations, 167–168
 Line control register (LCR),
 784.e16–784.e17
 Line fetch, 452
 Line status register (LSR),
 784.e16–784.e17,
 784.e26
 Line writeback, 452
 Linking, 34
 Lint, 203–204

Linters, 203
 Linux-based execution environments,
 50f
 Linux boot messages, 784.e3f
 Linux operating system (OS), 784.e48
 file
 binary, 784
 boot trace, 784.e3–784
 disassembled, 784.e3
 locations, 784t
 memory, 784.e3
 Hitchhiker's guide. *See* Hitchhiker's
 guide, Linux
 Livelock, 698
 Load/store unit (LSU),
 56–57,
 198,
 241–242, 314, 315f, 784.e130f
 block diagram, 517f
 cache, AHB cache interface, and EBU
 connections, 519f
 cache load hit, 534, 536–537
 cache store hit, 534–537
 data cache integration. *See* Data
 cache integration
 disabling interrupts, 533
 full MMU and TLB address
 translation, 530f
 memory operations, 529t
 response to stalls and flushes,
 533–534
 stall generation, 532–533
 test plan, 544
 TLB access fault, 543–544
 trap generation, 533
 uncached access, 537–540
 Wally LSU overview, 515–518
 Locality, 432
 Local repository\, 784.e170–784.e171
 Locks, 684–686
 Lock-step-compare, 240–241
 Log files
 area reports, 284, 285f
 mapped output files, 270–273, 271f
 other reports, 284
 performance reports,
 277–282, 277f–278f
 power reports, 282–284, 283f
 quality of results (QOR) reports,
 275, 276f
 synth.out, 269, 271f
 Logic design engineer, 254

Logic elements (LEs), 784.e78
 Logic simulators, 200t, 207–212
 Logic synthesis, 249
 Logic verification process
 definition, 193
 device under test (DUT), 193
 Lookup table (LUT), 784.e79
 Loopback testing, 784.e1
 Loop grade inflation, 41b
 LSU. *See* Load/store unit (LSU)

M

Machine architecture, 99–100, 100t
 Machine information registers, 346
 Machine language, 43–45, 44f
 MacOS, 784
 Main decoder, 312, 313t
 Market, RISC-V, 80, 81t–84t
 Maximum and minimum integer
 values, 784.e270t
 Memories, 160–168
 block characteristics, 266–267, 267t
 consistency models, 677
 exceptions, 494–495
 files, 784.e3
 hierarchy, 429
 Memory management and HPTW
 integration
 HPTW integration, 531–532
 PMA checker, 527–529
 PMP checker, 529
 TLB and cache integration, 529–531
 Memory management unit (MMU),
 241–242, 784.e132f
 definition, 469
 principles
 physical memory checking,
 477–478
 virtual memory, 469–477
 RISC-V. *See* RISC-V practices
 test plan, 495
 Wally implementation. *See* Wally
 implementation
 Memory management units (MMUs),
 784
 Memory map, 51–52, 51f, 386
 M extension
 principles
 carry-save addition, 592–593

division. *See* Division
 multiplication. *See* Multiplication
 signed arithmetic, 591–592
 unsigned arithmetic, 591–592
 riscv-arch-test, 606, 606t
 Wally implementation
 division and remainder,
 610–612, 611f
 hazard unit, 614, 615f
 multiplication, 607–610
 multiply and divide Unit,
 613, 614f
 Microarchitecture
 CPU and CPI, 56
 HDL code, 56
 single-cycle, 56–61, 58f
 Min/max special case handling,
 784.e270t
 Misprediction penalty, 545–547
 Miss penalty, 430–431
 MixColumns operation,
 725–726, 726f
 Mixed C and assembly language, 121
 Mixed-mode clock manager, 784.e81
 Modem control register (MCR),
 784.e16–784.e17
 Modem status register (MSR),
 784.e16–784.e17
 Modified Baugh-Wooley signed
 multiplier, 597f
 Modified booth encoding, 597
 Modules, 188–189
 Moore's law, 784.e79
 Multiple controllers, 387–389
 Multiple-input adders, 592f
 Multiple peripherals, 386–387
 Multiplication, 593–601
 array multiplier, 594
 booth encoding, 594, 597–598
 dot diagram, 594f
 logical AND, 593
 multiplier synthesis, 601
 other techniques, 601
 signed multiplication, 596
 tree multipliers, 594, 599
 unsigned, 593
 Multiplicative inverse table, 723f
 Multiplier dot diagram, 594f
 Multiplier synthesis, 601
 Multiported RAMs, 166
 Multithreaded/multicore processor,
 45

N

National Aeronautical and Space
 Administration (NASA),
 194
 Naturally aligned power-of-two
 (NAPOT), 784.e63
 Network on Chip (NoC), 390
 New User Setup and Toolchain
 Installation, 91
 Nightly build, 234–235
 Nightly regression, 234
 Nonrecurring engineering costs
 (NREs), 784.e78
 Nonvolatile program memory, 53
 Normalized floating-point formats,
 618
 N-way set associative cache, 436–437

O

On-chip peripheral bus (OPB), 410
 One-bit with many-bit gating, 170
 On-the-fly conversion (OTFC),
 784.e219f
 OpenEmbedded system, 784
 Open Hardware Manager, 784.e89f
 OpenHW Group, 200t, 227
 OpenSBI Firmware, 784.e60–784.e61
 PLIC initialization, 784.e62–784.e63
 PMP configuration, 784.e63
 UART initialization, 784.e60–784.e62
 Open-source build systems, 784
 Open Target, 784.e89f
 Operands, 299
 Operating system (OS), 784.e84
 OpenWrt, 784

P

Page-based memory types (Svpbmt),
 490–491
 Page coloring, 476–477
 Page table entry (PTE), 471–472
 Partial store order, 679–680
 Pattern history table (PHT), 552

- PC16550, 784.e16–784.e17
 - Performance measurement, 107
 - Performance, power, and area (PPA), 249–250
 - activity factor, 256
 - clock period and frequency, 250–251
 - datapath units, 260–266, 262f, 262t, 263f, 265t
 - dynamic and leakage power, 255
 - dynamic energy, 256
 - setup time constraint, 251
 - static timing analysis, 253–255, 255f
 - Synopsys Design Compiler, 257
 - time constraints, 251–253, 251f–253f
 - Peripherals
 - core local interruptor, 784
 - general-purpose input/output, 784
 - memory-mapped I/O, 784.e23
 - platform-level interrupt controller, 784
 - serial peripheral interface, 784.e1
 - testing, 784.e1
 - universal asynchronous receiver/transmitter, 784.e1
 - Personal computers (PCs), 784
 - Phase-locked loop (PLL), 391
 - Physical addressing, 469–470
 - Physical design engineer, 254
 - Physical harts, 45
 - Physical memory attributes checker, 784.e134f
 - Physical memory checking, 477–478
 - Physical memory protection (PMP), 784.e60
 - Physical memory protection checker, 784.e135f
 - Physical page number (PPN), 471–472
 - Pipelined microarchitecture, 65–67, 66f
 - Pipelined processor performance, 327–328
 - Pipeline hazards
 - CSR reads, 375–376
 - CSR writes, 376
 - description, 321–322
 - fences, 376
 - hazard example, 322–323
 - mret and sret, 375
 - precise traps, 374
 - WFI, 376–377
 - Pipeline registers, 323, 323f
 - Pipeline stalls, 466
 - Planar processes, 257–258
 - Platform-level interrupt controller (PLIC) peripheral, 784
 - hart context, 784
 - interrupt pending, 784
 - memory-mapped registers, 784.e1t
 - M- mode interrupt, 784
 - S- mode interrupt, 784
 - test plan, 784
 - Wally implementation, 784.e1
 - pLRU implementation, 456–459
 - PMA checker, 491–494, 501–502
 - memory management and HPTW integration, 527–529
 - page-based memory types (Svpbmt), 490–491
 - PMP checker, 490–491
 - regions, 493t
 - registers, 784.e63
 - Port connections, 171–172, 184–185
 - Positive edge-triggered flip-flop, 154
 - Power gating, 257
 - PPN. *See* Physical page number (PPN)
 - Precise traps, 352
 - Prefix comparators, 311f
 - Preprocessor, 172–173
 - Priority circuit, 151f
 - Privileged instructions
 - CSR access instructions, 341
 - exception generating instructions, 341–342
 - trap return instructions, 342
 - wait for interrupt, 342
 - wally implementation, 370
 - Privileged operations
 - instruction fetch unit and privileged changes, 377–378
 - principles
 - control and status registers (CSRs), 338
 - security and protection, 336–337
 - traps, 337–338
 - Privileged specification, 32
 - Privileged test plan, 360–362
 - generating exceptions, 362
 - generating interrupts, 363
 - Privileged unit, 784.e137f
 - Privilege modes, 46
 - Processes and threads, 46–47
 - Process ID (PID), 476
 - Process variation, 260, 261t
 - Producerconsumer/bounded buffer, 689–690
 - Product's and sum's exponents, 784.e194t
 - Product significand outputs, 784.e193t
 - Profile extensions
 - Za128rs and Za64rs, 706
 - Ziccamoa, 705
 - Ziccif, 705
 - Ziccrse, 705
 - Program counter (PC), 212–213
 - Programmable logic array (PLA), 196–197
 - Pseudoinstructions, 37–38
 - Pseudo least recently used (pLRU), 456–459
 - PTE. *See* Page table entry (PTE)
 - PTXdist, 784
 - Pull request, 784.e171
 - Python, 784.e166–784.e167
- Q**
- QEMU simulation
 - commands, 128t
 - open-source emulator, 127
 - run options, 96t
 - simulation of, 127–128
 - Quality of results (QOR) reports, 275, 276f
 - Questa GUI, 207–209
 - Quick Emulator (QEMU), 784.e48
 - Quotient selection constants (QSLC)
 - preferred radix-4 division, 784.e215t
 - viable radix-4 division, 784.e215t
- R**
- Radix-2 adder input generation, 784.e239t
 - Radix-4 Booth encoder, 599f

- Radix-2 division example, 784.e210f
- Radix-2 division P-D diagram, 784.e207f
- Radix-4 OTFC implementation, 784.e221f
- Radix-2 restoring unsigned integer division, 602–603, 603f
- Random access memory (RAM), 163–164, 163f
- RAS. *See* Return address stack (RAS)
- Read after write (RAW) data hazard, 322–323
- Read-only memory (ROM), 162–163, 162f
- Read/write port, 161
- Real-time operating system (RTOS), 784
- Receiver buffer register (RBR), 784.e1
- Recurrence division, 784.e205t
 - algorithm, 784.e205f
 - hardware, 784f
- Reduced instruction set computer (RISC)
 - architectures, 579
 - C extension
 - decompress block diagram, 584f
 - floating-point loads, 580
 - instruction decompression, 582–585
 - PC logic, 585–586, 586f
 - riscv-arch-test suite, 582, 583t
 - spills, 586–590, 586f, 588f–589f
 - Wally implementation, 582–590
 - Wally RV32IC/64IC, 584f
 - Zc* extension, 580–582
 - compressed instruction format, 579–580
 - compressed instruction maps, 580
 - compressed instruction sets, 579b
 - spill, 579
- Reference models, 198, 229
- Register, 784.e79
- Register transfer level (RTL), 249
- Relaxed ordering, 679–680
- Release consistency model, 674
- Relocation, 469–470
- Replacement policy, 439–441
- Reset behavior, 349
- Resettable flops, 155–156, 156f
- Reset vector, 52
- Residual sign correction and shift, 784.e228t
- Result multiplexers, 658
- Result selection logic, 784.e273t
- Return address stack (RAS), 547, 556–557
- RISCOF
 - definition, 137
 - Wally RISCOF test suites, 137, 138t
 - wally-riscv-arch-test, 138–140, 139f
- RISC-V
 - bit manipulation
 - test plan, 748
 - cryptography (Zk*), 715
 - M extension. *See* M extension
 - practices
 - division special cases, 604–605
 - signed operations, 604
 - software arithmetic implementations, 605–606
 - unsigned operations, 604
 - Zmmul, 606
- riscv-arch-test, 133–136, 134f, 331t–332t
- RISC-V atomic instructions
 - practices
 - atomic instructions, 700–704
 - atomic subsets, 705
 - consistency models, 698–699
 - fences, 699–700
 - profile extensions, 705–706
- principles, 673–698
 - coherence, 676
 - concurrency, 674–676
 - concurrent systems, 673
 - consistency, 677–682
 - deadlock, 696–697
 - livelock, 698
 - release consistency model, 674
 - starvation, 698
 - synchronization, 674, 682–696
- test plan, 706
- Wally implementation
 - atomic module, 708–710
 - timing diagrams, 711–712
- RISC-V International (RISC-V)
 - Foundation, 1–2, 26
- architecture, 32
- assembly language
 - B-type instructions, 35
 - C to, examples, 38–43
 - endianness, 37, 37f
 - instruction types, 32, 33t
 - I-type instructions, 34
 - J-type instructions, 35
 - pseudoinstructions, 37–38
 - register conventions, 37
 - R-type instructions, 32–34
 - RV64I additional instructions, 36
 - S-type instructions, 34
 - U-type instructions, 35–36
- CSRs, 31
- extensions and profiles
 - common extensions, 52, 52t
 - RISC-V profiles, 53–54
- load/store ISA, 31
- machine language, 43–45, 44f
- microarchitecture. *See* Microarchitecture
- microprocessors, 32
- vs.* other architectures, 54–55
- processor design, 31
- width XLEN, 31
- riscvISACOV, 214t, 222, 239f
- RV32I add covergroup, 224f
- RISC-V practices
 - carry-less multiplication, 738
 - crossbar permutations, 741
 - fences, 495
 - FPU
 - control and status register, 647f
 - move instructions, 649
 - NaNs, 621
 - registers and NaN boxing, 648
 - Zfa additional instruction
 - fcvtmod.w.d, 651
 - fli, 651
 - fltq and fleq, 651
 - fminm/fmaxm, 651
 - fmvh and fmvpl, 651
 - fround, 651
 - froundnx, 649–651
- hardware page table walker (HPTW), 495
- memory exceptions, 494–495
- memory protection, 489–494
 - PMA checker, 489–491
 - PMP checker, 491–494
- privileged operations
 - CSRs. *See* Control and status registers (CSRs)
 - privileged instructions, 340–342
 - privilege modes, 340
 - reset behavior, 349
 - trap handler, 357
 - traps, 350–357
- single-bit operations, 739

- systems
 - core local interruptor, 784.e23
 - platform-level interrupt controller, 784.e23
 - virtual memory, 478–489
 - dirty and accessed bits, 488
 - mstatus/sstatus register, 488–489
 - satp (supervisor address translation and protection) register, 478–479
 - variable page sizes, 483–488
 - virtual address and page table structure, 479–482
 - Zba, 734–735
 - Zbb, 736–738
 - counting, combining, reversing, 737–738
 - extension, 737
 - inverted operations, 736
 - Max/Min, 736–737
 - rotates, 736
 - Zbkb, 739
 - bit interleaving, 740–741
 - bit reversal, 740
 - packing, 740
 - Zknd and Zkne, 742–747
 - RVZknd: AES Decryption, 746–747
 - RV32Zkne: 32-bit AES Encryption, 745–746
 - RV64Zkne: 64-Bit AES Encryption, 743–745
 - RV64Zkn AES Key Scheduling, 747
 - Zknh, 747–748
 - Root file system, 784.e51–784
 - compressed, 784.e12
 - overlay, 784
 - using BusyBox, 784.e3f
 - Rounding logic, 784.e268t
 - Round to nearest ties away, 622
 - Round to nearest ties to maximum magnitude (RNM), 622
 - Round toward zero, 622
 - R-type instructions, 32–34
 - Running synthesis, 268–269
 - Run Spike, 107
 - RV64gc, area and delay *vs.* features, 295–297, 296f, 297t
 - RV32GC/RV32IFDC extension, 582
 - RVVI-TRACE Interface, 206t, 214
 - RVVI-TRACE standard interface (RVVI-TRACE), 213
- ## S
- Sail specification
 - definition, 124
 - RISC-V simulators, 124
 - Sample program, 57–59
 - S-box contents, 725f
 - Scratch register (SCR), 784–
 - Second-level cache, 429
 - Security and protection, 336–337
 - Self-checking testbench, 204f
 - Self-modifying code, 676
 - Semaphore-based mutex, 692
 - Semaphore-based producer-consumer, 693
 - Sequential consistency, 678
 - Sequential logic
 - bidirectional I/O, 154f
 - definition, 153–154
 - flip-flops, 154–157, 154f–155f
 - Sequentially consistent operation, 681
 - Serial clock (SCK), 784.e1
 - Serial communication, 784.e14
 - Serial data in (SDI), 784.e1
 - Serial data out (SDO), 784.e1
 - Serial peripheral interface (SPI), 784.e1
 - A/D converter interfaced, 784.e22f
 - controller-peripheral interface, 784f
 - loopback testing, 784
 - memory-mapped registers, 784–, 784.e1t
 - PC16550 block diagram, 784.e1f
 - physical interface, 784.e1
 - SiFive FE540, 784
 - signaling, 784f
 - test plan, 784.e23
 - top-level interface, 784f
 - Wally implementation, 784.e23
 - Setup time constraint, 251
 - SHA-2 algorithm, 733f
 - Shell, 784
 - Shell scripts, 784.e165–784.e166
 - Shifter, 306–307, 306f
 - Shift operations, 33
 - ShiftRows operation, 724–725, 726f
 - Signature checking, 212–214
 - Signed arithmetic, 591–592
 - Signed division, 604t
 - Signed multiplication, 596
 - Signed multiplier
 - modified Baugh-Wooley, 597f
 - partial products, 596f
 - simplified partial products, 596f
 - Sign injection, 616.e44t
 - Sign outputs, 784.e194t
 - Simulated harts, 45
 - Simulator, 51
 - Single-cycle core, 301–317, 302f
 - add/subtract, 305
 - architectural state, 302–303
 - arithmetic circuitry, 303–311, 304t
 - comparator, 308–311
 - immediate extension, 303
 - set less than, 308
 - shifter, 306–307, 306f
 - W-type instructions, 308
 - Single-cycle microarchitecture, 56–61, 58f
 - Single-precision
 - Single-threaded processor, 45
 - Sleep locks, 689
 - Snapshot, 784.e170–784.e171
 - SoC, 78–79, 79f
 - SoftFloat, 641–646
 - fpcalc, 643–646
 - softFloat demo, 643
 - Software interrupts, 784
 - Software threads, 46–47
 - Software tool flow
 - requirements and tool overview, 90
 - RISC-V toolchain, 89f
 - Spacing and indentation, 185–187
 - SPARC, 55
 - Spike simulation, 105–107, 106t
 - Spill
 - 32-byte cache line, 586f
 - IFU, 587f
 - instruction fetch, 587
 - module, 588f
 - spillfsm, 589f
 - timing diagram, 589f
 - Spin lock, 686
 - Split transfer, 389
 - Square root special case handling, 784.e269t

- SRAM interface, AHB, 400–401
 - Standard cell library, 249
 - Standard library and startup code, 122–123
 - Starvation, 698
 - Static branch prediction, 548
 - Static random access memory (SRAM), 266
 - Static timing analysis, 253–255, 255f
 - Status bits, 435
 - Status register, 343–346
 - Store address-misaligned exception, 353
 - Strongly Not Taken (SNT), 158
 - Strongly Taken (ST), 158
 - Structural hazards, 451–452
 - S-type instructions, 34
 - SubBytes operation, 724, 725f
 - Sub-cacheline read (SCR) module, 534
 - Subnormal numbers, 621–622
 - Subword loads, 34
 - Supervisor address translation and protection (satp) register, 478–479
 - Supervisor binary interface (SBI), 50, 99
 - Svinval TLB management extension, 777–779
 - Symmetric-key encryption, 716f
 - Synchronization, 673, 682–696
 - futex, 693–694
 - mutual exclusion
 - atomic instructions, 686–688
 - better locking, 686
 - locks, 684–686
 - sleep locks, 689
 - ticket locks, 688–689
 - producer-consumer: condition
 - variables, 689–692
 - semaphores, 692
 - semaphore-based mutex, 692
 - semaphore-based producer-consumer, 693
 - synchronization Libraries, 695–696
 - Synchronous design
 - memories and flip-flops, 187
 - sequential, 187
 - Synchronous memories, 319–321, 320f, 320t
 - Synonym problem, 476
 - Synopsys Design Compiler, 141, 257
 - Synopsys VCS, 212
 - Synthesis configurations, 287, 288t–289t
 - SystemC, 142
 - System-on-chip (SoC), 784.e48
 - address decoding and device Mux, 418–420
 - designs, 784.e185
 - SystemVerilog,
 - 61, 61b–65b, 141–142, 328–329
 - behavioral HDL (RTL) code, 145
 - code, 784.e78
 - definition, 145
 - hierarchically/structurally, 145
 - language subsets, 194t
 - modules, 145–148, 146f
 - SystemVerilog Assertions (SVAs), 214
- T
- Tcl
 - advanced commands
 - Aliases, 784.e192
 - control flow, 784.e189–784.e191
 - display data on screen, 784.e189
 - exiting Tcl, 784.e192
 - file, 784.e189
 - file-related, 784.e189
 - Glob, 784.e189
 - lists, 784.e188
 - procedures, 784.e191
 - running Tcl scripts, 784.e191–784.e192
 - embeddable scripting language, 784.e185
 - system-on-chip designs, 784.e185
 - use, 784.e185–784.e187
 - arguments and variables, 784.e186–784.e187
 - command syntax, 784.e187
 - nested commands, 784.e187
 - Technology readiness levels (TRLs), 194, 199, 200t
 - Testbenches, 204–206, 208f
 - Test-driven development, 202
 - TestFloat test cases, 654
 - Test plan, 329–333
 - riscv-arch-test, 651–652
 - TestFloat, 652–653
 - Test suites
 - architectural test pool, 134
 - definition, 133
 - riscv-arch-test, 133–136, 134f
 - test file structure, 135–136
 - Test vector generation, 196–197
 - Thrashing, 436
 - 32-Bit shifter source generator, 308f
 - 3-Bit register, 154f
 - Ticket locks, 688–689
 - Tightly integrated memory (TIM), 545
 - TileLink, 410
 - TIM. *See* Tightly integrated memory (TIM)
 - Time division multiple access (TDMA), 388
 - Timer interrupts, 784–
 - Tizen, 784
 - Toggle rates, 256
 - Tool Control Language (TCL), 250
 - Top-level cache logic
 - address, way, and cache line read, 460
 - cache flush logic, 463–466
 - read selection and bus interface logic, 462–463
 - write selection logic, 462
 - Top-level testbench and testvectors, 229–230
 - Total negative slack (TNS), 254
 - Total store ordering (TSO), 678–679
 - Tournament branch predictor, 557
 - Translation lookaside buffer (TLB), 474–475, 784.e133f
 - access fault, 543–544
 - and cache integration, 529–531
 - miss, 540–542
 - Transmit holding register (THR), 784.e26
 - Transparent/D latch, 157, 157f
 - Trap handler, 49, 357
 - external interrupts, 784
 - M-mode interrupts, 784.e12
 - S-mode interrupts, 784.e69–
 - software interrupts, 784
 - timer interrupts, 784.e69–784.e70
 - Trap return instructions, 342
 - Traps, privileged operations, 371–373
 - exceptions, 352–354
 - interruption, 354–357
 - precise traps, 352
 - Trap unit, 784.e138f
 - Tree multipliers, 594, 599

Tree pseudo-LRU, 440
 Troubleshooting, 784.e38
 Trunc multiplexer, 308f
 Trusted execution environments (TEEs), 784.e60
 TSMC28 ROM generators, 266
 Two-bit branch predictor state transition diagram, 158f–159f, 159b

U

UART initialization, 784.e62, 784.e62t
 Unbiased rounding, 622
 Uncached access, 537–540
 Unified Coverage Database (UCDB), 217–218
 Unified divide and square root
 addend generation, 784.e248
 bit widths, 784.e252–784.e254
 implementation, 784.e254–784.e256
 multiplicative algorithms, 784.e256–784.e257
 on-the-fly conversion, 784.e251
 postprocessing, 784.e251–784.e252
 preprocessing, 784.e248
 unified digit selection, 784.e248
 Unit in the last place, 623
 Unit *vs.* system testing, 198
 Universal asynchronous receiver/transmitter (UART)
 asynchronous, 784
 PC16550 register map, 784–, 784.e1f
 receiver buffer register, 784.e1
 RISC-V Linux, 784.e1
 SPI and I2C serial links, 784.e1
 system clock frequency, 784.e1, 784.e1
 test plan, 784
 timing, 784f
 top-level interface, 784, 784f
 transmit holding register, 784.e1
 VT100 text, 784f
 Wally implementation, 784.e1
 waveforms, 784
 Universal Asynchronous Receiver/Transmitter (UART), 784.e1, 784.e83
 Unpacking signals, 632

Unpack unit, 784.e145f
 Unprivileged Specification, 32
 Unreachable logic exclusion, 222–226
 Unsigned arithmetic, 591–592
 Unsigned multiplication, 593f
 Upstream repository, 784.e171
 U-type instructions, 35–36

V

Vectored interrupts, 357
 Vendor-specific electronic design automation tools, 784.e79–784.e80
 Verification strategies, 67–69, 70f, 76, 76t
 definition, 195–196
 design and verification approaches, 200–202
 emulation, 198–199
 reference models, 198
 test vector generation, 196–197
 TRL, 199, 204f
 unit *vs.* system testing, 198
 Verilator, 210–211
 Verilog Change Dump, 206
 Verilog syntax
 bit swizzling, 169
 delays, 170–171
 numbers, 168, 169t
 one-bit with many-bit gating, 170
 port connections, 171–172
 preprocessor
 define, 172
 include, 172
 Version control, 144
 Virtual harts, 45
 Virtual memory
 interaction of processes, virtual memory, and caches, 476–477
 multilevel page tables, 473–474
 page faults, 472–473
 pages, 471–472
 physical addressing, 469–470
 process, 469
 protection, 473
 translation lookaside buffer, 474–475

variable page size, 475–476
 Virtual peripherals, 784.e48
 Vivado, 784.e86
 VxWorks, 784

W

Wallace tree multipliers, 601
 Wally
 additional test, 784.e126t
 AHB and APB interfaces, 784.e136f
 configurations, 784.e108t
 additional parameters, 784.e113t, 784.e115t
 derivative, 784.e115t, 784.e118t
 key parameters, 784.e110t, 784.e112t
 default rvgc memory map, 784.e108t
 directory structure, 784.e107f
 key Wally commands, 784.e109, 784.e109
 PC logic, 784.e128f
 regression and test suites, 784.e109
 regression tests, 784.e125t
 repository, 784.e106
 RISC-V extensions and profiles, 784.e118t, 784.e124t
 test suites, 784.e125t, 784.e126t
 Wally block diagram, 73, 74f
 Wally code base, 74
 Wally configurations, 73, 190–191
 Wally diagrams
 AHB and APB interfaces, 784.e136f
 ALU with bit manipulation and cryptograph enhancements, 784.e140f
 atomic memory operations, 784.e135f
 basic bit manipulation, 784.e141f
 branch prediction unit, 784.e128f
 cache, 784.e131f
 carry-less multiplication, 784.e141f
 external bus unit, 784.e136f
 floating-point compare unit, 784.e148f
 floating-point convert unit, 784.e148f
 floating-point divide and square root unit, 784.e147f

- floating-point fused multiply-add unit, 784.e146f
- floating-point postprocessing unit, 784.e149f
- floating-point unit, 784.e144f
- hazard unit, 784.e129f
- high-level, 784.e109–
- instruction fetch unit, 784f
- integer divider, 784.e143f
- integer execution unit, 784f
- integer multiplier, 784.e142f
- integer multiply/divide unit, 784.e142f
- load/store unit, 784.e130f
- memory management unit, 784.e132f
- PC logic, 784.e128f
- physical memory attributes checker, 784.e134f
- physical memory protection checker, 784.e135f
- privileged unit, 784.e137f
- top-level schematic, 784.e127f
- translation lookaside buffer, 784.e133f
- trap unit, 784.e138f
- uncore unit, 784.e139f
- unpack unit, 784.e145f
- Wally device tree, 784.e3f
- Wally implementation
 - block diagram, 497–507
 - bus interface
 - bus hazards, 423–426
 - combined bus and tightly integrated memories, 420–421
 - external bus unit, 415–417
 - LSU and IFU AHB interface, 412–415
 - system-on-chip, 417–420
 - classify, 662–663
 - compare, 662
 - convert, 661
 - core local interruptor, 784
 - divide and square root, 660–661
 - division and remainder, 610–612
 - fence instruction, 511–512
 - floating-point unit
 - register file, 658
 - result and bypass multiplexers, 658–659
- FPGA
 - Arty A7. *See* Arty A7 board
 - external bus connection, 784.e83
 - hardware required, 784.e80
- mixed-mode clock manager, 784.e81
- SD card, 784.e83–784.e84
- SPI peripheral, 784.e83
- supporting other boards, 784.e96–784.e99
- uses IP, 784.e81
- Xilinx SC7A100TCSG324-1, 784.e80
- zero stage bootloader, 784.e99
- general-purpose input/output, 784
- hazard unit, 614
- hazard unit changes, 667
- HPTW
 - block diagram, 508
 - datapath, 508f
 - PTE status bit updates, 510–511
 - stalls, 510
 - state machine, 509f
- instruction fetch unit (IFU)
 - branch predictor. *See* Branch predictor
 - instruction cache, 560
 - ITLB misses, 574
 - PC logic, 560
 - reset, stalls, flushes, and traps, 574–576
- integer execution unit changes, 666
- load/store unit changes, 669
- MMU faults, 511
- move, 662
- multiplication, 607–610
- multiply-add, 660
- multiply and divide unit, 607, 613
- platform-level interrupt controller, 784.e1
- PMA checker, 498–500
- PMP checker, 501–502
- postprocessing, 666
- privileged unit changes, 666
- serial peripheral interface, 784.e23
- sign injection, 662
- TLB, 502–507
 - svnapot extension, 507
 - tlbcam, 502–503
 - tlbcontrol, 505–506
 - tlblru, 505
 - tlbmixer, 504
 - tlbram, 504
- top-level block diagram, 495–496, 496f
- universal asynchronous receiver/transmitter, 784.e1
- unpacking unit, 659
- Zfa additional floating-point instructions, 649–651
- fround, 649–652
- Wally LSU
 - application processor configuration, 517
 - high-level block diagram, 516f
 - LSU block diagram, 517f
- Wally memory map, 784.e1
- Wally pipelined processor, 318f
- Wally privileged unit, 364–370
- Wally-riscv-arch-test, 138–140, 139f
- Wally's branch predictor
 - branch direction predictor, 784.e40
 - branch target buffer, 784.e40
 - instruction class prediction, 784.e40
 - return address stack, 784.e40
- Wally's divide and square root unit, 661
- Wally's floating-point convert unit, 661
- Wally style guidelines
 - capitalization, 182, 182t
 - declaring signals, 183–184
 - lowRISC and Freescale, 181
 - port connections, 184–185
 - signal naming conventions, 181–185
 - spacing and indentation, 185–187
 - suffixes and prefixes, 183
- Wally system-on-chip, 70–79, 71t–72t
- Wally verification
 - assertions, 241
 - code coverage, 241–245
 - debugging hints, 235–239
 - design verification plan, 223t, 227–228
 - FPU, 239–240
 - functional coverage, 245–246
 - lint, 230
 - lock-step-compare, 240–241
 - reference model, 229
 - regression, 233–235
 - simulation, 231–233
 - top-level testbench and testvectors, 229–230
- Wally zero stage bootloader, 784.e99
- Waterfall project management, 143–144
- Way invalidation, 453
- Weakly Not Taken (WNT), 158

Weakly Taken (WT), 158
Windows, 784
Worst negative slack (WNS), 254
Write buffers, 677
Write port, 161
Write-through and writeback, 436
W-type instructions, 308

X

Xilinx, 784.e79

Z

Za64rs, 706
Za128rs, 706
Zca, 581
Zcb, 581
Zcd, 581
Zce shorthand, 582
Zcf, 581
Zc* family, 581
Zcmp, 581
Zcmt, 581
Zero stage bootloader (ZSBL), 784
Zfa additional floating-point
instructions, 649–651
Ziccamoa, 705

Ziccif, 705
ZICCLSM
misaligned loads and stores,
775–777
test plan, 776
Wally implementation, 776–777
Ziccrse, 705
Ziccond conditional instructions
alu with Ziccond support, 782f
operation sequences, 780t
riscv-arch-test, 781t
test plan, 781
Wally implementation, 782
Zicsr extension, 53

RISC-V Instruction Set Summary

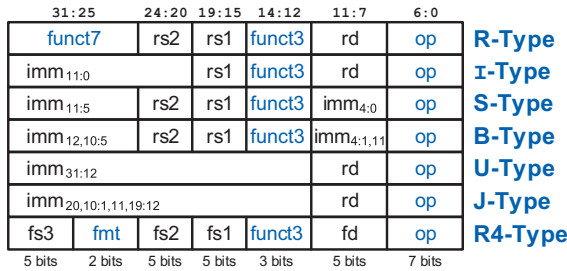


Fig. I.1 RISC-V 32-bit instruction formats

- XLEN: 32 for RV32 / 64 for RV64
- imm: signed immediate in imm_{11:0}
- uimm: 5/6-bit unsigned immediate in imm_{5/4:0}
- upimm: 20 upper bits of a 32-bit immediate, in imm_{31:12}
- Address: memory address: rs1 + SignExt(imm_{11:0})
- [Address]: data at memory location Address
- BTA: branch target address: PC + SignExt((imm_{12:1}, 1'b0))
- JTA: jump target address: PC + SignExt((imm_{20:1}, 1'b0))
- label: text indicating instruction address
- SignExt: value sign-extended to XLEN
- ZeroExt: value zero-extended to XLEN
- csr: control and status register

Table I.1 RV32/64I: RISC-V integer instructions

op	funct3/ imm _{14:12}	funct7/ instr _{31:25}	Type	Instruction	Description	Operation
0000011	000	imm _{11:5}	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address] _{7:0})
0000011	001	imm _{11:5}	I	lh rd, imm(rs1)	load half	rd = SignExt([Address] _{15:0})
0000011	010	imm _{11:5}	I	lw rd, imm(rs1)	load word	rd = SignExt([Address] _{31:0})
0000011	100	imm _{11:5}	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address] _{7:0})
0000011	101	imm _{11:5}	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address] _{15:0})
0010011	000	imm _{11:5}	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011	001	000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011	010	imm _{11:5}	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011	011	imm _{11:5}	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011	100	imm _{11:5}	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011	101	000000*	I	srlr rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011	101	010000*	I	srair rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011	110	imm _{11:5}	I	ori rd, rs1, imm	or immediate	rd = rs1 SignExt(imm)
0010011	111	imm _{11:5}	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111	imm _{14:12}	imm _{31:25}	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011	000	imm _{11:5}	S	sb rs2, imm(rs1)	store byte	[Address] _{7:0} = rs2 _{7:0}
0100011	001	imm _{11:5}	S	sh rs2, imm(rs1)	store half	[Address] _{15:0} = rs2 _{15:0}
0100011	010	imm _{11:5}	S	sw rs2, imm(rs1)	store word	[Address] _{31:0} = rs2 _{31:0}
0110011	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 - rs2
0110011	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2 _{5/4:0} (RV64/32)
0110011	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2 _{5/4:0} (RV64/32)
0110011	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2 _{5/4:0} (RV64/32)
0110011	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 rs2
0110011	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111	imm _{14:12}	imm _{31:25}	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011	000	imm _{12,10:5}	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011	001	imm _{12,10:5}	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011	100	imm _{12,10:5}	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011	101	imm _{12,10:5}	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011	110	imm _{12,10:5}	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011	111	imm _{12,10:5}	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111	000	imm _{11:5}	I	jalr rd, imm(rs1)	jump and link register	PC = [Address], rd = PC + 4
1101111	imm _{14:12}	imm _{20,10:5}	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

* = 0 for RV32, uimm₅ for RV64.

Table I.2 RV64I: Extra integer instructions

op	funct3	funct7/ imm _{11:5}	Type	Instruction	Description	Operation
0000011	011	imm _{11:5}	I	ld rd, imm(rs1)	load double word	rd = [Address] _{63:0}
0000011	110	imm _{11:5}	I	lwu rd, imm(rs1)	load word unsigned	rd = ZeroExt([Address] _{31:0})
0011011	000	imm _{11:5}	I	addiw rd, rs1, imm	add immediate word	rd = SignExt((rs1 + SignExt(imm)) _{31:0})
0011011	001	0000000	I	slliw rd, rs1, uimm	shift left logical immediate word	rd = SignExt((rs1 _{31:0} << uimm) _{31:0})
0011011	101	0000000	I	srliw rd, rs1, uimm	shift right logical immediate word	rd = SignExt((rs1 _{31:0} >> uimm) _{31:0})
0011011	101	0100000	I	sraiw rd, rs1, uimm	shift right arith. immediate word	rd = SignExt((rs1 _{31:0} >>> uimm) _{31:0})
0100011	011	imm _{11:5}	S	sd rs2, imm(rs1)	store double word	[Address] _{63:0} = rs2
0111011	000	0000000	R	addw rd, rs1, rs2	add word	rd = SignExt((rs1 + rs2) _{31:0})
0111011	000	0100000	R	subw rd, rs1, rs2	subtract word	rd = SignExt((rs1 - rs2) _{31:0})
0111011	001	0000000	R	sllw rd, rs1, rs2	shift left logical word	rd = SignExt((rs1 _{31:0} << rs2 _{4:0}) _{31:0})
0111011	101	0000000	R	srlw rd, rs1, rs2	shift right logical word	rd = SignExt((rs1 _{31:0} >> rs2 _{4:0}) _{31:0})
0111011	101	0100000	R	sraw rd, rs1, rs2	shift right arithmetic word	rd = SignExt((rs1 _{31:0} >>> rs2 _{4:0}) _{31:0})

In RV64I, registers are 64 bits, but instructions are still 32 bits. The term “word” generally refers to a 32-bit value. In RV64I, immediate shift instructions use 6-bit immediates: uimm_{5:0}, but for word shifts, the most significant bit of the shift amount (uimm₅) must be 0. Instructions ending in “w” (for “word”) operate on the lower half of the 64-bit registers. Sign- or zero-extension produces a 64-bit result.

Table I.3 RVF/D/Q/Zfh: RISC-V single-, double-, quad-, and half-precision floating-point instructions

op	funct3	funct7/ instr _{31:25}	rs2/ imm _{4:0}	Type	Flags NV DZ OF UF NX	Instruction	Description	Operation
1000011	rm	fs3, ffmt	fs2	R4	x - x x x x	fmadd.* fd, fs1, fs2, fs3	multiply-add	fd = fs1 * fs2 + fs3
1000111	rm	fs3, ffmt	fs2	R4	x - x x x x	fmsub.* fd, fs1, fs2, fs3	multiply-subtract	fd = fs1 * fs2 - fs3
1001011	rm	fs3, ffmt	fs2	R4	x - x x x x	fnmsub.* fd, fs1, fs2, fs3	negate multiply-sub	fd = -(fs1 * fs2 - fs3)
1001111	rm	fs3, ffmt	fs2	R4	x - x x x x	fnmadd.* fd, fs1, fs2, fs3	negate multiply-add	fd = -(fs1 * fs2 + fs3)
1010011	rm	00000, ffmt	fs2	R	x - x - x x	fadd.* fd, fs1, fs2	add	fd = fs1 + fs2
1010011	rm	00001, ffmt	fs2	R	x - x - x x	fsub.* fd, fs1, fs2	subtract	fd = fs1 - fs2
1010011	rm	00010, ffmt	fs2	R	x - x x x x	fmul.* fd, fs1, fs2	multiply	fd = fs1 * fs2
1010011	rm	00011, ffmt	fs2	R	x x x x x x	fdiv.* fd, fs1, fs2	divide	fd = fs1 / fs2
1010011	rm	01011, ffmt	00000	I	x - - - - x	fsqrt.* fd, fs1	square root	fd = sqrt(fs1)
1010011	000	00100, ffmt	fs2	R	- - - - -	fsgnj.* fd, fs1, fs2	sign injection	fd = fs1, sign = sign(fs2)
1010011	001	00100, ffmt	fs2	R	- - - - -	fsgnjn.* fd, fs1, fs2	negate sign injection	fd = fs1, sign = -sign(fs2)
1010011	010	00100, ffmt	fs2	R	- - - - -	fsgnjx.* fd, fs1, fs2	xor sign injection	fd = fs1, sign = sign(fs2) ^ sign(fs1)
1010011	000	00101, ffmt	fs2	R	x - - - - -	fmin.* fd, fs1, fs2	min	fd = min(fs1, fs2)
1010011	001	00101, ffmt	fs2	R	x - - - - -	fmax.* fd, fs1, fs2	max	fd = max(fs1, fs2)
1010011	010	10100, ffmt	fs2	R	x - - - - -	feq.* rd, fs1, fs2	compare =	rd = (fs1 == fs2)
1010011	001	10100, ffmt	fs2	R	x - - - - -	flt.* rd, fs1, fs2	compare <	rd = (fs1 < fs2)
1010011	000	10100, ffmt	fs2	R	x - - - - -	fle.* rd, fs1, fs2	compare ≤	rd = (fs1 ≤ fs2)
1010011	001	11100, ffmt	00000	I	- - - - -	fclass.* rd, fs1	classify	rd = classification of fs1
0000111	010	imm _{11:5}	imm _{4:0}	I	- - - - -	flw fd, imm(rs1)	load float	fd = [Address] _{31:0}
0100111	010	imm _{11:5}	fs2	S	- - - - -	fsw fs2, imm(rs1)	store float	[Address] _{31:0} = fs2
1010011	rm	11000, ffmt	000, lu	I	x - x - x x	fcvt.dst.src rd, fs1	convert fp to int	rd = dst(fs1)
1010011	rm	11010, ffmt	000, lu	I	- - x - x x	fcvt.dst.src fd, rs1	convert int to fp	fd = dst(rs1)
1010011	rm	10000, dfmt	000, sfmt	I	x - x x x x	fcvt.dst.src fd, fs1	convert fp to fp	fd = dst(fs1)
1010011	000	11100, ffmt	00000	I	- - - - -	fmv.x.{h/w/d} rd, fs1	move fp to int	rd = fs1
1010011	000	11110, ffmt	00000	I	- - - - -	fmv.{h/w/d}.x fd, rs1	move int to fp	fd = rs1

RVD only

0000111	011	imm _{11:5}	imm _{4:0}	I	- - - - -	fld fd, imm(rs1)	load double	fd = [Address] _{63:0}
0100111	011	imm _{11:5}	fs2	S	- - - - -	fsd fs2, imm(rs1)	store double	[Address] _{63:0} = fs2

RVQ only

0000111	100	imm _{11:5}	imm _{4:0}	I	- - - - -	flq fd, imm(rs1)	load quad	fd = [Address] _{127:0}
0100111	100	imm _{11:5}	fs2	S	- - - - -	fsq fs2, imm(rs1)	store quad	[Address] _{127:0} = fs2

RVH only

0000111	001	imm _{11:5}	imm _{4:0}	I	- - - - -	flh fd, imm(rs1)	load half	fd = [Address] _{15:0}
0100111	001	imm _{11:5}	fs2	S	- - - - -	fsh fs2, imm(rs1)	store half	[Address] _{15:0} = fs2

* = {h/s/d/q}. fs1, fs2, fs3, fd: floating-point registers. fs1, fs2, and fd are encoded in fields rs1, rs2, and rd; only R4-type also encodes fs3. ffmt: precision of computational instruction (half=10₂, single=00₂, double=01₂, quad=11₂). sfmt: source format. rm: rounding mode (0=toward zero, 1=toward zero, 2=down, 3=up, 4=to nearest (max magnitude), 7=dynamic). sign(fs1): the sign of fs1. dst or src = {h/s/d/q} for fp, {w/wu/lu} for 32/64-bit signed/unsigned ints. {w/wu} = 32-bit signed/unsigned int. {l/lu} = 64-bit (long) signed/unsigned int. For fp move instructions (fmv), {h/w/d} denotes a half-/single-/double-precision fp number held in a fp register, and x is a fp number (held in an integer register), with the same-precision as the fp source/destination. Floating-point exception flags are: NV: invalid operation, DZ: divide by zero, OF: overflow, UF: underflow, and NX: inexact.

Table I.4 Register names and numbers

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3–6	x28–31	Temporary registers

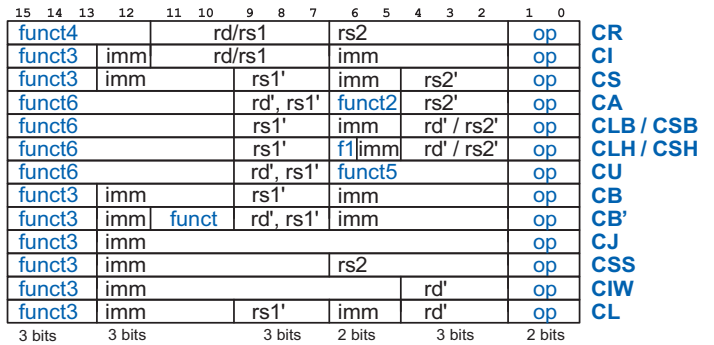


Fig. I.2. RISC-V compressed (16-bit) instruction formats

Table I.5 RVM: RISC-V multiply and divide instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0110011	000	0000001	R	mul rd, rs1, rs2	multiply	$rd = (rs1 * rs2)_{XLEN-1:0}$
0110011	001	0000001	R	mulh rd, rs1, rs2	multiply high signed signed	$rd = (rs1 * rs2)_{2*XLEN-1:XLEN}$
0110011	010	0000001	R	mulhsu rd, rs1, rs2	multiply high signed unsigned	$rd = (rs1 * rs2)_{2*XLEN-1:XLEN}$
0110011	011	0000001	R	mulhu rd, rs1, rs2	multiply high unsigned unsigned	$rd = (rs1 * rs2)_{2*XLEN-1:XLEN}$
0110011	100	0000001	R	div rd, rs1, rs2	divide (signed)	$rd = rs1 / rs2$
0110011	101	0000001	R	divu rd, rs1, rs2	divide unsigned	$rd = rs1 / rs2$
0110011	110	0000001	R	rem rd, rs1, rs2	remainder (signed)	$rd = rs1 \% rs2$
0110011	111	0000001	R	remu rd, rs1, rs2	remainder unsigned	$rd = rs1 \% rs2$

RV64M adds mulw, divw, remw, divuw, remuw with op = 59. These operate on only the lower 32 bits of a register.

Table I.6 RVC/Zca: RISC-V compressed (16-bit) integer instructions

op	instr _{15:10}	funct2	Registers	Type	Compressed Instruction	32-Bit Equivalent
C/Zca: Compressed instructions excluding floating-point loads and stores						
00	000---	-	imm ≠ 0	CIW	c.addi4spn rd', sp, imm	addi rd', sp, ZeroExt(imm)*4
00	010---	-		CL	c.lw rd', imm(rs1')	lw rd', (ZeroExt(imm)*4)(rs1')
00	110---	-		CS	c.sw rs2', imm(rs1')	sw rs2', (ZeroExt(imm)*4)(rs1')
01	000000	-	rs1 = 0, imm = 0	CI	c.nop	addi x0, x0, 0
01	000---	-	rd ≠ 0, imm ≠ 0	CI	c.addi rd, imm	addi rd, rd, SignExt(imm)
01	010---	-	rd ≠ 0	CI	c.li rd, imm	addi rd, x0, SignExt(imm)
01	011---	-	rd ≠ {0,2}, imm ≠ 0	CI	c.lui rd, imm	lui rd, {14(imm ₅), imm}
01	011---	-	rd = 2, imm ≠ 0	CI	c.addi16sp sp, imm	addi sp, sp, SignExt(imm)*16
01	100-00	-	imm ≠ 0	CB'	c.srli rd', imm	srli rd', rd', imm
01	100-01	-	imm ≠ 0	CB'	c.srai rd', imm	srai rd', rd', imm
01	100-10	-		CB'	c.andi rd', imm	andi rd', rd', SignExt(imm)
01	100011	00		CA	c.sub rd', rs2'	sub rd', rd', rs2'
01	100011	01		CA	c.xor rd', rs2'	xor rd', rd', rs2'
01	100011	10		CA	c.or rd', rs2'	or rd', rd', rs2'
01	100011	11		CA	c.and rd', rs2'	and rd', rd', rs2'
01	101---	-		CJ	c.j label	jal x0, label
01	110---	-		CB	c.beqz rs1', label	beq rs1', x0, label
01	111---	-		CB	c.bnez rs1', label	bne rs1', x0, label
10	000---	-	rd ≠ 0, imm ≠ 0	CI	c.slli rd, imm	slli rd, rd, imm
10	010---	-	rd ≠ 0	CI	c.lwsp rd, imm	lw rd, (ZeroExt(imm)*4)(sp)
10	1000--	-	rs1 ≠ 0, rs2 = 0	CR	c.jr rs1	jalr x0, 0(rs1)
10	1000--	-	rd ≠ 0, rs2 ≠ 0	CR	c.mv rd, rs2	add rd, x0, rs2
10	1001--	-	rs1 = 0, rs2 = 0	CR	c.ebreak	ebreak
10	1001--	-	rs1 ≠ 0, rs2 = 0	CR	c.jalr rs1	jalr ra, 0(rs1)
10	1001--	-	rd ≠ 0, rs2 ≠ 0	CR	c.add rd, rs2	add rd, rd, rs2
10	110---	-		CSS	c.swsp rs2, imm	sw rs2, (ZeroExt(imm)*4)(sp)
C/Zca (RV32 only)						
01	001---	-		CJ	c.jal label	jal ra, label
C/Zca (RV64 only, reuses encodings for c.jal above and c.flw, c.fsw, c.flwsp, and c.fswsp from Table I.22)						
00	011---	-		CL	c.ld rd', imm(rs1') (replaces c.flw)	ld rd', (ZeroExt(imm)*8)(rs1')
00	111---	-		CS	c.sd rs2', imm(rs1') (replaces c.fsw)	sd rs2, (ZeroExt(imm)*8)(rs1')
01	001---	-	rd ≠ 0	CI	c.addiw rd, imm (replaces c.jal)	addiw rd, rd, (SignExt(imm))
01	100111	00		CA	c.subw rd', rs2'	subw rd', rd', rs2'
01	100111	01		CA	c.addw rd', rs2'	addw rd', rd', rs2'
10	011---	-	rd ≠ 0	CI	c.ldsp rd, imm (replaces c.flwsp)	ld rd, (ZeroExt(imm)*8)(sp)
10	111---	-		CSS	c.sdsp rs2, imm (replaces c.fswsp)	sd rs2, (ZeroExt(imm)*8)(sp)

rs1', rs2', rd': 3-bit register designator for registers 8–15: 000₂ = x8 or f8, ..., 111₂ = x15 or f15. Table I.9 lists additional RISC-V compressed instructions.

Table I.7 Common RISC-V pseudoinstructions

Pseudoinstruction	RISC-V Instructions	Description	Operation
nop	addi x0, x0, 0	no operation	
li rd, imm _{11:0}	addi rd, x0, imm _{11:0}	load 12-bit immediate	rd = SignExtend(imm _{11:0})
li rd, imm _{31:0}	lui rd, imm _{31:12} [*] addi rd, rd, imm _{11:0}	load 32-bit immediate	rd = imm _{31:0}
mv rd, rs1	addi rd, rs1, 0	move (also called “register copy”)	rd = rs1
not rd, rs1	xori rd, rs1, -1	one’s complement	rd = ~rs1
neg rd, rs2	sub rd, x0, rs2	two’s complement	rd = -rs1
seqz rd, rs1	sltiu rd, rs1, 1	set if = 0	rd = (rs1 == 0)
snez rd, rs2	sltu rd, x0, rs2	set if ≠ 0	rd = (rs1 ≠ 0)
sltz rd, rs1	slt rd, rs1, x0	set if < 0	rd = (rs1 < 0)
sgtz rd, rs2	slt rd, x0, rs2	set if > 0	rd = (rs1 > 0)
beqz rs1, label	beq rs1, x0, label	branch if = 0	if (rs1 == 0) PC = label
bnez rs1, label	bne rs1, x0, label	branch if ≠ 0	if (rs1 ≠ 0) PC = label
blez rs2, label	bge x0, rs2, label	branch if ≤ 0	if (rs1 ≤ 0) PC = label
bgez rs1, label	bge rs1, x0, label	branch if ≥ 0	if (rs1 ≥ 0) PC = label
bltz rs1, label	blt rs1, x0, label	branch if < 0	if (rs1 < 0) PC = label
bgtz rs2, label	blt x0, rs2, label	branch if > 0	if (rs1 > 0) PC = label
ble rs1, rs2, label	bge rs2, rs1, label	branch if ≤	if (rs1 ≤ rs2) PC = label
bgt rs1, rs2, label	blt rs2, rs1, label	branch if >	if (rs1 > rs2) PC = label
bleu rs1, rs2, label	bgeu rs2, rs1, label	branch if ≤ (unsigned)	if (rs1 ≤ rs2) PC = label
bgtu rs1, rs2, label	bltu rs2, rs1, offset	branch if > (unsigned)	if (rs1 > rs2) PC = label
j label	jal x0, label	jump	PC = label
jal label	jal ra, label	jump and link	PC = label, ra = PC + 4
jr rs1	jalr x0, 0(rs1)	jump register	PC = rs1
jalr rs1	jalr ra, 0(rs1)	jump and link register	PC = rs1, ra = PC + 4
ret	jalr x0, 0(ra)	return from function	PC = ra
call label	jal ra, label	call nearby function	PC = label, ra = PC + 4
call label	auipc ra, offset _{31:12} [*] jalr ra, offset _{11:0} (ra)	call far away function	PC = PC + offset, ra = PC + 4
la rd, symbol	auipc rd, symbol _{31:12} [*] addi rd, rd, symbol _{11:0}	load address of global variable	rd = PC + symbol
l{b h w} rd, symbol	auipc rd, symbol _{31:12} [*] l{b h w} rd, symbol _{11:0} (rd)	load global variable	rd = [PC + symbol]
s{b h w} rs2, symbol, rs1	auipc rs1, symbol _{31:12} [*] s{b h w} rs2, symbol _{11:0} (rs1)	store global variable	[PC + symbol] = rs2
csrr rd, csr	csrrs rd, csr, x0	read CSR	rd = csr
csrw csr, rs1	csrrw x0, csr, rs1	write CSR	csr = rs1
csrs/csrrc csr, rs1	csrrs/csrrc x0, csr, rs1	set/clear bits in CSR	csr = csr rs1 / csr = csr & ~rs1

^{*}If bit 11 of the immediate/offset/symbol is 1, the upper immediate is incremented by 1. offset/symbol are the 32-bit PC-relative addresses of a label/global variable.

Table I.8 Privileged / CSR / fence instructions

op	funct3	imm _{11:0} /funct7	Registers	Type	Instruction	Description	Operation
1110011	000	000000000000	rs1,rd=0	I	ecall	transfer control to OS	
1110011	000	000000000001	rs1,rd=0	I	ebreak	transfer control to debugger	
1110011	000	000100000010	rs1,rd=0	I	sret	return from supervisor exception	PC = sepc
1110011	000	001100000010	rs1,rd=0	I	mret	return from machine exception	PC = mepc
1110011	000	000100000101	rs1,rd=0	I	wfi	wait for interrupt	
0001111	000	000000110011	rs1,rd=0	I	fence	synchronize loads and stores	
0001111	001	000000000000	rs1,rd=0	I	fence.i	synchronize instruction memory	
1110011	000	0001001	rd=0	R	sfence.vma rs1,rs2	synchronize page table	Note: rs1 = vaddr, rs2 = asid
1110011	001	csr		I	csrrw rd,csr,rs1	CSR read/write	rd=csr,csr = rs1
1110011	010	csr		I	csrrs rd,csr,rs1	CSR read/set	rd=csr,csr = rs1
1110011	011	csr		I	csrrc rd,csr,rs1	CSR read/clear	rd=csr,csr &= ~rs1
1110011	101	csr	rs1=uimm	I	csrrwi rd,csr,uimm	CSR read/write immediate	rd=csr,csr = ZeroExt(uimm)
1110011	110	csr	rs1=uimm	I	csrrsi rd,csr,uimm	CSR read/set immediate	rd=csr,csr = ZeroExt(uimm)
1110011	111	csr	rs1=uimm	I	csrrci rd,csr,uimm	CSR read/clear immediate	rd=csr,csr &= ~ZeroExt(uimm)

RISC-V Privileged Architecture (Version 1.12) Summary

(Excluding Hypervisor and Debug Features)

Table I.9 Control and Status Registers (CSRs)

Name	Address	h Address (RV32)	Size	Type	Description
Traps					
mstatus/sstatus	300/100	310	64/XLEN	MRW/SRW	Status
medeleg	302	312	XLEN	MRW	Exception delegation
mideleg	303		XLEN (12)	MRW	Interrupt delegation
mie/sie	304/104		XLEN (12)	MRW/SRW	Interrupt enable
mtvec/stvec	305/105		XLEN	MRW/SRW	Trap vector
mscratch/sscratch	340/140		XLEN	MRW/SRW	Scratch (for trap stack pointer)
mepc/sepc	341/141		XLEN	MRW/SRW	Exception program counter
mcause/scause	342/142		XLEN	MRW/SRW	Trap cause
mtval/stval	343/143		XLEN	MRW/SRW	Bad address or instruction
mip/sip	344/144		XLEN (12)	MRW/SRW	Interrupt pending
Virtual Memory					
satp	180		XLEN	SRW	Address translation & protection
Physical Memory Protection					
pmpcfg0/2/.../14	3A0/3A2/.../3AE	3A1/3A3/.../3AF	64 (8 x 8)	MRW	PMP configuration
pmpaddr0/1/.../63	3B0/3B1/.../3EF		XLEN	MRW	PMP address
Floating-Point					
fcsr/fflags/frm	003/001/002			URW	Floating-point CSR/flags/rounding
Counters / Performance Monitoring					
mcounteren/scounteren	306/106		32	MRW/SRW	Counter enable
mcountinhibit	320		32	MRW	Counter inhibit
mhpmevent3/4/.../31	323/324/.../33F	723/724/.../73F	XLEN	MRW	Counter event selectors
mcycle/cycle	B00/C00	B80/C80	64	MRW/URO	Cycle counter
time	C01	C81	64	URO	Time (from CLINT)
minstret/instret	B02/C02	B82/C82	64	MRW/URO	Instructions retired
mhpmcounter3/4/.../31	B03/B04/.../B1F	B83/B84/...	64	MRW/	Performance-monitoring counters
hpmcounter3/4/.../31	C03/C04/.../C1F	C83/C84/...		URO	
Machine Information Registers & Configuration					
misa	301		XLEN	MRW	Instruction set
menvcfg/senvcfg	30A/10A	31A/-	64/XLEN	MRW/SRW	Environment configuration
mseccfg	747	757	64	MRW	Security configuration
mvendorid	F11		32	MRO	JEDEC vendor ID
marchid	F12		XLEN	MRO	Vendor-specific architecture ID
mimpid	F13		XLEN	MRO	Vendor-specific implementation ID
mhartid	F14		XLEN	MRO	Hart ID
mconfigptr	F15		XLEN	MRO	Configuration pointer
Sstc Extension					
stimecmp	14D	15D	64	SRW	Supervisor timer compare
Zkr Extension					
seed	015		32	MRW	Seed to provide entropy for random number generator

{M/S/U}{RW/RO} = {Machine/Supervisor/User}{Read & Write/Read Only}. Gray registers are read only. For 64-bit registers in RV32, the bottom 32 are in the base register and the upper 32 are in a corresponding h register (e.g., mcycleh).

Table I.10 CSR bitfields

Register	XLEN-1	XLEN-2	XLEN-3	...	37	36	35	34	33	32	31-23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
mstatus	SD				MBE	SBE	SXL	UXL			TSR	TW	TVM	MXR	SUM	MPRV	XS	FS			MPP		VS	SPP	MPIE	UBE	SPIE			MIE		SIE						
sstatus	SD							UXL							MXR	SUM		XS	FS					VS	SPP		UBE	SPIE				SIE						
misa		MXL										V	U		S		Q				M				I	H			F	E	D	C		A				
menvcfg	STCE	PBMTE	ADUE																							CBZE	CBCFE	CBIE					FIOM					
senvcfg																										CBZE	CBCFE	CBIE					FIOM					
fcsr																												frm		NV	DZ	OF	UF	NX				
{m/s}cause	INT																																	cause				
medeleg																																						
mideleg																																						
mie																												MEI		SEI		MTI		STI		MSI		SSI
mip																												MEIE		SEIE		MTIE		STIE		MSIE		SSIE
sip																												MEIP		SEIP		MTIP		STIP		MSIP		SSIP
sie																												SEIE			STIE				SSIE			
sip																												SEIP			STIP				SSIP			

Gray register bits are read only. For 64-bit registers in RV32, the bottom 32 are in the base register and the upper 32 are in a corresponding h register (e.g., mcycleh). RV32 mstatus and sstatus place SD in bit 31. The bolded fields are in RV64 only: mstatus drops the SXL and UXL fields, leaving only MBE and SBE as bits 5:4; sstatus has no UXL field, so no sstatus is needed.

Table I.11 CSR bitfield legend

Bit Field	Meaning	Description
SD	State Dirty	1 = any of XS, FS, VS dirty
{M/S/U}BE	{Machine/Supervisor/User} Big Endian	1 = Big-endian data
{M/S/U}XL	{Machine/Supervisor/User} Base ISA XLEN	01 = 32; 10 = 64; 11 = 128
TSR	Trap SRET	For virtualization
TW	Timeout Wait	1 = Trap on WFI after timeout
TVM	Trap Virtual Memory	For virtualization
MXR	Make eXecutable Readable	1 = Allow loads from executable pages
SUM	Supervisor User Memory	1 = Supervisor can access user memory
MPRV	Modify PRiVilege	1 = Apply memory privileges from MPP
{X/F/V}S	{Custom(X)/Floating-point/Vector} State dirty	00 = off; 01 = initial; 10 = clean; 11 = dirty
{M/S}PP	{Machine/Supervisor} Previous Privilege	00 = User; 01 = Supervisor; 11 = Machine
{M/S}PIE	{Machine/Supervisor} Previous Interrupt Enable	Restore IE = PIE after MRET/SRET
{M/S}IE	{Machine/Supervisor} global Interrupt Enable	0 = disable all interrupts
STCE	Supervisor Timer Compare Enable	1 = enable Stsc extension (see Table I.9)
PBMTE	Page-Based Memory Types Enable	1 = enable Svpbmt extension (see Section 11.2.2.1.XREF)
ADUE	Access/Dirty Update Enable	1 = enable Svadu extension (see Section 8.2.3.5XREF)
CB{Z/CF}E	Cache Block {Zero/Clean&Flush} Enable	1 = enable CBO instructions (see Table I.21)
CBIE	Cache Block Invalidate Enable	00 = disable; 01 = flush; 11 = invalidate
FIOM	Fence of I/O implies Memory	For virtualization
frm	Floating-point Rounding Mode	0 = RNE; 1 = RTZ; 2 = RDN; 3 = RUP; 4 = RMM; 7 = DYN
flags {NV, DZ, OF, UF, NX}	{iNValid, Divide by Zero, OverFlow, UnderFlow, iNeXact}	Floating-point flags
INT	Trap is an INTerrupt	1 = trap is an interrupt
cause	Cause of trap	See Table I.13
{M/S}{E/T/S}{E/P}	{Machine/Supervisor}{External/Timer/Software} Interrupt{Enable/Pending}	MEIP set by PLIC MTIP set by CLINT when mtime > mtimecmp MSIP set by CLINT SEIP set by PLIC or mip.SEIP STIP set by mip.STIP or when mtime > stimecmp SSIP set by mip.SSIP

Table I.12 misa bitfields

Bit	Character	Description
0	A	Atomic
1	B	Bit manipulation (Zba+Zbb+Zbs)
2	C	Compressed
3	D	Double-precision floating-point
4	E	RV32E Embedded ISA
5	F	Single-precision floating-point
6	G	At least IMAFD supported
7	H	Hypervisor
8	I	RV32/64/128I integer ISA
12	M	Integer multiply/divide
16	Q	Quad-precision floating-point
18	S	Supervisor mode
20	U	User mode
21	V	Vector
XLEN-1:XLEN-2	MXL	XLEN: 01 = 32, 10 = 64, 11 = 128

Table I.13 cause interrupt and exception codes

Cause	Interrupt	Exception
0		Instruction address misaligned
1	Supervisor Software	Instruction access fault
2		Illegal instruction
3	Machine Software	Breakpoint
4		Load address misaligned
5	Supervisor Timer	Load access fault
6		Store/AMO address misaligned
7	Machine Timer	Store/AMO access fault
8		Environment call from U-mode
9	Supervisor External	Environment call from S-mode
10		
11	Machine External	Environment call from M-mode
12		Instruction page fault
13		Load page fault
14		
15		Store/AMO page fault

RISC-V Virtual Memory Summary

Table I.14 satp bitfields

Field	Meaning	RV64	RV32
Mode	RV64: 0 = bare; 8 = SV39; 9 = SV48; 10 = SV57 RV32: 0 = bare; 1 = RV32	63:60	31
ASID	Address space ID	59:44	30:22
PPN	Physical page number of root page table	43:0	21:0

Table I.16 XWR encodings

X	W	R	Meaning
0	0	0	Pointer to next level of page table
0	0	1	Read-only page
0	1	0	Reserved
0	1	1	Read-write page
1	0	0	Execute-only page
1	0	1	Read-execute page
1	1	0	Reserved
1	1	1	Read-write-execute page

Table I.15 Virtual address bitfields

Format	63:57	56:48	47:39	38:32	31:30	29:22	21	20:12	11:0
SV32	n/a				VPN[1]		VPN[0]		Page
SV39	extend			VPN[2]		VPN[1]		VPN[0]	Offset
SV48	extend		VPN[3]						
SV57	extend	VPN[4]							

Table I.17 Physical address bitfields

Format	55:48	47:39	38:34	33:30	29:22	21	20:12	11:0
SV32	n/a			PPN[1]		PPN[0]		Page
SV39	PPN[2]					PPN[1]		PPN[0]
SV48	PPN[3]		PPN[2]					
SV57	PPN[4]	PPN[3]						

Virtual addresses are 32 bits for RV32 and 64 bits for RV64.

Physical addresses are 34 bits for RV32 and 56 bits for RV64.

Table I.18 Page table entry bitfields

ISA	63	62:61	60:54	53:32	31:10	9:8	7	6	5	4	3	2	1	0
RV32	n/a					PPN	RSW:		Dirty	Accessed	Global	User	eXecutable	Writable
RV64	NAPOT:		PBMT:		reserved	PPN	Reserved for Software							Valid
	0 = normal 1 = 64 KiB huge page		0 = Use PMA 1 = Uncacheable Memory 2 = Uncacheable I/O											

RISC-V Physical Memory Protection (PMP) Summary

Table I.19 pmpaddr0-63 bitfields

ISA	63:54	53:32	31:0
RV32	n/a	n/a	33:2
RV64	0	55:34	33:2

Table I.20 pmpcfg0-15 bitfields

7	6:5	4:3	2	1	0
Locked	00	Alignment: 0 = off 1 = TOR: Top Of Range 2 = NA4: Naturally Aligned 4-byte 3 = NAPOT: Naturally Aligned Power of Two	eXecutable	Writable	Readable

Each pmpcfg register contains 8 (RV64) or 4 (RV32) copies of the 8-bit configuration fields shown in Table I.20, so each pmpcfg register supports up to 8 (RV64) or 4 (RV32) pmpaddr registers. To support all 64 pmpaddr registers, RV64 uses 64-bit pmpcfg0, 2, 8,... 14 and RV32 uses 32-bit pmpcfg0, 1, 2,...15.

RISC-V Cache Management Operation (CMO) Instructions

Table I.21 Zicbo*: CMO instructions

op	funct3	imm _{11:0}	Registers	Type	Instruction	Description	Operation
Zicbom: Cache Block Management							
0001111	010	000000000000	rd = 0	I	cbo.inval (rs1)	invalidate block	invalidate block holding [rs1]
0001111	010	000000000001	rd = 0	I	cbo.clean (rs1)	clean (write back) block	write back block holding [rs1]
0001111	010	000000000010	rd = 0	I	cbo.flush (rs1)	flush block	flush block holding [rs1]
Zicboz: Cache Block Zero							
0001111	010	000000000100	rd = 0	I	cbo.zero (rs1)	zero block	zero block holding [rs1]
Zicbop: Cache Block Prefetch							
0010011	110	imm[11:5], 00000	rd = 0	I	prefetch.i imm(rs1)	instruction prefetch	I\$ Prefetch [Address]
0010011	110	imm[11:5], 00001	rd = 0	I	prefetch.r imm(rs1)	data read prefetch	D\$ Read Prefetch [Address]
0010011	110	imm[11:5], 00011	rd = 0	I	prefetch.w imm(rs1)	data write prefetch	D\$ Write Prefetch [Address]

RISC-V Additional Compressed Instructions

Table I.22 RVC/Zcb/Zcf/Zcd: Additional compressed (16-bit) instructions (continued from Table I.6)

op	instr _{15:10}	funct	Type	Compressed Instruction	32-Bit Equivalent
Zcb: Simple code-saving compressed instructions (some require Zba, Zbb, M, or RV64)					
00	100000	–	CLB	c.lbu rd', imm(rs1')	lbw rd', (ZeroExt(imm))(rs1')
00	100001	1	CLH	c.lh rd', imm(rs1')	lh rd', (ZeroExt(imm)*2)(rs1')
00	100001	0	CLH	c.lhu rd', imm(rs1')	lhu rd', (ZeroExt(imm)*2)(rs1')
00	100010	–	CSB	c.sb rs2', imm(rs1')	sb rs2', (ZeroExt(imm))(rs1')
00	100011	0	CSH	c.sh rs2', imm(rs1')	sh rs2', (ZeroExt(imm)*2)(rs1')
01	100111	11000	CU	c.zext.b rd'	andi rd', rd', 0xFF
01	100111	11001	CU	c.sext.b rd'	sext.b rd', rd'
01	100111	11010	CU	c.zext.h rd'	zext.h rd', rd'
01	100111	11011	CU	c.sext.h rd'	sext.h rd', rd'
01	100111	11101	CU	c.not rd'	xori rd', rd', -1
01	100111	10	CA	c.mul rd', rs2'	mul rd', rd', rs2'
01	100111	11100	CU	c.zext.w rd'	add.uw rd', rd', zero
{C/Zcf}F (RV32 only): Compressed single-precision floating-point loads and stores					
00	011---	–	CL	c.flw fd', imm(rs1')	flw fd', (ZeroExt(imm)*4)(rs1')
00	111---	–	CS	c.fsw fs2', imm(rs1')	fsw fs2', (ZeroExt(imm)*4)(rs1')
10	011---	–	CI	c.flwsp fd, imm	flw fd, (ZeroExt(imm)*4)(sp)
10	111---	–	CSS	c.fswsp fs2, imm	fsw fs2, (ZeroExt(imm)*4)(sp)
{C/Zcd}D: Compressed double-precision floating-point loads and stores					
00	001---	–	CL	c.fld fd', imm(rs1')	fld fd', (ZeroExt(imm)*8)(rs1')
00	101---	–	CS	c.fsd fs2', imm(rs1')	fsd fs2', (ZeroExt(imm)*8)(rs1')
10	001---	–	CI	c.fldsp fd, imm	fld fd, (ZeroExt(imm)*8)(sp)
10	101---	–	CSS	c.fsdsp fs2, imm	fsd fs2, (ZeroExt(imm)*8)(sp)

RISC-V Bit Manipulation Instructions

Table I.23 RVB/Zb*: Bit manipulation instructions

op	funct3	funct7/ imm _{11:5}	rs2/ imm _{4:0}	Type	Instruction	Description	Operation
Zba: Address Generation							
0110011	010	0010000	rs2	R	shladd rd,rs1,rs2	shift left by 1 and add	rd = rs2 + (rs1 << 1)
0110011	100	0010000	rs2	R	sh2add rd,rs1,rs2	shift left by 2 and add	rd = rs2 + (rs1 << 2)
0110011	110	0010000	rs2	R	sh3add rd,rs1,rs2	shift left by 3 and add	rd = rs2 + (rs1 << 3)
Zba (RV64 only): Shift and Add 32-bit Unsigned Word (U.W.)							
0111011	010	0010000	rs2	R	shladd.uw rd,rs1,rs2	shift u.w. left 1, add	rd = rs2 + ZeroExt(rs1 _{31:0}) << 1
0111011	100	0010000	rs2	R	sh2add.uw rd,rs1,rs2	shift u.w. left 2, add	rd = rs2 + ZeroExt(rs1 _{31:0}) << 2
0111011	110	0010000	rs2	R	sh3add.uw rd,rs1,rs2	shift u.w. left 3, add	rd = rs2 + ZeroExt(rs1 _{31:0}) << 3
0111011	000	0000100	rs2	R	add.uw rd,rs1,rs2	add u.w.	rd = rs2 + ZeroExt(rs1 _{31:0})
0011011	001	000010*	uimm _{4:0}	I	slli.uw rd,rs1,uimm	shift left logical imm. u.w.	rd = ZeroExt(rs1 _{31:0}) << uimm
Zbb: Basic Bit Manipulation							
0110011	100	0000101	rs2	R	min rd,rs1,rs2	minimum	rd = min(rs1, rs2)
0110011	101	0000101	rs2	R	minu rd,rs1,rs2	unsigned minimum	rd = minu(rs1, rs2)
0110011	110	0000101	rs2	R	max rd,rs1,rs2	maximum	rd = max(rs1, rs2)
0110011	111	0000101	rs2	R	maxu rd,rs1,rs2	unsigned maximum	rd = maxu(rs1, rs2)
0110011	100	0100000	rs2	R	orn rd,rs1,rs2	or inverted operand	rd = rs1 ~rs2
0110011	111	0100000	rs2	R	andn rd,rs1,rs2	and inverted operand	rd = rs1 & ~rs2
0110011	110	0100000	rs2	R	xnor rd,rs1,rs2	xor inverted operand	rd = rs1 ^ ~rs2
0110011	001	0110000	rs2	R	rol rd,rs1,rs2	rotate left	rd = rol(rs1, rs2 _{4/5:0})
0110011	101	0110000	rs2	R	ror rd,rs1,rs2	rotate right	rd = ror(rs1, rs2 _{4/5:0})
0010011	101	0010100	00111	I	orc.b rd,rs1	bitwise or, combine on bytes	rd = orc.b(rs1)
011.011	100	0000100	00000	I	zext.h rd,rs1	zero-extend halfword	rd = ZeroExt(rs1 _{15:0})
0010011	001	0110000	00000	I	clz rd,rs1	count leading zeros	rd = clz(rs1)
0010011	001	0110000	00010	I	cpop rd,rs1	population count	rd = popcnt(rs1)
0010011	001	0110000	00001	I	ctz rd,rs1	count trailing zeros	rd = clz(rev(rs1))
0010011	001	0110000	00100	I	sext.b rd,rs1	sign-extend byte	rd = SignExt(rs1 _{7:0})
0010011	001	0110000	00101	I	sext.h rd,rs1	sign-extend halfword	rd = SignExt(rs1 _{15:0})
0010011	101	011010.	11000	I	rev8 rd,rs1	byte-wise reverse	rd = revbyte(rs1)
0010011	101	011000*	uimm _{4:0}	I	rori rd,rs1,uimm	rotate right immediate	rd = ror(rs1, uimm)
Zbb (RV64 only): Basic Bit Manipulation on 32-bit Words							
0111011	001	0110000	rs2	R	rolw rd,rs1,rs2	rotate left word	rd = SignExt(rol32(rs1 _{31:0} , rs2 _{4:0}))
0111011	101	0110000	rs2	R	rorw rd,rs1,rs2	rotate right word	rd = SignExt(ror32(rs1 _{31:0} , rs2 _{4:0}))
0011011	101	0110000	uimm _{4:0}	I	roriw rd,rs1,uimm	rotate right immediate word	rd = SignExt(ror32(rs1 _{31:0} , uimm))
0011011	001	0110000	00000	I	clzw rd,rs1	count leading zeros word	rd = clz(rs1 _{31:0})
0011011	001	0110000	00010	I	cpopw rd,rs1	population count word	rd = popcnt(rs1 _{31:0})
0011011	001	0110000	00001	I	ctzw rd,rs1	count trailing zeros word	rd = clz(rev(rs1 _{31:0}))
Zbc: Carry-Less (CL) Multiplication							
0110011	001	0000101	rs2	R	clmul rd,rs1,rs2	cl multiply (lower half)	rd = (rs1 @ rs2) _{XLEN-1:0}
0110011	011	0000101	rs2	R	clmulh rd,rs1,rs2	cl multiply (upper half)	rd = (rs1 @ rs2) _{2*XLEN-1:XLEN}
0110011	010	0000101	rs2	R	clmulr rd,rs1,rs2	cl multiply reversed	rd = rev((rev(rs1) @ rev(rs2)) _{XLEN-1:0})
Zbs: Single-Bit Operations							
0110011	001	0100100	rs2	R	bclr rd,rs1,rs2	bit clear	rd = rs1 & ~(1 << rs2 _{4/5:0})
0110011	001	0110100	rs2	R	binv rd,rs1,rs2	bit invert	rd = rs1 ^ (1 << rs2 _{4/5:0})
0110011	001	0010100	rs2	R	bset rd,rs1,rs2	bit set	rd = rs1 (1 << rs2 _{4/5:0})
0110011	101	0100100	rs2	R	bext rd,rs1,rs2	bit extract	rd = (rs1 & (1 << rs2 _{4/5:0}))
0010011	001	010010*	uimm _{4:0}	I	bclri rd,rs1,uimm	bit clear immediate	rd = (rs1 & ~(1 << uimm))
0010011	001	011010*	uimm _{4:0}	I	binvi rd,rs1,uimm	bit invert immediate	rd = (rs1 ^ (1 << uimm))
0010011	001	001010*	uimm _{4:0}	I	bseti rd,rs1,uimm	bit set immediate	rd = (rs1 (1 << uimm))
0010011	101	010010*	uimm _{4:0}	I	bexti rd,rs1,uimm	bit extract immediate	rd = (rs1 & (1 << uimm))

* = 0 for RV32, uimm₅ for RV64.

• = 0 for rv32, 1 for rv64. rev8 and zext.h are versions of more general GREV, PACK, and PACKW instructions that may be provided in a future enhancement of the bit manipulation specification and that use the op and funct7 fields in more general ways.

RISC-V Atomic Memory Operation (AMO) Instructions

Table I.24 RVA: RISC-V AMO instructions

op	funct3	funct7/ imm _{11:5}	rs2	Type	Instruction	Description	Operation
0101111	010	00010, aq, rl	00000	I	lr.w rd, (rs1)	load reserved	rd=SignExt([rs1]), reserve [rs1]
0101111	010	00011, aq, rl	rs2	R	sc.w rd, rs2, (rs1)	store conditional	if ([rs1] still reserved) [rs1] _{31:0} = rs2 _{31:0} , rd = 0 #success else rd = 1 #fail
0101111	010	00001, aq, rl	rs2	R	amoswap.w rd, rs2, (rs1)	amo swap	rd=[rs1], [rs1] _{31:0} = rs2
0101111	010	00000, aq, rl	rs2	R	amoadd.w rd, rs2, (rs1)	amo add	rd=[rs1], [rs1] _{31:0} = [rs1] + rs2
0101111	010	00100, aq, rl	rs2	R	amoxor.w rd, rs2, (rs1)	amo xor	rd=[rs1], [rs1] _{31:0} = [rs1] ^ rs2
0101111	010	01100, aq, rl	rs2	R	amoand.w rd, rs2, (rs1)	amo and	rd=[rs1], [rs1] _{31:0} = [rs1] & rs2
0101111	010	01000, aq, rl	rs2	R	amoor.w rd, rs2, (rs1)	amo or	rd=[rs1], [rs1] _{31:0} = [rs1] rs2
0101111	010	10000, aq, rl	rs2	R	amomin.w rd, rs2, (rs1)	amo min	rd=[rs1], [rs1] _{31:0} = min ([rs1], rs2)
0101111	010	10100, aq, rl	rs2	R	amomax.w rd, rs2, (rs1)	amo max	rd=[rs1], [rs1] _{31:0} = max ([rs1], rs2)
0101111	010	11000, aq, rl	rs2	R	amominu.w rd, rs2, (rs1)	amo min unsign.	rd=[rs1], [rs1] _{31:0} = minu([rs1], rs2)
0101111	010	11100, aq, rl	rs2	R	amomaxu.w rd, rs2, (rs1)	amo max unsign.	rd=[rs1], [rs1] _{31:0} = maxu([rs1], rs2)

In this table, [rs1] = SignExt([rs1]_{31:0}), the sign-extended contents of memory address rs1. RV64A adds .d versions of all these instructions (i.e., lr.d, amoswap.d, etc.), with funct3 = 011; these operate on a 64-bit operand (double word). The aq (acquire) and rl (release) bits indicate additional ordering. When aq = 1, later memory operations in this hart are only seen after AMO. When rl = 1, other RISC-V harts will see all previous memory operations in this hart before AMO. If aq = rl = 1, all later/previous memory operations in this hart must occur after/before AMO.

RISC-V Integer Conditional Instructions

Table I.25 Zicond: Integer conditional instructions

op	funct3	funct7	rs2	Type	Instruction	Description	Operation
0110011	101	0000111	rs2	R	czero.eqz rd, rs1, rs2	conditional zero on equal zero	rd=(rs2 == 0) ? 0 : rs1
0110011	111	0000111	rs2	R	czero.nez rd, rs1, rs2	conditional zero on not equal zero	rd=(rs2 != 0) ? 0 : rs1

RISC-V Additional Floating-Point Instructions

Table I.26 Zfa: Additional RISC-V floating-point instructions

op	funct3	funct7/ imm _{11:5}	rs2/ imm _{4:0}	Type	Flags NV DZ OF UF NX	Instruction	Description	Operation
1010011	000	11110, fmt	00001	I	- - - - -	fli.* fd, rs1	load immediate	fd = imm[rs1]
1010011	010	00101, fmt	fs2	R	x - - - -	fminm.* fd, fs1, fs2	min	fd = min(fs1, fs2)
1010011	011	00101, fmt	fs2	R	x - - - -	fmaxm.* fd1, fs1, fs2	max	fd = max(fs1, fs2)
1010011	rm	01000, fmt	00100	I	x - - - -	fround.* fd, fs1	round to integer	fd = round(fs)
1010011	rm	01000, fmt	00101	I	x - - - x	froundnx.* fd, fs1	round to integer	fd = round(fs)
1010011	101	10100, fmt	fs2	R	x - - - -	fltq.* rd, fs1, fs2	compare <	rd = (fs1 < fs2)
1010011	100	10100, fmt	fs2	R	x - - - -	flteq.* rd, fs1, fs2	compare ≤	rd = (fs1 ≤ fs2)
Only RV32D								
1010011	000	1110001	00001	I	- - - - -	fmvh.x.d rd, fs1	move high	rd = fs1[63:31]
1010011	000	1011001	rs2	R	- - - - -	fmvp.d.x fd, rs1, rs2	move pair	fd = {rs2, rs1}
Only RV64Q								
1010011	000	1110011	00001	I	- - - - -	fmvh.x.q rd, fs1	move high	rd = fs1[127:64]
1010011	000	1011011	rs2	R	- - - - -	fmvp.q.x fd, rs1, rs2	move pair	fd = {rs2, rs1}
Only D Extension								
1010011	001	1100001	01000	I	x - - - x	fcvtmod.w.d rd, fs1, rtz	modular convert	rd = SignExt(trunc(fs1) mod 2 ³²)

imm[rs1] is a floating-point immediate specified by rs1, as shown in Table 16.52. In fcvtmod.w.d, rtz is the rounding mode.

RISC-V Cryptography Instructions

Table I.27 : Zbk*/Zkn*: RISC-V scalar cryptography instructions

op	funct3	funct7/ instr _{31:25}	rs2/ instr _{24:20}	Type	Instruction	Description	Operation
Zbkb: Bit Manipulation for Cryptography (also includes rol, ror, rori, andn, orni, xnor, rev8 from Zbb, Table I.23)							
0110011	100	0000100	rs2	R	pack rd,rs1,rs2	pack XLEN/2 in XLEN	$rd = \{rs2_{XLEN/2-1:0}, rs1_{XLEN/2-1:0}\}$
0110011	111	0000100	rs2	R	packh rd,rs1,rs2	pack 8 in 16	$rd = ZeroExt\{rs2_{7:0}, rs1_{7:0}\}$
0010011	101	0110100	00111	I	brev8 rd,rs1	bit reverse	$rd = reversebitsinbytes(rs1)$
Zbkb (RV32 only): Bit Manipulation for Cryptography							
0010011	001	0000100	01111	I	zip rd,rs1	hi/low -> odd/even	$rd_{even} = rs1_{15:0}; rd_{odd} = rs1_{31:16}$
0010011	101	0000100	01111	I	unzip rd,rs1	odd/even -> hi/low	$rd_{15:0} = rs1_{odd}; rd_{31:16} = rs1_{even}$
Zbkb (RV64 only): Bit Manipulation for Cryptography on 32-bit Words (also includes rolw, rorw, rorwi, Table I.23)							
0111011	100	0000100	rs2	R	packw rd,rs1,rs2	pack 16 in 32	$rd = ZeroExt\{rs2_{15:0}, rs1_{15:0}\}$
Zbkc: Carry-less Multiply (see c1mul and c1mulh from Zbc, Table I.23)							
Zbkx: Bit Manipulation for Cryptography							
0110011	010	0010100	rs2	R	xperm4 rd,rs1,rs2	permute nibbles	$rd = lookup4(rs1, rs2)$
0110011	100	0010100	rs2	R	xperm8 rd,rs1,rs2	permute bytes	$rd = lookup8(rs1, rs2)$
Zknd (RV32 only): AES Decryption							
0110011	000	{bs, 10101}	rs2	R	aes32dsi rd,rs1,rs2,bs	final round	$rd = rs1^{(isbox(rs2[bs]) \ll bs * 8)}$
0110011	000	{bs, 10111}	rs2	R	aes32dsmi rd,rs1,rs2,bs	middle round	$rd = rs1^{(imixcols(isbox(rs2[bs])) \ll bs * 8)}$
Zknd (RV64 only): AES Decryption & Inverse Key Schedule							
0110011	000	0011101	rs2	R	aes64ds rd,rs1,rs2	final round	$rd = isbox(ishiftrows(rs2, rs1))$
0110011	000	0011111	rs2	R	aes64dsm rd,rs1,rs2	middle round	$rd = imixcols(isbox(shiftrows(rs2, rs1)))$
0010011	001	0011000	00000	I	aes64im rd,rs1	InvMixCols	$rd = imixcols(rs1)$
Zkne (RV32 only): AES Encryption							
0110011	000	{bs, 10001}	rs2	R	aes32esi rd,rs1,rs2,bs	final round	$rd = rs1^{(sbox(rs2[bs]) \ll bs * 8)}$
0110011	000	{bs, 10011}	rs2	R	aes32esmi rd,rs1,rs2,bs	middle round	$rd = rs1^{(mixcols(sbox(rs2[bs])) \ll bs * 8)}$
Zkne (RV64 only): AES Encryption							
0110011	000	0011001	rs2	R	aes64es rd,rs1,rs2	final round	$rd = sbox(shiftrows(rs2, rs1))$
0110011	000	0011011	rs2	R	aes64esm rd,rs1,rs2	middle round	$rd = mixcols(sbox(shiftrows(rs2, rs1)))$
Zknd and Zkne (RV64 only): AES Key Schedule							
0010011	001	0011000	{1, rnum}	I	aes64ksi rd,rs1,rnum	key schedule	$rd = \{2^{(rcon[rnum] \wedge sbox(rot(rs1)))}\}$
0110011	000	0111111	rs2	R	aes64ks2 rd,rs1,rs2	key schedule	$w0 = rs1_{63:32} \wedge rs2_{31:0}; rd = \{w0 \wedge rs2_{63:32}, w0\}$
Zknh: SHA2-256 Hash*							
0010011	001	0001000	00010	I	sha256sig0 rd,rs1	$\sigma_0^{(256)}$ function	$rd = (rs1 \text{ ror } 7) \wedge (rs1 \text{ ror } 18) \wedge (rs1 \gg 3)$
0010011	001	0001000	00011	I	sha256sig1 rd,rs1	$\sigma_1^{(256)}$ function	$rd = (rs1 \text{ ror } 17) \wedge (rs1 \text{ ror } 19) \wedge (rs1 \gg 10)$
0010011	001	0001000	00000	I	sha256sum0 rd,rs1	$\Sigma_0^{(256)}$ function	$rd = (rs1 \text{ ror } 2) \wedge (rs1 \text{ ror } 13) \wedge (rs1 \text{ ror } 22)$
0010011	001	0001000	00001	I	sha256sum1 rd,rs1	$\Sigma_1^{(256)}$ function	$rd = (rs1 \text{ ror } 6) \wedge (rs1 \text{ ror } 11) \wedge (rs1 \text{ ror } 25)$
Zknh (RV32 only): SHA2-512 Hash							
0110011	000	0101110	rs2	R	sha512sig0h rd,rs1,rs2	$\sigma_0^{(512)}$ high word	$rd = ((rs2, rs1) \gg 1)_{31:0} \wedge ((rs2, rs1) \gg 8)_{31:0} \wedge (rs1 \gg 7)$
0110011	000	0101010	rs2	R	sha512sig0l rd,rs1,rs2	$\sigma_0^{(512)}$ low word	$rd = ((rs2, rs1) \gg 1)_{31:0} \wedge ((rs2, rs1) \gg 8)_{31:0} \wedge ((rs2, rs1) \gg 7)_{31:0}$
0110011	000	0101111	rs2	R	sha512sig1h rd,rs1,rs2	$\sigma_1^{(512)}$ high word	$rd = ((rs2, rs1) \gg 29)_{31:0} \wedge ((rs2, rs1) \gg 19)_{31:0} \wedge (rs1 \gg 6)$
0110011	000	0101011	rs2	R	sha512sig1l rd,rs1,rs2	$\sigma_1^{(512)}$ low word	$rd = ((rs2, rs1) \gg 29)_{31:0} \wedge ((rs2, rs1) \gg 19)_{31:0} \wedge ((rs2, rs1) \gg 6)_{31:0}$
0110011	000	0101000	rs2	R	sha512sum0r rd,rs1,rs2	$\Sigma_0^{(512)}$ half	$rd = ((rs2, rs1) \gg 7)_{31:0} \wedge ((rs2, rs1) \gg 2)_{31:0} \wedge ((rs2, rs1) \gg 28)_{31:0}$
0110011	000	0101001	rs2	R	sha512sum1r rd,rs1,rs2	$\Sigma_1^{(512)}$ half	$rd = ((rs2, rs1) \gg 9)_{31:0} \wedge ((rs2, rs1) \gg 14)_{31:0} \wedge ((rs2, rs1) \gg 18)_{31:0}$
Zknh (RV64 only): SHA2-512 Hash							
0010011	001	0001000	00110	I	sha512sig0 rd,rs1	$\sigma_0^{(512)}$ function	$rd = (rs1 \text{ ror } 1) \wedge (rs1 \text{ ror } 8) \wedge (rs1 \gg 7)$
0010011	001	0001000	00111	I	sha512sig1 rd,rs1	$\sigma_1^{(512)}$ function	$rd = (rs1 \text{ ror } 19) \wedge (rs1 \text{ ror } 61) \wedge (rs1 \gg 6)$
0010011	001	0001000	00100	I	sha512sum0 rd,rs1	$\Sigma_0^{(512)}$ function	$rd = (rs1 \text{ ror } 28) \wedge (rs1 \text{ ror } 34) \wedge (rs1 \text{ ror } 39)$
0010011	001	0001000	00101	I	sha512sum1 rd,rs1	$\Sigma_1^{(512)}$ function	$rd = (rs1 \text{ ror } 14) \wedge (rs1 \text{ ror } 18) \wedge (rs1 \text{ ror } 41)$

bs=2-bit byte select. rnum=4-bit round number. *RV64's SHA2-256 instructions operate on rs1's low 32 bits and sign-extend the final result to 64 bits.