

ARM64

Assembly language on **Apple silicon**: registers, memory, the stack, the calling convention and how to read what the compiler wrote.

```
gcd.s - clang -target arm64-apple-macos14 -c
```

```
_gcd:
    cbz    x1, done
loop:
    udiv   x2, x0, x1    // q = a / b
    msub   x2, x2, x1, x0 // r = a - q*b
    mov    x0, x1
    mov    x1, x2
    cbnz   x1, loop
done:
    ret
```

```
9ac10802 9b018042 aa0103e0 aa0203e1 b5ffff81 d65f03c0
```

```
^ the same instructions as real 32-bit words, straight from the assembler
```

Contents

0	Before You Start	4
0.1	Who this is for	4
0.2	The three tools	4
0.3	What was verified, and how	5
1	How the Machine Thinks	6
1.1	The one trick	6
1.2	What assembly language is	6
1.3	Why bother?	7
1.4	The shape of A64	7
1.5	Your first look at compiler output	8
2	Registers and Data	10
2.1	The general-purpose registers	10
2.2	The special registers	11
2.3	Numbers in registers	12
2.4	Getting a constant into a register	12
2.5	Vector and floating-point registers	14
3	Instructions and Arithmetic	15
3.1	Anatomy of a line	15
3.2	Inside the 32 bits	15
3.3	Basic arithmetic and logic	16
3.4	Shifts, bit fields and "free" shifting	17
3.5	Aliases: why the disassembly looks different	18
4	Memory and Addressing	20
4.1	Memory is a long row of numbered bytes	20
4.2	Load and store	20
4.3	The addressing modes	21
4.4	Walking an array	22
4.5	Strings are just bytes	23
4.6	Structs: fields are fixed offsets	24
5	Flags and Branches	25
5.1	The four flags	25
5.2	Condition codes	26
5.3	Branches	26
5.4	If, else and loops	27
5.5	Branch-free code: <code>csel</code> and friends	28
5.6	A clever one: <code>ccmp</code>	28

6	Functions and the Stack	30
6.1	Calling and returning	30
6.2	The calling convention	30
6.3	The stack	31
6.4	A function that calls another	32
6.5	Recursion: one frame per call	33
6.6	Local variables on the stack	34
6.7	Calling a library function	35
7	Programs, Data and the OS	36
7.1	What the assembler gives you	36
7.2	Directives	37
7.3	Finding data: <code>adrp</code>	37
7.4	Hello, world with a raw system call	38
7.5	Hello, world the normal way	39
7.6	Variadic functions: Apple's special rule	40
8	Floating Point and NEON	42
8.1	Scalar floating point	42
8.2	NEON: SIMD	43
8.3	Two syntaxes for the same thing	44
8.4	What the compiler does by itself	45
8.5	Intrinsics: NEON from C and C++	46
9	Reading Compiler Output	47
9.1	A routine for reading any function	47
9.2	Division by a constant: no divide at all	47
9.3	Common transformations to recognise	49
9.4	Calls and frames in compiler output	49
9.5	Inlining: calls that vanish	50
9.6	Puzzles	50
10	Debugging and Tools	52
10.1	What stepping looks like	52
10.2	LLDB: the commands you need	53
10.3	Reading a crash	53
10.4	Pointer authentication (arm64e), briefly	54
10.5	Other tools on your Mac	54
11	Reference Pages	55
11.1	The function template	55
11.2	Instruction cards	55
11.3	Condition codes at a glance	57
11.4	Calling convention card	57
11.5	Common idioms	57
11.6	Glossary	58
11.7	Where to read next	58

Before You Start

What this booklet teaches, the three tools you need, and an honest account of what was and was not verified.

0.1 Who this is for

You can write a little C, C++ or Swift and want to know what the computer actually does with it. By the end you will be able to read the assembly that clang produces for an Apple silicon Mac (M1 through M5), write small routines by hand, follow a stack frame through a debugger, and understand why a compiler does what it does. We cover **AArch64** (the 64-bit Arm instruction set, called A64) as Apple uses it on macOS. iOS is the same instruction set with small differences that we do not cover.

Read chapters 1–6 in order; each one uses the last. Chapters 7–10 can then be read in any order. Chapter 11 is a reference you will come back to.

Remember • A teal box holds a rule of thumb.

Pitfall • A coral box is a trap people fall into.

Apple-specific • Violet marks something that differs from other Arm platforms (Linux, Windows) or is a macOS convention.

Reading a listing

✓ assembled & simulated

```
// comments are gray italics
add x0, x1, x2 // mnemonics
x0 w1 sp xzr v0 // registers
.globl .p2align // directives
```

SIMULATOR CHECKS

dark strips are tool output

0.2 The three tools

Everything below uses tools that ship with Apple's command-line developer tools (`xcode-select --install`). You do not need the full Xcode.

Command	What it does
<code>clang -c foo.s -o foo.o</code>	Assemble a file of assembly into an object file.
<code>clang foo.s -o foo</code>	Assemble <i>and</i> link into a program you can run (<code>./foo</code>).
<code>clang -S -O2 foo.c -o foo.s</code>	Compile C to assembly and stop, so you can read it.
<code>otool -tV foo.o</code>	Disassemble an object or executable (Apple's tool; <code>objdump -d</code> from LLVM does the same).
<code>lldb ./foo</code>	Run under the debugger: step one instruction, print registers, look at memory (chapter 10).

On an Apple silicon Mac no `-target` flag is needed; I used `-target arm64-apple-macos14` so the same commands work from the Linux machine I had.

Your first assemble-and-run (on your Mac)

```
cat > hello.s <<'EOS'      # (or save chapter 7's hello.s)
...
EOS
clang hello.s -o hello      # assemble + link
```

```
./hello                # -> Hello, ARM64!
otool -tV hello        # see your own instructions
```

0.3 What was verified, and how

I could not run ARM code while writing this: the machine that produced this booklet is not an Apple silicon Mac. So I used the strongest checks that machine allowed, and I tag every listing so you can see which checks it has.

✓ assembled

The file was assembled by clang 18 targeting arm64-apple-macos14. An invalid instruction or operand would have been rejected, so the *syntax and encodings are real*. The machine-code words printed in this book come from the assembler's output.

✓ assembled & simulated

As above, and I also ran the same text through a small A64 interpreter I wrote (about 600 lines of Python) and compared the results with a Python reference over several inputs. This checks the *logic*. It is my interpreter, not Apple's chip: it has no caches, no memory ordering and covers only the instructions used here. I verified it can fail by deliberately breaking a listing and watching the checks go red.

◇ real clang output

Assembly that clang 18 itself generated from the C shown above it (at -O2 unless stated). Real code, but not simulated. Your Xcode's clang may print slightly different instructions.

What this does not prove • Nothing here ran on silicon. Performance statements are explanations, not measurements. Syscall numbers, the 128-byte cache line, and anything about pointer authentication are things I know from documentation and experience, marked (*hedged*) where I am less sure; confirm with `sysctl` or Apple's documentation on your machine.

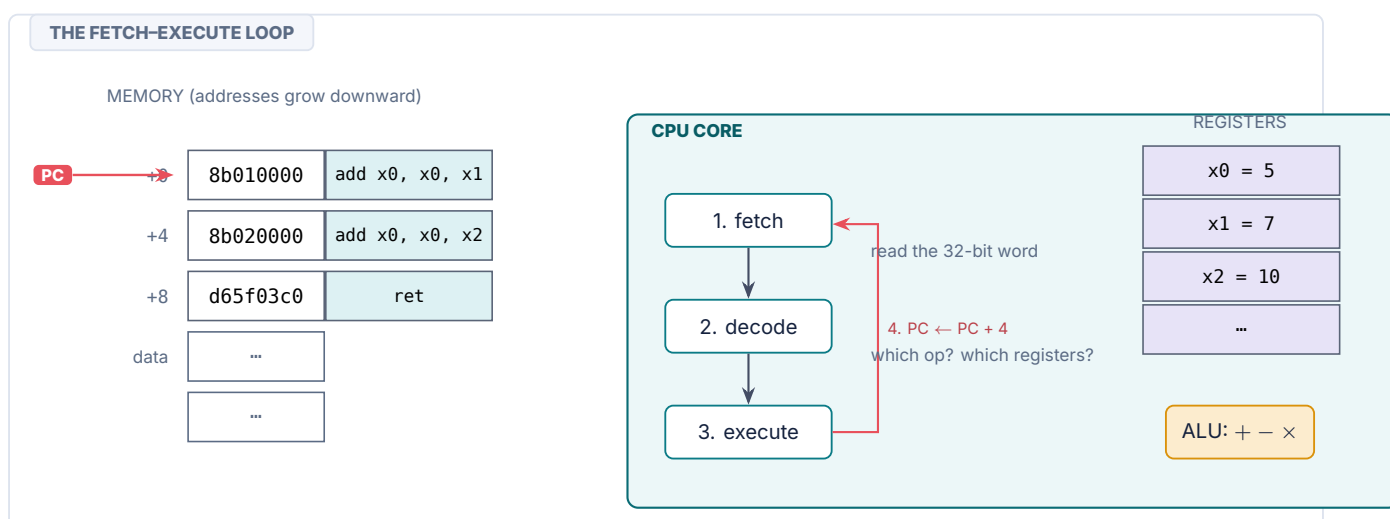
How the Machine Thinks

A processor does one thing, over and over: fetch an instruction, do it, move on. Assembly language is just those instructions spelled with letters.

01

1.1 The one trick

Your Mac's CPU has no idea what a function, a class or a loop is. It has a short list of tiny operations (add two numbers, copy a value to or from memory, jump somewhere else) and it performs them one after another, billions of times a second. A program is a long list of those operations stored in memory. A register called the **program counter** (PC) holds the address of the next one.



That is the whole story. Everything else (functions, objects, a window on screen) is built by arranging long lists of these steps. The fixed-size steps are called **instructions**, and on Apple silicon each one is exactly **32 bits (4 bytes)** wide. That is why the PC moves by 4 each time, and why the addresses in the picture are +0, +4, +8.

Remember • Registers are the CPU's own handful of very fast variables. Memory is large and comparatively slow. Almost all arithmetic happens *between registers*; memory is where values live when there are too many to keep in registers.

1.2 What assembly language is

The three words in the picture's memory column are real machine code: 8b010000, 8b020000, d65f03c0 are what a CPU actually reads. Nobody wants to write hexadecimal, so we write a text spelling of each word, and a program called the **assembler** turns the text back into words. The spelling is **assembly language**. The mapping is almost one-to-one: one line, one instruction.

Here is a complete function. It takes three integers and returns their sum.

add3: your first function

✓ assembled & simulated

```
// int64_t add3(int64_t a, int64_t b, int64_t c) -- args arrive in x0, x1, x2; result goes in x0
.globl _add3
.p2align 2
_add3:
    add    x0, x0, x1      // x0 = a + b
    add    x0, x0, x2      // x0 = (a + b) + c
    ret                                // jump to the address in x30 (the link register)
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

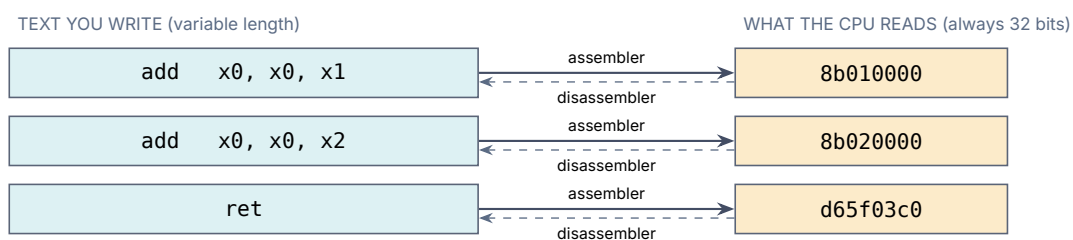
```
add3(1, 2, 3)           = 6
add3(-5, 2, 1)          = -2
add3(1000000000000, 1000000000000, -1) = 1999999999999
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: 8b010000  add x0, x0, x1
4: 8b020000  add x0, x0, x2
8: d65f03c0  ret
```

Read it like this. The function receives its three arguments in registers `x0`, `x1` and `x2` (a rule called the *calling convention*, chapter 6). The first `add` writes `x0 + x1` back into `x0`. The second adds `x2`. By convention the result of a function is left in `x0`, and `ret` jumps back to whoever called us. The C equivalent is `long add3(long a, long b, long c)` returning `a + b + c`.

Look at the machine code strip under the listing: the first column is the byte offset of each instruction from the start of the function (0, 4, 8: four bytes apart), the second is the 32-bit word, and the third is the assembly the disassembler recovered. The assembler wrote those words; I did not. In chapter 3 we will take one of them apart bit by bit.

LISTING ⇄ MACHINE WORD: ONE LINE, ONE INSTRUCTION**1.3 Why bother?**

You will rarely write large programs in assembly. You will often *read* it, and knowing how is a superpower:

- **Crash reports and debuggers** show you the instruction that faulted and the registers at that moment.
- **Performance work:** is the loop vectorized? is that call inlined? Only the assembly answers.
- **Understanding the language:** pointers, volatile, integer overflow, calling conventions and ABI compatibility stop being magic once you have seen what they turn into.
- **Low-level code:** tiny routines in a runtime, a GPU driver's CPU side, an interpreter's hot loop, SIMD kernels.

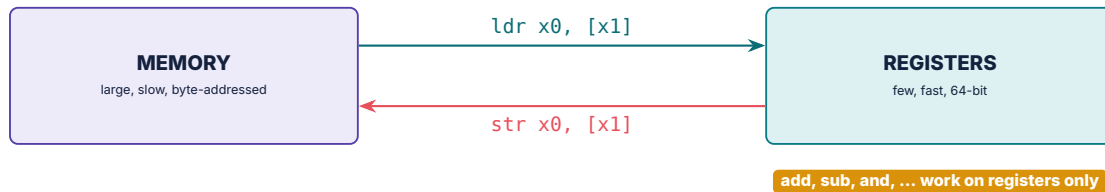
1.4 The shape of A64

Four facts will carry you through the rest of the book.

1. **Fixed width.** Every instruction is 32 bits. Compare with x86, where instructions are 1 to 15 bytes. Fixed width makes decoding simple and fast, but it means an instruction cannot hold a full 64-bit constant (chapter 2 shows the workaround).
2. **Load/store.** Only a few instructions (`ldr`, `str` and relatives) touch memory. Arithmetic works on registers only. To add a number from memory you load it, add, and maybe store it back.

3. **Lots of registers.** 31 general-purpose 64-bit registers (x0–x30) plus a stack pointer and a zero register, and 32 vector/floating-point registers. Compilers rarely run out.
4. **Three operands.** Most arithmetic names a destination and two sources: `add x0, x1, x2` means $x0 = x1 + x2$. The destination is *always first*.

LOAD/STORE: ONLY `ldr` AND `str` CROSS THE LINE



1.5 Your first look at compiler output

The best way to learn assembly is to ask the compiler for it. Take the smallest function there is:

Compile with `clang -S -O2` ◇ real clang 18 output; not simulated

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

```
_add:
    add    w0, w1, w0
    ret
```

Two instructions. Notice what the compiler did without being asked: it used `w` registers (the 32-bit halves, because the C type is `int`) and it wrote the operands as `w1, w0`, the opposite order from the source. Addition is commutative, so the order does not matter and the compiler is free to choose. Now the same function with optimization *off*:

The same function at `-O0` ◇ real clang 18 output; not simulated

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

```
_add:
    sub    sp, sp, #16
    str    w0, [sp, #12]
    str    w1, [sp, #8]
    ldr    w8, [sp, #12]
    ldr    w9, [sp, #8]
    add    w0, w8, w9
    add    sp, sp, #16
    ret
```

At `-O0` every argument is spilled to the stack, reloaded, added, and the stack is cleaned up: eight instructions where two will do. Debuggers love this (every variable lives in memory where they can see it), and optimizers hate it. Throughout the book, `-O2` output is what you will see in real programs, so that is what we read.

Pitfall • Compiler output changes between versions and flags. Never memorize *which* instructions it picks; learn to *read* any of them. The listings here come from clang 18; Apple's clang in your Xcode will differ in details.

Chapter 1 in five lines

- The CPU fetches a 32-bit instruction at the PC, executes it, and advances the PC by 4.
- Assembly is a text spelling of those words; the assembler converts, the disassembler converts back.

- A64 is fixed-width, load/store, has many registers, and is three-operand with the destination first.
- `ret` jumps back to the caller; arguments come in `x0...`; the result goes in `x0`.
- `clang -S -O2` shows you what the compiler wrote.

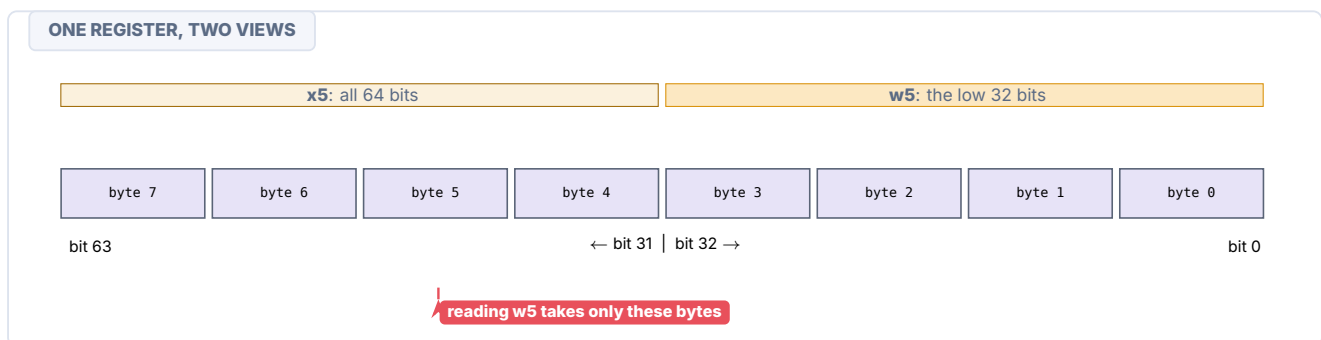
Registers and Data

The 31 general-purpose registers, their 32-bit views, the special ones, and how a 64-bit constant is built from 16-bit pieces.

02

2.1 The general-purpose registers

A64 gives you 31 general-purpose registers named `x0` to `x30`. Each holds 64 bits. You can also use the low half of any of them as a 32-bit register by changing the letter: `w0` is the bottom 32 bits of `x0`. They are the same physical storage seen two ways.



The rule that surprises everyone: **writing a w register zeroes the upper 32 bits of the matching x register.** The CPU never leaves half a register stale. This is also why 32-bit int code on a 64-bit machine needs no extra clean-up. See it happen:

w views, zero- and sign-extension ✓ assembled & simulated

```
// Starts with every bit of x0 set, then shows how w views, sign- and zero-extension behave.
.globl _widths
.p2align 2
_widths:
    mov     x0, #-1           // x0 = 0xffffffffffffffff (assembler picks movn)
    mov     w1, w0            // w1 = 0xffffffff; the TOP half of x1 is cleared to 0
    sxtw    x2, w1            // sign-extend 32 -> 64 bits: x2 = 0xffffffffffffffff
    uxtb    w3, w0            // zero-extend the low byte: w3 = 0xff
    mov     w0, #7            // writing w0 zeroes the top half of x0: x0 = 7
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
x0 after  mov w0, #7           = 0x7
x1 after  mov w1, w0           = 0xffffffff
x2 after  sxtw x2, w1          = 0xffffffffffffffff
x3 after  uxtb w3, w0          = 0xff
```

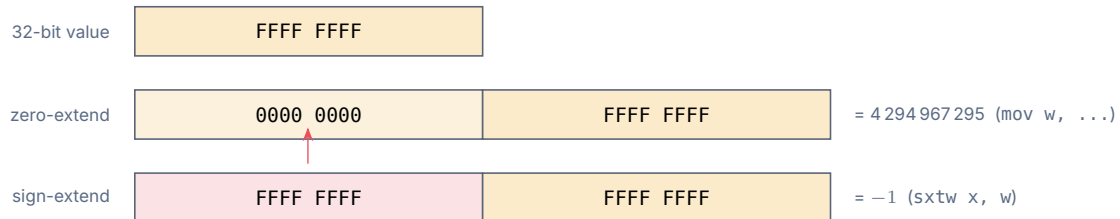
MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: 92800000  mov x0, #-0x1 ; =-1
4: 2a0003e1  mov w1, w0
8: 93407c22  sxtw x2, w1
c: 53001c03  uxtb w3, w0
10: 528000e0  mov w0, #0x7 ; =7
14: d65f03c0  ret
```

Four things are going on. `mov w1, w0` copies 32 bits and clears the rest (look at the encoding `2a0003e1`: the

disassembler calls it `mov`, but the real instruction is an `orr` with the zero register; the *alias* idea comes back in chapter 3). **sxtw** *sign-extends*: it copies bit 31 into all the upper bits, which is exactly what C does when you convert an `int` to a `long`. **uxtb** *zero-extends* a byte. And the last `mov w0, #7` leaves `x0` equal to exactly 7, not `0xffffffff00000007`.

ZERO-EXTEND vs SIGN-EXTEND A 32-BIT VALUE 0xFFFFFFFF INTO 64 BITS



sign bit (bit 31) is copied into every new upper bit

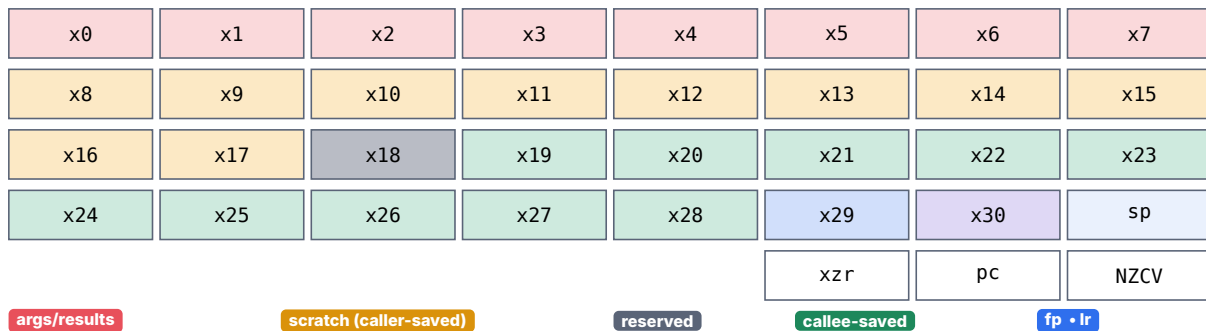
2.2 The special registers

Some register names mean something unusual. A few are hardware facts and a few are *conventions* that all Apple code follows.

Name	Kind	What it is
x0 - x7	convention	Arguments in, results out (chapter 6).
x8	convention	Indirect result address; otherwise a free scratch register.
x9 - x15	convention	Scratch: any function may overwrite them.
x16, x17	convention	Scratch for the linker's stubs (IP0, IP1). Also: x16 carries the syscall number on macOS.
x18	reserved	Never use on Apple platforms. Apple reserves it for the OS.
x19 - x28	convention	Callee-saved: a function must restore them before returning.
x29 (fp)	convention	Frame pointer. Apple requires it to point to a valid frame record.
x30 (lr)	hardware	Link register: <code>bl</code> stores the return address here; <code>ret</code> jumps to it.
sp	hardware	Stack pointer. Must be 16-byte aligned whenever it is used to access memory.
xzr, wzr	hardware	The zero register: reads as 0, writes are discarded.
pc	hardware	Program counter. You cannot name it in most instructions; only branches change it.
NZCV	hardware	Four condition flags set by compare instructions (chapter 5).

Pitfall • x18 is the platform register. Linux code sometimes uses it freely; on macOS and iOS you must treat it as untouchable, or the OS may overwrite your value or your program may be rejected. And `sp` is *not* a general register: most instructions that accept x registers treat register number 31 as `xzr`, and only a few (`add/sub` with an immediate, loads and stores) treat it as `sp`. The assembler picks correctly, but it explains some strange error messages.

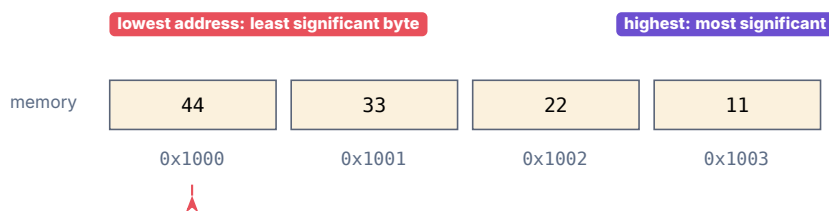
THE REGISTER MAP



2.3 Numbers in registers

A register holds just bits; whether they mean -1 or 18446744073709551615 depends on the instruction you apply. Signed integers use **two's complement**: to negate, invert every bit and add one. The top bit is the sign. So all 64 bits set is -1 , and $0x8000\ 0000\ 0000\ 0000$ is the most negative value.

Memory is **little-endian** on Apple silicon: the least significant byte of a multi-byte value is stored at the lowest address. The number $0x11223344$ in memory is the bytes `44 33 22 11`.

LITTLE-ENDIAN: THE 32-BIT VALUE $0x11223344$ STORED AT ADDRESS $0x1000$ 

Here are the C types and how they sit in registers. All of them fit in one register; narrower ones are extended to 32 or 64 bits as needed.

C / C++ type	Bytes	Register view	Notes
char, uint8_t	1	w	Loaded with <code>ldrb</code> , zero-extended.
short, uint16_t	2	w	<code>ldrh</code> , zero-extended (<code>ldrsh</code> to sign-extend).
int, uint32_t, float	4	w (float: s)	The common case; w registers.
long, long long, size_t	8	x	long is 8 bytes on macOS , unlike Windows.
pointers, int64_t	8	x	Pointers are plain 64-bit integers.
double	8	d	Lives in a vector register (chapter 8).

2.4 Getting a constant into a register

Every instruction is only 32 bits, so there is no room for a 64-bit constant inside one. A64 solves this in pieces: `movz` writes a 16-bit value into a chosen 16-bit slot and zeroes the rest; `movk` (*keep*) overwrites one slot and leaves the others alone. Four moves build any 64-bit value.

Building $0x123456789abcdef0$ one halfword at a time

✓ assembled & simulated

```
// uint64_t magic(void) -- building a 64-bit constant 16 bits at a time
.globl _magic
.p2align 2
```

```

_magic:
    movz    x0, #0xdef0                // x0 = 0x000000000000def0 (other bits zeroed)
    movk    x0, #0x9abc, lsl #16       // replace bits 16..31 only
    movk    x0, #0x5678, lsl #32       // replace bits 32..47
    movk    x0, #0x1234, lsl #48       // replace bits 48..63
    ret                                // x0 = 0x123456789abcdef0

```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
_magic() = 0x123456789abcdef0
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

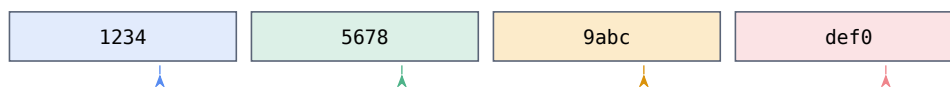
```

0: d29bde00  mov x0, #0xdef0 ; =57072
4: f2b35780  movk x0, #0x9abc, lsl #16
8: f2cacf00  movk x0, #0x5678, lsl #32
c: f2e24680  movk x0, #0x1234, lsl #48
10: d65f03c0  ret

```

FOUR HALFWORDS BECOME ONE 64-BIT CONSTANT

x0 after step 4:



- | | |
|------------------------------|------------|
| 1. mov x0, #0xdef0 | bits 15–0 |
| 2. movk x0, #0x9abc, lsl #16 | bits 31–16 |
| 3. movk x0, #0x5678, lsl #32 | bits 47–32 |
| 4. movk x0, #0x1234, lsl #48 | bits 63–48 |

The assembler does this for you only in a restricted way. Try to hand it a constant that does not fit and it refuses, with a message you will learn to recognize:

Assembler: a 64-bit constant in one mov ✓ real clang 18 error

```

a.s:2:11: error: expected compatible register or logical immediate
    mov x0, #0x123456789abcdef0
    ^

```

Assembler: add immediates are 12 bits (0–4095) ✓ real clang 18 error

```

b.s:2:15: error: expected compatible register, symbol or integer in range [0, 4095]
    add x0, x1, #5000
    ^

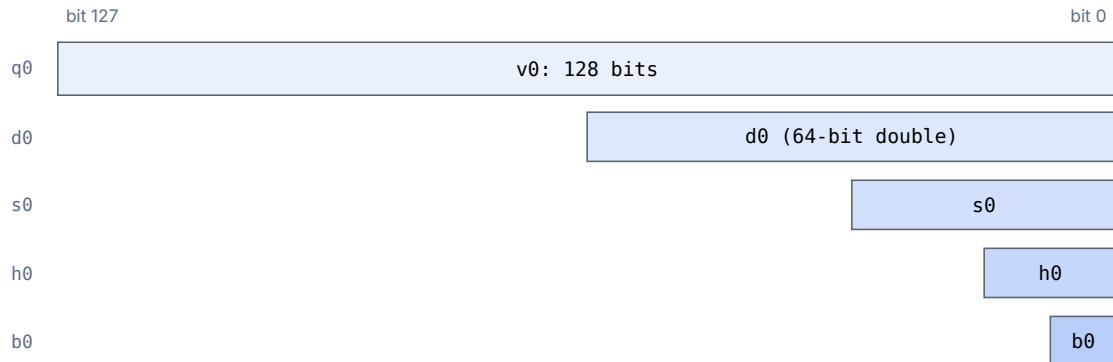
```

Remember • Small constants are cheap, big ones cost several instructions or a load from memory. A mov of #-1 is one instruction (movn, *move not*). Constants like 0x00ff00ff00ff00ff also fit one instruction as a *logical immediate* (a repeating bit pattern), which is why and, orr and eor accept them.

2.5 Vector and floating-point registers

Alongside the integer registers there are 32 vector registers, `v0` to `v31`, each 128 bits wide. Scalar floating point uses their low bits (`s0` for a float, `d0` for a double), and NEON SIMD instructions use all of them at once. Chapter 8 covers them. The only thing to know now is that they are a separate bank: `x0` and `d0` are unrelated storage.

ONE VECTOR REGISTER, MANY VIEWS



Chapter 2 in five lines

- `x0`–`x30` are 64-bit; `w0`–`w30` are their low halves. Writing `w` zeroes the top half.
- `xzr/wzr` read as zero; `sp` must stay 16-byte aligned; `x18` is off limits on Apple platforms.
- Memory is little-endian; `long` and pointers are 8 bytes.
- A 64-bit constant takes up to four instructions: `movz` then `movk` for each other halfword.
- `v0`–`v31` are a separate 128-bit bank for floating point and SIMD.

Instructions and Arithmetic

What is inside a 32-bit instruction, the shape of every arithmetic operation, and the small set of tricks (shifts, bit fields, multiply-add) that replace whole sequences elsewhere.

03

3.1 Anatomy of a line

ONE LINE OF ASSEMBLY

```
loop:  add  x2, x2, x3  // total += x3
```

label

mnemonic

operands: destination first

comment

A **label** (loop:) names an address so branches can find it. The **mnemonic** says what to do. The **operands** follow, comma-separated, *destination first*. Comments start with // (and /* */ also works); older Arm code uses ; or @, which clang for macOS also accepts in some places, but this book sticks to //.

Immediate (constant) operands start with #: add x0, x0, #1. Registers need no prefix.

3.2 Inside the 32 bits

You never need to decode instructions by hand, but doing it *once* makes everything else click: it shows why some constants are allowed and others are not, and why branches have limited reach. The pictures below were generated from the actual words the assembler produced, so each one is checked against a real encoding.

add x0, x0, x1 (REGISTER FORM)

31	30	29	28–24	23	21	20–16	15–10	9–5	4–0
1	0	0	01011	00	0	00001	000000	00000	00000
sf	op	S	shift			Rm	imm6	Rn	Rd

8b010000 = add x0, x0, x1

The top bit **sf** says 64-bit (1) or 32-bit (0) operation: this single bit is the difference between add x0, ... and add w0, ... **op**=0 means add (1 would mean subtract) and **S**=0 means "do not set flags". Then come the register numbers: **Rm** and **Rn** are the two sources and **Rd** is the destination, five bits each, which is exactly why there are 32 of them (register 31 is xzr or sp). **shift** and **imm6** let the second source be shifted left or right by up to 63 first; here the shift amount is 0.

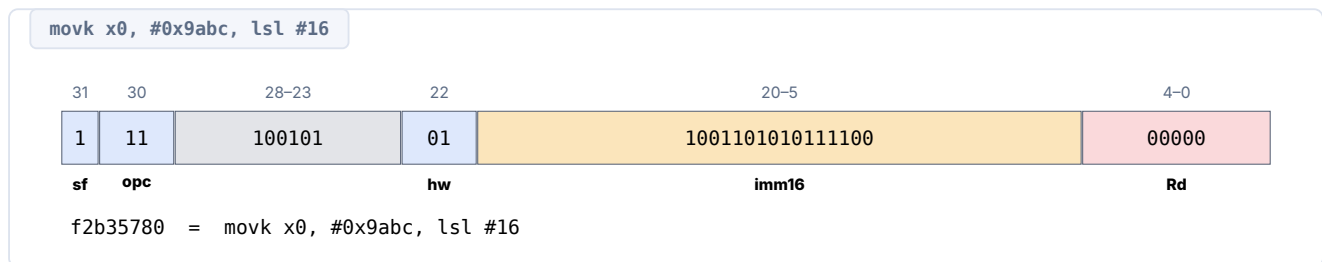
sub x1, x0, #1 (IMMEDIATE FORM)

31	30	29	28–23	22	21–10	9–5	4–0
1	1	0	100010	0	000000000001	00000	00001
sf	op	S	sh		imm12	Rn	Rd

d1000401 = sub x1, x0, #0x1

The constant lives *inside* the instruction in the 12-bit **imm12** field, which is the reason an add immediate must

be 0–4095 (you saw the assembler’s error in chapter 2). The **sh** bit optionally shifts that constant left by 12, so add `x0, x1, #5, lsl #12` can reach multiples of 4096 up to 16 773 120. Only **op**=1 changed from the previous picture: that is subtraction.



Sixteen bits of constant (**imm16**) and two bits (**hw**) choosing which 16-bit slot: slot 1 here (bits 16–31). This is the instruction from the last chapter’s constant-building listing, now seen from the inside.

Remember • Everything in an instruction word is either an *opcode* bit (fixed grey), a *register number* (red), a *constant* (amber) or a small *control* field (blue, green). Limits on constants, registers and branch distances all come from how many bits are left over.

3.3 Basic arithmetic and logic

Instruction	Meaning (C-like)
<code>add x0, x1, x2</code> / <code>add x0, x1, #5</code>	<code>x0 = x1 + x2</code> / <code>x0 = x1 + 5</code>
<code>sub x0, x1, x2</code>	<code>x0 = x1 - x2</code>
<code>neg x0, x1</code>	<code>x0 = -x1</code> (an alias: <code>sub x0, xzr, x1</code>)
<code>mul x0, x1, x2</code>	<code>x0 = x1 * x2</code> (low 64 bits of the product)
<code>madd x0, x1, x2, x3</code>	<code>x0 = x1 * x2 + x3</code>
<code>msub x0, x1, x2, x3</code>	<code>x0 = x3 - x1 * x2</code>
<code>udiv / sdiv x0, x1, x2</code>	unsigned / signed divide; divide by zero gives 0, no trap
<code>and / orr / eor x0, x1, x2</code>	bitwise AND / OR / XOR (orr not or)
<code>bic / orn / eon</code>	AND, OR, XOR with the second operand inverted
<code>mvn x0, x1</code>	<code>x0 = ~x1</code>
<code>lsl / lsr / asr x0, x1, #n</code>	shift left / logical right (zero fill) / arithmetic right (sign fill)
<code>ror x0, x1, #n</code>	rotate right

Pitfall • There is no remainder instruction. To compute `a % b` you divide and use `msub`: $r = a - (a/b) \cdot b$. The next listing does exactly that inside Euclid’s algorithm. There is also no “push” or “pop” and no instruction that adds memory to a register.

Seeing it work: Euclid’s GCD

gcd: divide, multiply-subtract, loop ✓ assembled & simulated

```
// uint64_t gcd(uint64_t a, uint64_t b)  Euclid: gcd(a, b) = gcd(b, a mod b)
    .globl _gcd
    .p2align 2
_gcd:
    cbz    x1, done                    // b == 0 -> answer is a
loop:
    udiv   x2, x0, x1                  // q = a / b
    msub   x2, x2, x1, x0              // r = a - q * b  (there is no 'mod' instruction)
```

```

mov    x0, x1           // a = b
mov    x1, x2           // b = r
cbnz   x1, loop
done:
ret                    // result is in x0

```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```

gcd(48, 18)           = 6
gcd(17, 5)            = 1
gcd(100, 0)           = 100
gcd(0, 7)             = 7
gcd(1099511627776, 3145728) = 1048576

```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```

0: b40000c1  cbz x1, 0x18 <done>
4: 9ac10802  udiv x2, x0, x1
8: 9b018042  msub x2, x2, x1, x0
c: aa0103e0  mov x0, x1
10: aa0203e1  mov x1, x2
14: b5ffff81  cbnz x1, 0x4 <loop>
18: d65f03c0  ret

```

TRACING gcd(48, 18) THROUGH THE LOOP

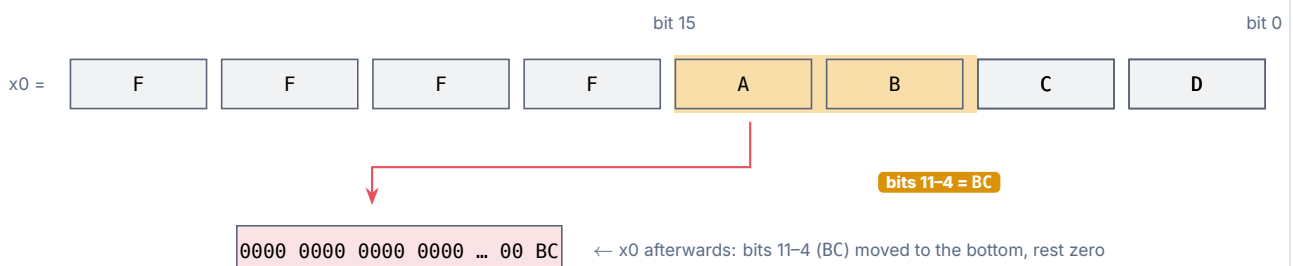
pass 1	x0 = 48	x1 = 18	udiv: x2 = 2	msub: x2 = 48 - 2×18= 12
pass 2	x0 = 18	x1 = 12	udiv: x2 = 1	msub: x2 = 18 - 1×12= 6
pass 3	x0 = 12	x1 = 6	udiv: x2 = 2	msub: x2 = 12 - 2×6= 0
result	x0 = 6	when x1 (the remainder) reaches 0, x0 holds the gcd		

3.4 Shifts, bit fields and “free” shifting

A64 can shift the *second source operand* as part of many instructions at no extra cost. That is the **shifted register** form, and compilers use it constantly for array indexing: `add x0, x1, x2, lsl #3` computes $x1 + x2 \times 8$, one instruction for “base plus index times eight”. (You will see this in chapter 9: even $x * 9$ compiles to `add x0, x0, x0, lsl #3`.)

Bit-field instructions pull a run of bits out of a register in one step. `ubfx x0, x1, #lsb, #width` (*unsigned bit-field extract*) is “shift right by `lsb` then mask to `width` bits”. Packed formats, hardware registers and flags words are full of these.

`ubfx x0, x0, #4, #8: EXTRACT BITS 11 DOWN TO 4`



field and is_pow2: a bit field and a bit trick

✓ assembled & simulated

```
// uint64_t field(uint64_t x) : bits [11:4] of x, i.e. (x >> 4) & 0xff
// bool is_pow2(uint64_t x) : true for 1, 2, 4, 8, ...
    .globl _field
    .p2align 2
_field:
    ubfx    x0, x0, #4, #8           // unsigned bit-field extract: start bit 4, width 8
    ret

    .globl _is_pow2
    .p2align 2
_is_pow2:
    sub     x1, x0, #1              // x-1 flips the lowest set bit and all below
    tst     x0, x1                  // x & (x-1) is zero exactly when x has one bit set...
    ccmp    x0, #0, #4, eq          // ...but 0 must not count: eq: compare x,0; else Z=1
    cset     w0, ne                  // w0 = 1 if (x & (x-1)) == 0 and x != 0
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
field(0xabcd)           = 0xbc
field(0xffffffffffff)   = 0xff
field(0x10)              = 0x1
is_pow2(0)               = 0
is_pow2(1)               = 1
is_pow2(2)               = 1
is_pow2(3)               = 0
is_pow2(4)               = 1
is_pow2(6)               = 0
is_pow2(8)               = 1
is_pow2(9223372036854775808) = 1
is_pow2(12345)           = 0
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: d3442c00  ubfx x0, x0, #4, #8
4: d65f03c0  ret
8: d1000401  sub x1, x0, #0x1
c: ea01001f  tst x0, x1
10: fa400804  ccmp x0, #0x0, #0x4, eq
14: 1a9f07e0  cset w0, ne
18: d65f03c0  ret
```

The second routine uses the classic identity $x \ \& \ (x-1) == 0$ for powers of two, plus a conditional-compare to rule out zero. We will take it apart properly in chapter 5 once flags are on the table.

3.5 Aliases: why the disassembly looks different

Many familiar names are *aliases*: the assembler accepts a friendly spelling and emits another instruction. The disassembler reverses this when it can. You have already met two:

You write	The assembler emits
mov x0, x1	orr x0, xzr, x1
cmp x0, x1	subs xzr, x0, x1 (result discarded, flags kept)
tst x0, x1	ands xzr, x0, x1
neg x0, x1	sub x0, xzr, x1
lsl x0, x1, #3	ubfm x0, x1, #61, #60
cset w0, eq	csinc w0, wzr, wzr, ne
ret	ret x30

This is why the zero register exists: with a register that always reads as zero, “move” needs no instruction of its own. When you read compiler output, expect to see the friendly names; when you read a disassembly of a binary, expect the same.

Chapter 3 in five lines

- An instruction is a 32-bit word: opcode bits, 5-bit register numbers, and small immediates.
- The `sf` bit selects 64-bit (`x`) or 32-bit (`w`); `imm12` limits add immediates to 0–4095.
- There is no remainder or divide-by-zero trap: use `udiv` + `msub`.
- The second source can be shifted for free: `base + index` $\ll$ `n` in one instruction.
- `mov`, `cmp`, `tst`, `neg`, `cset`, `ret` are aliases of more general instructions.

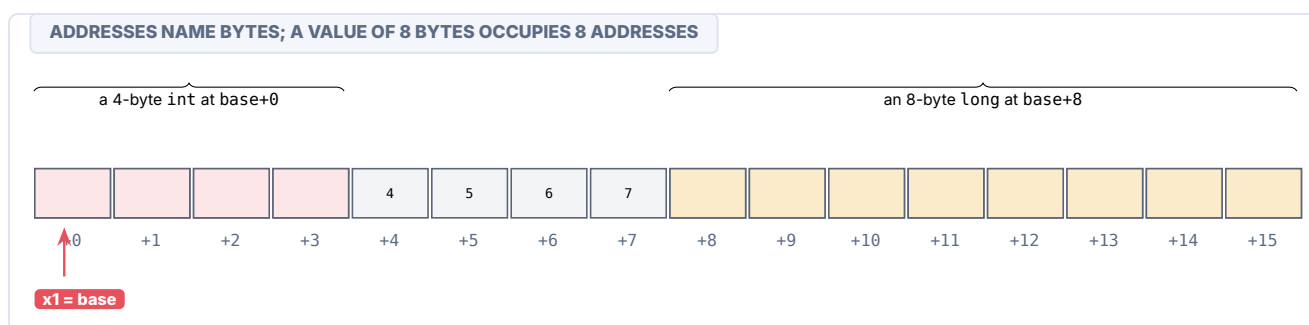
Memory and Addressing

Loads, stores, and the handful of addressing modes that make array walks, struct fields and stack frames cheap. Pointers stop being mysterious here: a pointer is a number in a register.

04

4.1 Memory is a long row of numbered bytes

Memory is one enormous array of bytes, each with an **address**. A pointer in C or C++ is nothing more than a 64-bit integer holding one of those addresses, and so it lives in an ordinary x register. Reading or writing memory means giving the CPU an address and a width.



4.2 Load and store

Two families of instructions cross between registers and memory.

- `ldr` (*load register*): read memory into a register.
- `str` (*store register*): write a register to memory.

The register letter picks the width: `ldr x0, [x1]` loads 8 bytes; `ldr w0, [x1]` loads 4 (and zeroes the top half of x0); `str w0, [x1]` stores the low 4 bytes. For narrower data use the suffixed forms, which *extend* the loaded value:

Instruction	Bytes	What lands in the register
<code>ldrb w0, [x1]</code>	1	the byte, zero -extended (unsigned char)
<code>ldrsw w0, [x1]</code>	1	the byte, sign -extended to 32 bits (signed char)
<code>ldrh w0, [x1]</code>	2	halfword, zero-extended (unsigned short)
<code>ldrsh x0, [x1]</code>	2	halfword, sign-extended to 64 bits
<code>ldr w0, [x1]</code>	4	word (int, float bits)
<code>ldrsw x0, [x1]</code>	4	word, sign-extended to 64 (int → long)
<code>ldr x0, [x1]</code>	8	doubleword (long, pointers)
<code>strb / strh / str w / str x</code>	1 / 2 / 4 / 8	store the low 1, 2, 4 or 8 bytes

Here is the smallest useful program that uses both: swap two integers through pointers, the machine-level version of `std::swap` on int.

swap32: two loads, two stores

✓ assembled & simulated

```
// void swap32(int32_t* a, int32_t* b)
    .globl _swap32
    .p2align 2
_swap32:
    ldr    w2, [x0]           // w2 = *a    (w = the low 32 bits of a register)
    ldr    w3, [x1]           // w3 = *b
    str    w3, [x0]           // *a = old *b
    str    w2, [x1]           // *b = old *a
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

swap32(&7, &-9) -> = (-9, 7)

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: b9400002 ldr w2, [x0]
4: b9400023 ldr w3, [x1]
8: b9000003 str w3, [x0]
c: b9000022 str w2, [x1]
10: d65f03c0 ret
```

Both pointers arrive in x0 and x1 (the calling convention again). We load *both* values into registers first, then store them crossed. If we stored before loading the second value, we would overwrite it; the C++ tmp variable is simply the register w2.

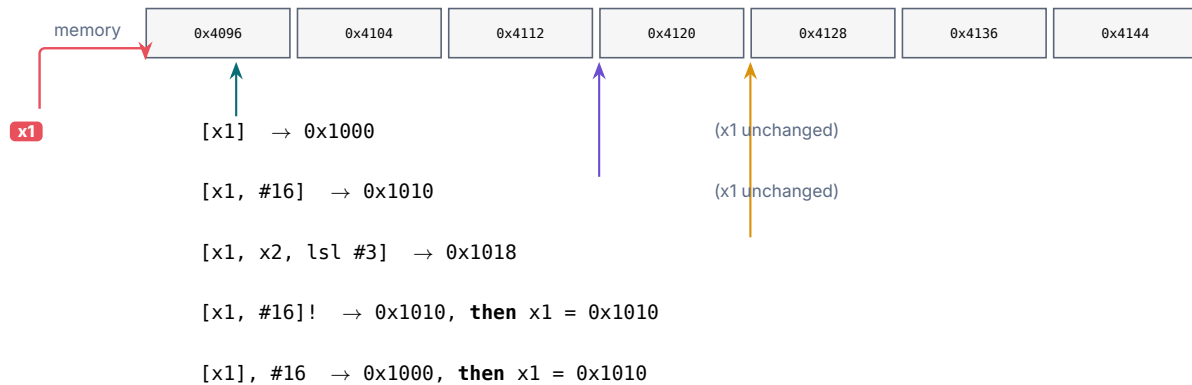
SWAP, STEP BY STEP (a = x0 POINTS TO 7, b = x1 POINTS TO -9)

start	*a	7	*b	-9	w2	--	w3	--
ldr w2, [x0]	*a	7	*b	-9	w2	7	w3	--
ldr w3, [x1]	*a	7	*b	-9	w2	7	w3	-9
str w3, [x0]	*a	-9	*b	-9	w2	7	w3	-9
str w2, [x1]	*a	-9	*b	7	w2	7	w3	-9

4.3 The addressing modes

The part inside the square brackets is the **effective address**: where to read or write. A64 gives a small, fixed menu. Suppose x1 = 0x1000 and x2 = 3:

Form	Address used	x1 afterwards
ldr x0, [x1]	0x1000	unchanged
ldr x0, [x1, #16]	0x1010	unchanged: immediate offset
ldr x0, [x1, x2, lsl #3]	0x1000 + 3×8 = 0x1018	unchanged: scaled register offset
ldr x0, [x1, #16]!	0x1010	0x1010: pre-index (update, then use)
ldr x0, [x1], #16	0x1000	0x1010: post-index (use, then update)

FIVE WAYS TO FORM AN ADDRESS ($x1 = 0x1000$)

Pre- and post-index forms are the secret of compact loops (the array walk below) and of the stack (chapter 6): they do a memory access and an add in one instruction.

Pitfall • The scaled register offset can only shift by the *size of the access*: `lsl #3` for 8-byte loads, `lsl #2` for 4-byte, and so on. Ask for the wrong shift and the assembler tells you:

Assembler: a mismatched shift on an 8-byte load

✓ real clang 18 error

```
d.s:2:20: error: expected 'lsl' or 'sxtx' with optional shift of #0 or #3
ldr x0, [x1, x2, lsl #2]
                ^
```

(An 8-byte load can shift by 0 or 3 only. For a 4-byte index into an array of `int`, use a `w` destination with `lsl #2`.)

Immediate offsets • `ldr x0, [x1, #imm]` takes a 12-bit *unsigned* offset that is *scaled by the access size*: for 8-byte loads, a multiple of 8 from 0 to 32760. For a negative or oddly aligned offset use the *unscaled* form `ldur` (range -256 to 255); clang writes `ldr` and the assembler picks `ldur` for you when needed.

4.4 Walking an array

Summing an array of 64-bit integers shows the loop idiom you will see everywhere: pointer in one register, count in another, post-index load, subs to count down and set flags, a conditional branch back.

sum: an array loop with a post-index load

✓ assembled & simulated

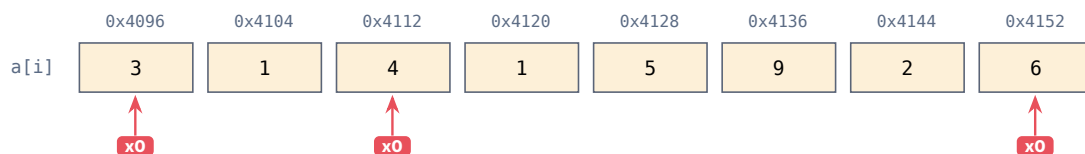
```
// int64_t sum(const int64_t* a, int64_t n)  -- x0 = pointer, x1 = count
.globl _sum
.p2align 2
_sum:
    mov     x2, #0                // total = 0
    cbz     x1, done              // n == 0 -> nothing to add
loop:
    ldr     x3, [x0], #8          // x3 = *a; a += 8 (post-index: use, THEN advance)
    add     x2, x2, x3            // total += x3
    subs    x1, x1, #1            // n--, and set the flags
    b.ne    loop                 // repeat while n != 0
done:
    mov     x0, x2                // return total
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
sum([3,1,4,1,5,9,2,6], 8)      = 31
sum(a, 0)                      = 0
sum(a, 1)                      = 3
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: d2800002  mov x2, #0x0 ; =0
4: b40000a1  cbz x1, 0x18 <done>
8: f8408403  ldr x3, [x0], #0x8
c: 8b030042  add x2, x2, x3
10: f1000421  subs x1, x1, #0x1
14: 54ffffa1  b.ne 0x8 <loop>
18: aa0203e0  mov x0, x2
1c: d65f03c0  ret
```

THE LOOP RUNNING ON {3, 1, 4, 1, 5, 9, 2, 6} (x0 = 0x1000 AT ENTRY)

The post-index load reads at x0 *then* adds 8 to it, so x0 sweeps along the array one slot per pass.

subs x1, x1, #1 counts down and sets the Z flag; b.ne loops while the count is not zero.

Zero-length arrays are handled by the cbz at the top, which skips the loop entirely.

4.5 Strings are just bytes

A C string is a run of bytes ending in a zero byte. Counting it needs a byte load and a *compare-and-branch*.

my_strlen: a byte loop

✓ assembled & simulated

```
// size_t my_strlen(const char* s)
    .globl _my_strlen
    .p2align 2
_my_strlen:
    mov     x1, x0                // x1 = start, kept so we can subtract at the end
loop:
    ldrb    w2, [x0], #1          // load one byte (zero-extended into w2); advance
    cbnz    w2, loop             // keep going until the byte is 0
    sub     x0, x0, x1           // x0 is one PAST the terminator...
    sub     x0, x0, #1           // ...so subtract 1 to exclude the '\0'
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
my_strlen('')      = 0
my_strlen('a')     = 1
my_strlen('hello') = 5
my_strlen('Apple silicon') = 13
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: aa0003e1  mov x1, x0
4: 38401402  ldrb w2, [x0], #0x1
8: 35ffffe2  cbnz w2, 0x4 <loop>
c: cb010000  sub x0, x0, x1
10: d1000400  sub x0, x0, #0x1
14: d65f03c0  ret
```

We load a byte with `ldrb` into `w2` and advance `x0` in the same instruction. `cbnz w2, loop` (*compare and branch if non-zero*) tests the byte without touching the flags. When the zero byte is found, `x0` already points one *past* it, so the length is the distance from the start minus one.

4.6 Structs: fields are fixed offsets

A struct is a block of memory in which every field sits at an offset the compiler computed. Accessing `p->w` is just a load at `p + offset`. The compiler inserts padding so each field is aligned to its own size.

A struct field access compiled by clang ◇ real clang 18 output; not simulated

```
struct P { int id; double w; long n; };
double get(const struct P* p) { return p->w * (double)p->n; }
```

```
_get:
    ldp    d0, d1, [x0, #8]
    scvtf  d1, d1
    fmul   d0, d0, d1
    ret
```

struct P { int id; double w; long n; }: 24 BYTES

0	1	2	3	pad	pad	pad	pad	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	---	---	-----	-----	-----	-----	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

id @ 0

w @ 8

n @ 16

`ldp d0, d1, [x0, #8]` loads two adjacent 8-byte values at once: `w` (offset 8) into `d0` and the 64 bits at offset 16 into `d1`. The compiler noticed they are neighbours.

Clang's output also has a surprise: it used `ldp` (*load pair*) to fetch both `w` and `n` in one instruction, putting the integer `n` into a *floating-point* register, then `scvtf` to convert it. Pairs are common: `ldp` and `stp` move two registers per instruction, and `stp` is how functions save registers on the stack (chapter 6).

Pitfall • Alignment matters for performance and, in a few cases (atomics, some special loads), for correctness. Ordinary loads and stores to unaligned addresses are allowed on Apple silicon but may be slower. The stack pointer, however, must be 16-byte aligned whenever it is used as a base register.

Chapter 4 in five lines

- A pointer is a 64-bit integer in an x register; `ldr/st r` move data with width chosen by the register letter.
- `ldrb/ldrh/ldrsw` zero- or sign-extend narrow values.
- Modes: `[x]`, `[x, #i]`, `[x, x, lsl #n]`, pre-index `[x, #i]!`, post-index `[x], #i`.
- Array loops pair a post-index load with `subs + b, ne`; string loops use `cbnz` on the loaded byte.
- Struct fields are fixed offsets from the base pointer; `ldp/stp` move two registers at once.

Flags and Branches

How a program decides: compare instructions set four flag bits, branches and conditional selects read them, and a few clever instructions avoid branching altogether.

05

5.1 The four flags

Arithmetic can leave a little summary of its result in four 1-bit flags, collectively **NZCV**. They are not written by every instruction, only by those that ask: the ones ending in s (adds, subs, ands) and the compare family (cmp, cmn, tst).

THE NZCV FLAGS

N	Negative: the result's top bit is 1
Z	Zero: the result was exactly 0
C	Carry: unsigned overflow (for subtraction: NO borrow)
V	oVerflow: signed overflow

cmp x0, x1 is just subs xzr, x0, x1: it subtracts, discards the answer and keeps the flags. Everything the program wants to know about how x0 relates to x1 is encoded in those four bits. We can watch them. The listing below is a single compare; the strip under it comes from running it in my interpreter for several pairs (the flags start at zero here; on real hardware they are whatever the caller last left).

cmp and nothing else ✓ assembled & simulated

```
// Run a compare and look at the flags afterwards (the test harness reads N Z C V).
.globl _cmpdemo
.p2align 2
_cmpdemo:
    cmp     x0, x1          // computes x0 - x1, throws the result away, keeps NZCV
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

21 checks: for 7 operand pairs, the flags agree with Python on equal (Z), signed less-than (N != V) and unsigned lower (C == 0)

FLAGS AFTER cmp x0, x1 (my interpreter)

x0	x1	N	Z	C	V
5	5	0	1	1	0
5	7	1	0	0	0
7	5	0	0	1	0
-1	1	1	0	1	0
1	-1	0	0	0	0
-9223372036854775808	1	0	0	1	1
0	0	0	1	1	0

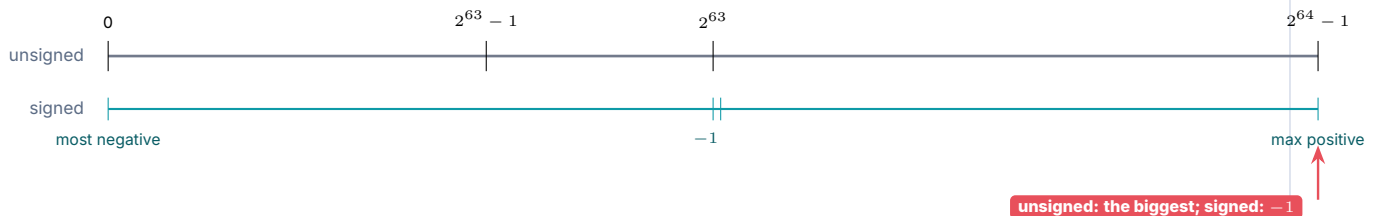
Look at the rows. Equal values give $Z=1$. $5 - 7$ is negative, so $N=1$. The row with $\text{INT64_MIN} - 1$ sets $V=1$: the true result does not fit in 64 signed bits, so the sign bit is wrong, and that is the signal that signed less-than" needs N and V together. For subtraction, **$C=1$ means no borrow was needed**, which is when the first operand was unsigned-greater-or-equal.

5.2 Condition codes

Instructions never test flags directly; they test a **condition**, a two-letter name that stands for a boolean formula over the flags. There are 16, and they come in a signed set and an unsigned set. Using the wrong set is a classic bug (the same bits, compared two different ways).

Cond	Flags tested	After <code>cmp a, b</code>	Use for
<code>eq</code>	$Z = 1$	$a = b$	any type
<code>ne</code>	$Z = 0$	$a \neq b$	any type
<code>hs</code> / <code>cs</code>	$C = 1$	$a \geq b$ unsigned	unsigned
<code>lo</code> / <code>cc</code>	$C = 0$	$a < b$ unsigned	unsigned
<code>hi</code>	$C = 1$ and $Z = 0$	$a > b$ unsigned	unsigned
<code>ls</code>	$C = 0$ or $Z = 1$	$a \leq b$ unsigned	unsigned
<code>ge</code>	$N = V$	$a \geq b$ signed	signed
<code>lt</code>	$N \neq V$	$a < b$ signed	signed
<code>gt</code>	$Z = 0$ and $N = V$	$a > b$ signed	signed
<code>le</code>	$Z = 1$ or $N \neq V$	$a \leq b$ signed	signed
<code>mi</code> / <code>pl</code>	$N = 1$ / $N = 0$	negative / non-negative	signs
<code>vs</code> / <code>vc</code>	$V = 1$ / $V = 0$	overflow / none	checked arithmetic
<code>al</code>	always	—	(unconditional)

THE SAME 64 BITS, TWO ORDERINGS: `0xFFFF...FF` IS BOTH -1 AND THE LARGEST UNSIGNED VALUE



The ring is cut at a different place. `hi/lo/hs/ls` compare along the top line, `gt/lt/ge/le` along the bottom one.

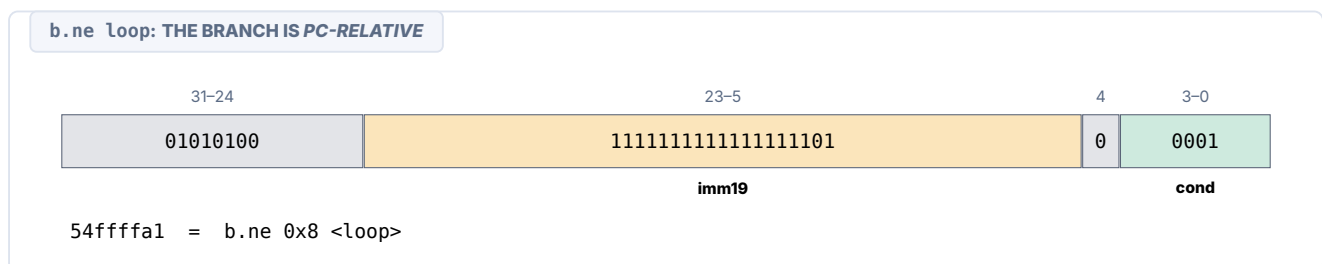
Pitfall • In C and C++, `if (a < b)` compiles to `lt` for signed types and `lo` for unsigned ones. If you hand-write assembly from a C function, copy the signedness of the C type, not what feels" right. The easiest way to be wrong is to use `lt` on an unsigned index.

5.3 Branches

A branch writes a new address into the PC. The ones you need:

Instruction	Meaning	Reach
<code>b label</code>	jump unconditionally	± 128 MiB
<code>b.cond label</code>	jump if the condition holds (<code>b.ne</code> , <code>b.lt</code> , ...)	± 1 MiB
<code>cbz / cbnz x0, label</code>	jump if <code>x0</code> is zero / non-zero; flags untouched	± 1 MiB
<code>tbz / tbnz x0, #n, label</code>	jump if bit <code>n</code> of <code>x0</code> is 0 / 1; flags untouched	± 32 KiB
<code>bl label</code>	call: <code>x30</code> = address of next instruction, then jump	± 128 MiB
<code>br x0 / blr x0</code>	jump / call to an address held in a register	anywhere
<code>ret</code>	jump to the address in <code>x30</code>	anywhere

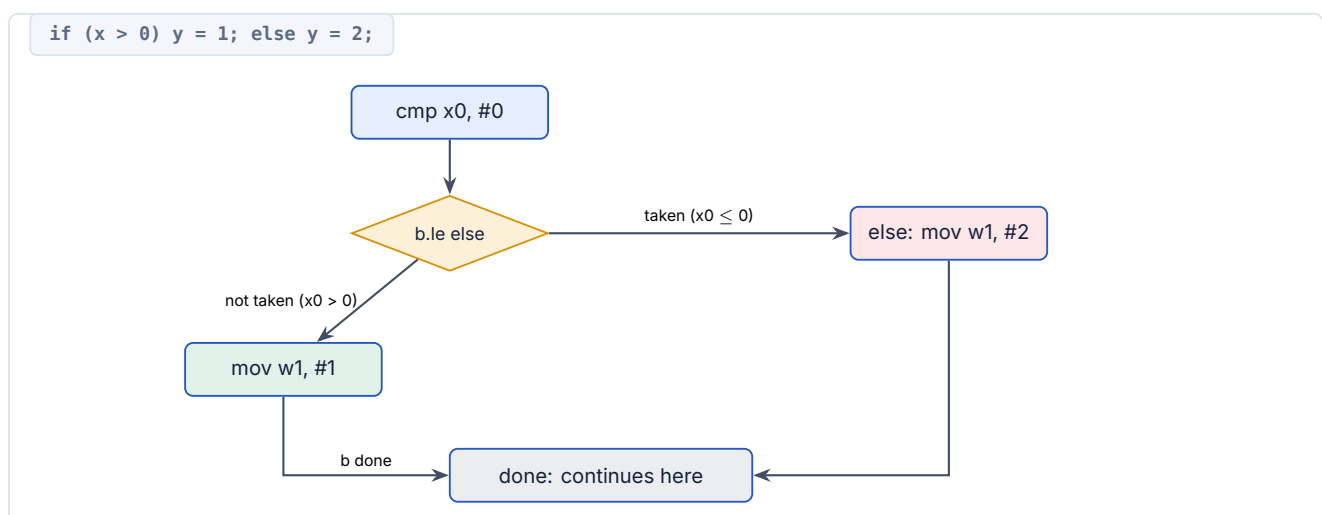
The reach figures come from the encoding: a branch holds a signed *count of instructions* (not bytes), and every instruction is 4 bytes. Here is the conditional branch `b.ne loop` from the array-sum loop, decoded:



The 19-bit **imm19** field is -3 (all ones except the last two bits). So the branch target is the branch's own address plus $-3 \times 4 = -12$ bytes, which lands on the `ldr` three instructions earlier. The **cond** field is `0001`, the code for `ne`. Because the target is stored as a distance rather than an address, the code works wherever the loader puts it in memory (this is called *position independence*).

5.4 If, else and loops

Every structured construct reduces to compare plus branch. Here is an `if/else` and the shape the compiler gives it:



A while or for loop becomes a label at the top, a body, and a conditional branch backwards. Compilers prefer to put the test at the *bottom* (so each pass costs one branch instead of two) and guard the first entry with a test at the top, exactly as in the sum listing of the last chapter: `cbz` up front, `b.ne loop` at the end. Euclid's loop in chapter 3 had the same shape.

5.5 Branch-free code: csel and friends

A mispredicted branch costs a pipeline flush, so when both outcomes are cheap the compiler uses a **conditional select**: compute both, then pick.

- `csel x0, x1, x2, cond: x0 = cond ? x1 : x2.`
- `csinc, csinv, csneg`: the else" value is incremented, inverted or negated.
- `cset w0, cond: w0 = cond ? 1 : 0`, which is how a C `bool` is made.
- `cneg x0, x1, cond: x0 = cond ? -x1 : x1.`

abs64 and max64 with no branches

✓ assembled & simulated

```
// int64_t abs64(int64_t x)      and      int64_t max64(int64_t a, int64_t b)
    .globl _abs64
    .p2align 2
_abs64:
    cmp     x0, #0                // sets the flags from x0 - 0
    cneg    x0, x0, lt            // if x0 < 0 (signed): x0 = -x0
    ret

    .globl _max64
    .p2align 2
_max64:
    cmp     x0, x1                // flags from a - b
    csel    x0, x0, x1, gt        // x0 = (a > b) ? a : b      (no branch!)
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
abs64(5)                = 5
abs64(-5)               = 5
abs64(0)                = 0
abs64(-9223372036854775807) = 9223372036854775807
max64(3, 9)             = 9
max64(9, 3)             = 9
max64(-4, -7)           = -4
max64(-7, -4)           = -4
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: f100001f  cmp x0, #0x0
4: da80a400  cneg x0, x0, lt
8: d65f03c0  ret
c: eb01001f  cmp x0, x1
10: 9a81c000  csel x0, x0, x1, gt
14: d65f03c0  ret
```

Both functions are straight-line code: a compare, then one conditional instruction. Nothing in them can be mispredicted.

5.6 A clever one: ccmp

`ccmp x0, #0, #4, eq` means: *if the previous condition was eq, compare x0 with 0 and set the flags from that; otherwise just load the flags with the literal 4 (binary 0100, which is Z=1).* It chains two conditions without a branch, which is how `a == b && c != d` is compiled. Our power-of-two test from chapter 3 uses it:

is_pow2: tst then ccmp

✓ assembled & simulated

```
// uint64_t field(uint64_t x) : bits [11:4] of x, i.e. (x >> 4) & 0xff
// bool is_pow2(uint64_t x)   : true for 1, 2, 4, 8, ...
    .globl _field
    .p2align 2
_field:
    ubfx    x0, x0, #4, #8        // unsigned bit-field extract: start bit 4, width 8
    ret
```

```

.globl _is_pow2
.p2align 2
_is_pow2:
    sub    x1, x0, #1           // x-1 flips the lowest set bit and all below
    tst    x0, x1              // x & (x-1) is zero exactly when x has one bit set...
    ccmp   x0, #0, #4, eq      // ...but 0 must not count: eq: compare x,0; else Z=1
    cset   w0, ne              // w0 = 1 if (x & (x-1)) == 0 and x != 0
    ret

```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```

field(0xabcd)           = 0xbc
field(0xffffffffffff)   = 0xff
field(0x10)             = 0x1
is_pow2(0)              = 0
is_pow2(1)              = 1
is_pow2(2)              = 1
is_pow2(3)              = 0
is_pow2(4)              = 1
is_pow2(6)              = 0
is_pow2(8)              = 1
is_pow2(9223372036854775808) = 1
is_pow2(12345)          = 0

```

FLAGS AFTER EACH INSTRUCTION (my interpreter; x = 8, 6, 0)

```

x = 8
sub    x1, x0, #1        N=0 Z=0 C=0 V=0
tst    x0, x1            N=0 Z=1 C=0 V=0
ccmp   x0, #0, #4, eq    N=0 Z=0 C=1 V=0
cset   w0, ne            N=0 Z=0 C=1 V=0
-> returns 1

x = 6
sub    x1, x0, #1        N=0 Z=0 C=0 V=0
tst    x0, x1            N=0 Z=0 C=0 V=0
ccmp   x0, #0, #4, eq    N=0 Z=1 C=0 V=0
cset   w0, ne            N=0 Z=1 C=0 V=0
-> returns 0

x = 0
sub    x1, x0, #1        N=0 Z=0 C=0 V=0
tst    x0, x1            N=0 Z=1 C=0 V=0
ccmp   x0, #0, #4, eq    N=0 Z=1 C=1 V=0
cset   w0, ne            N=0 Z=1 C=1 V=0
-> returns 0

```

For $x = 8$ the `tst` gives $Z=1$ ($8 \& 7 = 0$), so `eq` holds and `ccmp` actually compares 8 with 0: not equal, so Z becomes 0 and `cset ne` yields 1. For $x = 6$ the `tst` fails, `ccmp` *does not compare* and force-loads $Z=1$, so `cset ne` yields 0. For $x = 0$ the test passes ($0 \& -1 = 0$) but the compare finds $0 = 0$ and sets $Z=1$, so the result is 0: zero is correctly excluded. Notice that `sub` does not change the flags in the trace, while `tst` and `ccmp` do.

Chapter 5 in five lines

- N , Z , C , V are set only by `cmp/tst` and the `s`-suffixed instructions.
- Pick signed (`lt`, `le`, `gt`, `ge`) or unsigned (`lo`, `ls`, `hi`, `hs`) conditions to match the C type.
- Branch distances are signed instruction counts, so branches are position-independent.
- `cbz/tbz` test without touching the flags.
- `cset/cset/ccmp` replace small branches with straight-line code.

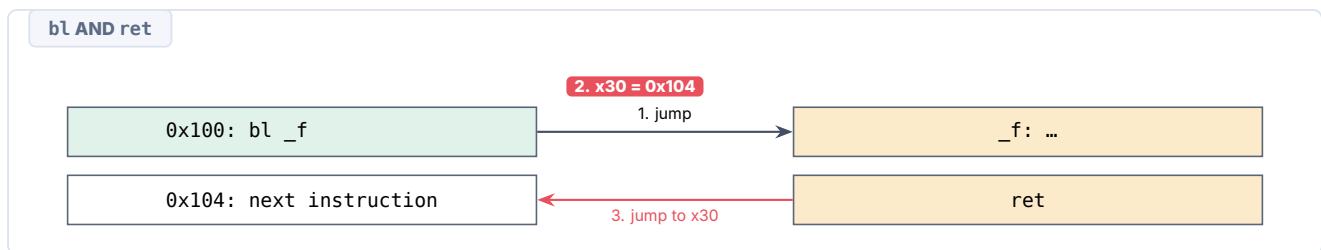
Functions and the Stack

How `bl` and `ret` work, the rules every function follows so that code from different compilers can call each other, and how the stack makes recursion possible.

06

6.1 Calling and returning

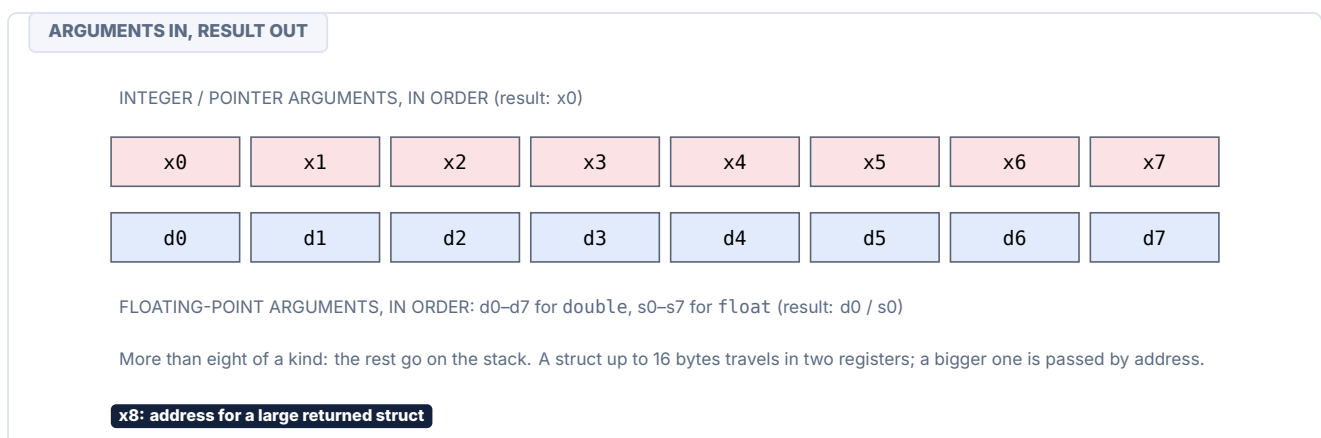
`bl label` (*branch with link*) does two things: it puts the address of the *next* instruction into `x30` (the link register), then jumps to the label. `ret` jumps to the address in `x30`. That is the entire call mechanism: no stack is involved, which is a significant difference from x86, where `call` pushes the return address in memory.



A function that calls *no one else* is a **leaf**. It never overwrites `x30`, so it needs no stack at all: `add3`, `gcd` and `square` are leaves. A function that makes a `bl` *does* overwrite `x30`, so it must save the old value first, or it will lose the way home.

6.2 The calling convention

For separately compiled code to work together, everyone agrees on where arguments and results live and which registers a function may disturb. Apple's rules (an adaptation of Arm's standard AAPCS64) are:



Example: `long f(long a, double b, int *p)` receives `a` in `x0`, `b` in `d0` and `p` in `x1`: integer and floating-point arguments are numbered *independently*. A double result goes back in `d0`.

The other half of the contract is who must preserve what:

CALLER-SAVED vs CALLEE-AVED

Caller-saved (scratch")

x0–x17, v0–v7, v16–v31

A call may destroy them.

If you need a value after b1, save it yourself.

x18: reserved, never touch

Callee-saved

x19–x28, x29, x30(!), low 64 bits of v8–v15

A function that uses them must restore

the caller's values before it returns.

sp: restored exactly

Apple-specific • Apple's arm64 ABI differs from the generic Arm one in a few places you may trip over. (1) **Variadic arguments go on the stack**, even the first few: calling `printf` from assembly means storing the extra arguments at `[sp]`, not in `x1...`. (2) `x18` is reserved. (3) The frame pointer `x29` must always point to a valid frame record (this makes backtraces reliable). (4) Rules for narrow arguments (a `char` passed in a register) differ from Linux; the safe approach is to follow Apple's "Writing ARM64 code for Apple platforms" document. (*Details of narrow-argument extension are from memory; confirm in the document.*)

6.3 The stack

The stack is a region of memory used for things that must survive across calls: saved registers, local arrays, spilled variables. It **grows downward**: pushing means *subtracting* from `sp`. The rules:

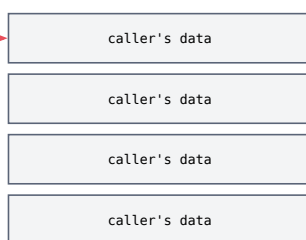
- `sp` must be a multiple of 16 whenever it is used to access memory, so frames come in multiples of 16 bytes.
- Whatever you subtract from `sp` on entry you must add back before `ret`.
- Memory below `sp` is not yours: signal handlers and interrupts may overwrite it.

A64 has no push and pop. Instead the *pre-index* store pair does the push, and the *post-index* load pair does the pop, both in one instruction:

```
stp x29, x30, [sp, #-32]!: PUSH FP AND LR, RESERVE 32 BYTES"
```

BEFORE

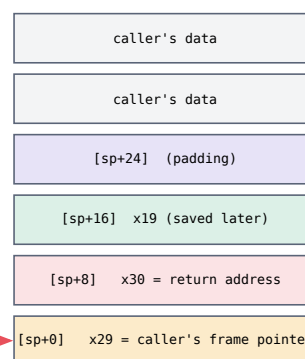
sp →



32 bytes

AFTER

sp →



`sp` moved DOWN by 32 and the pair was written, in ONE instruction.

6.4 A function that calls another

Now we can write a non-leaf function properly. `sumsq` calls `square` twice, and it needs a value ($a*a$) to survive the second call, so it keeps it in a callee-saved register, which in turn must be saved on entry and restored on exit.

square (leaf) and sumsq (calls it twice)

✓ assembled & simulated

```
// int64_t square(int64_t x) { return x * x; }          a LEAF: calls nothing
// int64_t sumsq(int64_t a, int64_t b) { return square(a) + square(b); } NOT a leaf
.globl _square
.p2align 2
_square:
    mul     x0, x0, x0                                // no bl inside: x30 is never overwritten, no frame
    ret

.globl _sumsq
.p2align 2
_sumsq:
    stp     x29, x30, [sp, #-32]!                    // prologue: make room, save fp and lr
    str     x19, [sp, #16]                            // callee-saved x19 will keep a value across a call
    mov     x29, sp                                    // fp = start of this frame
    mov     x19, x1                                    // stash b (x1 is caller-saved: a call may clobber it)
    bl      _square                                    // x0 = a * a (this bl overwrites x30)
    mov     x1, x0                                    // x1 = a*a
    mov     x0, x19                                    // x0 = b
    mov     x19, x1                                    // x19 = a*a (survives the next call)
    bl      _square                                    // x0 = b * b
    add     x0, x0, x19                                // a*a + b*b
    ldr     x19, [sp, #16]                            // epilogue: restore what we saved...
    ldp     x29, x30, [sp], #32                       // ...pop the frame, restoring lr
    ret                                              // return to OUR caller (x30 was restored)
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

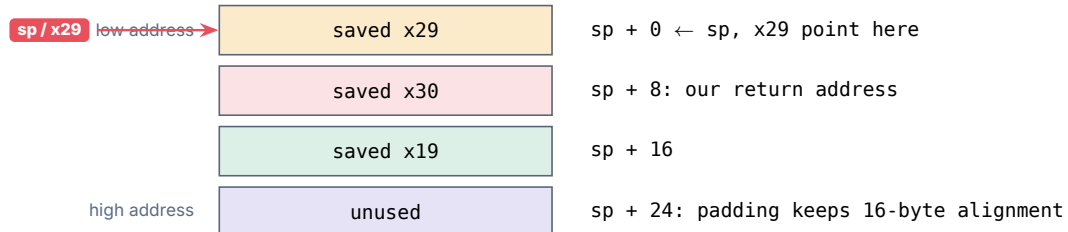
```
sumsq(3, 4) = 25
x19, x29, sp preserved = ('0xdead', '0x1234', True)
sumsq(0, 5) = 25
x19, x29, sp preserved = ('0xdead', '0x1234', True)
sumsq(1000000, 2) = 100000000000004
x19, x29, sp preserved = ('0xdead', '0x1234', True)
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: 9b007c00 mul x0, x0, x0
4: d65f03c0 ret
8: a9be7bfd stp x29, x30, [sp, #-0x20]!
c: f9000bf3 str x19, [sp, #0x10]
10: 910003fd mov x29, sp
14: aa0103f3 mov x19, x1
18: 97fffffa bl 0x0 <ltmp0>
1c: aa0003e1 mov x1, x0
20: aa1303e0 mov x0, x19
24: aa0103f3 mov x19, x1
28: 97fffff6 bl 0x0 <ltmp0>
2c: 8b130000 add x0, x0, x19
30: f9400bf3 ldr x19, [sp, #0x10]
34: a8c27bfd ldp x29, x30, [sp], #0x20
38: d65f03c0 ret
```

Read `sumsq` in three parts. The **prologue** (first three lines) builds a frame. The **body** calls twice. The **epilogue** (last three lines) reverses the prologue *in exactly the opposite order* and returns. The last row of the simulator strip checks the contract itself: after the call, `x19`, `x29` and `sp` have their original values, so a caller that was keeping something in `x19` would never know we used it.

ONE FRAME OF sumsq, AFTER ITS PROLOGUE



Pitfall • If `sumsq` forgot to save `x30`, the first `bl _square` would overwrite it, and the final `ret` would jump back into the middle of `sumsq` right after that call, looping forever. Forgetting to restore `x19` is subtler: your function seems fine, and your *caller* breaks later.

6.5 Recursion: one frame per call

The stack's real power is that every call gets its *own* frame, so a function can call itself. Here is factorial, written the obvious recursive way.

fact: recursion needs a frame per call

✓ assembled & simulated

```
// uint64_t fact(uint64_t n)    recursive:  n <= 1 ? 1 : n * fact(n - 1)
    .globl _fact
    .p2align 2
_fact:
    cmp     x0, #1
    b.hi    recurse           // unsigned higher: n > 1
    mov     x0, #1            // base case
    ret                               // (a "leaf path": no frame was needed)
recurse:
    stp     x29, x30, [sp, #-32]!   // push frame pointer and return address; sp -= 32
    str     x19, [sp, #16]         // save callee-saved x19 (we will use it)
    mov     x29, sp              // frame pointer = this frame
    mov     x19, x0              // keep n across the call (x0 will be clobbered)
    sub     x0, x0, #1
    bl      _fact               // x0 = fact(n - 1)    (bl sets x30 = return address)
    mul     x0, x19, x0          // n * fact(n - 1)
    ldr     x19, [sp, #16]       // restore x19
    ldp     x29, x30, [sp], #32  // pop frame; sp += 32
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
fact(0)           = 1
fact(1)           = 1
fact(5)           = 120
fact(10)          = 3628800
fact(20)          = 2432902008176640000
sp restored after fact(20) = 0x16f010000
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```

0: f100041f  cmp x0, #0x1
4: 54000068  b.hi 0x10 <recurse>
8: d2800020  mov x0, #0x1 ; =1
c: d65f03c0  ret
10: a9be7bfd  stp x29, x30, [sp, #-0x20]!
14: f9000bf3  str x19, [sp, #0x10]
18: 910003fd  mov x29, sp
1c: aa0003f3  mov x19, x0
20: d1000400  sub x0, x0, #0x1
24: 97ffffff  bl 0x0 <ltmp0>
28: 9b007e60  mul x0, x19, x0
2c: f9400bf3  ldr x19, [sp, #0x10]
30: a8c27bfd  ldp x29, x30, [sp], #0x20
34: d65f03c0  ret

```

Notice the *leaf path*: if $n \leq 1$ the function returns immediately, before building any frame (the early ret). Only the recursive path pays for a frame. Inside it, n must survive the recursive call, so it is parked in callee-saved $x19$ (which we saved first).

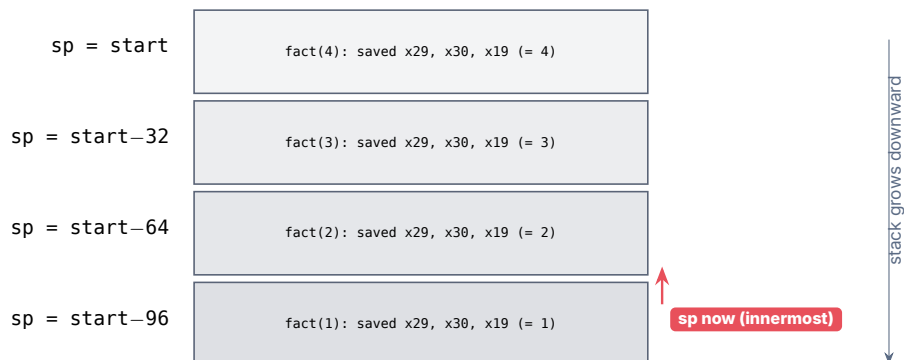
STACK POINTER AT THE START OF EACH CALL, fact(4) (my interpreter)

```

call  n  sp (relative to start)
1      4  +0
2      3  -32
3      2  -64
4      1  -96

```

THE STACK DURING fact(1), THE DEEPEST CALL (FRAME SIZE 32 BYTES EACH)



Each frame holds the previous `x29` too, so the frames form a linked list that a debugger can walk.

6.6 Local variables on the stack

A function can also reserve *space* on the stack just by subtracting from `sp`. At `-00`, compilers do this for every local variable. This function keeps its three arguments in memory purely to show the pattern:

Reserving and releasing stack space

✓ assembled & simulated

```

// int64_t sum3(int64_t a, int64_t b, int64_t c): spills everything to the stack (like -00)
.globl _sum3
.p2align 2
_sum3:
    sub    sp, sp, #32           // reserve 32 bytes (sp stays 16-aligned)
    str    x0, [sp, #0]         // a lives at [sp+0]
    str    x1, [sp, #8]         // b lives at [sp+8]
    str    x2, [sp, #16]        // c lives at [sp+16]
    ldr    x3, [sp, #0]
    ldr    x4, [sp, #8]
    ldr    x5, [sp, #16]

```

```

add    x0, x3, x4
add    x0, x0, x5
add    sp, sp, #32           // release the space: sp must end exactly where it began
ret

```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```

sum3(1, 2, 3)           = 6
sum3(100, -50, 7)       = 57
sp restored              = True

```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```

0: d10083ff  sub sp, sp, #0x20
4: f90003e0  str x0, [sp]
8: f90007e1  str x1, [sp, #0x8]
c: f9000be2  str x2, [sp, #0x10]
10: f94003e3  ldr x3, [sp]
14: f94007e4  ldr x4, [sp, #0x8]
18: f9400be5  ldr x5, [sp, #0x10]
1c: 8b040060  add x0, x3, x4
20: 8b050000  add x0, x0, x5
24: 910083ff  add sp, sp, #0x20
28: d65f03c0  ret

```

Always release exactly what you reserved, and always in multiples of 16. The last check in the strip confirms `sp` ends where it began.

6.7 Calling a library function

Calling `libc` from assembly is the same: put the arguments in `x0...`, `bl _function`, read the result from `x0`. Symbol names on macOS carry a leading underscore, so C's `strlen` is `_strlen`. Remember that `bl` clobbers `x30`, so the calling function needs a frame, and that any caller-saved register you care about must be saved across the call. Chapter 7 shows a complete example with `printf`, including Apple's variadic rule.

Chapter 6 in five lines

- `bl` sets `x30` and jumps; `ret` jumps to `x30`. Leaf functions need no frame.
- Arguments: `x0–x7` (integers/pointers) and `d0–d7` (floats); result in `x0` or `d0`.
- Caller-saved: `x0–x17`. Callee-saved: `x19–x28`, `x29`, `x30`. `x18` is reserved.
- The stack grows down in 16-byte multiples; `stp ... [sp, #-n]!` pushes, `ldp ... [sp], #n` pops.
- Prologue and epilogue mirror each other; every recursive call gets its own frame.

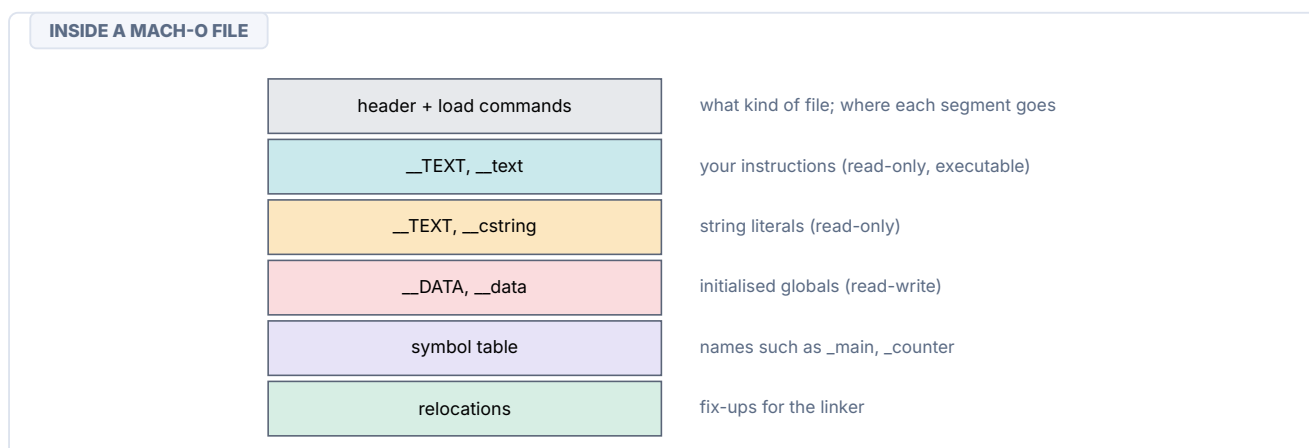
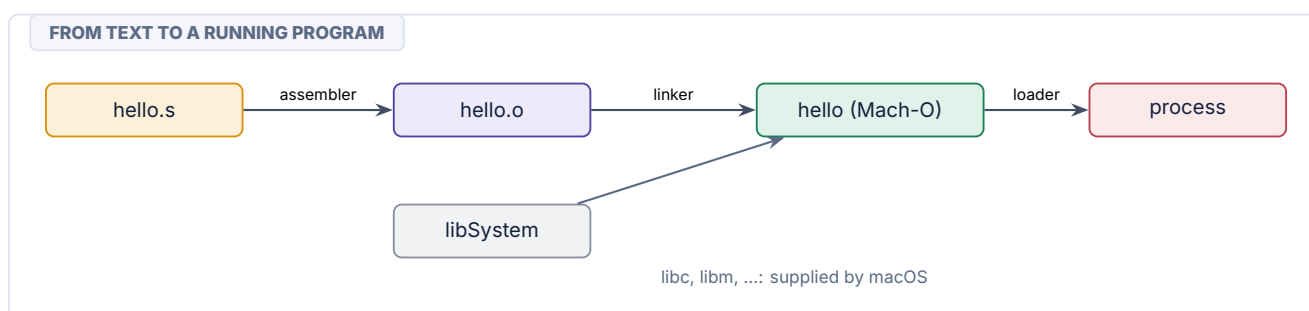
Programs, Data and the OS

What a Mach-O file contains, how code finds its data, how to print a string with a raw system call and with `libc`, and the Apple rule for variadic calls.

07

7.1 What the assembler gives you

Assembling a `.s` file produces an **object file** in Apple's **Mach-O** format. It holds the bytes of your instructions and data, grouped into **sections**, plus a **symbol table** (the names you made global) and **relocations** (a to-do list for the linker: patch this instruction once you know where that symbol ended up"). The linker combines objects into an executable, assigning final addresses.



Here is what the assembler's own tools report about the earlier `hello.s`. These are real command outputs (LLVM's `objdump` and `nm`, which print the same facts as Apple's `otool` and `nm`):

`objdump --macho --section-headers hello.o` ✓ real tool output

```

Sections:
Idx Name          Size      VMA           Type
 0  __text        00000024  0000000000000000 TEXT
 1  __cstring      0000000f  0000000000000024 DATA
  
```

nm: T = global code, D = global data, t/s = local ✓ real tool output

```
build/globals.o:
0000000000000000 T _bump
0000000000000018 D _counter
build/hello.o:
0000000000000000 T _main
0000000000000024 s msg
build/factrec.o:
0000000000000000 T _fact
0000000000000010 t recurse
```

Every global symbol has a leading underscore: the C function `main` is `_main` in assembly, always, on macOS.

7.2 Directives

Lines starting with a dot are instructions to the *assembler*, not to the CPU. You need only a few:

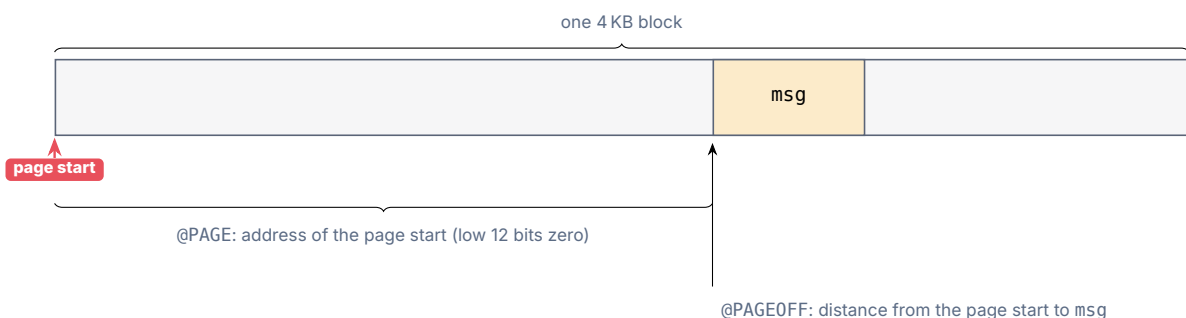
Directive	Meaning
<code>.globl _name</code>	Export the symbol so other files (and the linker) can see it.
<code>.p2align 2</code>	Align the next item to $2^2 = 4$ bytes (functions must be 4-byte aligned). <code>.p2align 3 = 8</code> bytes.
<code>.section __TEXT,__text</code>	Switch to the code section (the default).
<code>.section __TEXT,__cstring</code>	String literals.
<code>.section __DATA,__data</code>	Writable global data.
<code>.asciz "text"</code>	Emit the bytes of the string plus a terminating zero. (<code>.ascii</code> omits the zero.)
<code>.byte / .short / .word / .quad</code>	Emit 1-, 2-, 4- or 8-byte values.
<code>.space n</code>	Emit n zero bytes.

7.3 Finding data: `adrp`

Where is `msg`? The linker decides, so the assembler cannot put the address in the instruction. And an instruction is only 32 bits, too small to hold a 64-bit address. A64's answer is a two-step *page + offset* pair:

- `adrp x0, sym@PAGE`: put the address of the **4 KB-aligned block** that contains `sym` into `x0`. It is PC-relative with a reach of ± 4 GiB.
- `add x0, x0, sym@PAGEOFF`: add the symbol's offset within its page.

`adrp + add (OR THE OFFSET FOLDED INTO A LOAD)`



The second instruction needs only the *offset within the page*, a small number, so it fits: either as an `add` immediate or inside a load/store, as this global counter shows.

A global variable: adrp then a load with @PAGEOFF

✓ assembled & simulated

```
// int64_t counter; int64_t bump(void) { return ++counter; }      (data: __DATA, code: __TEXT)
.section __DATA,__data
.globl _counter
.p2align 3
_counter:
    .quad 0 // 8 bytes, initially 0

.section __TEXT,__text
.globl _bump
.p2align 2
_bump:
    adrp    x8, _counter@PAGE // x8 = address of the 4 KB page containing _counter
    ldr     x9, [x8, _counter@PAGEOFF] // x9 = counter (the offset is inside the ldr)
    add     x9, x9, #1
    str     x9, [x8, _counter@PAGEOFF] // counter = x9
    mov     x0, x9 // return the new value
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
bump() three times      = [1, 2, 3]
counter in memory      = 3
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: 90000008  adrp x8, 0x0 <tmp0>
4: f9400109  ldr x9, [x8]
8: 91000529  add x9, x9, #0x1
c: f9000109  str x9, [x8]
10: aa0903e0  mov x0, x9
14: d65f03c0  ret
```

In the machine-code strip the `adrp` says `0x0` and the `ldr` shows no offset at all. That is not a bug: an object file has not been linked yet, so those fields hold zeros and the three relocations tell the linker how to fill them:

objdump --macho -r globals.o: the linker's to-do list

✓ real tool output

```
Relocation information (__TEXT,__text) 3 entries
address pcrel length extern type scattered symbolnum/value
0000000c False long True PAGOF12 False _counter
00000004 False long True PAGOF12 False _counter
00000000 True long True PAGE21 False _counter
```

PAGE21 patches the `adrp` with the page; PAGOF12 patches each `ldr/str` with the 12-bit page offset. After linking, `otool -tV` on the executable would show real addresses.

Apple-specific • Do not confuse two pages". `adrp` always works in **4 KB** units; that is part of the instruction set. macOS on Apple silicon manages virtual memory in **16 KB** pages, which matters for alignment, `mmap` and cache-friendly layouts but not for how `adrp` computes. For symbols defined in *other* files or libraries, clang often routes through the **GOT**: `adrp x8, _sym@GOTPAGE` then `ldr x8, [x8, _sym@GOTPAGEOFF]` loads the symbol's address from a table the linker builds. I have not run a link here, so treat the GOT form as what you will see in compiler output".

7.4 Hello, world with a raw system call

The lowest-level way to print is to ask the kernel directly. On macOS an arm64 system call puts the **syscall number in x16**, arguments in `x0...`, and executes `svc #0x80` (supervisor call), a trap into the kernel. The numbers are 4 for `write(fd, buf, len)` and 1 for `exit(status)`.

Hello via svc: write then exit

✓ assembled & simulated

```
// Hello, world with a raw macOS system call (for learning; real programs call libc)
.globl _main
```

```

.p2align 2
_main:
    mov     x0, #1                // fd 1 = stdout
    adrp    x1, msg@PAGE          // x1 = address of the 4 KB page holding msg
    add     x1, x1, msg@PAGEOFF   // + the offset inside that page = &msg
    mov     x2, #14               // length
    mov     x16, #4               // syscall number: write
    svc     #0x80                 // trap into the kernel
    mov     x0, #0               // exit status
    mov     x16, #1               // syscall number: exit
    svc     #0x80

.section __TEXT,__cstring
msg: .asciz "Hello, ARM64!\n"

```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```

bytes written to fd 1      = 'Hello, ARM64!\n'
exit status                = 0
steps executed            = 9

```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```

0: d2800020  mov x0, #0x1 ; =1
4: 90000001  adrp x1, 0x0 <tmp0>
8: 91000021  add x1, x1, #0x0
c: d28001c2  mov x2, #0xe ; =14
10: d2800090  mov x16, #0x4 ; =4
14: d4001001  svc #0x80
18: d2800000  mov x0, #0x0 ; =0
1c: d2800030  mov x16, #0x1 ; =1
20: d4001001  svc #0x80

```

The strip shows my interpreter running" it: it implements just these two system calls and captured the bytes written. This proves the *logic* (right registers, right length, correct string address); it says nothing about the real kernel.

Pitfall • Raw syscalls are for learning. Apple does not promise that syscall numbers or this trap convention stay stable between macOS releases; the supported interface is `libSystem` (`write`, `puts`, ...). I have not run this listing on a real Mac. (Syscall numbers 4 and 1 are the classic BSD ones and have long been the same, but verify on your machine.)

7.5 Hello, world the normal way

Real programs call the C library. Same output, but now the OS interface is the library's problem and the code is robust.

Calling puts from assembly

✓ assembled

```

// Hello, world the normal way: call libc's puts, which does the write for us.
.globl _main
.p2align 2
_main:
    stp     x29, x30, [sp, #-16]! // we make a call: need a frame (bl clobbers x30)
    mov     x29, sp
    adrp    x0, msg@PAGE          // x0 = first argument: pointer to the string
    add     x0, x0, msg@PAGEOFF
    bl      _puts                 // puts(msg): prints it plus a newline
    mov     w0, #0                 // return 0 from main
    ldp     x29, x30, [sp], #16
    ret

.section __TEXT,__cstring
msg: .asciz "Hello from libc!"

```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```

0: a9bf7bfd stp x29, x30, [sp, #-0x10]!
4: 910003fd mov x29, sp
8: 90000000 adrp x0, 0x0 <ltmp0>
c: 91000000 add x0, x0, #0x0
10: 94000000 bl 0x10 <ltmp0+0x10>
14: 52800000 mov w0, #0x0 ; =0
18: a8c17bfd ldp x29, x30, [sp], #0x10
1c: d65f03c0 ret

```

main makes a call, so it builds a frame to protect x30. Returning 0 means setting w0 (an int). Link it with clang `hello.s -o hello`: clang adds the C library automatically.

7.6 Variadic functions: Apple's special rule

printf takes any number of arguments after the format. On Linux arm64 the extra arguments go in x1, x2...like any others. **On Apple platforms they go on the stack**, one 8-byte slot each, starting at [sp], while the format string (a fixed, named parameter) is still in x0.

printf with one variadic argument

✓ assembled

```

// printf("answer = %ld\n", 42);  Apple arm64: variadic arguments go ON THE STACK, not in x1.
.globl _main
.p2align 2
_main:
    stp    x29, x30, [sp, #-32]!    // frame; [sp] will hold the variadic argument
    mov    x29, sp
    mov    x8, #42
    str    x8, [sp]                // %ld argument goes at [sp], NOT in x1 (Apple)
    adrp   x0, fmt@PAGE            // x0 = the format string (a normal, named argument)
    add    x0, x0, fmt@PAGEOFF
    bl     _printf
    mov    w0, #0
    ldp    x29, x30, [sp], #32
    ret

.section __TEXT, __cstring
fmt: .asciz "answer = %ld\n"

```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```

0: a9be7bfd stp x29, x30, [sp, #-0x20]!
4: 910003fd mov x29, sp
8: d2800548 mov x8, #0x2a ; =42
c: f90003e8 str x8, [sp]
10: 90000000 adrp x0, 0x0 <ltmp0>
14: 91000000 add x0, x0, #0x0
18: 94000000 bl 0x18 <ltmp0+0x18>
1c: 52800000 mov w0, #0x0 ; =0
20: a8c27bfd ldp x29, x30, [sp], #0x20
24: d65f03c0 ret

```

CALLING printf("answer = %ld\n", 42) ON APPLE ARM64

x0 = fmt

the named parameter: in a register as usual

[sp+0] = 42

the first variadic argument: **on the stack**

[sp+8]

the second variadic argument would go here, and so on

Remember • This rule is why hand-written code that is correct on Linux can print garbage on a Mac. If you are ever porting Arm assembly between platforms, look at variadic calls first. As with the other `libc` listings, this one was assembled but not linked or run here.

Chapter 7 in five lines

- Assembling gives a Mach-O object: sections, symbols and relocations. Global names start with `_`.
- Directives (`.globl`, `.p2align`, `.asciz`, `.quad`, `.section`) steer the assembler.
- `adrp` + `@PAGEOFF` reaches any static data; `adrp` works in 4 KB units (macOS VM pages are 16 KB).
- A raw macOS syscall is `x16 = number` and `svc #0x80`; prefer `libc` in real programs.
- Variadic arguments go on the stack on Apple platforms.

Floating Point and NEON

The separate vector register bank: scalar float and double maths, then SIMD, where one instruction works on four floats or two doubles at once.

08

8.1 Scalar floating point

Floating-point values live in the v registers, seen through the narrow names s0–s31 (32-bit float) and d0–d31 (64-bit double) from chapter 2. A separate set of instructions operates on them; they all start with f.

Instruction	Meaning
fadd / fsub / fmul / fdiv d0, d1, d2	$d0 = d1 \pm \times \div d2$
fsqrt, fabs, fneg d0, d1	square root, absolute value, negate
fmadd d0, d1, d2, d3	$d0 = d1 \times d2 + d3$ with one rounding (fused)
fmov d0, d1 / fmov d0, #1.5	copy / load a small constant (limited set of values)
fmov d0, x1 / fmov x0, d1	move raw bits between integer and FP registers
scvtf d0, x1 / fcvtzs x0, d1	signed integer $\rightarrow$ double / double $\rightarrow$ integer (round toward zero)
fcmp d0, d1	compare; sets NZCV
ldr d0, [x1] / str d0, [x1]	load/store 8 bytes (s: 4, q: 16)

fcmp sets the same four flags as cmp, so the usual b.cond instructions work, with one twist: if either value is **NaN** the comparison is *unordered* and the flags become C=1, V=1 (N=Z=0). That is why float comparisons sometimes use different condition codes than integer ones (mi for less than, ls for less or equal), so that NaN makes the test fail.

A dot product of two arrays of doubles uses the fused multiply-add. This is the scalar baseline; notice it looks exactly like the integer array loop of chapter 4, with d registers.

dot: scalar double dot product with fmadd

✓ assembled & simulated

```
// double dot(const double* a, const double* b, int64_t n)    scalar floating point
    .globl _dot
    .p2align 2
_dot:
    fmov    d0, xzr                // d0 = 0.0 (accumulator; also the return reg)
    cbz     x2, done
loop:
    ldr     d1, [x0], #8           // d1 = *a++
    ldr     d2, [x1], #8           // d2 = *b++
    fmadd   d0, d1, d2, d0         // d0 = d1 * d2 + d0    (fused multiply-add)
    subs    x2, x2, #1
    b.ne    loop
done:
    ret
```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
dot([1.5,2,-3,4.25],[2,0.5,1,4], 4)    = 18.0
```

MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```

0: 9e6703e0  fmov d0, xzr
4: b40000c2  cbz x2, 0x1c <done>
8: fc408401  ldr d1, [x0], #0x8
c: fc408422  ldr d2, [x1], #0x8
10: 1f420020  fmadd d0, d1, d2, d0
14: f1000442  subs x2, x2, #0x1
18: 54ffff81  b.ne 0x8 <loop>
1c: d65f03c0  ret

```

`fmov d0, xzr` is the idiom for 0.0, because the all-zero bit pattern is $+0.0$. The return value of a function returning double goes in `d0`; the strip reads it back out of the interpreter's `v0` register.

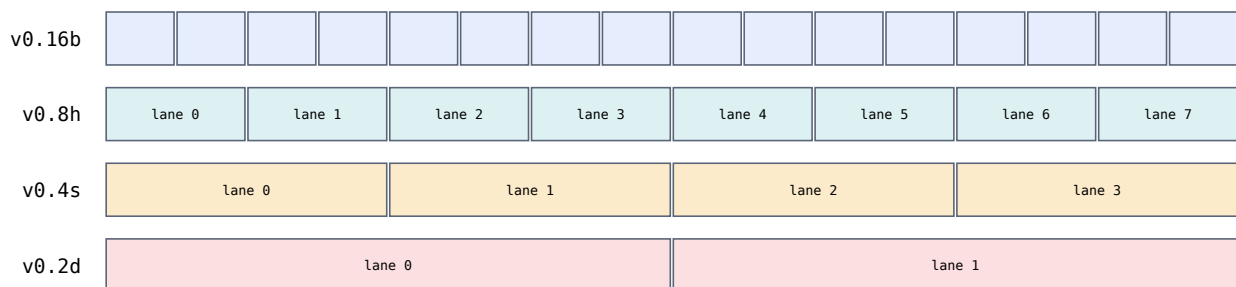
Remember • A fused multiply-add rounds once, not twice, so it is slightly *more accurate* than a separate multiply and add and can give a different last digit. Clang's default (`-ffp-contract=on`) fuses a multiply and add that appear in the *same expression*; `-ffp-contract=off` forbids it.

8.2 NEON: SIMD

Each vector register is 128 bits wide, and **NEON** instructions treat it as a row of equal-sized *lanes*. One instruction works on every lane at once: four float additions for the price of one.

You select the lane layout by an **arrangement specifier** after the register name:

ARRANGEMENT SPECIFIERS: HOW A 128-BIT REGISTER IS CUT UP



lane 0 is the *least* significant lane (on the right in a register diagram, at the lowest address in memory)

16 × 8-bit | 8 × 16-bit | 4 × 32-bit (float, int) | 2 × 64-bit (double, long)

(Half-width forms 8b, 4h, 2s use only the low 64 bits.) Our four-floats-at-a-time dot product:

dot4: NEON dot product, four floats per instruction

✓ assembled & simulated

```

// float dot4(const float* a, const float* b, int64_t n)  n % 4 == 0; 4 floats at a time
.globl _dot4
.p2align 2
_dot4:
    movi    v0.4s, #0                // four accumulators, all 0.0f
loop:
    ld1     {v1.4s}, [x0], #16        // load 4 floats from a, advance 16 bytes
    ld1     {v2.4s}, [x1], #16        // load 4 floats from b
    fmla    v0.4s, v1.4s, v2.4s        // v0[i] += v1[i] * v2[i] for all four lanes at once
    subs    x2, x2, #4
    b.ne    loop
    faddp    v0.4s, v0.4s, v0.4s        // pairwise add: (l0+l1, l2+l3, ...)
    faddp    s0, v0.2s                // add the two remaining halves -> scalar in s0
    ret

```

SIMULATOR CHECKS (my A64 interpreter vs a Python reference; not real hardware)

```
dot4([1..8],[0.5 x8], 8)           = 18.0
lanes of v0 after the loop         = [3.0, 4.0, 5.0, 6.0]
```

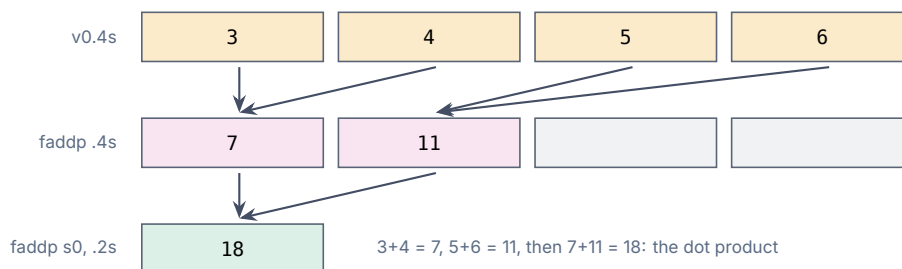
MACHINE CODE (offset, 32-bit encoding, disassembly: llvm-objdump on the object clang produced)

```
0: 4f000400  movi.4s v0, #0x0
4: 4cdf7801  ld1.4s { v1 }, [x0], #16
8: 4cdf7822  ld1.4s { v2 }, [x1], #16
c: 4e22cc20  fmla.4s v0, v1, v2
10: f1001042  subs x2, x2, #0x4
14: 54ffff81  b.ne 0x4 <loop>
18: 6e20d400  faddp.4s v0, v0, v0
1c: 7e30d800  faddp.2s s0, v0
20: d65f03c0  ret
```

Step through it. `movi v0.4s, #0` zeroes all four accumulator lanes. Each trip of the loop does two 16-byte loads (`ld1`, with a post-index like `ldr`) and one `fmla` (*fused multiply-accumulate*), then counts down by 4. After the loop we have four partial sums, one per lane, that must be added together: `faddp` is a *pairwise add* that adds neighbouring lanes, so two of them collapse four lanes to one.

LANES OF v0 (my interpreter, $a = 1 \dots 8, b = 0.5$)

```
a = 1 2 3 4 | 5 6 7 8      b = 0.5 everywhere
after loop: v0.4s = 3  4  5  6
after faddp x2: s0 = 18
```

THE HORIZONTAL REDUCTION: faddp TWICE

Pitfall • Floating-point addition is not associative. The vector version adds in a different *order* from a scalar loop (lane by lane, then across), so results can differ in the last bits. Compilers will not make that transformation on `float` reductions unless you allow it (`-ffast-math` or `-fassociative-math`). When hand-written code is faster, this is often the reason.

8.3 Two syntaxes for the same thing

You have seen `movi v0.4s, #0` in the listing above. But `clang -S` prints vector instructions like this:

```
movi.2d v0, #0000000000000000    fmla.4s v0, v2, v1    faddp.4s v0, v0, v0
```

The arrangement is attached to the *mnemonic* instead of to each register. This is Apple's older assembler dialect, and the assembler accepts both spellings. (I checked: both forms assemble to the same machine code.) Arm's documentation uses the `v0.4s` form, which this book prefers by hand.

8.4 What the compiler does by itself

You rarely need to write NEON by hand: the compiler **auto-vectorizes** simple loops. Here is the plain C sum from earlier, compiled with `-O2`:

An integer sum loop: what clang makes of it

◇ real clang 18 output; not simulated

```
long sum(const long* a, long n) {
    long t = 0;
    for (long i = 0; i < n; ++i) t += a[i];
    return t;
}
```

```
_sum:
    cmp     x1, #1
    b.lt    LBB0_3
    cmp     x1, #8
    b.hs    LBB0_4
    mov     x9, #0
    mov     x8, #0
    b       LBB0_7

LBB0_3:
    mov     x8, #0
    mov     x0, x8
    ret

LBB0_4:
    and     x9, x1, #0x7fffffffffffffff8
    add     x8, x0, #32
    movi.2d v0, #0000000000000000
    mov     x10, x9
    movi.2d v1, #0000000000000000
    movi.2d v2, #0000000000000000
    movi.2d v3, #0000000000000000

LBB0_5:
    ldp     q4, q5, [x8, #-32]
    ldp     q6, q7, [x8], #64
    add.2d  v0, v4, v0
    add.2d  v1, v5, v1
    add.2d  v2, v6, v2
    add.2d  v3, v7, v3
    subs    x10, x10, #8
    b.ne    LBB0_5
    add.2d  v0, v1, v0
    add.2d  v0, v2, v0
    add.2d  v0, v3, v0
    addp.2d d0, v0
    fmov    x8, d0
    cmp     x9, x1
    b.eq    LBB0_9

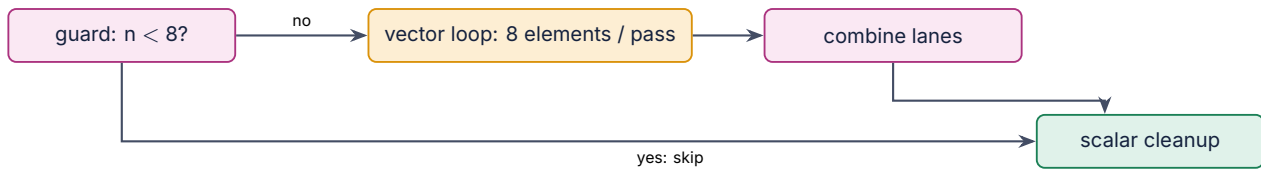
LBB0_7:
    sub     x10, x1, x9
    add     x9, x0, x9, lsl #3

LBB0_8:
    ldr     x11, [x9], #8
    add     x8, x11, x8
    subs    x10, x10, #1
    b.ne    LBB0_8

LBB0_9:
    mov     x0, x8
    ret
```

Walk through the output. First two guards: if $n < 1$ return 0, and if $n < 8$ skip straight to the scalar loop `LBB0_7`. Otherwise `LBB0_4` sets up *four* vector accumulators (`v0-v3`). The hot loop `LBB0_5` loads *eight* 64-bit values per pass (`ldp q4, q5` and `ldp q6, q7`: two pair loads of two 128-bit registers each) and adds with `add.2d`, two lanes at a time, in 4 independent chains. After the loop the accumulators are combined (`add.2d, addp.2d`) and moved to `x8` (`fmov x8, d0`), then a scalar *cleanup loop* finishes the remaining $n \bmod 8$ elements.

THE SHAPE OF A VECTORISED LOOP



8.5 Intrinsics: NEON from C and C++

When you want SIMD without writing assembly, use **intrinsics**: functions from `<arm_neon.h>` that compile to exactly one instruction each. Our dot product again:

Intrinsics: `vfmaq_f32` → `fmla`

◇ real clang 18 output; not simulated

```
#include <arm_neon.h>
float dot4(const float* a, const float* b, long n) {
    float32x4_t acc = vdupq_n_f32(0.0f);
    for (long i = 0; i < n; i += 4)
        acc = vfmaq_f32(acc, vld1q_f32(a + i), vld1q_f32(b + i));
    return vaddvq_f32(acc);
}
```

```
_dot4:
    cmp     x2, #1
    b.lt    LBB0_3
    mov     x8, #0
    movi.2d v0, #0000000000000000
LBB0_2:
    ldr     q1, [x0], #16
    ldr     q2, [x1], #16
    fmla.4s v0, v2, v1
    add     x8, x8, #4
    cmp     x8, x2
    b.lt    LBB0_2
    b       LBB0_4
LBB0_3:
    movi.2d v0, #0000000000000000
LBB0_4:
    faddp.4s v0, v0, v0
    faddp.2s s0, v0
    ret
```

The compiler produced essentially the hand-written listing: `ldr q, fmla.4s, faddp` twice. Intrinsics keep the compiler in charge of registers and scheduling and need no assembler file. We return to when to choose assembly, intrinsics or plain C++ in the “Where ARM64 helps” sections of the C++ and Graphics guides.

Apple-specific • Apple’s M-series cores execute NEON at 128 bits wide (four floats). As far as I know they do not implement Arm’s wider SVE vectors; newer generations add a separate matrix-oriented mode (SME). I am not certain which generations have which, so check Apple’s documentation or `sysctl hw.optional` on your Mac before relying on any extension beyond NEON.

Chapter 8 in five lines

- s/d registers are narrow views of the 128-bit v registers; FP instructions begin with f.
- `fmadd/fmla` fuse a multiply and an add with one rounding.
- NEON lanes: `.16b .8h .4s .2d`; one instruction operates on all lanes.
- Reductions across lanes use `faddp/addp`; FP addition order changes results slightly.
- Clang vectorizes loops itself and offers `<arm_neon.h>` intrinsics that map to single instructions.

Reading Compiler Output

The skill that pays off most: look at what clang wrote, recognise the patterns, and know why it picked them. Division by a constant, calls, inlining, and three puzzles.

09

9.1 A routine for reading any function

Compiler output looks dense until you have a routine. Use this one every time.

SIX QUESTIONS FOR ANY FUNCTION

- 1 Which registers are the *arguments*? (x0, x1, ... in order; d0... for floats)
- 2 Is there a *frame*? (stp x29,x30 ...) If not, it is a leaf.
- 3 Mark every *label* and every branch to it: that is the control-flow skeleton.
- 4 Find the *flag-setting* instruction before each b.cond / csel: that is the condition.
- 5 Follow the *loads and stores*: base register, offset, width. They are the data structure.
- 6 Find what ends up in x0 / d0 at each ret: that is the return value.

To get assembly without noise, compile with `clang -S -O2 -fno-asynchronous-unwind-tables x.c -o x.s`, and delete lines that start with a dot (directives) while you read. (This book's listings were cleaned the same way and *nothing else* was changed.) To read a built binary instead, use `otool -tV prog` or LLDB's `disassemble`.

9.2 Division by a constant: no divide at all

Integer division is slow, so when the divisor is a compile-time constant, compilers multiply by a precomputed reciprocal and shift. Read this output before the explanation:

Three tiny functions

◇ real clang 18 output; not simulated

```
unsigned div10(unsigned x) { return x / 10; }
int mod8(int x) { return x % 8; }
long mul9(long x) { return x * 9; }
```

```
_div10:
    mov     w8, #52429
    movk    w8, #52428, lsl #16
    umull   x8, w0, w8
    lsr     x0, x8, #35
    ret

_mod8:
    negs    w8, w0
    and     w8, w8, #0x7
```

```

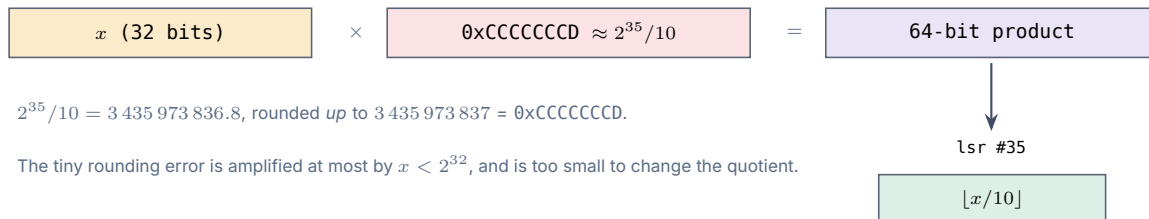
    and    w9, w0, #0x7
    csneg  w0, w9, w8, mi
    ret

_mul9:
    add    x0, x0, x0, lsl #3
    ret

```

`div10` computes $x/10$ with no `udiv`. The first two instructions build the constant `0xCCCCCCCD` (52429 is `0xCCCCD`, 52428 is `0xCCCC`), then `umull` multiplies it by x into a 64-bit product and `lsr #35` shifts right 35 places.

WHY $x / 10 == (x \times 0xCCCCCCD) \gg 35$



How do I know it is right for every input and not just the ones I thought of? I checked all of them: the verification harness evaluates the formula for all 2^{32} unsigned 32-bit values (vectorised on the CPU of the machine I used) and compares each with exact integer division. Zero mismatches. The same file also runs the compiler's three functions through the interpreter:

```

CHECKS ON THE ABOVE (my interpreter, plus the exhaustive 2^32 comparison)
div10(0) = 0
div10(9) = 0
div10(10) = 1
div10(99) = 9
div10(100) = 10
div10(4294967295) = 429496729
div10(123456789) = 12345678
mod8(0) = 0
mod8(7) = 7
mod8(8) = 0
mod8(13) = 5
mod8(-1) = -1
mod8(-7) = -7
mod8(-8) = 0
mod8(-13) = -5
mod8(2147483647) = 7
mod8(-2147483648) = 0
mul9(0) = 0
mul9(1) = 9
mul9(7) = 63
mul9(-3) = -27
mul9(1000000000) = 9000000000
all 2^32 inputs: (x*0xCCCCCCD)>>35 == x/10, mismatches = 0

```

The other two are worth a look. `mul9` is $x + (x \ll 3)$: one add with a shifted operand (chapter 3). `mod8` for a *signed* int is trickier because C says the result takes the sign of the dividend: `negs` computes $-x$ and sets flags, the two `ands` take the low three bits of x and of $-x$, and `csneg ... mi` picks one depending on whether x was negative (N flag). If it were unsigned, the answer would be a single `and w0, w0, #7`.

A habit worth forming • When a loop is slow because it divides or takes a remainder by something that is *not* a compile-time constant, that is the real `udiv` (about as expensive as several multiplies). Making the divisor a constant, or a power of two, lets the compiler do what you just read.

9.3 Common transformations to recognise

You see	It usually means
<code>lsl x0, x0, #k</code>	multiply by 2^k
<code>add x0, x0, x0, lsl #k</code>	multiply by $2^k + 1$ (e.g. $\times 9$)
<code>asr / lsr x0, x0, #k</code>	divide by 2^k (signed / unsigned)
<code>and w0, w0, #(2^k-1)</code>	remainder by 2^k (unsigned)
<code>mov xN, #imm; movk ...</code>	a large constant (often a hash multiplier or division magic)
<code>umull / smull + lsr / asr</code>	division by a constant
<code>cbz w0, label</code>	if ($x == 0$), no compare needed
<code>cset / cset / cinc</code>	a ternary or boolean without branching
<code>ldp / stp x29, x30</code>	a function prologue/epilogue
<code>ldr q / ldl, fmla, add.4s</code>	vectorised code
<code>bl _swift_retain, bl _objc_msgSend</code>	Swift / Objective-C runtime calls

9.4 Calls and frames in compiler output

Here is a plain C function that calls another function twice. It has to keep two values alive across calls, so it must use callee-saved registers, which means a frame:

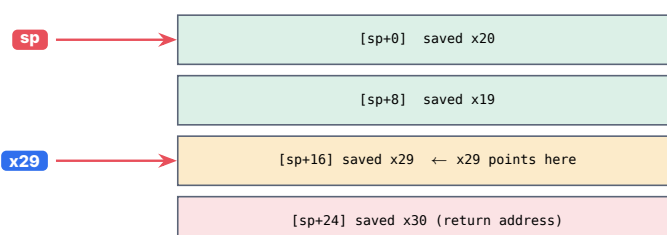
f(x) = twice(x) + twice(x+1) ◇ real clang 18 output; not simulated

```
long twice(long);
long f(long x) { return twice(x) + twice(x + 1); }
```

```
_f:
    stp     x20, x19, [sp, #-32]!
    stp     x29, x30, [sp, #16]
    add     x29, sp, #16
    mov     x19, x0
    bl      _twice
    mov     x20, x0
    add     x0, x19, #1
    bl      _twice
    add     x0, x0, x20
    ldp     x29, x30, [sp, #16]
    ldp     x20, x19, [sp], #32
    ret
```

Compare it with the hand-written `sumsq` of chapter 6; it is the same design. Clang saves `x19` and `x20` (callee-saved scratch for the values that survive calls) *and* the frame pair `x29, x30`, all with one pre-indexed `stp` followed by a second `stp`. Then `add x29, sp, #16` points the frame pointer at the saved pair, which is the link in the chain a debugger follows when it prints a backtrace.

f'S FRAME (32 BYTES)



Clang's `stp x20, x19` stores the pair at `[sp]` and `[sp+8]`; the second `stp x29, x30` fills `[sp+16]` and `[sp+24]`.

9.5 Inlining: calls that vanish

At -O2 small functions are usually *inlined*: copied into the caller, so the call disappears. Then other optimisations see through the boundary.

sq(a) + sq(a+1) with sq marked static

◇ real clang 18 output; not simulated

```
static long sq(long x) { return x * x; }
long f(long a) { return sq(a) + sq(a + 1); }

_f:
    mul     x8, x0, x0
    add     x9, x0, #1
    madd    x0, x9, x9, x8
    ret
```

There is no bl at all and no frame. The two squares became a mul and a madd: $(a + 1) \cdot (a + 1) + a \cdot a$ in one fused instruction. This is why a debugger showing an optimised build often cannot find the function you wrote: it no longer exists as a separate piece of code.

9.6 Puzzles

Below are three real clang outputs. Try to say what each function does *before* reading the answers. Use the six questions.

Puzzle 1

✓ real clang 18 output

```
_m1:
    cmp     w0, #0
    cset    w8, ne
    orr     w0, w8, w0, asr #31
    ret
```

Puzzle 2: three arguments

✓ real clang 18 output

```
_m2:
    cmp     x0, x2
    csel    x8, x0, x2, lt
    cmp     x0, x1
    csel    x0, x1, x8, lt
    ret
```

Puzzle 3: the argument is a pointer

✓ real clang 18 output

```
_m3:
    mov     x8, #0
LBB0_1:
    ldrb    w9, [x0, x8]
    orr     w9, w9, #0x20
    cmp     w9, #32
    b.eq    LBB0_3
    add     x8, x8, #1
    b       LBB0_1
LBB0_3:
    mov     x0, x8
    ret
```

Answers

1. The sign function, $(x > 0) - (x < 0)$, returning -1 , 0 or 1 . cset w8, ne gives 1 for any non-zero x. w0 asr #31 is -1 for negative x and 0 otherwise; ORing it in turns a 1 into -1 when x is negative. Two

instructions, no branch.

2. Clamp: $x < lo ? lo : (x > hi ? hi : x)$. First `cse1` computes x unless it exceeds `hi` (that is $\min(x, hi)$: it keeps x if $x < hi$, else `hi`), and the second `cse1` applies the lower bound: if $x < lo$ the answer is `lo`. Both are signed (`lt`), so the arguments were `long`.

3. Length up to the first zero byte or space: `ldrb` loads a byte; `orr w9, w9, #0x20` sets bit 5; the result equals `0x20` only if the original byte was `0x00` or `0x20` (a space), so `cmp #32 / b.eq` tests both in one comparison. Then `x8` counts. This is a nice example of a trick the compiler found that you would not write by hand.

I ran all three through the interpreter on several inputs to confirm these readings:

PUZZLE 3 CHECK (my interpreter vs the C semantics: first token length)

```
m3('hello world')    = 5
m3('abc')             = 3
m3('')               = 0
m3(' lead')          = 0
m3('a b c')          = 1
```

Chapter 9 in five lines

- Read in a fixed order: arguments, frame, labels, flag-setters, loads/stores, return value.
- Division by a constant becomes multiply-and-shift; I verified $x/10$ exhaustively for all 2^{32} inputs.
- Multiply by $2^k \pm 1$, remainder by 2^k , ternaries: shifts, and, and `cse1`.
- A frame (`stp x29, x30`) means the function calls something; no frame means a leaf or an inlined result.
- At -O2 small functions inline and their calls disappear.

Debugging and Tools

Stepping one instruction at a time, reading registers and memory, making sense of a crash, and the Apple tools that show you what the machine is doing.

10

Not run here • This chapter describes tools (LLDB on a Mac, Instruments, crash reports) that I could not run: the machine that produced this booklet has no Apple silicon and no macOS. Command names and options are from documentation and experience; output formats are described, not captured. The one thing that *is* real output is the step-by-step table below, which comes from my interpreter and shows what you would see if you stepped the same code. If an LLDB option does not behave as written, `help <command>` inside LLDB is authoritative.

10.1 What stepping looks like

Debugging at the instruction level means: stop the program, look at the registers, execute exactly one instruction, look again. Here is `gcd(48, 18)` from chapter 3, one instruction per row, with the three registers it changes. This table is from the interpreter, but it is exactly the sequence of states you would walk through with LLDB's `stepi` command.

REGISTERS AFTER EACH INSTRUCTION (my interpreter)

step	instruction	just executed	x0	x1	x2
1	cbz	x1, done	48	18	0
2	udiv	x2, x0, x1	48	18	2
3	msub	x2, x2, x1, x0	48	18	12
4	mov	x0, x1	18	18	12
5	mov	x1, x2	18	12	12
6	cbnz	x1, loop	18	12	12
7	udiv	x2, x0, x1	18	12	1
8	msub	x2, x2, x1, x0	18	12	6
9	mov	x0, x1	12	12	6
10	mov	x1, x2	12	6	6
11	cbnz	x1, loop	12	6	6
12	udiv	x2, x0, x1	12	6	2
13	msub	x2, x2, x1, x0	12	6	0
14	mov	x0, x1	6	6	0

Read down the x2 column: the quotient 2 appears after `udiv`, then the remainder 12 after `msub`. Rows 4 and 5 shift the pair (a, b) to (b, r). When the remainder reaches 0 the loop falls through and x0 holds the gcd.

THE DEBUGGER LOOP

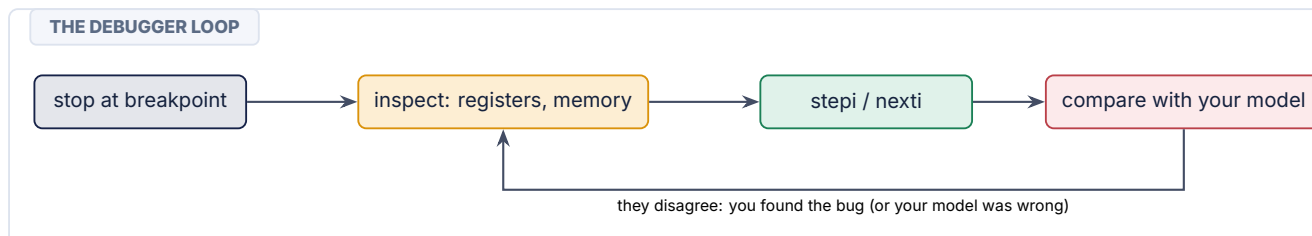
stop at breakpoint

inspect: registers, memory

stepi / nexti

compare with your model

they disagree: you found the bug (or your model was wrong)



10.2 LLDB: the commands you need

LLDB ships with the Xcode command-line tools. Build with debug info and no optimisation, then start it:

Build and start

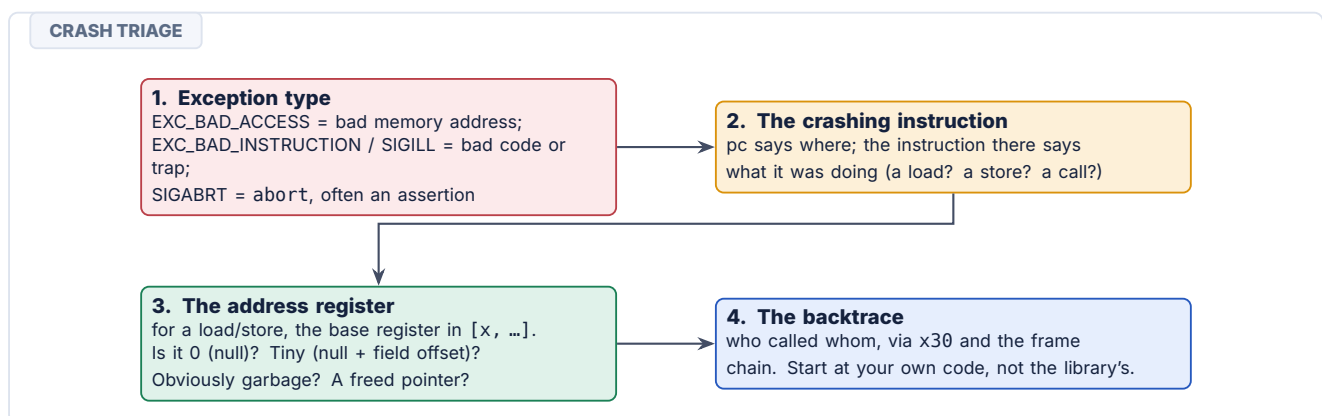
```
clang -g -O0 prog.c -o prog      # -g: debug info; -O0: variables stay where you wrote them
lldb ./prog                     # start the debugger (the program is not running yet)
```

Command (short form)	What it does
<code>breakpoint set -n gcd (b gcd)</code>	Stop whenever gcd is entered.
<code>run (r)</code>	Start the program.
<code>register read x0 x1 x2</code>	Print those registers. <code>register read</code> alone prints the general set; <code>--all</code> prints more.
<code>register write x0 5</code>	Change a register: experiment without editing code.
<code>disassemble -n gcd (di -n gcd)</code>	Show the instructions of a function. <code>disassemble -p</code> shows the ones around the current pc.
<code>thread step-inst (si)</code>	Execute one instruction , stepping <i>into</i> calls.
<code>thread step-inst-over (ni)</code>	One instruction, stepping <i>over</i> a bl.
<code>memory read -s8 -c4 -fx \$sp</code>	Print four 8-byte words at the stack pointer, in hex.
<code>x/4gx \$sp</code>	The same, in GDB's compact spelling (LLDB accepts it).
<code>bt</code>	Backtrace: the chain of calls that led here, found by following the frame pointers (x29).
<code>frame info / frame variable</code>	Where you are; and the named variables (needs -g).
<code>expression -- \$x0 + 1 (p \$x0)</code>	Evaluate an expression with registers as \$x0.
<code>continue (c)</code>	Run until the next breakpoint or the end.

Remember • `si` steps *into* a bl; `ni` runs the whole call and stops after it. At a function's first instruction, x0–x7 hold its arguments, and x30 is where it will return to: that one fact explains most of what you will want to look at.

10.3 Reading a crash

On macOS a crash produces a **crash report** (from the system, or from LLDB stopping with EXC_BAD_ACCESS). The parts that matter, in the order to read them:



A crash report for a 64-bit Arm process lists the thread state: x0...x28, fp (x29), lr (x30), sp, pc, the cpsr flags, and a far value, the **fault address register**: the address that caused the exception when it was a memory fault. (Typical layout from memory; field names may vary by macOS version.) A far of 0x0000000000000010 almost always means read a field at offset 16 of a null pointer", which you can confirm by looking for a load with [xN, #16] where xN is 0.

10.4 Pointer authentication (arm64e), briefly

Some Apple code, notably system libraries and the kernel, is compiled for **arm64e**, which adds *pointer authentication*: the unused high bits of a pointer hold a cryptographic signature that is checked before the pointer is used as a return address or function pointer. You may spot instructions such as `pacibsp`, `retab` or `autia` in a disassembly of system code. Ordinary third-party programs are plain `arm64`, which is what this book covers. *(I describe this from general knowledge and did not assemble arm64e code; ask `clang -arch arm64e` on your Mac if you want to see it.)*

10.5 Other tools on your Mac

Tool	Use
<code>otool -tv prog</code>	Disassemble the code of an executable or object.
<code>nm prog</code>	List symbols.
<code>lipo -archs prog</code>	Which architectures a binary contains (<code>arm64</code> , <code>x86_64</code> , ...).
<code>sysctl</code> <code>hw.cachelinesize</code>	Report the cache line size. I believe it is 128 bytes on Apple silicon (64 on most others); <i>confirm on your machine</i> , since it matters for false sharing.
<code>sysctl hw.optional</code>	List optional CPU features (NEON, atomics, ...).
Instruments → Time Profiler	Which functions use time; double-click to see the hot instructions.
Xcode: Debug → Debug Workflow → Always Show Disassembly	View assembly beside source while debugging.

Chapter 10 in five lines

- Build with `-g -O0` to debug; use `si/ni` to step one instruction and `register read` to look.
- At a function's entry, `x0–x7` are its arguments and `x30` is its return address.
- Crash triage: exception type, faulting instruction, address register, backtrace.
- A far near zero usually means a null pointer plus a field offset.
- Check `sysctl` for the real cache line size and CPU features; do not rely on my memory.

Reference Pages

Everything worth keeping within reach: a function template, instruction cards, the condition codes, the calling convention, common idioms, a glossary and where to read more.

11

11.1 The function template

A non-leaf function: prologue, body, epilogue ✓ assembled

```
// Template: a function that calls other functions and uses two callee-saved registers.
.globl _skeleton
.p2align 2
_skeleton:
    stp    x29, x30, [sp, #-32]!    // allocate a 16-byte-multiple frame; save fp and lr
    stp    x19, x20, [sp, #16]     // save the callee-saved registers we intend to use
    mov    x29, sp                // frame pointer = this frame
    mov    x19, x0                // keep arguments that must survive calls in x19, x20...
    mov    x20, x1
    // ... body: bl _helper, using x19/x20 across the call ...
    bl     _helper
    add    x0, x0, x19            // example use of a saved value
    ldp    x19, x20, [sp, #16]     // restore, in the mirror image of the prologue
    ldp    x29, x30, [sp], #32    // pop the frame (sp += 32) and restore fp, lr
    ret
```

Leaf functions need none of this: no `bl` means no frame. If you need no callee-saved registers, a frame of just `stp x29, x30, [sp, #-16]! ... ldp x29, x30, [sp], #16` is enough.

11.2 Instruction cards

Move / constants

<code>mov x0, x1</code> / <code>mov x0, #imm</code>	copy / small constant (alias)
<code>movz, movk, movn</code>	build constants 16 bits at a time
<code>adrp x0, sym@PAGE; add x0, x0, sym@PAGEOFF</code>	address of a static symbol
<code>sxtw, sxtb, sxth</code> / <code>uxtb, uxth</code>	sign- / zero-extend

Arithmetic and logic

<code>add, sub, adds, subs, neg, adc, sbc</code>	add / subtract (s: set flags; adc/sbc: with carry)
<code>mul, madd, msub, smull, umull, smulh, umulh</code>	multiply, multiply-add, widening multiply, high half
<code>udiv, sdiv</code>	divide (no remainder instruction; no trap on zero)
<code>and, orr, eor, bic, orn, eon, mvn, tst</code>	bitwise operations; <code>tst</code> sets flags only
<code>lsl, lsr, asr, ror</code>	shifts and rotate (immediate or register count)
<code>ubfx, sbfx, bfi, bfxil, ubfiz</code>	bit-field extract / insert
<code>clz, rbit, rev, rev16</code>	count leading zeros, reverse bits, reverse bytes

Compare / select

cmp, cmn, tst	set flags from a subtraction, addition, AND
ccmp, ccmn	conditional compare (chains conditions)
csel, csinc, csinv, csneg	conditional select with variants
cset, csetm, cinc, cneg	aliases: 0/1, 0/−1, conditional increment / negate

Branches

b, b.cond	jump; jump if condition
cbz, cbnz, tbz, tbnz	test register / bit and branch, flags untouched
bl, blr	call (label / register)
br, ret	jump to register / return via x30

Load / store

ldr, str (x, w, d, s, q)	8, 4, 8, 4, 16 bytes
ldrb, ldrh, ldrsb, ldrsh, ldrsw	narrow loads with zero-/sign-extension
strb, strh	narrow stores
ldp, stp	load / store two registers
ldur, stur	unscaled immediate offset (−256...255)
ldxr/stxr, ldaxr/stlxr, ldar/stlr, cas, ldadd, swp	exclusive, acquire/release and atomic operations (outside this book)

Floating point / NEON

fadd, fsub, fmul, fdiv, fsqrt, fabs, fneg	scalar arithmetic
fmadd, fmsub, fnmadd	fused multiply-add family
fcmp, fcsel	compare / conditional select
scvtf, ucvtf, fcvtzs, fcvtzu	integer ↔ float
fmov	register move, constant, or bit copy to/from x registers
ld1, st1, ldr q, str q	vector loads/stores
add, fadd, fmul, fmla (.4s etc.)	lane-wise arithmetic
faddp, addp, addv	pairwise / across-lane sums
dup, ins, umov, ext, zip1, uzp1, tbl	lane shuffles and moves

11.3 Condition codes at a glance

AFTER <code>cmp a, b</code>	
<code>eq</code>	$a = b$
<code>ne</code>	$a \neq b$
SIGNED	UNSIGNED
<code>lt</code>	<code>lo</code>
$a < b$	$a < b$
<code>le</code>	<code>ls</code>
$a \leq b$	$a \leq b$
<code>gt</code>	<code>hi</code>
$a > b$	$a > b$
<code>ge</code>	<code>hs</code>
$a \geq b$	$a \geq b$
OTHER	
<code>mi</code>	negative (N)
<code>pl</code>	positive or zero
<code>vs</code>	overflow (V)
<code>vc</code>	no overflow

11.4 Calling convention card

Integer arguments	<code>x0–x7</code> , then the stack
Float arguments	<code>d0–d7</code> / <code>s0–s7</code> , then the stack
Result	<code>x0</code> (<code>x1</code> for 128-bit), <code>d0</code> for floating point; <code>x8</code> = address for a large returned struct
Variadic (<code>printf</code>)	extra arguments on the stack (Apple)
Caller-saved	<code>x0–x17</code> , <code>v0–v7</code> , <code>v16–v31</code>
Callee-saved	<code>x19–x28</code> , <code>x29</code> (fp), <code>x30</code> (lr), low 64 bits of <code>v8–v15</code>
Reserved	<code>x18</code> (platform register)
Stack	grows down; <code>sp</code> 16-byte aligned; <code>x29</code> must point to a valid frame record
Names	C symbols get a leading underscore (<code>_main</code>)

11.5 Common idioms

Write	Does
<code>mov w0, wzr</code> or <code>mov x0, #0</code>	set to zero
<code>cmp x0, #0; b.eq L</code> or <code>cbz x0, L</code>	branch if zero (second form is shorter, flags kept)
<code>add x0, x0, #1</code>	increment
<code>add x0, x1, x2, lsl #3</code>	$x_1 + 8x_2$: address of <code>a[i]</code> for 8-byte elements
<code>ldr x0, [x1, x2, lsl #3]</code>	load <code>a[i]</code> (8-byte) in one instruction
<code>cset w0, ne</code>	<code>bool</code> from flags
<code>csel x0, x1, x2, lt</code>	$x0 = (a < b) ? x1 : x2$ after a <code>cmp</code>
<code>and x0, x0, #(1<<k)-1</code>	remainder by 2^k (unsigned)
<code>subs x1, x1, #1; b.ne loop</code>	count-down loop
<code>stp x29, x30, [sp, #-16]!</code>	push frame record
<code>ldp x29, x30, [sp], #16</code>	pop frame record
<code>nop</code>	do nothing (one instruction; useful for padding)
<code>brk #0</code>	trap into the debugger (like <code>__builtin_trap</code>)

11.6 Glossary

A64 / AArch64	The 64-bit Arm instruction set Apple silicon runs; fixed 32-bit instructions.
ABI	Application Binary Interface: the agreed rules for registers, stack and data layout between separately compiled code.
Alias	A friendly assembler spelling of a more general instruction (<code>mov</code> , <code>cmp</code> , <code>ret</code>).
Arm ARM	The Arm Architecture Reference Manual: the exhaustive specification of the instruction set.
Callee / caller	The function being called / the function making the call.
Frame (record)	The stack space of one function call, plus its saved <code>x29/x30</code> pair.
Immediate	A constant stored inside the instruction, written with <code>#</code> .
Lane	One element of a vector register (4 lanes of 32 bits in <code>.4s</code>).
Leaf function	A function that calls no other function.
Link register	<code>x30</code> : holds the return address after <code>bl</code> .
Mach-O	Apple's executable and object file format.
NEON	Arm's 128-bit SIMD instruction set (the <code>v</code> registers).
Prologue / epilogue	The entry code that builds a frame / the exit code that tears it down.
Relocation	A note telling the linker to patch an address into an instruction.
SIMD	Single instruction, multiple data: one instruction processes many lanes.

11.7 Where to read next

- **Arm Architecture Reference Manual for A-profile (Arm ARM[™])**: the definition of every instruction, with encodings and pseudocode. Large, but exact.
- **Apple: Writing ARM64 code for Apple platforms[™]** (Apple Developer documentation): the authoritative list of Apple's differences from the generic Arm ABI.
- **Compiler Explorer** (godbolt.org): paste C or C++, pick an AArch64 clang, and watch the assembly change as you edit.
- **LLVM's AArch64 documentation** and `clang --help`: flags such as `-mcpu=apple-m1`.
- The next step in this series: the **C++23** and **Graphics** field guides have a "Where ARM64 helps" section showing inline assembly, intrinsics and when to reach for them.

You now know

- How a CPU runs a program, and the A64 registers, flags, memory model and instruction encoding.
- How functions, the stack and the Apple calling convention work, with a reusable template.
- How to read what the compiler produced, from division tricks to vectorised loops.
- Which of this booklet was checked how, and which parts you should confirm on your own Mac.