

The Engineer's Mill

Charles Babbage's Analytical Engine, designed in the 1830s, had two parts: the Store, which held numbers, and the Mill, which worked on them. Every computer since has kept that split. This volume builds a machine from a single kind of logic gate upward, through adders, an ALU, registers and a CPU that runs real programs. It then follows a program from source text to a running process, and explains the memory hierarchy, pipelines and parallel hardware that decide how fast it runs. Every circuit and mechanism is simulated in Python you can run.

CHAPTERS

20

CODE EXAMPLES

20

EXERCISES

60

STAGE BUDGET

150 h

Contents

How to use this volume	4
Part I · Bits	6
Chapter 01 · Binary, hex and two's complement	7
Chapter 02 · Floating point, and why $0.1 + 0.2$ isn't 0.3	14
Part II · From gates to a computer	21
Chapter 03 · Logic gates, from a single NAND	22
Chapter 04 · Adders and the ALU	28
Chapter 05 · Flip-flops, registers and the clock	35
Chapter 06 · Instruction sets and machine code	41
Chapter 07 · The CPU: fetch, decode, execute	47
Chapter 08 · Reading assembly, and stack machines	54
Part III · Memory	61
Chapter 09 · The memory hierarchy and its real latencies	62
Chapter 10 · Caches: lines, sets, ways and misses	67
Chapter 11 · Cache-friendly code and data layout	73
Part IV · Parallelism in the core	80
Chapter 12 · Pipelining and hazards	81
Chapter 13 · Branch prediction and speculative execution	87
Chapter 14 · Instruction-level parallelism and out-of-order execution	93
Chapter 15 · SIMD, multicore and the limits of parallelism	99
Part V · From source to process	105
Chapter 16 · Compiling and assembling	106
Chapter 17 · Linking and loading	113
Chapter 18 · The stack and overflows	120
Part VI · The machine at work	126
Chapter 19 · I/O, interrupts and DMA	127

Chapter 20 · Performance as arithmetic	133
The deliverable	139
The twenty rules	142
Reading list	144

This stage is the gateway to three others. The syllabus puts computer architecture before operating systems, networking and databases because each of them assumes you know what a cache miss costs, what an interrupt is and how a program sits in memory. Without that model, every higher abstraction stays brittle. With it, performance stops being superstition and becomes arithmetic. The table shows where each of the stage's thirteen topics is taught here, and what to read or build alongside it.

STAGE 04 TOPIC	CHAPTERS	READ OR BUILD ALONGSIDE
Number representation: binary, hex, two's complement, overflow	Ch. 1	CS:APP ch. 2
IEEE 754 floating point, and why $0.1 + 0.2$ is not 0.3	Ch. 2	CS:APP ch. 2
Boolean logic, gates, multiplexers, adders, the ALU	Ch. 3–4	Nand2Tetris projects 1–2
Sequential logic: flip-flops, registers, the clock	Ch. 5	Nand2Tetris project 3
Instruction set architecture, machine code, reading assembly	Ch. 6, 8	Nand2Tetris project 4; CS:APP ch. 3
The fetch–decode–execute cycle; datapath and control unit	Ch. 7	Nand2Tetris project 5; CS:APP ch. 4
The memory hierarchy and its real latencies	Ch. 9	CS:APP ch. 6
Cache lines, locality, associativity, cache-friendly layout	Ch. 10–11	CS:APP ch. 6; Patterson & Hennessy ch. 5
Pipelining, hazards, branch prediction, speculative execution	Ch. 12–13	Patterson & Hennessy ch. 4
Instruction-level and data-level parallelism, SIMD	Ch. 14–15	CS:APP ch. 5; Patterson & Hennessy ch. 6
Compiling, assembling, linking and loading	Ch. 16–17	Nand2Tetris projects 6–11; CS:APP ch. 7
The stack frame, calling conventions, buffer overflows	Ch. 18	CS:APP ch. 3

Build it, then simulate it. Nand2Tetris, the stage's main project, has you build each layer in a hardware description language and test it in the course's simulators. This volume explains each layer and simulates it in plain Python, so you can see a mechanism working before you build it yourself: gates composed into an ALU, an assembler and a CPU running a program, a cache counting its misses, a pipeline counting its stalls, a branch predictor learning a pattern. Most examples count events, such as gates, cycles, misses and mispredictions, so they print the same numbers on every machine. The few that time real hardware say so, and show one recorded run. Every output shown is what the program printed when this volume was built.

The deliverable is a machine and a speed-up. The stage asks for the full Nand2Tetris stack, from gates to a compiler, and then for one slow program made four times faster using only cache locality. The appendix maps every Nand2Tetris project to the chapters that prepare for it, and gives a method for the locality project.

THE ONE HABIT

Every question about speed has an arithmetic answer: how many instructions, how many cycles each, how many trips to memory, and how far away the memory is.

An L1 cache hit costs about a nanosecond and a trip to main memory about a hundred. A mispredicted branch costs a dozen or more cycles, and a disk seek costs millions. So **estimate the cost before you measure it, and measure before you optimise**. When the estimate and the measurement disagree, one of your models is wrong, and finding out which is how you learn the machine.

PART I

Bits

Two chapters on how numbers live in hardware: integers as fixed-width patterns of bits, with the wrap-around that comes with them, and real numbers as binary scientific notation, with the rounding that comes with that. Every later chapter computes with these representations, and many famous bugs live in them.

Binary, hex and two's complement

3 exercises

Everything in a computer is a pattern of bits, and a type is just an agreement about how to read one. The same eight bits are 255 or -1 depending on who's asking. Two's complement, the representation every modern processor uses for signed integers, lets one adder serve both readings. That convenience comes at a price: integers have a fixed width, and when a result doesn't fit, most languages wrap around silently. This chapter covers the representations, the arithmetic, and the overflow bugs that have shipped in real software.

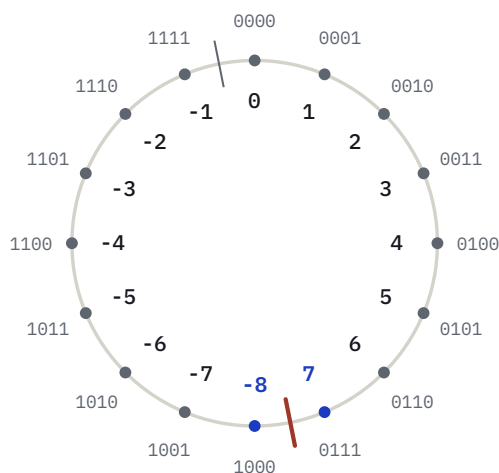
Bases, bits and bytes

A number written in base b has digits worth powers of b , so 2026 is $2 \cdot 10^3 + 0 \cdot 10^2 + 2 \cdot 10 + 6$ in decimal and 11111101010 in binary. Hardware uses base 2 because a wire is easiest to read as one of two states. People write binary in **hexadecimal**, base 16, because each hex digit stands for exactly four bits, a **nibble**, so 0x7EA is 0111 1110 1010 at a glance. Eight bits make a **byte**, the smallest unit memory is addressed in, and a **word** is the natural width of the processor's registers, now almost always 64 bits.

With n bits there are 2^n patterns. Read as **unsigned** integers they stand for 0 to $2^n - 1$, and arithmetic on them is arithmetic modulo 2^n : add 1 to the largest value and you get 0, exactly the clock arithmetic of *The Engineer's Compass*, chapter 5.

Two's complement

For signed integers, every modern processor uses **two's complement**. The top bit's weight is negative: in n bits it's worth -2^{n-1} instead of $+2^{n-1}$, and the other bits keep their usual positive weights. In 8 bits, 10000000 is -128 , 11111111 is $-128 + 127 = -1$, and 01111111 is $+127$. The range is -2^{n-1} to $2^{n-1} - 1$, one more negative number than positive. Equivalently, the pattern for $-x$ is the unsigned number $2^n - x$, which is why the representation lines up so neatly with modular arithmetic.



4-BIT TWO'S COMPLEMENT

adding 1 moves one step clockwise

unsigned: 0 to 15, wraps 1111 → 0000

signed: -8 to 7, wraps 0111 → 1000

the top bit is worth -8 instead of +8;
the other three bits keep +4, +2, +1

Integers of fixed width live on a circle, not a line. Unsigned and signed readings label the same circle differently, and each has one place where adding 1 jumps from the largest value to the smallest. That jump is overflow, and in most languages nothing stops you from making it.

To negate a two's complement number, invert every bit and add 1: inverting gives $(2^n - 1) - x$, and adding 1 gives $2^n - x$ (exercise 1.2). There's one zero, unlike older sign-and-magnitude schemes with a +0 and a -0. Most importantly, the hardware that adds unsigned numbers adds signed ones correctly too, because both are arithmetic modulo 2^n . The processor doesn't need to know which reading you intend. Only comparisons, division, widening and overflow detection differ. The asymmetry has one sharp edge: the most negative number has no positive partner, so negating -128 in 8 bits gives -128 again, and `abs(INT_MIN)` in C is negative.

Overflow

An unsigned sum overflows when it carries out of the top bit. A signed sum overflows when two operands of the same sign produce a result of the other sign. In 8 bits, $100 + 50$ gives the pattern 10010110, which is 150 unsigned and -106 signed. Languages disagree about what to do. C and C++ make signed overflow *undefined behaviour*, so compilers may assume it never happens and delete checks that rely on it, while unsigned overflow wraps by definition. Java and Go wrap signed integers silently. Rust panics in debug builds and wraps in release builds unless you ask for checked arithmetic. Python's integers grow without limit, so the only overflow is running out of memory.

Wrap-around has caused famous failures. In 2006 Joshua Bloch reported that the binary search in Java's standard library, and in many textbooks, computed its midpoint as `(low + high) / 2`. With arrays of over a billion elements, the sum overflows 32 bits and becomes negative. The fix is `low + (high - low) / 2`, and the example reproduces both. Systems that store time as a signed 32-bit count of seconds since 1970 run out on 19 January 2038 at 03:14:07 UTC, one second before they jump back to December 1901. And in 1996 the first

Ariane 5 rocket was lost about forty seconds after launch, when a conversion of a 64-bit floating-point value into a 16-bit signed integer overflowed in software reused from Ariane 4, whose slower trajectory had never produced values that large.

Shifts, widening and division

Shifting left by k multiplies by 2^k , dropping bits that fall off the top. Shifting right comes in two kinds. A **logical** shift fills with zeros, which suits unsigned values. An **arithmetic** shift copies the sign bit, so $-20 \gg 2$ is -5 , which suits signed values. Widening a signed value to more bits needs **sign extension**, copying the top bit into the new bits, so that -1 in 8 bits stays -1 in 32. Integer division of negative numbers differs too. Python's `//` and `>>` round down, so $-7 // 2$ is -4 , while C, Java and Go round integer division towards zero, giving -3 . Ported code that relies on either behaviour breaks quietly.

Byte order

A 32-bit value occupies four bytes, and machines disagree about their order. **Little-endian** machines, which include x86 and almost all ARM systems, store the least significant byte first, so `0x12345678` sits in memory as `78 56 34 12`. **Big-endian** order, `12 34 56 78`, is the convention for network protocols, where it's called network byte order, and for many file formats. Byte order never matters inside a program's own arithmetic. It matters the moment bytes cross a boundary: written to a file, sent over a network, or read from a memory dump. Then the format, not the machine, must say which order it uses, and code must convert explicitly, as Python's `struct` and `int.from_bytes` do.

Bits, run

The example converts between bases, encodes and decodes 8-bit two's complement, detects overflow, compares shifts and division rules, shows both byte orders, reproduces the midpoint and 2038 bugs in 32-bit arithmetic, and counts set bits with a classic trick.

EXAMPLE

`bits.py` · bases, two's complement, overflow, shifts, byte order and two real bugs

```
import struct
from datetime import datetime, timedelta, timezone

# 1. One number, four bases
n = 2026
print(f"{n} = 0b{n:b} = 0o{n:o} = 0x{n:X}; as 4 bytes, big-endian: {n.to_bytes(4, 'big')}.hex('
    ')}")

# 2. Two's complement in 8 bits: the bit pattern of -x is (2^8 - x)
BITS = 8
def encode(x): return x & ((1 << BITS) - 1)          # keep the low 8 bits
def decode(u): return u - (1 << BITS) if u & (1 << (BITS - 1)) else u
for x in (5, -5, 127, -128, -1):
    print(f"{x}>5} -> {encode(x):08b} (0x{encode(x):02X}) -> {decode(encode(x)):>5}")
print(f"negate 5 by inverting and adding 1: ~00000101 + 1 = {encode(~5 + 1):08b} =
    {decode(encode(~5 + 1))}")
print(f"negating -128 gives {decode(encode(128))}: there is no +128 in 8 bits")
```

```

# 3. Overflow: the same adder, read two ways
def add8(a, b):
    s = encode(a + b)
    signed_overflow = (a >= 0) == (b >= 0) and (decode(s) >= 0) != (a >= 0)  # same signs in,
    other sign out
    return s, signed_overflow
s, ov = add8(100, 50)
print(f"\n100 + 50 in 8 bits = {s:08b}: unsigned {s}, signed {decode(s)}, signed overflow:
      {ov}")
s, ov = add8(-100, -50)
print(f"-100 + -50 in 8 bits = {s:08b}: signed {decode(s)}, signed overflow: {ov}")
print(f"0xFF read as unsigned is {0xFF}, read as signed is {decode(0xFF)}")

# 4. Shifts: arithmetic shifts keep the sign; integer division rounds differently in different
    languages
x = -20
print(f"\n-20 = {encode(x):08b}; arithmetic >> 2 = {encode(x >> 2):08b} ({x >> 2}); "
      f"logical >> 2 = {encode(x) >> 2:08b} ({encode(x) >> 2})")
print(f"-7 >> 1 = {-7 >> 1} and -7 // 2 = {-7 // 2} (round down); C's -7 / 2 is -3 (round
      towards zero)")

# 5. Byte order
v = 0x12345678
print(f"\n0x12345678 little-endian: {struct.pack('<I', v).hex(' ')} big-endian:
      {struct.pack('>I', v).hex(' ')}")

# 6. Real overflow bugs, reproduced in 32-bit arithmetic
def i32(x): x &= 0xFFFFFFFF; return x - (1 << 32) if x >> 31 else x
lo, hi = 1_500_000_000, 1_600_000_000
print(f"binary search midpoint in 32-bit ints: (lo + hi) / 2 = {i32(lo + hi) // 2:,}; lo + (hi
      - lo) / 2 = {lo + (hi - lo) // 2:,}")
epoch = datetime(1970, 1, 1, tzinfo=timezone.utc)
last = epoch + timedelta(seconds=2 ** 31 - 1)
wrapped = epoch + timedelta(seconds=i32(2 ** 31))
print(f"a signed 32-bit time_t ends at {last:%Y-%m-%d %H:%M:%S} UTC; one second later it reads
      {wrapped:%Y-%m-%d %H:%M:%S}")

# 7. Bit tricks
def popcount(x):
    c = 0
    while x: x &= x - 1; c += 1  # clears the lowest set bit each
    time
    return c
print(f"\npopcount(2026) = {popcount(2026)} (bin(2026).count('1') = {bin(2026).count('1')}); "
      f"powers of two below 70: {[k for k in range(1, 70) if k & (k - 1) == 0]}")

```

OUTPUT

```

2026 = 0b11111101010 = 0o3752 = 0x7EA; as 4 bytes, big-endian: 00 00 07 ea
  5 -> 00000101 (0x05) ->    5
 -5 -> 11111011 (0xFB) ->   -5
127 -> 01111111 (0x7F) ->  127
-128 -> 10000000 (0x80) -> -128
 -1 -> 11111111 (0xFF) ->   -1
negate 5 by inverting and adding 1: ~00000101 + 1 = 11111011 = -5
negating -128 gives -128: there is no +128 in 8 bits

100 + 50 in 8 bits = 10010110: unsigned 150, signed -106, signed overflow: True
-100 + -50 in 8 bits = 01101010: signed 106, signed overflow: True
0xFF read as unsigned is 255, read as signed is -1

-20 = 11101100; arithmetic >> 2 = 11111011 (-5); logical >> 2 = 00111011 (59)
-7 >> 1 = -4 and -7 // 2 = -4 (round down); C's -7 / 2 is -3 (round towards zero)

0x12345678 little-endian: 78 56 34 12 big-endian: 12 34 56 78

```

```
binary search midpoint in 32-bit ints: (lo + hi) / 2 = -597,483,648; lo + (hi - lo) / 2 =
1,550,000,000
a signed 32-bit time_t ends at 2038-01-19 03:14:07 UTC; one second later it reads 1901-12-13
20:45:52

popcount(2026) = 8 (bin(2026).count('1') = 8); powers of two below 70: [1, 2, 4, 8, 16, 32,
64]
```

The last line uses the identity that `x & (x - 1)` clears the lowest set bit of `x`. Subtracting 1 turns the lowest 1 into 0 and the zeros below it into ones, and the `&` then erases all of them. Repeating it until `x` is zero counts the set bits, and `x` is a power of two exactly when one application gives zero. Bit tricks like these are the vocabulary of hash functions, compression and the flags fields of network headers. Processors also offer them as single instructions such as `popcnt`.

COMMON TRAPS

Mixing signed and unsigned in a comparison. In C, `-1 < 0u` is false, because `-1` is converted to the largest unsigned value.

Assuming overflow raises an error. In most languages it wraps, and in C it's undefined.

Taking the absolute value of the most negative integer, which overflows back to itself.

Right-shifting a negative number and expecting a logical shift, or the reverse.

Writing binary data without stating its byte order, then reading it on a machine with the other one.

Storing timestamps or counters in 32 bits that will outgrow them during the system's lifetime.

EXERCISES

EXERCISE 1.1 Subtraction is addition

Write `-37` in 8-bit two's complement, then compute `100 + (-37)` in binary, ignoring any carry out of the top bit. Check the result.

SOLUTION

37 is 00100101. Inverting gives 11011010 and adding 1 gives 11011011, which is 219 as an unsigned number, and $256 - 37 = 219$, as it should be. Adding `100 = 01100100` gives 1 00111111. Drop the carry out of bit 8 and the result is 00111111 = 63 = `100 - 37`. The carry out of the top bit doesn't

indicate signed overflow: the operands had different signs, so overflow was impossible. This is how processors subtract, by adding the two's complement, so a single adder circuit does both (chapter 4).

EXERCISE 1.2 Why invert-and-add-one negates

Prove that for an n -bit pattern x , inverting every bit and adding 1 gives the two's complement representation of $-x$.

SOLUTION

Inverting every bit of x gives the pattern whose bits are 1 exactly where x 's are 0, so x and its inverse add to $11\dots 1$, which is $2^n - 1$. The inverse is therefore $(2^n - 1) - x$, and adding 1 gives $2^n - x$. Working modulo 2^n , that's congruent to $-x$, and the two's complement pattern of $-x$ is by definition $2^n - x$. ■ For $x = 0$ the result is 2^n , which wraps to 0, so zero is its own negation. For $x = 2^{n-1}$, the most negative number, it gives 2^{n-1} again, which is the asymmetry described above.

EXERCISE 1.3 An overflow check the compiler deletes

In C, `if (a + b < a)` is a correct test for overflow when a and b are unsigned, and b is at most the maximum value. Why does it work for unsigned integers, and why can a compiler legally delete it when they're signed?

SOLUTION

Unsigned arithmetic wraps modulo 2^n by definition. If $a + b$ fits, the result is at least a . If it doesn't, the true sum is less than $a + 2^n$, so the wrapped result $a + b - 2^n$ is less than a . So the test fires exactly on overflow. For signed integers, overflow is undefined behaviour, so the compiler may assume it never happens. Under that assumption, $a + b < a$ can only be true when $b < 0$, and for a known non-negative b the compiler may conclude the condition is always false and remove it. Correct signed checks test before adding, for example `b > 0 && a > INT_MAX - b`, or use the `__builtin_add_overflow` intrinsic in GCC and Clang.

CHECKLIST

☐ Can you convert between binary, hex and decimal quickly?

- ☐ Can you encode and decode two's complement, and explain why one adder serves signed and unsigned numbers?
- ☐ Do you know what your languages do on overflow?
- ☐ Can you tell arithmetic from logical shifts, and explain sign extension?
- ☐ Do you know when byte order matters, and how to make it explicit?

GO DEEPER

BOOK Computer Systems: A Programmer's Perspective, chapter 2 Bryant & O'Hallaron

ARTICLE Nearly all binary searches and mergesorts are broken Joshua Bloch, 2006

REFERENCE Bit twiddling hacks Sean Eron Anderson

Floating point, and why 0.1 + 0.2 isn't 0.3

3 exercises

Floating point is scientific notation in binary, with a fixed number of significant bits. Most decimal fractions, 0.1 among them, have no exact binary form, so they're rounded the moment they're written down. Every operation rounds again. The gaps between representable numbers grow with their size, addition stops being associative, and subtracting two nearly equal numbers can wipe out every correct digit. None of this is a bug. IEEE 754 specifies it exactly, and it's what makes floating point fast and predictable. This chapter shows the layout, the rounding and the special values, then the habits that keep numerical code correct.

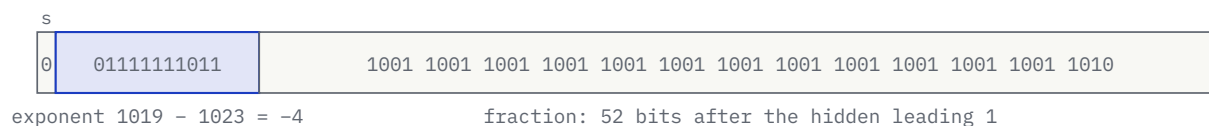
The layout

The IEEE 754 **double**, Python's `float` and JavaScript's only number type, packs a number into 64 bits: a sign bit s , an 11-bit exponent e and a 52-bit fraction f . For normal numbers the value is

$$(-1)^s \times 1.f \times 2^{e-1023}.$$

The leading 1 before the binary point isn't stored, since every normal number has one, so the 52 stored bits give 53 bits of precision. The exponent is stored with a **bias** of 1023 so that it can be compared as an unsigned number. The 32-bit **single** format has 8 exponent bits and 23 fraction bits, about 7 significant decimal digits against the double's 15 to 17.

THE DOUBLE NEAREST 0.1



value = $+1.6000000000000000888 \times 2^{-4} = 0.100000000000000055511\dots$

A TOY FORMAT WITH 2 FRACTION BITS



the same relative precision everywhere; the absolute gap doubles at each power of two

Top: the double nearest to 0.1. One tenth is 0.000110011... in binary, repeating for ever, so the 52 fraction bits cut it off and round the last bit up. Bottom: in any floating-point format the representable numbers are evenly spaced within each power of two, and the spacing doubles at the next one, so the relative precision stays constant while the absolute gaps grow.

Why 0.1 + 0.2 isn't 0.3

The double stored for 0.1 is exactly 0.10000000000000005551151231257827..., slightly too large, and the ones for 0.2 and 0.3 are rounded too. IEEE 754 guarantees that each basic operation is **correctly rounded**: the result is the exact result of the stored inputs, rounded to the nearest double, with ties going to the even neighbour. The sum of the stored 0.1 and 0.2 rounds to a double one step above the double stored for 0.3, so they differ by 2^{-54} , and Python prints 0.30000000000000004. The arithmetic was as accurate as it could be. The inputs weren't the numbers you typed. So **never compare floating-point results with ==**. Compare with a tolerance suited to the problem, as `math.isclose` does with a relative tolerance by default.

Precision, and gaps that grow

The distance from a double to the next one is its **unit in the last place**, or ulp. Near 1 it's $2^{-52} \approx 2.2 \times 10^{-16}$, the **machine epsilon**. Near a million it's about 10^{-10} , and above $2^{53} \approx 9 \times 10^{15}$ it's 2 or more, so doubles stop representing every integer. The example shows $2^{53} + 1$ rounding to 2^{53} . That matters outside numerical code. JavaScript stores every number as a double, so 64-bit identifiers sent as JSON numbers lose their low digits in a browser, which is why APIs send large IDs as strings. Single precision runs out of integers at $2^{24} = 16,777,216$, which is why games with large worlds see objects jitter far from the origin.

Rounding accumulates, and order matters

Each operation rounds, and errors build up over many operations. Addition isn't associative: $(10^{16} + 1) - 10^{16}$ is 0, because adding 1 to 10^{16} falls within half an ulp and is rounded away, while $(10^{16} - 10^{16}) + 1$ is 1. Adding 0.1 ten million times in a loop gives

999,999.99984, off by about 1.6×10^{-4} , because each small addend loses its low bits against the growing total. **Kahan summation** carries the lost low-order bits in a separate correction variable and adds them back, and Python's `math.fsum` goes further, tracking partial sums exactly and rounding once at the end. Both give 1,000,000.0. Order dependence has a practical consequence: a parallel sum, which adds in whatever order the threads finish, can give slightly different results on every run.

Special values

An exponent of all ones encodes **infinity**, when the fraction is zero, or NaN, "not a number", when it isn't. Overflow produces $\pm\infty$, and invalid operations such as $\infty - \infty$ or $0/0$ produce NaN, which then propagates through every later operation. NaN is unequal to everything, including itself, which breaks sorting and hash tables (*The Engineer's Compass*, chapter 4). There are two zeros, +0 and -0, equal under `==` but distinguishable by their sign, so $1/-0$ is $-\infty$. An exponent of all zeros encodes **subnormal** numbers, which fill the gap between the smallest normal double, about 2.2×10^{-308} , and zero. That's **gradual underflow**, and on some processors arithmetic on subnormals is much slower than normal arithmetic.

Catastrophic cancellation

Subtracting two nearly equal numbers cancels their shared leading digits and leaves only the low digits, which are the ones carrying rounding error. For small x , $\cos x$ is very close to 1, so $1 - \cos x$ keeps hardly any correct digits. At $x = 10^{-8}$, $\cos x$ rounds to exactly 1 and $(1 - \cos x)/x^2$ comes out as 0 instead of 0.5. The identity $1 - \cos x = 2 \sin^2(x/2)$ computes the same quantity without subtracting, and stays accurate. The same issue makes the textbook formula for variance, the mean of the squares minus the square of the mean, useless for data with a large mean and a small spread (exercise 2.3). The general cure is algebraic: rearrange the formula so it doesn't subtract nearly equal quantities.

Floats, measured

The example decodes the fields of three doubles, shows the exact stored value of 0.1, measures the gaps between doubles at four magnitudes, compares three ways of summing, exercises the special values, and computes $(1 - \cos x)/x^2$ both ways.

EXAMPLE

floats.py · the bits of a double, rounding, gaps, summation, special values and cancellation

```
import math, struct
from decimal import Decimal
from fractions import Fraction

# 1. The bits of a double: 1 sign bit, 11 exponent bits, 52 fraction bits
def fields(x):
    b = struct.unpack(">Q", struct.pack(">d", x))[0]
    return b >> 63, (b >> 52) & 0x7FFF, b & ((1 << 52) - 1)
for x in (1.0, -2.5, 0.1):
    s, e, f = fields(x)
```

```

    print(f"{x:>5}: sign {s}  exponent {e:011b} ({e} - 1023 = {e - 1023:>3})  fraction {f:013X}
          (hex)")
print(f"0.1 is stored as exactly {Decimal(0.1)}")

# 2. Why 0.1 + 0.2 != 0.3
print(f"\n0.1 + 0.2 = {0.1 + 0.2!r}; == 0.3: {0.1 + 0.2 == 0.3}; nearest doubles differ by
      {Fraction(0.1 + 0.2) - Fraction(0.3)}")
print(f"math.isclose(0.1 + 0.2, 0.3): {math.isclose(0.1 + 0.2, 0.3)}")

# 3. The gaps between doubles grow with their size
for x in (1.0, 1e6, 1e16, 2.0 ** 53):
    print(f"next double after {x:>22,.1f} is {math.ulp(x):.3g} away")
print(f"2^53 + 1 = {2 ** 53 + 1:,} becomes {float(2 ** 53 + 1):,.0f} as a double")
f32 = struct.unpack("f", struct.pack("f", 16_777_217.0))[0]
print(f"in single precision, 16,777,217 becomes {f32:,.0f}")

# 4. Addition is not associative, and summation order matters
print(f"\n(1e16 + 1) - 1e16 = {(1e16 + 1) - 1e16}; (1e16 - 1e16) + 1 = {(1e16 - 1e16) + 1}")
naive = 0.0
for _ in range(10_000_000): naive += 0.1
def kahan(values):
    total = c = 0.0
    for v in values:
        y = v - c; t = total + y
        c = (t - total) - y; total = t
        # c recovers the low bits lost
        in the addition
    return total
print(f"adding 0.1 ten million times: naive {naive!r}, Kahan {kahan(0.1 for _ in
range(10_000_000))!r}, "
      f"math.fsum {math.fsum(0.1 for _ in range(10_000_000))!r}")

# 5. Special values
inf, nan = float("inf"), float("nan")
print(f"\ninf - inf = {inf - inf}; nan == nan: {nan == nan}; -0.0 == 0.0: {-0.0 == 0.0}, "
      f"but copysign(1, -0.0) = {math.copysign(1, -0.0)}")
print(f"largest double {1.7976931348623157e308:.6e}; smallest normal {2.2250738585072014e-
308:.6e}; "
      f"smallest subnormal {5e-324:.1e}; 5e-324 / 2 = {5e-324 / 2}")

# 6. Catastrophic cancellation, and an algebraically identical formula that avoids it
for x in (1e-4, 1e-6, 1e-8):
    bad = (1 - math.cos(x)) / x ** 2
    good = 2 * math.sin(x / 2) ** 2 / x ** 2
    print(f"(1 - cos x)/x^2 at x = {x:.0e}: direct {bad:.10f}, rewritten {good:.10f} (true
          limit 0.5)")

```

OUTPUT

```

1.0: sign 0  exponent 01111111111 (1023 - 1023 = 0)  fraction 0000000000000 (hex)
-2.5: sign 1  exponent 10000000000 (1024 - 1023 = 1)  fraction 4000000000000 (hex)
0.1: sign 0  exponent 01111111011 (1019 - 1023 = -4)  fraction 99999999999A (hex)
0.1 is stored as exactly 0.1000000000000000055511151231257827021181583404541015625

0.1 + 0.2 = 0.300000000000000004; == 0.3: False; nearest doubles differ by 1/18014398509481984
math.isclose(0.1 + 0.2, 0.3): True
next double after 1.0 is 2.22e-16 away
next double after 1,000,000.0 is 1.16e-10 away
next double after 10,000,000,000,000,000.0 is 2 away
next double after 9,007,199,254,740,992.0 is 2 away
2^53 + 1 = 9,007,199,254,740,993 becomes 9,007,199,254,740,992 as a double
in single precision, 16,777,217 becomes 16,777,216

(1e16 + 1) - 1e16 = 0.0; (1e16 - 1e16) + 1 = 1.0
adding 0.1 ten million times: naive 999999.9998389754, Kahan 1000000.0, math.fsum 1000000.0

inf - inf = nan; nan == nan: False; -0.0 == 0.0: True, but copysign(1, -0.0) = -1.0

```

```
largest double 1.797693e+308; smallest normal 2.225074e-308; smallest subnormal 4.9e-324; 5e-324 / 2 = 0.0
(1 - cos x)/x^2 at x = 1e-04: direct 0.4999999970, rewritten 0.4999999996 (true limit 0.5)
(1 - cos x)/x^2 at x = 1e-06: direct 0.5000444503, rewritten 0.5000000000 (true limit 0.5)
(1 - cos x)/x^2 at x = 1e-08: direct 0.0000000000, rewritten 0.5000000000 (true limit 0.5)
```

The fields line up with the formula. 1.0 has exponent field 1023, meaning 2^0 , and fraction 0. -2.5 is -1.25×2^1 , with fraction 0x4000000000000, binary .01. 0.1 is 1.6×2^{-4} , whose fraction 0x999999999999A is the repeating pattern 1001 1001... ending in a rounded-up 1010. For money, use integers counting the smallest unit, or a decimal type such as Python's `Decimal`. Binary floating point can't hold 0.10 exactly, and an accounting system can't round away a cent.

COMMON TRAPS

Comparing floats with ==, including loop conditions like `while x != 1.0`.

Storing money in binary floating point. Use integer cents or a decimal type.

Sending 64-bit integers to JavaScript as numbers, which silently rounds them above 2^{53} .

Summing many small values into a large total without Kahan summation or `math.fsum`.

Subtracting nearly equal quantities, as in naive variance formulas.

Expecting identical results everywhere. Fused multiply-add, compiler flags such as `-ffast-math` and parallel reduction order all change the last bits.

EXERCISES

EXERCISE 2.1 Decode a float by hand

Decode the 32-bit single-precision pattern 0xC0490FDB. Single precision has 1 sign bit, 8 exponent bits with bias 127, and 23 fraction bits.

SOLUTION

In binary, 0xC0490FDB is 1100 0000 0100 1001 0000 1111 1101 1011. The sign bit is 1, so the number is negative. The next 8 bits, 1000 0000, give exponent field 128, so the scale is $2^{128-127} = 2$. The remaining 23 bits are the fraction .1001001 0000 1111 1101 1011, so the significand is $1.1001001000011111101101_2 \approx 1.5707964$. The value is

$-1.5707964 \times 2 \approx -3.1415927$: it's $-\pi$, rounded to single precision. The digits after the seventh are wrong, since $\pi = 3.14159265\dots$, which is what about 24 bits of precision buys.

EXERCISE 2.2 A loop that never ends

Why might `for (float f = 0.0f; f != 1.0f; f += 0.1f)` never terminate? How should the loop be written?

SOLUTION

0.1f isn't exactly one tenth, and each addition rounds. After ten steps, f is 1.0000001192... in single precision, one ulp above 1.0f, so `f != 1.0f` stays true and the loop runs past 1 for ever, or until something else stops it. Loop over an integer counter, and compute the float from it:

```
for (int i = 0; i < 10; i++) { float f = i * 0.1f; ... }.
```

Computing each value from the counter also prevents error from accumulating across iterations, since each value is rounded once instead of ten times.

EXERCISE 2.3 The variance formula that fails

The population variance equals $\overline{x^2} - \bar{x}^2$, the mean of the squares minus the square of the mean. For the values $10^9 + 4$, $10^9 + 7$, $10^9 + 13$ and $10^9 + 16$, this formula in double precision gives 0, while the true variance is 22.5. Explain why, and give a formula that works.

SOLUTION

Each square is about 10^{18} , where the gap between doubles is 128, so the squares are stored with an error of up to 64 each. The mean of the squares and the square of the mean are both about 10^{18} and agree in their first 16 digits, and the variance, 22.5, is far smaller than the rounding error in either. Subtracting them cancels everything except noise, which here happens to be 0. The fix is to subtract the mean before squaring: $\frac{1}{n} \sum (x_i - \bar{x})^2$ works with numbers like 4.5 and 2.5 instead of 10^{18} . For data arriving one value at a time, Welford's algorithm updates the mean and the sum of squared deviations incrementally with the same stability. Python's `statistics.pvariance` returns 22.5 for this data.

- ☐ Can you decode the sign, exponent and fraction of a double?
- ☐ Can you explain why $0.1 + 0.2$ isn't 0.3, and how to compare floats properly?
- ☐ Do you know where doubles and singles stop representing every integer?
- ☐ Can you explain non-associativity, and use compensated summation?
- ☐ Can you recognise catastrophic cancellation and rewrite a formula to avoid it?

GO DEEPER

ARTICLE What every computer scientist should know about floating-point arithmetic David Goldberg, 1991

REFERENCE Floating-point arithmetic: issues and limitations Python documentation

INTERACTIVE float.exposed: explore the bits of any float Bartosz Ciechanowski

PART II

From gates to a computer

Six chapters that build a working computer from one kind of logic gate, following the path of Nand2Tetris. Gates become adders and an ALU; feedback and a clock add memory; an instruction set defines what the hardware promises; a CPU fetches and executes those instructions; and reading assembly connects the machine back to the programs you write.

Logic gates, from a single NAND

3 exercises

A logic gate computes a boolean function of its inputs, and one gate is enough to build all the others. NAND, "not and", is functionally complete: Not, And, Or, Xor, the multiplexers that choose between signals and, eventually, a whole computer can be built from NAND gates alone. That's where Nand2Tetris starts. This chapter builds the basic gates, counts what each one costs in NANDs and in delay, turns any truth table into a circuit, and shows why the shape of a circuit matters as much as its size.

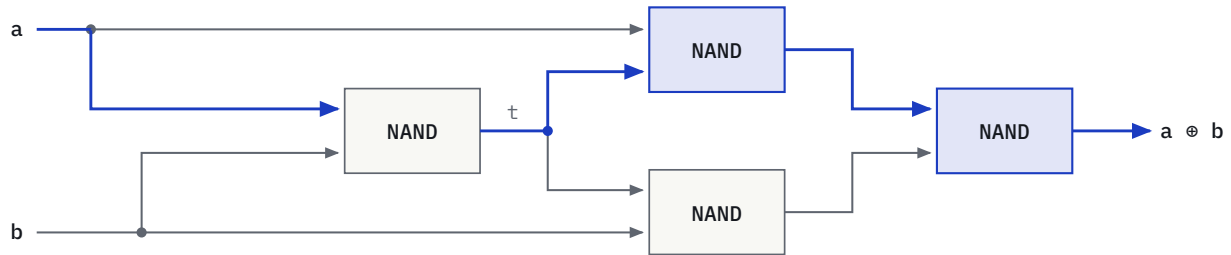
Gates are functions, and wires are variables

A **combinational** circuit has no memory. Its outputs depend only on its current inputs, so it computes a function, and its truth table describes it completely. Physically, a gate is a handful of transistors, and a modern processor contains billions of them. Two costs matter. **Size** is the number of gates, which costs chip area and power. **Depth** is the longest chain of gates from any input to the output. Each gate takes time to settle, so depth sets how long the circuit needs before its output is valid, and therefore, in chapter 5, how fast the clock can tick.

Everything from NAND

NAND is 0 only when both inputs are 1. *The Engineer's Compass*, exercise 1.3, proved that it can express any boolean function. Here are the constructions, built up so that each gate uses the ones before:

- **Not**(a) = Nand(a, a), one gate;
- **And**(a, b) = Not(Nand(a, b)), two gates;
- **Or**(a, b) = Nand(Not a, Not b), three gates, by De Morgan's law;
- **Xor**(a, b): with $t = \text{Nand}(a, b)$, the result is $\text{Nand}(\text{Nand}(a, t), \text{Nand}(b, t))$, four gates.



4 gates; the highlighted path from a to the output passes through 3 of them, so the depth is 3

Xor from four NANDs. The first gate's output t is shared by two others, which is what makes four gates enough. Sharing intermediate signals, called fan-out, is how real circuits stay small.

The **multiplexer**, or Mux, is the gate that makes computers programmable. It takes two data inputs and a selector, and passes through one input or the other: $\text{Mux}(a, b, \text{sel}) = \text{sel} ? b : a$, four NANDs. Its mirror image, the **demultiplexer**, routes one input to one of two outputs. Wider versions select among 4, 8 or more inputs and handle whole 16- or 64-bit words at once. A 16-bit And is just 16 Ands side by side: 32 NANDs, still depth 2, because the bits don't interact. Multiplexers are how a CPU chooses which register to read, which ALU function to apply and where the next instruction comes from.

From any truth table to a circuit

Every truth table has a circuit. For each input row where the output is 1, build an And of the inputs, negated where that row has a 0, and Or all of these terms together. This **sum of products** is correct by construction, and it's how logic synthesis starts, but it's rarely efficient. For the majority function, which is 1 when at least two of three inputs are 1, the sum of products uses 28 NANDs at depth 11, while a hand-designed version uses 6 NANDs at depth 4. Turning a specification into a small, shallow circuit is the job of synthesis tools. Like an optimising compiler, they apply algebraic identities until nothing more can be saved.

Shape matters as much as size. An 8-way Or, which is 1 if any of eight inputs is 1, can be a chain of seven Ors or a balanced tree of seven Ors. Both use 21 NANDs, but the chain is 14 gates deep and the tree only 6. Depth grows linearly in the chain and logarithmically in the tree. The same choice, at the scale of 64-bit additions, decides how fast a processor can add (chapter 4).

Built and checked

The example builds every gate above from a single `nand` function. Each signal carries its value and its depth, and a counter tracks how many NANDs each evaluation used. Every gate is checked against its full truth table.

EXAMPLE

gates.py · every gate from NAND, counted and checked, and a circuit made from a truth table

```
from itertools import product

# A signal is (value, depth): depth counts the gates on the longest path from the inputs.
gates_used = 0
def nand(a, b):
    global gates_used; gates_used += 1
    return (1 - (a[0] & b[0]), max(a[1], b[1]) + 1)

def NOT(a): return nand(a, a)
def AND(a, b): return NOT(nand(a, b))
def OR(a, b): return nand(NOT(a), NOT(b))
def XOR(a, b):
    t = nand(a, b)
    return nand(nand(a, t), nand(b, t))
def MUX(a, b, sel): return nand(nand(a, NOT(sel)), nand(b, sel)) # sel ? b : a
def DMUX(x, sel): return AND(x, NOT(sel)), AND(x, sel)

def check(name, circuit, reference, arity):
    global gates_used
    ok, depth = True, 0
    for bits in product((0, 1), repeat=arity):
        gates_used = 0
        out = circuit(*[(b, 0) for b in bits])
        outs = out if isinstance(out, tuple) and isinstance(out[0], tuple) else (out,)
        ok &= tuple(o[0] for o in outs) == tuple(reference(*bits) if
            isinstance(reference(*bits), tuple) else (reference(*bits),))
        depth = max(depth, max(o[1] for o in outs))
    print(f" {name:10} {gates_used:2} NANDs depth {depth} matches its truth table: {ok}")
print("every gate built from NAND alone:")
check("Not", NOT, lambda a: 1 - a, 1)
check("And", AND, lambda a, b: a & b, 2)
check("Or", OR, lambda a, b: a | b, 2)
check("Xor", XOR, lambda a, b: a ^ b, 2)
check("Mux", MUX, lambda a, b, s: b if s else a, 3)
check("DMux", DMUX, lambda x, s: (x & (1 - s), x & s), 2)

# Any truth table becomes a circuit: OR together one AND term per row that outputs 1
def from_truth_table(table, arity):
    def circuit(*ins):
        terms = []
        for bits in product((0, 1), repeat=arity):
            if table(*bits):
                lits = [x if b else NOT(x) for x, b in zip(ins, bits)]
                t = lits[0]
                for l in lits[1:]: t = AND(t, l)
                terms.append(t)
        out = terms[0]
        for t in terms[1:]: out = OR(out, t)
        return out
    return circuit
majority = lambda a, b, c: int(a + b + c >= 2)
print("\nmajority of three inputs:")
check("from table", from_truth_table(majority, 3), majority, 3)
check("by hand", lambda a, b, c: nand(AND(nand(a, b), nand(a, c)), nand(b, c)), majority, 3)

# Wider gates are copies: a 16-bit AND is 16 Ands side by side, still depth 2
gates_used = 0
word = [AND((1, 0), (b, 0)) for b in (1, 0) * 8]
print(f"\nAnd16: {gates_used} NANDs, depth {max(w[1] for w in word)}")
inputs = [(0, 0)] * 7 + [(1, 0)]
```

```

gates_used = 0
acc = inputs[0]
for b in inputs[1:]: acc = OR(acc, b)           # a chain of 7 Ors
chain_gates, chain_depth = gates_used, acc[1]
gates_used = 0
level = inputs
while len(level) > 1: level = [OR(level[i], level[i + 1]) for i in range(0, len(level), 2)]
print(f"Or8Way: {chain_gates} NANDs and depth {chain_depth} as a chain; {gates_used} NANDs and
      depth {level[0][1]} as a balanced tree")

```

OUTPUT

every gate built from NAND alone:

Not	1 NANDs	depth 1	matches its truth table: True
And	2 NANDs	depth 2	matches its truth table: True
Or	3 NANDs	depth 2	matches its truth table: True
Xor	4 NANDs	depth 3	matches its truth table: True
Mux	4 NANDs	depth 3	matches its truth table: True
DMux	5 NANDs	depth 3	matches its truth table: True

majority of three inputs:

from table	28 NANDs	depth 11	matches its truth table: True
by hand	6 NANDs	depth 4	matches its truth table: True

And16: 32 NANDs, depth 2

Or8Way: 21 NANDs and depth 14 as a chain; 21 NANDs and depth 6 as a balanced tree

COMMON TRAPS

Counting gates and forgetting depth. A small, deep circuit can be slower than a larger, shallow one.

Confusing a Mux with a DMux: a Mux chooses one of several inputs, a DMux sends one input to one of several outputs.

Trusting a hand-optimised circuit without checking its whole truth table. With n inputs there are only 2^n rows, so check them all.

Assuming a sum of products is a good circuit. It's a correct starting point, not an efficient one.

Thinking of gates as instantaneous. Every gate delays its output, and chapter 5's clock exists because of that.

EXERCISES

EXERCISE 3.1 A 4-way multiplexer

Build a Mux4Way, which selects one of four inputs a, b, c, d using a 2-bit selector s_1s_0 , from the 2-way Mux. How many NANDs does it use, and what's its depth?

SOLUTION

Use s_0 to choose within each pair, then s_1 to choose between the pairs: $ab = \text{Mux}(a, b, s_0)$, $cd = \text{Mux}(c, d, s_0)$, $\text{out} = \text{Mux}(ab, cd, s_1)$. That's 3 Muxes, so 12 NANDs. A Mux is 2 gates deep from its data inputs and 3 from its selector, which first goes through a Not. The selector s_0 reaches the output through the first Mux's selector and then the last Mux's data input, so the depth is $3 + 2 = 5$. In general an n -way Mux built as a tree of 2-way Muxes uses $n - 1$ of them and has depth proportional to $\log n$. Nand2Tetris asks you to build Mux4Way16 and Mux8Way16 exactly this way, 16 bits at a time.

EXERCISE 3.2 Why Or needs three NANDs

Prove that $\text{Or}(a, b) = \text{Nand}(\text{Not } a, \text{Not } b)$. Can Or be built from fewer than three 2-input NANDs?

SOLUTION

By De Morgan, $\neg(\neg a \wedge \neg b) = a \vee b$, and $\text{Nand}(x, y)$ is $\neg(x \wedge y)$, so $\text{Nand}(\text{Not } a, \text{Not } b) = a \vee b$. For the lower bound, any circuit with one NAND computes a NAND of two of its inputs or constants, which isn't Or. With two NANDs, the output gate is a NAND whose inputs are a, b, or the output of the first gate. Checking the few possibilities shows none of them is Or. For example, $\text{Nand}(\text{Nand}(a, b), x)$ is 1 whenever a and b are both 1, regardless of x, and also whenever x is 0, so it can't match Or in every row. So three is the minimum. Proving minimality is usually much harder than this, and synthesis tools rely on heuristics rather than proofs.

EXERCISE 3.3 A comparator

Design a circuit that outputs 1 when two 4-bit numbers are equal. How many NANDs does your design use, and what's its depth?

SOLUTION

Two bits are equal exactly when their Xor is 0. So compute $d_i = \text{Xor}(a_i, b_i)$ for the four bit positions, 16 NANDs at depth 3, then output 1 if no d_i is 1: $\text{out} = \text{Not}(\text{Or}(\text{Or}(d_0, d_1), \text{Or}(d_2, d_3)))$. Two levels of Or form a balanced tree, 3 Ors for 9 NANDs adding depth 4, and the final Not adds 1. That's 26 NANDs at depth 8. You can save the

final Not by building the last Or as a Nor directly, which shows how much room there is between a correct circuit and a minimal one. Comparators like this one decide branches, as in chapter 4's ALU, whose zero flag zr tests whether a 16-bit result is all zeros.

CHECKLIST

- ☐ Can you build Not, And, Or, Xor and Mux from NAND, and count their gates?
- ☐ Can you explain why depth matters as much as size?
- ☐ Can you turn any truth table into a circuit, and explain why the result is rarely minimal?
- ☐ Can you build a wide gate and an n-way Mux as a tree?
- ☐ Have you built Nand2Tetris project 1?

GO DEEPER

COURSE Nand2Tetris project 1: Boolean logic Nisan & Schocken

INTERACTIVE The Nand Game: build a computer from NAND gates in the browser Olav Junker Kjær

BOOK The Elements of Computing Systems, 2nd edition Nisan & Schocken

Adders and the ALU

3 exercises

Arithmetic is just more gates. A full adder adds three bits, a chain of them adds whole words, and subtraction reuses the same chain through two's complement. The simple chain is slow, because each bit waits for the carry from the bit below it, and the fix is the most important idea in fast arithmetic: compute all the carries in parallel. The arithmetic logic unit then wraps an adder in a few multiplexers, so that six control bits choose among eighteen functions. That's the Hack ALU, the heart of the Nand2Tetris computer, built and checked in this chapter.

Adding bits

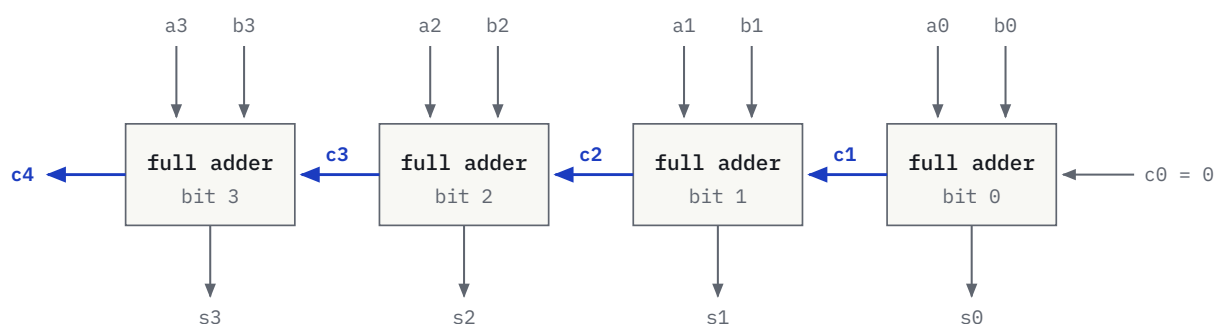
Adding two bits gives a sum bit and a carry: $1 + 1 = 10$ in binary, sum 0 carry 1. The sum is $a + b \bmod 2$, which is Xor, and the carry is And. That's a **half adder**. Adding a column of a long sum needs three inputs, the two bits plus the carry from the column to the right, and a **full adder** does that with two half adders and an Or:

$$s = a \oplus b \oplus c_{\text{in}}, \quad c_{\text{out}} = ab \vee (a \oplus b) c_{\text{in}}.$$

Built from this chapter's gates, a full adder takes 15 NANDs; a hand-optimised one takes 9.

The ripple-carry adder

To add two n -bit numbers, chain n full adders, feeding each one's carry out into the next one's carry in. This **ripple-carry adder** is exactly schoolbook addition, done in binary.



each sum waits for the carry from the bit to its right, so the delay grows with the width

A ripple-carry adder is four copies of the same full adder. It's small and simple, and its delay is the full length of the carry chain. In the worst case, adding 1 to $0111\dots1$, a carry must travel from the lowest bit all the way to the highest.

The example builds an 8-bit ripple adder from NAND gates and checks it on all 65,536 pairs of inputs. Its weakness is depth. Each full adder adds four gates to the carry path, so an 8-bit adder is 34 gates deep and a 64-bit adder 258. Chapter 5 shows that the clock can tick no faster than the slowest such path allows, so a 64-bit ripple adder would cap a processor at a few hundred megahertz.

Computing every carry at once

The way out is to notice when a column decides its carry on its own. A column **generates** a carry if both its bits are 1, $g_i = a_i b_i$, and **propagates** an incoming carry if exactly one bit is 1, $p_i = a_i \oplus b_i$. Then $c_{i+1} = g_i \vee p_i c_i$. Unrolling this shows that every carry is a function of the bits below it, and nothing forces the computation to go column by column:

$$c_4 = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \\ \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 p_1 p_0 c_0 .$$

The pairs (g, p) for adjacent spans of columns combine with an associative operation: a span generates a carry if its upper half generates one, or its upper half propagates and its lower half generates. Associativity means all the prefixes can be computed in a balanced tree, in $\log_2 n$ levels, the same trick as chapter 3's balanced Or. That's a **parallel-prefix adder**, here the Kogge–Stone design from 1973. The example's table shows the trade. At 64 bits the prefix adder is 28 gates deep against the ripple adder's 258, and it uses three times as many gates, 2,883 against 960. Real processors use prefix adders of various shapes that balance gate count, wiring and depth.

Subtraction for free

Chapter 1 showed that $-y$ is the bitwise inverse of y plus 1, so $x - y = x + \bar{y} + 1$. A subtractor is therefore an adder with a row of Xor gates that invert y on demand, and a carry-in of 1 that supplies the "+ 1". One circuit adds and subtracts, signed or unsigned. Comparisons come free too: $x < y$ exactly when $x - y$ is negative, ignoring overflow, so the sign bit of the difference answers the question.

The ALU

An **arithmetic logic unit** computes one of several functions of its inputs, chosen by control bits. The Hack ALU, designed for Nand2Tetris, does it with six bits applied in order:

- **zx** zeroes x , and **nx** then inverts it;
- **zy** zeroes y , and **ny** then inverts it;
- **f** chooses between $x + y$ and $x \& y$;
- **no** inverts the output.

Two status outputs report whether the result is zero (**zr**) and whether it's negative (**ng**). Those few operations produce eighteen useful functions, because inversion and addition combine in clever ways. With $\bar{v} = -v - 1$ in two's complement, setting **nx**, **f** and **no**

computes $x + y = -(-x - 1 + y) - 1 = x - y$. The example implements the ALU at the word level, runs all 18 control codes from the Nand2Tetris specification on 2,000 random pairs of 16-bit inputs, and checks every result against Python's arithmetic.

EXAMPLE

alu.py · a ripple-carry adder from NANDs, a parallel-prefix adder, and the Hack ALU's 18 functions

```
import random

# Signals are (value, depth), as in chapter 3, with a gate counter
count = 0
def nand(a, b):
    global count; count += 1
    return (1 - (a[0] & b[0]), max(a[1], b[1]) + 1)
def NOT(a): return nand(a, a)
def AND(a, b): return NOT(nand(a, b))
def OR(a, b): return nand(NOT(a), NOT(b))
def XOR(a, b):
    t = nand(a, b); return nand(nand(a, t), nand(b, t))

def half_adder(a, b): return XOR(a, b), AND(a, b) # sum, carry
def full_adder(a, b, c):
    s1, c1 = half_adder(a, b); s, c2 = half_adder(s1, c)
    return s, OR(c1, c2)

def bits(x, n): return [(x >> i) & 1, 0] for i in range(n)]
def value(sig): return sum(b[0] << i for i, b in enumerate(sig))

def ripple_add(a, b):
    carry, out = (0, 0), []
    for x, y in zip(a, b):
        s, carry = full_adder(x, y, carry); out.append(s)
    return out

# 1. An 8-bit ripple-carry adder, checked on all 65,536 input pairs
ok = all(value(ripple_add(bits(x, 8), bits(y, 8))) == (x + y) % 256 for x in range(256) for y
in range(256))
print(f"8-bit ripple-carry adder correct for all 65,536 input pairs: {ok}")

# 2. Ripple carry versus a parallel-prefix (Kogge-Stone) adder: gates and depth
def prefix_add(a, b):
    n = len(a)
    g = [AND(x, y) for x, y in zip(a, b)] # generate
    p = [XOR(x, y) for x, y in zip(a, b)] # propagate
    G, P, d = g[:], p[:], 1
    while d < n: # combine spans of
        length d
        G = [OR(G[i], AND(P[i], G[i - d])) if i >= d else G[i] for i in range(n)]
        P = [AND(P[i], P[i - d]) if i >= d else P[i] for i in range(n)]
        d *= 2
    return [p[0]] + [XOR(p[i], G[i - 1]) for i in range(1, n)]
print("\n bits    ripple: NANDs    depth    prefix: NANDs    depth")
for n in (8, 16, 32, 64):
    x, y = random.Random(n).getrandbits(n), random.Random(n + 1).getrandbits(n)
    count = 0; r = ripple_add(bits(x, n), bits(y, n)); rc, rd = count, max(s[1] for s in r)
    count = 0; q = prefix_add(bits(x, n), bits(y, n)); qc, qd = count, max(s[1] for s in q)
    assert value(r) == value(q) == (x + y) % 2 ** n
    print(f"{n:5} {rc:13} {rd:6} {qc:15} {qd:6}")

# 3. The Hack ALU: six control bits choose among 18 functions of two 16-bit inputs
```

```

MASK = 0xFFFF
def alu(x, y, zx, nx, zy, ny, f, no):
    if zx: x = 0
    if nx: x = ~x & MASK
    if zy: y = 0
    if ny: y = ~y & MASK
    out = (x + y) & MASK if f else x & y
    if no: out = ~out & MASK
    return out, int(out == 0), out >> 15 # out, zr, ng
def s16(v): return v - 0x10000 if v & 0x8000 else v
table = {"0": "101010", "1": "111111", "-1": "111010", "x": "001100", "y": "110000", "!x":
    "001101",
    "!y": "110001", "-x": "001111", "-y": "110011", "x+1": "011111", "y+1": "110111", "x-
    1": "001110",
    "y-1": "110010", "x+y": "000010", "x-y": "010011", "y-x": "000111", "x&y": "000000",
    "x|y": "010101"}
ref = {"0": lambda x, y: 0, "1": lambda x, y: 1, "-1": lambda x, y: -1, "x": lambda x, y: x,
    "y": lambda x, y: y,
    "!x": lambda x, y: ~x, "!y": lambda x, y: ~y, "-x": lambda x, y: -x, "-y": lambda x, y:
    -y,
    "x+1": lambda x, y: x + 1, "y+1": lambda x, y: y + 1, "x-1": lambda x, y: x - 1, "y-1":
    lambda x, y: y - 1,
    "x+y": lambda x, y: x + y, "x-y": lambda x, y: x - y, "y-x": lambda x, y: y - x,
    "x&y": lambda x, y: x & y, "x|y": lambda x, y: x | y}
rng = random.Random(4)
samples = [(rng.randrange(-32768, 32768), rng.randrange(-32768, 32768)) for _ in range(2000)]
good = 0
for name, code in table.items():
    ctl = [int(c) for c in code]
    good += all(s16(alu(x & MASK, y & MASK, *ctl)[0]) == s16(ref[name](x, y) & MASK) for x, y
        in samples)
print(f"\nHack ALU: {good} of {len(table)} functions correct on 2,000 random input pairs")
out, zr, ng = alu(5, 5 & MASK, *[int(c) for c in table["x-y"]])
print(f"x - y with x = y = 5: out = {out}, zr = {zr}, ng = {ng}; "
    f"y - x with x = 7, y = 5: out = {s16(alu(7, 5, *[int(c) for c in table['y-x']])[0])}, "
    f"ng = {alu(7, 5, *[int(c) for c in table['y-x']])[2]}")

```

OUTPUT

8-bit ripple-carry adder correct for all 65,536 input pairs: True

bits	ripple: NANDs	depth	prefix: NANDs	depth
8	120	34	195	16
16	240	66	499	20
32	480	130	1,219	24
64	960	258	2,883	28

Hack ALU: 18 of 18 functions correct on 2,000 random input pairs

x - y with x = y = 5: out = 0, zr = 1, ng = 0; y - x with x = 7, y = 5: out = -2, ng = 1

The status flags are how a computer decides anything. Every conditional branch in chapter 7 is a jump taken or not according to zr and ng, the results of an ALU operation such as $x - y$. Real ALUs add shifts, rotations and comparisons, and processors pair them with separate units for multiplication and division. Multiplying 64-bit integers takes a few cycles on a modern core, using trees of adders working in parallel. Division is iterative and takes tens of cycles, which is why compilers turn division by a constant into a multiplication and a shift whenever they can.

COMMON TRAPS

Forgetting the carry-in of 1 when subtracting by inverting.

Reading the carry out as signed overflow. Signed overflow is when the carries into and out of the sign bit differ (exercise 4.3).

Treating every arithmetic operation as equally cheap. Addition takes one cycle, multiplication a few, division tens.

Judging an adder by gate count alone. The carry chain decides its speed.

Hand-optimising multiplication by constants into shifts. Compilers already do it, and do it correctly.

EXERCISES

EXERCISE 4.1 Two more ALU codes

Show that the Hack control code 011111 ($zx = 0$, $nx = 1$, $zy = 1$, $ny = 1$, $f = 1$, $no = 1$) computes $x + 1$, and that 001110 computes $x - 1$.

SOLUTION

Use $\bar{v} = -v - 1$. For 011111: x is inverted to $-x - 1$; y is zeroed then inverted to $\bar{0} = -1$; $f = 1$ adds them, giving $-x - 2$; no inverts that, giving $-(-x - 2) - 1 = x + 1$. For 001110: x passes unchanged; y is zeroed and inverted to -1 ; $f = 1$ adds, giving $x - 1$; $no = 0$ leaves it. So the "constant 1" never needs its own circuit: it comes from inverting a zero and inverting again. The example confirms both codes, with the other sixteen, on random inputs.

EXERCISE 4.2 How deep is a lookahead carry?

Using the expansion of c_4 above, how deep is a 4-bit carry-lookahead unit built from 2-input gates, and why doesn't the same flat expansion work for 64 bits?

SOLUTION

The g and p signals take one gate level each, roughly, and each term of c_4 is an And of up to five signals and the result an Or of five terms. With 2-input gates, a 5-input And or Or is a balanced tree of depth 3, so c_4 is about $1 + 3 + 3 = 7$ gate levels, against $4 \times 4 = 16$ levels of carry path in a

4-bit ripple chain. Flattened for 64 bits, the last carry would be an Or of 65 terms, some of them Ands of 65 signals. The fan-in and the wiring explode, and the gate count grows quadratically. Prefix adders keep the logarithmic depth while reusing partial results, which is why they, rather than flat lookahead, are what wide adders use.

EXERCISE 4.3 The overflow flag

Show that a signed addition overflows exactly when the carry into the sign bit differs from the carry out of it. Check the rule on $100 + 50$ and on $-100 + -50$ in 8 bits.

SOLUTION

Let a and b be the sign bits and c the carry into the sign position. The sign of the result is $a \oplus b \oplus c$ and the carry out is the majority of a , b and c . If a and b differ, the result can't overflow, and the carry out equals c . If $a = b = 0$, overflow means the result's sign is 1, which happens exactly when $c = 1$, and the carry out is then 0, so they differ. If $a = b = 1$, overflow means the sign is 0, which happens exactly when $c = 0$, and the carry out is 1. So overflow happens exactly when $c \neq \text{carry out}$. ■ For $100 + 50 = 01100100 + 00110010$, there's a carry into bit 7 and none out: overflow. For $-100 + -50 = 10011100 + 11001110$, there's a carry out of bit 7 and none into it: overflow again, matching chapter 1. Processors compute this Xor as the V, or OF, flag after every addition.

CHECKLIST

- ☐ Can you build a half adder and a full adder, and chain them into a ripple-carry adder?
- ☐ Can you explain generate and propagate, and why prefix adders are logarithmic?
- ☐ Can you explain how one adder also subtracts?
- ☐ Can you work out what a Hack ALU control code computes?
- ☐ Have you built Nand2Tetris project 2?

GO DEEPER

COURSE Nand2Tetris project 2: Boolean arithmetic Nisan & Schocken

COURSE CS 61C: Great Ideas in Computer Architecture UC Berkeley

Flip-flops, registers and the clock

3 exercises

Combinational circuits forget their inputs the moment those inputs change. Memory needs feedback, a circuit whose output loops back into its own input. Two NAND gates wired to each other remember one bit. Add a clock, and you get a flip-flop that changes only at the instant the clock ticks, which turns the messy physics of gate delays into a simple rule: everything between two ticks has one clock period to settle. Registers, RAM and the program counter are flip-flops plus the multiplexers of chapter 3, and the slowest path between two flip-flops sets the processor's clock speed.

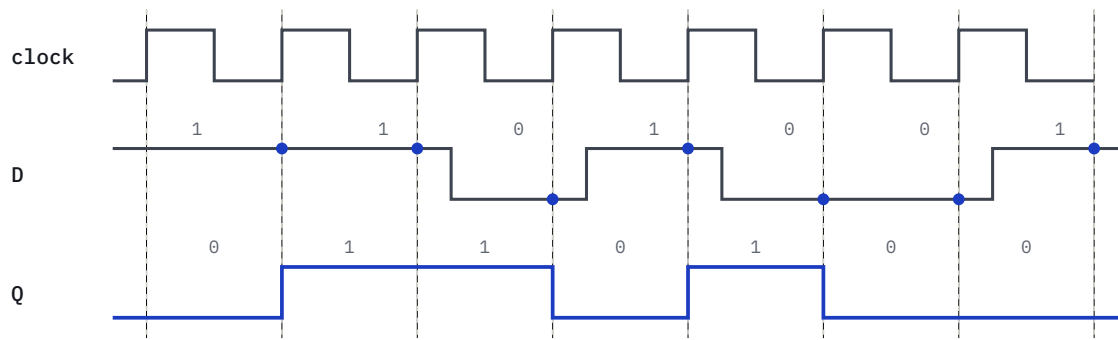
Feedback and the latch

Wire two NAND gates so that each one's output is one of the other's inputs, and leave one input on each free. Call the free inputs $\bar{S}$ and $\bar{R}$; they're active-low, meaning a 0 does something. Pulling $\bar{S}$ to 0 forces the output Q to 1, and the cross-connection then holds it there after $\bar{S}$ returns to 1. Pulling $\bar{R}$ to 0 forces Q to 0, which is also held. With both inputs at 1 the circuit just remembers its last state. That's an **SR latch**, one bit of memory made of two gates.

The forbidden input is both at 0, which forces both outputs to 1, breaking the rule that they're opposites. Releasing both at once creates a race, and the example's simulation, which updates both gates in lockstep, oscillates for ever. Real gates are never perfectly matched, so the latch eventually falls one way or the other, but it may hover between the two for an unpredictable time first, a condition called **metastability**. Designs avoid the forbidden input, and synchronise any signal arriving from outside the clock's control.

The clock and the D flip-flop

A latch changes whenever its inputs change, and signals in a real circuit change at different times as they ripple through gates of different depths. The solution is a **clock**, a signal that alternates between 0 and 1 at a fixed rate, and a **D flip-flop** that copies its input D to its output Q only at the clock's rising edge. Internally it's two latches in a row, master and slave, alternately enabled by the clock, so the input can never race straight through. Nand2Tetris treats the flip-flop as a primitive with a simple rule: $\text{out}(t) = \text{in}(t - 1)$, the output during each clock cycle is what the input was during the previous one.



Q changes only at rising edges, taking the value D held just before each one

D can change at any time during a cycle, as signals settle through the logic in front of it. The flip-flop samples it only at the rising edge, the dotted lines, so Q is D delayed by exactly one cycle, the same sequence the example prints.

For the sampling to work, D must be stable for a short **setup time** before the edge and a **hold time** after it. Meeting those two constraints everywhere on a chip is what timing analysis tools check, and violating them is how metastability gets in.

Registers, RAM and the program counter

A 1-bit **register** is a flip-flop with a Mux in front. When the load bit is 1 the Mux passes in the new value, and when it's 0 it feeds the flip-flop's own output back, so the bit is kept (exercise 5.1). Sixteen of them side by side make a 16-bit register. A **RAM** is many registers plus addressing. A DMux routes the load bit to the one register selected by the address, and a Mux selects that register's output. Nand2Tetris builds RAM8 from 8 registers, then RAM64 from 8 RAM8s, and so on up to the 16K-word RAM of the Hack computer, each level using three more address bits. Real memories aren't built from flip-flops. Static RAM, used for caches, takes six transistors per bit, and dynamic RAM, used for main memory, stores each bit as charge on a tiny capacitor with one transistor. That's dense and cheap, but the charge leaks away, so every row must be read and rewritten every few tens of milliseconds, and it's slow to read (chapter 9).

The **program counter** holds the address of the next instruction. On each cycle it does one of four things, in priority order: reset to 0, load a new address for a jump, increment to the next instruction, or hold. In the example it counts 0, 1, 2, 3, jumps to 100, counts on, and resets. That one register is the whole of a computer's control flow: chapter 7's CPU decides each cycle whether to increment it or load it.

Simulated

The example simulates the SR latch gate by gate, then models flip-flops, a register, an 8-word RAM and the program counter cycle by cycle, and finally turns chapter 4's adder depths into maximum clock rates.

EXAMPLE

sequential.py · a NAND latch, a flip-flop, a register, RAM8, the program counter and the clock limit

```
# 1. An SR latch: two NAND gates feeding each other. Inputs are active-low: s=0 sets, r=0
resets.
def nand(a, b): return 1 - (a & b)
def settle(s, r, q, qb):
    for step in range(10):
        # update both gates at once
        # until nothing changes
        nq, nqb = nand(s, qb), nand(r, q)
        if (nq, nqb) == (q, qb): return q, qb, step
        q, qb = nq, nqb
    return q, qb, None
# never settled
q, qb = 0, 1
print("SR latch made of two NANDs:")
for s, r, what in ((0, 1, "set"), (1, 1, "hold"), (1, 0, "reset"), (1, 1, "hold"), (0, 0, "both
low"), (1, 1, "release both")):
    q, qb, steps = settle(s, r, q, qb)
    print(f"    s={s} r={r} {what:13} -> Q={q} Q'={qb}" + ("    oscillates: no stable state" if
        steps is None else ""))

# 2. A clocked D flip-flop: its output is whatever its input was at the previous clock tick
class DFF:
    def __init__(self): self.state = 0
    def tick(self, d): out = self.state; self.state = d; return out
ff, ins = DFF(), [1, 1, 0, 1, 0, 0, 1]
outs = [ff.tick(d) for d in ins]
print(f"\nD flip-flop  in: {ins}\n                out: {outs} (one tick later)")

# 3. A register remembers until told to load; a RAM is registers plus address decoding
class Register:
    def __init__(self): self.ff = DFF(); self.value = 0
    def tick(self, x, load):
        self.value = x if load else self.value
        # a Mux in front of the flip-
        # flops
        return self.ff.tick(self.value)
reg = Register()
trace = [(x, load, reg.tick(x, load)) for x, load in ((7, 1), (9, 0), (9, 0), (4, 1), (5, 0),
(5, 0))]
print("\nregister (in, load) -> out: " + ", ".join(f"({x},{l})->{o}" for x, l, o in trace))
class RAM8:
    def __init__(self): self.regs = [0] * 8
    def access(self, x, addr, load):
        out = self.regs[addr]
        # Mux8Way16 selects one
        # register's output
        if load: self.regs[addr] = x
        # DMux8Way routes the load bit
        # to one register
        return out
ram = RAM8()
for addr, x in ((3, 42), (5, 7), (3, 43)): ram.access(x, addr, 1)
print(f"RAM8 after writing 42 to 3, 7 to 5, then 43 to 3: {ram.regs}")

# 4. The program counter: reset beats load beats increment
class PC:
    def __init__(self): self.value = 0
    def tick(self, x=0, load=0, inc=1, reset=0):
        out = self.value
        self.value = 0 if reset else x if load else (self.value + 1) & 0xFFFF if inc else
        self.value
        return out
pc = PC()
steps = [dict(), dict(), dict(), dict(x=100, load=1), dict(), dict(), dict(reset=1), dict()]
print(f"program counter: {[pc.tick(**s) for s in steps]} (jump to 100 after step 3, reset
```

```
after step 6)")
```

```
# 5. The clock can tick no faster than the slowest path between two registers
gate_ps = 15 # an illustrative delay per
gate, in picoseconds
for name, depth in (("64-bit ripple-carry adder", 258), ("64-bit prefix adder", 28)):
    period = depth * gate_ps
    print(f"{name:26}: {depth:3} gates deep -> {period / 1000:.2f} ns per cycle -> at most {1e6
        / period:.0f} MHz")
```

OUTPUT

SR latch made of two NANDs:

```
s=0 r=1 set      -> Q=1 Q'=0
s=1 r=1 hold     -> Q=1 Q'=0
s=1 r=0 reset    -> Q=0 Q'=1
s=1 r=1 hold     -> Q=0 Q'=1
s=0 r=0 both low -> Q=1 Q'=1
s=1 r=1 release both -> Q=1 Q'=1  oscillates: no stable state
```

```
D flip-flop  in:  [1, 1, 0, 1, 0, 0, 1]
               out: [0, 1, 1, 0, 1, 0, 0] (one tick later)
```

```
register (in, load) -> out: (7,1)->0, (9,0)->7, (9,0)->7, (4,1)->7, (5,0)->4, (5,0)->4
```

```
RAM8 after writing 42 to 3, 7 to 5, then 43 to 3: [0, 0, 0, 43, 0, 7, 0, 0]
```

```
program counter: [0, 1, 2, 3, 100, 101, 102, 0] (jump to 100 after step 3, reset after step
6)
```

```
64-bit ripple-carry adder : 258 gates deep -> 3.87 ns per cycle -> at most 258 MHz
```

```
64-bit prefix adder      : 28 gates deep -> 0.42 ns per cycle -> at most 2,381 MHz
```

How fast can the clock tick?

In a **synchronous** design every flip-flop updates at the same clock edge, and between edges the combinational logic computes the next values. Those values must settle before the next edge, so the clock period must exceed the delay of the slowest path from one flip-flop to the next, the **critical path**, plus the setup time. With an illustrative 15 picoseconds per gate, the 64-bit ripple adder's 258-gate path allows at most 258 MHz, while the prefix adder's 28 gates allow about 2.4 GHz. That's why chapter 4's carry trick matters. Designers shorten critical paths with faster circuits, and by splitting long paths into stages separated by registers, which is pipelining (chapter 12).

Clock rates rose for decades and then stopped near 4 to 5 GHz around 2005. The power a chip burns switching its transistors grows with the clock rate and with the square of the supply voltage. Voltages could no longer fall fast enough to compensate, and chips would have melted. Since then, more performance has come from doing more per cycle and from more cores (chapter 15), not from a faster clock.

COMMON TRAPS

Expecting a register to show a value in the same cycle it's written. It appears in the next cycle.

Creating feedback through combinational logic by accident, which makes a circuit oscillate

or latch unpredictably.

Feeding an unsynchronised external signal straight into clocked logic, inviting metastability.

Thinking of RAM as fast. Main memory is DRAM, and reading it takes hundreds of clock cycles.

Assuming clock speed is the measure of performance. Work per cycle matters as much.

EXERCISES

EXERCISE 5.1 A 1-bit register

Build a 1-bit register with inputs `in` and `load` and output `out`, from one D flip-flop and one Mux. Why must the flip-flop's output be wired back into the Mux?

SOLUTION

Feed the flip-flop's input from Mux(`a` = `out`, `b` = `in`, `sel` = `load`), and take out from the flip-flop. When `load` = 1, the Mux passes `in`, and the flip-flop stores it at the next edge. When `load` = 0, the Mux passes the flip-flop's own current output, so the next edge stores the same value again. A flip-flop has no notion of "keep", since it copies its input at every edge, and the feedback through the Mux is what makes keeping possible. In Nand2Tetris terms, $out(t) = load(t - 1) ? in(t - 1) : out(t - 1)$, exactly the register trace the example prints.

EXERCISE 5.2 The size of the Hack RAM

The Hack computer's data memory, RAM16K, holds 16,384 words of 16 bits, built as `RAM8` → `RAM64` → `RAM512` → `RAM4K` → `RAM16K`. How many flip-flops does it contain, how many address bits does each level use, and how does an address split across the levels?

SOLUTION

$16,384 \times 16 = 262,144$ flip-flops. Each of the first four levels multiplies the size by 8, using 3 more address bits: `RAM8` needs 3 bits, `RAM64` 6, `RAM512` 9, `RAM4K` 12. `RAM16K` is four `RAM4K`s rather than eight, so it adds 2 bits, for 14 in total ($2^{14} = 16,384$). In a 14-bit address, the top 2 bits choose a `RAM4K`, the next 3 a `RAM512` inside it, and so on down to the lowest 3 bits, which choose a register within a `RAM8`. Each level is a

DMux on the load bit and a Mux on the outputs, driven by its share of the address. Real memory chips decode addresses the same way, in rows and columns.

EXERCISE 5.3 Splitting the critical path

A circuit's critical path is 258 gates long, at 15 ps per gate. If you split it into four equal stages with registers between them, and each register adds the equivalent of 3 gate delays, what clock rate becomes possible? Why isn't the speed-up exactly 4?

SOLUTION

Each stage has $258 / 4 = 64.5$ gates of logic plus 3 of register overhead, 67.5 gate delays, or about 1.01 ns, so the clock can run at about 988 MHz instead of 258 MHz. That's 3.8 times faster, not 4, because the register overhead is paid once per stage, and it grows relative to the logic as the stages get shorter. Splitting into 100 stages would leave 2.6 gates of logic and 3 of overhead per stage, so the overhead would dominate. This is one reason pipelines stopped getting deeper, which chapter 12 takes up. Another is that a result needed by the next instruction then takes more stages to come back.

CHECKLIST

- ☐ Can you explain how an SR latch remembers, and what its forbidden input does?
- ☐ Can you explain why a clocked flip-flop makes timing manageable?
- ☐ Can you build a register, a RAM and a program counter from flip-flops and multiplexers?
- ☐ Can you compute a maximum clock rate from a critical path?
- ☐ Have you built Nand2Tetris project 3?

GO DEEPER

COURSE Nand2Tetris project 3: memory Nisan & Schocken

VIDEO Build an 8-bit computer from scratch Ben Eater

COURSE CS 61C: state, timing and synchronous digital systems UC Berkeley

Instruction sets and machine code

3 exercises

The instruction set architecture is the contract between hardware and software. It fixes the registers, the instructions and their binary encodings, and exactly what each instruction does. A program compiled for an instruction set runs on every processor that implements it, however different those processors are inside. This chapter compares the two main design philosophies, RISC and CISC, encodes and decodes real RISC-V instructions bit by bit, and encodes the two instruction types of the Hack computer that Nand2Tetris builds.

What an instruction set promises

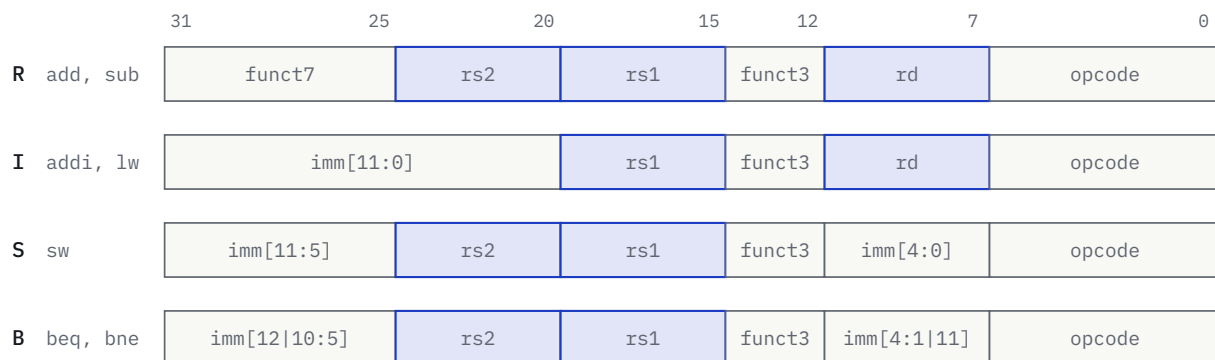
An **instruction set architecture (ISA)** specifies the programmer-visible state, meaning the registers, the program counter and memory, and every instruction's effect on it. RISC-V, for example, has 32 integer registers of 32 or 64 bits, named x0 to x31, where x0 always reads as zero. Memory is a large array of bytes, stored little-endian. Everything else is **microarchitecture**: how many instructions run at once, how big the caches are, how branches are predicted. A cheap microcontroller and a server processor can implement the same ISA and run the same binary, one at a millionth of the other's speed. Three ISAs dominate today: x86-64 in most PCs and servers, ARM64 in phones and Apple's computers, and RISC-V, an open standard designed at Berkeley in 2010 that anyone can implement without a licence.

RISC and CISC

The x86 family is the classic **CISC**, complex instruction set computer. Its instructions vary from 1 to 15 bytes, and many can read memory, compute and write memory in a single instruction. `ret` is the single byte C3, `mov eax, 1` is 5 bytes, and loading a 64-bit constant takes 10. **RISC** designs such as ARM64 and RISC-V, reduced instruction set computers, use fixed-size instructions, 4 bytes in RISC-V, and a **load/store architecture**: only loads and stores touch memory, and arithmetic works only on registers. Fixed sizes make decoding simple, since the processor knows where every instruction starts, and that matters when it wants to decode several per cycle (chapter 14). The old trade-off has mostly dissolved. Modern x86 processors translate each instruction into RISC-like **micro-operations** internally, while RISC designs added compressed 2-byte forms to recover CISC's code density.

Encoding RISC-V

Every RV32I instruction is 32 bits, in one of six formats. The lowest 7 bits, the **opcode**, say which format applies, and the fields that name registers sit at the same positions in every format.



rs1, rs2 and rd never move, so the register file can be read before the instruction is fully decoded

Four of RISC-V's six formats. The opcode at the bottom selects the format, funct3 and funct7 select the operation within it, and immediates are spread over whatever bits the register fields leave free. The immediate's sign bit is always bit 31, so sign extension can start immediately.

Immediates are two's complement and **sign-extended** to the register width, so `addi x6, x0, -1` stores the 12-bit pattern 0xFFF and loads -1. Branch offsets count bytes from the branch itself and are always even, so the lowest bit isn't stored, which lets a 12-bit field reach ± 4 KiB. The B-format's odd-looking bit shuffle keeps each immediate bit at the same position as in the S-format, so the hardware needs fewer multiplexers to assemble it.

Encoded and decoded

The example encodes nine RISC-V instructions into machine code, shows each one's fields, and decodes the binary back into assembly. It then encodes some Hack instructions.

EXAMPLE

isa.py · RISC-V instructions assembled to machine code and back, and Hack's two instruction types

```
# A slice of RISC-V (RV32I): every instruction is 32 bits, in one of six formats
REG = {f"x{i}": i for i in range(32)}
def bits(v, hi, lo): return (v >> lo) & ((1 << (hi - lo + 1)) - 1)
def R(f7, rs2, rs1, f3, rd, op): return f7 << 25 | rs2 << 20 | rs1 << 15 | f3 << 12 | rd << 7 | op
def I(imm, rs1, f3, rd, op): return (imm & 0xFFF) << 20 | rs1 << 15 | f3 << 12 | rd << 7 | op
def S(imm, rs2, rs1, f3, op): return bits(imm, 11, 5) << 25 | rs2 << 20 | rs1 << 15 | f3 << 12 | bits(imm, 4, 0) << 7 | op
def B(imm, rs2, rs1, f3, op): return (bits(imm, 12, 12) << 31 | bits(imm, 10, 5) << 25 | rs2 << 20 | rs1 << 15 | f3 << 12 | bits(imm, 4, 1) << 8 | bits(imm, 11, 11) << 7 | op)
def U(imm, rd, op): return (imm & 0xFFFF) << 12 | rd << 7 | op

def assemble(line):
```

```

op, args = line.split(None, 1)
a = [x.strip() for x in args.replace("(", " ").replace(")", " ").split(",")]
if op == "add": return R(0, REG[a[2]], REG[a[1]], 0, REG[a[0]], 0x33)
if op == "sub": return R(0x20, REG[a[2]], REG[a[1]], 0, REG[a[0]], 0x33)
if op == "addi": return I(int(a[2]), REG[a[1]], 0, REG[a[0]], 0x13)
if op == "lw": return I(int(a[1]), REG[a[2]], 2, REG[a[0]], 0x03)
if op == "sw": return S(int(a[1]), REG[a[0]], REG[a[2]], 2, 0x23)
if op == "beq": return B(int(a[2]), REG[a[1]], REG[a[0]], 0, 0x63)
if op == "bne": return B(int(a[2]), REG[a[1]], REG[a[0]], 1, 0x63)
if op == "lui": return U(int(a[1], 0), REG[a[0]], 0x37)
raise ValueError(op)

def sext(v, width): return v - (1 << width) if v >> (width - 1) else v
def disassemble(w):
    op, rd, f3, rs1, rs2, f7 = bits(w, 6, 0), bits(w, 11, 7), bits(w, 14, 12), bits(w, 19, 15),
        bits(w, 24, 20), bits(w, 31, 25)
    if op == 0x33: return f"{'sub' if f7 == 0x20 else 'add'} x{rd}, x{rs1}, x{rs2}"
    if op == 0x13: return f"addi x{rd}, x{rs1}, {sext(bits(w, 31, 20), 12)}"
    if op == 0x03: return f"lw x{rd}, {sext(bits(w, 31, 20), 12)}(x{rs1})"
    if op == 0x23: return f"sw x{rs2}, {sext(f7 << 5 | rd, 12)}(x{rs1})"
    if op == 0x63:
        imm = bits(w, 31, 31) << 12 | bits(w, 7, 7) << 11 | bits(w, 30, 25) << 5 | bits(w, 11,
            8) << 1
        return f"{'bne' if f3 == 1 else 'beq'} x{rs1}, x{rs2}, {sext(imm, 13)}"
    if op == 0x37: return f"lui x{rd}, 0x{bits(w, 31, 12):x}"
    return "?"

program = ["addi x5, x0, 10", "addi x6, x0, -1", "add x7, x5, x6", "sub x8, x5, x6",
    "lw x9, 8(x2)", "sw x9, 12(x2)", "beq x5, x7, -8", "bne x5, x0, 16", "lui x10,
    0x12345"]
print(f"{'assembly':20} {'machine code':>12} {'binary':^39} decoded")
for line in program:
    w = assemble(line)
    b = f"{w:032b}"
    print(f"{'line':20} 0x{w:08X} {b[:7]} {b[7:12]} {b[12:17]} {b[17:20]} {b[20:25]} {b[25:]}"
        f"{'disassemble(w)}")
print(f"every instruction round-trips: {all(disassemble(assemble(l)) == l for l in program)}")

# The Hack computer (Nand2Tetris): 16-bit instructions of two kinds
COMP = {"0": "0101010", "1": "0111111", "D": "0001100", "A": "0110000", "M": "1110000",
    "D+A": "0000010", "D+M": "1000010", "D-A": "0010011", "M-1": "1110010"}
DEST = {"": "000", "M": "001", "D": "010", "A": "100", "AM": "101"}
JUMP = {"": "000", "JGT": "001", "JEQ": "010", "JMP": "111"}
def hack(line):
    if line.startswith("@"): return f"0{int(line[1:]):015b}" # A-instruction: load a
        constant
    dest, rest = line.split("=") if "=" in line else ("", line)
    comp, jump = rest.split(";") if ";" in rest else (rest, "")
    return "111" + COMP[comp] + DEST[dest] + JUMP[jump] # C-instruction:
        compute, store, jump
print()
for line in ("@21", "D=A", "@5", "D=D+A", "M=D", "D;JGT", "0;JMP"):
    print(f"Hack {line:7} -> {hack(line)}")

```

OUTPUT

assembly	machine code	binary	decoded
addi x5, x0, 10	0x00A00293	00000000 01010 00000 000 00101 0010011	addi x5, x0, 10
addi x6, x0, -1	0xFFFF00313	11111111 11111 00000 000 00110 0010011	addi x6, x0, -1
add x7, x5, x6	0x006283B3	00000000 00110 00101 000 00111 0110011	add x7, x5, x6
sub x8, x5, x6	0x40628433	01000000 00110 00101 000 01000 0110011	sub x8, x5, x6
lw x9, 8(x2)	0x00812483	00000000 01000 00010 010 01001 0000011	lw x9, 8(x2)
sw x9, 12(x2)	0x00912623	00000000 01001 00010 010 01100 0100011	sw x9, 12(x2)
beq x5, x7, -8	0xFE728CE3	11111111 00111 00101 000 11001 1100011	beq x5, x7, -8
bne x5, x0, 16	0x00029863	00000000 00000 00101 001 10000 1100011	bne x5, x0, 16
lui x10, 0x12345	0x12345537	0001001 00011 01000 101 01010 0110111	lui x10, 0x12345

```
every instruction round-trips: True
```

```
Hack @21      -> 0000000000010101
Hack D=A      -> 1110110000010000
Hack @5       -> 0000000000000101
Hack D=D+A    -> 1110000010010000
Hack M=D      -> 1110001100001000
Hack D;JGT    -> 1110001100000001
Hack 0;JMP    -> 1110101010000111
```

Read the binary column against the figure. `add x7, x5, x6` has funct7 0, rs2 = 6, rs1 = 5, funct3 0, rd = 7 and opcode 0110011, and `sub` differs only in funct7, 0100000. The ALU's subtract control comes straight from that one bit. `lui x10, 0x12345` loads a 20-bit immediate into the upper bits of a register, and an `addi` fills in the low 12, which is how RISC-V builds 32-bit constants (exercise 6.2).

The Hack instruction set

The Hack computer has 16-bit instructions of just two kinds. An **A-instruction**, a 0 followed by 15 bits, loads a constant into the A register, as in `@21`. A **C-instruction**, 111 followed by 13 bits, computes something with the ALU, stores it, and maybe jumps. The bits are the control signals: one bit chooses whether the ALU's second input is A or M, the memory word at address A; six bits are the ALU's zx, nx, zy, ny, f and no from chapter 4; three bits say which of A, D and M receive the result; and three bits say whether to jump if the result is negative, zero or positive. `D=D+A` is 1110000010010000, and `0;JMP`, an unconditional jump to the address in A, is 1110101010000111. Hack has no call or return instructions and no stack. Nand2Tetris adds those in software, with a virtual machine (chapter 8).

COMMON TRAPS

Confusing the ISA with performance. The same instructions run at very different speeds on different implementations.

Forgetting that immediates are sign-extended, so a 12-bit field holding 0xFFF means -1, not 4,095.

Computing branch offsets in instructions instead of bytes, or from the wrong starting address.

Building a 32-bit constant with lui and addi without correcting for the sign of the low part (exercise 6.2).

Decoding x86 at an arbitrary byte. With variable-length instructions, you must know where one starts.

EXERCISE 6.1 Decode by hand

Decode the RISC-V instruction 0x02A00513.

SOLUTION

In binary: 0000 0010 1010 0000 0000 0101 0001 0011. The low 7 bits, 0010011, are the opcode for register-immediate arithmetic, I-format. Bits 11–7 are 01010, $rd = x10$. Bits 14–12 are 000, funct3 for `addi`. Bits 19–15 are 00000, $rs1 = x0$. Bits 31–20 are 000000101010, the immediate 42. So it's `addi x10, x0, 42`, which assemblers write as the pseudo-instruction `li a0, 42`: $x10$ is $a0$ in the calling convention, and adding to the always-zero $x0$ loads a constant.

EXERCISE 6.2 Loading 0xDEADBEEF

Load the 32-bit constant 0xDEADBEEF into $x5$ using `lui` and `addi`. Why is `lui x5, 0xDEADB` followed by `addi x5, x5, 0xEEF` wrong?

SOLUTION

`addi` sign-extends its 12-bit immediate, and 0xEEF has its top bit set, so it means $0xEEF - 0x1000 = -273$, not +3823. The sequence above would give $0xDEADB000 - 273 = 0xDEADAEEF$. Compensate by adding 1 to the upper part: `lui x5, 0xDEADC` then `addi x5, x5, -273` gives $0xDEADC000 - 273 = 0xDEADBEEF$. Assemblers do this automatically for the `li` pseudo-instruction, which is one reason to let them.

EXERCISE 6.3 Why the fields don't move

RISC-V keeps $rs1$, $rs2$ and rd at the same bit positions in every format, and the immediate's sign bit always at bit 31. What does that buy a hardware designer?

SOLUTION

Speed and simplicity in decoding. Because $rs1$ and $rs2$ are always in the same place, the processor can start reading both registers from the register file in the same cycle the instruction arrives, before it has worked out which format the instruction is. If the instruction turns out not to use them, the values are simply ignored. Fixed positions also mean fewer multiplexers in front of the register file. Keeping the sign bit at bit 31 means the sign extension of any immediate can begin immediately, in parallel with decoding, instead of waiting to find out

where the immediate's top bit is. Both decisions shorten the critical path of chapter 5, which helps the clock rate, at the cost of immediates whose bits look scrambled to humans.

CHECKLIST

- ☐ Can you separate what an ISA specifies from what the microarchitecture decides?
- ☐ Can you explain the RISC and CISC approaches, and why the difference matters less than it did?
- ☐ Can you encode and decode an R-, I-, S- and B-format RISC-V instruction?
- ☐ Can you explain sign extension of immediates, and build a 32-bit constant?
- ☐ Can you read Hack A- and C-instructions? Have you done Nand2Tetris project 4?

GO DEEPER

REFERENCE [The RISC-V instruction set manual](#) [RISC-V International](#)

COURSE [Nand2Tetris project 4: machine language](#) [Nisan & Schocken](#)

COURSE [CS 61C: RISC-V instruction formats](#) [UC Berkeley](#)

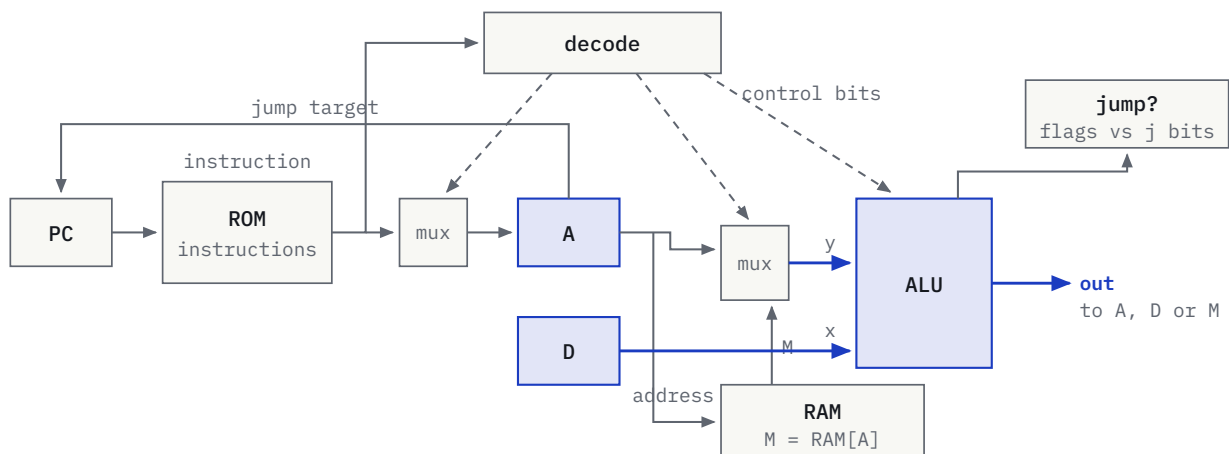
The CPU: fetch, decode, execute

3 exercises

A CPU is a loop. Fetch the instruction the program counter points to, decode its bits into control signals, execute it on the datapath of registers, ALU and memory, and decide where the next instruction is. Everything built so far fits together here: the ALU of chapter 4, the registers and program counter of chapter 5, and the instruction encoding of chapter 6. This chapter assembles the Hack CPU's datapath, writes a two-pass assembler, and runs real programs on an emulator cycle by cycle, counting exactly what they cost.

The datapath and the control unit

A processor has two parts. The **datapath** holds and transforms data: registers, the ALU, memory, and multiplexers that route values between them. The **control unit** decodes each instruction into the signals that drive the datapath: which inputs the multiplexers select, what the ALU computes, which registers load a new value, and whether the program counter jumps. In most processors the control unit is a substantial circuit. In Hack it's almost nothing, because the C-instruction's bits already are the control signals.



the PC loads A when the jump bits match the ALU's flags, and otherwise adds 1

The Hack CPU. An A-instruction sends its constant through the multiplexer into A. A C-instruction's bits choose A or M as the ALU's y input, set the ALU function, choose where the output goes, and decide the jump. The whole loop takes one clock cycle per instruction.

Fetch, decode, execute

Each clock cycle, the Hack CPU does all of the following:

- **Fetch:** read the instruction at ROM[PC].
- **Decode:** if the top bit is 0, it's an A-instruction; otherwise split it into the a bit, the six ALU control bits, three destination bits and three jump bits.
- **Execute:** for an A-instruction, load the constant into A. For a C-instruction, run the ALU on D and either A or M, and store the result in whichever of A, D and M the destination bits name.
- **Next:** if the jump bits match the ALU's zero and negative flags, load PC from A; otherwise increment it.

Hack is a **single-cycle** design: one instruction per cycle, and the clock period long enough for the slowest instruction to finish. It's also a **Harvard architecture**, with separate memories for instructions and data, so fetching and loading never compete. Most computers are **von Neumann** machines, with one memory holding both, which is what lets a program load another program, or a compiler write machine code. Modern processors are von Neumann outside and split again inside, with separate level-1 caches for instructions and data. Real CPUs also don't finish one instruction before starting the next. They overlap many, which Part IV explains, but the logical cycle stays the same.

An assembler in two passes

Programs are written in assembly, with names instead of numbers, and an **assembler** translates them into machine code. The difficulty is forward references: `@END` refers to a label defined later. The standard solution is two passes. The first pass walks the program counting instructions and records the ROM address of every label, which don't take up ROM themselves. The second pass translates each instruction, replacing labels with their addresses and giving every other new symbol, a variable, the next free RAM address from 16 upward. The example's assembler does exactly this in about thirty lines, which is Nand2Tetris project 6.

Programs, run

The example assembles a program that adds the numbers 1 to 100, shows its first machine words and the first eight cycles of execution, and runs it to completion. It then runs a multiplication program, which multiplies by repeated addition because Hack has no multiply instruction, on three pairs of inputs.

EXAMPLE

cpu.py · a two-pass Hack assembler and a cycle-by-cycle Hack CPU running two programs

```
# A Hack assembler and a cycle-by-cycle Hack CPU: fetch, decode, execute
COMP = {"0": "0101010", "1": "0111111", "-1": "0111010", "D": "0001100", "A": "0110000", "!D":
    "0001101",
    "!A": "0110001", "-D": "0001111", "-A": "0110011", "D+1": "0011111", "A+1": "0110111",
    "D-1": "0001110",
    "A-1": "0110010", "D+A": "0000010", "D-A": "0010011", "A-D": "0000111", "D&A":
    "0000000", "D|A": "0010101",
    "M": "1110000", "!M": "1110001", "-M": "1110011", "M+1": "1110111", "M-1": "1110010",
    "D+M": "1000010",
    "D-M": "1010011", "M-D": "1000111", "D&M": "1000000", "D|M": "1010101"}
```

```

DEST = {"": 0, "M": 1, "D": 2, "MD": 3, "A": 4, "AM": 5, "AD": 6, "AMD": 7}
JUMP = {"": 0, "JGT": 1, "JEQ": 2, "JGE": 3, "JLT": 4, "JNE": 5, "JLE": 6, "JMP": 7}

def assemble(source):
    lines = [l.split("//")[0].strip() for l in source.splitlines()]
    lines = [l for l in lines if l]
    symbols = {f"R{i}": i for i in range(16)} | {"SCREEN": 16384, "KBD": 24576}
    pc = 0
    for l in lines:
        addresses
        if l.startswith("("): symbols[l[1:-1]] = pc
        else: pc += 1
    code, next_var = [], 16
    for l in lines:
        variables from 16
        if l.startswith("("): continue
        if l.startswith("@"):
            s = l[1:]
            if not s.isdigit() and s not in symbols: symbols[s] = next_var; next_var += 1
            code.append(int(s) if s.isdigit() else symbols[s])
        else:
            dest, rest = l.split("=") if "=" in l else ("", l)
            comp, jump = rest.split(";") if ";" in rest else (rest, "")
            code.append(int("111" + COMP[comp] + f"{DEST[dest]:03b}" + f"{JUMP[jump]:03b}", 2))
    return code, symbols

def alu(x, y, zx, nx, zy, ny, f, no):
    # chapter 4's ALU
    if zx: x = 0
    if nx: x = ~x & 0xFFFF
    if zy: y = 0
    if ny: y = ~y & 0xFFFF
    out = (x + y) & 0xFFFF if f else x & y
    return ~out & 0xFFFF if no else out

def run(code, ram, max_cycles=100_000, trace=0):
    A = D = PC = cycles = 0
    while cycles < max_cycles:
        instr = code[PC]; cycles += 1
        if instr >> 15 == 0:
            # fetch
            # A-instruction: A = constant
            A, PC = instr, PC + 1
        else:
            # C-instruction: decode its
            fields
            a, c = (instr >> 12) & 1, [(instr >> (11 - i)) & 1 for i in range(6)]
            d, j = (instr >> 3) & 7, instr & 7
            out = alu(D, ram[A] if a else A, *c)
            # execute: the comp bits drive
            # the ALU directly
            neg, zero = out >> 15, out == 0
            jump = (j & 4 and neg) or (j & 2 and zero) or (j & 1 and not neg and not zero)
            if d & 1: ram[A] = out
            # write M before A changes
            if d & 2: D = out
            next_pc = A if jump else PC + 1
            if d & 4: A = out
            if jump and next_pc == PC - 1 and code[next_pc] == next_pc:
                return cycles
            # "@HALT, 0;JMP" loops for ever:
            # the program is done
            PC = next_pc
    if cycles <= trace:
        signed = D - 65536 if D & 0x8000 else D
        print(f"cycle {cycles:2}: next PC = {PC:2} A = {A:3} D = {signed:4}")
    return cycles

SUM = ""
@i
M=1 // i = 1
@sum
M=0 // sum = 0

```

```

(LLOOP)
    @i
    D=M
    @100
    D=D-A
    @END
    D;JGT          // if i > 100, stop
    @i
    D=M
    @sum
    M=D+M          // sum = sum + i
    @i
    M=M+1          // i = i + 1
    @LOOP
    0;JMP
(END)
    @sum
    D=M
    @R0
    M=D            // R0 = sum
(HALT)
    @HALT
    0;JMP
"""
code, symbols = assemble(SUM)
print(f"sum 1..100: {len(code)} instructions; symbols i -> RAM[{symbols['i']}], sum ->
      RAM[{symbols['sum']}], "
      f"LOOP -> ROM[{symbols['LOOP']}], END -> ROM[{symbols['END']}]"")
for addr in range(6): print(f"    ROM[{addr}] = {code[addr]:016b}")
print("first cycles:")
ram = [0] * 32768
cycles = run(code, ram, trace=8)
print(f"R0 = {ram[0]} after {cycles:,} cycles")

MULT = """
    @R2
    M=0            // R2 = 0
(LLOOP)
    @R1
    D=M
    @END
    D;JEQ          // while R1 != 0
    @R0
    D=M
    @R2
    M=D+M          // R2 = R2 + R0
    @R1
    M=M-1          // R1 = R1 - 1
    @LOOP
    0;JMP
(END)
    @END
    0;JMP
"""
mcode, _ = assemble(MULT)
for x, y in ((6, 7), (123, 45), (2, 1000)):
    ram = [0] * 32768; ram[0], ram[1] = x, y
    c = run(mcode, ram)
    print(f"multiply {x} x {y} by repeated addition: R2 = {ram[2]:,} in {c:,} cycles")

```

OUTPUT

```

sum 1..100: 24 instructions; symbols i -> RAM[16], sum -> RAM[17], LOOP -> ROM[4], END ->
      ROM[18]
ROM[0] = 00000000000010000
ROM[1] = 1110111111001000

```

```

ROM[2] = 0000000000010001
ROM[3] = 1110101010001000
ROM[4] = 0000000000010000
ROM[5] = 1111110000010000
first cycles:
cycle 1: next PC = 1   A = 16   D = 0
cycle 2: next PC = 2   A = 16   D = 0
cycle 3: next PC = 3   A = 17   D = 0
cycle 4: next PC = 4   A = 17   D = 0
cycle 5: next PC = 5   A = 16   D = 0
cycle 6: next PC = 6   A = 16   D = 1
cycle 7: next PC = 7   A = 100  D = 1
cycle 8: next PC = 8   A = 100  D = -99
R0 = 5050 after 1,416 cycles
multiply 6 x 7 by repeated addition: R2 = 42 in 92 cycles
multiply 123 x 45 by repeated addition: R2 = 5,535 in 548 cycles
multiply 2 x 1000 by repeated addition: R2 = 2,000 in 12,008 cycles

```

The trace shows the machine working at the lowest level. Cycles 1 to 4 set $i = 1$ and $\text{sum} = 0$. In cycle 5, `@i` puts i 's address, 16, into A; in cycle 6, `D=M` reads RAM[16] into D; and by cycle 8, D holds $i - 100 = -99$, ready for the jump test. Each loop iteration costs 14 instructions, so the program takes $14 \times 100 + 16 = 1,416$ cycles (exercise 7.2). The multiplication results show how much an algorithm depends on its inputs: 2×1000 takes 12,008 cycles, because the loop counts down the second operand. Swapping the operands would take 32.

Memory-mapped I/O

The Hack computer has no input or output instructions. Instead, part of the data memory is wired to devices. RAM[16384] to RAM[24575] are the 512×256 screen, one bit per pixel, so writing a word there turns 16 pixels on or off. RAM[24576] holds the code of the key currently pressed. Reading and writing devices with ordinary loads and stores is called **memory-mapped I/O**, and real computers use it extensively, as chapter 19 explains.

COMMON TRAPS

Using the new value of A in the same instruction that changes it. M and the jump target use the value A had at the start.

Counting labels as instructions. They mark addresses and occupy no ROM.

Assuming one instruction takes one cycle on real processors. Some take one, some dozens, and many overlap.

Forgetting to end a Hack program with an infinite loop. Otherwise the CPU runs on into whatever ROM holds next.

Ignoring the inputs' effect on running time, as with multiplication by repeated addition.

EXERCISE 7.1 A conditional in Hack

Write Hack assembly that sets RAM[2] to the larger of RAM[0] and RAM[1].

SOLUTION

Compute $\text{RAM}[0] - \text{RAM}[1]$ and jump on its sign: `@R0, D=M, @R1, D=D-M, @FIRST, D;JGT`, then `@R1, D=M, @STORE, 0;JMP`, then `(FIRST) @R0, D=M`, then `(STORE) @R2, M=D`, and finally `(HALT) @HALT, 0;JMP`. Every if/else compiles to this shape: compute a value, jump on a flag, and jump over the other branch. One subtlety: $D - M$ can overflow for very large values of opposite sign, which a careful version handles by comparing signs first.

EXERCISE 7.2 Counting cycles

Show that the sum program takes $14n + 16$ cycles to add up 1 to n , which is 1,416 for $n = 100$.

SOLUTION

The setup, `@i, M=1, @sum, M=0`, takes 4 cycles. Each of the n iterations runs all 14 instructions from `(LOOP)` to `0;JMP`. The final test, when $i = n + 1$, runs the first 6 instructions of the loop and jumps to END. The END block takes 4 cycles, and the halt loop's two instructions are counted once by the emulator before it stops. That's $4 + 14n + 6 + 4 + 2 = 14n + 16$. Counting like this is exactly the analysis of algorithms, at the level of machine instructions instead of abstract steps. Chapter 20's iron law of performance multiplies such counts by cycles per instruction and seconds per cycle.

EXERCISE 7.3 Faster multiplication

The multiplication program takes $12y + 8$ cycles to compute $x \times y$. How would you make it fast for all inputs, without hardware support?

SOLUTION

First, loop over the smaller operand, which needs a comparison at the start and costs at most $12 \min(x, y) + \text{a few cycles}$. Better, use **shift-and-add**, binary long multiplication: for each bit of y , if the bit is 1 add the current shifted copy of x to the result, then double x . With 16-bit numbers that's at most 16 iterations whatever the values, a few

hundred cycles instead of up to hundreds of thousands. Hack can double a number with `D=D+M` on itself, and test a bit by Anding with a mask held in memory. Hardware multipliers do the same thing in parallel, adding all the partial products at once in a tree of adders, which is why real multiplication takes a few cycles (chapter 4).

CHECKLIST

- ☐ Can you draw a datapath and say what the control unit drives in it?
- ☐ Can you describe one fetch–decode–execute cycle of the Hack CPU?
- ☐ Can you explain Harvard and von Neumann architectures?
- ☐ Can you write a two-pass assembler, and say why two passes are needed?
- ☐ Have you built Nand2Tetris projects 5 and 6?

GO DEEPER

COURSE Nand2Tetris projects 5 and 6: computer architecture and the assembler
Nisan & Schocken

BOOK Computer Systems: A Programmer's Perspective, chapter 4: processor
architecture Bryant & O'Hallaron

VIDEO Build an 8-bit computer from scratch: the control logic Ben Eater

Reading assembly, and stack machines

3 exercises

Compilers turn source code into assembly, and reading it is how you find out what a program really does: which values stay in registers, which go to memory, where the loops and calls are, and whether the compiler did what you hoped. This chapter reads and runs a RISC-V function, names the patterns every compiled program is made of, and then meets the other way to design an instruction set, the stack machine. Stack machines run Java, Python and WebAssembly, and the Nand2Tetris virtual machine is one.

Reading a function

The example's RISC-V program has a `main` that calls a function `sum(array, n)`. Reading it needs three pieces of vocabulary. The first is the **calling convention**, the agreement about registers that lets separately compiled functions call each other. In RISC-V, arguments arrive in `a0` to `a7` and the result returns in `a0`. The return address is in `ra`. The `t` registers are temporaries a function may overwrite, and the `s` registers must be preserved. The second is the shape of a **loop**: a test at the top that branches forward out of the loop, a body, and a jump backwards to the test. The third is **address arithmetic**. Element `i` of an array of 4-byte words lives at `base + 4i`, which the code computes with a shift, `slli t2, t1, 2`, because shifting left by 2 multiplies by 4.

Calls are two instructions. `jal ra, sum` jumps to `sum` and stores the address of the next instruction in `ra`, and `jalr zero, 0(ra)` jumps back to it. The example's final register dump shows `ra = 12`, the byte address of the instruction after the call, since `main`'s instructions sit at addresses 0, 4 and 8.

Patterns to recognise

Compiled code is built from a small set of shapes. Once you know them, a listing reads like a structured program:

- **if/else**: a compare and a branch forward over the "then" block, and a jump over the "else";
- **loops**: a backward branch, often with the test moved to the bottom so each iteration takes one branch;
- **function prologue and epilogue**: lowering `sp` to make room on the stack, saving `ra` and any `s` registers, and restoring them before returning (chapter 18);
- **switch**: a bounds check and an indirect jump through a table of addresses;

- **optimisations:** multiplications turned into shifts and adds, loop bodies duplicated (unrolling), and several elements processed per instruction (vectorisation, chapter 15).

Optimisation level changes everything. At `-O0` a compiler keeps every variable in memory and reloads it at each use, which is easy to follow and slow. At `-O2` variables live in registers, code is reordered and duplicated, and parts of the program disappear because the compiler proved them unnecessary. Read optimised code to learn about performance. Compiler Explorer, at godbolt.org, shows the assembly for any snippet in dozens of languages and compilers side by side, and `objdump -d` disassembles any binary on your own machine.

Stack machines

A register machine names its operands: `add t0, t0, t3` says exactly which registers to use. A **stack machine** names none. Its instructions take operands from the top of a stack and push results back. To compute $(a + b) \times c$ it pushes a and b , adds, which pops two values and pushes their sum, then pushes c and multiplies.

REGISTER MACHINE (RISC-V)

```
lw t0, a      t0 = a
lw t1, b      t1 = b
add t0, t0, t1 t0 = a + b
lw t1, c      t1 = c
mul t0, t0, t1 t0 = (a + b) c
```

every instruction names its registers

STACK MACHINE



the stack after each instruction; no register names at all

The same expression on a register machine and a stack machine. The stack code carries no register numbers at all, which makes it compact and trivial for a compiler to produce, at the price of more memory traffic unless the stack's top is kept in registers.

Stack code is compact and easy to generate: a compiler emits it by walking the expression tree in post-order, operands before operators. That's why most **virtual machines** use it, including the Java Virtual Machine, CPython's bytecode interpreter, WebAssembly, and the Nand2Tetris VM, whose 20-odd commands, push, pop, add, sub, neg, eq, gt, lt, and, or, not, label, goto, if-goto, function, call and return, are what the course's Jack compiler produces. A VM translator, Nand2Tetris projects 7 and 8, turns each VM command into a handful of Hack instructions, and a JIT compiler does the same for the JVM at run time, mapping the top of the stack onto real registers.

Three machines, one loop

The example runs the RISC-V function in a small interpreter, prints the bytecode that CPython compiles the same loop into, and runs a sum of 1 to 100 on a stack VM in the style of Nand2Tetris.

EXAMPLE

assembly.py · a RISC-V function run in an interpreter, CPython bytecode, and a Nand2Tetris-style stack VM

```
import dis, sys

# 1. A small RISC-V interpreter, enough to run a function that sums an array
ABI = ["zero", "ra", "sp", "gp", "tp", "t0", "t1", "t2", "s0", "s1"] + [f"a{i}" for i in
    range(8)] \
    + [f"s{i}" for i in range(2, 12)] + ["t3", "t4", "t5", "t6"]
REG = {name: i for i, name in enumerate(ABI)}
PROGRAM = """
main:   addi a0, zero, 256      # a0 = address of the array
        addi a1, zero, 5       # a1 = number of elements
        jal  ra, sum           # call sum(a0, a1); the return address goes in ra
        ebreak                 # stop
sum:    addi t0, zero, 0        # total = 0
        addi t1, zero, 0        # i = 0
loop:   bge t1, a1, done        # if i >= n, leave the loop
        slli t2, t1, 2          # t2 = i * 4, the byte offset of element i
        add  t2, a0, t2         # t2 = address of element i
        lw   t3, 0(t2)          # t3 = memory[t2]
        add  t0, t0, t3         # total += element
        addi t1, t1, 1          # i += 1
        jal  zero, loop        # jump back (the result address is discarded into zero)
done:   addi a0, t0, 0          # return value in a0
        jalr zero, 0(ra)        # return to the caller
"""

def parse(src):
    instrs, labels = [], {}
    for line in src.splitlines():
        line = line.split("#")[0].strip()
        if not line: continue
        if ":" in line:
            label, line = line.split(":", 1); labels[label.strip()] = len(instrs) * 4; line =
                line.strip()
        op, *args = line.replace(",", " ").replace("(", " ").replace(")", " ").split()
        instrs.append((op, args))
    return instrs, labels

def execute(instrs, labels, memory):
    x = [0] * 32; pc = 0; count = 0
    def val(a): return int(a) if a.lstrip("-").isdigit() else labels[a]
    while True:
        op, a = instrs[pc // 4]; count += 1; nxt = pc + 4
        if op == "addi": x[REG[a[0]]] = x[REG[a[1]]] + int(a[2])
        elif op == "add": x[REG[a[0]]] = x[REG[a[1]]] + x[REG[a[2]]]
        elif op == "slli": x[REG[a[0]]] = x[REG[a[1]]] << int(a[2])
        elif op == "lw": x[REG[a[0]]] = memory[x[REG[a[2]]] + int(a[1])]
        elif op == "bge": nxt = val(a[2]) if x[REG[a[0]]] >= x[REG[a[1]]] else nxt
        elif op == "jal": x[REG[a[0]]] = pc + 4; nxt = val(a[1])
        elif op == "jalr": x[REG[a[0]]] = pc + 4; nxt = x[REG[a[2]]] + int(a[1])
        elif op == "ebreak": return x, count
        x[0] = 0 # register zero is hard-wired to 0
        pc = nxt
    instrs, labels = parse(PROGRAM)
    memory = {256 + 4 * i: v for i, v in enumerate([3, 1, 4, 1, 5])}
    regs, count = execute(instrs, labels, memory)
```

```

print(f"RISC-V: sum of [3, 1, 4, 1, 5] = {regs[REG['a0']] after {count} instructions "
      f"(labels: {labels}); ra = {regs[REG['ra']]}, the address after the call")

# 2. The same function in Python compiles to bytecode for a stack machine
def total(values):
    t = 0
    for v in values:
        t += v
    return t
ops = [ins.opname for ins in dis.get_instructions(total)]
print(f"\nPython {sys.version_info.major}.{sys.version_info.minor} bytecode for the same loop
      ({len(ops)} instructions):")
print(" " + " ".join(ops))

# 3. A stack virtual machine in the style of Nand2Tetris: no registers, just push, pop and
    operate
VM = """
push constant 0
pop local 0          // sum = 0
push constant 1
pop local 1          // i = 1
label LOOP
push local 1
push constant 100
gt
if-goto END          // if i > 100, stop
push local 0
push local 1
add
pop local 0          // sum = sum + i
push local 1
push constant 1
add
pop local 1          // i = i + 1
goto LOOP
label END
push local 0
"""

def run_vm(src):
    prog = [l.split("//")[0].split() for l in src.strip().splitlines()]
    labels = {p[1]: i for i, p in enumerate(prog) if p[0] == "label"}
    stack, local, pc, steps = [], [0] * 8, 0, 0
    while pc < len(prog):
        p = prog[pc]; pc += 1; steps += 1
        if p[0] == "push": stack.append(int(p[2]) if p[1] == "constant" else local[int(p[2])])
        elif p[0] == "pop": local[int(p[2])] = stack.pop()
        elif p[0] in ("add", "gt"):
            b, a = stack.pop(), stack.pop(); stack.append(a + b if p[0] == "add" else -1 if a >
                b else 0)
        elif p[0] == "if-goto":
            if stack.pop(): pc = labels[p[1]]
        elif p[0] == "goto": pc = labels[p[1]]
    return stack[-1], steps
result, steps = run_vm(VM)
print(f"\nstack VM: sum 1..100 = {result} after {steps:,} VM commands")

```

OUTPUT

RISC-V: sum of [3, 1, 4, 1, 5] = 14 after 44 instructions (labels: {'main': 0, 'sum': 16, 'loop': 24, 'done': 52}); ra = 12, the address after the call

Python 3.14 bytecode for the same loop (15 instructions):

```

RESUME LOAD_SMALL_INT STORE_FAST LOAD_FAST_BORROW GET_ITER FOR_ITER STORE_FAST
LOAD_FAST_BORROW_LOAD_FAST_BORROW BINARY_OP STORE_FAST JUMP_BACKWARD END_FOR POP_ITER
LOAD_FAST_BORROW RETURN_VALUE

```

```
stack VM: sum 1..100 = 5050 after 1,411 VM commands
```

The CPython bytecode is a stack program. `LOAD_FAST` pushes a local variable, `BINARY_OP` pops two values and pushes the result, and `STORE_FAST` pops into a local. `FOR_ITER` and `JUMP_BACKWARD` are the loop's top and bottom. The exact instruction names change between Python versions, and this listing is from version 3.14. The stack VM takes 1,411 commands for the sum, where the hand-written Hack program of chapter 7 took 1,416 instructions. Translated into Hack, each VM command becomes several Hack instructions, so the VM version would run several times slower. That's the cost of a convenient abstraction, and recovering it is what optimising compilers and JITs are for.

COMMON TRAPS

Reading unoptimised code as representative of what runs in production.

Expecting assembly to mirror the source line by line. Optimisers reorder, merge and delete.

Mixing up AT&T and Intel syntax for x86, which put operands in opposite orders.

Depending on bytecode details, which change between interpreter versions.

Forgetting which registers a call may overwrite, the `t` and `a` registers in RISC-V.

EXERCISES

EXERCISE 8.1 Count the instructions

The example's program executed 44 instructions for an array of 5 elements. Account for every one, and give the formula for n elements.

SOLUTION

`main` runs 3 instructions before the call and `ebreak` after it. `sum` runs 2 setup instructions; each iteration runs 7 (`bge`, `slli`, `add`, `lw`, `add`, `addi`, `jal`); the final test that exits is 1 more; and the return takes 2. That's $3 + 1 + 2 + 7n + 1 + 2 = 7n + 9$, which is 44 for $n = 5$. The loop body dominates, so shaving one instruction from it matters more than anything else, which is what the next exercise does.

EXERCISE 8.2 Walk a pointer

Rewrite `sum` to step a pointer through the array instead of computing `base + 4i` on every iteration. How many instructions does it execute for 5 elements?

SOLUTION

Compute the end address once, `slli t2, a1, 2` then `add t2, a0, t2`, and loop: `bge a0, t2, done`; `lw t3, 0(a0)`; `add t0, t0, t3`; `addi a0, a0, 4`; `jal zero, loop`. The body drops from 7 instructions to 5, at the cost of 2 setup instructions, and `i` is no longer needed. For 5 elements that's $3 + 1 + 3 + 25 + 1 + 2 = 35$ instructions, and $5n + 10$ in general. Compilers perform this transformation, called **strength reduction**, automatically, which is why optimised loops rarely contain the multiplication you wrote.

EXERCISE 8.3 Translating the VM

In Nand2Tetris, the stack pointer lives in `RAM[0]` (called `SP`) and the stack grows upward from `RAM[256]`. Write the Hack instructions for the VM commands `push constant 7` and `add`.

SOLUTION

`push constant 7`: `@7, D=A (D = 7), @SP, A=M (A = address of the top), M=D (store 7 there), @SP, M=M+1 (bump SP)`. That's 7 instructions.
`add`: `@SP, AM=M-1 (pop: decrement SP and point at the old top), D=M (D = y), A=A-1 (point at x), M=D+M (replace x with x + y)`. That's 5 instructions, and `SP` correctly ends one lower. Every VM command becomes a fixed template like these, which is why a VM translator is a short program, and why VM code runs several times slower than hand-written assembly unless something optimises it.

CHECKLIST

- ☐ Can you follow a function's arguments, return value and return address through the calling convention?
- ☐ Can you recognise loops, conditionals and calls in compiled code?
- ☐ Do you know how optimisation changes what you'll see?
- ☐ Can you evaluate an expression on a stack machine, and explain why VMs use stacks?
- ☐ Have you built the Nand2Tetris VM translator, projects 7 and 8?

GO DEEPER

TOOL **Compiler Explorer** [Matt Godbolt](#)

COURSE **Nand2Tetris projects 7 and 8: the virtual machine** [Nisan & Schocken](#)

REFERENCE **dis: disassembler for Python bytecode** [Python documentation](#)

PART III

Memory

Three chapters on the Store, which now decides the speed of most programs more than the Mill does. First the hierarchy of memories and their real latencies, measured on real hardware; then how caches work and why they miss; and then how to arrange code and data so that they hit. The last of these is the stage's second deliverable.

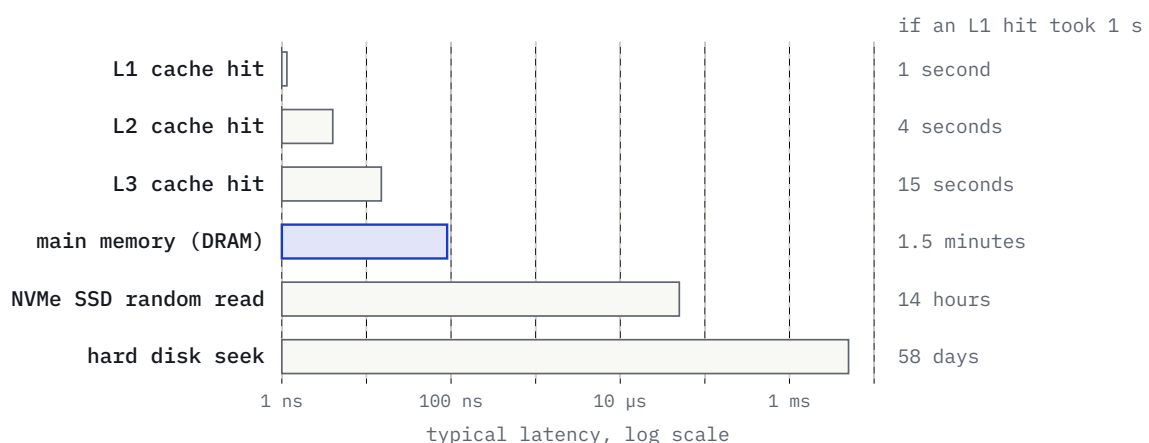
The memory hierarchy and its real latencies

3 exercises

Fast memory is small and expensive, and big memory is slow, so computers stack several kinds in a hierarchy. Registers and caches sit next to the processor, main memory sits further off, and storage further still, with each level roughly ten times larger and several times slower than the one above. The hierarchy works because programs have locality: they reuse what they used recently and touch what's next to it. This chapter gives the real numbers, measures the gap between cache and main memory on real hardware, and introduces the two quantities every performance argument needs, latency and bandwidth.

Why a hierarchy

Processor speed grew far faster than memory speed for decades. A modern core can execute several instructions per nanosecond, but a read from main memory, which is DRAM (chapter 5), takes around 100 nanoseconds. That's hundreds of instructions' worth of waiting, a gap often called the **memory wall**. Caches bridge it by keeping copies of recently used data in small, fast static RAM next to the core. A typical modern processor has a level-1 cache of 32 to 128 KiB per core, a level-2 cache of a few MiB, a level-3 cache of tens of MiB shared between cores, and then many gigabytes of DRAM.



Typical latencies, on a log scale; real values vary by a factor of two or so between machines. Scaled so that an L1 hit takes a second, a trip to main memory takes a minute and a half, and a disk seek takes two months. Each level hides the cost of the next only if most accesses never get that far.

Below memory comes storage. An NVMe solid-state drive answers a random read in tens of microseconds, and a spinning disk needs milliseconds to move its head. Networks add their own scale: a round trip inside a data centre takes about half a millisecond, and one

across a continent tens of milliseconds. Jeff Dean's list of "latency numbers every programmer should know" made these numbers famous, and their ratios matter more than their exact values, which keep changing.

Locality

The hierarchy works because real programs have **locality**. **Temporal locality** means that data used recently is likely to be used again soon, like a loop counter or the top of the stack. **Spatial locality** means that data near recently used data is likely to be used soon, like the next element of an array. Caches exploit both. They keep recently used data, and they move memory in fixed-size blocks called **cache lines**, 64 bytes on most x86 and ARM processors and 128 bytes on Apple's M-series chips, so one miss brings in the neighbours too. Hardware **prefetchers** watch the stream of misses, recognise sequential and strided patterns, and fetch lines before they're asked for. A program that walks memory in order gets almost every access from cache. A program that jumps around a large data set at random defeats all of it.

Measured on real hardware

The example performs the same work, 4 million reads of 8-byte integers, from arrays of growing size, once in sequential order and once in random order. The work per access is identical. Only where the data sits in the hierarchy changes. Unlike most examples in this volume it measures time, so its output depends on the machine: the run shown was recorded on an Apple M1 Pro laptop, and your numbers will differ.

EXAMPLE

hierarchy.py · the same reads in sequential and random order, from arrays of 16 KiB to 256 MiB

```
import random, time
from array import array

# Read 4 million elements from arrays of growing size, in order and in random order.
# The work per access is identical; only where the data sits in the memory hierarchy changes.
ACCESSSES = 1 << 22
random.seed(9)
def measure(n_elems, order):
    data = array("q", bytes(8 * n_elems))          # n 8-byte integers, all zero
    if order == "sequential":
        idx = [i % n_elems for i in range(ACCESSSES)]
    else:
        idx = random.choices(range(n_elems), k=ACCESSSES)
    best = float("inf")
    for _ in range(3):                             # keep the fastest of three runs
        t0 = time.perf_counter()
        s = 0
        for i in idx: s += data[i]
        best = min(best, time.perf_counter() - t0)
    return best / ACCESSSES * 1e9                   # nanoseconds per access

print(f'{array size':>12} {'sequential':>12} {'random':>10}    random / sequential")
for size_kib in (16, 256, 4096, 65536, 262144):
    n = size_kib * 1024 // 8
```

```
seq, rnd = measure(n, "sequential"), measure(n, "random")
label = f"{size_kib // 1024} MiB" if size_kib >= 1024 else f"{size_kib} KiB"
print(f"{label:>12} {seq:9.1f} ns {rnd:7.1f} ns {rnd / seq:5.2f}x")
```

OUTPUT

array size	sequential	random	random / sequential
16 KiB	19.0 ns	19.8 ns	1.04x
256 KiB	18.9 ns	19.3 ns	1.02x
4 MiB	19.3 ns	25.1 ns	1.30x
64 MiB	19.0 ns	71.4 ns	3.76x
256 MiB	18.9 ns	80.6 ns	4.26x

Python's interpreter adds about 19 ns to every access, which hides the difference between L1 and L2 entirely: at 16 KiB and 256 KiB both orders cost the same. At 4 MiB, random access starts to cost more, as the data spills out of the closest caches. At 64 MiB and 256 MiB the array no longer fits in any cache, and random access costs about 4 times as much as sequential, adding roughly 50 to 60 ns to each read, most of a trip to DRAM. Sequential access stays at 19 ns even for 256 MiB, because each 128-byte line serves 16 consecutive reads and the prefetcher fetches the next lines ahead of time. In a compiled language, without the interpreter's overhead, the same experiment shows a ratio of tens rather than 4.

Latency and bandwidth

Latency is how long one access takes. **Bandwidth** is how much data per second the memory system can deliver when many accesses are in flight. They're different limits. A laptop's DRAM delivers tens to hundreds of gigabytes per second, yet any single read still takes around 100 ns. **Little's law** connects them: the number of requests in flight equals throughput times latency. To use the bandwidth, the processor must keep many misses outstanding at once, which out-of-order cores do (chapter 14), and which a linked-list traversal prevents, because the address of the next node isn't known until the current load finishes. Workloads are accordingly **latency-bound**, waiting on one access after another, or **bandwidth-bound**, streaming more data than the memory can deliver. The cures differ, and chapter 20's roofline model tells them apart.

One more cache stands between a program and its data. Programs use virtual addresses, which the hardware translates to physical addresses through page tables kept by the operating system, and a small cache of recent translations, the **TLB**, makes that fast. Random access across hundreds of megabytes misses in the TLB as well as in the data caches, part of what the example's last rows measure. Stage 05 covers virtual memory in full.

COMMON TRAPS

Optimising arithmetic in a program that's waiting for memory. Find out which one it's bound by first.

Benchmarking with data that fits in cache and extrapolating to data that doesn't.

Treating big-O as the whole story. An $O(\log n)$ tree of scattered nodes can lose to an $O(n)$ scan of an array.

Pointer-heavy data structures in hot paths, where every hop is a potential trip to DRAM.

Mixing cold and warm measurements: the first run fills the caches, and later runs reuse them.

EXERCISES

EXERCISE 9.1 A list against an array

Estimate how long it takes to sum 100 million 8-byte integers stored in a contiguous array, and the same integers stored in a linked list whose nodes are scattered at random through memory. Assume about 100 ns per DRAM access and 20 GB/s of usable bandwidth.

SOLUTION

The array is 800 MB read sequentially. Prefetching keeps the memory system streaming, so it's bandwidth-bound: 800 MB at 20 GB/s is about 40 ms. The list is latency-bound. Each node's address comes from the previous node, so the loads can't overlap, and each one misses: $100 \text{ million} \times 100 \text{ ns} = 10 \text{ seconds}$. That's 250 times slower for the same values and the same number of additions. The list also needs extra memory for its pointers, which makes things worse. This is why performance-sensitive code prefers arrays, and why B-trees, which pack many keys into each node, beat binary trees on real hardware.

EXERCISE 9.2 How many misses in flight?

A memory system delivers 50 GB/s in 64-byte lines with a latency of 100 ns. How many cache misses must be outstanding at once to use that bandwidth?

SOLUTION

50 GB/s in 64-byte lines is about 781 million lines per second. By Little's law, requests in flight = throughput $\times$ latency = $781 \times 10^6 \times 100 \times 10^{-9} \approx 78$. One core can typically track a few dozen outstanding misses, so a

single thread can't saturate such a memory system, and several cores together can. It also explains the linked list of exercise 9.1: with one miss in flight at a time, it gets about 1/78 of the bandwidth.

EXERCISE 9.3 Reading the measurement

In the example, random access over 4 MiB costs only about 1.3 times as much as sequential access, but over 64 MiB it costs nearly 4 times as much. Explain the difference. What would you expect at 4 GiB?

SOLUTION

A 4 MiB array fits within the chip's larger caches, its level-2 cache and system-level cache, so random reads still hit in cache. They just hit in a slower level than the sequential reads, which are prefetched into L1. At 64 MiB the array is bigger than every cache, so most random reads go to DRAM, and the prefetchers can't predict them. At 4 GiB the data-cache behaviour is the same as at 256 MiB, all misses, but more accesses also miss in the TLB, because the pages touched no longer fit in it, so each read may need a page-table walk too. Expect it to be somewhat slower again. Using larger pages, which the operating system can provide, reduces that extra cost.

CHECKLIST

- ☐ Can you give the rough latency of each level of the hierarchy, and their ratios?
- ☐ Can you explain temporal and spatial locality, cache lines and prefetching?
- ☐ Can you tell a latency-bound workload from a bandwidth-bound one?
- ☐ Can you apply Little's law to memory requests?
- ☐ Have you measured the cache cliff on your own machine?

GO DEEPER

BOOK Computer Systems: A Programmer's Perspective, chapter 6: the memory hierarchy Bryant & O'Hallaron

PAPER What every programmer should know about memory Ulrich Drepper, 2007

REFERENCE Latency numbers every programmer should know after Jeff Dean and Peter Norvig

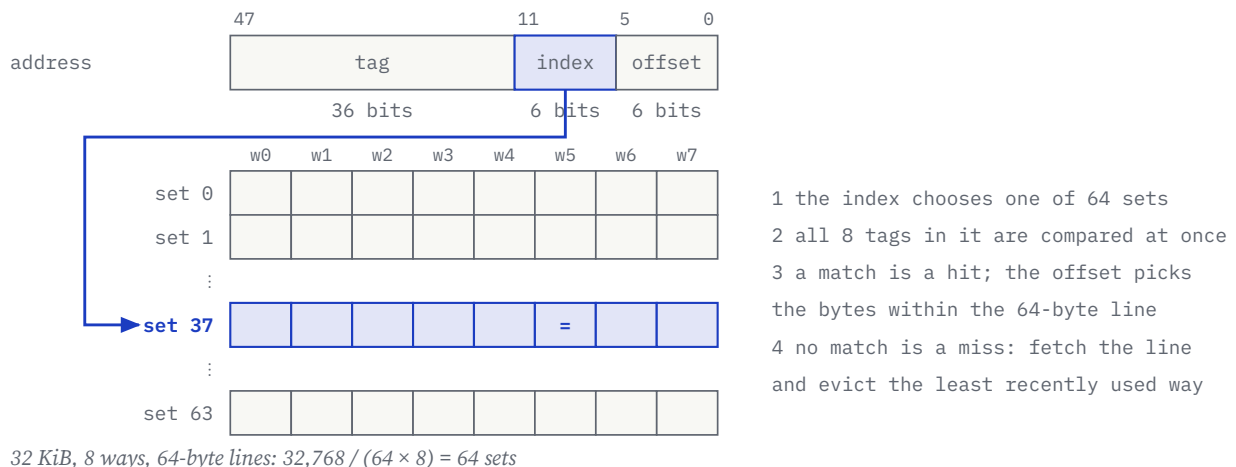
Caches: lines, sets, ways and misses

3 exercises

A cache keeps copies of recently used memory lines, and four design choices decide which access patterns it rewards: the line size, the number of sets, the number of ways in each set, and the rule for what to evict. Every miss falls into one of three kinds. Compulsory misses happen on first touch, capacity misses because the working set is too big, and conflict misses because too many lines compete for the same set. This chapter explains the organisation, then runs a cache simulator that sorts misses into these three kinds. It reproduces a classic trap, a matrix whose power-of-two row length makes a column walk thrash, along with its one-line fix.

Lines, sets and ways

A cache is divided into **lines** of, say, 64 bytes, and memory is moved in and out whole lines at a time. To find whether an address is cached, the hardware splits it into three fields. The lowest bits are the **offset** within the line, 6 bits for 64-byte lines. The next bits, the **index**, choose a **set**. The remaining high bits, the **tag**, identify which of the many memory lines that map to that set is actually there. A set holds several lines, its **ways**, and the hardware compares the tag against all of them at once.



A 32 KiB, 8-way cache with 64-byte lines has $32,768 / (64 \times 8) = 64$ sets. Each address can live in only one set, chosen by 6 of its bits, but in any of that set's 8 ways. A direct-mapped cache has one way per set; a fully associative cache has a single set with every line in it.

Associativity is a trade-off. A **direct-mapped** cache, with one way per set, is fast and simple but suffers when two frequently used addresses map to the same set. A **fully associative** cache, where any line can go anywhere, avoids that, but comparing every tag at once is too

expensive for anything but small structures such as TLBs. Level-1 caches are typically 8- to 12-way, and level-3 caches 12- to 16-way.

The three Cs

Mark Hill's classic classification sorts misses by cause, which tells you how to fix them:

- **Compulsory**, or cold, misses: the first access to a line can only miss. Prefetching hides them, and larger lines reduce their number.
- **Capacity** misses: the program's working set is bigger than the cache, so lines are evicted before they're reused. The fix is to use less data at a time, which is what chapter 11's blocking does.
- **Conflict** misses: the working set would fit, but too many of its lines map to the same set. The fix is to change the addresses, by padding data or changing strides, or to add associativity.

The simulator classifies each miss by also running a fully associative cache of the same size alongside. A miss that the fully associative cache would also have taken, on a line seen before, is capacity. A miss it would have avoided is conflict.

Replacement and the LRU cliff

When a set is full, a new line evicts one of the old ones. **Least recently used** (LRU) replacement evicts the line untouched for longest, a good bet under temporal locality. It has one pathological case: a loop over data slightly bigger than the cache. Each line is evicted just before the loop comes back for it, so the miss rate jumps from almost nothing to 100% as the array grows past the cache size. In the example, scanning a 32 KiB array repeatedly through a 32 KiB cache misses only on the first pass, 5%, while a 36 KiB array misses on every access. Real caches approximate LRU cheaply with pseudo-LRU bits, and recent designs use adaptive policies that resist such scans.

Writes

A **write-back** cache updates only its own copy and marks the line **dirty**, writing it to the next level when it's evicted. A **write-through** cache sends every write onward immediately. Almost all modern CPU caches are write-back and **write-allocate**, fetching a line on a write miss so it can be modified in place. With several cores, each with its own caches, a **coherence protocol** such as MESI keeps the copies consistent. A core that writes a line must first invalidate everyone else's copy. That's invisible to correct programs, but it costs time when two cores keep writing the same line, which is false sharing (chapter 11).

Simulated

The example's cache is set-associative with LRU replacement, 64-byte lines and configurable size and ways. It runs four experiments: a sequential scan, three ways of walking a 512×512 matrix, eight addresses that collide in one set, and repeated scans of arrays around the cache size.

EXAMPLE

caches.py · a set-associative LRU cache simulator that sorts misses into compulsory, capacity and conflict

```
from collections import OrderedDict

class Cache:
    """A set-associative cache with LRU replacement. It counts misses and sorts them into the 3
    Cs."""
    def __init__(self, size, line=64, ways=8):
        self.line, self.ways, self.sets = line, ways, size // (line * ways)
        self.data = [OrderedDict() for _ in range(self.sets)]
        self.full = OrderedDict(); self.capacity = size // line      # a fully associative
                               twin, same size
        self.seen = set()
        self.hits = self.compulsory = self.capacity_misses = self.conflict = 0
    def access(self, addr):
        block = addr // self.line
        s, tag = self.data[block % self.sets], block // self.sets
        in_full = block in self.full                                # would a perfect-placement
                               cache have hit?
        if in_full: self.full.move_to_end(block)
        else:
            self.full[block] = True
            if len(self.full) > self.capacity: self.full.popitem(last=False)
        if tag in s: s.move_to_end(tag); self.hits += 1; return
        if block not in self.seen: self.compulsory += 1              # first touch ever
        elif not in_full: self.capacity_misses += 1                  # even full associativity would
                               miss
        else: self.conflict += 1                                     # only placement caused it
        self.seen.add(block)
        s[tag] = True
        if len(s) > self.ways: s.popitem(last=False)                # evict the least recently used
                               line
    def report(self, name):
        n = self.hits + self.compulsory + self.capacity_misses + self.conflict
        m = n - self.hits
        print(f"{name:44} {n:>9} accesses miss rate {m / n:6.1%} "
              f"(compulsory {self.compulsory:}, capacity {self.capacity_misses:}, conflict
              {self.conflict:},)")

KB = 1024
# 1. A sequential scan: one miss per 64-byte line, 16 four-byte integers per line
c = Cache(32 * KB)
for i in range(64 * KB): c.access(4 * i)
c.report("scan 256 KiB of int32 once")

# 2. Row-major versus column-major traversal of a 512 x 512 int32 matrix (1 MiB)
N = 512
for order, pitch in (("row by row", 512), ("column by column", 512), ("column by column, rows
padded", 528)):
    c = Cache(32 * KB)
    for a in range(N):
        for b in range(N):
            i, j = (a, b) if order == "row by row" else (b, a)
            c.access(4 * (i * pitch + j))          # pitch: ints from one row's
                                                    start to the next
        c.report(f"512x512 matrix, {order}")

# 3. Conflict misses: 8 addresses exactly 32 KiB apart, touched over and over
for ways in (1, 8):
    c = Cache(32 * KB, ways=ways)
    for _ in range(1000):
        for k in range(8): c.access(k * 32 * KB)
```

```

c.report(f"8 addresses 32 KiB apart, {'direct-mapped' if ways == 1 else '8-way'}")

# 4. Working-set size: scan an array repeatedly; LRU falls off a cliff past the cache size
for size in (16, 32, 36, 64):
    c = Cache(32 * KB)
    for _ in range(20):
        for i in range(0, size * KB, 64): c.access(i)
    c.report(f"scan a {size} KiB array 20 times")

```

OUTPUT

scan 256 KiB of int32 once	65,536 accesses	miss rate	6.2%	(compulsory
4,096, capacity 0, conflict 0)				
512x512 matrix, row by row	262,144 accesses	miss rate	6.2%	(compulsory
16,384, capacity 0, conflict 0)				
512x512 matrix, column by column	262,144 accesses	miss rate	100.0%	(compulsory
16,384, capacity 0, conflict 245,760)				
512x512 matrix, column by column, rows padded	262,144 accesses	miss rate	6.2%	(compulsory
16,384, capacity 0, conflict 0)				
8 addresses 32 KiB apart, direct-mapped	8,000 accesses	miss rate	100.0%	(compulsory
8, capacity 0, conflict 7,992)				
8 addresses 32 KiB apart, 8-way	8,000 accesses	miss rate	0.1%	(compulsory
8, capacity 0, conflict 0)				
scan a 16 KiB array 20 times	5,120 accesses	miss rate	5.0%	(compulsory
256, capacity 0, conflict 0)				
scan a 32 KiB array 20 times	10,240 accesses	miss rate	5.0%	(compulsory
512, capacity 0, conflict 0)				
scan a 36 KiB array 20 times	11,520 accesses	miss rate	100.0%	(compulsory
576, capacity 10,944, conflict 0)				
scan a 64 KiB array 20 times	20,480 accesses	miss rate	100.0%	(compulsory
1,024, capacity 19,456, conflict 0)				

The sequential scan misses once per 64-byte line, 1 access in 16, and all of those misses are compulsory. Walking the matrix row by row does the same. Walking it column by column misses on every access, and the simulator calls these misses *conflict*, not capacity. The 512 lines a column walk touches would fit in the cache's 512 lines. But each row is $512 \times 4 = 2,048$ bytes, exactly 32 lines, so consecutive elements of a column are 32 lines apart, and with 64 sets they fall into only 2 of them. Sixteen slots for 512 lines means constant eviction. Padding each row to 528 integers makes the stride 33 lines, and since 33 shares no factor with 64, the column now spreads over all 64 sets and the miss rate drops back to 6.2%. The 8-way cache shrugs off eight addresses 32 KiB apart that a direct-mapped cache thrashes on, which is the case for associativity in one line.

COMMON TRAPS

Arrays whose rows, or strides, are large powers of two, which map many hot lines into the same few sets.

Loops over data just larger than the cache, where LRU replacement fails completely.

Assuming 64-byte lines everywhere. Apple's M-series chips use 128 bytes.

Benchmarking a data structure that fits in cache and expecting the same speed at full size.

Two threads writing neighbouring variables, which share a line and fight over it.

EXERCISES

EXERCISE 10.1 Split an address

A 48 KiB, 12-way cache has 64-byte lines and is addressed with 48-bit physical addresses. How many sets does it have, and how many bits are in the offset, index and tag?

SOLUTION

Lines: $48 \times 1024 / 64 = 768$. Sets: $768 / 12 = 64$. The offset needs 6 bits to address 64 bytes, the index 6 bits to choose among 64 sets, and the tag gets the remaining $48 - 12 = 36$ bits. That's the shape of many recent level-1 data caches. Making L1 bigger by adding ways rather than sets keeps the index within the lowest 12 bits, which are the same in the virtual and physical address with 4 KiB pages, so the cache can start looking up a set before address translation finishes.

EXERCISE 10.2 Why padding works

In the column walk, row i starts at line $32i$ with no padding and at line $33i$ with padding. Show which sets are used in each case for a cache with 64 sets, and explain why 33 works but 34 wouldn't work as well.

SOLUTION

Element (i, j) of a column lies in line $32i + c$ for a fixed c , so its set is $(32i + c) \bmod 64$, which takes only 2 values as i varies. With padding the set is $(33i + c) \bmod 64$, and since $\gcd(33, 64) = 1$, the multiples of 33 run through every residue mod 64 before repeating, so 64 consecutive rows land in 64 different sets and 512 rows fill each set with exactly 8 lines, matching its 8 ways. With a stride of 34 lines, $\gcd(34, 64) = 2$, so only the 32 even-numbered sets are used, 16 lines each, twice the associativity, and the cache thrashes again. The rule: choose a pitch whose number of lines is odd.

EXERCISE 10.3 Line size and a sequential sum

Summing an array of 8-byte integers in order, what's the miss rate with 64-byte lines, and with 128-byte lines? Why does a sequential sum still run near full speed when a quarter or more of its time would otherwise be spent on misses?

SOLUTION

A 64-byte line holds 8 integers, so 1 access in 8 misses, 12.5%. With 128-byte lines it's 1 in 16, 6.25%, all compulsory. If each miss cost a full trip to DRAM, the misses would dominate. The hardware prefetcher sees the sequential stream after a few misses and starts fetching lines ahead of the loop, so by the time an access arrives its line is already in cache, and the memory system works at its bandwidth instead of its latency. Compulsory misses are unavoidable, but predictable ones are free. That's why the example in chapter 9 found sequential access equally fast at 16 KiB and at 256 MiB.

CHECKLIST

- ☐ Can you split an address into tag, index and offset for a given cache?
- ☐ Can you explain direct-mapped, set-associative and fully associative caches?
- ☐ Can you sort a miss into compulsory, capacity or conflict, and name the fix for each?
- ☐ Can you explain the LRU cliff, and the power-of-two stride trap?
- ☐ Can you explain write-back caches and why coherence exists?

GO DEEPER

BOOK Computer Organization and Design, RISC-V edition, chapter 5 Patterson & Hennessy

COURSE CS 61C: caches UC Berkeley

PAPER What every programmer should know about memory, section 3 Ulrich Drepper

Cache-friendly code and data layout

3 exercises

The same computation can run ten times faster or slower depending only on the order in which it touches memory and how its data is laid out. The techniques are few and powerful. Walk memory in the order it's stored. Block a computation into tiles that fit in cache. Store the fields you scan together contiguously. Prefer arrays to pointers, and keep threads from writing to the same cache line. This chapter applies each one in the cache simulator. Blocking a matrix multiplication cuts its misses by a factor of 46, which is the stage's second deliverable in miniature: making a program several times faster through locality alone.

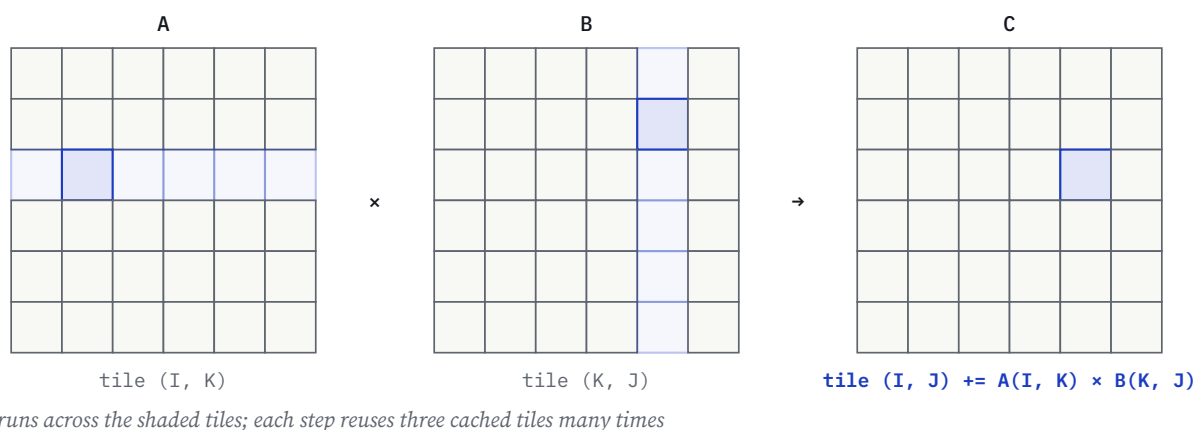
Walk memory in the order it's stored

C, C++, Rust, Go, NumPy's default and Python's lists of lists store two-dimensional arrays **row-major**: each row is contiguous, and moving along a row moves through memory one element at a time. Fortran, MATLAB, Julia and R store them **column-major**. A loop whose innermost index moves along the contiguous direction uses every byte of every cache line it fetches. A loop whose innermost index moves the other way strides through memory and may use 8 bytes of each 64-byte line. Fixing it is often a matter of swapping two loops, **loop interchange**, which compilers do automatically only when they can prove it's safe.

Matrix multiplication shows the effect. The textbook loop order, i-j-k, computes each C_{ij} as a dot product of row i of A with column j of B, so its innermost loop walks down a column of B. The order i-k-j instead adds A_{ik} times row k of B into row i of C, so its innermost loop walks along rows of both. Same arithmetic, same result, different access pattern.

Blocking

Even with the right loop order, a large matrix multiplication streams all of B through the cache once for every row of A, because B is too big to stay cached. **Blocking**, also called tiling, restructures the loops to work on small square tiles. Take a tile of A, a tile of B and a tile of C that fit in cache together, do all the work those three tiles allow, and only then move on.



One step of a blocked multiplication: the tile of C in block-row I and block-column J accumulates the product of a tile from A's block-row I and a tile from B's block-column J. With 16×16 tiles of doubles, each tile is 2 KiB, so all three stay in an 8 KiB cache while every element of each is used 16 times.

Inside a tile, every loaded line is reused many times before it's evicted. That's the principle behind every fast linear algebra library. BLAS implementations such as OpenBLAS and Intel's MKL block at several levels at once, for registers, L1, L2 and L3, and add SIMD instructions (chapter 15), which is how they reach close to the processor's peak while a naive triple loop reaches a few percent of it.

Data layout: arrays of structures and structures of arrays

A particle system might store each particle as a record with position, velocity, mass and charge, and keep an array of these records, an **array of structures** (AoS). A loop that updates every field of one particle at a time likes that layout. A loop that sums just one field over all particles fetches a 64-byte line for each 8-byte value it needs, wasting seven-eighths of the bandwidth. Storing each field in its own array, a **structure of arrays** (SoA), packs the values the loop needs together. In the example the scan drops from 100,000 misses to 12,500. Game engines built on entity-component systems, and the columnar databases described in *The Engineer's Ledger*, use the same idea at larger scale.

Pointer chasing

A linked list, a tree of small nodes, or a hash table whose buckets point to chains scatters its data wherever the allocator put it. Every step follows a pointer, the address of the next access isn't known until the current one completes, and the prefetcher can't help. In the example, traversing 100,000 values through a list whose nodes sit at random addresses misses 99,651 times, against 12,500 for an array of the same values. Under a simple model of 1 ns per hit and 100 ns per miss, that's about 100 ns per element against 13. Prefer contiguous containers: arrays and vectors, B-trees with wide nodes, hash tables with open addressing (*The Engineer's Workbench*, chapter 2), and nodes allocated together from an arena.

False sharing

Caches keep copies coherent a line at a time (chapter 10). If two threads on different cores write to *different* variables that happen to share a cache line, each write invalidates the other core's copy, and the line bounces between cores, every write costing a coherence round trip. That's **false sharing**: no data is shared, only the line. Per-thread counters stored next to each other in an array are the classic case. The fix is to pad or align each thread's data to its own line, 64 bytes on most machines and 128 on Apple's, using `alignas(64)` in C++ or Java's `@Contended`.

Simulated

The example uses chapter 10's cache to compare three versions of a 96×96 matrix multiplication, two layouts for a particle array, and a scattered linked list against an array. A small 8 KiB cache keeps the matrix simulation quick, standing in for a real 32 to 64 KiB level-1 cache facing proportionally larger matrices. It turns miss counts into an average access time with the simple model above.

EXAMPLE

layout.py · loop order, blocking, structure of arrays and pointer chasing, through the cache simulator

```
import random
from collections import OrderedDict

class Cache:
    # chapter 10's cache, without
    the 3C bookkeeping
    def __init__(self, size, line=64, ways=8):
        self.line, self.ways, self.sets = line, ways, size // (line * ways)
        self.data = [OrderedDict() for _ in range(self.sets)]; self.hits = self.misses = 0
    def access(self, addr):
        block = addr // self.line; s, tag = self.data[block % self.sets], block // self.sets
        if tag in s: s.move_to_end(tag); self.hits += 1
        else:
            self.misses += 1; s[tag] = True
            if len(s) > self.ways: s.popitem(last=False)
def amat_ns(c, hit_ns=1, miss_ns=100):
    # average memory access time
    model
    return (c.hits * hit_ns + c.misses * miss_ns) / (c.hits + c.misses)
KB = 1024

# 1. Matrix multiplication C = A B, n x n doubles: three loop orders and a blocked version
n, BS = 96, 16
A, B, C = 0, 8 * n * n, 16 * n * n
# base addresses of the three
matrices
def run(order):
    c = Cache(8 * KB)
    # a small cache keeps the
    simulation quick
    if order == "blocked":
        for ii in range(0, n, BS):
            for kk in range(0, n, BS):
                for jj in range(0, n, BS):
                    for i in range(ii, ii + BS):
                        for k in range(kk, kk + BS):
                            c.access(A + 8 * (i * n + k))
                        for j in range(jj, jj + BS):
```

```

        c.access(B + 8 * (k * n + j)); c.access(C + 8 * (i * n + j))
    return c
for x in range(n):
    for y in range(n):
        if order == "i-j-k":
            i, j = x, y; c.access(C + 8 * (i * n + j))
            for k in range(n): c.access(A + 8 * (i * n + k)); c.access(B + 8 * (k * n + j))
        else:
            rows of B and C
            i, k = x, y; c.access(A + 8 * (i * n + k))
            for j in range(n): c.access(B + 8 * (k * n + j)); c.access(C + 8 * (i * n + j))
    return c
print(f"multiplying two {n}x{n} matrices of doubles ({3 * 8 * n * n // KB} KiB in all) through
an 8 KiB cache:")
base = None
for order in ("i-j-k", "i-k-j", "blocked"):
    c = run(order); t = amat_ns(c)
    base = base or t
    print(f"    {order:8} {c.misses:>9}, misses of {c.hits + c.misses:,}    average access
    {t:5.2f} ns    speed-up {base / t:4.1f}x")

# 2. Array of structures versus structure of arrays: summing one field of 100,000 particles
count, fields = 100_000, 8
bytes
aos, soa = Cache(32 * KB), Cache(32 * KB)
for p in range(count):
    aos.access(64 * p + 0)
    bytes apart
    soa.access(8 * p)
    together
    # AoS: particle p's x field, 64
    # SoA: all x fields packed
print(f"\nsun one field of {count:,} particles: array of structs {aos.misses:,} misses, struct
of arrays {soa.misses:,}")

# 3. A linked list scattered through memory versus the same values in an array
random.seed(11)
nodes = 100_000
slots = random.sample(range(nodes * 4), nodes)
16-byte slot
ll, arr = Cache(32 * KB), Cache(32 * KB)
for k in range(nodes):
    ll.access(16 * slots[k])
    next node is
    arr.access(8 * k)
    bytes
    # follow the list: wherever the
    # walk the array: the next 8
print(f"traverse {nodes:,} values: scattered linked list {ll.misses:,} misses, array
{arr.misses:,}")
print(f"    with a 1 ns hit and a 100 ns miss: list {amat_ns(ll):.1f} ns per element, array
{amat_ns(arr):.1f} ns")

```

OUTPUT

```

multiplying two 96x96 matrices of doubles (216 KiB in all) through an 8 KiB cache:
i-j-k      923,520 misses of 1,778,688    average access 52.40 ns    speed-up  1.0x
i-k-j      112,896 misses of 1,778,688    average access  7.28 ns    speed-up  7.2x
blocked    20,022 misses of 1,824,768    average access  2.09 ns    speed-up 25.1x

sum one field of 100,000 particles: array of structs 100,000 misses, struct of arrays 12,500
traverse 100,000 values: scattered linked list 99,651 misses, array 12,500
    with a 1 ns hit and a 100 ns miss: list 99.7 ns per element, array 13.4 ns

```

Interchanging the loops cuts the misses from 923,520 to 112,896, and blocking cuts them to 20,022, 46 times fewer than the textbook order, with exactly the same arithmetic. The model's speed-ups, 7.2 and 25 times, overstate what real hardware shows, because real

processors overlap several misses at once and prefetch the regular streams, which helps the unblocked versions most. Measured speed-ups for blocking large matrix multiplications in compiled code are still typically several-fold, which is the size of the stage's goal. The method generalises. Measure where the misses are, find the loop whose accesses stride or scatter, and change the order or the layout until each fetched line is used fully, and used again before it's evicted.

COMMON TRAPS

Traversing in the wrong order for the language's layout, especially when porting between C and Fortran or MATLAB.

Choosing tiles that don't fit in cache together, or that are so small the loop overhead dominates.

Using AoS for data that's scanned one field at a time, or SoA for data that's always used whole.

Optimising without measuring misses. Use `perf stat -e cache-misses`, Instruments or VTune.

Per-thread counters packed side by side, the textbook false-sharing bug.

Hand-blocking matrix code that a BLAS library already does better. Call the library.

EXERCISES

EXERCISE 11.1 Choosing a tile size

For a blocked multiplication of doubles on a machine with a 48 KiB level-1 data cache, what's the largest square tile size b for which three tiles fit, and what would you actually choose?

SOLUTION

Three $b \times b$ tiles of 8-byte doubles take $3 \cdot 8b^2$ bytes, so $24b^2 \leq 49,152$, which gives $b \leq 45$. Choose somewhat less, such as 32. Other data also competes for the cache, including the loop's own variables, the stack and the code; set-associativity means tiles don't pack perfectly; and power-of-two tile sizes keep the loop arithmetic cheap. In practice the best size is found by measuring a few candidates, and libraries tune it per processor. Note the stride trap of chapter 10: if the matrix's row length is a large power of two, a tile's rows can collide in the same sets, and padding the matrix fixes it.

EXERCISE 11.2 Where i-k-j's misses come from

In the i-k-j order, the example counted 112,896 misses for $n = 96$. Show that streaming B accounts for almost all of them.

SOLUTION

For each of the 96 values of i , the k and j loops read all of B row by row: 96 rows of 96 doubles, 768 bytes or 12 lines each, so 1,152 lines. B is 72 KiB and the cache is 8 KiB, so nothing survives until the next i , and every line misses again: $96 \times 1,152 = 110,592$ misses. Rows of A and C are reused within each i and miss only once per i , adding $96 \times (12 + 12) = 2,304$. Together that's 112,896, exactly the simulator's count. Blocking cuts the dominant term by reusing each tile of B for 16 rows of A before evicting it.

EXERCISE 11.3 A cache-friendly transpose

Transposing a large matrix, $B[j][i] = A[i][j]$, reads A row by row but writes B column by column. Why is that slow, and how do you fix it?

SOLUTION

Every write to B lands in a different row, a full row length further along in memory, so each write touches a different cache line, and with large matrices each line is evicted before its neighbours are written. You can't make both the reads and the writes sequential, because a transpose inherently swaps the two directions. Blocking solves it. Transpose one $b \times b$ tile at a time, so the tile of A and the tile of B both fit in cache, and each line fetched is fully used: the reads cover b consecutive elements of each row of A, and the writes b consecutive elements of each row of B. Tiles of 8 to 32 elements on a side typically give several times the speed of the naive loop, and they're the same technique used for blocked matrix multiplication.

CHECKLIST

- ☐ Do you know whether your language stores arrays row-major or column-major?
- ☐ Can you block a loop nest, and choose a tile size?
- ☐ Can you choose between an array of structures and a structure of arrays?

- ☐ Can you explain why pointer chasing is slow, and name contiguous alternatives?
- ☐ Can you recognise and fix false sharing?

GO DEEPER

BOOK Computer Systems: A Programmer's Perspective, sections 6.5–6.6: cache-friendly code Bryant & O'Hallaron

ARTICLE Anatomy of high-performance matrix multiplication Goto & van de Geijn, 2008

ARTICLE Gallery of processor cache effects Igor Ostrovsky

PART IV

Parallelism in the core

Four chapters on how one processor core runs many instructions at once without changing what any program means. Pipelining overlaps the steps of successive instructions; branch prediction guesses which way the program will go so the pipeline never waits; out-of-order execution finds independent work and does it early; and SIMD and multiple cores apply the same idea to data and to threads.

Pipelining and hazards

3 exercises

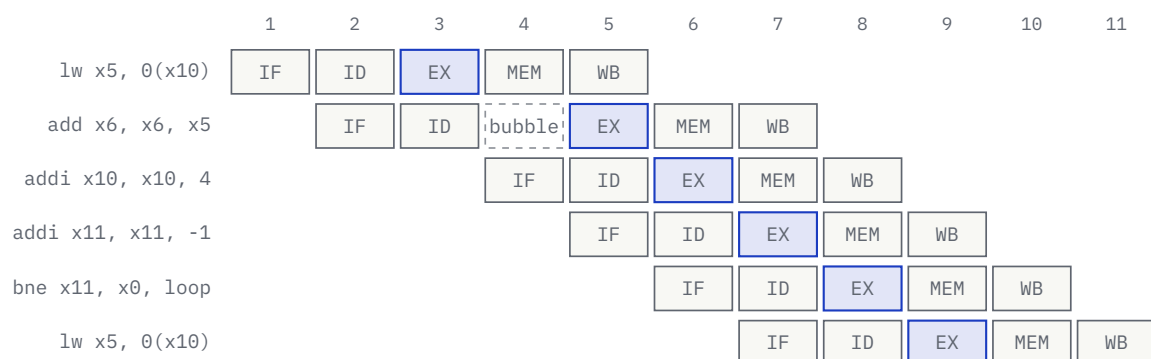
A laundry with a washer and a dryer doesn't wait for one load to dry before washing the next. Pipelining does the same with instructions. Split each one into stages, fetch, decode, execute, memory and write back, and work on five instructions at once, one in each stage. In the ideal case the processor finishes one instruction per cycle, even though each takes five. Hazards spoil the ideal: an instruction needs a result that isn't ready yet, or the pipeline doesn't know which instruction comes next. Forwarding, stalls, careful instruction order and branch prediction are how processors recover. This chapter simulates a five-stage pipeline and measures each one.

Five stages

The classic RISC pipeline has five stages, each with a register between it and the next:

- **IF**, instruction fetch, reads the instruction at PC;
- **ID**, instruction decode, decodes it and reads its source registers;
- **EX**, execute, runs the ALU, or computes an address for a load or store;
- **MEM** accesses data memory, for loads and stores;
- **WB**, write back, writes the result into the destination register.

Each stage does about a fifth of the work, so the clock can run roughly five times faster than a single-cycle design (chapter 5's exercise 5.3). Every instruction still takes five cycles from start to finish, its **latency**, but a new one can start every cycle, its **throughput**. The figure shows a loop's instructions overlapping in a staircase.



the add waits one cycle for the loaded value; every other cycle, one instruction enters each stage

One iteration of the example's loop, with forwarding and a correctly predicted branch. Five instructions are in flight at once. The add needs the value the load is still fetching from memory, so it starts a cycle late and leaves a bubble, the one cycle lost in this iteration.

Data hazards: forwarding and stalls

A **data hazard** occurs when an instruction needs a result that an earlier instruction hasn't written back yet. `add x6, x6, x5` needs x5 from the load just before it, but the load only writes x5 in its WB stage. The simplest fix is to **stall**: hold the dependent instruction in decode, inserting **bubbles**, until the value is in the register file. Without anything better, the example's loop runs at 2.2 cycles per instruction, CPI 2.20, instead of 1.

The better fix is **forwarding**, or bypassing. The value exists as soon as the ALU computes it, at the end of EX, so wires carry it straight from the pipeline register back to the ALU's input for the next instruction, without waiting for write back. With forwarding, ALU-to-ALU dependences cost nothing. Loads are the exception: a loaded value arrives only at the end of MEM, too late for an instruction in EX the very next cycle, so a **load-use** dependence still costs one bubble, the one in the figure. The fix for that is software. Move an independent instruction between the load and its use, as the example's reordered loop does, and the bubble disappears. Compilers schedule instructions this way automatically.

Control hazards

A branch decides which instruction comes next, but the pipeline must fetch the next instruction before the branch has been decided, which in this pipeline happens in EX, two stages after fetch. If the processor simply waits, or always fetches the next instruction in memory, every taken branch wastes two cycles. In the example's loop, where 1 instruction in 5 is a taken branch, that costs 0.4 cycles per instruction. The fix is to **predict** the branch and fetch from the predicted target immediately, discarding the wrong-path instructions if the guess was wrong. Chapter 13 shows how predictors get it right well over 90% of the time.

A third kind, **structural hazards**, happens when two instructions need the same hardware in the same cycle, for example a single memory port for both fetching instructions and loading data. Separate instruction and data caches and duplicated functional units remove most of them.

Simulated

The example schedules the loop on a five-stage pipeline under four assumptions, prints the pipeline diagram for two of them, and counts cycles over 1,000 iterations.

EXAMPLE

pipeline.py · a five-stage pipeline with and without forwarding, branch prediction and reordering

```
# A classic 5-stage pipeline: IF (fetch), ID (decode, read registers), EX (ALU), MEM (memory),  
WB (write back).  
# Each instruction: (text, destination register, source registers, kind)  
LOOP = [("lw   x5, 0(x10)", "x5", ["x10"], "load"),  
        ("add  x6, x6, x5", "x6", ["x6", "x5"], "alu"),  
        ("addi x10, x10, 4", "x10", ["x10"], "alu"),
```

```

        ("addi x11, x11, -1", "x11", ["x11"], "alu"),
        ("bne x11, x0, loop", None, ["x11", "x0"], "branch")]
REORDERED = [LOOP[0], LOOP[2], LOOP[3], LOOP[1], LOOP[4]] # the add moved away from the
load it needs

def schedule(program, iterations, forwarding, predict_taken):
    """Return the cycle of each instruction's EX stage, in order of execution."""
    last_writer, ex_cycles, prev_ex = {}, [], 2 # the first instruction is in
EX in cycle 3
    for it in range(iterations):
        for i, (text, dst, srcs, kind) in enumerate(program):
            ex = prev_ex + 1 # one instruction enters EX per
cycle at best
            for s in srcs:
                if s in last_writer:
                    p_ex, p_kind = last_writer[s]
                    if not forwarding: ex = max(ex, p_ex + 3) # wait for the producer's write-
back
                    elif p_kind == "load": ex = max(ex, p_ex + 2) # loaded value arrives
after MEM
                    else: ex = max(ex, p_ex + 1) # ALU result forwarded straight
to EX
            ex_cycles.append((it, i, ex))
            if dst: last_writer[dst] = (ex, kind)
            prev_ex = ex
            if kind == "branch" and not predict_taken and it < iterations - 1:
                prev_ex = ex + 2 # taken branch resolved in EX:
two fetches wasted
    return ex_cycles

def diagram(program, forwarding, predict_taken):
    sched = schedule(program, 2, forwarding, predict_taken)[:len(program) + 1]
    width = sched[-1][2] + 3
    print(f"{'':20}" + "".join(f"{c:>4}" for c in range(1, width + 1)))
    for it, i, ex in sched:
        row = [" " * 4] * width
        for stage, offset in (("IF", -2), ("ID", -1), ("EX", 0), ("MEM", 1), ("WB", 2)):
            row[ex + offset - 1] = f"{stage:>4}"
        print(f"{program[i][0]:20}" + "".join(row))

def cycles(program, n, forwarding, predict_taken):
    return schedule(program, n, forwarding, predict_taken)[-1][2] + 2 # the last
instruction's WB

print("one iteration plus the next fetch, no forwarding, branches not predicted:")
diagram(LOOP, False, False)
print("\nwith forwarding and a correct branch prediction:")
diagram(LOOP, True, True)
n = 1000
print(f"\nn:{n}, iterations, {5 * n}, instructions:")
for name, prog, fwd, pred in (("no forwarding, no prediction", LOOP, False, False),
                              ("forwarding, no prediction", LOOP, True, False),
                              ("forwarding, branch predicted", LOOP, True, True),
                              ("forwarding, predicted, reordered", REORDERED, True, True)):
    c = cycles(prog, n, fwd, pred)
    print(f"{name:34} {c:6} cycles CPI {c / (5 * n):.2f}")

```

OUTPUT

```

one iteration plus the next fetch, no forwarding, branches not predicted:

      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
lw   x5, 0(x10)    IF ID  EX MEM WB
add  x6, x6, x5          IF ID  EX MEM WB
addi x10, x10, 4          IF ID  EX MEM WB
addi x11, x11, -1          IF ID  EX MEM WB
bne  x11, x0, loop                IF ID  EX MEM WB

```

```

lw    x5, 0(x10)                                IF  ID  EX MEM  WB

with forwarding and a correct branch prediction:

      1  2  3  4  5  6  7  8  9 10 11 12
lw    x5, 0(x10)    IF  ID  EX MEM  WB
add    x6, x6, x5          IF  ID  EX MEM  WB
addi   x10, x10, 4          IF  ID  EX MEM  WB
addi   x11, x11, -1          IF  ID  EX MEM  WB
bne    x11, x0, loop        IF  ID  EX MEM  WB
lw     x5, 0(x10)          IF  ID  EX MEM  WB

```

```

1,000 iterations, 5,000 instructions:
no forwarding, no prediction      11,002 cycles   CPI 2.20
forwarding, no prediction        8,002 cycles   CPI 1.60
forwarding, branch predicted     6,004 cycles   CPI 1.20
forwarding, predicted, reordered 5,004 cycles   CPI 1.00

```

Each improvement removes one source of lost cycles. Forwarding removes the waits for write back, from CPI 2.20 to 1.60. Predicting the branch removes the two wasted fetches per iteration, giving 1.20. Moving the add away from the load removes the last bubble, giving the ideal CPI of 1.00. In the diagrams, the gaps before a stalled instruction's EX stage are its bubbles. Real hardware holds the stalled instruction in its stage rather than fetching it late, but the cycles lost are the same.

Deeper pipelines

More, shorter stages allow a faster clock, and in the early 2000s Intel's Pentium 4 used around 20 to 31 stages to reach high clock rates. The costs grow with depth. Forwarding paths multiply, every register between stages adds overhead (exercise 5.3), and a mispredicted branch throws away more work, because more instructions are in flight when the mistake is discovered. Designs settled at around 14 to 20 stages, with performance coming instead from issuing several instructions per cycle, which is chapter 14's subject.

COMMON TRAPS

Confusing latency and throughput. A pipelined instruction still takes five cycles; the pipeline finishes one per cycle.

Using a loaded value in the very next instruction in hand-written assembly for a simple in-order core.

Expecting deeper pipelines to be faster without limit. Overhead and misprediction cost grow with depth.

Ignoring that results depend on instruction order when timing tight loops on in-order processors.

Forgetting that every taken branch disturbs the fetch stream, even when it's predicted.

EXERCISES

EXERCISE 12.1 The ideal speed-up

An unpipelined processor takes 5 ns per instruction. A 5-stage pipeline splits the work evenly, adding 0.2 ns of register overhead per stage. What are the latency and throughput of the pipelined version, and what speed-up does it achieve on a long run of instructions without hazards?

SOLUTION

Each stage takes $5 / 5 + 0.2 = 1.2$ ns, so the clock period is 1.2 ns. Latency rises to $5 \times 1.2 = 6$ ns, longer than before, but throughput becomes one instruction per 1.2 ns. For a long run the speed-up approaches $5 / 1.2 \approx 4.2$, below the ideal of 5 because of the register overhead. Hazards reduce it further: with a CPI of 1.2, the effective time per instruction is 1.44 ns and the speed-up 3.5.

EXERCISE 12.2 Count the stalls

Without forwarding, how many bubbles does the example's loop suffer per iteration, and from which dependences? Check your answer against the CPI of 2.20.

SOLUTION

Without forwarding, a dependent instruction's EX can come no sooner than three cycles after its producer's EX, so a back-to-back dependence costs 2 bubbles. The add depends on the load just before it: 2 bubbles. The two addi instructions don't depend on their immediate predecessors. The branch depends on the addi just before it: 2 bubbles. The taken branch, unpredicted, wastes 2 more cycles. That's 6 lost cycles per 5-instruction iteration, so each iteration takes $5 + 6 = 11$ cycles, and $11 / 5 = 2.20$ cycles per instruction, matching the simulation.

EXERCISE 12.3 Schedule it yourself

With forwarding, schedule this code for the five-stage pipeline to avoid every stall: `lw x1, 0(x5)`, `add x2, x1, x3`, `lw x4, 4(x5)`, `add x6, x4, x7`.

SOLUTION

Each add uses the result of the load right before it, so as written there are two load-use bubbles. The second load doesn't depend on the first add, so move it up: `lw x1, 0(x5)` , `lw x4, 4(x5)` , `add x2, x1, x3` , `add x6, x4, x7` . Now each add comes two instructions after its load, and forwarding from MEM delivers the value in time. The code runs in 4 issue cycles with no bubbles, at the cost of needing x4 one instruction earlier. Compilers do this routinely, which is why reading optimised assembly often shows loads hoisted well above their uses.

CHECKLIST

- ☐ Can you name the five classic pipeline stages and draw a pipeline diagram?
- ☐ Can you distinguish data, control and structural hazards?
- ☐ Can you explain forwarding, and why a load-use dependence still costs a cycle?
- ☐ Can you reorder instructions to remove stalls?
- ☐ Can you explain why pipelines stopped getting deeper?

GO DEEPER

BOOK Computer Organization and Design, RISC-V edition, chapter 4 Patterson & Hennessy

COURSE CS 61C: pipelining and hazards UC Berkeley

BOOK Computer Systems: A Programmer's Perspective, section 4.5: pipelined implementations Bryant & O'Hallaron

Branch prediction and speculative execution

3 exercises

A modern core has well over a hundred instructions in flight, and a branch turns up every five or six of them, so waiting to learn which way each branch goes would leave most of the machine idle. Instead the processor predicts, runs ahead down the predicted path, and throws the work away if it guessed wrong. Predictors are right far more than 90% of the time on typical code, but a branch that depends on random data defeats them, and the difference can make the same loop several times slower. This chapter builds three predictors, measures them on four kinds of branch, and explains how speculation became the basis of the Spectre attacks.

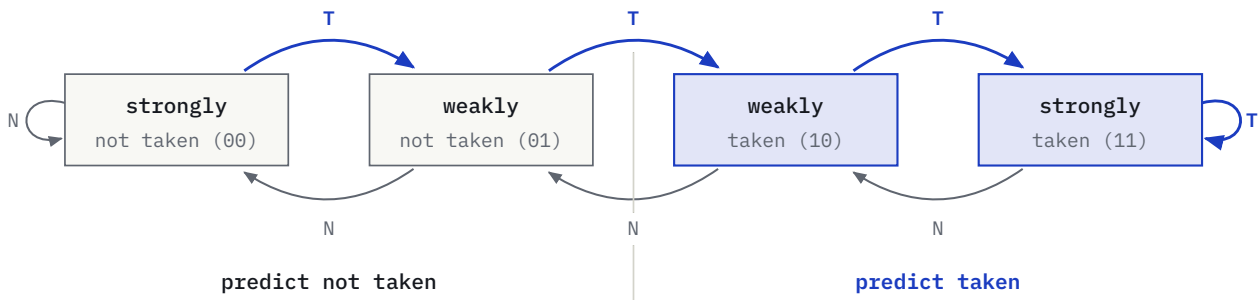
Why predict

Chapter 12's five-stage pipeline lost two cycles on every taken branch. A real core loses far more. It fetches several instructions per cycle into a pipeline 15 or more stages deep, so by the time a branch is resolved, dozens of later instructions have already been fetched. Stalling at every branch would waste most of that capacity. Predicting the branch and fetching from the guessed target keeps the pipeline full, and as long as the guesses are right the branches cost nearly nothing.

Predictors, from simple to clever

The simplest **static** rule predicts backward branches taken and forward branches not taken, which suits loops: the branch at the bottom of a loop jumps backwards and is usually taken. **Dynamic** predictors learn from history. A **1-bit** predictor remembers each branch's last outcome and predicts the same again. A **2-bit saturating counter** needs two mistakes in a row to change its mind, so the single not-taken exit at the end of a loop doesn't spoil the prediction for the next time the loop runs.

THE 2-BIT SATURATING COUNTER



T: the branch was taken, move right; N: not taken, move left; two surprises in a row flip the prediction

The 2-bit counter. Each outcome nudges the state one step, so a branch that is almost always taken stays in the "taken" half after a single not-taken surprise. Real predictors keep thousands of these counters, indexed by the branch's address.

Counters indexed by the branch address alone can't learn patterns such as "taken, not taken, taken, not taken", or "the inner loop exits every tenth time". **Global history** predictors also look at the outcomes of the last several branches. **Gshare**, from Scott McFarling in 1993, indexes its counters by the branch address XORed with a register of recent outcomes, so the same branch gets different counters in different contexts. Modern predictors such as TAGE combine several history lengths and reach over 95% accuracy on typical programs. A **branch target buffer** separately predicts *where* a taken branch goes, and a small **return address stack** predicts the targets of function returns.

Measured

The example implements a 1-bit predictor, a 2-bit predictor and gshare with 10 bits of history, and runs each on four branches: a loop that iterates 10 times, entered 1,000 times; an if-test on random bytes; the same test on the same bytes sorted; and a branch that alternates. It then turns accuracy into cycles per instruction for a 4-wide core with a 15-cycle misprediction penalty.

EXAMPLE

branches.py · 1-bit, 2-bit and gshare predictors on loops, random and sorted data, and alternation

```
import random

class TwoBit:
    """A table of 2-bit saturating counters: 0-1 predict not taken, 2-3 predict taken."""
    def __init__(self, bits=10): self.t = [1] * (1 << bits); self.mask = (1 << bits) - 1
    def predict(self, pc, hist): return self.t[pc & self.mask] >= 2
    def update(self, pc, hist, taken):
        i = pc & self.mask; self.t[i] = min(3, self.t[i] + 1) if taken else max(0, self.t[i] - 1)

class OneBit(TwoBit):
    def predict(self, pc, hist): return self.t[pc & self.mask] == 1
    def update(self, pc, hist, taken): self.t[pc & self.mask] = int(taken)
```

```

class Gshare(TwoBit):
    """2-bit counters indexed by the branch address XORed with the outcomes of recent
    branches."""
    def predict(self, pc, hist): return self.t[(pc ^ hist) & self.mask] >= 2
    def update(self, pc, hist, taken):
        i = (pc ^ hist) & self.mask; self.t[i] = min(3, self.t[i] + 1) if taken else max(0,
        self.t[i] - 1)

    def accuracy(predictor, trace, hist_bits=10):
        p, hist, right = predictor(), 0, 0
        for pc, taken in trace:
            right += p.predict(pc, hist) == taken
            p.update(pc, hist, taken)
            hist = ((hist << 1) | taken) & ((1 << hist_bits) - 1)
        return right / len(trace)

    def report(name, trace):
        print(f"{name:44} " + " ".join(f"{cls.__name__} {accuracy(cls, trace):6.1%}" for cls in
        (OneBit, TwoBit, Gshare)))

random.seed(13)
# 1. A loop that runs 10 times, entered 1,000 times: taken nine times, then not taken
loop = [(0x40, i < 9) for _ in range(1000) for i in range(10)]
report("loop branch, 10 iterations per entry", loop)

# 2. if (x >= 128) over 100,000 random bytes, unsorted and sorted
data = [random.randrange(256) for _ in range(100_000)]
report("if x >= 128, random data", [(0x80, x >= 128) for x in data])
report("if x >= 128, the same data sorted", [(0x80, x >= 128) for x in sorted(data)])

# 3. A branch that alternates: taken, not taken, taken, ...
report("alternating branch", [(0xC0, i % 2 == 0) for i in range(10_000)])

# 4. What mispredictions cost: 1 branch per 5 instructions, a 4-wide core, a 15-cycle penalty
def cpi(miss_rate, base=0.25, branch_frac=0.2, penalty=15): return base + branch_frac *
miss_rate * penalty
for label, trace in (("random", [(0x80, x >= 128) for x in data]), ("sorted", [(0x80, x >= 128)
for x in sorted(data)])):
    miss = 1 - accuracy(TwoBit, trace)
    print(f"cycles per instruction with {label} data: {cpi(miss):.2f}")

# 5. The branch-free version: compute the condition as a number and add it
total_branchy = sum(x for x in data if x >= 128)
total_branchless = sum(x * (x >= 128) for x in data) # the comparison becomes a 0
or 1 multiplier
print(f"sum of bytes >= 128: with a branch {total_branchy:},, without {total_branchless:}")

```

OUTPUT

```

loop branch, 10 iterations per entry          OneBit  80.0%  TwoBit  90.0%  Gshare  99.8%
if x >= 128, random data                      OneBit  50.2%  TwoBit  50.2%  Gshare  49.8%
if x >= 128, the same data sorted              OneBit 100.0%  TwoBit 100.0%  Gshare 100.0%
alternating branch                           OneBit   0.0%  TwoBit   0.0%  Gshare  99.9%
cycles per instruction with random data: 1.75
cycles per instruction with sorted data: 0.25
sum of bytes >= 128: with a branch 9,570,757, without 9,570,757

```

Each result has a simple explanation. On the loop, the 1-bit predictor misses twice per entry, at the exit and again at the first iteration of the next entry, for 80%. The 2-bit counter misses only at the exit, for 90%. Gshare sees the history of the last ten outcomes and learns exactly where the exit comes, reaching 99.8%. On random data every predictor scores about 50%, because there's no pattern to learn. On the same data sorted, the branch is not

taken for the first half and taken for the second, which every predictor gets almost entirely right. The alternating branch is the 2-bit counter's worst case, always one step behind, at 0%, while gshare's history makes it trivial.

What a misprediction costs

A wrong guess means flushing every instruction fetched after the branch and restarting from the right target, typically 15 to 20 cycles on a modern core. With 1 instruction in 5 a branch, a 50% miss rate costs $0.2 \times 0.5 \times 15 = 1.5$ cycles per instruction, so the example's loop over random data runs at a CPI of 1.75 against 0.25 for sorted data, seven times slower for identical work. In 2012 this became one of Stack Overflow's most famous questions, "Why is processing a sorted array faster than processing an unsorted array?", with exactly this explanation.

The cure for an unpredictable branch is not to branch. Compute the condition as a number and use it arithmetically, as the example's `x * (x >= 128)` does. Alternatively, use a **conditional move**, an instruction that selects between two values without changing the flow of control, which compilers emit when they judge a branch unpredictable. Branch-free code always does the work of both sides, so it's slower when the branch would have been predictable. Measure before choosing.

Speculation, and Spectre

Running ahead on a prediction is **speculative execution**. Instructions on the predicted path execute fully, and their results are kept in the pipeline until the branch is confirmed, then either committed or discarded. Architecturally the discarded work leaves no trace, but microarchitecturally it can: a speculative load still brings its line into the cache. In 2018 Paul Kocher, Jann Horn and colleagues published **Spectre**. An attacker trains a bounds-check branch to predict "in bounds", then supplies an out-of-bounds index. The processor speculatively reads secret memory and uses it to touch one of 256 cache lines before discovering the mistake, and timing which line is now cached reveals the secret byte. **Meltdown**, announced at the same time, let unprivileged code read kernel memory through speculative loads past a permission check on many Intel processors.

The defences cost performance and live at every level: speculation barriers and index masking in sensitive code, hardware fixes in new processors, operating-system changes such as separating kernel and user page tables, and browsers isolating each site in its own process. *The Engineer's Citadel* covers side channels in more depth. The lesson for this chapter is that performance tricks invisible to a program's results can still be visible to an attacker who measures time.

COMMON TRAPS

Unpredictable branches in hot loops, such as tests on random or hashed data.

Benchmarking with sorted or regular test data and deploying on real, messy data

Generalizing with sorted or regular test data and deploying on real, messy data.

Rewriting predictable branches as branch-free code, which makes them slower.

Branching on secret data in cryptographic code, which leaks it through timing; such code must be constant-time.

Assuming discarded speculative work leaves no trace. Caches remember it.

EXERCISES

EXERCISE 13.1 A period-3 pattern

A branch repeats the pattern taken, taken, not taken. What steady-state accuracy do a 1-bit predictor and a 2-bit counter achieve?

SOLUTION

1-bit: it predicts the previous outcome. After N it predicts N but sees T, wrong; after that T it predicts T and sees T, right; then it predicts T and sees N, wrong. One right in three, 33%. 2-bit: in steady state the counter sits at "weakly taken" when the pattern restarts. T is predicted correctly and moves it to "strongly taken"; T again, right; N is mispredicted and moves it back to "weakly taken". Two right in three, 67%. A global-history predictor like gshare learns the pattern completely, since the previous two outcomes always determine the next one.

EXERCISE 13.2 The price of a miss rate

A program's base CPI is 0.3. One instruction in five is a branch, mispredicted 10% of the time, with a 20-cycle penalty. What's its CPI, and how much faster would it run with a perfect predictor?

SOLUTION

Mispredictions add $0.2 \times 0.1 \times 20 = 0.4$ cycles per instruction, so $\text{CPI} = 0.3 + 0.4 = 0.7$. A perfect predictor would bring it back to 0.3, 2.3 times faster. A 90% accurate predictor sounds good, but on a wide, deep machine the remaining 10% of misses more than double the run time. That's why predictor accuracy keeps improving in each processor generation, and why the last few percent are worth so much hardware.

EXERCISE 13.3 Should you sort first?

Sorting made the example's loop seven times faster. Is it worth sorting an array just to run that loop over it once? What about a hundred times?

SOLUTION

Sorting n elements takes about $n \log_2 n$ comparisons, around 1.7 million for 100,000 elements, and the comparisons of a general-purpose sort are themselves hard to predict. The single loop makes only 100,000 branch decisions, so sorting costs far more than the mispredictions it saves. For a hundred passes over the same data, sorting once could pay for itself. But the better answer is usually branch-free code, which removes the mispredictions without sorting. For this particular test, the sum of the bytes at or above 128, a single pass with arithmetic is fastest, and the example confirms that it computes the same total.

CHECKLIST

- ☐ Can you explain why deep, wide pipelines need branch prediction?
- ☐ Can you trace a 1-bit and a 2-bit predictor on a pattern, and say what global history adds?
- ☐ Can you turn a miss rate into cycles per instruction?
- ☐ Can you write a branch-free version of a data-dependent test, and say when it helps?
- ☐ Can you explain how Spectre turns speculation into a side channel?

GO DEEPER

ARTICLE Branch prediction Dan Luu

PAPER Spectre attacks: exploiting speculative execution Kocher et al., 2018

ARTICLE Why is processing a sorted array faster than an unsorted array? Stack Overflow, 2012

Instruction-level parallelism and out-of-order execution

3 exercises

A modern core decodes several instructions per cycle, looks hundreds of instructions ahead, and executes each one as soon as its inputs are ready, in whatever order that turns out to be. It still commits results in program order, so the program can't tell. Three things limit how much it finds to do. Dependency chains force waiting, the number of functional units caps throughput, and the size of the instruction window limits how far ahead it can look. This chapter simulates an idealised out-of-order core and shows how to write code with more independent work in it: several accumulators instead of one, registers that don't create false dependences, and data that isn't reached by chasing pointers.

Superscalar, out of order, in-order commit

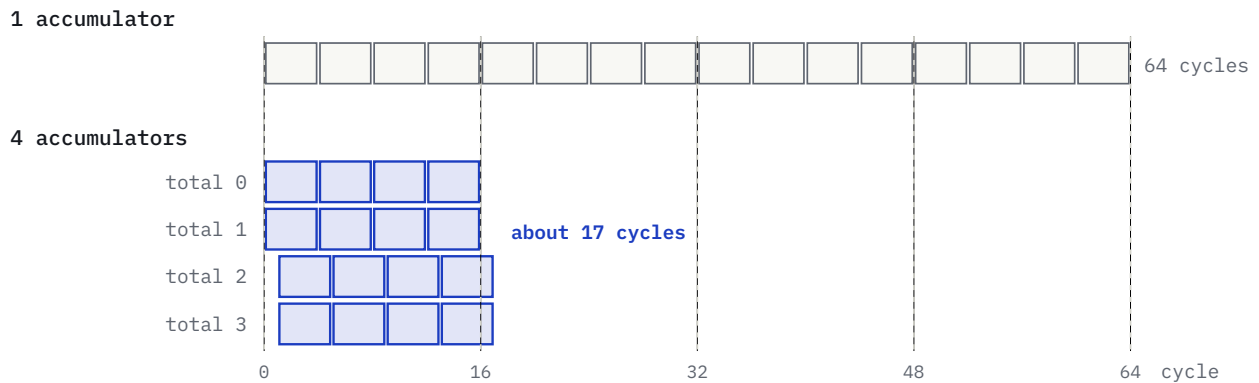
A **superscalar** processor issues more than one instruction per cycle. Today's large cores decode 6 to 8 or more per cycle and have several integer ALUs, two or more load and store units, and several floating-point and vector units. An **out-of-order** core fetches and decodes instructions in program order and places them in a large window, often called the reorder buffer, holding several hundred instructions. Any instruction whose operands are ready goes to a free unit, regardless of where it sits in the program. Finished instructions then **retire** in program order, which keeps exceptions precise: if an instruction faults, everything before it has completed and nothing after it has changed the architectural state. That in-order retirement is also what makes speculation, chapter 13, safe to undo.

Dependences, true and false

A **true dependence**, or read-after-write, means an instruction needs a value another computes, and nothing can avoid waiting for it. **False dependences** come from reusing register names. A write-after-read means an instruction overwrites a register an earlier one still has to read; a write-after-write means two instructions write the same register and the later value must win. Compilers reuse registers constantly, because there are only 32 names, so false dependences are everywhere. **Register renaming** removes them. The core has a much larger physical register file, often a few hundred registers, and gives every new result a fresh physical register, mapping the architectural name to whichever physical register holds its latest value. In the example, 64 independent load-multiply-add steps that all reuse x5 and x6 take 449 cycles in order and 260 out of order without renaming, but only 71 with it.

Latency, throughput and how many chains

Summing an array with one running total is a single chain of dependent additions, each waiting for the last. With a floating-point add latency of 4 cycles, the core manages one add every 4 cycles however many adders it has. Split the sum into k independent totals that are combined at the end, and k chains run side by side.



Sixteen additions with a latency of 4 cycles and two adders. One accumulator is one long chain, where every add waits for the one before it. Four accumulators give four chains that overlap, nearly four times faster for the same arithmetic.

How many chains are enough? To keep every adder busy, you need as many additions in flight as latency times throughput, here $4 \text{ cycles} \times 2 \text{ adders} = 8$. That's Little's law again, the same rule as chapter 9's outstanding cache misses. The example's sums of 1,024 floats run at 0.25, 0.5, 1 and then 1.9 adds per cycle for 1, 2, 4 and 8 accumulators, close to the 2-per-cycle limit. Sixteen accumulators are slightly slower, because combining them at the end takes a chain of 15 more adds. Combining them as a tree would fix that (exercise 14.2). Compilers make this transformation for integer sums, but not for floating point without permission such as `-ffast-math`, because floating-point addition isn't associative (chapter 2), and reordering a sum changes its last bits.

Simulated

The example's core starts each instruction as soon as its inputs are ready and a functional unit of the right kind is free, with 2 floating-point adders, 2 load units, 4 integer ALUs and 1 multiplier. Its window is unlimited, which makes it an upper bound on what real hardware finds.

EXAMPLE

ooo.py · an idealised out-of-order core: accumulators, register renaming and pointer chasing

```
# An idealised out-of-order core: an instruction starts once its inputs are ready and a unit is free.
LAT = {"fadd": 4, "load": 4, "add": 1, "mul": 3}
UNITS = {"fadd": 2, "load": 2, "add": 4, "mul": 1} # how many of each can start
```

per cycle

```
def run(program, rename=True, in_order=False):
    """program: list of (op, dest, sources). Returns the cycle the last result is ready."""
    ready, readers_done, busy, last_start, finish = {}, {}, {}, 0, 0
    for op, dst, srcs in program:
        t = max([ready.get(s, 0) for s in srcs] + [0])          # true dependences (read after
                                                                # write)
        if not rename and dst:                                  # without renaming, also wait
            for
                t = max(t, readers_done.get(dst, 0), ready.get(dst, 0)) # earlier readers and
                                                                # writers of dst
        if in_order: t = max(t, last_start)                     # in order: never start before
                                                                # an earlier one
        while busy.get((op, t), 0) >= UNITS[op]: t += 1        # find a cycle with a free
                                                                # unit
        busy[(op, t)] = busy.get((op, t), 0) + 1
        last_start = t
        done = t + LAT[op]
        for s in srcs: readers_done[s] = max(readers_done.get(s, 0), t)
        if dst: ready[dst] = done
        finish = max(finish, done)
    return finish

# 1. Summing 1,024 floats: one running total is a chain of dependent adds
N = 1024
print(f"summing {N:,} floats (add latency 4 cycles, 2 adders):")
for k in (1, 2, 4, 8, 16):
    prog = [("fadd", f"acc{i % k}", [f"acc{i % k}"]) for i in range(N)]          # k
                                                                # independent totals
    prog += [("fadd", "acc0", ["acc0", f"acc{j}"]) for j in range(1, k)]        # combine
                                                                # them at the end
    c = run(prog)
    print(f"    {k:2} accumulator{'s' if k > 1 else ' '}: {c:5,} cycles    ({N / c:.2f} adds per
    cycle)")

# 2. Register renaming: the same register reused for independent work creates false dependences
prog = []
for i in range(64):
    # each step: load, multiply,
    # add; all reuse x5, x6
    prog += [("load", "x5", [f"p{i}"]), ("mul", "x6", ["x5", "x5"]), ("add", f"s{i}", ["x6"])]
print(f"\n64 independent load-multiply-add steps that all reuse registers x5 and x6:")
print(f"    in order, no renaming:          {run(prog, rename=False, in_order=True):5,} cycles")
print(f"    out of order, no renaming:         {run(prog, rename=False):5,} cycles")
print(f"    out of order, renamed:              {run(prog, rename=True):5,} cycles")

# 3. Pointer chasing: each load needs the previous one's result, so nothing can overlap
print()
for lat, where in ((4, "L1 cache"), (100, "DRAM")):
    LAT["load"] = lat
    chain = [("load", "p", ["p"]) for _ in range(1000)]          # p = *p, a thousand times
    indep = [("load", f"v{i}", ["base"]) for i in range(1000)]    # a thousand loads from known
                                                                # addresses
    print(f"1,000 loads from {where:8} ({lat:3} cycles each): dependent chain {run(chain):7,}
    cycles, "
          f"independent {run(indep):4,} cycles")
```

OUTPUT

```
summing 1,024 floats (add latency 4 cycles, 2 adders):
  1 accumulator : 4,096 cycles    (0.25 adds per cycle)
  2 accumulators: 2,052 cycles    (0.50 adds per cycle)
  4 accumulators: 1,036 cycles    (0.99 adds per cycle)
  8 accumulators:   540 cycles    (1.90 adds per cycle)
 16 accumulators:   572 cycles    (1.79 adds per cycle)
```

```

64 independent load-multiply-add steps that all reuse registers x5 and x6:
  in order, no renaming:          449 cycles
  out of order, no renaming:      260 cycles
  out of order, renamed:         71 cycles

1,000 loads from L1 cache ( 4 cycles each): dependent chain 4,000 cycles, independent 503
cycles
1,000 loads from DRAM (100 cycles each): dependent chain 100,000 cycles, independent 599
cycles

```

The last two lines measure memory-level parallelism. A thousand loads from independent addresses overlap almost completely: 503 cycles from L1, and only 599 from DRAM, because in this idealised model all of them can be outstanding at once. A real core tracks a few dozen outstanding misses, so independent DRAM loads take longer than that, but still far less than one by one. A thousand dependent loads, each needing the previous one's result as its address, can't overlap at all: 100,000 cycles from DRAM. Out-of-order hardware is powerless against a dependency chain through memory, which is why chapter 11's linked list is slow on every processor ever built.

COMMON TRAPS

Reductions with a single accumulator in hot floating-point loops.

Expecting the compiler to reassociate floating-point arithmetic without being allowed to.

Long dependency chains through memory, such as linked structures, which no hardware can overlap.

Judging code by instruction count. Cycles depend on dependences and units, not just on how many instructions there are.

Assuming a wider core always helps. Without independent work, extra units sit idle.

EXERCISES

EXERCISE 14.1 Enough chains for an FMA unit

A core has 2 fused multiply-add units, each able to start one FMA per cycle, with a latency of 4 cycles. A dot product accumulates $s \leftarrow s + a_i b_i$ with FMAs. How many independent accumulators does it need to run at full speed? What if each FMA works on 8-lane vectors?

SOLUTION

In-flight operations needed = latency $\times$ throughput = $4 \times 2 = 8$ independent accumulators. With 8-lane vectors, each accumulator is a vector register holding 8 partial sums, so the core keeps $8 \times 8 = 64$ partial sums in flight, still in 8 registers, and processes 16 elements per cycle at peak. That's why hand-tuned dot products and matrix kernels use around 8 to 12 vector accumulators. It also shows why such kernels are usually limited by memory bandwidth rather than arithmetic: 16 elements per cycle at several gigahertz is far more data than memory can deliver.

EXERCISE 14.2 Combine as a tree

With 16 accumulators the example combined them in a chain of 15 dependent adds. How long does a tree combination take instead, and how many cycles would the 16-accumulator sum then need?

SOLUTION

A chain of 15 adds at 4 cycles each takes 60 cycles. A tree adds pairs in parallel: 16 values become 8, then 4, 2 and 1, in 4 levels, each 4 cycles long, so 16 cycles, and the 8 adds of the first level fit on 2 adders within those cycles. The main loop takes about 512 cycles, since 1,024 adds on 2 adders need at least 512, so the total falls from 572 to about 528. In general, combining k partial results takes about $\log_2 k$ latency steps as a tree, against $k - 1$ as a chain, the same tree-versus-chain argument as chapter 3's balanced Or.

EXERCISE 14.3 Why integers but not floats

Compilers happily split an integer sum into several accumulators, but not a floating-point sum. Why is the transformation exact for integers?

SOLUTION

Integer addition, including its wrap-around on overflow, is arithmetic modulo 2^n (chapter 1), which is associative and commutative: any grouping and order gives exactly the same bits. Splitting a sum into partial sums and combining them regroups the additions, so the result is identical. Floating-point addition rounds after every operation, so regrouping changes the rounding errors and can change the result, as chapter 2's non-associativity examples showed. Languages promise a specific order of evaluation for floating point, so a compiler may reorder it only with an explicit flag, or when the programmer writes the multiple accumulators by hand and accepts the slightly different result.

CHECKLIST

- ☐ Can you explain superscalar issue, out-of-order execution and in-order retirement?
- ☐ Can you distinguish true dependences from false ones, and explain register renaming?
- ☐ Can you compute how many independent chains a loop needs with latency $\times$ throughput?
- ☐ Can you rewrite a reduction with multiple accumulators?
- ☐ Can you explain why pointer chasing defeats out-of-order hardware?

GO DEEPER

BOOK

Computer Systems: A Programmer's Perspective, chapter 5: optimizing program performance Bryant & O'Hallaron

REFERENCE

Software optimization resources: microarchitecture and instruction tables
Agner Fog

ARTICLE

Modern microprocessors: a 90-minute guide Jason Robert Carey Patterson

SIMD, multicore and the limits of parallelism

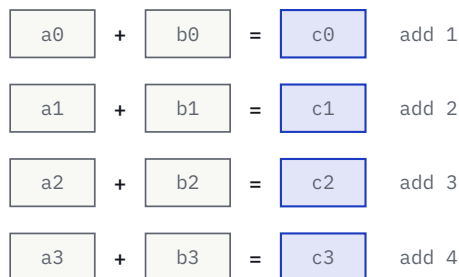
3 exercises

When clock rates stopped rising around 2005, performance had to come from doing more at once. SIMD instructions apply one operation to a whole vector of values. Multicore chips run several instruction streams side by side, and GPUs push the SIMD idea to thousands of lanes. Each pays off only when the work has the right shape: data laid out contiguously, iterations that don't depend on each other, and threads that don't fight over the same cache lines. Amdahl's law sets the ceiling, because whatever part of a program stays serial decides how much faster the rest can make it.

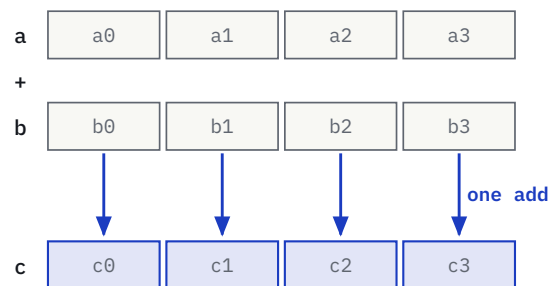
SIMD: one instruction, many lanes

A SIMD instruction, single instruction, multiple data, operates on a wide register holding several values, called **lanes**, at once. A 128-bit register holds four 32-bit floats, and one instruction adds four pairs. x86 has had 128-bit SSE since 1999, 256-bit AVX2 since 2013, and 512-bit AVX-512 on some processors. ARM has 128-bit NEON and the scalable SVE, whose vector length the hardware chooses. Adding two arrays of 1,024 floats takes 1,024 scalar adds, 256 with 128-bit vectors and 64 with 512-bit ones.

4 SCALAR INSTRUCTIONS



1 SIMD INSTRUCTION



same arithmetic: four instructions on the left, one on the right

Four scalar adds against one 4-lane SIMD add. The vector version needs its operands side by side in memory, so they can be loaded into a register in one go, and its lanes can't depend on one another.

SIMD has three requirements. The data must be **contiguous**, so a vector load can fetch several elements at once, which is another reason for chapter 11's structure-of-arrays layout. The iterations must be **independent**: `a[i] = a[i-1] + b[i]` can't run four lanes at once, because each lane needs the previous one's result. And **branches become masks**. Each lane computes both possibilities and a mask selects the result, so lanes whose

condition is false do wasted work, as in the example, where 5 of 8 lanes are useful. Compilers **auto-vectorise** simple loops that meet these conditions, and libraries such as NumPy, BLAS and image codecs use SIMD intrinsics written by hand.

GPUs

A GPU takes the same idea to an extreme. Thousands of simple lanes are grouped into units of 32 or so, called warps on NVIDIA hardware, which execute the same instruction in lockstep on different data, a model NVIDIA calls SIMT. A GPU keeps tens of thousands of threads in flight to hide memory latency, the throughput strategy of chapter 9 taken as far as it goes, and pairs them with very high memory bandwidth. It excels at large, regular, data-parallel work such as graphics, matrix multiplication and the training of neural networks. It suffers when threads in a warp take different branches, because the warp must run both paths with half its lanes masked off each time.

Multicore

A multicore chip puts several complete cores on one die, sharing the last-level cache and memory. Threads on different cores share memory, and the coherence protocol of chapter 10 keeps their caches consistent, which costs time whenever cores write to the same line. **False sharing** makes that cost appear where no data is actually shared. In the example's model, two cores each incrementing their own counter cause 200,000 line transfers when the counters sit on one cache line, and 2 when they sit on separate lines. Beyond coherence, parallel programs pay for creating and synchronising threads and for contention on shared locks, so splitting small tasks across cores often makes them slower. Stage 05 covers threads, locks and their bugs.

Amdahl's law

If a fraction p of a program's work can be parallelised perfectly over n processors and the rest stays serial, the speed-up is

$$S(n) = \frac{1}{(1 - p) + p/n},$$

which can never exceed $1/(1 - p)$, however many processors there are. Gene Amdahl made the argument in 1967. With 90% of the work parallel, 16 cores give a speed-up of 6.4 and no number of cores gives more than 10. The serial part, such as reading input, combining results or a lock everyone needs, dominates as soon as the parallel part has been divided up. **Gustafson's law** gives the optimistic counterpart: people use more processors to solve bigger problems, and if the parallel part grows with the problem, the serial fraction shrinks, which is how supercomputers stay useful.

Counted

The example counts instructions for SIMD addition at four vector widths, applies a mask in place of a branch, tabulates Amdahl's law, and counts the coherence traffic that false sharing causes.

EXAMPLE

simd.py · SIMD widths, masks instead of branches, Amdahl's law and false sharing

```
# 1. SIMD: one instruction applies the same operation to several lanes at once
def simd_add(a, b, lanes):
    out, instructions = [], 0
    for i in range(0, len(a), lanes):
        out += [x + y for x, y in zip(a[i:i + lanes], b[i:i + lanes])] # one vector
                                instruction
        instructions += 1
    return out, instructions
a, b = list(range(1024)), list(range(1024, 2048))
print("adding two arrays of 1,024 32-bit floats:")
for lanes, name in ((1, "scalar"), (4, "128-bit SIMD (SSE, NEON)"), (8, "256-bit SIMD (AVX2)"),
                    (16, "512-bit SIMD (AVX-512)")):
    out, n = simd_add(a, b, lanes)
    assert out == [x + y for x, y in zip(a, b)]
    print(f"    {name:26} {n:5,} add instructions")

# 2. Branches become masks: every lane computes, and a mask selects the results
xs = [3, -1, 4, -1, 5, -9, 2, 6]
mask = [x > 0 for x in xs] # one vector compare
result = [x * 2 if m else 0 for x, m in zip(xs, mask)] # compute everywhere, then
               blend
print(f"\ndouble the positive values with a mask: {xs} -> {result}; lanes doing useful work:
      {sum(mask)} of {len(xs)}")

# 3. Amdahl's law: the serial part of a program limits the speed-up from more cores
def amdahl(p, n): return 1 / ((1 - p) + p / n)
print("\nspeed-up on n cores when a fraction p of the work runs in parallel:")
print(f"    {p':>5} " + " ".join(f"{n:>8}" for n in (2, 4, 8, 16, 64, 1024)) + "    limit")
for p in (0.5, 0.9, 0.95, 0.99):
    print(f"    {p:>5} " + " ".join(f"{amdahl(p, n):8.2f}" for n in (2, 4, 8, 16, 64, 1024)) + f"
          {1 / (1 - p):5.0f}x")

# 4. False sharing: two cores each incrementing their own counter, on one cache line or on two
def coherence_traffic(same_line, increments=100_000):
    owner, transfers = {}, 0
    for i in range(increments):
        for core in (0, 1):
            line = 0 if same_line else core # which cache line this
                                           core's counter is on
            if owner.get(line) != core: # the line must move to
                this core first
                transfers += 1; owner[line] = core
    return transfers
print(f"\n2 cores x 100,000 increments: counters sharing one line need
      {coherence_traffic(True):,} line transfers, "
      f"counters on separate lines {coherence_traffic(False):,}")
```

OUTPUT

```
adding two arrays of 1,024 32-bit floats:
  scalar                1,024 add instructions
 128-bit SIMD (SSE, NEON)    256 add instructions
 256-bit SIMD (AVX2)        128 add instructions
 512-bit SIMD (AVX-512)     64 add instructions
```

double the positive values with a mask: [3, -1, 4, -1, 5, -9, 2, 6] -> [6, 0, 8, 0, 10, 0, 4, 12]; lanes doing useful work: 5 of 8

speed-up on n cores when a fraction p of the work runs in parallel:

p	2	4	8	16	64	1024	limit
0.5	1.33	1.60	1.78	1.88	1.97	2.00	2x
0.9	1.82	3.08	4.71	6.40	8.77	9.91	10x
0.95	1.90	3.48	5.93	9.14	15.42	19.64	20x
0.99	1.98	3.88	7.48	13.91	39.26	91.18	100x

2 cores x 100,000 increments: counters sharing one line need 200,000 line transfers, counters on separate lines 2

COMMON TRAPS

Data layouts that defeat vector loads, such as arrays of structures scanned one field at a time.

Loop-carried dependences that silently prevent vectorisation.

Expecting linear speed-up from more cores. Measure the serial fraction first.

Parallelising tiny tasks, where thread start-up and synchronisation cost more than the work.

Per-thread data packed into one cache line, the false-sharing trap.

Adding cores to a memory-bound program, which saturates bandwidth and stops scaling (chapter 20).

EXERCISES

EXERCISE 15.1 Amdahl in practice

A batch job spends 20% of its time in a serial setup phase and the rest in work that parallelises perfectly. What's its speed-up on 16 cores, and what's the limit? How much faster would 16 cores be if you also halved the setup time?

SOLUTION

With $p = 0.8$: $S(16) = 1 / (0.2 + 0.8/16) = 1 / 0.25 = 4$. The limit is $1 / 0.2 = 5$. Halving the setup makes the job take $0.1 + 0.8/16 = 0.15$ of the original single-core time on 16 cores, a speed-up of 6.7. Halving the serial part helped more than doubling the cores would have: 32 cores with the original setup give $1 / (0.2 + 0.025) = 4.4$. Optimise the serial part first.

EXERCISE 15.2 A loop that won't vectorise

Why can't `for i in 1..n: a[i] = a[i-1] + b[i]`, a running sum, be vectorised directly? How can a running sum still be computed in parallel?

SOLUTION

Each element depends on the one computed just before it, a loop-carried dependence, so four lanes can't start four iterations at once: lane 2 would need lane 1's result in the same instruction. But a running sum is a **prefix sum**, or scan, and addition is associative (for integers), so it can be restructured. Within a vector of 4, add each element to its neighbour one place to the left, then to the element two places to the left, which gives 4 prefix sums in 2 steps, $\log_2 4$. Then add the running total from the previous vector to all four lanes. Parallel prefix algorithms do the same at larger scale in $\log n$ steps, the same associativity trick as chapter 4's prefix adder.

EXERCISE 15.3 Vectorising a memory-bound loop

A loop adds two large arrays of floats that don't fit in cache, `c[i] = a[i] + b[i]`. Switching from scalar code to 8-lane SIMD makes it barely faster. Why?

SOLUTION

Each element needs 12 bytes of memory traffic, two 4-byte loads and one 4-byte store, for a single addition. A core can do billions of additions per second even without SIMD, but DRAM delivers perhaps 20 to 100 GB/s, so at 12 bytes per element the memory system supplies only a few billion elements per second. That's already about what scalar code achieves, so the loop is bandwidth-bound and the arithmetic was never the bottleneck. SIMD helps loops that do a lot of arithmetic per byte, or whose data is in cache. Chapter 20's roofline model makes this calculation routine: compare the arithmetic per byte with the machine's ratio of compute to bandwidth.

CHECKLIST

- ☐ Can you explain SIMD lanes and the three conditions for vectorising a loop?
- ☐ Can you replace a branch with a mask, and say what it costs?
- ☐ Can you explain how GPUs differ from CPUs, and what divergence does?

- ☐ Can you apply Amdahl's law, and explain Gustafson's counterpoint?
- ☐ Can you recognise and fix false sharing between threads?

GO DEEPER

BOOK Computer Organization and Design, RISC-V edition, chapter 6: parallel processors Patterson & Hennessy

REFERENCE Intel intrinsics guide Intel

PAPER Validity of the single processor approach to achieving large scale computing capabilities Gene Amdahl, 1967

PART V

From source to process

Three chapters on how a program written as text becomes a running process: compiling it in stages to lower-level code and finally machine code; linking separately compiled pieces into one executable and loading it into memory; and the call stack that every running function uses, along with the overflow bugs that have made it the most attacked structure in computing.

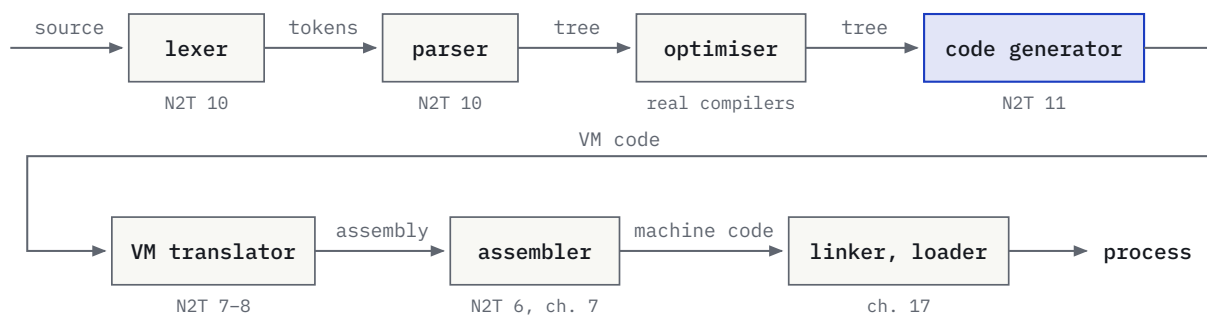
Compiling and assembling

3 exercises

A compiler is a pipeline of simple translations. Lexing turns characters into tokens, parsing turns tokens into a tree, analysis and optimisation transform the tree, and code generation turns it into instructions for some target machine. Each stage is a short program, and the tree in the middle is what makes compilers manageable. This chapter builds all four stages for a small language in about a hundred lines of Python, compiles a loop into chapter 8's stack VM, and runs it. That's the shape of Nand2Tetris projects 10 and 11, and of every real compiler.

The pipeline

Translation from source to machine happens in several steps, and Nand2Tetris builds every one of them. Each stage consumes the previous one's output and produces something closer to the machine.



From source text to a running process. The upper row is the compiler proper. The lower row is the translation the earlier chapters built: chapter 8's VM, chapter 7's assembler, and chapter 17's linker and loader. Real toolchains such as GCC and Clang follow the same outline, with an intermediate representation in place of the VM.

Lexing and parsing

The **lexer**, or tokenizer, groups characters into **tokens**: names, numbers, operators and punctuation, discarding whitespace and comments. A regular expression is often enough, as in the example. The **parser** checks that the tokens follow the language's **grammar** and builds an **abstract syntax tree** (AST), a tree whose nodes are operations and whose leaves are names and constants. **Recursive descent** is the simplest parsing method: write one

function per grammar rule, each calling the functions for the rules it contains. Operator precedence falls out of the structure. An expression is a sum of terms and a term is a product of factors, so `expr` calls `term`, which calls `factor`, and multiplication binds tighter than addition without any special handling. The tree for `x = (2 + 3) * y - 4 * 2` is `('-', ('*', ('+', 2, 3), 'y'), ('*', 4, 2))`, which makes the order of evaluation explicit.

Analysis and optimisation

Between parsing and code generation, a compiler builds **symbol tables**, mapping each name to its type and its storage location, such as the example's local slots. It checks types, and transforms the tree. **Constant folding** evaluates operations whose operands are known at compile time, so `(2 + 3) * y - 4 * 2` becomes `5 * y - 8`, and the generated code shrinks from 10 VM commands to 6. Real optimisers apply dozens of passes. They remove dead code, reuse common subexpressions, inline small functions, move work out of loops, and turn multiplication by a constant into shifts (chapter 4). They also assign variables to registers, which is graph colouring (*The Engineer's Workbench*, chapter 8), and schedule instructions around pipeline stalls (chapter 12). Most of a modern compiler is its optimiser, and most of that works on an intermediate representation such as LLVM's IR rather than on the source tree.

Code generation

For a stack machine, code generation is a post-order walk of the tree: emit code for the left operand, then for the right, then the operator. That's exactly the order a stack machine needs, operands pushed before the operation pops them. Control flow becomes labels and jumps. A `while` loop compiles to a label at the top, the condition, a jump to the end if it's false, the body, and a jump back to the top. Generating code for a register machine adds two harder problems, choosing instructions and allocating registers, which is why many compilers, and Nand2Tetris, first target a stack VM and translate from there.

Compiled and run

The example lexes and parses an assignment, folds its constants and generates VM code before and after folding. It then compiles a `while` loop that sums 1 to 100 and runs the result on a stack VM.

EXAMPLE

compiler.py · a lexer, a recursive-descent parser, constant folding and code generation for a stack VM

```
import re

# 1. Lexing: characters become tokens
def lex(src):
    return re.findall(r"\d+|[A-Za-z_]\w*|<=|[-+*/()=<{ }];", src)

# 2. Parsing: tokens become a tree, by recursive descent with one function per precedence level
```

```

class Parser:
    def __init__(self, tokens): self.t, self.i = tokens, 0
    def peek(self): return self.t[self.i] if self.i < len(self.t) else None
    def eat(self, tok=None):
        x = self.t[self.i]; self.i += 1
        assert tok is None or x == tok, f"expected {tok}, got {x}"
        return x
    def program(self):
        stmts = []
        while self.peek() not in (None, "}"): stmts.append(self.statement())
        return stmts
    def statement(self):
        if self.peek() == "while":
            self.eat(); cond = self.comparison(); self.eat("{"); body = self.program();
            self.eat("}")
            return ("while", cond, body)
        name = self.eat(); self.eat("="); value = self.expr(); self.eat(";")
        return ("let", name, value)
    def comparison(self):
        left = self.expr()
        if self.peek() in("<", "<="): op = self.eat(); return (op, left, self.expr())
        return left
    def expr(self):
        # + and - bind loosest
        node = self.term()
        while self.peek() in("+", "-"): op = self.eat(); node = (op, node, self.term())
        return node
    def term(self):
        # * and / bind tighter
        node = self.factor()
        while self.peek() in("*", "/"): op = self.eat(); node = (op, node, self.factor())
        return node
    def factor(self):
        tok = self.eat()
        if tok == "(": node = self.expr(); self.eat(")"); return node
        return int(tok) if tok.isdigit() else tok

# 3. An optimisation pass: fold operations on constants at compile time
def fold(n):
    if not isinstance(n, tuple): return n
    if n[0] == "let": return ("let", n[1], fold(n[2]))
    if n[0] == "while": return ("while", fold(n[1]), [fold(s) for s in n[2]])
    op, a, b = n[0], fold(n[1]), fold(n[2])
    if isinstance(a, int) and isinstance(b, int) and op in "+-*":
        return {"+": a + b, "-": a - b, "*": a * b}[op]
    return (op, a, b)

# 4. Code generation: walk the tree, children first, emitting stack-VM commands
def gen(n, out, slots, labels):
    if isinstance(n, int): out.append(f"push constant {n}")
    elif isinstance(n, str): out.append(f"push local {slots.setdefault(n, len(slots))}")
    elif n[0] == "let":
        gen(n[2], out, slots, labels); out.append(f"pop local {slots.setdefault(n[1], len(slots))}")
    elif n[0] == "while":
        k = len(labels); labels.append(k)
        out.append(f"label TOP{k}"); gen(n[1], out, slots, labels)
        out.append("not"); out.append(f"if-goto END{k}")
        for s in n[2]: gen(s, out, slots, labels)
        out.append(f"goto TOP{k}"); out.append(f"label END{k}")
    else:
        gen(n[1], out, slots, labels); gen(n[2], out, slots, labels)
        out.append({"+": "add", "-": "sub", "*": "mul", "/": "div", "<": "lt", "<=": "le"}[n[0]])

def run_vm(code, nlocals):
    labels = {c.split()[1]: i for i, c in enumerate(code) if c.startswith("label")}
    stack, local, pc, steps = [], [0] * nlocals, 0, 0

```

```

ops = {"add": lambda a, b: a + b, "sub": lambda a, b: a - b, "mul": lambda a, b: a * b,
      "div": lambda a, b: a // b, "lt": lambda a, b: int(a < b), "le": lambda a, b: int(a
      <= b)}
while pc < len(code):
    c = code[pc].split(); pc += 1; steps += 1
    if c[0] == "push": stack.append(int(c[2]) if c[1] == "constant" else local[int(c[2])])
    elif c[0] == "pop": local[int(c[2])] = stack.pop()
    elif c[0] in ops: b, a = stack.pop(), stack.pop(); stack.append(ops[c[0]](a, b))
    elif c[0] == "not": stack.append(int(not stack.pop()))
    elif c[0] == "if-goto":
        if stack.pop(): pc = labels[c[1]]
    elif c[0] == "goto": pc = labels[c[1]]
return local, steps

src = "x = (2 + 3) * y - 4 * 2;"
print("tokens:", lex(src))
tree = Parser(lex(src)).program()
print("tree: ", tree[0])
print("folded:", fold(tree[0]))
for label, t in (("unoptimised", tree), ("folded", [fold(s) for s in tree])):
    code = []; gen(t[0], code, {"y": 0}, [])
    print(f"{label:12} {len(code)} VM commands: {'; '.join(code)}")

program = """
sum = 0;
i = 1;
while i <= 100 {
    sum = sum + i;
    i = i + 1;
}
"""
ast = Parser(lex(program)).program()
code, slots = [], {}
for s in ast: gen(s, code, slots, [])
local, steps = run_vm(code, len(slots))
print(f"\nthe loop compiles to {len(code)} VM commands; running them leaves sum =
{local[slots['sum']]}"
      f"after {steps:,} commands")
for line in code: print(f"    {line}")

```

OUTPUT

```

tokens: ['x', '=', '(', '2', '+', '3', ')', '*', 'y', '-', '4', '*', '2', ';']
tree:  ('let', 'x', ('-', ('*', ('+', 2, 3), 'y'), ('*', 4, 2)))
folded: ('let', 'x', ('-', ('*', 5, 'y'), 8))
unoptimised 10 VM commands: push constant 2; push constant 3; add; push local 0; mul; push
    constant 4; push constant 2; mul; sub; pop local 1
folded      6 VM commands: push constant 5; push local 0; mul; push constant 8; sub; pop local
    1

the loop compiles to 20 VM commands; running them leaves sum = 5050 after 1,511 commands
push constant 0
pop local 0
push constant 1
pop local 1
label TOP0
push local 1
push constant 100
le
not
if-goto END0
push local 0
push local 1
add
pop local 0
push local 1

```

```
push constant 1
add
pop local 1
goto TOP0
label END0
```

The compiled loop runs 1,511 VM commands, where chapter 8's hand-written version ran 1,411. The difference is the `not` the code generator emits before every loop test, 101 times, because it compiles "jump to the end if the condition is false" literally. A slightly smarter generator would test the inverse condition directly (exercise 16.3). Compilers are full of small decisions like that, and their accumulated effect is the difference between naive and optimised code.

Assembling, and beyond

Below the compiler, the VM translator turns each VM command into a fixed sequence of assembly instructions (chapter 8, exercise 8.3), and the assembler turns assembly into machine code with chapter 7's two passes. Real toolchains split the same work across separate programs. `clang -S` stops after code generation and shows the assembly, `clang -c` runs the assembler too and produces an object file, and the linker (chapter 17) finishes the job. **Just-in-time** compilers, in the JVM, in JavaScript engines such as V8, and in PyPy, run this pipeline while the program executes, compiling the hottest code using the types and values they actually observe.

COMMON TRAPS

Left recursion in a recursive-descent parser, which recurses for ever; use a loop, as the example's `expr` does.

Getting associativity wrong, so that $a - b - c$ parses as $a - (b - c)$.

Ambiguous grammars, such as the dangling else, resolved silently in a way the programmer didn't expect.

Optimisations that change meaning, usually because the program relied on undefined behaviour.

Treating every layer as free. Each translation adds instructions unless something optimises them away.

EXERCISES

EXERCISE 16.1 Left associativity

The example's `expr` parses with a loop: a term, then repeatedly an operator and another term. Why not write it as the recursive rule `expr → term "-" expr`? Work through $10 - 4 - 3$ both ways.

SOLUTION

The recursive rule groups to the right: it parses 10, then "-", then the whole rest as an expression, giving $10 - (4 - 3) = 9$. Subtraction is left-associative, so the right answer is $(10 - 4) - 3 = 3$. The loop builds the tree from the left, wrapping the tree so far as the left child of each new operator, which gives ('-', ('-', 10, 4), 3). The left-recursive rule `expr → expr "-" term` would describe the right grouping, but a recursive-descent parser can't use it directly, because `expr` would call itself before consuming anything and never stop. The loop is the standard fix.

EXERCISE 16.2 Generate by hand

Write the VM code the example's generator produces for `y = a * (b + c);`, with `a`, `b` and `c` in local slots 0, 1 and 2 and `y` in slot 3. How deep does the stack get?

SOLUTION

A post-order walk gives: `push local 0, push local 1, push local 2, add, mul, pop local 3`. The stack reaches depth 3, just before the `add`, holding `a`, `b` and `c`. In general the depth needed for an expression tree is bounded by its height. Evaluating the deeper subtree first minimises it, a choice register allocators make too, because each stack slot corresponds to a register in compiled code.

EXERCISE 16.3 Drop the not

Change the `while` code generation so that the loop no longer needs a `not` before every test. How many VM commands does the sum loop then execute?

SOLUTION

Generate the inverse comparison directly. For `i <= 100`, emit `gt` instead of `le` followed by `not`, so the jump to END happens when `i > 100`. Alternatively, put the test at the bottom of the loop, jumping back to the body while the condition holds, which also removes the unconditional `goto` from each iteration. The first change saves one command per test, 101 in all, bringing the loop to 1,410 commands, the same as chapter 8's hand-written VM code apart from the final push.

Optimising compilers do both routinely: inverted branches and bottom-tested loops are among the first things you'll notice when reading their output.

CHECKLIST

- ☐ Can you name the stages of a compiler and what each produces?
- ☐ Can you write a recursive-descent parser with correct precedence and associativity?
- ☐ Can you implement constant folding on a syntax tree?
- ☐ Can you generate stack-VM code for expressions and loops?
- ☐ Have you built the Nand2Tetris compiler, projects 10 and 11?

GO DEEPER

BOOK [Crafting Interpreters](#) Robert Nystrom

COURSE [Nand2Tetris projects 10 and 11: the compiler](#) Nisan & Schocken

REFERENCE [LLVM language reference: an industrial intermediate representation](#) LLVM project

Linking and loading

3 exercises

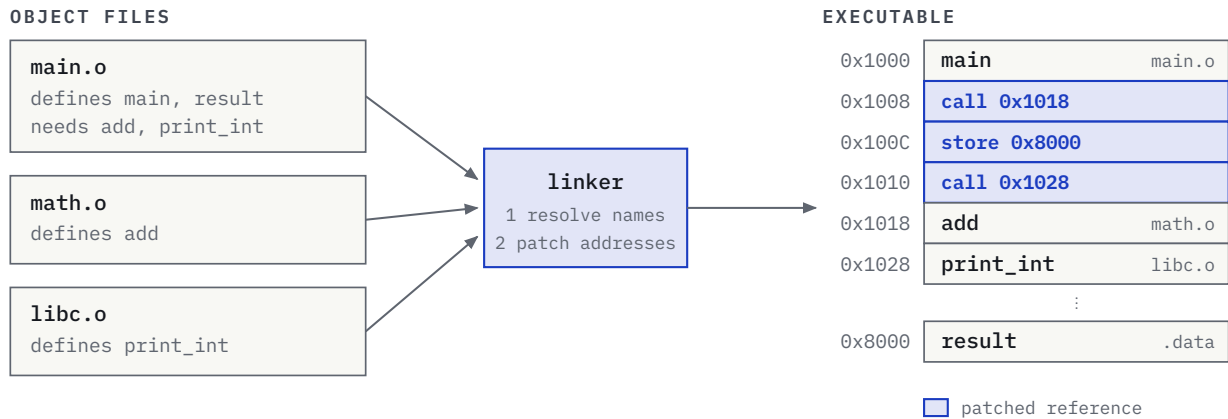
Real programs are compiled in pieces: one object file per source file, plus libraries compiled by someone else, years earlier. The linker combines them. It decides where every function and variable will live, matches each reference to exactly one definition, and patches the machine code with the final addresses. The loader then maps the result into memory and starts it. This chapter builds a toy linker that does all three steps and reports the two classic errors, then explains static and dynamic linking, how a process is laid out in memory, and why addresses are randomised on every run.

Object files

A compiler and assembler turn each source file into an **object file**: machine code that isn't yet runnable, because it refers to things whose addresses aren't known. An object file contains **sections**. `.text` holds the code, `.data` initialised global variables, `.rodata` constants, and `.bss` zero-initialised globals, which take no space in the file, only a size. It has a **symbol table** listing the names it defines and the names it uses but doesn't define, and **relocation entries** marking each place in the code that refers to a symbol and must be patched once its address is known. The formats differ by platform: ELF on Linux, Mach-O on macOS and PE/COFF on Windows. The tools to inspect them, `nm`, `objdump` and `readelf`, are worth learning.

What the linker does

Linking happens in two steps, much like chapter 7's assembler. **Symbol resolution** matches every undefined symbol in every input to exactly one definition elsewhere. **Relocation** lays the sections out at final addresses, computes every symbol's address, and patches each relocation entry with the right value.



Linking the example's three object files. Every name main.o uses but doesn't define is found in another input, every section gets a final address, and every reference in the code is patched with the address it names.

The two classic errors fall out of resolution. "Undefined reference" means some symbol has no definition in any input, usually a missing library on the command line, such as forgetting `-lm` for `sqrt`. "Duplicate symbol" or "multiple definition" means two inputs define the same name, often because a variable was defined, rather than declared, in a header included by several files. Static libraries, `.a` archives, are searched only for symbols still undefined at that point, which is why the order of libraries on a Unix linker command line matters.

Linked, by hand

The example's object files are lists of pseudo-instructions with symbol tables and relocation entries. The linker places the text sections from address 0x1000 and the data from 0x8000, resolves every reference, and patches the code. It then fails twice on purpose, and finally loads the result at three random base addresses.

EXAMPLE

linker.py · a two-step linker: layout, resolution and relocation, and loading with randomised addresses

```
import random

# Object files as an assembler would leave them: code, the symbols they define, the references
# to resolve
main_o = {"name": "main.o",
          "text": ["push 2", "push 3", "call add", "store result", "call print_int", "ret"],
          "data": {"result": 0},
          "defines": {"main": 0},
          "relocs": [(2, "add"), (3, "result"), (4, "print_int")]} # (instruction index,
                                                                    symbol)
math_o = {"name": "math.o",
          "text": ["load arg0", "load arg1", "add", "ret"],
          "data": {},
          "defines": {"add": 0},
          "relocs": []}
libc_o = {"name": "libc.o", "text": ["syscall write", "ret"], "data": {}, "defines":
```

```

{"print_int": 0}, "relocs": []}

def link(objects, text_base=0x1000, data_base=0x8000, instr_size=4):
    symbols, text, data, layout = {}, [], {}, []
    for obj in objects:
        collect definitions
        start = text_base + instr_size * len(text)
        layout.append((obj["name"], start, len(obj["text"])))
        for name, off in obj["defines"].items():
            if name in symbols: raise ValueError(f"duplicate symbol {name}")
            symbols[name] = start + instr_size * off
        for name in obj["data"]:
            symbols[name] = data_base + 8 * len(data); data[name] = obj["data"][name]
        obj["start_index"] = len(text); text += obj["text"]
    patched = list(text)
    for obj in objects:
        with its address
        for idx, name in obj["relocs"]:
            if name not in symbols: raise ValueError(f"undefined reference to {name} in {obj['name']}")
            k = obj["start_index"] + idx
            op = patched[k].split()[0]
            patched[k] = f"{op} 0x{symbols[name]:X}"
    return patched, symbols, layout

exe, symbols, layout = link([main_o, math_o, libc_o])
print("layout:", ", ".join(f"{n} at 0x{s:X} ({c} instructions)" for n, s, c in layout))
print("symbols:", {k: hex(v) for k, v in symbols.items()})
print("linked text:")
for i, ins in enumerate(exe): print(f"  0x{0x1000 + 4 * i:X}: {ins}")
try:
    link([main_o, math_o])
except ValueError as e:
    print(f"\nlinking without libc.o: error: {e}")
try:
    link([main_o, math_o, libc_o, {"name": "dup.o", "text": ["ret"], "data": {}, "defines": {"add": 0}, "relocs": []}])
except ValueError as e:
    print(f"linking a second definition of add: error: {e}")

# Loading with address space layout randomisation: the same program at a different base each run
random.seed(17)
print("\nloaded at random bases (ASLR): absolute addresses move, relative offsets don't:")
for run in range(3):
    base = random.randrange(0x5550_0000_0000, 0x5600_0000_0000, 0x1000)
    add_addr = base + (symbols["add"] - 0x1000)
    call_site = base + 4 * 2
    print(f"  run {run + 1}: base 0x{base:X}, add at 0x{add_addr:X}, call-site-relative offset {add_addr - call_site:+d}")

```

OUTPUT

```

layout: main.o at 0x1000 (6 instructions), math.o at 0x1018 (4 instructions), libc.o at 0x1028
(2 instructions)
symbols: {'main': '0x1000', 'result': '0x8000', 'add': '0x1018', 'print_int': '0x1028'}
linked text:
0x1000: push 2
0x1004: push 3
0x1008: call 0x1018
0x100C: store 0x8000
0x1010: call 0x1028
0x1014: ret
0x1018: load arg0
0x101C: load arg1
0x1020: add

```

```
0x1024: ret
0x1028: syscall write
0x102C: ret
```

```
linking without libc.o: error: undefined reference to print_int in main.o
linking a second definition of add: error: duplicate symbol add
```

```
loaded at random bases (ASLR): absolute addresses move, relative offsets don't:
run 1: base 0x55D5A0BCC000, add at 0x55D5A0BCC018, call-site-relative offset +16
run 2: base 0x55BA06E9A000, add at 0x55BA06E9A018, call-site-relative offset +16
run 3: base 0x559DAD298000, add at 0x559DAD298018, call-site-relative offset +16
```

Static and dynamic linking

Static linking copies the library code a program uses into the executable. The result is self-contained and starts fast, but every program carries its own copy, and fixing a bug in the library means relinking every program. **Dynamic linking** leaves library references unresolved until the program runs. Shared libraries, `.so` on Linux, `.dylib` on macOS and `.dll` on Windows, are loaded at run time and their code pages shared between all the processes that use them. A security fix to a shared library reaches every program at once. The costs are a little indirection on every call, through a **procedure linkage table** (PLT) and a **global offset table** (GOT) that the dynamic linker fills in, some start-up work, and a class of version-mismatch problems known on Windows as "DLL hell". Go and Rust link their own code statically by default, while most C and C++ programs link the system's C library dynamically.

Loading

To start a program, the operating system creates a fresh virtual address space and maps the executable's segments into it: code as read-only and executable, data as read-write. It reserves the bss and a stack, places the arguments and environment variables on the stack, and, for a dynamically linked program, maps the **dynamic linker**, `ld.so` on Linux, which loads the shared libraries and resolves their symbols. Control finally passes to the entry point, usually `_start` in the C runtime, which initialises the library and calls `main`. The resulting layout has code and data at the bottom, the **heap** above them growing upward as the program allocates, shared libraries and other mappings in the middle, and the **stack** near the top growing downward (chapter 18).

Address randomisation

If code always loads at the same address, an attacker who can overwrite a return address (chapter 18) knows exactly where to jump. **Address space layout randomisation** (ASLR) loads the executable, libraries, heap and stack at random addresses on every run, so a hard-coded address works only by luck. It requires **position-independent code**, which refers to its own functions and data relative to the current instruction's address rather than by absolute address. In the example, the base address changes on each run while the distance from `main`'s call site to `add` stays +16, so an instruction that encodes "call the function 16 bytes ahead" works wherever the code is loaded. Modern compilers produce position-independent executables (PIE) by default for exactly this reason.

COMMON TRAPS

Putting libraries before the objects that use them on a Unix linker command line.

Defining variables in headers, which produces duplicate symbols when the header is included twice.

Mixing library versions between build and deployment machines with dynamic linking.

Assuming fixed addresses for code, data or the stack. ASLR moves them every run.

Relying on the order of static initialisers across files in C++, which the language doesn't define.

EXERCISES

EXERCISE 17.1 Where does .bss go?

A program declares `static char buffer[100000000];` at file scope. Why doesn't the executable grow by 100 MB, and what happens when it runs?

SOLUTION

An uninitialised, or zero-initialised, global goes into `.bss`, and the object file records only its size, not 100 MB of zeros. When the program is loaded, the operating system reserves that much address space and maps it to zero-filled pages on demand. A page of physical memory is allocated only when the program first touches it. So the file stays small and the process uses memory only for the parts of the buffer it actually writes. Initialising the array with any non-zero value would move it to `.data` and put all 100 MB in the file.

EXERCISE 17.2 Read the error

Diagnose each linker error: "undefined reference to `sqrt'" when building a C program with `gcc main.c`, and "multiple definition of `counter'" in a program where `counter.h` contains `int counter = 0;`.

SOLUTION

The first: `sqrt` lives in the maths library, which on many Unix systems isn't linked by default. The compiler accepted the call because the header declared it, but no input defines it. Fix it with `gcc main.c -lm`,

with the library after the files that use it. The second: every file that includes `counter.h` gets its own definition of `counter`, so the linker sees several. Put `extern int counter;`, a declaration, in the header, and the definition `int counter = 0;` in exactly one source file. Both errors are about the difference between a declaration, which says a name exists, and a definition, which creates it.

EXERCISE 17.3 Why relative addressing survives relocation

Explain why an instruction that says "call the address 16 bytes after this one" needs no relocation when a library is loaded at a different base address, while "call address 0x1018" does. What does position-independent code do about global variables?

SOLUTION

Moving the whole library by some offset moves the calling instruction and the called function by the same amount, so the distance between them is unchanged and a PC-relative call stays correct, as the example's constant +16 shows. An absolute address names one fixed location, so it's wrong as soon as the code moves, and the loader would have to patch it, which would also make the page private to one process instead of shareable. For data whose address isn't known until load time, such as a variable in another shared library, position-independent code loads the address from the global offset table, which sits at a fixed distance from the code and which the dynamic linker fills in. One table entry per symbol is patched, rather than every instruction that uses it.

CHECKLIST

- ☐ Can you name the main sections of an object file and what goes in each?
- ☐ Can you explain symbol resolution and relocation?
- ☐ Can you diagnose undefined and duplicate symbol errors?
- ☐ Can you weigh static against dynamic linking?
- ☐ Can you describe a process's memory layout and explain ASLR and position-independent code?

GO DEEPER

BOOK Computer Systems: A Programmer's Perspective, chapter 7: linking Bryant & O'Hallaron

BOOK Linkers and Loaders John R. Levine

ARTICLE Position independent code in shared libraries Eli Bendersky

The stack and overflows

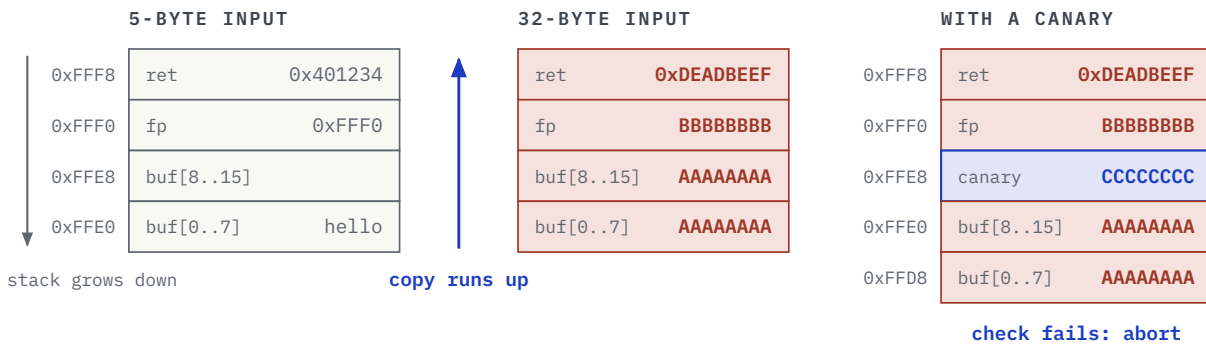
3 exercises

Every function call gets a frame on the stack: the address to return to, some saved registers, and the function's local variables. The stack is just memory and a pointer, which is why it's both fast and dangerous. Allocating a frame is one subtraction, but a local array sits a few bytes below the return address, and C's string functions don't check lengths. This chapter lays out a frame byte by byte, overflows it to take over the return address, watches a stack canary catch the same attack, and works out how deep recursion can go before the stack runs out.

Calls and frames

A call has to remember where to come back to. On x86 the `call` instruction pushes the return address onto the stack. On RISC-V, `jal` puts it in the register `ra` (chapter 6), and a function that makes calls of its own saves `ra` to the stack first. The function's **prologue** then saves the caller's frame pointer and any registers it promised to preserve, and allocates space for its locals by subtracting from the stack pointer. The **epilogue** undoes all of that, and `ret` jumps to the saved address. The rules for which registers hold arguments, which hold the return value, and which a function must preserve make up the **calling convention**. RISC-V passes the first eight arguments in `a0` to `a7` and returns in `a0`; on Linux and macOS, x86-64 uses `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`, and returns in `rax`. Further arguments go on the stack.

Two consequences follow. Stack allocation is almost free compared with the heap, where `malloc` has to search for a free block, which is one reason short-lived locals are fast. And a local variable stops existing when its function returns. Its bytes stay in memory until the next call reuses them, so a pointer to a local that outlives the call reads whatever the next frame put there.



One frame, three times, drawn to scale at one row per 8-byte word. The stack grows toward lower addresses, but a copy into the buffer runs toward higher ones, so too long an input overwrites the saved frame pointer and then the return address. A canary placed below them is overwritten first, and the function checks it before returning.

Overflowing a buffer

In C, `strcpy(buf, input)` copies until it finds the terminating zero byte, however big `buf` is. So do `gets`, which can't be used safely at all and was removed from C11, and `sprintf`. If the input comes from outside the program, its author decides what lands past the end of the buffer, and the return address starts only 8 bytes past its end. The example simulates a 64 KiB machine and runs the vulnerable function three times. A 5-byte input and a 16-byte one leave the frame intact. A 32-byte input fills the buffer, overwrites the saved frame pointer with the bytes `BBBBBBBB`, and replaces the return address with a value the attacker chose.

EXAMPLE

stack.py · a stack frame byte by byte: an overflow rewrites the return address, and a canary catches it

```
import struct, random

# A tiny machine: 64 KiB of memory, a stack growing down from the top, 8-byte words, little-endian
MEM = bytearray(65536)
def w64(addr, v): MEM[addr:addr + 8] = struct.pack("<Q", v)
def r64(addr): return struct.unpack("<Q", MEM[addr:addr + 8])[0]

def call_vulnerable(user_input, canary=None):
    """The caller pushes a return address; the callee saves the frame pointer, maybe a canary,
    and a 16-byte buffer."""
    sp = 0x10000
    sp -= 8; w64(sp, 0x401234); ret_slot = sp # return address into the caller
    sp -= 8; w64(sp, 0xFFFF0); fp_slot = sp # saved frame pointer
    if canary is not None: sp -= 8; w64(sp, canary); canary_slot = sp
    sp -= 16; buf = sp # char buf[16]
    MEM[buf:buf + len(user_input)] = user_input # strcpy(buf, input): no length check
    layout = f"buf at 0x{buf:X}, " + (f"canary at 0x{canary_slot:X}, " if canary is not None
    else "") + \
        f"saved fp at 0x{fp_slot:X}, return address at 0x{ret_slot:X}"
```

```

if canary is not None and r64(canary_slot) != canary:
    return layout, "stack smashing detected: the program aborts before returning"
return layout, f"returns to 0x{r64(ret_slot):X}"

print("frame:", call_vulnerable(b"hello")[0])
print(f"input of 5 bytes: {call_vulnerable(b'hello')[1]}")
print(f"input of 16 bytes: {call_vulnerable(b'A' * 16)[1]}")
attack = b"A" * 16 + b"B" * 8 + struct.pack("<Q", 0xDEADBEEF)           # fill buf, overwrite
                                fp, then the return address
print(f"input of 32 bytes: {call_vulnerable(attack)[1]}    <- the attacker chose this address")
random.seed(18)
canary = random.getrandbits(64) & ~0xFF                                # real canaries start
                                with a zero byte
attack_c = b"A" * 16 + b"C" * 8 + b"B" * 8 + struct.pack("<Q", 0xDEADBEEF)
print(f"with a stack canary, the same attack: {call_vulnerable(attack_c, canary)[1]}")

# How deep can recursion go? Frame size times depth against the stack limit
for frame_bytes in (48, 256, 4096):
    print(f"frames of {frame_bytes:5,} bytes: an 8 MiB stack holds about {8 * 1024 * 1024 //
        frame_bytes:,} nested calls")

```

OUTPUT

```

frame: buf at 0xFFE0, saved fp at 0xFFF0, return address at 0xFFF8
input of 5 bytes: returns to 0x401234
input of 16 bytes: returns to 0x401234
input of 32 bytes: returns to 0xDEADBEEF    <- the attacker chose this address
with a stack canary, the same attack: stack smashing detected: the program aborts before
    returning
frames of 48 bytes: an 8 MiB stack holds about 174,762 nested calls
frames of 256 bytes: an 8 MiB stack holds about 32,768 nested calls
frames of 4,096 bytes: an 8 MiB stack holds about 2,048 nested calls

```

In a real attack, the chosen address points at code the attacker supplied in the same input, or at existing code that does what they want. This is the oldest exploit technique in computing that still works. The Morris worm used it in 1988 against a `gets` call in the finger daemon, and Aleph One's 1996 article "Smashing the Stack for Fun and Profit" taught it to a generation.

Defences

A **stack canary** is a random value the prologue places between the locals and the saved registers, and that the epilogue checks before returning. An overflow from a buffer must pass through it to reach the return address, and the attacker doesn't know its value, so the check fails and the program aborts instead of jumping. GCC and Clang insert canaries with `-fstack-protector`, on by default in most Linux distributions. A **non-executable stack**, also called NX, DEP or W^X, makes the processor refuse to run instructions from stack pages, so injected code can't run. Attackers answered with **return-oriented programming**: chaining short fragments of the program's existing code, each ending in `ret`. That's where ASLR (chapter 17) comes in, since the fragments' addresses are no longer known in advance. The newest defences are in hardware. Intel's CET keeps a **shadow stack** of return addresses that ordinary stores can't write, and ARM's **pointer authentication**, used on Apple silicon, signs return addresses with a key held in the processor.

Each defence raises the cost of an attack, and none removes the bug. Microsoft and the Chromium project have each reported that about 70% of their serious security bugs are memory-safety errors. The lasting fix is code that can't write past the end of a buffer: bounds-checked languages such as Rust, Go, Java and Python, and in C and C++, functions that take the destination's size, such as `snprintf`, used correctly.

How deep can recursion go?

Each nested call adds a frame, and the stack has a fixed size. The main thread's default is 8 MiB on Linux and macOS, other threads often get less (512 KiB for secondary threads on macOS), and Windows defaults to 1 MiB. The example's last lines divide 8 MiB by the frame size: about 175,000 calls for a small 48-byte frame, but only 2,048 for a function with a 4 KiB local array. Below the stack sits an unmapped **guard page**, so running out is a crash, a segmentation fault, rather than silent corruption of the heap. Python sets its own recursion limit, 1,000 by default, well below the point where its interpreter's C stack would overflow. Recursion whose depth depends on the input, such as walking a long linked list or parsing deeply nested JSON, should be rewritten as a loop with an explicit stack on the heap.

COMMON TRAPS

Returning a pointer to a local variable. Its frame is gone as soon as the function returns.

Using `gets`, `strcpy` or `sprintf` on input whose length you don't control.

Forgetting the terminating zero byte when sizing a buffer for a string.

Putting large arrays on the stack, especially in threads with small stacks.

Recurring to a depth the input controls.

Assuming canaries and ASLR make an overflow harmless. They make it harder to exploit.

EXERCISES

EXERCISE 18.1 The off-by-one

In the example, a 16-byte input returned safely. Explain why `strcpy` with a 16-character string is still an overflow of `buf[16]` in C, and what it damages on this little-endian machine.

SOLUTION

`strcpy` also copies the terminating zero byte, so a 16-character string writes 17 bytes. The 17th lands at 0xFFFF0, the first byte of the saved frame pointer. On a little-endian machine that's its lowest byte, so the saved value 0xFFFF0 becomes 0xFF00. The function itself returns normally. But when the caller restores its frame pointer, it points 240 bytes too low, and the caller then reads its locals, and possibly its own return address, from the wrong place. One-byte zero overwrites like this have been exploited in practice. The fix is to size buffers for the string plus its terminator, and to copy with functions that take the destination's size.

EXERCISE 18.2 Will it fit?

A recursive function with a 256-byte frame walks a linked list of 100,000 nodes, one call per node. Does it fit in an 8 MiB stack? In a 512 KiB thread stack? What should the function do instead?

SOLUTION

It needs $100,000 \times 256$ bytes, about 25.6 MB, so it doesn't fit in either. The example shows an 8 MiB stack holds 32,768 frames of 256 bytes, and a 512 KiB stack holds a sixteenth of that, 2,048. The program crashes with a stack overflow at about node 32,000, or at about node 2,000 on the thread. A list walk needs no stack at all: rewrite it as a loop that follows the next pointer. For genuinely tree-shaped work, keep an explicit stack in a heap-allocated list, which can grow as large as memory allows. Raising the stack limit only moves the crash to a bigger input.

EXERCISE 18.3 What a canary misses

The example's canary has a zero lowest byte. Why does that help against `strcpy` overflows? Name two kinds of attack a canary doesn't stop.

SOLUTION

On a little-endian machine the lowest byte is stored first, so it's the first byte of the canary an overflow from the buffer reaches. To get past the canary without changing it, the input would have to contain a zero byte at that point, and `strcpy` stops copying at a zero byte. Canaries don't stop overflows that change other locals, such as a function pointer or a length, between the buffer and the canary; compilers reduce this by placing arrays directly below the canary. They don't stop writes that skip over it, such as `buf[i] = x` with an index the attacker controls, or

overflows on the heap. And if a separate bug leaks the canary's value, for instance a format-string bug, the attacker can write it back unchanged.

CHECKLIST

- ☐ Can you describe what a call, prologue, epilogue and return do to the stack?
- ☐ Can you state your platform's calling convention for arguments and return values?
- ☐ Can you explain, with addresses, how an overflow reaches the return address?
- ☐ Can you explain what canaries, non-executable stacks, ASLR and shadow stacks each stop?
- ☐ Can you estimate the maximum recursion depth from the frame size and the stack size?

GO DEEPER

ARTICLE [Smashing the Stack for Fun and Profit Aleph One, Phrack 49](#)

LAB [CS:APP Attack Lab: write your own overflow exploits Bryant & O'Hallaron](#)

ARTICLE [Memory safety The Chromium Projects](#)

PART VI

The machine at work

Two chapters on the machine as a whole: how the processor exchanges data with the devices around it, from keyboards to network cards and SSDs, without spending all its time waiting for them; and how to predict a program's performance with a few lines of arithmetic, and so know which optimisation is worth making before you make it.

I/O, interrupts and DMA

3 exercises

A processor talks to a device by reading and writing the device's registers. The hard part is knowing when something has happened. The processor can keep asking, which is polling. The device can interrupt it. Or the device can copy its data into memory by itself, which is direct memory access (DMA), and interrupt only when a batch is done. This chapter prices all three for a keyboard, a mouse, a 10 Gb/s network card and an NVMe SSD. It shows why fast devices need DMA, and why at the highest rates the answer turns back to polling.

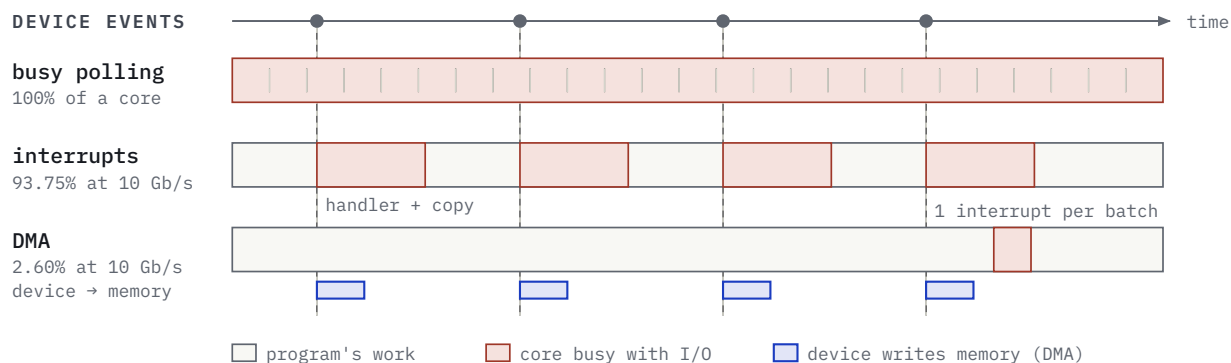
Talking to devices

Most devices are reached by **memory-mapped I/O**. Their control and status registers appear at physical addresses, and ordinary loads and stores to those addresses go to the device instead of memory, just as the Hack computer's screen and keyboard did in chapter 7. A store to a network card's register might mean "send the packet described here", and a load from its status register reports whether one has arrived. Those addresses must never be cached, and accesses to them must not be reordered or optimised away, so the page tables mark them uncached and drivers access them through `volatile` pointers or helpers such as Linux's `readl` and `writel`. In a modern computer most devices sit on PCI Express, and the code that knows a particular device's registers is its **driver**.

Polling and interrupts

Polling means reading the status register in a loop until it says data is ready. Each check costs only tens of cycles, but the core does nothing else, so polling a keyboard that produces ten events a second wastes a whole core. With **interrupts**, the device raises a signal when it needs attention. The processor finishes its current instruction, saves the program counter and some state, switches to kernel mode, and jumps to the handler registered for that interrupt in a table of handlers, the **interrupt vector table**. The handler acknowledges the device, does the minimum, schedules any longer work for later, and returns, and the interrupted program continues as if nothing had happened. The same mechanism handles **exceptions**, such as a page fault or a division by zero, and **system calls**, where a program deliberately traps into the kernel. A timer interrupt, a few hundred or a thousand times a second, is what lets the operating system take the processor back from a running program and schedule another.

An interrupt isn't free. Saving state, switching modes and running the handler costs around a microsecond, roughly 3,000 cycles, and the handler's code and data push the program's out of the caches. Even with interrupts, if the processor copies every byte from the device itself, a fast device keeps it busy.



One core, four device events, three ways of handling them. The percentages are the example's figures for a 10 Gb/s network card receiving 1,500-byte packets. Here the batch is four events; the example's is 32.

Direct memory access

With **DMA**, the device copies the data itself. The driver sets up a **ring** of descriptors in memory, each pointing at an empty buffer. The device writes each incoming packet or disk block into the next buffer, marks the descriptor as filled, and raises an interrupt only after a batch of them, or after a short timeout. This is **interrupt coalescing**. The processor never touches the data until it's ready to use it. Two practical details matter. On x86 machines, DMA is cache-coherent: the hardware makes sure the caches don't hold stale copies of buffers a device has written. On many ARM systems the driver must flush or invalidate the caches itself. And because a device that can write anywhere in memory can also overwrite the kernel, an **IOMMU** translates and restricts the addresses each device may use, the way page tables restrict programs.

The example prices one second of each device. Busy polling costs a whole core whatever the device, which is why it's worth it only when events never stop. With interrupts, the keyboard and mouse cost almost nothing, and their DMA column shows only that at low rates the choice doesn't matter; nobody batches keystrokes. The network card, with an interrupt and a copy per packet, takes 94% of a core. The SSD would need more than a whole core. DMA with one interrupt per 32 events brings both down to about 3%.

EXAMPLE

`io.py` · the CPU cost of busy polling, interrupts with copying, and DMA, for four devices

```
# How much CPU time does it take to move data in from a device? Three ways, for four devices.
GHZ = 3e9                                # cycles per second of one core
INTERRUPT = 3_000                        # cycles to take an interrupt, run the handler and
    return (about 1 us)
COPY_PER_BYTE = 0.25                     # cycles per byte for the CPU to copy data itself

devices = [ # name, events per second, bytes per event
    ("keyboard", 10, 8),
    ("mouse", 1_000, 8),
    ("10 Gb/s network, 1500-byte packets", 10e9 / 8 / 1500, 1500),
    ("NVMe SSD, 4 KiB reads at 1M IOPS", 1_000_000, 4096),
```

```

]
def busy_polling(rate, size):      # a core spends all its time checking: the whole core,
    whatever the rate
    return 1.0
def interrupts(rate, size):        # each event interrupts the CPU, which then copies the data
    return rate * (INTERRUPT + size * COPY_PER_BYTE) / GHZ
def dma(rate, size, batch=32):    # the device copies into memory itself and interrupts once
    per batch
    return rate / batch * INTERRUPT / GHZ

print(f"{'device':38} {'events/s':>11}    busy polling    interrupts + copy    DMA, 1 interrupt
per 32")
for name, rate, size in devices:
    irq = interrupts(rate, size)
    shown = f"{irq:.2%}" if irq <= 1 else f"{irq:.0%} (over 1 core)"
    print(f"{'name':38} {'rate':11,.0f}    {busy_polling(rate, size):>11.0%}    {shown:>16}
        {dma(rate, size):>14.2%}")

# Interrupt storms: with one interrupt per packet, a core saturates; coalescing and polling
under load help
for pps in (100_000, 1_000_000, 3_000_000):
    load = pps * INTERRUPT / GHZ
    print(f"{'pps':>9,} packets/s with one interrupt each: {min(load, 1):.0%} of a core" + ("
        (livelock: no time left for work)" if load >= 1 else ""))

```

OUTPUT

device	events/s	busy polling	interrupts + copy	DMA, 1
interrupt per 32				
keyboard	10	100%	0.00%	
0.00%				
mouse	1,000	100%	0.10%	
0.00%				
10 Gb/s network, 1500-byte packets	833,333	100%	93.75%	
2.60%				
NVMe SSD, 4 KiB reads at 1M IOPS	1,000,000	100%	134% (over 1 core)	
3.12%				
100,000 packets/s with one interrupt each: 10% of a core				
1,000,000 packets/s with one interrupt each: 100% of a core				(livelock: no time left for work)
3,000,000 packets/s with one interrupt each: 100% of a core				(livelock: no time left for work)

When interrupts are too many

The last lines of the output show the failure that made networking engineers rethink interrupts. At a million packets a second with one interrupt each, handling interrupts takes the whole core. The handler runs, receives the packet, and before the program can process it the next interrupt arrives. The machine is fully busy and gets nothing done. This is **receive livelock**, described by Mogul and Ramakrishnan in 1996. The answer was to go back to polling, but polling memory rather than device registers. Linux's **NAPI** takes one interrupt, switches it off, and polls the DMA ring until it's empty, switching interrupts back on only when traffic falls. Frameworks such as **DPDK**, for networking, and **SPDK**, for storage, go further: they give a device to a user-space program that polls its rings on dedicated cores and never takes an interrupt. Interrupts are best when events are rare, and polling a DMA ring is best when they never stop.

Coalescing trades latency for throughput. An interrupt per batch of 32 means the first packet waits for 31 more, or for the timeout. Network cards and storage drivers let the timeout and batch size be tuned, and latency-sensitive systems such as trading platforms set them low or poll.

COMMON TRAPS

Reading a device register through an ordinary pointer. The compiler may read it once and loop forever on the cached value.

Doing slow work in an interrupt handler. Acknowledge the device and defer the rest.

Assuming DMA buffers are coherent with the caches on every platform.

One interrupt per event at high rates, which ends in livelock.

Tuning coalescing for throughput on a system that needs low latency, or the reverse.

EXERCISES

EXERCISE 19.1 Coalescing and latency

A network card raises one interrupt per 32 packets or 50 μ s after the first packet of a batch, whichever comes first. At 1,000 packets a second, evenly spaced, how long can a packet wait, and how many interrupts are there per second? At 833,333 packets a second? Use 3,000 cycles per interrupt on a 3 GHz core.

SOLUTION

At 1,000 packets a second, packets are 1 ms apart, so 32 of them would take 32 ms. The 50 μ s timer always fires first, with one packet in the batch. Every packet waits up to 50 μ s, there are 1,000 interrupts a second, and they cost $1,000 \times 3,000$ cycles, 0.1% of the core. At 833,333 packets a second, packets are 1.2 μ s apart, so the 32nd packet of a batch arrives about 37 μ s after the first, before the timer. The first packet in each batch waits about 37 μ s, there are about 26,000 interrupts a second, and they cost 2.6% of the core, matching the example's DMA column. At low rates the timeout sets the latency, and at high rates the batch size does.

EXERCISE 19.2 The loop that never ends

A driver waits for a device with `while (*status == 0) {}`, where `status` is an ordinary `uint32_t *` pointing at the device's status register. Give two reasons it might loop forever after the device sets the register to 1.

SOLUTION

First, the compiler sees no store to `*status` inside the loop, so it may load the value once and test the same register copy forever. Declaring the pointer `volatile`, or using the kernel's `readl`, forces a real load on every iteration. Second, if the register's address were mapped as ordinary cacheable memory, the loads would hit the cache and never reach the device. Device memory must be mapped uncached, which the operating system's functions for mapping device registers, such as Linux's `ioremap`, do. A related rule applies to writes: when a driver fills in a descriptor and then writes a "go" register, a barrier must make sure the descriptor reaches memory before the device is told to read it.

EXERCISE 19.3 Choose the method

Choose polling, interrupts, or DMA with interrupts or polling for each, and justify it: a keyboard; a 100 Gb/s network card in a packet-processing appliance; a temperature sensor a microcontroller reads once a second; an NVMe SSD in a database server.

SOLUTION

The keyboard: interrupts. Events are rare, and polling would keep a core and its power use busy for nothing. The 100 Gb/s card: DMA with polling on dedicated cores, as DPDK does. With minimum-size 64-byte frames that's about 148.8 million packets a second, under 7 ns each, far too fast for interrupts. The sensor: polling, driven by a timer. Read the register once a second and go back to sleep; nothing is gained by having the sensor interrupt. The SSD: DMA with coalesced interrupts, which is what the NVMe protocol is built around, or polled completion queues, for example through `io_uring`, when the lowest latency is worth a core.

CHECKLIST

- ☐ Can you explain memory-mapped I/O and why device registers must be uncached and volatile?
- ☐ Can you describe what happens, step by step, when an interrupt arrives?

- ☐ Can you explain how a DMA descriptor ring works?
- ☐ Can you estimate the CPU cost of an I/O method from the event rate and per-event cost?
- ☐ Can you explain receive livelock and why NAPI and DPDK poll?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapter 36: I/O devices Arpaci-Dusseau & Arpaci-Dusseau

DOCS NAPI The Linux kernel documentation

PAPER Eliminating receive livelock in an interrupt-driven kernel Mogul & Ramakrishnan, USENIX 1996

Performance as arithmetic

3 exercises

Most performance questions can be answered, or at least bounded, before any code is written, with three formulas and a handful of numbers. The iron law splits running time into instruction count, cycles per instruction and clock period, and so says which of the earlier chapters' techniques can help. The roofline says whether a loop is limited by arithmetic or by memory, and how fast it can possibly go. Amdahl's law, applied to a profile, says which part is worth speeding up. This chapter applies all three, then turns to measurement, which checks the arithmetic.

The iron law

A program's running time is the product of three factors:

$$\text{time} = \text{instructions} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}.$$

Different things change each factor. The instruction count depends on the algorithm, the compiler and the instruction set; SIMD (chapter 15) cuts it by doing several operations per instruction. Cycles per instruction, or CPI, depends on how well the program suits the microarchitecture: cache misses (chapters 9 to 11), branch mispredictions (chapter 13) and dependency chains (chapter 14) all raise it. The clock period is set by circuit design and power (chapters 3 and 12). The factors interact, which is why the law is worth writing down. In the example, a better compiler removes 20% of the instructions, but the remaining ones are slower on average, and the gain is 1.18 times rather than 1.25. A cache-friendly data layout that leaves the instruction count alone but cuts CPI from 1.6 to 0.9 gains 1.78 times, much more than a 20% faster clock.

Hardware people often quote the inverse, instructions per cycle or IPC. A wide out-of-order core running well-behaved code retires three or four instructions per cycle, and the same core chasing pointers through memory retires a fraction of one. On Linux, `perf stat` reports a program's IPC along with its cache and branch misses, and is the quickest way to see which factor is the problem. Comparing CPI or clock rates between different instruction sets or different programs says little, because the instruction counts differ.

EXAMPLE

`perf.py` · the iron law, a roofline, read times, and Amdahl's law applied to a profile

```
import math
```

```
# 1. The iron law: time = instructions x cycles per instruction x seconds per cycle
```

```

def seconds(instructions, cpi, ghz): return instructions * cpi / (ghz * 1e9)
variants = [("original", 10e9, 1.6, 3.0), ("better compiler: 20% fewer instructions", 8e9, 1.7, 3.0),
            ("cache-friendly layout: lower CPI", 10e9, 0.9, 3.0), ("faster clock, same code", 10e9, 1.6, 3.6)]
print("the iron law of performance:")
base = None
for name, n, cpi, ghz in variants:
    t = seconds(n, cpi, ghz); base = base or t
    print(f"    {name:42} {n / 1e9:4.0f}G instructions x CPI {cpi} / {ghz} GHz = {t:5.2f} s ({base / t:.2f}x)")

# 2. The roofline: attainable speed is limited by compute or by memory bandwidth, whichever binds first
PEAK, BW = 200e9, 50e9 # 200 GFLOP/s of arithmetic, 50 GB/s of memory bandwidth
ridge = PEAK / BW
print(f"\nroofline for a machine with {PEAK / 1e9:.0f} GFLOP/s and {BW / 1e9:.0f} GB/s (ridge at {ridge:.0f} FLOP per byte):")
kernels = [("vector add c = a + b", 1, 12), ("dot product", 2, 8), ("5-point stencil", 6, 16), ("matrix multiply, naive", 2, 16), ("matrix multiply, blocked", 2 * 64, 16)]
for name, flops, bytes_ in kernels: # per element: arithmetic and memory traffic
    ai = flops / bytes_
    attainable = min(PEAK, ai * BW)
    bound = "memory-bound" if ai < ridge else "compute-bound"
    print(f"    {name:26} {ai:7.3f} FLOP/byte -> at most {attainable / 1e9:6.1f} GFLOP/s ({bound})")

# 3. Back-of-the-envelope: how long does it take to read 1 GB?
print("\nreading 1 GB:")
for where, rate in (("from L3-resident data, ~200 GB/s", 200e9), ("from DRAM, ~50 GB/s", 50e9), ("from an NVMe SSD, ~5 GB/s", 5e9), ("from a hard disk, ~200 MB/s", 200e6), ("over a 1 Gb/s network", 125e6)):
    s = 1e9 / rate
    print(f"    {where:36} {s * 1000:9.1f} ms")

# 4. Speeding up one part: Amdahl again, applied to a profile
profile = {"parse input": 0.15, "compute": 0.70, "write output": 0.15}
for part, factor in (("compute", 4), ("compute", 1e9), ("parse input", 4)):
    new = sum(f / factor if p == part else f for p, f in profile.items())
    print(f"making '{part}' {'infinitely' if factor > 1e6 else f'{factor:g}x'} faster: whole program {1 / new:.2f}x faster")

```

OUTPUT

```

the iron law of performance:
original                                10G instructions x CPI 1.6 / 3.0 GHz =  5.33 s
(1.00x)
better compiler: 20% fewer instructions  8G instructions x CPI 1.7 / 3.0 GHz =  4.53 s
(1.18x)
cache-friendly layout: lower CPI        10G instructions x CPI 0.9 / 3.0 GHz =  3.00 s
(1.78x)
faster clock, same code                 10G instructions x CPI 1.6 / 3.6 GHz =  4.44 s
(1.20x)

roofline for a machine with 200 GFLOP/s and 50 GB/s (ridge at 4 FLOP per byte):
vector add c = a + b      0.083 FLOP/byte -> at most    4.2 GFLOP/s (memory-bound)
dot product               0.250 FLOP/byte -> at most   12.5 GFLOP/s (memory-bound)
5-point stencil           0.375 FLOP/byte -> at most   18.8 GFLOP/s (memory-bound)
matrix multiply, naive     0.125 FLOP/byte -> at most    6.2 GFLOP/s (memory-bound)
matrix multiply, blocked   8.000 FLOP/byte -> at most  200.0 GFLOP/s (compute-bound)

reading 1 GB:
from L3-resident data, ~200 GB/s      5.0 ms
from DRAM, ~50 GB/s                  20.0 ms

```

```

from an NVMe SSD, ~5 GB/s          200.0 ms
from a hard disk, ~200 MB/s        5,000.0 ms
over a 1 Gb/s network              8,000.0 ms
making 'compute' 4x faster: whole program 2.11x faster
making 'compute' infinitely faster: whole program 3.33x faster
making 'parse input' 4x faster: whole program 1.13x faster

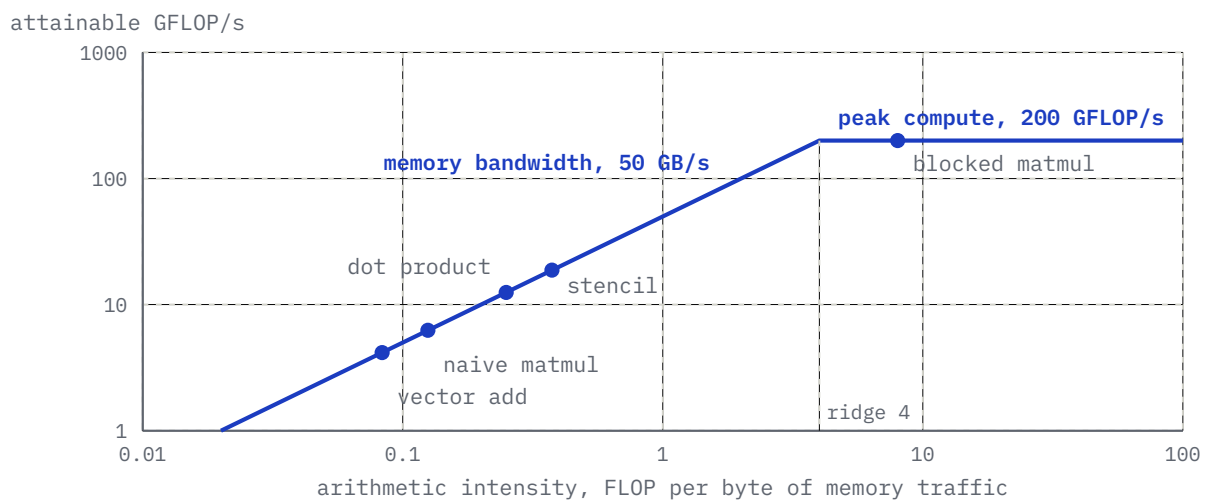
```

The roofline

A loop needs both arithmetic and data. Its **arithmetic intensity** I is the number of floating-point operations it does per byte it moves to or from main memory. A machine has a peak arithmetic rate P_{peak} and a memory bandwidth B , and the loop can go no faster than whichever limit it hits first:

$$P = \min(P_{\text{peak}}, I \times B).$$

Plotted on log-log axes, that's a sloped line, the bandwidth limit, meeting a flat one, the peak, at the **ridge point** P_{peak}/B . The example's machine, with 200 GFLOP/s and 50 GB/s, has its ridge at 4 FLOP per byte. Real CPUs have ridges between roughly 5 and 15, and GPUs often higher still, so most simple loops are on the slope. A vector add does one addition per 12 bytes moved, three 4-byte floats, and can't exceed 4.2 GFLOP/s on a machine that could do 200. Wider SIMD or more cores make no difference to a loop on the slope. Only reducing its memory traffic does: blocking (chapter 11), smaller data types, or fusing loops so that data is used several times while it's in the cache.



The example's five loops on the roofline of a machine with 200 GFLOP/s and 50 GB/s. Naive and blocked matrix multiply do the same arithmetic; blocking cuts the memory traffic per operation 64 times, which moves the loop past the ridge, from 6.2 to 200 GFLOP/s.

Matrix multiplication is the chapter 11 result seen on the roofline. The naive loop loads two values for every multiply-add, an intensity of 0.125. Blocking reuses each loaded value 64 times, which raises the intensity to 8 and moves the loop past the ridge, where it's limited

by arithmetic. That's also why matrix multiplication is the workload GPUs and AI accelerators are built around: it's one of the few important computations with enough reuse to use all their arithmetic.

Numbers to know

Estimates need a few numbers, known to within a factor of two: chapter 9's latencies, and the bandwidths in the example's third section. Reading a gigabyte takes about 20 ms from memory, 200 ms from an NVMe SSD, 5 s from a hard disk and 8 s over a 1 Gb/s network. Multiplying such numbers out answers design questions in a minute. A request that must read 1 GB from memory can't be served more than about 50 times a second per memory system, and a full scan of a 1 TB table on one SSD takes over three minutes however clever the query is. Throughput and latency connect through **Little's law** (chapters 9 and 14): the number of operations in flight equals throughput times latency, so a disk with a 100 μ s latency needs 100 outstanding requests to deliver a million operations a second.

Amdahl on a profile

Before speeding anything up, find out where the time goes. A **sampling profiler** such as `perf record` on Linux, Instruments on macOS or `py-spy` for Python interrupts the program thousands of times a second and records what it was running, which gives each function's share of the time with little overhead. Amdahl's law (chapter 15) then gives the ceiling. In the example's profile, computing takes 70% of the time. Making it four times faster speeds the whole program up 2.11 times, and making it infinitely fast only 3.33 times, $1/(1 - 0.7)$. Making parsing, 15% of the time, four times faster gains 1.13 times. Effort spent on a small part of the profile is mostly wasted, however good the optimisation.

Measurement has its own traps. Run several times and report the median, since the first run pays for cold caches and the rest vary with whatever else the machine is doing. Clock rates change with temperature and power: a laptop on battery or a core that has been busy for a minute runs slower than one that just woke up. Measure the work you care about at a realistic size, because a benchmark that fits in the cache tells you nothing about one that doesn't. The method that works is to estimate the bound first, measure, and compare. A loop running at 10% of its roofline has a problem worth finding, while one at 80% needs a different algorithm or a different machine, not more tuning.

COMMON TRAPS

Optimising before profiling, and so speeding up a part that takes 5% of the time.

Measuring once, with cold caches, on a machine whose clock rate is changing.

Comparing CPI or GHz across instruction sets or across programs.

Adding SIMD or cores to a memory-bound loop. Reduce its traffic instead.

Treating spec-sheet peaks as achievable. Measure sustained bandwidth, for example with the STREAM benchmark.

EXERCISES

EXERCISE 20.1 Fewer but slower

An optimisation removes 30% of a program's instructions but raises its CPI from 1.2 to 1.5. The clock rate is unchanged. Is the program faster or slower, and by how much?

SOLUTION

With the clock rate fixed, time is proportional to instructions times CPI. Before, that's $1 \times 1.2 = 1.2$; after, $0.7 \times 1.5 = 1.05$. The program is faster by $1.2/1.05 \approx 1.14$ times, far less than the 1.43 times that removing 30% of the instructions would suggest on its own. The change probably replaced many simple instructions with fewer complex or longer-latency ones, or made memory accesses less regular. Without the CPI figure, from `perf stat` for example, the instruction count alone would have been misleading.

EXERCISE 20.2 A roofline for daxpy

On the example's machine, a loop computes `y[i] = a * x[i] + y[i]` for 10^8 doubles. What's its arithmetic intensity, the fastest it can run, and the minimum time? Would 512-bit SIMD help? What would?

SOLUTION

Each element does 2 FLOPs, a multiply and an add, and moves 24 bytes: it reads `x[i]` and `y[i]` and writes `y[i]`, 8 bytes each. The intensity is $2/24 \approx 0.083$ FLOP per byte, far below the ridge at 4, so the loop is memory-bound at $0.083 \times 50 = 4.2$ GFLOP/s. The minimum time is the traffic over the bandwidth: 2.4 GB at 50 GB/s, 48 ms. Wider SIMD doesn't help, because arithmetic was never the limit. What helps is moving fewer bytes: single precision halves the traffic and the time, and fusing this loop with the next one that reads `y` avoids writing it out and reading it back.

EXERCISE 20.3 Find the bottleneck

A nightly job reads a 500 GB log from an NVMe SSD at 5 GB/s, parses it at 1 GB/s per core on 8 cores, and sends 50 GB of results over a 1 Gb/s network. The three stages can overlap. What's the shortest possible running time, and what would you change first?

SOLUTION

Reading takes $500/5 = 100$ s. Parsing runs at 8 GB/s, so it takes 62.5 s. Sending 50 GB at 125 MB/s takes 400 s. With the stages overlapped, the job can't finish faster than its slowest stage, so the minimum is 400 s, and the network is the bottleneck. Buying faster SSDs or more cores would change nothing. Change the network first: compressing the output four times brings it to 100 s, and a 10 Gb/s link to 40 s. Either way, reading at 100 s becomes the new bottleneck. That's Amdahl's law again: once one stage is fixed, the next slowest sets the time.

CHECKLIST

- ☐ Can you state the iron law and say which techniques change each factor?
- ☐ Can you compute a loop's arithmetic intensity and place it on a roofline?
- ☐ Can you estimate read times and throughput limits from a few remembered numbers?
- ☐ Can you use a profile and Amdahl's law to decide what to optimise?
- ☐ Can you benchmark without being fooled by cold caches, variance or clock changes?

GO DEEPER

PAPER Roofline: an insightful visual performance model for multicore architectures Williams, Waterman & Patterson

GUIDE Linux perf examples Brendan Gregg

BOOK Computer Organization and Design, RISC-V edition, chapter 1: performance Patterson & Hennessy

The stage asks for two things: the complete Nand2Tetris stack, from Nand gates to a compiler and operating system for the Hack computer, and one slow program made at least four times faster using only cache locality. Every chapter here prepares for one or both.

The Nand2Tetris stack

The course's twelve projects are on nand2tetris.org, with the hardware simulator, CPU emulator and test scripts for each. The table shows which chapters prepare for which project. Read the chapter, run its example, then build the project.

PROJECT	WHAT YOU BUILD	CHAPTERS
1 · Boolean logic	Not, And, Or, Xor, Mux and DMux, with their 16-bit and multi-way forms, all from Nand	Ch. 3
2 · Boolean arithmetic	half and full adders, Add16, Inc16 and the ALU	Ch. 1, 4
3 · Memory	a bit, a register, RAM8 up to RAM16K, and the program counter	Ch. 5
4 · Machine language	multiplication and a screen-filling program in Hack assembly	Ch. 6, 8
5 · Computer architecture	the Memory, CPU and Computer chips	Ch. 7
6 · Assembler	a translator from Hack assembly to binary	Ch. 7, 16
7 · VM I: stack arithmetic	the VM translator's arithmetic and memory commands	Ch. 8
8 · VM II: program control	branching, and function call and return	Ch. 8, 18
9 · High-level language	a program of your own in Jack	Ch. 16
10 · Compiler I: syntax analysis	the Jack tokenizer and parser	Ch. 16
11 · Compiler II: code	the code generator, producing VM code	Ch. 16,

12 · Operating system

the Math, Memory, Screen, Output, Keyboard, String, Array and Sys libraries

Ch. 1, 7, 17–19

Six habits make the build go smoothly:

- 01 Build the projects in order. Each one uses the chips or programs of the one before.
- 02 Run the supplied test scripts on every chip before moving on, and keep them passing.
- 03 Draw a chip on paper, with its internal wires named, before writing its HDL.
- 04 When a mechanism confuses you, simulate it in a few lines of Python first, as this volume's examples do.
- 05 Write the assembler, VM translator and compiler as small stages, each with its own tests.
- 06 In every software project, read the code your tool generates and step through it in the CPU emulator at least once.

The locality project

Take one slow program and make it at least four times faster by changing only how it uses the caches: the order in which it touches memory and the layout of its data. Keep the algorithm and the amount of arithmetic the same. Any of these works:

PROGRAM	WHY IT'S SLOW	FIX TO TRY	CHAPTER
Matrix multiply in i-j-k order	walks down a column, using one element per cache line	loop interchange to i-k-j, then blocking	Ch. 11
Matrix or image transpose	reads along rows and writes down columns	tiles that fit in L1	Ch. 11
Particle update over an array of structures	loads every field to use two	a structure of arrays	Ch. 11

Linked list or tree of small heap nodes	every hop is a dependent cache miss	nodes in one array, linked by index	Ch . 11, 14
Hash table with chaining	a miss for the bucket and another for each node	open addressing in one flat array	Ch . 10, 11
Grid with a power-of-two width, walked by column	conflict misses: the rows map to the same few sets	walk by row, or pad each row	Ch . 10
Per-thread counters in one array	false sharing: the threads fight over one line	one cache line per counter	Ch . 11, 15

Work through it in seven steps:

- 01 Write the program in C, C++ or Rust and compile it with optimisation on. Python's interpreter overhead hides the effects you're looking for.
- 02 Make the data at least four times the size of the last-level cache, and compute a checksum of the result to compare after every change.
- 03 Estimate first: count the bytes the program must move and the cache lines it actually touches, and work out the memory-bound time (chapter 20).
- 04 Measure the baseline: the median of at least five runs, and the cache misses from `perf stat`, Instruments or VTune.
- 05 Change one thing at a time, whether a loop order, a block size or a layout, and measure again the same way.
- 06 Explain each speed-up in terms of lines, sets and strides, and check that the miss counts fell by about as much as your explanation predicts.
- 07 Write it up: the baseline, each change with its time and misses, your estimate, and what you would try next.

Matrix multiplication is the safest choice. On large matrices, loop interchange alone often gives four times or more, and blocking adds to it. Turning a pointer-based structure into arrays teaches the most, because the fix is a redesign of the data rather than a reordering of loops. If your estimate and your measurements disagree, find out why before writing it up. That's the habit this whole volume is built on.

THE TWENTY RULES

One per chapter, each linked back to it. Read them before you choose a data type, trust a benchmark, or decide what to optimise.

- 01 Know the width and signedness of every integer, and decide in advance what happens when it overflows.
- 02 Never compare floats with `==`, never store money in them, and sum carefully when the terms differ in size.
- 03 Any logic can be built from NAND alone; judge a circuit by its depth as well as its gate count.
- 04 An adder is as fast as its carry chain, and subtraction is addition of the inverted operand plus one.
- 05 State changes only on the clock edge, and the slowest path between two registers sets the clock.
- 06 An instruction set is a contract, not a speed; know its formats, and remember that immediates are sign-extended.
- 07 A CPU is fetch, decode and execute in a loop, and an assembler is two passes and a symbol table.
- 08 Read the optimised assembly before believing anything about what the machine does.
- 09 Ask how far away the data is: a nanosecond in L1, a hundred in main memory, millions on a disk.
- 10 Know your cache's line size and sets, and keep hot data off large power-of-two strides.
- 11 Walk memory in the order it's stored, block what you reuse, and lay data out the way it's read.
- 12 A pipeline overlaps instructions but doesn't shorten them; dependences and taken branches cost stalls.

-
- 13 Make hot branches predictable or remove them, and remember that speculation leaves traces in the cache.
-
- 14 Speed comes from independent work, so split long dependency chains into several shorter ones.
-
- 15 Parallel hardware pays only for parallel work; measure the serial fraction before adding lanes or cores.
-
- 16 A compiler is a pipeline of small translations through a tree; check what it produced before optimising by hand.
-
- 17 Every reference needs exactly one definition: declare in headers, define once, and put libraries after their users.
-
- 18 Never copy into a buffer without knowing its size; defences slow attackers down, but only bounds checks stop them.
-
- 19 Use interrupts for rare events and DMA for bulk data, and poll a ring when the events never stop.
-
- 20 Estimate the bound, measure, and compare; optimise only what the profile and the roofline say matters.
-

READING LIST

The books, courses and papers behind this volume, and the chapters each one deepens. If you read only one book, make it *Computer Systems: A Programmer's Perspective*, which covers most of this volume from the programmer's side. If you do only one thing, build Nand2Tetris, because having built every layer of a computer yourself is what turns this stage's facts into a working model.

WORK	AUTHOR	DEEPENS
The Elements of Computing Systems, and Nand2Tetris	Nisan & Schocken	Ch. 3–8, 16
Computer Systems: A Programmer's Perspective	Bryant & O'Hallaron	Ch. 1–2, 9–11, 14, 17–19
Computer Organization and Design, RISC-V edition	Patterson & Hennessy	Ch. 4, 6, 10, 12, 15, 20
Computer Architecture: A Quantitative Approach	Hennessy & Patterson	Ch. 12–15, 20
Code: The Hidden Language of Computer Hardware and Software	Charles Petzold	Ch. 1, 3–7
CS 61C: Great Ideas in Computer Architecture	UC Berkeley	Ch. 1, 4–6, 10, 12, 15
What every computer scientist should know about floating-point arithmetic	David Goldberg	Ch. 2
What every programmer should know about memory	Ulrich Drepper	Ch. 9–11
Modern microprocessors: a 90-minute guide	Jason Robert Carey Patterson	Ch. 12–15
Software optimization resources	Agner Fog	Ch. 13–15, 20
Crafting Interpreters	Robert Nystrom	Ch. 8, 16
Linkers and Loaders	John R. Levine	Ch. 17
Operating Systems: Three Easy Pieces	Arpaci-Dusseau & Arpaci-Dusseau	Ch. 17–19

Compiled October 2026 as the companion to stage 04, Computer architecture, of The Engineer's Syllabus, alongside *The Engineer's Workbench*, *The Engineer's Compass*, *Exchange*, *Ledger*, *Craft*, *Shipyard*, *Fleet*, *Watch*, *Citadel* and *Foundry*. It follows the order of topics in the stage and draws on the works in the reading list, on Nisan and Schocken's Hack computer, which chapters 3 to 8 simulate, and on the original sources for its named results: Babbage's Analytical Engine (1837); von Neumann's report on the EDVAC (1945); Kogge and Stone (1973) on parallel-prefix adders; the IEEE 754 standard (1985, revised 2019) and Kahan (1965) on compensated summation; Hill (1987) on the three kinds of cache miss; McFarling (1993) on gshare; Kocher et al. (2018) on Spectre; Amdahl (1967) and Gustafson (1988); Little (1961); Aleph One (1996) on stack smashing; Mogul and Ramakrishnan (1996) on receive livelock; and Williams, Waterman and Patterson (2009) on the roofline. Clock rates, cache sizes, latencies and bandwidths are typical of late 2026 and change every year; the ratios between them change far more slowly, and those are what to rely on.

Formulas are written in TeX and converted at build time to MathML, which the browser renders natively, so the page loads no mathematics library. Every example was executed before publication, in pure-standard-library Python with fixed random seeds. Most examples count events, such as gates, cycles, misses and mispredictions, so they print the same numbers on any machine. Two depend on where they run: chapter 9's timings are one recorded run on an Apple M1 Pro, and chapter 8's bytecode listing comes from Python 3.14 and changes between Python versions. Re-run in October 2026 with Python 3.14.7, every example printed exactly the output shown, apart from chapter 9's timings, which differ from run to run and support the same conclusions. Tested here: integer widths, overflow, shifts and byte order; the bits of 0.1, the gaps between doubles, summation and cancellation; every gate from NAND, and a circuit from its truth table; ripple-carry and parallel-prefix adders and the Hack ALU; a latch, a flip-flop, RAM8 and the program counter; RISC-V encoding and decoding; a Hack assembler and CPU; a RISC-V interpreter, CPython bytecode and a stack VM; sequential and random reads from 16 KiB to 256 MiB; a cache simulator that sorts its misses; loop order, blocking, structures of arrays and pointer chasing; a five-stage pipeline with forwarding; 1-bit, 2-bit and gshare predictors; an out-of-order core with register renaming; SIMD widths, masks, Amdahl's law and false sharing; a compiler for a small language; a linker with relocation and randomised loading; a stack overflow and a canary; the CPU cost of polling, interrupts and DMA; and the iron law, a roofline and a profile.