

COMPUTER ARCHITECTURE

A quantitative study of how digital logic, instruction sets, memory hierarchies, pipelines, and parallel hardware execute programs.

OFFICIAL DOCUMENTATION

November 2026

CONTENTS

REPRESENTATION AND DIGITAL STATE

Digital Representation

Logic and Combinational Circuits

Sequential State

INSTRUCTION SETS AND EXECUTION

Instruction-Set Architecture

Assembly and Calling Conventions

Datapath and Control

Pipelining

Hazards and Branches

MEMORY, STORAGE, AND I/O

Cache Hierarchies

Virtual-Memory Hardware

Storage and Interconnects

I/O and Devices

PERFORMANCE AND PARALLEL HARDWARE

Performance Models

Superscalar and Out-of-Order Execution

Multicore and SIMD

Performance Investigation

SYNTHESIS AND CAPSTONE

Binary Design Lab

Architecture Capstone

Architecture Review

COURSE PRACTICE

Computer Architecture Practice Workshop

Representation and digital state

SECTION DETAIL

Digital Representation

Computer architecture begins with a contract about bits. A wire carries one of two reliably distinguishable signal ranges, but the signal itself is not intrinsically an integer, character, address, instruction, or color. Meaning is assigned by an encoding, and every later layer—from an arithmetic unit to a compiler, debugger, file format, and network protocol—must preserve that agreement. A systems architect therefore asks two questions for every bit pattern: “What representation is being assumed?” and “Which operations preserve its intended meaning?”

This chapter develops the representations used by a small processor. It treats unsigned and signed fixed-width integers as algebraic systems, derives two’s-complement operations instead of presenting them as a memorized trick, analyzes overflow and widening, and then turns to IEEE-style floating-point numbers. The final section connects representation choices to instruction behavior, C language semantics, serialization, and testing. The examples are deliberately finite: a 4-bit or 8-bit machine makes the wraparound visible, while the same proofs scale to 32- and 64-bit words.

Bits, words, and interpretation

DEFINITION

A w-bit word $b_{w-1}\dots b_1 b_0 \in \{0,1\}^w$ has no intrinsic type; its numerical meaning is determined by an explicit decoding map such as the unsigned bijection $U_w(b) = \sum_{i=0}^{w-1} b_i 2^i \in [0, 2^w - 1]$ or the two's-complement bijection $T_w(b) = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i \in [-2^{w-1}, 2^{w-1} - 1]$.

A bit is an element of the set $\{0,1\}$. A w-bit word is an element of $\{0,1\}^w$. If the bits are written from most significant to least significant as $b_{w-1}\dots b_1 b_0$, the conventional unsigned interpretation is

$$U_w(b_{w-1}\dots b_0) = \sum_{i=0}^{w-1} b_i 2^i.$$

The function U_w is a bijection from bit strings to the integers in $\{0, 1, \dots, 2^w - 1\}$. Thus an unsigned 4-bit word has sixteen possible values, from `0000` through `1111`, and no 4-bit pattern can represent 16. This is not a weakness of binary notation; it is a consequence of the finite state space. The processor must define what happens when an operation mathematically asks for a value outside the representable set.

The same word can have several meanings. `01000001` is 65 as an unsigned integer, 65 as a two's-complement integer, the ASCII character `A` under one character encoding, or an opcode under a particular ISA. The bits do not carry their own type. A type is a promise made by the surrounding computation, and a bug often consists of applying an operation belonging to one promise to data governed by another.

The term *word* is architectural rather than universal. An ISA may define 32-bit general-purpose registers while an implementation internally uses wider physical values. Memory is usually byte-addressed, so a 32-bit word occupies four consecutive bytes, but the byte order must also be specified. In little-endian order, the least significant byte occupies the lowest address; in big-endian order, the most significant byte does. Endianness does not change the numerical value when the bytes are reassembled correctly, but it changes the byte sequence exposed to a debugger, file format, or network peer.

Worked example: reconstructing a word. Suppose the four bytes at increasing addresses are `78 56 34 12`. Interpreted as little-endian hexadecimal, they form `0x12345678`; interpreted as big-endian, they form `0x78563412`. A memory dump is therefore incomplete without the byte-order convention. The error is not arithmetic—the reader has used the wrong reconstruction function.

BIT PATTERN INTERPRETATION MATRIX

Evaluate each statement regarding how bit patterns, types, and word representations function in computer architecture:

Answer: Physical bits carry no intrinsic type; their semantics are established by an interpretation contract.; Endian conventions define multi-byte storage order in memory without changing mathematical register values.

Hardware stores raw bits without inherent types. The surrounding instruction set, memory conventions, and software contracts determine whether bits represent integers, addresses, or instructions.

Unsigned arithmetic is modular arithmetic

The hardware adder has a fixed number of output wires. If it adds two w-bit operands, the low w output bits are naturally the residue modulo 2^w; a possible carry-out is an additional bit and is not part of the w-bit result unless the ISA explicitly exposes it. Thus fixed-width unsigned addition implements

$$(x + y) \bmod 2^w.$$

For 4-bit words, $1111 + 0001$ produces a low result of `0000` and a carry-out of 1. Numerically, $15 + 1 = 16$, but the 4-bit result represents $16 \bmod 16 = 0$. This is *wraparound*, not a claim that 15 and 1 sum to 0 in the integers. It is the behavior of the projection from an unbounded mathematical result into a finite ring.

Subtraction can use the same adder. In modular arithmetic, the additive inverse of x modulo 2^w is $(-x) \bmod 2^w$. The circuit computes $y - x$ as $y + (-x)$, with the same low-bit projection. For example, on 8 bits, the inverse of 3 is 253 because $3 + 253 = 256 \equiv 0 \pmod{256}$. Subtracting 3 from 5 becomes $5 + 253 = 258 \equiv 2 \pmod{256}$.

Multiplication is also modular when only the low w bits are retained. A full multiplier may produce a $2w$-bit product, and some instructions return both halves; an ordinary fixed-width multiply commonly returns the low half. The distinction matters for overflow detection, cryptographic arithmetic, array-size calculations, and compiler optimizations.

Carry versus overflow. For unsigned addition, the carry-out is the natural overflow indicator: the mathematical sum exceeds $2^w - 1$ exactly when a carry leaves the top bit. For signed two's-complement addition, carry-out alone is not the right test. Signed overflow occurs when two operands with the same sign produce a result with the opposite sign. Equivalently, for addition, overflow is the exclusive-or of the carry into and carry out of the sign bit. The two predicates answer different questions because unsigned and signed interpretations assign different meanings to the same bit pattern.

Worked example: 8-bit addition. Consider `01111111 + 00000001`. As unsigned values, this is $127 + 1 = 128$, which is representable, so there is no unsigned carry-out. As signed two's-complement values, the operands are 127 and 1, but the result pattern `10000000` represents -128. The mathematical signed result 128 is outside the signed range $[-128, 127]$, so signed overflow occurred. A processor may set a signed-overflow flag, an unsigned-carry flag, both, or neither depending on the ISA; software must use the flag whose definition matches the intended type.

THE BITS NEED AN INTERPRETATION

An eight-bit adder produces 10001100 from 01111000 + 00010100. One report calls the result 140; another calls it -116 and flags overflow.

Which statements correctly explain why these two reports differ without changing the output bits?

Reference decision: The bit pattern represents 140 under unsigned arithmetic and -116 under two-complement interpretation.

The same eight bits represent 140 unsigned and -116 in two's complement. The signed addition of 120 and 20 overflows its representable range even though the bit-level sum is well-defined modulo 256.

Signed integers and two's complement

Two's-complement representation defines the signed interpretation

$$T_w(b_{w-1} \dots b_0) = -b_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i.$$

The top bit has negative weight, while all lower bits retain positive powers of two. The representable range is

$$-2^{w-1} \le T_w(x) \le 2^{w-1} - 1.$$

There is one representation of zero, `000...000`. The asymmetry—one more negative value than positive value—is forced by using one of the 2^w patterns for each integer and reserving the sign bit as a negative weight. For 8 bits, the range is $[-128, 127]$; `11111111` is -1, `10000000` is -128, and `01111111` is 127.

To encode a negative integer $-a$ where $0 < a \le 2^{w-1}$, compute $2^w - a$. In bit terms, invert every bit of the positive representation and add one. For $a = 5$ on 8 bits, `00000101` becomes `11111010` after inversion and `11111011` after adding one; 251 interpreted modulo 256 is -5. The reason this procedure works is algebraic: $a + (2^w - a) = 2^w \equiv 0 \pmod{2^w}$, so the bit pattern $2^w - a$ is the additive inverse of a in the fixed-width ring.

The two's-complement sign rule is therefore not an independent arithmetic system bolted onto a binary adder. Signed addition can use the same modular addition as unsigned addition. If the mathematical signed result lies in range, the modular result has the correct signed interpretation. If it lies outside range, the bits still form a residue, but the signed interpretation is not the intended integer result; that is precisely overflow.

Worked example: $-6 + 9$ on 8 bits. Encode -6 as `11111010` and 9 as `00001001`. Add:

```
  11111010
+ 00001001
  --------
  00000011
```

The result is `00000011`, or 3. Both operands have different signs, so signed overflow is impossible: adding a positive and a negative value cannot exceed both ends of the signed interval in one step. This is a useful mental check, but the bit-level overflow predicate remains the authoritative one.

Predict the unsigned and signed interpretations of input 128 at width 8, then change the width to 9. The mathematical input is unchanged: only the projection and interpretation change. Width 8 yields unsigned 128 and signed –128; width 9 yields 128 under both interpretations because its sign bit is now zero.

EXPOSE FIXED-WIDTH OVERFLOW

```
01111111 → unsigned 127 · signed 127
```

Static state: 8-bit 127 is representable under both unsigned and two's-complement interpretations; adding one crosses the signed boundary.

Sign extension, zero extension, and truncation

WARNING

Counterexample: passing a negative signed 32-bit length `len = -1` (`0xFFFFFFFF`) into a bounds check `if ((size_t)len < buf_size)` converts `-1` to `4294967295` via unsigned reinterpretation; if a narrow signed variable is instead zero-extended or truncated before validation, the sign boundary collapses and permits out-of-bounds memory access.

Changing width is not a neutral operation. To preserve an unsigned value when moving from w to $k > w$ bits, prepend zeros; this is *zero extension*. To preserve a two's-complement signed value, replicate the sign bit; this is *sign extension*. For example, the 8-bit signed value `11110110` is -10. Sign-extending to 16 bits gives `11111111 11110110`, which is still -10. Zero-extending it gives `00000000 11110110`, which is 246.

The same bit pattern can therefore change value solely because the type used for extension changed. This is a common source of security bugs when a signed length or index is converted to an unsigned size before bounds checking. In a C-like language, the expression's *usual arithmetic conversions* may perform an implicit conversion that makes a negative value appear very large. A safe design makes the domain and conversion explicit, checks the sign before conversion, and checks multiplication or addition for range overflow before allocating or indexing.

Truncation from k bits to $w < k$ bits discards high bits and preserves the result modulo 2^w. It does not generally preserve a signed value. For example, 16-bit `00000000 10000000` is 128, while truncating to 8 bits gives `10000000`, which is -128 under two's complement. Truncation is safe only when the discarded bits are known to be redundant for the target interpretation: all zero bits for an unsigned value, or all copies of the sign bit for a signed value.

Exercise with solution outline. A 16-bit register contains `0xFF80`. Interpret it as signed, sign-extend it to 32 bits, zero-extend it to 32 bits, and truncate the original bit pattern to 8 bits.

The 16-bit signed value is -128, because the bit pattern is $65536 - 128$. Sign extension produces `0xFFFFFF80`, still -128. Zero extension produces `0x0000FF80`, which is 65408 as an unsigned 32-bit value. Truncation produces `0x80`; that is 128 unsigned or -128 signed. The final answer is intentionally two-valued until the target type is named.

Shifts and bit fields

> **EXAMPLE**
>
> Arithmetic right shift on a two's-complement negative odd integer rounds toward $-\infty$ (floor division), not toward 0: shifting 8-bit `-5` (`11111011`) right arithmetically by 1 bit yields `11111101` (`-3`), whereas C integer division `-5 / 2` truncates toward zero to yield `-2`.

Logical left shift by k positions multiplies an unsigned value by 2^k modulo 2^w when bits shifted out are discarded. Logical right shift inserts zeros and computes floor division by 2^k for an unsigned value. Arithmetic right shift inserts copies of the sign bit and approximates division by a power of two for negative two's-complement values, but the exact rounding rule must be specified by the ISA.

Bit fields use masks and shifts to pack several logical values into one word. To extract a field occupying bits p through $p + r - 1$, compute

$$(x \gg p) \mathbin{\&} (2^r - 1).$$

To insert a new unsigned field value v, first validate $0 \le v < 2^r$, clear the destination bits with the complement of the mask, then OR in $(v \ll p)$. Validation is essential: silently truncating a field can turn a valid identifier into a different identifier.

Worked example: a packed control word. Let bit 0 be `enabled`, bits 1–3 be a mode in the range 0–7, and bits 4–7 be a priority in the range 0–15. For word `10110101`, `enabled=1`; extracting mode gives `(0xB5>>1)&0x7 = 2`; extracting priority gives `(0xB5>>4)&0xF = 11`. To set mode to 6, clear mask `0x0E` and insert `6<<1`, obtaining `0xBD`. The operation preserves all bits outside the mode field by construction.

PACK AND EXTRACT BIT FIELDS WITHOUT CLOBBERING NEIGHBORS

```
uint8_t extract_mode(uint8_t word) {
  return (word >> 1) & 0x7;
}

uint8_t update_mode(uint8_t word, uint8_t new_mode) {
  assert(new_mode < 8);
  return (word & ~(0x7 << 1)) | (new_mode << 1);
}
```

Shifting right by 1 moves bits 1..3 into positions 0..2, and masking with 0×7 (binary 111) isolates the 3-bit unsigned field. To update bits 1..3 in place, clear them with ~(0×7 << 1) and OR in (new_mode << 1).

VERIFY TWO'S-COMPLEMENT SIGN EXTENSION AND OVERFLOW PREDICATES

```
def decode_word(bits: int, width: int):
  mask = (1 << width) - 1
  u = bits & mask
  sign_bit = 1 << (width - 1)
  s = u - (1 << width) if (u & sign_bit) else u
  return u, s

for pattern in [0b01111111, 0b10000000, 0b11110110]:
  u8, s8 = decode_word(pattern, 8)
  sext16 = (pattern | 0xFF00) if (pattern & 0x80) else pattern
  print(f"8-bit 0x{pattern:02X}: unsigned={u8:3d}, signed={s8:4d},
sext16=0x{sext16:04X}")
```

Bit-field layout creates an interface boundary. A producer and consumer must agree on width, signedness, reserved values, byte order, and whether unknown bits must be zero. The safest parser rejects impossible encodings rather than treating reserved patterns as future-compatible by accident.

Fixed-point and floating-point numbers

> **WARNING**
>
> Counterexample to floating-point associativity: in IEEE 754 binary32 ($p = 24$), $(2^{24} + 1.0) - 2^{24}$ evaluates to 0.0 because $2^{24} + 1.0$ lies halfway between 2^{24} and $2^{24} + 2$ and rounds to even (2^{24}), whereas $2^{24} + (-2^{24} + 1.0)$ evaluates to 1.0.

Fixed-point representation allocates a fixed number of fractional bits. With f fractional bits, an integer storage value n represents $n/2^f$. It gives uniform spacing and predictable integer operations, which is valuable in embedded control and financial formats, but its range and precision are fixed globally. Multiplying two values with f fractional bits produces a value scaled by 2^{2f} and requires a rescaling shift, with rounding policy made explicit.

Floating point instead represents values approximately as

$$(-1)^s \times m \times 2^e,$$

where the sign s, significand m, and exponent e are encoded in finite fields. IEEE 754 binary formats use normalized numbers with a biased exponent, subnormal numbers near zero, signed zero, infinities, and NaNs. The details matter because floating-point arithmetic is not ordinary real arithmetic: addition is not associative, overflow can produce infinity, and an invalid operation can produce NaN.

For a normalized binary format with precision p, adjacent values within one exponent bin differ by a fixed *unit in the last place* (ULP). Relative spacing is approximately constant within the normal range, while absolute spacing grows with magnitude. Near zero, subnormals provide gradual underflow with a fixed smaller spacing until the representation reaches zero.

Worked example: why 0.1 is not exact in binary. A finite binary fraction has the form $k/2^n$. If $1/10$ had such a form, then $k/2^n = 1/10$, giving $10k = 2^n$. The left side is divisible by 5 while the right side is not, so no such integers exist. A binary floating-point format must round 0.1 to a nearby representable value. Repeated addition of that approximation can accumulate error, and comparing a computed value directly with a decimal expectation can fail.

Rounding modes define how an exact result between representable numbers is chosen. Round-to-nearest, ties-to-even is common because it avoids a systematic upward bias in repeated operations. Even with this mode, cancellation can destroy significant digits: subtracting two nearly equal large values removes the leading digits that carried reliable information and leaves mostly rounding noise.

Numerical stability is therefore an architectural concern as well as a mathematical one. A program that sums a long sequence should consider pairwise summation, compensated summation, or a wider accumulator when the error budget requires it. A processor's floating-point unit implements specified operations and rounding, but it cannot infer the application's acceptable error or replace an error analysis.

Representation at software boundaries

> **WARNING**
>
> Never compute buffer allocation sizes as `width * height * bytes_per_pixel` in fixed-width 32-bit arithmetic before checking `width <= SIZE_MAX / (height * bytes_per_pixel)`; modular wraparound turns a huge dimension pair into a small allocation followed by a heap buffer overflow.

The representation contract becomes especially fragile at boundaries. A memory-mapped device may define bit 7 as a write-one-to-clear status flag, so a read-modify-write intended to update bit 0 can accidentally clear an event. A file format may store a 32-bit length in network byte order, while the host CPU is little-endian. A compiler may assume signed overflow is impossible in a language mode where the programmer expected modular wraparound. A cryptographic protocol may require canonical serialization so that two byte strings cannot encode the same signed message.

The remedy is not "be careful" but to define boundary functions. A decoder should accept bytes, validate length and reserved fields, and return a typed value or an error. An encoder should reject values outside field ranges. Arithmetic that can cross a representational boundary should use checked, saturating, or explicitly modular operations according to the requirement. Tests should include the minimum, maximum, zero, sign transition, one-past-the-end, endian reversal, NaN, infinity, and malformed encodings where applicable.

Worked boundary example. Suppose an image header contains width and height as unsigned 32-bit values, followed by `width*height*4` bytes of pixels. If the multiplication is performed in a 32-bit type, dimensions such as 70,000 by 70,000 wrap before allocation. A later write based on the mathematical product can overflow the allocated buffer. A safe decoder first checks each dimension against a policy limit, promotes to a wider type, checks `width <= SIZE_MAX/(height*4)`, and only then allocates. The representation lesson is that the bit width of an intermediate expression is part of the security proof.

Summary of invariants

REMARK

These six representation invariants underpin every subsequent hardware layer: ALU flags in ISA semantics, immediate sign-extension in datapaths, address-offset extraction in caches, and descriptor bounds checking in I/O.

Core representation invariants and failure modes:

Invariant	Mathematical condition	Hardware mechanism	Concrete failure mode when violated
1. Unsigned residue	$R = (A + B) \pmod{2^w}$	Discard carry-out bit w	Image buffer size calculation wraps to small value before allocation
2. Two's-complement	$[-2^{w-1}, 2^{w-1} - 1]$	MSB carries negative weight -2^{w-1}	Negating the minimum value -2^{w-1} produces -2^{w-1}
3. Extension safety	$\text{sext}(x) = x, \text{zext}(x) = x$	Replicate MSB for signed; zero-pad unsigned	Zero-extending negative signed index creates giant unsigned offset
4. Overflow vs carry	$V = c_w \oplus c_{w-1}$	Distinct CPU flags (`CF` vs `OF`)	Using unsigned carry (`CF`) to check signed overflow (`OF`)
5. Floating point	Finite binary approximations	IEEE 754 round-to-even and exception flags	Associativity failure in summation; non-exact representation of 0.1
6. Boundary contracts	Canonical serialization	Strict validation before field packing	Truncation silently aliases reserved bitfields to valid identifiers

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.1.1: Two's-Complement Arithmetic and Overflow Flags

Verification application

A 5-bit two's-complement machine adds `10110` and `01011`. Decode both operands, compute the bit result, decide whether signed overflow occurred, and decide whether unsigned carry occurred.

Recall that signed overflow detects whether the mathematical sum falls outside the 5-bit signed range [-16, 15], while unsigned carry checks whether the sum exceeds 31.

`10110` is $22 - 32 = -10$ and `01011` is 11. The mathematical sum is 1, so the result is `00001` with no signed overflow. As unsigned values, the operands are 22 and 11, whose sum 33 is outside the 5-bit unsigned range 0–31; the low result is 1 and carry-out is 1. This contrast is the point: one bit computation supports two different overflow predicates.

Exercise 1.1.2: Bitfield Masks and Extraction

Construction application

A 12-bit word stores a 3-bit tag at bits 9–11, a 5-bit length at bits 4–8, and a 4-bit flags field at bits 0–3. Give masks and extraction expressions for each field. What must be checked before inserting a length of 31?

Construct each mask with 1s at designated positions in hex. Extract with bitwise AND followed by right-shift by the field's starting index. Compare 31 against the 5-bit maximum capacity.

The tag mask is `0xE00`, the length mask is `0x1F0`, and the flags mask is `0x00F`; extraction shifts the masked value right by 9, 4, or 0 respectively. Length 31 fits the five-bit field exactly; length 32 does not and must be rejected rather than truncated to zero.

Exercise 1.1.3: Sign Extension Safety at Boundaries

Description transfer

Explain why zero-extending a negative 8-bit array index before checking it against an unsigned array length can be dangerous. Give a safe sequence of checks.

Trace what value -1 becomes when zero-extended versus sign-extended into a 32-bit or 64-bit unsigned type.

A negative signed index such as -1 becomes a very large unsigned value after conversion, potentially passing a poorly ordered comparison or changing which comparison is performed. First verify that the signed index is nonnegative, then convert to an unsigned size type, then compare against the array length, and only then calculate the address.

Exercise 1.1.4: Floating-Point Precision and Relative Spacing

Generalisation transfer

Derive the maximum relative spacing in a normalized binary floating-point format with precision p under round-to-nearest, ignoring subnormals. State the assumption about the exact value's location between adjacent representable numbers.

In a binade [2^e, 2^(e+1)), adjacent normalized values are spaced by 2^(e - p + 1). Relate the half-ULP maximum rounding error to the binade's lower endpoint.

In a binade, adjacent normalized values are separated by $2^{e-(p-1)}$. The relative gap to the lower value is therefore $2^{-(p-1)}$. Round-to-nearest makes the maximum rounding error half an ULP, at most 2^{-p} relative to a value near the lower endpoint; the exact bound varies slightly across the binade. The assumption is that the exact value is rounded to the nearest representable neighbor, not truncated.

Exercise 1.1.5: Canonical Serialization Design

Construction synthesis

Design a byte-level serialization for a signed 24-bit integer. Specify byte order, encoding range, and the decoder's response to out-of-range or noncanonical encodings.

Specify an exact 3-byte layout (e.g. little-endian two's complement), write down the explicit range [-2^23, 2^23 - 1], and state parser rejection rules for malformed lengths.

A 24-bit signed two's-complement value ranges from -2^{23} through $2^{23} - 1$. Encode its modular 24-bit pattern, choose big- or little-endian order, and require the decoder to sign-extend bit 23. A canonical decoder rejects a byte sequence of the wrong length and any alternate encoding not allowed by the format rather than silently accepting ambiguous representations.

References & Further Reading

- MIT 6.004 Computation Structures: lecture notes and examinations
- MIT 6.004 learning objectives
- IEEE 754-2019, Standard for Floating-Point Arithmetic
- CERT C: INT rules

SECTION DETAIL

Logic and Combinational Circuits

Digital design is the refinement of a mathematical function into a physical network whose outputs become valid after a bounded propagation delay. A *combinational circuit* has no intentional storage: once its inputs have remained stable long enough, its outputs depend only on those inputs. This property distinguishes it from sequential logic, where the circuit's state records history. The distinction is architectural, not merely terminological: an ALU, decoder, address calculator, and cache tag comparator are combinational components that must produce values within the timing budget of a clocked system.

Boolean functions as specifications

DEFINITION

A combinational logic block refines a pure mathematical function $f : \{0,1\}^n \to \{0,1\}^m$: for any input vector $x \in \{0,1\}^n$ held stable for at least the propagation delay t_{pd}, the physical output settles to $f(x)$ independently of prior input history.

The simplest specification is a Boolean function $f : \{0,1\}^n \to \{0,1\}^m$. A truth table lists all 2^n input combinations and the required output. A circuit is correct when, for every input row, its stable output equals the specified output. This exhaustive definition is practical for small circuits and becomes the semantic foundation for symbolic simplification, formal equivalence checking, and test generation for larger ones.

The primitive operations are NOT ($\neg x$), AND ($x \wedge y$), and OR ($x \vee y$). NAND and NOR are *functionally complete*: every Boolean function can be built from either one alone. For example, $\neg x = x \,\mathrm{NAND}\, x$, $x \wedge y = \neg(x \,\mathrm{NAND}\, y)$, and $x \vee y = (\neg x) \,\mathrm{NAND} (\neg y)$. Functional completeness is an expressiveness result; it does not imply that a NAND-only implementation is fastest, smallest, or easiest to verify.

Boolean algebra supports local transformations that preserve a function. Important identities include commutativity, associativity, distributivity, identity ($x \wedge 1 = x, x \vee 0 = x$), domination ($x \wedge 0 = 0, x \vee 1 = 1$), idempotence ($x \wedge x = x$), complement ($x \wedge \neg x = 0$), absorption ($x \vee (x \wedge y) = x$), and

De Morgan's laws:

$$\neg(x \wedge y) = \neg x \vee \neg y, \qquad \neg(x \vee y) = \neg x \wedge \neg y.$$

These are equations over truth values, so each transformation can be justified for all inputs. A designer should prefer a proof-preserving transformation to an intuition such as "this gate seems redundant."

Worked derivation. Simplify $F = xy + x\bar{y} + \bar{x}y$, where juxtaposition denotes AND and `+` denotes OR. First factor the first two terms: $xy + x\bar{y} = x(y + \bar{y}) = x$. Therefore $F = x + \bar{x}y$. Apply absorption in its dual form, $x + \bar{x}y = x + y$, which can be verified by distributivity: $(x + \bar{x})(x + y) = 1 \cdot (x + y) = x + y$. The original three-product circuit can therefore be implemented by one OR gate after the appropriate inputs are available. The simplification is a theorem about the function, not an approximation.

FROM TRUTH TABLE TO CIRCUIT

What property defines a combinational circuit?

Answer: Its stable output depends only on the current input values

A combinational circuit implements a function of current inputs. Storage and clocked state are introduced by sequential elements.

Canonical forms and minimization

Every Boolean function can be written as a sum of products (SOP). For each truth-table row where $F = 1$, form a *minterm* that is 1 only on that row, then OR the minterms. Likewise, a product of sums (POS) uses maxterms for rows where $F = 0$. Canonical forms are systematic and easy to derive, but they may contain many literals and gates.

Consider a three-input function that is 1 for inputs `001`, `010`, and `111`. With variables a, b, c in that order, its SOP is $\bar{a}\bar{b}c + \bar{a}b\bar{c} + abc$. If the function has don't-care input combinations—states that never occur or whose output does not matter—those rows may be assigned whichever value produces a cheaper implementation. A don't-care is not permission to ignore a reachable safety-critical state; it is a specification statement that must be defended by the surrounding design.

Karnaugh maps expose adjacent minterms that differ in one variable. Grouping powers of two allows the differing variable to disappear. For larger functions, Quine–McCluskey or binary decision diagrams provide systematic alternatives, while synthesis tools use cost models and technology libraries. “Minimal number of terms” and “minimal delay” are different objectives: factoring may reduce area but add a logic level; duplicating a term may reduce fanout and improve timing.

EQUIVALENT BOOLEAN FUNCTIONS, DIFFERENT HAZARDS

Two circuits have the same steady-state truth table. During a one-bit input change, one briefly pulses its output because paths have unequal delay.

Which evidence distinguishes the implementations?

Reference decision: A timing trace through reconvergent paths, including propagation-delay bounds

Boolean equivalence describes stable values, not transient behavior. Reconvergent paths with unequal delays can create hazards that only a timed path analysis or waveform reveals.

Multiplexers and decoders

DEFINITION

A 2^k-to-1 multiplexer implements indexed selection $y = d_{\mathrm{unsigned}(s_{k-1}\ldots s_0)}$ via a tree of $2^k - 1$ two-input mux stages of depth k, while a k-to-2^k decoder asserts the one-hot minterm $y_i = 1 \iff \mathrm{unsigned}(s_{k-1}\ldots s_0) = i$.

A 2-to-1 multiplexer is the conditional operator of hardware:

$$M(s, a, b) = \bar{s}a \vee sb.$$

When $s = 0$, the first product survives and $M = a$; when $s = 1$, the second survives and $M = b$. A 2^k-to-1 multiplexer can select one of 2^k inputs using k select bits. Multiplexers are powerful because a data path can be described as "choose this value when the control predicate holds" rather than drawn as a separate circuit for every case.

A decoder performs the inverse-shaped task: k input bits activate one of 2^k outputs. A 2-to-4 decoder with inputs a, b generates minterms $\bar{a}\bar{b}$, $\bar{a}b$, $a\bar{b}$, and ab. Decoders construct register-file enables, memory chip selects, instruction subfields, and one-hot state encodings. A one-hot representation uses one state bit per state, increasing storage but often simplifying next-state logic and making illegal-state detection explicit.

Worked construction. Implement a 4-to-1 multiplexer from three 2-to-1 multiplexers. First select between $d0$ and $d1$ using $s0$, and between $d2$ and $d3$ using $s0$. Select those two intermediate results using $s1$. The first layer has two parallel gates and the second layer one gate, so the data path has two multiplexer delays. A balanced tree is preferable to chaining three selections because its depth is logarithmic in the number of inputs.

Adders and arithmetic structure

The half-adder adds two bits. Its sum is $s = a \oplus b$ and carry is $c = ab$. A full adder adds a, b, and carry-in c_{in}:

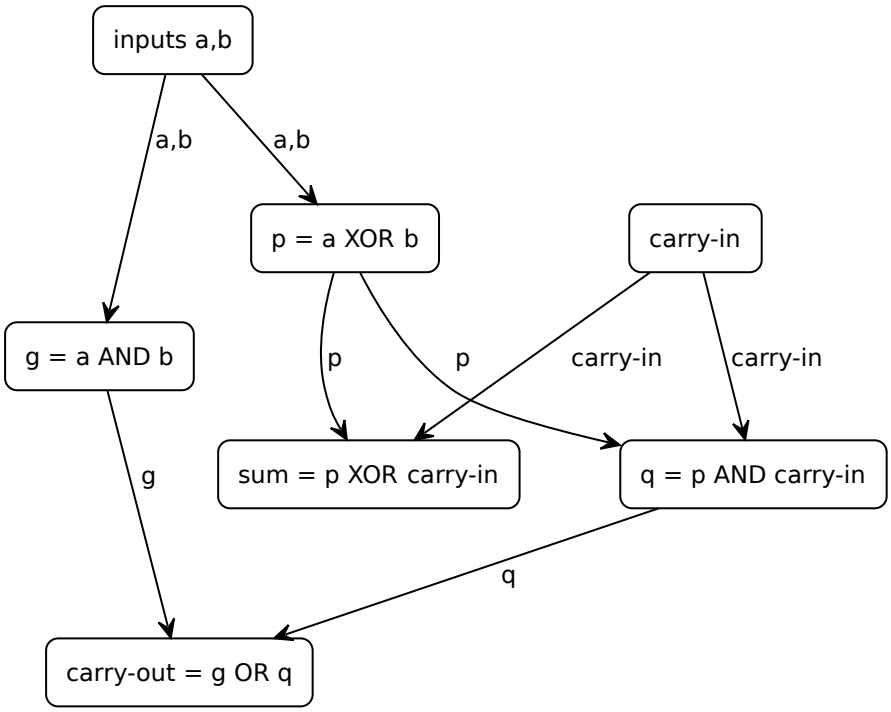
$$s = a \oplus b \oplus c_{in}, \qquad c_{out} = ab + ac_{in} + bc_{in}.$$

The carry expression says at least two of the three inputs are 1. It can also be written $ab + c_{in}(a \oplus b)$, separating the generate case from the propagate case. Chaining w full adders creates a ripple-carry adder. Its area is linear, but the carry may traverse every bit, so worst-case delay is $O(w)$ gate stages.

The carry-propagate structure motivates a carry-lookahead adder. Define per bit $g_i = a_i b_i$ and $p_i = a_i \oplus b_i$. Then $c_{i+1} = g_i + p_i c_i$. Expanding gives $c2 = g1 + p1\,g0 + p1\,p0\,c0$, and so on. Parallel prefix structures compute these equations with lower depth at greater wiring and area cost. This is a recurring architecture trade-off: asymptotic latency improves by investing physical resources and design complexity.

Subtraction uses the identity $a - b = a + \neg b + 1$ in two's complement. A single adder can therefore add either b or $\neg b$ selected by a control signal, with the same signal supplied as the initial carry-in. The ALU must define flag semantics: zero, negative, carry/borrow, and signed overflow are not interchangeable. A branch instruction that tests unsigned less-than uses a different interpretation from one that tests signed less-than even when both consume the same subtractor outputs.

The full-adder graph implements the derived generate/propagate equations. Each arrow carries a single Boolean value except the explicitly paired input `a,b`; no unspecified selector or stored state is needed.



Transfer check. For $(a, b, c_{in}) = (1, 0, 1)$, predict every internal wire and both outputs. Then change only b to 1: which carry term now establishes the output?

For `(1,0,1)`, $p = 1, g = 0, q = 1$, sum is 0, and carry-out is 1: the input carry propagates. For `(1,1,1)`, $p = 0, g = 1, q = 0$, sum is 1, and carry-out is 1: the operands generate the carry. Both satisfy $a + b + c_{in} = s + 2c_{out}$.

Comparators and predicates

LEMMA

In w-bit two's-complement arithmetic, $A <_{\text{signed}} B$ holds if and only if $(S_{\text{b}}^{\text{su}} \oplus V_{\text{b}}^{\text{su}}) = 1$, where S_{b}^{su} is the sign bit of $(A - B) \bmod 2^w$ and $V_{\text{b}}^{\text{su}} = c_{w-1} \oplus c_w$ is the signed subtraction overflow flag; for unsigned values, $A <_{\text{unsigned}} B$ holds if and only if the adder carry-out of $A + \neg B + 1$ is 0.

Equality of two w-bit words is the AND of bitwise equivalences:

$$eq(a, b) = \bigwedge_{i=0}^{w-1} \neg(a_i \oplus b_i).$$

An equality tree reduces delay compared with a serial chain. Unsigned less-than can be determined by the most significant bit at which operands differ. A lexicographic comparator therefore scans from high bits conceptually, although hardware uses a tree or prefix formulation. Signed comparison treats the sign bit specially: a negative operand is less than a nonnegative operand; if signs match, the remaining bits can be compared as an unsigned magnitude under the two's-complement ordering.

Predicates are often more valuable than numeric results. The ALU's output may be discarded while its zero or sign condition controls a branch, a memory-protection check, or an instruction exception. The control contract must say whether a comparison is signed, unsigned, exact, masked, or tolerant of invalid encodings.

Timing, hazards, and physical reality

WARNING

Two algebraically identical Boolean expressions can have different transient behavior: in $F = ab + \bar{a}c$ with $b = c = 1$, toggling a from $1 \to 0$ can briefly drop F to 0 (a static-1 hazard) if the ab gate turns off before the inverter on $\bar{a}$ turns on, unless the redundant consensus term bc is included or the output is sampled only after t_{pd}.

Boolean equivalence describes the stable function, but a physical circuit also has propagation delay. If an output feeds another gate before it has settled, the receiving logic can observe a transient value. The longest input-to-output path is the critical path and bounds the maximum clock frequency when the circuit lies between registers. Gate count is only a rough area proxy; fanout, wire length, loading, and technology-specific delays matter.

Unequal path delays can produce a *glitch*: an output briefly changes even though the logically correct value before and after the transition is the same. For example, in $F = ab + \bar{a}c$, changing a while $b = c = 1$ should leave $F = 1$, but the direct and inverted paths may not switch simultaneously, producing a temporary zero. Adding the consensus term bc yields a hazard-resistant two-level form for this transition, at the cost of an extra product. Whether a glitch matters depends on the receiver: a level-sensitive combinational consumer may ignore it after settling, while an asynchronous control input or clock edge can capture it.

Static timing analysis uses arrival and required times to verify that paths meet setup constraints. The logic designer cannot safely assume zero delay merely because a truth table is correct. Conversely, adding arbitrary "delay gates" is not a robust fix; synthesis may remove them, process variation may change their delay, and the real solution may require registering the signal or changing the protocol.

CMOS and implementation cost

> REMARK
>
> Because static CMOS inverting gates (NAND, NOR, NOT) restore full voltage rails in a single stage whereas non-inverting AND/OR gates require an inverting stage followed by an inverter, De Morgan transformations naturally map two-level Sum-of-Products expressions (`AND-OR`) into lower-delay `NAND-NAND` networks.

Transistor counts and stage delays in static CMOS:

Gate type	Complementary pull-up / pull-down topology	Transistor count	Stage delay and characteristics
`INVERTER`	1 pMOS pull-up, 1 nMOS pull-down	2	Single inverting stage; symmetric rise/fall with sized pMOS
`NAND2`	2 parallel pMOS, 2 series nMOS	4	Natural inverting gate; higher electron mobility makes series nMOS fast
`NOR2`	2 series pMOS, 2 parallel nMOS	4	Slower than NAND2 because lower hole mobility requires wider series pMOS
`AND2`	`NAND2` stage driving an `INVERTER`	6	Two stages; higher latency and area than native `NAND2`
`OR2`	`NOR2` stage driving an `INVERTER`	6	Two stages; higher latency and area than native `NOR2`
`XOR2`	Complex CMOS gate or transmission gates	8–12	High logic depth; forms the delay bottleneck in ripple carry adders

At the transistor level, complementary CMOS uses networks of p-type and n-type devices to implement logic with low static power in steady states. A NAND gate is naturally efficient: the pull-down network places nMOS devices in series for the AND condition, while the pull-up network is the dual parallel structure. A NOR gate has the opposite topology and often different delay characteristics because series and parallel transistor arrangements load differently.

The same Boolean expression can map to different physical costs. Factoring reduces repeated logic but may increase fanout; a NAND-NAND implementation can exploit the technology's native gate; a balanced tree reduces depth but increases wiring. Modern synthesis tools optimize against timing, area, power, and testability objectives. Architectural reasoning remains necessary because the tool only optimizes within the constraints and library the designer supplies.

Verification strategy

REMARK

Combinational verification combines structural equivalence checking against a reference specification with boundary-value vectors (`0x00000000` , `0xFFFFFFFF` , `0x7FFFFFFF` , `0x80000000`) that exercise full carry-chain propagation and sign-bit transitions.

For a small combinational block, exhaustive simulation covers all inputs. For a 32–bit ALU, exhaustive testing is impossible, so verification combines equivalence checking, constrained random tests, directed corner cases, and compositional proofs. Directed cases should include all-zero and all-one operands, one-hot patterns, sign transitions, carry chains, alternating bits, and illegal control encodings.

An assertion expresses a property such as “if operation is ADD, then result plus a widened carry equals the mathematical sum” or “if select is 0, output equals input A regardless of input B.” Assertions turn the specification into an executable proof obligation. A test that observes only the common path can miss a carry that propagates through every bit or a mode field that selects the wrong input.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.2.1: Full Adder Composition from Half Adders

Verification application

Derive a full-adder circuit from two half-adders. Give the equations for the intermediate carry and prove that the final carry equals $ab + ac_{in} + bc_{in}$.

Express sum and carry-out of the full adder using the intermediate sum and carry signals of two cascaded half adders.

The first half-adder computes $x = a \oplus b$ and $g = ab$; the second computes $s = x \oplus c_{in}$ and $h = xc_{in}$. The final carry is $g + h = ab + c_{in}(a \oplus b) = ab + ac_{in} + bc_{in}$ after expanding XOR and absorbing the duplicate term.

Exercise 1.2.2: Karnaugh Map Minimization and Symmetry

Construction application

Minimize $F(a, b, c, d) = \Sigma m(0, 2, 5, 7, 8, 10, 13, 15)$ and identify any symmetry. Verify the expression on at least one minterm and one zero row.

Group adjacent minterms on a 4-variable K-map and look for XOR/XNOR checkerboard patterns involving variables b and d.

The minterms pair by the high-bit and low-bit symmetries. Grouping the four cells where $b \oplus d = 0$ gives $\bar{b}\bar{d} + bd$; equivalently $F = b \oplus d$. A correct solution may use XOR/XNOR as a primitive or expand it into SOP. Check `0000` (1), `0001` (0), and a paired edge such as `1010` (1).

Exercise 1.2.3: Hierarchical Multiplexer Tree Synthesis

Construction application

Build an 8-to-1 multiplexer from 2-to-1 multiplexers. Give the number of gates and logic depth for a balanced tree and a serial chain.

Determine how many 2-to-1 multiplexers are needed to select 1 of 8 inputs, and compare a $\log_2(8)$-depth balanced tree against a serial linear chain.

A balanced tree uses seven 2-to-1 multiplexers and depth three. A serial chain also uses seven multiplexers but has depth seven and would require a different select-routing scheme; the balanced tree meets a shorter critical path.

Exercise 1.2.4: Data-Dependent Critical Path in Ripple-Carry Adders

Description transfer

Explain why a ripple-carry adder has a data-dependent delay even though its topology is fixed. Give an input pattern that propagates carry through every bit.

Find an operand combination where a carry generated at the least significant bit must propagate sequentially through all N bit positions.

Each full adder waits for the incoming carry. An operand pair such as `01111111` plus `00000001` generates a carry at the low bit and propagates it through all higher positions, exposing the linear chain delay.

Exercise 1.2.5: Signed Two's-Complement Comparator Correctness

Proof transfer

A comparator receives signed two’s-complement operands. Prove that comparing sign bits first and magnitudes second yields the correct order, including the minimum negative value.

Consider how sign bits order opposite-sign operands and how overflow flag V modifies the sign flag N during subtraction.

If signs differ, every negative two's-complement value is less than every nonnegative value, so the sign bit decides. If signs match, subtracting the common sign weight leaves an ordering by the lower bits; for two negative operands, the lower-bit ordering is reversed relative to unsigned magnitude, which the signed comparator's subtract/overflow logic accounts for. The minimum value is not negatable in range, so a proof must not assume symmetric magnitude bounds.

Exercise 1.2.6: Static-1 Hazard Detection and Consensus Term Elimination

Verification transfer

Construct a transition that can glitch in $ab + \bar{a}c$. Why does adding bc cover the static-1 hazard? What timing assumption makes the term useful?

Examine what happens when $b = c = 1$ and variable a changes state if an inverter propagation delay causes both product terms to momentarily drop to 0.

Set $b = c = 1$ and change a from 0 to 1. The function should stay 1, but ab may fall before $\bar{a}c$ rises. The consensus term bc remains 1 throughout, covering the static-1 gap. It helps only when the input transition is single-variable or the timing environment makes the consensus reasoning applicable.

Exercise 1.2.7: Formal Combinational ALU Invariant Assertions

Formalisation synthesis

Propose an assertion set for a 4-bit ALU with ADD, SUB, AND, OR, and XOR. Include one property about unused control codes.

Write algebraic invariants matching outputs to mathematical operations across all opcode modes, and specify defensive defaults for undefined control codes.

For each opcode, assert the output equation. For ADD and SUB, use widened arithmetic to relate result, carry, and operands; for bitwise operations assert bitwise equality. For unused codes, specify a safe result and side-effect behavior—such as no write enable—rather than leaving the output unconstrained.

References & Further Reading

- MIT 6.004 Computation Structures: lecture notes and examinations
- MIT 6.004 learning objectives
- Cornell CS 3410: Computer System Organization and Programming

SECTION DETAIL

Sequential State

Combinational logic answers a question about the present inputs. A processor, however, must remember a program counter, register contents, outstanding requests, protocol phase, and architectural mode. That memory is *state*. Sequential logic is the disciplined combination of storage elements and combinational next-state logic. Its correctness depends on both a functional relation—what state should follow what inputs—and a temporal relation—when a new state is allowed to become visible.

This chapter develops the synchronous design model used by most digital systems. It distinguishes latches from edge-triggered registers, derives setup and hold constraints, models finite-state machines formally, and treats asynchronous inputs and reset as sources of metastability and illegal behavior. The purpose is not to draw a traffic light but to make timing and state invariants explicit enough that a pipeline controller, bus arbiter, or cache controller can be reasoned about and tested.

State as a mathematical transition system

A synchronous machine can be modeled as a tuple $(S, I, O, \delta, \lambda)$: a set of states S, input alphabet I, output alphabet O, transition function $\delta : S \times I \to S$, and output function λ. At discrete clock boundaries, if the current state is s_t and the sampled input is i_t, then

$$s_{t+1} = \delta(s_t, i_t), \qquad o_t = \lambda(s_t, i_t) \text{ or } o_t = \lambda(s_t),$$

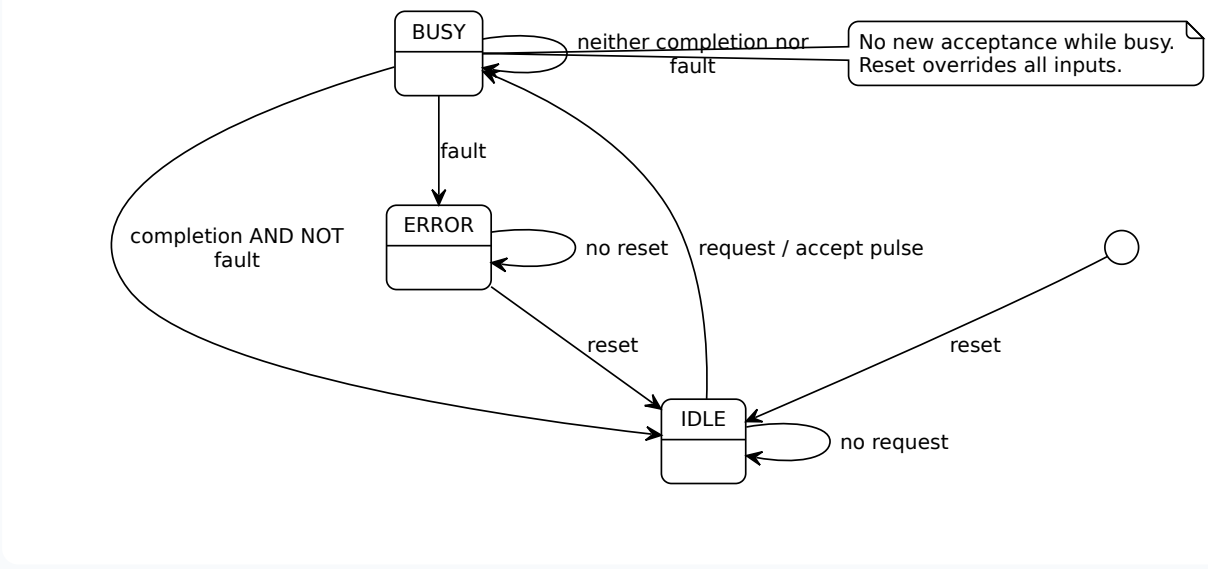
depending on whether the machine is Mealy or Moore. A Mealy output may respond combinationally to an input during a state; a Moore output depends only on registered state. Moore outputs often simplify timing and avoid input glitches at an interface, while Mealy machines may use fewer states and respond within a cycle. Neither is universally superior; the interface contract decides whether an output may change between clock edges.

The state set is an abstraction. A two-bit register physically has four encodings, but a protocol may use only `00`, `01`, and `10`; `11` is illegal. The transition function must define what happens from an illegal encoding, usually by moving to a safe recovery state. Leaving an encoding unconstrained is not the same as proving it unreachable. Faults, reset races, soft errors, X-propagation in simulation, and design bugs can all make an "impossible" state observable.

Worked example: a request controller. Let states be `IDLE`, `BUSY`, and `ERROR`. In `IDLE`, a request starts work and transitions to `BUSY`; no request keeps the machine in `IDLE`. In `BUSY`, completion transitions to `IDLE`, while a detected fault transitions to `ERROR`. In `ERROR`, reset returns

to `IDLE` and all normal inputs are ignored. The safety invariant is that a request is accepted only in `IDLE`; the liveness assumption is that a `BUSY` operation eventually either completes or reports a fault. The hardware design is incomplete until the priority between simultaneous completion and fault is specified.

The diagram completes that contract: reset has highest priority in every state, fault wins over completion in `BUSY`, and a request during `BUSY` is not queued. All ordinary transitions sample inputs on the active edge; the global reset rule is omitted from the arrows for legibility.



Trace challenge. Starting in `IDLE`, sample `(request, completion, fault)` as `(1,0,0)`, `(1,1,1)`, then `(1,1,0)`, with reset absent. Predict the post-edge states and count accepted requests.

The states are `BUSY`, `ERROR`, `ERROR`. Exactly one request is accepted, at the first edge. Fault outranks completion at the second edge; the third request cannot escape `ERROR` without reset. Accepting another request while busy would violate the specified no-queue contract.

STATE THE TRANSITION

What is required to specify a synchronous state machine?

Answer: A state space, next-state rule, output rule, and clock/reset assumptions

A transition system needs state and rules for how inputs change it. Clock and reset assumptions determine when the mathematical transition is applied.

Latches, flip-flops, and registers

A level-sensitive latch is transparent during an enable phase and holds its previous value otherwise. If enable is high, Q follows D after propagation delay; if enable is low, Q is stable. A pair of latches driven by complementary phases can form an edge-triggered element. A flip-flop is commonly abstracted as sampling D at a clock edge and retaining that sample until the next active edge:

$$Q(t^+) = D(t_{sample}),$$

where t^+ denotes the state just after the edge. A register is a bank of such storage elements, often with a shared clock-enable and reset. The abstract edge-triggered model prevents a combinational path from "seeing through" a storage element during the active edge, which makes cycle-by-cycle reasoning compositional.

The abstraction has limits. A physical flip-flop needs its input stable for a setup interval before the edge and a hold interval after it. If the input changes too close to the edge, the internal feedback may take an unusually long time to resolve to 0 or 1: metastability. The output is not a third logical value in the final system, but it can violate downstream timing while resolving. No finite synchronizer can make metastability impossible; it reduces the probability that unresolved behavior reaches logic by giving the first stage time to settle.

Two flip-flops in series are a common clock-domain crossing synchronizer for a single slowly changing control bit. The first may become metastable; the second samples it one cycle later. This is not safe for a multi-bit bus whose bits can settle on different cycles, nor does it preserve every pulse if the source toggles faster than the destination can observe. Multi-bit data requires a protocol such as a handshake, asynchronous FIFO, gray-coded pointer scheme, or a formally justified sampling method.

A CORRECT TRUTH TABLE CAN STILL MISS TIMING

A register's input changes 50 ps before the clock edge, but its setup requirement is 90 ps. Simulation using only Boolean values reports the expected next state.

Which conclusion is justified?

Reference decision: Functional logic is insufficient; the setup violation can make capture unreliable

Sequential correctness includes timing constraints. Violating setup time can produce an incorrect or metastable capture even when the Boolean next-state function is correct.

Timing constraints

DEFINITION

Synchronous edge-triggered timing holds if and only if every register-to-register path satisfies both the **setup constraint** $t_{\text{cq,max}} + t_{\text{comb,max}} + t_{\text{setup}} \leq T + t_{\text{skew}}$ and the **hold constraint** $t_{\text{cq,min}} + t_{\text{comb,min}} \geq t_{\text{hold}} + t_{\text{skew,hold}}$.

In a single-clock path, a launching register changes after the clock-to-Q delay t_{cq}, combinational logic requires t_{comb}, and the receiving register requires data to arrive at least t_{setup} before the next edge. With clock period T and skew t_{skew}, a simplified setup constraint is

$$t_{cq,max} + t_{comb,max} + t_{setup} \leq T + t_{skew}.$$

The corresponding hold constraint is a lower-bound condition: the launching register's earliest change plus the shortest combinational delay must not reach the receiving register too soon after the same edge. A simplified form is

$$t_{cq,min} + t_{comb,min} \geq t_{hold} + t_{skew,hold}.$$

Increasing the clock period can repair setup failure but cannot generally repair hold failure. This distinction matters in pipeline design: a path may be slow enough to miss the next edge, or so short that new data corrupts the old sample immediately after the edge. Adding a register changes latency and may repair setup while creating a new architectural timing contract.

Worked timing calculation. Suppose $t_{cq,max} = 80$ ps, the worst combinational path is 620 ps, setup time is 90 ps, and the worst skew provides 20 ps of relief. Then $T \geq 80 + 620 + 90 - 20 = 770$ ps, so the frequency bound from this path is approximately 1.30 GHz. If the shortest path has $t_{cq,min} = 40$ ps and $t_{comb,min} = 15$ ps, with hold requirement 45 ps and no helpful hold skew, the hold inequality is satisfied at 55 ps. Notice that the setup and hold calculations use different extrema; averaging delays would be unsound.

Clock gating and enables add another layer of correctness. A register with enable e has next-state equation $D = e \cdot f(s, i) + \bar{e} \cdot s$, meaning it retains state when disabled. A gated clock physically suppresses clock transitions and can save power, but the gate itself must be designed so enable changes cannot create a spurious pulse. A synchronous enable is usually easier to verify than ad hoc combinational clock gating.

Finite-state-machine design flow

> **EXAMPLE**
>
> In a ready/valid handshake controller, a transfer occurs on a rising clock edge if and only if $\mathrm{valid} \wedge \mathrm{ready} = 1$; once the sender asserts $\mathrm{valid} = 1$, it is forbidden to deassert valid or mutate the payload until the clock edge where $\mathrm{ready} = 1$ completes the transfer.

A robust design flow begins with an interface contract rather than an encoding. First define events, inputs, outputs, safety properties, liveness assumptions, reset behavior, and simultaneous-event priority. Next identify the smallest state that distinguishes futures: two histories may share a state only if every possible future input produces equivalent outputs and transitions. Then write a state-transition table or graph, choose an encoding, derive next-state logic, and verify the implementation against the transition relation.

State minimization is a behavioral question. If two states produce different outputs for the same input, they cannot be merged. If their outputs match, they can be merged only when their successor states are themselves equivalent under every input. This is the same idea as distinguishing prefixes in automata theory, now applied to a clocked controller. One-hot encoding uses a separate bit for each

state, binary encoding uses roughly $\lceil \log_2 |S| \rceil$ bits, and gray-like encodings can reduce simultaneous bit changes. Encoding affects physical timing and fault detection but not the abstract machine.

Worked example: a ready/valid interface. Suppose a producer presents `valid` and data, and a consumer presents `ready`. A transfer occurs on a clock edge exactly when `valid && ready`. If the producer is allowed to keep data stable while waiting, the controller needs `EMPTY` and `FULL` states: `EMPTY` may accept a new item, `FULL` holds an item until transfer. If simultaneous dequeue and enqueue are permitted, `FULL` with both signals high may remain `FULL` while replacing the data; if not, the enqueue must be rejected. The choice changes throughput and must be stated. A common bug is to clear `valid` as soon as it is asserted rather than when the handshake condition is true, losing data when the consumer is not ready.

Counters, timers, and derived state

> **WARNING**
>
> Off-by-one timeout bugs arise when a countdown register is loaded with N at acceptance edge t_0 without specifying whether expiration is tested on the pre-decrement state ($q_t = 0$, expiring after $N + 1$ edges) or the post-decrement state ($q_{t+1} = 0$, expiring after N edges).

A counter is a register plus an increment, decrement, hold, or load function. For a w-bit modulo counter, $q_{t+1} = (q_t + 1) \bmod 2^w$. A timer that must represent a duration of N cycles needs a clear convention: does zero mean "expired now," does it count the current edge, and can reload occur on the expiration edge? Off-by-one errors are semantic errors in δ, not merely coding mistakes.

Worked example: a three-cycle timeout. If a request is accepted at edge t_0 and the specification says failure should be raised if completion has not arrived by the third subsequent edge, then the relevant checks occur at t_1, t_2, t_3. Load 3 at t_0 and, at each later edge without completion, decrement then test the new value: it becomes 2, 1, 0 and expires at t_3. An equivalent convention loads 2 and tests the old value for zero before decrementing: old values at t_1, t_2, t_3 are 2, 1, 0. Loading 2 and testing after decrement instead expires at t_2, one edge too early. State whether completion wins over timeout at t_3, and test durations 0, 1, 2, 3, and 4 explicitly; duration zero needs its own acceptance-edge rule.

Derived state should be minimized when it can be recomputed without ambiguity. Storing both a count and an `empty` flag creates a consistency invariant; if one updates and the other does not, the controller can accept or discard data incorrectly. Redundant state is justified when it improves timing

or power, but then the equality and update invariants must be asserted.

Reset and initialization

> **WARNING**
>
> Releasing an asynchronous reset without synchronizing its deassertion edge to the local clock domain can violate reset-recovery timing on a subset of state flip-flops, throwing a multi-bit FSM or one-hot controller into an unreachable illegal state on the very first active clock edge.

Reset establishes an initial state but does not automatically initialize every physical bit in every technology. A synchronous reset is sampled by the clock; an asynchronous reset can affect state between edges but introduces deassertion timing concerns. A common safe pattern asserts reset asynchronously for rapid containment and deasserts it synchronously per clock domain.

Reset strategies and timing boundaries:

Reset architecture	Assertion timing	Deassertion timing	Hardware failure mode / race condition
Pure synchronous	On active clock edge	On active clock edge	Cannot reset if clock is stopped or gated; requires larger gates in logic path
Raw asynchronous	Immediate (level-triggered)	Asynchronous to clock	Deassertion near clock edge violates recovery/removal timing, causing metastability
Synchronized deassertion	Immediate (asynchronous)	Synchronized via 2-FF chain	Clean deassertion aligned to clock; eliminates cross-register reset skew

```
// Reset synchronizer: asynchronous assertion, synchronous deassertion
always @(posedge clk or posedge async_rst) begin
    if (async_rst) begin
        rst_meta <= 1'b1;
        rst_sync <= 1'b1;
    end else begin
        rst_meta <= 1'b0;
        rst_sync <= rst_meta;
    end
end
```

Reset values should be chosen for safety, not convenience. A bus arbiter should not grant multiple owners after reset; a valid/data queue should begin empty; a privilege controller should begin in the least-authority state. If the state encoding has illegal patterns, reset should enter a legal recovery state and the transition logic should route illegal encodings there as well.

Reset is also a protocol event. Downstream logic must know when outputs are meaningful, and cross-domain consumers must not observe one domain leaving reset while another remains held. A design that "works after reset" in simulation may fail in hardware if deassertion violates recovery/removal timing or if memories and registers initialize differently.

Verification and invariants

> **REMARK**
>
> Inductive invariant verification proves $I(s_0)$ at reset and $\forall s, i.\ I(s) \Rightarrow I(\delta(s, i))$ across all $2^{|I|}$ input combinations—including simultaneous events such as `enqueue && dequeue` or `completion && timeout`.

The strongest FSM tests are derived from properties. Safety properties say something bad never occurs: never grant two mutually exclusive resources, never dequeue an empty queue, never assert `done` without a prior accepted request. Liveness properties say something good eventually occurs under assumptions: every accepted request eventually completes or faults, a waiting arbiter eventually grants a continuously requesting client, and a nonempty queue eventually dequeues when the consumer remains ready.

An invariant proof has the form: establish the property in every initial state, assume it before an edge, and show every allowed transition preserves it. For a one-entry queue, the invariant is `valid=0` iff no item is stored, and `valid=1` implies stored data is stable until a handshake. The proof must consider no operation, enqueue only, dequeue only, and simultaneous enqueue/dequeue. Omitting the simultaneous case is a classic completeness failure.

Simulation explores traces; formal verification can quantify over all input combinations within a bounded or inductive model. Neither replaces a correct environment assumption. An assertion that assumes `ready` is always high proves little if the actual integration allows backpressure.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.3.1: Finite State Machine Controller Design

Construction application

Design a four-state elevator controller for floors 0 and 1. Specify state, request input, motor output, door output, reset behavior, illegal-state recovery, and simultaneous arrival/request priority.

Construct state transition diagrams and truth tables with explicit current and next state assignments.

A correct solution has explicit states such as `IDLE_0`, `MOVING_TO_1`, `IDLE_1`, and `MOVING_TO_0`, with outputs derived from state. A request received while moving must be queued, ignored, or merged by specification; silently choosing one is not a design. Illegal state encodings should disable the motor and return to a known idle state after reset or fault recovery.

Exercise 1.3.2: Setup and Hold Timing Constraint Equations

Verification application

A path has $t_{cq,max} = 120$ ps, $t_{comb,max} = 1.4$ ns, setup 80 ps, and clock skew of -30 ps. Compute the minimum clock period and explain the sign of skew in your equation.

Calculate maximum clock frequency from flip-flop clock-to-Q delay, combinational logic, and setup margins.

The bound is $T \geq 120 + 1400 + 80 - (-30) = 1630$ ps, because negative skew makes the receiving edge effectively earlier relative to the launching edge. The maximum frequency is about 613.5 MHz. The exact sign convention can vary, but the physical question is whether skew gives or removes available setup time.

Exercise 1.3.3: Metastability and Multi-Clock Synchronizers

Description transfer

Explain why synchronizing each bit of a binary counter independently can produce a value that never existed in the source domain. Give a safer transfer protocol.

Explain the mean time between failures (MTBF) when crossing asynchronous clock domains with dual flip-flops.

Bits do not cross at the same instant. A source changing from `0111` to `1000` can be sampled as `1111`, a mixed value never held by the source. Use a gray-coded counter for pointer transfer when its one-bit transition property is proven, or transfer data with a request/acknowledge handshake or asynchronous FIFO.

Exercise 1.3.4: Ready/Valid Handshake Protocol State Table

Construction transfer

Write a complete transition table for a one-entry ready/valid buffer, including simultaneous enqueue and dequeue. State whether it supports fall-through behavior.

Specify valid and ready assertion rules to prevent data loss or duplicate reads across boundaries.

The buffer is empty when `valid=0`. Enqueue without dequeue stores data and sets valid; dequeue without enqueue clears valid; neither preserves state; both can replace data and preserve valid if the protocol permits throughput one per cycle. Fall-through requires an additional combinational path from input to output when empty and a different timing contract; a registered buffer does not fall through.

Exercise 1.3.5: Queue Storage Invariant Verification

Proof transfer

Prove the queue-valid invariant for the four cases of input handshake and output handshake. Identify the case that changes the stored data without changing occupancy.

Prove that enqueue and dequeue operations maintain FIFO order and correct full/empty status flags.

In an empty state, only an accepted enqueue makes valid true. In a full state, only an accepted dequeue without enqueue makes it false. Neither handshake preserves it. Simultaneous handshake replaces the stored item and preserves occupancy. The proof must include the fact that the producer holds input data stable while `valid && !ready`.

Exercise 1.3.6: One-Hot vs Binary State Encoding Trade-offs

Description synthesis

Compare one-hot and binary state encoding for 12 states. Discuss flip-flop count, decode complexity, illegal encodings, and timing rather than declaring one universally better.

Compare flip-flop count, combinational decode logic speed, and power dissipation between encoding strategies.

One-hot uses 12 state bits and often simple local decode. Of its $2^{12} = 4096$ bit patterns, exactly 12 have one asserted bit, leaving 4084 illegal encodings, including all-zero and multi-hot values. Binary uses four bits: 12 encodings represent states and four of the 16 encodings remain unused. Binary decode may create deeper logic. Physical fanout, power, synthesis technology, and fault model determine the choice; both schemes need defined illegal-state behavior.

Exercise 1.3.7: Asynchronous Reset Assertion with Synchronous Release

Construction synthesis

Give a reset design for a multi-clock system with a producer, consumer, and asynchronous external interrupt. Identify which signals require synchronization and which require a handshake.

Explain how reset synchronizers prevent metastable transitions when releasing circuit resets.

Reset release should be synchronized independently in each clock domain. A single-bit interrupt request needs synchronization or an asynchronous capture circuit; a multi-bit event needs a handshake or FIFO. The producer and consumer need a defined reset interaction so neither treats stale valid/data as a new transfer.

References & Further Reading

- MIT 6.004 Computation Structures: lecture notes and examinations
- MIT 6.004 learning objectives
- Cornell CS 3410: Computer System Organization and Programming

Instruction sets and execution

SECTION DETAIL

Instruction-Set Architecture

An instruction-set architecture (ISA) is the stable contract presented by a processor to software. It specifies the programmer-visible state, instruction semantics, encodings, addressing rules, memory-access behavior, privilege boundaries, and exceptional events.

A microarchitecture may execute instructions in one cycle, in a deep pipeline, out of order, or through translation into internal operations; as long as it preserves the ISA's specified observations, it remains a valid implementation. This abstraction is what lets one binary run across generations of hardware and lets compilers target a machine without knowing its transistor-level organization.

> **REMARK** **Scope boundary and microarchitectural refinement**
>
> This lesson defines the functional ISA contract ($\sigma \rightarrow \sigma'$) that compilers target and hardware must refine; cycle-by-cycle datapath multiplexing, pipeline hazard forwarding, and out-of-order register renaming are microarchitectural realizations developed in subsequent datapath and pipeline lessons.

The contract is narrower than "what the hardware happens to do." Timing, cache size, branch-predictor state, and speculative execution are usually not architectural state, although they affect performance and can create side channels.

Conversely, the exact behavior of an unaligned access, an overflow flag, a memory ordering operation, or an illegal instruction is architectural if software can observe it. A rigorous ISA specification therefore separates *functional semantics* from *implementation accidents* and states what is guaranteed, permitted, or undefined.

Programmer-visible state

DEFINITION

An ISA state tuple $\sigma = (\mathrm{PC}, R, M, \mathrm{CSR}, \mathrm{priv})$ comprises the program counter PC, general-purpose and vector registers R, addressable memory M, architectural control/status registers CSR, and current privilege level priv. Each instruction word $w = M[\mathrm{PC}]$ defines a deterministic transition $\sigma' = [\![\mathrm{decode}(w)]\!](\sigma)$ that either commits updated architectural state or traps to an exception vector with architectural cause metadata.

The architectural state commonly includes a program counter (PC), a register file, memory, status/control registers, and privileged state. A register file has a finite set of named registers with a defined width. Some ISAs reserve a register whose reads always produce zero; others expose a stack pointer, link register, condition-code register, or vector register bank. The PC identifies the next instruction according to the ISA's fetch convention. It may advance by a fixed instruction width, by an instruction-length decoder, or by a branch/jump target.

An instruction is a state transition. If architectural state before execution is σ, instruction bytes decode to operation op, and the memory system supplies the required values, then execution defines $\sigma' = [\![op]\!](\sigma)$. The notation hides important partiality: an access may be misaligned, a privilege check may fail, a page may be absent, or a divide may have an invalid operand. In those cases, the ISA specifies an exception transition rather than an ordinary result.

Worked example: load/store semantics. A RISC-style load word `lw r1, 12(r2)` computes an effective address $EA = R[r2] + 12$, checks alignment and access permission, reads four bytes according to the memory endianness, sign- or zero-extends the word as specified, writes it to `r1`, and advances the PC. A store computes the same kind of address but writes register data to memory and does not update a general-purpose destination. The assembler syntax is short; the architectural transition includes address arithmetic, width, alignment, permission, and exception behavior.

LOCATE THE ABSTRACTION

Evaluate which elements belong strictly to the Instruction-Set Architecture (ISA) interface contract rather than a specific microarchitectural implementation:

Answer: Programmer-visible register files, instruction encodings, memory semantics, and trap exception behaviors.; Formal addressing mode specifications, instruction byte widths, and architectural memory alignment contracts.

An ISA specifies the functional contract between software and hardware. Microarchitectures implement this specification using diverse pipeline depths, execution units, and cache designs.

RISC and CISC as design families

A load/store ISA restricts memory access to explicit load and store operations; arithmetic instructions operate on registers. This regularity simplifies decode, register allocation, pipeline staging, and formal reasoning. It may increase instruction count for operations that could be expressed as memory-to-memory work. A more instruction-rich ISA can encode complex addressing modes or multi-step operations, reducing code size at the cost of decoding and implementation complexity.

The labels RISC and CISC describe tendencies, not a complete classification. Modern "complex" ISAs may translate instructions into simple internal operations, while a RISC ISA can contain vector, atomic, compressed, or privileged instructions with substantial semantics. The useful question is what software-visible contract is exposed and what implementation freedom it preserves.

Instruction length presents a trade-off. Fixed-width encodings simplify fetch alignment and parallel decode but may waste bits for small operations. Variable-length encodings improve code density but make instruction boundaries and branch targets harder to locate. Compressed encodings add a second width while preserving a common register or operation subset. Code density affects instruction-cache pressure, so encoding is a performance design choice as well as a compiler-interface choice.

ARCHITECTURAL STATE DEFINES THE INSTRUCTION

Two implementations execute an ADD instruction differently internally. Both read the same operands, update the destination register, advance the PC, and raise identical exceptions.

Evaluate what establishes these two different processor designs as valid implementations of the same ISA operation:

Reference decision: Their externally observable architectural state transitions and precise exception side effects are mathematically identical.

An ISA specifies programmer-visible state and functional behavior. Microarchitectures may differ radically in internal pipelines and resources while refining the identical architectural contract.

Encoding and decoding

An instruction encoding partitions bits into fields: opcode, register identifiers, immediate values, function modifiers, and sometimes mode or rounding controls. Field widths constrain the number of registers and immediate range. If a register file has 2^r entries, a register identifier needs r bits. An immediate may be sign-extended from a compact field; its range and alignment may be more important than its raw width.

Worked example: PC-relative branch. Suppose a branch encodes a signed k-bit displacement in instruction units of four bytes. The signed displacement range is $[-2^{k-1}, 2^{k-1} - 1]$ units, and the byte target is $PC + 4d$ if the architecture defines the PC as the current instruction address. A compiler cannot assume that every label is reachable; the assembler or linker must synthesize a longer branch sequence when the displacement does not fit. Misunderstanding whether the base is current PC or next PC creates an off-by-one-instruction error that passes short tests and fails at range boundaries.

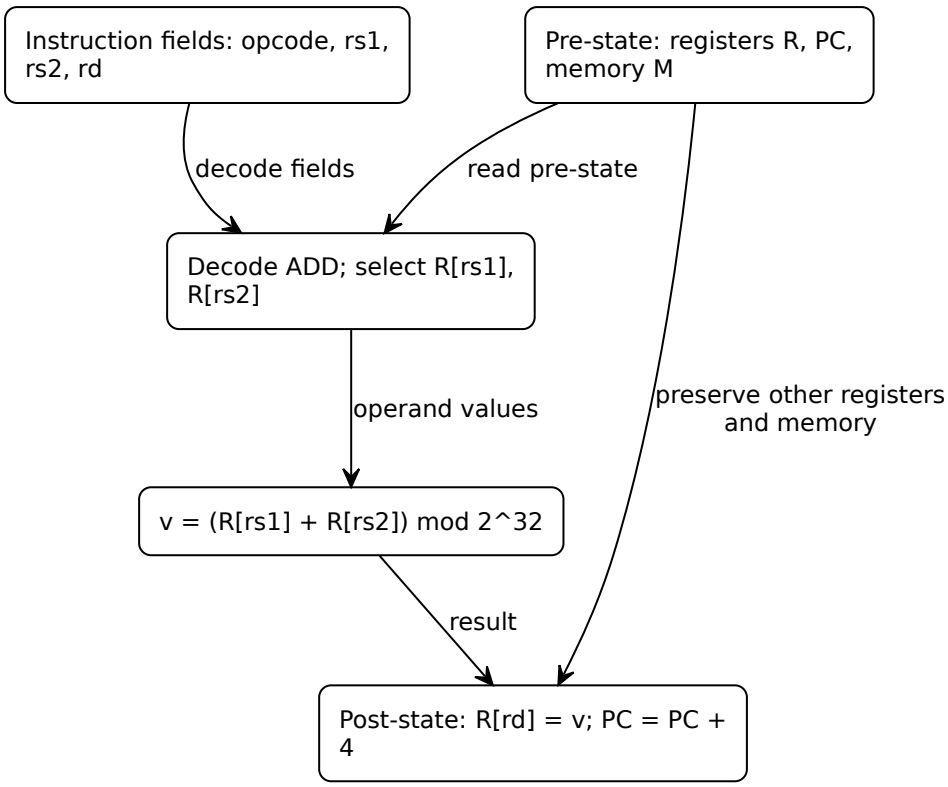
Immediate encoding also interacts with sign extension. An instruction that adds a signed 12-bit immediate must interpret `0xFFF` as -1, not 4095. A separate unsigned-immediate instruction or zero-extension rule is needed when the latter is intended. The opcode's semantic definition, not the bit pattern alone, determines the result.

> ***Pause & Predict: The 12-Bit Immediate Sign-Extension Trap.*** *Suppose you execute* `addi a0, zero, 0x800` *on a 32-bit RISC-V processor where* `addi` *accepts a 12-bit immediate field* `imm[11:0]`*. What 32-bit hexadecimal value is stored in* `a0`*? Is it* `0x00000800` $(+2048)$ *or* `0xFFFFF800` (-2048)*? Commit your reasoning before opening the analysis below.*

Check bit 11 of `0x800` (`0b1000_0000_0000`). If the most significant bit of the 12-bit immediate is 1, what does arithmetic sign extension do to bits 31..12?

Analysis: The value stored is `0xFFFFF800` (-2048)! In standard RISC-V I-type instructions, the 12-bit immediate is strictly **sign-extended** to 32 bits. The hex value `0x800` has bit 11 equal to `1` (`0b1000_0000_0000`). The sign bit `imm[11]` is copied into all upper 20 bits: `imm[31:12] = 0b1111_1111_1111_1111_1111` (`0xFFFFF`). Thus, the machine sign-extends `0x800` to `0xFFFFF800` . To load an unsigned $+2048$ or construct larger 32-bit values, one must use `lui` (Load Upper Immediate) combined with `addi` , carefully adjusting for the negative sign bit.

For a hypothetical 32-bit, fixed-width register-add ISA with modular arithmetic and no flags, the decoded fields select a precise state transition. This is a semantic map, not a claim about execution stages; the source operands are read from the pre-state even when a destination aliases a source.



Transfer check. Set `rs1=5`, `rs2=6`, `rd=5`, with initial `R[5]=0xffffffff` and `R[6]=2`. Determine the post-state and explain why destination/source aliasing does not change the operands.

> **REMARK** **Simultaneous state update and operand isolation**
>
> The mathematical sum is $2^{32} + 1$, so `R[5]` becomes 1. `R[6]` and memory remain unchanged; PC advances by four bytes. Both operands came from the pre-state. Writing the result early and then rereading `R[5]` as an input would implement a different transition. No carry flag is updated because this hypothetical ISA has no flags.

Addressing modes and memory consistency

An addressing mode defines how an instruction derives an effective address: base plus offset, base plus index times scale, PC-relative, absolute, or indirect through a register. Addressing modes are conveniences expressed in the ISA; a microarchitecture may compute them in one adder or several

internal steps. The mode must specify overflow behavior, alignment, privilege checks, and whether an address is computed before or after an update.

Worked example: effective address and instruction count. A RISC-V load `lw x5, 12(x6)` computes $EA = R[x6] + 12$. If `x6 = 0x1000`, then `EA = 0x100c`; alignment and permission are checked on that address before the four bytes are read. An indexed access to element `i` of an 8-byte array is not a single RISC-V addressing mode: with the index in `a1` and the base in `a0`, a compiler emits `slli t0, a1, 3`, `add t0, a0, t0`, then `ld x5, 0(t0)`. An x86-64 `mov rax, [rbx + rcx*8]` encodes the same base-plus-index arithmetic in one instruction. The trade is visible: the complex mode reduces instruction count but adds address-generation work and encoding constraints; the load/store ISA keeps the arithmetic explicit.

Memory is not an array of independent locations on a modern multiprocessor. The ISA specifies an ordering model for loads and stores. A sequentially consistent model places operations from all cores into one order consistent with each core's program order. Weaker models permit certain reorderings and require explicit fences or atomic operations when synchronization depends on visibility order. A compiler and processor must jointly preserve the language-level memory model; a lock implemented with ordinary unordered loads and stores is not automatically a lock.

Falsifiable trace: store buffering. Two cores execute the following program, with `x` and `y` initially zero:

```
Core 0:                 Core 1:
store 1 -> x            store 1 -> y
r0 <- load y            r1 <- load x
```

Under sequential consistency, at least one load must observe 1: the alternative requires a cycle in the single total order, because each core's store precedes its own load. Under x86-TSO and RISC-V's weak ordering, the outcome `r0 = 0` and `r1 = 0` is permitted: each store may remain buffered while its core loads the other location. Observing `(0,0)` therefore falsifies a claim that the implementation is sequentially consistent, but it does not falsify x86-TSO or RVWMO. Adding `mfence` between the store and load on x86, `fence rw, rw` on RISC-V, or sequentially consistent atomics forbids the outcome. Release/acquire ordering forbids the message-passing variant—store data, release-store a flag, acquire-load the flag, load data—but it does not order a store followed by a load on the same core, so it does not by itself forbid this store-buffering outcome. A lock built from plain loads and stores without these constraints has no such guarantee.

An atomic read-modify-write instruction can guarantee that no competing core observes an intermediate state. Compare-and-swap typically reads a location, compares it with an expected value, and writes a replacement only if equal, returning success or failure. Its semantics must specify

memory order, spurious failure if permitted, alignment, and exception behavior. Atomicity is about indivisibility at the specified granularity; it does not by itself prevent starvation, ABA problems, or incorrect higher-level invariants.

Procedure calls and ABI boundaries

The ISA supplies call, jump, and return primitives, but a usable software ecosystem also needs an application binary interface (ABI). The ABI assigns arguments and return values to registers or stack locations, classifies caller- and callee-saved registers, defines stack alignment, lays out aggregate values, and specifies symbol naming and object formats. A compiler-generated function and hand-written assembly interoperate only if both obey these conventions.

A call often saves a return address in a link register or stack, transfers control to a target, establishes a frame if needed, and returns through the saved address. Leaf functions may avoid a frame; recursive or reentrant functions cannot assume a single global copy of locals. The ABI's preservation rule is an invariant across the call boundary: after the callee returns, every callee-saved register has the value the caller expects, while caller-saved registers may have changed.

Worked example: summing an array. Suppose arguments are a pointer `p` and length `n`, and the result is returned in `r0`. A correct loop needs a pointer, remaining count, and accumulator. Each iteration loads a word, adds it to the accumulator, advances the pointer by the element size, decrements the count, and branches while nonzero. The proof invariant is: after k iterations, the accumulator is the sum of the first k elements and `p` points to element k. The machine sequence is correct only if the load width, pointer increment, signedness of the loop comparison, and return convention all match the source-level contract.

Exceptions, traps, and privilege

An exception transfers control from ordinary instruction execution to a handler with architectural state saved according to the ISA. Synchronous exceptions arise from the current instruction—illegal opcode, page fault, arithmetic fault, or system call. Interrupts are asynchronous events such as a timer or device completion, though the architectural boundary may use a common entry mechanism.

Precise exceptions mean the machine state appears as if all older instructions completed and no younger instruction took effect. In an in-order implementation this is natural; an out-of-order implementation needs a reorder or retirement mechanism to preserve the illusion. The exception record must identify a restart PC or faulting instruction, cause code, privilege transition, and saved status. A handler that cannot determine whether to retry or skip an instruction cannot safely recover.

Worked trace: a load page fault and restart. User code executes `lw a0, 0(s0)` with `PC = 0x4000_1000` and `s0 = 0x0000_0000_dead_b000`. The page-table walk finds no valid mapping. The hart records a synchronous exception with `scause = 13` (load page fault), `stval = 0xdead_b000`, and `sepc = 0x4000_1000`; it copies the previous privilege (U) into `sstatus.SPP`, saves the interrupt-enable state in `sstatus.SPIE`, disables supervisor interrupts, and enters the handler at `stvec`. The destination `a0` is not written: a precise exception leaves the register file as if the faulting instruction had not executed.

The handler allocates a page, installs a valid PTE, executes `sfence.vma` to discard stale translations on the handling hart, and executes `sret`. `sret` restores privilege from `SPP`, restores `SIE` from `SPIE`, and sets `PC = sepc`, so the load re-executes at `0x4000_1000` and now succeeds. Other harts that may have cached the old translation require a TLB shootdown. If the handler instead advanced `sepc` by four bytes, it would skip the load and silently leave `a0` undefined; if the hardware partially wrote `a0` before the fault was resolved, it would violate precise-exception semantics.

Privilege modes constrain access to memory, control registers, and devices. A system call deliberately enters a more privileged mode through a defined instruction; a page-protection violation enters it because an access violated a rule. The boundary is a security contract: user code should not be able to forge the saved privilege state, redirect the handler to an untrusted address, or retain access to privileged mappings after return.

Instruction semantics and compiler reasoning

Compilers transform programs by proving that the transformed program has equivalent permitted observations. If an ISA defines signed overflow as trapping, the compiler cannot replace a checked addition with modular addition. If overflow is undefined at the language level, the compiler may use that assumption to eliminate impossible branches—but only within the source language's rules. If an ISA operation wraps but the language operation is checked, the compiler must insert a range check or use a widening instruction.

Worked example: reassociation is not a semantic identity. IEEE 754 addition is not associative, so these two source expressions have different values:

```
double left  = (0.1 + 0.2) + 0.3;   /* 0.6000000000000001 */
double right = 0.1 + (0.2 + 0.3);   /* 0.6 */
```

A compiler may reassociate only when the language or an explicit fast-math mode permits it; otherwise the transformation changes the permitted observations. The same distinction separates the language from the ISA. C signed overflow is undefined behavior, so a compiler may fold `x + 1 >`

`x` to true; with `-fwrapv` it must preserve wraparound. The ISA's `add` instruction may wrap modulo the register width, but the compiler's obligation comes from the source language. If the language operation is checked, the compiler must insert a range check or use a wider instruction rather than relying on the hardware's wraparound.

Alias and alignment assumptions also affect legal transformations. A load can be hoisted above another operation only if the ISA and memory model permit the reordering and no aliasing or exception behavior changes. A vectorized load may require alignment or use a defined unaligned instruction. "The hardware usually handles it" is not an architectural proof.

Verification of an ISA implementation

An ISA simulator is an executable reference model, but a simulator bug can repeat the same misunderstanding as the hardware. Differential testing compares an implementation against a trusted model across random and directed programs. Directed tests target sign extension, branch boundaries, register-zero behavior, misalignment, privilege transitions, page faults, atomic races, and precise exception restart.

Worked example: directed tests and a random property. Differential testing is only as strong as the observations it compares. On RV64, the reference transition for `addi x5, x0, imm` is $R[x5] = \operatorname{sext}_{64}(imm_{12})$, with `x0` hardwired to zero and `PC` advancing by four. A test set that omits the sign bit passes a zero-extending bug.

Stimulus	Reference observation	Bug it exposes
`addi x5, x0, 0x000`	`x5 = 0`	wrong constant path
`addi x5, x0, 0x7ff`	`x5 = 2047`	truncation or off-by-one
`addi x5, x0, -2048` (immediate bits `0x800`)	`x5 = 0xffff_ffff_ffff_f800`	zero-extension instead of sign-extension
`addi x5, x0, -1` (immediate bits `0xfff`)	`x5 = -1`	failure to sign-extend
`addi x0, x0, 1`	`x0` remains 0	register-zero not hardwired
`addi x5, x0, imm` for random 12-bit `imm`	`x5 = sext(imm)`	any mismatch in the decode path

Random testing can exercise many additional encodings but does not guarantee all 4,096 cases are covered. Exhaustive enumeration of the 12-bit immediate field does; the directed cases explain why a mismatch matters. A simulator that agrees with the hardware on the same wrong interpretation is not a

reference: differential testing needs an independently written model, or formal refinement against the ISA's transition relation, before agreement is evidence.

Formal instruction proofs typically establish a refinement relation between microarchitectural state and architectural state. Internal pipeline registers, speculative values, and cache metadata are allowed to differ; at retirement, the visible state must match the reference transition. This relation explains why an implementation can execute instructions out of order while still satisfying a sequential ISA.

Counterexample & Failure Mode: The Unaligned Split-Access Trap & Atomicity Breakdown

Does an ISA's memory access contract guarantee atomicity and fault isolation for arbitrary pointers? Consider executing a 32-bit load `lw x5, 0(x1)` where `x1 = 0x0FFF_FFFE`.

1. **Boundary Breakdown**:
    - The 4-byte word occupies addresses `[0x0FFF_FFFE, 0x0FFF_FFFF, 0x1000_0000, 0x1000_0001]`.
    - These bytes straddle two distinct 4 KB virtual memory pages: Page 0 (`0x0FFF_F000`) and Page 1 (`0x1000_0000`).
2. **Split Fault Failure Mode**: Suppose Page 0 is resident in RAM with read permissions, but Page 1 is unmapped or paged out to disk.
    - A naive microarchitecture that reads bytes 0 and 1 before translating Page 1 will encounter a **Page Fault** on the second half of the instruction!
    - If the core partially wrote the destination register `x5` with the first two bytes, the architectural state is corrupted prior to exception entry, violating **precise exception semantics**.
3. **Multicore Atomicity Collapse**: Even if both pages are resident, an unaligned access straddling a 64-byte cache line boundary cannot be fulfilled by a single L1 cache tag match. It requires two distinct bus/interconnect transactions.
    - If another core executes a concurrent store to either cache line, it can interleave between the two halves.
    - The load reads bytes 0–1 from before the store, and bytes 2–3 from after the store, observing a "torn read" word that never existed in memory.
    - **ISA Contract Guarantee**: This is why strict RISC ISAs specify that misaligned accesses either raise an Address-Misaligned Trap or are explicitly non-atomic across cache boundaries.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.1.1: Signed Load Architectural State Transition Specification

Construction application

Specify the complete formal architectural state transition for a signed load instruction (e.g. `lw rd, offset(rs1)`), covering effective address arithmetic, privilege and alignment validation, byte endianness, sign extension into destination register `rd`, and precise exception handling.

Trace effective address generation, alignment verification, memory read, and sign extension into destination.

Compute effective address $EA = R[rs1] + \text{sign_extend}(offset)$. Verify user privilege level and that $EA \pmod 4 = 0$ (for 32-bit aligned accesses). If misaligned or access-violating, suppress destination update and trigger an Address-Misaligned or Load-Access-Fault trap saving faulting PC in exception state. If valid, read 4 bytes starting at EA according to machine endianness, sign-extend the 32-bit word to register width, write to $R[rd]$ (if $rd \neq x0$), and advance $PC \leftarrow PC + 4$.

Exercise 2.1.2: Fixed-Width 32-Bit Instruction Encoding Layout

Construction application

Design a 32-bit fixed-width encoding scheme supporting: (1) 16 three-register R-type operations; (2) 8 two-register I-type operations with a 12-bit signed immediate; and (3) 8 two-register B-type word-aligned conditional branches with a 12-bit signed displacement. Allocate explicit bitfields for opcode, registers, and immediates.

Allocate explicit bitfields for opcode, destination register, source registers, and immediate constants.

Use a 7-bit primary opcode field `inst[6:0]` to distinguish instruction classes. For R-type: `rd[11:7]` (5 bits), `funct3[14:12]` (3 bits), `rs1[19:15]` (5 bits), `rs2[24:20]` (5 bits), `funct7[31:25]` (7 bits). For I-type: `rd[11:7]`, `funct3[14:12]`, `rs1[19:15]`, `imm[31:20]` (12 bits signed $[-2048, 2047]$). For B-type: `funct3[14:12]`, `rs1[19:15]`, `rs2[24:20]`, with displacement bits split across `inst[31]`, `inst[7]`, `inst[30:25]`, and `inst[11:8]` to encode a ± 4 KB word-aligned branch offset.

Exercise 2.1.3: Assembly Loop Invariant Inductive Proof

Proof synthesis

Prove the partial correctness of an assembly array-sum loop using mathematical induction over loop iterations k, demonstrating that the accumulator satisfies $\mathrm{acc} = \sum_{i=0}^{k-1} A[i]$ at the loop header for arrays up to length N.

Induct on array index k, showing that the accumulator equals $\sum_{i=0}^{k-1} A[i]$ at the loop header, and discuss modular integer overflow.

Base case ($k = 0$): Before entering the loop, the accumulator is cleared to 0 and the base pointer references $A[0]$; the empty sum is 0, satisfying the invariant. Inductive step: Assume $\text{acc}_k = \sum_{i=0}^{k-1} A[i]$ at the start of iteration k. The loop body executes `lw t0, 0(ptr)`, `add acc, acc, t0`, and `addi ptr, ptr, 4`. At the end of the iteration, $\text{acc}_{k+1} = \text{acc}_k + A[k] = \sum_{i=0}^{k} A[i]$. When the branch condition $k = N$ terminates the loop, the accumulator contains $\sum_{i=0}^{N-1} A[i]$ under two's-complement modular arithmetic $(\text{mod } 2^{32})$.

Exercise 2.1.4: Memory Consistency and Litmus Test Synchronization

Verification application

In a multiprocessor system, thread 1 executes `data = 42; flag = 1;` while thread 2 executes `while (flag == 0); r1 = data;`. Verify why a weak memory model allows thread 2 to observe $r1 = 0$, and insert the necessary ISA memory barrier instructions to guarantee sequentially consistent message passing.

Write a classic 2-thread message passing pattern (data write, flag set vs flag check, data read) and insert memory fences or acquire/release operations.

Under weak memory architectures (e.g. RISC-V or ARM), the hardware memory controller or out-of-order execution engine may reorder independent stores ($W(data)$ and $W(flag)$) or independent loads ($R(flag)$ and $R(data)$). If $W(flag)$ reaches cache before $W(data)$, thread 2 sees $flag = 1$ but reads stale $data = 0$. Inserting a release store (`s-fence` or `fence w,w`) on thread 1 and an acquire load (`l-fence` or `fence r,r`) on thread 2 forces visibility order.

Exercise 2.1.5: Precise Exception Constraints on Speculative Store Commits

Description recall

Explain why an out-of-order processor cannot write speculative store results directly to the L1 data cache before an older floating-point divide instruction has completed without faulting.

Explain how committing a younger store to cache or memory before an older instruction's fault status is determined violates precise state.

Precise architectural exceptions require that when an exception occurs at instruction I_k, all instructions prior to I_k have completely retired their state, and no instruction at or subsequent to I_k has modified architectural registers or memory. If a younger store modifies cache before an older divide raises a divide-by-zero trap, persistent memory reflects side effects of instructions that should have been cancelled, destroying restartability and violating the ISA contract.

Exercise 2.1.6: Procedure Calling Convention and ABI Stack Frame Design

Construction application

Specify an ABI calling convention for an 8–register processor architecture that must support recursive function calls and 16–byte stack alignment. Partition registers and draw the resulting stack frame layout.

Partition registers into argument, temporary (caller-saved), and preserved (callee-saved) sets, and allocate stack slots for spilled return addresses and frames.

Register allocation:

- `r0`: Constant zero.
- `r1`: Return address `ra` (caller-saved by leaf, callee-saved across calls).
- `r2`: Stack pointer `sp`.
- `r3, r4`: Arguments and return values `a0, a1` (caller-saved).
- `r5`: Temporary `t0` (caller-saved).
- `r6, r7`: Saved registers `s0, s1` (callee-saved). Stack frame: Prologue decrements `sp` by 16 bytes, storing `ra` at `8(sp)` and `s0` at `0(sp)`. Epilogue restores both and increments `sp` by 16 before `ret`.

Exercise 2.1.7: Differential Testing of 12-Bit Immediate Sign Extension

Generalisation transfer

Design a property-based differential test suite to verify that a hardware decoder and software emulator decode 12-bit signed immediates identically. Specify the boundary vectors and mathematical invariant property.

Construct test vectors covering positive maximum ($+2047$), negative minimum (-2048), zero, -1, and property-based mathematical equivalence.

Test vectors:

- Boundary values: $0, +1, -1$ (`0xFFF`), $+2047$ (`0x7FF` , maximum positive), -2048 (`0x800` , minimum negative).
- Bit-pattern isolates: single bit set at each position $0 \ldots 11$. Property invariant: For all $b \in [0, 4095]$, if $b < 2048$, decoded value must equal b; if $b \geq 2048$, decoded value must equal $b - 4096$. Assert that hardware RTL simulation and software instruction emulator yield identical extended 64-bit integers across all 4096 inputs.

References & Further Reading

- Patterson, D. A., & Hennessy, J. L. (2020). *Computer Organization and Design: The Hardware/Software Interface (RISC-V Edition)* (2nd ed.), Chapter 2 ("Instructions: Language of the Computer"). Morgan Kaufmann.
- Waterman, A., & Asanović, K. (Eds.). (2019). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA (Document Version 20191213)*. RISC-V International.
- Waterman, A., & Asanović, K. (Eds.). *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V International. https://github.com/riscv/riscv-isa-manual
- Weaver, D. L., & Germond, T. (1994). *The SPARC Architecture Manual (Version 9)*. Prentice Hall.

SECTION DETAIL

Assembly and Calling Conventions

Assembly language exposes the contract between source-level procedures and the instruction-set architecture. A compiler can hide register names and stack offsets in a high-level language, but it cannot avoid choosing where arguments live, how a return address survives a call, which registers a callee must preserve, or how an aggregate is laid out. These choices form an application binary interface (ABI). An ABI is not just a compiler preference: separately compiled modules, debuggers, profilers, operating-system signal handlers, foreign-function interfaces, and hand-written assembly all depend on it.

From source to machine state

DEFINITION

An Application Binary Interface (ABI) calling convention partitions architectural registers into **caller-saved** (volatile across a `call`, owned by the callee during execution) and **callee-saved** (non-volatile across a `call`, which the callee must restore to their entry values before `ret`), and fixes the stack-pointer alignment and argument/return locations at every procedure boundary.

Consider a function $f(x, y) = x + 2y$. At the source level, parameters are abstract values and the result is an abstract integer. At the assembly level, an ABI may place `x` and `y` in argument registers, compute `2y` with a shift or add, add `x`, place the result in a return register, and jump back through a link register. The function's correctness has two parts: its arithmetic result is correct, and it preserves every part of the calling convention it does not own.

The compiler's liveness analysis identifies values needed after each instruction. A register can be reused only when its old value is dead or has been saved. A callee-saved register is an interprocedural liveness contract: the caller may keep a value there across a call, so the callee must preserve it. A caller-saved register is scratch space: the caller must save it if it remains live across a call. Neither classification makes a register intrinsically safe or unsafe; it specifies responsibility.

TRACE THE CALL

Evaluate which statements accurately describe the binary interface obligations established by an ABI calling convention:

Answer: Calling conventions define explicit ownership rules dividing hardware registers into caller-saved and callee-saved partitions.; ABIs specify required stack pointer alignment (such as 16 bytes in RV64) and argument spill frame layouts.

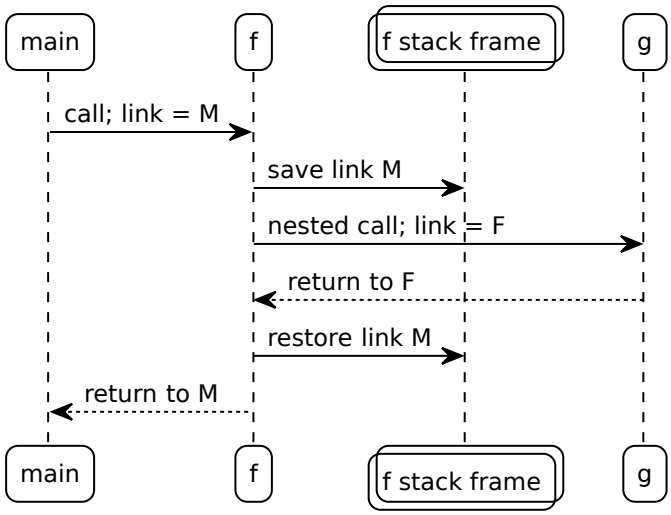
Calling conventions establish binary contracts for argument registers, callee-saved preservation, return values, and stack alignment, enabling independently compiled objects to interoperate correctly.

Calls, returns, and link state

A call transfers control to a target while recording a return address. One ISA may write the link into a dedicated register; another may push it on the stack; a compiler may synthesize either using ordinary instructions. A return loads the saved address into the PC. If the callee itself makes another call and uses the same link register, it must save the caller's link first. Leaf functions that make no calls can sometimes leave it in the link register, but a function cannot assume it is a leaf if a later implementation introduces a call for profiling, allocation, or error handling.

Worked call trace. Let `main` call `f`, and `f` call `g`. At the call from `main`, the link register holds the address after the call. On entry to `f`, that link is its return address, but `f` must save it before calling `g`. The call to `g` overwrites the link with the address after the nested call. After `g` returns, `f` restores its saved link and returns to `main`. If `f` omits the save, it returns to the instruction after the call to `g`, often looping or executing the wrong code. This failure is independent of whether the arithmetic inside `f` is correct.

The sequence exposes the two distinct continuations. `M` means the instruction after `main`'s call; `F` means the instruction after `f`'s call. The stack slot is private to this invocation of `f`.



> ***Pause & Predict: The Silent Leaf Function Bug.*** *Suppose an engineer writes a performance-critical leaf function in assembly that performs no function calls. It uses* `t0..t2` *for computation, reads arguments from* `a0..a1`*, and returns using* `ret` *(jumping to* `ra`*) without allocating a stack frame. Later, another developer adds an assertion macro:* `assert(val >= 0);`*. If the assertion fails, the macro calls* `__assert_fail(msg, file, line)`*. What catastrophic bug occurs when the assertion fails? Commit your reasoning before opening the analysis below.*

Think about what happens to the return address register `ra` when the processor executes a call instruction (`call` / `jal`).

Analysis: The function silently **ceases to be a leaf function** along the failure branch! When `jal __assert_fail` executes, the hardware automatically overwrites the return register `ra` with the address of the instruction immediately following the call to `__assert_fail`. Because the function never saved the caller's original `ra` onto the stack in a prologue, the caller's continuation address is permanently destroyed. If `__assert_fail` returns or attempts to unwind, execution will loop infinitely or jump into invalid memory. **Rule**: Any function that can execute even a conditional, exceptional, or debugging call must be compiled with a non-leaf stack frame that spills `ra`.

Transfer check. Suppose `g` itself calls `h`. Does `f` need an extra slot to preserve `g`'s return address?

REMARK **Continuation ownership in nested calls**

When `g` itself calls `h`, `f` does not need an extra slot to preserve `g`'s return address. `g` owns preservation of its incoming link before calling `h`; its invocation needs a separate save location. `f` preserves only its own continuation to `main`. A global shared return-address slot would fail under recursion or reentrancy because another invocation would overwrite it.

THE CALLEE CORRUPTED ITS CALLER

A function stores a temporary value in a callee-saved register and returns without restoring it. Its own return value is correct, but the caller later computes with corrupted state.

Evaluate which diagnostic evidence demonstrates this ABI contract violation:

Reference decision: Inspecting callee-saved register contents at function entry and comparing them immediately upon return reveals unpreserved caller state.

Calling conventions are boundary contracts. Comparing callee-saved registers before entry and after return definitively exposes whether a callee restored caller state.

Stack frames and alignment

The stack is a conventionally downward-growing region, though the ISA need not require that direction. A frame may contain the return address, saved callee registers, local variables, spill slots, outgoing arguments, and alignment padding. The stack pointer identifies the current allocation boundary; a frame pointer, when used, provides a stable reference if the stack pointer moves during the procedure.

A prologue allocates space and saves required state. An epilogue restores state, deallocates space, and returns. The ABI typically requires a stack alignment such as 16 bytes at a call boundary so that vector values, nested frames, and system interfaces can assume aligned storage. Alignment padding is not available for arbitrary data unless the frame layout accounts for it.

Worked frame layout. Suppose a function needs a saved return address (8 bytes), two saved registers (16 bytes), a 24-byte local array, and 16-byte alignment at calls. The raw total is 48 bytes, already a multiple of 16; a 48-byte frame can be allocated. If the raw total were 40 bytes, the frame would need 8 bytes of padding to reach 48. The prologue must establish the final stack pointer before storing at offsets, and the epilogue must restore in the reverse ownership order. A hand-written function that allocates 40 bytes while the ABI requires 16-byte alignment can fail only when a callee uses an aligned load, making the bug input- or call-path-dependent.

Stack frames are not automatically safe storage. A pointer to a local that escapes after return refers to reclaimed storage; a buffer write beyond its bounds can overwrite a return address or saved register. The architecture supplies memory access and privilege rules, while the ABI and compiler supply layout conventions. Control-flow integrity, shadow stacks, stack canaries, and memory tagging are additional defenses, not consequences of an ordinary call instruction.

Parameter passing and return values

ABIs classify scalar arguments, floating-point arguments, vectors, and aggregates. Small values often travel in registers; larger or variably sized values may use memory. A structure return may be implemented by a hidden pointer argument to caller-provided storage, shifting the visible argument positions. Variadic functions need metadata or a register-save area so the callee can retrieve arguments beyond those named in the prototype.

Worked example: placing a mixed argument list (RV64 psABI). Consider

```
struct pair { long x, y; };            /* 16 bytes = 2 x XLEN */
long sum4(long a, long b, long c, struct pair p, long d);
```

`a`, `b`, and `c` occupy `a0`–`a2`. The 16-byte `p` occupies the register pair `a3`/`a4`, with the lower-addressed eight bytes in `a3`; `d` then occupies `a5`. The placement is not left to the compiler's preference: the psABI fixes it so separately compiled callers and callees agree. Three consequences follow.

- If `p` were 24 bytes, it would exceed $2 \times \mathrm{XLEN}$ and be passed by reference: the visible argument list would contain the address of `p` in `a3`, and `d` would move to `a4`.

- If only one argument register remained for a 16-byte aggregate, the first eight bytes would go in that register and the second eight bytes on the stack; the psABI does not pad the aggregate to force a register pair.
- A returned `struct pair` uses `a0` / `a1`. A larger return aggregate is returned by reference: the caller allocates storage and passes its address as an implicit first argument, shifting the visible arguments by one.

Narrow scalars are widened by the same agreement. An integer argument narrower than XLEN is widened according to its signedness to 32 bits and then sign-extended to XLEN. Consequently a 32-bit `unsigned int` with bit 31 set arrives with the upper 32 bits set, not zeroed; a callee that reads the full register as a zero-extended 64-bit value is wrong. A hand-written callee that reads only the low 32 bits is immune to the upper bits.

The caller and callee must agree on signedness and width. Passing a 32-bit value in a 64-bit register may require sign extension or zero extension. A callee that assumes the wrong extension can interpret `0xFFFFFFFF` as -1 or 4,294,967,295. The same concern applies to Boolean values, pointers, and enum representations. ABI documentation is therefore part of the type system at a module boundary.

Recursion and reentrancy

Recursion works because each invocation has distinct storage for its parameters, locals, and return continuation. A recursive factorial function must preserve its current argument and return address while the smaller call executes. The invariant is that every active frame contains enough information to resume its caller and combine the recursive result with its own pending operation.

Worked recursion trace. For `fact(3)`, the first frame stores 3 and a continuation “multiply by 3.” The second stores 2 and continuation “multiply by 2.” The third stores 1 and returns 1. The second resumes, computes 2, returns; the first resumes, computes 6, returns. If the implementation keeps the argument only in a caller-saved register across the recursive call, the nested invocation may overwrite it and the multiplication uses the wrong value. A stack slot or callee-saved register is required.

Reentrancy adds interruption. An interrupt or signal handler may run while a procedure has partially updated a frame or global data. The handler’s ABI specifies which registers it saves and how it returns. Lock-free or signal-safe code must also avoid calling routines that assume nonreentrant global state. The ordinary function-call proof is insufficient if asynchronous control transfers can observe intermediate state.

Tail calls and frame reuse

A tail call occurs when a function's final action is to call another function and return its result without additional work. The current frame can be reused if arguments are placed according to the callee's convention and the caller's continuation is no longer needed. This turns recursion into iteration for tail-recursive functions and prevents unbounded stack growth.

Worked example: tail recursion versus a pending multiplication. The natural factorial, for $n \geq 0$,

```
long fact(long n) { return n <= 1 ? 1 : n * fact(n - 1); }
```

is not a tail call: after `fact(n - 1)` returns, the caller must still multiply by `n`, so every level needs a live frame holding `n` and `ra`. Rewriting with an accumulator,

```
long fact_acc(long n, long acc) {
    return n <= 1 ? acc : fact_acc(n - 1, n * acc);
}
```

makes the recursive call the final action. Every level has the same argument layout, so the implementation can place the new `n` and `acc` in `a0` / `a1` and jump to the entry point without writing a new link:

```
fact_acc:
    li   t0, 1
    ble  a0, t0, .Ldone
    mul  a1, a0, a1        # acc = n * acc
    addi a0, a0, -1
    j    fact_acc          # tail call: ra still points to fact_acc's
caller
.Ldone:
    mv   a0, a1
    ret
```

For `fact_acc(4, 1)`, the trace is `(4,1) -> (3,4) -> (2,12) -> (1,24)`, returning 24 with one active frame rather than four. The reuse is valid only because the callee's argument layout matches the current one, every callee-saved register has already been restored, and no cleanup or unwind action remains. If the implementation emitted `call fact_acc` instead of `j fact_acc`, it would overwrite `ra`; without saving the caller's link, the eventual return would be incorrect. An ordinary correct recursive call must preserve that link, typically using a separate frame per invocation.

Tail-call eligibility is constrained by ABI details: outgoing arguments may require more space, callee-saved registers must already be restored, and a frame address observable by debugging or exception machinery may prevent reuse. A compiler cannot apply tail-call elimination merely because the source expression looks final; it must preserve unwind, security, and calling-convention semantics.

Spills and register allocation

The register allocator colors an interference graph or uses a related strategy to assign live values to a finite register set. Values whose live ranges overlap cannot share a register. When pressure exceeds available registers, the allocator inserts spills: stores to stack slots and reloads before uses. Spilling changes instruction count, memory traffic, cache behavior, and possibly exception points.

Worked liveness example. In a loop with induction variable `i`, pointer `p`, accumulator `sum`, and temporary `t`, `i`, `p`, and `sum` may be live across the loop body while `t` is dead after its use. If a call occurs inside the body, the three long-lived values must be placed in callee-saved registers or saved by the caller around the call. A naive allocator that counts variables rather than live ranges may spill unnecessarily; an allocator that ignores the call boundary may lose values.

Debugging and unwinding

A debugger needs to reconstruct source-level frames from machine state. Frame pointers make this straightforward but consume a register; frame-pointer omission relies on compact unwind metadata describing how to recover the previous stack pointer and return address. Exceptions and signals need equivalent metadata to walk through optimized code, inlined functions, and partially completed prologues.

Worked example: reconstructing one caller frame. Suppose the current function uses `s0` as a frame pointer that points to the caller's stack pointer, and it has stored `ra` at `s0 - 8` and the caller's `s0` at `s0 - 16`. With `s0 = 0x8000_0020`, the saved `ra` is at `0x8000_0018` and the saved `s0` at `0x8000_0010`. If memory contains `ra = 0x4000_01a4` and saved `s0 = 0x8000_0040`, the debugger reconstructs the caller as `PC = 0x4000_01a4`, `sp = 0x8000_0020`, and `s0 = 0x8000_0040`; the next frame's saved `ra` is then at `0x8000_0038`. The chain is a linked list through memory, not a language-level call stack.

Compilers that omit the frame pointer encode the same facts as unwind rules tied to PC ranges. A minimal CFI fragment for a 32-byte frame is

```
fib:
    .cfi_startproc
    addi sp, sp, -32
    .cfi_def_cfa_offset 32
    sd   ra, 24(sp)
    .cfi_offset ra, -8
    sd   s0, 16(sp)
    .cfi_offset s0, -16
    ...
    .cfi_endproc
```

The CFA is the caller's `sp`, `sp + 32` after allocation. `.cfi_offset ra, -8` means the saved `ra` lives at `CFA - 8`, which is `24(sp)`; `.cfi_offset s0, -16` means `16(sp)`. Unwinding from any PC uses the row that covers that PC, so a prologue that adjusts `sp` before updating the CFI row becomes unwalkable in that interval. This is why hand-written assembly that omits `.cfi_*` directives can return correctly yet still break exception propagation or a profiler's backtrace.

A correct ABI implementation therefore has a temporal prologue contract: at each instruction boundary where an asynchronous event may occur, the unwind rule must describe where saved registers and the return address reside. Hand-written assembly that omits unwind metadata can break exception propagation even if ordinary calls return correctly.

Security-sensitive call boundaries

Indirect calls and returns use target addresses that may be corrupted by memory bugs or speculative mechanisms. Control-flow integrity restricts targets to a valid policy; shadow stacks maintain a protected copy of return addresses; pointer authentication or capabilities can bind pointers to context on suitable ISAs. These mechanisms change the calling convention's security properties but still rely on the same basic invariant: a return must transfer control to the intended continuation.

ABI boundaries also define privilege transitions. A system-call stub places arguments in kernel-defined registers, executes a trap instruction, and handles the return convention. The kernel validates pointers and sizes before accessing user memory; it cannot trust that a user-provided pointer remains mapped or unchanged between checks and use. The architecture supplies privilege separation, but correctness requires a complete protocol across the boundary.

Worked Trace: RISC-V (RV64) Stack Frame Lifecycle in Recursive Call

Consider compiling the recursive Fibonacci step:

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Under the standard RISC-V standard calling convention (psABI):

- `a0` holds the incoming argument `n` and the return value.
- `ra` holds the return link address.
- `s0` is a callee-saved register preserved across calls.
- The stack pointer `sp` must remain 16–byte aligned at all call boundaries.

The function must preserve three values across calls: `ra`, the caller's original `s0`, and the result of `fib(n - 1)` while `fib(n - 2)` runs; it also keeps `n` in the callee-saved `s0`. Those values need distinct storage. The saved `s0` and the spill cannot share a slot: storing the spilled result into the slot that holds saved `s0` silently replaces the caller's `s0` with a Fibonacci value, and the epilogue then restores the wrong register.

Step 1: Frame Allocation & Register Preservation (Prologue)

```
fib:
    addi sp, sp, -32      # 32 bytes: two saved registers + one spill
slot, 16-byte aligned
    sd   ra, 24(sp)       # Save return address link
    sd   s0, 16(sp)       # Save callee-preserved register s0
                          # 0(sp) and 8(sp) remain available for spills
```

Step 2: Base Case Test & Recursive Call Preparation

```
    li   t0, 1
    ble  a0, t0, .Lbase   # If n <= 1, jump to base case return
    mv   s0, a0           # Preserve original n in callee-saved s0
across nested calls
    addi a0, a0, -1       # Argument a0 = n - 1
    call fib              # Recursive call: fib(n - 1), returns result
in a0
```

Step 3: State Reconstruction Across Calls

```
    # a0 contains fib(n - 1). Spill it to its own slot while computing
fib(n - 2):
    sd   a0, 0(sp)        # Spill fib(n - 1) to the dedicated slot at
0(sp)
    addi a0, s0, -2       # a0 = n - 2 (using preserved s0)
    call fib              # Recursive call: fib(n - 2), returns result
in a0
    ld   t1, 0(sp)        # Reload fib(n - 1) from its slot
    add  a0, a0, t1       # Return value = fib(n - 2) + fib(n - 1)
    j    .Lexit
```

Step 4: Epilogue & Stack Restoration

```
.Lbase:
    # a0 already holds n (0 or 1)
.Lexit:
    ld   s0, 16(sp)       # Restore caller's original s0 from its own
slot
    ld   ra, 24(sp)       # Restore caller's original return address
    addi sp, sp, 32       # Deallocate stack frame
    ret                   # Return to caller via jr ra
```

The 32–byte size is not arbitrary: after the 16–byte-aligned `sp` is decremented by 32, it remains 16–byte aligned at each `call`. Slots `0(sp)` and `8(sp)` are free spill space; the saved `s0` at `16(sp)` and saved `ra` at `24(sp)` are never reused for temporaries. A 16–byte frame with the spill written to `0(sp)` overwrites saved `s0`. The function may then return the correct Fibonacci number while corrupting its caller's callee-saved register, a calling-convention violation that appears only after the caller uses `s0`.

COMPLETE THE RV64 NON-LEAF PROLOGUE AND EPILOGUE

```
fib:
  addi sp, sp, -32
  sd   ra, 24(sp)
  sd   s0, 16(sp)
  ...
.Lexit:
  ld   s0, 16(sp)
  ld   ra, 24(sp)
  addi sp, sp, 32
  ret
```

The prologue decrements sp by 32 (preserving 16–byte alignment), saves ra at 24(sp) and s0 at 16(sp), leaving 0(sp)..15(sp) for spill slots. The epilogue restores s0 from 16(sp) and ra from 24(sp) before adding 32 back to sp.

Counterexample & Failure Mode: Stack Smashing & Return Address Corruption

What happens when software violates stack frame boundaries? Consider a standard C routine that allocates a stack buffer:

```c
void vulnerable_read() {
    char buf[16];
    gets(buf); // Unbounded user input
}
```

1. **Stack Memory Layout**: For one common 64-bit frame layout, the compiler allocates 16 bytes for `buf` at `0(sp)`, then one saved callee register at `16(sp)`, then the saved return address `ra` at `24(sp)`. The exact offsets are ABI- and compiler-dependent; the invariant that matters is that `ra` is stored above the buffer in this layout, so a write past the buffer can reach it.

2. **The Buffer Overflow Attack**:

   - If an attacker supplies 24 bytes of payload, the buffer fills `0..15` and clobbers saved registers `16..23`.
   - If the attacker supplies 32 bytes, the final 8 bytes (`24..31`) directly overwrite the saved return address `ra`.

3. **Catastrophic Failure Mode**: When `vulnerable_read` executes its epilogue:

   ```
   ld  ra, 24(sp)  # Loads attacker-controlled bytes into ra
   ret             # Jumps directly to the attacker's address (PC =
   attacker_data)
   ```

   Instead of returning to `main`, the processor branches into arbitrary payload code or immediately crashes with `SIGSEGV`. This demonstrates that stack frames are purely logical abstractions over raw linear memory: hardware enforces no intrinsic partition between control metadata (`ra`) and local data arrays unless explicit hardware defenses (e.g., shadow stacks, non-executable stack pages, stack canaries) are activated.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.2.1: Recursive Stack Frame Layout and 16-Byte Alignment

Construction application

Design a stack frame for a recursive function with three 64–bit local variables, two function calls, one 32–byte vector spill area, and a 16–byte ABI alignment requirement. Show exact byte offsets, prologue allocation, and epilogue deallocation.

Account for return address, callee-saved registers, local variables, and alignment padding.

Account for saved `ra` (8 B), saved `s0` (8 B), three local variables ($3 \times 8 = 24$ B), and vector spill (32 B). Raw sum = $8 + 8 + 24 + 32 = 72$ B. To satisfy 16–byte alignment, round up to 80 B (adding 8 B padding). Prologue decrements `sp` by 80 and saves `ra` and `s0`; epilogue restores them and increments `sp` by 80 before `ret`.

Exercise 2.2.2: Link Register Overwrite in Nested Non–Leaf Invocations

Verification application

Trace a non-leaf function f that calls helper g. Explain what occurs if f omits saving register `ra` onto the stack before issuing `jal g`, and identify the instruction address loaded into `PC` when f executes `ret`.

Trace what happens to the return address register ra when f invokes another procedure g.

When `f` executes `jal g`, the CPU hardware overwrites `ra` with the address of the instruction immediately following the call to g. If f failed to spill its incoming return address to the stack, when f subsequently executes `ret` (`jr ra`), it jumps to the instruction after the call to g rather than returning to its original caller, entering an infinite loop or causing control-flow corruption.

Exercise 2.2.3: Caller-Saved vs. Callee-Saved Register Classification

Description recall

Classify registers `t0..t6`, `s0..s11`, and `a0..a7` under the standard RISC-V calling convention. Explain the interprocedural liveness contract governing which function bears responsibility for preservation across calls.

Identify which registers must be preserved across procedure calls versus which can be clobbered.

`t0..t6` and `a0..a7` are caller-saved (temporary/argument) registers; if the caller needs their values after a call, the caller must spill them before issuing `jal`. `s0..s11` are callee-saved registers; the caller can assume their values survive untouched across calls, obligating the callee to spill and restore any of them it mutates.

Exercise 2.2.4: Inductive Proof of Stack Frame Isolation

Proof synthesis

Formulate an inductive proof over recursion depth n showing that a recursive factorial routine preserves its intermediate multipliers when parameters are stored in stack frame slots rather than caller-saved registers.

State the base case and inductive step showing each frame retains isolated arguments and continuation.

Base case ($n \leq 1$): The function returns 1 without recursive calls, preserving frame integrity. Inductive step: Assume frame isolation holds for recursion depth k. At depth $k+1$, the routine spills its argument $k+1$ and continuation `ra` into its private frame before calling `fact(k)`. By induction, the child call executes at a distinct stack offset (`sp - frame_size`) without modifying memory at `sp`. Upon child return, the caller reloads $k+1$ from its intact private frame and computes $(k+1) \cdot \mathrm{fact}(k)$, preserving the invariant for all n.

Exercise 2.2.5: Tail Call Optimization Prerequisites and Frame Deallocation

Verification application

An optimizing compiler attempts to replace `call g; ret` with a direct jump `j g`. Specify three conditions under which this tail call optimization is illegal due to caller stack frame lifetime constraints.

Check whether the caller still requires local stack variables after the callee returns.

Tail-call optimization is invalid if: (1) the caller passes pointers to its own local stack variables as arguments to g; (2) the caller has pending deferred cleanup or destructor actions that must run after g completes; or (3) the callee requires larger outgoing stack arguments than the caller's frame can host without reallocation.

Exercise 2.2.6: Interference Graph Construction for Register Allocation

Construction application

Construct a register interference graph for four variables $v1, v2, v3, v4$ whose live ranges are $[1, 4]$, $[2, 5]$, $[3, 6]$, and $[5, 7]$. Determine the minimum number of registers required to color the graph without spills.

Add an undirected edge between any two variables whose live ranges overlap at any program point.

Interference edges connect variables whose intervals overlap:

- v_1 overlaps with v_2 (at 2, 3, 4) and v_3 (at 3, 4).
- v_2 overlaps with v_1, v_3 (at 3, 4, 5), and v_4 (at 5).
- v_3 overlaps with v_1, v_2, and v_4 (at 5, 6).
- v_4 overlaps with v_2, v_3. The maximum clique $\{v_1, v_2, v_3\}$ has size 3 (active simultaneously at time 3 and 4). Thus, the chromatic number is 3: three physical registers are both necessary and sufficient to allocate all variables without spilling.

Exercise 2.2.7: Pointer Validation and Security at System-Call Boundaries

Generalisation transfer

Explain why an operating system kernel cannot rely on user-space pointers having been verified prior to executing an `ecall` / `syscall` instruction, and prescribe the required kernel validation procedure.

Consider concurrency, unmapped memory, and privilege escalation if user addresses are dereferenced directly.

User-space pointers cannot be trusted because concurrent user threads can unmap or remap memory (TOCTOU race), pass addresses residing within kernel address space to read protected data, or trigger page faults inside critical kernel code. The kernel must validate that the user pointer and range reside entirely within the user address partition, verify memory permissions, and perform transfers using copy-with-fault-handling primitives (`copy_from_user`).

References & Further Reading

- Patterson, D. A., & Waterman, A. (2017). *The RISC-V Reader: An Open Architecture Atlas*, Chapter 3 (“Assembly Language and Calling Conventions”). Strawberry Canyon LLC.
- Bryant, R. E., & O’Hallaron, D. R. (2016). *Computer Systems: A Programmer’s Perspective* (3rd ed.), Chapter 3 (“Machine-Level Representation of Programs”). Pearson.
- RISC-V International (2023). *RISC-V ABIs Specification (psABI)*. GitHub Repository: `riscv-non-isa/riscv-elf-psabi-doc`.
- DWARF Standards Committee (2017). *DWARF Debugging Information Format, Version 5*. https://dwarfstd.org/doc/DWARF5.pdf.

SECTION DETAIL

Datapath and Control

A processor is not an instruction-shaped black box. It is a coordinated network of storage elements, combinational operators, multiplexers, decoders, and control state. The *datapath* carries and transforms values; the *control unit* determines which values move, which operation an arithmetic unit performs, which registers are written, and how the next program counter is selected. Separating these roles makes a design analyzable: the datapath can be checked against value-flow equations, while control can be checked against an instruction-state transition table.

This chapter builds a small single-cycle and multicycle processor model. The exact ISA is abstract, but the method applies to RISC-V, MIPS-like teaching processors, and custom embedded cores. The objective is not to memorize a block diagram. It is to derive the minimum hardware needed by a set of instruction semantics, identify resource conflicts, and prove that control signals cause no unintended architectural side effects.

From instruction semantics to resources

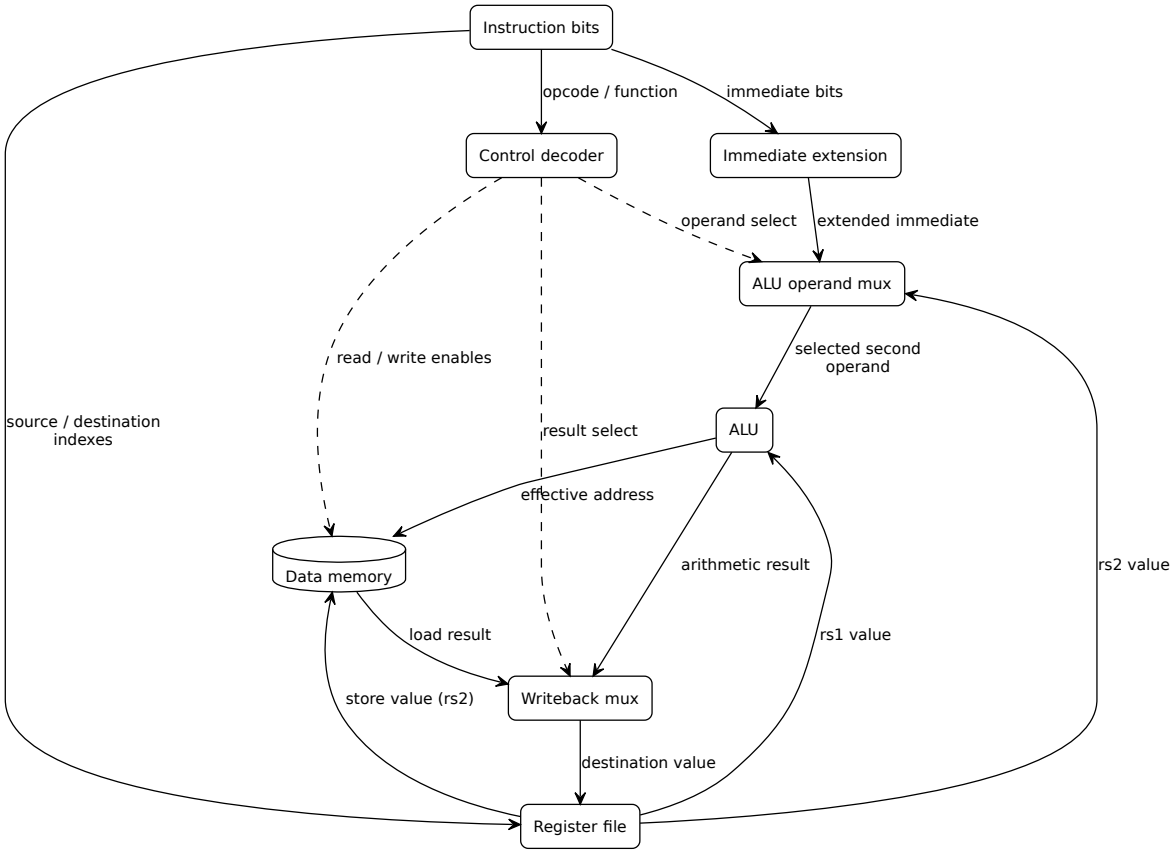
Consider four instruction families: register arithmetic `rd ← rs1 op rs2`, load `rd ← Mem[rs1+imm]`, store `Mem[rs1+imm] ← rs2`, and conditional branch `if rs1=rs2 then PC←PC+offset`. Their semantic equations identify the required resources:

- a program counter and instruction-memory read for fetch;
- an instruction register or equivalent decode path;
- a register file with two read ports and at least one write port;
- an immediate generator that extracts and extends encoding fields;
- an ALU for arithmetic, effective-address calculation, comparison, and target calculation;
- data memory with read and write controls;
- multiplexers selecting ALU operands, write-back data, and next PC;
- a control decoder deriving signals from opcode and function fields.

The list is a design hypothesis, not yet an implementation. One ALU may serve every role in a single-cycle design if all operations fit in the cycle; a multicycle design can reuse it across cycles. A register file may read combinationally and write on a clock edge, or both operations may be pipelined. The architecture is defined by the visible transition, while the implementation chooses resource sharing under timing, area, and power constraints.

Worked example: register arithmetic. The instruction fetches from address `PC`, decodes `rs1`, `rs2`, and `rd`, reads both operands, computes `ALU(rs1,rs2)`, writes the result to `rd` at the clock edge, and sets `PC←PC+4`. The control signals include `RegWrite=1`, `ALUSrc=0`, `MemRead=0`, `MemWrite=0`, `MemToReg=0`, and `PCSrc=sequential`. If `MemWrite` were accidentally high, the instruction would have an unintended store side effect even though its arithmetic result was correct. Correctness includes negative behavior: controls for irrelevant resources must disable them.

The data paths distinguish the store value from its effective address: the ALU computes `rs1 + immediate`, but the register file supplies `rs2` as the bytes to store. Opcode bits go to control decoding; register indexes select operands. The diagram omits next-PC selection to isolate this arithmetic/load/store slice.


Instruction bits
opcode / function
immediate bits
Control decoder
Immediate extension
operand select
extended immediate
ALU operand mux
read / write enables
selected second operand
source / destination indexes
result select
ALU
effective address
Data memory
arithmetic result
rs2 value
load result
rs1 value
store value (rs2)
Writeback mux
destination value
Register file

SEPARATE MECHANISM AND CONTROL

What is the role of control logic in a datapath?

Answer: It selects and sequences datapath operations according to instruction semantics

The datapath provides resources such as registers, ALU, and buses. Control signals select how those resources realize each instruction, often across several steps.

Single-cycle organization

In a single-cycle processor, one instruction completes between two clock edges. The path includes instruction memory, decode, register read, immediate generation, ALU, data memory if needed, write-back selection, and state updates. The clock period must exceed the longest instruction path plus register overhead. A load usually determines the period because it needs fetch, decode, address calculation, memory access, and write-back.

Single-cycle timing makes the control truth table easy to understand but wastes hardware time for short instructions. An add waits for the load-sized cycle even though it does not access data memory. The design also needs enough combinational resources to perform all phases without intermediate storage. A branch needs both a comparison and a target calculation; if the same ALU cannot do both in the cycle, the design adds hardware or changes the timing model.

Worked timing comparison. Suppose instruction fetch is 300 ps, register read 150 ps, ALU 200 ps, data memory 400 ps, register setup and clock overhead 100 ps, and multiplexing adds 50 ps. A load path may be $300 + 150 + 200 + 400 + 50 + 100 = 1200$ ps, so the clock cannot be shorter than 1.2 ns. An add may use only $300 + 150 + 200 + 50 + 100 = 800$ ps but still completes once every 1.2 ns. The throughput is one instruction per cycle; the latency is one cycle. Improving the add unit does not raise the clock until the load path or another critical path changes.

TRACE THE WRONG-PATH WRITE

A taken branch computes the correct target, but the next sequential instruction still writes a register before the PC redirects.

Which trace best localizes the control defect?

Reference decision: PC select, branch decision timing, pipeline validity, and register-write enable

The target calculation may be correct while the validity or write-enable path is late. A cycle trace connecting branch resolution to PC selection and side-effect suppression reveals the faulty control transition.

Multicycle organization

DEFINITION

A multicycle datapath decomposes instruction execution into a finite-state machine over non-architectural holding registers (`IR`, `MDR`, `A`, `B`, `ALUOut`), allowing a single memory port and a single ALU to be time-multiplexed across fetch, decode, address/execute, memory, and write-back cycles.

A multicycle design inserts state registers between conceptual phases: fetch, decode/register read, execute or address calculation, memory access, and write-back. Each phase can use a shorter clock period, and expensive resources such as one ALU or one memory port can be reused. The price is multiple cycles per instruction and a finite-state control machine that must preserve temporary values.

The program counter, instruction register, memory-data register, operand registers, and ALU-result register hold intermediate state. Fetch loads the instruction and increments the PC. Decode reads operands into registers. Execute computes an arithmetic result, effective address, or branch decision. A load's memory phase reads data into a temporary register; write-back then writes that value to the destination. A store ends after memory write, while an add can write back directly from the ALU-result register.

Control is now a temporal sequence, not a single opcode table. A state transition may depend on the decoded instruction class. The invariant is that each temporary register has a defined meaning in each control state. If an `ALUOut` register contains a branch target in one path and an arithmetic result in another, the write-back control must select it only under the corresponding instruction class.

Worked sequence: load. In `FETCH`, set memory address to PC, assert instruction-register write, select `PC+4` for the ALU, and update PC. In `DECODE`, read register operands and sign-extend the immediate. In `MEM_ADDR`, select base and immediate for the ALU and write the effective address to `ALUOut`. In `MEM_READ`, use `ALUOut` as data-memory address and load `MDR`. In `LOAD_WB`, select `MDR`, assert register-file write, and choose `rd`. A store follows the same address state but transitions to `MEM_WRITE`; an add transitions from execute to `ALU_WB`.

Control signals as a typed interface

> **WARNING**
>
> Treating write-enable control signals (`RegWrite`, `MemWrite`, `PCWrite`) as synthesis "don't-cares" on unused or illegal opcodes is a critical correctness bug: any architectural state element must default to write-disabled (`0`) unless the current instruction and state explicitly authorize mutation.

Control signals are often presented as bits, but their meanings are typed choices. `ALUSrc` may select register or immediate; `ResultSrc` may select ALU result, memory data, or PC-relative value; `PCSrc` may select sequential PC, branch target, jump target, or exception vector. Treating multi-way choices as independent Boolean bits creates illegal combinations. A decoded control record or enumerated signal makes illegal states easier to identify.

For an instruction class, build a control table with columns for every potentially side-effecting resource. A register arithmetic instruction writes exactly one destination, a load writes one destination from memory, a store writes memory and no register, and a branch writes no data state but may write PC. The table should include exceptions and illegal encodings. An unimplemented opcode must not inherit `RegWrite=1` from a neighboring decode row.

Worked control table reasoning. For a store, `RegWrite=0`, `MemWrite=1`, `MemRead=0`, `ALUSrc=1`, and `ResultSrc` is irrelevant because no write-back occurs. The data-memory address is the ALU result; store data comes from the second register operand. For a branch, `RegWrite=0`, `MemWrite=0`, `ALUSrc` may be 0 for comparison, and `PCSrc` is conditional on the ALU equality result. Marking irrelevant controls as "don't care" can help synthesis, but verification should constrain them to safe values where a malformed decode could otherwise cause an effect.

ALU sharing and operation encoding

EXAMPLE

A two-level ALU decoder maps the main control unit's 2-bit `ALUOp` (`00` for load/store address addition, `01` for `beq` subtraction comparison, `10` for R-type/I-type function decoding) together with `funct3` / `funct7` bits into the concrete ALU operation selector, isolating instruction-format decoding from arithmetic-unit control.

The ALU must support the operations required by the ISA: addition, subtraction, Boolean functions, shifts, comparisons, and perhaps address generation. Subtraction often supplies equality and signed/unsigned less-than predicates. A control field selects the operation. The decoder should derive ALU control from the instruction's primary opcode and function bits, while defaulting to a harmless operation for invalid combinations.

Address generation deserves special attention. A load and store use addition, but they differ in memory direction and write-back. Reusing the same ALU reduces area but means the control state must ensure its result is captured before another operation overwrites it. In a single-cycle design, no intermediate register is needed; in a multicycle design, `ALUOut` must be written in the address state and held through memory access.

Next-PC logic and control flow

DEFINITION

The next-PC transition function $\mathrm{PC}_{\mathrm{next}} = \mathrm{Mux}(\mathrm{PCSrc}, \mathrm{PC}+4, \mathrm{PC}+\mathrm{imm}_B, \mathrm{PC}+\mathrm{imm}_J, (R[rs1]+\mathrm{imm}_I)\ \&\ \sim 1, \mathrm{mtvec})$ selects the architectural continuation at the end of the instruction cycle, routing normal flow, taken branches, indirect jumps, and synchronous traps through one arbitrated state update.

Next-PC selection logic and target generation:

Instruction class or event	Selection condition	Target address arithmetic	`PCSrc` code
Normal sequential step	None (unconditional)	$\mathrm{PC} + 4$	`00`
Conditional branch (`beq`, `bne`, etc.)	Condition true ($\mathrm{Zero} = 1$, etc.)	$\mathrm{PC} + \mathrm{sext}(\mathrm{imm}_B)$	`01`
Direct jump and link (`jal`)	Unconditional	$\mathrm{PC} + \mathrm{sext}(\mathrm{imm}_J)$	`10`
Register-indirect jump (`jalr`)	Unconditional	$(R[rs1] + \mathrm{sext}(\mathrm{imm}_I)) \;\&\; \sim 1$	`11`
Architectural exception / trap	Synchronous fault or interrupt	mtvec (trap base vector)	`100`

The next PC is a selection problem. Common candidates are sequential `PC+instruction_length`, a conditional branch target, an unconditional jump target, a register-indirect target, and an exception vector. A branch target may be computed as `PC + sign_extend(offset << alignment)`; the shift encodes the fact that instruction addresses are aligned. The architecture must define whether the base is the current PC or the already incremented PC.

Branch condition and target selection must be coordinated. A not-taken branch chooses sequential PC even if the target arithmetic was computed. A taken branch must commit the target and suppress any unintended register or memory write from the branch itself. In a pipelined implementation, younger instructions may already be in flight; in a single-cycle or multicycle implementation, the control state can choose directly.

Exceptions add another PC candidate. A faulting load should record a restart address or faulting instruction address according to the ISA and transfer to a privileged handler. An asynchronous interrupt may be taken between instructions. The next-PC mux is therefore part of the protection boundary, not just a branch convenience.

Memory interfaces and side effects

> **WARNING**
>
> When a memory or MMIO port deasserts `ready` during a multicycle access, the control FSM must freeze state transitions and qualify `RegWrite` / `PCWrite` with `mem_ready`; otherwise the processor either latches garbage into `MDR` / `rd` or re-issues a non-idempotent MMIO store across multiple clock edges.

Worked cycle trace: multicycle memory stall on `lw`:

Cycle	Datapath state	`mem_req`	`mem_ready`	FSM action	State invariant
1	`MEM_ADDR`	1	0	Address generated in `ALUOut`; issued to bus	Enter wait state; freeze PC and IR
2	`MEM_WAIT`	1	0	Memory pipeline latency; data unready	Hold request; suppress `MDR` latch and `RegWrite`
3	`MEM_WAIT`	1	1	Memory asserts data valid	Latch read bus into `MDR`; advance FSM
4	`LOAD_WB`	0	—	Write `MDR` to register file $R[rd]$	Assert `RegWrite=1`; unfreeze fetch for next PC

An instruction or data memory interface has an address, read/write request, write data, read data, and completion or fault signal. A simple teaching model may respond in one cycle; a realistic memory can stall. The control unit must not advance state as though data arrived if `ready` is low. A load's destination write enable must be delayed until data is valid, and a store must not be repeated merely because the surrounding state was re-entered after a wait.

Idempotence matters. Repeating an instruction fetch is usually harmless; repeating a store to an ordinary memory location may not be if the store is a memory-mapped device write. A multicycle controller should use a transaction state or handshake protocol that distinguishes "request issued" from "response received." The ISA may define precise restart behavior, but the microarchitecture must ensure that a partially completed external side effect is not silently duplicated.

Verification by refinement and traces

REMARK

Datapath verification proves a simulation relation: at every instruction completion boundary, the architectural projection (PC, R, M) of the hardware state equals the single-step ISA transition $[\![\mathrm{instr}]\!](\sigma)$, and no spurious bus or register writes occurred during intermediate cycles.

Start with a reference transition for each instruction class. For every control state, write an assertion about which architectural state may change and which temporary values are valid. A trace table should show PC, instruction register, operand registers, ALU result, memory request, and write-back controls at each cycle for representative instructions.

For an add, prove that no memory write is asserted and that the value written to `rd` equals the reference ALU operation. For a load, prove that the address sent to memory is base plus properly extended immediate and that the eventual write-back occurs only after a successful read. For a fault, prove that the destination register remains unchanged and the next PC enters the handler with the documented cause.

Equivalence is not merely output comparison at the final cycle. A design that writes a wrong intermediate value then overwrites it before the end can pass a shallow test while violating a bus or side-effect contract. Observe memory requests, register write enables, privilege transitions, and handshake boundaries. Directed traces should include `rd=x0` or an architecturally ignored destination, maximum offsets, branch taken/not taken, memory stall, and illegal opcode.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.3.1: Minimal Datapath Resources

Construction application

Derive the minimal resources needed for register arithmetic, load, store, branch, and jump. For each, distinguish required from irrelevant controls.

Determine the distinct functional units required to read registers, perform ALU operations, and write back.

Arithmetic needs two register reads, ALU operation, one register write, and sequential PC. Load additionally needs effective-address ALU use, memory read, and memory-to-register selection. Store needs effective address, memory write, and store-data path but no register write. Branch needs comparison and target/PC selection but no data-state write. Jump needs target formation and PC selection.

Exercise 2.3.2: Multicycle State Machine Sequencing

Construction application

Draw a multicycle state machine for load, store, add, and branch using one ALU and one memory port. State which temporary register holds each intermediate value.

Trace execution steps from fetch and decode into instruction-specific memory, branch, or ALU states.

A sound sequence is `FETCH → DECODE →` instruction-specific execute. Load goes through `ADDR → MEM_READ → LOAD_WB`; store goes `ADDR → MEM_WRITE`; add goes `EXEC → ALU_WB`; branch goes `BRANCH`. `IR`, `MDR`, operand registers, and `ALUOut` preserve values between states.

Exercise 2.3.3: Single-Cycle Critical Path Timing

Verification application

Compute the single-cycle clock bound for a processor whose load path is 250 ps fetch, 180 ps decode, 220 ps ALU, 500 ps memory, 80 ps muxing, and 100 ps register overhead.

Sum the propagation delays across all units along the longest combinational path from clock to register setup.

The bound is $250 + 180 + 220 + 500 + 80 + 100 = 1330$ ps, so the maximum ideal frequency is about 752 MHz. The path is load-sized; optimizing only the add path does not change this bound.

Exercise 2.3.4: Stall Qualification for Memory Write-Back

Description transfer

Explain why a memory stall must not allow a load's write-back control to remain asserted across multiple cycles without a response qualifier.

Explain how memory latency stalls must block register destination updates until valid data arrives.

Without response qualification, the destination can be written repeatedly with stale or invalid data, and a stalled request may be duplicated. Write enable must be a conjunction of the write-back state and a valid successful response, with state advancing exactly once.

Exercise 2.3.5: Branch Control Flow and PC Update

Verification transfer

Write a trace for a taken branch and a not-taken branch. Include the PC base convention and prove that the branch instruction itself writes no general register.

Compare next-PC multiplexer selections when branch conditions evaluate to taken versus not-taken.

If PC denotes the current instruction address, target is `PC+offset`; if it denotes the already incremented address, target is `PC+4+offset`. A taken branch selects target, a not-taken branch selects sequential PC. Both set `RegWrite=0` and `MemWrite=0`.

Exercise 2.3.6: Default Safe Controls for Illegal Opcodes

Construction synthesis

Design safe default controls for an illegal opcode. Explain why “all don’t care” is unsafe when the decoder output feeds write enables.

Disable all state-modifying write enables and select benign ALU operations when decode fails.

Set all side-effecting enables low, select a benign ALU operation, choose sequential PC or a defined illegal-instruction trap, and prevent privileged state changes. Leaving enables as don’t-cares permits synthesis or a transient decode to activate a register or memory write.

Exercise 2.3.7: Load Instruction Refinement Proof

Formalisation synthesis

Give a refinement argument connecting a multicycle load implementation to the abstract ISA load transition.

Show that ALUOut captures effective address in cycle 2 and memory data reaches destination in cycle 4.

Show that after `FETCH` and `DECODE`, `ALUOut` equals the effective address; after `MEM_READ`, `MDR` equals the memory value or the machine enters the prescribed fault state; after `LOAD_WB`, only `rd` changes to the sign- or zero-extended `MDR` and PC is the correct next instruction. All other architectural state remains unchanged.

References & Further Reading

- MIT 6.004 Computation Structures: lecture notes and examinations
- MIT 6.004 learning objectives
- Cornell CS 3410: Computer System Organization and Programming

SECTION DETAIL

Pipelining

Pipelining improves instruction throughput by allowing different instructions to occupy different portions of a datapath at the same time. It does not make one instruction's combinational work disappear. Instead, registers divide a long path into stages and establish a repeating schedule. The architectural machine still appears to execute instructions in program order, while the implementation overlaps their internal phases.

The distinction between latency and throughput is fundamental. *Latency* is the time from an instruction entering the pipeline to its result becoming available or retiring. *Throughput* is the long-run completion rate after the pipeline is filled. A five-stage pipeline may have five-cycle instruction latency but complete one independent instruction per cycle. A pipeline that accepts one instruction per cycle but stalls frequently has a lower effective throughput than its ideal diagram suggests.

A pipeline as a timed refinement

DEFINITION

A k-stage pipeline partitions the single-cycle instruction transition into stage registers $R_1, \ldots, R_{k-1}$ such that an ideal sequence of N independent instructions completes in $k + N - 1$ clock cycles at clock period $T_{\mathrm{e}}^{\mathrm{pip}} \geq \max_{1 \leq i \leq k} (d_i + r_i)$, while retiring architectural state updates identical to sequential execution.

Suppose an unpipelined datapath performs fetch, decode/register read, execute, memory, and write-back in one cycle. Insert pipeline registers between these phases. The registers carry both data and control: an instruction's destination register, ALU operation, memory direction, and write-back choice must travel with the values they govern. A common five-stage model is IF, ID, EX, MEM, and WB.

At cycle t, instruction I_0 may be in EX while I_1 is in ID and I_2 is in IF. The hardware is not executing three complete instructions independently; it is executing three slices of three different instruction transitions. Correctness requires that each slice's data and controls remain aligned. A classic bug is to pipeline the operand value but not its destination register, causing a result to be written into the wrong architectural register several cycles later.

Worked timeline. For independent instructions I_0, I_1, I_2, I_3, I_4:

Cycle	IF	ID	EX	MEM	WB
1	I_0	-	-	-	-
2	I_1	I_0	-	-	-
3	I_2	I_1	I_0	-	-
4	I_3	I_2	I_1	I_0	-
5	I_4	I_3	I_2	I_1	I_0
6	-	I_4	I_3	I_2	I_1

The first result appears at cycle 5, but the fifth result appears at cycle 9. After filling, completion is one per cycle. For N instructions and k stages, ideal cycles are $k + N - 1$, not kN.

PIPELINE INVARIANT

What does pipelining change first: instruction semantics or the schedule of work?

Answer: It overlaps stages while preserving the architectural result

Pipelining is a timed implementation refinement. Hazards and control logic exist precisely to preserve the sequential semantics while overlapping work.

Clock period and stage balance

Each pipeline stage has a combinational delay d_i and a register overhead r_i including clock-to-Q, setup, skew, and margin. The clock period must exceed the maximum stage delay:

$$T \geq \max_i (d_i + r_i).$$

Splitting a path into more stages helps only if the new maximum decreases enough to justify added register overhead, bypass complexity, power, and branch-recovery cost. Perfectly equal stages are rarely possible because instruction fetch, memory access, and decode have different structures. A stage with 700 ps and four stages with 300 ps determines the clock; splitting a 300 ps stage further does nothing until the 700 ps bottleneck changes.

Worked speedup. An unpipelined path takes 10 ns. A four-stage partition has combinational delays 2.5 ns each and 0.1 ns register overhead, so its clock is 2.6 ns. For one instruction, unpipelined latency is 10 ns while pipelined latency is approximately $4 \times 2.6 = 10.4$ ns: the pipeline is slightly worse because of register overhead. For 100 independent instructions, the unpipelined time is 1000 ns; the pipeline takes $(4 + 100 - 1)2.6 = 267.8$ ns, a speedup of about 3.73 rather than the ideal 4. This example explains why pipeline depth is a throughput decision, not a promise of lower single-instruction latency.

A STAGE BOUNDARY CAN EXPOSE A HAZARD

A pipeline splits instruction memory and data memory logically but maps both to one single-ported physical memory. Fetch and load reach it in the same cycle.

What must the pipeline do to remain correct?

Reference decision: Arbitrate the structural conflict or stall one of the two accesses

One physical port cannot serve both accesses simultaneously. The design needs separate resources, multiporting, or a control rule that stalls and preserves both architectural operations.

Pipeline registers and control propagation

> **WARNING**
>
> Every instruction's destination register index `rd` and control signals (`RegWrite`, `MemRead`, `MemWrite`, `ResultSrc`) must advance through `ID/EX`, `EX/MEM`, and `MEM/WB` lockstep with its data values; driving the write-back stage's `rd` port directly from `IF/ID` overwrites a younger instruction's register with an older instruction's result.

A pipeline register is a typed bundle. In IF/ID, it may hold the fetched instruction and `PC`. ID/EX holds operands, immediate, destination identifiers, and decoded controls. EX/MEM holds ALU result, store data, destination, and memory controls. MEM/WB holds loaded data or ALU result, destination, and register-write control. Every field must have a defined value when the associated instruction is valid.

Pipeline Register	Stage Input -> Output	Propagated Control Fields	Datapath & Identification Fields	Validity & Hazard Invariant
`IF/ID`	Fetch → Decode	None (not yet decoded)	`Instr[31:0]`, `PC`, `PC_next`	Synchronous flush clears `valid` to 0 on branch misprediction
`ID/EX`	Decode → Execute	`ALUOp[1:0]`, `ALUSrc`, `Branch`, `MemRead`, `MemWrite`, `RegWrite`, `ResultSrc`	`rs1_data`, `rs2_data`, `ImmExt`, `rs1`, `rs2`, `rd`, `PC`	Stall freezes contents; bubble injects zeroed write controls
`EX/MEM`	Execute → Memory	`MemRead`, `MemWrite`, `RegWrite`, `ResultSrc`	`ALU_result`, `WriteData`, `rd`, `BranchTarget`, `Zero`	Preserves `rd` of executing instruction for forwarding comparison
`MEM/WB`	Memory → Writeback	`RegWrite`, `ResultSrc`	`ReadData`, `ALU_result`, `rd`	Delivers `rd` and `RegWrite` to register file write port

Validity is itself state. A bubble is an invalid pipeline entry that performs no architectural side effect. Reset typically fills the pipeline with invalid entries; a flush converts younger entries to bubbles. Treating a bubble as an instruction with arbitrary opcode is unsafe if control decode can assert writes. A valid bit must gate register writes, memory writes, exceptions, and retirement.

Control signals can be decoded early and pipelined, or regenerated from a pipelined instruction field later. Early decode may reduce repeated logic but increases bundle width; late decode may simplify the bundle but extend a stage's critical path. The invariant is that the controls used in a stage describe the instruction whose data is in that stage, not the instruction currently being fetched.

Data hazards

A data hazard arises when overlapping instructions have a dependence. A read-after-write (RAW) hazard is a true dependence: I_1 must read a value produced by I_0. Write-after-read (WAR) and write-after-write (WAW) hazards are name dependences that appear in machines allowing reads, writes, or retirement out of program order. A simple in-order five-stage pipeline primarily faces RAW hazards.

Consider:

```
add  r1, r2, r3
sub  r4, r1, r5
```

The add produces `r1` at the end of EX or later, while the sub needs it at the beginning of its EX stage. If the register file is read in ID, the sub may read the old value. *Forwarding* or bypassing sends the produced value from a later pipeline register directly to the consuming ALU input, avoiding a register-file round trip. The forwarding unit compares source identifiers against destination identifiers of older instructions whose results are available.

Forwarding does not eliminate every stall. For:

```
lw   r1, 0(r2)
add  r3, r1, r4
```

the load's data becomes available only after MEM, but the dependent add needs it in EX in the immediately following cycle. A one-cycle bubble lets the load reach its data-producing point before forwarding to the add. The hazard detector must freeze the PC and IF/ID register, insert a bubble into ID/EX, and then allow progress. Freezing everything would deadlock; advancing everything would compute with stale data.

Worked trace: load-use hazard schedule. With a load-use pair, cycle 3 has `lw` in EX and `add` in ID. The detector sees `ID/EX.MemRead=1` and `ID/EX.rd` equal to the add's source. In cycle 4, keep the add in ID, keep the PC and fetched instruction stable, and place a no-op control bundle in EX. In cycle

5, the load is in WB or its value is available for forwarding, while the add enters EX. The bubble costs one cycle and preserves architectural semantics.

Structural hazards

EXAMPLE

When instruction fetch (`IF`) and a data load (`MEM`) contend for a single-ported memory in cycle 4, the memory arbiter grants `MEM` (the older instruction) so the pipeline drains forward while holding `PC` and injecting a bubble into `IF/ID`; prioritizing `IF` over `MEM` would livelock `MEM` whenever fetches arrive every cycle.

A structural hazard occurs when two stages need a resource simultaneously. A single-ported unified memory cannot fetch the next instruction while a load or store accesses data in the MEM stage. Solutions include separate instruction and data caches, multiported memory, scheduling that inserts a stall, or a pipeline organization that separates requests in time. Hardware duplication increases area and power but can increase throughput.

Structural hazards are not data hazards and should not be hidden under a generic "stall." The cause determines the proof: a data stall waits for a value; a structural stall waits for ownership of a resource. A valid/ready protocol makes the resource conflict explicit and prevents the controller from assuming a response that was never accepted.

Control hazards and speculation

DEFINITION

For branch frequency $f_{\mathrm{r}}^{\mathrm{b}}$, predictor misprediction rate $m_{\mathrm{br}} = 1 - a$, and flush penalty p cycles, control hazards add $\Delta\mathrm{CPI}_{\mathrm{br}} = f_{\mathrm{r}}^{\mathrm{b}} \cdot m_{\mathrm{br}} \cdot p$ cycles per instruction and require annulling the `valid` bits of all p younger wrong-path pipeline slots upon misprediction resolution.

A conditional branch creates a control hazard because the next PC depends on a comparison that may not be resolved until EX. Meanwhile, the fetch stage has already fetched sequential instructions. The simplest policy stalls until the branch resolves or predicts not taken and flushes wrong-path instructions when taken. If the branch resolves after r stages and there are q younger wrong-path instructions, the recovery cost includes those invalidated cycles and any front-end refill.

Cycle	IF	ID	EX	MEM	WB	Architectural Action
1	`beq r1, r2, target`	—	—	—	—	Fetch branch; predict not taken
2	`sub r4, r5, r6` (wrong path)	`beq`	—	—	—	Fetch fall-through instruction
3	`and r7, r8, r9` (wrong path)	`sub`	`beq` (resolves TAKEN)	—	—	Condition evaluates true; target computed
4	`target: add r10, r11, r12`	bubble (`valid=0`)	bubble (`valid=0`)	`beq`	—	Pipeline flush: clear `IF/ID` and `ID/EX`; redirect PC to target
5	`target+4: lw r13, 0(r10)`	`target: add`	bubble (`valid=0`)	bubble (`valid=0`)	`beq`	Resume fetch along correct path; flush penalty = 2 cycles

The flush must be precise. A wrong-path instruction may have computed a value internally, but it must not write a register, memory, status register, or device. Valid bits and side-effect gating enforce this. A branch target calculation and condition comparison can be moved earlier to reduce penalty, but doing so increases the earlier stage's critical path and may introduce new forwarding requirements.

Branch prediction guesses direction and sometimes target. A static predictor uses a fixed rule; a dynamic predictor records history. Prediction accuracy, branch frequency, and misprediction penalty determine benefit. If branches occur with frequency f, prediction accuracy is a, and penalty is p cycles, the average branch cost is approximately $f(1-a)p$ cycles per instruction, ignoring secondary effects. Increasing predictor complexity is worthwhile only when its area, power, and access latency do not erase the gain.

Exceptions in a pipeline

> **WARNING**
>
> Because a younger instruction in `IF` (e.g., an instruction-fetch page fault) can detect an exception on an earlier clock cycle than an older instruction in `MEM` (e.g., a data-access fault), the pipeline must record exceptions in each instruction's stage bundle and take them strictly in program order at the commit boundary.

An exception may be detected in one stage while younger instructions have advanced. Precise exception handling requires all older instructions to retire, the faulting instruction to be represented at the specified boundary, and all younger instructions to become bubbles. A page fault on a load must not allow a younger store to modify memory before the handler observes the faulting state.

Pipeline Stage	Potential Exception	Detection Cycle	Commitment Order	Required Pipeline Control Action
`IF`	Instruction page fault / Misaligned fetch	Cycle t	Younger	Record status in `IF/ID.ExcCode`; suppress trap until WB
`ID`	Illegal instruction / Reserved opcode	Cycle $t+1$	Middle	Record status in `ID/EX.ExcCode`; suppress subsequent decoding
`EX`	Integer overflow / Arithmetic trap	Cycle $t+2$	Middle	Record status in `EX/MEM.ExcCode`; block register destination write
`MEM`	Data access page fault / Misaligned load	Cycle $t+3$	Older	Overrides younger exceptions; triggers trap handler at WB stage
`WB`	Commit boundary	Cycle $t+4$	Earliest	If `ExcCode != 0`, annul younger stages, write `sepc` / `scause`, redirect PC

Pipeline control therefore has three distinct actions: stall, which holds an instruction in place; bubble, which inserts invalid work into a stage; and flush, which discards younger work. Confusing them produces subtle bugs. On a load-use hazard, older stages continue, the dependent instruction is held, and one bubble is inserted. On a taken branch, younger instructions are flushed and a target fetch begins. On an exception, the precise recovery boundary is determined by the ISA and retirement design.

Performance model

> **REMARK**
>
> Pipelining reduces clock period T at the cost of hazard stalls and branch-flush penalties, so wall-clock time per instruction $\mathrm{TPI} = \mathrm{CPI} \times T = (1 + \mathrm{CPI}_{\mathrm{stall}} + \mathrm{CPI}_{\mathrm{flush}}) \times T$ must be evaluated as a product rather than optimizing T or CPI in isolation.

For a pipeline with clock period T, ideal time for N instructions is $(k + N - 1)T$. Add stall cycles S and flush/recovery cycles B:

$$T_{total} = (k + N - 1 + S + B)T.$$

The effective CPI for large N is approximately $1 + (S + B)/N$, plus any base issue limitation. A deeper pipeline can reduce T while increasing branch penalty, latch overhead, and hazard complexity. A useful comparison must include both terms.

Worked comparison. Design A has $T = 1$ ns, 10 stages, and 0.08 branch-mispredictions per instruction with penalty 8 cycles. Design B has $T = 1.3$ ns, 5 stages, and the same branch frequency with penalty 3 cycles. Ignoring other stalls, A has CPI $1 + 0.64 = 1.64$ and time per instruction 1.64 ns. B has CPI $1 + 0.24 = 1.24$ and time 1.612 ns. The deeper clock is not automatically better; workload-dependent control penalties nearly erase its frequency advantage.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.4.1: Pipeline Latency and Throughput Formula

Verification application

Schedule five independent instructions in a five-stage pipeline and derive the $k + N - 1$ formula.

Trace how each instruction progresses stage-by-stage from IF to WB to derive total cycles for N instructions.

Instruction I_j enters IF in cycle j, reaches WB in cycle $j + 4$, and the last of N instructions therefore completes in cycle $N + 4$. In general, the first instruction occupies all k stages and each additional instruction adds one cycle, giving $k + N - 1$.

Exercise 2.4.2: ALU vs Load Forwarding Path Comparison

Description application

For an add followed by a dependent add, identify the forwarding path and the cycle in which the value is available. Repeat for a load followed by a dependent add.

Compare when ALU results are available in EX/MEM versus load results returning after MEM stage.

An ALU result can usually forward from EX/MEM to the next instruction's EX input. A load result is not available until after MEM, so an immediately dependent instruction needs one bubble in a basic five-stage design and then receives the value from MEM/WB.

Exercise 2.4.3: Stall Control Logic for Load-Use Hazards

Construction application

Design the stall control signals for a load-use hazard. State exactly which registers hold, which receive a bubble, and which instruction proceeds.

Specify which pipeline registers must hold state and which must be cleared to insert a bubble.

Hold PC and IF/ID, force ID/EX controls to a no-op, and allow the load in EX to advance. On the next cycle release the hold. Holding ID/EX as well would repeat the load; advancing ID without a bubble would use stale data.

Exercise 2.4.4: Structural Conflicts in Unified Memory

Description transfer

A unified single-ported memory serves IF and MEM. Give a trace containing a load and explain the structural stall. Propose two hardware alternatives.

Trace cycle collisions when instruction fetch and data memory accesses contend for a single memory port.

When a load occupies MEM, IF cannot access the same port in that cycle, so fetch stalls and a bubble may appear in the front end. Separate instruction/data memories or a multiported/cache-banked design remove the conflict at hardware cost.

Exercise 2.4.5: Average CPI Calculation with Branch and Load Stalls

Verification transfer

Calculate average CPI for branch frequency 20%, accuracy 92%, and penalty 5 cycles, then add a one-cycle load-use stall for 10% of instructions.

Weight branch penalties and load-use delays by their instruction mix frequencies to find effective CPI.

Branch penalty is $0.20 \times 0.08 \times 5 = 0.08$ CPI. Load-use stalls add 0.10 CPI, so the estimate is 1.18 CPI. The terms assume independent frequencies and no interaction; real traces may correlate hazards.

Exercise 2.4.6: Flushed Store Cancellation Invariants

Formalisation synthesis

Explain why a flushed store must be prevented from reaching a memory write even if its address and data were already computed.

Ensure instructions squashed due to branch mispredictions or traps never assert memory write enables.

Computed internal values are not architectural effects. If the store writes after the branch is found taken or an exception is detected, a wrong-path or younger instruction changes memory and violates precise execution. The valid bit must gate the memory-write enable.

Exercise 2.4.7: Deep vs Shallow Pipeline Performance Comparison

Generalisation synthesis

Compare a 20-stage and 5-stage design under a workload with branch frequency 15%, misprediction accuracy 90%, and respective penalties 12 and 3 cycles. Include clock periods of 0.7 ns and 1.2 ns.

Evaluate whether higher clock frequency in a 20-stage pipeline outweighs larger misprediction penalties.

The 20-stage design has branch cost $0.15 \times 0.10 \times 12 = 0.18$, CPI 1.18, and 0.826 ns per instruction. The 5-stage design has cost 0.045, CPI 1.045, and 1.254 ns per instruction. Under these assumptions the deeper design wins despite its larger misprediction penalty, but the margin is workload-sensitive.

References & Further Reading

- MIT 6.004 Computation Structures: lecture notes and examinations
- MIT 6.004 learning objectives
- UC Berkeley CS 61C: pipeline and memory-system notes

SECTION DETAIL

Hazards and Branches

Pipelining creates overlap, and overlap creates situations in which an instruction cannot safely use a resource or value at the moment its ideal schedule requests it. These situations are *hazards*. A data hazard concerns dependence between values, a structural hazard concerns competition for hardware, and a control hazard concerns uncertainty about which instruction should execute next. Hazard handling is not an optimization layered on correctness; it is the mechanism that preserves the sequential architectural contract while internal work is concurrent.

Dependence graphs and hazard names

DEFINITION

Given instructions I_i preceding I_j in program order ($i < j$) with read sets $R(I)$ and write sets $W(I)$: a **Read-After-Write (RAW)** flow dependence holds when $W(I_i) \cap R(I_j) \neq \emptyset$; a **Write-After-Read (WAR)** anti-dependence holds when $R(I_i) \cap W(I_j) \neq \emptyset$; and a **Write-After-Write (WAW)** output dependence holds when $W(I_i) \cap W(I_j) \neq \emptyset$.

For instructions I_i and I_j with $i < j$, a true read-after-write (RAW) dependence exists when I_i writes a location that I_j reads. I_j must see the value produced by I_i. A write-after-read (WAR) anti-dependence occurs when an earlier instruction reads a location that a later instruction writes; the later write cannot occur before the earlier read. A write-after-write (WAW) output dependence occurs when both write the same location and program order determines the final value.

An in-order, single-issue five-stage pipeline reads registers in ID and writes in WB, so RAW hazards are the dominant register hazard. WAR and WAW do not arise when all reads happen before all writes in program order. An out-of-order or multi-issue machine can encounter all three and uses register renaming, issue constraints, and in-order retirement to preserve the architectural result.

The dependence graph is not the same as a hazard schedule. A dependence becomes a hazard only when the producer's availability and consumer's demand overlap unfavorably. Two instructions may be dependent but execute without a stall if forwarding provides the value in time. Conversely, a structural hazard can occur between independent instructions.

FIND THE DEPENDENCE

What creates a read-after-write data hazard?

Answer: A later instruction reads a value before its earlier producer makes it available

The dependency is semantic: the consumer reads a value whose producer has not yet completed. Forwarding or a stall must preserve the program's ordering.

Forwarding and interlocks

Consider:

```
add  r1, r2, r3
sub  r4, r1, r5
```

The `add` computes its result at the end of EX. In the next cycle, `sub` needs `r1` at the beginning of EX. A pipeline register such as EX/MEM already holds the result, so a forwarding mux can choose that value instead of the stale register-file read. The hazard unit compares the consuming source register with destination registers in older stages and selects the newest value that is known to be available.

The newest match wins. For:

```
add r1, r2, r3
add r1, r1, r4
add r5, r1, r6
```

the third instruction must receive the second instruction's result, not the first's. A forwarding network that checks only whether "some older instruction writes r1" can silently select stale data. The control priority must prefer the closest producing stage, subject to validity and register-zero conventions.

Forwarding is a form of dynamic bypass around the architectural register file. It does not change the ISA's register semantics; it changes where the implementation obtains a value. The proof obligation is observational: the consumer's ALU input must equal the value that a sequential execution would have read at that point.

FORWARD OR STALL?

In a five-stage pipeline, ADD produces r3 in EX. The following SUB needs r3 in its own EX stage before ADD reaches writeback.

Which evidence establishes that issuing SUB is safe?

Reference decision: A forwarding path delivers ADD's result to SUB before its EX use

A pipeline is a timed refinement of sequential semantics. The hazard rule is correct only when the consumer observes the producer's value and precise side effects remain ordered.

The load-use hazard

A load has a later availability point than an ALU instruction. In a basic five-stage pipeline, address calculation occurs in EX, memory returns data in MEM, and write-back occurs in WB. The instruction immediately after the load reaches EX before the data has returned:

```
lw   r1, 0(r2)
add  r3, r1, r4
```

Forwarding from EX/MEM cannot help because EX/MEM contains the load's address, not its loaded data. A one-cycle interlock is necessary unless the implementation has an earlier memory or special load-use path. The controller freezes the PC and IF/ID register, injects a bubble into ID/EX, and permits the load to advance. After the bubble, the dependent add reaches EX when the loaded value is available for forwarding from MEM/WB.

The stall is a temporal transformation of the schedule, not an alteration of the instruction stream. The load remains exactly once; the add remains exactly once; the bubble has no architectural identity. A common bug is to freeze the load in ID/EX as well as the dependent instruction, repeating the load indefinitely. Another is to freeze only the PC, allowing the fetched instruction to be overwritten or duplicated. A correct interlock identifies which pipeline registers represent the consumer and which represent older work that must advance.

Structural hazards and resource contracts

> **WARNING**
>
> When a structural stall in `MEM` coincides with a younger data hazard in `ID`, the pipeline controller must freeze all upstream stages (`IF`, `ID`, `EX`) rather than injecting a bubble that overwrites the stalled instruction in `EX/MEM`.

Suppose instruction fetch and data access share one single-ported memory. When a load is in MEM, the processor cannot simultaneously fetch the next instruction. The instructions are independent, but the resource is not. The controller must stall fetch, or the implementation must provide separate instruction/data paths, a second port, banking, or a cache hierarchy that permits both requests.

Contending Stages	Shared Resource	Arbitration Policy	Cycle Action for Stalled Stage	Cycle Action for Granted Stage
`MEM` (load/store) vs `IF` (fetch)	Single-ported memory port	Prioritize `MEM` (older instruction)	Freeze `PC` and `IF/ID`; inject bubble into `ID/EX`	Complete data access; advance `EX/MEM` to `MEM/WB`
`EX` (div) vs `EX` (alu)	Non-pipelined multicycle divider	Hold issue until divider ready	Assert `stall_EX`; freeze preceding pipeline stages	Continue iterative computation until `done=1`
Writeback stage (`WB`)	Register file write port	Single write-port contention	Stall multicycle functional unit completion	In-order pipeline commits standard ALU/load result

Resource conflicts should be specified as ownership contracts. A memory port may accept one request per cycle, or accept a request and return a response later. A multiplier may be pipelined and accept one operation per cycle, or be iterative and remain busy for several cycles. The hazard unit must use the actual valid/ready/busy protocol rather than assume every unit has one-cycle latency.

Structural and data hazards can interact. A load-use stall may coincide with a memory response stall. The controller needs a priority and a compositional invariant: older instructions retain their state, younger instructions do not pass an unavailable resource, and no request is duplicated unless the interface explicitly allows retries. The safe default is to make stalls hold all state that identifies an in-flight instruction while allowing independent older work to complete.

Control hazards

A conditional branch determines the next PC from a comparison. Until the condition and target are known, fetch does not know whether the sequential instruction is correct. In the single-issue five-stage pipeline considered here, a branch in EX can have at most two younger instructions in IF and ID. If the branch is taken, those instructions must be invalidated; if it is not taken, they may continue. Wider machines or extra fetch buffers can contain more younger work, but that is a different model.

The simplest policy stalls fetch until resolution. This avoids wrong-path work but costs cycles on every branch. Predict-not-taken fetches sequentially and flushes only taken branches. A static backward-taken/forward-not-taken rule exploits common loop behavior. A dynamic predictor records recent outcomes, perhaps with a branch-history table, two-bit saturating counters, a global history register, or a target buffer. The predictor is an implementation optimization, not part of the ordinary architectural state, unless the ISA exposes prediction hints.

A two-bit saturating counter has states strongly-not-taken, weakly-not-taken, weakly-taken, and strongly-taken. A taken outcome increments toward strongly taken; a not-taken outcome decrements toward strongly not taken. The prediction changes only after crossing the middle boundary, so one anomalous outcome does not immediately reverse a stable loop prediction. This reduces oscillation on branches that are usually taken with one exit.

Target prediction is separate from direction prediction. Knowing that a branch is likely taken is not enough if the target address is unavailable until decode or execute. A branch target buffer caches the target associated with a fetch PC; a return-address stack predicts procedure returns. Predictor structures trade storage and access latency for fewer front-end bubbles.

Worked trace — a two-bit counter on a loop

A loop branch is taken five times and then not taken on exit. The six outcomes below start from the strongly-not-taken state `00`. A taken outcome moves one step toward strongly taken; a not-taken outcome moves one step toward strongly not taken.

Outcome	Counter before	Prediction	Counter after	Correct?
T	00	not taken	01	no
T	01	not taken	10	no
T	10	taken	11	yes
T	11	taken	11	yes
T	11	taken	11	yes
N	11	taken	10	no

The first two taken outcomes are mispredicted while the counter learns; subsequent taken outcomes are predicted correctly, and the exit is mispredicted. If the loop is short, warm-up costs matter more. A predictor that reacts faster can learn sooner but may also flip after a single anomalous outcome.

A single counter indexed by PC ignores correlation between branches. A **global history register** records the recent outcomes of other branches and indexes a table of counters, so `if (a)` followed by `if (a && b)` can predict `b` using the outcome of `a`. A **TAGE** predictor combines several tables indexed by different history lengths, selects the longest matching entry when it has enough confidence, and falls back to a shorter history or a base predictor when it does not. Longer history captures correlated branches; shorter history limits damage from aliasing and cold entries. Predictor state is microarchitectural: it must be checkpointed or repaired on misprediction but never becomes part of the architectural result.

Use the trace to predict when the dependent `add` reaches MEM and WB. Then toggle the branch outcome: which instruction disappears, and which older instructions must be unaffected? The stage model assumes forwarding into EX and separate instruction/data memories, so a load-use stall is its only data stall.

TRACE A LOAD-USE AND BRANCH HAZARD

- **I0**: `lw r1,0(r2)`
- **I1**: `add r3,r1,r4`
- **I2**: `beq r3,r0,target`
- **I3**: `or r5,r6,r7`

T = first target instruction; · = invalid bubble; — = empty. Each row shows occupancy during the cycle; branch redirection takes effect at its end.

Cycle	IF	ID	EX	MEM	WB
1	I0	—	—	—	—
2	I1	I0	—	—	—
3	I2	I1	I0	—	—
4	I2	I1	·	I0	—
5	I3	I2	I1	·	I0
6	—	I3	I2	I1	·
7	T	·	·	I2	I1
8	—	T	·	·	I2
9	—	—	T	·	·
10	—	—	—	T	·
11	—	—	—	—	T

Static state: I0 writes back at cycle 5, I1 at cycle 7, and I2 reaches WB at cycle 8. The taken branch squashes I3 before EX; the target reaches WB at cycle 11.

The load occupies MEM at cycle 4 and WB at cycle 5. Its consumer is held in ID at cycle 4, reaches EX at cycle 5, MEM at cycle 6, and WB at cycle 7. The branch reaches EX at cycle 6 and resolves at that cycle's end. A taken branch removes the younger `or`, but must not remove or accelerate the older `add`. Advancing `add` directly from EX to WB is not a legal trace in this model.

Flushes, speculation, and side effects

WARNING

Squashing a wrong-path instruction by clearing its `RegWrite` and `MemWrite` control bits preserves architectural state, but if the wrong-path instruction already initiated a data-cache line fill before the branch resolved in `EX`, the resulting cache-state change remains observable to timing side-channel attacks (Spectre-class transient execution).

Speculation lets the processor execute predicted-path instructions before a branch is resolved. Speculative values can occupy internal registers and caches, but architectural side effects must wait until the instruction is known to be on the correct path and older exceptions have been handled. A wrong-path store cannot write memory. A wrong-path system call cannot enter the kernel. A wrong-path register value cannot retire.

Speculative State Layer	Action on Predicted Path	Action on Branch Resolution (Mispredict)	Architectural Visibility Barrier
Register file	Allocates rename physical register	Restores committed rename map; frees speculative register	In-order reorder buffer (ROB) commit
Data cache hierarchy	Line fills and tag updates occur	Lines remain allocated in cache (microarchitectural leakage)	Line invalidation / speculative fence
Store buffer	Buffers uncommitted store data	Squashes uncommitted store queue entries	Drain to cache only after instruction retires
Memory-mapped I/O	Speculative access prohibited	Trapped or rejected by memory management unit	Strict in-order commit before MMIO transaction

The security literature shows why "not architecturally visible" is not identical to "unobservable." Speculative execution can affect caches and other microarchitectural state, and a later measurement may infer information about a discarded path. Architectural hazard handling still requires flushing invalid instructions; security hardening may additionally constrain speculation around sensitive checks. The microarchitecture must document or mitigate such channels according to its threat model.

Renaming and out-of-order hazards

EXAMPLE

Worked trace: in the sequence `I1: r1 = r2 + r3; I2: r2 = r4 + r5; I3: r6 = r1 + r2`, renaming maps architectural registers to physical registers `I1: p11 = p2 + p3`, `I2: p12 = p4 + p5`, and `I3: p13 = p11 + p12`. This eliminates the WAR anti-dependence on `r2` between `I1` and `I2` (allowing `I2` to execute before `I1`), while preserving the true RAW dependencies of `I3` on `p11` and `p12`.

In a superscalar machine, the pipeline may issue instructions out of program order. WAR and WAW hazards are name dependences: two instructions use the same architectural register name even when no true data dependence exists. Register renaming maps each architectural destination to a fresh physical register. A later write no longer destroys a physical register that an earlier read still needs, and two independent writes no longer contend for one storage location.

The rename map changes on dispatch; a reorder buffer or retirement map commits the architectural mapping in program order. If an instruction faults or a branch mispredicts, speculative mappings are discarded and the older committed map is restored. Renaming removes false dependences but cannot remove RAW dependence: a consumer still waits for the physical register produced by its true predecessor.

Worked dependence analysis. For `r1=r2+r3; r2=r4+r5; r6=r1+r2`, the second instruction has a WAR relationship with the first because it overwrites `r2` after the first reads it; renaming gives the second result a new physical register and preserves the first's source. The third instruction has true RAW dependencies on both earlier results and cannot execute until both physical values are ready. It may issue after the second result even if unrelated instructions between them are still waiting.

Recovery and precise ordering

DEFINITION

Precise recovery requires that when instruction I_k raises a synchronous fault or resolves a mispredicted branch, every older instruction I_j $(j < k)$ completes and commits in program order, while every younger instruction I_m $(m > k)$ has its reorder-buffer entry, store-queue entry, and speculative rename mappings annulled before redirecting `PC`.

When a branch prediction fails, the processor identifies the branch's younger instructions, cancels them, restores the rename state or checkpoint associated with the branch, and redirects fetch to the correct target. When an exception occurs, all younger instructions are discarded while older instructions retire. The recovery mechanism must also restore predictor or speculative state sufficiently to prevent a stale instruction from committing.

Recovery Phase	Triggering Event	Hardware Mechanism	Invariant Enforced
Pipeline Slot Flush	Branch mispredict in `EX`	Clear `valid` bits in `IF/ID` and `ID/EX`	No wrong-path instruction modifies registers or memory
Rename Map Repair	Misprediction or trap	Revert speculative map table from checkpoint or walk ROB	Architectural register names resolve to committed state
Store Queue Drain	In-order retirement	Retire head of ROB; write data to coherent cache	Stores become globally visible strictly in program order
Target PC Redirection	Branch resolution in `EX`	Drive target address into `PC_next`; resume `IF`	Next fetched instruction belongs to correct program path

The cost of recovery includes detection latency, cancellation, rename-map restoration, front-end redirection, instruction-cache access, and pipeline refill. A deeper machine may have a shorter clock but a larger recovery window. This is why branch prediction accuracy is a first-order performance variable rather than a minor implementation detail.

Verification by adversarial traces

> **REMARK**
>
> A hazard unit is verified only when directed traces exercise simultaneous hazards—such as an `EX/MEM` and `MEM/WB` double-write to the same register (`add r1, r1, r2; add r1, r1, r3; add r4, r1, r5`), a load-use stall targeting `x0`, and a taken branch immediately following a stalled instruction.

Hazard verification should use traces designed to defeat the common case. Include chains of dependent ALU instructions, alternating destinations, load-use pairs, destination register zero, taken and not-taken branches, back-to-back stores, memory stalls, simultaneous hazards, and exceptions immediately after branches. For each trace, record stage occupancy, source values, forwarding selections, stall signals, valid bits, and committed state.

Assertions can state that a consumer never enters EX with an unavailable source, a memory request is issued only once per transaction, a flushed instruction never asserts a side-effect enable, and a retired instruction's operands equal the reference-model operands. A performance counter can

measure stalls but cannot establish correctness; a processor may be fast because it silently uses wrong values.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.5.1: Data Dependency Classification

Description application

Classify every dependence in `add r1,r2,r3; sub r4,r1,r5; xor r1,r6,r7; add r8,r1,r9` and identify forwarding sources.

Distinguish true RAW dependencies from name-based WAR and WAW anti-dependencies.

The first-to-second pair is RAW on r1; the second-to-third pair is a WAW on r1 if the third writes it, while the third also reads r1 from the second? Here `xor r1,r6,r7` overwrites the first result, so the fourth consumer must receive the third result. The implementation must prioritize the newest valid producer. A careful solution distinguishes the architectural names from the actual instruction order.

Exercise 2.5.2: Load-Use Hazard Stall and Forwarding

Construction application

Design the exact stall/forward behavior for a load followed by two consumers, one immediately adjacent and one separated by an independent instruction.

Determine the cycle in which memory data is returned and calculate required stall cycles for dependent ALUs.

The adjacent consumer requires one bubble in a basic five-stage machine; after the bubble, forwarding from the load's data-producing stage supplies it. The second consumer may use the first loaded value if one independent instruction provides enough separation, but the exact answer depends on the implementation's memory and forwarding timing; state that assumption instead of applying a universal rule.

Exercise 2.5.3: Structural Memory Conflict Trace

Description transfer

Give a structural hazard trace for a unified memory and compare separate caches, multiporting, and stalling.

Show the collision when instruction fetch and data memory access contend for a single unified port.

A load in MEM conflicts with instruction fetch in the same cycle. Separate instruction/data caches remove the conflict, multiporting duplicates access hardware and bandwidth, and stalling preserves correctness at lower throughput. The choice is architectural only if latency or ordering is exposed; otherwise it is a microarchitectural trade-off.

Exercise 2.5.4: Branch Misprediction Penalty Calculation

Verification transfer

Derive the expected branch penalty for a predictor with frequency 18%, accuracy 94%, and resolution penalty 6 cycles.

Multiply the branch instruction frequency by the misprediction rate and pipeline recovery cycle cost.

Misprediction frequency is $0.18(1 - 0.94) = 0.0108$ per instruction; multiplied by 6 cycles gives 0.0648 CPI, before other stalls.

Exercise 2.5.5: Two-Bit Saturating Predictor State Transitions

Construction transfer

Explain how two-bit prediction behaves on a loop with nine taken outcomes followed by one not-taken exit.

Trace state changes in the four-state machine (strongly/weakly taken/not-taken) across loop iterations.

Starting weakly not-taken (`01`), the first taken outcome is mispredicted and increments the counter to weakly taken (`10`). The second taken outcome is correctly predicted and moves it to strongly taken (`11`); the remaining taken outcomes leave it there. The not-taken exit is mispredicted and decrements the counter once, to `10` . Other initial states change the warm-up.

Exercise 2.5.6: Register Renaming for Hazard Elimination

Construction synthesis

Rename a sequence containing WAR and WAW hazards, then identify the remaining RAW dependencies.

Map architectural register names to distinct physical registers to eliminate false WAR and WAW hazards.

Give each architectural destination a fresh physical register. Reads retain the physical versions visible at their dispatch time; WAR and WAW disappear, while a later read of a value produced by an earlier instruction remains RAW and waits for readiness.

Exercise 2.5.7: Wrong-Path Instruction Squashing Invariant

Formalisation synthesis

State the invariants required to prove that a mispredicted branch cannot commit a store.

Ensure all pipeline registers populated on wrong-path execution are cleared before state commitment.

Every wrong-path instruction has a valid bit that becomes false on flush; memory write enable is gated by valid and retirement/commit permission; the store queue or memory system accepts only committed stores; recovery removes younger entries. Together these establish that a mispredicted store can be computed internally but never changes architectural memory.

References & Further Reading

- Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach* (6th ed.), Chapter 3 (“Instruction-Level Parallelism and Its Exploitation”). Morgan Kaufmann. (Hazards, forwarding, speculation, and branch prediction.)
- Smith, J. E. (1981). *A Study of Branch Prediction Strategies*. ISCA, 135–148. (Two-bit saturating counters and branch-history prediction.)
- Seznec, A., & Michaud, P. (2006). *A Case for (Partially) TAgged GEometric History Length Branch Prediction*. Journal of Instruction-Level Parallelism, 8, 1–23. (TAGE.)
- MIT 6.004 Computation Structures: lecture notes and examinations
- MIT 6.004 learning objectives
- UC Berkeley CS 61C: five-stage pipeline notes

Memory, storage, and I/O

SECTION DETAIL

Cache Hierarchies

The processor can execute many arithmetic operations in the time required for one main-memory access. A cache hierarchy reduces the effective cost by keeping recently or nearby used data in smaller, faster storage close to the core. The hierarchy is not a magical "fast memory"; it is a carefully specified prediction mechanism. A cache contains copies of lower-level data, decides whether a request hits, chooses where a block may reside, and defines how writes propagate. Its performance depends on an access trace, its correctness depends on coherence and permission rules, and its cost includes area, energy, refill bandwidth, and miss handling.

Locality and the two-level model

DEFINITION

For a cache level with hit time t_h, local miss rate m, and miss penalty t_{penalty}, the Average Memory Access Time is $\text{AMAT} = t_h + m \cdot t_{\text{penalty}}$. In a multi-level hierarchy, the local miss rate of level k is conditional on missing in level $k-1$, whereas the global miss rate is the fraction of all processor memory references that miss in level k ($m_{\text{global},2} = m_{L1} \cdot m_{L2}$).

Temporal locality means that an address recently used is likely to be used again. Spatial locality means that addresses near a recently used address are likely to be used. A cache exploits both by transferring a *block* or *line* containing several adjacent bytes, not just the requested word. The block size is a design compromise: larger lines capture more spatial locality but consume bandwidth and may evict useful data; smaller lines reduce overfetch but increase tag and refill overhead.

In a two-level model, let hit time be t_h, miss rate be m, and miss penalty be the additional time to obtain the block from the next level. The average memory access time is

$$AMAT = t_h + m \cdot t_{penalty}.$$

If the lower level's access time includes the first level's lookup, some texts instead write $AMAT = t_h + (1 - h) t_{lower}$; the variables must be defined before numbers are substituted. For a hierarchy with L1 and L2, one useful form is

$$AMAT = t_{L1} + m_{L1}\left(t_{L2} + m_{L2} t_{mem}\right).$$

The L2 miss rate is usually conditional on an L2 access, while a global miss rate is relative to all processor accesses. Confusing local and global rates is a frequent source of apparently impressive but incorrect performance calculations.

Worked AMAT example. Suppose L1 hit time is 1 cycle and its miss rate is 5%. L2 takes 8 cycles on a hit and has a local miss rate of 20%; DRAM costs 80 cycles after an L2 miss. Then $AMAT = 1 + 0.05(8 + 0.2 \cdot 80) = 2.2$ cycles. The L2 lookup contributes $0.05 \cdot 8 = 0.4$ cycles per processor access through the L1 miss path, while DRAM contributes $0.05 \cdot 0.2 \cdot 80 = 0.8$ cycles. Improving L1 hit time from 1 to 0.8 cycles saves 0.2; halving the DRAM penalty saves $0.05 \cdot 0.2 \cdot 40 = 0.4$ cycles under this workload. The bottleneck is identified by the product of frequency and cost, not by the largest raw latency alone.

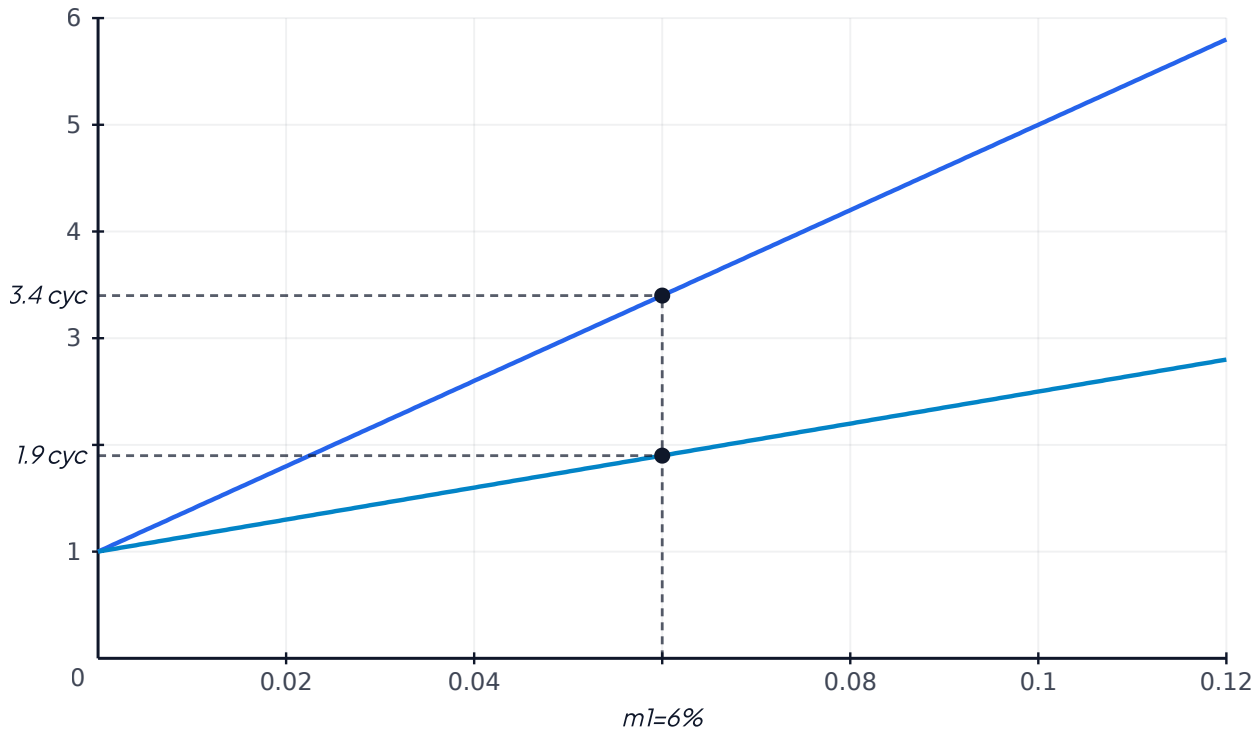


Figure 3.1.1: Average Memory Access Time (AMAT) vs. L1 miss rate m_1, contrasting a direct main-memory penalty ($t_{\text{miss}} = 40$) with an intermediate L2 cache ($t_{\text{miss}} = 15$).

MEMORY LOCALITY MATRIX

Evaluate each statement regarding spatial and temporal locality principles in hardware cache hierarchies:

Answer: Sequential iteration across elements of a contiguous array amortizes block refill costs through spatial locality.; Repeatedly accessing loop counters or inner accumulator variables within a hot loop exploits temporal locality.

Hardware cache lines transfer contiguous blocks to exploit spatial locality, while temporal locality relies on multiple references to the same addresses over short time intervals.

Address decomposition

For a cache with capacity C bytes, line size B bytes, and A ways, the number of sets is $S = C/(B \cdot A)$. Assuming powers of two, a byte address divides into block offset bits, set-index bits, and tag bits. The offset has $\log_2 B$ bits; the index has $\log_2 S$ bits; the remaining address bits form the tag.

Worked decomposition. A 32-bit byte address accesses a 16 KiB, 4-way cache with 64-byte lines. There are $16384/(64 \cdot 4) = 64$ sets. The line offset is $\log_2(64) = 6$ bits (`bits [5:0]`), the set index is $\log_2(64) = 6$ bits (`bits [11:6]`), and the tag is $32 - 6 - 6 = 20$ bits (`bits [31:12]`). For byte address `0x12345678`, the low 12 bits are `0x678 = 0b0110_0111_1000`, so the block offset is `0b111000 = 0x38` (56), the set index is `0b011001 = 0x19` (25), and the 20-bit tag is `0x12345`. Two addresses with set index 25 but different 20-bit tags compete for the four ways of set 25; they may conflict even if the other 63 sets are empty.

Alignment is part of this arithmetic. A word spanning two lines can require two tag lookups and two transfers, or the ISA may forbid the access. If a structure is aligned to a line boundary, its fields may still cross lines if its size and offsets are not chosen carefully. Performance analysis should use actual byte addresses and strides rather than assume every element maps independently.

INSTRUCTION COUNT HIDES LOCALITY

Two loops execute the same number of loads. One scans adjacent array elements; the other follows randomly scattered pointers through the same amount of data.

Which architectural metrics and evidence explain the performance divergence between these two loops?

Reference decision: Address traces, cache-line spatial reuse, and local miss rates across hierarchy levels.

Instruction count ignores memory hierarchy dynamics. Cache-line reuse, miss rates, memory-level parallelism, and DRAM row-buffer behavior govern actual execution latency.

Mapping organizations

A direct-mapped cache has one possible line per set. It is inexpensive and fast to index, but two frequently alternating blocks mapping to one set cause conflict misses. A fully associative cache allows a block anywhere and minimizes conflict misses, but comparing its tag against every line is expensive. A set-associative cache is a compromise: each block maps to one set but can occupy one of A ways.

Associativity does not eliminate capacity or compulsory misses. A compulsory miss occurs on the first access to a block. A capacity miss occurs because the working set exceeds the cache's usable capacity. A conflict miss occurs because the mapping forces competing blocks into too few locations. The same trace can exhibit all three at different cache sizes and associativities.

Worked conflict trace. A direct-mapped cache with four lines receives accesses to blocks 0, 4, 0, 4. Both block numbers map to set `block mod 4 = 0`. Every access misses, even though only two blocks are needed and the cache holds four. A 2–way cache with four total lines has two sets; blocks 0 and 4 map to set 0 but can occupy its two ways, so after warm-up the alternating trace hits. If a third block 8 is added, the two-way set again thrashes. Associativity is a targeted response to mapping conflicts, not a universal capacity increase.

The explorer holds the number of sets at two and varies ways per set, so total capacity also varies. Its two-block trace still diagnoses a mapping conflict: a one-way two-set cache has capacity for both blocks, but both compete for set 0. Predict the first hit, then add a third competitor and inspect the most-recently-used to least-recently-used order after each access.

MAKE ASSOCIATIVITY EXPLAIN THE TRACE

Block	Set	Result	Resident: MRU → LRU
B0	0	miss	B0
B4	0	miss	B4 → B0
B0	0	hit	B0 → B4
B4	0	hit	B4 → B0
B0	0	hit	B0 → B4
B4	0	hit	B4 → B0

Static state: blocks 0 and 4 alternate in a 2-way, two-set cache. Both map to set 0, but both fit, so only compulsory misses remain.

Replacement policy

EXAMPLE

In a 2-way set-associative cache where blocks A, B, and C all map to set 0, the cyclic access stream A, B, C, A, B, C experiences a 100% miss rate under Least Recently Used (LRU) replacement because the third distinct block always evicts the block that will be requested next, whereas a 3-way set or random replacement breaks the deterministic thrashing cycle.

When a set is full, the cache needs a replacement policy. Least recently used (LRU) evicts the line whose most recent access is farthest in the past; it exploits temporal locality but becomes expensive to track as associativity grows. Pseudo-LRU approximates LRU with fewer bits. Random replacement

is simple and can avoid adversarial patterns that defeat a deterministic policy. The policy affects performance, not the abstract contents that must eventually be returned, provided coherence and validity are correct.

Replacement metadata is itself state and must be reset or initialized consistently. A valid bit distinguishes an unused way from a line whose tag happens to match a reset value. A dirty bit records whether a write-back line differs from the lower-level copy. A coherence state may add shared, exclusive, modified, or invalid information in a multiprocessor.

Write policies

> **WARNING**
>
> In a write-back cache with a store buffer, a younger load that misses in the cache or reads a line being evicted must snoop or drain pending store-buffer entries for the same physical address; otherwise the load can return stale lower-level bytes while the most recent write sits uncommitted in the store buffer.

A write-through cache sends every write to the lower level, simplifying consistency but increasing traffic. A write-back cache updates only the cache line and writes it back on eviction if dirty, reducing bandwidth but requiring dirty tracking and careful eviction ordering. A write-allocate cache brings a missing line into the cache before applying a write; no-write-allocate sends a write miss directly downward. Common pairings are write-back/write-allocate and write-through/no-write-allocate, but the choice is workload- and coherence-dependent.

Write buffers decouple a processor from lower-level write latency. They introduce ordering obligations: a later load cannot observe stale data if an older buffered store targets the same address, and a memory barrier must drain or order the buffer as the ISA requires. Memory-mapped I/O often uses uncached or specially ordered regions because caching a device register can turn one device operation into repeated reads of an old copy.

Worked write example. A write-back cache has a dirty line for address A. A miss to another block mapping to the same set selects that way for replacement. The controller must first write the dirty line to the lower level, wait for acceptance or record a safe buffered transaction, install the new tag/data, and then satisfy the new request. If it overwrites the line first, the old value is lost. A write-through cache avoids this particular dirty eviction but can still require ordering if multiple writes are outstanding.

Miss handling and nonblocking caches

DEFINITION

A Miss-Status Holding Register (MSHR) tracks an in-flight cache miss by recording the missing physical block address, the target cache set/way, and the list of coalesced processor loads or stores waiting for that refill, enabling a lockup-free cache to overlap multiple independent misses (Memory-Level Parallelism).

A blocking cache stalls the requesting pipeline until a miss is complete. A nonblocking cache can continue serving hits and track outstanding misses with miss-status holding registers (MSHRs). It must know which requests depend on which refill, merge multiple accesses to one missing line, and preserve ordering for stores and fences. Multiple outstanding misses improve memory-level parallelism but increase control-state complexity.

MSHR Field	Data Width / Format	Hardware Purpose	Transition on Memory Refill
`Valid`	1 bit	Indicates active in-flight cache line miss request	Cleared to 0 when line arrives and targets are served
`Block Address`	Tag + Set index bits	Matches secondary misses to coalesce requests to same block	Compared against incoming pipeline loads/stores
`Way Allocation`	$\log_2(\text{Ways})$ bits	Pre-selected replacement way identified by cache controller	Receives refilled cache line data from interconnect
`Merged Load Queue`	Entry bitmask / register list	Identifies destination registers awaiting missing words	Delivers words directly to execution units / register file
`Dirty / Write Mask`	Line size bitmask	Tracks byte offsets modified by coalesced stores before refill	Merged with incoming memory line before writing back

The refill path may receive critical words first, allowing the processor to resume before the entire line arrives. This is useful when the requested word is available early but creates hazards if a later word is consumed before refill completion. The cache's valid/ready protocol must make partial data visible only under a defined rule.

Arrays, loops, and locality

EXAMPLE

For a row-major matrix `int32_t A[1024][1024]` with 64-byte cache lines, row-stride inner loop `A[i][j]` (incrementing `j`) touches 16 consecutive 4-byte integers per cache line ($1/16 = 6.25\%$ miss rate), whereas column-stride inner loop `A[i][j]` (incrementing `i`) jumps by $1024 \times 4 = 4096$ bytes per access —both wasting 15 of the 16 fetched words per line and mapping every access in column `j` to the exact same cache set in any cache whose set-indexing span divides 4096.

Row-major traversal of a matrix stored with rows contiguous accesses neighboring elements and uses each fetched line efficiently. Column-major traversal jumps by the row stride; if the stride exceeds line size and the working set exceeds the cache, each access may fetch a line whose other words are not used. Loop interchange can transform the trace without changing the mathematical result.

Worked stride calculation. With 64-byte lines and 4-byte integers, a sequential loop uses 16 integers per line and approaches one miss per 16 accesses after warm-up. A stride-16 loop touches one integer per line; it gets little spatial benefit and can cause one miss per access if the active lines do not fit. A stride of 1024 bytes maps to the same set periodically in a power-of-two cache, potentially causing conflict thrashing. Cache behavior is a property of line size, capacity, associativity, and address mapping together.

Data layout also controls locality. An array of structures is convenient when every field of one record is used together. A structure of arrays is better when a loop scans one field across many records, because each line contains more useful values and vector loads can be contiguous. Padding can prevent false sharing between cores, but increases footprint and may alter index mapping.

Coherence and consistency

DEFINITION

Cache coherence guarantees single-location serialization (writes to the *same* memory location are observed in a single order by all cores, and a read returns the latest write to that location), whereas a **memory consistency model** specifies the legal visibility ordering of loads and stores across *different* memory locations.

In a shared-memory multiprocessor, each core may cache a copy of a line. Cache coherence answers whether writes to one location eventually become visible consistently and whether a reader can obtain the latest value under the protocol's rules. A protocol such as MESI tracks whether a line is modified, exclusive, shared, or invalid and sends invalidations or updates over an interconnect.

Current MESI State	Local Core Read	Local Core Write	Bus Snoop Read (Peer Read)	Bus Snoop Invalidate (Peer Write)
Invalid (I)	Miss: issue BusRd; $\rightarrow$ $\mathbf{S}$ or $\mathbf{E}$	Miss: issue BusRdX; $\rightarrow$ $\mathbf{M}$	No-op; ignore transaction	No-op; ignore transaction
Shared (S)	Hit; no bus action; remain $\mathbf{S}$	Hit: issue BusUpgrade; $\rightarrow$ $\mathbf{M}$	No-op; peer receives shared copy	Invalidate local tag; $\rightarrow$ $\mathbf{I}$
Exclusive (E)	Hit; no bus action; remain $\mathbf{E}$	Hit (silent upgrade); $\rightarrow$ $\mathbf{M}$	Respond Shared; transition to $\mathbf{S}$	Invalidate local tag; $\rightarrow$ $\mathbf{I}$
Modified (M)	Hit; no bus action; remain $\mathbf{M}$	Hit; write to line; remain $\mathbf{M}$	Intercept; flush data to bus; $\rightarrow$ $\mathbf{S}$	Intercept; write back dirty data; $\rightarrow$ $\mathbf{I}$

Coherence is per cache line, not per language object or field. Two independent variables in one line can cause false sharing: cores repeatedly invalidate each other's copies even though they never access the same variable. Coherence also does not define all cross-address ordering; the ISA memory model and synchronization operations determine whether a flag becoming visible implies that preceding data is visible.

Atomic read-modify-write and fences interact with caches. An atomic operation may acquire exclusive ownership of a line, perform the update without an intervening conflicting write, and impose ordering. A fence may prevent a compiler or processor from reordering specified memory operations, but it does not magically flush every cache on every architecture. The precise semantics belong to the ISA.

Security and correctness boundaries

> **WARNING**
>
> Microarchitectural cache state (which sets are occupied and how fast a load completes) is observable through timing even when architectural access checks reject an unauthorized read; furthermore, in a Virtually Indexed, Physically Tagged (VIPT) cache whose index field extends into the virtual page number ($\text{sets} \times \text{line_size} > \text{page_size}$), two virtual aliases for the same physical frame can map to different sets (virtual synonyms) unless hardware or OS page coloring prevents stale duplicate lines.

Caches can create timing side channels because hits and misses have different latency. Isolation mechanisms may partition cache ways, flush state on context switch, use constant-access patterns, or accept a residual risk. A correct cache from the functional viewpoint can still leak access patterns. Security analysis must distinguish architectural data returned by loads from microarchitectural observations such as timing and replacement state.

Attack / Microarchitectural Hazard	Concrete Mechanism	Target Resource	Hardware / OS Countermeasure
Flush+Reload	Attacker flushes shared line via `clflush`, times victim access	Inclusive shared LLC lines	Way partitioning (CAT), unshared address mappings
Prime+Probe	Attacker fills target set, detects victim eviction via reload latency	Cache set conflict replacement	Random cache indexing, partitioned set allocation
VIPT Virtual Synonyms	Two virtual aliases map to different sets ($b + s > p$)	L1 virtually-indexed sets	Hardware alias detection snoops, OS page coloring
Transient Speculative Fill	Squashed speculative loads fetch secrets into data cache	L1/L2 data cache tags	Speculative load hardening, barrier fences (`lfence`)

The cache must also respect virtual memory permissions. Every cached access must still pass TLB permission checks for the current address space, and in virtually indexed caches where index bits extend above the page offset, synonym handling must prevent two virtual mappings of one physical

frame from holding inconsistent dirty lines in different sets. The order of translation and cache lookup—physically indexed, virtually indexed, synonym handling—belongs to the implementation and must be proven equivalent to the architectural memory model.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.1.1: Address Bitfield Decomposition

Construction application

Derive tag, index, and offset bits for a 64-bit address, 32 KiB, 8-way cache with 64-byte lines.

Compute offset bits from line size, index bits from set count, and assign remaining bits to the tag.

A 64-byte line needs 6 offset bits. There are $32768/(64 \cdot 8) = 64$ sets, requiring 6 index bits. The remaining $64 - 6 - 6 = 52$ bits are tag bits.

Exercise 3.1.2: Multilevel AMAT Calculation

Verification application

A hierarchy has $t_1 = 1$ cycle, $t_2 = 5$ cycles, $t_3 = 30$ cycles, and main-memory penalty $t_m = 100$ cycles. Its local miss rates are $m_1 = 0.08$, $m_2 = 0.25$, and $m_3 = 0.40$. Compute AMAT under the serial additive model, then report the global L2 and L3 miss rates and explain the difference between local and global L2 miss rate.

Apply the serial memory hierarchy equation where each level's miss penalty includes lower-level access times.

The serial model gives $1 + 0.08(5 + 0.25(30 + 0.40(100))) = 1 + 0.08(5 + 17.5) = 2.8$ cycles. The global L2 miss rate is $0.08 \cdot 0.25 = 0.02$ (2% of processor references), and the global L3 miss rate is $0.08 \cdot 0.25 \cdot 0.40 = 0.008$ (0.8%). The local L2 rate, 25%, is conditional on an L1 miss; treating it as 25% of all accesses would overstate lower-level traffic and cost.

Exercise 3.1.3: Direct-Mapped Conflict Miss Trace

Construct a trace that produces conflict misses in a direct-mapped cache but hits in a 2-way cache. Then add a third competing block.

Construction transfer

Identify memory addresses that map to the identical cache set index and trigger ping-pong evictions.

Blocks 0 and 4 alternate in a four-line direct-mapped cache and both map to set 0, so every access misses after each replacement. A 2-way cache can retain both; adding block 8 makes three blocks contend for two ways and can thrash under LRU.

Exercise 3.1.4: Write Allocation Policies Comparison

Description transfer

Compare write-through/no-write-allocate and write-back/write-allocate for a streaming store and a repeatedly updated small object.

Analyze bus traffic differences when writing streaming sequential data versus repeatedly updating hot variables.

A streaming store may waste a read-for-ownership under write-allocate and benefit from write-through or no-write-allocate; a hot object benefits from write-back because repeated writes stay local. The answer changes with reuse, bandwidth, durability, and coherence assumptions.

Exercise 3.1.5: Dirty Line Eviction Invariants

Formalisation transfer

Explain why a dirty line must be written back before replacement and what state a write buffer must preserve.

Consider data loss if memory is overwritten before a modified cache block is copied back.

The dirty data is the only current copy. Replacement must preserve its address tag and bytes until lower-level acceptance; a write buffer must also preserve order relative to later loads, stores, fences, and device accesses.

Exercise 3.1.6: Loop Interchange for Spatial Locality

Generalisation synthesis

Design a loop transformation that improves locality for a matrix operation. State the cache assumptions under which it helps.

Reorder nested loops so that memory accesses step contiguously through row-major arrays.

Interchange loops so the innermost dimension follows contiguous storage. For row-major data, iterate columns inside rows. The benefit assumes lines are larger than one element and the working set or tile fits enough cache to exploit reuse.

Exercise 3.1.7: Coherence vs Consistency vs False Sharing

Description synthesis

Distinguish coherence, consistency, and false sharing using one two-core example for each.

Contrast cache line ownership conflicts with memory ordering rules and independent variable packing.

Coherence: two cores write the same memory address `X` and must observe a single protocol-defined serialization order for `X`. Consistency: a producer writes `data = 42` and executes a release store `flag = 1`, while a consumer executes an acquire load of `flag == 1` before reading `data`, guaranteeing cross-address visibility. False sharing: two cores write distinct variables (`counter[0]` and `counter[1]`) that reside in the same 64–byte cache line, causing MESI invalidation ping-pong without any logical data sharing.

References & Further Reading

- MIT 6.004 Computation Structures: cache and memory-hierarchy notes
- UC Berkeley CS 61C: memory hierarchy and AMAT
- UC Berkeley CS 61C: cache terminology
- UC Berkeley CS 61C: AMAT

SECTION DETAIL

Virtual-Memory Hardware

Virtual memory is an address-translation and protection system that gives each process an address space with a private naming convention, even though physical memory is shared. The illusion is not "infinite RAM" by itself. It is a contract among the ISA, memory-management unit (MMU), operating system, page tables, translation caches, and fault handler. A load from virtual address v either returns the byte authorized by the current address space, or produces a defined fault without silently reading another process's data.

Pages and address decomposition

DEFINITION

For page size 2^p bytes, virtual-address translation decomposes $VA = (\text{VPN} \ll p) \mid \text{offset}$ where $\text{offset} = VA \,\&\, (2^p - 1)$ is invariant across translation and the MMU maps $\text{VPN} \mapsto (\text{PFN}, \text{perms})$ to form $PA = (\text{PFN} \ll p) \mid \text{offset}$ when perms authorize the access.

If the page size is 2^p bytes, a virtual address divides into a virtual page number (VPN) and a page offset. The low p bits are the offset and remain unchanged through translation; the upper bits identify the virtual page. A page-table entry maps the VPN to a physical frame number (PFN), so the physical address is

$$PA = (PFN \ll p) \mid (VA \,\&\, (2^p - 1)).$$

The page offset is not translated because every page and frame have the same size. This fixed-size mapping permits a process's virtual pages to occupy noncontiguous physical frames, removes external fragmentation from page allocation, and allows the OS to give different processes different mappings for the same virtual address.

Worked trace: virtual-to-physical translation. For a 32-bit address and 4 KiB pages, $p = 12$. Address `0x12345` has offset `0x345` and VPN `0x12`. If the page table maps VPN `0x12` to PFN `0x9AB`, the physical address is `0x9AB345`. Relocating the page changes the PFN and therefore the high physical address bits, but not the offset `0x345`. The same virtual address in another process may map to a different frame or be invalid.

Page size is a trade-off. Larger pages reduce page-table entries and TLB pressure and make sequential transfers more efficient, but increase internal fragmentation, fault granularity, and the amount of data transferred when only one byte is needed. Small pages improve allocation precision and reduce wasted transfer but require more translation metadata. Architectures may support multiple page sizes so applications can choose based on workload.

TRANSLATE SAFELY

What does a page-table translation add to a virtual address?

Answer: A physical-frame mapping plus permission and validity checks

Translation maps a virtual page to a frame while hardware checks present, readable, writable, executable, and other protection bits.

Page-table structures

A flat page table has one entry for every virtual page, which is wasteful for sparse address spaces. A multi-level page table divides the VPN into indexes. A top-level entry points to a lower-level table only when that region is used; unused portions need no allocated table. The physical address is formed from the final PTE's PFN and the original offset.

For a 48-bit virtual address and 4 KiB pages, there are 36 VPN bits. If each page-table page holds 512 eight-byte entries, each level consumes 9 VPN bits, so four levels cover 36 bits. A translation walk reads one entry per level, checks validity and permissions, and obtains the final PFN. The walk itself requires memory accesses and can be more expensive than the data load, motivating TLBs and page-walk caches.

Page-table entries commonly contain a present/valid bit, read/write and execute permissions, user/supervisor access, accessed/reference state, dirty state, caching attributes, and the PFN. The exact bit meanings are ISA-specific. A PTE with a valid bit clear may indicate an unmapped page, a swapped-out page encoded by the OS, or a reserved fault state. Hardware must not interpret OS-private encodings as ordinary physical addresses.

A TLB MISS IS NOT A PAGE FAULT

A load misses in the TLB. The page-table entry is present, readable, and maps to a resident physical frame.

Which sequence should the hardware or operating system perform?

Reference decision: Walk the page table, refill the translation, and retry the load

A TLB miss means the translation is absent from the cache, not that the page is absent from memory. A valid present PTE supplies the mapping; a page fault is needed only for absence or another exceptional condition.

TLBs and translation performance

DEFINITION

A Translation Lookaside Buffer (TLB) caches active $(\mathrm{ASID}, \mathrm{VPN}) \mapsto (\mathrm{PFN}, \mathrm{flags})$ entries; with TLB hit time t_{TLB}, miss rate m_{TLB}, and page-table walk penalty t_{walk}, translation adds $t_{\mathrm{TLB}} + m_{\mathrm{TLB}} t_{\mathrm{walk}}$ to unoverlapped memory accesses.

The translation lookaside buffer (TLB) caches recent VPN-to-PFN translations together with address-space identity and permission metadata. On a TLB hit, the MMU can form the physical address without walking the page table. On a TLB miss, hardware or software performs a page walk; if the PTE is valid, the translation is installed and the instruction retries or continues. A page fault is different: the translation is absent or disallowed and requires the operating system.

TLB reach is the number of bytes covered by cached translations. A TLB with 64 entries and 4 KiB pages covers 256 KiB if all entries are useful and nonoverlapping. A sequential scan over a larger region may have good cache locality but poor TLB locality if it revisits pages after the TLB has evicted their translations. Huge pages increase reach but can waste memory and complicate allocation.

Worked effective access time. Suppose a TLB hit costs 1 cycle, a TLB miss with a four-entry page walk costs 20 additional cycles, and the data cache hit costs 4 cycles after translation. With a 2% TLB miss rate and no page faults, the average cost is approximately $1 + 0.02 \cdot 20 + 4 = 5.4$ cycles under the stated additive model. If a page walk accesses the same cache hierarchy as data, the costs are not independent; a serious analysis must state whether the 20-cycle figure already includes cache effects.

Protection and address spaces

Virtual memory enforces isolation through permissions and address-space selection. A user process should not read kernel pages, execute data pages when non-executable policy forbids it, or write read-only code. A context switch changes the active page-table root or address-space identifier. TLB entries from the old process must be invalidated or tagged so a matching VPN cannot reuse an old process's translation.

Page Protection Bit	Evaluated Condition	Hardware Action on Violation	Use Case / Security Invariant
`U` (User)	`U = 0` and Current Privilege Mode is User	Trap: Page Fault (Permission Violation)	Prevents user-mode code from reading or writing kernel memory
`R` (Read)	Read access requested with `R = 0`	Trap: Load Page Fault	Prevents access to unmapped or execute-only memory regions
`W` (Write)	Store access requested with `W = 0`	Trap: Store Page Fault	Enforces read-only shared text and copy-on-write page protections
`X` (Execute)	Instruction fetch requested with `X = 0`	Trap: Instruction Page Fault	Enforces $W \oplus X$ (Data Execution Prevention) on data/stack pages
`ASID` Tag	TLB lookup with mismatched address space	Suppresses TLB match; forces page walk	Prevents cross-process translation reuse without flushing TLB

Shared memory is an intentional exception. Two processes can map the same physical frame at different virtual addresses, perhaps with read-only permission or controlled write access. Shared library text can be mapped read-only into many processes; copy-on-write pages can be shared until one process writes, at which point the OS allocates a private frame and updates that process's PTE.

The permission check belongs to the access, not merely the page-table lookup. An instruction fetch, data read, and data write may have different permission requirements. A TLB entry must cache all permissions relevant to future use, or the MMU must recheck them. Reusing a translation while ignoring a permission change is a security violation.

The explorer uses deliberately small 256-byte pages so the offset is the low eight bits. For process A, address `0x4a3` names VPN 4 and offset `0xa3`; its cached translation maps to frame 2, giving `0x2a3`. Predict what changes if the process or access permission changes. A cache hit cannot override permissions, and equal virtual addresses in different address spaces need not have equal translations.

TRANSLATE A VIRTUAL ADDRESS—OR TRIGGER A FAULT



TLB hit. The ASID-tagged translation is cached; permissions are still checked.

Static state: process A reads 0×4A3. VPN 4 is cached and maps to frame 2, preserving offset 0×A3, so the physical address is 0×2A3.

Page faults and demand paging

WARNING

Because a page-fault exception restarts the faulting instruction after the OS repairs the mapping, the processor pipeline must squash the faulting load/store and all younger instructions before any architectural register or memory state is modified.

A page fault is a controlled exception caused by an absent or prohibited translation. The hardware records the faulting virtual address, access type, instruction context, and cause. The operating system examines the PTE and process policy. It may allocate a zero-filled page, load a file-backed

page, fetch a swapped-out page, perform copy-on-write, or deliver a segmentation/access error to the process.

Sequence Step	Active Agent	Action Taken	State Transformation
1	Hardware MMU	Detects invalid PTE (`V = 0`) or permission violation	Captures faulting address in `stval` / `CR2`, cause in `scause`
2	Processor Pipeline	Squashes faulting instruction and younger pipeline slots	Restores architectural registers to pre-instruction checkpoint
3	Trap Dispatch	Switches to supervisor mode; jumps to kernel `trap_vector`	Hardware saves `sepc` (faulting instruction PC); disables interrupts
4	OS Page Fault Handler	Identifies cause: anonymous demand page, file-backed, or COW	Allocates physical frame; populates data from backing store
5	OS Memory Subsystem	Installs valid PTE (`V = 1` , permissions, PFN) in RAM	Executes `sfence.vma` / `invlpg` to invalidate stale TLB entries
6	Hardware Return	Handler executes `sret` / `iret`	Restores user privilege mode; restarts faulting instruction at `sepc`

The fault handler must be restartable. The faulting instruction generally has not committed its architectural effect; after the mapping is repaired, the instruction is retried. If the instruction performed a partially visible device operation before faulting, ordinary restart assumptions do not apply, which is one reason memory-mapped I/O and access permissions need precise architecture rules.

Demand paging creates an additional timing regime. A TLB miss may cost tens of cycles; a page fault may cost millions of cycles because storage or decompression is involved. A benchmark that averages both without reporting fault counts can mistake a small number of faults for random CPU noise. Working-set size, replacement policy, locality, and prefetch decisions determine whether paging improves capacity or causes thrashing.

Replacement and dirty state

EXAMPLE

When the hardware page walker or TLB sets the PTE Accessed (`A = 1`) bit on any read/write and the Dirty (`D = 1`) bit on any store, the OS page reclaimer can discard a clean file-backed frame (`D = 0`) immediately upon eviction, whereas a dirty frame (`D = 1`) must be written back to storage before its physical frame is reallocated.

When physical frames are scarce, the OS chooses a victim page. The accessed/reference bit approximates use; the dirty bit says whether a private or writable page differs from its backing store. A clean file-backed page can be discarded and re-read. A dirty anonymous or file-backed page must be written before reuse. The OS may use clock-like approximations rather than exact LRU because tracking every reference is expensive.

The page-replacement analogy to cache replacement is useful but incomplete. A cache line is normally replaced by hardware in a microsecond-scale critical path; a page is managed by software and may involve storage, process scheduling, and asynchronous write-back. A page fault also changes the process's mapping and protection state, not just a data copy.

Cache and virtual-memory interaction

LEMMA

A Virtually Indexed, Physically Tagged (VIPT) cache with line size 2^b bytes and 2^s sets is free of virtual-address synonyms in hardware if and only if $b + s \leq p$ (equivalently, the size of one cache way 2^{b+s} does not exceed the page size 2^p), because all set-index bits then come from the untranslated page offset.

A memory access can require both translation and cache lookup. In a physically indexed, physically tagged (PIPT) cache, translation occurs first, then the physical address selects the cache set and tag. This is simple for synonyms but adds translation latency. A virtually indexed, physically tagged cache can overlap some lookup with translation, but the page offset must contain enough bits for the index or aliases require special handling.

Architecture Class	Set Index Derivation	Tag Match Derivation	Translation Critical Path	Synonym Vulnerability
PIPT (Physically Indexed, Physically Tagged)	Physical address	Physical tag	Cache access waits for TLB translation	None (physical address is unambiguous)
VIVT (Virtually Indexed, Virtually Tagged)	Virtual address	Virtual tag	Translation bypassed on cache hit	High (requires ASID tags and synonym flushes)
VIPT (Virtually Indexed, Physically Tagged)	Virtual offset ($b + s \leq p$)	Physical tag	Index lookup overlaps concurrently with TLB walk	None if way size $\leq$ page size; aliases possible otherwise

The page offset and cache block offset are related but not identical. If page size is 4 KiB and line size is 64 bytes, the low 6 bits select a byte within a cache line while the low 12 bits select a byte within a page. If the cache has 64 sets, its 6–bit index also lies within the page offset; this permits the cache to begin indexing before translation finishes. If the index extends above the page offset, different virtual pages may map to the same cache set unpredictably and synonym management becomes harder.

Address-space identifiers (ASIDs) tag TLB entries with a process identity, reducing full TLB flushes on context switch. Cache lines are usually tagged by physical address in a PIPT design, so a shared physical frame has one coherent cache identity even when mapped at different virtual addresses. The exact implementation may use virtual tags, page coloring, or explicit invalidation; the architectural requirement is that permitted aliases observe a coherent memory value and prohibited aliases cannot bypass permissions.

Translation invalidation and concurrency

> **WARNING**
>
> Clearing a PTE's valid or writable bit in RAM does not revoke access on cores that still hold the old translation in their private TLBs: the OS must complete local TLB invalidation (`sfence.vma` / `invlpg`) and synchronous remote TLB shootdown IPIs before freeing or repurposing the physical frame.

Changing a page table is a concurrent operation. A CPU may have a stale TLB entry while the OS updates the PTE. An invalidation instruction or interprocessor shootdown ensures that other cores stop using the old translation before the frame is reused or permissions are weakened. The OS must

order page-table writes, TLB invalidation, and reuse of physical memory. Otherwise one process may access a frame after it has been reassigned to another.

Shootdown Phase	Initiating Core (Core 0)	Target Cores (Cores 1..N)	Synchronization Invariant
Phase 1: PTE Update	Clears PTE valid bit (`V = 0`) in RAM	Executing threads using stale TLB	PTE is invalid in memory; old mappings still cached
Phase 2: IPI Broadcast	Issues interprocessor interrupt (IPI) to cores running process	Receive IPI; enter kernel shootdown trap handler	Initiator spins waiting on atomic acknowledge counter
Phase 3: Local Invalidate	Executes local `sfence.vma va, asid`	Execute local `sfence.vma va, asid`; increment ack	All private TLBs flush matching virtual translation
Phase 4: Frame Reclaim	Observes all acks received; reclaims or reassigns physical frame	Return from trap handler to user mode	No CPU can access freed frame through stale mapping

Permission upgrades and downgrades require different reasoning. Before making a page writable, the OS must ensure the intended private copy exists. Before removing access, it must ensure no core can continue using a stale cached translation. A correct PTE in memory is not sufficient evidence that every processor has observed it.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.2.1: Multilevel Page Table Virtual Address Decomposition

Construction application

Decompose virtual address `0x123456789abc` for 4 KiB pages and four 9-bit page-table indexes, ordered from level 4 (top) to level 1 (leaf). The leaf PTE supplies PFN `0xABCDE`. Identify all fields and compute the physical address.

Extract VPN levels and page offset bits from a 48–bit or 64–bit virtual memory address.

Four 9–bit indexes consume 36 VPN bits and the 12–bit offset completes 48 bits. From the most-significant VPN bits downward, the indexes are $36, 209, 179, 393$ (level 4 through level 1), and the offset is `0xabc`. With leaf PFN `0xABCDE`, $PA = (0xABCDE \ll 12) | 0xABC = 0xABCDEABC$; the offset is preserved exactly. A different page-table walk ordering must be stated rather than silently swapping levels.

Exercise 3.2.2: TLB Reach Comparison Across Page Sizes

Verification application

Compare TLB reach for 64 entries with 4 KiB pages and 2 MiB pages. State the fragmentation trade-off.

Calculate total addressable memory covered by fixed TLB entries when using 4 KiB versus 2 MiB huge pages.

64 entries cover 256 KiB with 4 KiB pages and 128 MiB with 2 MiB pages. Huge pages improve reach and reduce walk pressure but can waste up to nearly a page per allocation and make fine-grained protection or sharing less flexible.

Exercise 3.2.3: Copy-On-Write Fault Control Sequence

Description transfer

Explain the complete control flow for a copy-on-write fault after a `fork`-like operation.

Mark shared page table entries read-only and allocate fresh physical frames on first write trap.

Mark shared pages read-only, let both processes point to the same frame, and fault on a write. The handler verifies the fault is a valid copy-on-write case, allocates a private frame, copies data, updates only the writer's PTE to writable, invalidates the stale translation, and retries the store. A failed allocation must not leave the page writable.

Exercise 3.2.4: TLB Hit Rate Impact on Effective Access Time

Verification transfer

Compute AMAT-like cost for a TLB with hit/miss probabilities and a data cache, stating whether costs overlap.

Compute average memory access time taking into account multi-level page walk penalties on TLB misses.

Use $t_{hit} + mTLBt_{walk} + t_{cache}$ only if the stated costs are additive and page walks do not evict data. If translation and cache lookup overlap, use a conditional timing model rather than adding both full latencies.

Exercise 3.2.5: Stale TLB Entries and Cross-Core Shootdown

Description transfer

Explain why stale TLB entries can become a security issue when a physical frame is reused.

Explain why revoking permissions in memory requires interprocessor interrupt shootdowns across all cores.

If process A's translation remains cached after the frame is assigned to process B, A may read or write B's physical memory through the old mapping. ASIDs, shootdowns, and ordered reuse prevent this stale authority.

Exercise 3.2.6: PIPT vs VIPT Cache Hierarchy Trade-offs

Description synthesis

Compare PIPT and virtually indexed/physically tagged caches in terms of latency, synonyms, and page-offset constraints.

Compare virtually indexed physically tagged caches with PIPT regarding translation latency and page coloring.

PIPT avoids virtual synonyms naturally but waits for translation before cache tag/index work. VIPT can overlap index lookup when index bits lie within the page offset, but aliases mapping one frame at different virtual addresses may require synonym handling or page coloring.

Exercise 3.2.7: Page Table Permission Security Invariants

Formalisation synthesis

Design page-table and fault-handler invariants that prevent a user process from writing a read-only shared code page.

Enforce user vs supervisor and write-xor-execute (W^X) bit configurations to prevent privilege escalation.

A user PTE for shared code has user-read and execute permission but no write permission; hardware checks the TLB-cached permission on every store. The kernel changes permissions only through validated transitions, invalidates translations after updates, and never reuses a frame until all stale translations are removed.

References & Further Reading

- MIT 6.004 Computation Structures: virtual memory and memory hierarchy notes
- Cornell CS 3410: Virtual Memory
- UC Berkeley CS 61C: virtual memory and cache interaction

SECTION DETAIL

Storage and Interconnects

The processor is one participant in a system that includes DRAM, persistent storage, network devices, accelerators, and controllers. An interconnect transports requests and responses among them; a storage hierarchy determines where data resides and what latency, bandwidth, ordering, and durability guarantees apply. Systems architects must distinguish capacity from availability, visibility from persistence, and link bandwidth from end-to-end throughput. A device can accept a request quickly while the data remains queued, buffered, or volatile.

Interconnect transactions

DEFINITION

A split-transaction interconnect decouples request transmission from response completion by tagging each request with a transaction identifier (`TID`) and flow-control credits, allowing multiple non-blocking reads and posted writes to be in flight simultaneously while responses return in arbitrary order.

An interconnect transaction generally contains an address or route, command, attributes, write data or a response identifier, and completion information. A memory read request may be split from a later response; a posted write may have no immediate response. Credits, ready/valid handshakes, or request IDs control flow and prevent a fast producer from overrunning a receiver.

Ordering is a protocol property. A system may guarantee that requests from one source preserve order, allow reads to pass writes, or permit responses to return out of order with IDs. The CPU's memory model, cache coherence protocol, device DMA rules, and interconnect ordering must compose. A memory fence that orders ordinary cacheable stores may not order a device transaction unless the address space and ISA explicitly assign that meaning.

Worked trace: descriptor and doorbell ordering. A CPU stores a descriptor, rings a device doorbell, and expects the device to read the descriptor. The descriptor write must become visible to the device before the doorbell write is observed. A release fence or device-memory ordering operation establishes this. If the doorbell passes the descriptor on the interconnect, the device may fetch uninitialized fields even though the CPU executed the stores in source order.

TRACE THE TRANSACTION

Why must an interconnect protocol define ownership and completion separately?

Answer: A request can be accepted before the operation's data or response is complete

Pipelined or buffered interconnects decouple request acceptance from completion. The protocol needs IDs, ordering, backpressure, and error behavior.

Buses, links, and topologies

A shared bus arbitrates access among masters and broadcasts or routes transactions over common wires. It is simple but its bandwidth and arbitration latency degrade as participants increase. Point-to-point links scale better by providing independent lanes, switches, virtual channels, and flow control, but require more routing and protocol state. A crossbar offers multiple simultaneous paths at area cost; a mesh or ring trades diameter and bisection bandwidth against wiring.

Topology determines contention and failure domains. A link with peak bandwidth R does not imply every pair can communicate at R if traffic shares a bottleneck. Bisection bandwidth bounds the total traffic crossing a partition of the network. Arbitration policy affects fairness, latency, and priority inversion. A high-priority control request can starve bulk traffic, while strict fairness can delay latency-sensitive interrupts.

LATENCY AND BANDWIDTH ANSWER DIFFERENT QUESTIONS

A link needs 80 ns before the first byte arrives and then transfers 16 GB/s. A workload issues one dependent 64-byte request at a time.

Which quantity dominates this workload?

Reference decision: Per-request latency dominates because dependencies prevent concurrent transfers

A dependent request stream cannot overlap enough work to approach peak bandwidth, so first-byte and protocol latency dominate. Independent batched requests could expose a different bottleneck.

DRAM behavior

EXAMPLE

Accessing an already open DRAM row in an active bank requires only the column-access strobe delay ($t_{\mathrm{CAS}} \approx 14\ \mathrm{ns}$), whereas accessing a different row in the same bank (a row-buffer conflict) must first precharge the bitlines and activate the new row ($t_{\mathrm{RP}} + t_{\mathrm{RCD}} + t_{\mathrm{CAS}} \approx 42\ \mathrm{ns}$), tripling access latency.

DRAM stores charge in cells that must be refreshed and accessed through rows and columns. A row-buffer hit can be faster than opening a different row; bank-level parallelism allows multiple outstanding operations when addresses distribute well. The memory controller schedules reads, writes, refreshes, and priorities while respecting timing constraints.

A sequential stream benefits from burst transfers and row locality. Random accesses expose activate/precharge and queueing costs. Two threads can interfere by targeting the same bank or row, so per-thread bandwidth is workload-dependent. The controller's policy may favor row hits for throughput or aging requests for fairness and tail latency.

Persistent storage

DEFINITION

End-to-end storage service time decomposes as $T_{\mathrm{I/O}} = T_{\mathrm{queue}} + T_{\mathrm{cmd}} + T_{\mathrm{media}} + T_{\mathrm{transfer}} + T_{\mathrm{completion}}$, where T_{media} is dominated by mechanical seek plus rotational delay on magnetic disks and by flash page read/program plus FTL garbage-collection stalls on SSDs.

Storage devices add controller queues, translation layers, flash erase blocks, wear leveling, garbage collection, and firmware scheduling. A solid-state drive may complete small random reads quickly relative to a disk, but latency still varies with queue depth, garbage collection, thermal state, and whether data is resident in a device cache. A disk's seek and rotation make sequential access much cheaper than random access.

Storage Architecture	4 KiB Random Read Latency	Sustained Sequential Bandwidth	Overwrite / Erase Granularity	Queue Concurrency Model
Mechanical HDD	$5 - 15\,\mathrm{ms}$ (seek + rotation)	$150 - 250\,\mathrm{MB/s}$	In-place 512B/4KiB sector rewrite	Native Command Queueing (NCQ, 32 commands)
SATA SSD (AHCI)	$50 - 100\,\mu\mathrm{s}$	$500 - 550\,\mathrm{MB/s}$	Flash page read/write (4–16 KiB); block erase (2–8 MiB)	Single queue (32 entries); host register bottleneck
NVMe SSD (PCIe Gen4)	$10 - 25\,\mu\mathrm{s}$	$5{,}000 - 7{,}500\,\mathrm{MB/s}$	Flash page write; background FTL garbage collection	Up to 64K submission/completion queue pairs (64K entries each)
Persistent Memory (CXL)	$100 - 300\,\mathrm{ns}$	$20 - 32\,\mathrm{GB/s}$	Byte/cache-line direct addressability (`clwb` + `sfence`)	Direct load/store instructions via memory bus

The cost of an I/O request can be decomposed as queueing, command setup, media access, transfer, and completion notification. At low queue depth, media latency dominates; at high queue depth, queueing and contention dominate. A benchmark must state queue depth and request size. Reporting only peak sequential bandwidth cannot predict small synchronous writes.

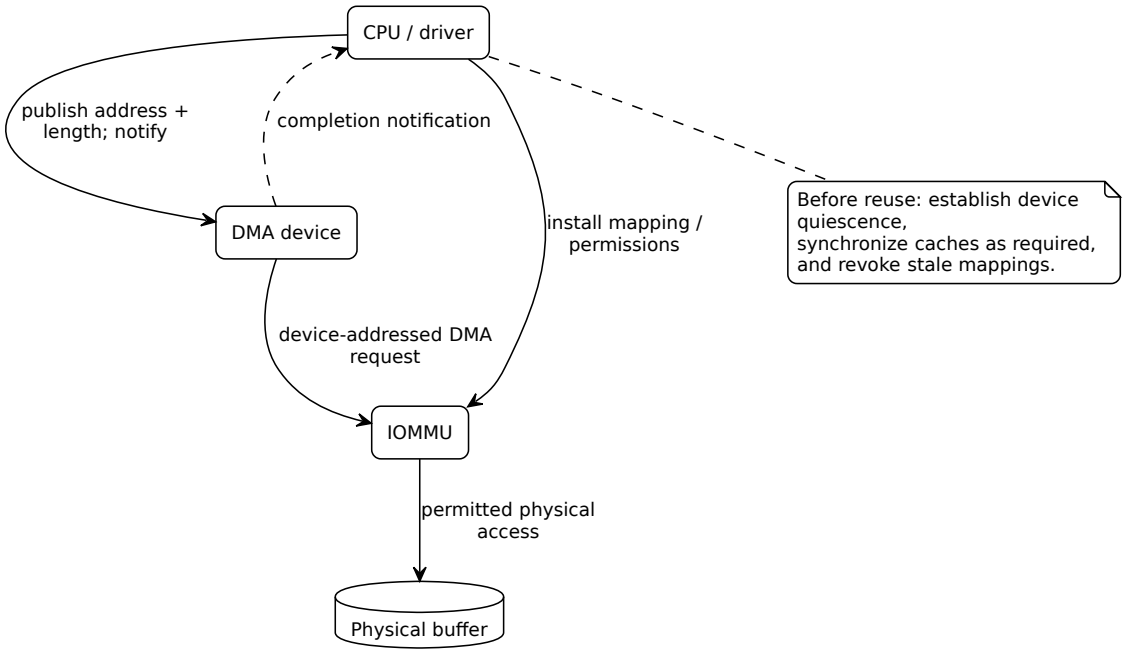
DMA and ownership

Direct memory access (DMA) allows a device to transfer a buffer without a CPU instruction for each word. The OS configures a descriptor containing buffer address, length, direction, permissions, and perhaps a completion policy. The device becomes owner of the buffer for the transfer; the CPU must not modify or reuse it until ownership returns.

Worked receive lifecycle. For a 1 MiB network receive, the OS allocates or maps a buffer, establishes a device-visible address, records length and permissions, and publishes a descriptor with a memory-ordering barrier. The device writes bytes, marks completion, and raises an interrupt or updates a completion queue. The driver acknowledges the completion, invalidates or synchronizes CPU cache state if the platform is noncoherent, validates the reported length and packet metadata, transfers ownership to the network stack, and posts a replacement buffer. A stale cache line can make the CPU see old bytes; a missing bounds check can turn a malformed device length into a memory-safety bug.

On an IOMMU-equipped system, device virtual addresses are translated and permission-checked, limiting a compromised device or driver from DMA-writing arbitrary physical memory. The IOMMU mapping has its own invalidation and lifetime protocol. Unmapping a buffer while the device still owns it creates a use-after-free at the hardware boundary.

This diagram separates control-plane setup from the DMA request path. The driver supplies a device-visible address and length after establishing the mapping. The device issues requests through the IOMMU; the IOMMU translates and checks them before memory is accessed. A completion notification alone does not prove that teardown has quiesced all outstanding requests.



Cache coherence and DMA

> **WARNING**
>
> On an interconnect where posted DMA writes and interrupt-message writes travel over separate virtual channels or paths, a completion interrupt can reach the CPU before the last DMA data cache lines become visible unless the device or bridge orders data visibility ahead of interrupt assertion.

A coherent DMA device participates in the cache-coherence domain or uses a coherent interconnect, so CPU and device copies follow a defined protocol. A noncoherent device requires explicit cache clean/invalidate operations. For a CPU-to-device buffer, dirty CPU cache lines must be written back before the device reads. For a device-to-CPU buffer, stale CPU cache lines must be invalidated before the CPU reads. The exact operation order depends on direction and platform.

DMA Transfer Direction	Platform Architecture	Required Pre-DMA Action	Required Post-DMA Action	Correctness Invariant
CPU → Device (Transmit)	Coherent Interconnect	Issue memory barrier (`sfence` / `dmb`)	Reclaim buffer on completion interrupt	Coherence engine snoops dirty L1/L2 lines automatically
CPU → Device (Transmit)	Noncoherent Interconnect	Clean / Write-back dirty cache lines to RAM	Reclaim buffer on completion interrupt	Device reads valid payload from DRAM; no dirty CPU eviction overwrites
Device → CPU (Receive)	Coherent Interconnect	Allocate and map buffer; publish descriptor	Read payload after acquire barrier	Snooping invalidates stale CPU cache lines on device DMA write
Device → CPU (Receive)	Noncoherent Interconnect	Invalidate buffer range in CPU cache	Invalidate cache again before reading payload	CPU cannot read stale cached data; uncommitted CPU writes cannot clobber DMA

A descriptor ring has two agents updating different fields: the producer publishes a descriptor, and the consumer observes it and later returns ownership. Use release/acquire ordering or the platform's DMA primitives. Do not infer safety from interrupt delivery alone; an interrupt can arrive before earlier writes become visible to the CPU if the architecture permits that ordering.

Interrupts, polling, and completion queues

EXAMPLE

At 2×10^6 packets/s, taking one interrupt per packet at $1.5\ \mu\text{s}$ per handler requires $3.0\ \text{CPU seconds/s}$ (saturating 3 cores in interrupt overhead alone); polling a completion ring in batches of 64 entries amortizes notification overhead across 64 descriptors.

Interrupts reduce CPU waste when events are infrequent, but each interrupt has entry, save/restore, scheduling, and cache costs. A high-rate device can cause an interrupt storm, so drivers batch completions, coalesce interrupts, or switch to polling under load. Polling reduces notification latency and can improve throughput at the expense of a busy core.

A completion queue decouples device progress from handler execution. The producer writes entries and advances a producer index; the consumer reads entries and advances a consumer index. Ring-buffer wraparound needs sequence numbers or an empty/full convention. The indices themselves are shared state and need atomicity and ordering. A consumer must not read an entry before the producer has published it; a producer must not overwrite an entry before ownership returns.

Reliability and error handling

DEFINITION

Single-Error Correcting, Double-Error Detecting (SECDED) Hamming codes add parity bits to each 64-bit DRAM word (typically 8 ECC bits for a 72-bit codeword of Hamming distance 4), correcting any 1-bit flip transparently and raising a synchronous machine-check exception on any 2-bit flip.

Interconnects and storage controllers detect corruption with parity, ECC, CRC, retries, or end-to-end checksums. A corrected error may be logged; an uncorrectable error may raise an exception, return a failed request, or mark a device block bad. Software must distinguish transient retry from permanent failure and must not report durable success before the relevant persistence guarantee holds.

Codeword Parity (P)	Syndrome Vector (S)	Error Classification	Hardware Action	Architectural Consequence
Even ($P = 0$)	$S = \mathbf{0}$	No error detected	Forward DRAM word to memory controller	Normal memory transaction completes
Odd ($P = 1$)	$S \neq \mathbf{0}$	Single-bit error (correctable)	Invert bit at index S; log syndrome in register	Corrected data delivered; correctable error counter increments
Even ($P = 0$)	$S \neq \mathbf{0}$	Double-bit error (uncorrectable)	Suppress data delivery; assert error flag	Machine check exception (MCE) raised; kernel panics or kills task
Odd ($P = 1$)	$S = \mathbf{0}$	Parity bit itself corrupted	Correct parity bit; log single-bit event	Correct payload delivered transparently to CPU

ECC memory can correct a one-bit error and detect certain multi-bit patterns depending on code. The architectural consequence may be transparent correction or a machine-check exception. Redundancy improves reliability but does not make silent corruption impossible; end-to-end checksums detect errors that occur outside the protected segment or in software.

Durability and ordering

> **WARNING**
>
> A posted write completion from a PCIe/interconnect controller or volatile drive DRAM cache means the request was accepted into a buffer, not that the bytes survived power loss; crash durability requires an explicit storage flush/FUA barrier down to non-volatile media.

A write reaching a controller's volatile cache is visible to the issuing software but may disappear on power loss. A flush, force-unit-access operation, capacitor-backed cache, or storage protocol may provide stronger persistence. Filesystem and database durability depend on the entire chain: CPU ordering, cache write-back, interconnect completion, device cache, flash translation layer, and power-failure behavior.

Worked durability sequence. To make a journal record durable before exposing a metadata update, software writes the record, issues the required data barrier/flush, waits for completion, writes the metadata, and flushes again if the protocol demands it. If the first flush is omitted, a power failure can leave metadata pointing to a record that never persisted. The correct sequence is a cross-layer contract, not merely two source-level writes in order.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.3.1: Ready/Valid Bus Backpressure Protocol

Construction application

Design a ready/valid transaction with backpressure and out-of-order responses identified by request ID.

Enforce that transaction payload data remains stable until both master and slave assert handshakes.

The sender holds `valid` and data until the receiver asserts `ready`; a transfer occurs only on both. Responses carry IDs so an out-of-order return can match its request. State whether ordering is restored before software observes completions.

Exercise 3.3.2: DMA Transmit Buffer Ownership Lifecycle

Description application

Trace a DMA transmit buffer from CPU preparation through device completion and reuse, including cache operations.

Trace cache flushing, ring descriptor posting, device DMA reading, and completion interrupt handling.

CPU writes data, orders it, transfers ownership, device reads/transmits, device returns completion, CPU observes completion and reclaims only after the completion is ordered and validated. Noncoherent systems need direction-specific cache clean/invalidate operations.

Exercise 3.3.3: Polling vs Interrupt Coalescing Trade-offs

Description transfer

Compare polling, interrupts, and interrupt coalescing for a low-rate sensor and a 100 Gb/s network interface.

Contrast immediate interrupt response with batched interrupt timers to balance latency and CPU overhead.

Polling suits high-rate traffic when a core can remain dedicated; interrupts suit sparse events; coalescing batches high-rate notifications and trades latency for lower overhead. The choice depends on arrival rate, latency target, and available CPU capacity.

Exercise 3.3.4: IOMMU DMA Translation and Access Isolation

Description transfer

Explain how an IOMMU limits DMA damage and why unmapping before completion is unsafe.

Show how an IOMMU prevents buggy or malicious devices from overwriting arbitrary host kernel memory.

The IOMMU maps only permitted device virtual addresses to frames and can revoke mappings. Revocation must wait for outstanding transactions or synchronize the device; otherwise a late DMA uses a freed or reassigned frame.

Exercise 3.3.5: Interconnect Bandwidth and Packet Latency Model

Verification transfer

Compute time to transfer a block from bandwidth and setup latency for small and large requests; identify the crossover.

Sum arbitration delay, serialization latency, and propagation time across on-chip interconnects.

For setup L and bandwidth R, transfer time is $L + n/R$. Small requests are setup-dominated; large requests approach n/R. Compare two systems by solving where their total times cross rather than using peak R alone.

Exercise 3.3.6: Write-Ahead Journaling and Crash Recovery

Construction synthesis

Design a durable two-record journal sequence and name every flush/order assumption.

Trace log record persistence and metadata flushes to ensure file systems recover to consistent states.

Persist the journal record and flush it before publishing metadata; persist metadata and flush before acknowledging the operation. The exact barriers depend on the storage protocol, but each acknowledgement must follow the stated durable point.

Exercise 3.3.7: Descriptor Ring Buffer Producer-Consumer Invariants

Formalisation synthesis

Give invariants for a descriptor ring with producer and consumer indices.

Define head and tail pointer wrap-around rules to prevent overwrite of unconsumed transfer requests.

A producer never overwrites an entry whose ownership remains with the consumer; a consumer never reads an unpublished entry; each index advances monotonically modulo ring size with an unambiguous full/empty rule; descriptor fields become visible before publication and completion becomes visible before reclamation.

References & Further Reading

- MIT 6.004 Computation Structures: devices, interrupts, and interconnect notes
- Cornell CS 3410: Computer System Organization and Programming
- RISC-V IOMMU specification

SECTION DETAIL

I/O and Devices

Input/output architecture connects a general-purpose processor to components whose timing, state, and failure behavior are not controlled by the instruction pipeline. A device may be a keyboard, disk, network interface, sensor, accelerator, timer, or virtual machine endpoint. It exposes a programming interface through registers, memory-mapped queues, interrupts, DMA, or a combination. A correct driver is a protocol implementation: it establishes ownership, orders commands and data, handles backpressure and errors, and does not assume that a device responds like ordinary memory.

Device register models

> **WARNING**
>
> Reading a memory-mapped status or FIFO register can clear pending interrupt bits or dequeue hardware data as a physical side effect; MMIO regions must be mapped uncached and accessed with explicit width and ordering rather than ordinary speculative memory loads.

Programmed I/O exposes control, status, data, and interrupt registers. A control register may start an operation; a status register may report ready, busy, error, or completion; a data register may consume a byte on read or clear an event on write. These side effects make repeated reads and read-modify-write operations semantically significant. A compiler optimization that repeats an ordinary memory read cannot be applied blindly to a device register.

Memory-mapped I/O assigns device registers addresses in the processor's address space. Port I/O uses a separate instruction space on some ISAs. In either model, the ISA defines ordering, width, alignment, privilege, and caching attributes. A 32-bit load from a register documented as 8-bit may trigger an invalid access or read adjacent state. The driver must use the device's access width and avoid speculative or prefetching accesses when the mapping forbids them.

Worked register sequence. A device has `STATUS.BUSY`, `CONTROL.START`, and a data FIFO. To start a transaction, the driver validates the buffer, writes the command and length, executes the required device-memory barrier, and writes `START`. It then either polls `BUSY` with a bounded timeout or waits

for an interrupt. On completion it reads status, checks error bits before consuming data, and acknowledges the interrupt according to the device specification. Writing `START` before the descriptor is visible can launch a transaction with stale parameters.

CROSS THE DEVICE BOUNDARY

Why does a memory-mapped device register need a volatile access rule in many C implementations?

Answer: The device can change or observe it outside ordinary program flow

The compiler must not remove or merge accesses whose effects are observable to hardware. Volatile does not by itself guarantee atomicity, ordering, or synchronization.

Polling and interrupts

Polling repeatedly reads status until a condition changes. It has predictable control flow and can achieve low latency when a dedicated core polls at high frequency, but it consumes cycles and power while idle. Interrupts let a device notify the processor, reducing idle work, but entry/exit costs, prioritization, cache disruption, and interrupt storms add overhead.

For an event arrival rate λ, polling period P gives average detection delay roughly $P/2$ under uniform arrivals, while consuming polling work every period. An interrupt has a fixed notification cost plus queuing delay and may be cheaper for low λ. At high rates, interrupt coalescing batches events; the system accepts more notification latency to reduce per-event overhead.

Worked choice. A 10 kHz sensor needs samples within 100 microseconds and has a small data payload. A dedicated polling loop at 20 kHz gives a nominal 25-microsecond average detection delay but consumes a core and must account for scheduling and cache effects. An interrupt per sample may meet latency at low system load but could overload the CPU if handler cost is high. A timer-driven batch or device-side FIFO may be better. A bursty disk, by contrast, benefits from interrupts or completion queues because long idle periods make continuous polling wasteful.

An interrupt handler should do the minimum urgent work: identify and acknowledge the source, capture status, and schedule deferred processing. If it performs unbounded parsing or blocking I/O, it can delay higher-priority interrupts and create priority inversion. Nested interrupts require a defined

masking and priority policy. A shared interrupt line requires reading each possible source and acknowledging only the one that is actually pending.

COMPLETION MUST PRECEDE CONSUMPTION

A driver starts DMA into a buffer and immediately reads the buffer before observing device completion or applying the platform's synchronization rule.

Which trace exposes the ordering defect?

Reference decision: Descriptor ownership, DMA writes, completion event, barrier, and CPU read

DMA creates another agent that can modify memory. Ownership, completion, and ordering rules determine when the CPU may safely consume the buffer; final bytes alone erase the race.

Queues and descriptors

DEFINITION

In a producer-consumer descriptor ring of capacity N indexed by unsigned monotonic counters prod and cons, the in-flight occupancy is $k = \mathrm{prod} - \mathrm{cons} \in [0, N]$; the driver must execute a release memory barrier between writing descriptor fields at slot $\mathrm{prod} \bmod N$ and publishing $\mathrm{prod} \leftarrow \mathrm{prod} + 1$.

High-throughput devices use rings rather than one register per item. A descriptor contains an address, length, operation, flags, ownership, and completion status. The producer publishes descriptors by writing fields first and advancing a producer index last. The consumer reads only published descriptors, performs the operation, writes completion fields, and advances a completion index.

The ring has a capacity invariant. If producer and consumer counters are monotonic, occupancy is `producer - consumer`; the producer may advance only when occupancy is below capacity, and the consumer may advance only when occupancy is positive. If counters wrap, use a sufficiently wide counter or sequence bits so full and empty are distinguishable. A single shared "valid" bit without ownership semantics can be observed in the wrong order.

Worked trace: descriptor ownership. A network transmit descriptor is `FREE`, then `READY`, then `IN_FLIGHT`, then `DONE`, then `FREE`. The CPU may write the buffer in `READY` preparation but must not modify it in `IN_FLIGHT`. The device may write completion status only in `IN_FLIGHT`. The driver checks `DONE`, validates length and error status, and returns ownership. A timeout does not prove the device stopped; reclaiming the buffer before reset or cancellation completes can cause a late DMA to corrupt a new owner.

DMA and IOMMUs

DMA transfers data between a device and memory. It avoids CPU copy loops but expands the set of agents that can access a buffer. The OS must establish a device-visible mapping, set direction and permissions, synchronize caches if needed, and revoke the mapping only after all transactions complete. A device writing past a programmed length is a hardware boundary violation; an IOMMU can restrict its address range and fault the transaction.

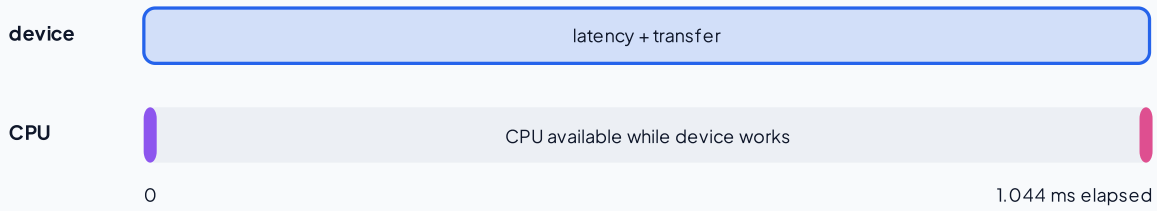
The IOMMU translates device virtual addresses to physical frames, often using page tables and permissions analogous to CPU virtual memory. It may cache translations, so invalidation and synchronization are required when mappings change. A driver must not unmap a buffer and immediately hand its frame to another process while the device can still issue an old translated request.

For a CPU-to-device buffer on a noncoherent platform, dirty CPU cache lines must be cleaned to memory before device ownership. For a device-to-CPU buffer, CPU cache lines that could contain stale data must be invalidated before reading completion data. On coherent systems, the platform's DMA API may combine or abstract these actions, but the ownership and ordering proof remains necessary.

The investigation below separates CPU occupancy from elapsed device time. For B bytes, its assumed device time is $D = L + 1000B/10^8$ milliseconds, where L is latency in milliseconds. A 3 GHz CPU executes 3×10^6 cycles per millisecond. Interrupt-driven CPU copying costs $K = 500 + 1200 + 2B$ CPU cycles: setup, interrupt, and copying. Thus busy time is $K/(3 \times 10^6)$ ms and elapsed time is $D + K/(3 \times 10^6)$ ms. These constants define a teaching model, not a device measurement.

At 4096 bytes and 1 ms latency, the device consumes 1.04096 ms; the CPU consumes $9892/(3 \times 10^6) \approx 0.003297$ ms. At 100 operations/s, the CPU demand is about 0.033%, despite more than 1 ms elapsed per operation. DMA replaces byte-proportional CPU copying with 1800 setup cycles plus 1200 interrupt cycles. Polling adds the entire device interval to CPU busy time. Predict which term dominates before changing transfer size or request rate.

CHOOSE POLLING, INTERRUPTS, OR DMA FROM WORKLOAD EVIDENCE



Static state: 4 KiB transfers, 1 ms device latency, 100 operations/s, interrupt-driven CPU copying. The CPU is free during device latency but pays setup, interrupt, and copy work.

Device state machines

EXAMPLE

When a driver transitions a device from `RUNNING` to `RESET` after a watchdog timeout, it must hold all `IN_FLIGHT` DMA buffers pinned until the hardware reset-complete status bit asserts and the IOMMU page-table entries are invalidated, transitioning to `QUIESCED` before any buffer returns to the page allocator.

A driver should model the device as a state machine rather than a collection of register writes. States may include `RESET`, `CONFIGURED`, `RUNNING`, `STOPPING`, `ERROR`, and `QUIESCED`. Events include reset completion, command acceptance, completion, timeout, hot-unplug, and fatal error. Each transition must specify which queues are valid, which interrupts are enabled, and which buffers the device may access.

Worked recovery sequence. If a network device stops responding, the driver disables new submissions, masks interrupts, records outstanding descriptors, requests a device reset, waits for reset completion with a bounded timeout, invalidates device mappings, reinitializes rings, and either

reconciles or fails outstanding operations. Reusing descriptors before the device is quiesced creates a race between old and new commands. Recovery is successful only when the state machine proves no old agent retains ownership.

Timeouts and failure interpretation

> **WARNING**
>
> A software timer expiration proves only that the CPU has not yet observed completion; it does not revoke the device's ability to perform DMA into the buffer or prevent a delayed interrupt from arriving for an earlier generation.

A timeout is evidence that an expected event did not occur by a deadline; it is not proof of device termination. The device may be slow, the interrupt may be lost, the request may be queued, or the completion may be visible but unread because of a cache/order bug. Recovery must be idempotent or use a device generation number so a late completion from an old operation cannot satisfy a new request.

Generation tags are useful when descriptors are reused. A completion includes the generation that was active when submitted; the driver ignores a completion whose generation does not match the current owner. This turns a temporal ambiguity into an explicit validation check. The tag does not stop a malicious device from writing arbitrary data, so permissions and bounds remain necessary.

Virtualization and emulated devices

> **REMARK**
>
> Paravirtualized ring interfaces (such as Virtio) amortize expensive VM-exit traps across batches of descriptors in shared guest-physical memory, while direct PCIe device assignment (VFIO) eliminates data-path exits only when paired with hardware IOMMU translation and interrupt remapping.

A virtual machine may see a virtual device whose registers are implemented by a hypervisor. Traps, shared-memory queues, or paravirtual interfaces carry requests between guest and host. The guest driver still needs a device contract, but performance depends on exit frequency, batching, and

shared-page synchronization. A virtual interrupt can be delayed or coalesced, so guest timeouts must account for scheduling.

Virtualization Strategy	Guest Access Interface	Hypervisor Interception Mechanism	I/O Datapath Overhead	Hardware Isolation Requirement
Trap-and-Emulate	Standard legacy MMIO registers	Stage-2 page table fault $\rightarrow$ VM-exit	High: each register read/write incurs $\approx 1{,}000 - 2{,}000$ cycle trap	Software boundary in hypervisor
Paravirtualization (Virtio)	Shared split-ring descriptor buffers	Batch doorbell notification via hypercall / PIO	Low: exits amortized across batches of N descriptors	Guest-physical to host-physical translation
Direct Device Assignment (VFIO)	Native physical PCIe registers & DMA	Direct hardware execution (no VM-exits)	Near zero: native hardware throughput and latency	Hardware IOMMU (two-stage) + Interrupt Remapping

Device assignment to a guest requires isolation. An assigned device may DMA into guest memory only through an IOMMU context; interrupt remapping prevents it from targeting arbitrary host vectors. Reset and teardown are especially important because a device can retain in-flight state across ownership changes.

Verification and testing

> **REMARK**
>
> Driver verification requires deterministic fault injection at descriptor publication, completion decoding, and timeout boundaries rather than relying solely on happy-path device emulation.

Test drivers with delayed responses, duplicate interrupts, out-of-order completions, malformed lengths, queue-full conditions, device reset during DMA, and concurrent teardown. A device model or simulator can inject these cases deterministically. Assertions should check ownership, descriptor bounds, index arithmetic, one-time completion, and state transitions.

Observability needs event IDs, device generation, descriptor index, command, timestamp, status, and error cause without logging sensitive payloads. A trace that says “timeout” is insufficient to distinguish a lost interrupt from a device hang or ordering bug.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.4.1: Register-Level Peripheral Protocol Design

Construction application

Design a register-level protocol for a device with command, status, FIFO, interrupt, and reset registers.

Define explicit reset states, read/write side-effects, access widths, and write-ordering sequences for peripheral control registers.

The protocol must define reset values, access widths, side effects, ordering, and error acknowledgment. The driver writes configuration before start, waits for the documented ready condition, and never treats a read of a side-effect register as a pure observation.

Exercise 3.4.2: Ring Buffer Occupancy and Wrap-Around Invariants

Formalisation application

Derive ring-buffer full/empty invariants under wrapping producer and consumer counters.

State full vs empty conditions in terms of monotonic head and tail indices modulo capacity (N) without ambiguous state aliasing.

With monotonic counters, occupancy is `prod-cons`; empty means equal, full means difference equals capacity. The implementation must prevent producer advancement at full and consumer advancement at empty, and use enough counter bits to distinguish wraparound.

Exercise 3.4.3: I/O Notification Mechanism Trade-offs

Description transfer

Compare polling, interrupts, coalescing, and completion queues for three workloads.

Evaluate CPU overhead, latency distribution, and throughput trade-offs across polling, basic interrupts, interrupt coalescing, and completion queues.

Polling favors a dedicated CPU and predictable low latency; interrupts favor sparse events; coalescing favors high-rate events where batching is acceptable; completion queues favor many outstanding operations and decouple notification from work. State the workload and latency objective.

Exercise 3.4.4: DMA Buffer Ownership and Noncoherent Cache Management

Description transfer

Trace CPU-to-device and device-to-CPU DMA ownership on coherent and noncoherent platforms.

Specify clean (flush) operations before device reads and invalidate operations before CPU reads on non-cache-coherent platforms.

Transfer ownership only after buffer writes and publication ordering. Clean CPU lines before device reads on noncoherent systems; invalidate before CPU reads device-written data. Reclaim only after completion and quiescence.

Exercise 3.4.5: Device Timeout and Generation-Tagged Recovery

Construction transfer

Design a timeout/recovery state machine with generation tags.

Use generational sequence IDs and safe buffer quarantine to prevent late-arriving device completions from corrupting reused memory.

Stop submissions, mask/acknowledge events, invalidate or reset the device, advance a generation, and ignore late completions with old generation. Recovery must not reuse a buffer while the old device context can still access it.

Exercise 3.4.6: IOMMU Address Translation and Interrupt Remapping

Description synthesis

Explain how an IOMMU and interrupt remapping protect a virtual machine.

Explain how page-level DMA address isolation and message-signaled interrupt remapping enforce VM security boundaries.

IOMMU mappings restrict DMA targets; interrupt remapping restricts interrupt destinations. Teardown requires quiescing the device and invalidating translations before reassignment.

Exercise 3.4.7: Fault Injection for Asynchronous Device Completions

Construction synthesis

Propose fault-injection tests for duplicate, delayed, and malformed completions.

Design tests that simulate dropped interrupts, duplicate completion notices, corrupt payload lengths, and mid-flight device resets.

Delay completion, deliver duplicate completion, report excessive length, reorder fields, reset during transfer, and complete after timeout. Verify no double-free, buffer reuse, stale generation acceptance, or unbounded handler loop occurs.

References & Further Reading

- MIT 6.004 Computation Structures: devices, interrupts, and interconnect notes
- Cornell CS 3410: Computer System Organization and Programming

- RISC-V IOMMU specification

Performance and parallel hardware

SECTION DETAIL

Performance Models

Performance engineering begins by defining what is being measured. A processor can be faster in cycles per instruction and slower in wall-clock time if its clock is lower, or faster on one input and slower on another because its memory behavior changes. A credible performance claim names the workload, correctness condition, measurement boundary, resource metric, and baseline. Only then do equations such as CPI, AMAT, Amdahl's law, and throughput become useful rather than decorative.

Time, instruction count, and CPI

For a program executing IC instructions, average cycles per instruction CPI, and clock frequency f, idealized CPU time is

$$T = \frac{IC \cdot CPI}{f} = IC \cdot CPI \cdot T_{cycle}.$$

The formula is a decomposition, not a causal explanation. IC depends on compiler, ISA, optimization, and input. CPI depends on pipeline width, hazards, cache behavior, branches, memory-level parallelism, and contention. Frequency depends on critical paths, voltage, temperature, and design point. Improving one term can worsen another.

If a processor retires 3 billion instructions in 1.5 seconds at 2 GHz, then $IC = 3 \cdot 10^9$, cycles $= 3 \cdot 10^9$, and $CPI = 1$. If a second implementation retires 2.4 billion instructions in 1.4 seconds at 2.5 GHz, its cycles are $3.5 \cdot 10^9$ and CPI is about 1.458. It is faster in wall-clock time despite a worse CPI because it performs fewer instructions and runs at a higher frequency. Reporting CPI alone would reverse the conclusion.

$$CPI = CPI_{base} + \alpha_{br} p_{br} + \alpha_{load} p_{load} + \alpha_{L1} p_{L1} + \cdots$$

is useful only when α represents the relevant frequency and p the conditional penalty. Double-counting a miss penalty that is already embedded in a measured CPI produces a false diagnosis.

> ***Pause & Predict: Does lower CPI always imply faster execution?*** *Suppose an optimizing compiler eliminates 50% of the instructions in an inner loop. All eliminated instructions were simple 1-cycle integer additions; all long-latency memory loads remain.*
>
> - *What happens to the program's average* CPI*? Does it increase, decrease, or remain constant?*
> - *Does total execution time* T *improve or degrade? Commit your reasoning before opening the analysis below.*

Calculate $\mathrm{CPI} = \mathrm{Cycles}/\mathrm{Instructions}$ before and after removing 50 fast instructions when 50 slow loads remain.

Analysis: The average CPI actually **increases**! Suppose the original program had 100 instructions: 50 ALU ops (1 cycle each) and 50 cache-miss loads (20 cycles each).

- Original cycles: $50(1) + 50(20) = 1050$ cycles. Original $\mathrm{CPI} = 1050/100 = 10.5$.
- New program: The 50 ALU ops are eliminated. Remaining instructions: 50 loads.
- New cycles: $50(20) = 1000$ cycles. New $\mathrm{CPI} = 1000/50 = 20.0$. Average CPI nearly doubled from 10.5 to 20.0, yet total execution time dropped from 1050 cycles to 1000 cycles. Optimizing for CPI alone is a classic pitfall: total wall-clock time ($T = IC \cdot \mathrm{CPI}/f$) is the only true figure of merit.

Latency, throughput, and queueing

Latency is the time for one operation from request to completion. Throughput is the sustainable completion rate. A pipelined multiplier may have latency four cycles and throughput one result per cycle; a nonpipelined multiplier may have latency and initiation interval both four cycles. A system can have low latency but poor throughput, or high latency with excellent throughput after filling.

Queueing adds waiting time. If a server's utilization approaches one, small increases in offered load can produce large increases in response latency even when service time is unchanged. A network or memory benchmark that measures only peak bandwidth hides queueing effects and can overstate performance under contention. Tail latency—p95, p99, or higher—matters when a user request depends on many parallel subrequests.

Worked pipeline example. A four-stage unit has 2 ns latency per result after fill and accepts one input every 0.5 ns. For a single request, latency is approximately 2 ns; for 100 independent requests, completion time is $2 + (99)(0.5) = 51.5$ ns. A nonpipelined unit with 2 ns service time takes 200 ns. If the requests are dependent, however, the consumer may need to wait for each two-nanosecond result and receive little benefit from throughput.

BENCHMARK CLAIM UNDER REVIEW

One warm benchmark run places CPU A eight percent ahead of CPU B. The report attributes the difference to A's wider pipeline.

Evaluate which evidentiary standards are required before accepting this causal architectural claim:

Reference decision: Controlled repeated executions must isolate pipeline width from frequency scaling, cache footprint, and OS scheduler variance.

Performance is an empirical claim. Microarchitectural models predict the causal mechanism, while repeated measurements and hardware counter telemetry distinguish pipeline effects from thermal, scheduling, and cache confounders.

LATENCY VERSUS THROUGHPUT

Evaluate which statements accurately distinguish computational latency from throughput in processor pipelines:

Answer: Latency measures the total elapsed time required for a single operation to complete from initiation to retirement.; Throughput measures the total volume of independent operations successfully completed per unit of time.

Latency measures single-operation duration, while throughput measures rate of completion. Pipelining decouples the two by allowing multiple overlapping in-flight instructions.

Amdahl's law and bottlenecks

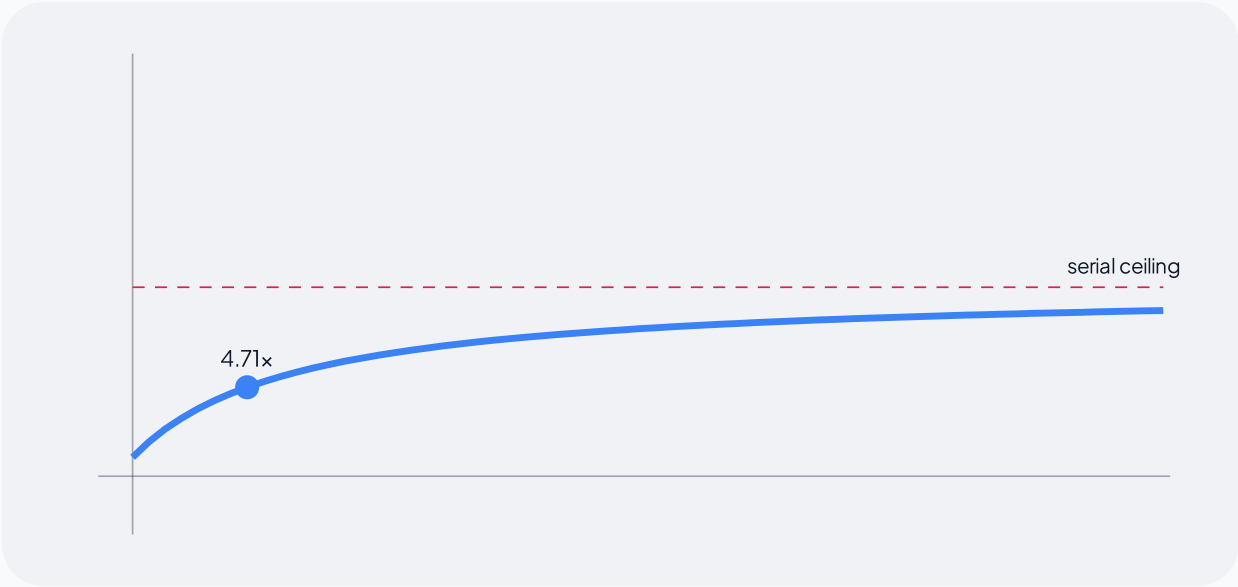
If fraction p of the original execution time is improved by factor s, total speedup is

$$S = \frac{1}{(1-p) + p/s}.$$

The unimproved fraction limits the result. If memory accounts for 40% and is made twice as fast, $S = 1/(0.6 + 0.2) = 1.25$. If serialization accounts for 70% and becomes free, the maximum speedup is $1/0.3 \approx 3.33$, not infinite.

The fraction must be measured in the baseline execution, not inferred from the fraction of source code or hardware blocks. Improving cache hit rate can reduce memory time but also change overlap with computation; the simple law may need a more detailed model. Gustafson-style scaling asks a different question: as the problem grows with processor count, how much parallel work can be usefully added? Neither law replaces a workload definition.

EXPOSE AMDAHL'S SERIAL CEILING



Even abundant parallel hardware cannot remove the execution time of the serial fraction.

Roofline and resource ceilings

For data-intensive kernels, arithmetic intensity is operations per byte transferred from a bottleneck memory level. If peak compute throughput is P_{peak} operations per second and sustainable bandwidth is B, attainable performance is bounded by

$$P \le \min(P_{peak}, B \cdot I),$$

where I is arithmetic intensity. A low-intensity kernel is bandwidth-bound; adding arithmetic units does little unless reuse or compression increases effective intensity. A high-intensity kernel is compute-bound until another resource—register capacity, cache bandwidth, instruction issue, or synchronization—becomes limiting.

Worked roofline example. A processor sustains 1 TB/s of relevant memory bandwidth and 4 TFLOP/s peak arithmetic. A kernel performs 2 floating operations per byte, so the bandwidth ceiling is 2 TFLOP/s and the kernel is bandwidth-bound under the model. Doubling peak compute to 8 TFLOP/s changes nothing; doubling reuse to 4 operations per byte raises the bandwidth ceiling to 4 TFLOP/s, at which point peak compute becomes the next ceiling.

Measurement design

A benchmark is an experiment, not a stopwatch call. Define the hypothesis, independent variable, dependent metric, controlled variables, repetitions, warm-up policy, and stopping rule. The timing boundary must exclude setup when measuring steady-state execution and include it when measuring end-to-end user experience. Report distribution and uncertainty, not only the fastest run.

Worked example: an experiment that could falsify a cache claim. Suppose the claim is that 32×32 tiling makes a 1024×1024 double-precision transpose faster because it lowers L1 data-cache misses.

- **Hypothesis:** tiling reduces L1D miss rate relative to the naive traversal, and that reduction is the mechanism behind any wall-time gain.
- **Independent variable:** traversal order (naive, 16×16, 32×32, 64×64) with the same matrix and element type.
- **Dependent variables:** median wall time over repeated runs, L1D and LLC miss rates, and memory bandwidth, measured with hardware counters.
- **Controls:** identical compiler and flags, single thread pinned to one core, fixed frequency governor, same input values and alignment, warm-up before measurement, and interleaved A/B/B/A ordering to spread thermal and background drift.
- **Stopping rule and analysis:** report all runs or their median and confidence interval, not the fastest run; use enough repetitions that the interval for the tiled-versus-naive difference is narrower than the effect the claim asserts.

Under the stated hypothesis, either reduced misses without a time improvement or a time improvement without the predicted miss reduction fails to support the proposed mechanism. These observations can suggest another ceiling or cause, such as TLB behavior or code generation, but do not identify it by themselves. Reporting only a warm, best-of-three timing cannot distinguish these cases.

Warm-up may populate instruction/data caches, trigger JIT compilation, stabilize branch predictors, or move a frequency governor. Randomized input order can prevent an algorithm from receiving a pre-sorted favorable trace, but randomization itself must be reproducible and reported.

System-level confounders can dominate small architectural effects. CPU affinity, thread count, background OS load, thermal throttling, compiler optimization flags, and heap allocator behavior must all be tightly pinned and monitored.

Use counters to test mechanisms. If a cache hypothesis predicts that row-major traversal lowers L1 misses, measure misses and bandwidth as well as time. If a branch hypothesis predicts misprediction stalls, measure branch events and compare a branchless variant. Counters are not infallible—multiplexing, sampling, speculative counting, and privilege restrictions matter—but they are stronger evidence than a narrative about “cache friendliness.”

Statistical interpretation

A mean is sensitive to outliers; a median describes a typical run; high percentiles describe tail behavior. Confidence intervals quantify uncertainty under assumptions about sampling, but repeated measurements from the same warmed system are not automatically independent. Effect size matters: a 1% improvement with 0.8% measurement variation is weak evidence unless the experiment is strengthened.

Worked example: a 1% mean gain against run-to-run variation. Five independent baseline runs give a mean of 100.0 ms with sample standard deviation 0.8 ms; five independent variant runs give 99.0 ms with the same standard deviation. The observed difference is 1.0 ms, or 1%.

Treating the runs as independent and approximately normal, the standard error of the difference is $0.8\sqrt{2/5} \approx 0.51$ ms. A Welch interval with about eight degrees of freedom and $t_{0.975} \approx 2.31$ gives a 95% interval of roughly 1.0 ± 1.17 ms, or $[-0.17, 2.17]$ ms. The interval includes zero, so the data do not establish the 1% gain even though the sample mean improved. If the same five input pairs are run in alternating order, the paired differences might be $[0.8, 1.2, 0.9, 1.1, 1.0]$ ms: mean 1.0 ms, sample standard deviation 0.16 ms, standard error 0.07 ms, and a 95% interval of about $[0.80, 1.20]$ ms, which excludes zero. Pairing removes between-run drift but does not make repeated runs on one warmed machine independent; the report must state which design was used and check for interference. Neither analysis protects against a sample that is too small to represent the workload, and neither says anything about the tail: a 1% mean gain can coexist with a worse p99.

Benchmark suites should include representative and adversarial workloads. A compiler optimization that wins on one synthetic loop may lose on a real application because instruction-cache pressure or code size changes. A cache design that wins on a small working set may lose when the active data exceeds L2. The strongest report explains where the model predicts failure and tests that boundary.

Energy and performance per watt

Performance is not the only constraint. Dynamic energy is related to switching activity, capacitance, and voltage; leakage and static power matter at idle and high density. A higher frequency may require higher voltage and disproportionately increase power.

Memory traffic often consumes more energy than an arithmetic operation. Consequently, a cache optimization can improve both time and energy, or increase energy by fetching large unused lines.

Performance per watt is a ratio with a workload denominator. A design that finishes quickly but uses a high power burst may compare differently from one that runs longer at low power, depending on whether the system is energy-limited, thermally limited, or latency-limited. State the objective before optimizing.

Worked example: voltage-frequency scaling and performance per watt. Consider a simplified dynamic-power model $P_{dyn} = \alpha C V^2 f$, with leakage and static power omitted. A core runs at 2.0 GHz and 1.0 V with $P_{dyn} = 10$ W. To reach 3.0 GHz, the design raises the supply to 1.2 V. With IPC and instruction count unchanged, throughput rises by $3.0/2.0 = 1.5\times$, but power rises by $(1.2/1.0)^2 \times (3.0/2.0) = 2.16\times$, to 21.6 W. Performance per watt therefore falls to $1.5/2.16 \approx 0.69$ of its original value, about 31% lower, and energy per instruction rises by $2.16/1.5 = 1.44$, or 44%. Including leakage and other static power can change the ratio in either direction: it must be measured or modeled at both operating points. The arithmetic does not prove that the higher frequency is wrong; it proves that the frequency choice must be evaluated against the stated objective, and that a 50% clock increase is not a 50% performance-per-watt increase.

Worked Trace: Multi-Component CPI Stall Decomposition & Frequency Trade-off

Consider evaluating an architectural proposal to deepen a pipeline from 5 stages to 10 stages:

- **Baseline Processor (Arch-A)**: Clock frequency $f_A = 2.0\ \text{GHz}$ ($T_{cycle} = 0.5\ \text{ns}$). Base $CPI = 1.0$.
  - Branch instructions: 20% of workload, branch penalty is 2 cycles, misprediction rate $p_{br} = 5\%$.
  - Memory load instructions: 25% of workload, L1 D-cache miss rate $p_{miss} = 8\%$, miss penalty 40 cycles.
- **Proposed Processor (Arch-B)**: Deeper pipeline allows clock frequency to increase to $f_B = 3.0\ \text{GHz}$ ($T_{cycle} = 0.333\ \text{ns}$).
  - Deeper pipeline increases branch misprediction penalty to 6 cycles.
  - Higher frequency increases L1 D-cache miss penalty to 60 cycles (main memory latency in nanoseconds is constant at $20\ \text{ns}$).

Step 1: Compute Effective CPI for Arch-A:

$$CPI_A = CPI_{base} + f_{br} \cdot p_{br} \cdot \text{Penalty}_{br,A} + f_{load} \cdot p_{miss} \cdot \text{Penalty}_{mem,A}$$

$$CPI_A = 1.0 + (0.20 \cdot 0.05 \cdot 2) + (0.25 \cdot 0.08 \cdot 40) = 1.0 + 0.02 + 0.80 = 1.82.$$

Step 2: Compute Effective CPI for Arch-B:

$$CPI_B = CPI_{base} + f_{br} \cdot p_{br} \cdot \text{Penalty}_{br,B} + f_{load} \cdot p_{miss} \cdot \text{Penalty}_{mem,B}$$

$$CPI_B = 1.0 + (0.20 \cdot 0.05 \cdot 6) + (0.25 \cdot 0.08 \cdot 60) = 1.0 + 0.06 + 1.20 = 2.26.$$

Step 3: Compare Wall-Clock Execution Time: For an instruction trace of $IC = 10^9$ instructions:

$$T_A = \frac{IC \cdot CPI_A}{fA} = \frac{10^9 \cdot 1.82}{2.0 \times 10^9 \text{ Hz}} = 0.910 \text{ seconds.}$$

$$T_B = \frac{IC \cdot CPI_B}{fB} = \frac{10^9 \cdot 2.26}{3.0 \times 10^9 \text{ Hz}} = 0.753 \text{ seconds.}$$

Conclusion: Even though Arch-B suffers a 24.2% penalty in CPI ($1.82 \rightarrow 2.26$) due to worse stall penalties in cycles, the 50% frequency increase ($2.0 \rightarrow 3.0$ GHz) dominates, producing an overall speedup of $\frac{0.910}{0.753} \approx 1.21\times$ (21% faster).

Counterexample & Failure Mode: The MFLOPS / MIPS Mirage (The Algorithm vs Rate Trap)

Can a lower operation count or a higher execution rate mislead an optimization decision? Consider multiplying two $N \times N$ dense matrices with $N = 1024$.

1. **Standard Triple-Loop $O(N^3)$**:
    - Operations: $2N^3 = 2(1024)^3 \approx 2.15 \times 10^9$ floating-point operations.
    - Execution on a SIMD machine: Highly optimized BLAS achieves 80% peak compute (800 GFLOP/s), finishing in 2.68 ms.
2. **Strassen's Sub-Cubic Algorithm $O(N^{2.807})$**:
    - Strassen performs 7 recursive multiplications instead of 8, so the number of scalar multiplications is $7^{10} \approx 2.82 \times 10^8$ at $N = 1024$—about $3.8\times$ fewer than the naive algorithm's N^3 multiplications. Total floating-point operations also include the 18 matrix additions/subtractions per level; under the common 7-multiplication/18-addition count, Strassen needs about 1.97×10^9 floating-point operations at this size, only about 8% fewer than the naive algorithm's $2N^3$.
    - Execution on the same SIMD machine: complex recursive memory layouts, matrix additions, and poor vector-register utilization lower compute efficiency to only 20% peak (200 GFLOP/s).
    - Wall time: $\frac{1.97 \times 10^9}{200 \times 10^9} \approx 9.9$ ms.

3. **The Mirage Breakdown**:
    - The standard algorithm sustains $800\ \mathrm{GFLOP/s}$ and finishes in $2.68\ \mathrm{ms}$.
    - Strassen sustains only $200\ \mathrm{GFLOP/s}$; at this size its operation-count advantage is too small to compensate, so it finishes in about $9.9\ \mathrm{ms}$.
    - The lesson is not that Strassen is always slower. It is that a rate or an asymptotic exponent does not determine wall time: the comparison must count the same operations, include additions and data movement, and use the same implementation efficiency. Reporting 2.82×10^8 as Strassen's total work silently drops the additions, and reporting $200\ \mathrm{GFLOP/s}$ as a performance verdict silently drops the algorithm's cost model.
    - **Rule**: Optimizing for MFLOPS, MIPS, or CPI without fixing the algorithmic work and the instruction count (IC) produces invalid engineering decisions.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.1.1: CPU Performance Equation Derivation and Analysis

Construction application

Compute CPU time, cycle count, and CPI from instruction count, frequency, and measured execution time. Explain why a lower CPI does not guarantee lower wall-clock time when instruction count or frequency differs.

Relate instruction count, cycles per instruction (CPI), and clock cycle time to total execution runtime.

Use $T = IC \cdot CPI / f$. Cycle count is $T \cdot f$ and CPI is cycles divided by IC. An optimization that reduces CPI by adding simple instructions (raising IC) or lowering clock frequency f can increase total execution runtime T despite a numerically lower CPI.

Exercise 4.1.2: Amdahl's Law with Multiple Serial and Parallel Bottlenecks

Construction application

Apply Amdahl's law to a workload spending 25% of time in memory access, 15% in synchronization, and 60% in compute, when memory access is accelerated by $4\times$ and synchronization by $2\times$. Calculate the overall speedup and identify the limiting bottleneck.

Compute the maximum achievable speedup when an accelerated component represents a fraction of execution time.

The baseline fraction sum is $0.25 + 0.15 + 0.60 = 1.0$. The improved execution time fraction is $T_{\text{new}} = 0.25/4 + 0.15/2 + 0.60 = 0.0625 + 0.075 + 0.60 = 0.7375$. Overall speedup is $S = 1/0.7375 \approx 1.356$. The unaccelerated compute portion (60%) now accounts for $0.60/0.7375 \approx 81.4\%$ of total runtime, establishing the primary bottleneck.

Exercise 4.1.3: Roofline Model Kernel Classification and Ceiling Shifts

Description recall

Using the Roofline model with memory bandwidth B, peak compute performance P_{peak}, and arithmetic intensity I, explain how to classify a kernel as compute-bound versus memory-bound. What architectural and code transformations raise each ceiling?

Compare operational intensity times memory bandwidth ($I \cdot B$) against peak arithmetic throughput to classify memory-bound vs compute-bound kernels.

Compute the operational throughput boundary $I \cdot B$. If $I \cdot B < P_{\text{peak}}$, the kernel is memory-bound; if $I \cdot B \geq P_{\text{peak}}$, it is compute-bound. Increasing SIMD vector width or FMA units raises the compute ceiling P_{peak}. Loop tiling, register blocking, cache reuse, and data compression raise arithmetic intensity I, shifting the kernel rightward along the bandwidth slope.

Exercise 4.1.4: Experimental Methodology for Cache-Bound Array Traversals

Verification application

Design a rigorous experimental harness to test whether a 2D matrix traversal is cache-bound. Identify at least five confounding variables and how to control for them.

Vary working set size across L1/L2/LLC boundaries and control for frequency scaling, page faults, compiler vectorization, and cache warmup.

Vary matrix dimensions such that working set sizes span $8\ \mathrm{KB}$ (L1-resident), $256\ \mathrm{KB}$ (L2-resident), $16\ \mathrm{MB}$ (LLC-resident), and $128\ \mathrm{MB}$ (DRAM-bound). Control confounders by: (1) locking CPU clock governor to prevent dynamic frequency scaling; (2) pinning thread affinity to a single core; (3) pre-faulting and warming pages to eliminate page-fault overhead; (4) using identical compiler optimization flags (`-O3 -fno-tree-vectorize`); and (5) reading hardware performance counters (`PAPI_L1_DCM` , `PAPI_L2_DCM`) to confirm cache misses.

Exercise 4.1.5: Average Throughput versus Tail Latency Divergence

Description recall

Explain how architectural optimizations like batching and aggressive caching can simultaneously improve average throughput while degrading 99th-percentile (p99) tail latency.

Consider how caching or request batching can accelerate the fast path while queuing complex or evicted requests into high p99/p99.9 percentiles.

Batching buffers incoming requests to saturate wide vector units, maximizing throughput at the cost of holding individual early requests in queues, inflating tail latency. Similarly, cache admission policies accelerate common requests but cause tail requests encountering cold evictions or lock contention to stall for milliseconds. System performance evaluation must report percentiles rather than relying exclusively on mean throughput.

Exercise 4.1.6: Pipeline Latency, Initiation Interval, and Throughput Metrics

Construction application

Distinguish between pipeline latency, initiation interval, throughput, and hardware utilization. Give the exact completion time for N independent operations traversing a k-stage pipeline with clock period τ and initiation interval $II = 1$.

Differentiate between total end-to-end service delay (latency), minimum input issue separation (initiation interval), and completions per second (throughput).

Latency is total propagation time from issue to retirement ($k \cdot \tau$). Initiation interval (II) is the cycle delay between successive accepted inputs ($1 \cdot \tau$). Throughput is retired operations per second ($1/\tau$). Total completion time for N operations is $T(N) = (k + N - 1) \cdot \tau$. For $N = 1, T = k\tau$; as $N \to \infty$, incremental completion time per operation approaches τ.

Exercise 4.1.7: Benchmarking Methodology Critique: Best-of-Three Reporting

Generalisation transfer

Critique the practice of publishing only the minimum (fastest) elapsed runtime from three experimental runs. Detail why this anti-pattern invalidates scientific reproducibility and prescribe the proper reporting protocol.

Identify how minimum-of-three filtering suppresses system variance, thermal throttling, OS context switches, and cache eviction noise.

Reporting the minimum of three runs censors natural system variance, OS scheduler preemption, thermal throttling, and cache eviction effects, presenting an unrepresentative best-case artifact. The proper protocol requires: performing at least 30 independent iterations after verified cache warmup; reporting median, interquartile range (IQR), and full percentiles (p50, p90, p99); and applying non-parametric statistical tests (e.g. Mann-Whitney U) with confidence intervals.

References & Further Reading

- Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach* (6th ed.), Chapter 1 ("Fundamentals of Quantitative Design and Analysis") & Appendix B. Morgan Kaufmann.
- Williams, S., Waterman, A., & Patterson, D. (2009). *Roofline: An insightful visual performance model for multicore architectures*. Communications of the ACM, 52(4), 65–76.
- Amdahl, G. M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities*. AFIPS Conference Proceedings, Vol. 30, 483–485.

- Strassen, V. (1969). *Gaussian elimination is not optimal*. Numerische Mathematik, 13(4), 354–356. https://doi.org/10.1007/BF02165411

SECTION DETAIL

Superscalar and Out-of-Order Execution

An in-order scalar pipeline issues at most one instruction per cycle and waits when the next instruction cannot proceed. Modern high-performance cores attempt to exploit *instruction-level parallelism* (ILP): independent instructions can execute simultaneously, even when they appear sequentially in the program. A superscalar processor has multiple issue slots or functional pipelines; an out-of-order processor allows a ready younger instruction to execute while an older instruction waits for a cache miss or long-latency operation. The implementation must nevertheless preserve the ISA's ordered architectural observations.

Limits of instruction-level parallelism

DEFINITION

Within a finite instruction window of N instructions with true data-dependence DAG $G = (V, E)$ weighted by execution latencies $l(v)$, the achievable Instructions Per Cycle on a W-wide machine is bounded above by $\mathrm{IPC} \le \min\left(W,\ N / L_{\mathrm{crit}}(G)\right)$, where $L_{\mathrm{crit}}(G)$ is the longest latency-weighted RAW path in G.

The dependence graph determines the parallelism available in a finite instruction window. Independent additions can execute concurrently; a chain `r1=r2+r3; r4=r1+r5; r6=r4+r7` has a critical path of three additions even if the core has many ALUs. Memory aliases, branches, exceptions, and finite registers limit how far the processor can look ahead.

Superscalar width W is an upper bound on issue per cycle, not a guarantee. If a workload has insufficient independent instructions, one functional unit is busy while others idle. The effective IPC is bounded by dependencies, front-end fetch/decode width, execution-unit availability, cache misses, and retirement bandwidth.

Worked dependency example. For `A: r1=r2+r3`, `B: r4=r5+r6`, and `C: r7=r1+r4`, A and B can execute in parallel on two adders. C waits for both. Define two-cycle latency to mean that an instruction issued at the start of cycle t has a usable result at the start of $t + 2$. A and B issue at cycle 1, become ready at cycle 3, and C issues at cycle 3 with its result ready at cycle 5. A single-issue

machine with a pipelined adder issues A at cycle 1 and B at cycle 2; C waits until cycle 4 and produces its result at cycle 6. Neither schedule includes fetch or retirement latency. Issue width reduces elapsed time here, but does not shorten the true dependence through C.

Renaming and physical registers

Architectural register names create false dependencies. In:

```
add r1, r2, r3
add r2, r4, r5
```

the second instruction writes `r2` after the first reads it, a WAR anti-dependence. It is not a true flow of data. Register renaming maps each destination to a fresh physical register and records which physical register currently represents each architectural name. The first instruction reads the old physical `r2`; the second writes a new physical register. WAW hazards similarly disappear because distinct writes use distinct physical destinations.

Renaming does not remove RAW dependencies. If a later instruction reads the architectural register after a new mapping is installed, it receives the producer's physical register and waits for its readiness. The rename map is speculative state: a branch checkpoint records enough mapping information to restore the pre-branch name map if prediction fails.

Physical registers need reclamation. When a new mapping replaces an old architectural mapping and no in-flight instruction can still read the old physical register, the old register can return to the free list. Reference tracking, retirement maps, or equivalent mechanisms prevent premature reuse. A physical-register leak eventually stalls dispatch even when execution units are idle.

> ***Pause & Predict: Can Renaming Accelerate a Pure RAW Chain?*** *Suppose you have an infinite supply of execution units and physical registers, and you run the following code:*
>
> ```
> add a0, a1, a2
> add a0, a0, a3
> add a0, a0, a4
> add a0, a0, a5
> ```
>
> *Will register renaming allow these four instructions to execute concurrently? Commit your reasoning before opening the analysis below.*

Recall the difference between true data dependencies (RAW) and false name dependencies (WAR, WAW). Which ones can renaming eliminate?

Analysis: No! Not by a single cycle. Register renaming eliminates only **name dependencies**:

- Write-After-Read (WAR / anti-dependence)
- Write-After-Write (WAW / output dependence)

It does **not** eliminate Read-After-Write (RAW / true data dependence). Each instruction strictly consumes the mathematical value produced by its predecessor. Renaming will assign physical registers:

- `P1 = a1 + a2`
- `P2 = P1 + a3` (must wait for P1 to be computed)
- `P3 = P2 + a4` (must wait for P2 to be computed)
- `P4 = P3 + a5` (must wait for P3 to be computed)

The critical path remains exactly 4 serial addition latencies. Only algorithmic reassociation (e.g. tree reduction $(a1 + a2) + (a3 + a4)$) or compiler loop unrolling can expose true ILP from serialized chains.

EXECUTE EARLY, RETIRE IN ORDER

A younger independent instruction finishes before an older cache-missing load. A branch older than both is later found mispredicted.

Evaluate which hardware structures and mechanisms preserve precise architectural state in this scenario:

Reference decision: Register renaming eliminates false hazards, while reorder buffer entries prevent speculative results from committing until in-order retirement.

Register renaming removes anti- and output-dependencies, while in-order ROB retirement buffers speculative results until older branches and memory faults are definitively resolved.

Reservation stations and wakeup

Out-of-order cores hold decoded instructions in issue queues or reservation stations. Each entry stores the operation, source physical-register identifiers, readiness bits or values, destination, and exception/speculation metadata. When a producer completes, a wakeup mechanism marks dependent operands ready. A select mechanism chooses ready instructions for available functional units.

Worked trace: tag wakeup and select. Suppose two reservation-station entries hold

```
RS1: fadd f3, f0, f4    sources (P10, P4)   ready bits (0, 1)
RS2: fsub f5, f1, f2    sources (P6, P8)    ready bits (1, 1)
```

and only one FP adder is free. The producer of `P10` broadcasts at the end of cycle 3. At that broadcast, RS1 compares `P10` against its outstanding source tags, sets the first ready bit, and becomes selectable for cycle 4. RS2 was already selectable; if both request the adder in cycle 4, the selector must choose by age or another policy, and the loser waits. In this timing model the broadcast at the end of cycle 3 cannot also drive an issue earlier in cycle 3. Other designs can combine or pipeline wakeup and select differently, trading latency against the delay of the compare-and-select path. The timing model, not the ISA, decides when a dependent can issue.

Wakeup and select are timing-critical because a wide machine compares many tags every cycle. A centralized issue queue is flexible but expensive; clustered or partitioned designs reduce wiring but can require operand movement between clusters. The architecture is not changed by these choices, but performance and power are.

Long-latency operations create pressure. A cache-miss load may occupy a queue entry and a physical destination while independent work proceeds. If the reorder buffer or load/store queue fills, dispatch eventually stops—a phenomenon called back-end or window pressure. Increasing the window can expose more ILP but raises energy, area, and recovery cost.

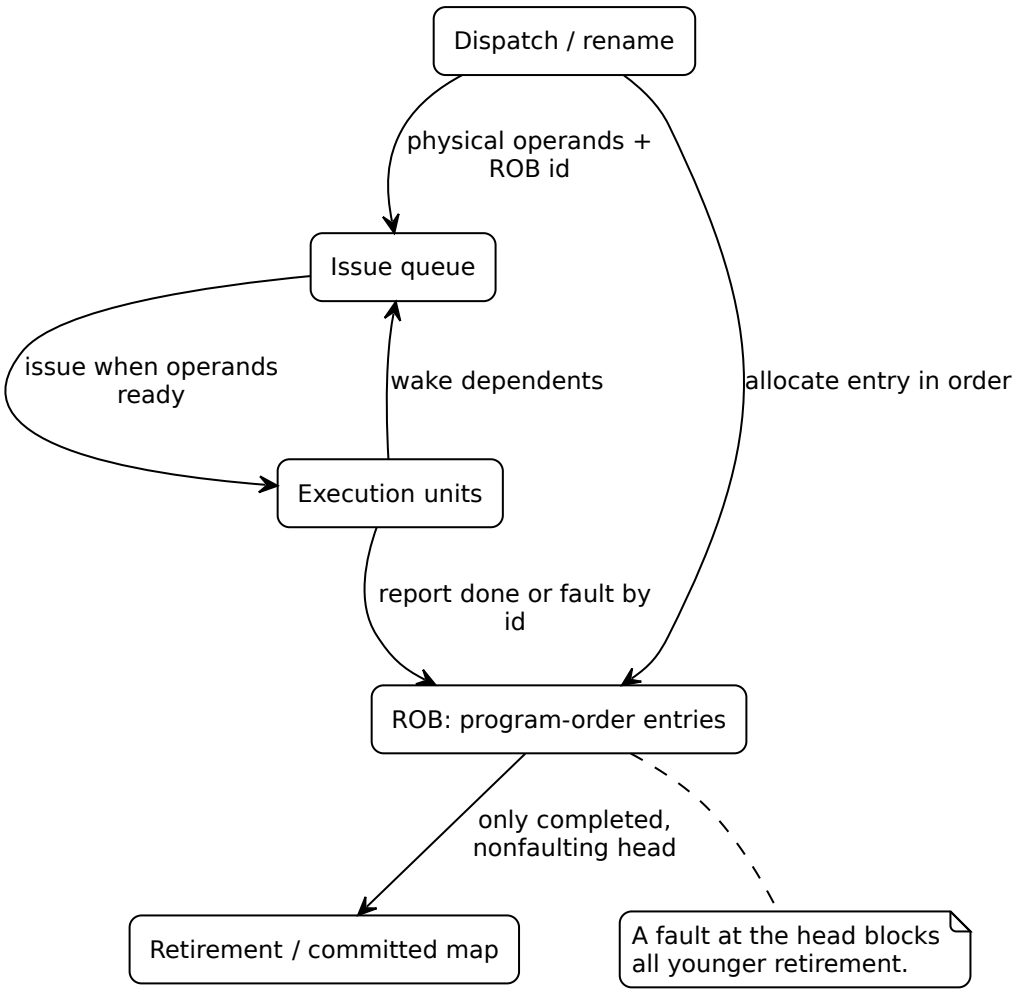
Reorder buffer and precise retirement

Execution completes out of order, but retirement commits architectural effects in program order. A reorder buffer (ROB) records program order, destination information, exception state, and completion. The head entry retires when its operation is complete and no older fault exists. Younger completed instructions wait.

This design provides precise exceptions. If instruction 10 faults, instructions 0–9 may retire, instruction 10 records the fault, and instructions 11 onward are discarded even if they executed internally. Stores often enter a store buffer and become globally visible only at retirement or under a carefully defined memory-ordering protocol. Register results can be kept in physical registers while the committed map changes at retirement.

Worked exception trace. Instruction A is an older load that faults; instruction B is a younger add that completes; instruction C is a younger store that completes. The ROB marks B and C done but cannot retire them because A is at the head and faulting. The machine discards B and C, leaves architectural registers and memory as though A had not committed, and enters the handler with A's fault information. If C had written memory immediately at execute completion, precise exception semantics would be violated.

A ROB entry is allocated in program order at dispatch, before execution completes. Otherwise a younger fast instruction could appear older than a stalled load. The graph separates allocation from completion reporting and from the permission to retire.



REMARK **ROB precise exception boundary**

If the reorder buffer contains older load A followed by independent add B, and B completes while A waits on a page fault, both entries were allocated in order at dispatch. Although B holds a valid speculative result, B cannot retire past A. At the fault boundary when A reaches the ROB head, A traps and younger instruction B is squashed; neither instruction writes an architectural register result. Older instructions committed prior to A remain architectural.

ARCHITECTURAL STATE PRESERVATION MATRIX

Evaluate each statement regarding what an out-of-order superscalar processor must guarantee to software to preserve the sequential execution model:

Answer: Register values and memory stores must be made architectural strictly in program order at retirement.; Exceptions must be reported precisely so no younger instructions have modified architectural state when the handler runs.

Internal execution may reorder independent operations arbitrarily, but retirement, store buffering, and reorder buffer tracking guarantee precise exceptions and sequential program semantics.

Load/store ordering and memory disambiguation

Registers expose explicit dependences, but memory addresses may be unknown until execution. A younger load and older store can execute out of order only if the core proves their addresses do not alias, or if it speculates and can recover when the guess is wrong. The load/store queue records program order, addresses, data, and permissions. Store-to-load forwarding lets a load obtain data from an older store that has not reached cache, provided the addresses and byte masks match.

Partial overlap is difficult. A 4-byte load may overlap two older stores with different byte ranges; forwarding must merge bytes or stall. An address predictor may speculate that a load does not alias an older store; a later conflict triggers replay of the load and its dependents. Memory consistency rules further restrict when stores become visible to other cores.

Worked example: byte-mask forwarding (little-endian). Two older stores are in the load/store queue:

```
# Assume x1 = 0x100.
S1: sw x5, 0(x1)     bytes 0x100..0x103 = A0 A1 A2 A3
S2: sw x6, 4(x1)     bytes 0x104..0x107 = B0 B1 B2 B3
```

Assume an implementation that permits these misaligned accesses and can forward bytes from multiple stores. A younger `lh x7, 2(x1)` reads bytes `0x102..0x103`, so it can forward `A2 A3` from S1 without waiting for S2. A younger `lw x8, 2(x1)` reads `0x102..0x105` and needs `A2 A3` from S1 and `B0 B1` from S2. If S2's data has not arrived, the load cannot complete by forwarding from S1 alone: it must wait, or it must have speculated and be prepared to replay. A younger `lw x9, 0(x1)` is fully covered by S1 and can forward the whole word. The byte masks, not just the base addresses, determine whether forwarding is possible; an address match with a partial overlap is not enough.

Speculation and branch recovery

A branch predictor supplies a next PC before the branch resolves. The front end fetches and renames along that path. Each speculative instruction carries a branch-mask or checkpoint identifier. When the branch resolves correctly, the work may continue; when it mispredicts, the core restores the rename map, frees or invalidates younger ROB and issue entries, clears speculative stores, and redirects fetch.

Worked recovery trace. A branch `B` occupies ROB entry 20 and is predicted taken. The front end fetches and renames 24 younger instructions into ROB entries 21–44. When `B` resolves mispredicted, the core restores the rename map from `B`'s checkpoint, marks entries 21–44 invalid, clears any speculative stores from that range, and redirects fetch to the correct target. Instructions older than `B` are unaffected and may retire.

State	Before resolution	After recovery
Rename map	includes speculative mappings from 21–44	restored to `B`'s checkpoint
ROB	20–44 occupied	21–44 squashed; 20 may retire when all older instructions have retired
Store buffer	speculative stores from 21–44	those stores discarded
Fetch PC	wrong-path target	corrected target

The invariant is falsifiable: no instruction younger than `B` may retire or make a store globally visible after the misprediction is known. A machine that lets a wrong-path store drain before the squash violates precise state even if the register results are later discarded. The recovery cost is not just the 24 discarded entries; it includes the refill latency and the time to re-establish the front-end stream.

Recovery bandwidth matters. A machine that can dispatch eight instructions per cycle may need several cycles to clear a large wrong-path window. Predictor accuracy and branch resolution depth determine the average penalty. Speculation also consumes cache and predictor capacity, so wrong-path work can slow the correct path even before a flush.

Exceptions, interrupts, and replay

Out-of-order execution may discover exceptions before an instruction reaches the ROB head. The exception is recorded, but the machine delays architectural entry until older instructions retire. Some operations are restartable; others require suppressing duplicate effects. A speculative page-walk fault, for example, may be handled differently from an architectural load fault depending on the design.

Worked trace: replay is not an exception. An older store `S` is at ROB entry 9 and a younger load `L` is at entry 11. `L` executes before `S`'s address is known because the memory disambiguator predicts no alias. When `S`'s address resolves and matches `L`, the LSQ does not raise an architectural fault: it marks `L` and its dependents for replay. `L` re-executes after the store data is available, and the dependents re-execute from the replayed value. No handler runs, no `sepc` is written, and no architectural register or memory state is exposed for the wrong value.

A true page fault on `L` follows a different path: the fault is recorded in `L`'s ROB entry and delivered only when `L` reaches the head; `sepc` receives `L`'s PC, younger entries are squashed, and the handler may retry the load. An interrupt is external and is taken at a precise instruction boundary: older instructions retire, younger speculative work is discarded or held, and the saved PC identifies the next instruction to execute on return.

Event	Architectural fault?	Saved PC	Recovery
Memory-order violation	no	none	replay load and dependents
Load page fault	yes	faulting load PC	handler retries or terminates
Timer interrupt	no (asynchronous)	next instruction to execute	return to saved PC

The common requirement is that no younger effect becomes visible before the event's architectural boundary.

Interrupts are taken at a precise boundary chosen by the ISA. The implementation must ensure all older effects are committed and all younger effects are recoverable. Replay mechanisms for memory-order violations and cache misses must preserve the same visible instruction ordering. The more aggressively a core speculates, the more state it must retain to restore the architectural model.

Performance and energy trade-offs

Increasing width, window size, physical registers, and speculation can improve IPC on ILP-rich code. It can hurt on branchy, pointer-heavy, or synchronization-heavy code because the extra machinery consumes power and may lengthen critical wakeup/select paths. A high IPC can also be misleading if the code performs more instructions or stalls at memory.

Worked example: width, IPC, and energy per instruction. Compare two same-frequency cores at 3.0 GHz on a workload whose IPC rises from 2.0 to 2.3 when issue width increases from four to six:

Core	Width	IPC	Power	Time per instruction	Energy per instruction	Throughput per watt
A	4	2.0	20 W	0.167 ns	3.33 nJ	0.300×10^9 instructions/s/W
B	6	2.3	26 W	0.145 ns	3.77 nJ	0.265×10^9 instructions/s/W

Core B executes 15% more instructions per second, reducing time per instruction by about 13%, but uses about 13% more energy per instruction and delivers about 12% fewer instructions per second per watt. The correct choice depends on the objective: B is preferable in a latency- or throughput-constrained system with power headroom, while A is preferable when the system is power- or thermally limited. The table also shows why an IPC claim alone is insufficient: B's higher IPC is real, but it does not by itself establish a better design point. If B's frequency also rose to 3.5 GHz, its time and energy per instruction would improve, but only if the power estimate still holds after the voltage increase; a 26 W budget at 3.0 GHz cannot be assumed at 3.5 GHz.

The optimal width depends on workload mix. A wide core may waste energy on a serial dependency chain; a modest core with a better cache or memory system may finish sooner. Architectural evaluation therefore uses traces, counters, and representative suites, not a single synthetic instruction stream.

Verification strategy

An out-of-order implementation is verified against a reference ISA model at retirement. Random instruction streams stress renaming and commit; constrained tests stress dependencies, exceptions, mispredictions, overlapping memory accesses, and full queues. Assertions should ensure that no instruction retires before all older instructions, no physical register is reused while live, and no younger store becomes visible before the allowed commit point.

Formal refinement abstracts away issue order. It relates each microarchitectural state to a sequence of retired architectural states. Internal execution can be arbitrary when it produces no prohibited external effect; retirement must correspond to the next reference transition. This proof style explains why caches, predictors, and speculative values need not appear in the ISA.

Worked Trace: Cycle-by-Cycle Tomasulo Renaming & Out-of-Order Execution

Consider four floating-point instructions dispatched in program order on a 2-way superscalar out-of-order processor:

```
I1: fld    f0, 0(x1)        # Load: 2 execution cycles (address gen + cache hit)
I2: fmul.d f4, f0, f2        # Multiply: 4 execution cycles (RAW on f0 from I1)
I3: fsub.d f2, f6, f8        # Subtract: 1 execution cycle (WAR on f2 with I2)
I4: fadd.d f6, f4, f2        # Add: 1 execution cycle (RAW on f4 from I2 and f2 from I3)
```

Suppose the initial map is $f0 \to P_0, f2 \to P_2, f4 \to P_4, f6 \to P_6, f8 \to P_8$, with fresh registers $P_0^1, P_2^1, P_4^1, P_6^1$. Define one timing model: at most two instructions dispatch in program order per cycle; execution starts no earlier than the cycle after dispatch; a result broadcasts at the end of its last execution cycle and is usable starting in the next cycle. Separate result paths permit two broadcasts in a cycle. At most two completed ROB entries retire in order per cycle, starting the cycle after broadcast. All input registers and the load's address are ready initially; the load hits the cache.

Instruction	Dispatch: destination and sources	Execution	Broadcast	Retirement
I1 `fld f0`	cycle 1: P_0^1; address ready	cycles 2–3	end of 3	cycle 4
I2 `fmul f4`	cycle 1: $P_2^1 \leftarrow P_0^1$, P_2	cycles 4–7	end of 7	cycle 8
I3 `fsub f2`	cycle 2: $P_4^1 \leftarrow P_6$, P_8	cycle 3	end of 3	cycle 8
I4 `fadd f6`	cycle 2: $P_6^1 \leftarrow P_2^1$, P_4	cycle 8	end of 8	cycle 9

At the end of cycle 3, I1 and I3 have broadcast. I2's reservation station now has both operands; I4 has P_4^1 but still waits for P_2^1. I3 can finish first because I2 captured the *old* P_2^1 at dispatch; I3 writes the distinct P_4^1 and cannot overwrite I2's source. The true dependence from I2 to I4 remains: I4 cannot execute until the cycle after I2 broadcasts.

End of cycle	ROB in program order, after retirement	Reservation-station state
2	I1 executing; I2 waiting; I3 and I4 dispatched	I2 waits for P_0^1; I4 waits for P_2^1, P_4^1
3	I1 done; I2 waiting; I3 done; I4 waiting	I2 ready; I4 waits for P_2^1
4	I1 retired; I2 executing; I3 done; I4 waiting	I4 still waits for P_2^1
7	I2 done; I3 done; I4 waiting	I4 becomes ready after the broadcast
8	I2 and I3 retired; I4 done	I4 has broadcast but cannot retire until cycle 9
9	empty	no waiting instruction

I3 completes before I2 but cannot retire before it. A different issue width, broadcast bandwidth, cache latency, or retirement policy changes the cycle numbers, not the dependencies or the in-order-retirement requirement.

Counterexample & Failure Mode: Memory Disambiguation Violation (Speculative Load Bypass Hazard)

Unlike registers where dependencies are explicitly encoded in instruction bitfields, memory addresses are computed dynamically at runtime.

Consider the following program sequence:

```
I1: sw  x5, 0(x1)     # Store word: the load that produces x1 is still
waiting on a cache miss
I2: lw  x6, 0(x2)     # Load word: address x2 is known immediately
I3: add x7, x6, x8    # Dependent consumer
```

1. **Speculative Execution Decision**: Because the load producing `x1` is stalled on a cache miss, an aggressive memory disambiguator speculates that `x1 != x2` (no memory aliasing). The processor executes the load `I2` immediately out-of-order and feeds `x6` to `I3`.

2. **The Hazard Manifestation**: Five cycles later, the load's data arrives from L2 cache and writes `x1`; the hardware then evaluates `0(x1)`. Crucially, `x1 == x2`!

   - `I1` was supposed to store the new value into memory location `0(x1)`.
   - `I2` speculatively read the **stale** value prior to `I1`'s write!
   - `I3` and all downstream instructions computed with corrupt, non-architectural data.

3. **Required Microarchitectural Recovery**: If the Load-Store Queue (LSQ) fails to detect this address collision when `I1` resolves, a silent data corruption bug occurs. A correct out-of-order core must snoop the store address against all younger speculative loads in the LSQ. Upon collision detection, it marks `I2` and its dependents for replay; if they have already executed, it squashes and re-executes them after `I1`'s data is available. The store `I1` itself is older and remains; recovery does not roll architectural state back to before `I1`.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.2.1: Register Renaming for Dependency Resolution

Construction application

Rename an instruction sequence containing RAW, WAR, and WAW dependencies and draw the resulting physical-register mappings.

Assign fresh physical registers to eliminate false WAR and WAW naming hazards while preserving RAW chains.

Allocate fresh physical destinations for every write, while each read uses the mapping visible at dispatch. WAR and WAW disappear; a consumer still waits for the physical register produced by its true source instruction.

Exercise 4.2.2: Superscalar Dual-Issue Instruction Scheduling

Verification application

Schedule four instructions on two ALUs with one three-cycle multiply and one dependent consumer. State issue, completion, and retirement cycles.

Match independent instructions with complementary functional unit requirements in the same issue cycle.

The multiply can issue alongside an independent add if units and operands are ready. Its consumer cannot issue until three-cycle completion. Retirement remains in program order even if the independent add finishes first.

Exercise 4.2.3: Memory Disambiguation and Store-to-Load Forwarding

Description transfer

Explain why an older unresolved store can prevent a younger load from executing even when the load's register operands are ready.

Explain why unknown older store addresses force younger loads to wait or verify against store queues.

The store address may equal the load address. Executing the load first could read an older value and violate program order. The queue can prove nonaliasing, forward from the store, or wait and replay if speculation fails.

Exercise 4.2.4: Reorder Buffer Precise Exception Handling

Description transfer

Trace ROB state for an exception at the head with two younger completed instructions.

Trace instruction retirement at the ROB head to ensure exceptions take effect at exact architectural boundaries.

Older instructions before the head retire; the faulting head records cause; younger completed entries do not retire and are discarded. Architectural register and memory state contains no younger effects.

Exercise 4.2.5: Branch Misprediction Impact on Out-of-Order IPC

Generalisation transfer

Derive the expected IPC effect of a branch frequency, prediction accuracy, and recovery cost.

Calculate IPC degradation from branch flush recovery when speculative pipelines retire wrong-path work.

Branch cost is frequency times misprediction probability times penalty. IPC or CPI must then be combined with base issue width and clock period; a lower CPI does not automatically mean lower wall-clock time.

Exercise 4.2.6: Speculative Checkpointing and Architectural Restore

Description synthesis

Identify the state that must be checkpointed for branch recovery.

Identify the rename tables and register allocation states that must be restored on branch mispredictions.

Save the rename map or a recoverable checkpoint, the ROB boundary/branch mask, speculative store state, and fetch PC/target state. Predictor updates may use a separate policy but cannot allow wrong-path architectural effects.

Exercise 4.2.7: Physical Register Free-List Management Invariants

Formalisation synthesis

Give three assertions that would catch premature physical-register reuse.

Assert that physical registers are released to the free list only when younger overwriting instructions retire.

Assert that a physical register is allocated only when free, not allocated while referenced by any in-flight source, and not returned to the free list until retirement has removed every architectural and speculative reference.

References & Further Reading

- Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach* (6th ed.), Chapter 3 (“Instruction-Level Parallelism and Its Exploitation”). Morgan Kaufmann.
- Tomasulo, R. M. (1967). *An efficient algorithm for exploiting multiple arithmetic units*. IBM Journal of Research and Development, 11(1), 25–33.
- Yeager, K. C. (1996). *The MIPS R10000 superscalar microprocessor*. IEEE Micro, 16(2), 28–40.

SECTION DETAIL

Multicore and SIMD

When a single instruction stream cannot provide enough parallel work, architects add parallel execution resources. A multicore processor contains several cores, each with a program counter, registers, pipeline, and usually private upper-level caches, connected to shared memory or shared lower-level caches. SIMD (single instruction, multiple data) applies one instruction to multiple lanes of a vector register. Both approaches can increase throughput, but neither turns an inherently sequential algorithm into a parallel one. The limiting resources are serial dependencies, synchronization, memory bandwidth, communication, locality, and the cost of coordinating independent work.

Parallel speedup and its limits

THEOREM

For a fixed workload with serial fraction $s \in (0, 1]$ and parallelizable fraction $1 - s$ across P homogeneous processors, Amdahl's law bounds speedup by $S(P) = \frac{1}{s + (1-s)/P} < \frac{1}{s}$.

Let a fraction s of a program be inherently serial and the remaining $1 - s$ be perfectly parallel. With P processors, Amdahl's law gives

$$S(P) = \frac{1}{s + (1 - s)/P}.$$

As $P \to \infty$, speedup approaches $1/s$. If $s = 0.1$, the limit is 10× even with unlimited cores. Real programs also pay scheduling, synchronization, load imbalance, cache-coherence, and communication costs, so the observed speedup is lower.

Gustafson's law describes a scaled problem where the parallel portion grows with processor count. It can justify using more cores for larger datasets, but it does not invalidate Amdahl's limit for a fixed problem. A performance claim must specify whether input size is fixed or scaled.

Worked example. A workload is 90% parallel and runs in 100 seconds on one core. On 8 cores, the ideal time is $10 + 90/8 = 21.25$ seconds, speedup 4.71, not 8. If synchronization adds 2 seconds and load imbalance adds 10% to the parallel component, the real time is larger still. Measuring only the

parallel kernel can hide serial setup and teardown that dominate end-to-end time.

BOUND THE SPEEDUP

Why can doubling the number of cores fail to double application speed?

Answer: Serial work, synchronization, memory bandwidth, and imbalance limit speedup

Amdahl-style limits and shared resources constrain the useful parallel fraction. Extra cores may add contention rather than throughput.

Shared-memory execution model

In a shared-memory model, threads communicate by loads and stores to a common address space. A thread's registers and private cache are local; the shared memory abstraction is implemented through coherence and an interconnect. A data race occurs when conflicting accesses from different threads are not ordered by synchronization and at least one is a write. The language and ISA may treat a data race as undefined or weakly ordered, so "the hardware usually returns the latest value" is not a portable correctness argument.

Locks provide mutual exclusion if acquire and release operations impose the required ordering. A producer should publish data before setting a ready flag with release semantics; a consumer should acquire the flag before reading the data. Without ordering, the consumer may see the flag while some data stores remain invisible or reordered.

Worked message-passing example. The producer writes `payload=42`, then performs a release store `ready=1`. The consumer loops with an acquire load of `ready` and, after observing 1, reads `payload`. The synchronization creates a happens-before relation from the payload write to the consumer's read. Replacing both with ordinary relaxed operations may permit the consumer to observe `ready=1` and an old payload under a weak memory model. Coherence of each individual location does not establish ordering between locations.

FALSE SHARING WITHOUT SHARED VARIABLES

Two threads update different counters placed on the same cache line. Performance collapses as cores increase even though no thread reads the other's counter.

Which mechanism best explains the slowdown?

Reference decision: Coherence repeatedly transfers ownership of the shared cache line

Coherence operates at cache-line granularity. Independent words on one line can therefore invalidate and transfer the line between cores, creating false sharing without a source-level data dependency.

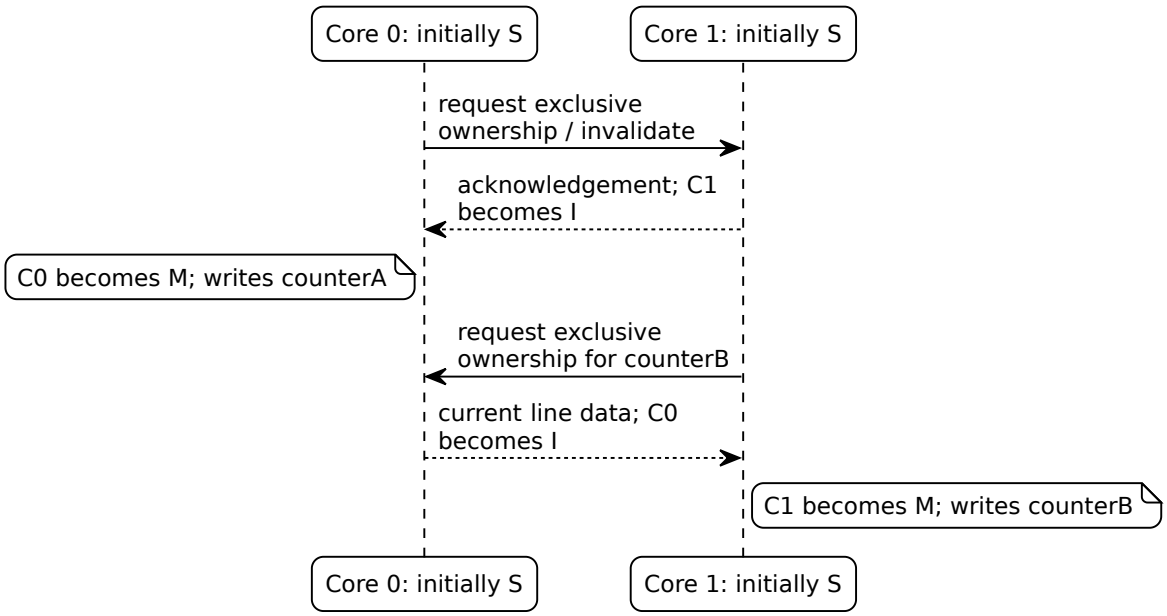
Cache coherence and false sharing

Each core may cache a line, so a write requires a coherence action. An invalidation protocol makes other copies invalid before or as the writer obtains ownership; an update protocol sends new data to sharers. MESI-like protocols track modified, exclusive, shared, and invalid states. A line in modified state is the current copy and must be written back or transferred before another agent obtains it.

Coherence granularity is a cache line. If thread A increments `counterA` and thread B increments `counterB`, but both fields share one line, each write can invalidate the other core's copy. This is false sharing: logically independent variables generate physical contention. Padding or alignment can separate them, but increases memory footprint and may affect locality.

Coherence traffic is not free even when a value is correctly synchronized. A lock that bounces between cores causes ownership transfers; a producer-consumer queue may repeatedly move lines between private caches. A scalable algorithm reduces shared writes, batches communication, uses per-core data, or chooses a topology-aware layout.

The sequence uses a simplified two-core write-invalidate protocol. Both copies initially share the same clean line. A writer waits for the other copy to become invalid before modifying the line. `M` means unique modified ownership; `S` means shared; `I` means invalid. A line cannot simultaneously be `M` on one core and `S` on another in this model. Directory routing and transient states are omitted.



Transfer check. The two counters are distinct variables with one writer each. Why do the writes still exchange the line, and what changes if the counters occupy different lines?

The variables have independent writers, but a cache-line ownership request covers both words. On one shared line, each core must obtain the other core's latest copy before its own write. Separating the words onto different lines allows each core to retain ownership of its own line after warm-up. Padding changes physical sharing and footprint, not the synchronization rules of the programming language.

SIMD semantics

WARNING

Counterexample: vectorizing `for (i = 1; i < N; i++) a[i] = a[i - 1] + b[i]` with a 4-lane SIMD load of `a[i-1 .. i+2]` produces wrong results because each iteration depends on the previous iteration's updated `a[i - 1]` (a loop-carried RAW dependence of distance 1).

SIMD represents a vector as lanes of equal-width elements. A vector add performs lane-wise addition, optionally with masking, saturation, widening, or narrowing. The compiler can vectorize a loop when iterations are independent, memory accesses are regular enough, and the chosen operation has matching semantics.

For `for i: c[i]=a[i]+b[i]`, four-lane SIMD loads four values from each array, performs four additions, and stores four results. The scalar remainder handles a length not divisible by four unless the ISA provides masked operations. Alignment may be required or merely performance-relevant depending on the instruction. Aliasing can block vectorization: if `c` overlaps `a`, a vector store might overwrite values that later scalar iterations still need.

Floating-point vectorization can change results because reassociation is not generally valid under strict IEEE semantics. The sum `(a+b)+c` may round differently from `a+(b+c)`. A compiler needs permission from the language and numerical contract before reassociating reductions. Integer overflow has the same issue: modular vector addition may not match a checked scalar operation.

Branches and masks

EXAMPLE

In a 8-lane SIMD loop with predicate mask $m = \texttt{0b00101101}$, predicated execution evaluates the vector instruction across all lanes and suppresses register/memory writes where $m_k = 0$, eliminating branch misprediction penalties at the cost of executing wasted lane work when m is sparse.

Data-dependent branches reduce SIMD efficiency because lanes may require different control paths. Masked execution computes both or selectively disables lanes. If only one lane satisfies a condition, executing the operation for all lanes may still be cheaper than scalarizing, but the cost depends on operation latency and branch predictability.

Worked conditional example. To compute `max(a[i],0)`, a vector compare produces a mask, and a blend selects each lane from the input or zero. No branch is needed. For a rare expensive function `f(x)` taken by one lane, evaluating `f` for all lanes may be wasteful; compaction or scalar fallback may be better. The algorithmic condition is the same, but the hardware cost model differs.

Reduction and synchronization patterns

> **WARNING**
>
> Reassociating an IEEE 754 floating-point reduction from a sequential left-fold into a SIMD/multicore tree reduction changes rounding order and can alter low-order bits or overflow behavior; bit-reproducible parallel reductions require either a fixed tree topology independent of thread scheduling or an widened/exact accumulator.

A reduction such as summing an array has a dependency chain if written as one accumulator. SIMD uses partial sums in lanes, then combines them in a tree. A multicore reduction gives each thread a chunk, computes a local sum, and combines partials. The tree reduces span from $O(n)$ to $O(\log n)$ after parallel work, but floating-point order changes and synchronization/combination costs remain.

Reduction Step	Active Elements / Vector Lanes	Pairwise Computation	Cumulative Span (D)	Total Parallel Work (W)
Step 0	8 elements: $x0, x1, x2, x3, x4, x5, x6, x7$	Initial data distribution	0	0 operations
Step 1	4 pairs: $(x0 + x1), (x2 + x3), (x4 + x5), (x6 + x7)$	SIMD vector add (lane stride 1)	1 addition	4 operations
Step 2	2 pairs: $(s01 + s23), (s45 + s67)$	SIMD vector add (lane stride 2)	2 additions	6 operations
Step 3	1 final pair: $(s0123 + s4567)$	Scalar or horizontal reduction add	3 additions	7 operations

Barrier synchronization lets threads exchange a phase boundary: no thread proceeds until all arrive. Barriers are safe only when every participating thread executes the same barrier count under the same control conditions. A thread that skips a barrier can deadlock its peers. Fences order memory operations but do not by themselves wait for another thread; a barrier usually combines a control rendezvous with memory-order semantics.

Scheduling and load balance

DEFINITION

In the work-span task-graph model with total work $T_1 = W$ and critical-path span $T_\infty = D$, execution time T_P on P processors satisfies the lower bound $T_P \geq \max(W/P, D)$ and greedy-scheduler upper bound $T_P \leq W/P + D$.

Static partitioning assigns predictable chunks and has low overhead, but uneven work creates idle cores. Dynamic scheduling distributes tasks from a shared queue, improving balance at synchronization and contention cost. Work stealing lets idle workers take tasks from others, often scaling better for recursive irregular workloads but complicating locality and determinism.

Scheduling Strategy	Task Assignment Timing	Dispatch Overhead	Dynamic Imbalance Tolerance	Cache Locality Preservation
Static Block	Compile time / loop start	Minimal ($O(1)$ loop bounds arithmetic)	Poor (stragglers stall entire parallel phase)	High (contiguous chunks remain in local L1/L2 cache)
Dynamic Shared Queue	Runtime worker demand	High (atomic locks/CAS on centralized queue)	High (fast workers drain more tasks)	Poor (tasks ping-pong between core cache domains)
Guided Self-Scheduling	Dynamic decreasing chunks	Moderate (coarse initial chunks, fine tails)	Moderate-High (balances overhead vs imbalance)	Moderate (initial large chunks preserve spatial locality)
Work Stealing	Thread-local deques + steal	Low on local work; contention only when idle	Exceptional (handles irregular recursive trees)	High (threads work locally until starvation occurs)

The critical path or span limits parallel speedup. If a task graph has total work W and span D, an ideal lower bound with P processors is at least $\max(W/P, D)$; scheduling overhead adds more. Amdahl's serial fraction is one coarse view of span; a task graph exposes parallelism at finer granularity.

Memory bandwidth and locality

EXAMPLE

For STREAM triad `A[i] = B[i] + s * C[i]` on 8-byte doubles (24 bytes moved per 2 FLOPs, or 32 bytes with write-allocate), arithmetic intensity is at most $2/24 = 0.083\ \mathrm{FLOP/byte}$. On a socket with $40\ \mathrm{GB/s}$ DRAM bandwidth, throughput is capped at $3.3\ \mathrm{GFLOP/s}$ even if 16 cores provide $500\ \mathrm{GFLOP/s}$ of vector compute.

Adding cores can saturate shared memory bandwidth before compute units are full. A streaming loop may already approach the bandwidth ceiling on one or a few cores. Tiling, reuse, compression, and reducing bytes per result can improve scaling more than adding cores. NUMA systems make locality explicit: memory access cost depends on which socket owns the page, so first-touch allocation and thread placement matter.

SIMD and multicore often amplify cache concerns. A four-lane vector accesses 16 bytes or more per instruction; multiple cores can evict each other's lines or saturate a shared cache. Performance analysis should measure bandwidth, cache misses, coherence traffic, and utilization rather than infer scalability from core count.

Verification and determinism

> **REMARK**
>
> Parallel verification must establish data-race freedom under the target memory model (every conflicting access pair ordered by happens-before) separately from numerical determinism across thread counts and SIMD widths.

Parallel programs may be functionally nondeterministic even when each execution is legal. A race-free reduction with associative integer addition is deterministic in value; floating-point reduction may vary by order. Testing one schedule is insufficient. Stress tests vary thread placement, delays, input sizes, and scheduling; race detectors and model checkers can expose missing synchronization but depend on coverage and memory-model assumptions.

The implementation must prove ownership and lifetime. A thread accessing a shared object needs a lock, atomic protocol, immutable snapshot, or documented data-race-free ownership transfer. A vector load needs bounds and alias assumptions. A DMA or device buffer adds another agent to the coherence and lifetime proof.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.3.1: Amdahl's Law Scaling Limits

Verification application

Apply Amdahl's law to a 90% parallel workload on 4, 16, and unlimited cores.

Apply $S = 1/((1-p) + p/N)$ with parallel fraction $p = 0.90$ across core counts $N \in \{4, 16, \infty\}$.

With $s = 0.1$, speedups are $1/(0.1 + 0.9/4) = 3.08, 1/(0.1 + 0.9/16) = 6.4$, and limit 10. These are ideal fixed-work results; synchronization and imbalance reduce them.

Exercise 4.3.2: Acquire–Release Synchronization Protocol

Construction application

Design a release/acquire message-passing protocol and explain why relaxed operations are insufficient.

Contrast release stores and acquire loads with relaxed atomic operations for publishing shared data payloads.

The producer performs ordinary payload stores followed by a release store to `ready`; the consumer performs an acquire load and then reads the payload. The acquire/release pair establishes ordering. Relaxed operations provide atomicity for each location but not the cross-location publication guarantee.

Exercise 4.3.3: False Sharing Elimination in Thread-Local Counters

Description transfer

Identify false sharing in a per-thread counter array and propose two layouts.

Analyze how multiple cores updating adjacent counter array elements in the same cache line cause invalidation ping-pong.

Place counters in separate cache lines with padding or use per-core arrays aligned to line boundaries; alternatively aggregate less often. Padding costs memory, while frequent aggregation costs synchronization.

Exercise 4.3.4: SIMD Conditional Selection and Tail Handling

Construction transfer

Vectorize a conditional clamp and handle a non-multiple-of-lane-count length.

Use vector comparison masks or blend instructions for branches, and consider scalar cleanup loops for residual elements.

Use a compare mask and blend input with zero, then process the scalar tail or use a masked vector operation. Verify the length and avoid out-of-bounds masked loads under the ISA's mask semantics.

Exercise 4.3.5: Floating-Point Non-Associativity in Parallel Reduction

Description transfer

Analyze a floating-point reduction under different tree shapes and state the reproducibility trade-off.

Explain how changing reduction tree geometry or thread counts alters rounding order and numerical results.

A tree changes addition order and therefore rounding. Pairwise trees reduce error growth compared with a serial chain but may produce different last bits across thread counts. Reproducibility requires a fixed reduction tree or compensated/accurate summation policy.

Exercise 4.3.6: Work-Span Analysis and Speedup Bounds

Verification synthesis

Use work/span bounds to evaluate a task graph with 1,000 units of work and span 40 on 32 processors.

Calculate $T_p \geq \max(T_1/p, T_\infty)$ using total work $T_1 = 1000$ and critical-path span $T_\infty = 40$ on $p = 32$ cores.

Work bound is $1000/32 = 31.25$ units and span bound is 40, so ideal time is at least 40 units. More processors cannot beat the critical path; scheduling overhead and dependencies make actual time larger.

Exercise 4.3.7: Multicore Scaling Under Memory Bandwidth Saturation

Description synthesis

Explain why doubling cores may not improve a bandwidth-bound kernel.

Consider what happens when aggregate off-chip memory traffic from all active cores meets the memory bus saturation ceiling.

Once memory bandwidth is saturated, extra cores compete for the same bytes. Improving locality, reducing traffic, increasing reuse, or using a higher-bandwidth memory system is required before additional arithmetic resources help.

References & Further Reading

- MIT 6.004 Computation Structures: parallel-processing notes
- UC Berkeley CS 61C: multicore execution model
- MIT 6.004 learning objectives

SECTION DETAIL

Performance Investigation

This laboratory treats a performance claim as a falsifiable engineering hypothesis. The subject is two implementations of a matrix kernel: a naive traversal and a tiled/vectorized implementation. The aim is not to produce a chart that flatters an optimization. It is to determine which resource limits each version, quantify uncertainty, and identify the input boundary where the conclusion changes.

Question and model

DEFINITION

A falsifiable performance hypothesis specifies (1) the workload regime W, (2) the mechanism intervention T relative to baseline B, (3) the intermediate hardware event counter E (such as L2 misses per element) predicted to change, and (4) the outcome that would reject the mechanism explanation.

Suppose the kernel computes $C = A + B$ for square matrices of 32-bit values. A naive row-major implementation reads two values and writes one for each element. A column-major or poorly laid-out version may touch a new cache line for each access. The hypothesis is: "For matrices larger than the last-level cache, tiled row-major traversal reduces time by improving spatial and temporal locality, and the improvement will correlate with lower cache-miss traffic rather than merely higher clock frequency."

The hypothesis predicts several observations: cache misses per instruction should decrease, memory bandwidth may become more efficient, instructions per element may change modestly, and the effect should grow when dimensions exceed cache capacity but weaken when the entire matrix fits in a higher cache. If time improves while misses and bandwidth do not change, the proposed mechanism is incomplete; perhaps vector width, compiler unrolling, or frequency changed instead.

MEASURE BEFORE TUNING

What is the first step in a defensible performance investigation?

Answer: State a workload and hypothesis with a measurable metric

A workload and hypothesis define what evidence could support or falsify the diagnosis. Without them, a benchmark is an anecdote.

Experimental controls

Fix element type, matrix dimensions, initialization, compiler version, optimization flags, thread count, CPU affinity, frequency policy where possible, and correctness check. Use the same memory alignment and allocation method. Randomize or carefully define matrix contents so dead-code elimination cannot remove the computation. Make the output observable, for example by checking a checksum after timing, but exclude checksum time from the kernel measurement unless end-to-end time is the object of study.

Warm-up matters. The first iteration may include page faults, instruction-cache fills, dynamic linking, frequency ramp, and allocator effects. Decide whether the target is cold-start latency or steady-state throughput. For steady state, run warm-up iterations and report that policy. For cold start, reset or isolate state rather than silently discarding the first observation.

Repeat enough times to observe scheduling and thermal variation. Report median, interquartile range, and high percentiles; show raw repetitions when sample count is small. A single fastest run is a lower bound on incidental interference, not an estimate of typical performance.

ONE WARM RUN IS NOT A RESULT

One warm benchmark run places design A eight percent ahead of design B. The report attributes the difference to A's cache policy without collecting cache events.

What evidence is needed before accepting that causal claim?

Reference decision: Repeated controlled trials plus cache measurements predicted by the policy model

Performance is an empirical claim. A model predicts the mechanism, while controls and repeated measurements distinguish that mechanism from cache state, thermal drift, scheduling, or workload bias.

Measurement boundaries

WARNING

Reading a timestamp counter (`rdcycle` / `rdtsc`) without compiler and pipeline serialization barriers allows out-of-order execution to sample the stop timestamp before slow memory stores or loop iterations have retired.

Use a monotonic high-resolution timer or hardware cycle counter with a documented serialization method. The start must not be moved after work by compiler or CPU reordering; the end must not be read before stores complete if the result includes stores. A barrier or fence may be needed around timing, but its cost must be excluded or included consistently.

Measurement Phase	Executed Operation	Boundary Invariant	Serialization Primitive Required
Preamble Setup	Buffer allocation & pre-touch	Excluded from timed kernel	Compiler barrier + full memory fence
Start Timestamp	Sample cycle counter (T_{start})	No kernel instruction executes before timestamp read	`cpuid` / `fence rw, rw` before counter read
Kernel Execution	Compute loop / memory accesses	Isolated from timing calls and verification logic	Unmodified compiler-generated sequence
Stop Timestamp	Sample cycle counter (T_{stop})	All stores and loop iterations retire before read	`fence rw, rw` + pipeline serialize before counter read
Post-run Verify	Compute output checksum	Excluded from reported execution cycles	Bitwise assertion against reference oracle

Measure at least wall-clock time, cycles, retired instructions, cache references/misses at relevant levels, branch instructions/misses, and memory bandwidth if counters are available. Record CPU model, core, frequency, operating system, compiler, input dimensions, alignment, and thermal state. Hardware counter names and sampling may differ; report their interpretation rather than treating a counter label as a universal physical quantity.

Workload matrix

EXAMPLE

Worked calculation: for three `float32` matrices A, B, C of dimension $N \times N$, the total active footprint is $3 \times 4N^2 = 12N^2$ bytes. Choosing $N = 32$ ($12\,\mathrm{KiB}$, fits in 32 KiB L1), $N = 128$ ($192\,\mathrm{KiB}$, exceeds L1 but fits in 256 KiB L2), and $N = 2048$ ($48\,\mathrm{MiB}$, exceeds LLC) isolates each level of the memory hierarchy.

Use dimensions that cross the cache hierarchy: small enough to fit L1, larger than L1 but within L2, larger than L2, and larger than the shared cache or memory capacity available to the process. Include non-square or leading-dimension cases if the implementation claims generality. Test dimensions aligned and misaligned to vector width and cache-line size. For tiled code, include tile sizes smaller than, near to, and larger than the candidate cache footprint.

For each dimension, compute the expected bytes of logical input and output. A simple lower bound for $C = A + B$ is 12 bytes per element if every value is read/written once, though write allocation and line transfers can add traffic. Compare measured bandwidth with the hardware's sustainable stream bandwidth rather than its advertised peak. If the naive version is far below the ceiling due to poor locality, tiling may improve effective bandwidth without changing the physical DRAM limit.

Statistical interpretation

REMARK

Always pair central tendency (median cycles/element) with dispersion (interquartile range) and tail behavior (p99), and evaluate paired trial differences $(T_{i,B} - T_{i,A})$ across randomized block order rather than comparing single best runs.

Plot normalized time per element or cycles per element against matrix size. A cache-capacity transition should appear as a change in slope or a step when the active working set exceeds a level, though prefetchers and replacement can smooth it. Plot cache misses per element and bandwidth alongside time. Use paired runs where both implementations execute under similar system conditions, but avoid alternating patterns that cause one version to inherit the other's cache state unless that is the production scenario.

Compute effect size and uncertainty. If implementation A takes 10.0 ± 0.2 ms and B takes 9.8 ± 0.2 ms with strongly overlapping paired variation, the 2% improvement needs more evidence. If B takes 7.0 ± 0.1 ms across stable repetitions, the mechanism and practical significance are clearer. Do not call a result "statistically significant" without stating the test and independence assumptions.

Confounders

WARNING

If the compiler auto-vectorizes the tiled loop with 8-wide SIMD while leaving the naive loop scalar, a 3× speedup may reflect instruction-count reduction rather than cache locality; always inspect disassembled loop bodies and retired instruction counts alongside cache-miss counters.

Compiler vectorization can turn both source variants into similar machine code. Inspect generated assembly or compiler optimization reports. A different allocation can change page coloring and TLB behavior. Transparent huge pages can change translation reach. Frequency scaling can favor the version that keeps the core busy differently. NUMA placement can dominate large matrices. Background processes, interrupts, thermal throttling, garbage collection in a host runtime, and page faults can create outliers.

Confounder Source	Mechanism of Distortion	Controlled Experimental Protocol	Verification Metric / Evidence
DVFS Frequency Scaling	Turbo Boost alters clock rate between runs	Pin governor to `performance`; fix CPU frequency	Measure CPU frequency via PMU `APERF` / `MPERF`
Core Migration & Context Switches	Thread moves across cores, losing L1/L2 cache	Pin benchmark process to isolated physical core (`taskset`)	Read OS context-switch counters (`getrusage`)
Compiler Vectorization Asymmetry	One variant vectorizes while other stays scalar	Compile with `-fopt-info-vec` and inspect assembly	Compare retired instruction counts and vector instructions
Page Faults & Cold Allocation	First-touch allocation costs page faults in run	Run untimed warm-up passes to populate page tables	Assert zero minor/major page faults during timing
NUMA Remote Memory Access	Buffers allocated on remote socket have $3\times$ latency	Bind memory allocation to local NUMA node (`numactl`)	Read uncore remote memory access counters

A benchmark that fixes all confounders may no longer represent deployment. Therefore use two phases: a controlled mechanism experiment and a representative end-to-end experiment. The former supports causal understanding; the latter tests product value. Do not use the representative result to claim a cache mechanism without counters, or the controlled result to claim universal application speed.

Correctness and adversarial cases

> **WARNING**
>
> Never report timing from a kernel whose output has not been verified against an independent reference oracle across non-multiple-of-tile dimensions ($N = 1023$), misaligned base pointers, and aliased inputs (`C == A`).

Compare outputs against a high-precision or reference implementation on random values, zeros, maximum and minimum representable values, NaNs if floating point is used, and dimensions that exercise tails. Vectorized code can read beyond logical bounds if masked loads are misunderstood. Tiled code can omit fringe rows or reuse a stale tile. A performance result from incorrect output is not a performance result.

Adversarial Test Vector	Input Parameter Structure	Targeted Failure Mode	Pass Invariant
Non-Multiple Tile Dimensions	Matrix size $N = 1023$ (tile $B = 32$)	Fringe tile loop boundary omission or out-of-bounds write	Output matches reference for all N^2 elements
Pointer Misalignment	Base pointer offset by 1 byte (`ptr + 1`)	Unaligned SIMD load trap or silent fault	Kernel completes without bus error; results identical
Input/Output Aliasing	In-place update (`C == A`)	RAW hazard clobbers inputs before subsequent tap reads	Prohibited by assertion or handled via staging buffer
Adversarial Stride Thrashing	Row stride equals exact power of two cache set span	Direct-mapped / low-associativity conflict thrashing	Performance degradation diagnosed via L1/L2 miss counters

Test aliasing and alignment assumptions. If the API permits overlapping input/output, a transformation that assumes no alias may change semantics. If aligned vector loads are used, test misaligned input and either reject it explicitly or provide a safe path. Document whether the compiler is allowed to reassociate floating-point operations; bitwise differences may be acceptable only under a stated numerical tolerance.

Worked investigation plan

Run four matrix sizes: 64×64, 256×256, 1024×1024, and 4096×4096, with three tile sizes and both implementations. Warm up five times, measure ten repetitions per cell, pin to one core, and record cycles, instructions, L1/L2/LLC misses, branch misses, and memory bytes. For each cell, verify a checksum and report median cycles per element and p99.

If tiling helps only at 1024 and 4096 while L2/LLC misses per element fall, the cache hypothesis gains support. If it helps at all sizes with unchanged misses but lower instructions, vectorization or loop overhead is likely. If p50 improves but p99 worsens, investigate preemption, memory contention, or tile-size-dependent allocation. If the compiler emits identical loops, the source-level optimization has no measured implementation effect and the hypothesis must be revised.

Worked example: paired-trial evidence calculation. Consider this deliberately small fixture, where each row is one paired trial and the cache counter is normalized per element:

size	variant	median cycles/element	L2 misses/element	p99 cycles/element
1024^2	naive	8.0	0.22	10.1
1024^2	tiled	6.4	0.11	7.0
64^2	naive	2.1	0.01	2.3
64^2	tiled	2.0	0.01	2.2

At 1024^2, tiling reduces median time by $(8.0 - 6.4)/8.0 = 20\%$ and L2 misses by 50%; at 64^2 it changes time by only 4.8% with no miss reduction. This pattern supports the stated cache hypothesis only for the larger working set. It does not prove causation: vector instructions, frequency, and allocation must still be controlled or measured. The table is a shipped deterministic teaching fixture, not a claim about measured hardware.

REPRODUCE THE FIXTURE'S EFFECT SIZES

```
rows = [(8.0, 0.22, 6.4, 0.11), (2.1, 0.01, 2.0, 0.01)]
for naive_time, naive_miss, tiled_time, tiled_miss in rows:
  time_change = (naive_time - tiled_time) / naive_time
  miss_change = (naive_miss - tiled_miss) / naive_miss if naive_miss else 0.0
  print(f"time reduction={time_change:.1%}, L2-miss reduction={miss_change:.1%}")
```

EXPECTED OUTPUT

```
time reduction=20.0%, L2-miss reduction=50.0%
time reduction=4.8%, L2-miss reduction=0.0%
```

The runner computes only the supplied fixture; it is not a hardware benchmark and cannot establish cache causality. A learner's report must separate this deterministic arithmetic from evidence collected on a real target.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.4.1: Cache Optimization Hypothesis Formulation

Construction application

Write a complete hypothesis and prediction table for a cache optimization.

Specify the concrete physical mechanism, input size threshold relative to cache capacities, and quantitative observables (miss rates, cycles per element).

A good hypothesis names a mechanism, input regime, and predicted observable. “Tiling is faster” is weak; “for working sets exceeding L2, tile reuse lowers L2 misses per element and therefore cycles per element” is testable.

Exercise 4.4.2: Working-Set Dimension Selection Across Cache Tiers

Construction application

Choose dimensions that cross three cache levels and justify them from capacity.

Calculate active working set bytes for input and output matrices against L1, L2, and LLC capacity boundaries.

Compute bytes per active working set and choose dimensions just below and above L1, L2, and LLC capacities, then include larger cases. State whether matrices are shared or private and whether all three arrays fit simultaneously.

Exercise 4.4.3: Measurement Boundary Design and Store Ordering

Construction transfer

Design a timing boundary that includes stores correctly without measuring result verification.

Place monotonic timer reads and memory/compiler fences so all stores commit without timing output validation.

Use a monotonic timer with compiler/CPU ordering barriers, start immediately before the kernel, end after the final stores are ordered, and verify output afterward. If fences are inside the measured boundary, include their cost consistently in both variants.

Exercise 4.4.4: Experimental Controls for Benchmark Confounders

Description transfer

List six confounders and a control or measurement for each.

Identify system and environment variables (frequency governors, vectorization, NUMA placement, cache warm-up, background noise) and define isolation controls.

Examples: compiler vectorization—inspect assembly; frequency scaling—pin and log frequency; NUMA—bind memory; page faults—pre-touch or measure cold start separately; alignment—control allocation; interference—isolated core and repeated runs.

Exercise 4.4.5: Disentangling Instruction Count from Cache Effects

Description transfer

Interpret time improving with unchanged cache misses but lower instruction count.

Consider whether the compiler vectorized loops, unrolled iterations, or eliminated scalar operations without altering memory traffic.

The improvement may come from fewer instructions, better issue utilization, or vectorization rather than cache behavior. Measure vector instructions, CPI, and generated code before attributing causality.

Exercise 4.4.6: Vector Loop Bounds and Buffer Aliasing Validation

Construction synthesis

Design correctness tests for vector tails and overlapping buffers.

Test boundary vector lengths (0, 1, vector lane width ± 1) and distinct spatial overlap relationships against a scalar reference.

Test lengths 0, 1, lane-1, lane, lane+1, and large nonmultiples; compare against a scalar reference. For overlap, test exact alias, forward overlap, backward overlap, and disjoint buffers, or document that overlap is forbidden.

Exercise 4.4.7: Peak Bandwidth vs Random Memory Access Limits

Description synthesis

Explain why a peak-bandwidth comparison is insufficient for a random-access kernel.

Contrast streaming hardware prefetcher bandwidth with pointer-chasing latency, TLB reach, and memory-level parallelism limits.

Peak bandwidth assumes large sequential transfers and ideal concurrency. Random access has latency, poor locality, TLB misses, and limited memory-level parallelism; sustainable bandwidth and latency distribution are the relevant measures.

References & Further Reading

- MIT 6.004 Computation Structures: cost/performance notes
- UC Berkeley CS 61C: memory hierarchy and AMAT

Synthesis and capstone

SECTION DETAIL

Binary Design Lab

This lab turns representation theory into an interface that another engineer could implement. Design a 16-byte binary record for a sensor sample and a parser that is safe under malformed input. The record must contain a version, flags, signed temperature, unsigned pressure, timestamp, sequence number, and checksum. You must specify bit widths, byte order, alignment, valid ranges, reserved encodings, checksum coverage, and behavior when the record is truncated or inconsistent.

Specification before encoding

Choose a wire format rather than inheriting the host machine's struct layout. Host padding, compiler ABI, endianness, and alignment are not stable interchange contracts. Let byte 0 contain an 8-bit version, byte 1 flags, bytes 2–3 a signed 16-bit temperature in hundredths of a degree, bytes 4–7 an unsigned pressure in pascals, bytes 8–11 a 32-bit timestamp, bytes 12–13 a 16-bit sequence, and bytes 14–15 a CRC or checksum. State that all multibyte fields use big-endian order.

The version field supports evolution, but a parser must define whether it accepts unknown versions. Flags need a mask of defined bits; reserved bits should be zero in canonical encodings and rejected or ignored according to policy. Temperature has a signed range $[-327.68, 327.67]$ degrees if represented in hundredths. Pressure's valid physical range may be narrower than the unsigned bit range; validation belongs above bit decoding and should not be conflated with syntax validity.

> ***Pause & Predict: The Packed Struct Portability Trap.*** *Why should systems engineers never parse incoming network or sensor byte streams using a C struct cast like* `struct Sensor* s = (struct Sensor*)buf;` *even with* `#pragma pack(1)`*? Consider CPU alignment traps, endianness, and aliasing rules before opening the analysis below.*

Consider what happens when an ARM Cortex-M0 accesses a 32-bit integer at an odd address, or when an x86 host decodes a big-endian network packet.

Analysis: Directly overlaying a C struct onto a byte stream introduces three fatal hazards:

1. **Hardware Alignment Traps**: On architectures requiring strict natural alignment (such as ARMv6-M Cortex-M0 or older SPARC/MIPS cores), reading a 32-bit field (`uint32_t pressure`) from an unaligned offset generates an immediate **HardFault / Bus Error** hardware trap exception.
2. **Endianness Corruption**: A multi-byte integer serialized in network big-endian order (`0x00018BCD`) read on a little-endian host (such as x86-64 or ARM64) is silently interpreted in reverse as `0xCD8B0100` ($3{,}448{,}439{,}040$ Pa instead of $101{,}325$ Pa).
3. **Compiler Undefined Behavior**: The C standard prohibits aliasing arbitrary byte buffers through incompatible struct pointers (the strict aliasing rule). Compilers may optimize out subsequent memory re-reads entirely.

Rule: Always deserialize byte streams by explicitly shifting and masking individual octets (`val = (buf[0] << 24) | (buf[1] << 16) | ...`).

SPECIFY BEFORE ENCODE

Why must a binary wire format be formally specified before implementing bit-manipulation and serialization routines?

Answer: Field widths, endianness, and reserved values establish the semantic contract that parser routines must preserve.; Formal field constraints define unambiguous rejection boundaries for malformed, truncated, or hostile inputs.

A bit pattern has meaning only through a format contract. Writing code without a specification introduces boundary ambiguities and vulnerability to malformed input.

Worked encoding

Suppose version is 1, flags are `0b00000101`, temperature is -12.34°C, pressure is 101325 Pa, timestamp is 1,700,000,000, and sequence is 42. Temperature scales to -1234. Its 16-bit two's-complement encoding is $65536 - 1234 = 64302 = 0xFB2E$. Big-endian bytes are `FB 2E`. Pressure is `0x00018BCD`; timestamp and sequence are encoded similarly. The parser must sign-extend `0xFB2E` before dividing by 100; interpreting it as unsigned produces 643.02 and a physically absurd but representable value.

The checksum must cover a defined byte range and have a defined initial value, polynomial or addition rule, reflection policy, and byte order. A simple sum can detect many accidental errors but is weak against rearrangements and deliberate changes. A CRC gives stronger burst-error detection under its specified polynomial, but it is not authentication. If the record crosses an adversarial boundary, add a cryptographic integrity mechanism rather than treating CRC as a security control.

VERIFY BEFORE MINIMIZING

A lab minimizes a four-input control function directly from prose. The resulting circuit is small, but one reserved input combination unexpectedly asserts write-enable.

Evaluate which engineering design artifacts are essential before performing logic minimization on control circuits:

Reference decision: A complete truth table specifying explicit outputs for reserved and unassigned inputs prevents accidental control assertions.

Minimization preserves the function it is given. Complete truth tables and deliberate don't-care assignments define safe behavior across all input combinations.

Parser state and failure behavior

A safe parser first checks that at least 16 bytes are available before reading any field. It then decodes into a temporary typed structure, checks version and reserved bits, verifies checksum before trusting payload, validates scaled ranges and sequence policy, and only then publishes the sample. If input may be streamed, use a state machine that accumulates exactly the required header and body bytes; never assume one read returns one record.

The order is a security and correctness boundary. Checking physical range before checking that the bytes belong to a complete record can read beyond the buffer. Publishing temperature before checksum verification can expose attacker-controlled data to control logic. Updating the last-seen sequence before all validation succeeds can permanently reject a later valid record. Use a temporary object and a commit point: prior to commit, parsing is private speculation; after commit, every field satisfies the record invariant.

Failure categories should be distinguishable: truncated input, unsupported version, invalid reserved bits, checksum failure, range violation, and replayed sequence. A caller may choose to drop a corrupt sample but should not treat a checksum failure as a valid zero measurement. Error paths must not leave partially updated shared state.

Worked malformed cases. A 15-byte buffer must be rejected before the parser reads the checksum. A record with version 2 may be rejected or dispatched to a versioned parser, but the default parser must not interpret new fields using old offsets. A valid checksum over a temperature

outside the physical sensor range is syntactically valid but semantically invalid. A sequence lower than the last accepted value may be stale, wrapped, or out of order; the protocol must state which.

Implementation and portability

Implement field access with explicit shifts and masks or byte reads, not a packed host struct whose layout is compiler-dependent. Check arithmetic before conversion: a signed input converted to unsigned can become large, and scaling a value can overflow an intermediate type even when the final physical range would fit. Use a wider intermediate and checked multiplication/addition.

The encoder should be the inverse of the decoder on canonical values. Property-based tests can generate valid records, encode and decode them, and assert equality. Negative tests mutate one bit, truncate each possible length, set every reserved bit, and exercise minimum/maximum values. A parser that passes only a hand-written happy-path sample has not established a robust representation contract.

Worked Trace: 16-Byte Sensor Frame Serialization & Decoupled Verification

Let us construct a step-by-step serialization and parser trace for a complete 16-byte sensor telemetry frame:

Input Data Specification:

- Version = 1 (`0x01`, 1 byte)
- Flags = `0b00000101` (`0x05`, active + calibrated bits, 1 byte)
- Temperature = $-12.34\,^{\circ}\mathrm{C}$ (signed 16-bit fixed point, $100\times$ scale)
  - Scaled integer: -1234.
  - 16-bit two's complement: $65536 - 1234 = 64302 = 0\mathrm{xFB2E}$.
- Pressure = $101325\ \mathrm{Pa}$ (unsigned 32-bit integer)
  - Hexadecimal: `0x0001_8BCD` (4 bytes).
- Timestamp = $1{,}700{,}000{,}000$ Unix epoch seconds (unsigned 32-bit integer)
  - Hexadecimal: `0x6554_F100` (4 bytes).
- Sequence ID = 42 (unsigned 16-bit integer)
  - Hexadecimal: `0x002A` (2 bytes).

Step 1: Byte Layout & Big-Endian Octet Placement

```
Byte [00]: 0x01                  (Version = 1)
Byte [01]: 0x05                  (Flags = 0b00000101)
Byte [02..03]: 0xFB, 0x2E        (Temp: high byte 0xFB, low byte 0x2E)
Byte [04..07]: 0x00, 0x01, 0x8B, 0xCD (Pressure: 0x00018BCD)
Byte [08..11]: 0x65, 0x54, 0xF1, 0x00 (Timestamp: 0x6554F100)
Byte [12..13]: 0x00, 0x2A        (Sequence: 0x002A)
```

Step 2: Checksum / CRC-16 Calculation (Bytes 0–13) Using CRC-16-CCITT (polynomial $x^{16} + x^{12} + x^5 + 1 = 0\text{x}1021$, init `0xFFFF`):

- Processing the 14 payload bytes yields CRC value `0x9D47`.
- Byte [14..15]: `0x9D, 0x47`.
- Full wire frame: `01 05 FB 2E 00 01 8B CD 65 54 F1 00 00 2A 9D 47` (exactly 16 bytes).

Step 3: Invariant Verification during Decoding

1. Length Check: $16 \geq 16$ bytes present.
2. Checksum Verification: Compute CRC over bytes 0..13, compare against `(buf[14] << 8) | buf[15]`. Matches `0x9D47`.
3. Safe Fixed-Point Reconstruction:

```
int16_t raw_temp = (int16_t)((buf[2] << 8) | buf[3]); // Preserves
signedness: 0xFB2E -> -1234
double temp_c = (double)raw_temp / 100.0;             // Yields
exact -12.34
```

Counterexample & Failure Mode: The Fixed-Point Unsigned Sign-Extension Disaster

What occurs when a decoder misclassifies the signedness of a fixed-point telemetry stream? Consider an environmental sensor deployed in an Antarctic research station measuring $-25.00\,^{\circ}\text{C}$.

1. **The Correct Wire Pattern**:
   - Scaled value: -2500.
   - Two's complement 16-bit encoding: $65536 - 2500 = 63036 = 0\text{xF63C}$.
   - Wire bytes: `F6 3C`.

2. **The Naive Parser Flaw**: Suppose the receiving firmware decodes the bytes as an unsigned integer before scaling:

```
uint16_t raw = (uint16_t)((buf[2] << 8) | buf[3]); // Evaluates to
63036
float temp = raw / 100.0f;                          // Computes 63036
/ 100.0 = +630.36 C!
```

3. **Systemic Failure Mode**:

- Instead of $-25^{\circ}\mathrm{C}$, the flight computer or environmental monitor reads $+630.36^{\circ}\mathrm{C}$ (hotter than molten lead).
- Emergency cooling protocols trigger, fire suppressants discharge, or heaters shut off in sub-zero conditions, destroying equipment.
- **Root Cause**: The software treated bitfield syntax as an unsigned scalar rather than enforcing the two's complement arithmetic invariant defined in the interface contract.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.1.1: Byte-Offset Specification and Alignment Table

Construction application

Produce a byte-offset specification table for a 16-byte telemetry record with explicit field names, widths, signedness, integer scaling factors, valid ranges, and endianness, without relying on compiler-dependent struct padding.

Specify explicit bit/byte widths, signedness, valid numerical ranges, and endianness for every field.

A complete table names every byte and makes no reliance on host padding. The signed temperature field must explicitly specify two's complement before scaling; pressure and timestamps must state whether all bit patterns are syntactically valid.

Exercise 5.1.2: Two's-Complement Signed Fixed-Point Encoding

Construction application

Derive the encoded byte sequences for five representative sensor samples, including minimum and maximum representable temperatures and negative scaling boundaries.

Convert negative scaled values using modular complement arithmetic before emitting high-order bytes.

Encode negative scaled temperature as $2^{16} - |x|$ and emit high byte first. Check the result by decoding the emitted bytes and comparing the integer scaled value before converting to floating point.

Exercise 5.1.3: Streaming Parser State Machine and Buffer Framing

Description application

Specify a streaming parser state machine with explicit framing states (header, payload, checksum) and deterministic error rejection paths for partial reads.

Define explicit states (header, payload, checksum) and deterministic error rejection paths.

States should collect header/body bytes, then checksum, then validate and publish. Each read may return fewer bytes than requested. On any error, discard the current record and do not retain partial data as a future header unless the framing protocol says how resynchronization works.

Exercise 5.1.4: CRC Error Detection and Bit-Flip Sensitivity

Verification application

Define a 16-bit CRC polynomial and demonstrate mathematical detection of single-bit corruption and two-byte transpositions over the telemetry frame.

Specify polynomial, initial value, and bit coverage to detect single-bit flips and transpositions.

A CRC specification must include polynomial, initial value, reflection, final XOR, and coverage. Demonstrate the computed checksum before and after mutation; do not claim security or collision resistance from an unspecified sum.

Exercise 5.1.5: Version-2 Extension and Backward Compatibility

Construction transfer

Design a version-2 record extension that enables legacy version-1 parsers to safely reject newer frames while version-2 parsers decode both generations.

Ensure version-1 parsers reject newer versions cleanly while version-2 parsers can decode legacy frames.

Put new records under version 2 and keep byte offsets unambiguous. A version-1 parser should reject version 2 before interpreting payload, while a version-2 parser can decode a shared prefix and additional fields.

Exercise 5.1.6: Encode–Decode Bijective Round-Trip Invariant Proof

Proof synthesis

Provide a formal proof sketch showing that decoding an encoded canonical sample $D(E(x))$ bijectively recovers x for all valid domain values.

Show field-level injectivity, exact integer scaling preservation, and invertibility of byte reversal.

Show each field's encoding is injective over its valid range, byte reversal is undone by decoding, sign extension inverts two's-complement encoding, and scaling uses an exact integer representation. Canonicality excludes alternate encodings.

Exercise 5.1.7: Hostile Byte Stream Mutation and Fuzzing Matrix

Generalisation transfer

Construct an adversarial test matrix containing truncated frames, reserved bit corruptions, out-of-range sensor readings, and invalid checksums, stating expected parser outcomes.

Include zero-length frames, partial truncations, reserved-field mutations, and out-of-range sensor values.

Include length 0, every length 1–15, exact 16, extra trailing bytes, all-zero/all-one values, signed boundaries, each reserved bit, one-bit and burst mutations, stale sequence, and physical limits. Expected outcomes must distinguish reject, accept, and accept-with-warning.

Proof obligations

Let E be the encoder and D the decoder. For every canonical sample x in the declared domain, the primary correctness property is $D(E(x)) = x$.

This requires more than matching a few examples: each field encoding must be injective, scaling must use an exact integer representation, byte-order reversal must be involutive, and the checksum computed by the decoder must equal the encoder's checksum over the same bytes.

A second property is rejection soundness: if a byte string fails length, version, reserved-bit, checksum, or range validity, the decoder must not publish a sample. A third is framing progress: after rejecting a record, the streaming parser must either discard exactly the documented frame or resynchronize according to an explicit delimiter rule.

Prove the properties compositionally. First prove field encode/decode lemmas, then prove concatenation preserves offsets, then prove checksum agreement, and finally prove the parser's commit condition. This mirrors hardware refinement: a large interface becomes manageable when each boundary has a local invariant and the composition preserves it.

Assessment standard

An excellent lab report contains a complete byte-level specification, integer derivations, pseudocode with failure states, generated examples, a test matrix, and proofs tied to stated assumptions. It explains why a packed C struct is insufficient, why CRC is not authentication, why a checksum can be valid for semantically invalid data, and how a future version remains unambiguous. A report that only prints a hexadecimal sample has demonstrated encoding practice but not architectural competence.

An independent engineer or automated test harness should be able to implement the encoder and parser independently from the report and obtain identical bytes for every canonical example.

References & Further Reading

- IEEE Computer Society. (2019). *IEEE Standard for Floating-Point Arithmetic* (IEEE Std 754–2019). IEEE.
- Patterson, D. A., & Hennessy, J. L. (2020). *Computer Organization and Design: The Hardware/Software Interface (RISC-V Edition)*, Chapter 2 & 3. Morgan Kaufmann.
- Software Engineering Institute, Carnegie Mellon University. (2021). *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*. Addison-Wesley.

SECTION DETAIL

Architecture Capstone

This capstone integrates representation, instruction-set semantics, datapath/control, pipelining, memory hierarchy, I/O, performance modeling, and parallel execution. The problem is to design a processor-backed image-filter service that accepts frames from a device, applies a configurable convolution and color transform, and emits results under a latency and energy budget. The goal is not to choose a fashionable CPU feature. It is to make a defensible architecture argument in which every abstraction, optimization, and failure mode is explicit.

System specification

DEFINITION

The capstone service contract requires sustaining 30 frames/s of 1920×1080 four-channel 8-bit images with 99th-percentile end-to-end frame latency T_{99} $\leq 25\ \mathrm{ms}$ from frame availability to output completion while enforcing zero out-of-bounds DMA accesses and explicit overflow/saturation semantics.

Assume a sensor produces 1920×1080 frames at 30 frames per second. Pixels arrive as four 8-bit channels; the filter applies a 3×3 kernel to each channel, clamps results to 0–255, and writes an output frame. The system has a 2 GHz 64-bit RISC-style core, 32 KiB 4-way L1 data cache with 64-byte lines, 256 KiB private L2, shared DRAM bandwidth of 20 GB/s, and a DMA-capable input device. These are modeling assumptions, not facts about one real processor; change them if your design justifies it.

The service must sustain the input rate, keep 99th-percentile frame processing below 25 ms after the complete frame is available, reject malformed descriptors, and never allow the device to write outside an owned buffer. The output may be delayed by buffering, but the system must state its queue bound and behavior when input outruns processing. An energy-aware design should report memory traffic and active-core time rather than claim that SIMD is efficient.

START WITH THE BOTTLENECK

What should a capstone specification identify before choosing a cache or pipeline?

Answer: A workload, correctness contract, and measurable latency, throughput, or energy budget

Hardware choices are meaningful relative to a workload and contract. A budget turns a design narrative into a falsifiable engineering argument.

Baseline arithmetic model

A 3×3 convolution performs nine multiplies and eight additions per output channel if implemented directly. A 1920×1080 frame contains 2,073,600 pixels; at four channels this is approximately 74.65 million multiplies and 66.36 million additions per frame, before boundary handling, loads, stores, clamping, address arithmetic, and loop control. At 30 frames per second the direct arithmetic demand is roughly 2.24 billion multiplies and 1.99 billion additions per second.

This count is a first estimate, not a timing result. If the kernel coefficients are fixed, multiplication may become shifts/adds or vector instructions. If the image is tiled, input pixels can be reused across neighboring outputs. If the filter is separable, a 3×3 kernel may be expressed as two 1×3 passes, reducing arithmetic at the cost of an intermediate buffer and additional traffic. A complete proposal compares these transformations against cache and bandwidth effects.

Worked arithmetic-intensity estimate. A naive output pixel reads 9 input pixels and writes one output pixel. At four bytes per pixel, ignoring reuse, that is 40 bytes and roughly 68 arithmetic operations, giving 1.7 operations per byte. With 20 GB/s of DRAM bandwidth, the roofline bandwidth ceiling is about 34 GFLOP/s-equivalent operations per second under this simplified count. Tiling can reuse input values from cache and raise effective intensity; it cannot be credited until the tile fits and the access trace is shown.

CONNECT THE CLAIM TO FALSIFIABLE EVIDENCE

A capstone claims a new prefetcher improves performance by reducing memory stalls, but reports only end-to-end speedup on one input.

Which evidence package can test the proposed mechanism?

Reference decision: Defined workloads, baseline controls, stall events, coverage, accuracy, and repeated timing

The mechanism predicts changes in stalls and prefetch quality before end-to-end speedup. Controlled workloads and repeated measurements connect those intermediate effects to the final claim and reveal harmful cases.

Representation and ISA choices

EXAMPLE

Worked trace: for a 3×3 stencil with signed 8-bit weights $w_{i,j} \in [-127, 127]$ and unsigned 8-bit channels $p_{i,j} \in [0, 255]$, the worst-case magnitude $\sum |w_{i,j}|p_{i,j} = 9 \times 127 \times 255 = 291{,}465$ overflows a signed 16-bit accumulator ($[-32{,}768, 32{,}767]$), requiring at least a 20-bit signed accumulator (typically a 32-bit lane) before final clamping to $[0, 255]$.

Specify whether pixels are stored interleaved RGBA or in separate channel planes. Interleaved layout is convenient when all channels of one pixel are processed together; planar layout can improve SIMD scans of one channel and simplify separable filters. State byte order, alignment, signedness of coefficients, accumulator width, and saturation behavior. A signed 8-bit coefficient multiplied by an unsigned 8-bit pixel can require a signed 32-bit accumulator to avoid intermediate overflow.

The ISA needs loads/stores, integer or vector multiply-add, widening arithmetic, min/max or clamp, pointer increments, branches or masked operations, and a memory-ordering primitive for DMA ownership. If coefficients are programmable, define their encoding and validation. If the vector

instruction saturates, verify whether it rounds, truncates, or wraps; using modular arithmetic where saturation is required produces visually subtle but systematic errors.

Worked overflow bound. If pixels are at most 255 and kernel coefficients have absolute value at most 127, one channel's absolute convolution sum is at most $9 \cdot 255 \cdot 127 = 291,465$. That requires 19 magnitude bits plus a sign bit (20 signed bits): signed 19-bit range ends at 262,143. A 32-bit accumulator provides margin for bias, intermediate transforms, and implementation variation. The final result must be rounded and clamped after the full sum, not after each product unless the specification permits that approximation.

Datapath and pipeline implications

> **WARNING**
>
> If a pipelined multiply-accumulate unit has a 4-cycle latency and the inner loop accumulates all 9 taps into a single register, RAW data hazards serialize execution to one tap every 4 cycles; unrolling across multiple independent output pixels or channels is required to hide pipeline latency.

A scalar core may load one pixel at a time, perform arithmetic, and store output. A vector core handles multiple pixels or channels per instruction, but boundary pixels, alignment, tails, and coefficient broadcast must be designed. A pipelined multiply-add has latency greater than one cycle, so independent accumulators or software scheduling are needed to maintain throughput. A dependent single accumulator can underutilize the pipeline even if the arithmetic unit accepts one operation per cycle.

Implementation Alternative	Issue Width & Arithmetic Units	Unrolling / Latency Hiding	Cycles per RGBA Pixel	Clock Target to Meet 30 fps
Scalar In-Order Core	1-way issue; 1 pipelined MAC (latency 4)	Single accumulator (RAW stalls serialize taps)	36 cycles (9 taps × 4 cycles)	$\geq 2.24\,\mathrm{GHz}$ (exceeds scalar envelope)
Scalar Unrolled Core	1-way issue; 1 pipelined MAC (latency 4)	4 interleaved channel accumulators (R, G, B, A)	9 cycles (fully hides MAC latency)	$\geq 560\,\mathrm{MHz}$ (within thermal envelope)
SIMD Vector Core	4-way SIMD vector unit; 128-bit vector MAC	16-way SIMD vector taps across 4 pixels	2.25 cycles	$\geq 140\,\mathrm{MHz}$ (low-power edge envelope)
Dedicated Hardware Stencil	9-multiplier parallel 2D convolver pipeline	Hardwired streaming systolic array	1 cycle / pixel	$62.2\,\mathrm{MHz}$ (ultra-low power ASIC/FPGA)

Branch handling matters at image boundaries. A branch per pixel for clamping or bounds can cause mispredictions and front-end pressure. Vector min/max or masked loads can make the inner loop regular. Separate a prologue, aligned steady-state kernel, and epilogue when the ISA and workload justify it; the code-size and maintenance cost belongs in the comparison.

Memory hierarchy and tiling

EXAMPLE

For a tile of $w = 64, h = 32$ four-byte RGBA pixels, the input Halo footprint is $4 \times (64 + 2)(32 + 2) = 8{,}976$ bytes and the output tile is $4 \times 64 \times 32 = 8{,}192$ bytes, totaling $17{,}168$ bytes—comfortably fitting within the 32 KiB L1 data cache while leaving ways free for stack and coefficient reads.

The frame is far larger than L1 and L2, so a naive traversal may be bandwidth-bound. A tile should include enough neighboring rows for the 3×3 stencil and fit the chosen cache level with output and coefficient data. For a tile of width w and height h, the input footprint is approximately $(w + 2)(h + 2)$ pixels and output footprint wh pixels, adjusted for channels and layout. Choose w, h so the footprint plus other active data fits with a safety margin; associativity conflicts can invalidate a capacity-only calculation.

Row-major layout favors a row-wise inner loop. A line of 64 bytes contains 16 four-byte RGBA pixels, so sequential access can use most fetched data. A column stride of 1920 pixels is 7680 bytes, causing poor spatial locality and potentially repeated set conflicts. Tiling changes the reuse distance: neighboring output pixels reuse input rows before eviction.

Design comparison. A row streaming design minimizes temporary storage but may reload rows for each output channel. A channel-plane tiled design improves vector reuse and coefficient broadcast but may require a format conversion. A separable design cuts arithmetic but writes and rereads an intermediate image. Measure bytes moved at each hierarchy level; “fewer operations” is not enough if traffic doubles.

DMA and buffering

> **WARNING**
>
> Never allow the filter kernel to read an input frame buffer before checking the device completion status and executing the required memory/cache synchronization barrier, and never recycle an output buffer to the DMA ring while the display or output engine still holds in-flight read ownership.

The input device writes into a ring of owned frame buffers. The driver validates descriptor addresses and lengths, maps them through an IOMMU if present, publishes ownership with the required memory barrier, and receives a completion with generation and length. The processing core must not read a buffer until completion is ordered and validated. After processing, ownership returns only after all CPU writes are visible to the output device or host.

Buffer Ring Slot	Buffer State	Current Writer	Current Reader	Memory / Cache Synchronization Action	Ownership Transfer Event
Slot 0	`CAPTURING`	Camera DMA Engine	None (CPU access trapped)	Driver issues acquire fence before publishing descriptor	Camera asserts frame interrupt on completion
Slot 1	`PROCESSING`	CPU Filter Kernel	CPU Filter Kernel	CPU performs cache line prefetch; processes 64×32 tiles	Kernel completes output frame checksum
Slot 2	`TRANSMITTING`	None (writes blocked)	Display DMA Engine	CPU issues `sfence` / write-back dirty lines to RAM	Display engine signals VSYNC completion
Slot 3	`FREE`	None	None	OS resets buffer generation tag; clears stale descriptors	Driver enqueues slot back to Camera receive ring

Double buffering permits one frame to arrive while another is processed; triple buffering absorbs jitter but increases memory. If processing time occasionally exceeds the frame period, the queue grows. A bounded system must choose drop-oldest, drop-newest, backpressure, or degraded-quality behavior. The policy is an architecture requirement because it affects memory capacity, latency, and user-visible correctness.

Performance budget

DEFINITION

Under the stage-budget decomposition $T_{\text{e}}^{\text{fram}} = T_{\text{n}}^{\text{DMA_i}} + T_{\text{e}}^{\text{queu}} + T_{\text{e}}^{\text{comput}} + T_{\text{l}}^{\text{memory_stal}} + T_{\text{t}}^{\text{DMA_ou}}$, meeting the 30 fps throughput constraint requires $\mathrm{E}[T_{\text{e}}^{\text{stag}}] \leq 33.3\ \text{ms}$ per pipeline stage, while meeting the tail constraint requires $P(T_{\text{e}}^{\text{fram}} \leq 25\ \text{ms}) \geq 0.99$.

At 30 frames per second, the average frame budget is 33.3 ms, but the p99 requirement is 25 ms after availability. If one core handles 2 billion useful operations per second and the model requires 4.2 billion weighted operations per second, the design needs vectorization, multiple cores, lower arithmetic, or a hardware accelerator. Estimate instruction count and CPI separately from arithmetic count. Memory stalls and DMA wait time belong in the end-to-end timeline.

Pipeline Stage	Quantitative Budget (T_{stage})	Limiting Resource Ceiling	Architecture Mitigations for Tail Latency (p99)
Camera DMA Ingest	$1.2\,\text{ms}$	PCIe Gen3 $\times 1$ ($985\,\text{MB/s}$)	Dedicated continuous physical frames; no IOMMU page faults
L1/L2 Cache Prefetch	$0.8\,\text{ms}$	L2 $\rightarrow$ L1 cache bandwidth ($64\,\text{B/cycle}$)	Hardware stream prefetcher tuned to row stride
2D Stencil Convolution	$14.5\,\text{ms}$	Vector MAC throughput ($4\,\text{ops/cycle}$)	Dual 4-way vector accumulators; loop boundary peel
Output Cache Write-back	$1.5\,\text{ms}$	L2 write-back bandwidth	Streaming non-temporal stores (`movntdq`) bypass L1
Display DMA Egress	$1.2\,\text{ms}$	Display engine memory channel	High-priority QoS traffic class on memory arbiter
Total Frame Latency	$19.2\,\text{ms}$	**Headroom:** $5.8\,\text{ms}$ **vs** $25\,\text{ms}$ **SLA**	**Deterministic margin guarantees** $P(T \leq 25\,\text{ms}) \geq 0.99$

Use a stage budget: input completion, cache warm-up, compute, output write-back, and scheduling. For a vector design, calculate lanes per instruction and tail fraction. For a multicore design, account for partitioning, synchronization, shared-cache traffic, and load imbalance. A design that meets mean time but not p99 must identify the source of tail variation—page faults, queueing, interrupts, contention, or device behavior.

Reliability and security

> **WARNING**
>
> Multiplying untrusted descriptor `width * height * 4` in 32-bit unsigned arithmetic can wrap around and pass a buffer-size check while causing the filter loop or DMA engine to read or write past the allocated frame buffer.

Validate every descriptor length against the allocated buffer, prevent integer overflow in stride and frame-size calculations, and reject unsupported dimensions before DMA. Device completion is untrusted input: a malformed length or status must not become an unchecked loop bound. Memory mappings must be revoked after device reset, and stale completions must be rejected by generation.

Failure Mode / Threat Vector	Trigger Condition	System Impact	Hardware & Architectural Verification Invariant
Descriptor Integer Overflow	Untrusted camera reports $W \times H \times 4 > 2^{32} - 1$	DMA writes outside allocated frame into kernel memory	Reject descriptor if $W > 4096 \vee H > 2160$ before DMA setup
Stale DMA Overwrite	Host reallocates frame buffer before DMA engine quiesces	Active DMA corrupts newly assigned tenant process memory	Enforce generation counters; poll device status bit before reuse
Frame Dropping Under Jitter	Processing exceeds 33.3 ms during transient background load	Ring buffer overflows; camera DMA drops subsequent frames	Triple-buffering absorbs 1-frame jitter; fallback to drop-oldest
Speculative Cache Snooping	Secret camera data left resident in shared LLC	Cross-tenant Flush+Reload side channel leaks frame bytes	Explicit cache line clear / way-partitioning before core release

If the image contains sensitive data, cache timing, DMA isolation, and buffer clearing become part of the threat model. A faster vector implementation that leaves pixels in a shared cache or recycled buffer may have a different security posture. State what confidentiality is required and which layers enforce it.

Architecture specification package

> **REMARK**
>
> Every architectural diagram and specification table must pair quantitative estimates (operations, bytes transferred, cycle bounds) with explicit correctness and ownership invariants.

Produce a block diagram and a sequence diagram from device arrival through output completion. Specify pixel representation, ISA operations, accumulator width, pipeline schedule, cache mapping, tile dimensions, DMA ownership, queue policy, exception behavior, and security assumptions. Give a baseline and at least two alternatives, with a table of arithmetic work, estimated bytes moved, memory ceilings, compute ceilings, area/energy implications, and unverified assumptions.

Include a correctness argument for one output pixel, an invariant for buffer ownership, an AMAT or roofline calculation, and an experiment plan. The experiment must identify what would falsify the claim that the design is compute-bound, cache-bound, bandwidth-bound, or synchronization-bound. Report how you would measure p99 latency and how you would inject a delayed or malformed device completion.

Verification criteria

> **REMARK**
>
> Architectural designs are evaluated on falsifiable engineering rigor: a slower design with a sound overflow proof, verified DMA ownership, and a discriminating Roofline/AMAT experiment is superior to an unverified speedup claim.

A rigorous design specification is precise about boundaries and uncertainty. It derives rather than asserts arithmetic cost, connects data layout to cache lines, distinguishes architectural guarantees from microarchitectural assumptions, handles tail and overflow cases, and proposes measurements that could disprove its own model. A diagram without invariants is incomplete; a benchmark without workload and measurement boundaries is weak evidence; a speedup claim without a baseline is not a result.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.2.1: L1 Tile Working-Set Footprint and Compulsory DRAM Traffic

Verification application

Verify that a 64×32 RGBA output tile with a 1-pixel halo fits inside the 32 KiB 4-way L1 data cache and calculate the compulsory DRAM traffic per output pixel compared to an untiled naive kernel.

Include the 1-pixel border on all four sides: $(64 + 2) \times (32 + 2)$ input pixels plus 64×32 output pixels, at 4 bytes per RGBA pixel.

1. **Input tile with halo:** $(64+2)\times(32+2)=66\times 34=2{,}244$ pixels $\times 4\ \mathrm{B}=8{,}976\ \mathrm{B}$.
2. **Output tile:** $64\times 32=2{,}048$ pixels $\times 4\ \mathrm{B}=8{,}192\ \mathrm{B}$.
3. **Total working set:** $8{,}976+8{,}192=17{,}168\ \mathrm{B}\approx 16.77\ \mathrm{KiB}<32\ \mathrm{KiB}$, leaving nearly half of L1 capacity for stack frames, filter coefficients, and set-associativity conflict headroom.
4. **Traffic reduction:** Without reuse, each output pixel reads $9\times 4=36\ \mathrm{B}$ and writes $4\ \mathrm{B}$ ($40\ \mathrm{B/pixel}$). With L1 tiling, each output pixel amortizes to $\frac{8{,}976+8{,}192}{2{,}048}\approx 8.38\ \mathrm{B/pixel}$, reducing external memory traffic by $4.77\times$.

Exercise 5.2.2: Stencil Accumulator Width and Saturation Ordering

Verification application

Suppose the 3×3 convolution kernel uses signed 8-bit weights $w_{i,j}\in[-128,127]$ and unsigned 8-bit pixels $p_{i,j}\in[0,255]$. Determine the exact minimum signed two's-complement accumulator width required to prevent intermediate overflow before final clamping to $[0,255]$, and construct a 9-tap counterexample showing why per-tap clamping violates exact convolution semantics.

Compute the worst-case negative sum $9\times(-128)\times 255$ and compare against signed k-bit bounds $[-2^{k-1},2^{k-1}-1]$.

1. **Minimum accumulator width:** The extreme negative sum is $9 \times (-128) \times 255 = -293{,}760$, and the extreme positive sum is $9 \times 127 \times 255 = +291{,}465$. A signed 19-bit integer covers $[-262{,}144, +262{,}143]$, which overflows both bounds. A signed 20-bit integer covers $[-524{,}288, +524{,}287]$, so **20 signed bits** is the exact mathematical minimum (implemented in hardware with a standard 32-bit lane).
2. **Per-tap saturation counterexample:** Suppose 8 taps yield $+255 \times 1 = +255$ each and the 9th tap yields -1800 (e.g., $w = -8, p = 225$). Exact accumulation gives $8(255) - 1800 = 240 \in [0, 255]$. Clamping to $[0, 255]$ after every tap instead saturates at 255 across taps $2 \ldots 8$ and then drops to $\max(0, 255 - 1800) = 0$, introducing a 240-unit error.

Exercise 5.2.3: Pipeline RAW Hazard Hiding via Accumulator Unrolling

Construction application

Consider a 2 GHz core whose pipelined 4-channel MAC instruction completes all four RGBA lanes of one stencil tap with initiation interval $1\ \text{cycle}$ and latency $4\ \text{cycles}$. A naive loop updates one pixel accumulator across all 9 taps ($9 \times 4 = 36\ \text{cycles/pixel}$). Show how unrolling across $k = 4$ independent output pixels removes RAW stalls to achieve $9\ \text{cycles/pixel}$, and calculate the resulting compute-only frame time for a 1920×1080 image.

Interleave tap t across four independent pixel accumulators A_0, A_1, A_2, A_3 before issuing tap $t + 1$ to A_0.

1. **Hazard elimination:** With $k = 4$ independent pixel accumulators, the loop schedules `MAC A0, tap_t`, `MAC A1, tap_t`, `MAC A2, tap_t`, `MAC A3, tap_t` in consecutive cycles $0, 1, 2, 3$. By cycle 4, the 4-cycle latency of `A0` has completed, so `MAC A0, tap_{t+1}` executes with zero RAW stalls.
2. **Amortized cycle rate:** Four pixels complete all 9 taps in $4 \times 9 = 36$ cycles (plus negligible 3-cycle drain), giving 9 cycles/pixel.
3. **Compute-only frame time:** A 1920×1080 frame has $2{,}073{,}600$ pixels. At 9 cycles/pixel, compute requires $2{,}073{,}600 \times 9 = 18{,}662{,}400$ cycles. At 2 GHz ($2 \times 10^9 \text{ cycles/s}$), compute takes $\frac{18{,}662{,}400}{2 \times 10^9} = 9.33 \text{ ms}$, leaving 15.67 ms of headroom within the 25 ms p99 budget (compared to 37.32 ms without unrolling).

Exercise 5.2.4: DMA Descriptor Integer Overflow and Generation-Counter Race

Verification transfer

A camera driver allocates an $8{,}294{,}400$-byte frame buffer (`1920 * 1080 * 4`) and validates an incoming descriptor in 32-bit unsigned arithmetic with `uint32_t bytes = width * height * 4; if (bytes <= buf_capacity) accept();`. Construct a concrete `(width, height)` input pair that bypasses this check via wraparound, and write both the overflow-safe validation invariant and the generation-tag check required before recycling a DMA slot.

Choose factors of $2^{32} = 4{,}294{,}967{,}296$ (for instance powers of two whose product times 4 wraps modulo 2^{32} into $[0, 8{,}294{,}400]$).

1. **Overflow exploit vector:** Let `width = 65536` (2^{16}) and `height = 16384` (2^{14}). In 32-bit unsigned arithmetic, $65{,}536 \times 16{,}384 \times 4 = 2^{32} \equiv 0 \pmod{2^{32}}$. Since $0 \leq 8{,}294{,}400$, the naive check accepts the descriptor while the DMA loop attempts to transfer $4\ \mathrm{GiB}$ past the buffer. (Similarly, `width = 32768, height = 32769` wraps to `131,072` bytes $\leq 8{,}294{,}400$.)
2. **Safe validation invariant:** Reject zero dimensions and enforce explicit architectural bounds plus division-checked 64-bit bounds before multiplying: `width > 0 && height > 0 && width <= 4096 && height <= 2160 && (uint64_t)width * height <= buf_capacity / 4`.
3. **Buffer recycling invariant:** Never return a slot to `FREE` until an MMIO status read confirms the DMA engine is idle on that descriptor and its returned generation token matches the slot's expected generation counter (incremented monotonically on each reuse).

Exercise 5.2.5: Falsifying Memory-Bandwidth vs. Compute Bottlenecks

Generalisation synthesis

Compare the untiled naive stencil ($68\ \mathrm{ops/pixel}$, $40\ \mathrm{B/pixel}$, intensity $1.7\ \mathrm{ops/B}$) and the 64×32 L1-tiled stencil ($68\ \mathrm{ops/pixel}$, $8.38\ \mathrm{B/pixel}$ DRAM traffic) on a $2\ \mathrm{GHz}$ core with a $20\ \mathrm{GB/s}$ DRAM ceiling and a $16\ \mathrm{Gops/s}$ SIMD compute ceiling. Determine which roofline bound constrains each variant, and design a single controlled experiment that falsifies the hypothesis that the tiled pipeline is DRAM-bandwidth-bound.

Compute the roofline ridge point ($16\ \mathrm{Gops/s} \div 20\ \mathrm{GB/s}$) and test sensitivity to core clock scaling versus DRAM bandwidth scaling.

1. **Roofline classification:** The machine ridge point is $\frac{16\ \text{Gops/s}}{20\ \text{GB/s}} = 0.8\ \text{ops/B}$.
   - **Untiled kernel** ($1.7\ \text{ops/B}$): Memory roofline ceiling is $1.7 \times 20 = 34\ \text{Gops/s} > 16\ \text{Gops/s}$ theoretically if L1/L2 had infinite conflict-free streaming, though in practice poor row-strided reuse drops effective intensity below $0.8\ \text{ops/B}$ and stalls on DRAM bandwidth.
   - **L1-tiled kernel** ($\frac{68}{8.38} \approx 8.11\ \text{ops/B}$): Exceeds the $0.8\ \text{ops/B}$ ridge point by over $10\times$ (DRAM ceiling $8.11 \times 20 \approx 162.3\ \text{Gops/s}$), placing the tiled kernel squarely under the $16\ \text{Gops/s}$ **compute ceiling**.

2. **Falsification experiment:** Hold input resolution, tile size (64×32), and DRAM clock fixed while stepping CPU core frequency from $2.0\ \text{GHz}$ down to $1.0\ \text{GHz}$ (and measuring DRAM CAS byte counters and L2 miss stalls). If frame time doubles while total DRAM bytes stay constant at $\approx 17.4\ \text{MB/frame}$, compute is the bottleneck; conversely, if frame time stays nearly invariant under core clock halving while L2/DRAM stall cycles remain high, the hypothesis of compute-boundedness is falsified and the bottleneck lies in memory stalls or cache conflicts.

References & Further Reading

- MIT 6.004 Computation Structures: learning objectives and lecture notes
- UC Berkeley CS 61C: memory hierarchy and AMAT
- Cornell CS 3410: Computer System Organization and Programming

SECTION DETAIL

Architecture Review

Architecture review is a structured argument that a proposed machine satisfies its software contract under its intended workload and failure model. A block diagram can show components without proving that instruction state, memory visibility, exceptions, device ownership, and performance budgets compose. This review uses a hypothetical 4-core image-processing SoC to demonstrate the evidence a reviewer should request.

Review packet

DEFINITION

An architecture design review audits whether a proposed microarchitecture and system organization refine the programmer-visible ISA contract and meet stated latency, bandwidth, and isolation budgets under adversarial traces.

The proposal contains an ISA summary, pipeline diagram, cache hierarchy, page-table model, DMA rings, and benchmark results. The cores are four-wide out of order, have private 32 KiB L1 caches and shared 2 MiB L2, use write-back coherence, and share DRAM. A camera writes frames through DMA; software applies a filter and sends output to a display engine. The design claims 30 frames per second and "strong consistency" without defining memory-ordering operations or the p99 latency boundary.

The first review question is scope: which claims are architectural and which are implementation assumptions? If software binaries depend on an instruction's exact overflow or alignment behavior, that behavior belongs to the ISA. If the claim is that a loop usually hits in L1, it is a workload-dependent performance hypothesis. If a driver assumes a DMA buffer is not reused until completion, that is an ownership protocol. Each claim needs a different proof or measurement.

REVIEW THE BOUNDARY

Why must an architecture review test both the functional contract and the performance model?

Answer: A correct design can still violate latency or bandwidth constraints

A processor or memory system can compute correct results while missing a service budget. Review needs separate evidence for semantics and quantitative behavior.

Contract completeness

Review the architectural state: registers, PC, privilege modes, memory model, exception causes, atomic operations, vector state, and device address-space attributes. For every instruction class, ask what changes and what remains unchanged. An illegal instruction must not inherit a neighbor's write enable. A faulting load must specify whether the PC points to the faulting instruction and whether retry is legal.

Check type boundaries: signed and unsigned arithmetic, extension rules, unaligned accesses, byte order, vector tails, and floating-point rounding. An ISA that says "integer add" without overflow semantics forces compilers and debuggers to guess. A device interface that permits arbitrary access widths can create incompatible implementations.

Worked audit finding. The proposed ISA says loads return a word but does not say whether a byte load sign-extends, zero-extends, or leaves upper bits unchanged. This is a blocking defect because software cannot write a portable decoder and a compiler cannot preserve source-level signedness. The reviewer must request a precise table of width and extension behavior.

REVIEW THE REFINEMENT BOUNDARY

A review claims a speculative core is correct because its final benchmark output matches once. No trace connects speculative state to in-order architectural retirement.

Which evidence addresses the missing correctness argument?

Reference decision: A commit trace showing ordered visible effects and precise recovery on faults

Microarchitectural work may be reordered, but committed architectural effects must refine sequential ISA behavior. A retirement and recovery trace exposes that boundary directly.

Datapath and pipeline audit

WARNING

In an out-of-order or speculative pipeline, verifying that non-faulting instructions compute the right value is insufficient: the review must confirm that flushed or younger instructions after a faulting load or mispredicted branch never commit register, memory, or device side effects.

Trace one instruction of each class through every stage. For each stage, identify its inputs, outputs, valid bit, control bundle, and side effects. Confirm that destination identifiers travel with their results and that bubbles disable all architectural writes. Check the critical path and the clock constraint using worst-case rather than average delay.

Audit hazards with adversarial sequences: dependent ALU chains, load-use pairs, branch-after-compare, back-to-back writes to one register, memory stalls, and exceptions after speculation. Ask which forwarding paths exist, which dependencies stall, and how the controller prevents deadlock. A review finding must include a trace that demonstrates the failure or a proof that the path is safe.

For out-of-order execution, inspect rename-map checkpoints, physical-register reclamation, load/store disambiguation, reorder-buffer retirement, and store visibility. The statement “execution is out of order but results are correct” is not evidence. The design needs a refinement argument to the ISA and a precise exception story.

Memory-system audit

EXAMPLE

If Core 0 writes `data = 1` then `flag = 1` using write-back L1 caches, cache coherence ensures Core 1 eventually observes `flag == 1`, but only a defined memory-consistency operation (such as a release store paired with an acquire load, or `fence rw, rw`) guarantees Core 1 cannot read the stale pre-write value of `data` after observing `flag == 1`.

Compute cache tag/index/offset fields from capacity, line size, and associativity. Identify conflict traces and estimate AMAT from local miss rates. Examine write-back eviction ordering, dirty state, coherence transitions, and fence semantics. A write-back cache without a coherence policy is not a complete shared-memory design: two cores may read different copies after one writes.

Review virtual-memory interaction: TLB tags or ASIDs, page-table levels, permission checks, page faults, cache aliases, invalidation, and frame reuse. A stale TLB entry after context switch is a security defect, not merely a performance problem. A page-table update without interprocessor shootdown can expose a frame to the wrong process.

Review the memory model with a message-passing test. A producer writes data then a flag; a consumer observes the flag then reads data. State which release/acquire or fence operations make the intended order visible. Coherence of each location is not enough to prove cross-location ordering.

I/O and ownership audit

> **WARNING**
>
> Mapping MMIO control or status registers as cacheable or allowing speculative loads into read-to-clear device FIFO ranges corrupts device state machines; I/O regions must enforce non-cacheable, non-speculative access attributes.

The camera DMA path must specify buffer allocation, IOMMU mapping, cache synchronization, descriptor publication, completion ordering, interrupt behavior, timeout, reset, and buffer reuse. Draw the ownership states and mark which agent may write each field. A completion carrying an invalid length must be rejected before it becomes an array bound.

Audit Dimension	Verification Standard	Failure Mode Observed	Audit Remediation Action
MMIO Attributes	Strongly-ordered, non-cacheable	Cache hit returns stale device status register	Configure page table entry with `Device-nGnRE` / `PMA` I/O attributes
Side-Effect FIFO Read	Non-speculative load dispatch	Speculative load silently drains hardware FIFO entry	Enforce load fence (`fence r, r` / `lfence`) before FIFO read
DMA Descriptor Publication	Strict release barrier	Device reads partially initialized descriptor	Insert `fence w, o` (write-to-output fence) before ringing doorbell
Buffer Ownership Handshake	Single-writer / single-reader	CPU processes buffer while camera DMA writes	Require explicit generation match and ownership bit check

Review device-memory attributes. Caching a status register can return stale state; speculative reads can acknowledge a side-effect register unexpectedly. A barrier used for ordinary memory may not order a device doorbell. The driver and hardware specification must name the required primitive.

Performance and measurement audit

EXAMPLE

At 30 frames/s, a 32 ms mean processing time with 15 ms standard deviation satisfies average throughput while violating a 25 ms p99 latency SLO on a large fraction of frames; the review must demand percentile distributions and hardware stall-counter breakdowns rather than mean FPS alone.

The proposal's 30-frame-per-second claim should be decomposed into input, processing, output, scheduling, and queueing. Count arithmetic operations, estimate bytes moved, compute a roofline ceiling, and identify whether the design is compute- or bandwidth-bound. Then examine p50 and p99, not only average frame time. If input bursts or DMA interrupts create queueing, mean throughput can hide missed latency targets.

Benchmark methodology must state input dimensions, pixel distribution, alignment, compiler flags, warm-up, core placement, frequency, cache state, repetitions, and timing boundaries. Counters should test mechanisms: cache misses, branch misses, bandwidth, queue occupancy, and stall reasons. A review should reject "2× faster" when the baseline uses different correctness settings or excludes setup that the product must pay.

Safety and security audit

WARNING

Without an IOMMU restricting the camera DMA engine to validated frame-buffer pages, a malformed descriptor or compromised device firmware can overwrite kernel page tables or privileged code in host DRAM.

Enumerate assets and adversaries: user frames, page tables, device buffers, privileged code, and inter-core messages. Check memory-safety arithmetic, privilege boundaries, IOMMU mappings, speculative side channels, cache timing, and stale ownership. Identify the blast radius of a compromised device or driver. A security control must map to a threat and have a test or operational signal.

Hardware Security Dimension	Attack Surface Evaluated	Concrete Vulnerability Mechanism	Verified Defense Invariant
DMA Access Authority	PCIe bus-mastering endpoints	Compromised peripheral overwrites OS page tables	Hardware IOMMU enforces page-table isolation per device domain
Transient Execution Leakage	Branch predictor & out-of-order execution	Spectre v1 bounds-check bypass reads host memory	Speculative execution barrier (`sb` / `lfence`) on untrusted indexing
Rowhammer DRAM Disturbance	Repeated activation of adjacent DRAM rows	Bit-flips induced in adjacent physical memory pages	Target Row Refresh (TRR) enabled in DRAM controller
Fault Recovery & Privilege Containment	Bus error / ECC double-bit parity trap	Kernel panic or uncontained privilege escalation	Synchronous Machine Check Exception (MCE) kills only faulting task

Check fault containment: illegal instruction, page fault, ECC error, DMA overrun, device timeout, thermal throttling, and power loss. For each, state whether the processor retries, kills a process, resets a device, degrades service, or enters a machine-check path. An architecture that only describes the happy path is incomplete.

Review decision and evidence levels

> REMARK
>
> Every review finding must separate its severity classification (blocking correctness defect vs. performance risk) from the evidence level required to close it (formal invariant/trace vs. statistical benchmark distribution).

Classify findings as blocking correctness defects, unverified assumptions, performance risks, or maintainability concerns. A design decision is strong when it has a precise invariant, a worked trace or proof, an experiment plan, and a rollback or recovery story. “The simulator passed” is evidence only for the tested traces and simulator model. “The benchmark improved” is evidence only for the stated workload and measurement boundary.

Worked decision. The proposal uses write-back private L1 caches but gives no coherence protocol. Reject as incomplete because a two-core program can write a shared location and observe stale values, violating the memory model. A revision must specify a protocol, fence interaction,

invalidation/ownership transitions, and tests for write-write, read-after-write, false sharing, and eviction. The reviewer should not approve the design merely because single-core benchmarks pass.

Required review report

> **REMARK**
>
> Use the boundary checklist below to structure the final sign-off report so that every hardware-software interface (ISA, MMU, cache, DMA, interrupt) has an explicit pass/fail oracle.

Construct a one-page architecture claim table, a block and sequence diagram, three adversarial traces, two quantitative models, a threat model, and a list of open assumptions. For every claim, label evidence as theorem/proof, simulation, benchmark, specification, or expert judgment. Include one finding you expect the design team to disagree with and the measurement that would settle it.

Before signing off, perform a boundary review. Ask what happens at zero and maximum lengths, the first and last cache line, a branch taken at a page boundary, an exception after a younger DMA request, a context switch while a TLB entry is cached, a device reset while a descriptor is in flight, and a power failure after a storage acknowledgement. These cases expose whether the design has a real state model or only a collection of nominal diagrams. Record the expected architectural observation and the internal mechanism that enforces it for each case.

The final review meeting should separate disagreement about a requirement from disagreement about an implementation. If two reviewers disagree about whether a frame may be dropped, first resolve the service contract; if they agree on the contract but disagree about a cache or DMA mechanism, compare evidence against the invariant. This prevents architecture review from becoming a preference contest and makes the decision reversible when a measured assumption changes.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.3.1: ISA Width, Extension, and Exception Contract Audit

Verification application

A proposed 64-bit ISA manual defines `LB rd, offset(rs1)` as "loads one byte from effective address `rs1 + offset` into `rd`", omits unaligned-access fault rules for `LW` / `LD`, and leaves `EPC` unspecified when a data TLB miss traps during a speculative load. Identify the three blocking contract defects and write the exact architectural state rules needed to close them.

Check upper-bit semantics in the 64-bit destination register, alignment fault behavior, and precise exception state.

1. **Upper-bit extension ambiguity:** The manual must distinguish signed sign-extension (`LB`: bits `63..8` filled with bit `7`) from unsigned zero-extension (`LBU`: bits `63..8` zeroed); leaving upper bits undefined or preserved prevents portable compilation of `int8_t` vs. `uint8_t`.
2. **Alignment contract omission:** For multi-byte loads (`LW`, `LD`), specify whether unaligned addresses trap with an alignment fault or split atomically/non-atomically across cache lines, and define the trap vector and memory side-effect guarantee.
3. **Imprecise trap specification:** On a data TLB miss, the ISA must enforce precise exceptions: `EPC` records the faulting load's PC, `rd` remains unmodified, and younger speculative instructions are squashed before architectural commitment.

Exercise 5.3.2: Out-of-Order Speculative Store and Forwarding Audit

Verification application

A four-wide out-of-order core verifies single-cycle branch prediction recovery in simulation, yet allows executed stores to write L1 cache lines before reaching the head of the reorder buffer (ROB) and forwards from the store buffer by matching only word-aligned addresses without byte-mask checking. Construct an adversarial instruction sequence demonstrating architectural failure for each flaw and state the hardware invariants that fix them.

Trace a store younger than a faulting load for early L1 writes, and a partial byte store followed by a wider load for forwarding.

1. **Premature store write trace:** Suppose instruction I_1 (`LD x1, 0(x2)`) raises a page fault while younger I_2 (`SW x3, 0(x4)`) writes L1 early. When I_1 traps at ROB retirement, architectural memory already contains I_2's speculative store, violating precise exceptions and potentially corrupting shared data visible to other cores or DMA. **Invariant:** Stores remain buffered in a speculative store queue and commit to L1 only upon reaching the non-faulting head of the ROB.
2. **Width-blind forwarding trace:** Suppose I_1 (`SB x5, 0(x10)`) writes byte 0 while I_2 (`LW x6, 0(x10)`) reads all four bytes `[0..3]`. Word-address-only matching either returns `x5` with stale/zero upper bytes or fails to merge bytes `[1..3]` from L1. **Invariant:** Store-to-load forwarding requires full containment of the load's byte mask within the matching store's byte mask (or stalling/merging byte-by-byte from L1).

Exercise 5.3.3: Weakly Ordered Message-Passing Failure Trace and Fence Repair

Construction transfer

Given Core 0 executing `M[data] = 42; M[flag] = 1;` and Core 1 executing `while (M[flag] == 0); r1 = M[data];` on a weakly ordered multicore processor with coherent caches (initially `M[data] = 0`, `M[flag] = 0`), construct a trace showing how `r1 == 0` can occur and specify the architectural ordering repair.

Coherence guarantees single-address serialization, not cross-address store-to-store or load-to-load ordering across `data` and `flag`.

1. **Failure trace without barriers:** Even with hardware cache coherence (MESI), Core 0's store buffer or interconnect can make the write `M[flag] = 1` visible to Core 1 before `M[data] = 42` drains, or Core 1's out-of-order load unit can speculatively read `M[data]` (observing `0`) before the loop exits on `M[flag] == 1`.
2. **Architectural repair:** Core 0 must issue a release store (`stlr` or `fence w, w` between the stores to `data` and `flag`), and Core 1 must issue an acquire load (`ldar` or `fence r, r` after observing `M[flag] == 1` and before reading `M[data]`).

Exercise 5.3.4: Camera DMA Ownership Invariant and MMIO Attribute Audit

Formalisation transfer

In the camera SoC proposal, the driver maps MMIO device status and read-to-clear FIFO registers as normal write-back cacheable memory and updates the DMA ring tail doorbell immediately after writing descriptor fields in cached DRAM. Identify two correctness/hardware corruption failures this causes and formulate the memory-attribute, ordering, and IOMMU isolation invariants required for a sound I/O interface.

Examine stale/speculative MMIO reads, descriptor publication order across memory/I/O domains, and device physical address bounds.

1. **MMIO cacheable failure:** Caching status registers returns stale values from L1 instead of live device state, while speculative loads into a read-to-clear FIFO register silently drain queued hardware words on mispredicted paths. **Invariant:** Map all MMIO register ranges with strongly ordered, non-cacheable, non-speculative attributes (`Device-nGnRE` / I/O PMA) and guard side-effect reads with load barriers (`fence r, r` / `lfence`).
2. **Doorbell race & DMA isolation:** Without a write-to-output barrier and cache flush (on non-coherent DMA), the device can fetch uninitialized descriptor bytes after seeing the doorbell write; without an IOMMU, a corrupted descriptor can target kernel page tables. **Invariant:** Flush/clean descriptor lines if non-coherent, issue `fence w, o` before ringing the MMIO doorbell, verify buffer generation/ownership tags on completion, and confine device bus mastering through an IOMMU page-table domain covering only active frame buffers.

Exercise 5.3.5: Separating Throughput Averages from p99 Latency and Roofline Evidence

Generalisation synthesis

The review packet claims the 30 fps / 25 ms p99 latency SLO is met because mean frame latency is $21.5\ \text{ms}\,(46.5\ \text{fps})$ and cites a $2.1\times$ speedup over a scalar baseline compiled at `-O0`. Explain why neither metric establishes the SLO or microarchitectural claim, and design a falsifiable measurement protocol using latency distributions, hardware counters, and roofline arithmetic intensity.

Compare mean vs. tail percentiles under jitter, critique unoptimized baselines, and bind DRAM traffic to frame latency ceilings.

1. **Why mean and `-O0` evidence fail:** A $21.5\ \text{ms}$ mean easily violates a $25\ \text{ms}$ p99 bound whenever DMA interrupts, TLB shootdowns, or DRAM refresh/queueing create a heavy tail (e.g., 5% of frames taking $35\ \text{ms}$). Comparing against `-O0` attributes basic register allocation and dead-code removal to the proposed hardware.
2. **Falsifiable measurement protocol:**
   - Benchmark against an `-O3` /vectorized reference implementation with pinned frequencies, warm-up policy, and fixed input resolution (1920×1080 RGBA).
   - Record full per-frame latency percentiles ($p50, p95, p99, p99.9$) across $\geq 1{,}000$ contiguous frames with concurrent camera DMA load.
   - Collect hardware counters (retired ops, DRAM CAS read/write bytes, L1/L2 miss stalls, TLB misses) to compute measured operational intensity (ops/byte) on the roofline model and verify which stage bounds $p99$.

References & Further Reading

- MIT 6.004 Computation Structures: learning objectives and lecture notes
- UC Berkeley CS 61C: memory hierarchy and multicore notes
- Cornell CS 3410: Computer System Organization and Programming

Course Practice

SECTION DETAIL

Computer Architecture Practice Workshop

Problem Workshop

This workshop generates algorithmic practice problems spanning computer architecture: signed two's complement representation, pipelined data hazards and forwarding, multi-level cache AMAT calculations, and branch misprediction penalties.

Recommended Workflow

1. Calculate cycle counts, bit decompositions, and memory access times independently before inspecting hints.
2. Consult progressive hints when encountering pipeline dependencies or hierarchy parameters.
3. Verify derivations against the complete worked solution, checking hardware state transitions and penalty cycles.

INTERACTIVE PRACTICE WORKSHOP

Randomized parameter generation, progressive hints, and worked derivations are available in the interactive web edition. Covered topics:

- Digital Representation
- Pipelining & Hazards
- Cache Hierarchies & AMAT
- Instruction Set Architecture & Datapath

References & Further Reading

- David A. Patterson and John L. Hennessy (2020). *Computer Organization and Design: The Hardware/Software Interface (RISC-V Edition)*. Morgan Kaufmann.
- John L. Hennessy and David A. Patterson (2019). *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.