CS M151B - Computer Systems Architecture

COMPUTER MODEL AND ABSTRACTION

1 Abstraction

Abstraction. An abstraction is a simplified representation of a complex system that exposes only the necessary functionality while hiding implementation details. In computer science, abstractions allow programmers and engineers to manage complexity by working at higher conceptual levels instead of dealing directly with low-level mechanisms such as hardware signals, memory layouts, or device control.

For example, when calling a function such as `printf` in C++, the programmer does not need to understand how characters are rendered on the screen, how the operating system schedules the process, or how electrical signals control the display hardware. All of these details are hidden behind well-defined abstraction layers.

Abstraction is a fundamental concept in computer science and appears in nearly every system we design.

Computer System Abstraction Stack. Modern computer systems are structured as a hierarchy of abstraction layers:

- Applications (e.g., YouTube, video codecs)
- System Libraries (e.g., media decoders, standard libraries)
- Operating System Kernel (e.g., Windows Kernel)
- Hardware Abstraction Layer (HAL)
- Hardware (CPU, memory, peripherals)

Each layer depends only on the interface provided by the layer below it, not its internal implementation. This separation allows individual layers to be modified or replaced without affecting the entire system.

Abstraction in Networking: OSI Model. The OSI networking model demonstrates how abstraction is applied to communication systems by dividing networking responsibilities into layers:

1. Physical Layer (cables, electrical signals)
2. Data Link Layer (Ethernet)
3. Network Layer (IP)
4. Transport Layer (TCP)
5. Session Layer
6. Presentation Layer (e.g., JPEG encoding)
7. Application Layer (Email, RPC)

Each layer solves a specific problem and builds upon the services provided by the layer beneath it, enabling interoperability and scalability.

Hardware Abstraction Levels. Hardware systems are also designed using multiple abstraction layers:

- System level
- Register-transfer level (RTL)
- Gate level (e.g., NAND gates, multiplexers)
- Transistor level
- Mathematical operations (e.g., matrix multiplication)

These layers allow hardware designers to reason about functionality at different levels of detail without needing to consider transistor-level behavior at all times.

2 A Basic Computer Model

Basic Model Overview. At a high level, a computer performs tasks by following a simple conceptual model:

1. Accept input
2. Read instructions
3. Operate on data
4. Execute instructions
5. Produce output
6. Return a result

This abstract model applies to all modern computers, regardless of their internal complexity.

PERFORMANCE

Performance Equation. CPU performance is commonly modeled as:

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$\text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

Here, CPI is the clock cycles per instruction.

Performance can be improved by reducing instruction count, reducing CPI, or increasing clock rate.

Component	Affects	How
Algorithm	IC, CPI	Slower ops raise CPI
Language	IC, CPI	Abstraction adds indirect calls
Compiler	IC, CPI	Shapes instruction count and CPI
ISA	IC, Clock, CPI	Affects all three factors

Table 1: Factors Affecting CPU Performance

THE VON NEUMANN ARCHITECTURE

3 Introduction

Architecture Components. A Von Neumann system consists of:

- Input devices
- Central Processing Unit (CPU)
- Memory unit
- Output devices

4 Memory and Addressing

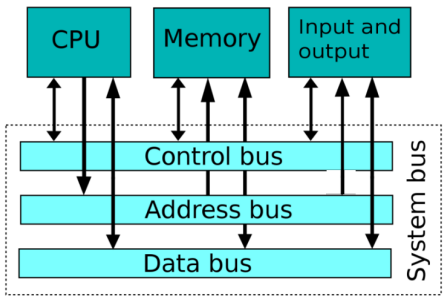
Memory Unit. The memory unit provides digital storage for both instructions and data. Because instructions are stored as data, programs can be modified, loaded, or generated dynamically, providing significant flexibility.

Shared Address Space. All memory and I/O devices share a single address space, which is logically partitioned into instruction, data, and input/output regions.

5 System Bus

Bus Structure. Communication between components occurs through a system bus composed of three parts:

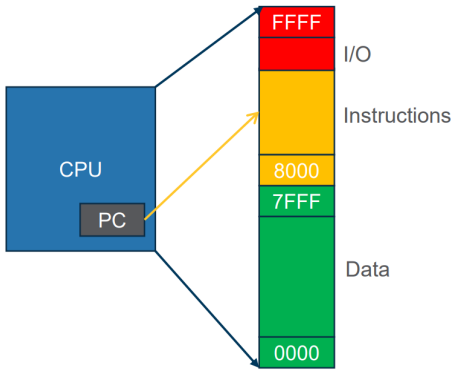
- **Address bus**: Specifies memory or I/O addresses and is unidirectional.
- **Data bus**: Transfers data and is bidirectional.
- **Control bus**: Manages read/write operations and device selection and is bidirectional.



6 Program Counter

Program Counter (PC). The program counter (PC), also known as the instruction pointer (IP), is a specialized register in the CPU that stores the memory address of the next instruction to be executed.

After each instruction, the PC is updated, enabling sequential execution unless altered by control-flow instructions.



7 Limitations of the Von Neumann Architecture

Von Neumann Bottleneck. Because instructions and data share the same bus, they compete for memory access. This shared-bus design leads to performance bottlenecks, excessive control switching, and CPU idle time while waiting for memory or I/O operations.

RISC-V BASE INSTRUCTION FORMATS

8 Representing Instructions in the Computer

How computers interpret instructions. Computers interpret instructions through a sequence of hardware steps often called the instruction cycle. At the lowest level, instructions are just binary numbers stored in memory, and the CPU has dedicated circuits that decode and execute them.

Programs written in languages like C++ are eventually compiled into machine code, which consists of binary instruction words. This binary value is stored in RAM. The CPU does not see "add". It only sees bit patterns. This binary value is stored in RAM. The CPU does not see "add". It only sees bit patterns.

Example (RISC-V style instruction):

```
add x9, x10, x20
```

After compilation this might look like:

```
0000000 10101 10100 000 01001 0110011
```

This layout of the instruction is called the instruction format. As you can see from counting the number of bits, this RISC-V instruction takes exactly 32 bits—a word
For the full RISC-V instruction set reference, see here

9 Base Instruction Formats

What is an instruction format? An **instruction format** defines how the 32 bits of a machine instruction are partitioned into *fields*—named groups of bits that the processor hardware reads to determine what operation to perform and which operands to use. Every RISC-V base instruction is exactly 32 bits wide. The key fields that appear across formats are:

- `opcode` (bits [6:0]): identifies the general category of the instruction (e.g., arithmetic, load, store, branch). Present in *every* format, always in the same position.
- `rd` (bits [11:7]): the **destination register** number (0–31) where the result is written.
- `funct3` (bits [14:12]): a secondary opcode that further specifies the operation (e.g., distinguishing `ADD` from `AND` within the same opcode group).
- `rs1` (bits [19:15]): the first **source register**.
- `rs2` (bits [24:20]): the second source register (only in formats that need two register sources).
- `funct7` (bits [31:25]): an additional function field used in R-type to distinguish operations that share the same `opcode` and `funct3` (e.g., `ADD` vs. `SUB`).
- `imm`: an **immediate** value (constant embedded in the instruction). Its width and position vary by format.

Why multiple formats? RISC-V uses several fixed 32-bit instruction *formats* so the hardware can decode quickly and consistently. Each format is chosen based on

1. the **addressing mode** (register vs. immediate vs. memory),

2. the **operation type** (ALU, branch, load/store, etc.). The slides classify the base formats as R, I, S, B, U, and J.

Types of Formats.

- **R-type (register-register). Bit layout:**

  `[funct7 | rs2 | rs1 | funct3 | rd | opcode]`,

  i.e. 7+5+5+3+5+7 = 32 bits.

  R-type instructions take two source registers (`rs1, rs2`) and write a result to a destination register (`rd`). The operation is fully specified by `opcode` + `funct3` + `funct7`. Because there is no immediate, all 32 bits are used to encode registers and function fields, giving the largest space for distinguishing operations.

  Examples: `ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU`.

- **I-type (immediate / loads). Bit layout:**

  `[imm[11:0] | rs1 | funct3 | rd | opcode]`,

  i.e. 12+5+3+5+7 = 32 bits.

  I-type instructions contain one source register (`rs1`), a destination register (`rd`), and a **12-bit signed immediate** (`imm[11:0]`), which is sign-extended to 32 bits before use. The immediate replaces the `rs2` and `funct7` fields from R-type.

  This format serves two purposes: immediate ALU ops (`ADDI, ORI, ANDI, XORI, SLTI`) and *loads* (`LW, LH, LB, LWU, ...`), where the immediate is added to `rs1` to form the memory address: `rd = Mem[rs1 + imm]`.

  Note: `JALR` also uses I-type format.

- **S-type (stores). Bit layout:**

  `[imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode]`.

  Stores write register data to memory, so there is no destination register `rd`. Instead, the 12-bit immediate is *split*: the upper 7 bits occupy bits [31:25] (where `funct7` sits in R-type) and the lower 5 bits occupy bits [11:7] (where `rd` sits in R-type). This splitting keeps `rs1` and `rs2` in their usual positions so the register file doesn't need extra muxing.

  Effective address: `Mem[rs1 + imm] = rs2`.
  Examples: `SW, SH, SB`.

- **B-type (branches). Bit layout:**

  `[imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode]`.

  B-type is a variant of S-type. It compares `rs1` and `rs2` and conditionally adds a signed offset to the PC. The immediate encodes a **13-bit signed offset in multiples of 2** (the lowest bit is always 0 and is not stored), giving a branch range of ±4 KiB from the current instruction.

  The bits are *shuffled* compared to S-type so that all immediate bits except the sign bit align with S-type—this simplifies hardware muxing between the two formats.

  Examples: `BEQ, BNE, BLT, BGE, BLTU, BGEU`.

- **U-type (upper immediate). Bit layout:**

  `[imm[31:12] | rd | opcode]`, i.e. 20+5+7 = 32 bits.

  U-type provides a **20-bit immediate** placed in the upper 20 bits of the result (bits [31:12]), with the lower 12 bits set to zero. It only needs `rd` and the immediate—no source registers.

  `LUI rd, imm` loads the upper immediate into `rd`.
  `AUIPC rd, imm` adds the upper immediate to the current PC and stores the result in `rd`.
  Together with an I-type `ADDI`, `LUI` can construct any arbitrary 32-bit constant in a register.

- **J-type (jump). Bit layout:**

  `[imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode]`.

  J-type encodes a **21-bit signed offset in multiples of 2** for PC-relative jumps, giving a range of ±1 MiB. Like B-type, the lowest bit is implicit. The immediate bits are permuted to maximize overlap with other formats and simplify the hardware sign-extension / muxing logic.

  `JAL rd, offset` stores `PC+4` in `rd` (the return address) and jumps to `PC + offset`.

Design insight: why the immediate bits are "shuffled." At first glance the scattered immediate fields in S/B/J-type look confusing, but the placement is deliberate: the sign bit is *always* in bit 31, and overlapping immediate bits share the same position across formats wherever possible. This means the hardware can use a single sign-extension unit and a small mux to reconstruct the immediate for any format—a key reason RISC-V decoding is simpler than in CISC architectures.

10 Compressed Instructions (RV32C / the "C" Extension)

Motivation. In standard RISC-V, every instruction is 32 bits wide. This is simple for hardware but wastes code memory: studies of real programs show that a large fraction of instructions use only small immediates, a handful of common registers, and a small set of frequent operations. The **C (compressed) extension** introduces 16-bit versions of the most common instructions, reducing code size by roughly 25–30%—comparable to ARM Thumb-2 or MIPS16.

A processor that implements RV32C can freely intermix 32-bit and 16-bit instructions in the same instruction stream. The hardware distinguishes them by looking at the lowest two bits: if `bits[1:0]` $\neq$ `11`, the instruction is 16 bits; if `bits[1:0] = 11`, it is 32 bits (or wider, for future extensions).

Key constraints of compressed instructions. Because only 16 bits are available, compressed instructions make trade-offs:

1. **Fewer registers.** Many C-format instructions can only address 8 registers (`x8`–`x15`, also known as `s0`–`s1` and `a0`–`a5`) using a 3-bit register specifier, rather than the full 32-register file.

2. **Smaller immediates.** Immediate fields are narrower (5 or 6 bits in many cases), so only small constants and short offsets can be encoded.

3. **Implicit operands.** Some formats hardwire a source or destination to a specific register. For example, `C.ADDI` implicitly uses the same register as both source and destination (`rd = rd + imm`), and stack-pointer-relative loads/stores implicitly use `x2` (the stack pointer).

4. **Restricted operations.** Only the most frequently occurring instructions have compressed equivalents—there is no compressed form of every R-type or I-type instruction.

Compressed instruction formats. The C extension defines its own set of 16-bit formats, analogous to the base 32-bit formats:

- **CR-type (compact register).** Two register operands, no immediate. Used for instructions like `C.ADD rd, rs2` and `C.MV rd, rs2`. Both register fields are the full 5 bits, so all 32 registers are accessible.
- **CI-type (compact immediate).** One register plus a small immediate. Used for `C.ADDI`, `C.LI`, `C.LUI`, and stack-pointer-relative loads like `C.LWSP`. The destination register is 5 bits (full range); the immediate is typically 5–6 bits.
- **CSS-type (compact stack-store).** Used for stack-pointer-relative stores like `C.SWSP`. The base register is implicitly `x2` (sp), so only `rs2` and the offset are encoded.
- **CIW-type (compact immediate wide).** Used for `C.ADDI4SPN`, which adds a scaled immediate to the stack pointer and writes the result to one of the 8 "popular" registers (`x8`–`x15`). The immediate field is 8 bits.
- **CL-type / CS-type (compact load / compact store).** Used for register-offset loads and stores like `C.LW` and `C.SW`. Both the base and data registers are limited to the 8-register subset, and the offset is a small scaled immediate.
- **CB-type (compact branch).** Used for `C.BEQZ` and `C.BNEZ`, which compare a single register (from the 8-register subset) against zero. Only equality/inequality with zero is supported—general two-register comparisons require the full 32-bit B-type.
- **CJ-type (compact jump).** Used for `C.J` and `C.JAL` (RV32 only). Encodes an 11-bit signed offset for short PC-relative jumps.

Mapping to base instructions. Every compressed instruction is defined as an *alias* for exactly one base 32-bit instruction. The assembler can automatically substitute the 16-bit form when the operands fit. For example:

- `C.ADD rd, rs2` $\equiv$ `ADD rd, rd, rs2`
- `C.LW rd', offset(rs1')` $\equiv$ `LW rd', offset(rs1')` (with restricted registers and offset range)
- `C.BEQZ rs1', offset` $\equiv$ `BEQ rs1', x0, offset`
- `C.J offset` $\equiv$ `JAL x0, offset`

This means the C extension adds no new architectural state or semantics—it is purely a code-size optimization. A program compiled with C-extension instructions will produce the same results as one using only base instructions.

Why the 8-register subset? The registers `x8`–`x15` were chosen because the RISC-V calling convention assigns the most frequently used roles to them: `s0`–`s1` (callee-saved), `a0`–`a5` (function arguments and return values). Profiling real code shows these 8 registers account for the vast majority of operand references, so restricting to them in compressed formats loses very little in practice.

Alignment implications. Without the C extension, all instructions are 4-byte aligned. With C enabled, instructions can be 2-byte aligned, which means the PC can take any even value. This has minor implications for instruction fetch logic (the fetcher must handle instructions that straddle cache-line or word boundaries) and for branch target addresses (which must be 2-byte aligned rather than 4-byte aligned).

11 Datapath Overview

Datapath. The **datapath** is the collection of hardware components and interconnections that perform the actual computation inside a processor. It includes

- functional units (ALU, adders)
- storage elements (register file, memory, pipeline registers)
- the multiplexers and wires that route data between them

Control Unit. The **control unit** is separate—it reads the instruction's opcode and function fields and generates the signals that tell the datapath *what* to do at each step.

Together, the datapath and control unit form the processor's **execution engine**. The datapath answers "with what hardware?" while the control unit answers "in what configuration?"

Single-cycle datapath. In the simplest implementation, every instruction executes in **one clock cycle**. The cycle must be long enough to accommodate the slowest instruction (typically `LW`, which must read the instruction memory, read the register file, compute an address in the ALU, access data memory, and write back).

The major components, in the order data flows through them:

1. **Program Counter (PC).** A register holding the address of the current instruction. At each clock edge it updates to either `PC+4` (sequential) or a branch/jump target.

2. **Instruction Memory.** Takes the PC as input and outputs the 32-bit instruction word. In a real system this is the instruction cache; in a teaching model it is often shown as a separate read-only memory.

3. **Register File.** Has two read ports (`rs1`, `rs2`) and one write port (`rd`). The 5-bit register specifiers from the instruction select which registers to read; the result is written at the end of the cycle if the `RegWrite` control signal is asserted.

4. **Immediate Generator (ImmGen).** Extracts and **sign-extends** the immediate field from the instruction. Because different formats scatter the immediate bits differently (recall I/S/B/U/J layouts), the ImmGen uses the opcode to select the correct extraction and sign-extension logic.

5. **ALU (Arithmetic Logic Unit).** Performs the actual computation: addition for `ADD`/`LW`/`SW` address calculation, subtraction for `SUB` and branch comparisons, bitwise operations for `AND`/`OR`/`XOR`, shifts, etc. One input always comes from `rs1`; the other comes from either `rs2` (R-type) or the immediate (I/S/B-type), selected by the **ALUSrc** multiplexer. A separate **ALU control** unit decodes `funct3`/`funct7` to choose the specific operation.

6. **Data Memory.** Used only by loads and stores. For `LW`, it reads from the address computed by the ALU; for `SW`, it writes the value from `rs2` to that address. Control signals `MemRead` and `MemWrite` enable the appropriate operation.

7. **Write-back multiplexer.** Selects what value is written to `rd`: the ALU result (R-type, I-type ALU), the memory read data (loads), or `PC+4` (for `JAL`/`JALR`). Controlled by the **MemtoReg** (or **WBSel**) signal.

8. **Branch/jump logic.** An adder computes `PC + immediate` for branch and jump targets. A multiplexer selects between `PC+4` and the branch target, controlled by the **PCSrc** signal, which is asserted when a branch condition is met or the instruction is an unconditional jump.

In short,

Program Counter → Instruction Memory → Register File → Immediate Generator → ALU → Data Memory → Write-back multiplexer → Branch/jump logic.

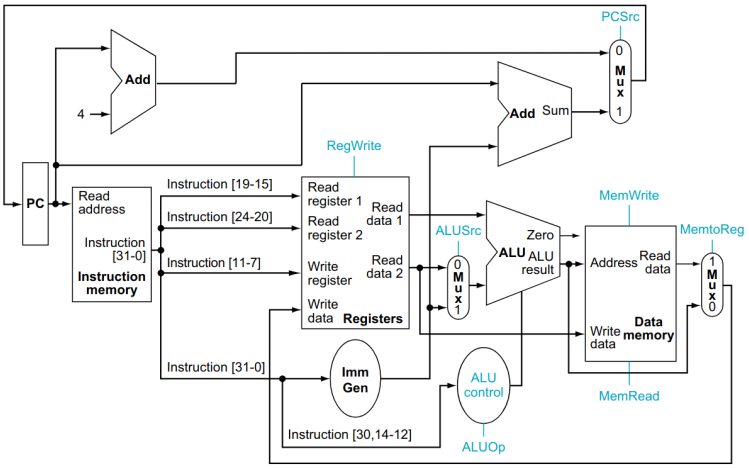


FIGURE 4.19 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added, which depends on the funct3 field and part of the funct7 field. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

Key control signals in the single-cycle datapath.

- `RegWrite`: enables writing to the register file.
- `ALUSrc`: selects the second ALU input (register vs. immediate).
- `ALUOp`: tells the ALU control what category of operation to perform.
- `MemRead` / `MemWrite`: enable data memory access.
- `MemtoReg` (or `WBSel`): selects the write-back source.
- `PCSrc`: selects the next PC value.
- `Branch`: combined with the ALU's zero/comparison output to determine `PCSrc`.

The control unit is essentially a lookup table: given the 7-bit `opcode` (and sometimes `funct3`), it outputs the correct combination of these signals.

Limitation of the single-cycle design. The clock period equals the delay of the longest instruction path. Every instruction—even a simple `ADD` that never touches data memory—takes this worst-case time. This wastes time and limits clock frequency.

MULTICYCLE IMPLEMENTATION

12 Motivation

The problem with single-cycle. In the single-cycle datapath, every instruction completes in one clock cycle. The clock period must be long enough for the *slowest* instruction—typically `LW`, which traverses instruction memory, the register file, the ALU, data memory, and the write-back mux. But a simple `ADD` never touches data memory at all, yet it pays the same long cycle time. This is wasteful.

The multicycle idea. Instead of doing everything in one long cycle, break each instruction into **multiple shorter steps**, where each step takes exactly one clock cycle. Different instructions take a *different number of steps* depending on what they need to do:

- `ADD`: might take 4 cycles (fetch, decode, execute, write-back).
- `LW`: might take 5 cycles (fetch, decode, ALU address calc, memory read, write-back).
- `SW`: might take 4 cycles (fetch, decode, ALU address calc, memory write).
- `BEQ`: might take 3 cycles (fetch, decode, compare and update PC).

The clock period is now set by the slowest *single step* (one ALU operation, or one memory access), not the slowest entire instruction. This is shorter than the single-cycle clock period, so faster instructions genuinely execute faster.

13 Key Differences from Single-Cycle

Shared functional units. In the single-cycle design, instruction memory and data memory are separate units (or at least separate ports), and there are multiple adders (one for ALU operations, one for `PC+4`, one for branch targets). The multicycle design **reuses** hardware across steps because only one step happens per cycle:

- A **single memory unit** serves both instruction fetch and data access—just not in the same cycle. This is possible because fetch happens in step 1 and data access happens in step 4 or 5, so they never overlap.
- A **single ALU** handles arithmetic operations, address calculation, `PC+4` computation, and branch target computation—in different cycles.

This reduces hardware cost compared to single-cycle but requires **multiplexers** at the ALU inputs and memory address input to select different sources in different cycles.

Internal registers between steps. In single-cycle, data flows combinationally from one component to the next within a single cycle. In multicycle, each step produces results that must be **saved** for use in a later step. This requires additional internal (non-architectural) registers that are *not* visible to the programmer:

- **Instruction Register (IR):** holds the instruction word after it is fetched in step 1, so subsequent steps can decode it.
- **Memory Data Register (MDR):** holds data read from memory (used by `LW` to hold the loaded value until write-back).
- **A and B registers:** hold the values read from the register file (`rs1` and `rs2`) after the decode step, so they are available in the execute step.
- **ALUOut register:** holds the ALU result from the execute step so it can be used in the memory or write-back step.

These are distinct from **pipeline registers** in a pipelined design: multicycle internal registers hold data for *one instruction at a time* because only one instruction is in flight. Pipeline registers hold data for *different instructions simultaneously*.

14 The Five Steps in Detail

Step 1: Instruction Fetch (IF).

- Read instruction from memory at the address in the PC: `IR = Mem[PC]`.
- Compute `PC + 4` using the ALU and store it back in the PC: `PC = PC + 4`.
- Both operations happen in the same cycle—the memory provides the instruction while the ALU computes the incremented PC.

Step 2: Instruction Decode / Register Fetch (ID).

- Decode the instruction held in IR: determine opcode, register specifiers, immediate.
- Read `rs1` and `rs2` from the register file and store the values in internal registers A and B.
- Sign-extend the immediate and compute `PC + imm` speculatively using the ALU (stored in ALUOut). This is done "just in case" the instruction turns out to be a branch—the branch target is ready early.
- Crucially, decoding and register reading happen **in parallel** because the register specifiers are in fixed bit positions and can be sent to the register file before the full decode is complete.

Step 3: Execution / Address Calculation (EX). What happens here depends on the instruction type:

- **R-type:** ALU performs the operation on A and B. Result goes to ALUOut. `ALUOut = A op B`.
- **I-type (ALU):** ALU performs the operation on A and the sign-extended immediate. `ALUOut = A op imm`.
- **Load/Store:** ALU computes the effective address. `ALUOut = A + imm`.
- **Branch:** ALU compares A and B (e.g., tests equality). If the branch condition is true, the PC is updated to the branch target that was speculatively computed in step 2: `PC = ALUOut` (from step 2). If not taken, the PC already holds `PC+4` from step 1.

For branch instructions, execution is **complete** after this step—no further cycles needed.

Step 4: Memory Access (MEM). Only executed by loads and stores:

- **Load:** read data memory at the address in ALUOut. Store the result in MDR. `MDR = Mem[ALUOut]`.
- **Store:** write the value in B to memory at the address in ALUOut. `Mem[ALUOut] = B`. Store is **complete** after this step.

For R-type and I-type ALU instructions, this step is **write-back** instead: write ALUOut to the destination register `rd`. These instructions are complete after this step (4 cycles total).

Step 5: Write-Back (WB). Only executed by load instructions:

- Write the value in MDR to the destination register `rd`: `Reg[rd] = MDR`.

This is why `LW` takes 5 cycles—it is the only instruction that needs all five steps.

15 Control Unit: Finite State Machine

Single-cycle control vs. multicycle control. The single-cycle control unit is purely **combinational**: given the opcode, it outputs all control signals at once because everything happens in one cycle. The multicycle control unit must be **sequential**: it needs to issue *different* control signals in each step and remember which step it is currently in.

This is implemented as a **finite state machine (FSM)**. Each state corresponds to one step of execution and sets the appropriate control signals for that step. Transitions between states depend on the instruction type:

- States 0 and 1 (IF and ID) are **shared** by all instructions—every instruction starts the same way.
- State 2 (EX) branches into different next-states depending on opcode: R-type goes to one state, load/store goes to another, branch is done.
- Subsequent states continue until the instruction is complete, then the FSM returns to state 0 to fetch the next instruction.

The FSM can be implemented with a ROM (lookup table) or with discrete combinational logic (using the current state and opcode as inputs to produce the next state and control signals). The ROM approach is called **microprogramming**: each state is essentially a "microinstruction" that specifies what the datapath should do in that cycle.

Microprogramming. In a microprogrammed control unit, the FSM is stored in a small control memory (microcode ROM). Each entry contains one microinstruction specifying all control signals plus the address of the next microinstruction. This makes the control unit very flexible—changing the behavior of an instruction only requires changing the microcode, not rewiring logic. Complex ISAs (like x86) rely heavily on microcode, while simple RISC designs (like RISC-V) often use hardwired control because the FSM is small enough.

16 Multicycle Control Signals

In the single-cycle design, each control signal is set once per instruction. In the multicycle design, the *same signal* may take different values in different steps. Some additional control signals are needed that do not exist in the single-cycle design:

- **IorD:** selects the memory address source—PC (for instruction fetch) or ALUOut (for data access).
- **IRWrite:** enables writing to the Instruction Register (only asserted in step 1).
- **ALUSrcA:** selects the first ALU input—PC (for address calculations in steps 1–2) or register A (for execution in step 3).
- **ALUSrcB:** selects the second ALU input among four options: register B, the constant 4, the sign-extended immediate, or the shifted immediate (for branch offset).

- **PCSource:** selects what value to write to the PC—ALU output (for `PC+4`), ALUOut register (for branch target), or a jump target.

- **PCWrite / PCWriteCond:** `PCWrite` unconditionally writes the PC (used in fetch and jumps). `PCWriteCond` writes the PC only if the ALU's comparison output indicates the branch is taken.

Note that `RegWrite`, `MemRead`, `MemWrite` still exist but are now asserted only in the specific step that needs them, not for the entire instruction.

17 Performance Comparison

Single-cycle:

$$\text{Execution time} = \text{IC} \times 1 \times T_{\text{long}}$$

where T_{long} is the worst-case instruction latency (the full path through all components).

Multicycle:

$$\text{Execution time} = \text{IC} \times \text{CPI}_{\text{avg}} \times T_{\text{step}}$$

where T_{step} is the duration of one step (the slowest *single* stage), and CPI_{avg} is the average number of cycles per instruction, weighted by instruction frequency.

If the instruction mix is 25% loads (5 cycles), 10% stores (4 cycles), 45% R-type (4 cycles), 15% branches (3 cycles), and 5% jumps (3 cycles):

$$\text{CPI}_{\text{avg}} = 0.25(5) + 0.10(4) + 0.45(4) + 0.15(3) + 0.05(3) = 4.05$$

The multicycle design wins if $\text{CPI}_{\text{avg}} \times T_{\text{step}} < 1 \times T_{\text{long}}$, which depends on the instruction mix and how much shorter T_{step} is compared to T_{long}.

18 Multicycle vs. Pipelined

- **Multicycle:** only one instruction is in the datapath at a time. Different instructions take different numbers of cycles. Hardware is shared across steps. No hazard issues (since there is no overlap between instructions).

- **Pipelined:** multiple instructions overlap—while one is in EX, the next is in ID, and the one after is in IF. The clock period is the same as multicycle (T_{step}), but the ideal CPI drops to 1 because one instruction *completes* every cycle at steady state. The cost is hazard detection/resolution logic, forwarding, and pipeline registers.

- The multicycle design is best understood as a **stepping stone** between single-cycle and pipelined: it introduces the idea of breaking execution into stages and reusing hardware, which directly leads to the pipelined design when you allow multiple instructions to occupy different stages simultaneously.

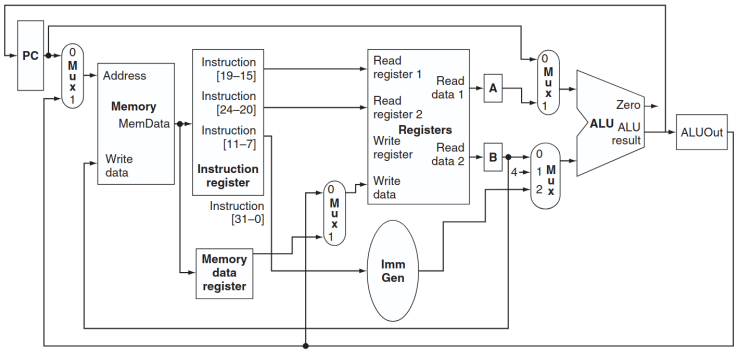


FIGURE e4.5.2 Multicycle datapath for RISC-V handles the basic instructions. Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several registers (IR, MDR, A, B, ALUOut), a multiplexor for the memory address, a multiplexor for the top ALU input, and expanding the multiplexor on the bottom ALU input into a four-way selector. These small additions allow us to remove two adders and a memory unit.

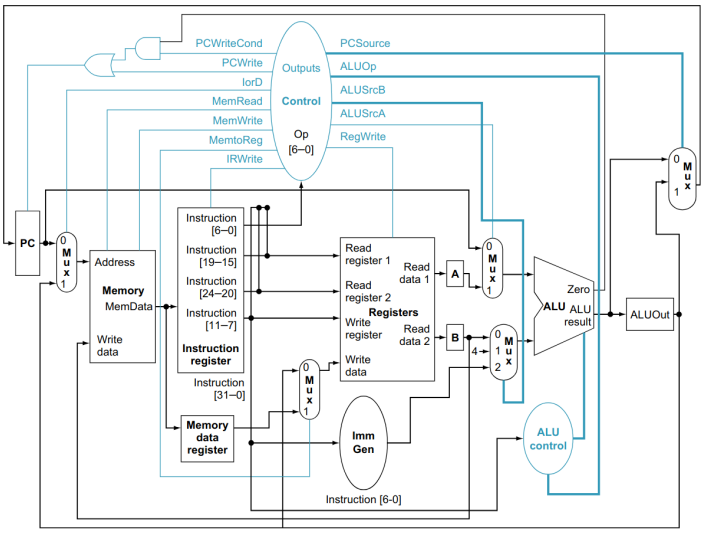


FIGURE e4.5.4 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure e4.5.3 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 4.29 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken.

DATA HAZARDS IN PIPELINED PROCESSORS

19 Data Hazards

Why hazards exist. A pipelined processor overlaps the execution of multiple instructions: while one instruction is in the Execute stage, the next is in Decode, and the one after that is being Fetched. This overlap improves throughput but creates **hazards**—situations where the pipeline cannot simply proceed at full speed because an instruction depends on the result of a previous instruction that has not yet completed.

A **data hazard** occurs specifically when an instruction needs to read or write a value that another in-flight instruction is also reading or writing. If the pipeline does nothing about it, the dependent instruction will read a stale (wrong) value from the register file.

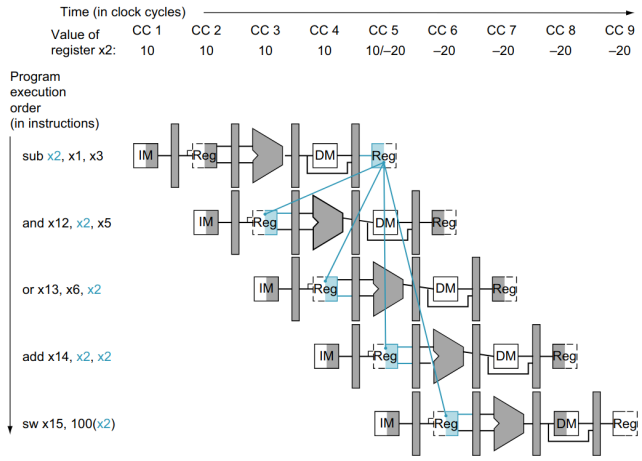


FIGURE 4.54 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into x2, and all the following instructions read x2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards*.

20 Types of data hazards

Data hazards are classified by the order of the conflicting read (R) and write (W) operations:

- **RAW (Read After Write)—true dependence.** Instruction i writes a register and a later instruction j reads it. If j reaches the Decode/read stage before i has written its result back, j reads the old value.

  Example:

  ```
  ADD  x1, x2, x3    # writes x1
  SUB  x4, x1, x5    # reads x1 -- needs the value ADD produces
  ```

  This is the most common and most important hazard in practice. It reflects a genuine data dependence in the program.

- **WAW (Write After Write)—output dependence.** Two instructions both write the same register. If the earlier instruction's write happens *after* the later instruction's write (due to pipeline timing), the register ends up with the wrong value.

  Example:

  ```
  ADD  x1, x2, x3    # writes x1
  MUL  x1, x4, x5    # also writes x1 -- must finish second
  ```

  WAW hazards are uncommon in simple 5-stage in-order pipelines (because instructions write back in order) but arise in *out-of-order* and *multi-cycle* pipelines where a later instruction might complete before an earlier one.

- **WAR (Write After Read)—anti-dependence.** A later instruction writes a register that an earlier instruction has not yet read.

  Example:

  ```
  SUB  x4, x1, x5    # reads x1
  ADD  x1, x2, x3    # writes x1 -- must not overwrite
                       before SUB reads
  ```

  Like WAW, this is rare in simple in-order pipelines (reads happen in Decode, writes in Write-Back, so the read always comes first). WAR hazards become relevant in out-of-order execution and in pipelines where operand read and write stages can be reordered.

Note: there is no "RAR" hazard—two reads of the same register never conflict.

21 Solutions to data hazards

1. **Stalling (pipeline bubbles).** The simplest approach: when the hardware detects a hazard, it *stalls* the dependent instruction by inserting one or more "bubble" cycles (NOPs) into the pipeline until the needed value is available. This is correct but costs performance because the pipeline sits idle during the stall.

   In the classic 5-stage pipeline (IF–ID–EX–MEM–WB), a RAW hazard between two back-to-back instructions can require up to **2 stall cycles** if no other technique is used, because the producing instruction does not write back until WB while the consuming instruction needs the value in ID or EX.

2. **Forwarding (bypassing).** Instead of waiting for the result to be written to the register file, the hardware *forwards* the result directly from the pipeline stage where it is first available (typically the end of EX or MEM) back to the input of the stage that needs it.

   Common forwarding paths in a 5-stage RISC-V pipeline:

   - **EX-to-EX forwarding:** the ALU result at the end of the EX stage is forwarded to the ALU input of the next instruction's EX stage. This eliminates the stall for back-to-back ALU instructions:

     ```
     ADD  x1, x2, x3    # result available end of EX
     SUB  x4, x1, x5    # needs x1 at start of EX --
                          forwarded, no stall
     ```

   - **MEM-to-EX forwarding:** the result at the end of the MEM stage is forwarded to the EX input of an instruction two cycles behind. This handles the case where one instruction separates the producer and consumer.

   Forwarding eliminates *most* RAW stalls but requires extra multiplexers and control logic in the datapath.

3. **Load-use hazard (the case forwarding cannot fully solve).** A `LW` instruction does not have its data until the *end of the MEM stage*, one stage later than an ALU instruction. If the very next instruction needs the loaded value, even EX-to-EX forwarding is too early—the data does not exist yet. This is the **load-use hazard**:

   ```
   LW   x1, 0(x2)     # data available end of MEM
   ADD  x4, x1, x5    # needs x1 at start of EX -- one cycle
                        too soon
   ```

   The hardware must insert **one stall cycle**, after which the loaded value can be forwarded from MEM to EX. This single-cycle penalty is sometimes called a **load delay slot**.

4. **Compiler scheduling (software reordering).** The compiler can reorder independent instructions to fill the gap after a load, avoiding the stall without any hardware cost:

   ```
   LW   x1, 0(x2)
   ADD  x6, x7, x8    # independent -- fills the load delay slot
   ADD  x4, x1, x5    # x1 now available via MEM-to-EX forwarding,
                        no stall
   ```

   Good compilers perform this **instruction scheduling** automatically. This is why code compiled with optimization (`-O2`) often has fewer pipeline stalls than unoptimized code.

5. **Register renaming (for WAW and WAR).** Out-of-order processors eliminate WAW and WAR hazards by mapping architectural registers to a larger pool of **physical registers**. Two instructions that both write `x1` are renamed to write different physical registers, removing the false dependence entirely. This is beyond the scope of most introductory pipeline discussions but is the standard technique in modern high-performance cores.

Detecting hazards in hardware. The pipeline control logic compares register specifiers across stages:

- If `rs1` or `rs2` of the instruction in ID matches the `rd` of an instruction in EX, MEM, or WB, a potential RAW hazard exists.
- The **hazard detection unit** checks whether forwarding can resolve it; if not (load-use case), it asserts a stall signal.
- A stall freezes the IF and ID stages (they re-fetch/re-decode the same instructions) and injects a bubble (NOP control signals) into EX.

An important edge case: writes to `x0` are ignored in RISC-V (`x0` is hardwired to zero), so the hazard detection logic must *not* trigger forwarding or stalls when `rd = x0`.

PIPELINING DATAPATH

What it is. Pipelining is a CPU optimization technique that increases instruction throughput by overlapping the execution of multiple instructions using a multi-stage pipeline, similar to an assembly line. It breaks instructions into steps (e.g., fetch, decode, execute) so that different hardware units work

on different instructions simultaneously

Idea. Break the single-cycle datapath into stages separated by **pipeline registers**. Each stage does one piece of work per cycle, and multiple instructions occupy different stages simultaneously, like an assembly line.

Stages of Pipelining. The classic RISC-V pipeline has five stages:

1. **IF (Instruction Fetch).** Read the instruction from memory at the address in the PC. Compute `PC+4` as the default next address. Both the instruction and `PC+4` are written into the **IF/ID pipeline register**.

2. **ID (Instruction Decode / Register Read).** Decode the instruction: extract `opcode`, `funct3`, `funct7`, register specifiers, and immediate. Read `rs1` and `rs2` from the register file. Generate control signals. All decoded information and control signals are written into the **ID/EX pipeline register**.

3. **EX (Execute / Address Calculation).** The ALU performs its operation: arithmetic for R/I-type, address computation for loads/stores, comparison for branches. The branch target address `PC + imm` is also computed here. Results are written into the **EX/MEM pipeline register**.

4. **MEM (Memory Access).** For loads, read data memory at the ALU-computed address. For stores, write to data memory. For all other instructions, this stage simply passes the ALU result through. Results are written into the **MEM/WB pipeline register**.

5. **WB (Write-Back).** Write the final result (ALU output or loaded data) back to the destination register `rd` in the register file.

Pipeline registers. The registers between stages (IF/ID, ID/EX, EX/MEM, MEM/WB) are critical: they hold *all* information an instruction needs in later stages—data values, register numbers, control signals, and the PC. Without them, data from one instruction would collide with data from the next. Each pipeline register is clocked on the rising edge, so all stages advance in lockstep every cycle.

An important detail: the control signals generated in ID must travel through every subsequent pipeline register alongside the data, because they are not needed until later stages. For instance, `MemRead` is generated in ID but used in MEM, so it must pass through the ID/EX and EX/MEM registers.

Forwarding paths in the pipelined datapath. To resolve RAW data hazards (discussed in the previous section), the datapath includes **forwarding multiplexers** at the ALU inputs. These muxes can select their input from three sources:

- The normal value from the ID/EX pipeline register (no hazard).
- The ALU result from the EX/MEM register (EX-to-EX forwarding).
- The result from the MEM/WB register (MEM-to-EX forwarding).

A **forwarding unit** compares the `rs1`/`rs2` fields of the instruction in EX with the `rd` fields of instructions in MEM and WB to decide which mux input to select.

Hazard detection unit. A separate piece of logic handles the **load-use hazard**. It compares the `rd` of a load instruction in EX with the `rs1`/`rs2` of the instruction in ID. If there is a match, it:

1. stalls the IF and ID stages (prevents the PC and IF/ID register from updating),

2. inserts a bubble into EX (sets all EX control signals to zero / NOP).

After one stall cycle, the loaded value reaches MEM/WB and can be forwarded normally.

Branch handling. In the basic pipelined datapath, the branch decision is made at the end of EX. By that point, two instructions have already entered the pipeline after the branch. If the branch is taken, those instructions must be **flushed** (their pipeline registers are zeroed out). This costs a **branch penalty** of 1–2 cycles depending on where the comparison is performed.

To reduce this penalty, some designs move the branch comparison earlier (into ID), add a dedicated comparator, and use branch prediction to guess the outcome before it is known. These optimizations trade hardware complexity for fewer wasted cycles.

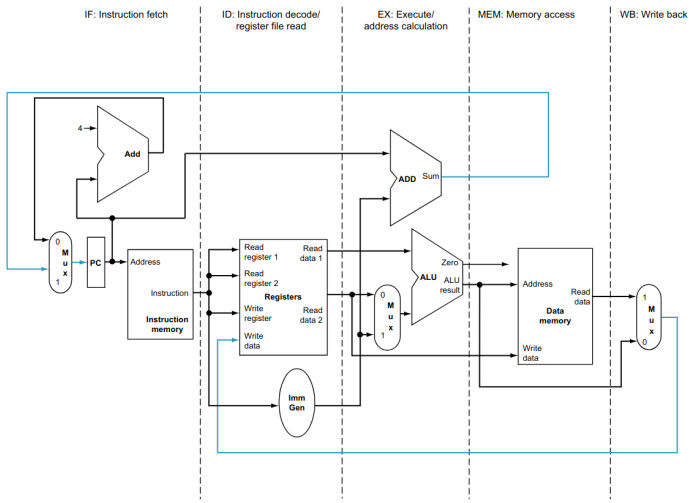


FIGURE 4.35 The single-cycle datapath from Section 4.4 (similar to Figure 4.21). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

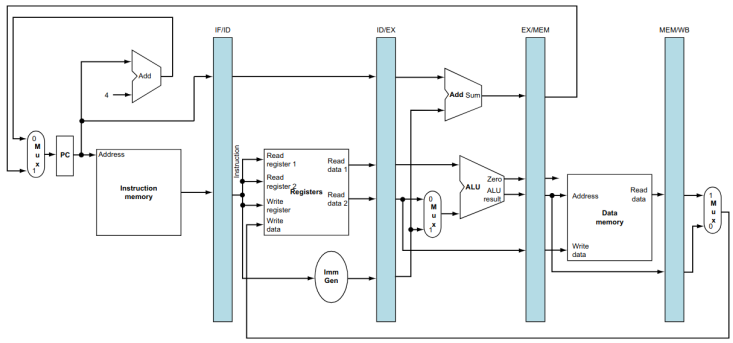


FIGURE 4.37 The pipelined version of the datapath in Figure 4.35. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 96 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 64-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 256, 193, and 128 bits, respectively.

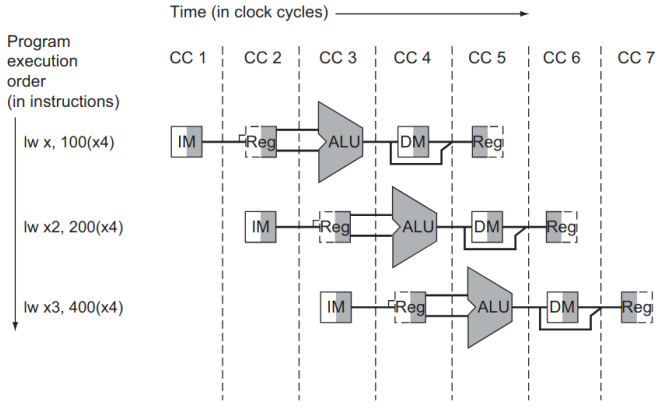


FIGURE 4.36 Instructions being executed using the single-cycle datapath in Figure 4.35, assuming pipelined execution. Similar to Figures 4.30 through 4.32, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.48. *IM* represents the instruction memory and the PC in the instruction fetch stage, *Reg* stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on. To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

22 Single-Cycle vs. Pipelined: Summary

- **Single-cycle:** simple to understand, one instruction per cycle, but the cycle time is dictated by the slowest instruction. CPI = 1, but clock period is large.

- **Pipelined:** ideally also CPI = 1 (one instruction *completes* per cycle at steady state), but the clock period is only as long as the slowest *stage*, which is much shorter. Throughput improves by roughly a factor equal to the number of stages ($\sim 5\times$), though hazards, stalls, and flushes reduce the ideal speedup in practice.
- Both execute the same instructions with the same ISA semantics—pipelining is a **microarchitectural** optimization invisible to software.

23 Control Hazards

What is a control hazard? A **control hazard** (also called a **branch hazard**) occurs when the pipeline does not know which instruction to fetch next because the outcome of a branch or jump has not yet been determined. In the classic 5-stage pipeline, the branch decision (taken or not taken) and the branch target address are computed in the EX stage. But by the time EX finishes, the pipeline has already fetched and begun decoding the *next one or two instructions* after the branch—instructions that may turn out to be wrong if the branch is taken.

If the branch is taken and the pipeline fetched the wrong instructions, those instructions must be **flushed** (squashed), and the pipeline restarts at the correct target address. The wasted cycles are called the **branch penalty**.

Branch penalty depends on pipeline depth. In the basic 5-stage pipeline where the branch resolves at the end of EX, the penalty is **2 cycles** (the IF and ID stages each contain a wrong instruction). If the hardware is modified to resolve branches earlier—say in ID by adding a dedicated comparator—the penalty drops to **1 cycle**. In deeper pipelines used by modern processors (10–20+ stages), the branch penalty can be 10–20 cycles, making branch prediction critical.

Why control hazards matter. Branches are common: roughly 15–25% of instructions in typical programs are branches. If every branch caused a 2-cycle stall, the effective CPI would increase significantly. For a 20% branch frequency and a 2-cycle penalty:

$$\text{CPI} = 1 + 0.20 \times 2 = 1.4$$

That is a 40% performance loss from branches alone. Reducing or hiding this penalty is essential.

24 Simple Strategies (No Prediction)

Stall until resolved. The simplest approach: when a branch is detected in ID, freeze the pipeline and do not fetch any new instructions until the branch outcome is known. This guarantees correctness but pays the full branch penalty on *every* branch, whether taken or not. This is rarely used in practice because the cost is too high.

Flush on taken (assume not taken). Fetch the instruction sequentially after the branch (i.e., assume the branch is **not taken**). If the branch turns out to be not taken, the pipeline continues with no penalty—the fetched instruction was correct. If the branch is taken, flush the incorrectly fetched instructions and redirect the PC to the branch target.

This is a simple form of **static prediction**: always predict not taken. The penalty is paid only when the prediction is wrong (i.e., when the branch is actually taken). For programs where roughly half of branches are taken, this cuts the average penalty roughly in half compared to always stalling.

Delayed branching. The architecture defines a **branch delay slot**: the instruction immediately after the branch is *always executed*, regardless of whether the branch is taken. The compiler is responsible for filling this slot with a useful instruction (ideally one that needs to execute regardless of the branch direction). If no such instruction can be found, the compiler inserts a NOP.

This was used in MIPS and early SPARC. RISC-V chose **not** to include delay slots because they complicate the ISA, make it harder to change the pipeline depth in future implementations, and modern branch prediction largely makes them unnecessary.

25 Static Branch Prediction

Static prediction strategies are fixed rules that do not adapt based on program behavior at runtime:

- **Always predict not taken:** as described above. Simple, zero hardware cost. Works well for branches that are rarely taken (e.g., error-checking branches).
- **Always predict taken:** always fetch from the branch target. This sounds counterintuitive, but in practice many branches *are* taken (e.g., loop-back branches). However, there is a subtlety: the target address may not be known until after decode, so the pipeline may still need to stall for one cycle to compute the target even if it "predicts taken."
- **Backward taken, forward not taken (BTFNT):** predict **taken** if the branch target is at a *lower* address (backward branch—likely a loop), and **not taken** if the target is at a higher address (forward branch—likely an if-statement skip). This heuristic captures the common case that loops iterate many times, so the backward branch at the bottom of the loop is almost always taken. BTFNT is simple and surprisingly effective, achieving roughly 65–70% accuracy.
- **Compiler hints:** some ISAs allow the compiler to encode a prediction bit in the branch instruction based on profiling data. RISC-V does not currently use this approach, but it has been used in other architectures (e.g., PowerPC).

Static prediction is cheap but limited: it cannot adapt to the actual runtime behavior of individual branches, which often varies depending on input data.

26 Dynamic Branch Prediction

Dynamic predictors use **hardware tables** that record the history of past branch outcomes and use that history to predict future behavior. These adapt at runtime and can achieve very high accuracy (95–99%).

1-bit predictor. The simplest dynamic predictor: maintain a table indexed by the low-order bits of the branch's PC address. Each entry stores a single bit recording whether the branch was taken or not taken *last time* it executed.

Prediction rule: predict the same outcome as last time.

Problem: a branch that is taken 9 out of 10 times (e.g., a loop that iterates 10 times) will mispredict **twice per loop invocation**—once when the loop exits (taken → not taken), and once on re-entry (the bit was flipped to not-taken, but the loop starts taking again). The predictor "overreacts" to a single deviation.

2-bit saturating counter. To fix the overreaction problem, use a **2-bit counter** per entry with four states:

- **00: Strongly Not Taken**
- **01: Weakly Not Taken**
- **10: Weakly Taken**
- **11: Strongly Taken**

The prediction is based on the MSB of the counter: if bit 1 is 1, predict taken; if 0, predict not taken. When a branch is taken, the counter increments (saturating at 11). When not taken, it decrements (saturating at 00).

The key advantage: a single misprediction only moves the counter one step, so a branch that is usually taken (sitting at 11) will move to 10 (weakly taken) on one deviation, but still predict taken next time. It takes **two consecutive** mispredictions to actually change the prediction direction. This reduces the loop-exit problem to one misprediction per invocation instead of two.

Branch History Table (BHT). A practical implementation uses a table of 2-bit counters indexed by the low-order bits of the branch PC. This is called a **Branch History Table** (or **Pattern History Table**). A typical BHT might have 1024 or 4096 entries.

Because the index uses only the low bits of the PC, different branches can map to the same entry (**aliasing**). This can cause interference, but in practice the table is large enough that aliasing is infrequent, and even when it occurs, the 2-bit counter provides some resilience.

Correlating predictors (two-level predictors). Some branches are correlated with the outcomes of *other* recent branches. For example:

```
if (x == 0) ...     # branch A
if (x > 0) ...      # branch B -- outcome depends on A
```

A simple BHT indexed only by the branch's own PC cannot capture this correlation. A **correlating predictor** (also called a two-level predictor) addresses this by incorporating **global branch history**—a shift register that records the taken/not-taken outcomes of the last n branches. The BHT is then indexed by a combination of the branch PC and the global history register, so the same branch in different contexts maps to different counter entries.

An (m, n) correlating predictor uses m bits of global history and n-bit counters. For example, a (2, 2) predictor uses the last 2 branch outcomes plus 2-bit saturating counters, with the global history selecting among $2^2 = 4$ different sets of counters for each branch.

Tournament predictors. Rather than choosing a single prediction scheme, a **tournament predictor** combines multiple predictors and dynamically selects the one that has been more accurate for each branch. A typical design uses:

- A **local predictor** that tracks each branch's own history (good for branches with regular patterns like loop counters).
- A **global predictor** that uses global branch history (good for correlated branches).
- A **selector** (another table of 2-bit counters) that learns which predictor is more accurate for each branch and follows its prediction.

Tournament predictors achieve very high accuracy and were used in processors like the Alpha 21264 and many modern designs.

27 Branch Target Buffer (BTB)

Prediction accuracy is only half the problem. Even if you correctly predict *taken*, you still need to know *where* to fetch from—the branch target address. Computing the target requires decoding the instruction and adding the offset, which takes at least one cycle.

A **Branch Target Buffer (BTB)** is a cache that maps branch PC addresses to their **target addresses**. It is consulted during the IF stage, before the instruction is even decoded:

1. The fetch unit sends the current PC to the BTB.

2. If the BTB has an entry for this PC (a hit), it provides the predicted target address and the direction prediction.

3. If the prediction is "taken," the fetch unit immediately redirects to the target address *in the very next cycle*—no decode needed.

4. If the BTB misses (the instruction is not a branch, or is a branch seen for the first time), fetch continues sequentially.

The BTB effectively converts the branch penalty to **zero cycles** for correctly predicted taken branches, because the redirect happens during IF with no gap. The penalty is only paid on mispredictions, first-time branches, or BTB misses.

28 Return Address Stack (RAS)

Function returns (e.g., `JALR x0, 0(x1)` in RISC-V) are indirect branches whose target changes depending on where the function was called from. A BTB would only remember the *most recent* return address, which is wrong if the function is called from multiple sites.

A **Return Address Stack (RAS)** is a small hardware stack (typically 8–32 entries) that:

- **Pushes** the return address (`PC+4`) when a `JAL` (function call) is detected during fetch.
- **Pops** the top entry as the predicted target when a return instruction is detected.

Because function calls and returns are naturally nested (LIFO order), the RAS achieves very high prediction accuracy for returns—often above 95%. It fails only when the stack overflows (deeply nested or recursive calls) or when non-standard control flow (e.g., `setjmp/longjmp`) breaks the call-return pairing.

29 Misprediction Recovery

When the actual branch outcome disagrees with the prediction, the processor must:

1. **Flush** all instructions that were fetched along the wrong path. In a simple 5-stage pipeline, this means squashing 1–2 instructions in IF/ID. In a deep out-of-order pipeline, this can mean discarding dozens of speculatively executed instructions.

2. **Redirect** the PC to the correct address (the branch target if the branch was mispredicted as not-taken, or the fall-through if mispredicted as taken).

3. **Restore** the processor state to what it was at the branch. In out-of-order processors, this involves restoring the register rename map and discarding speculative register writes. Mechanisms for this include checkpoint-based recovery (snapshot the rename table at each branch) and reorder-buffer-based recovery (walk back the ROB to undo speculative results).

4. **Update** the prediction tables (BHT, BTB, etc.) with the correct outcome so future predictions improve.

The cost of misprediction is directly proportional to how deep the pipeline is and how many instructions were speculatively executed down the wrong path. This is why prediction accuracy is so critical in modern processors—even a 95% accurate predictor wastes significant cycles if the pipeline is 15+ stages deep and issues 4+ instructions per cycle.

30 Putting It All Together

The branch prediction subsystem in a modern processor typically combines all of these mechanisms:

- A **BTB** identifies branches during fetch and provides target addresses.
- A **direction predictor** (tournament or correlating) predicts taken vs. not taken.
- A **RAS** predicts return targets.
- The results feed into the fetch unit so that, in the common case, the pipeline never stalls for branches at all—the correct instruction stream flows in continuously. Penalties are paid only on mispredictions, which for well-tuned predictors occur on fewer than 2–5% of branches.

SCOREBOARDING

31 Motivation

The problem with in-order execution. In a simple pipelined processor, instructions are issued and executed strictly in **program order**. If an instruction stalls (e.g., waiting for a slow memory access or a long-latency operation like division), every instruction behind it also stalls—even if those later instructions are completely independent and could execute immediately. This wastes functional unit cycles and limits performance.

The idea. Scoreboarding is a technique for **out-of-order execution**: it allows independent instructions to proceed past a stalled instruction, keeping functional units busy as much as possible. It was first implemented in the CDC 6600 (1964), one of the earliest supercomputers.

The key insight is that the pipeline can *issue* and *execute* instructions out of order as long as the results are still correct—meaning all true data dependences (RAW hazards) are respected. The **scoreboard** is a centralized hardware structure that tracks the status of every instruction, every functional unit, and every register, and uses this information to decide when each instruction can safely proceed to the next step.

32 Hardware Assumptions

A scoreboarded processor typically has:

- **Multiple functional units:** several ALUs, a multiplier, a divider, a floating-point adder, etc. This is essential—out-of-order execution is only useful if there are multiple units that can work in parallel.
- **A single issue point:** instructions are still *fetched and issued* in program order (one at a time), but they may *execute and complete* out of order.
- **No forwarding/bypassing:** the original CDC 6600 scoreboard did not use forwarding. Results are written to the register file, and dependent instructions read from the register file. This simplifies the scoreboard logic but means some stalls that a forwarding pipeline would avoid still occur.

33 The Four Stages

Every instruction passes through four stages managed by the scoreboard. The scoreboard decides when each instruction is allowed to advance from one stage to the next.

1. **Issue (decode and check structural hazards).** The scoreboard checks whether the required functional unit is **available** and whether there is a **WAW hazard** (another in-flight instruction writing to the same destination register). If the functional unit is free and there is no WAW conflict, the instruction is issued to that unit. Otherwise, the instruction stalls at issue, and no further instructions can issue behind it (issue is in-order).

When issued, the scoreboard records: which functional unit the instruction uses, which registers it reads (`rs1`, `rs2`) and writes (`rd`), and which other instructions (if any) will produce the source operands it needs.

2. **Read Operands (check RAW hazards).** The scoreboard checks whether the source registers are **currently available**—meaning no earlier issued instruction is still going to write them. If both operands are ready, the functional unit reads them from the register file and begins execution. If one or both operands are not yet available, the instruction **waits** at this stage (without blocking other functional units).

This is where RAW hazards are resolved: the instruction simply does not read its operands until the producing instruction has written its result.

3. **Execution (operate).** The functional unit performs the computation. This may take one cycle (simple ALU) or many cycles (divide, floating-point multiply). The scoreboard is notified when execution completes. Other instructions can issue, read operands, and execute in parallel during this time—this is where the out-of-order benefit comes from.

4. **Write Result (check WAR hazards).** When execution finishes, the instruction is ready to write its result to the register file. But first, the scoreboard checks for **WAR hazards**: is there an earlier-issued instruction that has *not yet read* a register that this instruction is about to *overwrite*? If so, the write is delayed until that earlier instruction reads its operand.

Once safe, the result is written to the register file, the scoreboard updates its tables to reflect that the destination register is now available, and any instruction waiting on this result (stalled at Read Operands) can proceed.

34 Scoreboard Data Structures

The scoreboard maintains three tables:

1. Instruction Status Table. Tracks which of the four stages each in-flight instruction is currently in. This is used for overall coordination and to determine when the pipeline is done with an instruction.

2. Functional Unit Status Table. One row per functional unit, with the following fields:

- `Busy`: whether the unit is currently in use.
- `Op`: the operation being performed.
- `Fi`: the destination register (`rd`).
- `Fj, Fk`: the source registers (`rs1`, `rs2`).
- `Qj, Qk`: which functional unit will produce the value for `Fj` and `Fk` (if they are not yet available). This is how the scoreboard tracks which instruction each source operand depends on.
- `Rj, Rk`: flags indicating whether `Fj` and `Fk` are ready (already available in the register file). When both are "yes," the instruction can proceed to Read Operands.

3. Register Result Status Table. One entry per register, recording which functional unit (if any) will write to that register. If the entry is blank, the register is available. This table is used to detect WAW hazards at issue time and to determine operand availability at read-operand time.

35 Example Walkthrough

Consider these instructions with two functional units (an adder and a multiplier):

```
MUL  x1, x2, x3     # long-latency multiply
ADD  x4, x1, x5     # depends on MUL result (RAW on x1)
SUB  x6, x7, x8     # independent of both
```

- Cycle 1: `MUL` issues to the multiplier. Scoreboard records that x1 will be produced by the multiplier.
- Cycle 2: `ADD` issues to the adder. Scoreboard notes that `Fj = x1` is not yet ready (`Qj` = multiplier, `Rj` = no). `ADD` stalls at Read Operands.
- Cycle 3: `SUB` issues to the adder—wait, the adder is occupied by `ADD`. If there is a second ALU, `SUB` issues there. If not, `SUB` stalls at Issue.
- Meanwhile, `MUL` is executing (takes many cycles). `SUB` (if issued) reads its operands immediately (x7 and x8 are ready) and executes—completing *before* `ADD`, even though it was fetched later. This is out-of-order execution.
- When `MUL` finishes and writes x1, the scoreboard marks x1 as available. `ADD` can now read operands and proceed.

36 Limitations of Scoreboarding

- **In-order issue:** instructions are still issued in program order. If one instruction cannot issue (structural or WAW hazard), everything behind it stalls. This limits the amount of parallelism the hardware can exploit.
- **WAR hazards cause real stalls:** because there is no register renaming, a WAR hazard forces the writing instruction to wait. In the example above, if `SUB` wrote to x5 and `ADD` hadn't yet read x5, `SUB` would have to delay its write. These are "false" dependences that waste cycles.
- **WAW hazards prevent issue:** if two instructions write the same register, the second cannot issue until the first finishes. Again, register renaming would eliminate this.
- **No forwarding:** results must go through the register file, adding latency compared to designs with bypass paths.
- **Limited instruction window:** the scoreboard only tracks instructions that have been issued, and issue is in-order, so the "window" of instructions it can examine for parallelism is relatively small.

37 Scoreboarding vs. Tomasulo's Algorithm

Tomasulo's algorithm (developed for the IBM 360/91) addresses the main limitations of scoreboarding by adding **register renaming** and **reservation stations**:

- WAR and WAW hazards are eliminated entirely through renaming, so they never cause stalls.
- Results are forwarded directly from functional units to waiting reservation stations via a **Common Data Bus (CDB)**, without going through the register file first.
- Instructions wait at reservation stations rather than at a centralized scoreboard, distributing the control logic.

Scoreboarding is simpler to understand and implement, and serves as the conceptual foundation for Tomasulo's more powerful approach. Most modern out-of-order processors use techniques descended from Tomasulo's algorithm rather than pure scoreboarding.

TOMASULO'S ALGORITHM

38 Overview

What problem does it solve? Scoreboarding allows out-of-order execution but still stalls on WAR and WAW hazards because instructions read and write the same architectural registers. These are **false dependences**—they arise from register reuse, not from any real data flow between instructions. Tomasulo's algorithm, developed by Robert Tomasulo for the IBM 360/91 floating-point unit (1967), eliminates false dependences through **register renaming**, allowing more instructions to execute in parallel.

The key ideas.

1. **Register renaming** eliminates WAR and WAW hazards entirely. Instead of tracking architectural register names, the hardware renames them to internal "tags" that uniquely identify each result. Two instructions that both write `x1` get different internal tags, so they do not conflict.

2. **Reservation stations** replace the centralized scoreboard. Each functional unit has a small set of reservation station entries where instructions wait for their operands. The operands are stored *in the reservation station itself* (or replaced by a tag indicating which result to wait for), so the instruction no longer depends on the register file once issued.

3. **Common Data Bus (CDB)** broadcasts results from functional units to all reservation stations simultaneously. Any reservation station waiting for a particular result grabs it directly off the bus—no need to go through the register file first. This is a form of forwarding/bypassing.

39 Data Structures

Reservation stations. Each entry holds one issued instruction and contains:

- `Op`: the operation to perform.
- `Vj, Vk`: the **values** of the source operands (once available).
- `Qj, Qk`: the **tags** identifying which reservation station will produce each source operand. If `Qj` is empty, it means `Vj` already holds the actual value. If `Qj` is set to some tag, the instruction is still waiting for that result.
- `Busy`: whether this entry is occupied.

The critical insight: once an instruction copies its operand value into `Vj/Vk` (or records the tag in `Qj/Qk`), it no longer references the architectural register at all. The register can be freely overwritten by a later instruction without causing a WAR hazard, because the earlier instruction already has its own copy of the value.

Register status table (register alias table). One entry per architectural register. Each entry records which reservation station (if any) will produce the next value for that register. This is how renaming works: when an instruction issues and writes register `x1`, the table is updated to say "the value of x1 will come from reservation station X." Subsequent instructions that read `x1` will record tag X in their `Qj/Qk` rather than reading the register file.

40 The Three Stages

Unlike scoreboarding's four stages, Tomasulo's algorithm uses three:

1. **Issue (dispatch).** Fetch the next instruction from the instruction queue (in program order). Check whether a reservation station in the appropriate functional unit is **free**. If so:

- Allocate the reservation station entry.
- Read operands from the register file if available, storing them in `Vj/Vk`. If an operand is not yet available (the register status table says it will be produced by some reservation station), record that tag in `Qj/Qk` instead.
- Update the register status table: mark this reservation station as the producer of the destination register.

If no reservation station is free, the instruction stalls (structural hazard). Note: there is **no WAW stall**—even if a previous instruction writes the same destination register, the register status table simply updates to point to the *new* producer, and the old instruction's result will still complete and broadcast but will not update the register file (because the register no longer points to it).

2. **Execute.** When both `Qj` and `Qk` are clear (both operands available in `Vj` and `Vk`), the functional unit begins execution. Instructions can begin execution **out of order**—whichever reservation station has its operands ready first goes first.

While waiting, the reservation station monitors the CDB. When a result is broadcast with a matching tag, the station captures the value into `Vj` or `Vk` and clears the corresponding `Q` field. This happens in the same cycle the result is broadcast.

3. **Write Result.** When execution finishes, the functional unit broadcasts the result and its tag on the **Common Data Bus**. Three things happen simultaneously:

- Every reservation station whose `Qj` or `Qk` matches the tag captures the value (resolving RAW dependences).
- The register file is updated if the register status table still points to this reservation station (i.e., no later instruction has "re-renamed" the same register).
- The reservation station entry is freed.

There is **no WAR check** needed at write time—unlike scoreboarding. By the time a result is written, any earlier instruction that needed the old register value has already copied it into its reservation station. The old value is safe.

41 How Register Renaming Eliminates False Dependences

Consider this sequence:

```
MUL  x1, x2, x3
ADD  x4, x1, x5
SUB  x1, x6, x7    # WAW with MUL, WAR with ADD (both on x1)
```

In a scoreboard, `SUB` cannot issue until `MUL` finishes (WAW on x1), and `SUB` cannot write its result until `ADD` reads x1 (WAR). Both stalls are caused by reuse of the name `x1`, not by any real data flow.

In Tomasulo's algorithm:

- `MUL` issues, and the register status table records "x1 ← RS_mul."
- `ADD` issues, sees that x1 is pending from RS_mul, so it records `Qj = RS_mul` in its reservation station. It does *not* depend on the register file entry for x1 anymore.
- `SUB` issues freely. The register status table updates to "x1 ← RS_sub," overwriting the MUL entry. This is fine because ADD already captured its dependence on MUL via the tag, not the register name.
- When MUL finishes and broadcasts its result, ADD's reservation station grabs it. The register file for x1 is *not* updated because the status table now points to RS_sub, not RS_mul.
- SUB can execute and write x1 whenever ready—no WAW or WAR stall.

All three instructions can execute with **zero false-dependence stalls**. Only the true RAW dependence (ADD waiting for MUL's result) causes any waiting, and that is resolved as soon as MUL broadcasts.

42 Limitations

- **In-order issue:** like scoreboarding, instructions are still issued in program order. If the instruction queue is blocked (no free reservation station), later independent instructions cannot issue.
- **Single CDB bottleneck:** in the original design, there is one Common Data Bus. If two functional units finish in the same cycle, one must wait. Modern designs use multiple result buses to alleviate this.
- **Imprecise exceptions:** because instructions complete out of order, an exception (e.g., arithmetic overflow) may be detected *after* later instructions have already written results. This makes it difficult to recover to a precise program state. Modern processors solve this with a **Reorder Buffer (ROB)**—a queue that forces instructions to *commit* (make results architecturally visible) in program order, even though they *execute* out of order. The ROB effectively adds a fourth stage: Issue → Execute → Write Result (to ROB) → Commit (to register file in order).
- **Complexity:** the associative matching logic (every reservation station checking its tags against every CDB broadcast every cycle) requires content-addressable comparators, which are expensive in area and power.

43 Scoreboarding vs. Tomasulo: Summary

- **WAR/WAW hazards:** scoreboarding stalls on them; Tomasulo eliminates them through renaming.
- **Operand storage:** scoreboarding reads operands from the register file when ready; Tomasulo stores operands in reservation stations.
- **Result distribution:** scoreboarding writes results to the register file, then dependents read; Tomasulo broadcasts on the CDB, and dependents capture directly (bypassing the register file).
- **Control:** scoreboarding is centralized (one scoreboard tracks everything); Tomasulo is distributed (each reservation station tracks its own operands).
- **Outcome:** Tomasulo extracts more parallelism and achieves higher throughput, at the cost of more complex hardware.

SUPERSCALAR PROCESSORS

44 Motivation

The limits of a scalar pipeline. In a standard pipelined processor, the best achievable throughput is one instruction per cycle (CPI = 1). In practice, hazards push the CPI above 1. But even if every hazard were eliminated, the pipeline still completes at most one instruction per cycle. To go faster, we need to **issue multiple instructions per cycle**.

What is superscalar? A **superscalar** processor fetches, decodes, and issues **more than one instruction per cycle**. A 2-wide superscalar can issue up to 2 instructions per cycle; a 4-wide can issue up to 4. The ideal CPI becomes $1/n$ for an n-wide machine (or equivalently, the ideal IPC—instructions per cycle—becomes n). Modern high-performance processors are typically 4–8 wide.

This is distinct from pipelining. Pipelining overlaps different *stages* of different instructions (fetch one while decoding another). Superscalar overlaps different *instructions in the same stage* (fetch two instructions simultaneously, decode two simultaneously, etc.).

45 How It Works

Widened pipeline stages. Every stage of the pipeline is replicated or widened:

- **Fetch:** read multiple instructions from the instruction cache per cycle. This requires a wider memory interface (e.g., reading 8 or 16 bytes at once). Branch prediction becomes critical because the fetch unit needs to know which instructions to grab—if a branch is among the fetched group, the predictor must redirect mid-fetch.
- **Decode:** decode multiple instructions in parallel, determine dependences between them, and decide which can issue together. This is the most complex part—the hardware must check for hazards *within* the same group of instructions, not just across pipeline stages.
- **Issue / dispatch:** send multiple instructions to functional units. This requires enough functional units to absorb the instructions—a 4-wide machine might need 2 ALUs, 1 load/store unit, and 1 branch unit, for example.
- **Execute:** multiple functional units operate in parallel.
- **Write-back / commit:** multiple results are written back per cycle.

In-order vs. out-of-order superscalar. A superscalar processor can be either:

- **In-order superscalar:** instructions are issued in program order, multiple at a time, but only if they are independent. If the second instruction depends on the first, only the first issues that cycle. Simpler but limited—dependences frequently prevent full-width issue.
- **Out-of-order superscalar:** combines superscalar issue with Tomasulo-style out-of-order execution. Instructions are dispatched to reservation stations and execute as soon as their operands are ready, regardless of program order. A reorder buffer ensures they commit in order. This is what virtually all modern high-performance processors use (e.g., Apple M-series, AMD Zen, Intel Core).

46 Challenges

Dependence detection within a group. In a scalar pipeline, hazard detection compares the instruction in decode with instructions in later stages. In a superscalar, the hardware must *also* check for dependences *among* the instructions being issued in the same cycle. For a 4-wide machine, this means checking all six pairwise combinations every cycle—and the logic grows quadratically with issue width.

Register renaming at scale. An out-of-order superscalar must rename multiple destination registers per cycle. If two instructions in the same issue group both write `x1`, the renamer must assign them different physical registers and ensure the second rename "sees" the first. This requires the rename logic to handle cascading dependences within a single cycle.

Functional unit balance. Issuing 4 instructions per cycle is only useful if there are enough functional units to execute them. But adding units costs area and power. Designers must balance the mix of units (ALUs, multipliers, load/store units, branch units) against the expected instruction mix of real programs.

Branch prediction pressure. With multiple instructions fetched per cycle, the processor encounters branches more frequently per cycle. A single misprediction still flushes the pipeline, but now the flush discards more in-flight work because more instructions were issued speculatively. High prediction accuracy becomes even more critical.

Memory bandwidth. Fetching and committing multiple instructions per cycle demands higher bandwidth from the instruction cache and data cache. The register file also needs more read and write ports to service multiple simultaneous operations, which increases its size and access latency.

47 Superscalar vs. VLIW

An alternative approach to multiple issue is **VLIW (Very Long Instruction Word)**, where the *compiler* (not the hardware) decides which instructions execute in parallel:

- **Superscalar:** the hardware dynamically detects parallelism at runtime. The compiler produces a normal sequential instruction stream, and the processor figures out what can run in parallel. This is flexible and handles unpredictable behavior well but requires complex hardware.
- **VLIW:** the compiler packs multiple operations into a single wide instruction word. Each "slot" in the word corresponds to a specific functional unit. The hardware is simple—it just executes whatever the compiler put in each slot. But the compiler must find enough parallelism to fill the slots, and if it cannot, it inserts NOPs, wasting issue slots. VLIW also suffers from code compatibility issues: a program compiled for a 4-wide VLIW does not run on a 6-wide VLIW without recompilation.

VLIW is used in some DSPs and embedded processors (e.g., TI C6000 series) and was attempted in general-purpose computing (Intel Itanium / IA-64). Superscalar dominates general-purpose processors today because it does not require the compiler to predict runtime behavior.

48 Practical Limits of Superscalar

Increasing issue width gives diminishing returns. Going from 1-wide to 2-wide gives a large speedup; going from 4-wide to 8-wide gives much less:

- **Limited ILP in programs:** most programs simply do not have enough independent instructions in a small window to keep 8+ functional units busy every cycle. Studies show that typical programs have an average ILP of about 2–4 instructions per cycle, even with perfect branch prediction and unlimited resources.
- **Quadratic hardware cost:** dependence checking, renaming, and issue logic grow roughly as $O(n^2)$ with issue width n. An 8-wide machine is not twice the cost of a 4-wide—it is significantly more.
- **Memory wall:** wider issue means more memory operations per cycle, increasing pressure on the cache hierarchy. Cache misses stall the entire wide pipeline.
- **Branch misprediction cost:** a wider machine has more speculative instructions in flight, so each misprediction wastes more work.

This is a major reason modern processors have shifted toward **multicore** designs rather than ever-wider superscalar: instead of trying to extract more parallelism from a single thread (which hits diminishing returns), put multiple independent cores on the chip and run multiple threads in parallel. The parallelism comes from the software (multiple threads or processes) rather than from the hardware trying to find it within a single instruction stream.

49 Overview

What are they? An **exception** is an unexpected event that disrupts normal instruction execution and requires the processor to transfer control to a special handler routine. The terms "exception" and "interrupt" are often used interchangeably, but in RISC-V (and Patterson & Hennessy) they are distinguished:

- **Exception (internal):** caused by something the processor itself does—an illegal instruction, arithmetic overflow, a system call (`ECALL`), or a page fault. These are **synchronous**: they are tied to a specific instruction and will occur every time that instruction executes in the same context.
- **Interrupt (external):** caused by something outside the processor—an I/O device signaling completion, a timer expiring, or a hardware failure. These are **asynchronous**: they arrive at unpredictable times, unrelated to any particular instruction.

Despite the different causes, the hardware response is essentially the same: save enough state to resume later, then jump to a handler.

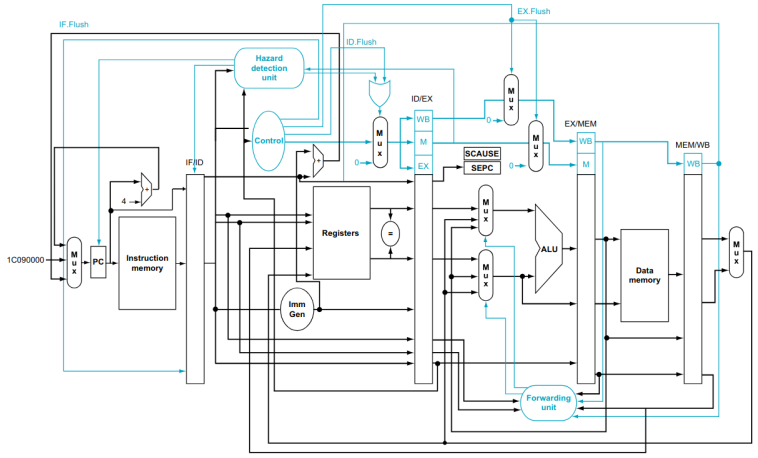


FIGURE 4.67 The datapath with controls to handle exceptions. The key additions include a new input with the value 0000 0000 1C09 0000$_{hex}$ in the multiplexor that supplies the new PC value; an SCAUSE register to record the cause of the exception; and an SEPC register to save the address of the instruction that caused the exception. The 0000 0000 1C09 0000$_{hex}$ input to the multiplexor is the initial address to begin fetching instructions in the event of an exception.

50 What the Hardware Does

When an exception or interrupt is detected, the processor must:

1. **Save the PC** of the offending (or interrupted) instruction into a special register. In RISC-V, this is the `SEPC` (Supervisor Exception Program Counter) or `MEPC` (Machine Exception Program Counter), depending on the privilege level.

2. **Record the cause** in a status register (`SCAUSE` / `MCAUSE` in RISC-V) so the handler knows *why* the exception occurred.

3. **Transfer control** to the exception handler by loading the PC with the address from a handler base register (`STVEC` / `MTVEC`).

4. **Disable further interrupts** (optionally) to prevent nested exceptions from corrupting state before it is saved.

The handler (software, typically part of the OS) then examines the cause, takes appropriate action (e.g., kills the offending process, services the I/O device, handles the page fault), and returns to normal execution using a special return instruction (`SRET` / `MRET`), which restores the saved PC.

51 Exceptions in a Pipelined Processor

Exceptions become tricky in a pipeline because multiple instructions are in flight simultaneously:

- Different exceptions can occur in different stages: an illegal instruction is detected in ID, arithmetic overflow in EX, a page fault in MEM.
- A later instruction (further along the pipeline) might raise an exception *before* an earlier instruction does, even though the earlier instruction comes first in program order.
- Instructions that entered the pipeline *after* the excepting instruction may have already partially executed or modified state.

Precise exceptions. A processor supports **precise exceptions** if, when an exception is handled:

1. All instructions *before* the excepting instruction have completed and their results are committed.

2. The excepting instruction and all instructions *after* it have had no visible effect on the architectural state.

This gives the illusion that instructions executed one at a time in order, which makes it possible for the handler to inspect a clean program state, fix the problem, and resume exactly where execution left off. Precise exceptions are essential for virtual memory (page faults must be restartable) and for debugging.

How the pipeline achieves precise exceptions. In a simple 5-stage pipeline, the hardware handles this by:

- Recording exceptions as they are detected in each stage but **not acting on them immediately**. Instead, the exception status is carried along in the pipeline registers alongside the instruction.
- Waiting until the instruction reaches WB (the *commit point*) to actually raise the exception. At this point, all earlier instructions have already committed, and all later instructions can be flushed cleanly.
- Flushing all instructions behind the excepting instruction by zeroing their pipeline registers.

In out-of-order processors, the **Reorder Buffer (ROB)** provides precise exceptions: instructions execute out of order but commit in order. If an instruction in the ROB has an exception, all later instructions (even those that have already executed) are discarded, and the architectural state reflects only the instructions that committed before the exception.

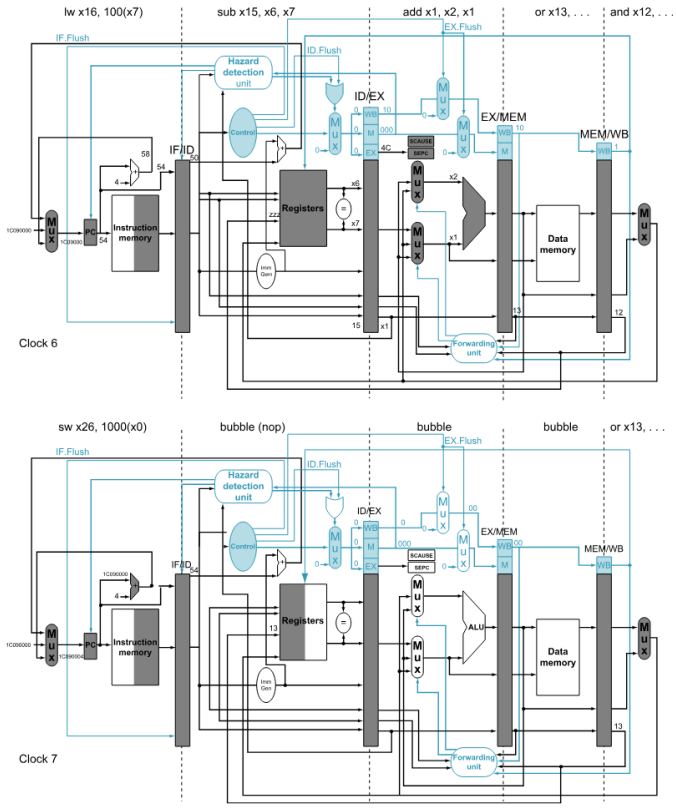


FIGURE 4.68 The result of an exception due to hardware malfunction in the add instruction. The exception is detected during the EX stage of clock 6, saving the address of the add instruction in the SEPC register (4C$_{hex}$). It causes all the Flush signals to be set near the end of this clock cycle, deasserting control values (setting them to 0) for the `add`. Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine—`sw x26, 1000(x0)`—from instruction location 0000 0000 1C09 0000$_{hex}$. Note that the `and` and `or` instructions, which are prior to the `add`, still complete.

52 RISC-V Privilege Levels

RISC-V defines up to three privilege levels that determine which exception-related CSRs (Control and Status Registers) are used:

- **Machine mode (M-mode):** the highest privilege, always present. Handles traps via `MEPC`, `MCAUSE`, `MTVEC`. Typically runs firmware or a bootloader.
- **Supervisor mode (S-mode):** used by operating system kernels. Handles traps via `SEPC`, `SCAUSE`, `STVEC`.
- **User mode (U-mode):** lowest privilege, runs application code. Cannot directly handle exceptions—they are delegated upward to S-mode or M-mode.

Exceptions can be **delegated** from a higher privilege level to a lower one (e.g., M-mode can configure certain exceptions to be handled in S-mode) using delegation registers (`MEDELEG`, `MIDELEG`).

CACHE MEMORY

53 Why Caches Exist

The processor-memory gap. Modern processors can execute instructions in sub-nanosecond cycles, but main memory (DRAM) takes on the order of 50–100 nanoseconds to respond. Without intervention, the processor would spend the vast majority of its time *waiting* for memory. A **cache** is a small, fast SRAM buffer placed between the processor and main memory that stores recently accessed data so that future accesses can be served quickly.

Locality of reference. Caches work because real programs exhibit two forms of predictable access patterns:

- **Temporal locality:** if a memory address was accessed recently, it is likely to be accessed again soon (e.g., loop variables, stack frames, frequently called functions).
- **Spatial locality:** if an address was accessed, nearby addresses are likely to be accessed soon (e.g., sequential array traversal, sequential instruction fetch).

Cache design exploits both: temporal locality by *keeping* recently used data, and spatial locality by fetching data in **blocks** (also called **cache lines**) rather than individual bytes.

54 Cache Terminology

- **Hit:** the requested data is found in the cache. The access is served at cache speed (typically 1–3 cycles for L1).
- **Miss:** the requested data is not in the cache. The processor must fetch the block from a lower level of the memory hierarchy (L2, L3, or main memory), incurring a **miss penalty** of many cycles.
- **Hit rate:** the fraction of accesses that are hits. Even small caches often achieve hit rates above 90%.
- **Miss rate:** $1 -$ hit rate.
- **Hit time:** the time to access the cache and determine hit/miss (includes tag comparison).
- **Miss penalty:** the additional time to fetch data from the next level on a miss.
- **Average Memory Access Time (AMAT):**

$$\boxed{\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}}$$

  This is the single most important equation in cache analysis. For multi-level caches, the miss penalty of level n is the AMAT of level $n+1$.
- **Block (cache line):** the unit of data transfer between cache levels, typically 32–64 bytes. Every cache entry stores one block.
- **Tag:** the upper bits of the memory address stored alongside each cache block to identify *which* block from main memory is currently occupying that cache entry.
- **Valid bit:** a single bit per cache entry indicating whether the entry contains valid data (necessary because the cache starts empty or may contain stale entries after a flush).
- **Dirty bit:** used in write-back caches to indicate that the cached block has been modified and must be written to the next level before eviction.

55 Address Decomposition

For any cache access, the processor splits the memory address into three fields. For a cache with 2^s sets, block size of 2^b bytes, and an m-bit address:

- **Block offset** (low-order b bits): selects the specific byte within the cache block. For a 64-byte block, $b = 6$.
- **Index** (next s bits): selects which **set** (row) of the cache to look in. For a cache with 256 sets, $s = 8$.
- **Tag** (remaining $m - s - b$ upper bits): compared against the stored tag in the selected set to determine if this is a hit. The tag uniquely identifies which main memory block is stored in this entry.

The total cache size (data only) is:

$$\text{Cache size} = \text{Number of sets} \times \text{Associativity} \times \text{Block size}$$

56 Cache Organization (Associativity)

The **associativity** of a cache determines how many locations (ways) a given memory block can be placed in. This is the most fundamental design choice in cache architecture.

Direct-mapped cache (1-way associative). Each memory block maps to *exactly one* cache entry, determined by the index bits. To check for a hit, the hardware reads that single entry and compares the tag.

Advantages: simplest and fastest to access—only one tag comparison, one data read. Minimal hardware cost.

Disadvantage: **conflict misses.** If two frequently accessed blocks happen to map to the same index, they repeatedly evict each other even if the rest of the cache is empty. This is sometimes called **thrashing**.

Example: in a direct-mapped cache with 256 entries, addresses `0x0000` and `0x4000` might map to the same index. A loop alternating between these two addresses would miss on *every* access.

Fully associative cache. A block can be placed in *any* cache entry. There is no index field; the entire address (minus the offset) is the tag, and *every* entry's tag must be compared simultaneously on each access.

Advantages: eliminates conflict misses entirely. The only misses are compulsory (first access) and capacity (cache is full).

Disadvantage: requires a comparator for every cache entry, making it prohibitively expensive and power-hungry for large caches. Only practical for very small structures (e.g., TLBs with 16–64 entries).

N-way set-associative cache. A compromise: the cache is divided into sets, each containing N entries (ways). A block maps to a specific *set* (determined by the index bits) but can occupy *any way* within that set. On an access, all N tags in the selected set are compared in parallel.

- 2-way set associative: each set has 2 entries. Halves the number of sets compared to a same-size direct-mapped cache (one more index bit becomes part of the tag). Dramatically reduces conflict misses.
- 4-way, 8-way: further reduce conflict misses with diminishing returns. Most modern L1 caches are 4–8 way; L2/L3 caches are often 8–16 way.

A direct-mapped cache is 1-way set associative. A fully associative cache is an N-way cache where N equals the total number of blocks (one single set).

57 Replacement Policies

When a miss occurs in a set-associative or fully associative cache and all ways in the target set are occupied, one block must be **evicted** to make room. The replacement policy decides which one:

- **LRU (Least Recently Used):** evict the block that has not been accessed for the longest time. This directly exploits temporal locality. Optimal for small associativities but tracking exact LRU order becomes expensive for high associativity (tracking $N!$ orderings for N ways).
- **Pseudo-LRU:** approximations of LRU that use fewer state bits. A common approach is a binary tree of bits: for a 4-way cache, 3 bits can approximate the LRU victim. Used in many real L1 caches.
- **FIFO (First In, First Out):** evict the block that was loaded earliest, regardless of whether it has been accessed since. Simpler than LRU but can evict frequently used blocks.
- **Random:** choose a victim at random. Surprisingly competitive with LRU for large caches and very cheap to implement (just a counter or LFSR). Often used in L2/L3 caches.

The ideal policy would be **Belady's optimal algorithm** (evict the block that will not be used for the longest time in the future), but this requires knowledge of future accesses and is only useful as a theoretical benchmark.

58 Write Policies

What happens when the processor writes to a cached block?

There are two independent decisions: what to do on a **write hit** and what to do on a **write miss**.

Write-hit policy:

- **Write-through:** every write updates *both* the cache and the next level of the hierarchy immediately. The cache and memory are always consistent. Simple but generates heavy write traffic to the lower level.

  A **write buffer** is almost always used with write-through: writes are queued in a small FIFO buffer so the processor does not stall waiting for the slower memory. As long as writes do not arrive faster than the buffer can drain, the processor runs at full speed.
- **Write-back:** writes update *only* the cache. The modified block is marked with a **dirty bit**. The block is written to the next level only when it is **evicted**. This reduces write traffic significantly (many writes to the same block are "absorbed" by the cache) but complicates the design because the cache and memory can be inconsistent.

Write-miss policy:

- **Write-allocate (fetch on write):** on a write miss, fetch the block from the next level into the cache, then perform the write in the cache. This is the natural pairing with write-back, because subsequent writes to the same block will hit.
- **No-write-allocate (write around):** on a write miss, write directly to the next level without loading the block into the cache. This avoids polluting the cache with data that may not be read. Natural pairing with write-through.

Most modern caches use **write-back with write-allocate** because it minimizes traffic to slower memory levels.

59 Types of Cache Misses (The Three C's)

Every cache miss falls into one of three categories:

- **Compulsory (cold) misses:** the very first access to a block that has never been in the cache. These are unavoidable regardless of cache size or associativity. The only way to reduce them is **prefetching** (loading blocks before they are requested).
- **Capacity misses:** the cache is too small to hold all the blocks the program actively uses (its **working set**). Even a fully associative cache of the same size would miss. The solution is a larger cache.
- **Conflict misses:** the cache has enough total space, but multiple active blocks map to the same set and evict each other. These are the misses that disappear if you switch from N-way to fully associative with the same total size. Increasing associativity reduces conflict misses.

A useful diagnostic technique: simulate the same access trace on a fully associative cache of the same size. Any miss that occurs in the real cache but *not* in the fully associative simulation is a conflict miss. Any miss that occurs in both is either compulsory or capacity.

60 Multi-Level Cache Hierarchies

Modern processors use two or three levels of cache:

- **L1 cache:** closest to the processor, smallest (typically 32–64 KB), and fastest (1–3 cycle hit time). Usually split into separate **L1-I** (instruction) and **L1-D** (data) caches so that instruction fetch and data access can proceed simultaneously without structural hazards.
- **L2 cache:** larger (256 KB – 1 MB), slower (5–15 cycles), unified (holds both instructions and data). Serves as a backstop for L1 misses.
- **L3 cache:** even larger (2–64 MB), slower (20–50 cycles), often shared among multiple cores. The last level before main memory.

Inclusion policies between levels:

- **Inclusive:** every block in L1 is also in L2. Simplifies coherence (a snoop that misses in L2 is guaranteed to miss in L1) but wastes some L2 capacity duplicating L1 contents.
- **Exclusive:** a block resides in exactly one level. Maximizes effective capacity (L1 + L2 hold distinct data). On an L1 eviction, the block moves to L2 ("victim cache" behavior). On an L2 hit that was an L1 miss, the block swaps between levels.
- **Non-inclusive non-exclusive (NINE):** no strict guarantee either way. L2 may or may not contain L1's blocks. Simpler to implement than strict inclusion/exclusion.

AMAT with multiple levels:

$$\text{AMAT} = \text{L1 Hit Time} + \text{L1 Miss Rate} \times \big(\text{L2 Hit Time} + \text{L2 Miss Rate} \times \text{L2 Miss Penalty}\big)$$

The L2 miss rate here is the **local** miss rate (fraction of accesses *to L2* that miss). Sometimes the **global** miss rate is quoted instead (L1 miss rate × L2 local miss rate), which gives the fraction of *all* accesses that miss both levels.

61 Cache Performance Optimization

Techniques to improve cache performance can be grouped by which term in the AMAT equation they target:

Reducing miss rate:

- Increase block size (exploits spatial locality, but too large increases miss penalty and can increase conflict misses).
- Increase associativity (reduces conflict misses, but increases hit time and power).
- Increase cache size (reduces capacity misses, but costs area, power, and potentially hit time).
- Software optimization: loop transformations such as **blocking/tiling** restructure array accesses to fit within the cache, and **loop interchange** can change the access order to match the memory layout (row-major vs. column-major).
- **Prefetching:** hardware or software speculatively fetches blocks before they are demanded. Hardware prefetchers detect stride patterns (e.g., accessing every 64th byte) and issue fetches ahead of time. Compiler-inserted `prefetch` instructions do the same under software control.

Reducing miss penalty:

- Add more cache levels (L2, L3) so that most misses are served by a fast on-chip cache rather than slow DRAM.
- **Critical word first:** when fetching a multi-word block, request the specific word that caused the miss first and deliver it to the processor immediately, then fill the rest of the block. The processor does not wait for the entire block.
- **Early restart:** similar idea—resume execution as soon as the needed word arrives, before the full block transfer completes.
- **Non-blocking (lockup-free) caches:** allow the cache to continue serving hits while a miss is being resolved. The cache tracks outstanding misses in **MSHRs (Miss Status Holding Registers)**. This is essential for out-of-order processors that can have many memory operations in flight.
- Write buffers and victim caches absorb eviction traffic so that misses are not delayed by write-backs.

Reducing hit time:

- Keep L1 caches small and low-associativity (direct-mapped or 2-way) to minimize access latency.
- **Virtually-indexed, physically-tagged (VIPT):** use virtual address bits for the index (available immediately, no TLB lookup needed) but physical address bits for the tag (checked after the TLB lookup completes in parallel). This overlaps the TLB translation with the cache access, reducing effective hit time. Works cleanly when the number of index + offset bits does not exceed the page offset size.
- Pipeline the cache access itself across multiple cycles (common for L2/L3).

62 Cache Coherence (Multiprocessor Context)

When multiple cores each have private caches but share main memory, a new problem arises: one core may modify a cached block while another core holds a stale copy. This is the **cache coherence** problem.

Coherence definition. A memory system is coherent if:

1. a read by core P to address X returns the most recent write to X by P (program order is preserved),

2. a read by core P to address X eventually returns the value written by another core Q (writes become visible),

3. writes to the same address are **serialized**: all cores observe them in the same order.

Snooping protocols. In small-scale multiprocessors, all caches monitor ("snoop") a shared bus. When one core writes, the others see the bus transaction and take action. The most common snooping protocol is **MESI**, where each cache line is in one of four states:

- **Modified (M):** this cache has the only valid copy, and it has been written. Memory is stale. The cache must write back before any other core can read.
- **Exclusive (E):** this cache has the only copy, but it has not been modified. Memory is up to date. Can transition to M on a local write without bus traffic.
- **Shared (S):** multiple caches may hold this block, all consistent with memory. A write requires invalidating all other copies first.
- **Invalid (I):** the entry is not valid (equivalent to not being in the cache).

On a write hit in S state, the writing core issues an **invalidation** on the bus, causing all other caches to transition their copy to I. This is called a **write-invalidate** protocol.

Directory-based protocols. For larger systems where a shared bus is impractical, a **directory** (a centralized or distributed table) tracks which caches hold each block. On a write, the directory sends targeted invalidations only to the caches that hold the block, avoiding broadcast traffic. This scales better but adds latency and storage overhead for the directory.

False sharing. A subtle performance problem: two cores access *different variables* that happen to reside in the *same cache line*. The coherence protocol treats the entire line as a unit, so writes by one core invalidate the other core's copy even though they are modifying unrelated data. The solution is to pad data structures so that frequently written variables by different cores fall on different cache lines.

VIRTUAL MEMORY

63 Motivation

What is it. Virtual memory is an operating system technique that creates an illusion for applications of having a large, contiguous block of main memory (RAM), even if physical RAM is limited. It works by using hard drive or SSD space to extend RAM, swapping inactive data between RAM and storage, allowing computers to run larger or more applications than physical memory allows.

Why virtual memory? Without virtual memory, every program would need to manage physical memory addresses directly. This creates serious problems: programs could accidentally (or maliciously) overwrite each other's data, every program would need to know exactly how much physical memory is available, and running multiple programs simultaneously would require manually partitioning memory among them.

Virtual memory solves all of these by giving each program the illusion that it has its own large, private, contiguous address space. The hardware and operating system collaborate to **translate** virtual addresses (what the program sees) into physical addresses (where data actually lives in DRAM). This translation happens transparently—the program never knows or cares where its data physically resides.

Three key benefits.

1. **Protection / isolation:** each process has its own virtual address space. Process A cannot access process B's physical memory because the translation mechanism simply will not map A's virtual addresses to B's physical frames. The OS controls the mapping, so it can enforce access permissions (read, write, execute) on each region.

2. **Abstraction of physical memory:** programs are written as if they have a large contiguous address space (e.g., 32-bit = 4 GB, 64-bit = vastly more). The actual physical memory may be smaller, fragmented, or shared—the translation layer hides all of this.

3. **Memory as a cache for disk:** physical memory acts as a cache for a much larger virtual address space. Pages that do not fit in DRAM can be stored on disk (in a **swap** region). When accessed, they are brought into physical memory on demand. This allows programs to use more memory than physically exists.

64 Paging

Pages and frames. Virtual memory divides both the virtual address space and physical memory into fixed-size chunks:

- A **page** is a fixed-size block of virtual memory (typically 4 KB in RISC-V and most modern systems, though larger pages of 2 MB or 1 GB exist).
- A **frame** (or **physical page**) is a same-sized block of physical memory.

Translation maps each virtual page to a physical frame. The key advantage of fixed-size pages over variable-sized segments is that any page can be placed in any frame, eliminating **external fragmentation** (unusable gaps between allocated regions).

Virtual address decomposition. A virtual address is split into two parts:

- **Virtual Page Number (VPN):** the upper bits, identifying which page the address belongs to.
- **Page offset:** the lower bits, identifying the specific byte within the page. For a 4 KB page, the offset is 12 bits ($2^{12} = 4096$).

Translation replaces the VPN with a **Physical Page Number (PPN)** while keeping the page offset unchanged:

$$\text{Physical Address} = \text{PPN} \,\|\, \text{Page Offset}$$

65 Page Tables

What is a page table? The **page table** is a data structure maintained by the operating system that stores the mapping from virtual page numbers to physical page numbers. Each process has its own page table (this is how isolation is enforced—different processes have different mappings).

Each entry in the page table is called a **Page Table Entry (PTE)** and contains:

- **PPN:** the physical page number this virtual page maps to.
- **Valid bit:** whether this page is currently in physical memory. If 0, the page is either on disk or has never been allocated.
- **Protection bits:** read (R), write (W), execute (X) permissions. The hardware checks these on every access and raises an exception if a permission is violated.
- **Dirty bit:** whether the page has been written to since it was loaded. If a dirty page must be evicted, it must be written back to disk first.
- **Reference (accessed) bit:** set by hardware whenever the page is accessed. The OS uses this to implement replacement policies (e.g., approximate LRU).

The problem with flat page tables. A single-level page table for a 32-bit address space with 4 KB pages has $2^{32}/2^{12} = 2^{20} \approx 1$ million entries. If each PTE is 4 bytes, that is 4 MB per process—large but manageable. However, for a 64-bit address space, a flat page table would require 2^{52} entries (with 4 KB pages), which is impossibly large.

Multi-level page tables. The solution is to use a **hierarchical (multi-level) page table**. Instead of one giant table, the VPN is split into multiple fields, each indexing a different level:

- The first-level table (the **page directory**) is always in memory. Each entry points to a second-level table.
- Second-level tables are only allocated if that region of the virtual address space is actually used by the program.
- The final level contains the actual PPN.

This is how RISC-V organizes its page tables. The Sv32 scheme (for RV32) uses a 2-level page table. The Sv39 scheme (for RV64) uses a 3-level page table with a 39-bit virtual address:

- VPN[2] (9 bits) indexes the first-level table.
- VPN[1] (9 bits) indexes the second-level table.
- VPN[0] (9 bits) indexes the third-level table, yielding the PPN.
- Page offset (12 bits) passes through unchanged.

Each level's table is exactly one page (4 KB) containing $2^9 = 512$ entries of 8 bytes each. The key advantage: a program that uses only a small fraction of its address space only needs a handful of page table pages in memory, rather than a single enormous table.

The **page table base register** (`SATP` in RISC-V) stores the physical address of the root (first-level) page table for the currently running process. On a context switch, the OS updates `SATP` to point to the new process's page table.

The cost of translation. With a multi-level page table, translating a single virtual address requires **multiple memory accesses**—one per level of the page table, plus the final access to the actual data. For Sv39, that is 3 page table accesses + 1 data access = 4 memory accesses for every load or store. This would be catastrophically slow without caching the translations.

66 Translation Lookaside Buffer (TLB)

What is a TLB? The **Translation Lookaside Buffer (TLB)** is a small, fast, **fully associative** (or high-associativity set-associative) cache that stores recent virtual-to-physical page translations. Instead of walking the multi-level page table on every access, the hardware first checks the TLB.

- **TLB hit:** the VPN is found in the TLB, which immediately provides the PPN. The translation adds essentially zero extra latency (the TLB lookup happens in parallel with the cache access in a VIPT design). Typical TLB hit rates exceed 99%.
- **TLB miss:** the VPN is not in the TLB. The hardware (or software, depending on the architecture) must **walk the page table** to find the mapping. This is called a **page table walk** and costs several memory accesses. The resulting PTE is then loaded into the TLB for future use.

TLB structure. A typical TLB has 16–64 entries (sometimes more in modern processors). Because it is small, full associativity is practical, which maximizes the hit rate. Each TLB entry contains:

- The VPN (used as the tag for lookup).
- The PPN (the translated result).
- Protection bits (R, W, X) copied from the PTE.
- Valid bit.
- An **ASID (Address Space Identifier)** in some designs: a small tag identifying which process the entry belongs to, so the TLB does not need to be flushed entirely on every context switch.

TLB misses: hardware vs. software.

- **Hardware-managed TLB:** the processor itself walks the page table on a TLB miss. This is what RISC-V and x86 use. The walk is handled by dedicated hardware (a **page table walker**) and is transparent to software.
- **Software-managed TLB:** a TLB miss raises an exception, and the OS handler walks the page table and loads the TLB entry. This was used in MIPS and some other architectures. It gives the OS more flexibility but adds overhead.

67 Page Faults

What is a page fault? If the page table walk finds that the valid bit of the PTE is 0, the requested page is **not in physical memory**. The hardware raises a **page fault exception**, and control transfers to the OS page fault handler.

The handler determines what to do:

1. If the page exists on disk (it was **swapped out**), the OS reads it from disk into a free physical frame, updates the page table to point to the new frame, and restarts the faulting instruction.

2. If no free frame is available, the OS must **evict** an existing page. If the evicted page is dirty, it must be written back to disk first. The OS typically uses an approximate LRU policy (based on reference bits) to choose the victim.

3. If the access is to an address that was never allocated (e.g., dereferencing a null pointer), the OS delivers a **segmentation fault** to the process, typically killing it.

Why page faults are expensive. Disk access takes on the order of **milliseconds**—roughly 10^6 times slower than a DRAM access. This enormous miss penalty is why virtual memory systems use large pages (4 KB instead of 64-byte cache lines), write-back (never write-through), and highly aggressive replacement policies. Even a small page fault rate has a massive impact on performance.

Page faults and precise exceptions. A page fault must be **restartable**: after the OS brings the page into memory, the processor re-executes the exact instruction that caused the fault, and it must produce the correct result. This is why precise exceptions (discussed in the Exceptions section) are essential—the architectural state must be cleanly rolled back to the point of the faulting instruction.

68 Interaction with the Cache

Where does translation happen relative to the cache? The cache can be indexed and tagged using either virtual or physical addresses, leading to different design points:

- **Physically Indexed, Physically Tagged (PIPT):** the cache is accessed using the physical address. The TLB lookup must complete *before* the cache can be accessed, adding the TLB latency to every cache access. Simple and avoids aliasing issues but slower.
- **Virtually Indexed, Virtually Tagged (VIVT):** the cache is accessed entirely with virtual addresses—no translation needed for cache hits. Very fast, but creates **aliasing** problems: different virtual addresses in different processes can map to the same physical address, leading to multiple stale copies. Requires flushing the cache on context switches. Rarely used today.
- **Virtually Indexed, Physically Tagged (VIPT):** the **index** comes from the virtual address (available immediately), while the **tag** is the physical address (provided by the TLB). The cache lookup and TLB lookup happen *in parallel*: the index selects the set while the TLB produces the physical tag for comparison. This hides the TLB latency and is the most common design for L1 caches.

  The VIPT design works cleanly when the number of index bits + offset bits does not exceed the page offset. For a 4 KB page with a 12-bit offset, and a cache with 64-byte blocks and 64 sets, the index needs 6 bits and the offset needs 6 bits, totaling 12 bits—exactly the page offset. All index bits come from the page offset, which is the same in both virtual and physical addresses, so no aliasing occurs. If the cache is larger and requires more index bits, aliasing must be handled explicitly.

L2 and L3 caches, which are less latency-sensitive, typically use PIPT because translation has already been resolved by the time an L1 miss reaches them.

69 TLB and Context Switches

When the OS switches from one process to another, the virtual-to-physical mappings change (the new process has a different page table). The TLB entries from the old process are now invalid for the new process. Two approaches:

- **Flush the TLB:** invalidate all TLB entries on every context switch. Simple but causes a burst of TLB misses (and thus page table walks) when the new process starts running.
- **Tagged TLB with ASIDs:** each TLB entry includes an **Address Space Identifier** that identifies which process it belongs to. On a context switch, only the ASID register is updated. Entries from different processes coexist in the TLB, and a lookup only matches if both the VPN *and* the ASID match. This avoids flushing and preserves useful entries from processes that will run again soon. RISC-V supports ASIDs in the `SATP` register.

70 Virtual Memory as a Level of the Memory Hierarchy

Virtual memory fits naturally into the memory hierarchy, with physical memory (DRAM) acting as a cache for the much larger virtual address space (backed by disk):

- **Block size:** one page (4 KB or larger)—much bigger than a cache line, to amortize the enormous disk latency.
- **Associativity:** fully associative—any virtual page can map to any physical frame. The OS has full control over placement.
- **Replacement:** handled by the OS in software (approximate LRU using reference bits), not hardware.
- **Write policy:** always write-back (writing through to disk on every store would be absurdly slow).
- **Miss penalty:** millions of cycles (disk access), making the miss rate the dominant performance factor.

MULTICORE PROCESSORS

71 Why Multicore?

The end of frequency scaling. For decades, processor performance improved by increasing clock frequency—from MHz in the 1980s to GHz by the early 2000s. But around 2004, this approach hit a wall. Higher frequencies require higher voltages, and power consumption grows roughly as:

$$P \propto C \times V^2 \times f$$

where C is capacitance, V is voltage, and f is frequency. Pushing frequency further caused chips to generate more heat than could be practically dissipated. This is often called the **power wall**.

At the same time, the superscalar approach was hitting diminishing returns: making a single core wider than 4–6 issue gave minimal speedup because programs lack sufficient instruction-level parallelism, and the hardware cost grows quadratically with issue width.

Dennard Scaling (and its end). For decades, a principle called **Dennard scaling** (after Robert Dennard, 1974) allowed clock frequencies to rise without power density increasing. The observation was: as transistors shrink, their voltage and current scale down proportionally, so each transistor uses less power. A transistor half the size can switch at twice the speed, but at lower voltage it consumes roughly the same power *per unit area*. This meant chip designers could pack more transistors, run them faster, and still cool the chip with the same heatsink.

Dennard scaling is why clock rates grew from 25 MHz to 3+ GHz over two decades while power only increased by a factor of roughly 30—even though clock rates increased by a factor of 1000. Each generation reduced voltage by about 15%, and since power is proportional to V^2, the voltage reduction largely offset the increase from higher frequency and more transistors.

Why it broke down (∼2004). As transistors shrank below about 65 nm, voltage could no longer be reduced without making transistors too "leaky." A transistor at very low voltage cannot fully turn off—current leaks through even in the off state, like a faucet that cannot be completely shut. This **static (leakage) power** now accounts for roughly 40% of power consumption in modern server chips, and it grows with transistor count regardless of whether those transistors are actively switching.

With voltage stuck near ∼1 V, any increase in frequency or transistor count directly increases power. Chips hit roughly 100–150 watts—the practical limit for air cooling in commodity systems. This is the **power wall**: not a theoretical limit, but an economic and engineering one. More exotic cooling exists but is too expensive for consumer and most server hardware.

The breakdown of Dennard scaling is the single most important reason the industry shifted from making one core faster to putting multiple cores on a chip. Transistor counts still grow (Moore's Law continued longer than Dennard scaling), so designers spend those transistors on *more cores* rather than on wider or faster single cores.

The multicore solution. Instead of making one core faster, put **multiple independent cores** on the same chip, each running its own thread or process. A quad-core processor running four threads can achieve four times the throughput of a single core—not by making any one thread faster, but by running more threads simultaneously. This is called **thread-level parallelism (TLP)**, as opposed to the instruction-level parallelism (ILP) exploited by superscalar.

The power advantage is significant. Two cores running at frequency f with voltage V consume roughly the same power as one core at frequency $2f$ (which would need higher voltage), but deliver more total throughput. Multicore trades single-thread performance for better performance per watt.

72 Multicore Architecture

Basic structure. A typical multicore processor consists of:

- **Multiple cores:** each is an independent processor with its own pipeline, register file, and control logic. Each core can be a full out-of-order superscalar design.
- **Private L1 caches:** each core has its own L1-I and L1-D cache for fast access without contention.
- **Private or shared L2 cache:** some designs give each core its own L2; others share it.
- **Shared last-level cache (LLC):** typically L3, shared among all cores. Reduces off-chip memory traffic and allows data sharing between cores.
- **Shared main memory:** all cores see the same physical address space and can access the same DRAM.
- **Interconnect:** a bus, ring, or mesh network connecting cores to each other and to the shared cache/memory.

Homogeneous vs. heterogeneous.

- **Homogeneous multicore:** all cores are identical. Simpler to design and program. Example: early Intel Core processors.
- **Heterogeneous multicore (big.LITTLE / hybrid):** the chip contains a mix of high-performance cores and power-efficient cores. Demanding tasks run on the big cores; background tasks run on the little cores to save energy. Examples: ARM big.LITTLE, Apple M-series (performance + efficiency cores), Intel Alder Lake and later (P-cores + E-cores).

73 Parallel Programming

The fundamental challenge. Multicore only helps if the software can divide its work across multiple cores. A single-threaded program runs on one core and gains nothing from having additional cores. This makes **parallel programming** essential to exploiting multicore.

Amdahl's Law. Not all parts of a program can be parallelized. Amdahl's Law quantifies this limitation:

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{n}}$$

where f is the fraction of the program that can be parallelized and n is the number of cores. Even with infinite cores, the maximum speedup is $1/(1-f)$. If only 90% of a program is parallelizable ($f = 0.9$), the maximum speedup is 10×, no matter how many cores you add. The serial portion becomes the bottleneck.

Example: with $f = 0.75$ and $n = 4$ cores:

$$\text{Speedup} = \frac{1}{0.25 + \frac{0.75}{4}} = \frac{1}{0.25 + 0.1875} = \frac{1}{0.4375} \approx 2.29\times$$

Four cores give only a 2.29× speedup, not 4×, because 25% of the work is serial.

Sources of parallelism.

- **Data parallelism:** the same operation is applied to many data elements independently. Example: adding two large arrays element by element—each core handles a portion of the array.
- **Task parallelism:** different tasks or functions execute concurrently. Example: one core handles the user interface while another processes data in the background.
- **Pipeline parallelism:** work is divided into stages, with each core handling one stage and passing results to the next. Example: a video processing pipeline where one core decodes, another applies filters, and a third encodes.

74 Shared Memory and Synchronization

Shared memory model. In a shared-memory multicore (which is what most multicore processors are), all cores can read and write the same memory. This makes communication easy—one core writes a value, another reads it—but introduces the problem of **coordinating access** to shared data.

Race conditions. A **race condition** occurs when two cores access the same memory location, at least one is a write, and the accesses are not ordered. The result depends on which core gets there first, making the program's behavior unpredictable.

Example: two cores both executing `x = x + 1` on a shared variable. Each core reads `x`, adds 1, and writes back. If both read before either writes, the final value of `x` increases by 1 instead of 2. This is called a **lost update**.

Synchronization primitives. To prevent race conditions, cores must synchronize using hardware-supported primitives:

- **Locks (mutexes):** a lock protects a critical section of code so that only one core can execute it at a time. Before entering the critical section, a core *acquires* the lock; when done, it *releases* it. Other cores trying to acquire the lock must wait (**spin** or **block**).
- **Atomic operations:** special instructions that perform a read-modify-write sequence as a single indivisible operation. RISC-V provides atomic instructions through the A extension, including:
  - `LR` (Load Reserved) and `SC` (Store Conditional): `LR` loads a value and sets a reservation on the address. `SC` writes to the address only if the reservation is still valid (no other core has written to it since the `LR`). If `SC` fails, the software retries the entire sequence. This `LR`/`SC` pair is the foundation for building locks and other synchronization without requiring complex atomic instructions.
  - `AMO` (Atomic Memory Operations): instructions like `AMOADD` and `AMOSWAP` that atomically read, modify, and write a memory location in a single instruction.
- **Barriers:** a synchronization point where all cores must arrive before any can proceed. Used to ensure that all cores have finished one phase of computation before starting the next.

The cost of synchronization. Synchronization adds overhead: acquiring and releasing locks takes time, contended locks cause cores to wait, and atomic operations require the cache coherence protocol to gain exclusive access to the cache line (invalidating other cores' copies). Excessive synchronization can serialize execution and negate the benefits of multicore. Good parallel programs minimize the amount of shared mutable state and the frequency of synchronization.

75 Cache Coherence Revisited

Multicore processors rely on the cache coherence mechanisms discussed in the cache section (MESI protocol, snooping or directory-based). A few points worth emphasizing in the multicore context:

- Coherence is what makes shared-memory programming *possible*. Without it, one core's write might never become visible to another core, making communication impossible.
- Coherence traffic is a significant source of overhead. Every write to shared data generates invalidations or updates on the interconnect. As core counts grow, this traffic can saturate the interconnect.
- **False sharing** (discussed in the cache section) becomes a more pressing problem with more cores, because more cores means more potential for unrelated variables on the same cache line being written by different cores.

76 Memory Consistency

Coherence vs. consistency. Cache coherence guarantees that all cores eventually see the same value for a given address. **Memory consistency** defines the order in which writes to *different* addresses become visible to other cores. These are different guarantees:

- Coherence: "writes to address X are seen by all cores in the same order."
- Consistency: "if core 1 writes X then writes Y, will core 2 see the write to X before the write to Y?"

Sequential consistency. The simplest and most intuitive model: the result of execution is as if all memory operations from all cores were interleaved in some total order that respects each core's program order. In other words, it looks like the cores are taking turns, one operation at a time. This is easy to reason about but expensive to implement because it restricts reordering optimizations.

Relaxed consistency models. Most modern processors (including RISC-V) use **relaxed** memory models that allow certain reorderings for performance. For example, a core might let a later load bypass an earlier store to a different address, because waiting would waste cycles. When strict ordering is needed (e.g., in synchronization code), the programmer inserts explicit **fence** instructions. RISC-V provides the `FENCE` instruction, which ensures that all memory operations before the fence are visible to other cores before any memory operations after the fence.

HARDWARE MULTITHREADING

77 Motivation: Functional Unit Underutilization

The problem. Even with out-of-order execution, branch prediction, and caches, a single thread often cannot keep all functional units busy every cycle. Data dependences, cache misses, and branch mispredictions create bubbles in the pipeline where functional units sit idle. In a superscalar processor with 4 issue slots, a single thread might only fill 1–2 slots on average—the rest are wasted.

The idea. Instead of trying to squeeze more parallelism from one thread, give the processor **multiple hardware threads**. When one thread stalls or cannot fill all issue slots, the processor can issue instructions from *another* thread, keeping the functional units busy. This is **hardware multithreading**—it exploits **thread-level parallelism (TLP)** to improve utilization of existing hardware, without replicating the entire core.

What each thread needs. All hardware threads share the same functional units, caches, and pipeline. But each thread requires its own:

- **Program Counter (PC):** each thread executes a different instruction stream.
- **Register File:** each thread has its own architectural registers so they do not interfere.
- **Execution state (CSRs):** each thread has its own control/status registers and exception state.

A **thread scheduler** decides which thread(s) get to use the pipeline each cycle. The additional hardware cost is modest—duplicating the PC and register file is cheap compared to duplicating the entire core.

78 Types of Hardware Multithreading

Fine-Grain Multithreading (FGMT). The processor switches to a **different thread every cycle** in a round-robin fashion. Only one thread issues instructions in any given cycle, but the thread changes each cycle. If thread T0 fetches in cycle 1, thread T1 fetches in cycle 2, T0 again in cycle 3, and so on.

- **How it hides latency:** if T0 encounters a cache miss or pipeline stall, the processor simply continues issuing instructions from T1, T2, etc. on subsequent cycles. By the time T0's turn comes around again, the stall may have resolved. With enough threads, the pipeline never sits idle.
- **Advantage:** simple hardware. No need to detect or resolve inter-thread dependences within a cycle, because only one thread occupies the pipeline at a time. No slot sharing.
- **Disadvantage:** single-thread performance suffers. Even if T0 has plenty of independent instructions ready, it only gets to issue every Nth cycle (where N is the number of threads). The pipeline bandwidth is divided equally among threads regardless of demand.

Simultaneous Multithreading (SMT). The processor issues instructions from **multiple threads in the same cycle**, sharing the superscalar issue slots dynamically. In a 4-wide superscalar with SMT, cycle 1 might issue 2 instructions from T0 and 2 from T1; cycle 2 might issue 3 from T0 and 1 from T1, depending on what is ready.

- **How it hides latency:** when one thread stalls (e.g., waiting on a cache miss), its issue slots are immediately filled by instructions from other threads. There is no wasted cycle waiting for a round-robin turn.
- **Advantage:** maximizes utilization of superscalar issue slots. Each cycle, the scheduler picks the best instructions from all threads, filling as many slots as possible.
- **Disadvantage:** significantly more complex hardware. The rename logic, issue queue, and scheduling logic must handle instructions from multiple threads simultaneously, tracking which physical registers belong to which thread, and ensuring correct execution across threads sharing the same functional units.
- **Real-world example:** Intel's **Hyper-Threading Technology** is SMT with 2 threads per core. The OS sees each physical core as 2 logical processors.

79 SMT vs. FGMT: Pipeline Comparison

Consider two threads (T0 and T1) each running two instructions: a load (`LW`) followed by an `ADD` that depends on the load result. The load takes 3 cycles to execute, and the `ADD` must wait until the load commits (no forwarding). The pipeline has 4 stages: Fetch (F), Issue (I), Execute (E), Commit (C).

FGMT (round-robin, 1 instruction/cycle):

```
Cycle:   1   2   3   4   5   6   7   8   9
T0_LW:   F   I   E   E   E   C
T1_LW:       F   I   E   E   E   C
T0_ADD:          F   I   -   -   -   E   C
T1_ADD:              F   -   I   -   -   E  ...
```

T0 and T1 alternate fetch cycles. The ADDs stall at Issue waiting for their respective loads to commit. Total: 4 instructions in 9 cycles, IPC $= 4/9 \approx 0.44$.

SMT (2 instructions/cycle):

```
Cycle:   1   2   3   4   5   6   7   8
T0_LW:   F   I   E   E   E   C
T1_LW:   F   I   E   E   E   C
T0_ADD:      F   I   -   -   -   E   C
T1_ADD:      F   I   -   -   -   E   C
```

Both threads fetch and issue in the same cycle. Both ADDs stall until their loads commit, then execute together. Total: 4 instructions in 8 cycles, IPC $= 4/8 = 0.5$.

SMT achieves higher throughput because it can issue from both threads simultaneously rather than alternating.

80 Multithreading vs. Multicore

Hardware multithreading and multicore are complementary strategies that address different bottlenecks:

- **Hardware multithreading (SMT/FGMT):** multiple threads share *one* set of functional units and caches. Improves *utilization* of existing resources. Low additional hardware cost (just duplicate PC, register file, and thread state). But threads compete for the same resources—cache thrashing, functional unit contention, and scheduling complexity limit scalability. Typically 2–4 threads per core.
- **Multicore:** replicates the *entire core*, giving each thread its own pipeline, functional units, and private caches. Threads do not compete for execution resources (only for shared LLC and memory bandwidth). Scales better but costs much more area and power per additional thread.

Modern high-performance processors combine both: multiple cores, each with SMT. For example, a 4-core processor with 2-way SMT presents 8 logical processors to the OS. This exploits both thread-level parallelism (across cores) and functional unit utilization (within each core).

The progression of techniques for increasing throughput can be summarized as:

$$\text{Scalar} \rightarrow \text{Superscalar} \rightarrow \text{SMT/FGMT} \rightarrow \text{Multicore} \rightarrow \text{Multicore + SMT}$$

Each step addresses a different bottleneck: pipelining hides instruction latency, superscalar exploits ILP, multithreading fills idle slots with other threads, and multicore provides dedicated resources for independent threads.