



Multicycle and Pipelined Processors

From one long cycle to many short ones: multicycle datapaths, microprogramming, pipelining, hazards, branch handling and branch prediction

What this handbook covers

- | | |
|------------------------------------|--|
| 1 The multicycle microarchitecture | 5 Control dependences |
| 2 Microprogrammed control | 6 Branch prediction |
| 3 Pipelining fundamentals | 7 Exceptions, multicycle operations & beyond |
| 4 Data dependences and hazards | |

Dr. Bikram Paul

School of Computing and Electrical Engineering (SCEE)

Indian Institute of Technology Mandi

bikram@iitmandi.ac.in

Jan–June 2026

Contents

1	The Multicycle Microarchitecture	1
1.1	The instruction processing cycle	1
1.1.1	Architectural state and micro-states	1
1.2	Benefits of the multicycle design	2
1.3	A multicycle RISC-V datapath	2
1.4	The steps of each instruction	3
1.5	FSM control	3
1.6	Performance analysis	4
	Exercises	6
2	Microprogrammed Control	7
2.1	Microprogram control: the vocabulary	7
2.2	Example: the LC-3b state machine	8
2.3	Microcoding the single-bus processor	9
2.4	Advantages of microprogrammed control	9
2.5	Horizontal and vertical microcode	10
2.6	Nanocode and millicode	11
2.7	A microprogram for the multicycle RISC-V machine	11
2.8	Hardwired or microprogrammed?	12
	Exercises	12
3	Pipelining Fundamentals	14
3.1	The idea	14
3.1.1	The laundry analogy	14
3.2	The ideal pipeline	14
3.2.1	Throughput	15
3.2.2	Throughput and cost with latch overhead	15
3.3	Pipelining the RISC-V datapath	16
3.3.1	Dividing the single-cycle datapath into stages	16
3.3.2	Pipeline registers	17
3.3.3	Illustration of pipeline operation	18

3.4	Control in a pipeline	18
3.5	From ideal to practical pipelines	18
3.5.1	Issues in pipeline design	19
	Exercises	19
4	Data Dependences and Pipeline Hazards	21
4.1	Causes of pipeline stalls	21
4.2	Resource contention	21
4.3	Data dependences	22
4.3.1	Handling flow dependences	22
4.4	Interlocking	22
4.4.1	Detection technique 1: scoreboarding	23
4.4.2	Detection technique 2: combinational dependence-check logic	23
4.5	Analysing register dependences in the five-stage pipeline	23
4.6	Stalling	24
4.6.1	Implementing stalls	24
4.6.2	Stall condition logic	24
4.6.3	Impact of stalls on performance	25
4.7	Data forwarding (bypassing)	25
4.7.1	Resolving RAW with forwarding	26
4.7.2	Forwarding logic	26
4.7.3	The load-use hazard	26
4.8	Who handles dependences: hardware or software?	28
	Exercises	28
5	Control Dependences	30
5.1	Control dependence as a special data dependence	30
5.2	Stalling on branches	30
5.3	Guessing NextPC = PC + 4	31
5.4	Predicate combining and predicated execution	32
5.4.1	Predicate combining (not predicated execution)	32
5.4.2	Predicated execution	33
5.5	Delayed branching	33
5.5.1	Filling the delay slot	34
5.6	Fine-grained multithreading	35
	Exercises	36

6	Branch Prediction	37
6.1	Guessing the next instruction to fetch	37
6.2	What must be predicted?	37
6.3	Static branch prediction	38
6.4	Dynamic branch prediction	40
6.4.1	The last-time predictor	40
6.4.2	Improving the last-time predictor: the two-bit counter	40
6.5	The branch problem in wide, deep pipelines	42
6.6	Correlating (two-level) predictors	42
	Exercises	43
7	Exceptions, Multicycle Operations and Beyond	45
7.1	Long-latency (multicycle) operations	45
7.2	Exceptions and interrupts in a pipeline	46
7.3	Putting it together: a CPI stack	47
7.4	Factorial on three microarchitectures	48
7.5	RTL of the hazard and forwarding units	48
7.6	Beyond the scalar in-order pipeline	49
7.6.1	Superscalar execution	49
7.6.2	Out-of-order execution	49
	Exercises	50
A	Solved Problems	51
B	Formula Sheet	55

The Multicycle Microarchitecture

The single-cycle processor of Unit 2 forces every instruction to take as long as the slowest one. The multicycle microarchitecture breaks each instruction into several short steps, one per clock cycle, so that simple instructions finish in fewer cycles than complex ones and hardware units can be reused within an instruction. This chapter revisits the instruction-processing cycle, builds a multicycle RISC-V datapath and its finite-state-machine control, and analyses when the multicycle design actually wins.

1.1 The instruction processing cycle



Figure 1.1: The instruction processing cycle (von Neumann model): six conceptual phases, repeated for every instruction.

The phases are *conceptual*: an ISA does not require them to be separate in time. Not every instruction needs every phase (an `add` has no address to evaluate; a `jal` stores its result into the PC), and a microarchitecture may merge phases into one clock cycle or spread one phase over several cycles.

1.1.1 Architectural state and micro-states

- The ISA specifies abstractly what the next architectural state AS' should be, given an instruction and the current AS . It defines an abstract finite-state machine in which *state* = programmer-visible state and *next-state logic* = the instruction execution specification. From the ISA's point of view there are no intermediate states between AS and AS' : one state transition per instruction.
- The microarchitecture implements *how* AS is transformed into AS' . There are many choices. We may introduce programmer-*invisible* state to optimize the speed of execution: multiple state transitions per instruction.
 - Choice 1: $AS \rightarrow AS'$ (transform AS to AS' in a single clock cycle) — the single-cycle machine.
 - Choice 2: $AS \rightarrow AS+MS_1 \rightarrow AS+MS_2 \rightarrow AS+MS_3 \rightarrow AS'$ — the multicycle machine,

where MS_i are micro-states.

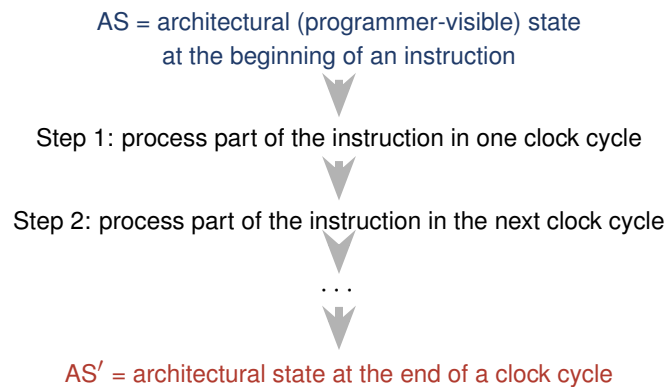


Figure 1.2: Multicycle execution: one instruction, several clock cycles, invisible micro-states in between.

1.2 Benefits of the multicycle design

- **Critical-path design.** The clock is set by the slowest *step*, not the slowest *instruction*. We can keep reducing the critical path independently of the worst-case processing time of any instruction.
- **Bread-and-butter (common-case) design.** We can optimize the number of states (cycles) it takes to execute “important” instructions that make up much of the execution time, while rare instructions may take many cycles.
- **Balanced design.** There is no need to provide more capability or resources than really needed. An instruction that needs resource X multiple times does not require multiple X's to be implemented: a single ALU can compute $PC+4$, the branch target and the operation in different cycles. This leads to more efficient hardware: components are reused multiple times for one instruction.

1.3 A multicycle RISC-V datapath

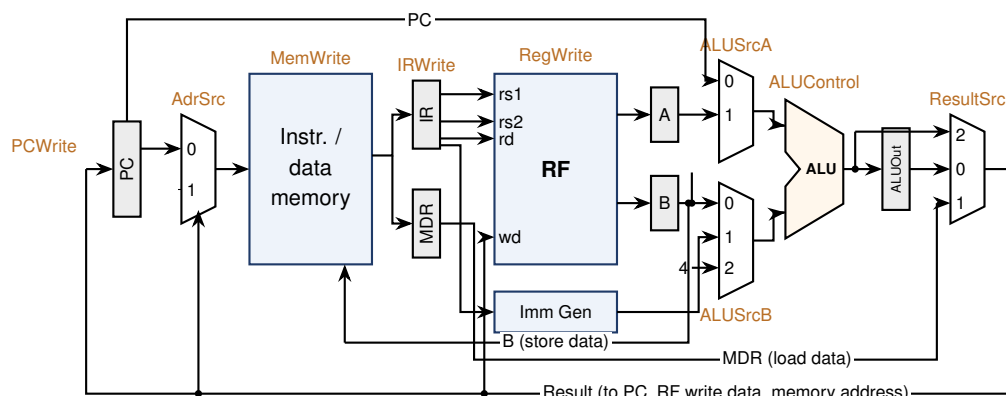


Figure 1.3: A multicycle RISC-V datapath (after Harris & Harris): one memory for instructions and data, one ALU, and non-architectural registers IR, MDR, A, B and ALUOut that hold values between cycles. Orange labels are control signals generated by the FSM.

Compared with the single-cycle datapath:

- **One memory** (instructions and data). The *AdrSrc* multiplexer selects the PC (fetch) or *ALUOut* (data access). This is possible because fetch and data access now happen in different cycles.
- **One ALU** computes $PC+4$, the branch/jump target and the operation, in different cycles. The *ALUSrcA*/*ALUSrcB* multiplexers grow accordingly. The two extra adders of the single-cycle design disappear.
- **Non-architectural registers** (*IR*, *MDR*, *A*, *B*, *ALUOut*) hold the results of one step for use in the next. They are the micro-state MS_i .
- **Write enables** on the PC (*PCWrite*) and *IR* (*IRWrite*), because these are now updated only in specific cycles.

1.4 The steps of each instruction

Table 1.1: Register-transfer steps of the multicycle RISC-V machine.

Step	Action	Used by
1 Fetch	$IR \leftarrow M[PC]$; $OldPC \leftarrow PC$; $PC \leftarrow PC+4$	all
2 Decode	$A \leftarrow R[rs1]$; $B \leftarrow R[rs2]$; $ALUOut \leftarrow OldPC+imm$ (target, computed speculatively)	all
3 Execute	<i>R</i> : $ALUOut \leftarrow A \text{ op } B$; <i>I</i> : $ALUOut \leftarrow A \text{ op } imm$; load/store: $ALUOut \leftarrow A+imm$; branch: if ($A \text{ cmp } B$) $PC \leftarrow ALUOut$; JAL: $PC \leftarrow ALUOut, R[rd] \leftarrow PC$	per class
4 Memory / WB	load: $MDR \leftarrow M[ALUOut]$; store: $M[ALUOut] \leftarrow B$; <i>R/I</i> : $R[rd] \leftarrow ALUOut$	load, store, <i>R</i> , <i>I</i>
5 Load WB	$R[rd] \leftarrow MDR$	load

Resulting cycle counts: **load 5, store 4, R-type and I-ALU 4, branch 3, JAL/JALR 3** (and **LUI/AUIPC 4**).

1.5 FSM control

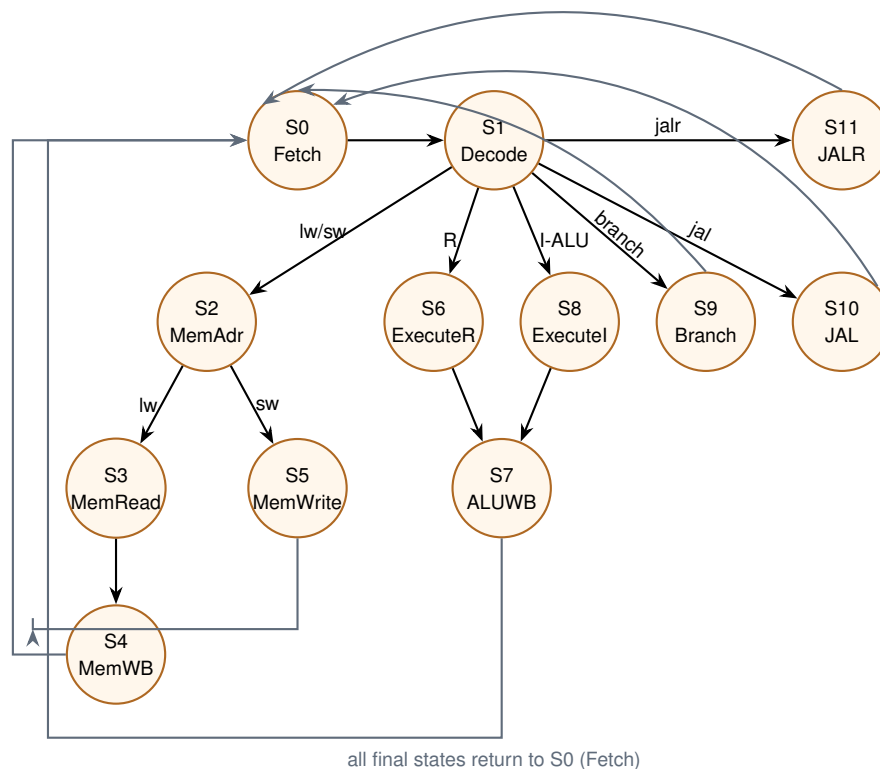


Figure 1.4: Multicycle control FSM. Each state asserts a specific set of control signals.

Worked Example 1.1: Control signals in each state

State	AdrSrc	ALUSrcA, ALUSrcB	ALUOp	ResultSrc	Enables
S0 Fetch	PC	PC, 4	ADD	ALU (direct)	IRWrite, PCWrite
S1 Decode	—	OldPC, imm	ADD	—	(A, B, ALUOut always load)
S2 MemAdr	—	A, imm	ADD	—	—
S3 MemRead	ALUOut	—	—	—	(MDR loads)
S4 MemWB	—	—	—	MDR	RegWrite
S5 MemWrite	ALUOut	—	—	—	MemWrite
S6 ExecuteR	—	A, B	funct	—	—
S7 ALUWB	—	—	—	ALUOut	RegWrite
S9 Branch	—	A, B	SUB/cmp	ALUOut	PCWrite if taken

The control unit is now a *sequential* circuit: a state register plus next-state and output logic. 12 states need $\lceil \log_2 12 \rceil = 4$ state bits.

1.6 Performance analysis

Execution time of an instruction = $\text{CPI} \times \text{clock cycle time}$. Execution time of a program = $\sum_{\text{instructions}} (\text{CPI} \times \text{cycle time}) = (\# \text{ instructions}) \times (\text{average CPI}) \times (\text{clock cycle time})$.

- **Single-cycle:** $\text{CPI} = 1$; clock cycle time = long.
- **Multicycle:** $\text{CPI} = \text{different for each instruction}$; average CPI hopefully small; clock cycle

time = short.

Worked Example 1.2: Multicycle versus single-cycle

Delays (Unit 2): memory 250 ps, register read 150 ps, ALU 200 ps, multiplexer 25 ps, register $t_{cq} + t_{su} = 50$ ps. The longest *step* is a memory access: $t_{C,mc} = 50 + 25 + 250 = 325$ ps (address MUX + memory + register overhead).

Instruction mix: loads 20%, stores 10%, R/I-ALU 50%, branches 17%, jumps 3%.

$$CPI_{mc} = 0.20 \cdot 5 + 0.10 \cdot 4 + 0.50 \cdot 4 + 0.17 \cdot 3 + 0.03 \cdot 3 = 4.00.$$

Time per instruction: multicycle $4.00 \times 325 = 1300$ ps; single-cycle 960 ps. **The multicycle machine is 35% slower!** Why? (1) Register overhead (50 ps) is paid 4 times per instruction instead of once. (2) The steps are unbalanced: register read (150) and ALU (200) steps waste part of a 325 ps cycle.

Now add a slow FP multiply (1600 ps in single-cycle; 2% of instructions). Single-cycle $t_C = 2110$ ps. Multicycle: the FP multiplier takes $\lceil 1600/325 \rceil = 5$ execute cycles, so FP multiply instructions take 8 cycles: $CPI = 0.98 \cdot 4.00 + 0.02 \cdot 8 = 4.08$, time $4.08 \times 325 = 1326$ ps.

Multicycle is now $2110/1326 = 1.59\times$ faster.

Multicycle designs pay off when instruction latencies vary widely — precisely the situation of CISC processors (and of microcoded implementations of complex instructions, next chapter).

Worked Example 1.3: Cycle count for factorial

The factorial loop of Unit 2 (blt, mul, addi, jal) executes 5 times plus setup and exit. Assume mul takes 4 execute cycles (7 total). Instruction counts: 3 addi (setup, 4 cycles each), 6 blt (3), 5 mul (7), 5 addi (4), 5 jal (3), 1 sw (4):

$$12 + 18 + 35 + 20 + 15 + 4 = 104 \text{ cycles, } CPI = 104/25 = 4.16, \quad 104 \times 325 \text{ ps} = 33.8 \text{ ns}$$

versus 24.0 ns single-cycle (where mul must fit in one cycle) — unless the single-cycle clock is lengthened to accommodate the multiplier, as computed in Unit 2.

Advanced Insight: Multicycle cores today

Very small embedded cores (e.g. SERV, the bit-serial RISC-V core with 32 cycles per bit-slice operation, or the multicycle PicoRV32 with $CPI \approx 4$) use exactly this approach to minimise area: PicoRV32 fits in roughly 750–2000 LUTs and reaches 250–450 MHz on modern FPGAs because its cycle is short. Where area and power matter more than throughput — sensor nodes, management controllers, security enclaves — multicycle is often the right answer.

Chapter summary

- Multicycle: $AS \rightarrow AS+MS_1 \rightarrow \dots \rightarrow AS'$; one short step per cycle; non-architectural registers hold micro-state.
- One memory and one ALU are reused across steps; control becomes an FSM.
- CPI depends on the instruction (3–5 here); $T = N \times \overline{CPI} \times t_C$.
- Multicycle beats single-cycle only when instruction latencies vary widely; register overhead and unbalanced steps otherwise make it slower. Pipelining (Chapter 3) keeps

the short cycle and recovers $\text{CPI} \approx 1$.

Exercises

E1.1 Recompute CPI_{mc} for a mix of 30% loads, 10% stores, 40% ALU, 20% branches.

Answer: $1.5 + 0.4 + 1.6 + 0.6 = 4.1$.

E1.2 If the memory were split into two 125 ps halves accessed over two cycles, what would the new t_C and CPI be? Is it better?

Answer: $t_C = 50 + 25 + 200 = 275$ ps (ALU-bound); loads 7, stores 6, ALU 5, branches and jumps 4:
 $\text{CPI} = 1.4 + 0.6 + 2.5 + 0.68 + 0.12 = 5.30$; $5.30 \times 275 = 1458$ ps — worse.

E1.3 List the multicycle steps of auipc and lui. Can they be done in 3 cycles?

Answer: Yes: S1 already computes OldPC+imm, so AUIPC can write ALUOut in cycle 3.

E1.4 Why does the decode step compute OldPC+imm even for non-branch instructions?

Answer: The ALU is otherwise idle in S1; computing the target speculatively saves a cycle for branches and jumps at no cost.

E1.5 Draw the state-transition table for the FSM of Figure 1.4 and implement the next-state logic as a ROM. How many bits?

Answer: Inputs: 4 state bits + 7 opcode bits ($\rightarrow$ 11 address bits); outputs: 4 next-state bits + ≈ 15 control bits: $2^{11} \times 19 \approx 39$ kbit — motivating the microsequencer of the next chapter.

2

Microprogrammed Control

A multicycle processor is controlled by a finite-state machine. When the FSM grows to dozens or hundreds of states — as it does for a CISC instruction set — designing it as random logic becomes impractical. **Microprogramming**, invented by Maurice Wilkes in 1951, replaces the next-state and output logic by a memory: each state becomes a *microinstruction*, the FSM becomes a *microprogram*, and the designer “programs” the control unit. This chapter explains the structure of a microprogrammed control unit, the LC-3b and single-bus examples from the lectures, horizontal versus vertical microcode, nanocode and millicode, and how to size a control store.

2.1 Microprogram control: the vocabulary

- The **control signals** associated with the current state form a **microinstruction**.
- The act of transitioning from one state to another — determining the next state and the microinstruction for the next state — is **microsequencing**.
- The **control store** stores the control signals for every possible state: it holds the microinstructions for the entire FSM.
- The **microsequencer** determines which set of control signals will be used in the next clock cycle (i.e. the next state).

Definition 2.1: Multicycle microarchitecture as an FSM

The instruction-processing cycle is divided into “states”; a stage in the instruction-processing cycle can take multiple states. A multicycle microarchitecture sequences from state to state to process an instruction. The behaviour of the machine in a state is completely determined by the control signals in that state, and the behaviour of the entire processor is specified fully by a finite-state machine. In a state (clock cycle), control signals control two things: **how the datapath should process the data**, and **how to generate the control signals for the next clock cycle**.

The microsequencer typically chooses among:

1. the **next sequential** microinstruction ($\mu\text{PC}+1$);
2. a **jump** to an address in the microinstruction (field J);
3. a **conditional branch** on a datapath condition (COND field selects which condition, e.g. memory ready, branch enable, addressing-mode bit);
4. a **dispatch** on the opcode: the instruction’s opcode (or a ROM indexed by it) gives the first microinstruction of that instruction’s routine (IRD: “instruction register decode”).

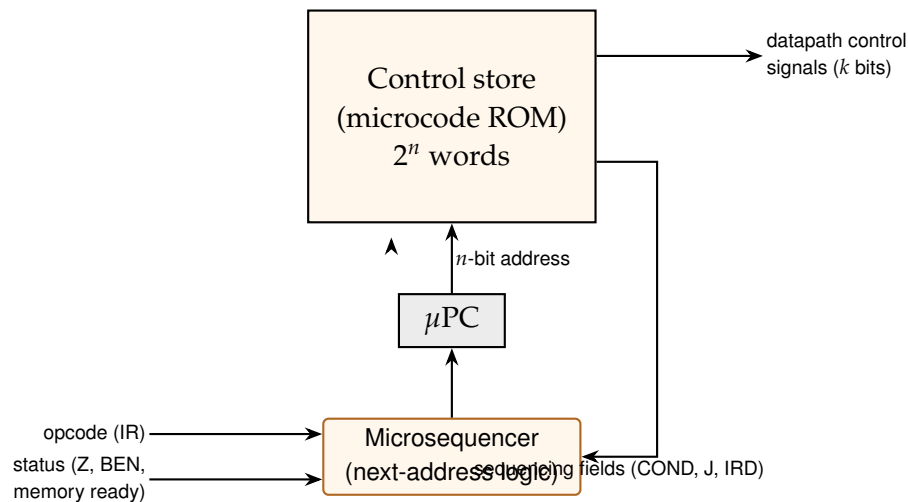
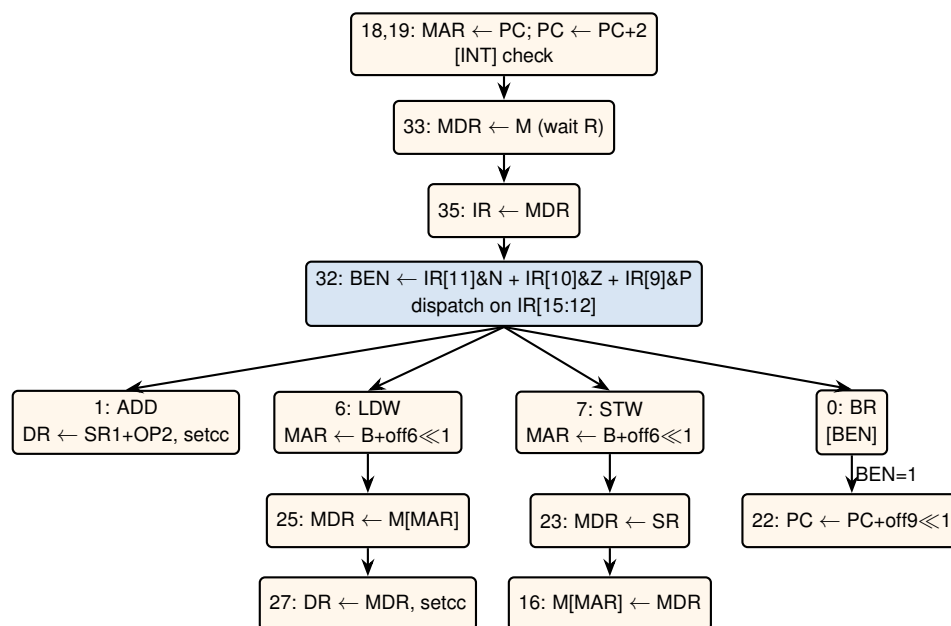


Figure 2.1: Microprogrammed control unit: the μ PC addresses the control store, whose output drives the datapath and tells the microsequencer how to choose the next μ PC.

2.2 Example: the LC-3b state machine

The LC-3b microarchitecture (Patt & Patel) is the canonical teaching example shown in the lecture. Its FSM has 31 used states (out of 64 addresses); every instruction starts at the fetch states (18/19 $\rightarrow$ 33 $\rightarrow$ 35 $\rightarrow$ 32), where state 32 performs a 16-way **dispatch** on the 4-bit opcode, so each opcode's routine begins at state number = opcode.



(plus states for AND, XOR, SHF, LEA, JMP, JSR, TRAP, LDB, STB, RTI; all routines return to state 18)

Figure 2.2: Simplified fragment of the LC-3b state machine. “wait R” states loop until the memory signals ready.

Worked Example 2.1: Sizing the LC-3b control store

The LC-3b microinstruction has 26 datapath-control bits (load enables LD.MAR, LD.MDR, LD.IR, LD.BEN, LD.REG, LD.CC, LD.PC; gate signals GatePC, GateMDR, GateALU,

GateMARMUX, GateSHF; MUX selects PCMUX[2], DRMUX, SR1MUX, ADDR1MUX, ADDR2MUX[2], MARMUX; ALUK[2]; MIO.EN, R.W, DATA.SIZE, LSHF1) and 9 microsequencer bits (IRD 1, COND 2, J 6). The control store has $2^6 = 64$ words:

$$64 \times (26 + 9) = 64 \times 35 = 2240 \text{ bits.}$$

Its microsequencer is tiny: if IRD = 1 the next address is {00, IR[15:12]} (dispatch); otherwise it is J with bits J[2:0] Ored with the selected condition (COND = 01: memory ready into J[1]; 10: BEN into J[2]; 11: IR[11] into J[0]). A 2240-bit ROM plus a few gates replaces a complex random-logic FSM.

2.3 Microcoding the single-bus processor

Recall the single-bus processor of Unit 1 (PC, MAR, MDR, IR, Y, Z, TEMP, registers on one internal bus). Each control step of an instruction becomes one microinstruction:

Table 2.1: Microroutine for ADD R1, (R3) on the single-bus processor (each line is one microinstruction; “End” returns to the fetch routine).

μ -address	Control signals asserted	Sequencing
0	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}	next
1	Z _{out} , PC _{in} , Y _{in} , WMFC	next (wait for memory)
2	MDR _{out} , IR _{in}	dispatch on opcode
... ADD with register-indirect source starts at, say, 25:		
25	R3 _{out} , MAR _{in} , Read	next
26	R1 _{out} , Y _{in} , WMFC	next
27	MDR _{out} , SelectY, Add, Z _{in}	next
28	Z _{out} , R1 _{in} , End	goto 0

2.4 Advantages of microprogrammed control

- **A new abstraction for the hardware designer.** The concept of a control store of microinstructions enables *microprogramming*. The designer can translate any desired operation into a sequence of microinstructions. All the designer needs to provide is:
 - the sequence of microinstructions needed to implement the desired operation;
 - the ability for the control logic to correctly sequence through the microinstructions;
 - any additional datapath elements and control signals needed (no need if the operation can be “translated” into existing control signals).
- **Very simple design to do powerful computation** by controlling the datapath with a sequencer. A high-level ISA is translated into microcode (a sequence of μ -instructions); microcode enables a minimal datapath to emulate an ISA. Microinstructions can be thought of as a *user-invisible ISA* (μ -ISA).
- **Easy extensibility of the ISA.** A new instruction can be supported by changing the microcode; complex instructions can be supported as a sequence of simple microinstructions.

- **Updatable machine behaviour.** A buggy implementation of an instruction can be fixed by changing the microcode in the field.

Advanced Insight: Microcode updates in practice

Modern x86 processors still contain a microcode sequencer (MSROM) for complex instructions (string operations, CPUID, transcendental x87 instructions, system instructions), and a small writable patch RAM loaded by the BIOS or OS at boot. Microcode updates have been used to disable a buggy transactional-memory feature (Intel TSX, 2014), to mitigate Spectre-class vulnerabilities (2018, adding the IBRS/IBPB controls), and to fix voltage-related instability (2024). The ability to change behaviour after shipping is worth a great deal when a design bug would otherwise mean a recall.

2.5 Horizontal and vertical microcode

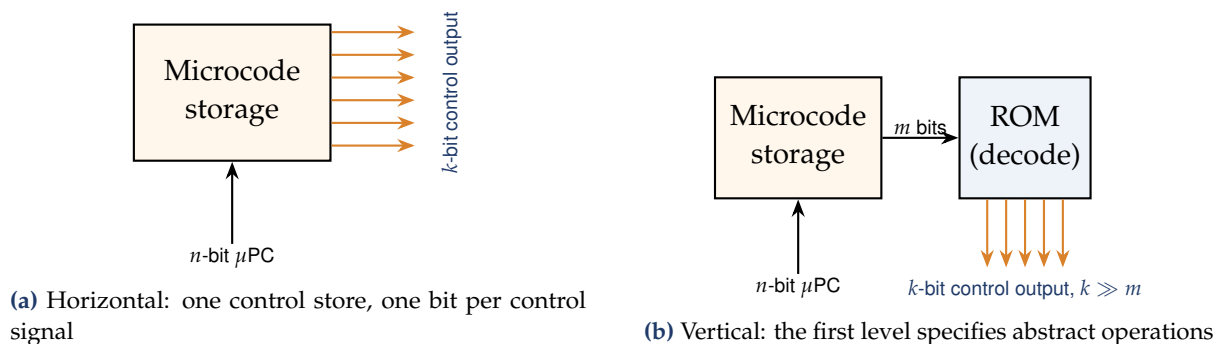


Figure 2.3: Horizontal versus vertical microcode. In vertical microcode a 1-bit signal in the first-level word can mean “do this register transfer” (or a combination of RTs), decoded by a second level into many control signals.

- **Horizontal microcode:** a single control store provides the control signals directly, one bit per signal. Maximum parallelism, fastest (no decoding), but wide microinstructions with many unused combinations.
- **Vertical microcode:** a two-level control store; the first level specifies abstract operations (e.g. “ $PC \leftarrow PC+4$ ”, “ $ALUOut \leftarrow A \text{ op } B$ ”) in encoded fields; a decoder or second ROM expands them into control signals. Narrow words, but less parallelism and an extra decode delay.

Worked Example 2.2: Horizontal versus vertical control-store size

A datapath has 48 control signals, and the microprogram has 512 microinstructions (9-bit μPC). Sequencing fields need 9 (next address) + 3 (condition select) = 12 bits.

Horizontal: $512 \times (48 + 12) = 30\,720$ bits.

Encoded fields: Suppose the 48 signals fall into groups of mutually exclusive signals: ALU operation (12 alternatives $\rightarrow$ 4 bits), bus source (10 $\rightarrow$ 4 bits), bus destination (14 $\rightarrow$ 4 bits), memory (read/write/none $\rightarrow$ 2 bits), and 8 independent single-bit signals. Width = $4 + 4 + 4 + 2 + 8 + 12 = 34$ bits: $512 \times 34 = 17\,408$ bits (43% smaller), plus four small field decoders.

Vertical: if only 64 distinct combinations of control signals actually occur, the first level stores a 6-bit index plus sequencing: $512 \times (6 + 12) = 9\,216$ bits; the second level stores the 64 combinations: $64 \times 48 = 3\,072$ bits. Total 12\,288 bits (60% smaller than horizontal), at the

cost of a second ROM access in the critical path.

2.6 Nanocode and millicode

- **Nanocode:** a level *below* traditional microcode — microprogrammed control for a subsystem (e.g. a complicated floating-point module) that acts as a slave in a microcontrolled datapath. We refer to a μ coded subsystem embedded in a μ coded system as nanocode (e.g. a “ μ coded” FPU implementation inside a “ μ coded” processor).
- **Millicode:** a level *above* traditional microcode — ISA-level subroutines that can be called by the microcontroller to handle complicated operations and system functions. Example: Heller and Farrell, “Millicode in an IBM zSeries processor,” IBM J. R&D, May/July 2004.
- In both cases we avoid complicating the main μ -controller. They can be thought of as “microcode” at different levels of abstraction.

Worked Example 2.3: The Motorola 68000’s two-level store

The MC68000 (1979) used a 544-word microROM of 17 bits and a 336-word nanoROM of 68 bits. Many microinstructions share identical control-signal patterns, so each microinstruction stores only sequencing information plus a pointer into the nanoROM:

$$544 \times 17 + 336 \times 68 = 9248 + 22\,848 = 32\,096 \text{ bits,}$$

versus $544 \times (68 + \approx 10) \approx 42\,400$ bits for a single horizontal store — a 24% saving that mattered on a die with only $\approx 68\,000$ transistors.

2.7 A microprogram for the multicycle RISC-V machine

The FSM of Chapter 1 can be microcoded directly. Fields: AddrSrc, ALUSrcA, ALUSrcB, ALUOp, ResultSrc, write enables (IR, PC, Reg, Mem), and a sequencing field (Seq: NEXT, DISPATCH1, DISPATCH2, FETCH, BRANCH).

Table 2.2: Microprogram for the multicycle RISC-V datapath (“—” = don’t care / not asserted).

Addr	Label	AddrSrc	SrcA	SrcB	ALUOp	Writes	Seq
0	FETCH	PC	PC	4	ADD	IR, PC	NEXT
1	DECODE	—	OldPC	imm	ADD	(A, B, ALUOut)	DISPATCH1
2	MEMADR	—	A	imm	ADD	—	DISPATCH2
3	MEMRD	ALUOut	—	—	—	(MDR)	NEXT
4	MEMWB	—	—	—	—	Reg (MDR)	FETCH
5	MEMWR	ALUOut	—	—	—	Mem	FETCH
6	EXECL	—	A	B	funct	—	NEXT
7	ALUWB	—	—	—	—	Reg (ALUOut)	FETCH
8	EXECL	—	A	imm	funct	—	GOTO 7
9	BRANCH	—	A	B	SUB	PC if taken	FETCH
10	JAL	—	OldPC	4	ADD	PC (ALUOut), Reg	FETCH

Dispatch ROM 1 (indexed by opcode): load/store → 2, R-type → 6, I-ALU → 8, branch → 9, JAL

→ 10. Dispatch ROM 2: load → 3, store → 5.

Worked Example 2.4: Microinstruction count and control-store size

11 microinstructions; with about 18 control bits plus a 3-bit Seq field and a 4-bit jump address, each word is 25 bits: $16 \times 25 = 400$ bits (μ PC of 4 bits). The two dispatch ROMs map 7-bit opcodes, but only the 9 relevant opcodes matter; implemented as logic they are a handful of gates. Executing `lw` visits addresses $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$: 5 cycles, matching the FSM.

2.8 Hardwired or microprogrammed?

Table 2.3: Hardwired versus microprogrammed control.

	Hardwired (random logic / PLA)	Microprogrammed
Speed	fast: signals generated by gates	slower: control-store access each cycle
Design effort for complex ISAs	very high; error-prone	systematic; “write a program”
Changes, bug fixes	require new silicon	change the ROM / patch RAM
Best fit	RISC, pipelined cores, simple FSMs	CISC, complex/multi-step instructions
Examples	RISC-V, ARM, MIPS cores	IBM System/360 (most models), VAX, 68000, x86 complex ops

The IBM System/360 (1964) used microprogramming to implement *one* architecture across a family of machines with very different datapaths: the narrow Model 30 executed many microinstructions per instruction, the wide Model 65 fewer. Microcode made the architecture/organization separation of Unit 1 economically possible.

Chapter summary

- Microinstruction = the control signals of one state; microsequencing = choosing the next state; control store = all microinstructions; microsequencer = next- μ PC logic (next, jump, conditional branch, opcode dispatch).
- LC-3b: 64×35 -bit control store; dispatch on IR[15:12]; tiny microsequencer.
- Microprogramming gives a μ -ISA abstraction, easy ISA extension and field fixes.
- Horizontal (wide, fast) vs vertical (narrow, decoded) microcode; nanocode below and millicode above microcode.

Exercises

E2.1 A control unit has 30 control signals and 200 microinstructions with 2-way conditional branching. Compute the horizontal control-store size.

Answer: 8-bit μ PC; word = $30 + 8$ (next address) + 2 (condition select) = 40 bits; $256 \times 40 = 10\,240$ bits (or $200 \times 40 = 8000$ if sized exactly).

E2.2 Write the microroutine for `SUB R2, R4` and for a `JMP (PC ← R5)` on the single-bus processor.

E2.3 Add the instruction `lw.post rd, imm(rs1)` (load, then $rs1 \leftarrow rs1 + imm$) to the RISC-V microprogram of Table 2.2. How many new microinstructions and cycles?

Answer: After MEMWB add one microinstruction: $SrcA = A$, $SrcB = imm$, ADD, write $Reg[rs1]$ (needs a write-address select): 6 cycles.

E2.4 Why does vertical microcode lengthen the critical path? Where in the cycle does the second-level ROM access happen?

Answer: The control signals are available only after two ROM accesses; the decode must complete early in the cycle or be pipelined.

E2.5 Compare the number of bits needed to store the multicycle FSM as (a) a single ROM indexed by {state, opcode} and (b) a microcode store with dispatch ROMs.

Answer: (a) $2^{4+7} \times (4 + 18) \approx 45 \text{ kbit}$; (b) $16 \times 25 + 2 \times 2^7 \times 4 \approx 1.4 \text{ kbit}$.

3

Pipelining Fundamentals

The multicycle design keeps most hardware idle in most cycles: while the ALU executes one instruction, the memory, register file and PC logic wait. **Pipelining** puts that idle hardware to work by overlapping the execution of several instructions, each in a different stage — exactly like an assembly line. This chapter develops the idea, quantifies throughput, latency and cost, builds the five-stage RISC-V pipeline with its pipeline registers and control, and lists the reasons real pipelines fall short of the ideal.

3.1 The idea

More systematically, pipelining means:

- pipeline the execution of multiple instructions — the “assembly-line processing” of instructions;
- divide the instruction processing cycle into distinct **stages** of processing;
- ensure there are enough hardware resources to process one instruction in each stage;
- process a **different instruction in each stage**: instructions consecutive in program order are processed in consecutive stages.

Benefit: it increases instruction-processing *throughput* ($1/\text{CPI}$), without (ideally) increasing the latency of any single instruction.

3.1.1 The laundry analogy

Four loads of laundry (A, B, C, D) each need washing, drying, folding and putting away, 30 minutes each.

Each load still takes 2 hours (latency unchanged), but a load finishes every 30 minutes (throughput $4\times$). The speedup for 4 loads is only $8/3.5 = 2.29$ because of the time to fill and drain the pipeline; for many loads it approaches 4. The lecture’s further variants show what goes wrong in practice: if the dryer takes twice as long, the dryer becomes the bottleneck and loads queue up; if two loads need the same machine at once, one must wait.

3.2 The ideal pipeline

Goal: increase throughput with little increase in cost (hardware cost, in the case of instruction processing). The ideal pipeline assumes:

1. **Repetition of identical operations:** the same operation is repeated on a large number of different inputs (e.g. all laundry loads go through the same steps).

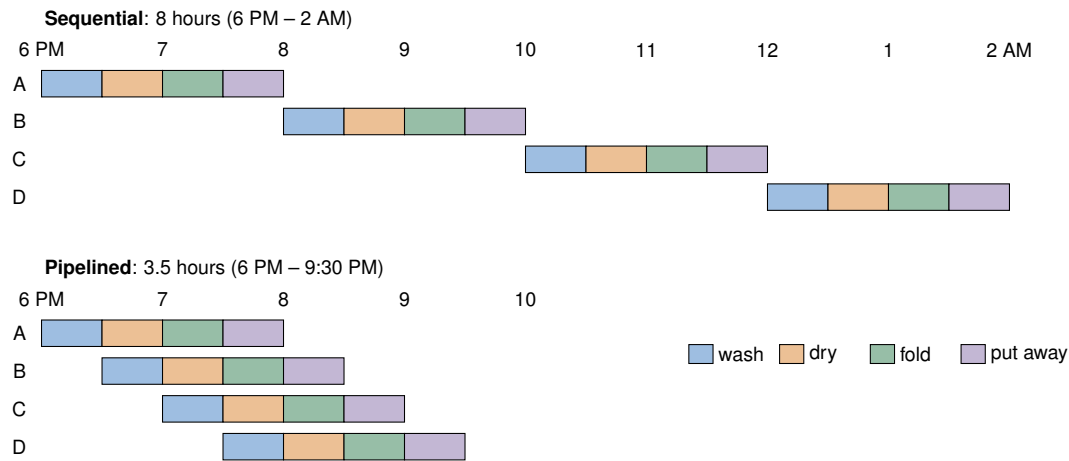


Figure 3.1: Laundry example: sequential versus pipelined processing of four loads.

2. **Repetition of independent operations:** no dependences between repeated operations.
3. **Uniformly partitionable suboperations:** processing can be evenly divided into uniform-latency suboperations that do not share resources.

Fitting examples: automobile assembly lines, doing laundry, and arithmetic pipelines such as a floating-point multiplier.

3.2.1 Throughput

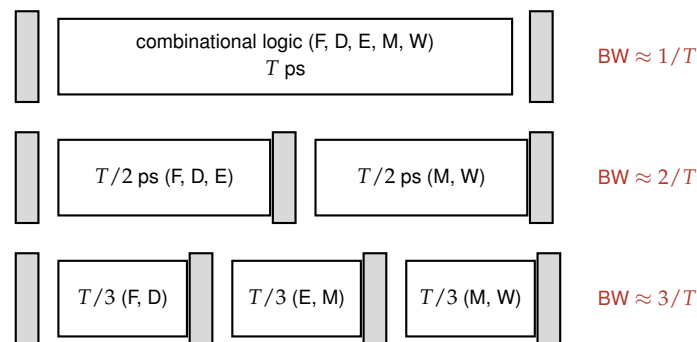


Figure 3.2: Ideal pipelining: cutting combinational logic of delay T into k equal stages separated by latches multiplies bandwidth by k (ignoring latch overhead).

3.2.2 Throughput and cost with latch overhead

Let the unpipelined logic have delay T and gate cost G ; let each pipeline latch (register) level add delay S and cost L .

$$\text{Non-pipelined: } BW = \frac{1}{T + S}, \quad \text{Cost} = G + L, \quad (3.1)$$

$$k\text{-stage pipelined: } BW_k = \frac{1}{T/k + S}, \quad \text{Cost}_k = G + Lk. \quad (3.2)$$

As $k \rightarrow \infty$, $BW_{\max} = 1/(1 \text{ gate delay} + S)$: latch delay reduces throughput (switching overhead between stages), and latches increase hardware cost.

Worked Example 3.1: Throughput and the optimal pipeline depth

Let $T = 4000$ ps of logic, $S = 100$ ps per latch, $G = 200\,000$ gates and $L = 6000$ gates per latch level (a 64-bit-wide datapath with control).

k	Cycle $T/k + S$ (ps)	BW (GIPS)	Cost (kgates)	BW/Cost (per Mgate)
1	4100	0.244	206	1.18
5	900	1.111	230	4.83
10	500	2.000	260	7.69
20	300	3.333	320	10.42
26	254	3.937	356	11.06
40	200	5.000	440	11.36
80	150	6.667	680	9.80

Maximizing $\text{BW/Cost} = 1/[(T/k + S)(G + Lk)]$ gives (Kogge's result)

$$k_{\text{opt}} = \sqrt{\frac{GT}{LS}} = \sqrt{\frac{200\,000 \times 4000}{6000 \times 100}} = \sqrt{1333} \approx 36.5.$$

Throughput keeps rising with k , but each extra stage buys less: from 40 to 80 stages, throughput rises 33% while cost rises 55%. In real processors the optimum is much smaller (≈ 10 – 20 stages) because the ideal assumptions fail: dependences and branch mispredictions cost more cycles in deeper pipelines (Chapters 4–6).

3.3 Pipelining the RISC-V datapath

3.3.1 Dividing the single-cycle datapath into stages

Using the classic stage delays from the lecture:

200 ps	100 ps	200 ps	200 ps	100 ps
IF: instruction fetch	ID: decode / register read	EX: execute / address calc.	MEM: memory access	WB: write back

The single-cycle clock must cover the sum, 800 ps (for a load). The pipelined clock must cover the slowest stage, 200 ps; the ID and WB stages waste 100 ps each.

Worked Example 3.2: Speedup of the five-stage pipeline

For N instructions on a k -stage pipeline with cycle t_p , execution takes $(k + N - 1)t_p$; single-cycle takes Nt_s .

Three loads: single-cycle $3 \times 800 = 2400$ ps; pipelined $(5 + 3 - 1) \times 200 = 1400$ ps; speedup 1.71.

$N = 10^6$: single-cycle $800 \mu\text{s}$; pipelined $(10^6 + 4) \times 200 \text{ ps} \approx 200 \mu\text{s}$; speedup $\rightarrow 800/200 = 4.0$, not 5, because the stages are unbalanced (the ideal speedup k needs every stage to take $800/5 = 160$ ps). With 20 ps of pipeline-register overhead per stage, the cycle becomes 220 ps and the speedup 3.64.

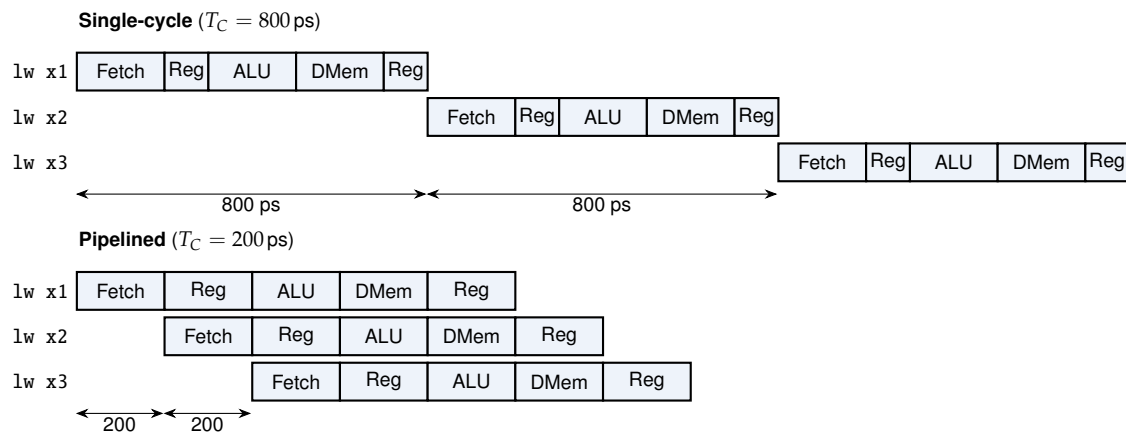


Figure 3.3: Instruction pipeline throughput for three loads: 2400 ps sequentially versus 1400 ps pipelined; in steady state one instruction completes every 200 ps.

3.3.2 Pipeline registers

Between consecutive stages, **pipeline registers** hold everything the later stages will need: the stage's results *and* the information (register numbers, control signals, immediates, PC) that travels with the instruction. They are named after the stages they separate: IF/ID, ID/EX, EX/MEM, MEM/WB.

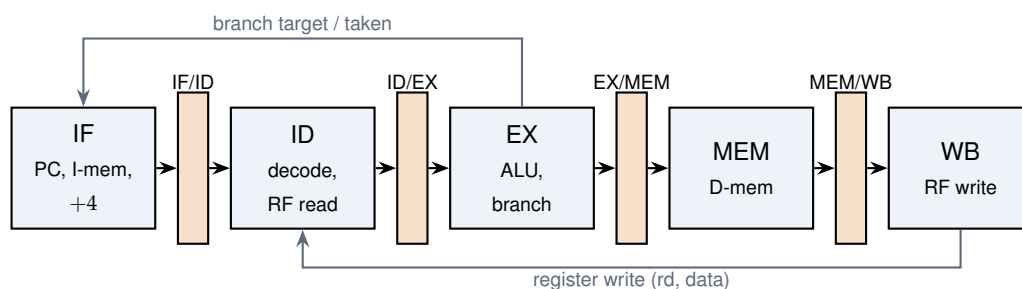


Figure 3.4: Five-stage pipeline with its four pipeline registers. Two paths flow backwards: register write-back (WB $\rightarrow$ ID) and the branch outcome (EX $\rightarrow$ IF); both are sources of hazards.

Worked Example 3.3: How wide are the pipeline registers?

Register	Contents	Bits
IF/ID	PC (32), instruction (32)	64
ID/EX	PC (32), rs1 data (32), rs2 data (32), imm (32), rd (5), rs1/rs2 numbers (10), funct3 (3), control (EX: 7, MEM: 2, WB: 3)	158
EX/MEM	ALU result (32), rs2 data for store (32), PC+4 (32), rd (5), control (MEM 2, WB 3), taken (1)	107
MEM/WB	ALU result (32), load data (32), PC+4 (32), rd (5), control (WB 3)	104
Total		≈ 433 flip-flops

At ≈ 24 transistors per flip-flop that is $\approx 10\,400$ transistors of pipeline registers — the “*Lk*” cost term. Their clock-to-Q plus setup ($S \approx 50\text{ ps}$) is paid in every stage.

3.3.3 Illustration of pipeline operation

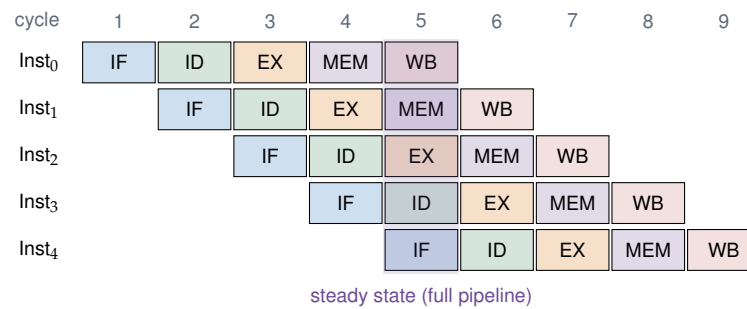


Figure 3.5: In cycle 5 all five stages are busy, each with a different instruction.

3.4 Control in a pipeline

For a given instruction the pipeline needs the *same* control signals as the single-cycle design, but the signals are required in *different cycles*, depending on the stage.

- **Option 1:** decode once, using the same logic as single-cycle, and buffer the signals in the pipeline registers until consumed (EX signals are used and dropped at ID/EX → EX; MEM signals travel to EX/MEM; WB signals travel to MEM/WB).
- **Option 2:** carry the relevant “instruction word/field” down the pipeline and decode locally within each stage (or in a previous stage).

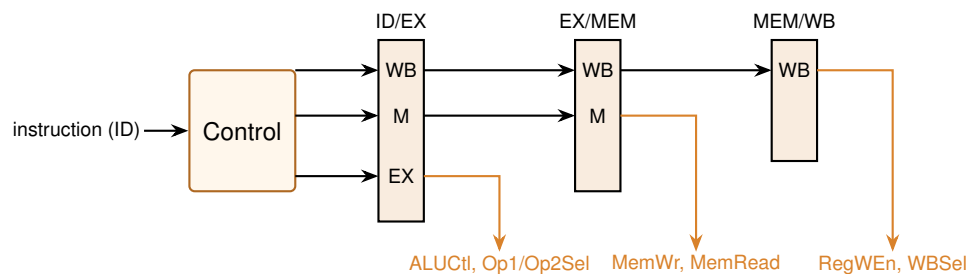


Figure 3.6: Option 1: control signals are generated in ID and travel with the instruction; each group is consumed in its stage.

3.5 From ideal to practical pipelines

- **Identical operations ... NOT!** Different instructions do not all need the same stages. Forcing different instructions to go through the same pipe stages leaves some stages idle for some instructions: *external fragmentation*.
- **Uniform suboperations ... NOT!** Different pipeline stages do not have the same latency, yet each stage must be controlled by the same clock: some pipe stages are too fast but all take the clock-cycle time: *internal fragmentation*.
- **Independent operations ... NOT!** Instructions are not independent of each other. We need to detect and resolve inter-instruction dependences to ensure the pipeline provides correct results: *pipeline stalls* (the pipeline is not always moving).

Worked Example 3.4: Measuring internal and external fragmentation

Internal: stage delays $\{200, 100, 200, 200, 100\}$ in a 200 ps cycle: useful time 800 of $5 \times 200 = 1000$ ps $\Rightarrow$ 20% wasted per instruction.

External: an R-type instruction does nothing in MEM, a store nothing in WB, a branch nothing in MEM or WB. With the mix 50% ALU, 20% load, 10% store, 20% branch/jump, the number of idle stage-slots per instruction is $0.5 \cdot 1 + 0.2 \cdot 0 + 0.1 \cdot 1 + 0.2 \cdot 2 = 1.0$: one of five stages is idle on average, another 20%. (Idle stages cost energy unless gated, but not throughput.)

3.5.1 Issues in pipeline design

- **Balancing work in pipeline stages:** how many stages and what is done in each.
- **Keeping the pipeline correct, moving and full** in the presence of events that disrupt pipeline flow: handling dependences (data and control), handling resource contention, handling long-latency (multicycle) operations.
- **Handling exceptions and interrupts.**
- **Advanced: improving pipeline throughput** — minimizing stalls.

Advanced Insight: How deep should a pipeline be?

The Intel Pentium 4 (2000–2004) pushed pipeline depth to 20 and then 31 stages to reach 3.8 GHz; its branch misprediction penalty grew accordingly, and its successor (Core, 2006) returned to ≈ 14 stages with a much higher IPC. Studies by Hartstein and Puzak (2002) and Sprangle and Carmean (2002) found the performance-optimal logic depth per stage to be about 6–8 FO4 inverter delays when power is ignored, and considerably more when power is included, because every extra latch burns clock power. Modern high-performance cores use 13–20 stages; microcontroller cores 2–5.

Chapter summary

- Pipelining overlaps instructions in different stages; throughput rises up to $k\times$, latency does not improve.
- Ideal pipeline: identical, independent, uniformly partitionable operations. Real pipelines suffer external and internal fragmentation and dependences.
- $BW_k = 1/(T/k + S)$, $Cost_k = G + Lk$, $k_{opt} = \sqrt{GT/(LS)}$.
- The five-stage RISC-V pipeline: IF, ID, EX, MEM, WB with registers IF/ID, ID/EX, EX/MEM, MEM/WB that carry data and control; N instructions take $k + N - 1$ cycles.

Exercises

E3.1 A 900 ps combinational datapath is split into stages of 150, 250, 200, 300 ps with 40 ps register overhead. Compute the cycle, the latency of one instruction, and the throughput speedup over the unpipelined version (with one register).

Answer: Cycle 340 ps; latency $4 \times 340 = 1360$ ps; speedup $940/340 = 2.76$.

E3.2 Rebalance the previous pipeline into 5 stages of equal delay (assume any split is possible).

New speedup?

Answer: $900/5 + 40 = 220$ ps; $940/220 = 4.27$.

E3.3 Derive k_{opt} by maximizing $1/[(T/k + S)(G + Lk)]$.

E3.4 How many cycles does a 5-stage pipeline need for 100 instructions without stalls? What is the CPI?

Answer: 104 cycles; CPI = 1.04.

E3.5 Which control signals of Unit 2 belong to each stage of the pipeline?

Answer: IF: PCSel; ID: ImmSel; EX: Op1Sel, Op2Sel, ALUCtl, BrUn; MEM: MemWr; WB: WBSel, RegWEn.

4

Data Dependences and Pipeline Hazards

The ideal pipeline assumes independent instructions. Real programs are full of dependences: an instruction often needs the result of the one just before it. This chapter classifies the causes of pipeline stalls, analyses which dependences are dangerous in the five-stage pipeline, and develops the two hardware mechanisms that preserve correctness and performance: **interlocking** (detect and stall) and **data forwarding** (bypassing).

4.1 Causes of pipeline stalls

A **stall** is a condition in which the pipeline stops moving. Its causes:

- **Resource contention** (structural hazards);
- **Dependences** between instructions — *data* dependences and *control* dependences;
- **Long-latency (multicycle) operations**.

Dependences dictate ordering requirements between instructions. Resource contention is sometimes called a *resource dependence*.

4.2 Resource contention

Resource contention happens when instructions in two pipeline stages need the same resource.

- **Solution 1: eliminate the cause of contention.** Duplicate the resource or increase its throughput: e.g. use separate instruction and data memories (caches), or use multiple ports for memory structures.
- **Solution 2: detect the resource contention and stall one of the contending stages.** Which stage should stall? Usually the younger instruction, so that older instructions keep moving and eventually free the resource.

Worked Example 4.1: Cost of a unified single-ported memory

If the five-stage pipeline had a single memory port, every load or store in MEM would block the instruction fetch of a younger instruction in the same cycle. With 30% loads/stores, each causing one stall cycle:

$$\text{CPI} = 1 + 0.30 \times 1 = 1.30 \quad \Rightarrow \quad 23\% \text{ slower.}$$

Split L1 instruction and data caches (a Harvard organization at L1) remove this hazard entirely. Similarly, the register file is given two read ports and one write port, and is written in the first half of the cycle and read in the second half, so that ID and WB never conflict.

4.3 Data dependences

Definition 4.1: Types of data dependence

For instructions I_a (earlier) and I_b (later):

- **Flow dependence** (true data dependence, **read-after-write, RAW**): I_b reads a register that I_a writes.
- **Anti dependence** (**write-after-read, WAR**): I_b writes a register that I_a reads.
- **Output dependence** (**write-after-write, WAW**): both write the same register.

Flow (RAW)	Anti (WAR)	Output (WAW)
$r_3 \leftarrow r_1 \text{ op } r_2$	$r_3 \leftarrow r_1 \text{ op } r_2$	$r_3 \leftarrow r_1 \text{ op } r_2$
$r_5 \leftarrow r_3 \text{ op } r_4$	$r_1 \leftarrow r_4 \text{ op } r_5$	$r_5 \leftarrow r_3 \text{ op } r_4$
		$r_3 \leftarrow r_6 \text{ op } r_7$

Which ones cause stalls in a pipelined machine? For all of them we need to ensure that the semantics of the program are correct. **Flow dependences always need to be obeyed**, because they constitute a true dependence on a *value*. **Anti and output dependences exist due to the limited number of architectural registers**: they are dependences on a *name*, not a value, and can be removed by renaming.

4.3.1 Handling flow dependences

Anti and output dependences are easier to handle: in an in-order pipeline that writes the destination in one stage and in program order, they never cause problems. Flow dependences are more interesting. There are five fundamental ways of handling them:

1. **Detect and wait** until the value is available in the register file (interlocking + stall).
2. **Detect and forward/bypass** the data to the dependent instruction.
3. **Detect and eliminate** the dependence at the software level (no need for the hardware to detect it).
4. **Predict** the needed value(s), execute “speculatively”, and verify.
5. **Do something else** (fine-grained multithreading): no need to detect.

4.4 Interlocking

Interlocking is the detection of dependences between instructions in a pipelined processor to guarantee correct execution.

- **Software-based interlocking**: the compiler guarantees that dependent instructions are far enough apart, inserting nops if necessary. The original MIPS was named “Microprocessor without Interlocked Pipeline Stages”.
- **Hardware-based interlocking**: the hardware detects dependences and stalls.

4.4.1 Detection technique 1: scoreboarding

Each register in the register file has a **Valid bit**. An instruction that writes a register resets its Valid bit (when it is decoded) and sets it again when it writes back. An instruction in Decode checks whether all its source *and destination* registers are Valid: if yes, no stall; if no, it stalls.

- + Simple: one bit per register.
- Stalls for all types of dependences (RAW, WAR, WAW), not only flow dependences.

4.4.2 Detection technique 2: combinational dependence-check logic

Special logic checks whether any instruction in later stages is supposed to write to any source register of the instruction being decoded. If yes, stall; if no, no flow dependence.

- + No need to stall on anti and output dependences.
- More complex logic than a scoreboard, and it grows as the pipeline becomes deeper and wider (think superscalar execution: every source compared with every in-flight destination).

4.5 Analysing register dependences in the five-stage pipeline

Table 4.1: Where each instruction type reads and writes the register file.

Stage	R/I-type	LW	SW	Br	J	JR
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

For a given pipeline, when is there a potential conflict between two data-dependent instructions? It depends on the dependence type (RAW, WAR, WAW), the instruction types involved, and the *distance* between the two instructions.

The distance rule

Let I_i be the producer and I_j the consumer, $\text{dist}(i, j) = j - i$ instructions apart. For a RAW dependence where I_j reads in stage X (ID) and I_i writes in stage Y (WB):

- $\text{dist}(i, j) \leq \text{dist}(X, Y) \Rightarrow$ **unsafe** to keep j moving;
- $\text{dist}(i, j) > \text{dist}(X, Y) \Rightarrow$ **safe**.

In the five-stage pipeline $\text{dist}(\text{ID}, \text{WB}) = 3$ (with write-before-read in the same cycle; 4 otherwise). WAR and WAW are always safe because reads happen in ID, before any later instruction's WB, and writes happen in program order.

So instructions I_A and I_B (I_A before I_B) have a hazardous RAW dependence iff

- I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW), and
- $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$.

In other words, if I_B in the ID stage reads a register written by I_A in the EX, MEM or WB stage, then the operand required by I_B is not yet in the register file.

4.6 Stalling

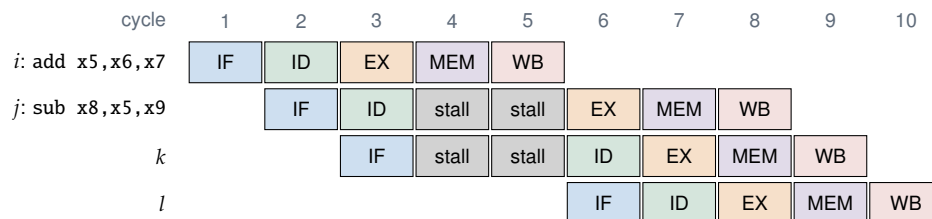


Figure 4.1: RAW dependence with distance 1 and no forwarding: j is held in ID (cycles 4–5, shaded) until i writes x5 in WB in cycle 5; with write-before-read, j reads the new value in cycle 5 and enters EX in cycle 6. Two bubbles flow down the pipeline behind i ; k is held in IF. (Without write-before-read the penalty would be three cycles.)

Stall = make the dependent instruction wait until its source data value is available:

1. stop all up-stream stages (IF and ID keep their instructions);
2. drain all down-stream stages (older instructions continue; bubbles fill the gap).

4.6.1 Implementing stalls

- Disable PC and IF/ID latching: the stalled instruction stays in its stage (ID) and the one behind it stays in IF.
- Insert “invalid” instructions (nops, *bubbles*) into the stage following the stalled one: clear the control signals in ID/EX ($\text{RegWen} = \text{MemWr} = 0$ and a valid bit = 0).

It is crucial that the EX, MEM and WB stages continue to advance normally during stall cycles.

4.6.2 Stall condition logic

Helper functions: $\text{rs}(I)$ returns the rs field of instruction I ; $\text{use_rs}(I)$ returns true if I requires $\text{RF}[\text{rs}]$ and $\text{rs} \neq \text{x0}$. Stall when:

$$\begin{aligned}
 &(\text{rs}(\text{IR}_{\text{ID}}) == \text{dest}_{\text{EX}}) \ \&\& \ \text{use_rs}(\text{IR}_{\text{ID}}) \ \&\& \ \text{RegWrite}_{\text{EX}} \quad \text{or} \\
 &(\text{rs}(\text{IR}_{\text{ID}}) == \text{dest}_{\text{MEM}}) \ \&\& \ \text{use_rs}(\text{IR}_{\text{ID}}) \ \&\& \ \text{RegWrite}_{\text{MEM}} \quad \text{or} \\
 &(\text{rs}(\text{IR}_{\text{ID}}) == \text{dest}_{\text{WB}}) \ \&\& \ \text{use_rs}(\text{IR}_{\text{ID}}) \ \&\& \ \text{RegWrite}_{\text{WB}} \quad \text{or} \\
 &(\text{rt}(\text{IR}_{\text{ID}}) == \text{dest}_{\text{EX}}) \ \&\& \ \text{use_rt}(\text{IR}_{\text{ID}}) \ \&\& \ \text{RegWrite}_{\text{EX}} \quad \text{or} \\
 &(\text{rt}(\text{IR}_{\text{ID}}) == \text{dest}_{\text{MEM}}) \ \&\& \ \text{use_rt}(\text{IR}_{\text{ID}}) \ \&\& \ \text{RegWrite}_{\text{MEM}} \quad \text{or} \\
 &(\text{rt}(\text{IR}_{\text{ID}}) == \text{dest}_{\text{WB}}) \ \&\& \ \text{use_rt}(\text{IR}_{\text{ID}}) \ \&\& \ \text{RegWrite}_{\text{WB}}
 \end{aligned}$$

(rt is the MIPS name for RISC-V’s rs2.) If the register file writes in the first half of the cycle and reads in the second, the two WB terms can be dropped (distance 3 becomes safe).

Worked Example 4.2: Cost of the stall logic

Each term compares two 5-bit register numbers (5 XNORs + a 5-input AND) and ANDs in two more signals: about 8 gates. Six terms plus an OR: about 50 gates — tiny. For a 4-wide superscalar with 8 sources in decode and 12 destinations in flight, the comparator count is $8 \times 12 = 96$ five-bit comparators: the logic grows as $(\text{width})^2 \times (\text{depth})$, the “disadvantage” noted for combinational dependence checking.

4.6.3 Impact of stalls on performance

Each stall cycle corresponds to one lost cycle in which no instruction can be completed. For a program with N instructions and S stall cycles,

$$\text{average CPI} = \frac{N + S}{N}.$$

S depends on the frequency of RAW dependences, the exact distance between the dependent instructions, and the distance between dependences (suppose i_1, i_2, i_3 all depend on i_0 : once i_1 's dependence is resolved, i_2 and i_3 must be okay too).

Worked Example 4.3: Stalls in a bubble-sort inner loop

The lecture's example is the MIPS code for `for (j=i-1; j>=0 && v[j] > v[j+1]; j--)` { ... } (Patterson & Hennessy):

```

    addi    $s1, $s0, -1      # j = i - 1
for2tst:  slti    $t0, $s1, 0    # 3 stalls (waits for $s1 from addi)
    bne     $t0, $zero, exit2  # 3 stalls (waits for $t0)
    sll     $t1, $s1, 2
    add     $t2, $a0, $t1      # 3 stalls (waits for $t1)
    lw      $t3, 0($t2)        # 3 stalls (waits for $t2)
    lw      $t4, 4($t2)
    slt     $t0, $t4, $t3      # 3 stalls (waits for $t4)
    beq     $t0, $zero, exit2  # 3 stalls (waits for $t0)
    ...

```

(RISC-V equivalent: `addi s1,s0,-1; slti t0,s1,0; bne t0,x0,exit2; slli t1,s1,2; add t2,a0,t1; lw t3,0(t2); lw t4,4(t2); slt t0,t4,t3; beq t0,x0,exit2.`)

Without forwarding, each back-to-back RAW costs 3 stalls (no write-before-read assumed in the lecture's count). The 9 instructions shown incur 6 such dependences:

$$\text{CPI} = \frac{9 + 6 \times 3}{9} = \frac{27}{9} = 3.0.$$

With write-before-read in the register file, each costs 2 stalls: $\text{CPI} = (9 + 12)/9 = 2.33$. The pipeline is barely better than the multicycle machine — dependences are the norm, not the exception, in real code.

4.7 Data forwarding (bypassing)

Problem: a consumer (dependent) instruction has to wait in the decode stage until the producer instruction writes its value into the register file. **Goal:** we do not want to stall the pipeline unnecessarily. **Observation:** the data value needed by the consumer can be supplied directly

from a later stage in the pipeline, instead of only from the register file. **Idea:** add additional dependence-check logic and data-forwarding paths (buses) to supply the producer's value to the consumer right after the value is available. **Benefit:** the consumer can move in the pipeline until the point the value can be supplied $\Rightarrow$ less stalling.

The register file is a communication abstraction

It is intuitive to think of the register file as state: “add rx, ry, rz” literally means get values from RF[ry] and RF[rz] and put the result in RF[rx]. But the RF is just a part of a *communication abstraction*: “add rx, ry, rz” means (1) get the results of the last instructions to define the values of RF[ry] and RF[rz]; (2) until another instruction redefines RF[rx], younger instructions that refer to RF[rx] should use this instruction's result. What matters is to maintain the correct “data flow” between operations — recall the dataflow model of Unit 1.

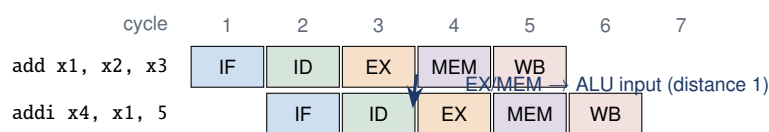


Figure 4.2: Forwarding: the add's result, available at the end of its EX stage (in the EX/MEM register), is fed directly to the ALU input of the addi in the next cycle. No stall.

4.7.1 Resolving RAW with forwarding

Instructions I_A and I_B have a RAW dependence iff I_B reads a register written by I_A and $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$. In other words, if I_B in ID reads a register written by I_A in EX, MEM or WB, then the operand required by I_B is not yet in the RF $\Rightarrow$ retrieve the operand from the datapath instead of the RF $\Rightarrow$ retrieve it from the *youngest* definition if multiple definitions are outstanding.

4.7.2 Forwarding logic

For the ALU's first operand in EX (and symmetrically for the second):

```
if      (rs1_EX != 0) && (rs1_EX == rd_MEM) && RegWrite_MEM then
    forward operand from MEM stage      // dist = 1
else if (rs1_EX != 0) && (rs1_EX == rd_WB) && RegWrite_WB  then
    forward operand from WB stage       // dist = 2
else
    use operand from register file      // dist >= 3
```

The order of the tests implements “youngest definition wins”: if both the instruction in MEM and the one in WB write the register, the MEM-stage value is newer.

4.7.3 The load-use hazard

A load's value is available only at the end of MEM. A dependent instruction immediately after the load needs it at the beginning of EX — one cycle too early. Forwarding cannot travel back in time: one stall (bubble) is unavoidable.

The stall condition becomes: $\text{MemRead}_{\text{EX}} \ \&\& \ (\text{rd}_{\text{EX}} == \text{rs1}_{\text{ID}} \ || \ \text{rd}_{\text{EX}} == \text{rs2}_{\text{ID}}) \ \&\& \ \text{rd}_{\text{EX}} \neq 0$.

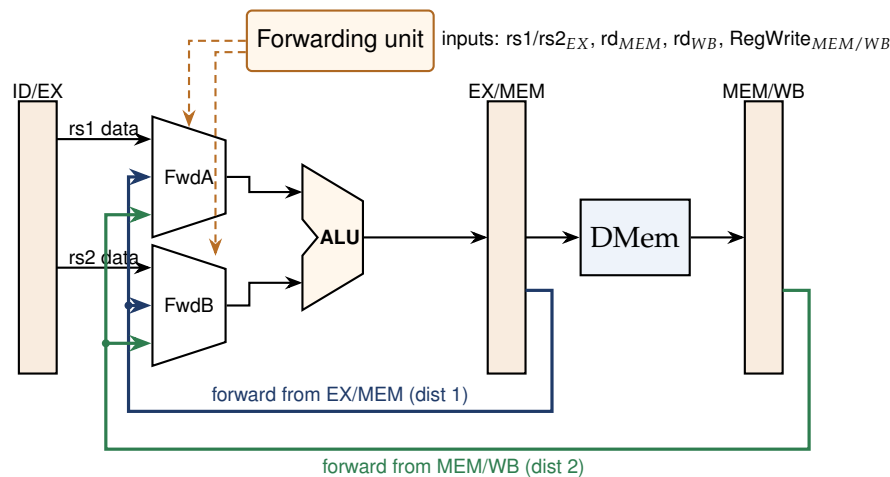


Figure 4.3: Forwarding datapath: two 3:1 multiplexers in front of the ALU select the register-file value, the EX/MEM value or the MEM/WB value.

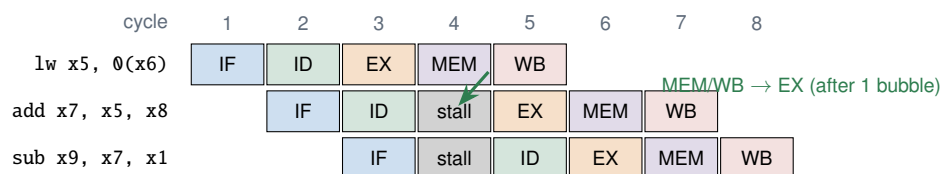


Figure 4.4: Load-use hazard: even with forwarding, the consumer of a load must wait one cycle.

Worked Example 4.4: The bubble-sort loop with forwarding

With full forwarding, the only remaining stall in the 9-instruction sequence is between `lw $t4, 4($t2)` and `slt $t0, $t4, $t3` (load-use): the lecture's version shows a `nop` inserted there. (We assume branches resolve in EX, so the ALU results feeding `bne/req` are forwarded in time; branch penalties are treated in Chapter 5.)

$$\text{CPI} = \frac{9 + 1}{9} = 1.11 \quad \text{versus } 3.0 \text{ without forwarding: } 2.7 \times \text{ faster.}$$

The compiler can often remove the remaining bubble by *scheduling*: moving an independent instruction between the load and its use. Swapping the two loads does not help (both feed `slt`), but in the full loop the `addi $s1, $s1, -1` of the next iteration could be hoisted into the slot.

Worked Example 4.5: CPI with load-use stalls

In a program, 25% of instructions are loads, and 40% of loads are immediately followed by a dependent instruction that the compiler could not separate.

$$\text{CPI} = 1 + 0.25 \times 0.40 \times 1 = 1.10.$$

In a pipeline with a 2-cycle data cache (deeper pipeline), the load-use penalty is 2 cycles: $\text{CPI} = 1.20$ — one reason why L1 data-cache latency is so carefully optimized.

4.8 Who handles dependences: hardware or software?

- What is the role of the hardware versus the software in data-dependence handling? Software-based interlocking versus hardware-based interlocking: who inserts and manages the pipeline bubbles? Who finds the independent instructions to fill “empty” pipeline slots?
- What is the role of hardware versus software in the order in which instructions are executed in the pipeline? Software-based instruction scheduling → **static scheduling**; hardware-based instruction scheduling → **dynamic scheduling**.

Static scheduling. The compiler orders the instructions; the hardware executes them in that order. Contrast with dynamic scheduling, in which the hardware can execute instructions out of the compiler-specified order. How does the compiler know the latency of each instruction? What information does the compiler *not* know that makes static scheduling difficult? Anything determined at run time: variable-length operation latency, memory addresses (aliasing), branch directions. How can the compiler alleviate this, i.e. estimate the unknown? **Profiling.**

Advanced Insight: Register renaming and out-of-order execution

Anti and output dependences are dependences on *names*. An out-of-order core maps the 31 architectural registers onto a larger pool of physical registers (e.g. 128–300): every new destination gets a fresh physical register, which eliminates WAR and WAW hazards entirely and lets independent instructions execute as soon as their operands are ready (Tomasulo’s algorithm, IBM 360/91, 1967). Only true (RAW) dependences remain; forwarding becomes a broadcast of results on “common data buses” to all waiting instructions. The fourth way of handling dependences — *value prediction* — goes further and guesses the value of a load before it arrives; it has been studied extensively and appears in limited form (e.g. load-address and zero-value prediction) in some commercial designs.

Chapter summary

- Stall causes: resource contention, data and control dependences, long-latency operations.
- RAW dependences are true value dependences; WAR/WAW are name dependences, harmless in an in-order pipeline and removable by renaming.
- Hazard rule: a RAW dependence is dangerous if $\text{dist}(i, j) \leq \text{dist}(\text{ID}, \text{WB})$. Detection by scoreboard (stalls on all dependences) or comparators (RAW only).
- Stall = freeze PC and IF/ID, inject bubbles into ID/EX; $\text{CPI} = (N + S) / N$.
- Forwarding from EX/MEM and MEM/WB removes most stalls; the load-use case still costs one bubble, which the compiler tries to hide by scheduling.

Exercises

E4.1 Identify all RAW, WAR and WAW dependences: `add x1,x2,x3; sub x4,x1,x5; and x1,x6,x7; or x8,x1,x4; sw x8,0(x1)`.

Answer: RAW: `add`→`sub` (x1), `sub`→`or` (x4), and→`or` (x1), and→`sw` (x1), or→`sw` (x8). WAR: `sub` reads x1, and writes x1. WAW: `add` and `and` both write x1.

E4.2 Draw the pipeline diagram for the sequence above (a) without forwarding (write-before-read RF) and (b) with full forwarding. Count cycles.

Answer: (a) sub waits 2 cycles for x1, or waits 2 for x1 (from and), sw waits 2 for x8: 15 cycles (6 stalls); (b) 9 cycles, no stalls.

E4.3 Write the forwarding-unit equations for the second ALU operand and for the store-data input (rs2 for sw in MEM).

E4.4 A program has 30% loads; 50% of them have a dependent instruction immediately after, of which the compiler can reschedule 60%. Compute CPI.

Answer: $1 + 0.3 \times 0.5 \times 0.4 = 1.06$.

E4.5 Why must the forwarding logic check $rd \neq 0$? Give a failing example.

Answer: `addi x0, x1, 5` followed by `add x2, x0, x0` would forward 5 into the add instead of 0.

E4.6 With a scoreboard, how many stall cycles does `lw x1, 0(x2); lw x1, 4(x3)` incur? Why would a comparator-based design incur none?

Answer: The second load stalls in ID until the first writes back and sets the valid bit (WAW on x1): 2 cycles with write-before-read; comparator logic only checks sources (RAW), so it sees no hazard.

5

Control Dependences

Every instruction depends on the one before it for one value: the program counter. For sequential instructions the next PC is trivially $PC+4$, but for branches and jumps it is known only after the instruction has been decoded and (for conditional branches) its condition evaluated — several cycles after the pipeline needed it to fetch the next instruction. This chapter quantifies the cost of control dependences and surveys the techniques that reduce it: stalling, predicting $PC+4$ and flushing, delayed branches, predication, and fine-grained multithreading. Branch prediction proper is the subject of Chapter 6.

5.1 Control dependence as a special data dependence

- Control dependence is a data dependence on the instruction pointer / program counter.
- **Question:** what should the fetch PC be in the next cycle?
- **Answer:** the address of the next instruction. All instructions are control dependent on previous ones. Why? Because each could, in principle, change the PC.
- If the fetched instruction is a non-control-flow instruction, the next fetch PC is the address of the next sequential instruction — easy to determine if we know the size of the fetched instruction.
- If the instruction that is fetched is a control-flow instruction, how do we determine the next fetch PC?

Keeping the pipeline full with the correct sequence of dynamic instructions is critical. Potential solutions if the instruction is a control-flow instruction:

1. **Stall** the pipeline until we know the next fetch address;
2. **Guess** the next fetch address (*branch prediction*);
3. Employ **delayed branching** (branch delay slot);
4. **Do something else** (fine-grained multithreading);
5. **Eliminate** control-flow instructions (predicated execution);
6. Fetch from **both** possible paths, if you know the addresses of both (multipath execution).

5.2 Stalling on branches

In the five-stage pipeline the branch condition and target are known at the end of EX. If the pipeline stalls fetch until then, every branch costs 2 cycles (fetch resumes in the cycle after EX).

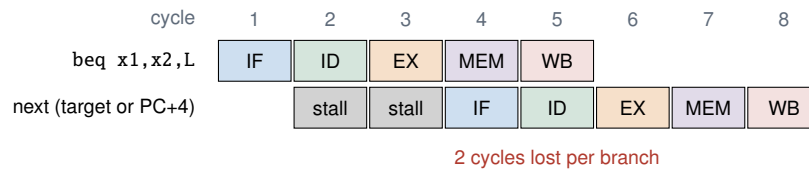


Figure 5.1: Stalling fetch until the branch resolves in EX.

Worked Example 5.1: CPI with branch stalls

20% of instructions are control-flow instructions; each costs 2 stall cycles: $CPI = 1 + 0.20 \times 2 = 1.40$. In a 15-stage pipeline that resolves branches in stage 10, the penalty is 9 cycles: $CPI = 1 + 0.20 \times 9 = 2.8$. Stalling is not an option for deep pipelines.

5.3 Guessing NextPC = PC + 4

Always predict that the next sequential instruction is the next instruction to be executed. This is a form of next-fetch-address prediction (and branch prediction). When the branch resolves:

- if not taken, the guess was right: no penalty;
- if taken, the instructions fetched since the branch (“wrong-path” instructions) must be **flushed** and the branch target fetched.

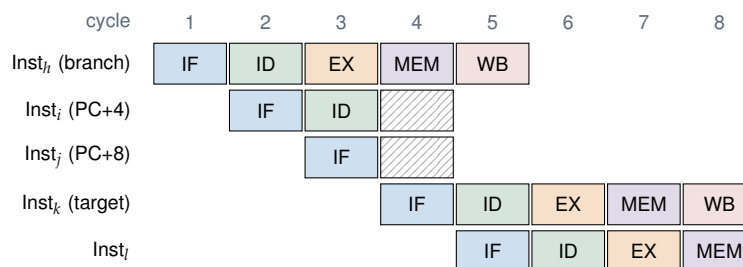


Figure 5.2: Pipeline flush on a misprediction: when Inst_i resolves as taken in EX (cycle 3), the two wrong-path instructions are killed (hatched cells: converted to bubbles) and the target is fetched in cycle 4.

Worked Example 5.2: Performance of predict-not-taken (the lecture’s analysis)

Assume: correct guess $\Rightarrow$ no penalty ($\sim 86\%$ of the time); incorrect guess $\Rightarrow$ 2 bubbles; no data-dependence stalls; 20% control-flow instructions; 70% of control-flow instructions are taken.

$$CPI = [1 + (0.20 \times 0.7) \times 2] = [1 + 0.14 \times 2] = 1.28.$$

Here 0.14 is the probability of a wrong guess and 2 is the penalty for a wrong guess.

Making PC+4 more effective. Maximize the chance that the next sequential instruction is the next instruction to be executed:

- **Software:** lay out the control-flow graph such that the “likely next instruction” is on the not-taken path of a branch — *profile-guided code positioning* (Pettis & Hansen, PLDI 1990).
- **Hardware:** cache traces of executed instructions so that the dynamically following instruc-

tions are stored contiguously — the *trace cache*.

Worked Example 5.3: Resolving branches in ID

Moving the comparison and target addition into ID (an equality comparator after the register read, a dedicated adder) reduces the misprediction penalty to 1 cycle: $\text{CPI} = 1 + 0.14 \times 1 = 1.14$. But the comparator now needs operands that may still be in EX or MEM: an ALU result needed by a branch immediately following it now costs 1 stall, and a load feeding a branch costs 2. With 30% of branches depending on the preceding ALU instruction: $\text{CPI} \approx 1.14 + 0.2 \times 0.3 \times 1 = 1.20$ — still better than 1.28, and the comparator path may lengthen the ID stage. Classic MIPS resolved branches in ID for this reason; most modern deep pipelines resolve them late and rely on prediction instead.

5.4 Predicate combining and predicated execution

5.4.1 Predicate combining (not predicated execution)

Complex predicates are converted into multiple branches: `if ((a == b) && (c < d) && (a > 5000)) { ... }` becomes 3 conditional branches. Problem: this increases the number of control dependences. **Idea:** combine predicate operations to feed a *single* branch instruction instead of having one branch for each. Predicates are stored and operated on using *condition registers*; a single branch checks the value of the combined predicate.

- + Fewer branches in code $\Rightarrow$ fewer mispredictions/stalls.
- Possibly unnecessary work: if the first predicate is false, there was no need to compute the other predicates.

(IBM POWER has eight 4-bit condition-register fields and CR logical instructions `crand`, `cror` for exactly this purpose. In RISC-V the same effect is obtained with `slt/sltu/xor/seqz` into general registers and `and/or`, followed by one branch.)

Worked Example 5.4: One branch instead of three in RISC-V

```
# a0=a, a1=b, a2=c, a3=d ; if (a==b && c<d && a>5000) goto L
xor    t0, a0, a1      # t0 = 0 iff a==b
seqz   t0, t0           # t0 = (a==b)
slt     t1, a2, a3      # t1 = (c<d)
li      t2, 5000
slt     t2, t2, a0       # t2 = (5000<a)
and     t0, t0, t1
and     t0, t0, t2
bnez    t0, L           # single branch
```

8 instructions and 1 branch versus about 5 instructions and 3 branches. If each branch mispredicts 10% of the time at a 10-cycle penalty, the three-branch version loses on average $\approx 3 \times 0.1 \times 10 = 3$ cycles; the combined version $0.1 \times 10 = 1$ cycle plus 3 extra instructions. Combining wins when branches are hard to predict.

5.4.2 Predicated execution

Idea: convert control dependence into data dependence. Simple example: suppose we had a conditional move instruction `CMOV condition, R1 ← R2`: $R1 = (\text{condition} == \text{true}) ? R2 : R1$. Employed in most modern ISAs (x86 `cmov`, Alpha, ARM conditional execution, RISC-V Zicn0d's `czero.eqz/czero.nez`).

With branches	With CMOVs
<code>if (a == 5) {b = 4;} else {b = 3;}</code>	<code>CMPEQ condition, a, 5;</code> <code>CMOV condition, b ← 4;</code> <code>CMOV !condition, b ← 3;</code>

- **Eliminates branches** $\Rightarrow$ enables straight-line code (i.e. larger basic blocks in code).
- **Advantages:** always-not-taken prediction works better (no branches); the compiler has more freedom to optimize code (no branches): control flow does not hinder instruction reordering optimizations; code optimizations are hindered only by data dependences.
- **Disadvantages:** useless work — some instructions are fetched and executed but discarded (especially bad for easy-to-predict branches); requires additional ISA support.

Worked Example 5.5: When does predication pay?

An if-then-else has 4 instructions on each path. **Branching version:** 1 branch + 4 instructions executed; misprediction rate m , penalty $P = 10$ cycles. Expected cycles = $5 + 10m$. **Predicated version:** both paths executed, 8 instructions + 1 compare = 9 cycles, never mispredicted. Break-even: $5 + 10m = 9 \Rightarrow m = 0.4$. Predication wins only if the branch mispredicts more than 40% of the time — i.e. for genuinely data-dependent, unpredictable branches (sorting, compression, search on random data). For a 2-instruction `max(a, b)` the predicated version is 2–3 instructions and wins at much lower m .

5.5 Delayed branching

Change the semantics of a branch instruction: *branch after N instructions* (or after N cycles). **Idea:** delay the execution of a branch; the N instructions (*delay slots*) that come after the branch are **always** executed, regardless of the branch direction.

Problem: how do you find instructions to fill the delay slots? The branch must be independent of the delay-slot instructions. Unconditional branches: easier to find instructions to fill the delay slot. Conditional branches: the condition computation should not depend on instructions in the delay slots $\Rightarrow$ difficult to fill.

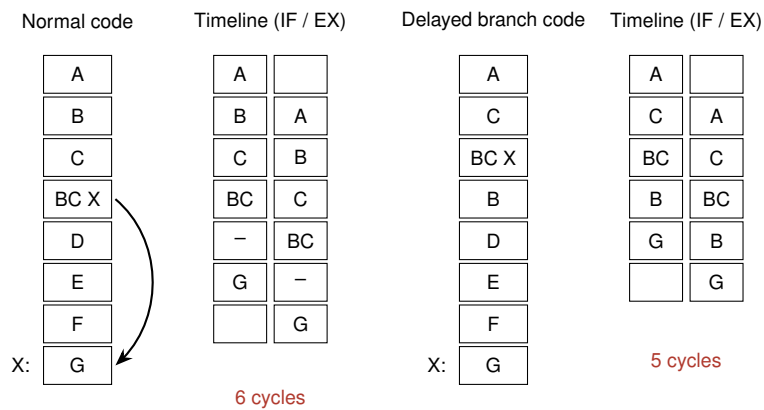


Figure 5.3: Delayed branching in a 2-stage (IF, EX) pipeline with one delay slot: instruction B, which does not affect the branch condition, is moved into the slot after BC X, so the slot does useful work instead of a bubble.

5.5.1 Filling the delay slot

Strategy	Requirement	When useful
(a) From before the branch	independent of the branch condition	always safe (within same basic block)
(b) From the target	must be safe to execute on the fall-through path	when the branch is mostly taken
(c) From the fall-through	must be safe to execute on the taken path	when mostly not taken

Reordering data-independent (RAW, WAW, WAR) instructions does not change program semantics. For (b) and (c), correctness may require adding a new instruction on the other path (or the slot instruction must be harmless there, e.g. writing a dead register).

Delayed branch with squashing. In SPARC, if the branch falls through (i.e. it is not taken), the delay-slot instruction is *not* executed (annulled). Why could this help? It allows the slot to be filled from the *target* path (strategy b) without requiring the instruction to be harmless on the fall-through path.

Worked Example 5.6: Value of a delay slot

A compiler fills 65% of delay slots with useful instructions (strategy a), and the rest with nop. One delay slot, branch resolved after 1 cycle, 20% branches. Without delay slots (stall): $\text{CPI} = 1 + 0.2 \times 1 = 1.2$. With delay slots: the slot cycle is wasted only when filled with a nop: $\text{CPI} = 1 + 0.2 \times 0.35 = 1.07$ (counting nops as wasted cycles).

Advantages and disadvantages.

- + Keeps the pipeline full with useful instructions in a simple way, assuming (1) the number of delay slots equals the number of instructions needed to keep the pipeline full before the branch resolves, and (2) all delay slots can be filled with useful instructions.
- Not easy to fill the delay slots (even with a 2-stage pipeline): (1) the number of delay slots increases with pipeline depth and superscalar execution width; (2) the number of delay

slots should be variable with variable-latency operations. Why?

- **Ties ISA semantics to hardware implementation.** SPARC, MIPS and HP-PA have 1 delay slot. What if the pipeline implementation changes with the next design? The delay slot stays forever — one reason RISC-V has no delay slots.

5.6 Fine-grained multithreading

Idea: the hardware has multiple thread contexts (PC + registers). Each cycle, the fetch engine fetches from a different thread. By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread: branch/instruction-resolution latency is overlapped with the execution of other threads' instructions.

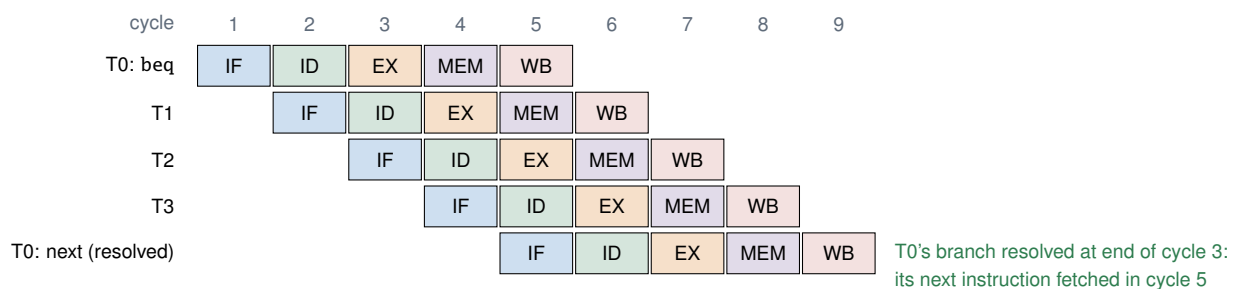


Figure 5.4: Fine-grained multithreading with 4 threads in a 5-stage pipeline: consecutive instructions in the pipeline belong to different threads, so there are no intra-thread data or control hazards.

- + No logic needed for handling control and data dependences within a thread: no dependence checking between instructions (only one instruction in the pipeline from a single thread), no branch prediction logic.
- + Otherwise-bubble cycles are used for executing useful instructions from different threads; improved system throughput, latency tolerance and utilization.
- Single-thread performance suffers: one instruction fetched every N cycles from the same thread.
- Extra logic for keeping thread contexts (multiple PCs, register files), thread-selection logic.
- Does not overlap latency if there are not enough threads to cover the whole pipeline.
- Resource contention between threads in caches and memory; some dependence-checking logic *between* threads remains (loads/stores to the same address).

History. The CDC 6600's peripheral processing unit (Thornton, "Parallel Operation in the Control Data 6600", AFIPS 1964) was fine-grained multithreaded: the processor executed a different I/O thread every cycle, and an operation from the same thread was executed every 10 cycles. The Denelcor HEP (Heterogeneous Element Processor; Smith, "A pipelined, shared resource MIMD computer", ICPP 1978) supported 120 threads per processor with available/unavailable (waiting) queues; each thread could have only one instruction in the pipeline, so to each thread the processor looked like a non-pipelined machine. The trade-off is system throughput versus single-thread performance. Modern examples: Sun/Oracle UltraSPARC T1 "Niagara" (4 threads per core, thread-select mux before decode), and GPUs, which interleave warps of threads to hide pipeline and memory latency.

Worked Example 5.7: Throughput and latency with fine-grained multithreading

A 5-stage pipeline with branch penalty 2 and load-use penalty 1. Single-thread CPI (with predict-not-taken) = $1.28 + 0.25 \times 0.4 = 1.38$.

With 4 threads interleaved round-robin, no intra-thread hazards exist: the pipeline completes one instruction per cycle (aggregate CPI = 1.0, i.e. IPC = 1.0), a 38% throughput gain. But each thread issues only every 4th cycle: a thread's program runs at CPI = 4.0, i.e. $4.0/1.38 = 2.9\times$ slower than if it had the pipeline alone. If one thread stalls on a 100-cycle cache miss, the others continue — this latency tolerance is the main reason GPUs use the technique.

Chapter summary

- Control dependence is a dependence on the PC; the five-stage pipeline learns the next PC only in EX.
- Stalling costs the full resolution latency per branch; predict-PC+4 costs it only on taken branches (CPI = $1 + 0.2 \cdot 0.7 \cdot 2 = 1.28$).
- Predicate combining reduces the number of branches; predication (CMOV) removes them at the cost of executing both paths.
- Delayed branches fill the resolution latency with always-executed instructions but tie the ISA to one pipeline.
- Fine-grained multithreading hides dependences entirely by interleaving threads, trading single-thread performance for throughput.

Exercises

E5.1 Compute CPI for predict-not-taken with 25% branches, 60% taken, 3-cycle penalty.

Answer: $1 + 0.25 \times 0.6 \times 3 = 1.45$.

E5.2 With the same branches, what CPI does predict-taken achieve if the target is known in ID (1-cycle bubble when correctly predicted taken) and resolution in EX?

Answer: Taken: 1 bubble; not-taken mispredict: 3: $1 + 0.25(0.6 \cdot 1 + 0.4 \cdot 3) = 1.45$ — no better here.

E5.3 Rewrite `max = (a > b) ? a : b` with branch-free RISC-V code using `slt`, `sub`, and `/neg`.

Answer: `slt t0,a,b; neg t0,t0; sub t1,b,a; and t1,t1,t0; add max,a,t1.`

E5.4 Fill the delay slot in: `add x1,x2,x3; beq x4,x5,L; [slot]` for each strategy and state the safety condition.

E5.5 With 8 threads on a 5-stage pipeline, what fraction of each thread's issue slots is used? What happens to throughput if 3 threads are blocked on cache misses?

Answer: $1/8$; 5 ready threads still cover the 5-stage pipeline: throughput stays at 1 IPC.

6

Branch Prediction

Predicting PC+4 is right only when branches are not taken. A **branch predictor** guesses, at fetch time and before the instruction is even decoded, whether the fetched instruction is a branch, which direction it will go, and where it will go. With good predictors, modern processors keep pipelines of 15–20 stages and 4–8 instructions wide busy with correct-path work more than 95% of the time. This chapter builds the predictors from the lecture — BTB, static schemes, last-time and two-bit-counter predictors — quantifies their benefit, and looks at the correlating predictors used in real cores.

6.1 Guessing the next instruction to fetch

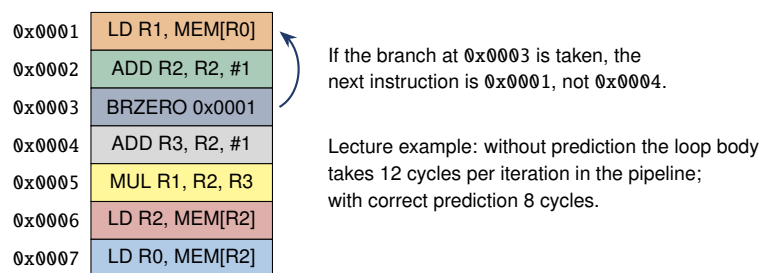


Figure 6.1: The next fetch address after a branch is unknown at fetch time.

Misprediction penalty. When a branch resolves, if the prediction was wrong, the branch target (Inst_k) is fetched and all instructions fetched since the branch (the “wrong-path” instructions) must be flushed. The penalty equals the number of cycles between fetch and resolution.

6.2 What must be predicted?

Requires three things to be predicted *at fetch stage*:

1. whether the fetched instruction is a branch;
2. the (conditional) branch direction;
3. the branch target address (if taken).

Observation: the target address remains the same for a conditional direct branch across dynamic instances. **Idea:** store the target address from the previous instance and access it with the PC: the **Branch Target Buffer (BTB)** or Branch Target Address Cache.

- The target (3) is provided by the BTB: it remembers the target address computed the last time the branch was executed.

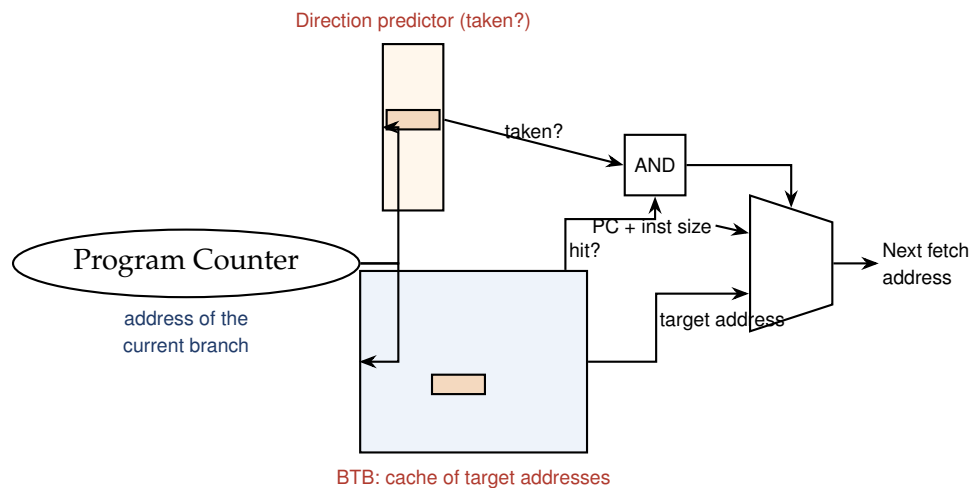


Figure 6.2: BTB and direction prediction. The BTB answers “is this a branch, and where does it go?”; the direction predictor answers “will it be taken?”.

- Whether the instruction is a branch (1) is also answered by the BTB: if the BTB provides a target address for the program counter, then it must be a branch. Alternatively, “branch metadata” bits can be stored in the instruction cache/memory (partially decoded instructions).
- The direction (2) is the hard part: how do we predict it?

Worked Example 6.1: Always-taken with a BTB (the lecture's numbers)

With a BTB, “always taken” is possible at fetch: 20% branches, 70% taken; only the 30% not-taken branches mispredict, with 2 bubbles each:

$$\text{CPI} = 1 + (0.20 \times 0.3) \times 2 = 1.12.$$

(Assuming the BTB always hits for taken branches. A BTB miss on a taken branch costs the full resolution penalty.)

Worked Example 6.2: BTB storage

A 512-entry, direct-mapped BTB for a 32-bit RISC-V core: index with PC[10:2] (9 bits), tag PC[31:11] (21 bits), target (30 bits; the 2 low bits are known), valid bit: 52 bits/entry, $512 \times 52 = 26\,624$ bits = 3.25 KiB. Storing a partial tag (e.g. 8 bits) saves 13 bits/entry (25%) at the cost of occasional false hits, which merely cause a misprediction. A 4096-entry table of 2-bit direction counters costs only 1 KiB.

6.3 Static branch prediction

Static predictors make the same prediction every time a given branch executes; the decision is made at compile time or fixed in hardware.

Always not-taken. Simple to implement: no need for a BTB, no direction prediction. Low accuracy: ~30–40% (for conditional branches). The compiler can lay out code such that the likely path is the “not-taken” path.

Always taken. No direction prediction; better accuracy: ~60–70%. Backward branches (i.e. loop branches) are usually taken; a backward branch's target address is lower than the branch PC.

Backward taken, forward not taken (BTFN). Predict backward (loop) branches as taken, others not-taken. Combines both observations.

Profile-based. **Idea:** the compiler determines the likely direction for each branch using a profile run, and encodes that direction as a hint bit in the branch instruction format.

- + Per-branch prediction (more accurate than the schemes above) → accurate if the profile is representative.
- Requires hint bits in the branch instruction format.
- Accuracy depends on the dynamic branch behaviour: TTTTTTTTTTNNNNNNNNNN → 50% accuracy; TNTNTNTNTNTNTNTNTNTN → 50% accuracy.
- Accuracy depends on the representativeness of the profile input set.

Program-based (program analysis). Use heuristics based on program analysis to determine the statically predicted direction. Example opcode heuristic: predict BLEZ as NT (negative integers used as error values in many programs). Example loop heuristic: predict a branch guarding a loop execution as taken (i.e. execute the loop). Pointer and FP comparisons: predict not equal. Ball and Larus, “Branch prediction for free,” PLDI 1993.

- + Does not require profiling.
- Heuristics might not be representative or good; requires compiler analysis and ISA support.

Programmer hints (pragmas). **Idea:** the programmer provides the statically predicted direction via *pragmas* or annotations in the programming language. Examples: the Linux kernel's `likely()/unlikely()` macros, implemented with GCC's `__builtin_expect`, and the C++20 attributes `[[likely]]` and `[[unlikely]]`:

```
if (likely(x)) { ... }
if (unlikely(error)) { ... }
```

Used by Linux and modern compilers.

- + Does not require profiling or program analysis; the programmer may know some branches and their program better than other analysis techniques.
- Requires programming-language, compiler and ISA support; burdens the programmer.

Limitation of all static schemes

All static branch-prediction schemes (compile-time techniques) cannot adapt to dynamic changes in branch behaviour. This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its own overheads).

Worked Example 6.3: Static schemes on a small program

A program's branches: loop-closing backward branches (40% of dynamic branches, 90% taken), forward error checks (35%, 5% taken), forward if-then-else (25%, 50% taken).

Scheme	Loop	Error	If-else	Overall accuracy
Always NT	10%	95%	50%	$0.04 + 0.3325 + 0.125 = 49.8\%$
Always T	90%	5%	50%	$0.36 + 0.0175 + 0.125 = 50.3\%$
BTFN	90%	95%	50%	$0.36 + 0.3325 + 0.125 = 81.8\%$

With 20% branches and a 2-cycle penalty: $\text{CPI}_{\text{BTFN}} = 1 + 0.2 \times 0.182 \times 2 = 1.073$.

6.4 Dynamic branch prediction

Idea: predict branches based on dynamic information collected at run time.

- + Prediction based on the history of the execution of branches; it can adapt to dynamic changes in branch behaviour; no need for static profiling (input-set representativeness problem goes away).
- More complex (requires additional hardware).

6.4.1 The last-time predictor

Last-time predictor: a single bit per branch (stored in the BTB) indicates which direction the branch went the last time it executed. TTTTTTTTTTNNNNNNNNNNNN $\rightarrow$ 90% accuracy. Always mispredicts the last iteration and the first iteration of a loop branch: accuracy for a loop with N iterations $= (N - 2) / N$.

- + Loop branches for loops with a large N (number of iterations).
- Loop branches for loops with a small N ; TNTNTNTNTNTNTNTNTNTN $\rightarrow$ 0% accuracy.

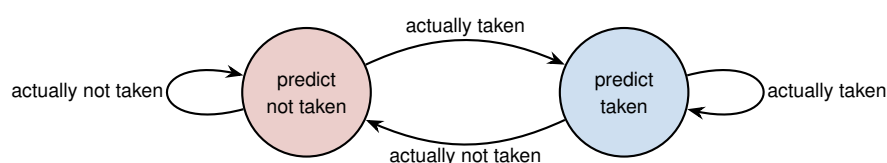


Figure 6.3: State machine of the last-time (1-bit) predictor.

Implementation. A table of 1-bit entries (the *branch history table*, BHT, or the BTB itself) indexed by low PC bits; the bit is read at fetch to predict, and written with the actual outcome when the branch resolves. Different branches that map to the same entry *alias* and may disturb each other's predictions.

6.4.2 Improving the last-time predictor: the two-bit counter

Problem: the last-time predictor changes its prediction from $T \rightarrow NT$ or $NT \rightarrow T$ too quickly, even though the branch may be mostly taken or mostly not taken. Solution idea: add **hysteresis** to the predictor so that the prediction does not change on a single different outcome. Use two bits to

track the history of predictions for a branch instead of a single bit: a **saturating two-bit counter** per branch (Smith, 1981).

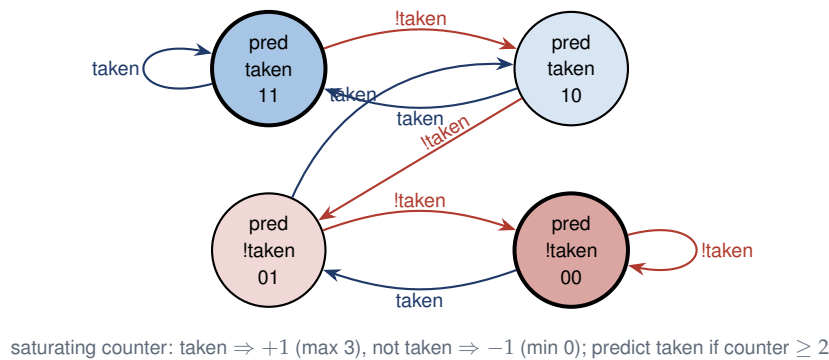


Figure 6.4: Two-bit saturating counter predictor. States 11 and 00 are “strong”, 10 and 01 “weak”; the prediction changes only after two consecutive mispredictions from a strong state.

Each branch is associated with a two-bit counter; one more bit provides hysteresis; a strong prediction does not change with one single different outcome.

- Accuracy for a loop with N iterations = $(N - 1)/N$.
- TNTNTNTNTNTNTNTNTN $\rightarrow$ 50% accuracy (assuming the counter is initialized to weakly taken).
- + Better prediction accuracy.
- More hardware cost (but the counter can be part of a BTB entry).

Worked Example 6.4: Tracing 1-bit and 2-bit predictors

A branch inside a loop that runs 4 iterations, executed twice: outcomes T T T N T T T N. Both predictors start predicting taken (1-bit: T; 2-bit: state 10).

Outcome	T	T	T	N	T	T	T	N	Correct
1-bit prediction	T	T	T	T	N	T	T	T	
1-bit result	✓	✓	✓	×	×	✓	✓	×	5/8
2-bit state before	10	11	11	11	10	11	11	11	
2-bit prediction	T	T	T	T	T	T	T	T	
2-bit result	✓	✓	✓	×	✓	✓	✓	×	6/8

In steady state the 1-bit predictor mispredicts twice per loop execution (exit and re-entry): $(N - 2)/N = 50\%$ for $N = 4$; the 2-bit counter only once: $(N - 1)/N = 75\%$.

The alternating pattern TNTN...: from state 10 (weakly taken) the 2-bit counter goes 10 $\rightarrow$ 11 (T, correct), 11 $\rightarrow$ 10 (N, wrong), 10 $\rightarrow$ 11 (T, correct)...: 50%. From state 01 (weakly not taken) it goes 01 $\rightarrow$ 10 (T, wrong), 10 $\rightarrow$ 01 (N, wrong)...: 0%. Initialization matters, and alternating patterns defeat all per-branch counters — motivating history-based predictors (Section 6.6).

Worked Example 6.5: CPI with a two-bit predictor (the lecture’s numbers)

~85–90% accuracy for many programs with 2-bit counter-based prediction (also called *bimodal* prediction). With 90% accuracy:

$$\text{CPI}_{2\text{BC}} = [1 + (0.20 \times 0.10) \times 2] = 1.04.$$

Compared with predict-not-taken (1.28) that is a $1.23\times$ speedup from about 1 KiB of counters.

6.5 The branch problem in wide, deep pipelines

Control-flow instructions (branches) are frequent: 15–25% of all instructions. Problem: the next fetch address after a control-flow instruction is not determined until N cycles later in a pipelined processor (N = minimum branch-resolution latency). If we are fetching W instructions per cycle (i.e. the pipeline is W wide), a branch misprediction leads to $N \times W$ wasted instruction slots.

Worked Example 6.6: Importance of the branch problem (the lecture's numbers)

Assume $N = 20$ (20 pipe stages), $W = 5$ (5-wide fetch); 1 out of 5 instructions is a branch; each 5-instruction block ends with a branch. How long does it take to fetch 500 instructions?

- 100% accuracy: 100 cycles (all instructions fetched on the correct path); no wasted work.
- 99% accuracy: 100 (correct path) + 20 (wrong path) = 120 cycles; 20% extra instructions fetched.
- 98% accuracy: $100 + 20 \times 2 = 140$ cycles; 40% extra instructions fetched.
- 95% accuracy: $100 + 20 \times 5 = 200$ cycles; 100% extra instructions fetched.

(100 branches; each misprediction costs 20 cycles of wrong-path fetch.) Going from 95% to 99% accuracy speeds up fetch by $200/120 = 1.67\times$. In a big core, branch prediction accuracy is worth more than almost any other single feature.

6.6 Correlating (two-level) predictors

Last-time and 2-bit counter predictors exploit “last-time” predictability. Two further realizations:

- **Realization 1:** a branch’s outcome can be correlated with other branches’ outcomes — *global branch correlation*.
- **Realization 2:** a branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last time” it was executed) — *local branch correlation*.

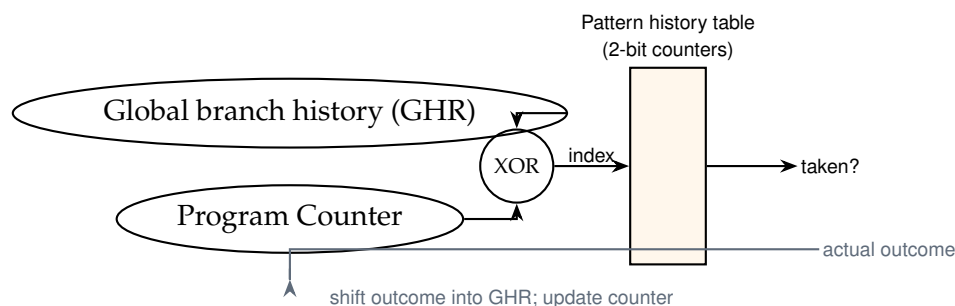


Figure 6.5: The *gshare* predictor (McFarling, 1993): the global history of the last h branch outcomes is XORed with the PC to index a table of 2-bit counters.

Worked Example 6.7: gshare learns the alternating pattern

A single branch alternates T, N, T, N, ...; use a 2-bit global history (no other branches). After warm-up, the most recent outcome in the history before every T is N, and before every N it is T, so the two kinds of outcomes always see *different* histories and therefore index *different* counters: one counter is trained only by T outcomes, the other only by N outcomes. After at most two training visits each, both counters are in the correct strong state and the branch is predicted **100%** correctly — versus 50% (or 0%) for the bimodal predictor.

Similarly, in `if (x>0) a=1; if (x>0) b=2;` the second branch is perfectly correlated with the first; a global history captures this, a per-branch counter cannot.

Advanced Insight: State-of-the-art predictors

Tournament predictors (Alpha 21264, 1998) run a local-history and a global-history predictor side by side and use a third table of counters to choose between them per branch. **Perceptron** predictors (Jiménez & Lin, 2001; used in AMD Zen cores) compute the prediction as the sign of a weighted sum of history bits, allowing very long histories with linear hardware cost. **TAGE** (Seznec & Michaud, 2006) uses several tables indexed with geometrically increasing history lengths (e.g. 5, 9, 15, 25, 44, 76, 130 bits) and partial tags, predicting with the longest matching history; TAGE-derived designs win most branch-prediction championships and appear in many high-end cores. Typical accuracies on SPEC-like code: bimodal $\approx 90\%$, gshare $\approx 93\text{--}95\%$, TAGE $\approx 97\text{--}98\%$. Returns are predicted by a **return address stack** (push on `jal ra`, pop on `ret`), and indirect jumps by history-indexed target caches (ITTAGE).

Chapter summary

- Three things to predict at fetch: is it a branch, its direction, its target. The BTB provides (1) and (3).
- Static schemes: always NT (30–40%), always T (60–70%), BTFN, profile hints, program heuristics, pragmas — none adapts at run time.
- Dynamic: last-time (1 bit, $(N - 2)/N$ on loops) and 2-bit saturating counters (hysteresis, $(N - 1)/N$, 85–90% overall).
- Misprediction cost scales with depth $\times$ width; 95% $\rightarrow$ 99% accuracy can speed up a 20-stage, 5-wide machine by $1.67\times$.
- Correlating predictors (gshare, tournament, perceptron, TAGE) exploit global and local history to reach 95–98%.

Exercises

E6.1 A loop with 10 iterations is executed 100 times. Compute the accuracy of the loop branch under always-taken, last-time and 2-bit prediction.

Answer: 90%; 80% (after the first entry); 90%.

E6.2 Compute CPI for 22% branches, 3-cycle penalty, with (a) predict NT and 65% taken; (b) a 2-bit predictor with 92% accuracy and a BTB that always hits.

Answer: (a) $1 + 0.22 \cdot 0.65 \cdot 3 = 1.43$; (b) $1 + 0.22 \cdot 0.08 \cdot 3 = 1.05$.

E6.3 Trace a 2-bit counter starting at 00 on the pattern NNTNNTNNT. Accuracy?

Answer: Predictions all N; correct on N outcomes: $6/9 = 67\%$.

E6.4 Design a gshare predictor with a 12-bit history and 4096 counters. What is its storage? What happens if two hot branches alias?

Answer: 8 Kbit = 1 KiB plus 12-bit GHR; aliasing causes destructive interference, reducing accuracy.

E6.5 For a 15-stage, 4-wide machine with 20% branches, how much faster is 97% accuracy than 93%? (Model: fetch time = useful cycles + mispredictions $\times$ 15.)

Answer: Per 400 instructions: 100 cycles + 80 branches $\times$ (0.03 or 0.07) $\times$ 15 = 136 vs 184: 1.35 $\times$.

Exceptions, Multicycle Operations and Beyond

The five-stage pipeline of the previous chapters handles single-cycle integer operations and assumes nothing unusual happens. This final chapter addresses the remaining items on the list of pipeline-design issues — long-latency operations and exceptions — then puts everything together in a complete performance model (a “CPI stack”), runs the factorial program of Unit 2 on the pipeline, gives the RTL of the hazard and forwarding units, and previews superscalar and out-of-order execution.

7.1 Long-latency (multicycle) operations

Floating-point operations, integer multiply and divide, and cache misses take more than one cycle in EX or MEM. A common organization gives each long operation its own functional unit, pipelined where possible:

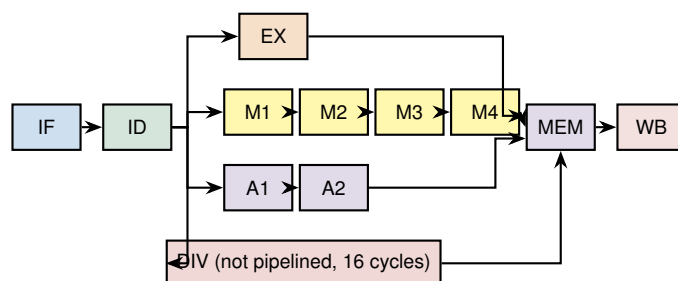


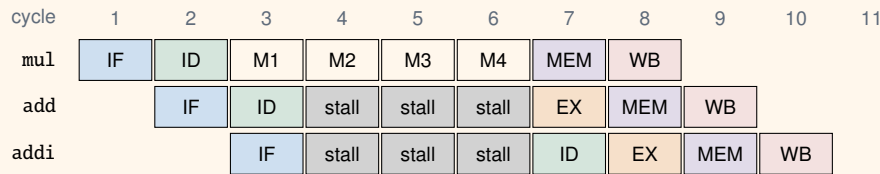
Figure 7.1: Pipeline with multiple functional units of different latencies: integer EX (1 cycle), FP/integer multiply (4, pipelined), FP add (2, pipelined), divide (16, iterative).

New problems appear:

- **Structural hazards:** the divider is not pipelined (a second divide must wait); several instructions may reach MEM/WB in the same cycle and compete for the register-file write port.
- **RAW hazards last longer:** a consumer of a 4-cycle multiply must wait up to 3 extra cycles even with forwarding.
- **WAW hazards reappear:** instructions now complete *out of order*. A `fmul f1, ...` followed by `fadd f1, ...` could write `f1` in the wrong order.
- **Precise exceptions become difficult:** a later, short instruction may already have written its result when an earlier, long instruction raises an exception.

Worked Example 7.1: Timing a sequence with a 4-cycle multiplier

```
mul  x5, x6, x7      # 4-cycle EX (M1-M4)
add  x8, x5, x9      # needs x5
addi x10, x11, 1     # independent
```



With forwarding from the end of M4 to EX, add waits in ID for 3 cycles. In an in-order pipeline, addi is stuck behind it even though it is independent: 3 cycles lost. *Out-of-order* issue would let addi proceed — the motivation for dynamic scheduling (Section 7.6.2).

7.2 Exceptions and interrupts in a pipeline

Definition 7.1: Precise exception

An exception (or interrupt) is **precise** if, when the handler starts, (1) all instructions before the excepting instruction have completed and updated the architectural state, (2) no instruction after it has changed the architectural state, and (3) mepc points to the excepting instruction (or, for interrupts, the first instruction not executed).

Precise exceptions make it possible to restart the program after the handler (page faults, emulation of missing instructions, debugging). In a pipeline, exceptions are detected in different stages:

Stage	Exceptions detected
IF	instruction-address misaligned, instruction page fault / access fault
ID	illegal instruction, ecall, ebreak
EX	(none in RV32I; arithmetic overflow in MIPS; FP exceptions set flags only)
MEM	load/store misaligned, load/store page fault / access fault

Several instructions can raise exceptions in the same cycle, and a younger instruction (in an earlier stage) may detect its exception *before* an older one. The standard solution:

1. When a stage detects an exception, it does not trap immediately; it records the cause in the instruction's pipeline register (an "exception" bit plus cause and PC) and turns the instruction into a no-op for state-changing purposes (no register or memory write).
2. The exception is taken when the instruction reaches a designated **commit point** (e.g. the end of MEM, before WB): there, older instructions have all completed. All younger instructions in the pipeline are flushed, mepc and mcause are written, and fetch is redirected to mtvec.
3. Because only the oldest excepting instruction reaches the commit point first, exceptions are taken in program order — precise.

Worked Example 7.2: Two exceptions in flight

Cycle t : a `lw` with a misaligned address is in MEM (older); an illegal instruction is in ID (younger). The illegal instruction was flagged one cycle earlier, but it is younger, so it must not trap first. At the commit point the `lw` traps: `mepc` = PC of `lw`, `mcause` = 4; the illegal instruction is flushed along with everything else. After the handler fixes the misaligned load and returns, the illegal instruction is fetched again and traps then (cause 2). Handling them in pipeline order is what makes this work.

Advanced Insight: Precise exceptions with out-of-order completion

When instructions complete out of order (multicycle units, out-of-order execution), designs keep results in a **reorder buffer** (ROB) and write the architectural state only at *retirement*, in program order; exceptions are taken at retirement. Alternatives are a **history buffer** (keep old values to undo younger writes) and a **future file**. Every modern out-of-order core uses a ROB; it is also the mechanism that recovers from branch mispredictions.

7.3 Putting it together: a CPI stack

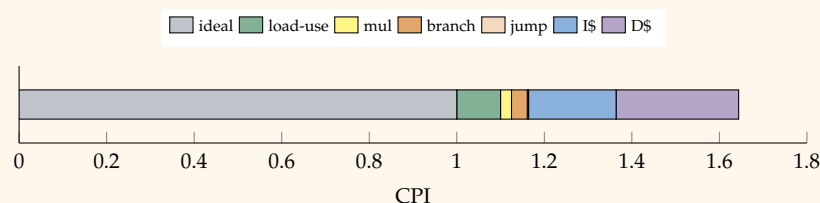
Performance of a real pipeline is the ideal CPI plus contributions from each type of stall:

$$\text{CPI} = \text{CPI}_{\text{ideal}} + \text{CPI}_{\text{data}} + \text{CPI}_{\text{control}} + \text{CPI}_{\text{struct}} + \text{CPI}_{\text{mem}}.$$

Worked Example 7.3: CPI stack of a five-stage RISC-V pipeline

Workload: 25% loads (40% followed by a dependent instruction the compiler could not separate), 10% stores, 18% conditional branches (2-bit predictor, 90% accurate, penalty 2), 3% jumps (BTB hit 95%, penalty 2 on miss), 2% multiplies (4-cycle, 50% followed by a dependent instruction within 1 cycle, average 2.5 stall cycles), instruction-cache misses 1% of instructions $\times$ 20 cycles, data-cache misses 4% of loads/stores $\times$ 20 cycles.

Component	Computation	CPI
Ideal		1.000
Load-use	$0.25 \times 0.40 \times 1$	0.100
Multiply	$0.02 \times 0.5 \times 2.5$	0.025
Branches	$0.18 \times 0.10 \times 2$	0.036
Jumps	$0.03 \times 0.05 \times 2$	0.003
I-cache	0.01×20	0.200
D-cache	$0.35 \times 0.04 \times 20$	0.280
Total		1.644



Memory stalls (0.48) now dominate the pipeline's own hazards (0.16) — which is why Unit 4 is about the memory hierarchy.

7.4 Factorial on three microarchitectures

Worked Example 7.4: The factorial program on the pipeline

Recall the loop from Unit 2:

```

    addi a0,x0,5 ; addi a1,x0,1 ; addi t0,x0,1
loop: blt  a0,t0,done
      mul  a1,a1,t0
      addi t0,t0,1
      jal  x0,loop
done:  sw   a1,256(x0)

```

Assumptions: full forwarding, single-cycle mul in EX, branches and jumps resolved in EX with predict-not-taken (2-cycle penalty when taken).

Dependences. blt needs t0 from addi two instructions earlier (distance 2, forwarded from MEM/WB: no stall). mul needs t0 from three instructions earlier: from the register file. No data stalls.

Control. jal is always taken: 2 bubbles per iteration. blt is not taken for 5 evaluations (correctly predicted) and taken once at exit (2 bubbles).

Cycle count. 25 instructions + 4 cycles to fill the pipeline + 5×2 (jal) + 1×2 (final blt) = 41 cycles.

Microarchitecture	Cycles	Clock (ps)	Time (ns)	CPI
Single-cycle (Unit 2)	25	960	24.0	1.00
Multicycle (Chapter 1)	104	325	33.8	4.16
5-stage pipeline, predict NT	41	300	12.3	1.64
5-stage pipeline + BTB (jal predicted)	31	300	9.3	1.24

(The pipeline clock: slowest stage 250 ps memory + 50 ps register overhead.) For this tiny program the fill time and taken branches are a large fraction; for long programs CPI approaches the steady-state value of the loop: $6/4 = 1.5$ with predict-NT and $4/4 = 1.0$ with a BTB.

7.5 RTL of the hazard and forwarding units

Listing 7.1: Forwarding and hazard-detection units for the five-stage pipeline.

```

// Forwarding: select ALU operand A (operand B is identical with rs2)
// 2'b00 = register file (ID/EX), 2'b10 = EX/MEM, 2'b01 = MEM/WB
always @* begin
    fwdA = 2'b00;
    if (regwrite_mem && rd_mem != 0 && rd_mem == rs1_ex)      fwdA = 2'b10; // youngest
    else if (regwrite_wb && rd_wb != 0 && rd_wb == rs1_ex)    fwdA = 2'b01;
end

// Load-use hazard: stall IF and ID one cycle, insert a bubble into ID/EX
wire load_use = memread_ex && rd_ex != 0 &&
                ((rd_ex == rs1_id && use_rs1_id) || (rd_ex == rs2_id && use_rs2_id));
assign pc_write    = ~load_use;    // hold PC
assign ifid_write  = ~load_use;    // hold IF/ID

```

```

assign idex_bubble = load_use;      // zero control bits in ID/EX

// Control hazard: branch/jump resolved in EX with predict-not-taken
assign flush = take_branch_ex;      // squash IF/ID and ID/EX (wrong path)
assign idex_flush = flush | idex_bubble;

```

Note the asymmetry: a load-use *stall* freezes the front of the pipeline and inserts one bubble, while a misprediction *flush* kills the two wrong-path instructions and redirects fetch. Both act by clearing the control bits of the pipeline register, never by modifying architectural state.

7.6 Beyond the scalar in-order pipeline

7.6.1 Superscalar execution

A **superscalar** pipeline fetches, decodes and executes W instructions per cycle (ideal $\text{CPI} = 1/W$, i.e. $\text{IPC} = W$). Every structure widens: W -wide fetch, $2W$ register read ports, W write ports, W ALUs, and dependence checking between instructions issued in the same cycle as well as with those in flight.

Worked Example 7.5: Dual issue

A 2-wide in-order pipeline can issue a pair only if the second instruction does not depend on the first and at most one of them is a load/store (one data-cache port). For the factorial loop body: (blt, mul) is independent $\Rightarrow$ pair; (addi, jal) independent $\Rightarrow$ pair. With a BTB, the loop runs at 2 cycles per iteration: $\text{IPC} = 2$. For typical integer code, in-order dual issue achieves $\text{IPC} \approx 1.3\text{--}1.5$ because many adjacent pairs are dependent.

7.6.2 Out-of-order execution

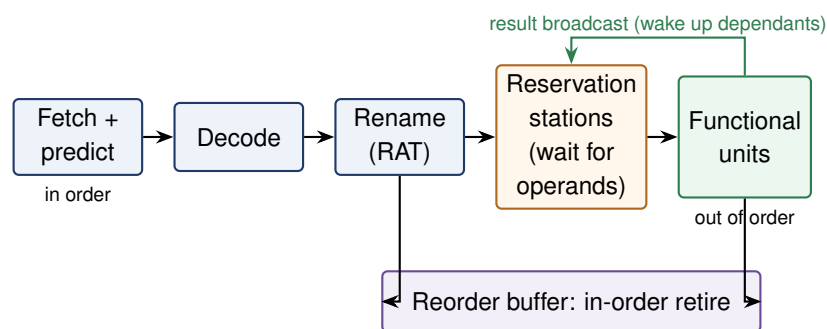


Figure 7.2: Structure of an out-of-order core: in-order front end, dataflow-style execution, in-order retirement.

An out-of-order core renames registers (removing WAR and WAW), lets each instruction execute when its operands are ready (restricted dataflow), and retires in order through the reorder buffer (precise exceptions, branch recovery). In the multiply example above, `addi` would execute while `add` waits, recovering the 3 lost cycles. Modern cores keep 200–600 instructions in flight to hide memory latency; this requires branch prediction accuracy well above 95% (Chapter 6), because a misprediction discards everything after the branch.

Worked Example 7.6: Renaming removes false dependences

```

I1: lw    x5, 0(x10)    # long-latency load (miss)
I2: add   x6, x5, x1    # RAW on x5: must wait
I3: addi  x5, x2, 4     # WAR with I2, WAW with I1
I4: sub   x7, x5, x3    # RAW on the new x5

```

Rename: I1 $x5 \rightarrow p20$; I2 $x6 \rightarrow p21$ (reads $p20$); I3 $x5 \rightarrow p22$; I4 reads $p22$, $x7 \rightarrow p23$. Now I3 and I4 depend only on $x2$, $x3$ and each other: they execute immediately while I1 waits for memory and I2 waits for I1. Without renaming, I3 could not write $x5$ until I2 had read it.

Chapter summary

- Multicycle functional units cause longer RAW stalls, structural hazards on write ports, WAW hazards and imprecise exceptions.
- Precise exceptions: record in the pipeline, take at a commit point in program order, flush younger instructions; reorder buffers generalize this.
- A CPI stack decomposes performance into ideal, data, control, structural and memory components; memory usually dominates.
- Factorial: 24 ns single-cycle, 33.8 ns multicycle, 12.3 ns pipelined, 9.3 ns with a BTB.
- Superscalar issue raises the ideal IPC to W ; out-of-order execution with renaming and a ROB approaches dataflow limits while preserving sequential semantics.

Exercises

E7.1 Recompute the CPI stack if the D-cache miss rate falls to 2% and the branch predictor improves to 95%.

Answer: $1.644 - 0.14 - 0.018 = 1.486$.

E7.2 In the multiply example, how many cycles are saved if the compiler moves `addi` between `mul` and `add`?

Answer: 1 (the `add` now waits 2 cycles instead of 3, and `addi` is not delayed).

E7.3 An instruction in IF causes a page fault in the same cycle that an older `sw` in MEM causes a misaligned-store exception. Which trap is taken, and what is `mepc`?

Answer: The store's (older): `mepc` = PC of the `sw`, `mcause` = 6.

E7.4 Draw the pipeline diagram for one iteration of the factorial loop with predict-not-taken and confirm the 6-cycle iteration time.

E7.5 Rename the sequence `add x1,x2,x3; sub x2,x1,x4; mul x1,x2,x5; add x6,x1,x2` using physical registers starting at $p10$. Which dependences remain?

Answer: $x1 \rightarrow p10$, $x2 \rightarrow p11$ (reads $p10$), $x1 \rightarrow p12$ (reads $p11$), $x6 \rightarrow p13$ (reads $p12$, $p11$): only RAW chains remain.

A

Solved Problems

Worked Example A.1: Stage-by-stage register transfers of lw and add

In the pipelined datapath every stage reads only its input pipeline register and writes only its output pipeline register (plus, in WB, the register file). For `lw x9, 8(x2)` and `add x7, x5, x6`:

Stage	<code>lw x9, 8(x2)</code>	<code>add x7, x5, x6</code>
IF	IF/ID.IR $\leftarrow$ M[PC]; IF/ID.PC $\leftarrow$ PC; PC $\leftarrow$ PC+4	same
ID	ID/EX.A $\leftarrow$ R[x2]; ID/EX.imm $\leftarrow$ 8; ID/EX.rd $\leftarrow$ 9; ctrl	ID/EX.A $\leftarrow$ R[x5]; ID/EX.B $\leftarrow$ R[x6]; rd $\leftarrow$ 7
EX	EX/MEM.Y $\leftarrow$ A + imm	EX/MEM.Y $\leftarrow$ A + B
MEM	MEM/WB.D $\leftarrow$ M[EX/MEM.Y]	MEM/WB.Y $\leftarrow$ EX/MEM.Y
WB	R[9] $\leftarrow$ MEM/WB.D	R[7] $\leftarrow$ MEM/WB.Y

Note that the destination register number (9 or 7) must travel with the instruction all the way to WB: using the rd field of the instruction *currently* in ID would write the result into the wrong register — a classic first-pipeline bug.

Worked Example A.2: Moving jump resolution into ID

JAL's target ($PC + \text{imm}_J$) needs no register value, so it can be computed in ID with a dedicated adder. Then a JAL costs 1 bubble instead of 2 under predict-not-taken. For the factorial loop (one JAL per 4-instruction iteration) the steady-state iteration time falls from 6 to 5 cycles (CPI 1.25). JALR needs rs1 and is usually left in EX (or predicted by a return address stack). The hardware cost: a 32-bit adder in ID and a third PC-source input; the hazard unit must flush only IF/ID (one wrong-path instruction) when the ID-stage jump redirects fetch.

Worked Example A.3: Scheduling to remove load-use stalls

```
# a[i] = b[i] + c[i] for i = 0..n-1  (a0=a, a1=b, a2=c, a3=n)
loop: lw  t0, 0(a1)
      lw  t1, 0(a2)
      add t2, t0, t1      # load-use on t1: 1 stall
      sw  t2, 0(a0)
      addi a1, a1, 4
      addi a2, a2, 4
      addi a0, a0, 4
      addi a3, a3, -1
```

```
bne a3, x0, loop    # taken: 2 bubbles (predict NT)
```

Per iteration: 9 instructions + 1 stall + 2 bubbles = 12 cycles. Scheduling: move `addi a1, a1, 4` between the second `lw` and the `add` (and fix nothing else, since `a1` is not used afterwards in the iteration): 11 cycles. With a branch predictor that predicts the loop branch correctly: 9 cycles (CPI 1.0). Unrolling twice and interleaving the two iterations' loads removes the stall even without the `addi` trick and amortizes the loop overhead: 13 instructions (4 loads, 2 adds, 2 stores, 4 `addis`, 1 branch) per 2 elements, i.e. 6.5 cycles per element.

Worked Example A.4: Hazards in a six-stage pipeline with a two-cycle data cache

Split MEM into M1 and M2 (IF, ID, EX, M1, M2, WB) so that a larger data cache fits. Load data is now available at the end of M2. Forwarding paths: EX/M1 → EX (ALU results, distance 1), M1/M2 → EX (distance 2), M2/WB → EX (distance 3, includes load data). Load-use penalties: consumer at distance 1 waits 2 cycles, at distance 2 waits 1 cycle, at distance 3 none. With 25% loads, 40% of which have a consumer at distance 1 and 20% at distance 2:

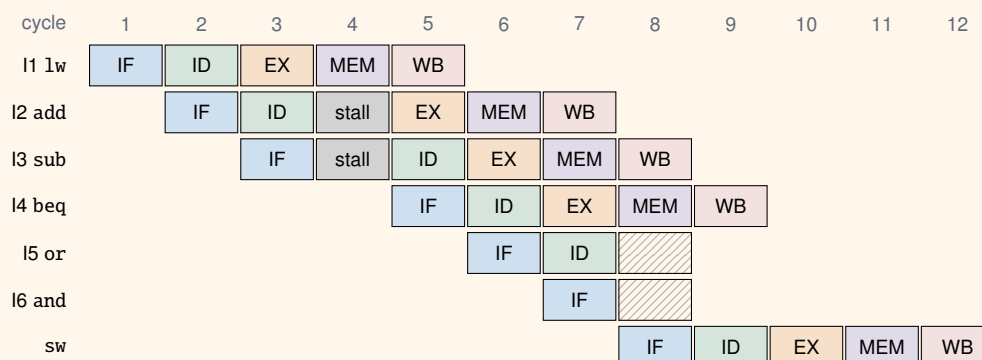
$$\text{CPI}_{\text{load}} = 0.25 \times (0.40 \times 2 + 0.20 \times 1) = 0.25$$

versus 0.10 in the five-stage pipeline. If splitting MEM lets the clock shrink from 300 to 260 ps and the cache grows enough to cut the miss contribution from 0.28 to 0.18 CPI, the time per instruction changes from $1.644 \times 300 = 493$ ps (CPI stack of Chapter 7) to $(1.644 + 0.15 - 0.10) \times 260 = 440$ ps: 11% faster despite the extra stalls. Deeper pipelines are justified only when the cycle-time and miss-rate gains outweigh the added stalls.

Worked Example A.5: A complete pipeline diagram

Five-stage pipeline, full forwarding, register file written in the first half of the cycle and read in the second, branches resolved in EX, predict not taken. Draw the diagram and count cycles for (assume the `beq` is taken):

```
I1: lw    x1, 0(x2)
I2: add   x3, x1, x4    # load-use on x1
I3: sub   x5, x3, x6    # RAW on x3 (forwarded)
I4: beq   x5, x0, L     # RAW on x5 (forwarded); taken
I5: or    x7, x8, x9    # wrong path
I6: and   x10, x7, x1   # wrong path
L:  sw    x5, 4(x2)
```



I2 stalls one cycle (load-use) and receives x1 from MEM/WB; I3 gets x3 from EX/MEM; I4 gets x5 from EX/MEM; I4 resolves taken in EX (cycle 7) and the two wrong-path instructions are flushed; sw is fetched in cycle 8 and writes back (does nothing) in cycle 12. Total: 12 cycles for 5 useful instructions (1 load-use stall + 2 flush bubbles + 4 fill cycles).

Worked Example A.6: Optimal pipeline depth with branch penalties

Logic delay $T = 3200$ ps, latch overhead $S = 60$ ps. A deeper pipeline also resolves branches later: assume the misprediction penalty is $k/2$ cycles for a k -stage pipeline. Branches are 20% of instructions, mispredicted 8% of the time. Time per instruction:

$$t(k) = \left(\frac{T}{k} + S\right) \left(1 + 0.2 \times 0.08 \times \frac{k}{2}\right) = \left(\frac{3200}{k} + 60\right) (1 + 0.008k).$$

k	cycle (ps)	CPI	time/instr (ps)
5	700	1.040	728
10	380	1.080	410
15	273	1.120	306
20	220	1.160	255
30	167	1.240	207
40	140	1.320	185
60	113	1.480	168
80	100	1.640	164
100	92	1.800	166

Setting the derivative to zero: $t(k) = 3200/k + 25.6 + 60 + 0.48k$, so $k_{\text{opt}} = \sqrt{3200/0.48} \approx 82$, $t \approx 164$ ps. Doubling the misprediction rate to 16% halves the last term's denominator: $k_{\text{opt}} = \sqrt{3200/0.96} \approx 58$. Better branch prediction allows deeper pipelines. (Real optima are lower still because of power and the latch overhead's growth with clock skew.)

Worked Example A.7: Predictors on a nested-loop trace

```
for (i = 0; i < 3; i++)           // outer branch B1 (backward, taken 2 of 3)
  for (j = 0; j < 2; j++)         // inner branch B2 (backward, taken 1 of 2)
    ...;
```

Dynamic sequence (compiled as bottom-tested loops): B2: T N, B1: T, B2: T N, B1: T, B2: T N, B1: N. Counting per predictor (2-bit counters start at 10, 1-bit at T, BTFN treats both as backward $\Rightarrow$ taken):

Predictor	B2 correct (of 6)	B1 correct (of 3)	Accuracy
BTFN / always taken	3	2	5/9 = 56%
1-bit last-time	1	2	3/9 = 33%
2-bit counter	3	2	5/9 = 56%
gshare, 2-bit history (steady state)	6	2	8/9 = 89%

With such short loops, only history-based prediction helps: the global history “B2 was just taken” perfectly predicts that B2 will now exit; “B2 just exited” predicts that B1 follows; B1’s own exit (the third time) is distinguishable only with a history long enough to span a whole

outer iteration (≥ 6 outcomes), with which the trace becomes 100% predictable.

Worked Example A.8: Multicycle versus pipeline for a floating-point workload

Mix: 30% FP add (latency 2), 20% FP mul (latency 4), 25% load, 10% store, 15% branch. **Multicycle**, 300 ps cycle: FP add 5 cycles, FP mul 7, load 5, store 4, branch 3: $\text{CPI} = 1.5 + 1.4 + 1.25 + 0.4 + 0.45 = 5.00 \Rightarrow 1500 \text{ ps/instruction}$. **Pipeline**, 300 ps cycle, fully pipelined FP units, forwarding; average RAW stalls: 0.5 per FP mul, 0.3 per FP add, 0.3 per load; branches 0.15 penalty-cycles each (predictor): $\text{CPI} = 1 + 0.2 \times 0.5 + 0.3 \times 0.3 + 0.25 \times 0.3 + 0.15 \times 0.15 = 1.29 \Rightarrow 387 \text{ ps}$. Speedup $3.9\times$.

Worked Example A.9: Microcode for a CISC-style instruction

Implement `memadd (rd), (rs):  $M[rd] \leftarrow M[rd] + M[rs]$` on the multicycle datapath (single memory, one ALU). Microroutine after fetch/decode: After decode, $A = R[rd]$ (destination address) and $B = R[rs]$ (source address).

1. $\text{MDR} \leftarrow M[A]$ (AdrSrc selects A; first operand).
2. $\text{TEMP} \leftarrow \text{MDR}$.
3. $\text{MDR} \leftarrow M[B]$ (second operand).
4. $\text{ALUOut} \leftarrow \text{TEMP} + \text{MDR}$.
5. $M[A] \leftarrow \text{ALUOut}$.

Five cycles after decode (7 in total), an extra temporary register and an extra AdrSrc input: easy in microcode, but an instruction that accesses memory three times is a nightmare for a pipeline — the essence of the RISC argument.

Formula Sheet

Quantity	Formula
Execution time	$T = N \times \overline{\text{CPI}} \times t_C$
Average CPI	$\overline{\text{CPI}} = \sum_i p_i \text{CPI}_i$
Single-cycle clock	$t_C > t_{\text{IFetch}} + t_{\text{RFetch}} + t_{\text{ALU}} + t_{\text{DMem}} + t_{\text{RWB}}$
Multicycle clock	$t_C \geq \max_{\text{step}}(t_{\text{step}}) + t_{cq} + t_{su}$
Pipelined clock	$t_C = \max_{\text{stage}}(t_{\text{stage}}) + S$
k -stage pipeline, N instructions	$(k + N - 1)$ cycles, no stalls
Pipelined bandwidth and cost	$\text{BW}_k = 1/(T/k + S)$, $\text{Cost}_k = G + Lk$
Optimal depth (cost/performance)	$k_{\text{opt}} = \sqrt{GT/(LS)}$
CPI with stalls	$\text{CPI} = (N + S)/N = 1 + \text{stalls per instruction}$
RAW hazard condition (5-stage)	$\text{dist}(i, j) \leq \text{dist}(\text{ID}, \text{WB}) = 3$ (2 with write-before-read)
Load-use penalty (with forwarding)	1 cycle
Branch CPI	$\text{CPI} = 1 + f_{\text{br}} \times (1 - \text{accuracy}) \times \text{penalty}$
Predict-not-taken accuracy	$1 - f_{\text{taken}}$
Loop accuracy, N iterations	last-time $(N - 2)/N$; 2-bit counter $(N - 1)/N$
Wasted slots per misprediction	$N_{\text{resolve}} \times W$
Control store (horizontal)	$2^n \times (k + \text{sequencing bits})$