



The Single-Cycle RISC-V Processor

Building a complete RV32I CPU: hardware elements, datapath, control, timing, and running a real program from C source to machine code

What this handbook covers

- | | |
|---|---------------------------------------|
| 1 Hardware elements of a CPU | 5 Timing and performance |
| 2 The five steps of instruction execution | 6 From C to machine code: factorial |
| 3 Building the datapath | 7 RTL implementation and verification |
| 4 The control unit | 8 Extensions: LSU, M, traps, MMIO |

Dr. Bikram Paul

School of Computing and Electrical Engineering (SCEE)
Indian Institute of Technology Mandi
bikram@iitmandi.ac.in

Jan–June 2026

Contents

1	Hardware Elements of a CPU	1
1.1	Combinational versus state elements	1
1.2	Multiplexers and decoders	1
1.2.1	Multiplexers	2
1.2.2	Decoders	2
1.3	The arithmetic–logic unit	2
1.3.1	A one-bit ALU slice	2
1.3.2	Comparisons and flags	3
1.3.3	The shifter	3
1.3.4	ALU control encoding	4
1.4	State elements	4
1.4.1	Flip-flops and registers	4
1.4.2	The register file	5
1.5	The memory model	6
1.6	Timing of state-to-state paths	7
	Exercises	7
2	The Five Steps of Instruction Execution	9
2.1	Instruction processing as a state transformation	9
2.2	The five steps	9
2.3	The RV32I subset implemented	10
2.4	The immediate generator	11
2.5	Decoding a short program by hand	12
	Exercises	13
3	Building the Single-Cycle Datapath	14
3.1	Overview	14
3.2	Instruction fetch	14
3.3	R-type (register–register) instructions	15
3.4	I-type ALU instructions	15
3.5	Load and store instructions	16

3.6	Conditional branches	17
3.7	Jumps and upper immediates	17
3.8	The complete single-cycle datapath	18
3.9	Tracing instructions through the datapath	18
	Exercises	20
4	The Control Unit	21
4.1	Control signals	21
4.2	The main control table	22
4.3	ALU control	22
4.3.1	Two-level decoding	22
4.3.2	RV32I ALU control	23
4.4	Main decoder logic	24
4.4.1	Branch condition logic	25
4.5	Implementing the control unit	25
4.6	Control in action and control faults	26
	Exercises	27
5	Timing and Performance of the Single-Cycle Processor	28
5.1	The clock-period constraint	28
5.2	Component delays	28
5.3	Critical path of each instruction class	28
5.4	Execution time	29
5.5	Hardware utilization and energy	30
5.6	Summary of the single-cycle design	31
	Exercises	31
6	From C to Machine Code: Running Factorial	32
6.1	The C program	32
6.2	The compilation toolchain	32
6.3	Unoptimized compiler output	33
6.4	Hand-optimized RV32IM version	34
6.4.1	Assembling by hand	34
6.4.2	The memory image	35
6.5	Executing factorial on the single-cycle processor	35
6.6	Compressed instructions in the real binary	36

6.7	ELF files, bare metal and Linux	37
	Exercises	38
7	RTL Implementation and Verification	39
7.1	From block diagram to modules	39
7.2	The Verilog model	39
7.2.1	Walking through the code	41
7.3	Running factorial	42
7.4	A self-checking instruction test	43
7.5	Verification of real processors	44
7.6	Synthesis and FPGA implementation	44
	Exercises	45
8	Extending the Single-Cycle Processor	47
8.1	A load/store unit for sub-word accesses	47
8.2	The M extension: multiply and divide	48
8.2.1	Multiplier hardware	49
8.2.2	Division cannot be single-cycle	49
8.3	Exceptions and control/status registers	50
8.3.1	What must happen on a trap	50
8.3.2	Datapath additions	50
8.4	Memory-mapped I/O	51
	Exercises	53
A	RV32I(M) Reference Card	54
B	Solved Design Problems	56

Hardware Elements of a CPU

A processor is built from exactly two kinds of hardware: **combinational circuits**, whose outputs are a function of their present inputs only, and **synchronous state elements**, which remember values from one clock cycle to the next. In this unit we assemble a complete RISC-V processor that executes every instruction in a single clock cycle. Before drawing any datapath we need precise models of the parts: multiplexers, decoders, the ALU and its shifter, flip-flops and registers, the register file, and memory. This chapter builds each of them, with the timing behaviour that will later determine the processor's clock period.

1.1 Combinational versus state elements

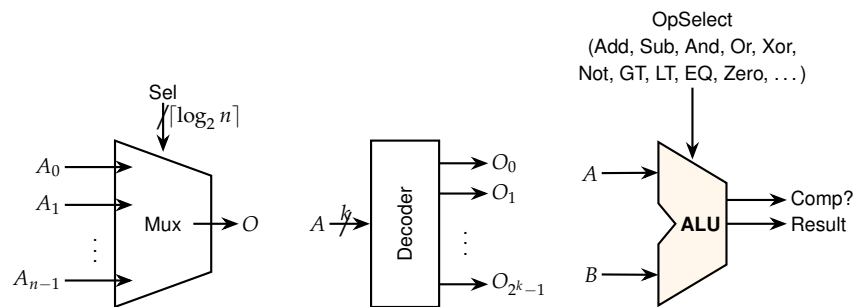


Figure 1.1: The three workhorse combinational blocks: multiplexer, decoder and ALU.

- **Combinational circuits:** multiplexers, decoders, adders, ALUs, comparators, shifters, sign-extendors. No memory; output settles a propagation delay after the inputs.
- **Synchronous state elements:** flip-flops, registers, the register file, SRAM and DRAM. They capture new values only at a clock edge.

Edge-triggered clocking methodology

All state elements in our processor are written on the *rising* edge of a single clock. Between edges, the combinational logic computes the next values from the current state. A signal therefore travels from a state element, through combinational logic, into a state element within one clock period. This discipline makes timing analysis tractable: there is no combinational feedback loop, and every path is bounded by two clock edges.

1.2 Multiplexers and decoders

1.2.1 Multiplexers

An n -input multiplexer selects one of n inputs $A_0 \dots A_{n-1}$ using $\lceil \log_2 n \rceil$ select bits. A 2:1 multiplexer computes $O = \bar{s}A_0 + sA_1$; wider multiplexers are trees of 2:1 multiplexers, or a single AND–OR level driven by a decoder of the select lines.

You cannot just join wires together

When two sources must feed the same input (e.g. the register-file write data may come from the ALU *or* from data memory; the PC may come from PC+4 *or* from a branch target), the wires cannot simply be tied together: two drivers fighting on one net produce an undefined voltage and excessive current. A multiplexer — with a select signal generated by the control unit — is always required. Every multiplexer in a datapath implies a control signal.

Worked Example 1.1: Cost and delay of a 32-bit 4:1 multiplexer

As a tree of 2:1 MUXes: per bit, 3 two-input MUXes in 2 levels; for 32 bits, 96 MUX2 cells. With a MUX2 delay of 25 ps the delay is 50 ps.

As AND–OR with a 2:4 decoder: the decoder (shared by all 32 bits) produces $\bar{s}_1\bar{s}_0$, $\bar{s}_1s_0$, $s_1\bar{s}_0$, s_1s_0 ; each bit uses four 2-input ANDs and one 4-input OR. Delay: decoder + AND + OR ≈ 3 gate delays. The select lines fan out to 32 bits, so they need buffering — in practice adding one more gate delay. For an n -input multiplexer the tree delay grows as $\log_2 n$.

1.2.2 Decoders

A k -to- 2^k decoder asserts exactly one of its 2^k outputs: output O_i is 1 iff the input equals i . Decoders select one register of the register file for writing and one word of a memory array.

Worked Example 1.2: The 5-to-32 write decoder of the register file

Register numbers are 5 bits. Output $O_i = WE \cdot [\text{rd} = i]$. A direct implementation uses 32 five-input AND gates (plus the write-enable): output O_{13} for $13 = 01101_2$ is $\bar{r}_4 r_3 r_2 \bar{r}_1 r_0 \cdot WE$. A *predecoded* implementation splits the address into 2+3 bits: a 2:4 and a 3:8 decoder (12 small gates) followed by 32 two-input ANDs — fewer transistors and smaller fan-in, the standard structure in SRAM row decoders.

1.3 The arithmetic–logic unit

1.3.1 A one-bit ALU slice

Figure 1.2 shows a one-bit ALU slice supporting AND, OR, ADD, SUB and the “set on less than” comparison. Thirty-two such slices, with carries chained, form a 32-bit ALU.

- **AND / OR:** computed directly by gates.
- **ADD:** full adder with carry chain.
- **SUB:** $A - B = A + \bar{B} + 1$ in two’s complement. Setting Binvert=1 inverts B through the XOR gates, and the carry-in of bit 0 is set to 1. The same adder therefore serves both operations.

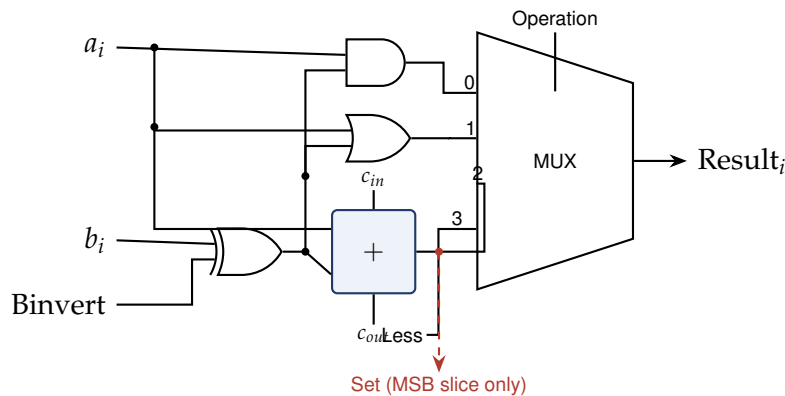


Figure 1.2: One bit of a simple ALU. Binvert with $c_{in} = 1$ turns the adder into a subtractor; the “Less” input of bit 0 receives the “Set” output of bit 31.

- **SLT (set on less than):** compute $A - B$; the result is 1 if it is negative. The MSB slice produces Set, which is routed to the Less input of bit 0; all other Less inputs are 0.

1.3.2 Comparisons and flags

Definition 1.1: ALU flags

Zero $Z = \overline{r_{31} + r_{30} + \dots + r_0}$ (a 32-input NOR); **Negative** $N = r_{31}$; **Carry** $C = c_{32}$ (carry out of the MSB); **Overflow** $V = c_{32} \oplus c_{31}$ (carry into and out of the MSB differ).

Signed and unsigned less-than from one subtraction

After computing $A - B$ (i.e. $A + \overline{B} + 1$):

- **signed** $A < B$ iff $N \oplus V = 1$;
- **unsigned** $A < B$ iff $C = 0$ (no carry out means a borrow occurred).

Using N alone for signed comparison is wrong whenever the subtraction overflows.

Worked Example 1.3: Flags for four subtractions (32-bit)

A	B	$A - B$ (hex)	N	Z	C	V	Comparison
5	3	00000002	0	0	1	0	slt=0, sltu=0
3	5	FFFFFFFE	1	0	0	0	slt=1, sltu=1
-5 (FFFFFFFB)	3	FFFFFFF8	1	0	1	0	slt=1, sltu=0
7FFFFFFF	FFFFFFF (-1)	80000000	1	0	0	1	slt=0, sltu=1

Row 3: $-5 < 3$ signed, but as unsigned $0xFFFFF8 > 3$, so sltu=0 — consistent with $C = 1$.
 Row 4: $2^{31} - 1 - (-1) = 2^{31}$ overflows: $N = 1$ but $V = 1$, so $N \oplus V = 0$ and slt=0 (correct: $2^{31} - 1 > -1$). Using N alone would give the wrong answer.

1.3.3 The shifter

RV32I has logical left (sll), logical right (sr) and arithmetic right (sra) shifts by 0–31 positions. A **barrel shifter** performs any shift in a single cycle using $\log_2 32 = 5$ stages of 2:1 multiplexers;

stage k shifts by 2^k if bit k of the shift amount is set.

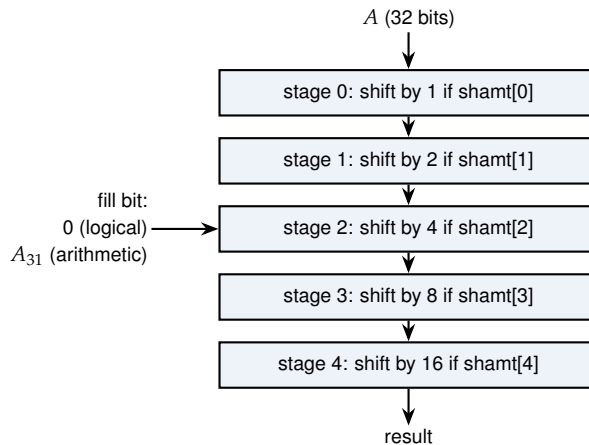


Figure 1.3: A logarithmic barrel shifter: five stages of 32 two-input multiplexers (160 MUX2s).

Worked Example 1.4: Shifting `0xF0000035` right by 9

`shamt = 9 = 010012`: stages 0 (by 1) and 3 (by 8) are active.

- `srl`: fill with 0. `0xF0000035` $\gg 9 = 0x00780000$ (the low 9 bits `0x035` are lost; `0xF0000000` $\gg 9 = 0x00780000).$
- `sra`: fill with the sign bit (1): `0xFFFF80000`. As signed numbers: `0xF0000035` $= -268\,435\,403$; $\lfloor -268\,435\,403/512 \rfloor = -524\,288 = 0xFFFF80000$. Arithmetic right shift is floor division by 2^k .

A right shift is often built from the left shifter by bit-reversing input and output, saving area at the cost of two reversal stages (wires only).

1.3.4 ALU control encoding

Our ALU receives a 4-bit control code (the lecture's table):

ALU control	Function	ALU control	Function
0000	AND	0110	subtract
0001	OR	0111	set on less than
0010	add	1100	NOR

Bit 2 is exactly `Binvert` (and the carry-in): subtract and `SLT` both invert B . Bits 1–0 select the MUX input (00 AND, 01 OR, 10 adder, 11 Less). Bit 3 inverts A (`NOR` $= \overline{A} \cdot \overline{B}$). Full RV32I additionally needs XOR, the three shifts and `SLTU`; a practical RV32I ALU therefore uses a 4-bit code derived from `funct3` and one bit of `funct7` (Chapter 4).

1.4 State elements

1.4.1 Flip-flops and registers

An edge-triggered D flip-flop samples D on the rising edge of Clk ; with an enable, it loads only when $En=1$ (Figure 1.4). A **register** is a row of such flip-flops sharing clock and enable.

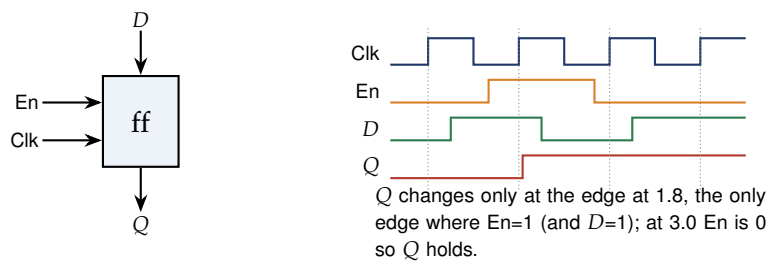


Figure 1.4: Flip-flop with enable: edge-triggered, data sampled at the rising edge only when enabled.

1.4.2 The register file

The RISC-V register file holds 32 registers of 32 bits. A single-cycle datapath must read two source registers ($rs1$, $rs2$) and write one destination (rd) in the same cycle: a **2-read, 1-write (2R1W)** register file.

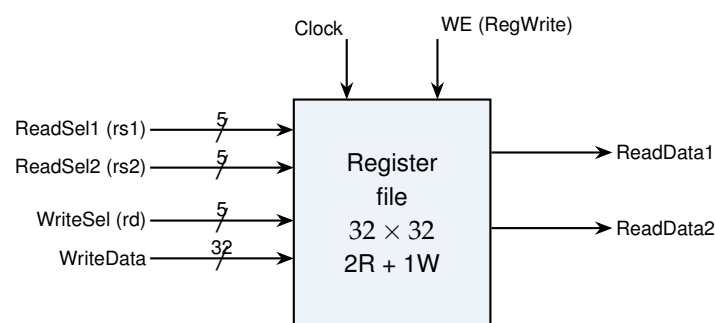


Figure 1.5: Interface of the 2R1W register file.

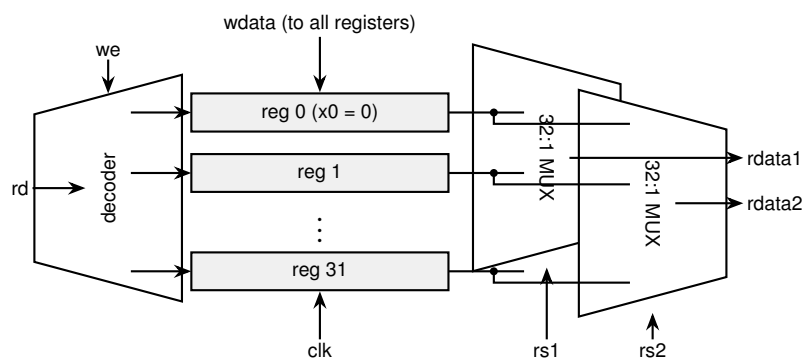


Figure 1.6: Internal structure: a write decoder generates one enable per register; two 32:1 multiplexers implement the two combinational read ports.

Behaviour.

- **Reads are combinational:** $ReadData1 = R[rs1]$ and $ReadData2 = R[rs2]$ appear a read-access time after the register numbers become valid.
- **Writes are synchronous:** $R[rd] \leftarrow WriteData$ at the rising edge if $WE=1$ and $rd \neq 0$.
- Register $x0$ is not stored: its read port simply returns 0, and writes to it are discarded.

Worked Example 1.5: Sizing the register file

Storage: 31 registers $\times$ 32 bits = 992 flip-flops (x0 needs none). **Read ports:** each is a 32-bit 32:1 multiplexer = $32 \times 31 = 992$ MUX2s, so two ports cost 1984 MUX2s, depth $\log_2 32 = 5$ MUX levels ($5 \times 25 = 125$ ps). **Write:** a 5:32 decoder plus 31 enable gates. With ≈ 24 transistors per flip-flop and 12 per MUX2, the register file needs about $992 \cdot 24 + 1984 \cdot 12 \approx 47\,600$ transistors — more than ten times an Intel 4004.

Port scaling. A p -read, w -write register file implemented as an SRAM array has area roughly proportional to $(p + w)^2$ (each port adds a word line and a bit line per cell). Going from 2R1W (3 ports) to the 8R4W (12 ports) file of a 4-wide superscalar core multiplies area by about $(12/3)^2 = 16$. This is why wide out-of-order cores split or cluster their register files.

Read-during-write in the same cycle

If an instruction reads a register that is written at the end of the *same* cycle, it sees the *old* value — correct for a single-cycle machine, since the write belongs to the same instruction (e.g. `addi x5, x5, 1` must read the old x5). In a pipeline, a later instruction may read a register in the same cycle an earlier one writes it; pipelined designs therefore often write in the first half of the cycle and read in the second half (“write before read”), or add a bypass. Chapter 3 of Unit 3 returns to this.

1.5 The memory model

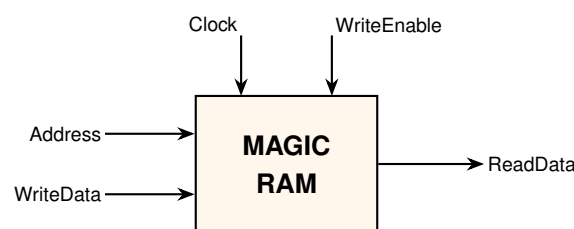


Figure 1.7: The idealised memory used for single-cycle design.

For the single-cycle processor we assume a “magic” memory in which reads and writes are always completed in one cycle:

- a **read** can be done at any time, i.e. it is combinational: ReadData follows Address after the access time;
- a **write** is performed at the rising clock edge if WriteEnable is asserted; the write address and data must be stable at the clock edge (setup and hold).

We use two such memories: an **instruction memory** (read-only from the processor’s point of view) and a **data memory**. This is a Harvard organization, needed because a single-cycle machine must fetch an instruction *and* access data in the same cycle (recall the structural argument of Unit 1).

Advanced Insight: Why real memories are not magic

Real SRAM arrays are synchronous: an address is registered on one clock edge and data appear after the next. FPGA block RAMs (e.g. 36 Kb BRAMs on Zynq) behave this way, which is why a straightforward single-cycle CPU cannot use BRAM for instruction memory

without either (a) clocking the memory on the falling edge, (b) using distributed (LUT) RAM with asynchronous read, or (c) moving to a multi-cycle or pipelined design. A large DRAM needs tens of nanoseconds — dozens of processor cycles — which is the origin of the cache hierarchy studied in Unit 4. Magic memory is a teaching abstraction that lets us focus on the datapath; the real memory system is a unit of its own.

1.6 Timing of state-to-state paths

Every path in the processor starts at a state element (PC, register file, memory) and ends at a state element. Its length bounds the clock period:

$$T_{clk} \geq t_{clk \rightarrow q} + \sum t_{\text{combinational}} + t_{\text{setup}} + t_{\text{skew}}. \quad (1.1)$$

In the single-cycle processor the longest such path runs from the PC, through instruction memory, the register file, the ALU, data memory, and back to the register file's write port. Chapter 5 quantifies it.

Worked Example 1.6: A first timing budget

Component delays (a representative 28 nm-class design): PC $t_{clk \rightarrow q} = 30$ ps, instruction memory 250 ps, register-file read 150 ps, ALU 200 ps, data memory 250 ps, write-back MUX 25 ps, register-file setup 20 ps. For a load: $30 + 250 + 150 + 200 + 250 + 25 + 20 = 925$ ps $\Rightarrow f_{\max} \approx 1.08$ GHz. An add uses the same path minus data memory: 675 ps. The load sets the clock for *every* instruction — the central weakness of single-cycle design.

Chapter summary

- Datapaths combine combinational blocks (MUX, decoder, ALU, shifter, sign extender) with state elements (PC, register file, memories) under a single-edge clocking discipline.
- One adder with an inverted B and $c_{in} = 1$ provides ADD, SUB, SLT and SLTU; signed $<$ is $N \oplus V$, unsigned $<$ is $\overline{C}$.
- A 2R1W register file = write decoder + 31 registers + two 32:1 read multiplexers; reads are combinational, writes happen at the clock edge; x0 is hard-wired to zero.
- “Magic” memories read combinatorially and write at the edge; separate instruction and data memories are required for single-cycle execution.

Exercises

E1.1 Compute N, Z, C, V for $A - B$ with $A = 0x80000000, B = 1$. What do slt and sltu return?

Answer: $A - B = 0x7FFFFFFF$; $N = 0, Z = 0, C = 1, V = 1$. slt = $N \oplus V = 1$ ($-2^{31} < 1$); sltu = $\overline{C} = 0$.

E1.2 Compute sra and srl of $0x800000F0$ by 4 and by 31.

Answer: by 4: $0xF80000F0, 0x080000F0$; by 31: $0xFFFFFFFF, 0x00000001$.

E1.3 How many MUX2s and levels does a 64-bit barrel shifter need?

Answer: $6 \text{ stages} \times 64 = 384 \text{ MUX2s}$, 6 levels.

E1.4 Estimate the transistor count of a 2R1W 16×32 register file (RV32E) using the numbers in this chapter.

Answer: $15 \cdot 32 = 480 \text{ FFs}$ (11 520 T) + two 16:1 ports ($2 \cdot 32 \cdot 15 = 960 \text{ MUX2}$, 11 520 T): $\approx 23\,000$.

E1.5 Why must the write address and data of a register file be stable around the clock edge, while the read address may change at any time?

E1.6 Design the “Zero” detector for a 32-bit result using 4-input NOR/NAND gates only. How many levels?

Answer: $8 \text{ NOR4} \rightarrow 2 \text{ NAND4} \rightarrow 1 \text{ NOR2}$: 3 levels, 11 gates.

2

The Five Steps of Instruction Execution

Before building hardware we must be precise about *what* each instruction does. This chapter breaks instruction execution into five steps, lists the RV32I subset our processor implements together with its register-transfer semantics, and studies the one piece of datapath that is unique to RISC-V's encoding: the immediate generator.

2.1 Instruction processing as a state transformation

The ISA describes the machine abstractly as a finite-state machine whose state is the **architectural state** AS — the PC, the 32 registers and memory. Executing one instruction transforms AS into AS'. From the ISA's point of view there are no intermediate states: one state transition per instruction.

The microarchitecture decides *how* AS becomes AS'. The single-cycle microarchitecture makes the transformation in exactly **one clock cycle**: at the start of the cycle the state elements hold AS; combinational logic computes AS'; at the next rising edge the PC, register file and data memory are updated simultaneously. (Unit 3 introduces the alternative: $AS \rightarrow AS+MS_1 \rightarrow AS+MS_2 \rightarrow \dots \rightarrow AS'$, using programmer-invisible micro-states over several cycles.)

2.2 The five steps

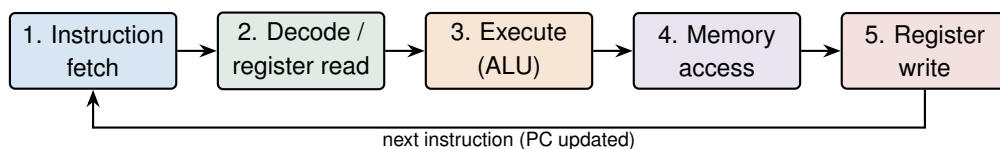


Figure 2.1: The five steps of executing a RISC-V instruction.

Step 1: Instruction fetch (IF). The 32-bit instruction word must first be fetched from memory (in a real machine, through the cache–memory hierarchy). This is also where the PC is incremented: $PC \leftarrow PC+4$, pointing to the next instruction, because memory is byte addressed and each instruction is 4 bytes.

Step 2: Instruction decode (ID). Gather data from the fields — decode all necessary instruction data:

1. read the opcode to determine the instruction type and the field lengths;
2. read data from all necessary registers: add reads two registers, addi reads one, jal reads none.

Because RISC-V places `rs1` and `rs2` at fixed positions, the register file can be read *in parallel* with decoding, even before we know whether the values are needed.

Step 3: Execute (ALU). The real work of most instructions is done here:

- arithmetic ($+$, $-$, $\times$, $\div$), shifts, logic ($\&$, $|$), comparisons (`slt`);
- loads and stores: `lw t0, 40(t1)` accesses memory at address $t1 + 40$ — the addition is done in this stage;
- conditional branches: the comparison is done in this stage (one solution).

Step 4: Memory access (MEM). Only loads and stores do anything here; other instructions remain idle during this step (or skip it altogether). Because these instructions have a unique step, we need this extra stage to account for them. Thanks to caches, this stage is expected to be fast.

Step 5: Register write (WB). Most instructions write the result of some computation into a register: arithmetic, logical, shifts, loads, `slt`. Stores and branches write nothing into a register at the end and remain idle during this step. Jumps (`jal`, `jalr`) write the return address $PC+4$.

Table 2.1: What each instruction class does in each step (RTL notation; $M[]$ is memory).

Class	IF	ID	EX	MEM	WB
R-type	$IR \leftarrow M[PC]$	$A \leftarrow R[rs1],$ $B \leftarrow R[rs2]$	$Y \leftarrow A \text{ op } B$	—	$R[rd] \leftarrow Y$
I-ALU	$IR \leftarrow M[PC]$	$A \leftarrow R[rs1], \text{imm}$	$Y \leftarrow A \text{ op } \text{imm}$	—	$R[rd] \leftarrow Y$
Load	$IR \leftarrow M[PC]$	$A \leftarrow R[rs1], \text{imm}$	$\text{addr} \leftarrow A + \text{imm}$	$D \leftarrow M[\text{addr}]$	$R[rd] \leftarrow D$
Store	$IR \leftarrow M[PC]$	A, B, imm	$\text{addr} \leftarrow A + \text{imm}$	$M[\text{addr}] \leftarrow B$	—
Branch	$IR \leftarrow M[PC]$	A, B, imm	$\text{cond} \leftarrow \text{cmp}(A, B);$ $\text{tgt} \leftarrow PC + \text{imm}$	—	—
JAL	$IR \leftarrow M[PC]$	imm	$\text{tgt} \leftarrow PC + \text{imm}$	—	$R[rd] \leftarrow PC + 4$
JALR	$IR \leftarrow M[PC]$	A, imm	$\text{tgt} \leftarrow (A + \text{imm}) \& \sim 1$	—	$R[rd] \leftarrow PC + 4$
LUI	$IR \leftarrow M[PC]$	imm	$Y \leftarrow \text{imm}$	—	$R[rd] \leftarrow Y$
AUIPC	$IR \leftarrow M[PC]$	imm	$Y \leftarrow PC + \text{imm}$	—	$R[rd] \leftarrow Y$

All instructions also update the PC: $PC \leftarrow PC + 4$, except taken branches and jumps, which load the target.

2.3 The RV32I subset implemented

Only bit 30 (`funct7[5]`) ever differs among R-type ALU operations; it distinguishes ADD/SUB and SRL/SRA. The I-type ALU instructions reuse the same `funct3` codes; `srai` sets the same bit 30 in its immediate field. Branch `funct3` codes: 000 BEQ, 001 BNE, 100 BLT, 101 BGE, 110 BLTU, 111 BGEU. Load/store `funct3`: 000 byte, 001 half, 010 word, 100 LBU, 101 LHU.

Table 2.2: Opcodes of the implemented RV32I instructions.

opcode[6:0]	Format	Instructions	Semantics
0110011	R	add, sub, sll, slt, sltu, xor, srl, sra, or, and	$rd \leftarrow rs1 \text{ op } rs2$
0010011	I	addi, slti, sltiu, xori, ori, andi, slli, srli, srai	$rd \leftarrow rs1 \text{ op } imm$
0000011	I	lb, lh, lw, lbu, lhu	$rd \leftarrow M[rs1 + imm]$
0100011	S	sb, sh, sw	$M[rs1 + imm] \leftarrow rs2$
1100011	B	beq, bne, blt, bge, bltu, bgeu	if cond: $PC \leftarrow PC + imm$
1101111	J	jal	$rd \leftarrow PC + 4; PC \leftarrow PC + imm$
1100111	I	jalr	$rd \leftarrow PC + 4; PC \leftarrow (rs1 + imm) \& \sim 1$
0110111	U	lui	$rd \leftarrow imm$
0010111	U	auipc	$rd \leftarrow PC + imm$

Table 2.3: R-type function encodings (opcode 0110011).

funct7	funct3	op	funct7	funct3	op
0000000	000	ADD	0100000	000	SUB
0000000	001	SLL	0000000	101	SRL
0000000	010	SLT	0100000	101	SRA
0000000	011	SLTU	0000000	110	OR
0000000	100	XOR	0000000	111	AND

2.4 The immediate generator

Five of the six formats contain an immediate, encoded differently in each. The **immediate generator** (“Imm Select” or “sign-extend” block on the slides) extracts the immediate bits and produces a 32-bit sign-extended operand.

Table 2.4: Construction of the 32-bit immediate. $i[k]$ denotes instruction bit k ; $\{n\{i[31]\}\}$ replicates the sign bit n times.

Format	32-bit immediate
I	$\{21\{i[31]\}, i[30:25], i[24:21], i[20]\}$
S	$\{21\{i[31]\}, i[30:25], i[11:8], i[7]\}$
B	$\{20\{i[31]\}, i[7], i[30:25], i[11:8], 1'b0\}$
U	$\{i[31], i[30:20], i[19:12], 12'b0\}$
J	$\{12\{i[31]\}, i[19:12], i[20], i[30:25], i[24:21], 1'b0\}$

Worked Example 2.1: Extracting immediates from instruction words

(a) I-type 0xFF810113: binary 1111 1111 1000 0001 0000 0001 0001 0011. Opcode 0010011 (I-ALU), $rd=2$, $funct3=000$ (addi), $rs1=2$, $imm[11:0] = 111111111000 = -8$. Instruction: `addi sp, sp, -8`.

(b) S-type 0xFE112E23: 1111111 00001 00010 010 11100 0100011. $imm[11:5]=1111111$, $imm[4:0]=11100$: $imm = 111111111100 = -4$; $rs2=x1$, $rs1=x2$, $funct3=010$: `sw ra, -4(sp)`.

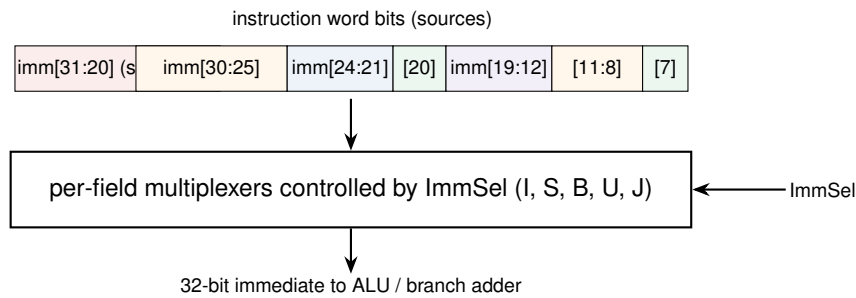


Figure 2.2: Immediate generator: bit 31 is always the sign; every output bit chooses among at most a few instruction bits (or 0), so the MUXes are small and the delay is short.

(c) **B-type** `0x00B50463`: `0 000000 01011 01010 000 0100 0 1100011`. $\text{imm}[12]=0$, $\text{imm}[10:5]=000000$, $\text{imm}[4:1]=0100$, $\text{imm}[11]=0$: $\text{imm} = 00\,000000\,0100\,0 = 8$. $\text{rs1}=\text{x10}$, $\text{rs2}=\text{x11}$, $\text{funct3}=000$: `beq a0, a1, +8`.

(d) **J-type** `0x0100006F`: $\text{bits}[31:12] = 0000\,0001\,0000\,0000\,0000$: $i[31]=0$, $i[30:21]=00000001000$, $i[20]=0$, $i[19:12]=000000000$. $\text{imm} = \{0, 00000000, 0, 0000001000, 0\} = 1000_2 \cdot 2 = 16$: `jal x0, +16` (i.e. `j +16`).

(e) **U-type** `0x000102B7`: $\text{imm}[31:12] = 0x00010$, $\text{rd}=5$: `lui t0, 0x10` $\Rightarrow$ `t0 = 0x00010000`.

Branch and jump offsets are in units of 2 bytes

B- and J-type immediates do not store bit 0 (always 0), so the encoded field reaches twice as far. The slide phrases this as “the 12-bit immediate encodes the branch target address as a signed offset from PC, in units of 16 bits (i.e. shift left by 1 then add to PC)”. Half-word granularity (rather than 4-byte) is chosen so that 16-bit compressed (C) instructions can be branch targets.

Worked Example 2.2: Branch reach and target computation

A `bne` at $\text{PC} = 0x0000\,1F40$ has immediate field bits giving $\text{imm} = -0x60$ (-96). $\text{Target} = 0x1F40 - 0x60 = 0x1EE0$. The 13-bit signed B-immediate spans $[-4096, +4094]$ bytes, i.e. ± 1024 instructions. A loop body longer than that must branch around an unconditional `jal`, whose 21-bit offset reaches ± 1 MiB.

2.5 Decoding a short program by hand

Worked Example 2.3: From hex to datapath actions

PC	Word	Instruction	Steps used
0x00	00500513	addi a0, x0, 5	IF, ID (read x0), EX (0 + 5), WB (a0 ← 5)
0x04	00A02223	sw a0, 4(x0)	IF, ID (x0, a0), EX (0 + 4), MEM (M[4] ← 5)
0x08	00402583	lw a1, 4(x0)	IF, ID, EX (0 + 4), MEM (read 5), WB (a1 ← 5)
0x0C	00B50633	add a2, a0, a1	IF, ID (a0, a1), EX (5 + 5), WB (a2 ← 10)
0x10	FEB60CE3	beq a2, a1, -8	IF, ID, EX (10 ≠ 5: not taken), PC ← 0x14

Every instruction uses IF and ID; only the load uses all five steps. In a single-cycle machine each instruction nonetheless occupies one full (load-length) clock cycle.

Chapter summary

- Executing an instruction transforms the architectural state AS into AS'; the single-cycle microarchitecture does it in one clock cycle.
- The five steps: fetch (and PC+4), decode/register read, execute (ALU), memory access (loads/stores only), register write (all but stores and branches).
- Our subset covers all RV32I computational, load/store, branch, jump and upper-immediate instructions; R-type operations differ only in funct3 and bit 30.
- The immediate generator assembles a sign-extended 32-bit immediate from five different layouts; branch and jump offsets are multiples of 2.

Exercises

E2.1 For each of sub, lbu, sh, bgeu, auipc, list the steps used and the registers read and written.

E2.2 Extract the immediate and disassemble 0xFFFF00513, 0x00C12083, 0xFE0718E3.

Answer: addi a0, x0, -1 (li a0, -1); lw ra, 12(sp); bne a4, x0, -16.

E2.3 Write the immediate-generator output for jal ra, -2048.

Answer: 0xFFFFF800; encoding 0x801FF0EF.

E2.4 Why does jalr clear bit 0 of the target address?

Answer: So that the target is at least 2-byte aligned (C extension), and bit 0 can be used by software as a tag.

E2.5 Show that the U-type immediate of lui followed by an addi can build any 32-bit constant; what correction is needed when bit 11 of the low part is 1?

Answer: Add 1 to the upper 20 bits, because addi sign-extends the low 12 bits.

3

Building the Single-Cycle Datapath

We now assemble the datapath incrementally, adding hardware one instruction class at a time and merging the pieces with multiplexers. At every step we ask two questions: *which state elements must be read and written?* and *which values must reach which inputs?* The control signals created along the way (shown in orange) are generated by the control unit of Chapter 4. The chapter ends with the complete single-cycle RV32I datapath and numeric traces of real instructions through it.

3.1 Overview

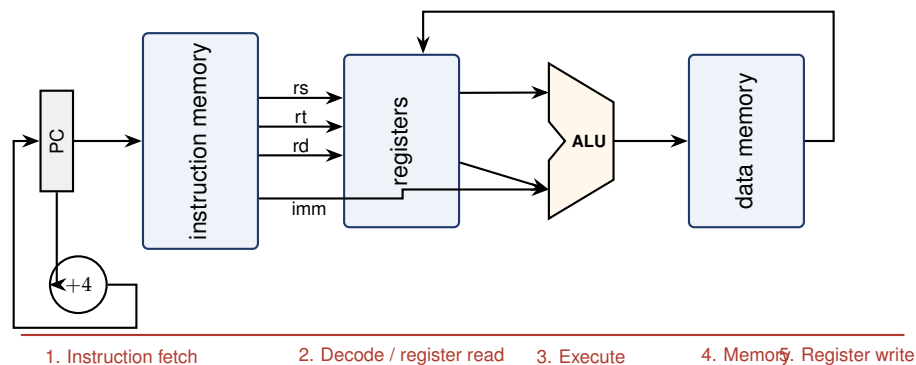


Figure 3.1: The datapath at a glance (after the lecture’s “DataPath Overview”): PC → instruction memory → registers → ALU → data memory → registers, with a +4 incrementer feeding the PC.

The basic operation of every instruction is:

- PC → instruction memory: fetch the instruction;
- register numbers → register file: read the registers;
- depending on the instruction class, use the ALU to calculate an arithmetic result, a memory address for a load/store, or a branch condition and target address;
- access data memory for a load/store;
- $PC \leftarrow \text{target address or } PC+4$.

3.2 Instruction fetch

The PC is a 32-bit register written every cycle (no enable needed in a single-cycle design). The adder is a dedicated incrementer — we cannot use the main ALU because it is busy with the instruction’s own operation in the same cycle.

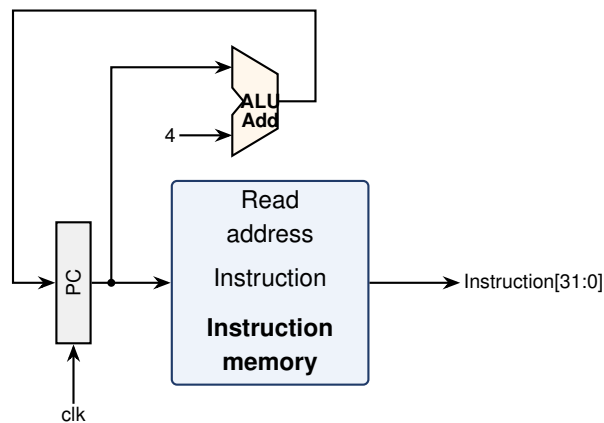


Figure 3.2: Instruction fetch: the PC addresses instruction memory; an adder computes $PC+4$, written back to the PC at the next rising edge.

3.3 R-type (register–register) instructions

R-type instructions read two registers, perform an ALU operation and write one register: $rd \leftarrow (rs1) \text{ func } (rs2)$.

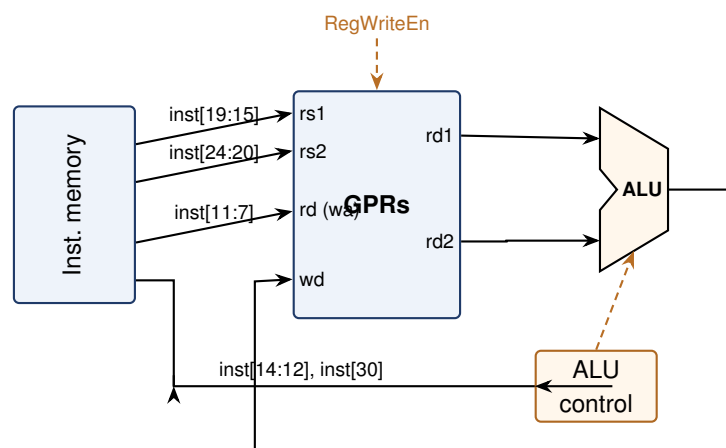


Figure 3.3: Datapath for R-type ALU instructions. The ALU-control block decodes funct3 and bit 30 into the ALU operation.

3.4 I-type ALU instructions

$rd \leftarrow (rs1) \text{ op immediate}$. The second ALU operand now comes from the immediate generator instead of $rs2$. We add a multiplexer in front of the ALU's B input, controlled by **Op2Sel** (Reg / Imm), and the immediate generator controlled by **ImmSel**.

Merging the two classes (the lecture's "Merging (Reg-Reg) & (Reg-Imm) DataPath") costs one 32-bit 2:1 multiplexer and the immediate generator. The register-write path is identical: rd is always $inst[11:7]$.

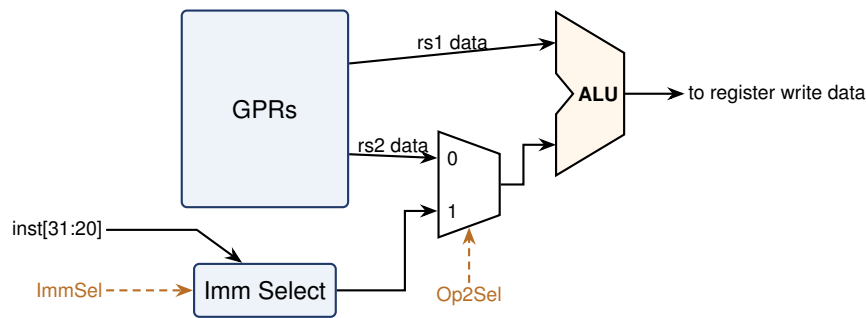


Figure 3.4: Adding an immediate operand: the Op2Sel multiplexer chooses rs2 data or the immediate.

3.5 Load and store instructions

Loads and stores calculate the address using the ALU but with a sign-extended offset ($rs1 + imm$), then:

- **load:** read data memory and write the value into register rd;
- **store:** write register rs2's value into data memory.

In both, rs1 is the base register. For a load, rd is the destination; for a store, rs2 is the data source. Note that the store's immediate is split (S-type), but rs1 and rs2 stay in their usual places, so no new register-file multiplexing is needed.

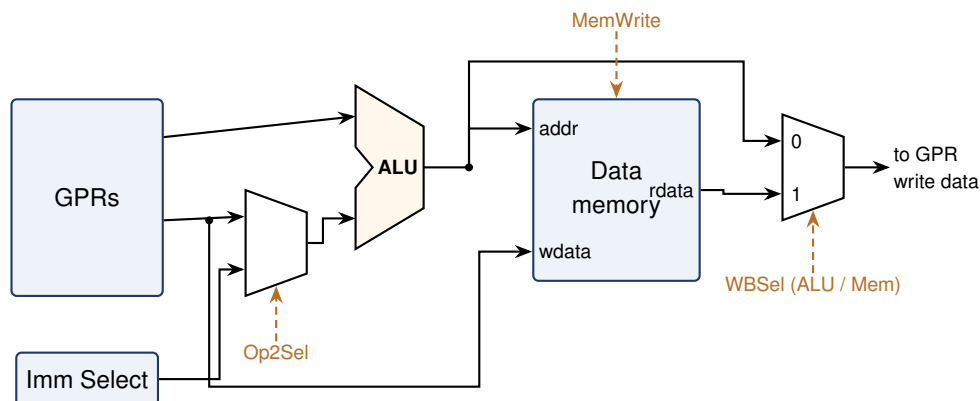


Figure 3.5: Load/store datapath: the ALU computes $rs1 + imm$; data memory is read for loads and written (with rs2 data) for stores; the WBSel multiplexer chooses ALU result or memory data for write-back.

Worked Example 3.1: Sub-word loads and stores

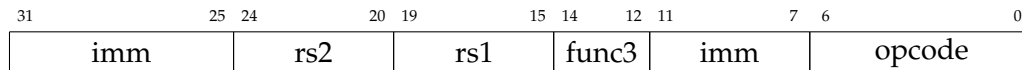
Memory is byte addressed, but our data memory returns a 32-bit aligned word. For lb a0, 6(x0) with $M[4..7] = 0x80\ 0x7F\ 0x12\ 0xF3$ at addresses 4,5,6,7 (little-endian word $0xF3127F80$):

- word address = $6 \gg 2 = 1$ (bytes 4–7), byte offset = $6 \& 3 = 2$;
- byte = $(\text{word} \gg (8 \cdot 2)) \& 0xFF = 0x12$;
- lb sign-extends: $0x00000012$; lb at address 7 gives $0xF3 \rightarrow 0xFFFFFFFFF3$, while lbu gives $0x000000F3$.

For sb the memory needs **byte-write enables**: sb t0, 5(x0) asserts enable bit 1 of the 4-bit

mask 0010 and places $t0$'s low byte on byte lane 1. A small “load extend” block after the memory and a “store align” block before it implement these using funct3 and the address's two low bits.

3.6 Conditional branches



Branches compare two integer registers for equality (BEQ/BNE) or signed magnitude (BLT/BGE) or unsigned magnitude (BLTU/BGEU). The 12-bit immediate encodes the target as a signed PC-relative offset in units of 2 bytes.

A branch therefore needs two things:

1. **compare the operands** — use the ALU (subtract and check the Zero/flags output) or a dedicated comparator;
2. **calculate the target address** — sign-extend the displacement, shift left (already done by the immediate encoding in RISC-V), and add to the PC (the PC of the branch, not PC+4).

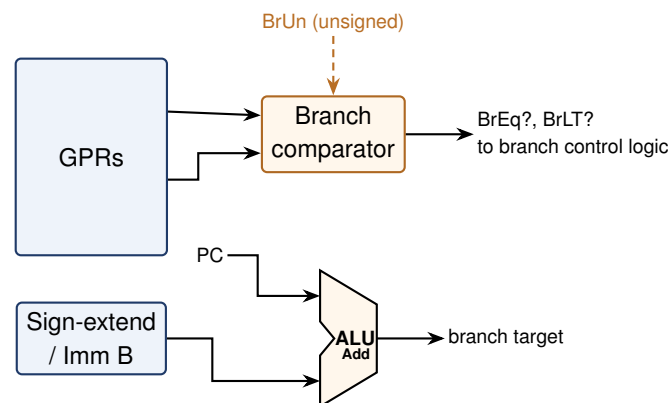


Figure 3.6: Branch datapath: a comparator evaluates the condition while a dedicated adder computes $PC + imm$. (The MIPS-style slide uses the ALU's Zero output plus an adder fed by a “shift left 2”.)

Using a separate comparator rather than the ALU lets RISC-V keep the ALU free for other uses and evaluate all six conditions with one subtractor-like circuit producing **BrEq** and **BrLT** (with **BrUn** selecting signed or unsigned). The control logic then decides: BEQ taken if **BrEq**; BNE if $\overline{\text{BrEq}}$; BLT/BLTU if **BrLT**; BGE/BGEU if $\overline{\text{BrLT}}$.

3.7 Jumps and upper immediates

- **JAL**: $rd \leftarrow PC + 4$; $PC \leftarrow PC + imm_J$. The target can be computed by the same PC-relative adder used for branches (with ImmSel = J).
- **JALR**: $rd \leftarrow PC + 4$; $PC \leftarrow (rs1 + imm_I) \& \sim 1$. The target is computed by the ALU (it is register-relative), so the ALU output must be able to reach the PC: a new PCSel input (“rind” on the slides).
- **LUI**: $rd \leftarrow imm_U$. The ALU can pass operand B through (or add it to x0).
- **AUIPC**: $rd \leftarrow PC + imm_U$. The ALU's A operand must be able to come from the PC: an

Op1Sel multiplexer (rs1 / PC).

The write-back multiplexer gains a third input, PC+4, for the link value. The lecture's control table uses $\text{PCSel} \in \{\text{pc}+4, \text{br}, \text{jabs}, \text{rind}\}$ and $\text{WBSel} \in \{\text{ALU}, \text{Mem}, \text{PC}\}$.

3.8 The complete single-cycle datapath

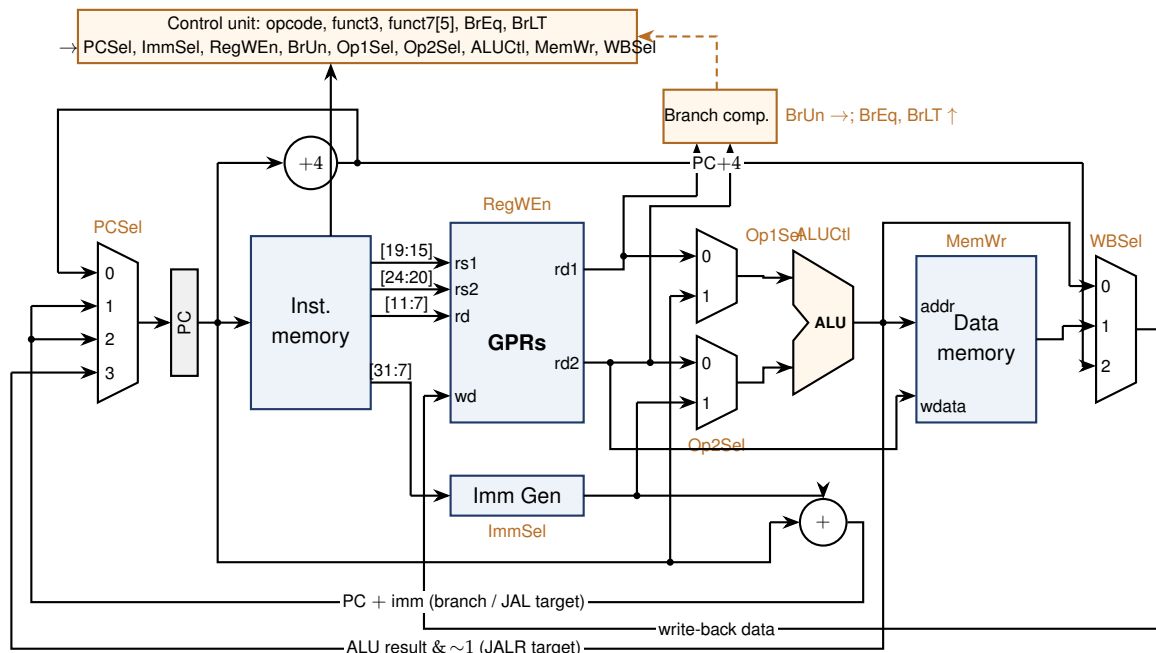


Figure 3.7: The complete single-cycle RV32I datapath. Orange labels mark the control points. PCSel: 0 = PC+4, 1 = branch target (taken), 2 = JAL target ("jabs", PC+imm_J, also from the target adder), 3 = JALR target from the ALU ("rind"). WBSel: 0 = ALU, 1 = memory, 2 = PC+4.

Figure 3.7 combines all the pieces. Counting the multiplexers reveals the control points: PCSel (4:1), Op1Sel (2:1), Op2Sel (2:1), WBSel (3:1), plus ImmSel inside the immediate generator, the ALU control, BrUn, RegWEn and MemWr.

3.9 Tracing instructions through the datapath

The best way to understand a datapath is to put numbers on its wires. Assume the initial state: $x1 = 0x00000010$, $x2 = 0x00001000$, $x5 = 0xFFFFFFFFC$ (-4), $x6 = 0x00000005$, and data memory $M[0x1008] = 0xDEADBEEF$.

Worked Example 3.2: add x7, x5, x6 at PC = 0x40

Word 0x006283B3.

Signal	Value
rs1, rs2, rd	5, 6, 7
rd1, rd2	0xFFFFFFFF, 0x00000005
Op1Sel / Op2Sel	rs1 / rs2
ALU control, ALU out	ADD, 0x00000001 (−4 + 5)
MemWr, WBSel, RegWEn	0, ALU, 1
PCSel, next PC	PC+4, 0x44

At the rising edge: $x7 \leftarrow 1$, $PC \leftarrow 0x44$.

Worked Example 3.3: `lw x9, 8(x2)` at $PC = 0x44$

Word `0x00812483`: I-type, $imm = 8$. $rd1 = 0x1000$; $Op2Sel = Imm$; $ALU\ out = 0x1008 =$ address; data memory read = `0xDEADBEEF`; $WBSel = Mem$; $RegWEn = 1$. The value on $rd2$ ($x8$, whatever it holds) reaches the data memory's write-data input but $MemWr = 0$, so it is ignored. Next $PC = 0x48$.

Note that the branch comparator and target adder also compute values (meaningless for a load); in a single-cycle design *all* hardware computes *every* cycle, and the control signals select which results matter.

Worked Example 3.4: `sw x6, -4(x1)` at $PC = 0x48$

Word: $imm = -4$ split as $imm[11:5] = 1111111$, $imm[4:0] = 11100$; encoding `0xFE60AE23`. $rd1 = 0x10$; $ALU: 0x10 + 0xFFFFFFFF = 0x0C$ (carry discarded); $wdata = rd2 = x6 = 5$; $MemWr = 1 \Rightarrow M[0x0C] \leftarrow 5$ at the clock edge; $RegWEn = 0$ ($WBSel$ is a don't-care).

Worked Example 3.5: `blt x5, x6, +12` at $PC = 0x4C$

$rd1 = -4$, $rd2 = 5$. Comparator ($BrUn = 0$): $BrLT = 1$, $BrEq = 0$. Target adder: $0x4C + 12 = 0x58$. Control: BLT and $BrLT \Rightarrow PCSel = br$; next $PC = 0x58$. Had the instruction been `bltu`, $BrUn = 1$ and $0xFFFFFFFF <_u 5$ is false: not taken, next $PC = 0x50$.

Worked Example 3.6: `jalr x1, 0(x5)` with $x5 = 0x00000201$

$ALU: 0x201 + 0 = 0x201$; bit 0 cleared $\Rightarrow$ target `0x200`; $PCSel = rind$; $WBSel = PC+4$, so $x1 \leftarrow PC+4$; $RegWEn = 1$. Function return is the special case `jalr x0, 0(x1)` (pseudo-instruction `ret`).

Worked Example 3.7: `auipc x10, 0x2` at $PC = 0x00003004$

$Op1Sel = PC$, $Op2Sel = Imm$ ($ImmSel = U$, $imm = 0x00002000$); $ALU\ out = 0x00005004 \rightarrow x10$. Combined with a following `addi x10, x10, 1012` this forms any PC-relative address — the basis of position-independent code (1a pseudo-op).

Advanced Insight: Where the hardware is idle

In a single-cycle datapath every block is used at most once per instruction; many are idle for most instructions (data memory for 70% of them, the branch adder for 80%). A multi-cycle design (Unit 3) reuses one ALU for $PC+4$, branch target and the operation, eliminating the

two extra adders — at the cost of more cycles per instruction. A pipelined design does the opposite: it keeps all the hardware and *fills* the idle blocks with other instructions.

Chapter summary

- Datapath components are added per instruction class: fetch (PC, instruction memory, +4), R-type (register file, ALU), I-type (immediate generator, Op2Sel), loads/stores (data memory, WBSel), branches (comparator, target adder, PCSel), jumps and U-type (PC to ALU via Op1Sel, PC+4 to write-back, ALU to PC).
- Each merge point needs a multiplexer and therefore a control signal.
- In a single cycle all blocks compute in parallel; control signals select which results update the state.

Exercises

E3.1 Trace `sub x7, x6, x5` and `sra x7, x5, x6` with the state above.

Answer: $5 - (-4) = 9$; $-4 \gg_a 5 = -1$ (`0xFFFFFFFF`).

E3.2 Which datapath elements are unused by `lui`?

Answer: data memory, branch comparator, branch adder (computing but ignored), register-file read ports.

E3.3 Modify the datapath to support a hypothetical `lwr rd, rs1, rs2` ($rd \leftarrow M[rs1 + rs2]$). What changes?

Answer: None structurally: Op2Sel = rs2 with WBSel = Mem; only the control table needs a new row.

E3.4 Modify the datapath to support `addm rd, rs1, imm(rs2)` ($rd \leftarrow rs1 + M[rs2 + imm]$). Why does this break the single-cycle “one ALU pass” structure?

Answer: It needs an address addition *before* memory and an addition *after* it: a second adder after data memory (or a second cycle).

E3.5 In `sw x6, -4(x1)` above, which value appears on the WBSel output and why is it harmless?

Answer: Depends on WBSel’s (don’t-care) setting; RegWEn = 0 so nothing is written.

4

The Control Unit

The datapath can perform every RV32I instruction, but only if each multiplexer, write enable and ALU operation is set correctly in every cycle. In a single-cycle processor these settings are a pure *combinational function* of the current instruction (and, for branches, of the comparison result): this is **hardwired control**. This chapter tabulates the control signals, derives the ALU control and the main decoder, implements them as logic, and shows how to reason about control faults.

4.1 Control signals

Table 4.1: Control signals of the single-cycle RV32I datapath.

Signal	Width	Values	Effect
PCSel	2	pc+4, br, jabs, rind	selects the next PC
ImmSel	3	I, S, B, U, J	selects the immediate format
RegWEn	1	0/1	writes the register file at the clock edge
BrUn	1	0/1	unsigned comparison in the branch comparator
Op1Sel	1	Reg / PC	ALU operand A: rs1 data or PC
Op2Sel	1	Reg / Imm	ALU operand B: rs2 data or immediate
ALUCtl	4	ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU, COPYB	ALU operation
MemWr	1	0/1	writes data memory at the clock edge
WBSel	2	ALU, Mem, PC+4	selects register write-back data

The control unit (Figure 4.1) is a block of combinational logic whose inputs are the opcode, funct3, bit 30 (funct7[5]) and the comparator outputs, and whose outputs are the signals above.

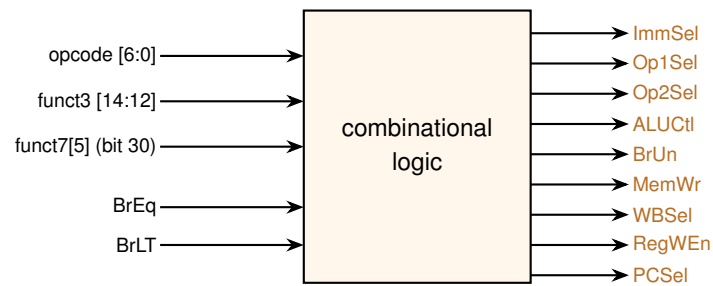


Figure 4.1: Hardwired control: a combinational function of the instruction fields and the branch comparison result.

4.2 The main control table

The lecture’s table (from the Berkeley CS152 single-cycle RISC-V design) lists, per instruction class, the values of ImmSel, Op2Sel, FuncSel (ALU function source), MemWr, RFWen, WBSel, WASel (write-address select) and PCSel:

Table 4.2: Hardwired control table from the lecture. * = don’t care. “Func” means the ALU function comes from funct fields; “Op” from the opcode; “+” forces addition.

Opcode	ImmSel	Op2Sel	FuncSel	MemWr	RFWen	WBSel	WASel	PCSel
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	IType ₁₂	Imm	Op	no	yes	ALU	rd	pc+4
LW	IType ₁₂	Imm	+	no	yes	Mem	rd	pc+4
SW	SType ₁₂	Imm	+	yes	no	*	*	pc+4
BEQ _{true}	SBType ₁₂	*	*	no	no	*	*	br
BEQ _{false}	SBType ₁₂	*	*	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	X1	jabs
JALR	*	*	*	no	yes	PC	rd	rind

Notes: Op2Sel = Reg / Imm; WBSel = ALU / Mem / PC; WASel = rd / X1 (an early draft of RISC-V had a separate J and an implicit-x1 JAL); PCSel = pc+4 / br / rind / jabs. In ratified RV32I, J is just JAL x0, and JAL writes its explicit rd field, so WASel is always rd and disappears. Table 4.3 gives the control for the datapath of Chapter 3.

Don’t-cares are design freedom

A * entry means the instruction does not care about the signal — typically because the value it selects is never stored (RegWEn = 0 and MemWr = 0). Assigning don’t-cares cleverly simplifies the logic. But two signals must *never* be don’t-care: **RegWEn** and **MemWr**, because they change architectural state. A wrong write enable corrupts the machine silently.

4.3 ALU control

4.3.1 Two-level decoding

A common technique (shown on the lecture’s “ALU Control” slide, in MIPS form) is **two-level decoding**: the main decoder produces a 2-bit **ALUOp** that says *what kind* of instruction this is,

Table 4.3: Control table for the RV32I single-cycle datapath (Figure 3.7).

Class	opcode	ImmSel	Op1	Op2	ALUCtl	MemWr	RegWEn	WBSel	PCSel
R-type	0110011	*	Reg	Reg	funct	0	1	ALU	pc+4
I-ALU	0010011	I	Reg	Imm	funct [†]	0	1	ALU	pc+4
Load	0000011	I	Reg	Imm	ADD	0	1	Mem	pc+4
Store	0100011	S	Reg	Imm	ADD	1	0	*	pc+4
Branch	1100011	B	*	*	*	0	0	*	br / pc+4 [‡]
JAL	1101111	J	*	*	*	0	1	PC+4	jabs
JALR	1100111	I	Reg	Imm	ADD	0	1	PC+4	rind
LUI	0110111	U	*	Imm	COPYB	0	1	ALU	pc+4
AUIPC	0010111	U	PC	Imm	ADD	0	1	ALU	pc+4

[†] funct3 selects the operation; bit 30 matters only for srai (not for addi: there is no subi). [‡] br if the condition selected by funct3 holds, else pc+4. BrUn = funct3[1] for branches.

and a small ALU-control decoder combines ALUOp with the function field to produce the 4-bit ALU control:

Table 4.4: ALU control in the lecture's MIPS-style formulation.

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The ALU is used for: load/store ($F = \text{add}$), branch ($F = \text{subtract}$), R-type (F depends on the funct field). Two-level decoding keeps the main decoder independent of the (many) funct codes and keeps the ALU decoder small and fast.

4.3.2 RV32I ALU control

For RV32I the ALU decoder's inputs are ALUOp (2 bits: 00 = add, 01 = branch/unused, 10 = R-type, 11 = I-ALU), funct3 (3 bits) and bit 30.

Worked Example 4.1: Why addi must ignore bit 30

addi x5, x5, -1024 has imm = -1024 = 110000000000. Its bit 30 (imm[10]) is 1! If the ALU decoder treated I-type exactly like R-type, this addi would perform a *subtraction*: $x5 - (-1024) = x5 + 1024$ instead of $x5 - 1024$ — a nasty, data-dependent bug that only appears for certain immediates. The rule: use bit 30 for funct3 = 000 only if ALUOp = R-type; use it for funct3 = 101 (shift right) in both formats, because srai/srli encode it explicitly in the upper immediate bits.

A convenient encoding makes the decoder almost trivial: let $\text{ALUCtl} = \{\text{bit30} \cdot q, \text{funct3}\}$ where q

Table 4.5: RV32I ALU decoder (for ALUOp = R-type or I-ALU).

funct3	bit30	(R / I)	ALUCtl	funct3	bit30	(R / I)	ALUCtl
000	0	both	ADD	100	x	both	XOR
000	1	R only	SUB	101	0	both	SRL
000	1	I	ADD (addi)	101	1	both	SRA
001	x	both	SLL	110	x	both	OR
010	x	both	SLT	111	x	both	AND
011	x	both	SLTU				

= 1 for R-type or funct3 = 101. Then ADD = 0000, SUB = 1000, SLL = 0001, SLT = 0010, SLTU = 0011, XOR = 0100, SRL = 0101, SRA = 1101, OR = 0110, AND = 0111; ADD is forced for loads, stores, JALR and AUIPC, and a separate COPYB for LUI (or LUI adds to x0 using Op1Sel = zero). The ALU itself then decodes these 4 bits internally.

4.4 Main decoder logic

With one-hot class signals decoded from the opcode, each control output is an OR of classes. Using opcode bits $o_6 \dots o_0$ (bits $o_1 o_0$ are always 11 for 32-bit instructions):

Class	Decode term (on $o_6 o_5 o_4 o_3 o_2$)
R-type (01100)	$\overline{o_6} o_5 o_4 \overline{o_3} \overline{o_2}$
I-ALU (00100)	$\overline{o_6} \overline{o_5} o_4 \overline{o_3} \overline{o_2}$
Load (00000)	$\overline{o_6} o_5 \overline{o_4} o_3 \overline{o_2}$
Store (01000)	$\overline{o_6} o_5 \overline{o_4} \overline{o_3} \overline{o_2}$
Branch (11000)	$o_6 o_5 \overline{o_4} o_3 \overline{o_2}$
JALR (11001)	$o_6 o_5 \overline{o_4} \overline{o_3} o_2$
JAL (11011)	$o_6 o_5 \overline{o_4} o_3 o_2$
LUI (01101)	$\overline{o_6} o_5 o_4 \overline{o_3} o_2$
AUIPC (00101)	$\overline{o_6} \overline{o_5} o_4 \overline{o_3} o_2$

Then, for example:

$$\begin{aligned}
 \text{RegWEn} &= R + I + \text{Load} + \text{JAL} + \text{JALR} + \text{LUI} + \text{AUIPC} \\
 &= \overline{\text{Store} + \text{Branch}} \quad (\text{for legal opcodes}), \\
 \text{MemWr} &= \text{Store}, \\
 \text{Op2Sel(Imm)} &= \overline{R\text{-type}} \quad (\text{don't-cares for branch/JAL set to Imm}), \\
 \text{Op1Sel(PC)} &= \text{AUIPC}, \\
 \text{WBSEL} &= \begin{cases} \text{Mem} & \text{Load} \\ \text{PC+4} & \text{JAL} + \text{JALR} \\ \text{ALU} & \text{otherwise} \end{cases} \\
 \text{PCSEL} &= \begin{cases} \text{rind} & \text{JALR} \\ \text{jabs} & \text{JAL} \\ \text{br} & \text{Branch} \cdot \text{taken} \\ \text{pc+4} & \text{otherwise.} \end{cases}
 \end{aligned}$$

4.4.1 Branch condition logic

With $\text{funct3} = f_2 f_1 f_0$ of a branch, $\text{BrUn} = f_1$ and

$$\text{taken} = \begin{cases} \text{BrEq} \oplus f_0 & f_2 = 0 \text{ (BEQ: } f_0 = 0, \text{ BNE: } f_0 = 1) \\ \text{BrLT} \oplus f_0 & f_2 = 1 \text{ (BLT/BLTU: } f_0 = 0, \text{ BGE/BGEU: } f_0 = 1) \end{cases}$$

i.e. $\text{taken} = (\overline{f_2} \text{BrEq} + f_2 \text{BrLT}) \oplus f_0$: one 2:1 MUX and one XOR. The RISC-V funct3 encoding was chosen to make this logic this small.

Worked Example 4.2: Evaluating three branches

Let $\text{rs1} = \text{0xFFFFFFFF0}$ (−16) and $\text{rs2} = \text{0x000000010}$ (16).

Instruction	funct3	BrUn	BrEq	BrLT	taken?
bne	001	0	0	1	$(\text{BrEq}) \oplus 1 = 1$
bge	101	0	0	1	$(\text{BrLT}) \oplus 1 = 0$
bltu	110	1	0	0	$(\text{BrLT}) \oplus 0 = 0$

For `bltu` the comparator works unsigned: $\text{0xFFFFFFFF0} >_u \text{0x10}$, so $\text{BrLT} = 0$ — the same register values give opposite answers for `blt` and `bltu`.

4.5 Implementing the control unit

Worked Example 4.3: ROM versus logic for the control unit

Naive ROM. Inputs: opcode (7), funct3 (3), bit 30 (1), BrEq , BrLT (2): 13 address bits. Outputs: PCSel 2 + ImmSel 3 + RegWEn 1 + BrUn 1 + Op1Sel 1 + Op2Sel 1 + ALUCtl 4 + MemWr 1 + WBSEL 2 = 16 bits. ROM size $2^{13} \times 16 = 131\,072$ bits — absurdly large for what is really a few dozen gates.

Decomposed. Only bits $o_6..o_2$ matter for the class (5 address bits), giving a 32×12 -bit main ROM (384 bits); the ALU decoder needs ALUOp, funct3, bit30 (6 bits: $64 \times 4 = 256$ bits); branch logic is 2 gates. Total ≈ 640 bits plus a few gates. **As logic:** the class decoder is nine 5-input ANDs; the outputs are ORs of 1–7 classes; the ALU decoder is about 15 gates. Total ≈ 60 –80 gates with a depth of 3–4 levels (≈ 100 ps) — negligible next to memory and ALU delays. This is the main argument for hardwired control in RISC processors.

Illegal instructions

A real control unit must also detect opcodes outside the table (and reserved funct combinations) and raise an *illegal-instruction exception* instead of executing garbage. Otherwise a jump into data memory may silently execute random bytes with RegWEn = 1. In RV32I, the all-zeros word `0x00000000` is deliberately defined as illegal, so that executing zero-filled memory traps immediately.

4.6 Control in action and control faults

Worked Example 4.4: Control signals for a short program

Instruction	ImmSel	Op1	Op2	ALUCtl	MemWr	RegWEn	WBSEL	PCSel
<code>addi t0, x0, 1</code>	I	Reg	Imm	ADD	0	1	ALU	pc+4
<code>slli t1, t0, 3</code>	I	Reg	Imm	SLL	0	1	ALU	pc+4
<code>sub t2, t1, t0</code>	*	Reg	Reg	SUB	0	1	ALU	pc+4
<code>sw t2, 0(sp)</code>	S	Reg	Imm	ADD	1	0	*	pc+4
<code>lw t3, 0(sp)</code>	I	Reg	Imm	ADD	0	1	Mem	pc+4
<code>beq t3, t2, L</code>	B	*	*	*	0	0	*	br (taken)
<code>jal ra, f</code>	J	*	*	*	0	1	PC+4	jabs
<code>jalr x0, 0(ra)</code>	I	Reg	Imm	ADD	0	0*	PC+4	rind

* RegWEn is 1 for JALR, but $rd = x0$, so the register file discards the write — the hard-wired zero saves a control special case.

Register values: $t0 = 1$, $t1 = 8$, $t2 = 7$; memory at sp gets 7; $t3 = 7$; the branch is taken.

Worked Example 4.5: Diagnosing a stuck-at control fault

A manufacturing defect forces **Op2Sel** to always select the register (stuck-at Reg). Which instructions still work?

- R-type: correct (they use Reg anyway).
- Branches, JAL: correct (they do not use the ALU result).
- I-ALU: compute $rs1 \text{ op } R[\text{imm}[4 : 0]]$ — wrong (the $rs2$ field position holds $\text{imm}[4:0]$, so the ALU sees a register selected by the low immediate bits).
- Loads/stores: address $rs1 + R[\text{rs2 field}]$ — wrong. (For a store the $rs2$ field is the data register, so the address becomes $rs1 + \text{data}$.)
- JALR: target $rs1 + R[\text{imm}[4 : 0]]$ — wrong unless that register holds the immediate.

A test program to detect the fault: `addi x1, x0, 5` followed by a check that `x1 == 5` (with the fault, `x1 = 0 + R[5]`). Systematically choosing such programs is *functional fault testing*; in industry, scan chains and ATPG test the gates directly.

Advanced Insight: Decoder verification and formal equivalence

Because the control unit is small and purely combinational, it is an ideal target for *formal* verification: a SAT solver can prove that the RTL decoder is equivalent to a golden model derived from the ISA specification for *all* 2^{32} instruction words, something simulation can never cover. Open-source flows (e.g. riscv-formal with SymbiYosys) check every instruction's architectural effect against a formal ISA model; decoder bugs such as the `addi/bit-30` example above are found in seconds.

Chapter summary

- Single-cycle control is a combinational function of opcode, funct3, bit 30 and the branch comparison: hardwired control.
- Two-level decoding: main decoder $\rightarrow$ ALUOp; ALU decoder (ALUOp, funct3, bit30) $\rightarrow$ ALUctl. Bit 30 must be ignored for `addi`.
- Branch outcome = $(\overline{f_2}\text{BrEq} + f_2\text{BrLT}) \oplus f_0$, $\text{BrUn} = f_1$.
- `RegWEn` and `MemWr` are never don't-cares; illegal opcodes must trap.
- The whole control unit is $\sim 60\text{--}80$ gates — tiny compared to the datapath.

Exercises

E4.1 Fill the control table row for `sltiu`, `sb` and `bgeu`.

E4.2 Derive a minimal logic expression for `WBSel[1]` (PC+4) and `WBSel[0]` (Mem) from opcode bits.

Answer: `WBSel[1]` = $o_6 o_5 o_2$ (JAL, JALR); `WBSel[0]` = $\overline{o_6} \overline{o_5} \overline{o_4} \overline{o_2}$ (load), using other opcodes as don't-cares.

E4.3 What goes wrong if `MemWr` is stuck at 1? Which instruction class is the only one unaffected *architecturally*?

Answer: Every instruction writes `rs2` data to address ALU result; only stores behave correctly.

E4.4 Suppose `RegWEn` is stuck at 1. Show that a `sw` would then also write the register file. Which register and value?

Answer: Register `imm[4:0]` (rd field position) gets the `WBSel` output (e.g. the address).

E4.5 Add the M-extension instruction `mul1`. What changes in the control (opcode, funct7 = 0000001)?

Answer: ALU decoder must examine `funct7[0]` (bit 25) for R-type and select a multiplier unit; main decoder unchanged.

5

Timing and Performance of the Single-Cycle Processor

A single-cycle processor has $CPI = 1$ by construction, so its performance is decided entirely by the clock period. This chapter finds the critical path of each instruction class, computes the clock period and the resulting execution time, and quantifies why the single-cycle design — although simple and correct — is fundamentally inefficient. These numbers motivate the multi-cycle and pipelined designs of Unit 3.

5.1 The clock-period constraint

The clock period must be sufficiently long for all of the following steps to be “completed”:

1. instruction fetch;
2. decode and register fetch;
3. ALU operation;
4. data fetch, if required;
5. register write-back setup time.

$$t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB} \quad (5.1)$$

At the rising edge of the following clock, the PC, the register file and the data memory are updated together. Because the same clock serves every instruction, t_C must cover the *slowest* instruction.

5.2 Component delays

5.3 Critical path of each instruction class

Worked Example 5.1: Path delays per instruction class

Load (lw): PC t_{cq} → IMem → RF read (the immediate generator and control finish earlier) → Op2Sel MUX → ALU → DMem → WBSel MUX → RF setup:

$$30 + 250 + 150 + 25 + 200 + 250 + 35 + 20 = 960 \text{ ps.}$$

Store: same up to DMem, which must have address and data stable before the edge (setup ≈ 20 ps): $30 + 250 + 150 + 25 + 200 + 20 = 675$ ps (the write itself happens at the edge).

R-type: $30 + 250 + 150 + 25 + 200 + 35 + 20 = 710$ ps.

Branch: PC → IMem → RF → comparator → branch logic (2 gates, 30 ps) → PCSel MUX → PC setup: $30 + 250 + 150 + 150 + 30 + 40 + 20 = 670$ ps. The target adder path ($30 + 250 +$

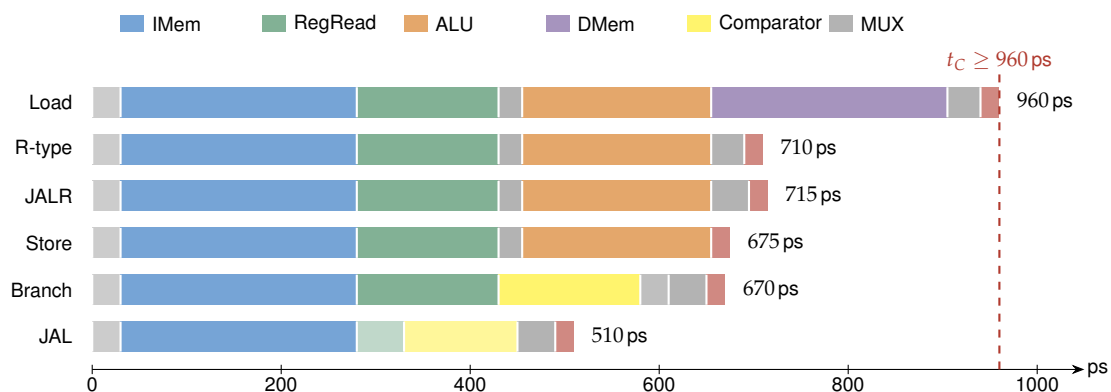
Table 5.1: Representative component delays used in this chapter.

Component	Delay (ps)	Notes
PC clock-to-Q	30	register
Instruction memory read	250	t_{IFetch}
Decoder / control unit	60	in parallel with register read
Register-file read	150	t_{RFetch}
Immediate generator	50	in parallel with register read
2:1 / 3:1 / 4:1 multiplexer	25 / 35 / 40	
ALU (32-bit, prefix adder)	200	t_{ALU}
Branch comparator	150	
PC+4 adder, branch-target adder	120	
Data memory read	250	t_{DMem}
Register-file setup	20	t_{RWB}
PC setup	20	

50 + 120 + 40 + 20 = 510 ps) is shorter.

JALR: 30 + 250 + 150 + 25 + 200 + 40 + 20 = 715 ps (ALU → PCSel → PC).

JAL: 30 + 250 + 50 + 120 + 40 + 20 = 510 ps for the PC; the link write (30 + 120 + 35 + 20 = 205 ps from PC+4) is short.

**Figure 5.1:** Critical-path composition for each instruction class. The load sets the clock period.

With the numbers above, $t_C \geq 960$ ps, i.e. $f_{\max} \approx 1.04$ GHz. Every instruction takes 960 ps, although an R-type needs only 710 ps and a JAL 510 ps.

5.4 Execution time

Worked Example 5.2: Program execution time on the single-cycle machine

A program executes $N = 10^9$ instructions with the lecture's RISC-V mix: 40% ALU, 30% load/store (say 20% loads, 10% stores), 20% branch/jump (17% branches, 3% JAL/JALR), 5% immediate (I-ALU), 5% other (treat as ALU).

Single-cycle: CPI = 1, $t_C = 960$ ps:

$$T_{\text{single}} = 10^9 \times 1 \times 960 \text{ ps} = 0.960 \text{ s.}$$

Ideal variable-length clock (each instruction takes exactly its own path delay — impossible with a synchronous clock, but a useful bound):

$$\begin{aligned} t_{\text{avg}} &= 0.50 \cdot 710 + 0.20 \cdot 960 + 0.10 \cdot 675 + 0.17 \cdot 670 + 0.03 \cdot 613 \\ &= 355 + 192 + 67.5 + 113.9 + 18.4 = 746.8 \text{ ps} \end{aligned}$$

(using the average of JAL and JALR, 613 ps, for jumps). $T_{\text{ideal}} = 0.747 \text{ s}$. The fixed clock wastes $1 - 746.8/960 = 22\%$ of the time.

Worked Example 5.3: Adding a slow instruction

Suppose we add a single-precision floating-point multiply whose FPU path takes 1 600 ps instead of the ALU's 200 ps, and it appears in only 2% of instructions. The new critical path is $30 + 250 + 150 + 25 + 1600 + 35 + 20 = 2110 \text{ ps}$: *every* instruction becomes $2.2\times$ slower, so the program takes 2.11 s instead of 0.96 s — to speed up 2% of the instructions with hardware support. With a variable clock the cost would be only $0.02 \times (2110 - 710) = 28 \text{ ps}$ per instruction on average.

This is the decisive argument against the single-cycle design: **the cycle time is dictated by the worst-case instruction**, violating the principle “make the common case fast”. A multi-cycle FPU (taking, say, 8 short cycles) or a pipelined one solves it.

5.5 Hardware utilization and energy

In a single-cycle machine every block is clocked and every combinational block switches every cycle, even when its result is discarded. Each block is also used at most once per cycle, so the design needs separate adders for PC+4 and the branch target, and separate instruction and data memories.

Worked Example 5.4: Utilization of the data memory and ALU

With the mix above, data memory is used by 30% of instructions; for the other 70% its access is pure waste (unless gated off). The ALU is used by ALU, I-ALU, load/store, JALR and AUIPC instructions — about 80%; but it is busy only for 200 ps of each 960 ps cycle, i.e. its *time* utilization is $0.8 \times 200/960 = 17\%$. The instruction memory is busy $250/960 = 26\%$ of the time. Pipelining (Unit 3) overlaps these activities so that each unit is busy almost all the time.

Worked Example 5.5: Dynamic energy per instruction

Dynamic energy per event $E = \alpha CV^2$. Assume at $V = 0.9 \text{ V}$: instruction-memory access 8 pJ, register-file read (two ports) 2 pJ, write 1.5 pJ, ALU operation 1.2 pJ, data-memory access 10 pJ, PC and adders 0.6 pJ, control 0.2 pJ.

Ungated (every block active every cycle): $8 + 2 + 1.5 + 1.2 + 10 + 0.6 + 0.2 = 23.5 \text{ pJ}$ per instruction. **Gated** (data memory enabled only for loads/stores, register write only when RegWEn): $8 + 2 + 0.85 \times 1.5 + 1.2 + 0.3 \times 10 + 0.6 + 0.2 = 16.3 \text{ pJ}$ — a 31% saving from two enable signals. At 10^9 instructions per second: 23.5 mW versus 16.3 mW. Memory accesses dominate energy, which is another reason caches (small, nearby memories) matter.

5.6 Summary of the single-cycle design

Advantages	Disadvantages
Simple, easy to understand and verify	Cycle time set by the slowest instruction
CPI = 1, no hazards, no stalls	Duplicated hardware (adders, memories)
Control is purely combinational	Long instructions (FP, multiply) are impractical
Good teaching and verification reference	Requires single-cycle (“magic”) memories

Advanced Insight: Where single-cycle cores are actually used

Single-cycle (or near-single-cycle) cores survive where frequency is low and simplicity is valuable: tiny soft cores on FPGAs (e.g. minimal RISC-V cores for control tasks), cores in deeply embedded sensors running at a few MHz, and *reference models* used to verify complex processors — an ISA-level single-cycle model (like the Spike simulator or a cycle-free RTL model) is the golden reference against which an out-of-order core is co-simulated instruction by instruction.

Chapter summary

- $t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$; the load is the critical instruction.
- With CPI = 1, $T = N \cdot t_C$: all instructions pay for the slowest one; in our example 22% of time is wasted versus an ideal variable clock.
- Adding one slow operation slows every instruction; utilization and energy efficiency are poor. Multi-cycle and pipelined organizations fix this.

Exercises

E5.1 Recompute t_C if the memories are replaced by caches with 150 ps access.

Answer: $30 + 150 + 150 + 25 + 200 + 150 + 35 + 20 = 760$ ps; $f = 1.32$ GHz.

E5.2 With the delays of Table 5.1, which component would you speed up first to shorten the clock? By how much does a $2\times$ faster ALU help?

Answer: Memories (500 ps of 960); a 100 ps ALU gives 860 ps, a 10% gain.

E5.3 The ALU is replaced by one that can also do a 32-bit multiply in 900 ps. Compute the new t_C and the slowdown for programs without multiplies.

Answer: 1660 ps; $1.73\times$ slower.

E5.4 Compute the energy per instruction for a program with 40% loads/stores using the gated model.

Answer: $8 + 2 + 1.275 + 1.2 + 4 + 0.8 = 17.3$ pJ (assuming 85% register writes).

E5.5 Why is the branch path (670 ps) shorter than the load path even though both include register reads and a comparison/ALU step?

6

From C to Machine Code: Running Factorial

This chapter follows one small program — the factorial of 5 — all the way from C source to the bits stored in instruction memory, and then executes it cycle by cycle on the single-cycle processor. Along the way we meet the compilation toolchain, the stack frame produced by an unoptimized compiler, hand-optimized RV32 assembly, instruction encoding, compressed instructions, ELF files and memory images, and the differences between bare-metal and Linux executables.

6.1 The C program

The lecture's program is a *bare-metal* program: there is no operating system, so the entry point is `_start` rather than `main`, and the program must never return (there is nothing to return to).

Listing 6.1: fact.c (bare-metal).

```
void _start() {
    volatile int num = 5;
    volatile long long result = 1;

    for (int i = 1; i <= num; i++)
        result *= i;

    while (1);    // Infinite loop to prevent exit
}
```

Why `volatile`? Without it, an optimizing compiler would compute $5! = 120$ at compile time and emit no loop at all (or, since `result` is never used, nothing). `volatile` forces every access to be performed in memory, which preserves the loop for study — and, as we will see, makes the unoptimized code very memory-intensive.

6.2 The compilation toolchain

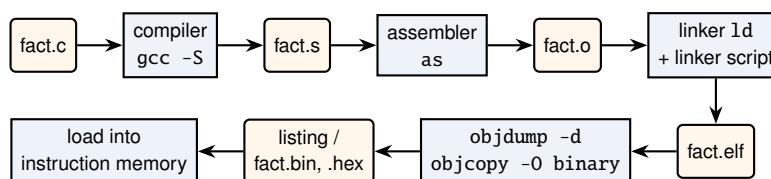


Figure 6.1: From C source to an image in instruction memory.

Listing 6.2: Building the bare-metal executable.

```
riscv64-unknown-elf-gcc -O0 -march=rv64gc -mabi=lp64d -nostdlib -S fact.c      # fact
.S
riscv64-unknown-elf-gcc -O0 -march=rv64gc -mabi=lp64d -nostdlib -o fact.elf fact.c
riscv64-unknown-elf-objdump -d fact.elf      # disassembly with addresses
riscv64-unknown-elf-objcopy -O binary fact.elf fact.bin    # raw memory image
```

`-nostdlib` omits the C runtime, so `_start` is the first code executed. For our 32-bit processor we would use `-march=rv32im -mabi=ilp32`.

6.3 Unoptimized compiler output

At `-O0` the compiler keeps every variable in the stack frame and loads/stores it at each use. The lecture's slide shows the generated RV64 assembly; its structure is:

Listing 6.3: Representative `-O0` output (RV64, structure as on the slide).

```
_start:
    addi    sp, sp, -32      # allocate 32-byte frame
    sd      s0, 24(sp)       # save caller's frame pointer
    addi    s0, sp, 32       # s0 = frame pointer (old sp)
    li      a5, 5
    sw      a5, -24(s0)      # num = 5          (int, 4 bytes)
    li      a5, 1
    sd      a5, -32(s0)      # result = 1       (long long, 8 bytes)
    li      a5, 1
    sw      a5, -20(s0)      # i = 1
    j       .L2
.L3:     lw      a5, -20(s0)  # a5 = i
    ld      a4, -32(s0)      # a4 = result
    mul     a5, a4, a5       # a5 = result * i
    sd      a5, -32(s0)      # result = a5
    lw      a5, -20(s0)
    addiw   a5, a5, 1        # i++ (32-bit add, sign-extended)
    sw      a5, -20(s0)
.L2:     lw      a5, -24(s0)  # a5 = num    (volatile: reload every time)
    sext.w  a4, a5
    lw      a5, -20(s0)      # a5 = i
    sext.w  a5, a5
    bge     a4, a5, .L3      # if num >= i, loop
.L4:     j       .L4         # while(1);
```

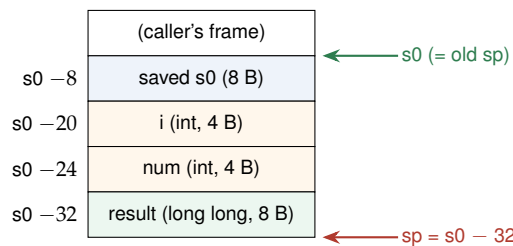


Figure 6.2: Stack frame of `_start` at `-O0` (addresses grow upward; the stack grows downward). There is padding: `i` at `-20` and `num` at `-24` are 4 bytes each, `result` at `-32` is 8 bytes, and the saved `s0` occupies `-8...-1`.

Reading the code.

- The prologue allocates a 32-byte frame (the RISC-V ABI keeps `sp` 16-byte aligned) and sets up a frame pointer `s0`.
- `int` is 32 bits even on RV64, hence `sw/lw` for `num` and `i`, `addiw` (32-bit add) and `sext.w` (sign-extend word, a pseudo-instruction for `addiw rd, rs, 0`). `long long` is 64 bits: `sd/ld`.
- The loop is compiled “test at the bottom”: `j .L2` jumps to the condition first. The condition `i <= num` becomes `bge num, i`.
- `mul` requires the M extension (part of the “G” in `rv64gc`).

Worked Example 6.1: Dynamic instruction count of the -O0 code

Prologue up to and including `j .L2`: 10 instructions. Each iteration executes the 7-instruction body `.L3` plus the 5-instruction test `.L2`: 12 instructions. The test runs 6 times (5 times true, once false), the body 5 times:

$$N = 10 + 5 \times 7 + 6 \times 5 = 75 \text{ instructions (before the final infinite loop).}$$

Memory operations: prologue 4 (`sd, sw, sd, sw`); body 5 per iteration; test 2 per evaluation: $4 + 25 + 12 = 41$ loads/stores — 55% of all instructions, versus $\approx 30\%$ in typical code. That is the cost of volatile and `-O0`.

6.4 Hand-optimized RV32IM version

Keeping the variables in registers (as an optimizing compiler would without `volatile`) and using a 32-bit result ($5! = 120$ fits easily), the program for our RV32 processor is:

Listing 6.4: `fact32.s`: hand-written RV32IM factorial.

```
.text
.globl _start
_start: addi a0, x0, 5      # num = 5
        addi a1, x0, 1      # result = 1
        addi t0, x0, 1      # i = 1
loop:   blt a0, t0, done    # if (num < i) exit loop
        mul a1, a1, t0      # result *= i
        addi t0, t0, 1      # i++
        jal x0, loop        # j loop
done:   sw a1, 256(x0)      # store result at address 0x100
end:    jal x0, end         # while(1);
```

6.4.1 Assembling by hand

Worked Example 6.2: Encoding the branch and the backward jump

`blt a0, t0, done` at `0x0C`; `done` is at `0x1C`, so the offset is $+16 = 00\,000000\,1000_2$. B-type fields: `imm[12]=0`, `imm[10:5]=000000`, `rs2=t0=00101`, `rs1=a0=01010`, `funct3=100` (BLT), `imm[4:1]=1000`, `imm[11]=0`, opcode `1100011`:

$$0\,000000\,00101\,01010\,100\,1000\,0\,1100011 = 0x00554863.$$

Table 6.1: Machine code of fact32.s (instruction memory starts at address 0).

Address	Word	Instruction	Fields
0x00	00500513	addi a0,x0,5	imm=5 rs1=0 f3=000 rd=10 op=0010011
0x04	00100593	addi a1,x0,1	imm=1 rs1=0 f3=000 rd=11 op=0010011
0x08	00100293	addi t0,x0,1	imm=1 rs1=0 f3=000 rd=5 op=0010011
0x0C	00554863	blt a0,t0,+16	imm=+16 rs2=5 rs1=10 f3=100 op=1100011
0x10	025585B3	mul a1,a1,t0	f7=0000001 rs2=5 rs1=11 f3=000 rd=11 op=0110011
0x14	00128293	addi t0,t0,1	imm=1 rs1=5 f3=000 rd=5 op=0010011
0x18	FF5FF06F	jal x0,-12	imm=-12 rd=0 op=1101111
0x1C	10B02023	sw a1,256(x0)	imm=256 rs2=11 rs1=0 f3=010 op=0100011
0x20	0000006F	jal x0,0	imm=0 rd=0 op=1101111

jal x0, loop at 0x18; loop is at 0x0C: offset -12 . As a 21-bit two's-complement number, $-12 = 1\ 11111111\ 1\ 11111110\ 10\ 0$ (imm[20], imm[19:12], imm[11], imm[10:1], imm[0]). The J-format order is imm[20] | imm[10:1] | imm[11] | imm[19:12]:

$$1\ 11111110\ 1\ 11111111\ 00000\ 1101111 = 0xFF5FF06F.$$

mul is an R-type instruction with funct7 = 0000001 — the bit that distinguishes the M extension (bit 25). Our base single-cycle datapath needs a multiplier added to the ALU (and ALU-decoder support for bit 25) to run it.

6.4.2 The memory image

Instructions are stored little-endian. The first three words in memory, byte by byte:

address	00	01	02	03	04	05	06	07	08	09	0A	0B
byte	13	05	50	00	93	05	10	00	93	02	10	00

A hexdump of the binary (xxd fact.bin) shows exactly these bytes: 1305 5000 9305 1000 9302 1000 6348 5500 The lecture's "actual memory block" slide is a hexdump of the full ELF file — the program bytes appear inside the .text section, surrounded by headers, symbol tables and strings.

6.5 Executing factorial on the single-cycle processor

Worked Example 6.3: Cycle-by-cycle trace

Initial state: all registers 0, PC = 0.

Cycle	PC	Instruction	Effect at the end of the cycle
1	0x00	addi a0,x0,5	a0 = 5
2	0x04	addi a1,x0,1	a1 = 1
3	0x08	addi t0,x0,1	t0 = 1

4	0x0C	blt a0,t0,done	5 < 1? no: PC ← 0x10
5	0x10	mul a1,a1,t0	a1 = 1
6	0x14	addi t0,t0,1	t0 = 2
7	0x18	jal x0,loop	PC ← 0x0C
8–11		(iteration 2)	a1 = 2, t0 = 3
12–15		(iteration 3)	a1 = 6, t0 = 4
16–19		(iteration 4)	a1 = 24, t0 = 5
20–23		(iteration 5)	a1 = 120, t0 = 6
24	0x0C	blt a0,t0,done	5 < 6: taken, PC ← 0x1C
25	0x1C	sw a1,256(x0)	M[0x100] = 120
26–	0x20	jal x0,end	PC ← 0x20 (forever)

25 instructions complete the computation (CPI = 1, so 25 cycles). With $t_C = 960$ ps: $25 \times 0.96 = 24.0$ ns. The -00 RV64 version needs 75 instructions (72 ns on an equivalent single-cycle RV64 core) — the compiler optimization (register allocation) is worth $3\times$, more than any single microarchitectural trick in this unit.

Worked Example 6.4: Control signals during one loop iteration

Instr.	ImmSel	Op1	Op2	ALUCtl	BrUn	MemWr	RegWEn	WBSel / PCSel
blt	B	*	*	*	0	0	0	* / pc+4 (not taken)
mul	*	Reg	Reg	MUL	*	0	1	ALU / pc+4
addi	I	Reg	Imm	ADD	*	0	1	ALU / pc+4
jal	J	*	*	*	*	0	1*	PC+4 / jabs

* rd = x0, so the write of PC+4 is discarded.

Worked Example 6.5: A loop without mul

If the processor has no M extension, multiplication by the small integer i can be done by repeated addition or shift-and-add. A shift-and-add multiply of a 32-bit value by an n -bit multiplier takes, per multiplier bit: `andi`, `beq`, `add`, `slli`, `srli`, branch-back ≈ 6 instructions. For $i \leq 5$ (3 bits) that is about 18 instructions instead of one `mul`: the loop grows from 4 to ≈ 21 instructions and the total from 25 to about 110 — a $4.4\times$ slowdown. This is the quantitative case for the M extension in any core that multiplies often.

6.6 Compressed instructions in the real binary

The lecture's `objdump` listing (compiled for `rv64gc`) shows many 16-bit words such as `1101`, `ec22`, `1000`, `4795`. These are **C extension** instructions: each expands to one 32-bit base instruction.

Worked Example 6.6: Decoding compressed instructions

0x1101 = `000 1 00010 00000 01`: `op = 01`, `funct3 = 000` $\Rightarrow$ `c.addi`; `rd = 00010` = `sp`; `imm = {bit12, bits[6:2]}` = `1 000002` = `-32`. Expansion: `addi sp, sp, -32`.

0xEC22 = 111 011000 01000 10: op = 10, funct3 = 111 $\Rightarrow$ c.sdsp (RV64); offset bits [12:7] = 011000 give uimm[5:3] = 011, uimm[8:6] = 000 $\Rightarrow$ 24; rs2 = 01000 = s0. Expansion: sd s0, 24(sp).
0x1000 = 000 100000000 000 00: c.addi4spn with nzuimm = 32, rd' = 000 $\rightarrow$ x8 (s0): addi s0, sp, 32.
0x4795 = 010 0 01111 00101 01: c.li rd = 01111 = a5, imm = 5: li a5, 5 (i.e. addi a5, x0, 5).

Worked Example 6.7: Code-size saving

The lecture's listing starts at **0x100b0** and the first nine instructions occupy **0x100b0–0x100c7**: 24 bytes, versus 36 bytes uncompressed. Across typical programs, RVC reduces static code size by 25–30%, which improves instruction-cache hit rates (Unit 4) and reduces instruction-memory energy. The cost: a decompression stage in the front end and instructions that are no longer 4-byte aligned (hence branch offsets in units of 2 bytes).

6.7 ELF files, bare metal and Linux

An ELF (Executable and Linkable Format) file contains a header (machine EM_RISCV, entry point), program headers telling the loader which segments to place where in memory, and sections: .text (code), .rodata (constants), .data (initialized globals), .bss (zero-initialized globals), .symtab/.strtab (symbols), .comment (compiler version) and .riscv.attributes (the ISA string, e.g. rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0).

Table 6.3: Bare-metal versus Linux user-space builds of the same program.

	Bare metal (riscv64-unknown-elf)	Linux (riscv64-unknown-linux-gnu)
Entry	_start is the program	_start in crt1 sets up argc/argv, calls main
Load address	chosen by the linker script (e.g. 0x10000 or 0)	position-independent, chosen by the OS loader
Libraries	none (-nostdlib) or newlib	glibc, dynamic linking via PLT/GOT
Exit	infinite loop or ebreak	ecall with the exit system call
Privilege	machine mode, owns the hardware	user mode under supervisor-mode kernel

The lecture's Linux listing shows code at small addresses (**0x30a...**) with branch targets such as `<_PROCEDURE_LINKAGE_TABLE_+0x8a>`: the program was linked as a position-independent executable, and objdump labels addresses relative to the nearest symbol it knows, which happens to be the PLT. The instruction sequence of the loop itself is identical.

Advanced Insight: Running the program on a simulator and on your processor

spike pk fact_linux runs the Linux-style binary on the RISC-V ISA simulator with a proxy kernel; spike -l -isa=rv32im fact32.elf prints a commit log of every instruction, PC and register write — exactly the trace table above. For a processor in RTL, the standard flow converts the ELF into a \$readmemh hex file (objcopy -O verilog), loads it into the instruction-memory model in the testbench, runs the simulation, and compares the processor's commit log with Spike's line by line. Any mismatch pinpoints the first divergent

instruction. Chapter 7 builds this flow.

Chapter summary

- The toolchain: gcc → as → ld (linker script) → ELF → objcopy → memory image; -nostdlib bare-metal programs start at `_start` and never return.
- `-O0` keeps variables on the stack: 75 instructions and 55% memory operations for 5!; register allocation reduces it to 25.
- Hand-encoding exercises every format: B and J offsets are PC-relative and scrambled; `mul` is R-type with `funct7 = 0000001`.
- The single-cycle CPU executes the 25 instructions in 25 cycles (24 ns at 1.04 GHz).
- Compressed (C) instructions halve common instruction sizes; ELF files wrap code in headers, sections and symbols.

Exercises

E6.1 Modify `fact32.s` to compute $n!$ for n in `a0` given at entry and return the result in `a0` with `jalr x0, 0(ra)`. How many instructions execute for $n = 10$?

Answer: With the same loop: $2 + 4n + 1 + 1 = 44$ for $n = 10$ (init 2, $4n$ loop, final `blt`, `ret`).

E6.2 $13!$ overflows 32 bits. What value does the RV32 program produce for $n = 13$?

Answer: $13! = 6\,227\,020\,800$; $\text{mod } 2^{32} = 1\,932\,053\,504$.

E6.3 Encode `bge a4, a5, .L3` if `.L3` is 48 bytes before the branch.

Answer: `imm = -48: 0xFCF758E3`.

E6.4 Decode the compressed instructions `0x4785` and `0x6105`.

Answer: `c.li a5,1; c.addi16sp sp,32 (= addi sp,sp,32)`.

E6.5 Count loads, stores and ALU operations in Listing 6.3 and compute the CPI on a multi-cycle machine with $\text{CPI}_{ld} = 5$, $\text{CPI}_{st} = 4$, $\text{CPI}_{ALU} = 4$, $\text{CPI}_{br} = 3$, $\text{CPI}_j = 3$.

7

RTL Implementation and Verification

A datapath on paper becomes a processor when it is written in a hardware description language, simulated, verified and synthesized. This chapter presents a complete, compact Verilog model of the single-cycle processor of Chapters 3–4 (plus the mul instruction needed by the factorial program), a testbench that runs factorial, a self-checking instruction test, and the verification and FPGA-implementation practices used for real RISC-V cores. Every listing in this chapter was simulated with Icarus Verilog; the outputs shown are the actual simulation results.

7.1 From block diagram to modules

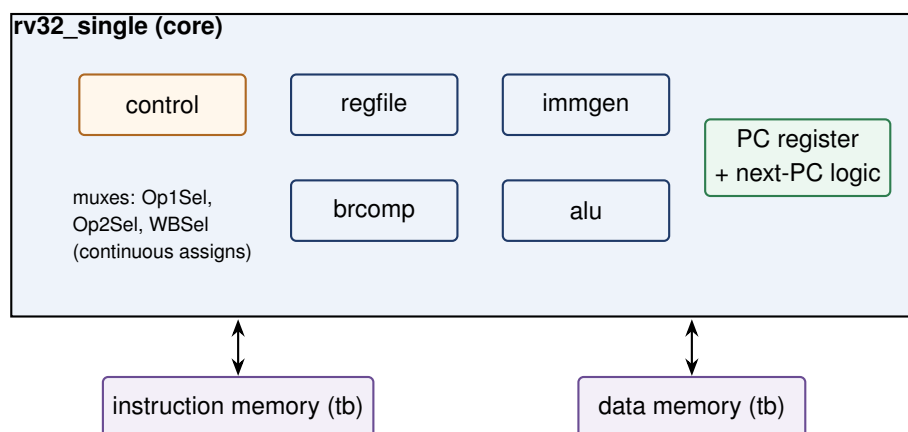


Figure 7.1: Module hierarchy. Memories live in the testbench so that the core is reusable with any memory system.

Design rules followed by the code:

- Combinational blocks use always @* or assign; every output is assigned on every path (default values at the top of always blocks) so that synthesis does not infer latches.
- State changes only in always @(posedge clk) with non-blocking assignments (<=).
- The register file reads combinatorially and writes at the clock edge; x0 is not stored and reads as zero.
- The core’s memory interface is a plain combinational-read, edge-write port — the “magic memory” of Chapter 1.

7.2 The Verilog model

Listing 7.1: rv32_single.v — the complete single-cycle core (about 110 lines).

```

1 // =====
2 // rv32_single.v -- single-cycle RV32I(+mul) processor, VL326 Unit 2
3 // =====
4 module regfile(input clk, we, input [4:0] ra1, ra2, wa,
5               input [31:0] wd, output [31:0] rd1, rd2);
6     reg [31:0] r [1:31];
7     assign rd1 = (ra1 == 0) ? 32'b0 : r[ra1];      // x0 reads as zero
8     assign rd2 = (ra2 == 0) ? 32'b0 : r[ra2];
9     always @(posedge clk) if (we && wa != 0) r[wa] <= wd;
10 endmodule
11
12 module immgen(input [31:0] i, input [2:0] sel, output reg [31:0] imm);
13     localparam I=0, S=1, B=2, U=3, J=4;
14     always @* case (sel)
15         I: imm = {{21{i[31]}}, i[30:20]};
16         S: imm = {{21{i[31]}}, i[30:25], i[11:7]};
17         B: imm = {{20{i[31]}}, i[7], i[30:25], i[11:8], 1'b0};
18         U: imm = {i[31:12], 12'b0};
19         J: imm = {{12{i[31]}}, i[19:12], i[20], i[30:21], 1'b0};
20         default: imm = 32'b0;
21     endcase
22 endmodule
23
24 module alu(input [31:0] a, b, input [4:0] op, output reg [31:0] y);
25     // op = {funct7[0] (mul), funct7[5] qualified, funct3}; 5'b11111 = copy b
26     always @* case (op)
27         5'b00000: y = a + b;                      // ADD
28         5'b01000: y = a - b;                      // SUB
29         5'b00001: y = a << b[4:0];                // SLL
30         5'b00010: y = {31'b0, $signed(a) < $signed(b)}; // SLT
31         5'b00011: y = {31'b0, a < b};              // SLTU
32         5'b00100: y = a ^ b;                      // XOR
33         5'b00101: y = a >> b[4:0];                // SRL
34         5'b01101: y = $signed(a) >>> b[4:0];       // SRA
35         5'b00110: y = a | b;                      // OR
36         5'b00111: y = a & b;                      // AND
37         5'b10000: y = a * b;                      // MUL (M extension)
38         5'b11111: y = b;                          // COPYB (LUI)
39         default: y = 32'bx;
40     endcase
41 endmodule
42
43 module brcomp(input [31:0] a, b, input un, output eq, lt);
44     assign eq = (a == b);
45     assign lt = un ? (a < b) : ($signed(a) < $signed(b));
46 endmodule
47
48 module control(input [31:0] inst, input breq, brlt,
49               output reg [2:0] immisel, output reg oplpc, op2imm, memwr, regwen,
50               output reg [1:0] wbsel, pcsel, output reg [4:0] aluop, output brun);
51     wire [6:0] opc = inst[6:0]; wire [2:0] f3 = inst[14:12];
52     wire taken = ((f3[2] ? brlt : breq) ^ f3[0]);
53     assign brun = f3[1];
54     localparam PC4=0, BR=1, JABS=2, RIND=3, WALU=0, WMEM=1, WPC=2;
55     always @* begin
56         {immisel, oplpc, op2imm, memwr, regwen, wbsel, pcsel} = 0; aluop = 5'b00000;
57         case (opc)
58             7'b0110011: begin regwen=1;                // R-type
59                 aluop = {inst[25], inst[30], f3}; end
60             7'b0010011: begin regwen=1; op2imm=1;      // I-ALU
61                 aluop = {1'b0, (f3==3'b101) & inst[30], f3}; end
62             7'b0000011: begin regwen=1; op2imm=1; wbsel=WMEM; end // load (lw)
63             7'b0100011: begin immisel=1; op2imm=1; memwr=1; end // store (sw)
64             7'b1100011: begin immisel=2; pcsel = taken ? BR : PC4; end
65             7'b1101111: begin immisel=4; regwen=1; wbsel=WPC; pcsel=JABS; end
66             7'b1100111: begin op2imm=1; regwen=1; wbsel=WPC; pcsel=RIND; end
67             7'b0110111: begin immisel=3; op2imm=1; regwen=1; aluop=5'b11111; end
68             7'b0010111: begin immisel=3; oplpc=1; op2imm=1; regwen=1; end
69             default: ; // illegal: NOP here
70         endcase
71     end

```

```

71   end
72   endmodule
73
74   module rv32_single(input clk, rst,
75       output [31:0] imem_addr, input [31:0] inst,
76       output [31:0] dmem_addr, dmem_wdata, output dmem_we, input [31:0] dmem_rdata);
77   reg [31:0] pc;
78   wire [2:0] imm_sel; wire op1pc, op2imm, memwr, regwen, brun, breq, brlt;
79   wire [1:0] wbsel, ptsel; wire [4:0] aluop;
80   wire [31:0] rd1, rd2, imm, alu_y, wd;
81   wire [31:0] pc4 = pc + 4, pcimm = pc + imm;
82
83   control cu(inst, breq, brlt, imm_sel, op1pc, op2imm, memwr, regwen, wbsel, ptsel, aluop, brun);
84   regfile rf(clk, regwen, inst[19:15], inst[24:20], inst[11:7], wd, rd1, rd2);
85   immgen ig(inst, imm_sel, imm);
86   brcomp bc(rd1, rd2, brun, breq, brlt);
87   alu al(op1pc ? pc : rd1, op2imm ? imm : rd2, aluop, alu_y);
88
89   assign wd = (wbsel == 2) ? pc4 : (wbsel == 1) ? dmem_rdata : alu_y;
90   assign imem_addr = pc;
91   assign dmem_addr = alu_y; assign dmem_wdata = rd2; assign dmem_we = memwr;
92
93   always @(posedge clk)
94       if (rst) pc <= 0;
95       else case (ptsel)
96           0: pc <= pc4;    1, 2: pc <= pcimm;    3: pc <= alu_y & ~32'b1;
97       endcase
98   endmodule

```

7.2.1 Walking through the code

regfile. The array `r[1:31]` has no entry 0. The two read ports are continuous assignments with an explicit zero test, so reading `x0` costs one comparator and a MUX per port. The write is guarded by `wa != 0`.

immgen. A direct transcription of Table 2.4: each format is one concatenation. The `{21{i[31]}}` replication is the sign extension.

alu. The 5-bit operation code is `{funct7[0], qualified funct7[5], funct3}` — the “almost trivial” encoding of Chapter 4. `$signed()` selects signed comparison and arithmetic shift. `a * b` infers a 32×32 multiplier whose low 32 bits are the RV32M `mul` result. The default `32'bx` tells synthesis the remaining codes are don’t-cares and makes simulation flag accidental use.

control. One case on the opcode sets exactly the non-zero entries of Table 4.3; everything else defaults to 0. The branch decision is the one-line expression `(f3[2] ? brlt : breq) ^ f3[0]`. Note the I-ALU ALU code: bit 30 is passed only when `funct3 == 101` (shifts) — the `addi/bit-30` trap of Chapter 4 avoided.

rv32_single. Two dedicated adders compute `pc4` and `pcimm`; the three data multiplexers are conditional expressions; the PC register implements the `PCSel` multiplexer in its case. Branch (BR) and JAL (JABS) both use `pcimm` because RISC-V’s JAL is PC-relative.

Blocking versus non-blocking assignments

Writing `pc = pc4;` (blocking) instead of `pc <= pc4;` inside the clocked block happens to work in this tiny design, but in any design where two clocked blocks read each other's registers, blocking assignments create simulation races: the result depends on which always block the simulator happens to run first, and the simulated behaviour may differ from the synthesized hardware. Rule: `<=` in clocked blocks, `=` in combinational blocks, never mix them in one block.

7.3 Running factorial

Listing 7.2: `tb.v` — testbench running the factorial program.

```

1  `timescale 1ns/1ps
2  module tb;
3      reg clk = 0, rst = 1; always #0.5 clk = ~clk;          // 1 GHz
4      reg [31:0] imem [0:255]; reg [31:0] dmem [0:255];
5      wire [31:0] ia, da, dw, dr; wire we;
6      rv32_single cpu(clk, rst, ia, imem[ia[9:2]], da, dw, we, dr);
7      assign dr = dmem[da[9:2]];
8      always @(posedge clk) if (we) dmem[da[9:2]] <= dw;
9      initial begin
10         $readmemh("fact32.hex", imem);
11         #1.2 rst = 0;
12         repeat (40) @(posedge clk);
13         $display("M[0x100] = %0d (expected 120)", dmem[64]);
14         $display("a1 = %0d, t0 = %0d, PC = %h", cpu.rf.r[11], cpu.rf.r[5], cpu.pc);
15         $finish;
16     end
17     integer n = 0; // print the instruction executing in each cycle (state before commit)
18     always @(posedge clk) if (!rst) begin
19         n = n + 1;
20         if (n <= 26) $display("cycle %2d PC=%h inst=%h a1=%0d t0=%0d",
21                               n, cpu.pc, imem[ia[9:2]], cpu.rf.r[11], cpu.rf.r[5]);
22     end
23 endmodule

```

The file `fact32.hex` contains the nine words of Table 6.1, one per line. Compiling and running:

```
$ iverilog -g2012 -o sim tb.v rv32_single.v && vvp sim
```

produces (abridged):

```

cycle 1  PC=00000000 inst=00500513 a1=x t0=x
cycle 2  PC=00000004 inst=00100593 a1=x t0=x
cycle 3  PC=00000008 inst=00100293 a1=1 t0=x
cycle 4  PC=0000000c inst=00554863 a1=1 t0=1
cycle 5  PC=00000010 inst=025585b3 a1=1 t0=1
...
cycle 21 PC=00000010 inst=025585b3 a1=24 t0=5
cycle 22 PC=00000014 inst=00128293 a1=120 t0=5
cycle 23 PC=00000018 inst=ff5ff06f a1=120 t0=6
cycle 24 PC=0000000c inst=00554863 a1=120 t0=6
cycle 25 PC=0000001c inst=10b02023 a1=120 t0=6
cycle 26 PC=00000020 inst=0000006f a1=120 t0=6
M[0x100] = 120 (expected 120)

```

Each line is printed at the rising edge that ends the cycle, so the register values are those seen by the instruction *during* the cycle, i.e. before it commits — which is why `a1` becomes 120 one line after the `mul` of cycle 21. The trace matches the hand trace of Chapter 6 exactly: 25 instructions,

and the result 120 stored at address `0x100`. The `x` values in the first cycles are uninitialized registers — real hardware would show arbitrary values; the program never reads them before writing.

7.4 A self-checking instruction test

Running one program proves little. A **directed test** exercises each instruction class and checks the results automatically.

Table 7.1: Directed test program (`test.hex`) and the expected architectural results.

PC	Instruction	Expected effect
00	<code>addi t0, x0, -4</code>	<code>t0 = FFFFFFFC</code>
04	<code>addi t1, x0, 5</code>	<code>t1 = 5</code>
08	<code>add t2, t0, t1</code>	<code>t2 = 1</code>
0C	<code>sub t3, t1, t0</code>	<code>t3 = 9</code>
10	<code>sra t4, t0, t1</code>	<code>t4 = FFFFFFFF</code>
14	<code>srl t5, t0, t1</code>	<code>t5 = 07FFFFFF</code>
18	<code>slt t6, t0, t1</code>	<code>t6 = 1</code> ($-4 < 5$)
1C	<code>sltu s0, t0, t1</code>	<code>s0 = 0</code> (<code>FFFFFFFC</code> $\not<_u$ 5)
20	<code>lui s1, 0x12345</code>	<code>s1 = 12345000</code>
24	<code>addi s1, s1, 0x678</code>	<code>s1 = 12345678</code>
28	<code>auipc s2, 1</code>	<code>s2 = 00001028</code>
2C	<code>sw s1, 64(x0)</code>	<code>M[64] = 12345678</code>
30	<code>lw s3, 64(x0)</code>	<code>s3 = 12345678</code>
34	<code>beq s3, s1, +8</code>	taken $\rightarrow$ 3C
38	<code>addi s4, x0, 99</code>	must be skipped
3C	<code>bgeu t0, t1, +8</code>	taken (unsigned) $\rightarrow$ 44
40	<code>addi s5, x0, 99</code>	must be skipped
44	<code>jal ra, +12</code>	<code>ra = 48, $\rightarrow$ 50</code>
48	<code>addi s6, x0, 7</code>	<code>s6 = 7</code> (after return)
4C	<code>jal x0, 0</code>	halt
50	<code>xori s7, x0, 0xFF</code>	<code>s7 = 000000FF</code>
54	<code>jalr x0, 0(ra)</code>	return $\rightarrow$ 48

The testbench (`tb_test.v`, supplied with the sources) runs 30 cycles and calls a `check(reg, expected)` task for every destination register, plus a check that the two “must be skipped” instructions did not execute. Result of the actual run: 0 errors.

Worked Example 7.1: How a bug shows up

Suppose the immediate generator’s B-format line were mistyped as `{{20{i[31]}}}, i[7], i[30:25], i[11:8], 1'b1` (bit 0 forced to 1). Then `beq s3, s1, +8` would jump to `0x3D`; the instruction memory indexed by `ia[9:2]` would fetch word `0x3C` anyway (the low bits are ignored) but the PC would now be odd, the following `bgeu` would compute a target of `0x45`, the `jal` would link `0x49`... The directed test reports `ra = 00000049`, expected `00000048` — a single failing check that points straight at PC arithmetic. A co-simulation against an ISS (below) would flag the very first divergent PC.

7.5 Verification of real processors

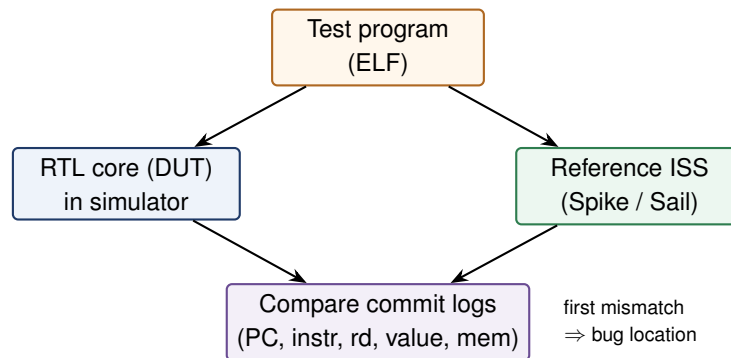


Figure 7.2: Lock-step co-simulation: the DUT and a reference instruction-set simulator execute the same program; their per-instruction commit records are compared.

- **Directed tests:** hand-written programs for each instruction (riscv-tests rv32ui-p-*: one test per instruction, each self-checking and ending with a pass/fail code written to a “tohost” address).
- **Compliance / architecture tests:** the RISC-V International riscv-arch-test suite produces a memory “signature” that must match the reference.
- **Random instruction generation:** generators such as RISC-V DV produce millions of constrained-random programs, compared against the reference ISS by co-simulation (Figure 7.2).
- **Coverage:** functional coverage (every opcode, every register as rd/rs1/rs2, every branch taken and not taken, corner values like 0x80000000) measures what the tests have exercised.
- **Formal verification:** riscv-formal proves, for bounded depths, that every retired instruction’s effect matches the ISA specification.

Worked Example 7.2: How many tests for exhaustive coverage?

Exhaustively testing even a single add over all operand values needs $2^{64} \approx 1.8 \times 10^{19}$ cases; at 10^9 simulated instructions per second that is 585 years. Random testing samples this space, coverage tells us which interesting *classes* (signs, overflow, zero results, x0 as destination) were hit, and formal methods prove the remaining properties symbolically. Verification typically consumes 60–70% of a processor project’s engineering effort — the topic of VL-401.

7.6 Synthesis and FPGA implementation

Worked Example 7.3: Estimating clock frequency on an FPGA

Typical 7-series (speed grade –1) delays: LUT ≈ 0.12 ns, net ≈ 0.3 – 0.5 ns per hop, carry chain ≈ 0.03 ns per bit, distributed-RAM read ≈ 1.0 ns, DSP multiply ≈ 3.5 ns (unpipelined). Critical path of a load: PC (t_{cq} 0.4) $\rightarrow$ instruction LUTRAM ($1.0 + 0.5$ net) $\rightarrow$ register-file LUTRAM ($1.0 + 0.5$) $\rightarrow$ operand MUX ($0.12 + 0.4$) $\rightarrow$ 32-bit adder ($0.2 + 32 \times 0.03 + 0.4$) $\rightarrow$ data LUTRAM ($1.0 + 0.5$) $\rightarrow$ write-back MUX ($0.12 + 0.4$) $\rightarrow$ register-file setup (0.3): about 8.7 ns, i.e. ≈ 115 MHz. With mul on the critical path (3.5 ns instead of the adder), about 10.7 ns (≈ 93 MHz) — the single-cycle penalty of Chapter 5 again: one slow operation slows

Table 7.2: How the Verilog constructs map to an AMD/Xilinx 7-series FPGA (e.g. the Zynq-7020 on the ZedBoard).

Construct	Typical implementation
regfile (2 async reads, 1 sync write)	distributed (LUT) RAM: two copies of a 32×32 RAM, one per read port ($\approx 2 \times 24$ LUTs used as RAM)
$a * b$	3–4 DSP48E1 slices (each a 25×18 multiplier)
adders ($pc+4$, $pc+imm$, ALU add/sub)	LUTs + CARRY4 fast-carry chains
shifts ($\ll$, $\gg$, $\gg>$)	5 levels of LUT-based multiplexers
control, immgen, muxes	a few hundred 6-input LUTs
PC	32 flip-flops
instruction/data memory with asynchronous read	distributed RAM (block RAM requires a synchronous read)

every instruction.

Advanced Insight: Moving to block RAM

Block RAMs register their address on a clock edge. A standard trick keeps the single-cycle behaviour by clocking the instruction BRAM with the *inverted* clock: the address (PC) is captured at the falling edge, half a cycle after the PC updates, and the instruction is available for the second half of the cycle. This halves the time available for the rest of the path, so it only works at low frequency. The clean solution is to treat the BRAM's internal register as a pipeline register — which is exactly the first step from the single-cycle design to the pipelined design of Unit 3, where the IF/ID register *is* the BRAM output register.

Chapter summary

- The single-cycle core maps one-to-one onto Verilog: regfile, immgen, alu, brcomp, control and a PC register, joined by conditional-expression multiplexers.
- Combinational logic in always @* with defaults; state in always @(posedge clk) with <=.
- Simulation of factorial reproduces the hand trace (25 cycles, 120 at 0x100); a self-checking directed test covers every instruction class.
- Real cores are verified by directed tests, architecture tests, random generation with ISS co-simulation, coverage and formal proofs.
- On an FPGA the core runs at roughly 90–115 MHz; the asynchronous-read memories prevent the use of block RAM, motivating pipelining.

Exercises

E7.1 Add lb, lh, lbu, lhu support to the core (load extension unit using funct3 and dmem_addr[1:0]). Write a directed test for it.

E7.2 Add sb and sh: change the data-memory interface to provide a 4-bit byte-enable mask.

What mask does `sh t0, 6(x0)` produce?

Answer: 1100 (bytes 2 and 3 of word 1).

E7.3 Modify `control` so that an unknown opcode sets an output `illegal`. In the testbench, stop the simulation and print the PC when it is asserted.

E7.4 The testbench prints `a1=x` in the first cycles. Add a reset loop to `regfile` that clears all registers. Why do ASIC designers often *not* reset register files?

Answer: Area and power: resetting 992 flip-flops needs reset-capable cells and routing; the ISA does not require initial values.

E7.5 Write the RTL for a directed test of `bltu` and `bge` with the operands `0x80000000` and `0x7FFFFFFF`. Predict the outcomes first.

Answer: bltu: $0x80000000 <_u 0x7FFFFFFF$ false; bge: $-2^{31} \geq 2^{31} - 1$ false.

Extending the Single-Cycle Processor

The core of Chapter 7 runs integer programs with word-sized memory accesses. A processor that can sit inside an SoC needs more: byte and halfword accesses, multiplication and division, a way to handle errors and system calls (exceptions and control/status registers), and a way to talk to peripherals (memory-mapped I/O). This chapter adds each of these to the single-cycle design, and in doing so meets the first situations where a single cycle is not enough.

8.1 A load/store unit for sub-word accesses

RV32I has five loads (lb, lh, lw, lbu, lhu) and three stores (sb, sh, sw). With a 32-bit-wide data memory, the processor needs:

- a **store alignment** block: replicate the low byte/halfword of rs2 onto the correct byte lanes and generate a 4-bit **byte-enable** mask;
- a **load extension** block: select the addressed byte/halfword from the 32-bit word and sign- or zero-extend it.

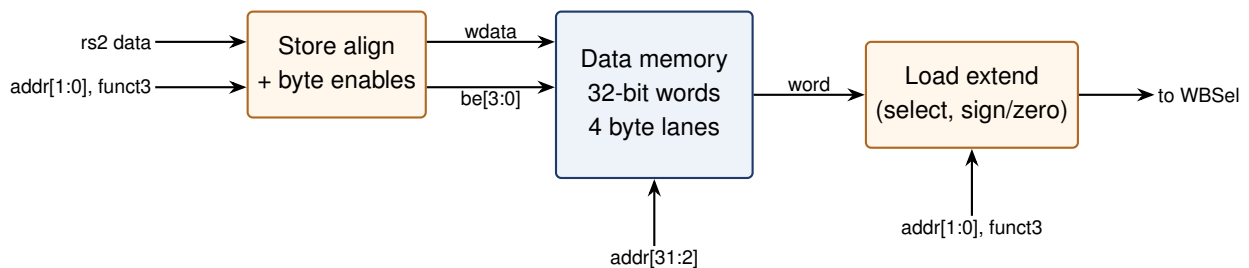


Figure 8.1: Load/store unit around a word-organized data memory.

Table 8.1: Byte enables and lane placement for stores (little-endian).

Store	addr[1:0]	be[3:0]	wdata
sw	00	1111	rs2
sh	00	0011	{rs2[15:0], rs2[15:0]}
sh	10	1100	{rs2[15:0], rs2[15:0]}
sb	00	0001	{4{rs2[7:0]}}
sb	01	0010	{4{rs2[7:0]}}
sb	10	0100	{4{rs2[7:0]}}
sb	11	1000	{4{rs2[7:0]}}

Replicating the data onto all lanes and letting the byte enables choose avoids a shifter: the alignment logic is just wiring plus a 2-to-4 decoder.

Worked Example 8.1: A sequence of sub-word accesses

Memory word at 0x200 initially 0x00000000; t0 = 0x8899AABB.

Instruction	be	Word at 0x200 after	Register result
sw t0, 0x200(x0)	1111	8899AABB	
sb x0, 0x201(x0)	0010	889900BB	
sh t0, 0x202(x0)	1100	AABB00BB	
lb t1, 0x203(x0)	—		t1 = FFFFFFFA
lbu t2, 0x203(x0)	—		t2 = 000000AA
lh t3, 0x200(x0)	—		t3 = 000000BB
lhu t4, 0x202(x0)	—		t4 = 0000AABB

For lh t3, 0x200: the low halfword is 0x00BB, bit 15 = 0, so sign extension gives 0x000000BB.

Misaligned accesses. A lw from address 0x202 spans two words. RISC-V allows an implementation either to handle it (two memory accesses, i.e. two cycles — not possible in a strict single-cycle design) or to raise a *load-address-misaligned* exception (Section 8.3) and let software emulate it. Small cores choose the exception.

8.2 The M extension: multiply and divide

Table 8.2: RV32M instructions (opcode 0110011, funct7 0000001).

funct3	Instruction	Result
000	mul	low 32 bits of $rs1 \times rs2$
001	mulh	high 32 bits, signed $\times$ signed
010	mulhsu	high 32 bits, signed $\times$ unsigned
011	mulhu	high 32 bits, unsigned $\times$ unsigned
100	div	signed quotient (rounded toward zero)
101	divu	unsigned quotient
110	rem	signed remainder (sign of dividend)
111	remu	unsigned remainder

No traps on division

RISC-V division never traps. Division by zero returns all ones (div: -1 , divu: $2^{32} - 1$) and the remainder equals the dividend; the signed overflow case $-2^{31} \div -1$ returns -2^{31} with remainder 0. Software that needs a trap checks the divisor with one branch. This keeps the pipeline simple: no instruction except loads, stores and system instructions can raise an exception.

Worked Example 8.2: M-extension results

Let $a_0 = 0xFFFFFFFFE$ (-2 signed, $4\,294\,967\,294$ unsigned) and $a_1 = 3$.

Instruction	Result	Explanation
<code>mul</code>	<code>FFFFFFFA</code>	-6
<code>mulh</code>	<code>FFFFFFF</code>	-6 as 64 bits = <code>FFFFFFF FFFFFFFFA</code>
<code>mulhu</code>	<code>00000002</code>	$4\,294\,967\,294 \times 3 = 2\,FFFFFFFA$
<code>div</code>	<code>00000000</code>	$-2/3 = -0.67 \rightarrow 0$ (toward zero)
<code>rem</code>	<code>FFFFFFFE</code>	$-2 = 3 \cdot 0 + (-2)$
<code>divu</code>	<code>55555554</code>	$4\,294\,967\,294/3 = 1\,431\,655\,764$
<code>remu</code>	<code>00000002</code>	$4\,294\,967\,294 = 3 \cdot 1\,431\,655\,764 + 2$

A full 64-bit product is obtained with the pair `mulh rdh, a, b; mul rdl, a, b`, which high-performance cores fuse into one operation.

8.2.1 Multiplier hardware

An $n \times n$ **array multiplier** forms n^2 partial-product bits with AND gates and adds them with $n - 1$ rows of adders. Its delay is about $2n$ full-adder delays (the carry ripples along rows and down columns). A **Wallace/Dadda tree** reduces the n partial products with carry-save adders in $\approx \log_{1.5}(n/2)$ levels, followed by one fast carry-propagate adder. **Radix-4 Booth recoding** halves the number of partial products.

Worked Example 8.3: Multiplier delay versus the single-cycle clock

Full-adder delay 40 ps, 64-bit prefix adder 250 ps.

- 32×32 array: $\approx 2 \cdot 32 \cdot 40 = 2560$ ps.
- Booth radix-4 + Wallace tree: 17 partial products; 3:2 compressors reduce 17 rows to 2 in 6 levels ($17 \rightarrow 12 \rightarrow 8 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 2$): $6 \cdot 40 = 240$ ps, plus Booth encoding (60 ps) and the final adder (250 ps): ≈ 550 ps.

Inserted into the 200 ps ALU slot, even the fast tree raises the single-cycle clock from 960 to ≈ 1310 ps (37% slower for *every* instruction). Practical cores pipeline the multiplier over 2–4 cycles.

8.2.2 Division cannot be single-cycle

The simple restoring or non-restoring divider produces one quotient bit per iteration: 32 iterations of a 32-bit subtract-and-shift, i.e. $32 \times (250 + 25) \approx 8.8$ ns if unrolled combinationally — nine times our clock period. Division is therefore implemented as an **iterative multi-cycle unit**: the processor *stalls* (holds the PC and does not write the register file) for ≈ 32 cycles (or 16 with radix 4) until the divider signals completion. This requires the first piece of genuinely multi-cycle control: a busy signal that freezes the PC — a preview of the stall logic of Unit 3.

Worked Example 8.4: Cost of division in a loop

A loop of 20 instructions contains one `div`. With a 32-cycle divider the loop takes $19 + 32 = 51$ cycles instead of 20: $CPI = 2.55$ for this loop. Replacing $x / 10$ by multiplication with

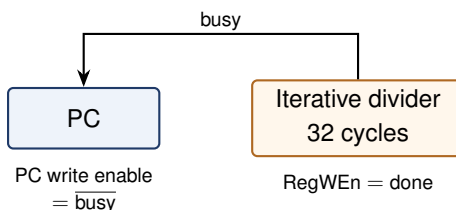


Figure 8.2: Stalling the single-cycle processor for a multi-cycle divider.

the reciprocal (`mulhu` by `0xCCCCCCCD` and a shift right by 3) brings it back to ≈ 22 cycles — which is exactly what compilers do for division by constants.

8.3 Exceptions and control/status registers

8.3.1 What must happen on a trap

An **exception** is an unusual condition caused by an instruction (illegal opcode, misaligned address, `ecall`, `ebreak`); an **interrupt** is an external event (timer, I/O). In machine mode, RISC-V handles both identically — a **trap**:

1. `mepc` $\leftarrow$ PC of the trapping instruction (for interrupts: of the next instruction to execute);
2. `mcause` $\leftarrow$ cause code (MSB = 1 for interrupts);
3. `mtval` $\leftarrow$ faulting address or instruction bits;
4. `mstatus.MPIE` $\leftarrow$ MIE; MIE $\leftarrow$ 0 (disable interrupts);
5. PC $\leftarrow$ `mtvec` (trap-handler address).

The handler ends with `mret`, which restores MIE and sets PC $\leftarrow$ `mepc`. For `ecall` the handler must add 4 to `mepc` first, or it will re-execute the `ecall` forever.

Table 8.3: Common machine-mode trap causes (`mcause`).

Code	Exception	Code	Interrupt (MSB set)
0	instruction address misaligned	3	machine software interrupt
2	illegal instruction	7	machine timer interrupt
3	breakpoint (<code>ebreak</code>)	11	machine external interrupt
4	load address misaligned		
6	store address misaligned		
11	environment call from M-mode		

8.3.2 Datapath additions

The **Zicsr** instructions read and modify CSRs atomically: `csrrw rd, csr, rs1` (swap), `csrrs` (set bits), `csrrc` (clear bits) and their immediate forms. They are I-type with opcode 1110011 (SYSTEM) and the 12-bit CSR address in the immediate field — e.g. `mtvec = 0x305`, `mepc = 0x341`, `mcause = 0x342`.

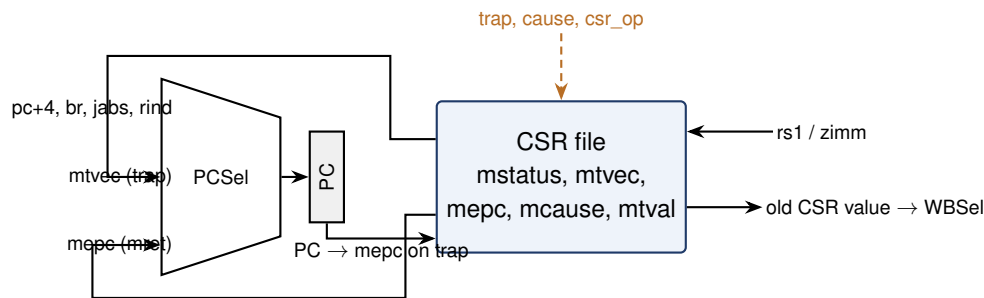


Figure 8.3: Additions for traps: a CSR file, two more PCSel inputs (mtvec, mepc) and a CSR read path to write-back for the csrrw/csrrs/csrrc instructions.

Worked Example 8.5: A system call, step by step

Setup: `mtvec = 0x00000100`. At `PC = 0x00000040` the program executes `ecall` (`0x00000073`) with `a7 = 1` (“print” service).

1. Control decodes `SYSTEM/funct3 = 000/imm = 0`: `ecall`. It asserts trap with cause 11.
2. At the clock edge: `mepc` $\leftarrow$ `0x40`, `mcause` $\leftarrow$ 11, `MPIE` $\leftarrow$ `MIE`, `MIE` $\leftarrow$ 0, `PC` $\leftarrow$ `0x100`. No register is written (`RegWen = 0`).
3. Handler at `0x100`: `csrr t0, mcause` (`csrrs t0, mcause, x0`), checks it equals 11, performs the service selected by `a7`, then `csrr t1, mepc`; `addi t1, t1, 4`; `csrw mepc, t1`.
4. `mret`: `PC` $\leftarrow$ `0x44`, `MIE` $\leftarrow$ `MPIE`. Execution resumes after the `ecall`.

In a single-cycle machine the “precise exception” requirement is free: the trapping instruction has not changed any state when the trap is taken, and all earlier instructions have completed. In a pipeline, achieving this is a major design problem (Unit 3).

Worked Example 8.6: Timer interrupts

The machine timer `mtime` is a 64-bit counter incremented at a fixed rate (say 10 MHz); an interrupt is pending when `mtime` $\geq$ `mtimecmp`. To get a 1 ms tick: `mtimecmp` $\leftarrow$ `mtime` + 10 000 after each interrupt. If the handler takes 150 instructions on our 1.04 GHz single-cycle core (`CPI = 1`), the interrupt overhead is $150 \times 0.96 \text{ ns} = 144 \text{ ns}$ per 1 ms, i.e. 0.014% of the processor’s time.

8.4 Memory-mapped I/O

Peripherals are controlled through registers that appear at memory addresses. A load or store to such an address is routed by an **address decoder** to the peripheral instead of RAM.

Worked Example 8.7: Driving LEDs and a UART

GPIO output register at `0x4000_1000`; UART: transmit-data register at `0x4000_0000`, status register at `0x4000_0004` whose bit 0 is “TX busy”.

```
lui    s0, 0x40001          # s0 = 0x40001000 (GPIO)
addi   t0, x0, 0x55
```

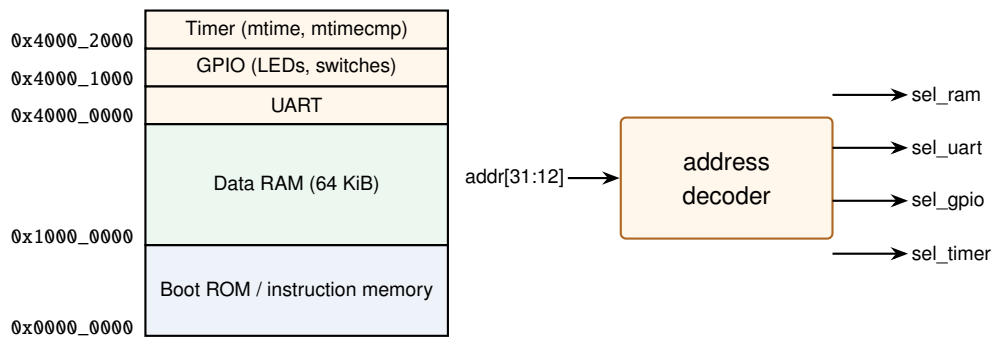


Figure 8.4: An example SoC address map and the decoder that selects the target of each load/store.

```

sw    t0, 0(s0)           # LEDs <- 01010101
lui   s1, 0x40000         # s1 = 0x40000000 (UART)
addi  t1, x0, 'A'         # 0x41
wait: lw    t2, 4(s1)      # read status
andi  t2, t2, 1
bne   t2, x0, wait        # spin while busy
sw    t1, 0(s1)           # transmit 'A'

```

At 115 200 baud (10 bits per character) a character takes $86.8 \mu\text{s}$; at 1.04 GHz the 3-instruction polling loop spins $\approx 86.8 \mu\text{s} / (3 \times 0.96 \text{ ns}) \approx 30\,000$ times per character. This waste is why real systems use UART interrupts or DMA.

Decoding cost. Comparing `addr[31:12]` against four regions takes four 20-bit comparators — or, with a carefully chosen map, just a few top bits (here `addr[30]` selects I/O, `addr[12]` selects GPIO vs. UART). Peripheral registers must also be declared *volatile* in C and must not be cached (Unit 4).

Advanced Insight: From a single-cycle core to a microcontroller SoC

Add the load/store unit, M extension (with a stalling divider), Zicsr with machine-mode traps, a timer and a UART behind an address decoder, and the single-cycle RV32IM core becomes a complete microcontroller able to run C programs compiled with `-march=rv32im_zicsr`. Open designs of exactly this kind (e.g. PicoRV32-based SoCs, the educational “single-cycle RISC-V on FPGA” labs) are the usual first project of a processor-design course. Replacing the core with a pipelined one (Unit 3) and adding caches (Unit 4) leaves the rest of the SoC unchanged — the ISA and the memory map are the stable interfaces.

Chapter summary

- Sub-word stores use replicated data and byte enables; sub-word loads select and sign- or zero-extend. Misaligned word accesses trap in simple cores.
- RV32M adds 8 instructions; division never traps. A fast multiplier (Booth + Wallace) is $\approx 2.5\times$ an ALU delay; a divider needs ≈ 32 iterations and forces the first multi-cycle stall.
- Machine-mode traps save PC in `mepc`, the cause in `mcause`, and jump to `mtvec`; `mret` returns. Exceptions are trivially precise in a single-cycle CPU.
- Memory-mapped I/O routes loads/stores to peripherals via an address decoder.

Exercises

E8.1 Give `be[3:0]` and the stored word for `sb t0, 0x103(x0)` when `t0 = 0x12345678` and the word at `0x100` was `0xAABBCCDD`.

Answer: `be = 1000`; word becomes `0x78BBCCDD`.

E8.2 Compute `div`, `rem`, `divu`, `remu` for `a0 = -7`, `a1 = 2`.

Answer: `-3`, `-1`; `0x7FFFFFFC`, `1`.

E8.3 What do `div` and `rem` return for $-2^{31} \div -1$ and for $5 \div 0$?

Answer: -2^{31} , `0`; `-1`, `5`.

E8.4 Write a trap handler fragment that, on an illegal instruction (cause 2), skips the faulting instruction and returns.

Answer: `csrr t0,mepc; addi t0,t0,4; csrw mepc,t0; mret` (after checking `mcause = 2`).

E8.5 How many cycles does a 32-bit radix-4 iterative divider need? With 3% divisions in a program, what is the CPI of the stalling single-cycle machine?

Answer: 16 cycles; $\text{CPI} = 0.97 + 0.03 \times 16 = 1.45$.

E8.6 Design an address map in which a single address bit selects RAM versus I/O, and another bit selects among two peripherals. Write the decoder equations.

RV32I(M) Reference Card

Instr.	Fmt	opcode	funct3	funct7 / imm	Datapath settings	Semantics
lui	U	0110111	—	—	Op2=Imm, COPYB	$rd = imm \ll 12$
auipc	U	0010111	—	—	Op1=PC, Op2=Imm	$rd = PC + (imm \ll 12)$
jal	J	1101111	—	—	PCSel=jabs, WB=PC+4	$rd = PC + 4; PC += imm$
jalr	I	1100111	000	—	PCSel=rind, WB=PC+4	$rd = PC + 4; PC = (rs1 + imm) \& \sim 1$
beq	B	1100011	000	—	BrEq	if $rs1 = rs2$
bne	B	1100011	001	—	$\overline{\text{BrEq}}$	if $rs1 \neq rs2$
blt	B	1100011	100	—	BrLT, BrUn=0	if $rs1 <_s rs2$
bge	B	1100011	101	—	$\overline{\text{BrLT}}$, BrUn=0	if $rs1 \geq_s rs2$
bltu	B	1100011	110	—	BrLT, BrUn=1	if $rs1 <_u rs2$
bgeu	B	1100011	111	—	$\overline{\text{BrLT}}$, BrUn=1	if $rs1 \geq_u rs2$
lb	I	0000011	000	—	WB=Mem, sign-ext byte	$rd = sx(M_8[rs1 + imm])$
lh	I	0000011	001	—	WB=Mem, sign-ext half	$rd = sx(M_{16}[\dots])$
lw	I	0000011	010	—	WB=Mem	$rd = M_{32}[rs1 + imm]$
lbu	I	0000011	100	—	WB=Mem, zero-ext	$rd = zx(M_8[\dots])$
lhu	I	0000011	101	—	WB=Mem, zero-ext	$rd = zx(M_{16}[\dots])$
sb	S	0100011	000	—	MemWr, be=1 lane	$M_8[rs1 + imm] = rs2$
sh	S	0100011	001	—	MemWr, be=2 lanes	$M_{16}[\dots] = rs2$
sw	S	0100011	010	—	MemWr, be=1111	$M_{32}[\dots] = rs2$
addi	I	0010011	000	—	ADD	$rd = rs1 + imm$
slti	I	0010011	010	—	SLT	$rd = (rs1 <_s imm)$
sltiu	I	0010011	011	—	SLTU	$rd = (rs1 <_u imm)$
xori	I	0010011	100	—	XOR	$rd = rs1 \oplus imm$
ori	I	0010011	110	—	OR	$rd = rs1 imm$
andi	I	0010011	111	—	AND	$rd = rs1 \& imm$
slli	I	0010011	001	0000000	SLL	$rd = rs1 \ll sh$
srli	I	0010011	101	0000000	SRL	$rd = rs1 \gg_l sh$
srai	I	0010011	101	0100000	SRA	$rd = rs1 \gg_a sh$
add	R	0110011	000	0000000	ADD	$rd = rs1 + rs2$
sub	R	0110011	000	0100000	SUB	$rd = rs1 - rs2$
sll	R	0110011	001	0000000	SLL	$rd = rs1 \ll rs2[4 : 0]$
slt	R	0110011	010	0000000	SLT	$rd = (rs1 <_s rs2)$
sltu	R	0110011	011	0000000	SLTU	$rd = (rs1 <_u rs2)$
xor	R	0110011	100	0000000	XOR	$rd = rs1 \oplus rs2$
srl	R	0110011	101	0000000	SRL	$rd = rs1 \gg_l rs2[4 : 0]$
sra	R	0110011	101	0100000	SRA	$rd = rs1 \gg_a rs2[4 : 0]$
or	R	0110011	110	0000000	OR	$rd = rs1 rs2$
and	R	0110011	111	0000000	AND	$rd = rs1 \& rs2$
mul	R	0110011	000	0000001	MUL	$rd = (rs1 \times rs2)[31 : 0]$
mulh	R	0110011	001	0000001	MULH	$rd = (rs1 \times_{ss} rs2)[63 : 32]$

Instr.	Fmt	opcode	funct3	funct7 / imm	Datapath settings	Semantics
div	R	0110011	100	0000001	DIV (iterative)	$rd = rs1 \div_s rs2$
rem	R	0110011	110	0000001	REM (iterative)	$rd = rs1 \bmod_s rs2$
ecall	I	1110011	000	imm=0	trap, cause 11	system call
ebreak	I	1110011	000	imm=1	trap, cause 3	breakpoint
csrrw	I	1110011	001	imm=csr	CSR swap	$rd = csr; csr = rs1$
csrrs	I	1110011	010	imm=csr	CSR set	$rd = csr; csr = rs1$
mret	R	1110011	000	0011000 00010	PCSel=mepc	return from trap

Common pseudo-instructions. `nop = addi x0,x0,0; li rd,imm = addi (or lui+addi); mv rd,rs = addi rd,rs,0; not rd,rs = xori rd,rs,-1; neg rd,rs = sub rd,x0,rs; seqz rd,rs = sltiu rd,rs,1; beqz rs,L = beq rs,x0,L; bgt rs,rt,L = blt rt,rs,L; j L = jal x0,L; call f = auipc ra,..; jalr ra,..(ra); ret = jalr x0,0(ra); la rd,sym = auipc+addi; csrr rd,csr = csrrs rd,csr,x0.`

B

Solved Design Problems

Worked Example B.1: Adding a new instruction: `bnei rs1, imm5, label`

Specification. Branch if $rs1 \neq$ a 5-bit sign-extended immediate placed in the $rs2$ field (a hypothetical custom instruction in the custom-0 opcode space).

Datapath changes. The branch comparator's second input must be able to take a sign-extended $inst[24:20]$: add a 2:1 multiplexer (BrOp2Sel) in front of the comparator's B input and a 5-to-32 sign extender. The branch target still comes from the B-type immediate. Op1/Op2/ALU are unaffected.

Control. New row: ImmSel = B, BrOp2Sel = imm5, MemWr = 0, RegWEn = 0, PCSel = br if $\overline{BrEq}$ else pc+4.

Timing. The new MUX is on the comparator path: branch path 670 $\rightarrow$ 695 ps — still shorter than the load's 960 ps, so the clock is unchanged. **Benefit.** Loops ending in `addi t0, t0, 1; bnei t0, 10, loop` save the `li t1, 10` outside the loop (a static saving) and free a register; dynamic count is unchanged. Probably not worth the opcode.

Worked Example B.2: Adding `lwpi rd, imm(rs1)`: load with post-increment

Specification. $rd \leftarrow M[rs1]$; $rs1 \leftarrow rs1 + imm$.

Problem. Two register writes in one instruction: the register file has only one write port. Options: (a) add a second write port (costly: the RF area grows by $\approx (4/3)^2$, and all write decoders double); (b) take two cycles (breaks single-cycle). Also the memory address is $rs1$ (not $rs1 + imm$), so the address must bypass the ALU: a new MUX on the data-memory address input (AddrSel = $rs1 / ALU$).

Conclusion. With option (a): datapath +1 MUX, +1 write port, control +2 signals. Loads become slightly slower (MUX on address path: +25 ps $\Rightarrow t_C = 985$ ps, +2.6%). If 8% of instructions are loads that can use post-increment and each saves one `addi`, N drops 8%: net speedup $1/(0.92 \times 1.026) = 1.059$. A 6% gain for a second write port — the classic RISC trade-off discussed in Unit 1.

Worked Example B.3: Shortening the critical path by moving the comparator

The load path (960 ps) is dominated by two 250 ps memories. Suppose instead the instruction memory is a fast 150 ps ROM and the data memory remains 250 ps.

Path	Before (ps)	After (ps)
Load	960	860
R-type	710	610
Branch	670	570
JALR	715	615

t_C falls by 10.4%. Speeding up the data memory to 150 ps as well gives 760 ps; then the R-type (610) and load (760) differ by only 150 ps. Halving the ALU delay now gains 100 ps on every path (13%). The general lesson: after each optimization, *recompute the critical path*, because it moves.

Worked Example B.4: Diagnosing a failing program from a trace

A student's core runs the program `addi t0,x0,-1; srli t1,t0,28; srai t2,t0,28` and reports `t1 = 0xFFFFFFFF, t2 = 0xFFFFFFFF`. Expected: `t1 = 0x0000000F, t2 = 0xFFFFFFFF`. **Analysis.** `srli` and `srai` differ only in bit 30 (`inst[30]`). The core produced an arithmetic shift for both, so its ALU decoder must be taking bit 30 as 1 for `srli`, or ignoring it and always choosing SRA. Checking the encoding: `srli t1,t0,28 = 0x01C2D313`, bit 30 = 0. So the decoder ignores bit 30 for I-type — the student “fixed” the `addi` bug by masking bit 30 for *all* I-type instructions, breaking `srai/srli`. Correct rule: qualify bit 30 with `funct3 = 101` for I-type (as in Listing 7.1).

Worked Example B.5: How long does a memory copy take?

Copy n words with the loop

```
loop: lw    t0, 0(a0)
      sw    t0, 0(a1)
      addi  a0, a0, 4
      addi  a1, a1, 4
      addi  a2, a2, -1
      bne   a2, x0, loop
```

6 instructions per word, so for $n = 1024$: $6144 \text{ cycles} \times 0.96 \text{ ns} = 5.9 \mu\text{s}$, a bandwidth of $4096 \text{ B} / 5.9 \mu\text{s} = 0.69 \text{ GB/s}$. Unrolling by 4 (4 `lw`, 4 `sw`, 3 `addi`, 1 `bne` per 4 words = 3 instructions per word) doubles the bandwidth to 1.39 GB/s. A DMA engine moving one word per cycle at 1 GHz would reach 4 GB/s with the processor free — which is why SoCs include DMA controllers.