RV32I Single-Cycle Processor Design

YARP — Yet Another RISC-V Processor

FROM FIRST PRINCIPLES TO FULL RTL IMPLEMENTATION

A complete engineering reference covering computer architecture foundations, the RISC-V ISA, all six instruction types, and a full SystemVerilog implementation of a single-cycle RV32I core — Fetch, Decode, Register File, ALU, Data Memory, and Control Unit.

RISC-V RV32I | Single-Cycle | SystemVerilog RTL | In-Order Execution | Computer Architecture

March 2026 | Basics to Advanced | 14 Chapters

Table of Contents

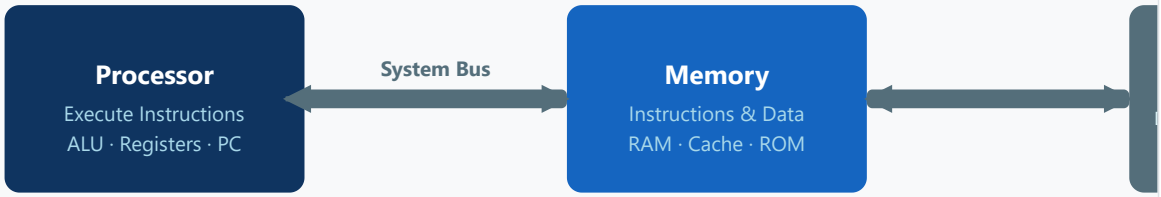
CHAPTER 1 — PART I: FOUNDATIONS

Computer Architecture & The Processor

1.1 What is Computer Architecture?

Computer architecture describes the **structure, organization, and functionality** of a computer system. It defines how the major components — processor, memory, and I/O devices — are designed and how they interact with each other to execute programs.

FIGURE 1.1 — COMPUTER SYSTEM COMPONENTS

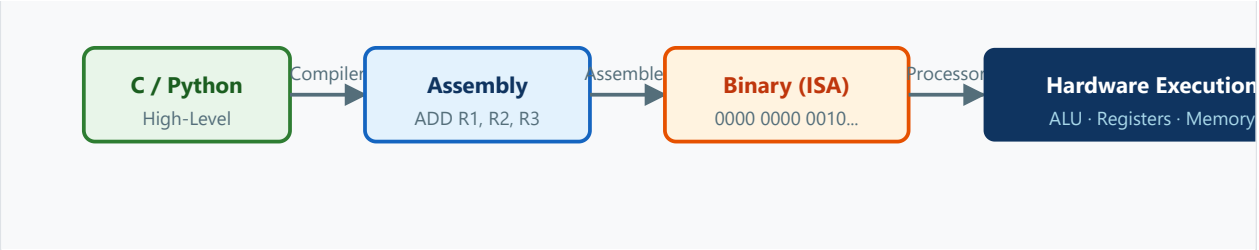


Data & Instructions flow bidirectionally across the system bus

1.2 The Role of the Processor

The processor's primary function is to **execute instructions**. A user writes a program in a high-level language (C, Python). That program is compiled to *assembly*, then assembled into *binary* — strings of 1s and 0s that the processor understands directly.

FIGURE 1.2 — SOFTWARE-TO-HARDWARE TRANSLATION CHAIN



1.3 ISA — The Contract Between Software and Hardware

The **Instruction Set Architecture (ISA)** defines the contract between software and hardware. It specifies:

- The complete set of instructions the hardware can execute
- How those instructions are encoded in binary
- The registers visible to the programmer
- Memory addressing rules and alignment constraints
- Behaviour of arithmetic, logic, load/store, and control flow operations

> **Key Insight**
>
> The ISA is what allows software to be compiled once and run on any processor that implements that ISA — regardless of the underlying microarchitecture.

CHAPTER 2

RISC-V — Philosophy & Classification

2.1 What Makes RISC-V Different

RISC-V (pronounced "risk-five") is an **open-source, royalty-free ISA** originally developed at UC Berkeley for research and education. Unlike proprietary ISAs (x86, ARM), anyone can implement RISC-V without licensing fees. It has since grown into a mainstream architecture adopted by industry — including AMD, Western Digital, Google, and many SoC vendors.

Design Philosophy

Clean, minimal base ISA with optional extensions. The base integer ISA (I) is deliberately small — sufficient for compilers, OS kernels, and embedded systems — with optional extensions (M for multiply, F for float, etc.) added only when needed.

Avoid Over-Architecting

RISC-V avoids committing to a specific microarchitecture style. The ISA is defined by *what* an instruction does, not *how* the hardware implements it.

2.2 Unprivileged vs Privileged ISA

RISC-V separates its specification into two distinct parts:

Feature	Unprivileged ISA	Privileged ISA
Who uses it	User programs	Operating system / hypervisor
Hardware access	No	Yes — full control
System control	No	Yes — full control
Examples	ADD, LOAD, STORE	Set page table, I/O control
CPU mode	User mode	Kernel / Machine mode

Why This Separation Exists

Without it, any user program could crash the system, access other programs' memory, or control hardware directly — leading to catastrophic security and stability failures. CPUs enforce: User Mode → only unprivileged ISA. Kernel Mode → access to privileged ISA.

2.3 Base ISA Variants: RV32I and RV64I

There are two primary base integer variants. `XLEN` refers to the integer register width in bits.

Variant	XLEN	Address Space	Use Case
RV32I	32 bits	32-bit (4 GB)	Embedded, microcontrollers, our YARP design
RV64I	64 bits	64-bit (16 EB)	Linux servers, application processors

M Extension (not in base)

By default, the base RV32I ISA does NOT include hardware multiply or divide. Integer multiply (MUL, DIV, REM) requires the optional

M extension

. Our YARP processor implements the pure RV32I base — no M extension.

CHAPTER 3

Processor Execution Models

3.1 In-Order Processors

An in-order processor **executes instructions exactly in the sequence they appear** in the program. If one instruction stalls (e.g., waiting for data from memory), every instruction behind it must wait too.

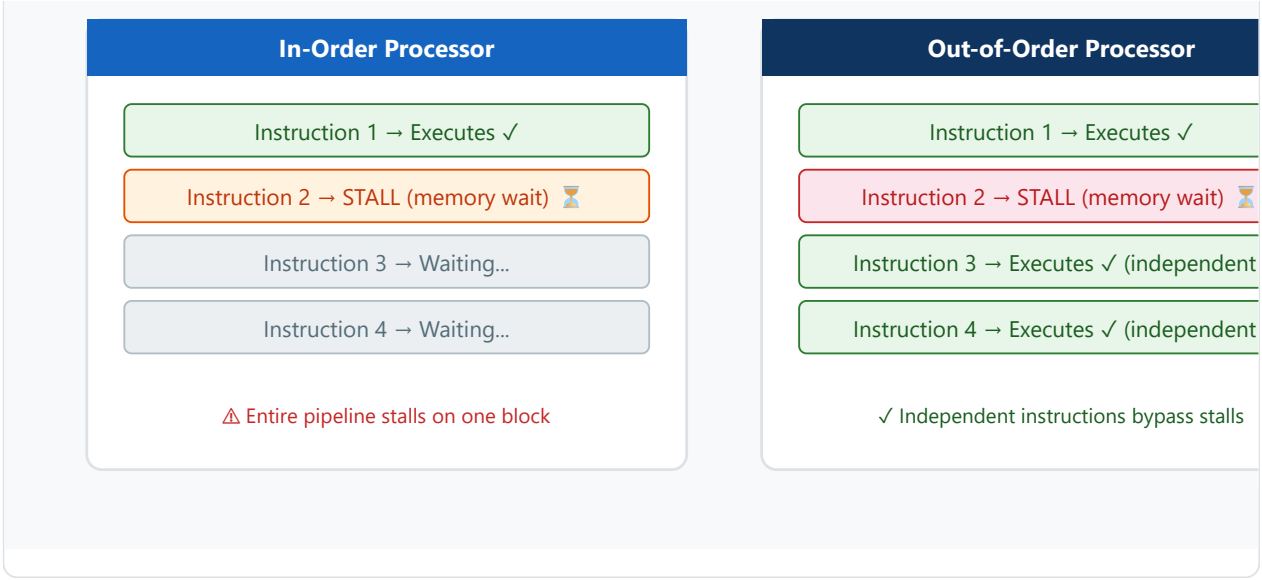
> **Characteristics**
>
> Simple hardware, predictable timing, deterministic behaviour. Lower performance under stall-heavy workloads.

3.2 Out-of-Order Processors

An out-of-order (OOO) processor looks ahead at multiple instructions and executes independent ones **as soon as their inputs are ready**, even if earlier instructions are stalled. Results are committed back in the original program order to preserve correctness (called *precise state*).

> **OOO Hardware Requirements**
>
> Instruction window · Reorder buffer (ROB) · Dependency checking logic · Reservation stations. This is what makes OOO processors complex and power-hungry.

FIGURE 3.1 — IN-ORDER VS OUT-OF-ORDER EXECUTION



3.3 Why YARP is Single-Cycle In-Order

A **single-cycle processor** completes every instruction in exactly **one clock cycle**. Each instruction flows through: Fetch → Decode → Execute → Memory → Register Write-Back — all within a single cycle. The next instruction does not begin until the current one completes fully.

✓ In-Order

Instructions execute strictly sequentially. No reordering, no speculation.

✗ No Pipelining

All five stages active in the same cycle for one instruction.

✗ No Hazard Handling

Since only one instruction is in flight, data/control hazards cannot arise.

One-Line Summary

An RV32I single-cycle processor is strictly in-order because it executes one instruction completely per cycle — with no overlap, no reordering, and no dynamic scheduling.

CHAPTER 4 — PART II: RV32I INSTRUCTION SET ARCHITECTURE

Instruction Formats & Encoding

4.1 The Six Instruction Types

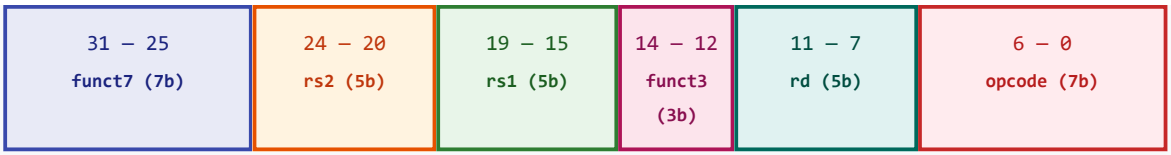
All RV32I instructions are **exactly 32 bits wide** and word-aligned. The first 7 bits (`instr[6:0]`) are the **opcode** — they tell the processor which type of instruction this is and what operation to perform. There are six instruction formats:

Type	Opcode	Purpose	Examples
R-type	0x33	Register-register arithmetic/logic	ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT
I-type (arith)	0x13	Register-immediate arithmetic	ADDI, ANDI, ORI, XORI, SLTI, SLLI, SRLI, SRAI
I-type (load)	0x03	Load from memory	LW, LH, LHU, LB, LBU
I-type (JALR)	0x67	Jump and link register	JALR
S-type	0x23	Store to memory	SW, SH, SB
B-type	0x63	Conditional branch	BEQ, BNE, BLT, BGE, BLTU, BGEU
U-type	0x17, 0x37	Upper immediate operations	AUIPC, LUI
J-type	0x6F	Unconditional jump	JAL

4.2 R-Type: Register-Register Operations

R-type instructions operate on **two source registers (rs1, rs2)** and write the result to a **destination register (rd)**. The funct3 and funct7 fields together identify the specific operation.

FIGURE 4.1 — R-TYPE INSTRUCTION FORMAT (32 BITS)

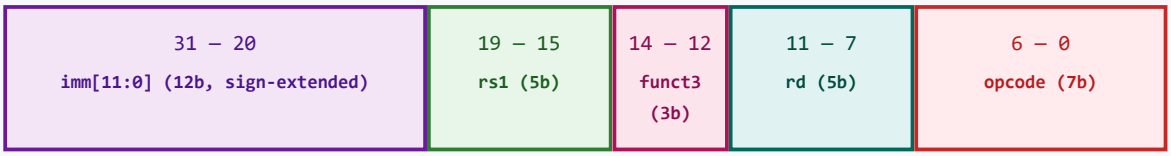



```
Example: ADD R1, R2, R3
  opcode = 0x33  (R-type)
  funct3 = 0x0   funct7 = 0x00  → ADD
  funct3 = 0x0   funct7 = 0x20  → SUB
  funct3 = 0x4   funct7 = 0x00  → XOR
  funct3 = 0x6   funct7 = 0x00  → OR
  funct3 = 0x7   funct7 = 0x00  → AND
```

4.3 I-Type: Immediate & Load Operations

I-type instructions use **one source register (rs1)** and a **12-bit sign-extended immediate** as the second operand. The result goes to rd. The same format is used for arithmetic immediates, loads, and JALR.

FIGURE 4.2 — I-TYPE INSTRUCTION FORMAT

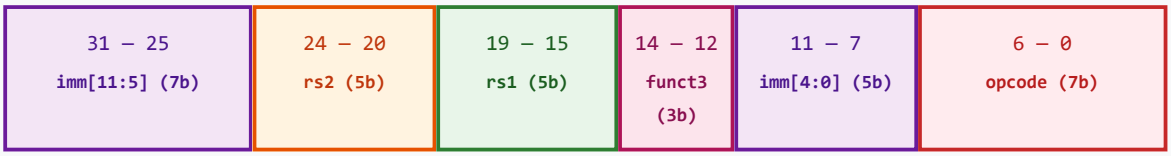


Instruction	opcode	Operation
ADDI rd, rs1, imm	0x13	rd = rs1 + sign_ext(imm)
LW rd, imm(rs1)	0x03	rd = Mem[rs1 + sign_ext(imm)]
JALR rd, rs1, imm	0x67	rd = PC+4; PC = rs1 + imm

4.4 S-Type: Store Operations

S-type instructions store a register value to memory. The immediate is **split across two fields** (to keep rs1, rs2, funct3 in the same bit positions as R-type — simplifying decoder logic). There is no rd field — no result is written to a register.

FIGURE 4.3 — S-TYPE INSTRUCTION FORMAT

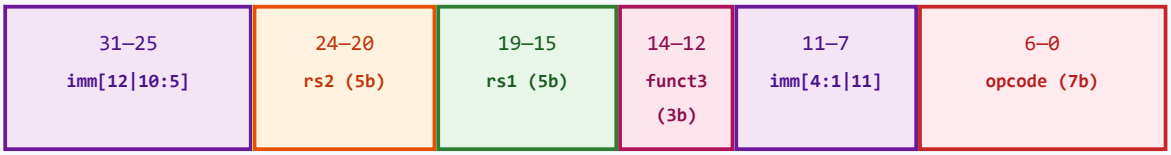



```
Effective address = rs1 + sign_ext({imm[11:5], imm[4:0]})
SW rs2, imm(rs1)  → Mem[rs1+imm] = rs2   (32-bit word store)
SH rs2, imm(rs1)  → Mem[rs1+imm] = rs2[15:0]
SB rs2, imm(rs1)  → Mem[rs1+imm] = rs2[7:0]
```

4.5 B-Type: Branch Operations

B-type is similar to S-type but encodes a PC-relative branch **offset**. The immediate is again split and scrambled to maintain rs1/rs2/funct3 register field alignment. Note: bit 0 of the offset is always 0 (branches target word-aligned addresses).

FIGURE 4.4 — B-TYPE INSTRUCTION FORMAT



Instruction	funct3	Condition
BEQ rs1, rs2, offset	0x0	Branch if rs1 == rs2
BNE rs1, rs2, offset	0x1	Branch if rs1 ≠ rs2
BLT rs1, rs2, offset	0x4	Branch if rs1 < rs2 (signed)
BGE rs1, rs2, offset	0x5	Branch if rs1 ≥ rs2 (signed)
BLTU rs1, rs2, offset	0x6	Branch if rs1 < rs2 (unsigned)
BGEU rs1, rs2, offset	0x7	Branch if rs1 ≥ rs2 (unsigned)

4.6 U-Type: Upper Immediate Operations

U-type instructions load a **20-bit immediate into the upper 20 bits** of a destination register (bits [31:12]), with the lower 12 bits set to zero.

FIGURE 4.5 — U-TYPE INSTRUCTION FORMAT

LUI (Load Upper Immediate) — opcode 0x37

rd = {imm[31:12], 12'b0}
Directly loads a 20-bit constant into the upper portion of rd.

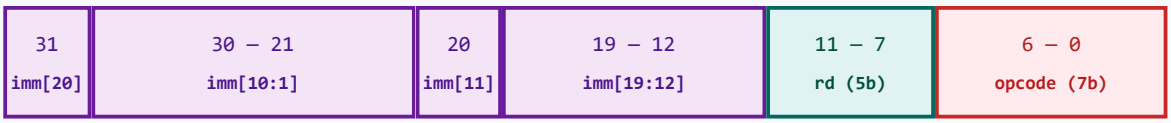
AUIPC (Add Upper Immediate to PC) — opcode 0x17

rd = PC + {imm[31:12], 12'b0}
Used to build PC-relative addresses. Paired with ADDI to form any 32-bit address.

4.7 J-Type: Jump Operations

J-type (JAL) encodes a 21-bit PC-relative offset into a heavily scrambled immediate field — the scrambling keeps rs1 aligned with other types to reduce mux complexity in the decoder.

FIGURE 4.6 — J-TYPE INSTRUCTION FORMAT




```
JAL rd, offset
  rd  = PC + 4          (save return address)
  PC  = PC + sign_ext(offset)    (jump to target)
  Note: offset is always even (bit[0]=0, branch to word-aligned)
```

CHAPTER 5

Programmer's Model — Registers & PC

5.1 The 32 General-Purpose Registers

RV32I provides **32 integer registers**, each 32 bits wide (XLEN=32). They are named `x0` through `x31`. Register addresses are 5 bits wide (2^5 = 32 addresses), which is why rs1, rs2, and rd fields in instruction encodings are all 5 bits.

FIGURE 5.1 — RV32I PROGRAMMER'S MODEL

Reg	ABI Name	Role
x0	zero	Hardwired Zero (always 0)
x1	ra	Return Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x5–x7	t0–t2	Temporaries
x8	s0/fp	Saved / Frame Pointer
x9	s1	Saved Register
x10–x11	a0–a1	Function Args / Return Values
x12–x17	a2–a7	Function Arguments
x18–x27	s2–s11	Saved Registers (callee-saved)
x28–x31	t3–t6	Temporaries

Reg	ABI Name
PC — Program Counter	
32-bit register	
Holds address of current instruction	
Increments by 4 each cycle (word-aligned)	

Register File Summary

- 32 registers total (x0–x31)
- Each register is XLEN=32 bits wide
- x0 is hardwired to 0 — always reads 0
- All other registers are general-purpose
- No mandatory reset value in ISA
- **5-bit address → 32 addresses (2^5)**

5.3 X0: The Hardwired Zero Register

Register `x0` always reads as zero and ignores all writes. This is not a software convention — it is enforced in hardware. This enables useful pseudo-instructions:

```
MV  rd, rs1        →  ADDI rd, rs1, 0      (copy register)
NOP                →  ADDI x0, x0, 0     (no operation)
NEG rd, rs1        →  SUB  rd, x0, rs1   (negate)
```

```
SEQZ rd, rs1       →  SLTIU rd, rs1, 1   (set if equal zero)
J   target         →  JAL  x0, target    (jump, discard return addr)
```

CHAPTER 6 — PART III: YARP MICROARCHITECTURE

YARP Overview & Full Datapath

6.1 Single-Cycle Execution Philosophy

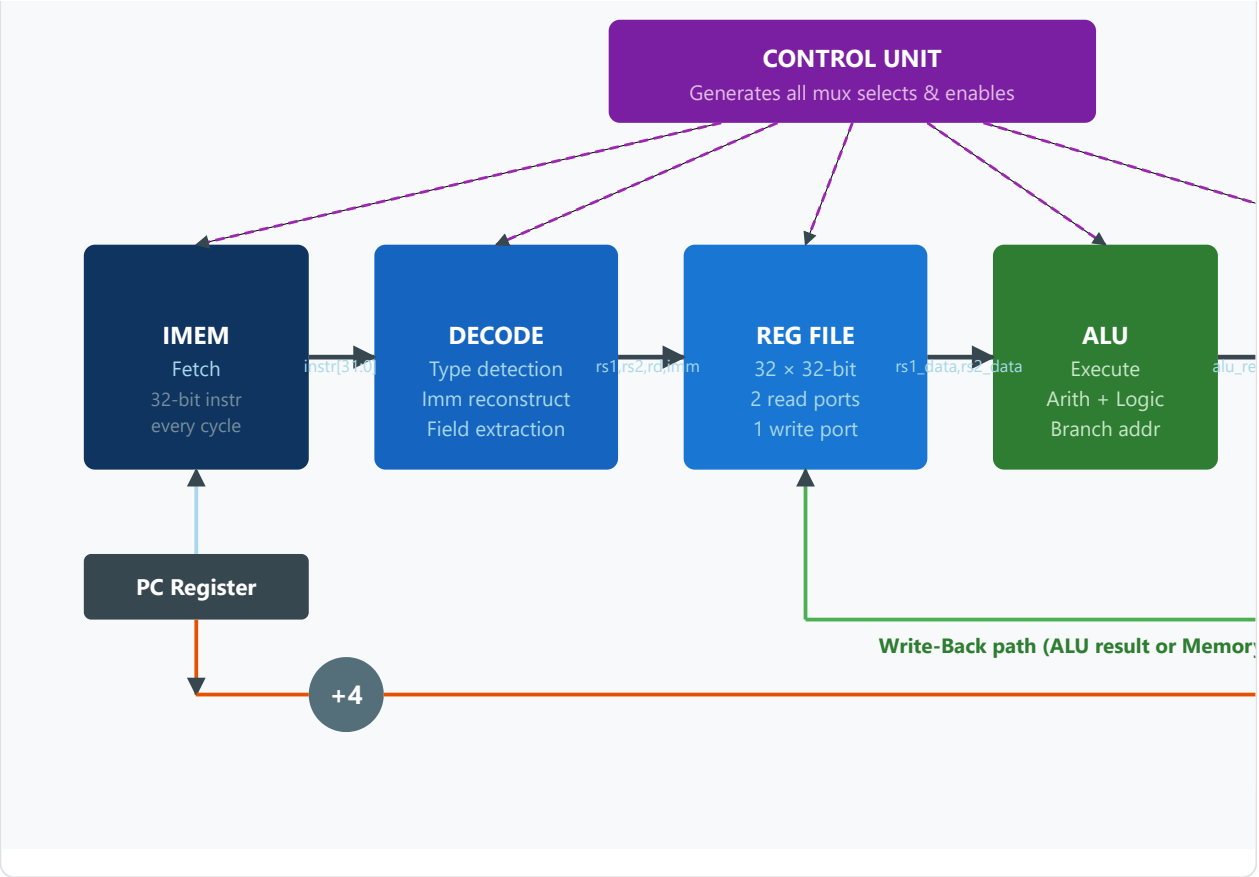
In YARP, **every instruction completes within a single clock cycle**. The full datapath — from instruction fetch to register writeback — is combinational (or registered at the very end). This means:

- No pipeline registers between stages
- No hazard detection or forwarding logic needed
- No out-of-order scheduling hardware
- The clock period must accommodate the worst-case instruction (typically a load)

```
Per-cycle operation:
  Fetch → Decode → [Register Read] → ALU/Execute → Data Memory → Register Write-Back
  ──────────────────────────────────────────────────────────────────────────────────
  All of the above happens in ONE clock cycle for ONE instruction.
```

6.2 The Five Functional Blocks

FIGURE 6.1 — YARP FULL SINGLE-CYCLE DATAPATH



The **Control Unit** receives instruction type signals from the decoder and generates all the mux-select signals and enable bits that route data correctly through the datapath. It is pure combinational logic — no state.

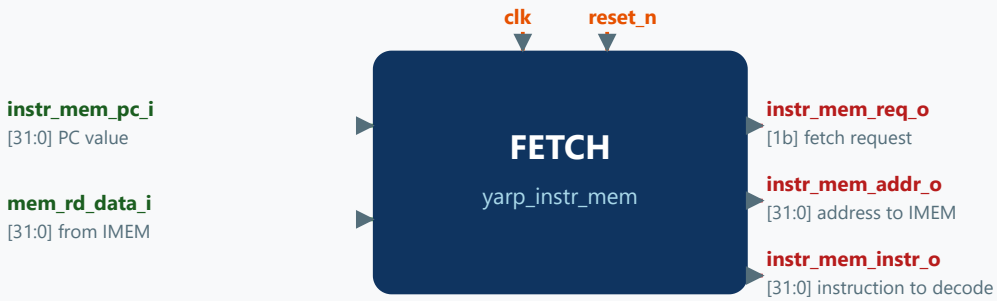
CHAPTER 7

Fetch Stage — YARP IMEM

7.1 Signals & Interface

The Fetch stage is responsible for **requesting instructions from memory** every cycle. It takes the current PC value, asserts a read request to instruction memory, and forwards the returned 32-bit instruction to the Decode stage.

FIGURE 7.1 — YARP FETCH MODULE BLOCK DIAGRAM



Signal	Direction	Width	Description
`instr_mem_pc_i`	Input	32b	Current PC — address of instruction to fetch
`mem_rd_data_i`	Input	32b	Instruction word returned by instruction memory
`instr_mem_req_o`	Output	1b	Fetch request signal to instruction memory (asserted when active)
`instr_mem_addr_o`	Output	32b	Address sent to instruction memory — equals PC
`instr_mem_instr_o`	Output	32b	Fetched instruction forwarded to the Decode stage

7.2 Fetch Logic Explained

Since this is a single-cycle processor, `instr_mem_addr_o` is simply wired to `instr_mem_pc_i` — the PC is the instruction address. The fetch request (`instr_mem_req_o`) is de-asserted synchronously on reset using an async-negedge flip-flop, then permanently asserted once out of reset.

> **Design Note — Why Async Reset for req?**
>
> The instruction memory request must be deasserted the instant reset is asserted, even mid-cycle. An asynchronous negedge-reset FF achieves this without waiting for a clock edge.

7.3 RTL Implementation

```
// YARP Instruction Memory — Fetch Stage RTL
module yarp_instr_mem (
  input  logic         clk,
  input  logic         reset_n,
  input  logic [31:0]  instr_mem_pc_i,   // PC address value

  // Memory interface — read request to IMEM
  output logic         instr_mem_req_o,
  output logic [31:0] instr_mem_addr_o,
  input  logic [31:0]  mem_rd_data_i,    // instruction from memory

  // Instruction to decoder
  output logic [31:0] instr_mem_instr_o
);

  // Assert req after reset deasserts (async negedge reset)
  always_ff @(posedge clk, negedge reset_n)
    if (~reset_n) instr_mem_req_o <= 1'b0;
    else          instr_mem_req_o <= 1'b1;

  // PC passes directly to memory address port
  assign instr_mem_addr_o  = instr_mem_pc_i;

  // Memory data is the instruction — forward to decoder
  assign instr_mem_instr_o = mem_rd_data_i;

endmodule
```

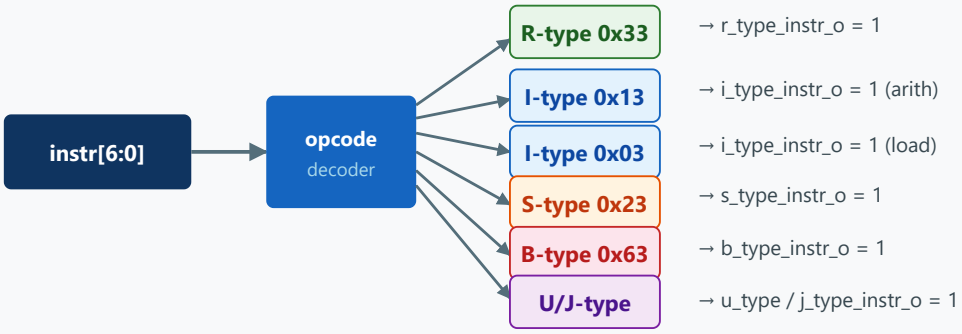
CHAPTER 8

Decode Stage — yarp_decode

8.1 Opcode Map & Type Detection

The Decode unit receives the raw 32-bit instruction from Fetch and performs three jobs simultaneously: identify the instruction type, extract register addresses and function fields, and reconstruct the sign-extended immediate value.

FIGURE 8.1 — OPCODE TO INSTRUCTION TYPE MAPPING



8.2 Immediate Reconstruction

Each instruction type encodes its immediate differently — bits are scattered across the instruction word to keep rs1/rs2/funct3 aligned. The decoder reassembles these into a clean 32-bit sign-extended immediate:

```
I-type imm = sign_ext({ instr[31:20] })
S-type imm = sign_ext({ instr[31:25], instr[11:7] })
B-type imm = sign_ext({ instr[31], instr[7], instr[30:25], instr[11:8], 1'b0 })
U-type imm = { instr[31:12], 12'b0 }
J-type imm = sign_ext({ instr[31], instr[19:12], instr[20], instr[30:21], 1'b0 })
```

Why Bit Scrambling?

> The RISC-V ISA intentionally scrambles immediate bits so that rs1 (bits 19:15), rs2 (bits 24:20), and funct3 (bits 14:12) always sit in the same bit positions across all instruction types. This reduces the number of muxes needed in the decode unit.

8.3 RTL Implementation

```
module yarp_decode import yarp_pkg::*; (
  input  logic [31:0] instr_i,      // 32-bit instruction from fetch
  output logic [4:0]  rs1_o,        // source reg 1 address
  output logic [4:0]  rs2_o,        // source reg 2 address
  output logic [4:0]  rd_o,         // destination reg address
  output logic [6:0]  op_o,         // opcode field
  output logic [2:0]  funct3_o,
  output logic [6:0]  funct7_o,
  output logic        r_type_instr_o,
  output logic        i_type_instr_o,
  output logic        s_type_instr_o,
  output logic        b_type_instr_o,
  output logic        u_type_instr_o,
  output logic        j_type_instr_o,
  output logic [31:0] instr_imm_o   // reconstructed sign-extended immediate
);

  // ── Field extraction (direct wiring) ──────────────────
  assign rs1_o    = instr_i[19:15];
  assign rs2_o    = instr_i[24:20];
  assign rd_o     = instr_i[11:7];
  assign funct3_o = instr_i[14:12];
  assign funct7_o = instr_i[31:25];
  assign op_o     = instr_i[6:0];

  // ── Type decode and immediate reconstruction ──────────
  always_comb begin
    r_type_instr_o = 1'b0; i_type_instr_o = 1'b0;
    s_type_instr_o = 1'b0; b_type_instr_o = 1'b0;
    u_type_instr_o = 1'b0; j_type_instr_o = 1'b0;
    instr_imm_o    = 32'b0;

    case (op_o)
      7'b0110011: begin  // R-type (0x33)
        instr_imm_o    = 32'b0;
        r_type_instr_o = 1'b1;
      end
      7'b0000011,          // I-type load (0x03)
      7'b0010011,          // I-type arith (0x13)
      7'b1100111: begin    // JALR (0x67)
```

```
        instr_imm_o    = {{20{instr_i[31]}}, instr_i[31:20]};
        i_type_instr_o = 1'b1;
      end
      7'b0100011: begin  // S-type (0x23)
        instr_imm_o    = {{20{instr_i[31]}}, instr_i[31:25], instr_i[11:7]};
        s_type_instr_o = 1'b1;
      end
      7'b1100011: begin  // B-type (0x63)
        instr_imm_o    = {{20{instr_i[31]}}, instr_i[7], instr_i[30:25], instr_i[11:8]
        b_type_instr_o = 1'b1;
      end
      7'b0010111,          // AUIPC (0x17)
      7'b0110111: begin   // LUI   (0x37)
        instr_imm_o    = {instr_i[31:12], 12'b0};
        u_type_instr_o = 1'b1;
      end
      7'b1101111: begin  // J-type JAL (0x6F)
        instr_imm_o    = {{11{instr_i[31]}}, instr_i[31], instr_i[19:12],
                          instr_i[20], instr_i[30:21], 1'b0};
        j_type_instr_o = 1'b1;
      end
      default: ; // all zeros (NOP)
    endcase
  end

endmodule
```

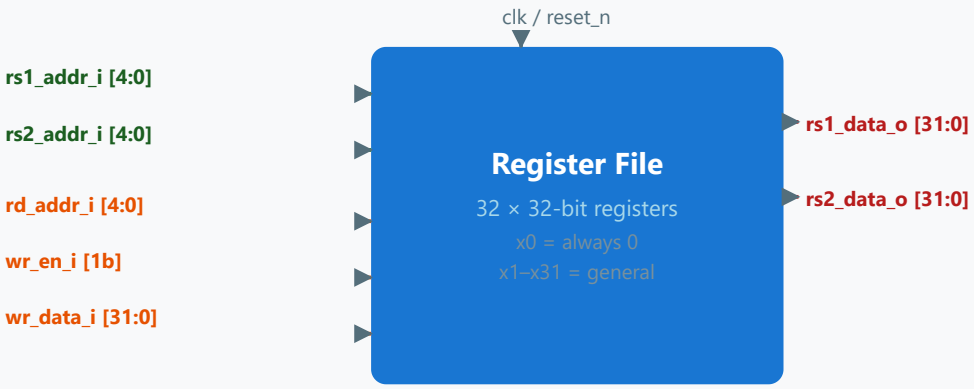
CHAPTER 9

Register File — yarp_regfile

9.1 Interface & Requirements

The Register File is the **central data store** of the processor. For every instruction, it simultaneously provides data from up to two source registers (rs1, rs2) and accepts a write-back value into the destination register (rd) — all in the same cycle.

FIGURE 9.1 — REGISTER FILE INTERFACE



Signal	Dir	Width	Description
`rs1_addr_i`	In	5b	Address of source register 1 (from decode)
`rs2_addr_i`	In	5b	Address of source register 2 (from decode)
`rd_addr_i`	In	5b	Destination register address (write port)
`wr_en_i`	In	1b	Write enable — from control unit
`wr_data_i`	In	32b	Data to write into rd (from ALU/memory/PC+4)
`rs1_data_o`	Out	32b	Data read from rs1 — goes to ALU operand A

`rs2_data_o`	Out	32b	Data read from rs2 — goes to ALU operand B or memory write data

9.2 X0 Immutability & Read Latency

Requirement	RV32I Specification	YARP Implementation
X0 value	Always reads 0, writes ignored	Write guarded: only write if rd_addr_i ≠ 0. Read: always returns 0 for addr=5'b0
Read latency	Combinational (same cycle)	Assign statements — no clock needed for reads
Write timing	Registered — takes effect on next cycle or same cycle for single-cycle	posedge clk write; combinational read handles read-after-write

9.3 RTL Implementation

```
module yarp_regfile (
  input  logic        clk, reset_n,
  input  logic [4:0] rs1_addr_i, rs2_addr_i,
  input  logic [4:0] rd_addr_i,
  input  logic        wr_en_i,
  input  logic [31:0] wr_data_i,
  output logic [31:0] rs1_data_o, rs2_data_o
);

  // 32 registers, each 32-bit wide
  logic [31:0] regfile [31:0];

  // ── Write port (registered) ──────────────────────
  always_ff @(posedge clk, negedge reset_n) begin
    if (~reset_n) begin
      for (int i = 0; i < 32; i++) regfile[i] <= 32'b0;
    end else begin
      // Write protection for X0: only write if rd_addr_i != 0
      if (wr_en_i && (rd_addr_i != 5'b0))
        regfile[rd_addr_i] <= wr_data_i;
    end
  end

  // ── Read ports (combinational — zero-latency) ────
  // X0 always returns 0 regardless of stored value
  assign rs1_data_o = (rs1_addr_i == 5'b0) ? 32'b0 : regfile[rs1_addr_i];
  assign rs2_data_o = (rs2_addr_i == 5'b0) ? 32'b0 : regfile[rs2_addr_i];
```

```
endmodule
```

Four Key Design Decisions

1. X0 is hardwired to zero via write-guard AND read-mux. 2. Combinational reads eliminate read-after-write hazards in single-cycle. 3. X0 forced to return 0 on reads. 4. Eliminates read-after-write hazard since read is purely combinational.

CHAPTER 10

Execute Stage — ALU (yarp_execute)

10.1 ALU Operations

The ALU (Arithmetic Logic Unit) is a **pure combinational block** that takes two 32-bit operands and a 4-bit operation selector, and outputs a 32-bit result. In YARP, the ALU covers all integer computation defined in the RV32I base ISA.

ALU Op	Operation	RV32I Instructions
OP_ADD	A + B	ADD, ADDI, LW, SW, AUIPC, JAL, JALR, branches
OP_SUB	A − B	SUB
OP_SLL	A << B[4:0]	SLL, SLLI
OP_SRL	A >> B[4:0] (logical)	SRL, SRLI
OP_SRA	A >>> B[4:0] (arithmetic)	SRA, SRAI
OP_OR	A | B	OR, ORI
OP_AND	A & B	AND, ANDI
OP_XOR	A ^ B	XOR, XORI
OP_SLTU	A < B (unsigned) → 1 or 0	SLTU, SLTIU
OP_SLT	A < B (signed) → 1 or 0	SLT, SLTI

10.2 Signed vs Unsigned Operations

SLTU / SLTIU — Unsigned Comparison

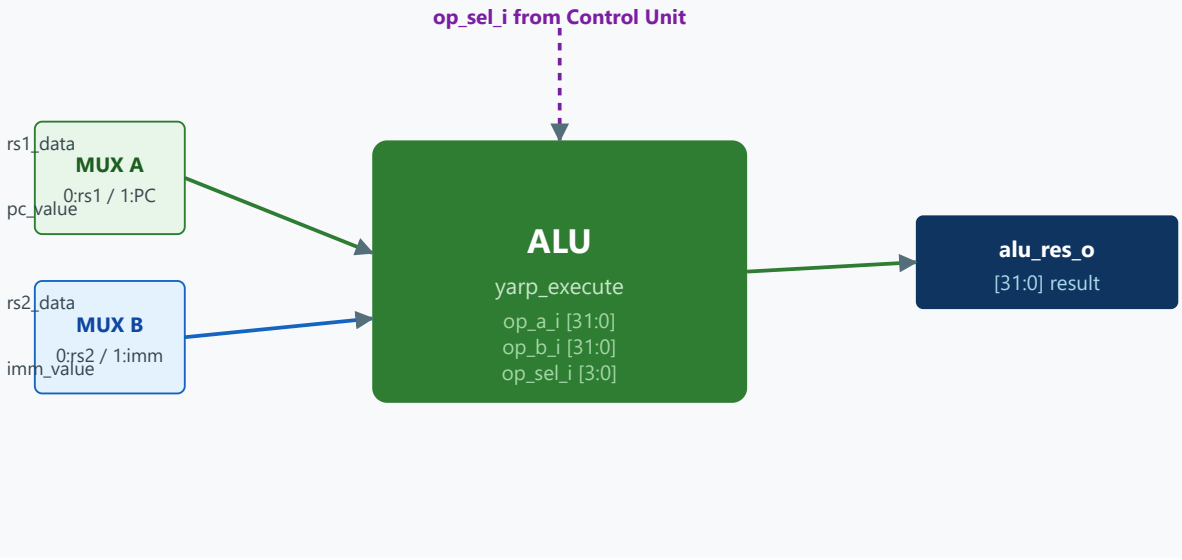
Treats both operands as unsigned 32-bit integers. Comparison happens on the raw bit pattern (no sign extension).

SLT / SLTI — Signed Comparison

Treats both operands as 2's complement signed integers. If bit[31] = 1, the value is

negative. A signed negative value is always less than any positive value.

FIGURE 10.1 — ALU BLOCK WITH OPERAND MUXES



10.3 RTL Implementation

```
module yarp_execute import yarp_pkg::*; (
  input  logic [31:0] opr_a_i,    // operand A (rs1 or PC)
  input  logic [31:0] opr_b_i,    // operand B (rs2 or imm)
  input  logic [3:0]  op_sel_i,   // ALU operation select
  output logic [31:0] alu_res_o   // 32-bit ALU result
);

  logic [31:0] twos_compl_a, twos_compl_b, alu_res;

  // 2's complement for signed operands
  assign twos_compl_a = ~opr_a_i[31] ? opr_a_i + 32'h1 : opr_a_i;
  assign twos_compl_b = ~opr_b_i[31] ? opr_b_i + 32'h1 : opr_b_i;

  always_comb begin
    case (op_sel_i)
      OP_ADD : alu_res = opr_a_i + opr_b_i;
      OP_SUB : alu_res = opr_a_i - opr_b_i;
      OP_SLL : alu_res = opr_a_i << opr_b_i[4:0];
      OP_SRL : alu_res = opr_a_i >> opr_b_i[4:0];
      OP_SRA : alu_res = $signed(opr_a_i) >>> opr_b_i[4:0];
      OP_OR  : alu_res = opr_a_i | opr_b_i;
      OP_AND : alu_res = opr_a_i & opr_b_i;
```

```
      OP_XOR : alu_res = opr_a_i ^ opr_b_i;
      OP_SLTU: alu_res = (opr_a_i < opr_b_i) ? 32'h1 : 32'h0;
      OP_SLT : begin
        if (opr_a_i[31] == opr_b_i[31])
          alu_res = (twos_compl_a < twos_compl_b) ? 32'h0 : 32'h0;
        else
          alu_res = opr_a_i[31] ? 32'h1 : 32'h0; // negative < positive
      end
      default: alu_res = opr_a_i + opr_b_i;
    endcase
  end

  assign alu_res_o = alu_res;

endmodule
```

CHAPTER 11

Data Memory — yarp_data_mem

11.1 Load & Store Instructions

The Data Memory stage handles the two instruction classes that access external memory:

Load Instructions

Read data FROM memory into a register. The effective address is computed by the ALU: `rs1 + sign_ext(imm)`. LW, LH, LHU, LB, LBU.

Store Instructions

Write data FROM a register TO memory. Address = `rs1 + sign_ext(imm)`. SW, SH, SB. No register write-back occurs.

All other arithmetic instructions (ADD, AND, JAL, etc.) skip this stage and route the ALU result directly to the register write-back path.

11.2 Byte, Half-Word, Word Access

RV32I supports three access widths. Memory is always **byte-addressable**. Address alignment constraints apply:

Instruction	Access Width	Bytes	Address Alignment	Sign Extend?
LW / SW	Word	4	addr[1:0] = 2'b00 (4-byte aligned)	N/A (full 32 bits)
LH / SH	Half-word	2	addr[0] = 0 (2-byte aligned)	LH: sign-extended to 32b
LHU	Half-word	2	addr[0] = 0	Zero-extended to 32b
LB / SB	Byte	1	Any address	LB: sign-extended to 32b
LBU	Byte	1	Any address	Zero-extended to 32b

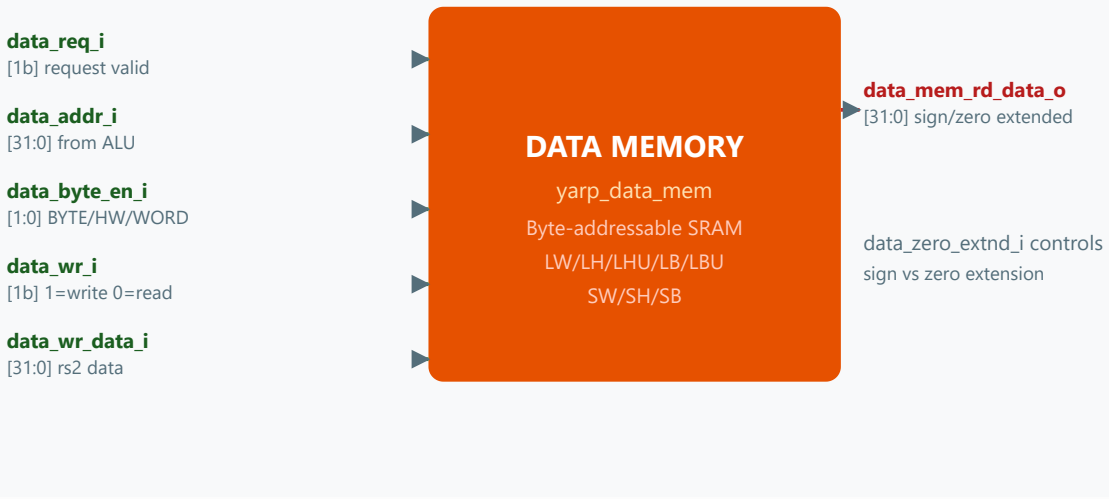
Alignment Rule — Word Store Example

Valid SW addresses (last 2 bits must be 00): 0x0000, 0x0004, 0x0008, 0x000C...

Valid SH addresses (last bit must be 0): 0x0000, 0x0002, 0x0004...

SB: any address is valid.

FIGURE 11.1 — DATA MEMORY INTERFACE



11.3 RTL Implementation

```
module yarp_data_mem import yarp_pkg::*; (
  input  logic        clk, reset_n,
  // Request from current instruction
  input  logic        data_req_i,
  input  logic [31:0] data_addr_i,
  input  logic [1:0]  data_byte_en_i, // 2'b00=byte, 2'b01=hw, 2'b11=word
  input  logic        data_wr_i,      // 1=write, 0=read
  input  logic [31:0] data_wr_data_i,
  input  logic        data_zero_extnd_i, // 0=sign-extend, 1=zero-extend
  // Output to memory bus
  output logic        data_mem_req_o,
  output logic [31:0] data_mem_addr_o,
  output logic [1:0]  data_mem_byte_en_o,
  output logic        data_mem_wr_o,
  output logic [31:0] data_mem_wr_data_o,
  // Read data from memory
  input  logic [31:0] mem_rd_data_i,
  output logic [31:0] data_mem_rd_data_o
);
```

```
  logic [31:0] rd_data_sign_extnd, rd_data_zero_extnd, data_mem_rd_data;

  // ── Sign / zero extension based on access size ──────────
  assign rd_data_sign_extnd =
    (data_byte_en_i == HALF_WORD) ? {{16{mem_rd_data_i[15]}}, mem_rd_data_i[15:0]} :
    (data_byte_en_i == BYTE)      ? {{24{mem_rd_data_i[7]}}},  mem_rd_data_i[7:0]}  :
                                     mem_rd_data_i;

  assign rd_data_zero_extnd =
    (data_byte_en_i == HALF_WORD) ? {16'b0, mem_rd_data_i[15:0]} :
    (data_byte_en_i == BYTE)      ? {24'b0, mem_rd_data_i[7:0]}  :
                                     mem_rd_data_i;

  // LHU/LBU → zero extend, LH/LB → sign extend
  assign data_mem_rd_data    = data_zero_extnd_i ? rd_data_zero_extnd : rd_data_sign_e

  // ── Output assignments (pass-through to memory bus) ─────
  assign data_mem_req_o     = data_req_i;
  assign data_mem_addr_o    = data_addr_i;
  assign data_mem_byte_en_o = data_byte_en_i;
  assign data_mem_wr_o      = data_wr_i;
  assign data_mem_wr_data_o = data_wr_data_i;
  assign data_mem_rd_data_o = data_mem_rd_data;

endmodule
```

CHAPTER 12

Control Unit — yarp_control

12.1 Control Signals Overview

The Control Unit is the "brain" of the datapath. It takes instruction type signals from the Decoder and generates all the mux-select and enable signals that route data correctly through the processor. It is entirely **combinational logic — no state registers**.

FIGURE 12.1 — CONTROL UNIT INPUTS AND OUTPUTS

12.2 MUX Selects & Data Routing

Control Signal	Width	Meaning
`pc_sel_o`	1b	0 = PC+4 (sequential) | 1 = Branch/Jump target from ALU
`op1sel_o`	1b	0 = rs1 register data | 1 = Current PC (for AUIPC, JAL)
`op2sel_o`	1b	0 = rs2 register data | 1 = Sign-extended immediate

`alu_func_o`	4b	ALU operation select — matches OP_ADD, OP_SUB... enum
`rf_wr_en_o`	1b	1 = instruction writes to register file, 0 = no write (S/B type)
`rf_wr_data_o`	2b	00=ALU result | 01=DMEM read data | 10=32-bit immediate | 11=PC+4
`data_req_o`	1b	1 = this instruction accesses data memory (Load or Store)
`data_wr_o`	1b	1 = write to memory (Store) | 0 = read from memory (Load)
`data_byte_o`	2b	Access size: BYTE=2'b00, HALF_WORD=2'b01, WORD=2'b11
`zero_extnd_o`	1b	1 = zero-extend load (LHU, LBU) | 0 = sign-extend

12.3 Complete Control Truth Table

Instruction Type	pc_sel	op1sel	op2sel	alu_func	rf_wr_en	rf_wr_data	data_req	data_wr
R-type (ADD/SUB...)	PC+4	rs1	rs2	funct-derived	1	ALU	0	—
I-type arith (ADDI...)	PC+4	rs1	imm	funct-derived	1	ALU	0	—
I-type load (LW...)	PC+4	rs1	imm	ADD	1	DMEM	1	0
S-type (SW...)	PC+4	rs1	imm	ADD	0	—	1	1
B-type (BEQ...)	branch	rs1	imm	ADD	0	—	0	—
U-type LUI	PC+4	PC	imm	passthrough	1	IMM	0	—
U-type AUIPC	PC+4	PC	imm	ADD	1	ALU	0	—
J-type JAL	jump	PC	imm	ADD	1	PC+4	0	—
I-type JALR	jump	rs1	imm	ADD	1	PC+4	0	—

CHAPTER 13

Worked Execution Examples

Example 1: ADD R1, R2, R3

```
Instruction: ADD x1, x2, x3
Binary encoding:
  [31:25] funct7 = 0000000
  [24:20] rs2    = 00011  (x3)
  [19:15] rs1    = 00010  (x2)
  [14:12] funct3 = 000
  [11:7]  rd     = 00001  (x1)
  [6:0]   opcode = 0110011 (R-type = 0x33)

Execution trace:
  FETCH:    PC → IMEM → instr[31:0] returned
  DECODE:   opcode=0x33 → r_type. rs1=x2, rs2=x3, rd=x1, funct3=0, funct7=0
  RF READ:  rs1_data = regfile[x2], rs2_data = regfile[x3]
  CONTROL:  op1sel=rs1, op2sel=rs2, alu_func=OP_ADD, rf_wr_en=1, rf_wr_data=ALU
  ALU:      alu_res = rs1_data + rs2_data
  DMEM:     data_req=0 — skipped
  WB:       regfile[x1] ← alu_res  (on posedge clk)
  PC:       PC ← PC + 4
```

Example 2: LW x5, 8(x1)

```
Instruction: LW x5, 8(x1)   → Load word from address (x1 + 8) into x5
Encoding: I-type, opcode=0x03, funct3=010

Execution trace:
  FETCH:    PC → IMEM → LW instruction fetched
  DECODE:   opcode=0x03 → i_type (load). rs1=x1, rd=x5, imm=8
  RF READ:  rs1_data = regfile[x1]
  CONTROL:  op1sel=rs1, op2sel=imm, alu_func=OP_ADD, rf_wr_en=1, rf_wr_data=DMEM
            data_req=1, data_wr=0 (read)
  ALU:      alu_res = regfile[x1] + 8    → effective address
  DMEM:     data_mem_addr = alu_res → Mem[alu_res] returned as 32-bit word
```

```
  WB:      regfile[x5] ← dmem_rd_data  (sign-extended to 32 bits)
  PC:      PC ← PC + 4
```

Example 3: BEQ x1, x2, offset

```
Instruction: BEQ x1, x2, +16   → If x1==x2, branch PC+16
Encoding: B-type, opcode=0x63, funct3=000

Execution trace:
  FETCH:   PC → IMEM → BEQ instruction fetched
  DECODE:  opcode=0x63 → b_type. rs1=x1, rs2=x2, imm=16 (PC-relative offset)
  RF READ: rs1_data = regfile[x1], rs2_data = regfile[x2]
  CONTROL: op1sel=rs1, op2sel=rs2, alu_func=OP_SUB  (or dedicated comparator)
           rf_wr_en=0, data_req=0
           Branch Unit: (rs1_data == rs2_data) → branch_taken = 1
  ALU:     Target address = PC + sign_ext(imm) = PC + 16
  PC MUX:  pc_sel = branch_taken ? (PC+16) : (PC+4)
           → Since x1==x2: PC ← PC + 16
```

Example 4: SW x3, 12(x2)

```
Instruction: SW x3, 12(x2)   → Store word in x3 to memory address (x2 + 12)
Encoding: S-type, opcode=0x23, funct3=010

Execution trace:
  FETCH:   PC → IMEM → SW instruction fetched
  DECODE:  opcode=0x23 → s_type. rs1=x2, rs2=x3, imm={12[11:5],12[4:0]}=12
  RF READ: rs1_data = regfile[x2]  (base address register)
           rs2_data = regfile[x3]  (data to store)
  CONTROL: op1sel=rs1, op2sel=imm, alu_func=OP_ADD
           rf_wr_en=0, data_req=1, data_wr=1 (write), data_byte=WORD
  ALU:     alu_res = rs1_data + 12  → store address
  DMEM:    Mem[alu_res] ← rs2_data  (32-bit word write)
  WB:      No register write (S-type)
  PC:      PC ← PC + 4
```

CHAPTER 14

Glossary

Term	Definition
ALU	Arithmetic Logic Unit — performs all integer arithmetic and logic operations
AUIPC	Add Upper Immediate to PC — builds PC-relative addresses
B-type	Branch instruction format — PC-relative conditional branches
funct3	3-bit field at instr[14:12] — sub-type selector within an opcode
funct7	7-bit field at instr[31:25] — distinguishes ADD vs SUB, SRL vs SRA
I-type	Immediate instruction format — uses one source reg + 12-bit immediate
IMEM	Instruction Memory — read-only memory storing the program
In-Order	Execution model where instructions complete strictly in program order
ISA	Instruction Set Architecture — the contract between SW and HW
J-type	Jump instruction format — JAL, unconditional PC-relative jump
JALR	Jump and Link Register — indirect jump via rs1 + immediate
LUI	Load Upper Immediate — loads 20-bit constant into bits[31:12]
MUX	Multiplexer — selects one of N inputs based on a control signal
opcode	7-bit field at instr[6:0] — identifies instruction type
PC	Program Counter — 32-bit register holding the address of the current instruction
PLRU	Pseudo-LRU — approximate Least-Recently-Used replacement policy
R-type	Register-Register instruction format — both operands are registers
rd	Destination register — where the instruction result is written
RISC-V	Open-source, royalty-free ISA; "RISC" = Reduced Instruction Set Computer

rs1, rs2	Source registers 1 and 2 — provide operands to the ALU
RTL	Register-Transfer Level — the abstraction layer describing hardware in HDL
RV32I	RISC-V base integer ISA with 32-bit registers; "I" = integer only
S-type	Store instruction format — writes register data to memory
Sign Extension	Expanding a shorter value to 32 bits by replicating the sign bit
Single-Cycle	Processor where every instruction completes in exactly one clock cycle
SRA	Shift Right Arithmetic — fills with sign bit; preserves sign of value
SRL	Shift Right Logical — fills with zeros
SystemVerilog	IEEE 1800 hardware description and verification language
U-type	Upper-immediate format — 20-bit immediate in bits[31:12]; LUI and AUIPC
Write-Back	The final stage: writing the instruction result into the destination register
X0 (zero)	Register x0 — hardwired to 0; reads always return 0, writes are ignored
XLEN	The integer register width in bits: 32 for RV32I, 64 for RV64I
YARP	Yet Another RISC-V Processor — the single-cycle RV32I core documented here
Zero Extension	Expanding a shorter value to 32 bits by filling upper bits with 0s

RV32I Single-Cycle Processor Design — YARP

Yet Another RISC-V Processor · From ISA Foundations to Full SystemVerilog RTL

To save as PDF: Open in browser → Print → Save as PDF → Enable Background Graphics