RISC-V

System-on-Chip Design

Harris, Stine, Thompson, & Harris

Chapter 7:

Pipelined Core

Chapter 7 :: Topics

Pipelined Core

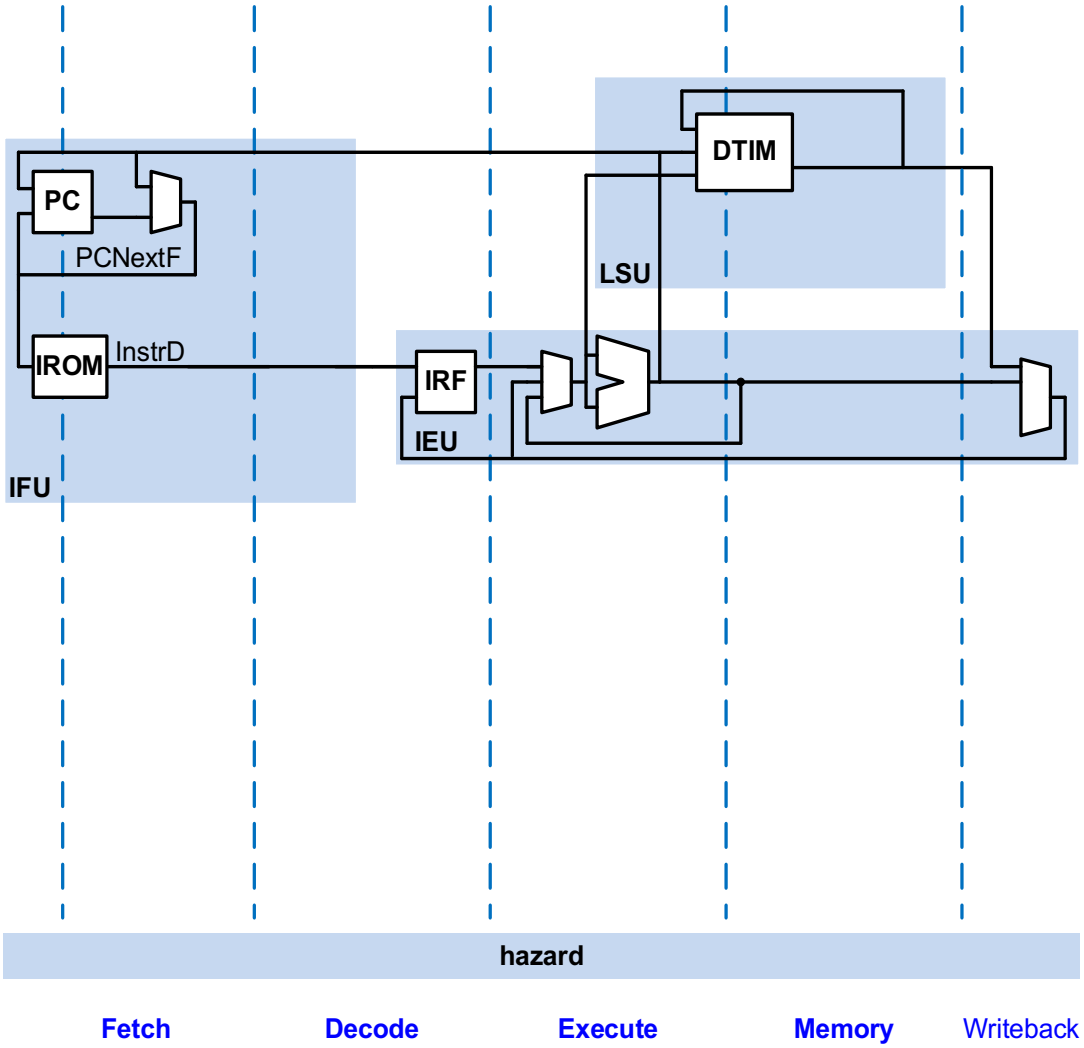
7.1 Single-Cycle Core

7.2 Pipelined RV32I Core

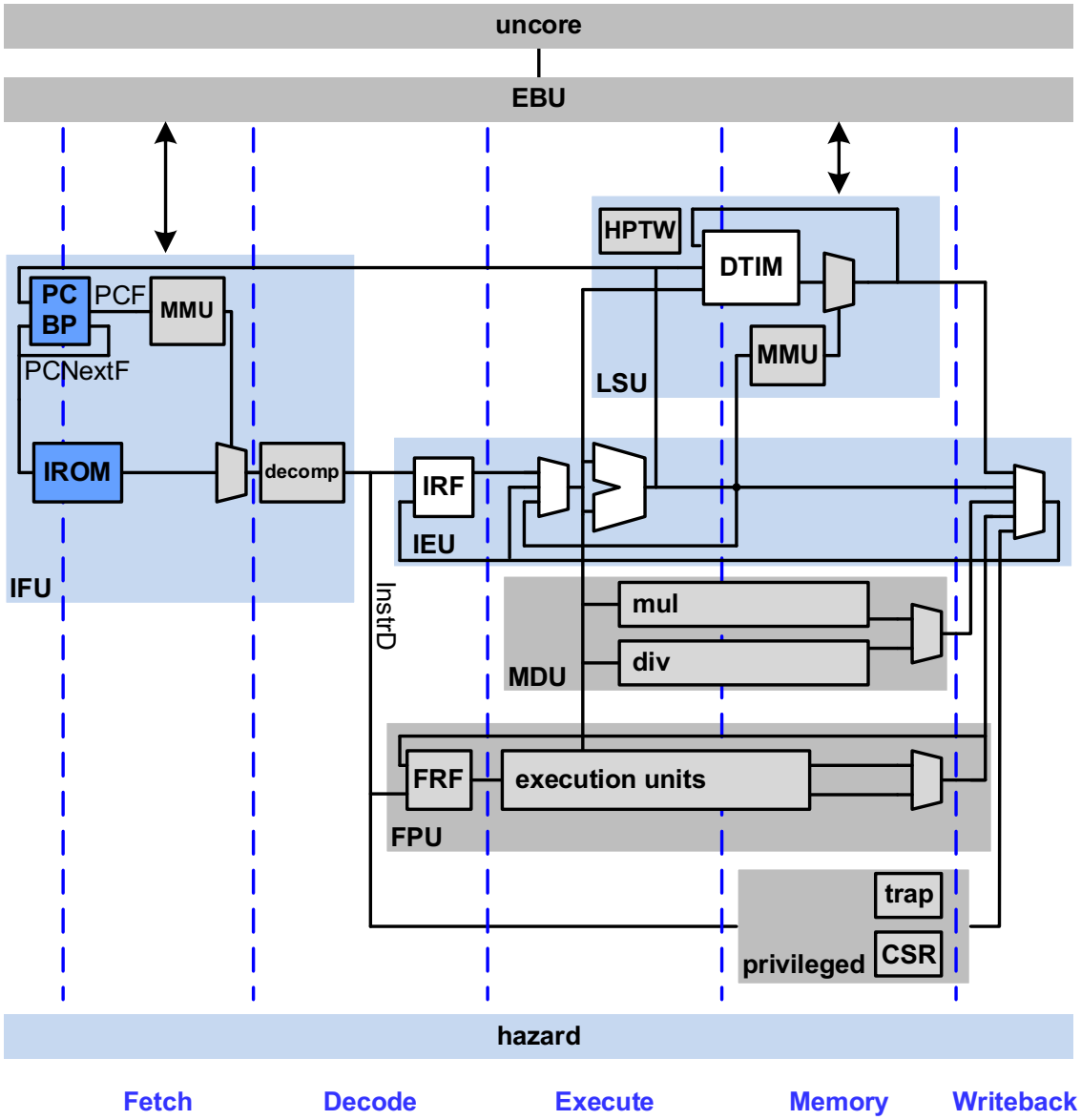
Simplified Pipelined Core

- **RV32I** or **RV64I**
- Assumes **ideal instruction & data memories**
  - IROM instruction read-only memory
  - DTIM: data tightly-integrated memory
- **No bus interface** (for external memory or peripherals)
- **Trivial branch predictor:** predicts not taken
- **No MDU** (multiply-divide unit)
- **No FPU** (floating-point unit)
- **No privileged unit, so:**
  - Always run in machine-mode (M-Mode)
  - No traps
  - No virtual memory
  - Doesn't support CSRs

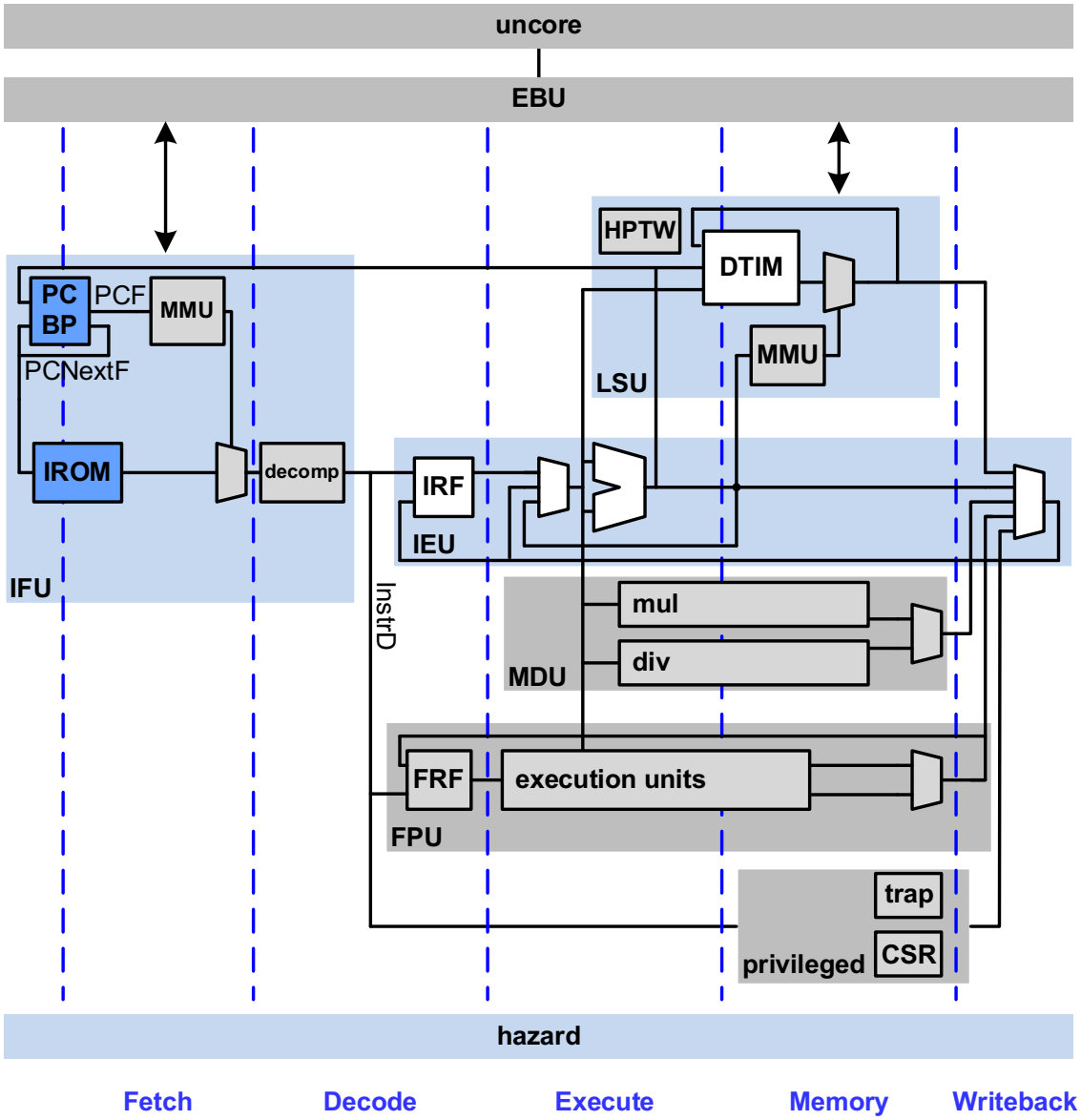
Simplified Pipelined Core



Simplified Pipelined Core



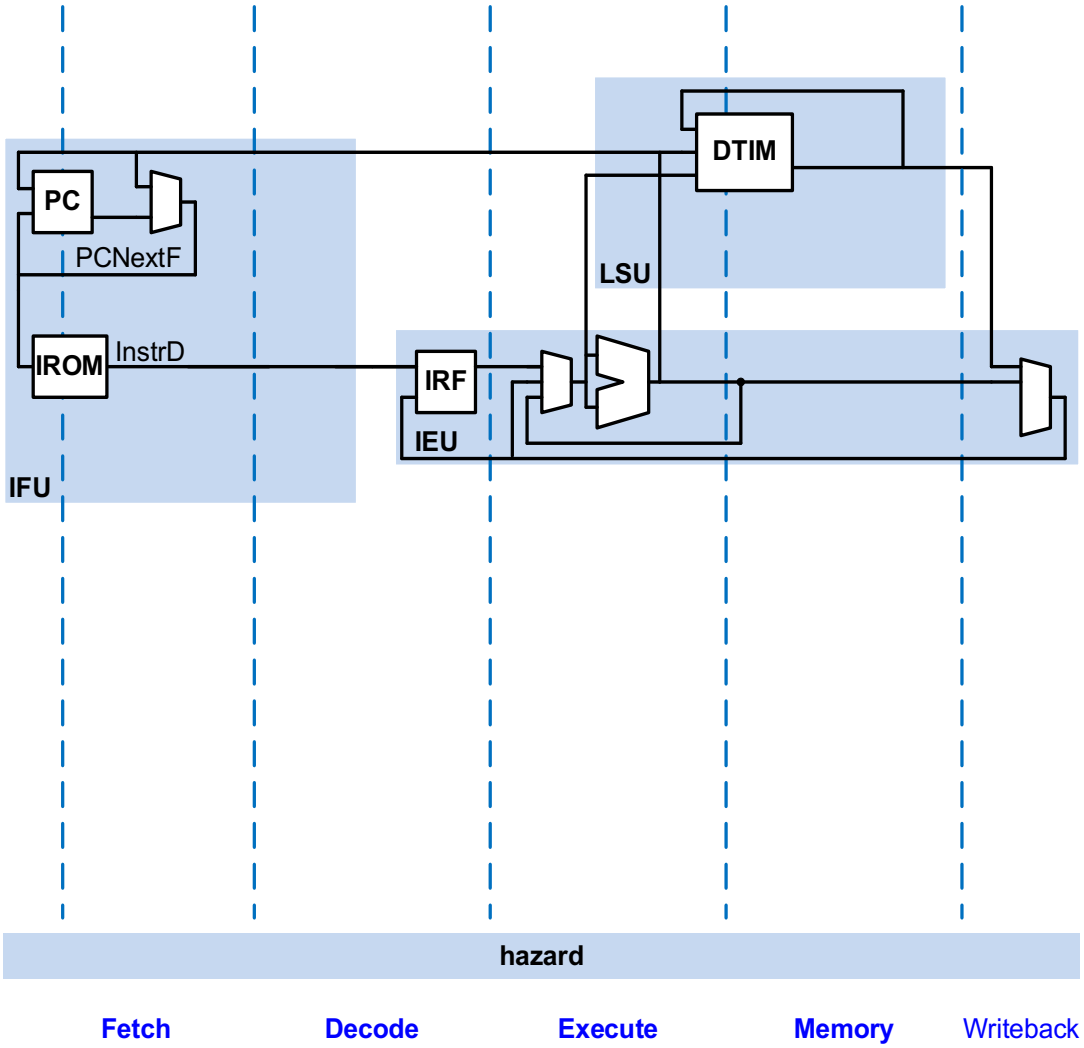
Simplified Pipelined Core



Chapters:

- **8 – PRIV:** Privileged Unit
- **9 – EBU:** External Bus Unit
- **10 – Cache**
- **11 – MMU:** Memory Management Unit
- **12 – LSU:** Load/Store Unit
- **13 – IFU:** Instruction Fetch Unit, branch predictors
- **14-19 – Extensions:**
  - **14 – C:** Compressed
  - **15 – M:** Multiply/Divide
  - **16 – FPU:** Floating-Point
  - **17 – AMO:** Atomic
  - **18 – B/Zk*:** Bit Manip & Crypto
  - **19 –** Other Extensions
- **20 – Peripherals**

Simplified Pipelined Core



Supports RV32I / RV64I Instructions

Category	Op	Format	RV32I/RV64I	RV64I Extra
R-Type ALU	51 / 59	R	add, sub, and, or, xor, sll, srl, sra, slt, sltu	addw, subw, sllw, srlw, sraw
I-Type ALU	19 / 27	I	addi, andi, ori, xori, slli, srli, srai, slti, sltiu	addiw, slliw, srliw, sraiw
Loads	3	I	lw, lh, lhu, lb, lbu	ld, lwu
Stores	35	S	sw, sh, sb	sd
Branches	99	B	beq, bne, blt, bge, bltu, bgeu	
Jump and Link	111	J	jal	
Jump and Link Register	103	I	jalr	
Load Upper Immediate	55	U	lui	
Add Upper Immediate PC	23	U	auipc	

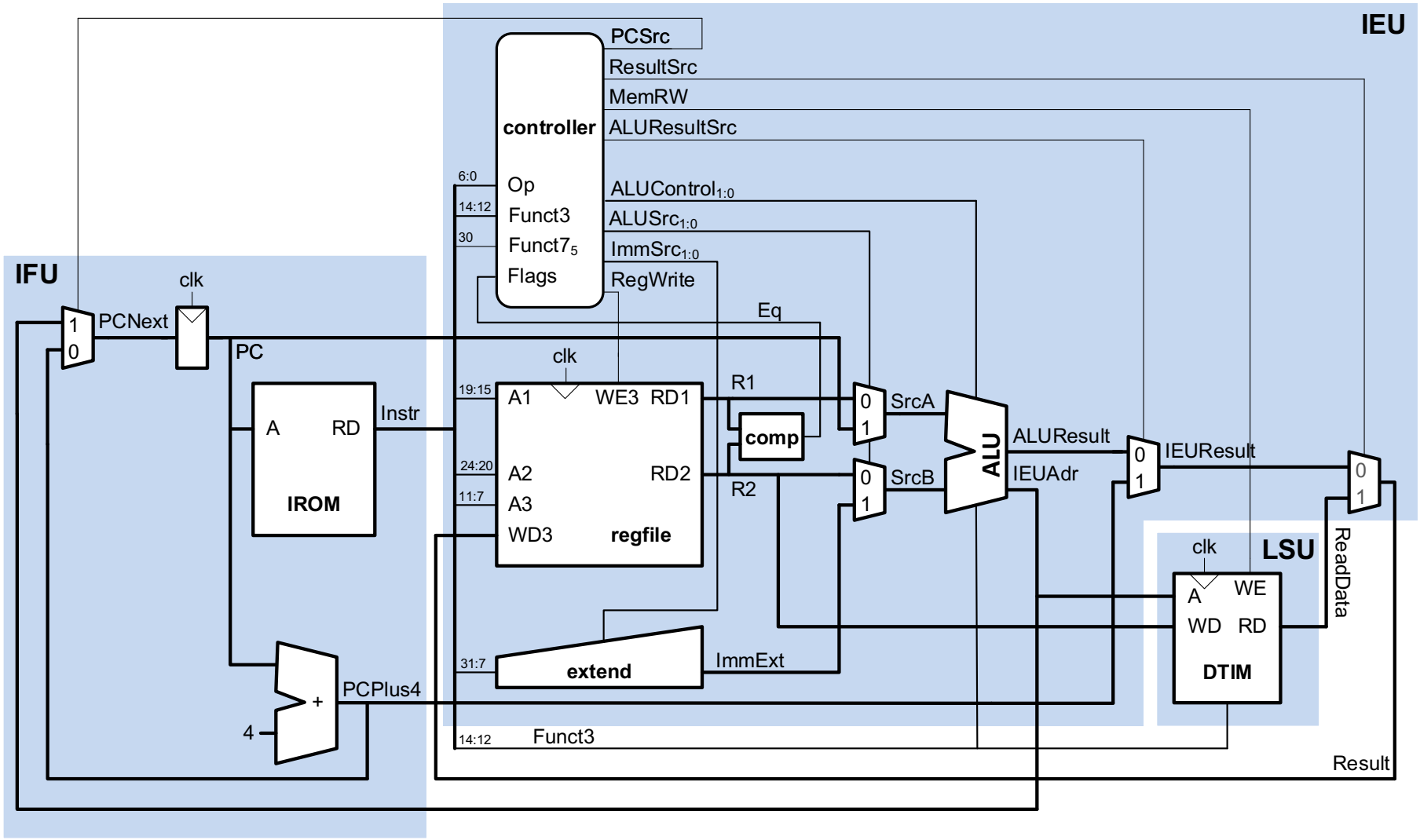
Single-Cycle → Pipelined Core

- Start by developing a single-cycle core
- Then convert it into a pipelined core

Chapter 7: Pipelined Core

Single-Cycle Core

Simple Single-Cycle Core from Ch. 2



Supports: `add`, `sub`, `and`, `or`, `slt`, `addi`, `andi`, `ori`, `slti`, `lw`, `sw`, `beq`, `jal`

Extend Simple Core From Ch. 2

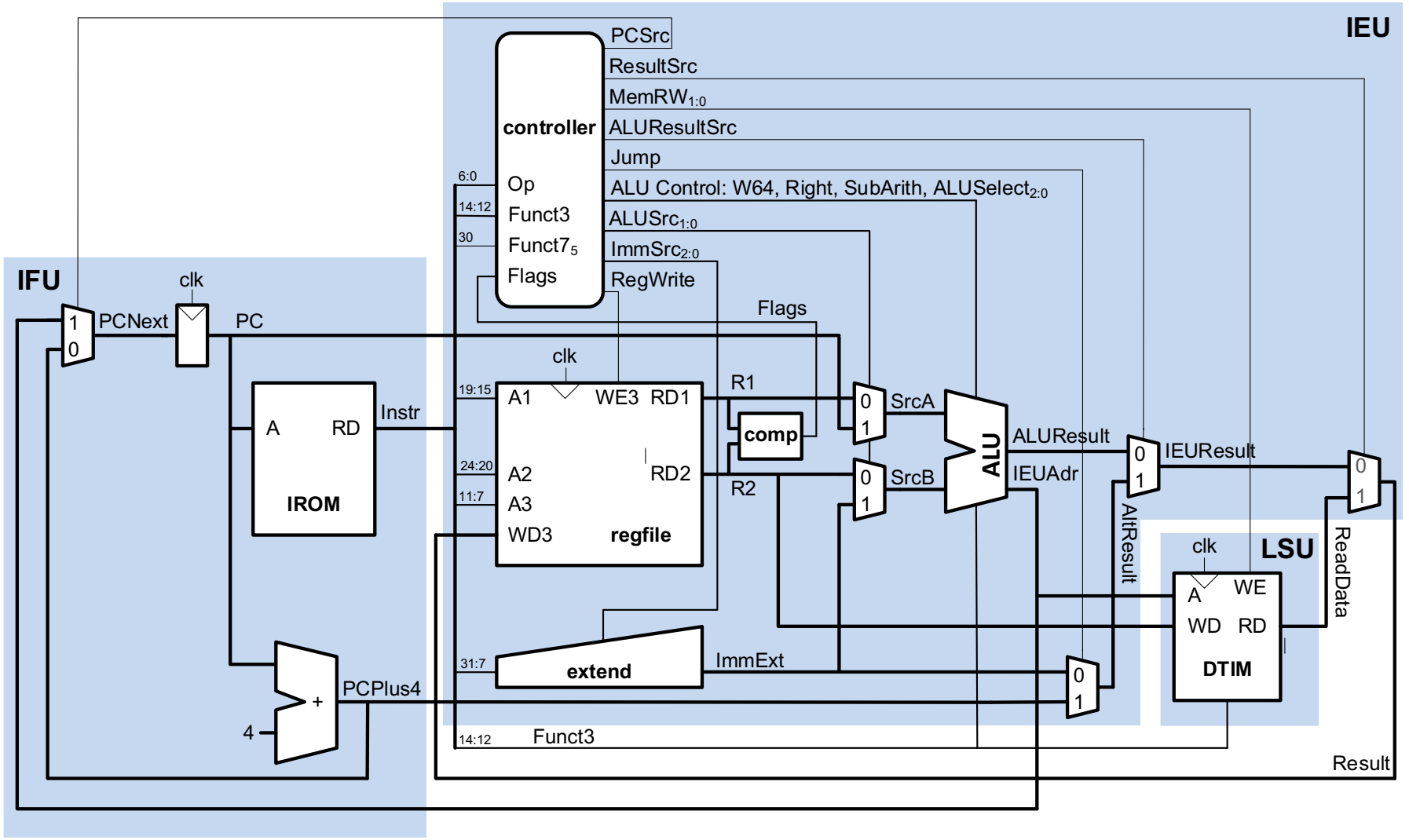
Support all RV32I/RV64I Instructions:

- More R- and I-type ALU instructions, including RV64's W instructions, which operate on 32-bit values
- Byte, half-word, and double-word loads and stores
- More flavors of branches
- Upper immediate instructions: `lui` and `auipc`
- Jump and link register: `jalr`

Baseline vs. New Instructions

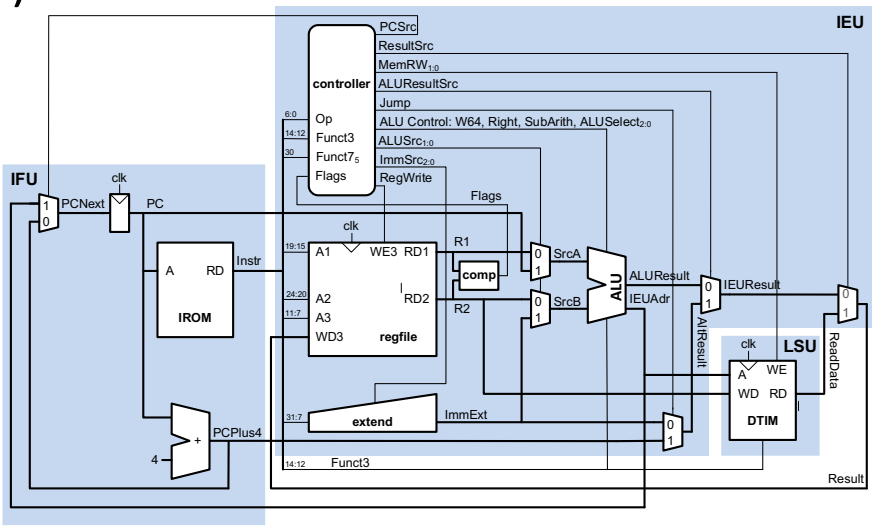
Category	Op	Format	RV32I/RV64I	RV64I Extra
R-Type ALU	51 / 59	R	add, sub, and, or, xor, sll, srl, sra, slt, sltu	addw, subw, sllw, srlw, sraw
I-Type ALU	19 / 27	I	addi, andi, ori, xori, slli, srli, srai, slti, sltiu	addiw, slliw, srliw, sraiw
Loads	3	I	lw, lh, lhu, lb, lbu	ld, lwu
Stores	35	S	sw, sh, sb	sd
Branches	99	B	beq, bne, blt, bge, bltu, bgeu	
Jump and Link	111	J	jal	
Jump and Link Register	103	I	jalr	
Load Upper Immediate	55	U	lui	
Add Upper Immediate PC	23	U	auipc	

Extended Simple Single-Cycle Core



Enhancements

- **Configurable datapath:**
  - XLEN = 32 or 64 bits (for RV32 or RV64)
- **ALU enhanced:**
  - Supports **additional operations**
    - xor, set less than unsigned, shifts
  - Has an **extra bit of ALUControl** to specify RV64's RW/IW instruction
    - e.g. `addw` and `addiw`
- LSU adds circuitry for **subword loads and stores**
- Comparator generates **more flags**: equal, less than, less than unsigned (to handle all the flavors of branches)
- Extender enhanced to support **U-format** instructions
- The controller adds entries for the **new instruction categories:** `jalr`, `lui`, and `auipc`.

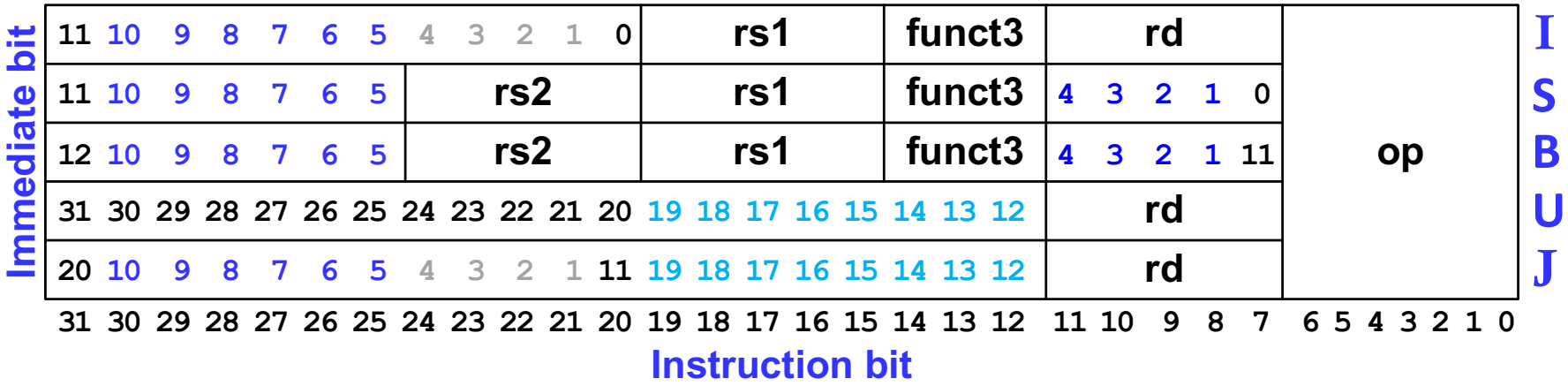


Architectural State

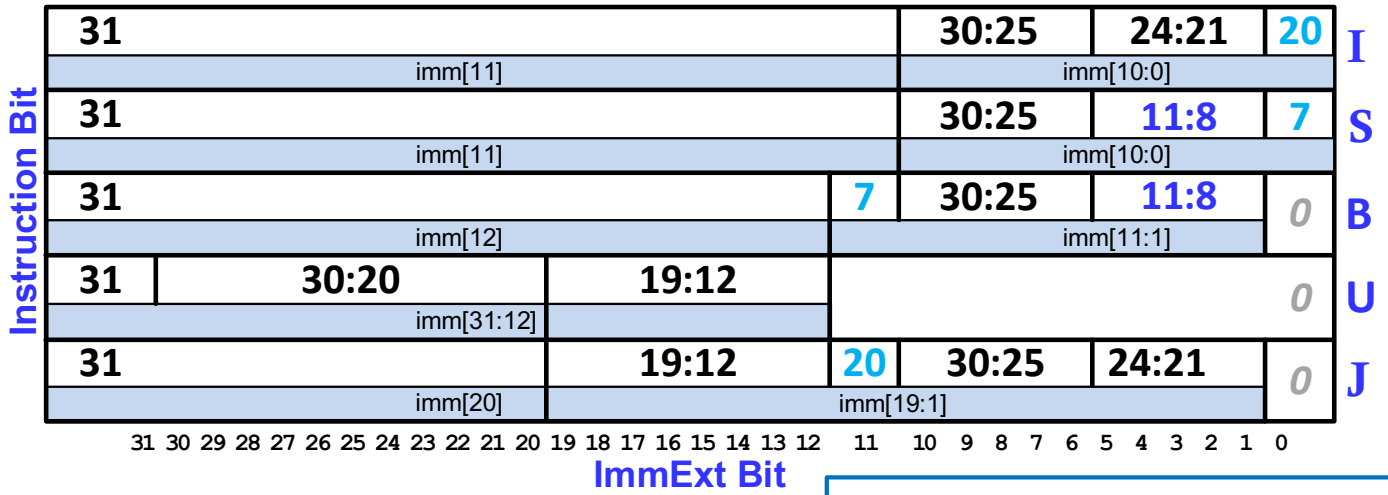
- **PC** (program counter)
- **Integer registers** (x1 - x31)

Immediate Encodings & Extensions

Instruction Bits



Immediate Extension

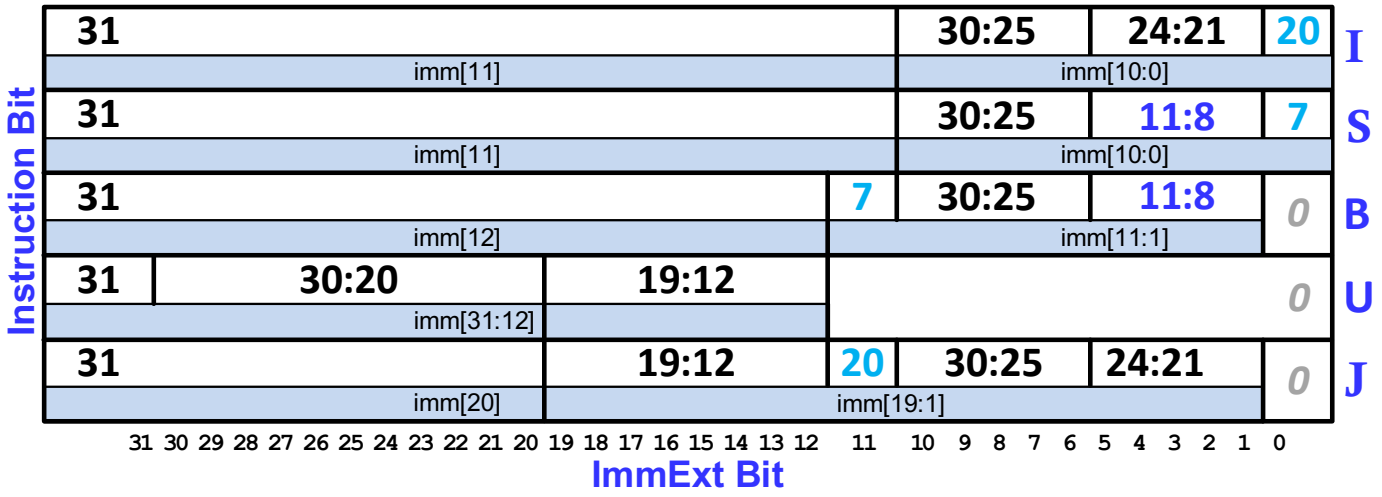


12-, 13-, 21-, or 32-bit immediates

Immediate Extensions

ImmSrc	ImmExt	Type	Description
000	{{20{Instr[31]}}, Instr[31:20]}	I	12-bit signed
001	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S	12-bit signed
010	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}	B	13-bit signed
011	{{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}	J	21-bit signed
100	{Instr[31:12], 12'b0}	U	32-bit signed

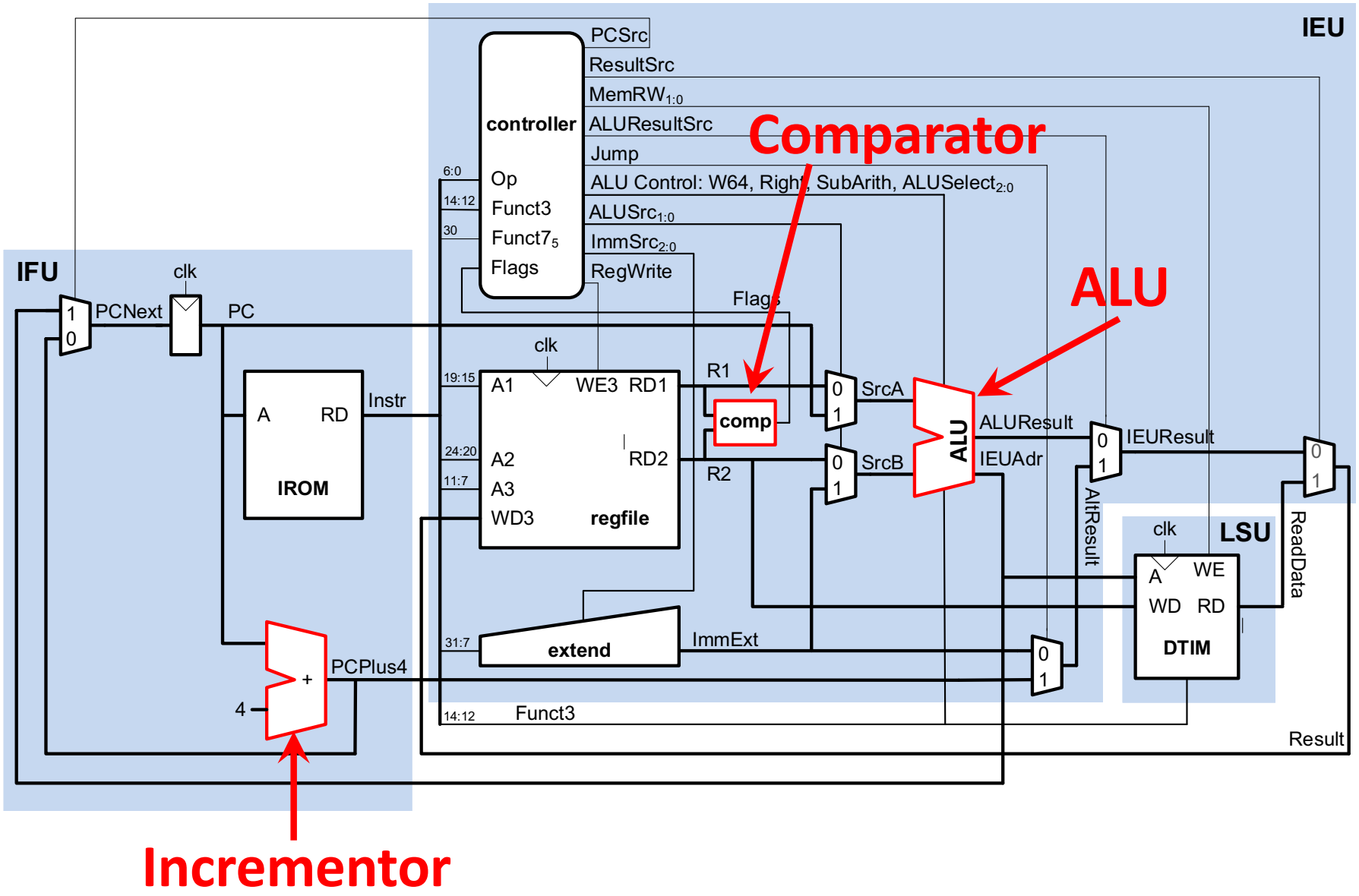
Immediate Extension



Arithmetic Circuitry

- **ALU:** Perform an arithmetic, shift, or logical operation and address calculation
- **Comparator:** Compare two registers to produce flags used for branches
- **Incrementor:** Increment the PC by 4 to calculate the next PC

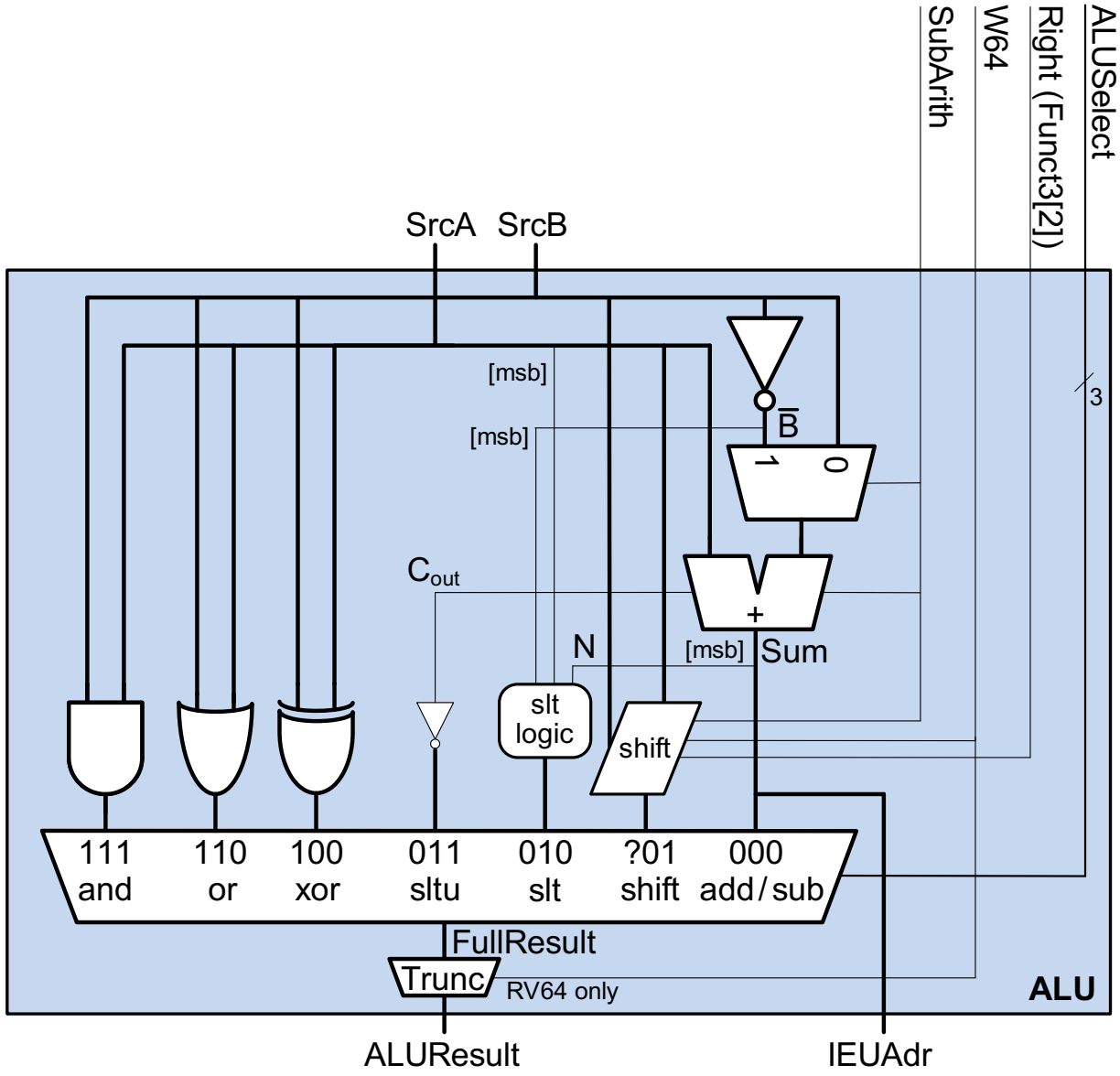
Arithmetic blocks



ALU Operations

Category	ALUResult	IEUResult	IEUAdr Destination	Comparator
R-Type ALU	R1 op R2	ALUResult	x	
I-Type ALU	R1 op ImmExt	ALUResult	x	
Loads/Stores	x	x	R1 + ImmExt → LSU	
Branches	x	x	PC + ImmExt → IFU	R1 cmp R2
`jal`	x	PCPlus4	PC + ImmExt → IFU	
`jalr`	x	PCPlus4	R1 + ImmExt → IFU	
`lui`	x	Imm	x	
`auipc`	PC + ImmExt	ALUResult	x	

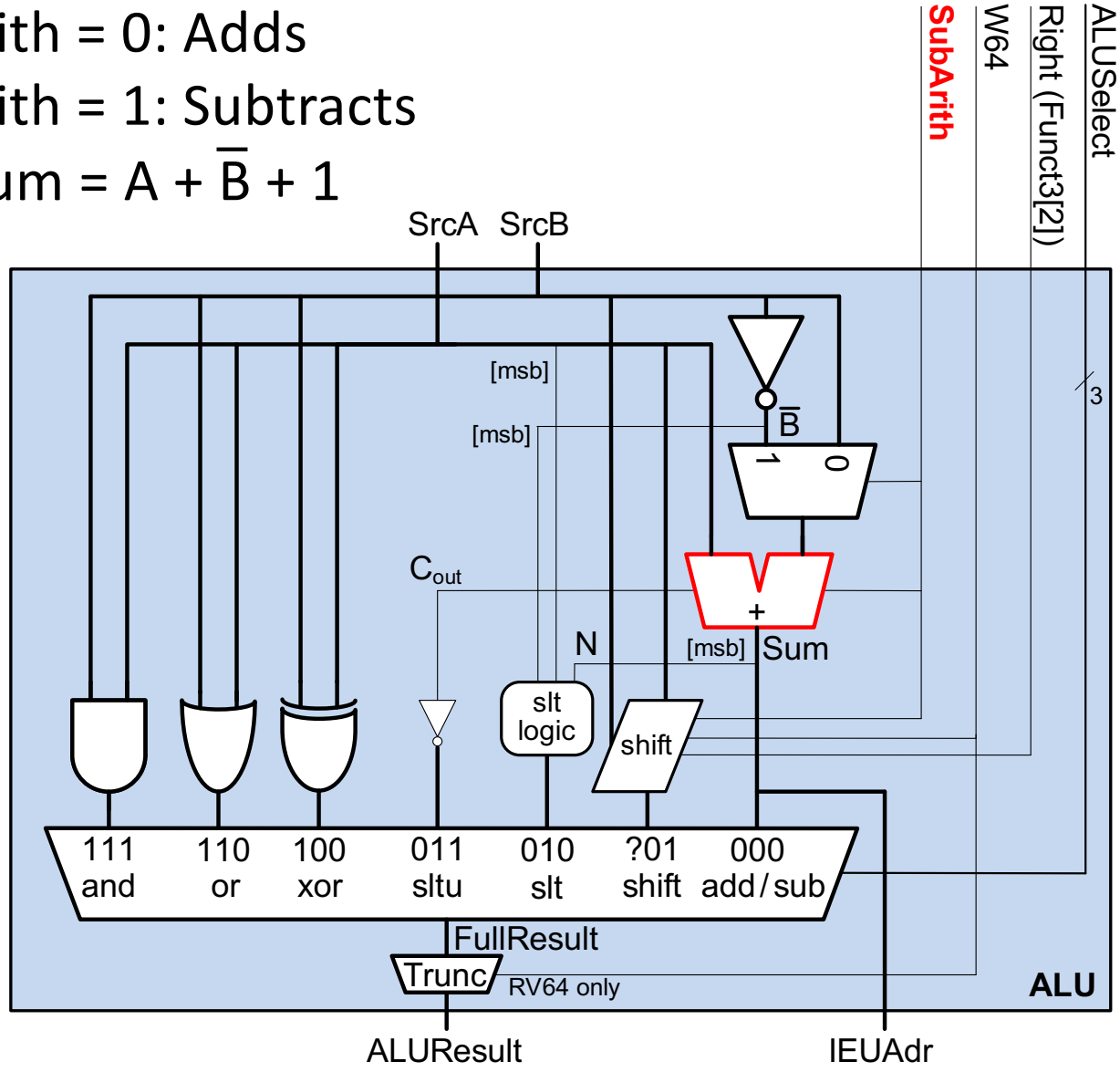
ALU Schematic



Adder Operations

- SubArith = 0: Adds
- SubArith = 1: Subtracts

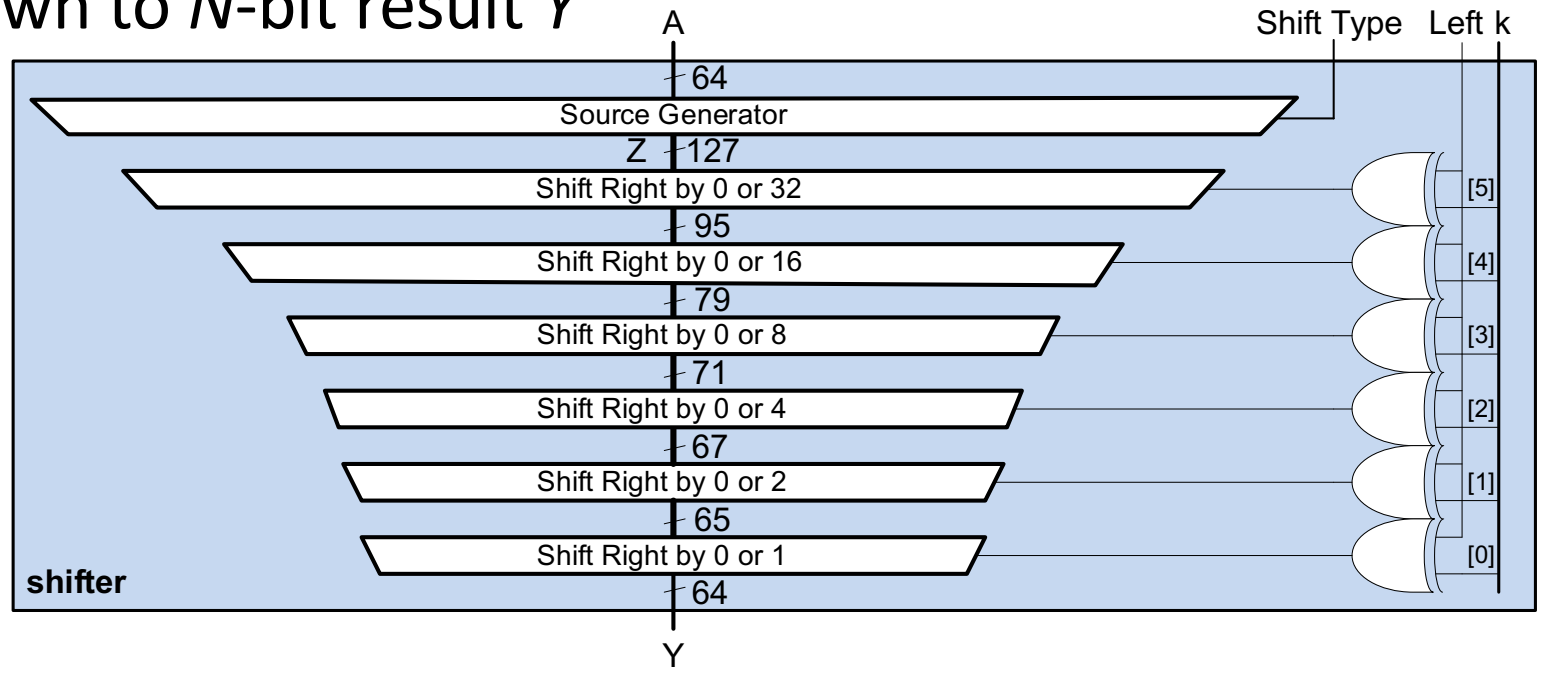
  Sum = A + $\overline{B}$ + 1



Shifter

- **Funnel shifter**
  - Source generator creates 2*N*-1 bit signal *Z* from *A*
  - Mux tree funnels down to *N*-bit result *Y*

Shift Type	Z	Shift Amount
srl	{63'b0, $A_{63:0}$}	k
sra	{{63{A_{63}}}, $A_{63:0}$}	k
sll	{$A_{63:0}$, 63'b0}	~k
srlw	{95'b0, $A_{31:0}$}	k
sraw	{64'b0, {31{A_{31}}}, $A_{31:0}$}	k
sllw	{32'b0, $A_{31:0}$, 63'b0}	~k



Comparator

- **Handles `eq`, `lt`, `ltu`**
  - Use signed for `lt` vs. unsigned for `ltu`

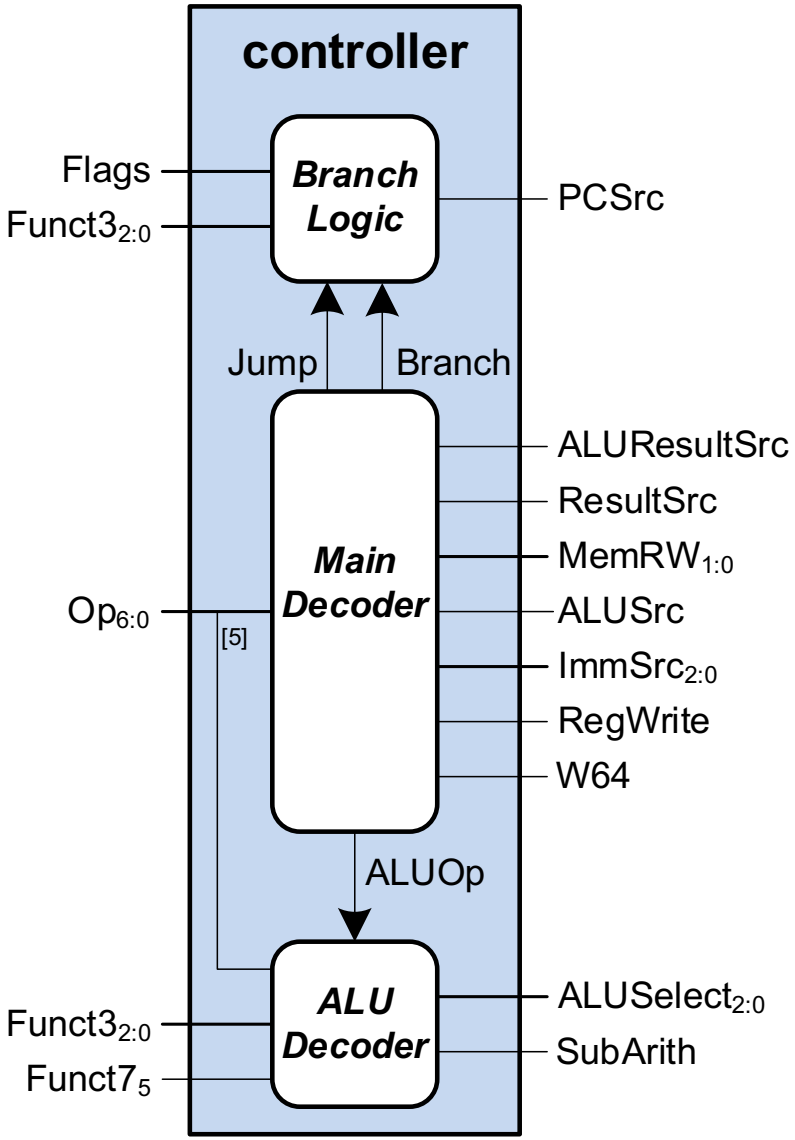
```
module comparator #(parameter WIDTH=64) (
  input  logic [WIDTH-1:0] a, b, // Operands
  input  logic sgnd,             // Signed operands
  output logic [1:0] flags);     // Output flags: {eq, lt}

  logic             eq, lt; // Flags: equal (eq), less than (lt)
  logic [WIDTH-1:0] af, bf; // Operands with msb flipped when signed

  // For signed numbers, flip most significant bit
  assign af = {a[WIDTH-1] ^ sgnd, a[WIDTH-2:0]};
  assign bf = {b[WIDTH-1] ^ sgnd, b[WIDTH-2:0]};

  // Behavioral description gives best results
  assign eq = (a == b); / / eq = 1 when operands are equal, 0 otherwise
  assign lt = (af < bf); // lt = 1 when a less than b (accounting for sign)
  assign flags = {eq, lt};
endmodule
```

Controller


controller
Branch Logic
Flags
Funct3$_{2:0}$
PCSrc
Jump
Branch
Main Decoder
Op$_{6:0}$
[5]
ALUResultSrc
ResultSrc
MemRW$_{1:0}$
ALUSrc
ImmSrc$_{2:0}$
RegWrite
W64
ALUOp
ALU Decoder
Funct3$_{2:0}$
Funct7$_5$
ALUSelect$_{2:0}$
SubArith

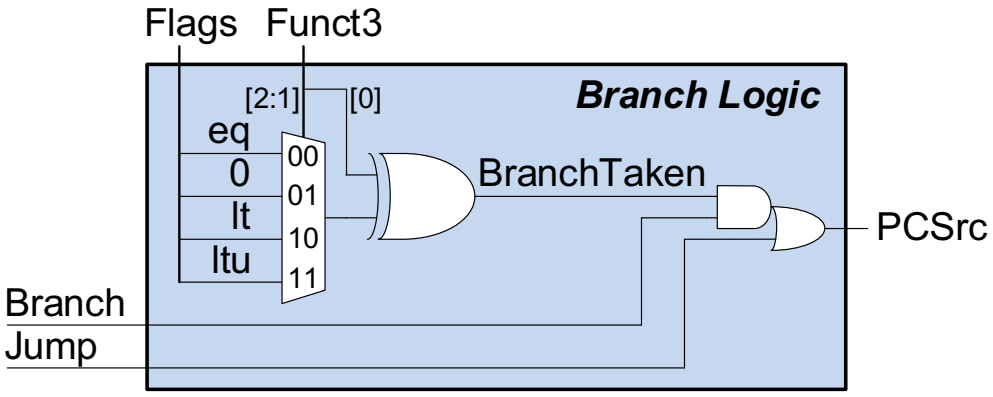
Main Decoder

Category	op	Branch	Jump	ALUSrc	ImmSrc	ALUOp	W64	ALUResultSrc	ResultSrc	RegWrite	MemRW
R-Type ALU	51	0	0	00	xxx	1	0	0	0	1	00
I-Type ALU	19	0	0	01	000	1	0	0	0	1	00
RW-Type ALU*	59	0	0	00	xxx	1	1	0	0	1	00
IW-Type ALU*	27	0	0	01	000	1	1	0	0	1	00
Loads	3	0	0	01	000	0	0	x	1	1	10
Stores	35	0	0	01	001	0	0	x	x	0	01
Branches	99	1	0	11	010	0	0	x	x	0	00
`jal`	111	0	1	11	011	0	0	1	0	1	00
`jalr`	103	0	1	01	000	0	0	1	0	1	00
`lui`	55	0	0	xx	100	x	0	1	0	1	00
`auipc`	23	0	0	11	100	0	0	0	0	1	00

*** Required for RV64 only**

Branch Logic

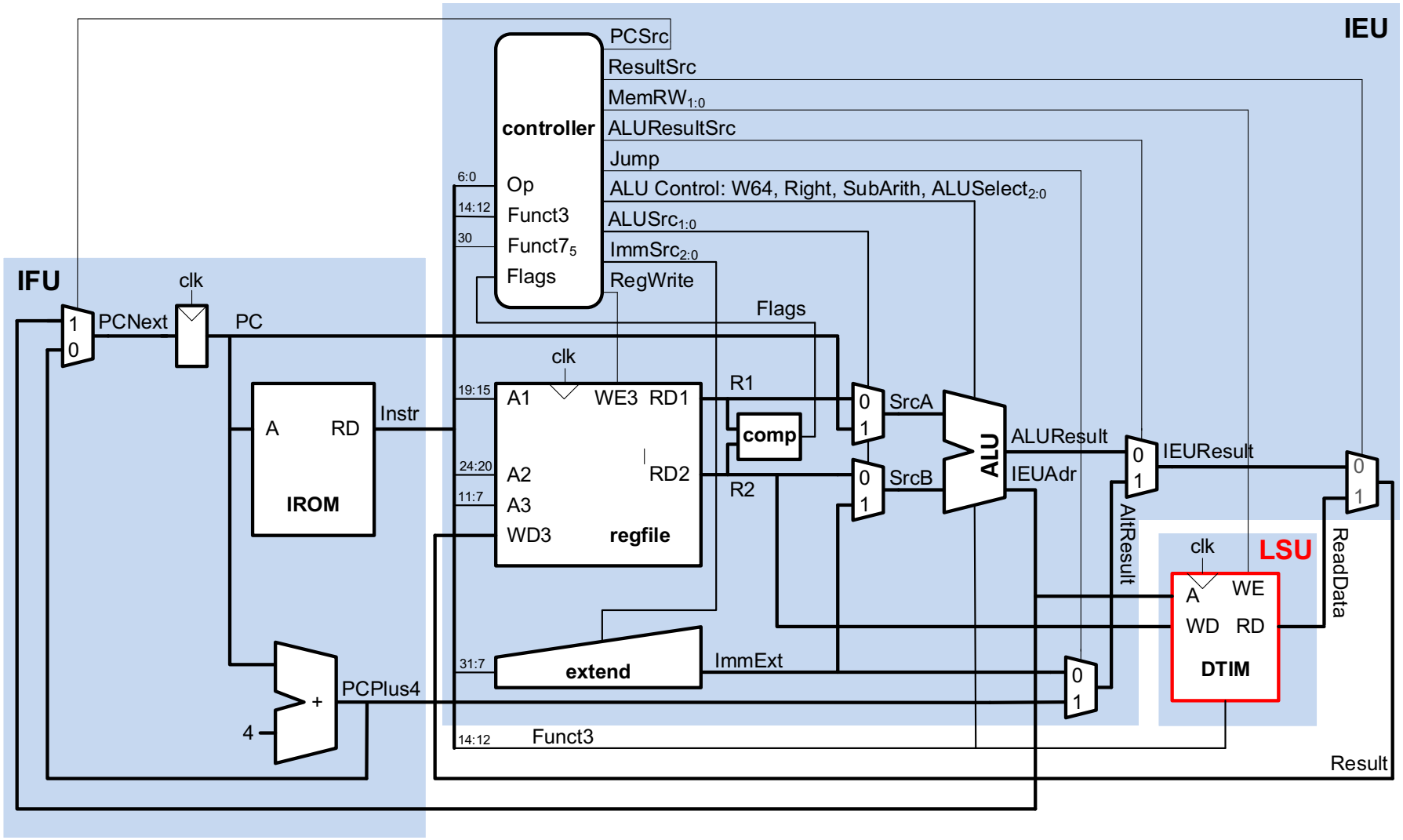
Type	Funct3	BranchTaken
beq	000	eq
bne	001	~eq
<undefined>	01x	0
blt	100	lt
bge	101	~lt
bltu	110	ltu
bgeu	111	~ltu



When Funct3[0] = 1, the flag inverts

- **Example:** for bne, `funct3` = 001, so `BranchTaken` = ~eq (i.e., eq XOR 1 = ~eq)

LSU: Load/Store Unit

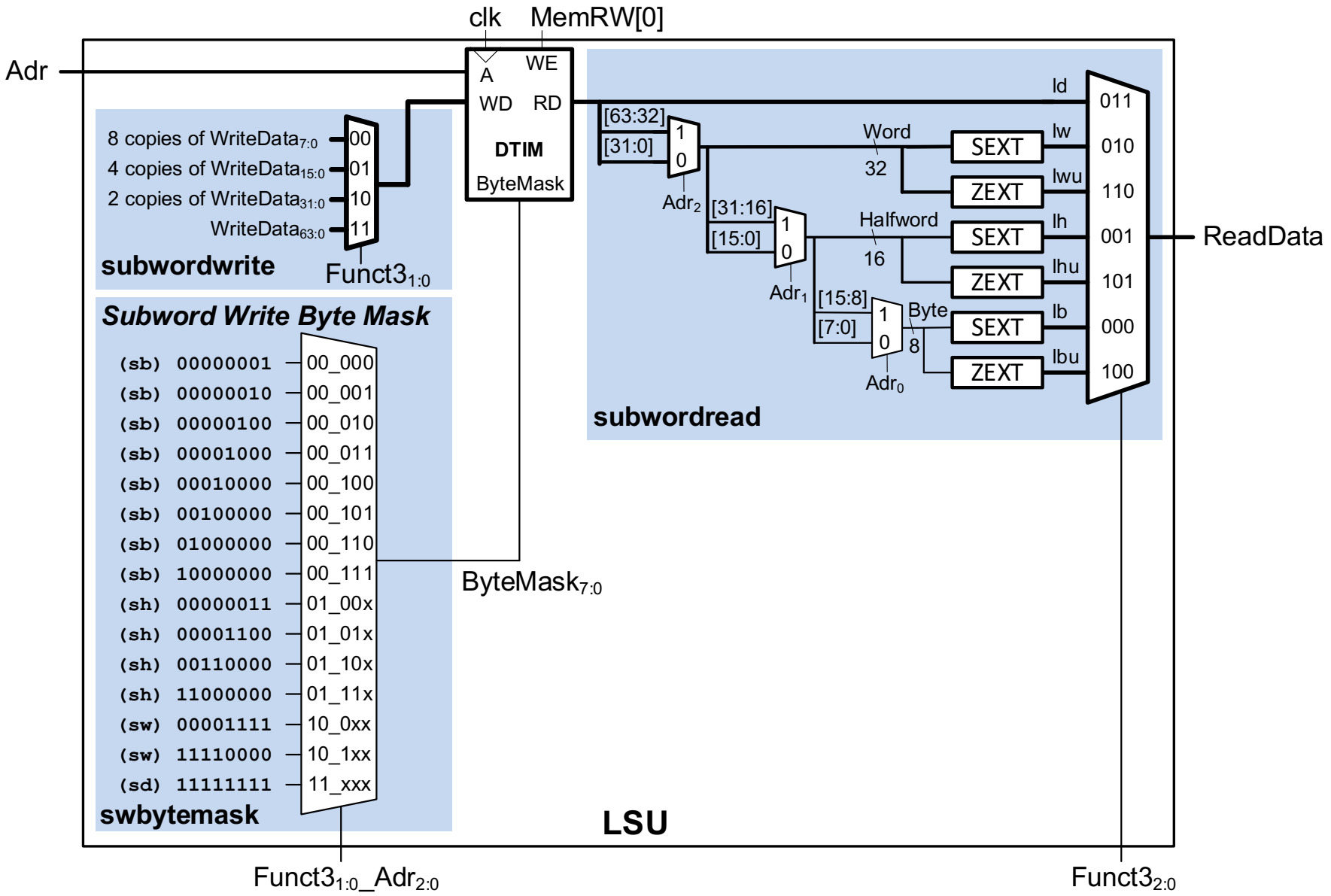


LSU: Load/Store Unit

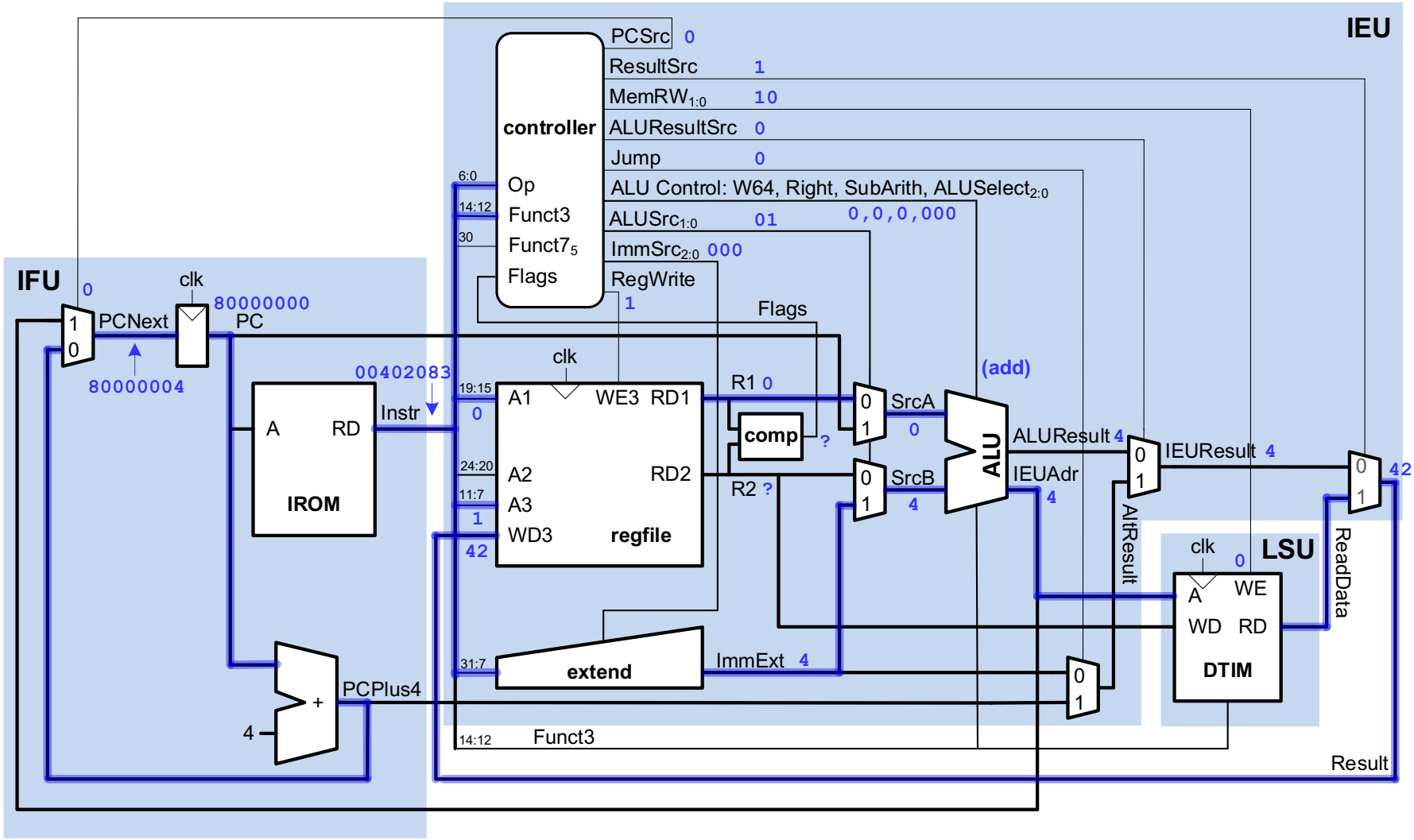
Contains a **DTIM** (data tightly integrated memory)

- TIMs have single-cycle access
- Reads occur asynchronously
- Writes occur on rising edge of clock
- Subword reads:
  - DTIM reads full word
  - Extracts appropriate byte(s)
  - Places them in least significant byte(s) of result
  - Sign-extends result
- Subword writes:
  - DTIM replicates subword into all positions in its data write signal
  - Asserts byte write signals (that indicate which bytes within word to write)

Detailed LSU



Example Execution

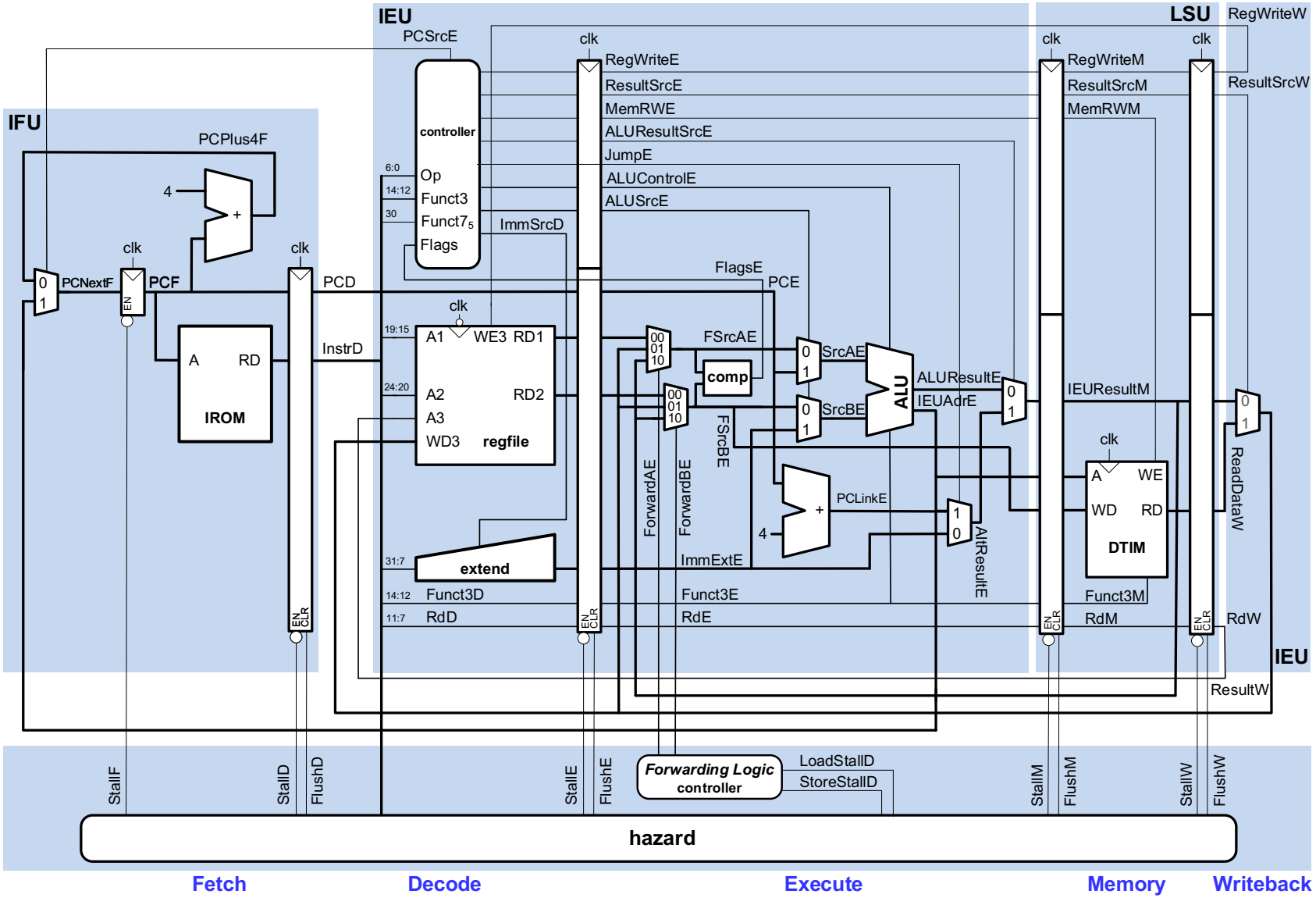



```
// address instruction      machine code
80000000   lw x1, 4(x0)      00402083
```

Chapter 7: Pipelined Core

Pipelined RV32I Core

Pipelined Processor

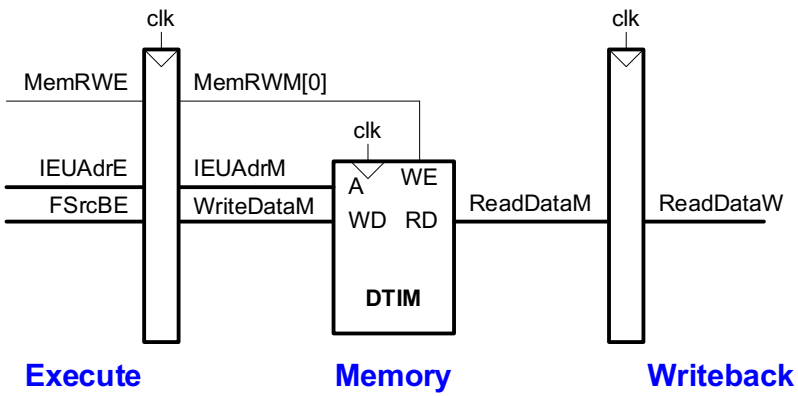


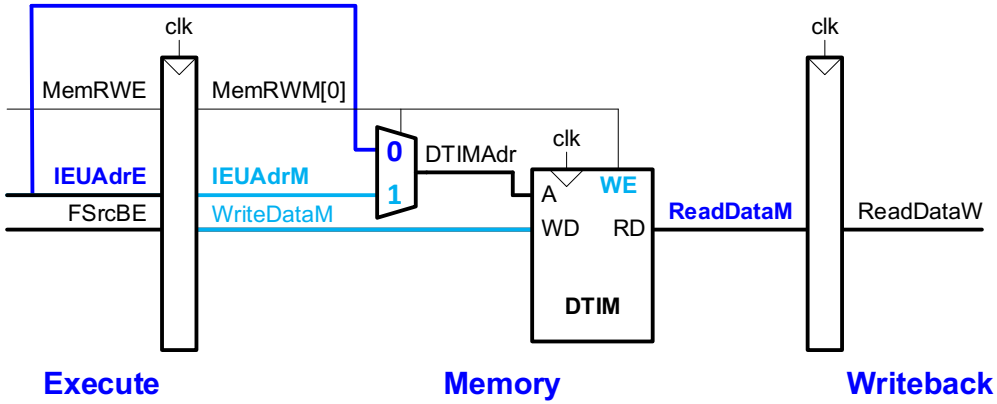
- Pipeline registers
- Hazard unit
- Bypass muxes

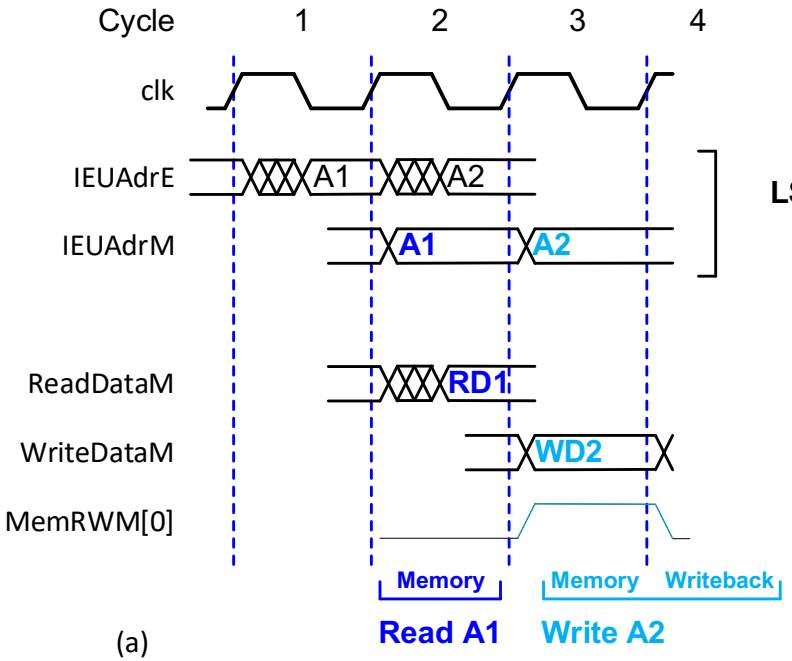
Synchronous Memories

- **Read may be synchronous or asynchronous** – memories with asynchronous reads are called "asynchronous memories"
- **Writes are always synchronous** (on rising or falling edge)
- **Synchronous (vs. asynchronous) memories:**
  - Smaller area
  - Higher Performance

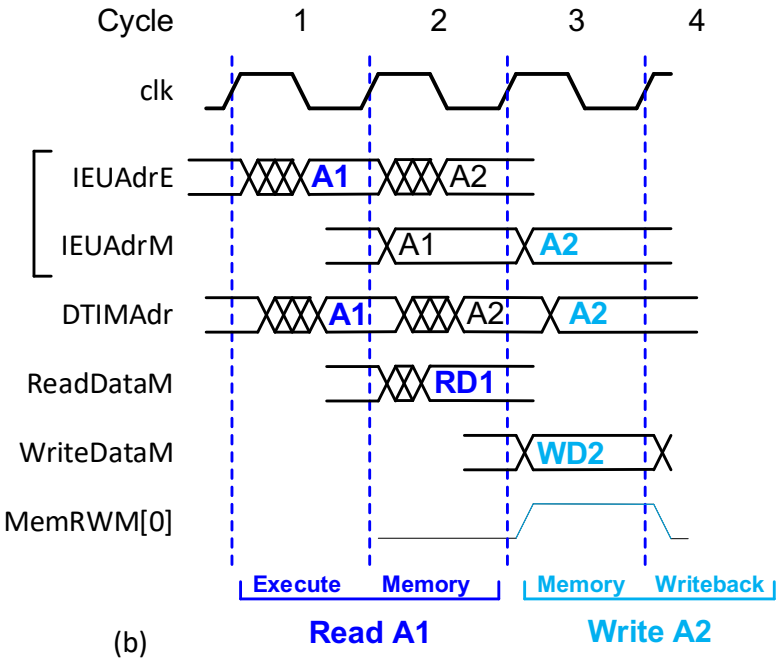
Synchronous Memories: lw then sw







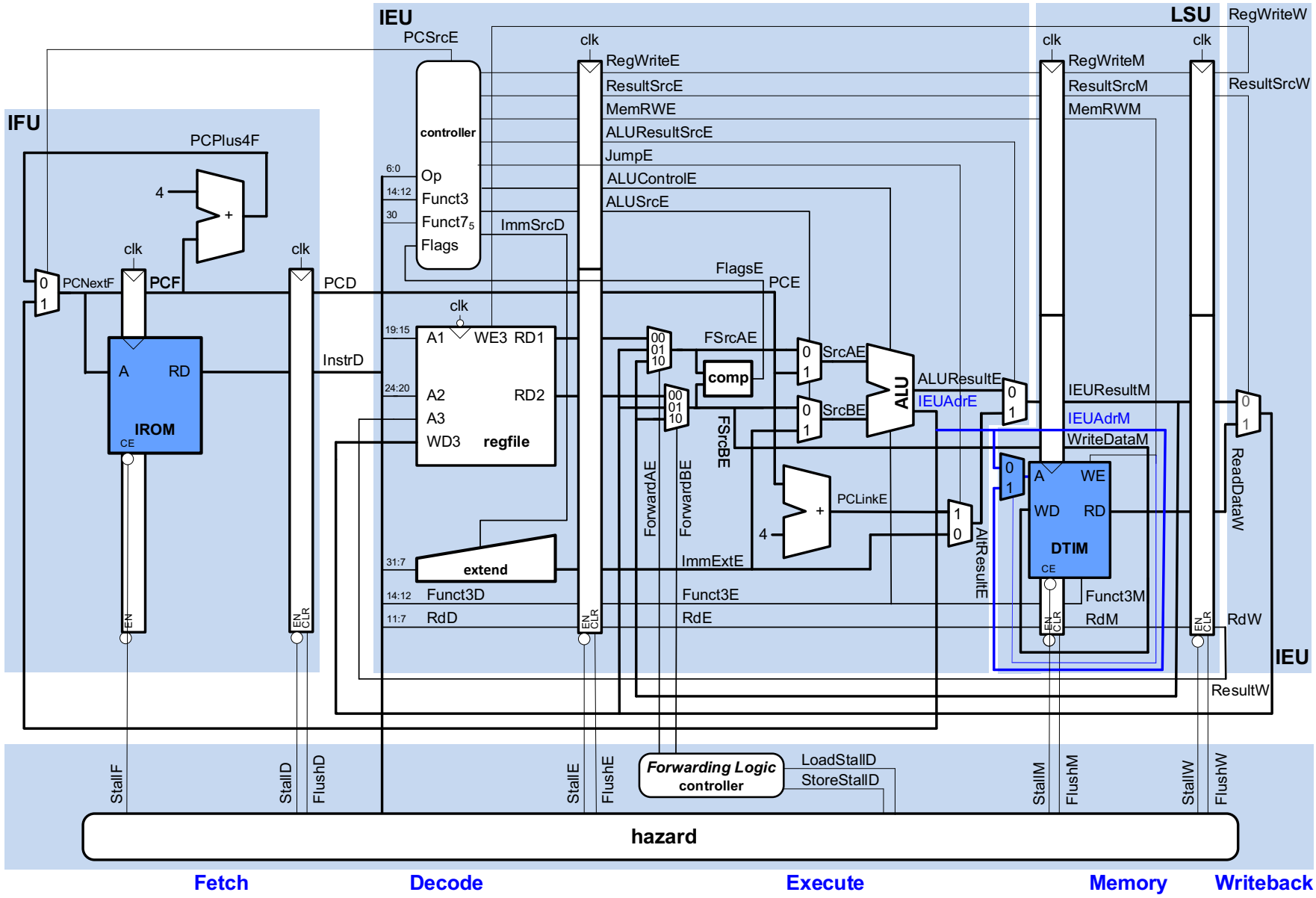
LSU Address Signals



Synchronous IROM

- **Similar to DTIM:**
  - Address must set up before rising edge of Fetch stage
  - So IROM uses **`PCNextF`** (instead of `PCNext`)

Pipeline with Synch. Memories



Chapter 7: Pipelined Core

Hazards

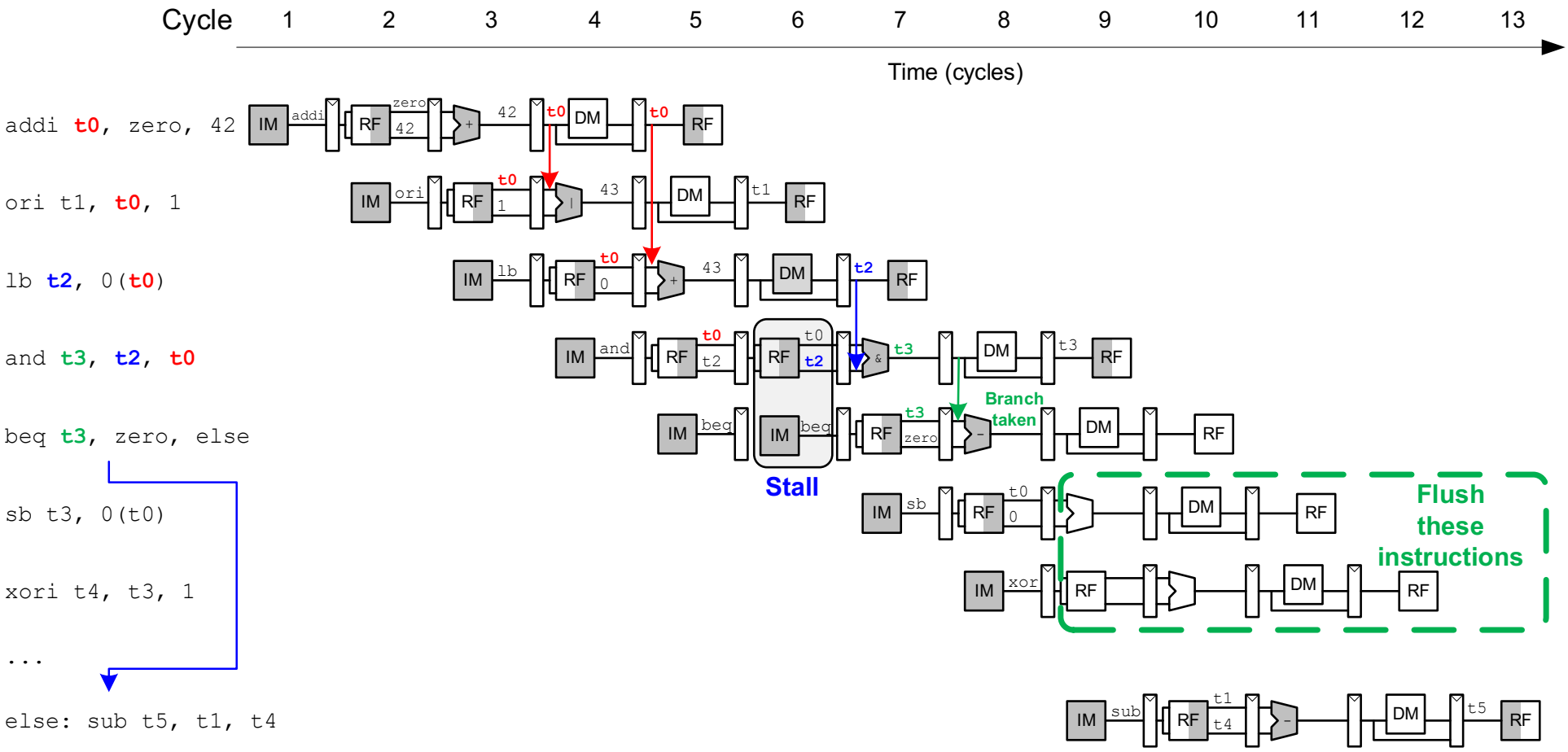
Types of Hazards

- **Data hazard:** Instruction needs a result from a previous instruction
- **Control hazard:** Next instruction address has not been calculated yet (because of branch or jump)

Hazard Resolution

- **Forward:** Send result from in-flight instruction
- **Stall:** Stop pipeline
- **Flush:** Discard incorrect instructions from pipeline

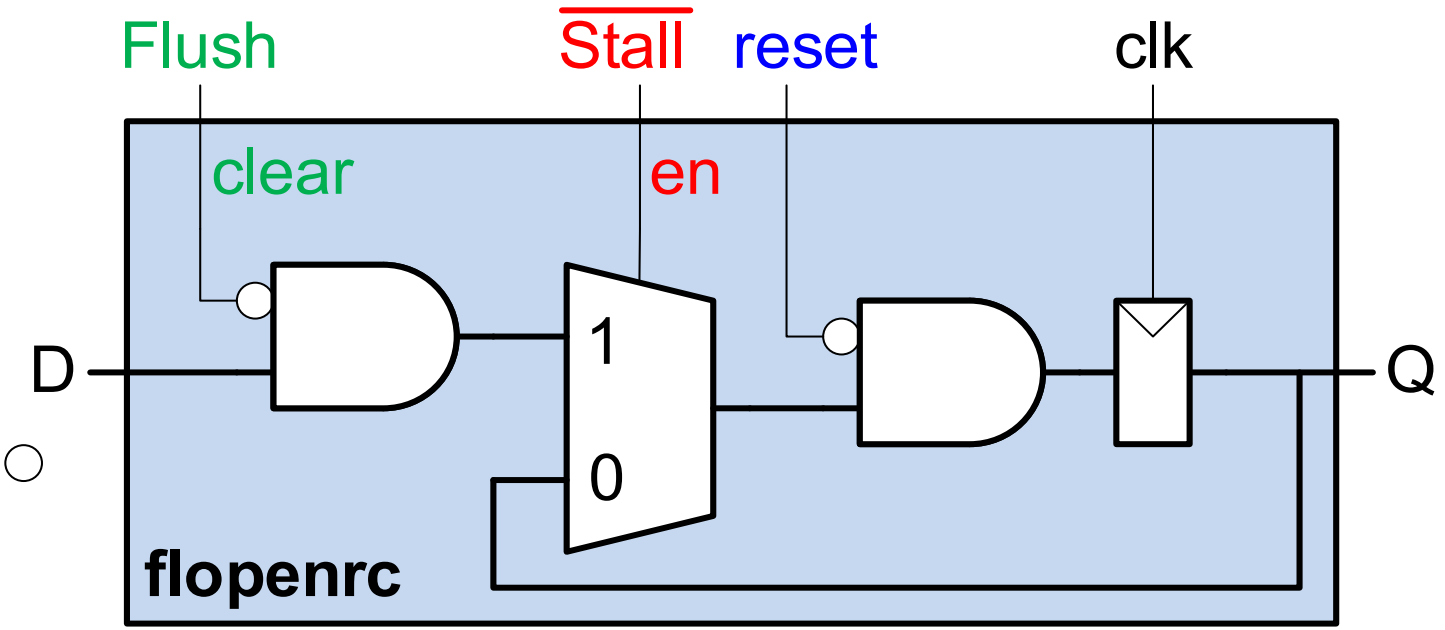
Program with Hazards



Data hazards: `t0` and `t2` accesses

Control hazard: `beq`

Pipeline Registers



All pipeline registers have:

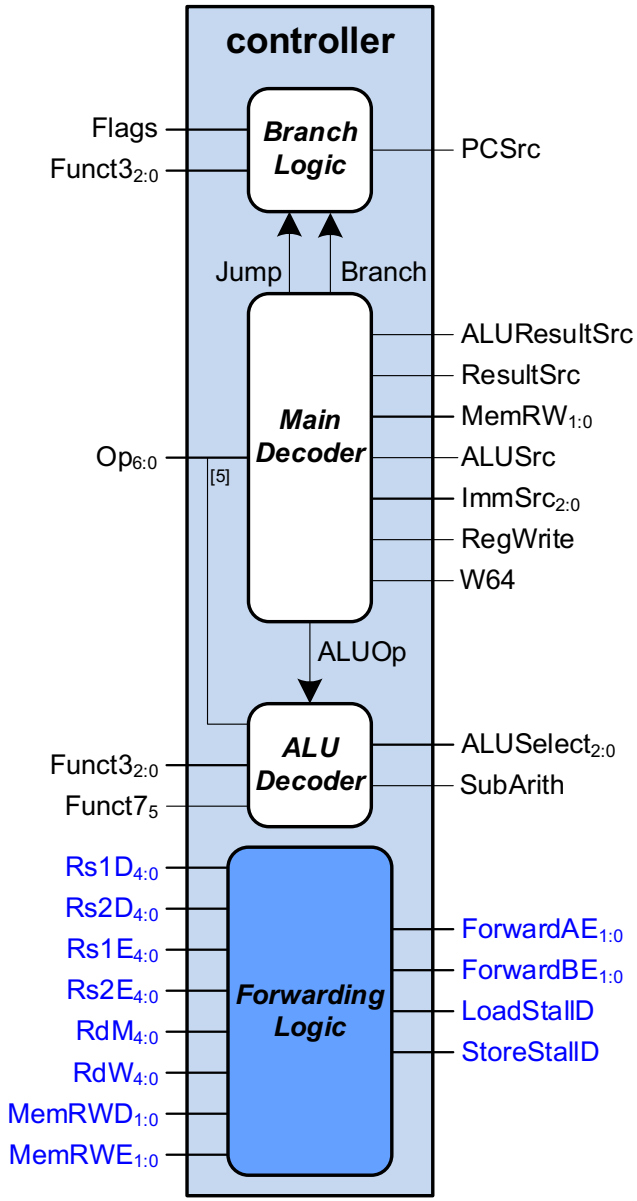
- `clear` (synchronous): connected to `Flush`
- `en`: connected to $\overline{\texttt{Stall}}$
- `reset` (asynchronous): global reset

Forwarding Logic

- **Handles:**
  - **Forwards:** When source register in Execute stage same as destination register in Memory or Writeback stage
- The Forwarding Logic is inside the **controller unit**

Forwarding Logic

- **Handles:**
  - **Forwards:** When source register in Execute stage same as destination register in Memory or Writeback stage
- The Forwarding Logic is inside the **controller unit**



Forwarding Logic

- **Handles:**
  - **Forwards:** Source register in Execute stage same as destination register in Memory or Writeback stage

Example: **`ForwardAE`** (for SrcA of ALU):

```
# mem stage
if      ((Rs1E == RdM) & RegWriteM) & (Rs1E != 0) ForwardAE = 10
# writeback stage
else if ((Rs1E == RdW) & RegWriteW) & (Rs1E != 0) ForwardAE = 01
# no forward
else                                              ForwardAE = 00
```

`ForwardBE` (for SrcB) is similar (replace `Rs1E` with `Rs2E`).

Hazard Unit

- **Handles:**
  - **Stalls:**
    - Load followed by use
    - Store followed by a load

Hazard Unit

- **Handles:**
  - **Stalls:**
    - **Load followed by use:** detected in Decode stage
      - Load is in Execute stage:
        MemRW[1] = 1
      - Either source register in Decode stage is same as Execute's destination register:
        `MatchDE = (Rs1D == RdE | Rs2D = RdE)`

      Thus, `LoadStallD = MemRWE[1] & MatchDE`
    - **Store followed by a load:**

      `StoreStallD = MemRWE[0] & MemRWD[1]`

Recall:

`MemRW[1]` = 1 for loads

`MemRW[0]` = 1 for stores

Hazard Unit

- Also includes flushes for control hazards:
  - `PCSrcE` = 1 for jumps or taken branches
  - So, when `PCSrcE` = 1, Decode and Execute stages must be flushed (on next cycle). Thus,

```
FlushDCause = PCSrcE
FlushECause = PCSrcE
```

Summary: Forwarding & Hazards

Forward:

```
if      ((Rs1E == RdM) & RegWriteM) & (Rs1E != 0) ForwardAE = 10
else if ((Rs1E == RdW) & RegWriteW) & (Rs1E != 0) ForwardAE = 01
else                                              ForwardAE = 00

if      ((Rs2E == RdM) & RegWriteM) & (Rs2E != 0) ForwardBE = 10
else if ((Rs2E == RdW) & RegWriteW) & (Rs2E != 0) ForwardBE = 01
else                                              ForwardBE = 00
```

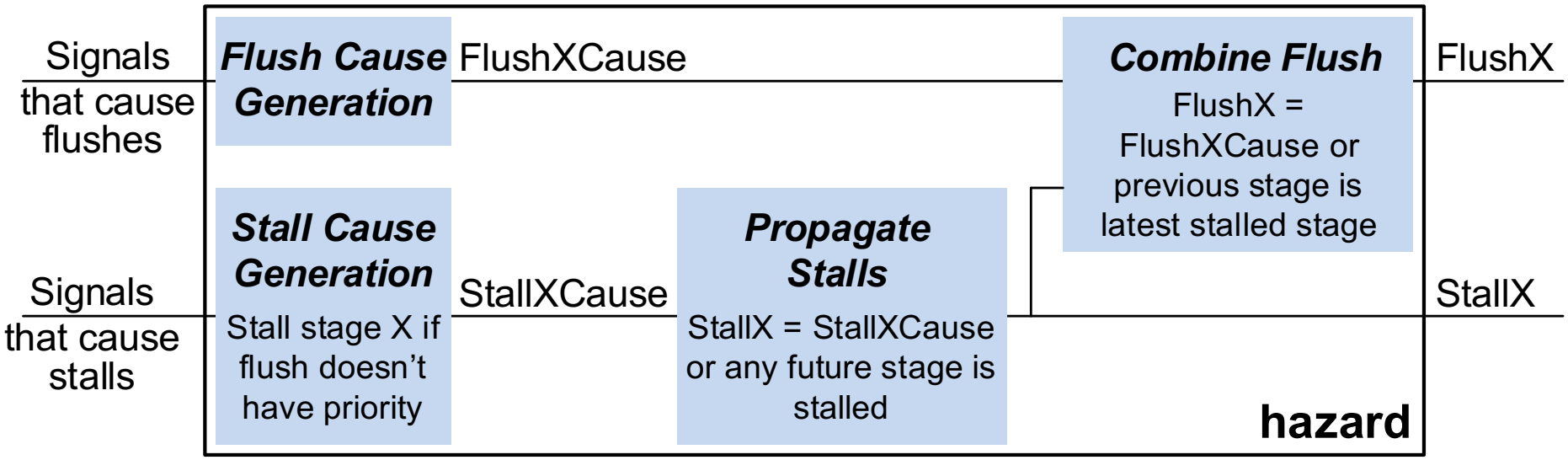
Stall:

```
MatchDE     = (Rs1D == RdE | Rs2D == RdE) & (RdE != 0)
LoadStallD  = MemRWE[1] & MatchDE
StoreStallD = MemRWE[0] & MemRWD[1]
```

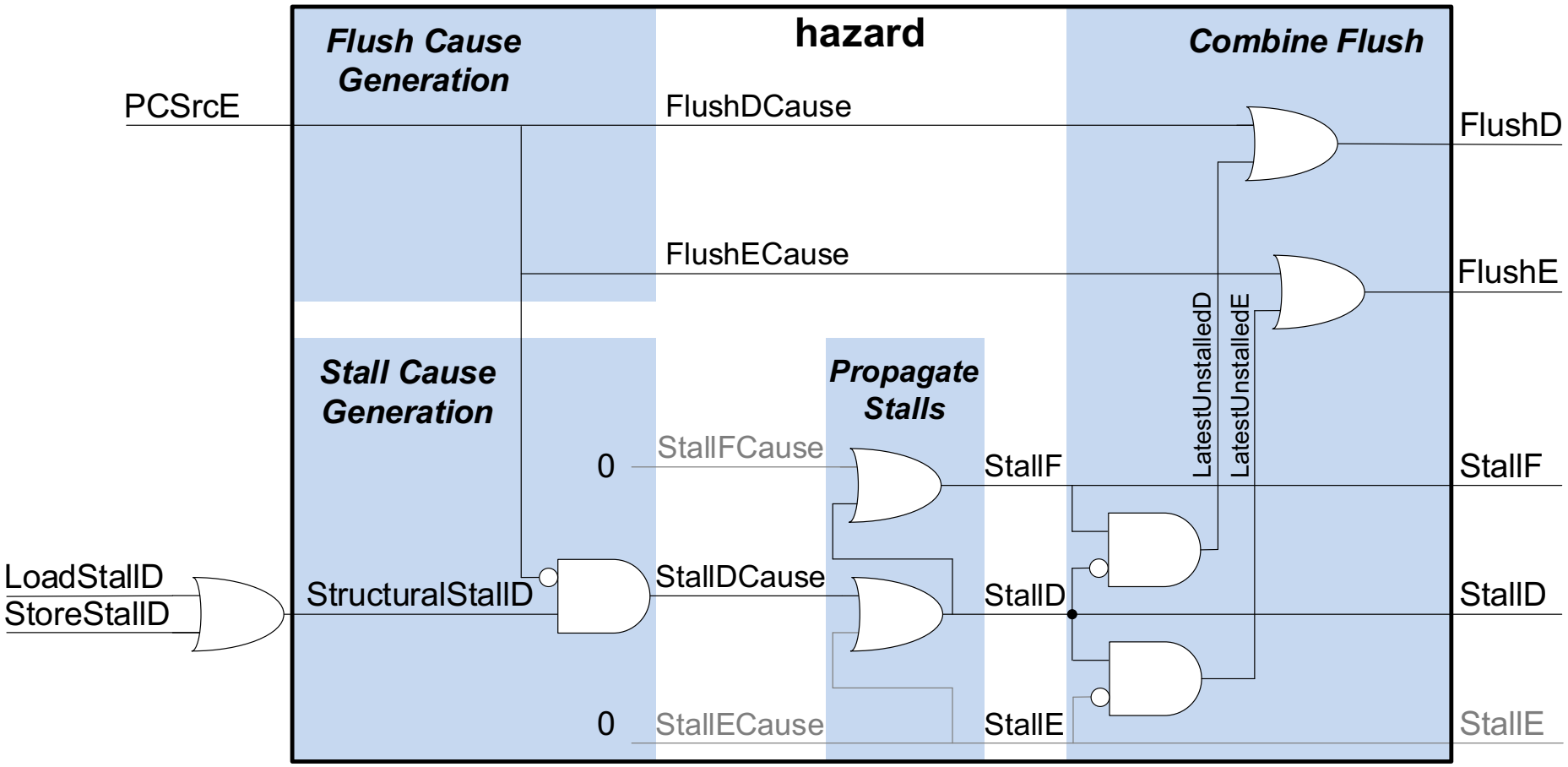
Flush:

```
FlushDCause = PCSrcE
FlushECause = PCSrcE
```

Hazard Unit Architecture



Stall and Flush Logic



Pipelined Performance

Execution Time = # instructions × *CPI* × T_C

Example: CoreMark Benchmark

Type	# Instructions	% of Total Instructions	CPI
R- or I-type ALU	1,320,259	49.7%	1
Jumps	46,750	1.8%	3
Branches	588,905	22.2%	1.84
Loads/stores	698,738	26.3%	1.41
Total	2,654,652	100%	1.34

- **Branches:** 11% mispredicted (costs 2 more cycles to flush two fetched instructions)
  - So, branch CPI = 1 + 0.11 x 2 = 1.22
- **Jumps:** always take 3 cycles (1 to execute, 2 flushed). So, CPI = 3
- **40.1% of loads/stores** cause a stall (either a store followed by load or a load that is immediately used by next instruction)
  - So, load/store CPI = 1 + 0.40 x 1 = 1.46
- **Overall CPI:**

  $(0.5 \times 1) + (0.02 \times 3) + (0.222 \times 1.84) + (0.263 \times 1.41) = \mathbf{1.34}$

CoreMark's empirically reported CPI: 1.27

About these Notes

RISC-V System-on-Chip Design Lecture Notes