

# Digital Design & Computer Architecture

Sarah Harris & David Harris

## Chapter 7: Microarchitecture





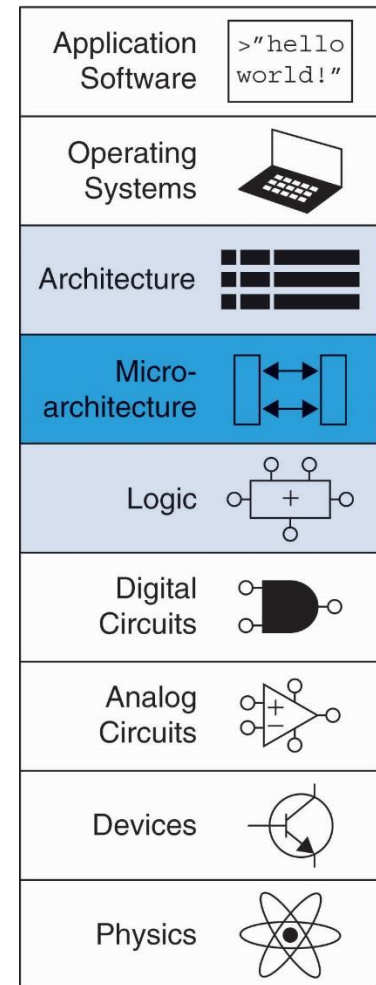






# Chapter 7 :: Topics

- Introduction
- Performance Analysis
- Single-Cycle Processor
- Multicycle Processor
- Pipelined Processor
- Advanced Microarchitecture









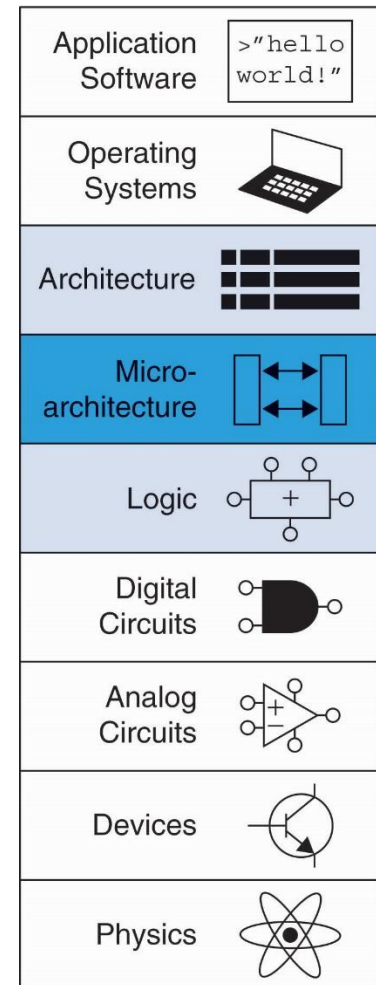






# Introduction

- **Microarchitecture:** how to implement an architecture in hardware
- Processor:
  - **Datapath:** functional blocks
  - **Control:** control signals















# Microarchitecture

- **Multiple implementations** for a single architecture:
  - **Single-cycle:** Each instruction executes in a single cycle
  - **Multicycle:** Each instruction is broken up into series of shorter steps
  - **Pipelined:** Each instruction broken up into series of steps & multiple instructions execute at once













# Processor Performance

- **Program execution time**

**Execution Time = (#instructions)(cycles/instruction)(seconds/cycle)**

- **Definitions:**

- CPI: Cycles/instruction
- clock period: seconds/cycle
- IPC: instructions/cycle = IPC

- **Challenge is to satisfy constraints of:**

- Cost
- Power
- Performance











# RISC-V Processor

- Consider **subset** of RISC-V instructions:
  - R-type ALU instructions:
    - **add, sub, and, or, slt**
  - Memory instructions:
    - **lw, sw**
  - Branch instructions:
    - **beq**













# Architectural State Elements

Determines everything about a processor:

- **Architectural state:**

- 32 registers
- PC
- Memory





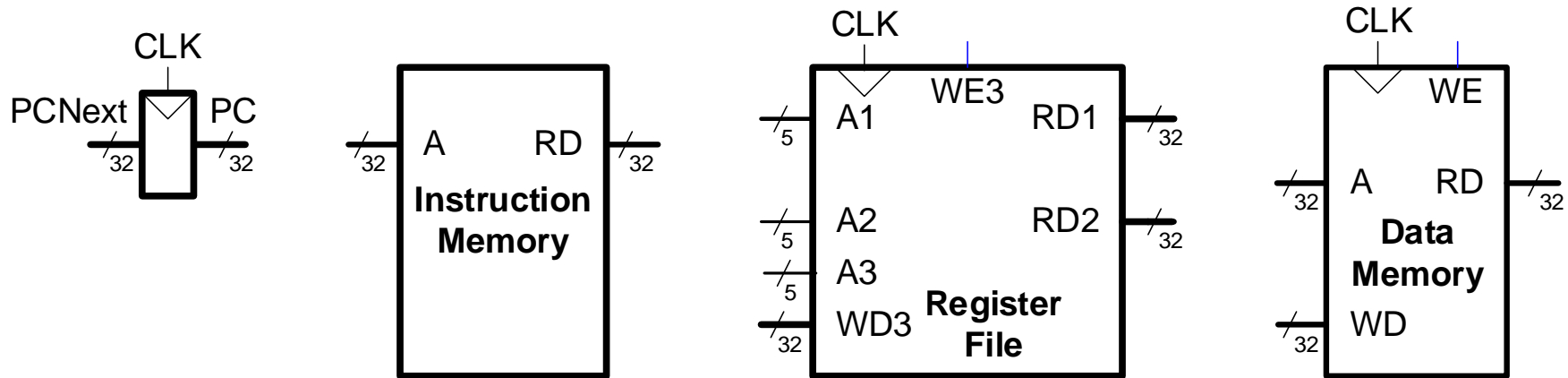








# RISC-V Architectural State Elements















# Chapter 7: Microarchitecture

## **Single-Cycle RISC-V Processor**











# Single-Cycle RISC-V Processor

- Datapath
- Control













# Example Program

- Design datapath
- View example program executing

## Example Program:

Address	Instruction	Type	Fields					Machine Language	
0x1000	L7: lw x6, -4(x9)	I	imm <sub>11:0</sub>	rs1	f3	rd	op		
			1111111111100	01001	010	00110	0000011	FFC4A303	
0x1004	sw x6, 8(x9)	S	imm <sub>11:5</sub>	rs2	rs1	f3	imm <sub>4:0</sub>	op	
			0000000 00110	01001	010	01000	0100011	0064A423	
0x1008	or x4, x5, x6	R	funct7	rs2	rs1	f3	rd	op	
			0000000 00110	00101	110	00100	0110011	0062E233	
0x100C	beq x4, x4, L7	B	imm <sub>12,10:5</sub>	rs2	rs1	f3	imm <sub>4:1,11</sub>	op	
			1111111 00100	00100	000	10101	1100011	FE420AE3	









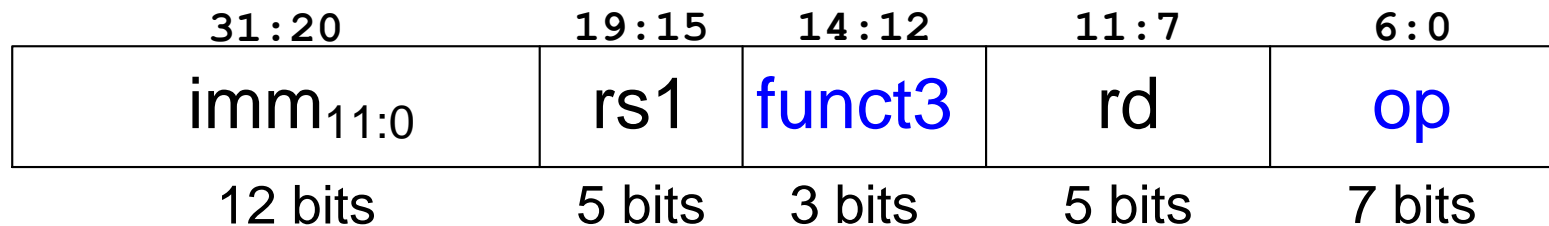




# Single-Cycle RISC-V Processor

- **Datapath:** start with `lw` instruction
- **Example:** `lw x6, -4(x9)`  
`lw rd, imm(rs1)`

## I-Type









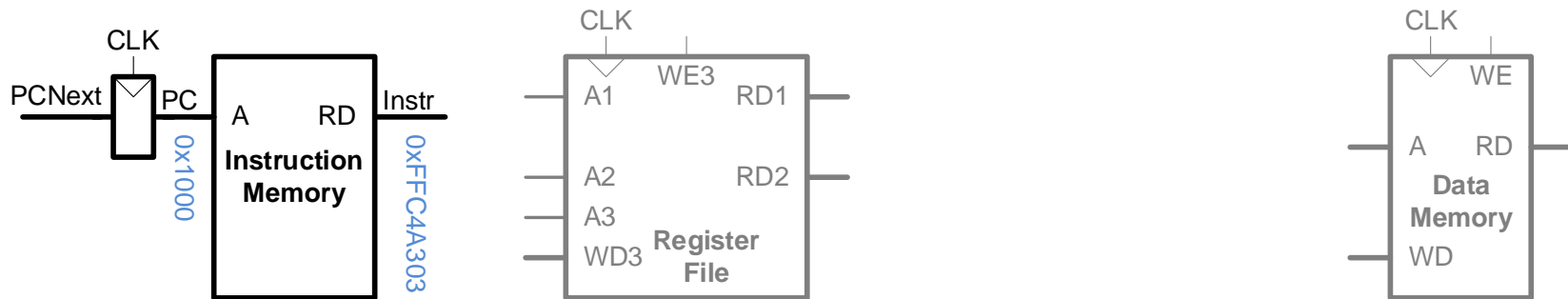






# Single-Cycle Datapath: $l_w$ fetch

## STEP 1: Fetch instruction



Address	Instruction	Type	Fields	Machine Language										
0x1000	L7: lw x6, -4(x9)	I	<table><tr><th>imm<sub>11:0</sub></th><th>rs1</th><th>f3</th><th>rd</th><th>op</th></tr><tr><td>111111111100</td><td>01001</td><td>010</td><td>00110</td><td>0000011</td></tr></table>	imm <sub>11:0</sub>	rs1	f3	rd	op	111111111100	01001	010	00110	0000011	FFC4A303
imm <sub>11:0</sub>	rs1	f3	rd	op										
111111111100	01001	010	00110	0000011										





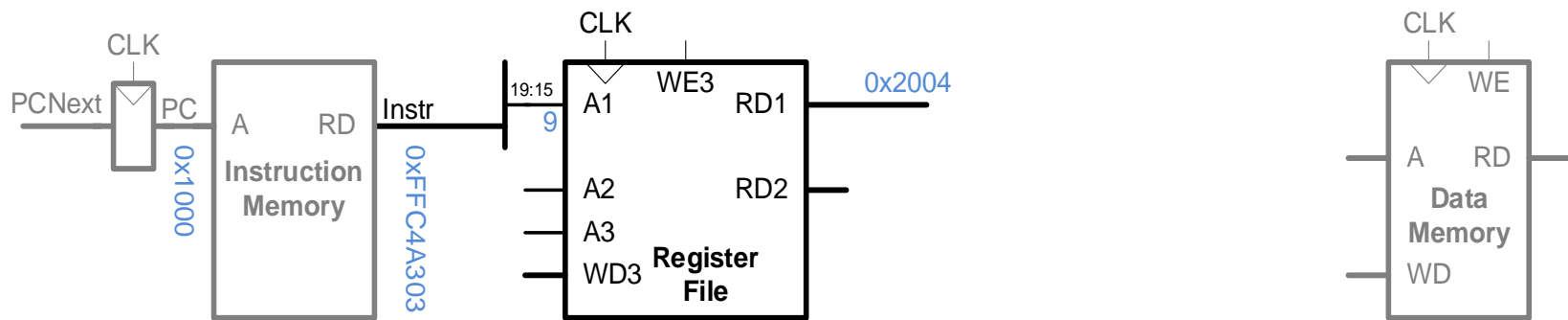






# Single-Cycle Datapath: $lw$ Reg Read

## STEP 2: Read source operand (**rs1**) from RF



Address	Instruction	Type	Fields	Machine Language
0x1000	L7: lw x6, -4(x9)	I	<div><div>imm<sub>11:0</sub></div><div>111111111100</div></div> <div><div>rs1</div><div>01001</div></div> <div><div>f3</div><div>010</div></div> <div><div>rd</div><div>00110</div></div> <div><div>op</div><div>0000011</div></div>	FFC4A303







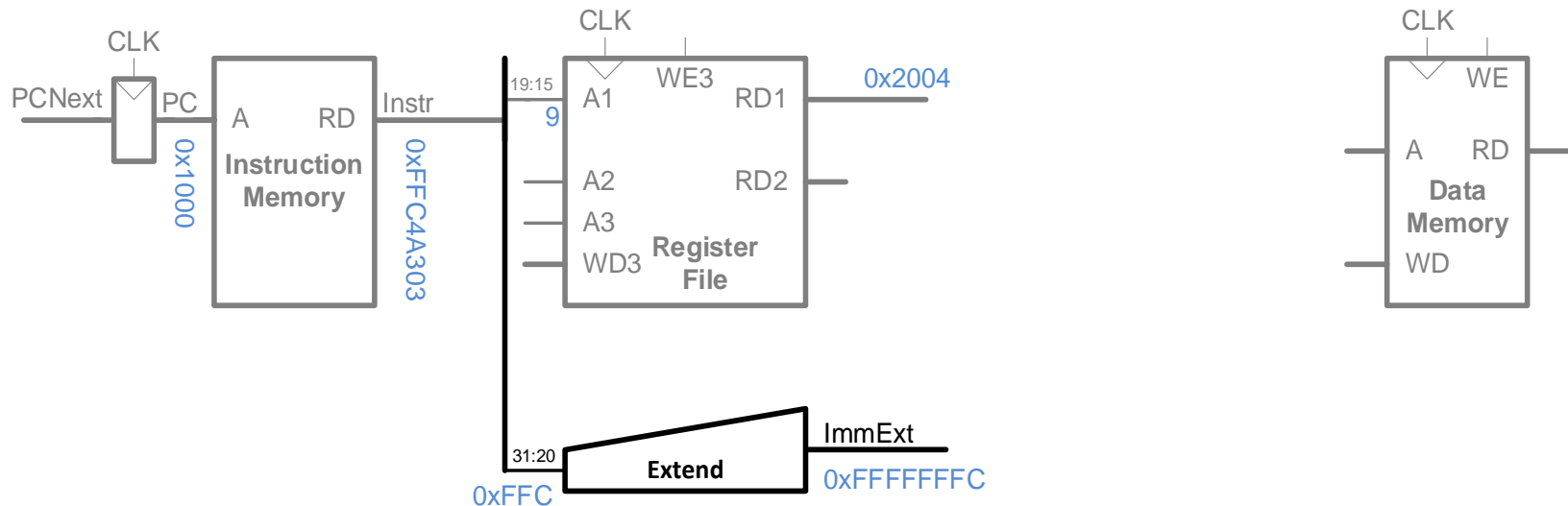






# Single-Cycle Datapath: 1w Immediate

## STEP 3: Extend the immediate



Address	Instruction	Type	Fields	Machine Language
0x1000	L7: lw x6, -4(x9)	I	<div><div>imm<sub>11:0</sub></div><div>1111111111100</div></div> <div><div>rs1</div><div>01001</div></div> <div><div>f3</div><div>010</div></div> <div><div>rd</div><div>00110</div></div> <div><div>op</div><div>0000011</div></div>	FFC4A303







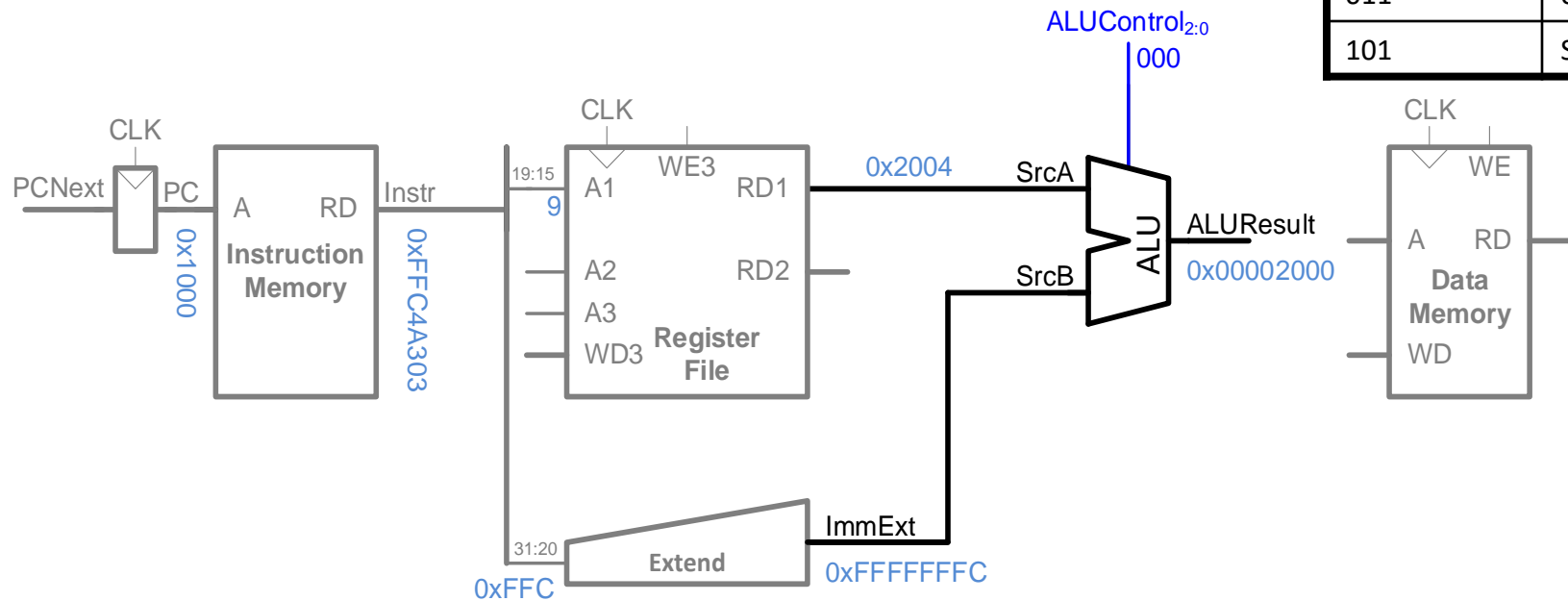






# Single-Cycle Datapath: lw Address

## STEP 4: Compute the memory address



ALUControl <sub>2:0</sub>	Function
000	add
001	subtract
010	and
011	or
101	SLT

Address	Instruction	Type	Fields	Machine Language										
0x1000	L7: lw x6, -4(x9)	I	<table><tr><td>imm<sub>11:0</sub></td><td>rs1</td><td>f3</td><td>rd</td><td>op</td></tr><tr><td>111111111100</td><td>01001</td><td>010</td><td>00110</td><td>0000011</td></tr></table>	imm <sub>11:0</sub>	rs1	f3	rd	op	111111111100	01001	010	00110	0000011	FFC4A303
imm <sub>11:0</sub>	rs1	f3	rd	op										
111111111100	01001	010	00110	0000011										







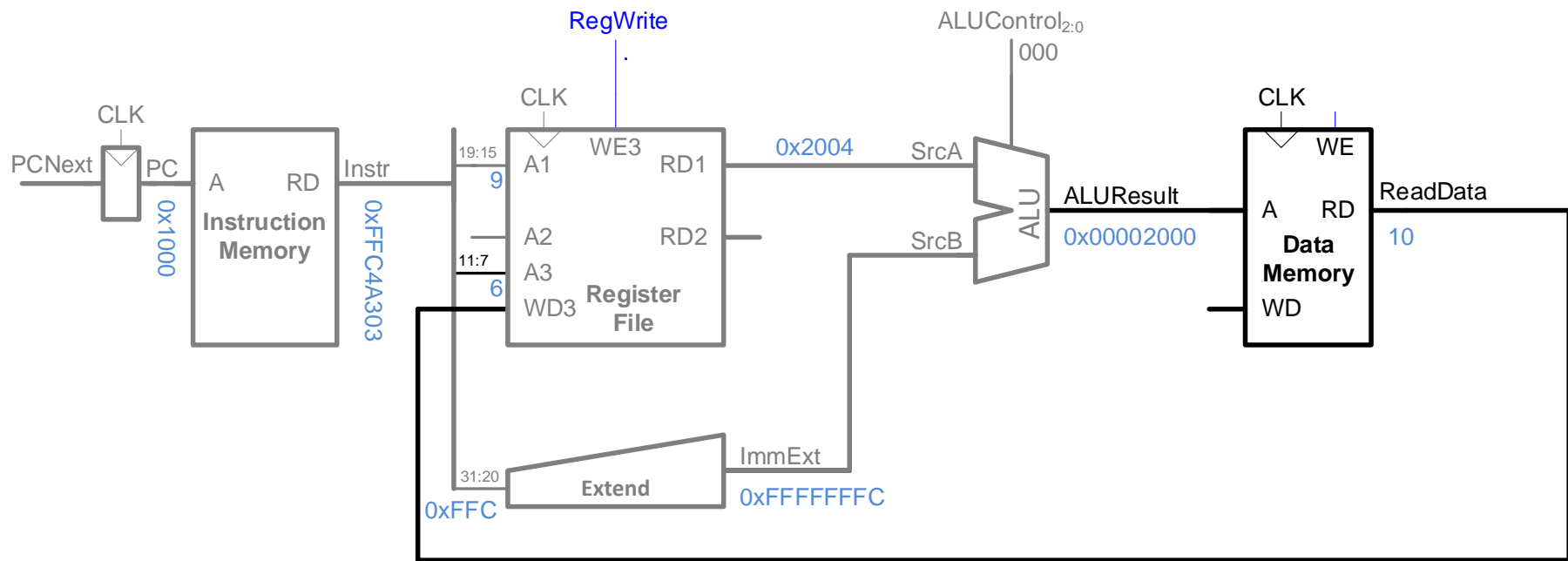






# Single-Cycle Datapath: $lw$ Mem Read

**STEP 5:** Read data from memory and write it back to register file



Address	Instruction	Type	Fields	Machine Language										
0x1000	L7: lw x6, -4(x9)	I	<table><tr><th>imm<sub>11:0</sub></th><th>rs1</th><th>f3</th><th>rd</th><th>op</th></tr><tr><td>111111111100</td><td>01001</td><td>010</td><td>00110</td><td>0000011</td></tr></table>	imm <sub>11:0</sub>	rs1	f3	rd	op	111111111100	01001	010	00110	0000011	FFC4A303
imm <sub>11:0</sub>	rs1	f3	rd	op										
111111111100	01001	010	00110	0000011										





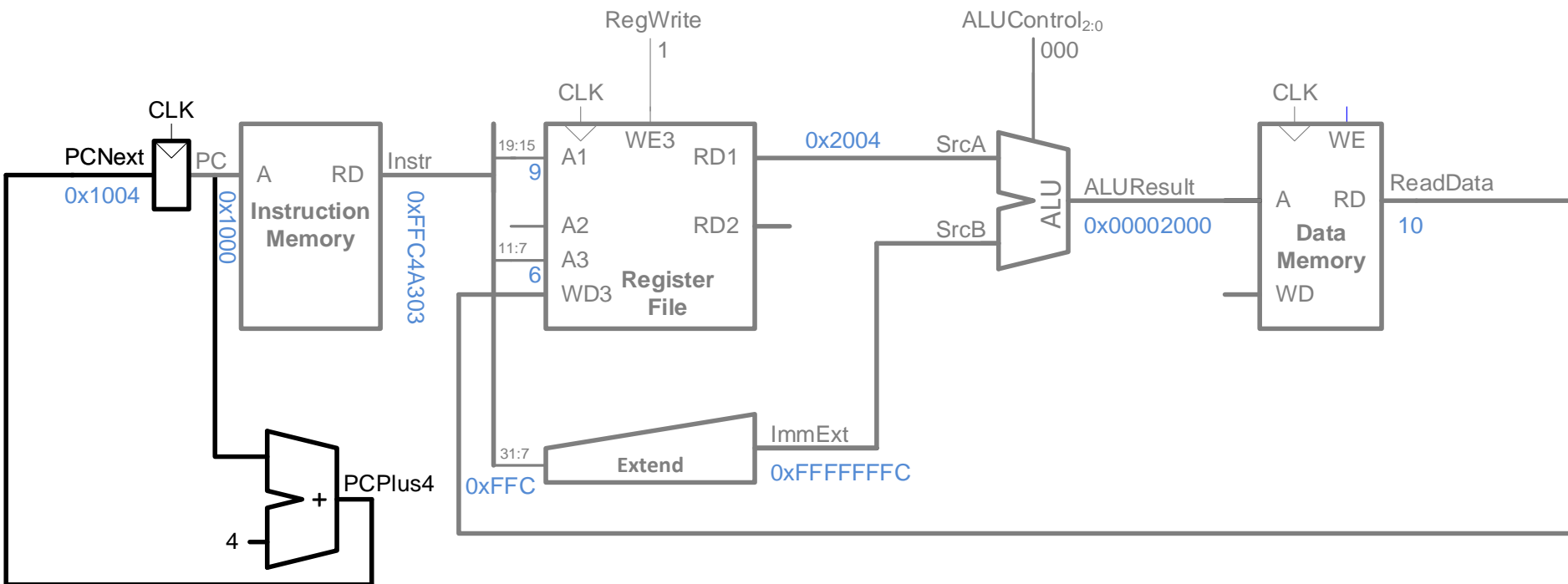






# Single-Cycle Datapath: PC Increment

## STEP 6: Determine address of next instruction



Address	Instruction	Type	Fields			Machine Language	
			imm <sub>11:0</sub>	rs1	f3	rd	op
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011
							FFC4A303













# Chapter 7: Microarchitecture

**Single-Cycle**

**Datapath: Other  
Instructions**







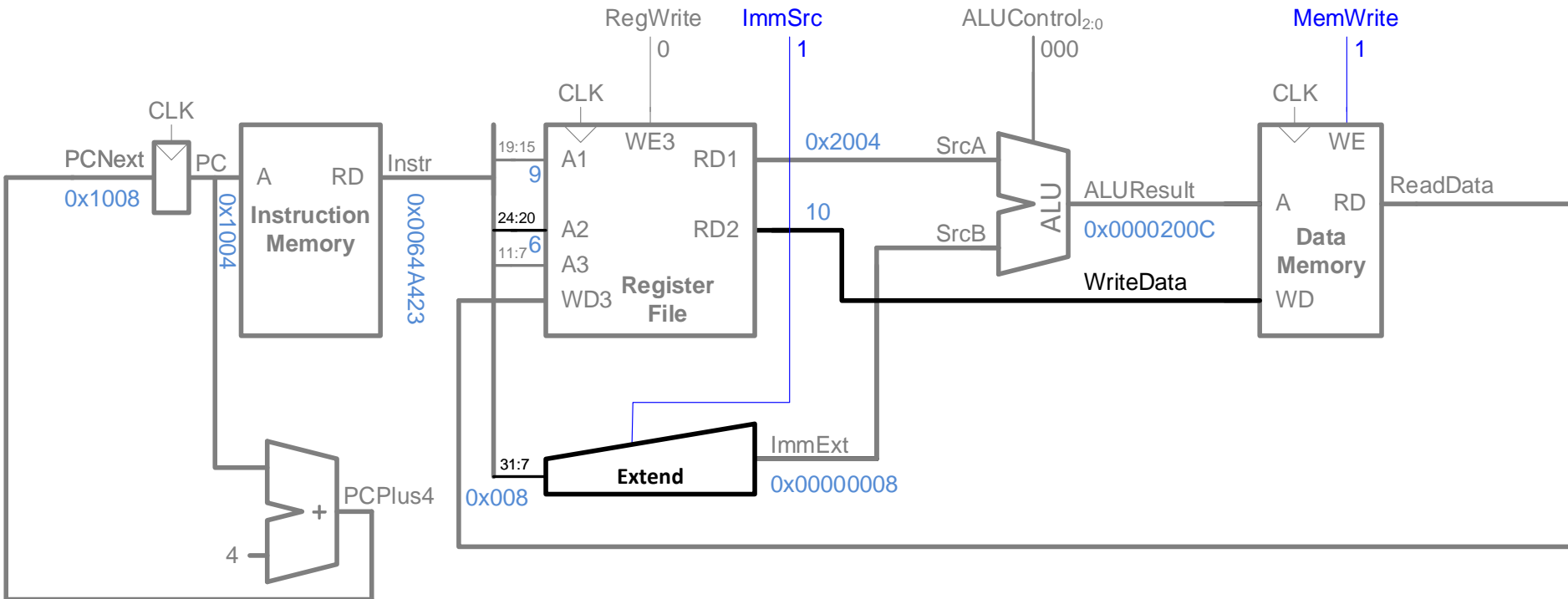






# Single-Cycle Datapath: sw

- **Immediate:** now in {instr[31:25], instr[11:7]}
- **Add control signals:** ImmSrc, MemWrite



Address	Instruction	Type	Fields					Machine Language	
			imm <sub>11:5</sub>	rs2	rs1	f3	imm <sub>4:0</sub>	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423









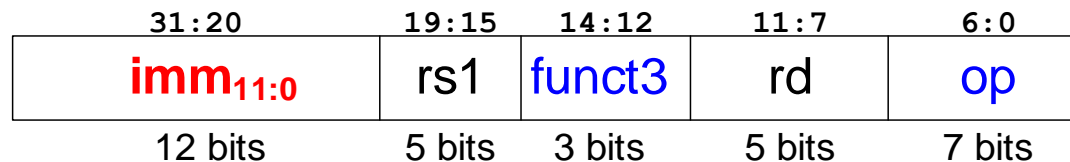




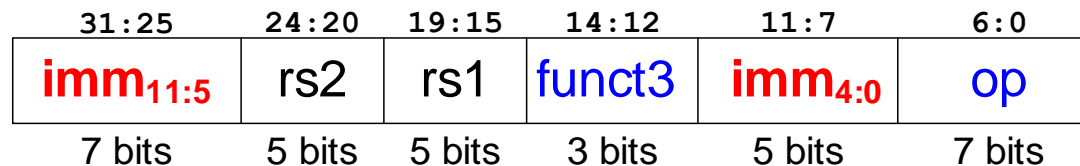
# Single-Cycle Datapath: Immediate

ImmSrc	ImmExt	Instruction Type
0	{{20{instr[31]}}, <b>instr[31:20]</b> }	I-Type
1	{{20{instr[31]}}, <b>instr[31:25], instr[11:7]</b> }	S-Type

## I-Type



## S-Type







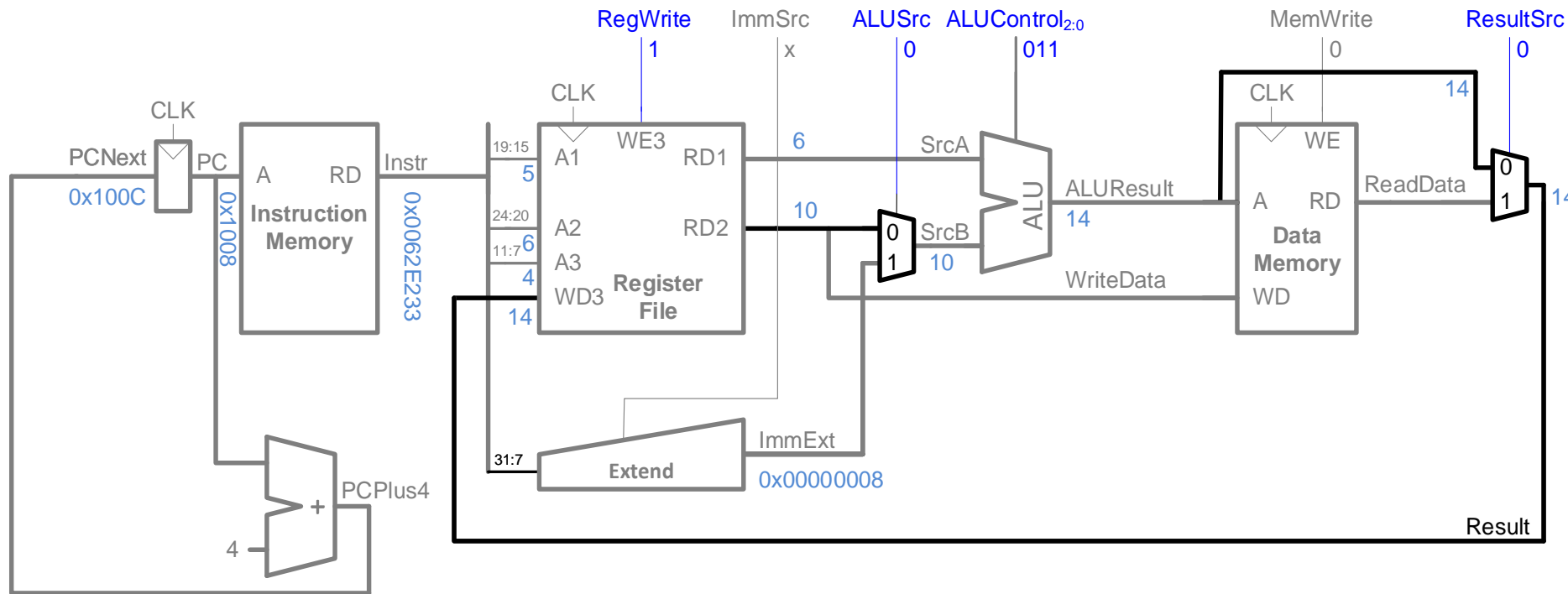






# Single-Cycle Datapath: R-type

- Read from **rs1** and **rs2** (instead of **imm**)
- Write *ALUResult* to **rd**



Address	Instruction	Type	Fields					Machine Language
			funct7	rs2	rs1	f3	rd	op
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011
								0062E233







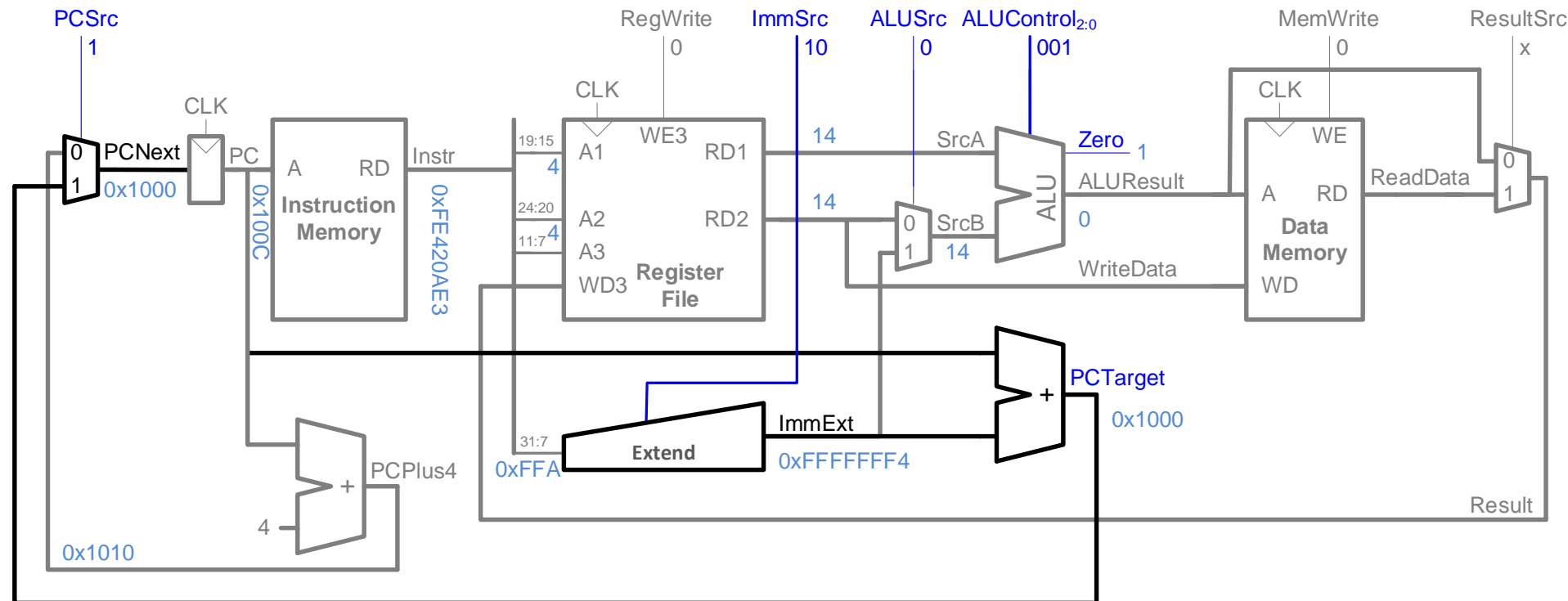






# Single-Cycle Datapath: beq

Calculate **target address**:  $PCTarget = PC + imm$



Address	Instruction	Type	Fields					Machine Language	
			imm <sub>12,10:5</sub>	rs2	rs1	f3	imm <sub>4:1,11</sub>	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3









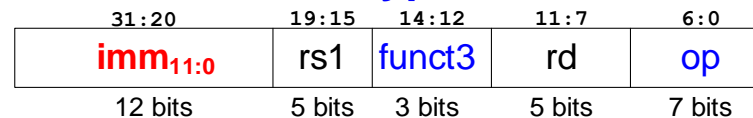




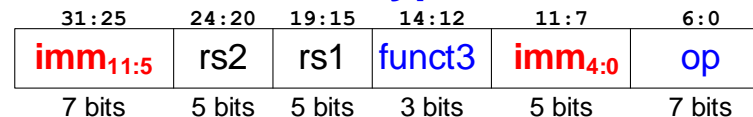
# Single-Cycle Datapath: ImmExt

ImmSrc <sub>1:0</sub>	ImmExt	Instruction Type
00	{{20{instr[31]}}, <b>instr[31:20]</b> }	I-Type
01	{{20{instr[31]}}, <b>instr[31:25], instr[11:7]</b> }	S-Type
10	{{19{instr[31]}}, <b>instr[31], instr[7], instr[30:25], instr[11:8], 1'b0</b> }	B-Type

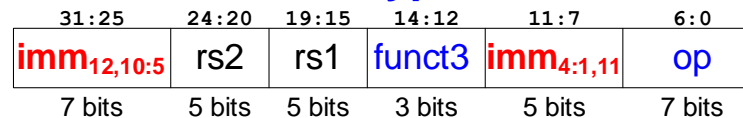
## I-Type



## S-Type



## B-Type







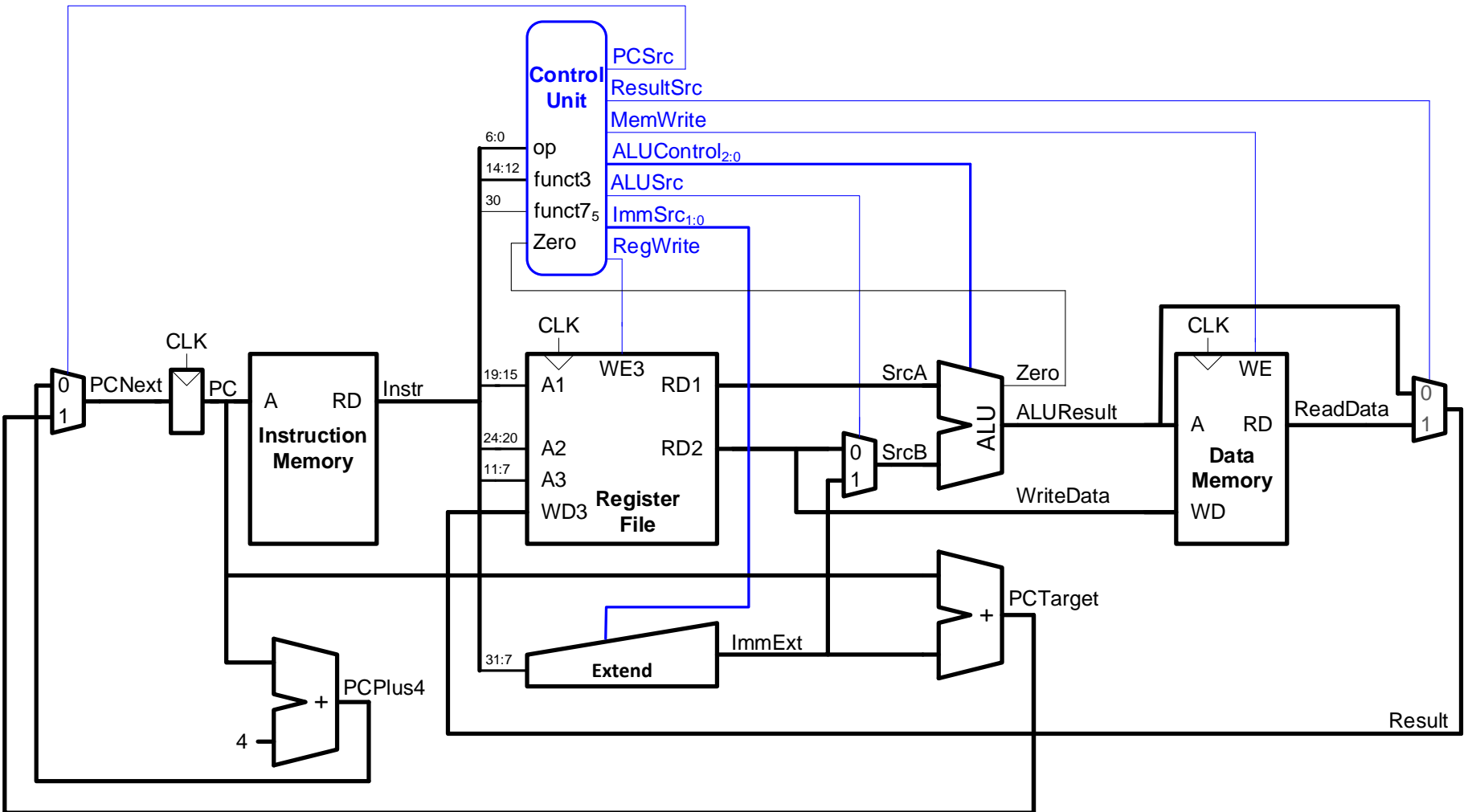








# Single-Cycle RISC-V Processor













# Chapter 7: Microarchitecture

## **Single-Cycle Control**







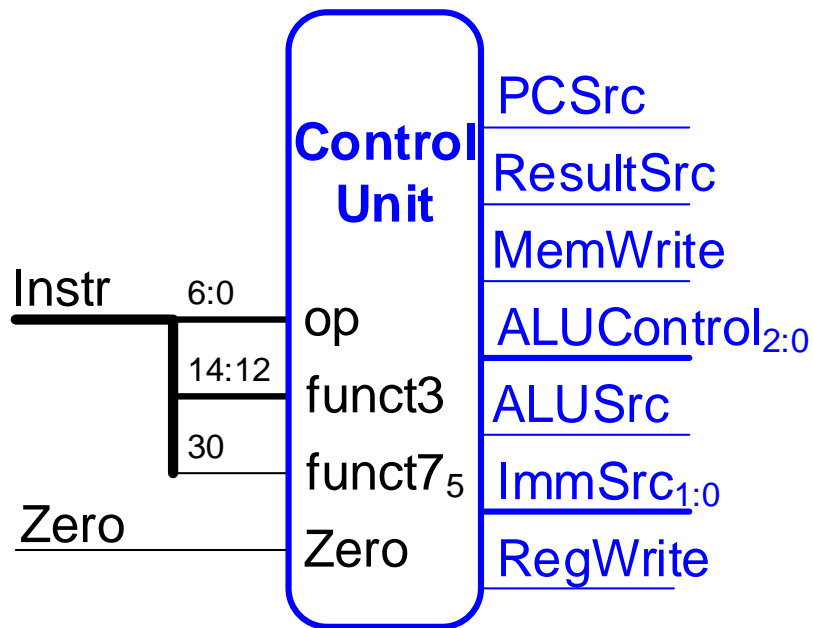




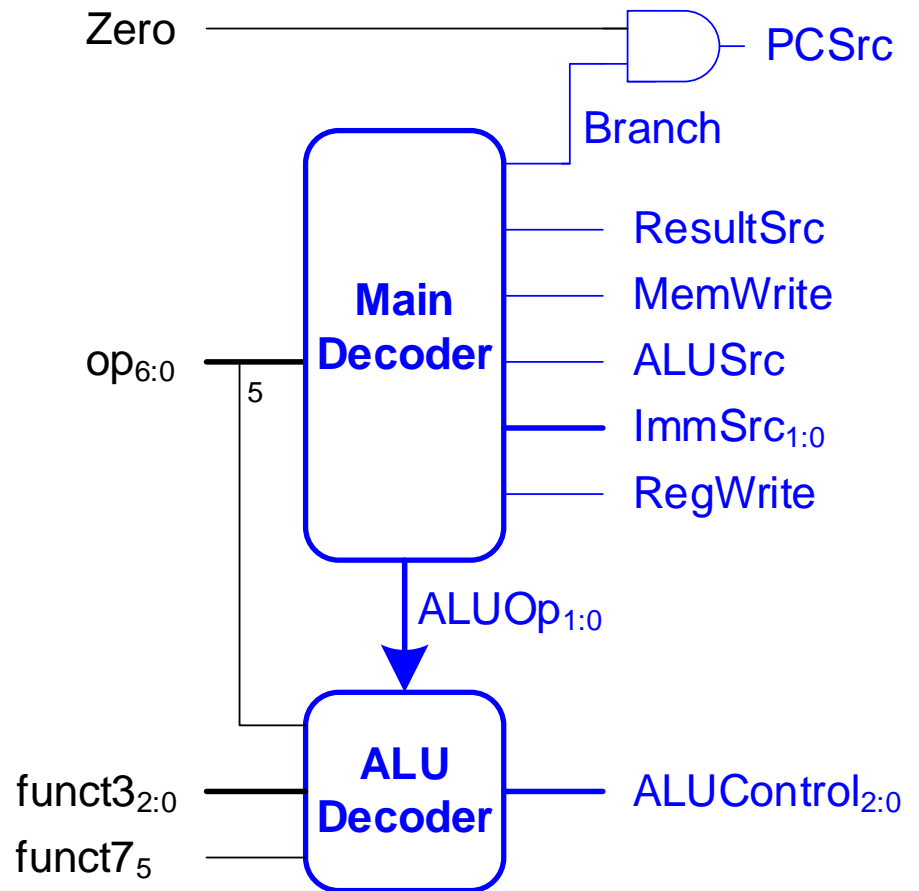


# Single-Cycle Control

## High-Level View



## Low-Level View









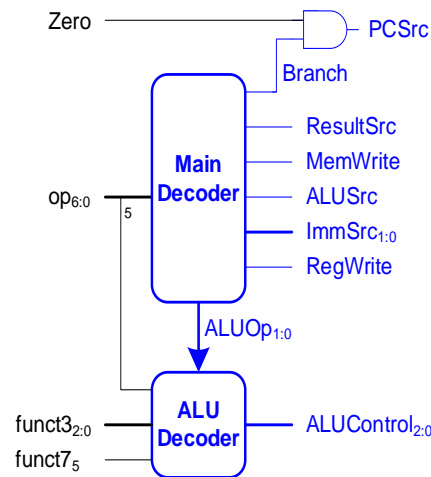






# Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw							
35	sw							
51	R-type							
99	beq							









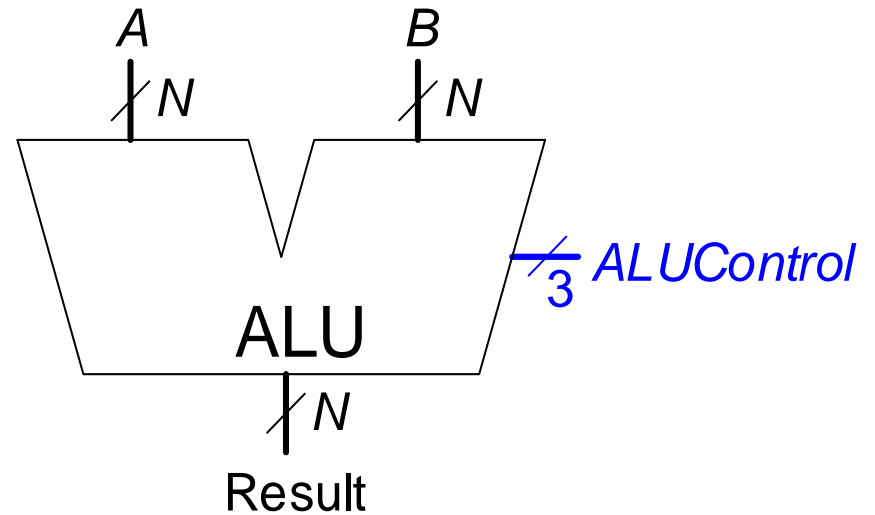






# Review: ALU

ALUControl <sub>2:0</sub>	Function
000	add
001	subtract
010	and
011	or
101	SLT







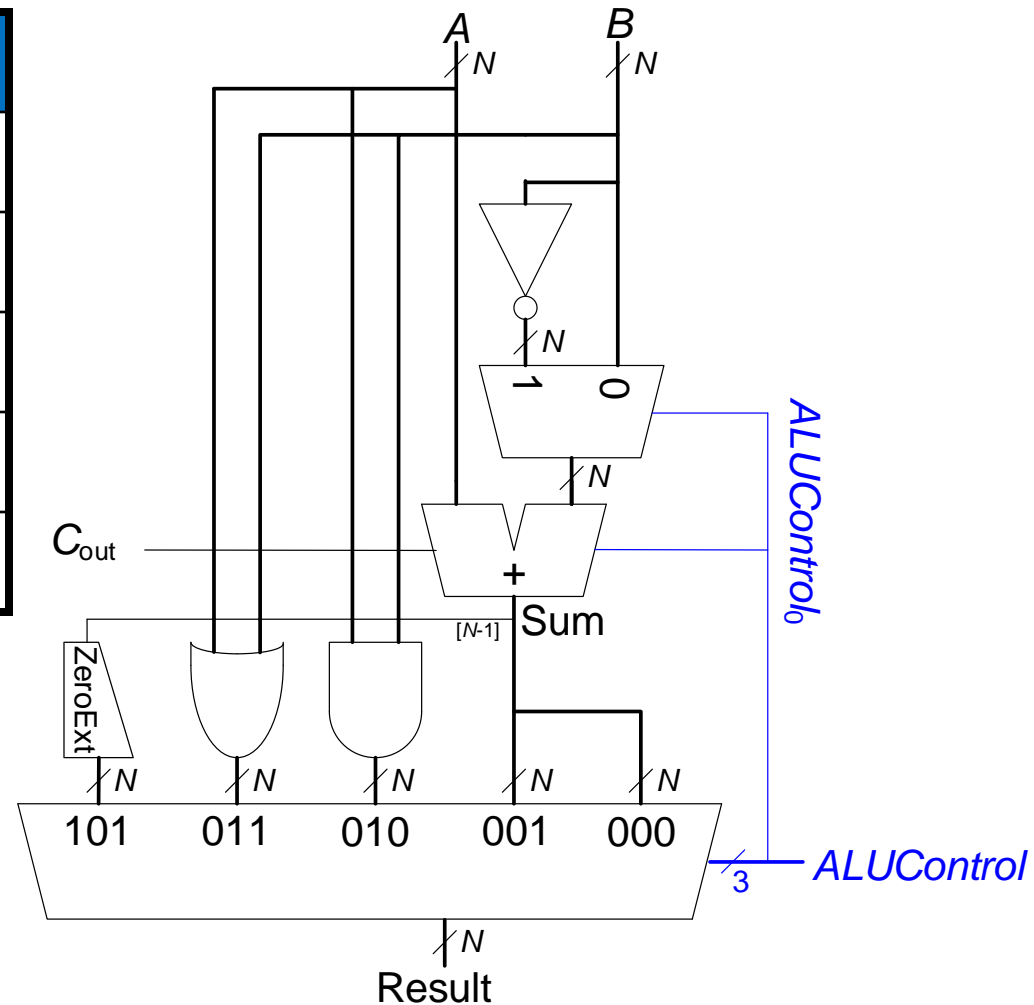






# Review: ALU

ALUControl <sub>2:0</sub>	Function
000	add
001	subtract
010	and
011	or
101	SLT







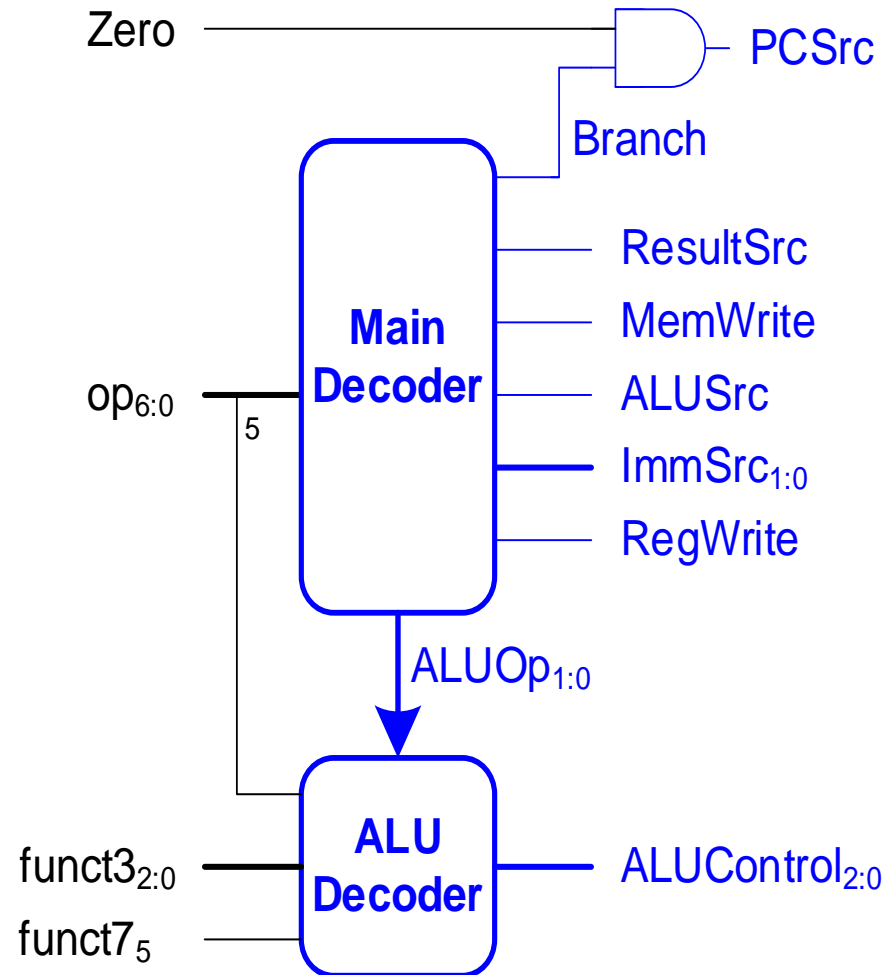








# Single-Cycle Control: ALU Decoder









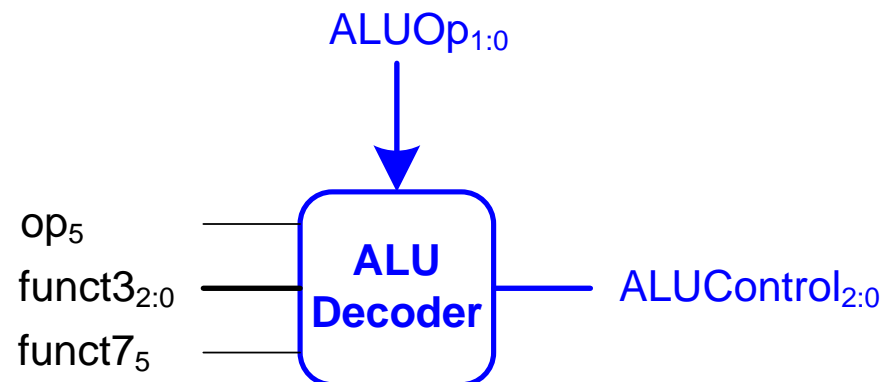






# Single-Cycle Control: ALU Decoder

<i>ALUOp</i>	<i>funct3</i>	<i>op<sub>5</sub></i> , <i>funct7<sub>5</sub></i>	Instruction	<i>ALUControl<sub>2:0</sub></i>
00	x	x	<b>lw, sw</b>	000 (add)
01	x	x	<b>beq</b>	001 (subtract)
10	000	00, 01, 10	<b>add</b>	000 (add)
	000	11	<b>sub</b>	001 (subtract)
	010	x	<b>slt</b>	101 (set less than)
	110	x	<b>or</b>	011 (or)
	111	x	<b>and</b>	010 (and)









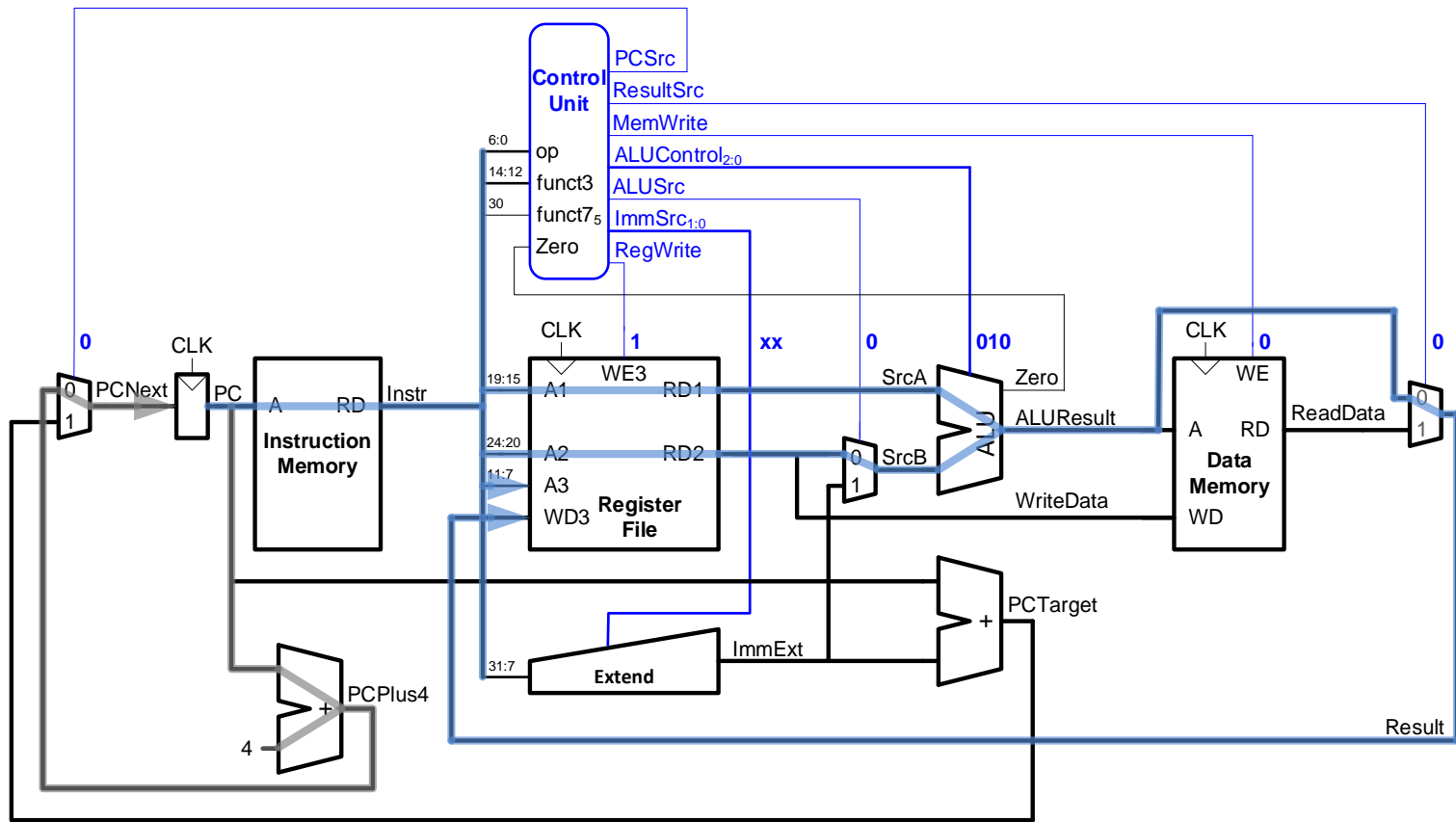






# Example: and

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
51	R-type	1	XX	0	0	0	0	10



and x5, x6, x7











## Chapter 7: Microarchitecture

# **Extending the Single-Cycle Processor**













# Extended Functionality: I-Type ALU

Enhance the single-cycle processor to handle **I-Type ALU instructions**: `addi`, `andi`, `ori`, and `slli`

- **Similar to R-type** instructions
- But **second source** comes from **immediate**
- Change ***ALUSrc*** to select the immediate
- And ***ImmSrc*** to pick the correct immediate













# Extended Functionality: I-Type ALU

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	beq	0	10	0	0	X	1	01
19	I-type	1	00	1	0	0	0	10







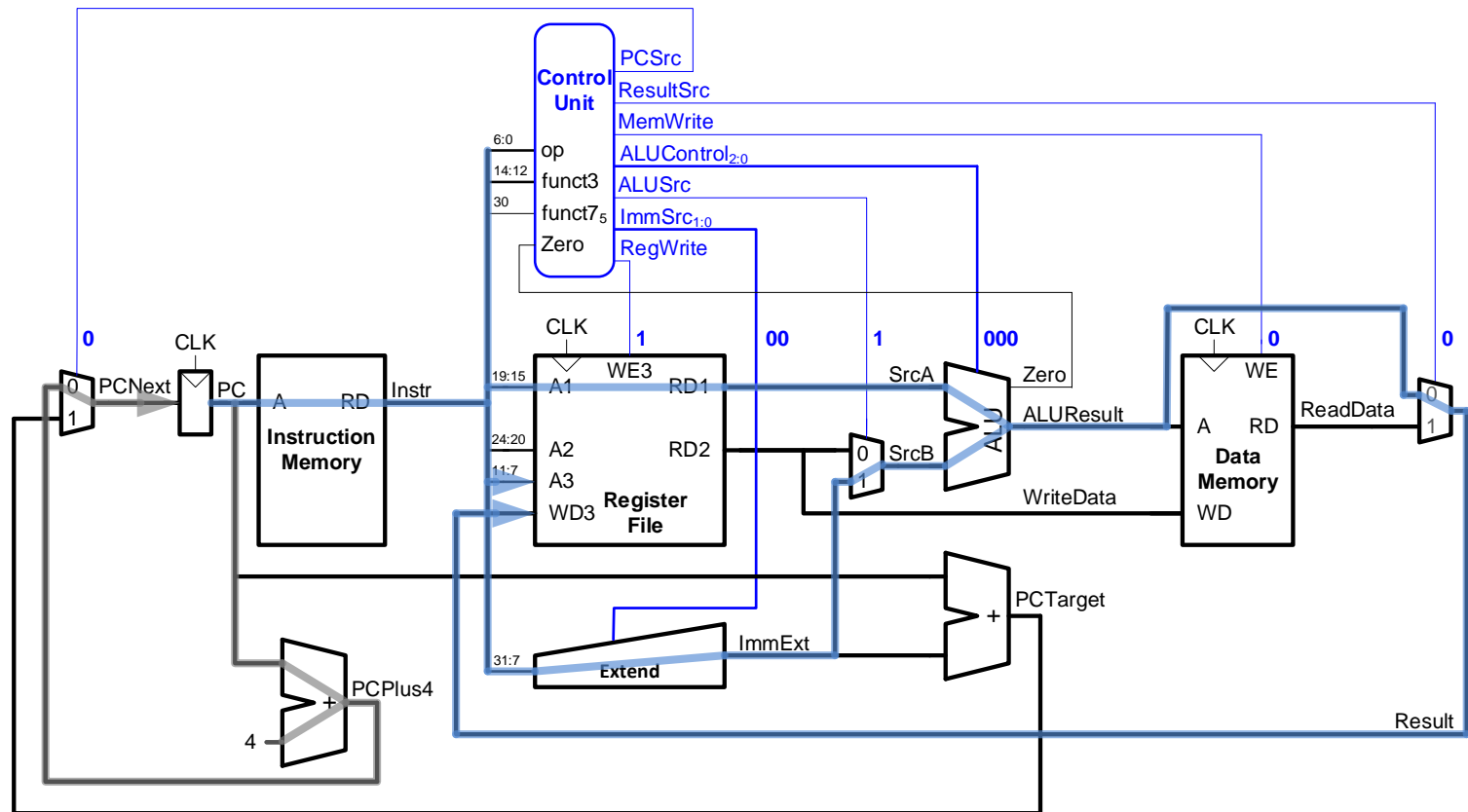






# Extended Functionality: addi

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
19	I-type	1	00	1	0	0	0	10



addi x5, x6, -33











# Extended Functionality: `jal`

Enhance the single-cycle processor to handle `jal`

- **Similar to `beq`**
- But jump is **always taken**
  - *PCSrc* should be 1
- **Immediate format** is different
  - Need a new *ImmSrc* of 11
- And `jal` must **compute  $PC+4$**  and **store in `rd`**
  - Take  $PC+4$  from adder through ResultMux





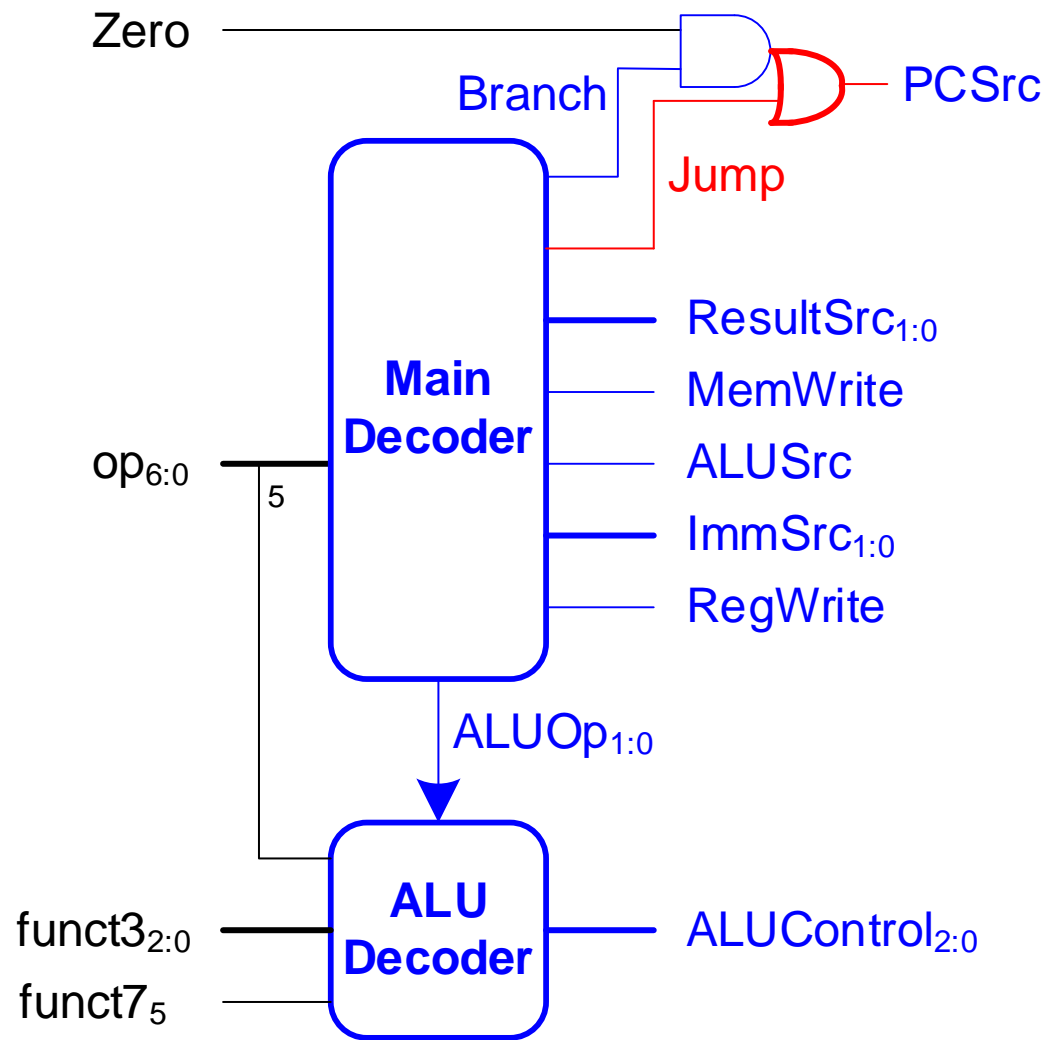








# Extended Functionality: jal











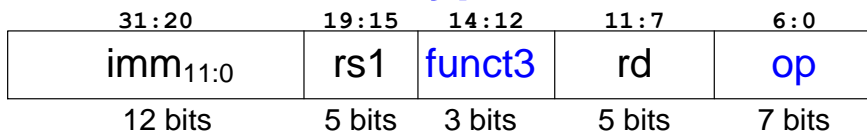




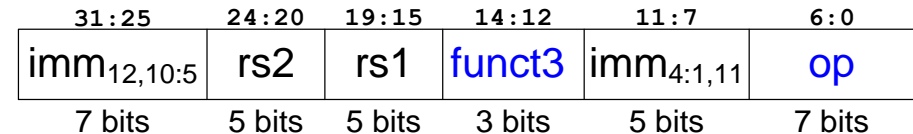
# Extended Functionality: *ImmExt*

ImmSrc <sub>1:0</sub>	ImmExt	Instruction Type
00	{{20{instr[31]}}, instr[31:20]}	I-Type
01	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S-Type
10	{{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type
<b>11</b>	<b>{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}</b>	<b>J-Type</b>

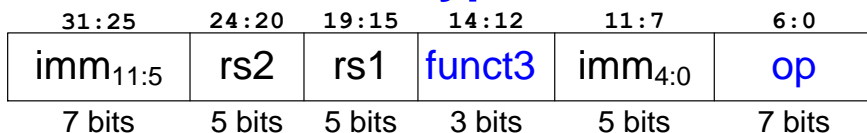
## I-Type



## B-Type



## S-Type



## J-Type















# Extended Functionality: jal

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
3	lw	1	00	1	0	01	0	00	0
35	sw	0	01	1	1	XX	0	00	0
51	R-type	1	XX	0	0	00	0	10	0
99	beq	0	10	0	0	XX	1	01	0
19	l-type	1	00	1	0	00	0	10	0
111	jal	1	11	X	0	10	0	XX	1





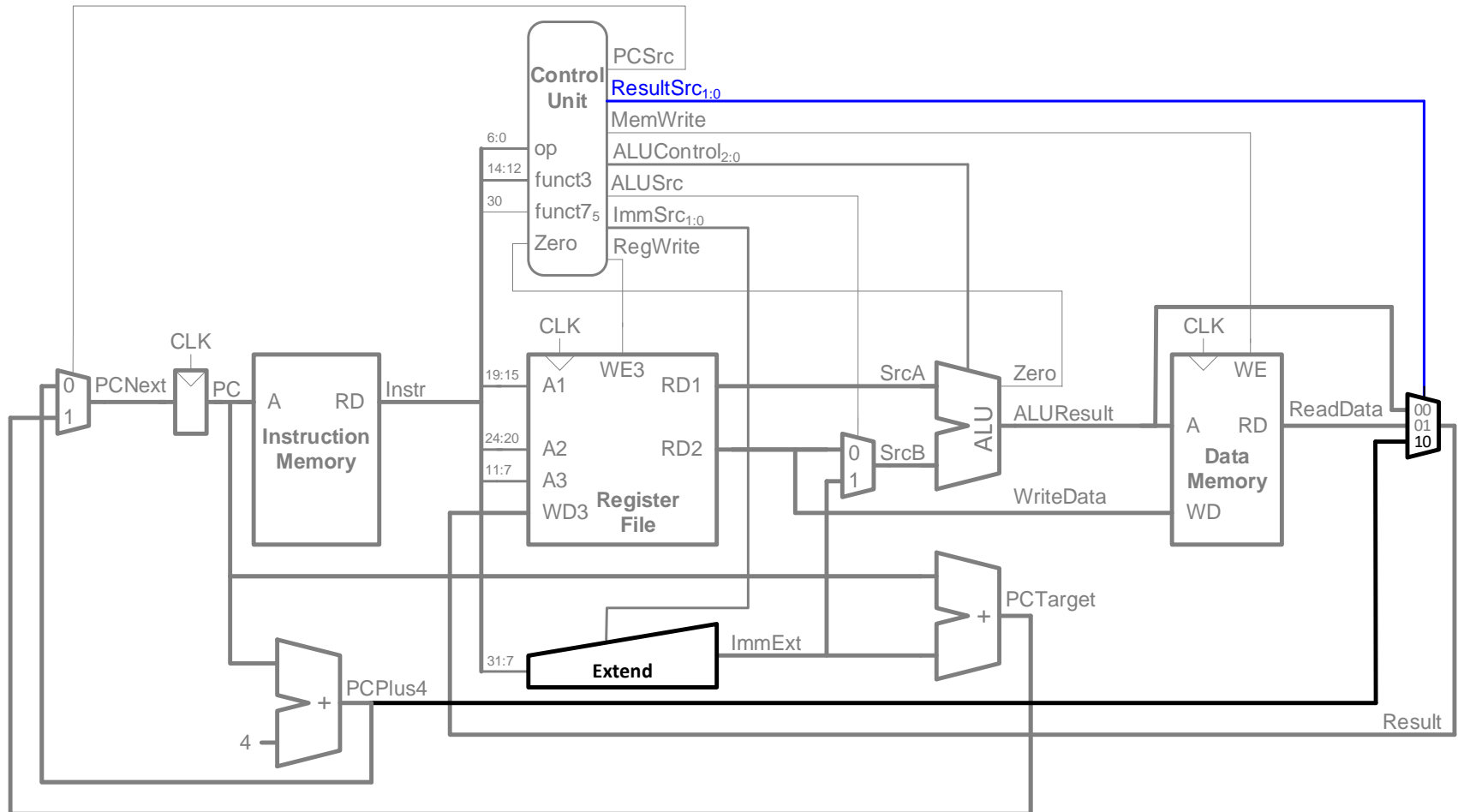






# Extended Functionality: jal

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
111	jal	1	11	X	0	10	0	XX	1















# Chapter 7: Microarchitecture

## **Single-Cycle Performance**













# Processor Performance

## Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x  $T_c$





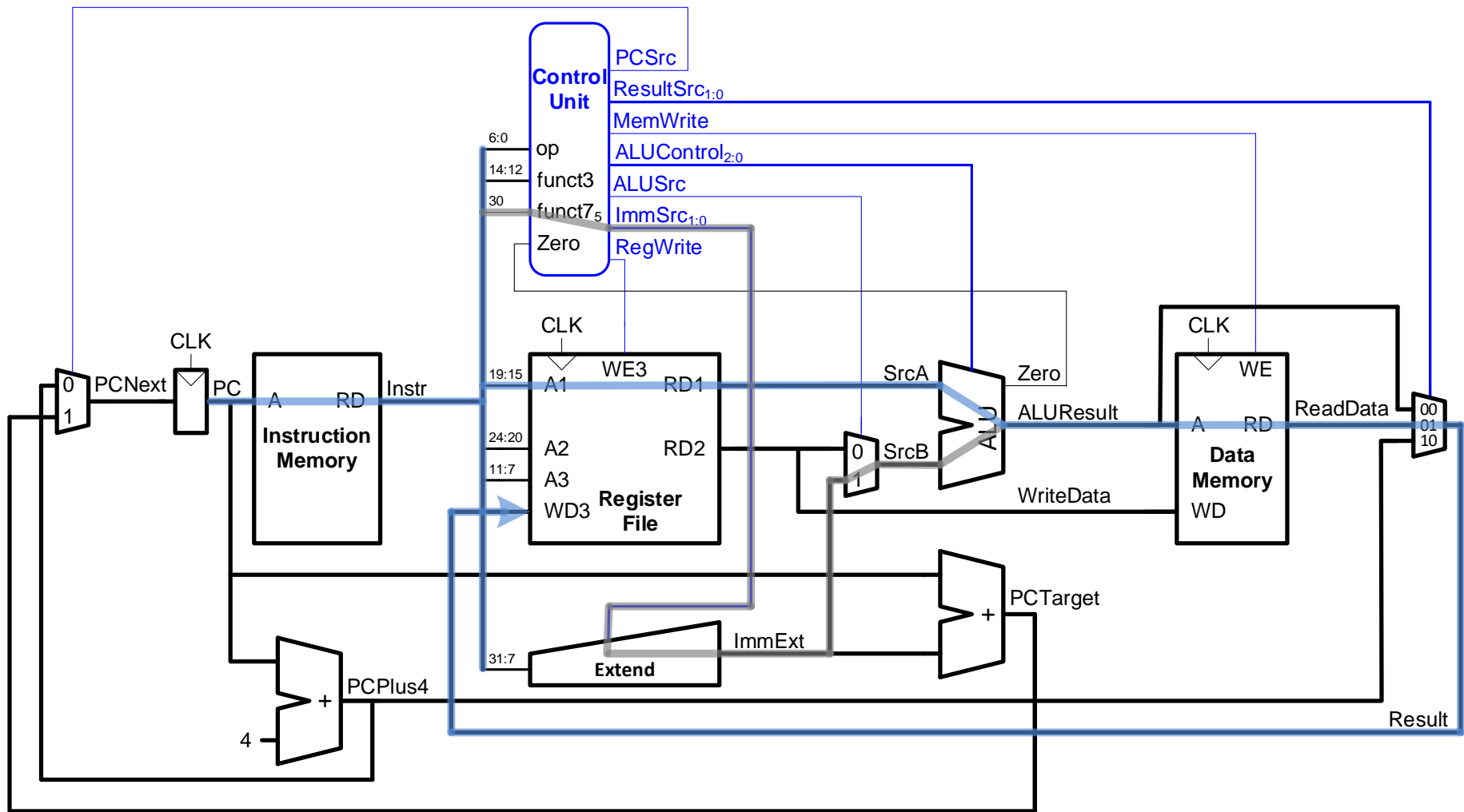








# Single-Cycle Processor Performance



$T_c$  limited by critical path (1w)











# Single-Cycle Processor Performance

- **Single-cycle critical path:**

$$T_{c\_single} = t_{pcq\_PC} + t_{mem} + \max[t_{RFread}, t_{dec} + t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

- **Typically, limiting paths are:**

- memory, ALU, register file

- So,  $T_{c\_single} = t_{pcq\_PC} + t_{mem} + t_{RFread} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$   
 $= t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$













# Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	$t_{pcq\_PC}$	40
Register setup	$t_{setup}$	50
Multiplexer	$t_{mux}$	30
AND-OR gate	$t_{AND-OR}$	20
ALU	$t_{ALU}$	120
Decoder (Control Unit)	$t_{dec}$	25
Extend unit	$t_{ext}$	35
Memory read	$t_{mem}$	200
Register file read	$t_{RFread}$	100
Register file setup	$t_{RFsetup}$	60

$$T_{c\_single} = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$$
$$=$$













# Single-Cycle Performance Example

Program with 100 billion instructions:

$$\text{Execution Time} = \# \text{ instructions} \times \text{CPI} \times T_c$$













## Chapter 7: Microarchitecture

# **Multicycle RISC-V Processor**











# Single- vs. Multicycle Processor

- **Single-cycle:**
  - + simple
  - cycle time limited by longest instruction ( $1_w$ )
  - separate memories for instruction and data
  - 3 adders/ALUs
- **Multicycle processor** addresses these issues by breaking instruction into **shorter steps**
  - shorter instructions take fewer steps
  - can re-use hardware
  - cycle time is faster













# Single- vs. Multicycle Processor

- **Single-cycle:**

- + simple
- cycle time limited by longest instruction ( $\tau_w$ )
- separate memories for instruction and data
- 3 adders/ALUs

- **Multicycle:**

- + higher clock speed
- + simpler instructions run faster
- + reuse expensive hardware on multiple cycles
- sequencing overhead paid many times

**Same design steps  
as single-cycle:**

- **first datapath**
- **then control**







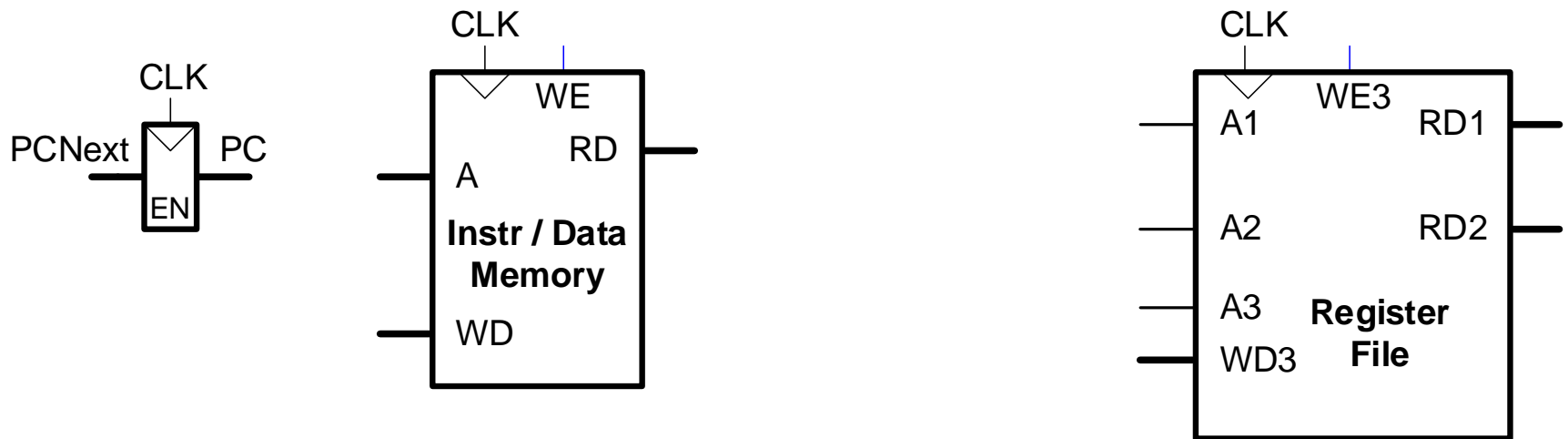






# Multicycle State Elements

Replace separate Instruction and Data memories with a **single unified memory** – more realistic









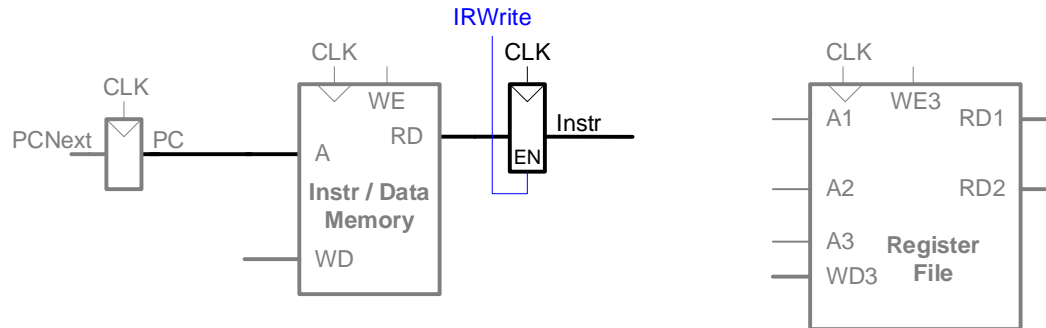






# Multicycle Datapath: Instruction Fetch

## STEP 1: Fetch instruction







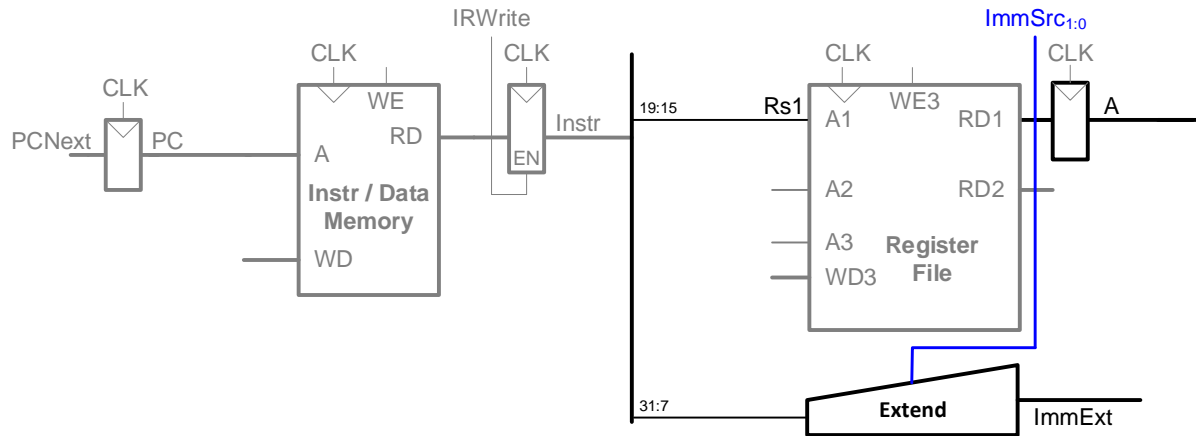






# Multicycle Datapath: $1_w$ Get Sources

**STEP 2:** Read source operand from RF and extend immediate









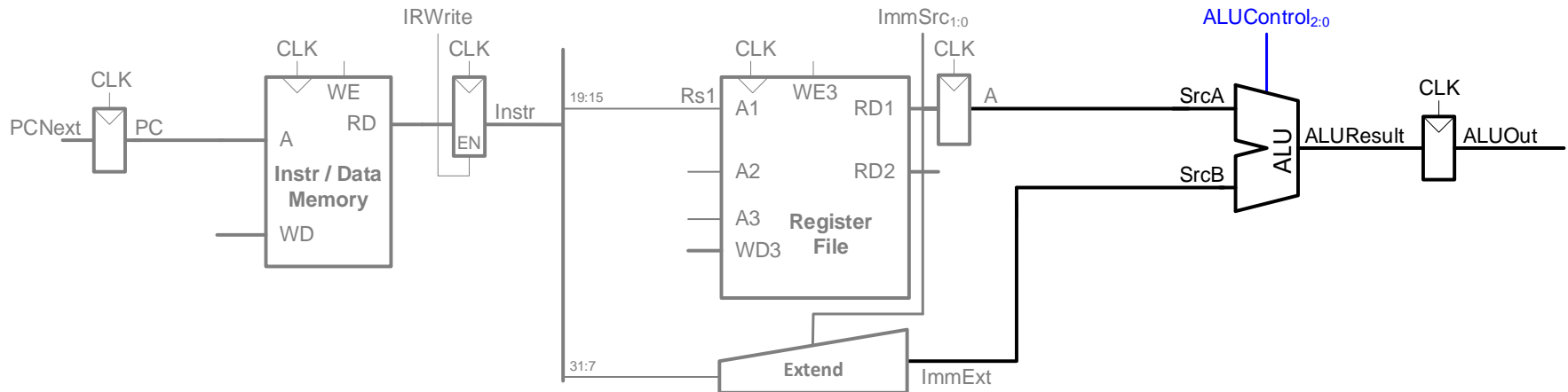






# Multicycle Datapath: $1_w$ Address

## STEP 3: Compute the memory address









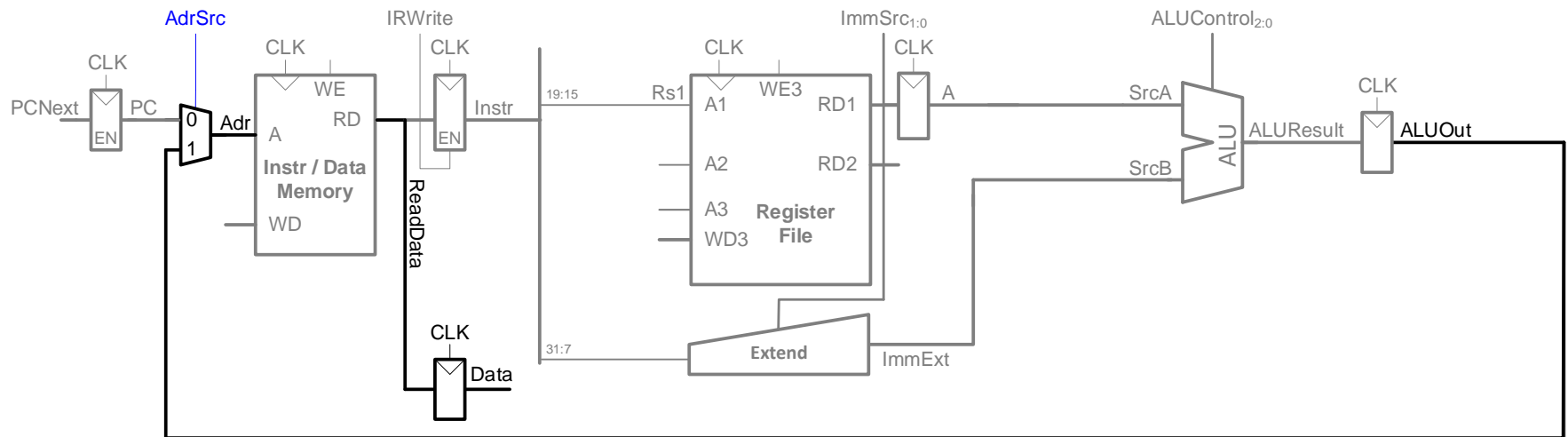






# Multicycle Datapath: $1_W$ Memory Read

## STEP 4: Read data from memory









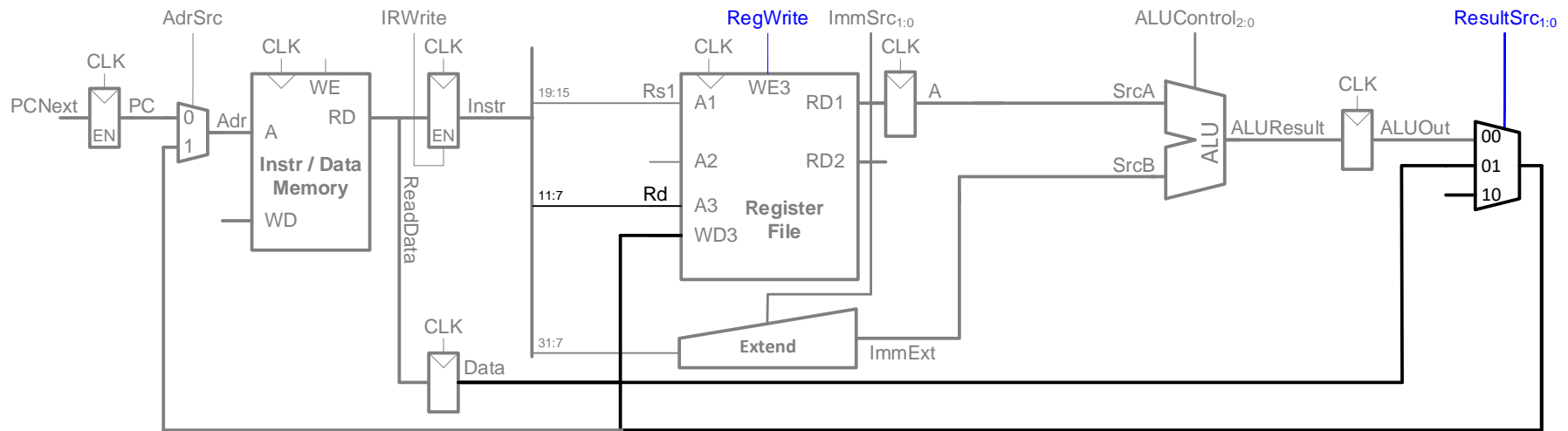






# Multicycle Datapath: $1_W$ Write Register

## STEP 5: Write data back to register file







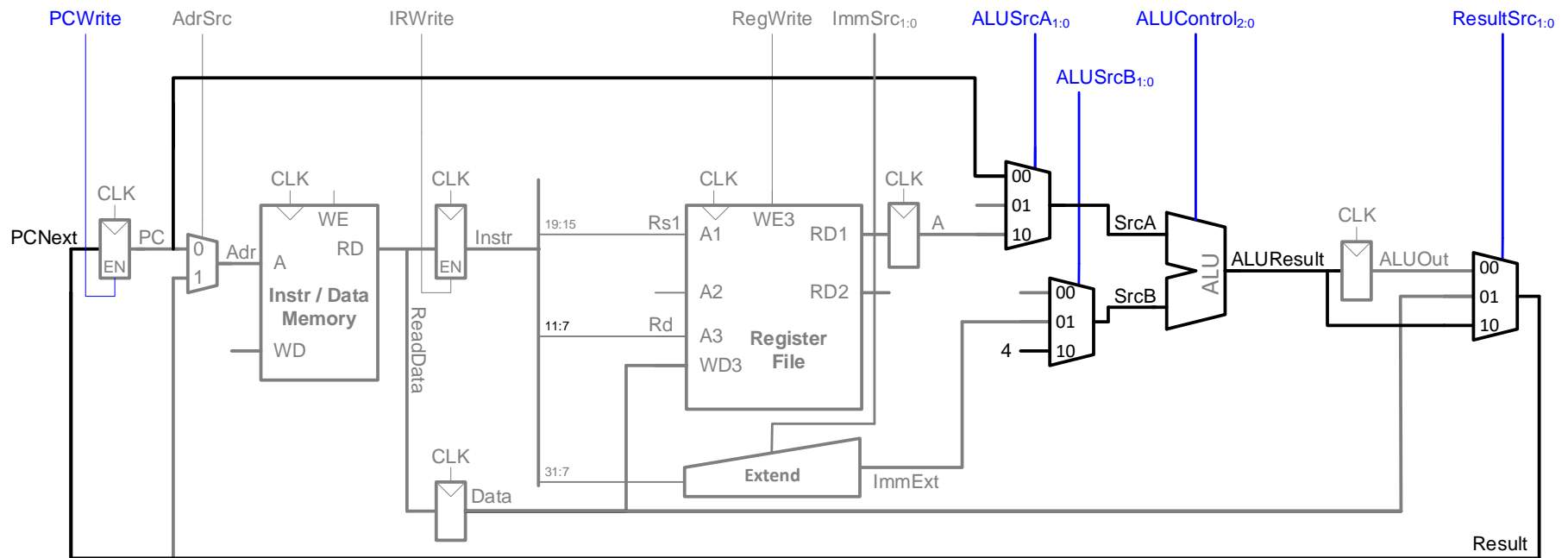






# Multicycle Datapath: Increment PC

**STEP 6:** Increment PC:  $PC = PC + 4$















## Chapter 7: Microarchitecture

# **Multicycle Datapath: Other Instructions**







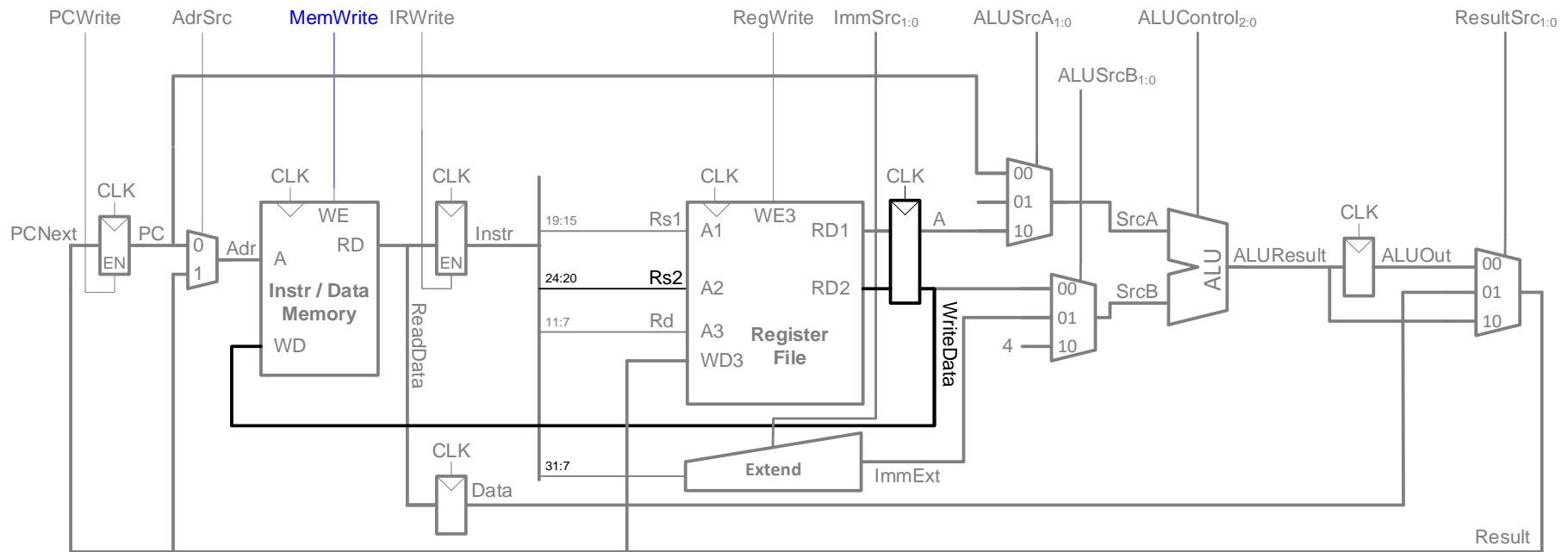






# Multicycle Datapath: sw

Write data in **rs2** to memory









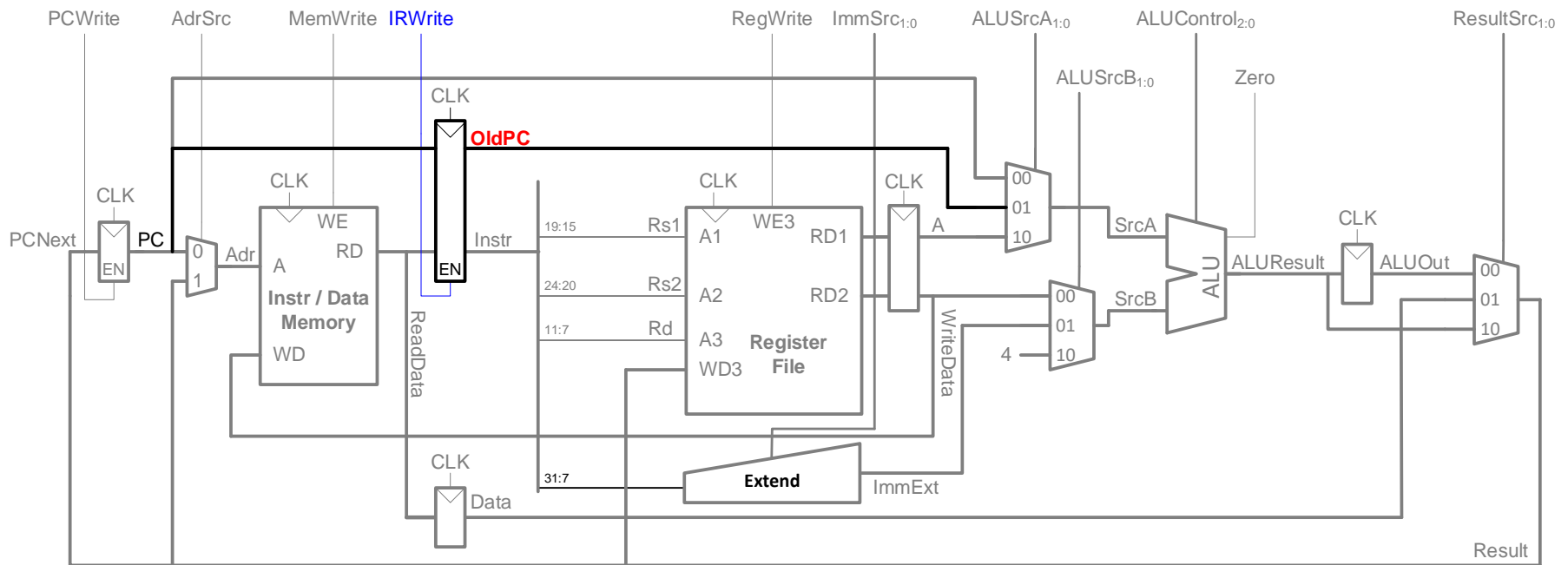






# Multicycle Datapath: beq

Calculate branch target address:  
 $BTA = PC + imm$



PC is updated in Fetch stage, so need to save **old (current) PC**



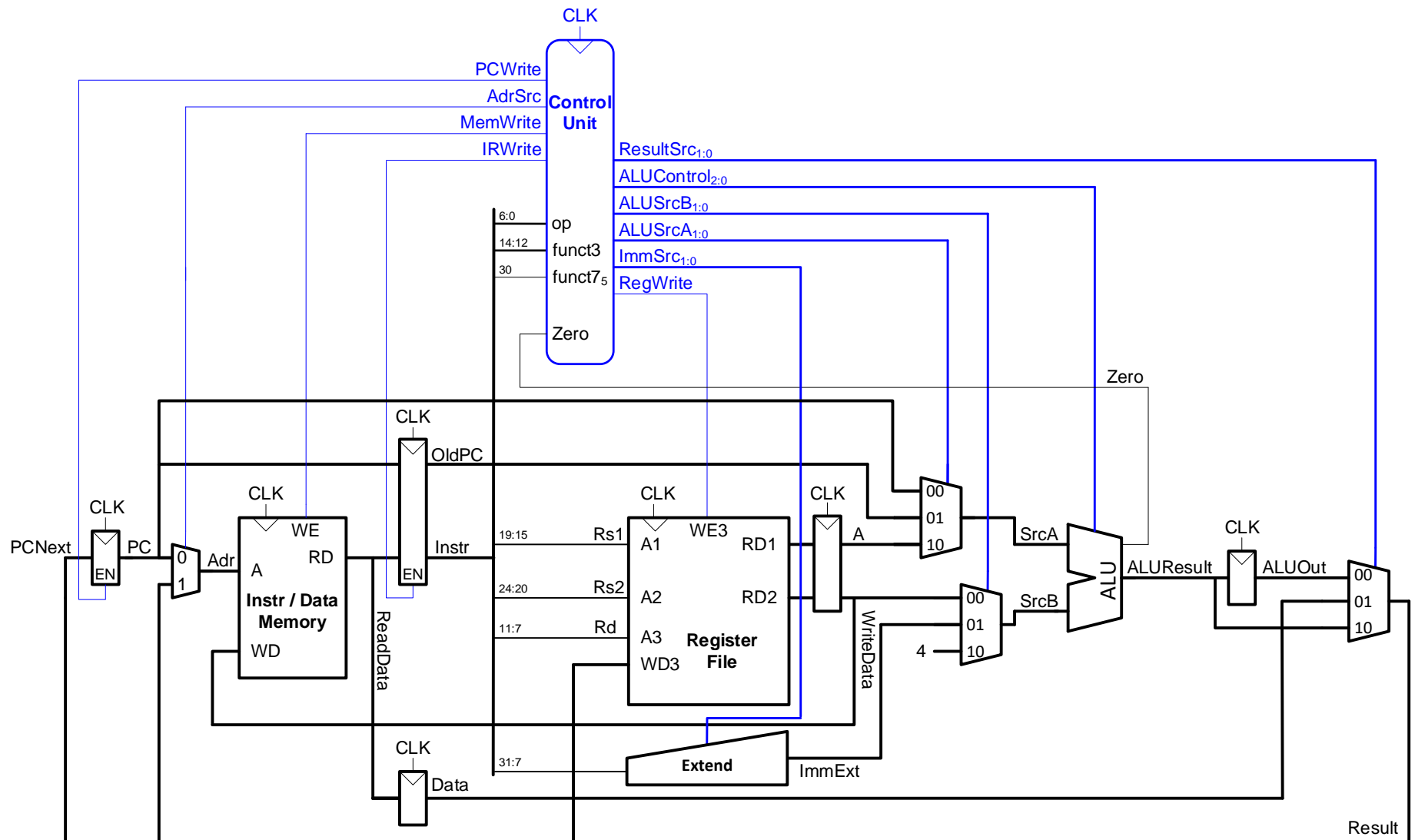








# Multicycle RISC-V Processor















## Chapter 7: Microarchitecture

# **Multicycle Control**







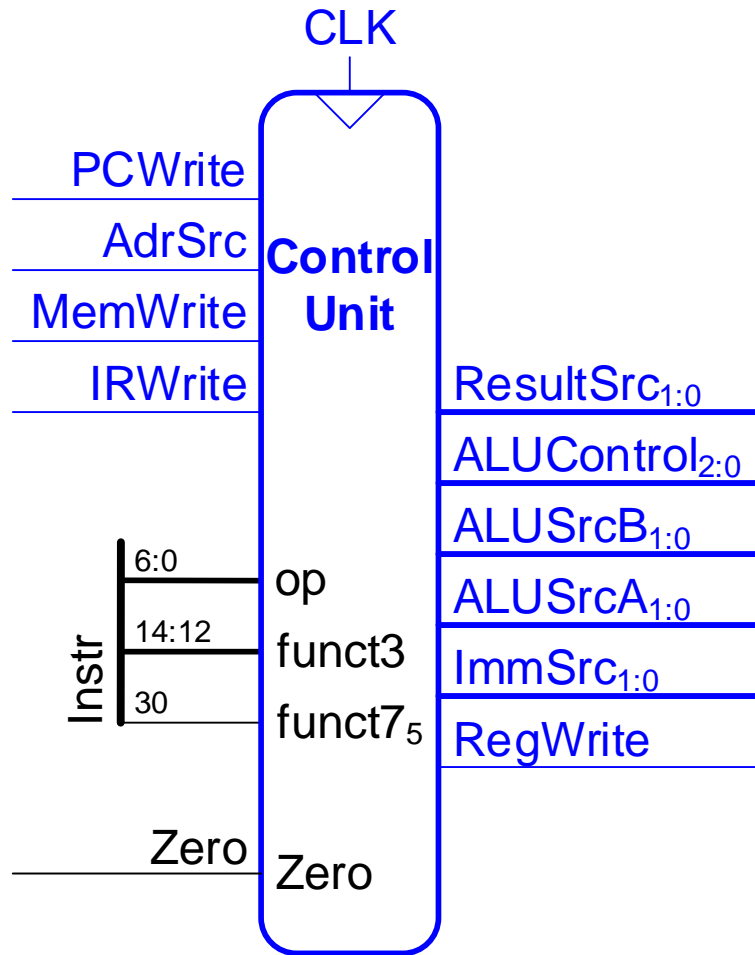




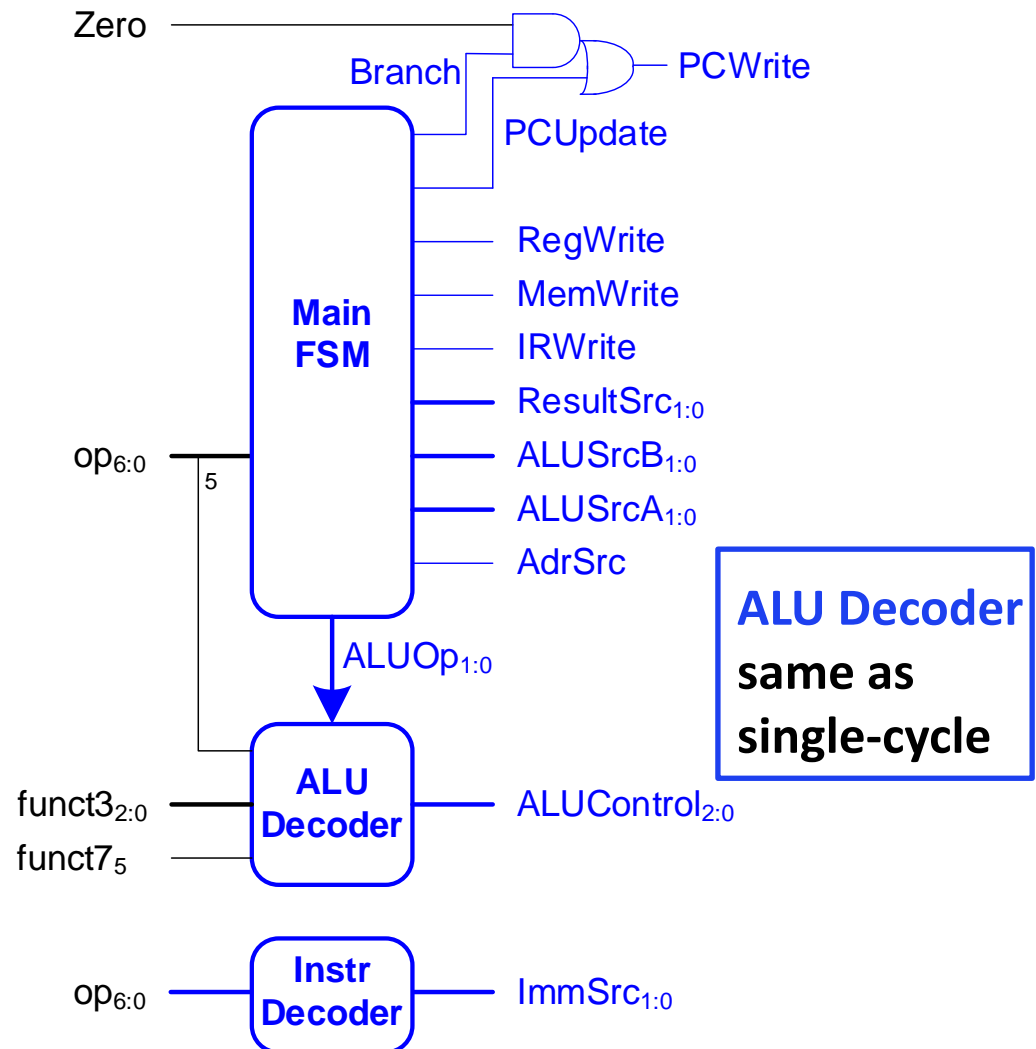


# Multicycle Control

## High-Level View



## Low-Level View







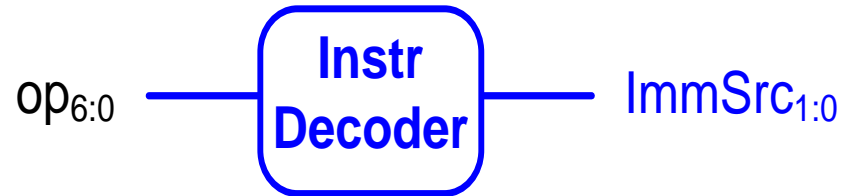








# Multicycle Control: Instruction Decoder



op	Instruction	ImmSrc
3	lw	00
35	sw	01
51	R-type	XX
99	beq	10



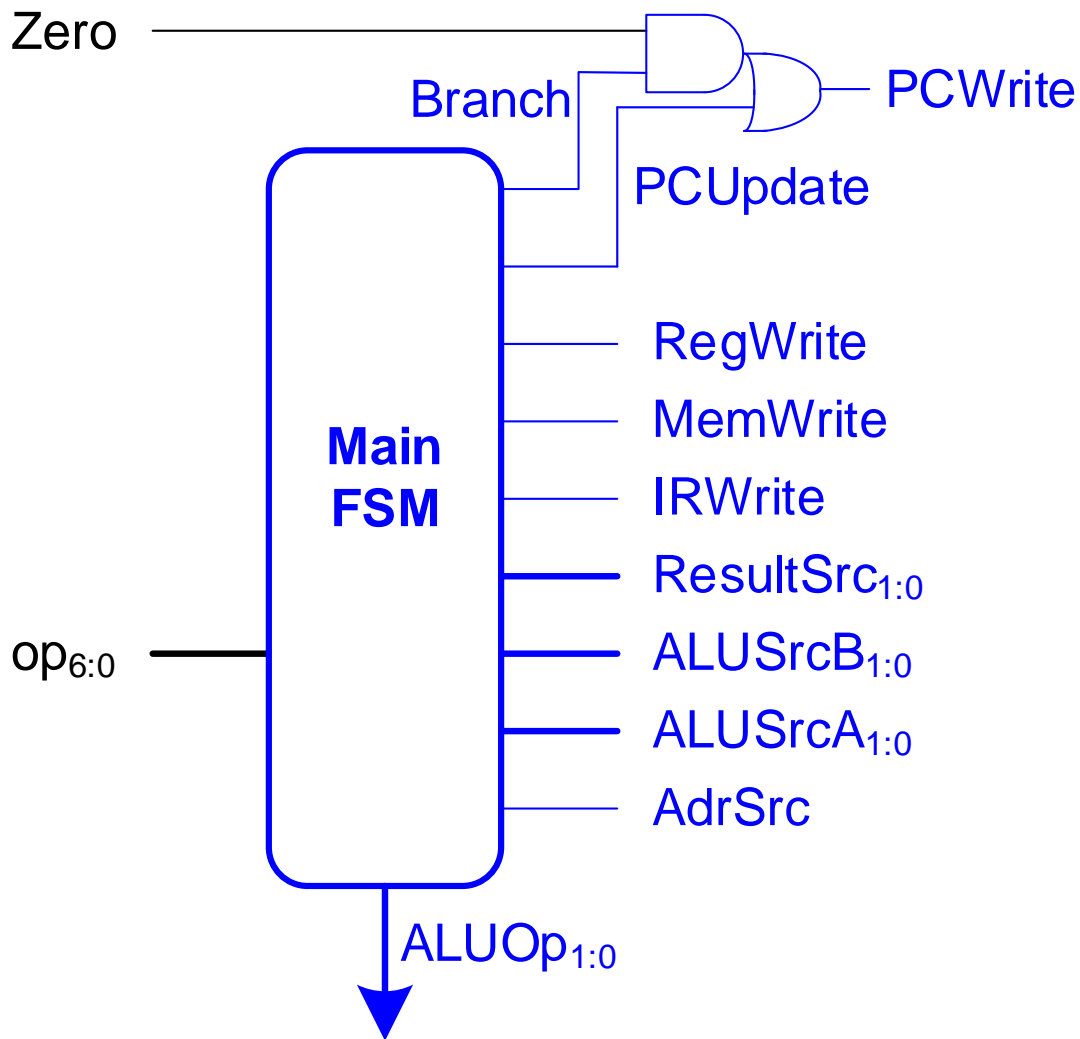








# Multicycle Control: Main FSM



**To declutter FSM:**

- **Write enable signals** (RegWrite, MemWrite, IRWrite, PCUpdate, and Branch) are **0** if not listed in a state.
- **Other signals are don't care** if not listed in a state





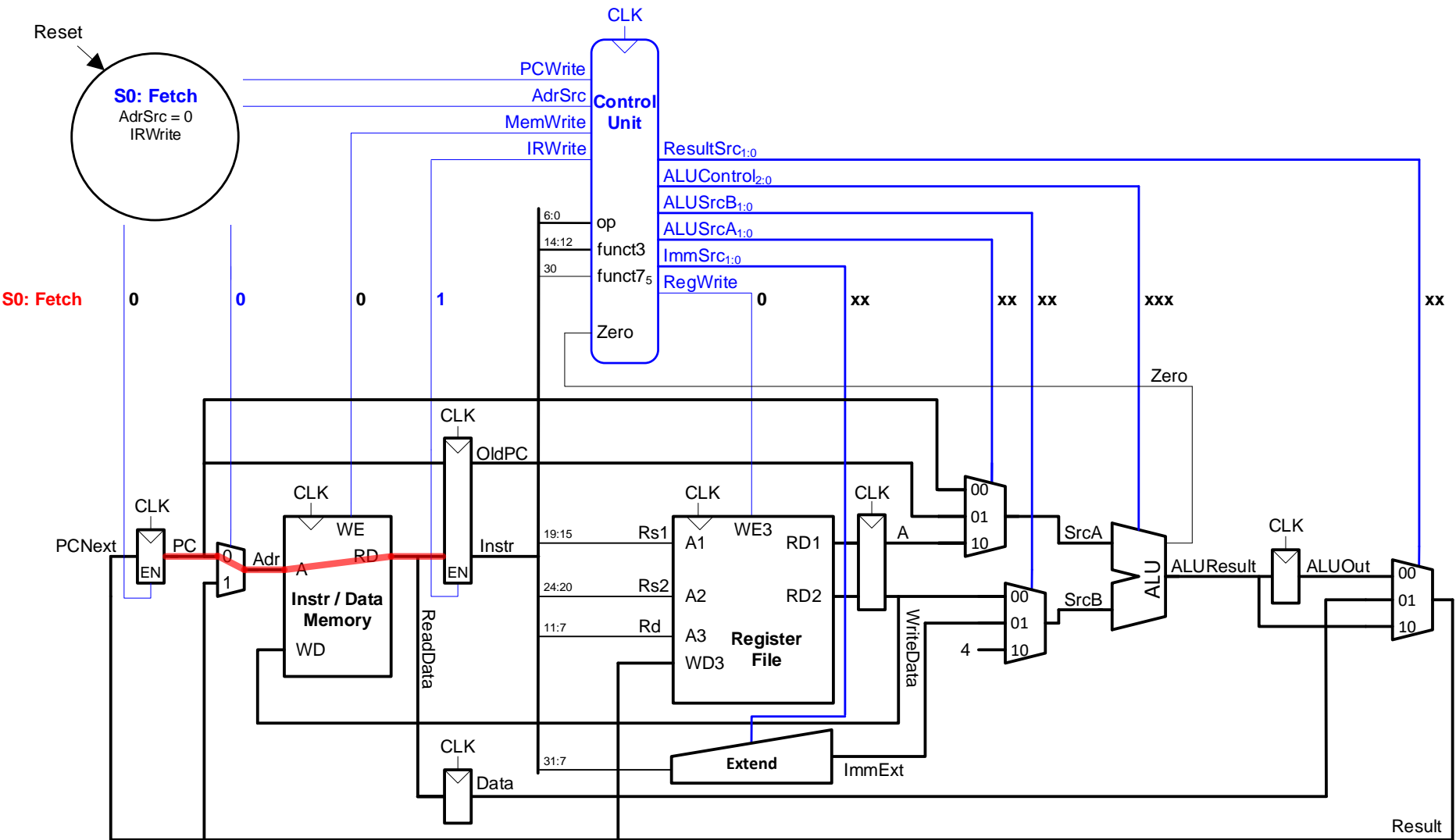








# Main FSM: Fetch







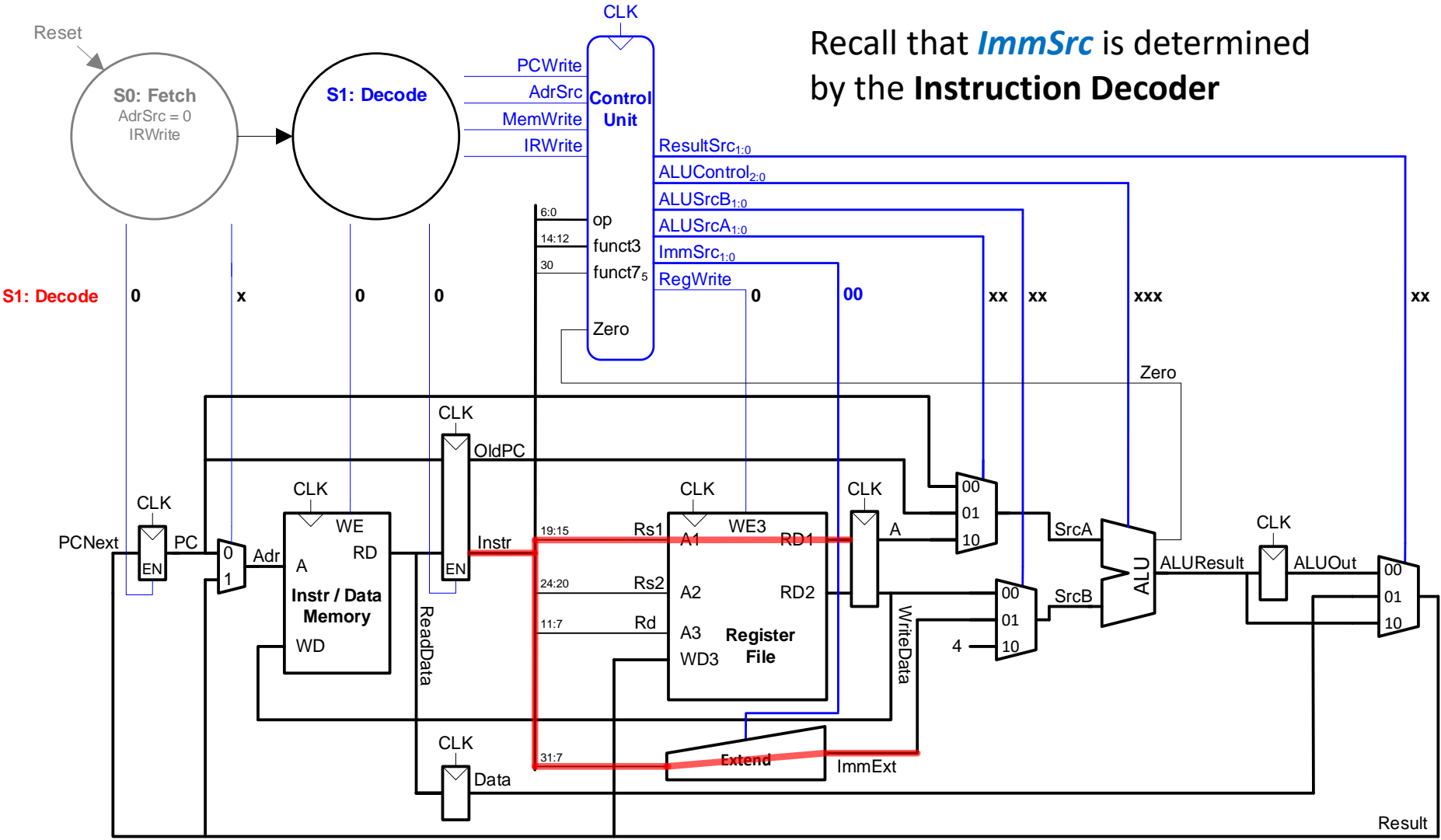








## Main FSM: Decode







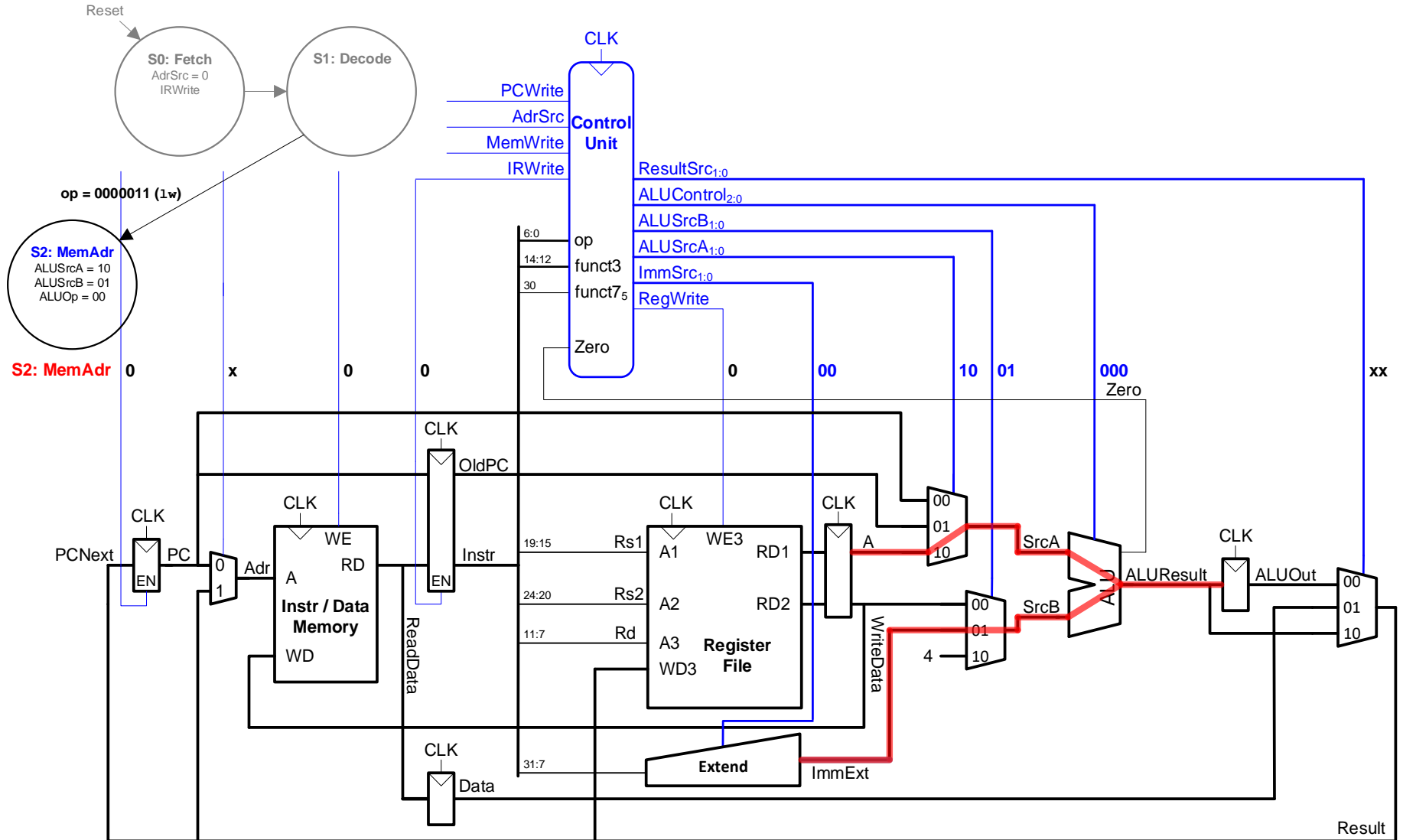








# Main FSM: Address





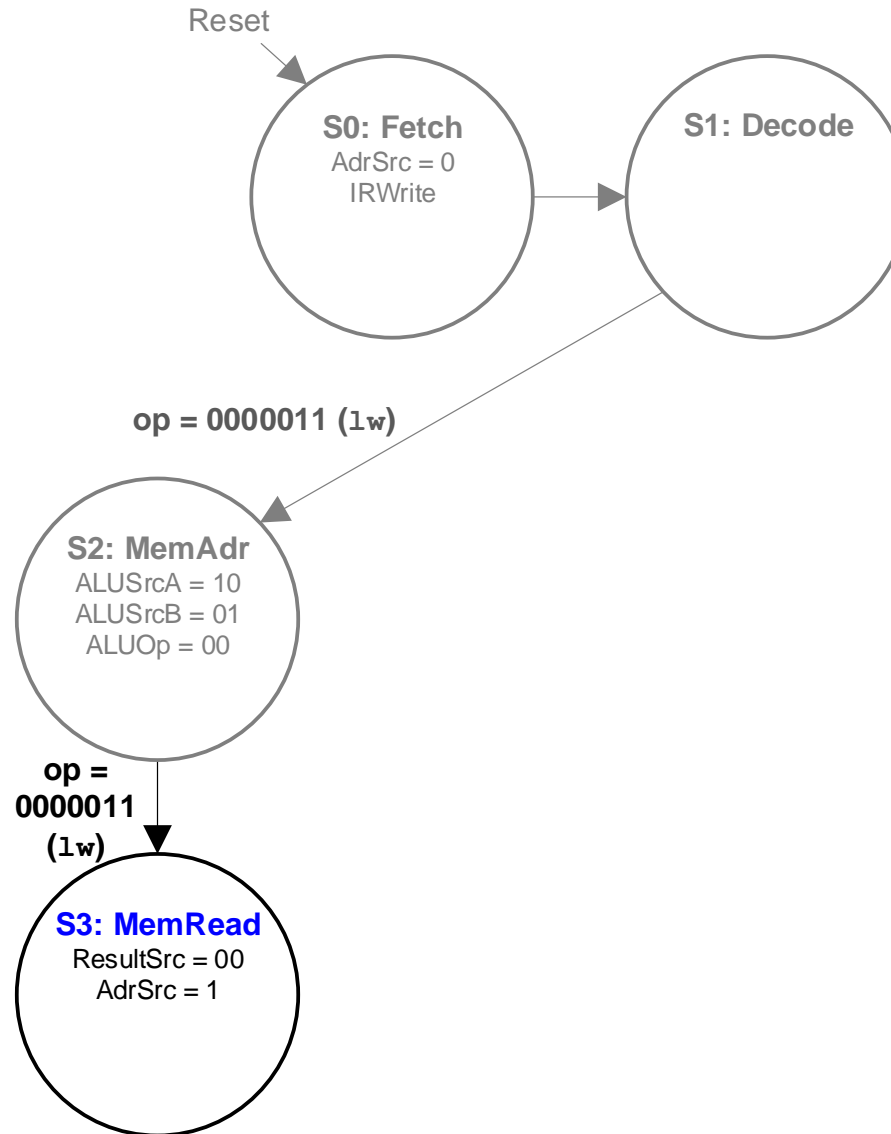








# Main FSM: Read Memory







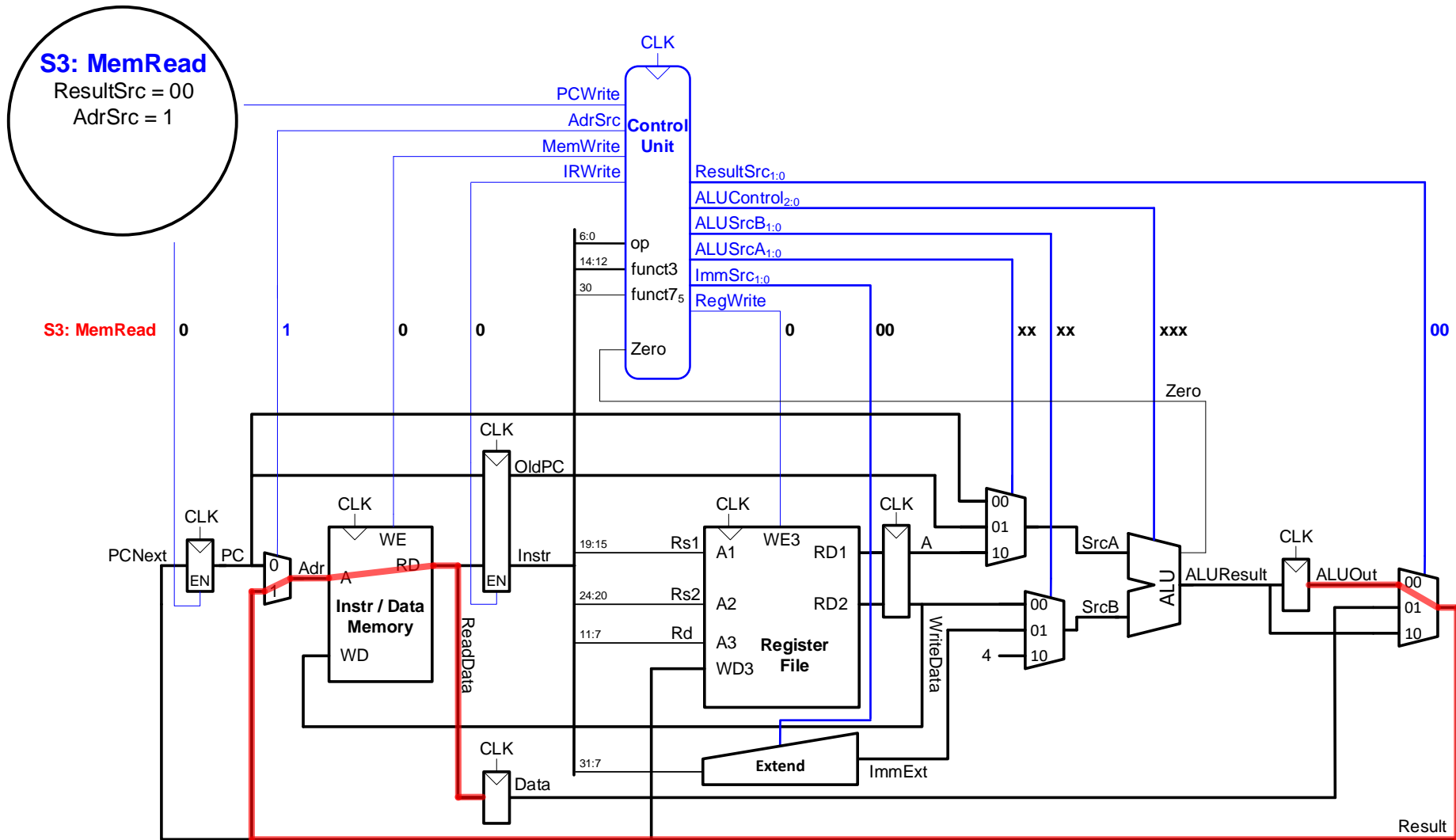








# Main FSM: Read Memory Datapath







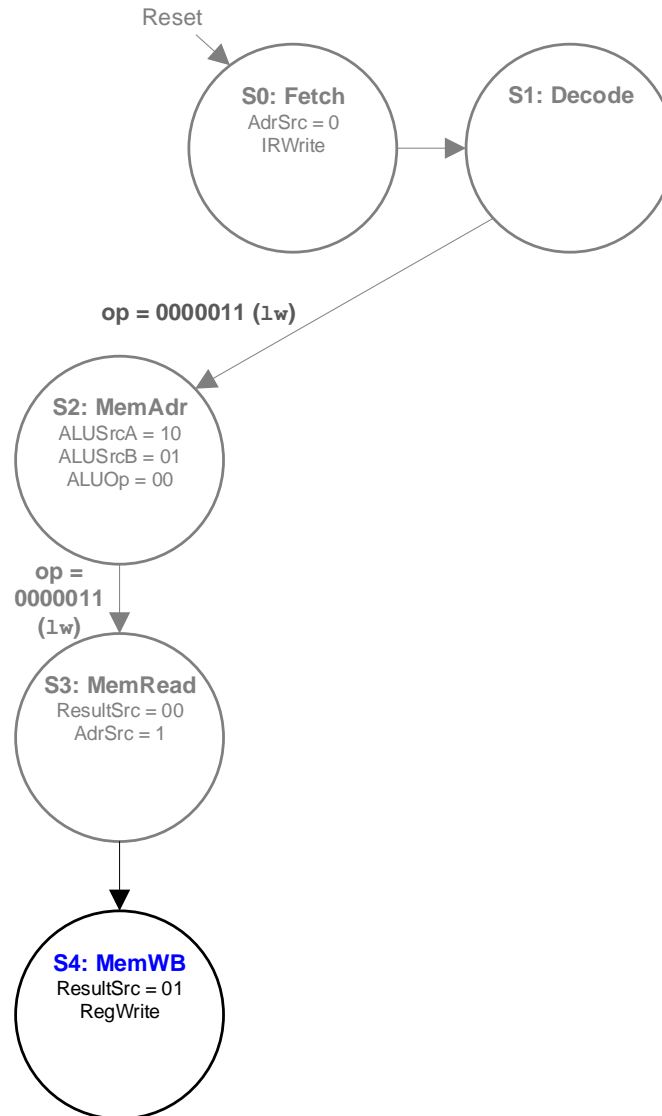








# Main FSM: Write RF







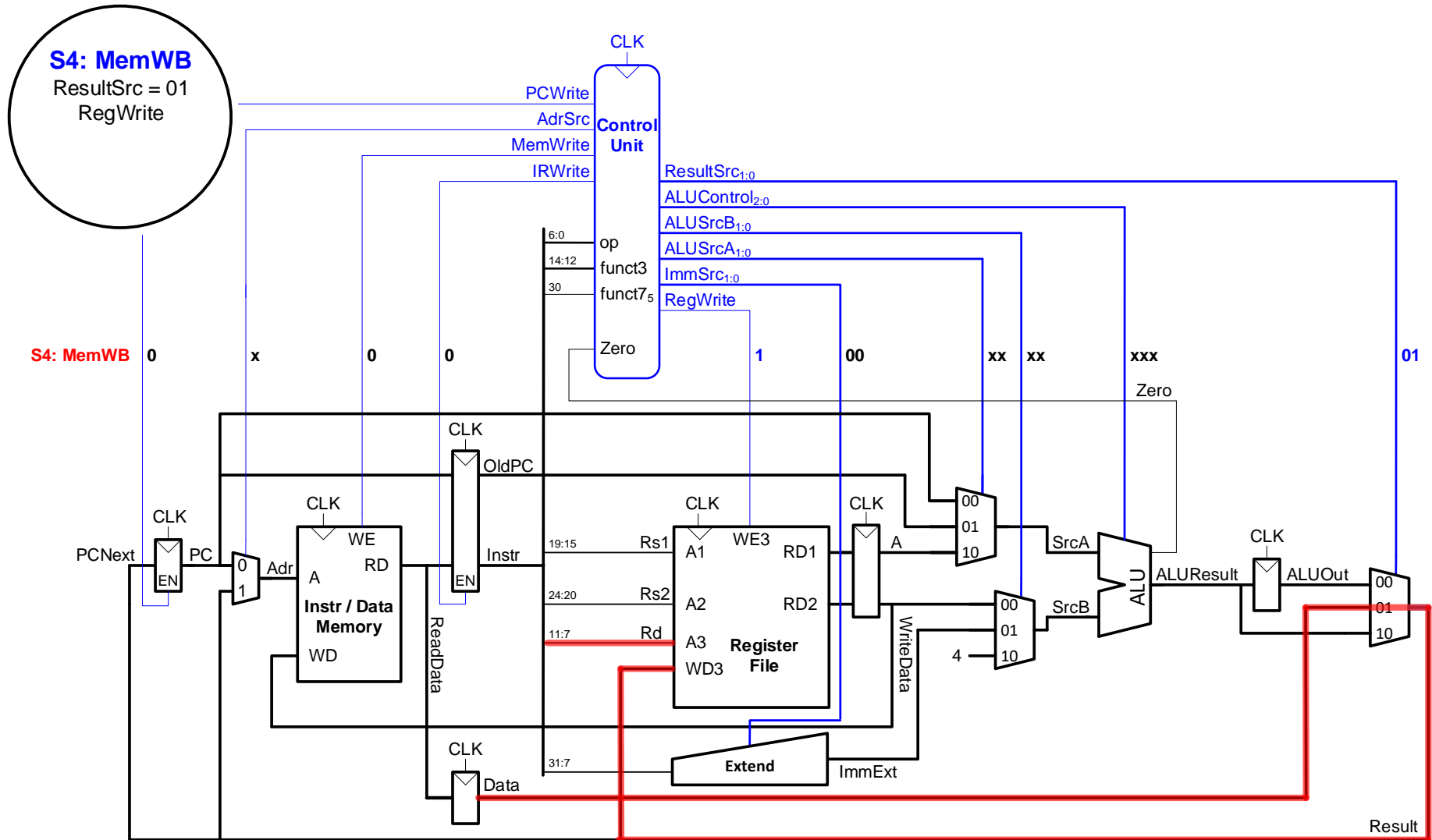








# Main FSM: Write RF Datapath







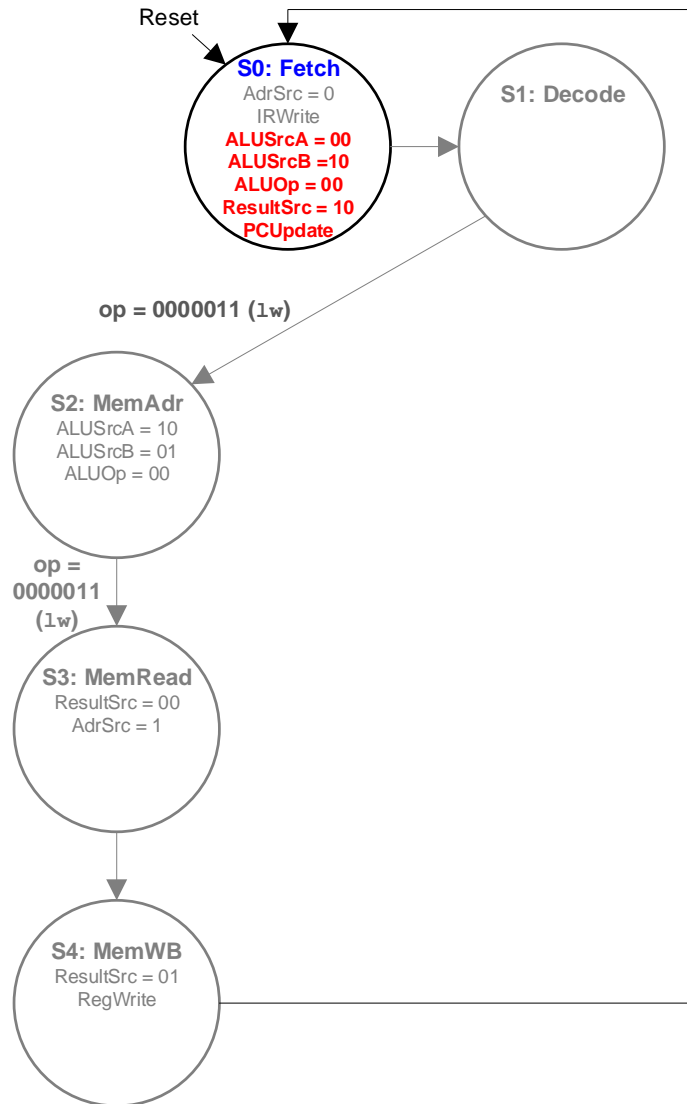






# Main FSM: Fetch Revisited

Calculate **PC+4**  
during Fetch stage  
(ALU isn't being  
used)







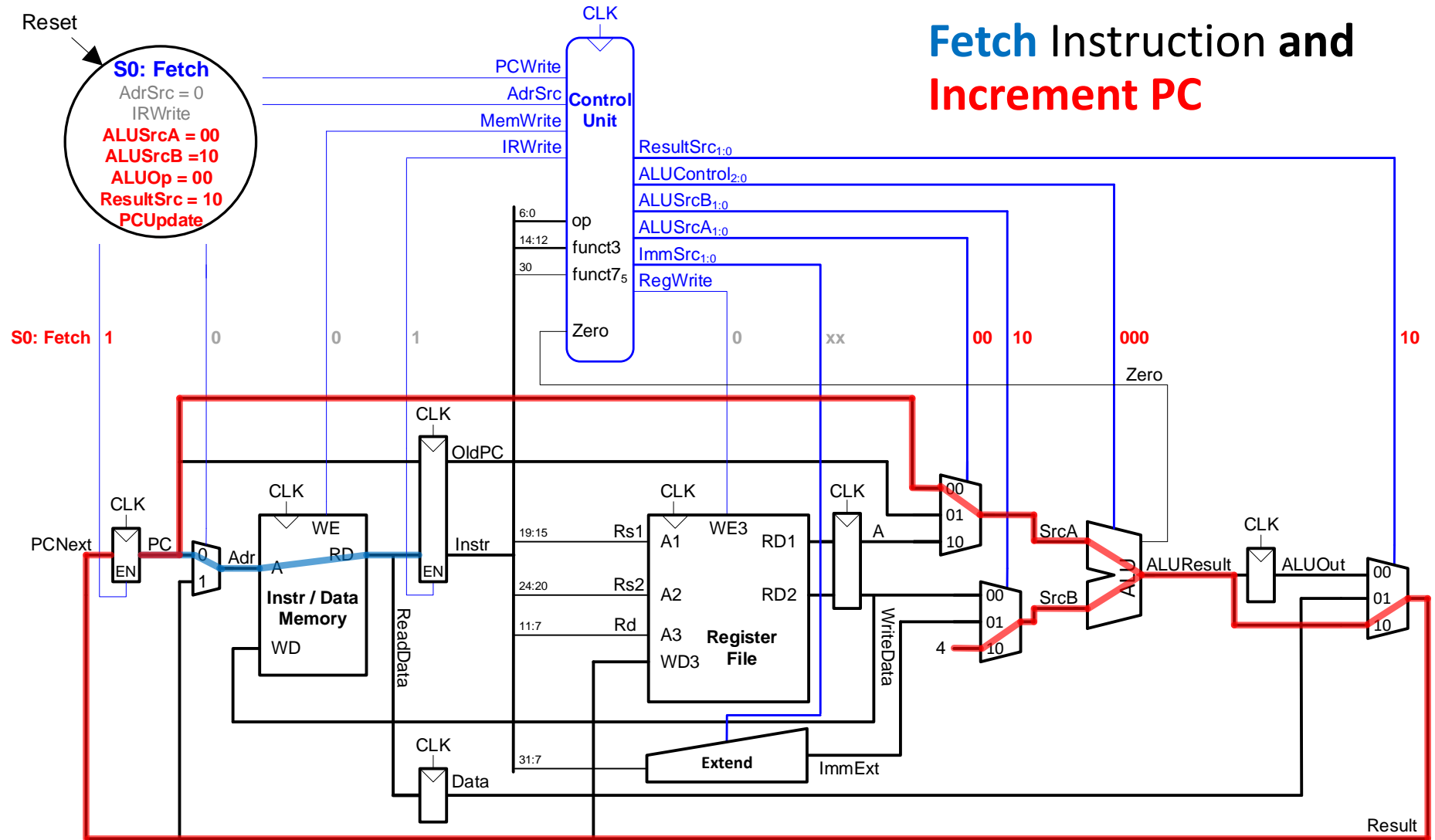








# Main FSM: Fetch (PC+4) Datapath















## Chapter 7: Microarchitecture

# **Multicycle Control: Other Instructions**





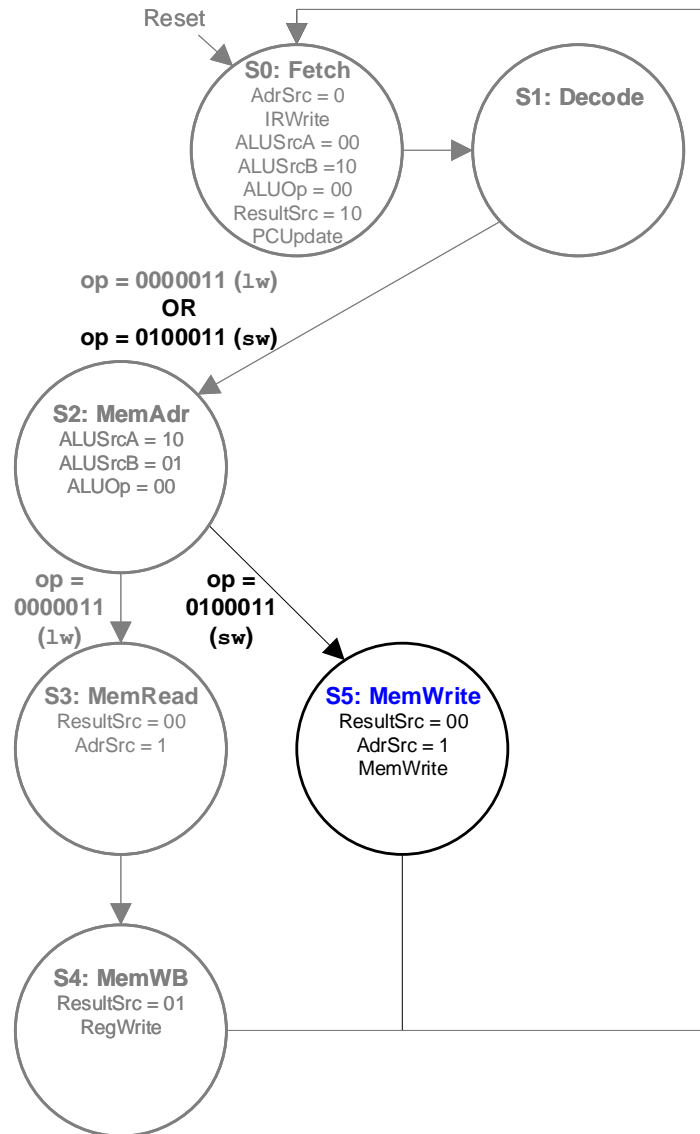








# Main FSM: SW





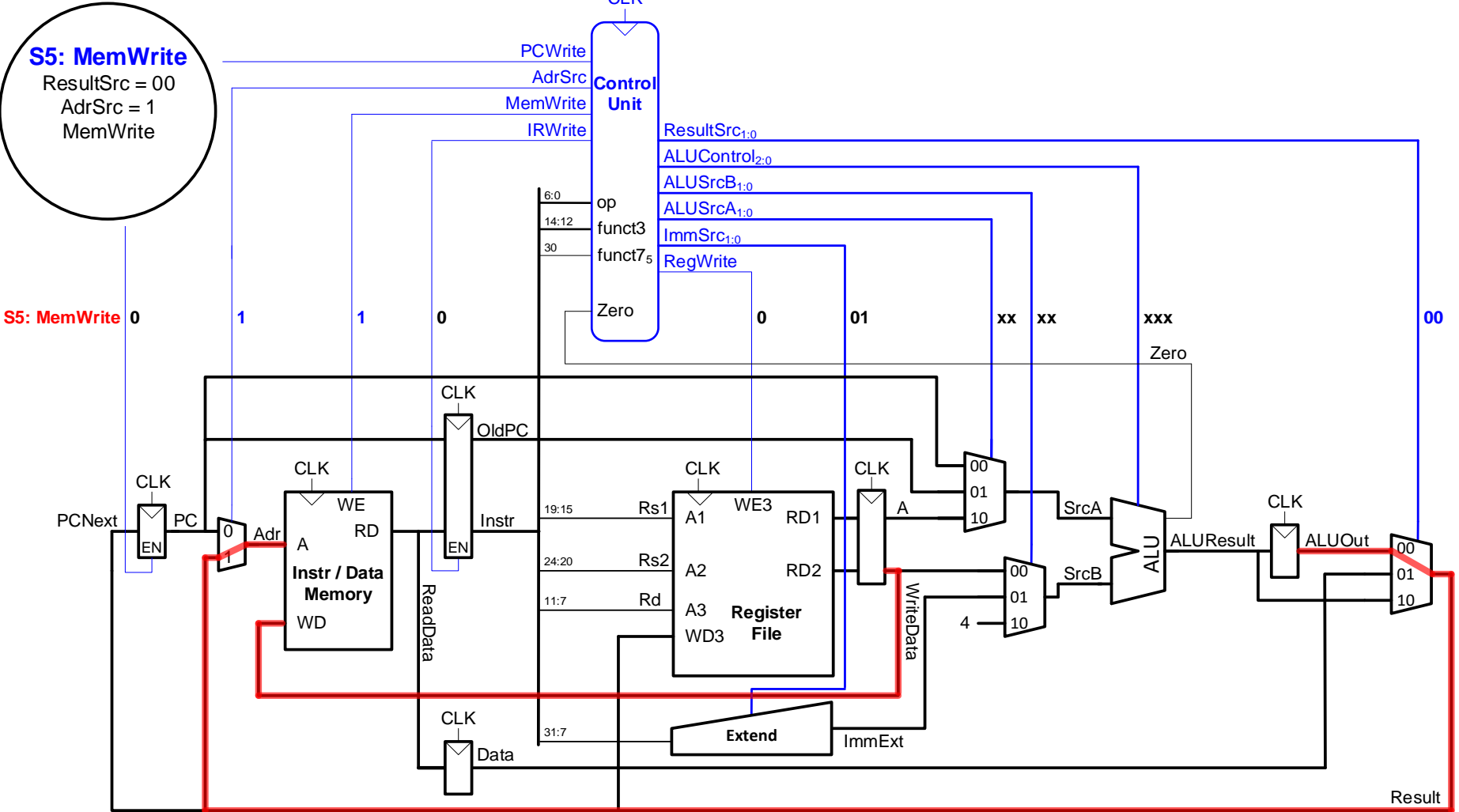








# Main FSM: $s_w$ Datapath







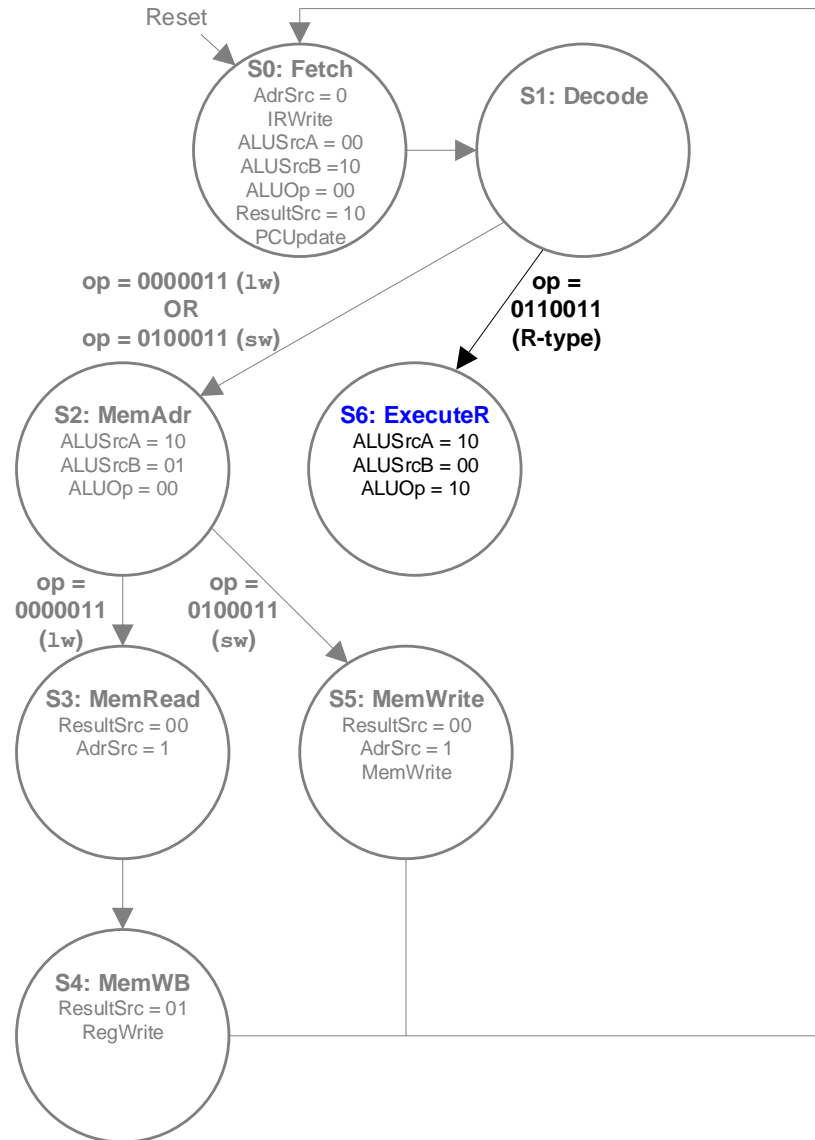








# Main FSM: R-Type Execute











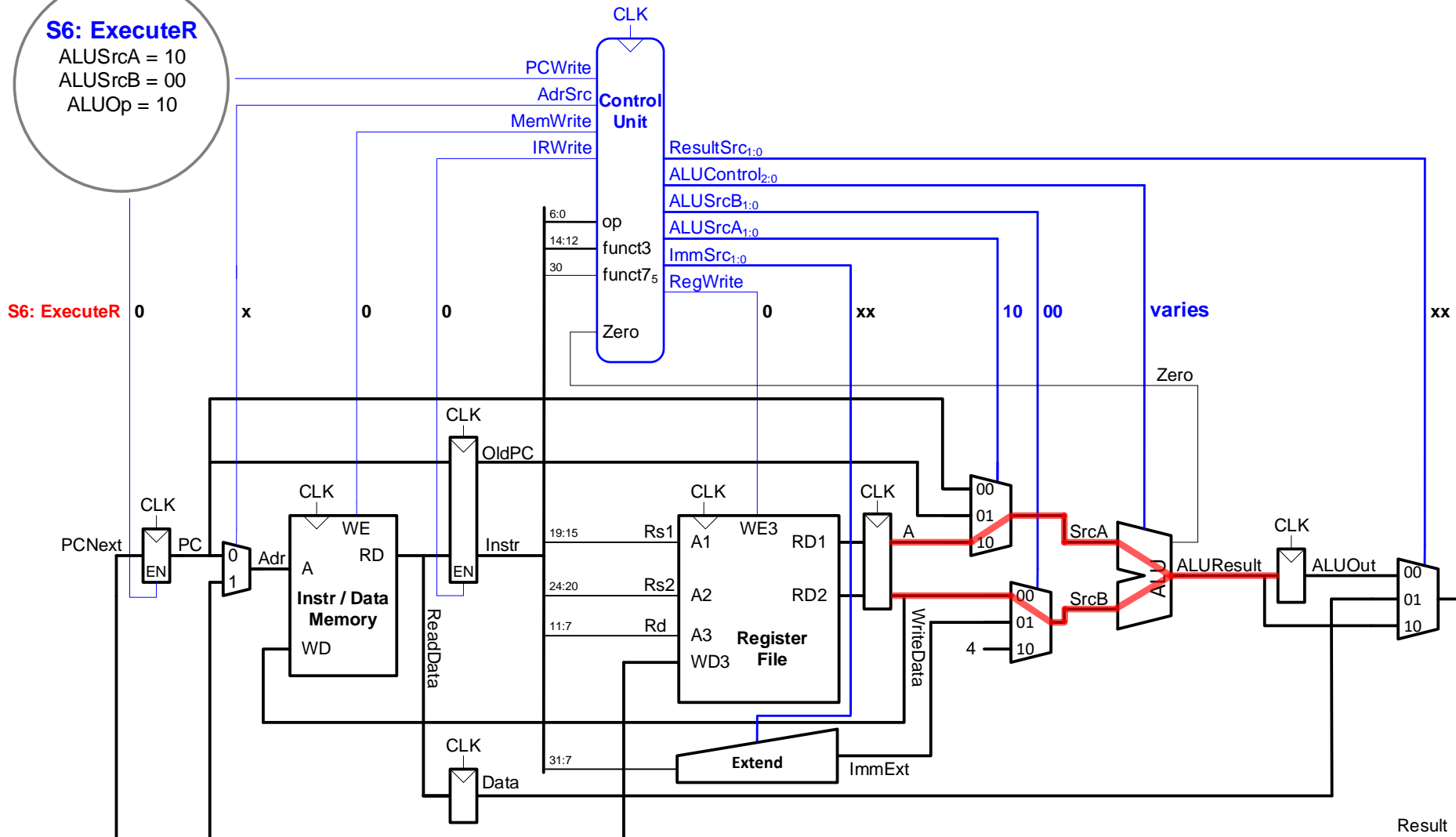




# Main FSM: R-Type Execute Datapath

## S6: ExecuteR

ALUSrcA = 10  
ALUSrcB = 00  
ALUOp = 10







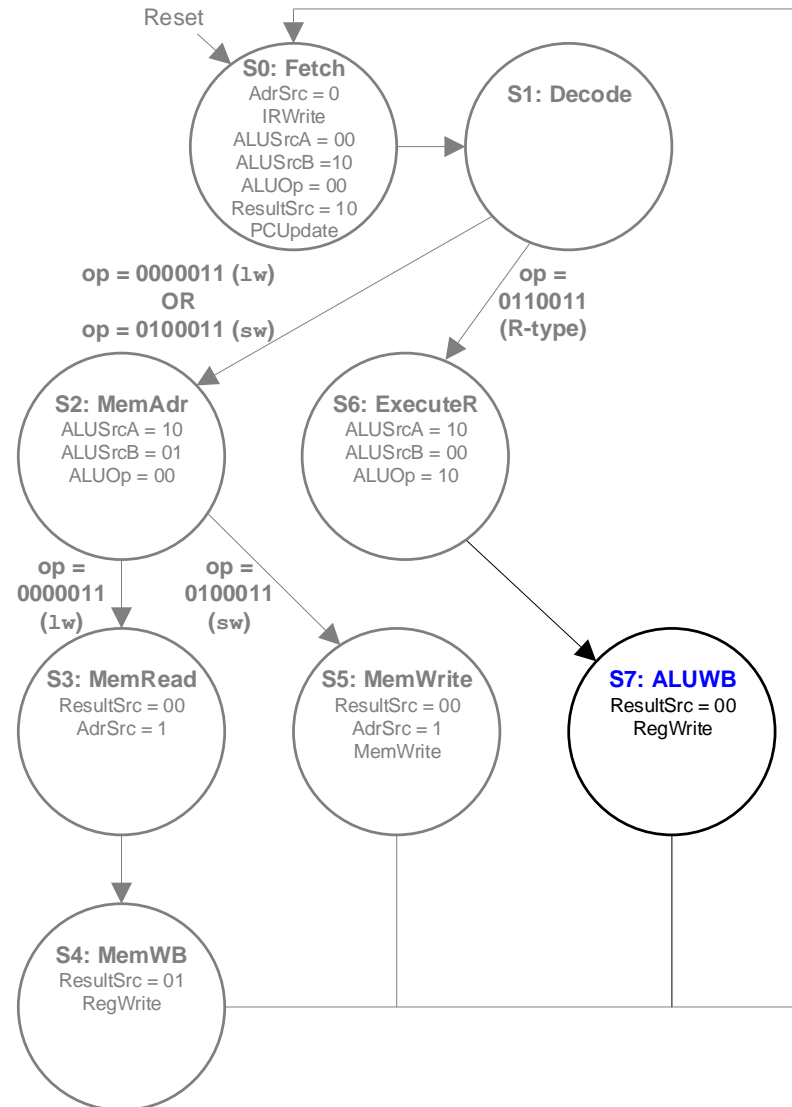








# Main FSM: R-Type ALU Write Back





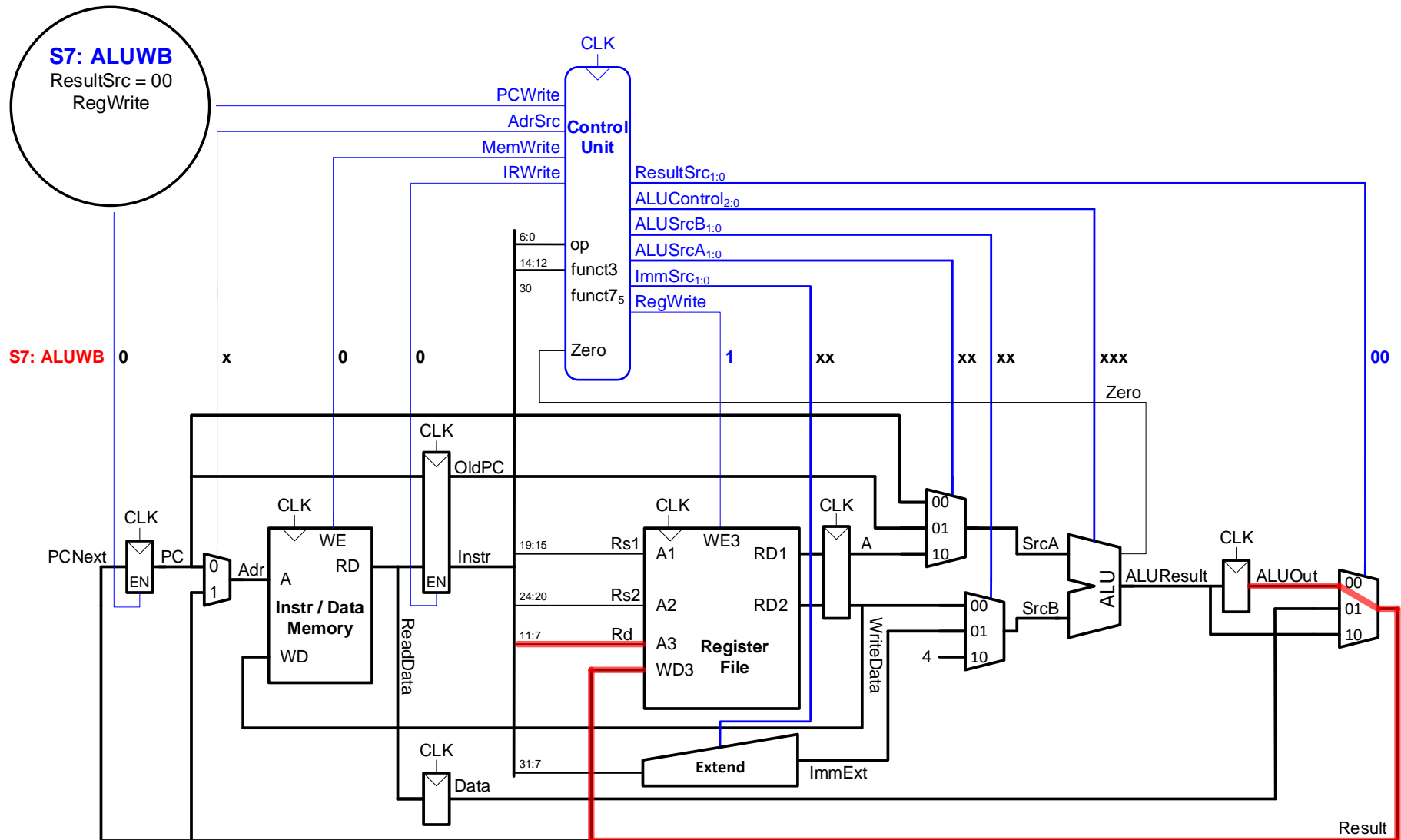








# Main FSM: R-Type ALU Write Back















# Main FSM: beq

- **Need to calculate:**
  - Branch Target Address
  - **rs1** - **rs2** (to see if equal)
- **ALU** isn't being used in Decode stage
  - Use it to calculate Target Address ( $PC + imm$ )





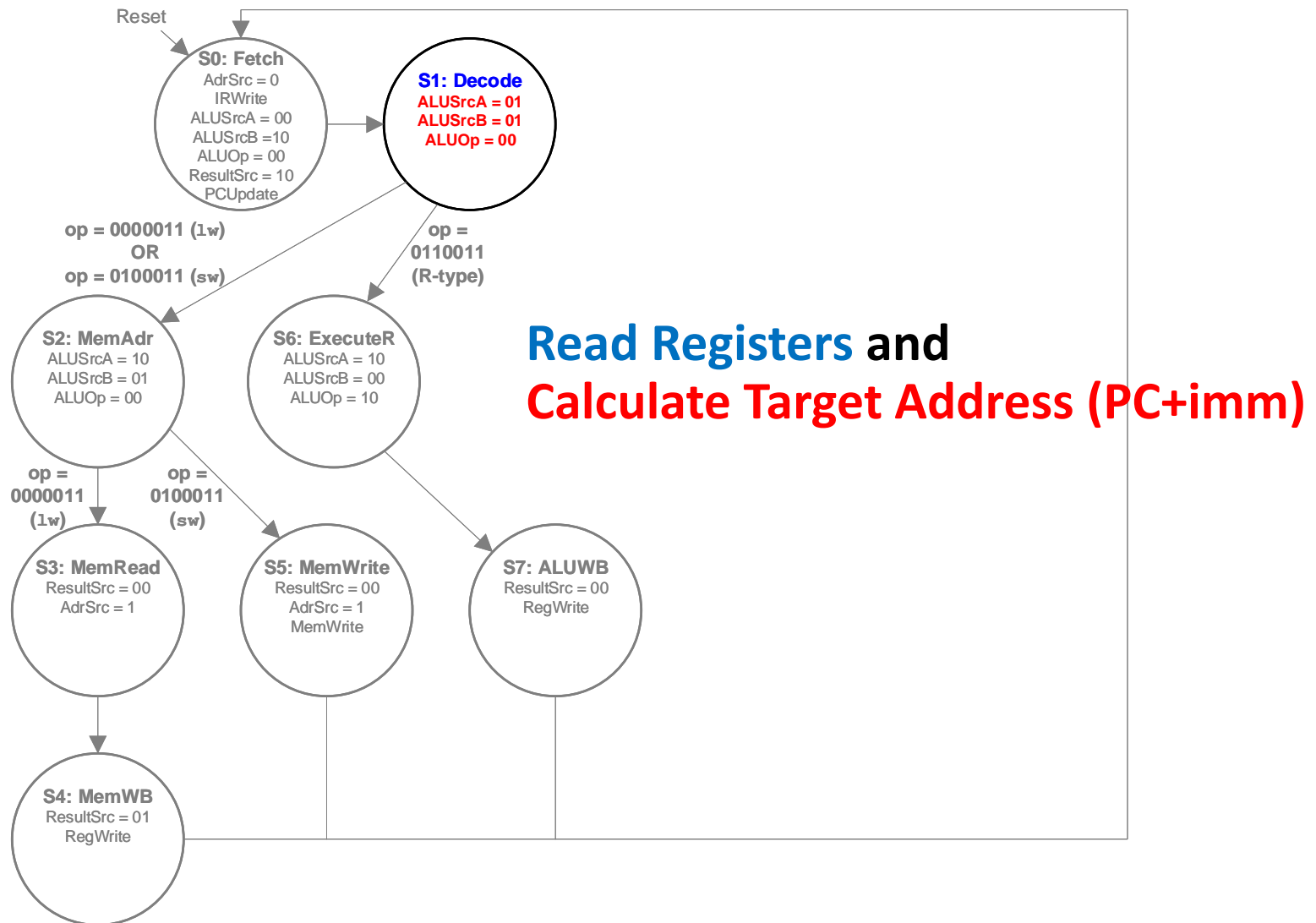








# Main FSM: Decode Revisited

















## Main FSM: Decode (Target Address)

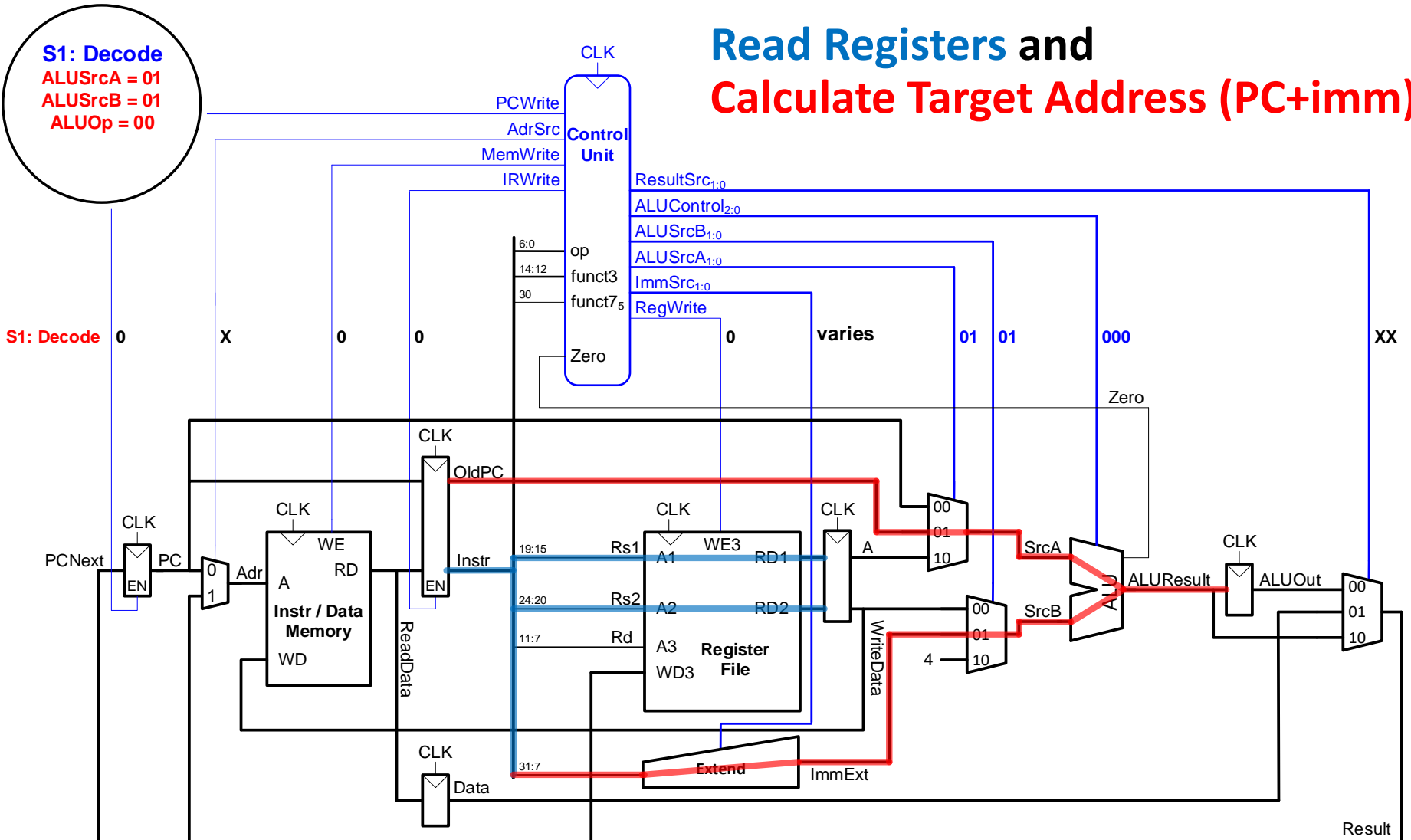


**S1: Decode**  
 ALUSrcA = 01  
 ALUSrcB = 01  
 ALUOp = 00



**S1: Decode**  
 ALUSrcA = 01  
 ALUSrcB = 01  
 ALUOp = 00

## Read Registers and Calculate Target Address (PC+imm)





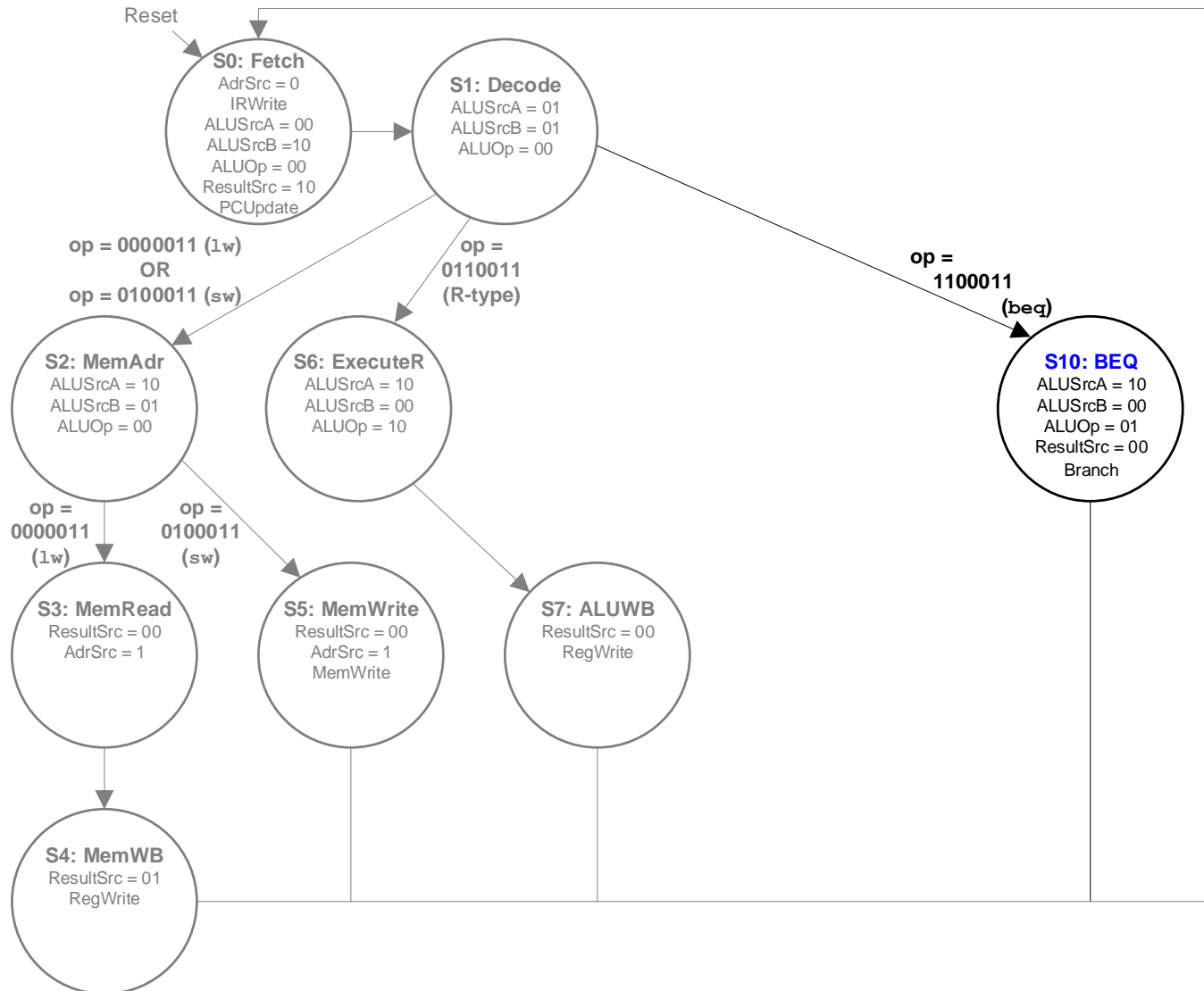








# Main FSM: beq









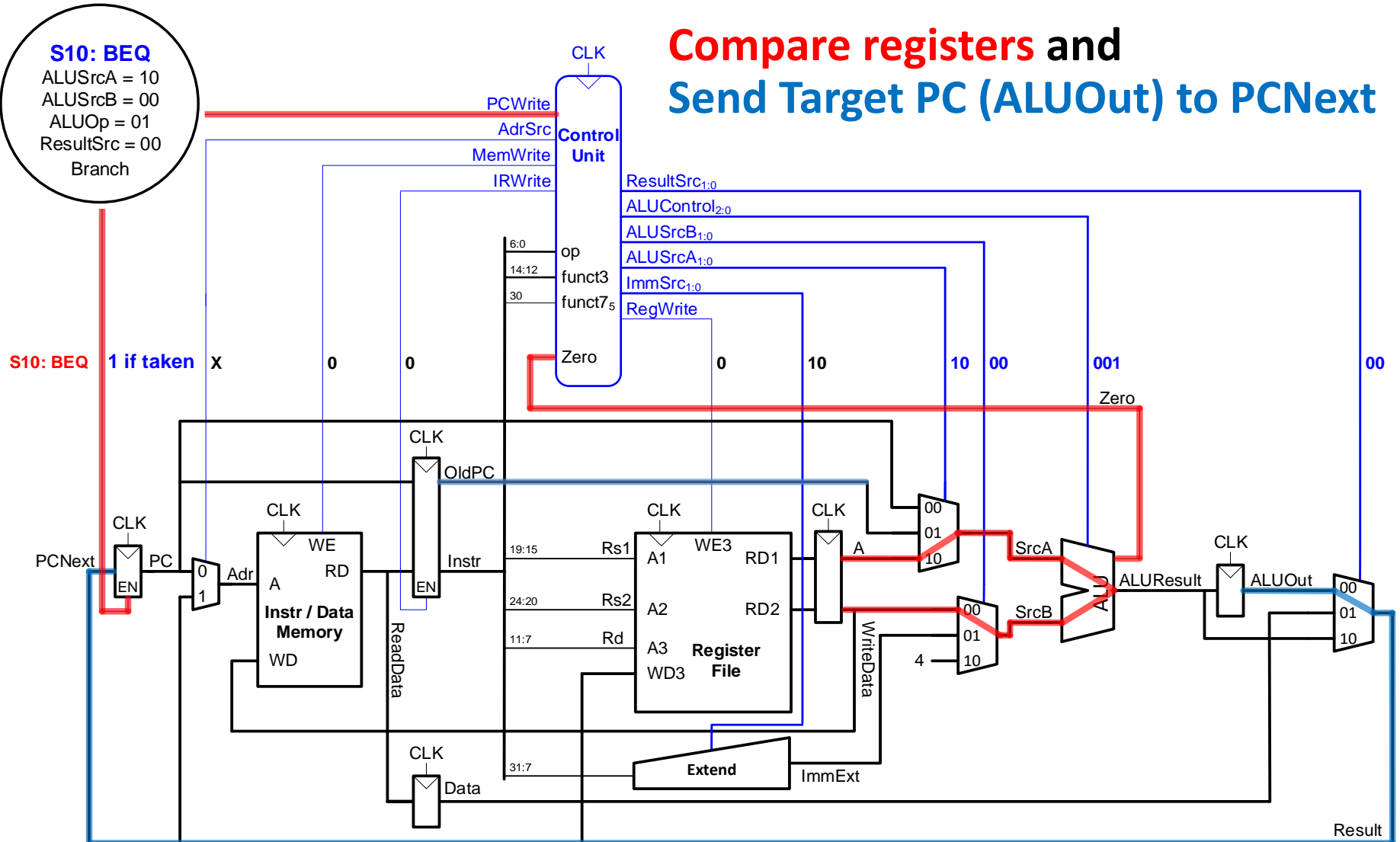






# Main FSM: beq Datapath

**Compare registers and  
Send Target PC (ALUOut) to PCNext**















## Chapter 7: Microarchitecture

# **Extending the RISC-V Multicycle Processor**





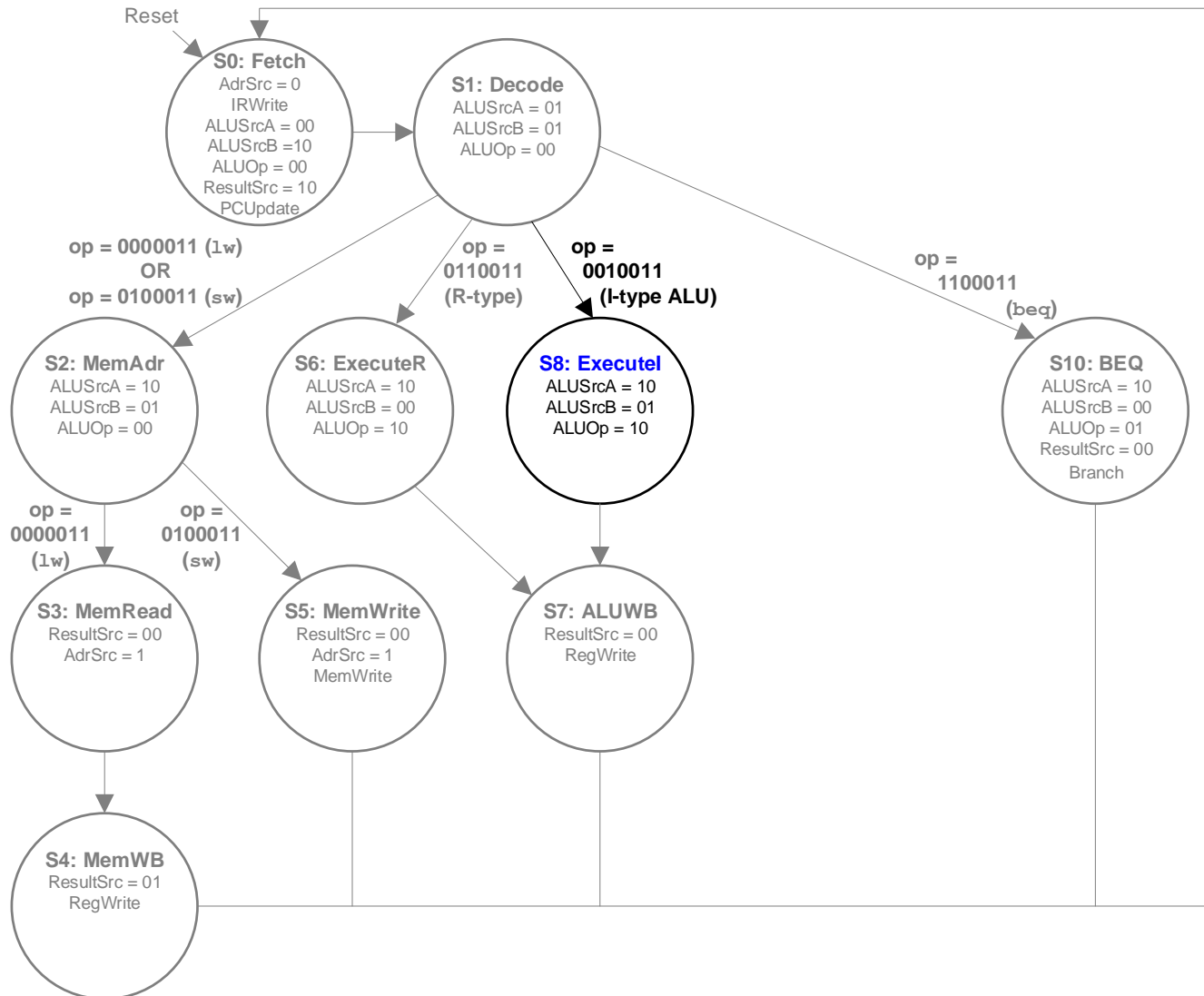








# Main FSM: I-Type ALU Execute











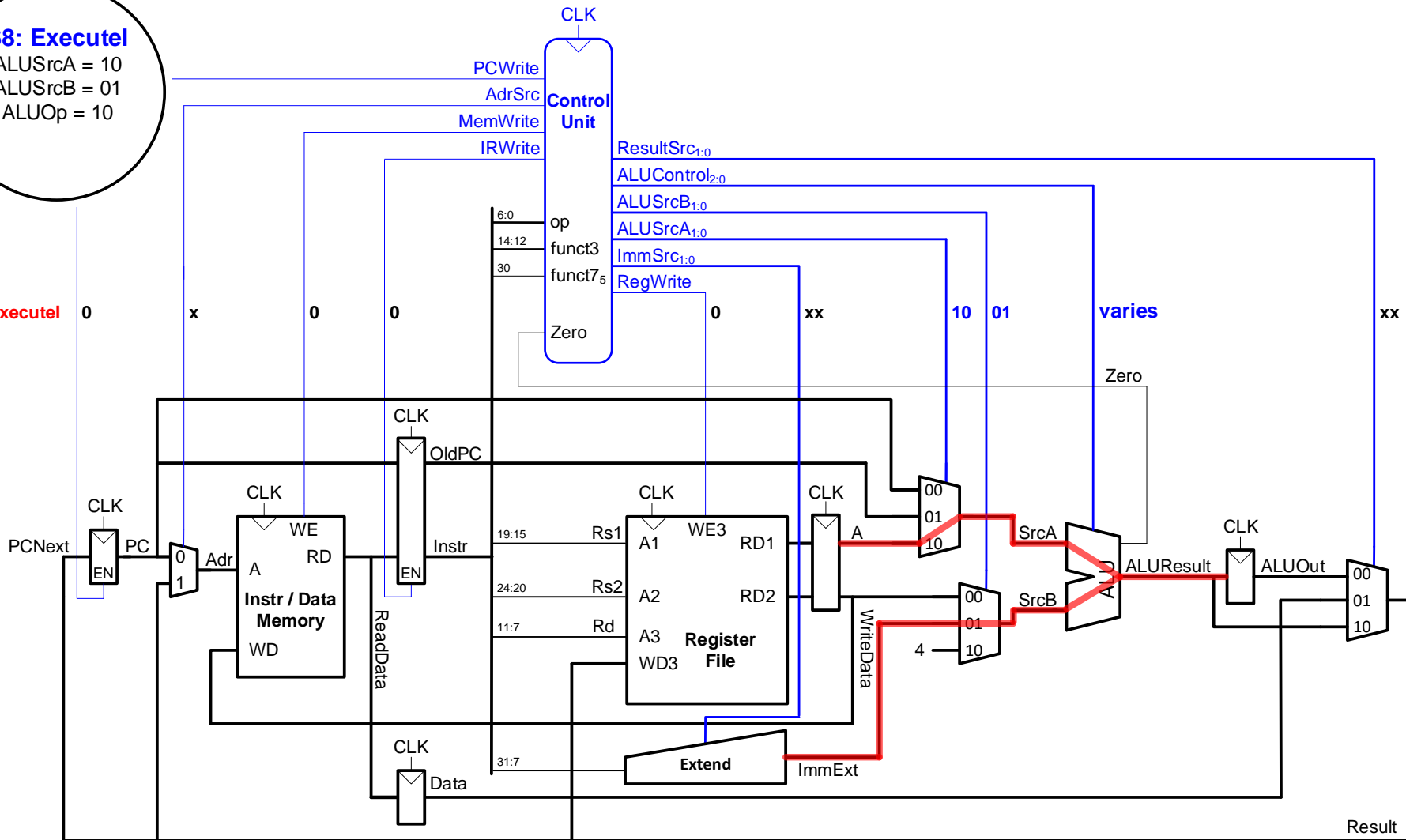


# Main FSM: I-Type ALU Exec. Datapath

## S8: Executel

ALUSrcA = 10  
ALUSrcB = 01  
ALUOp = 10

S8: Executel







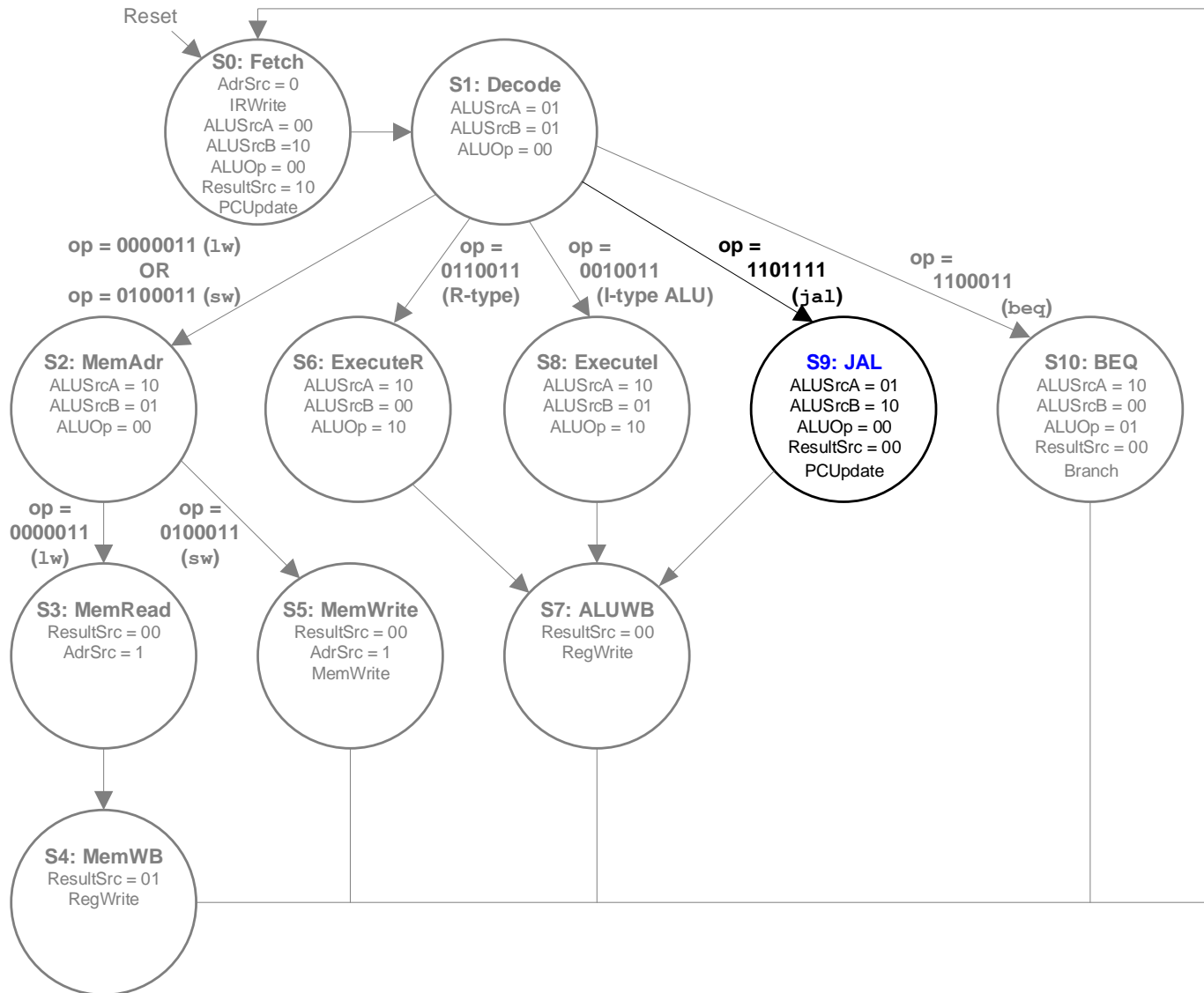








# Main FSM: jal













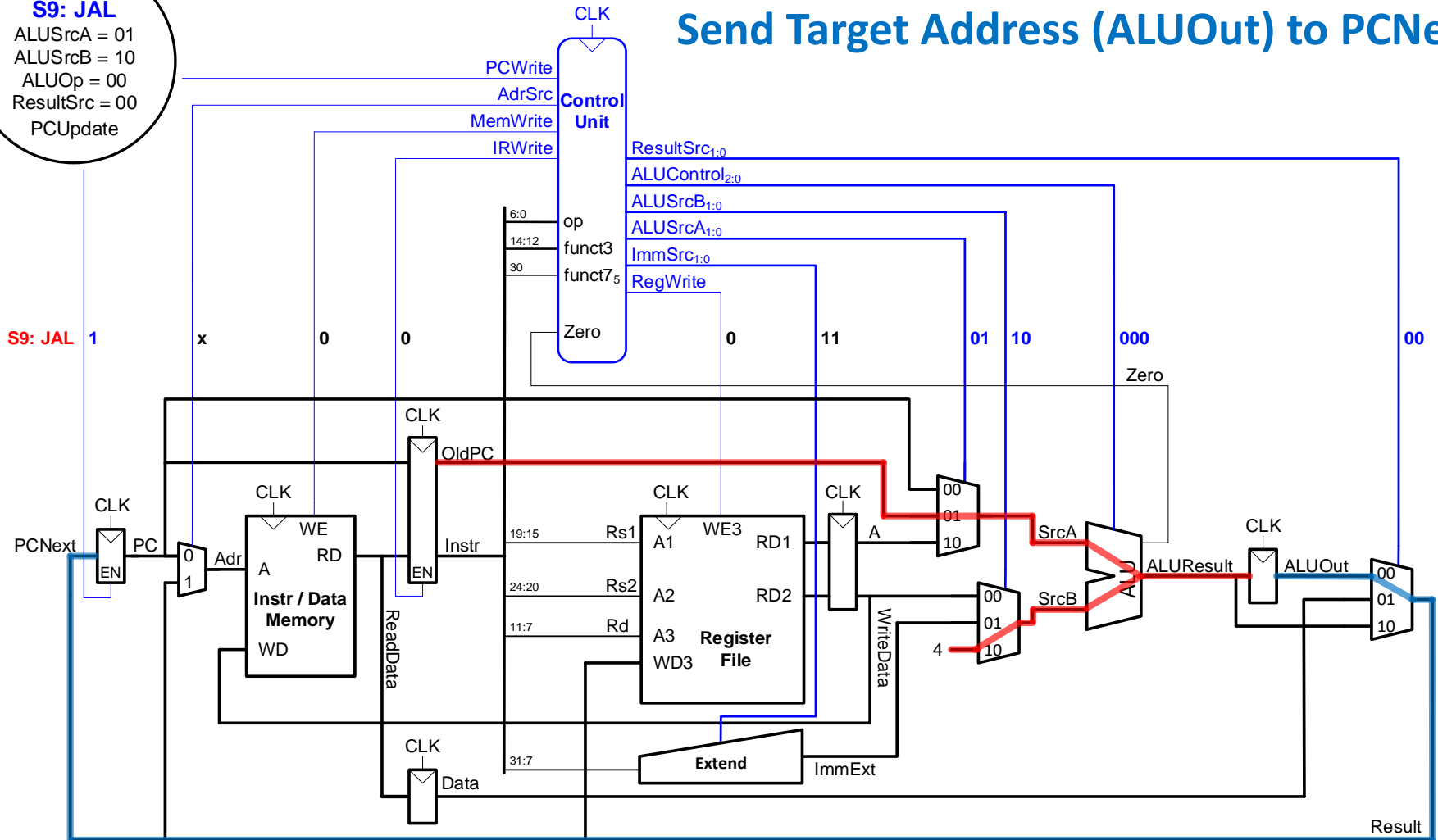


# Main FSM: jal Datapath

**Calculate PC + 4 and**  
**Send Target Address (ALUOut) to PCNext**

**S9: JAL**

ALUSrcA = 01  
ALUSrcB = 10  
ALUOp = 00  
ResultSrc = 00  
PCUpdate









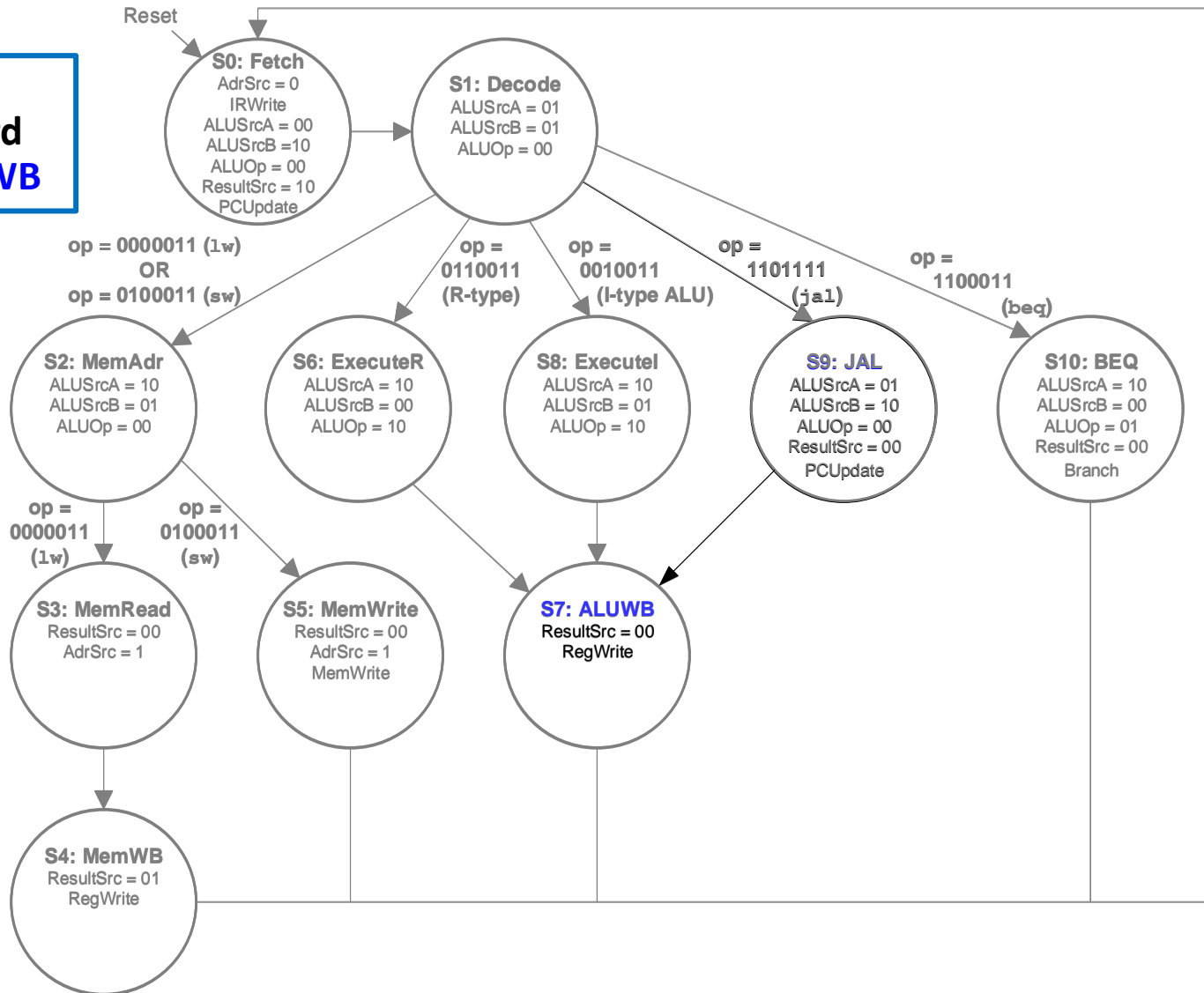






# Main FSM: jal

**PC + 4** is  
written to **rd**  
in **S7: ALUWB**







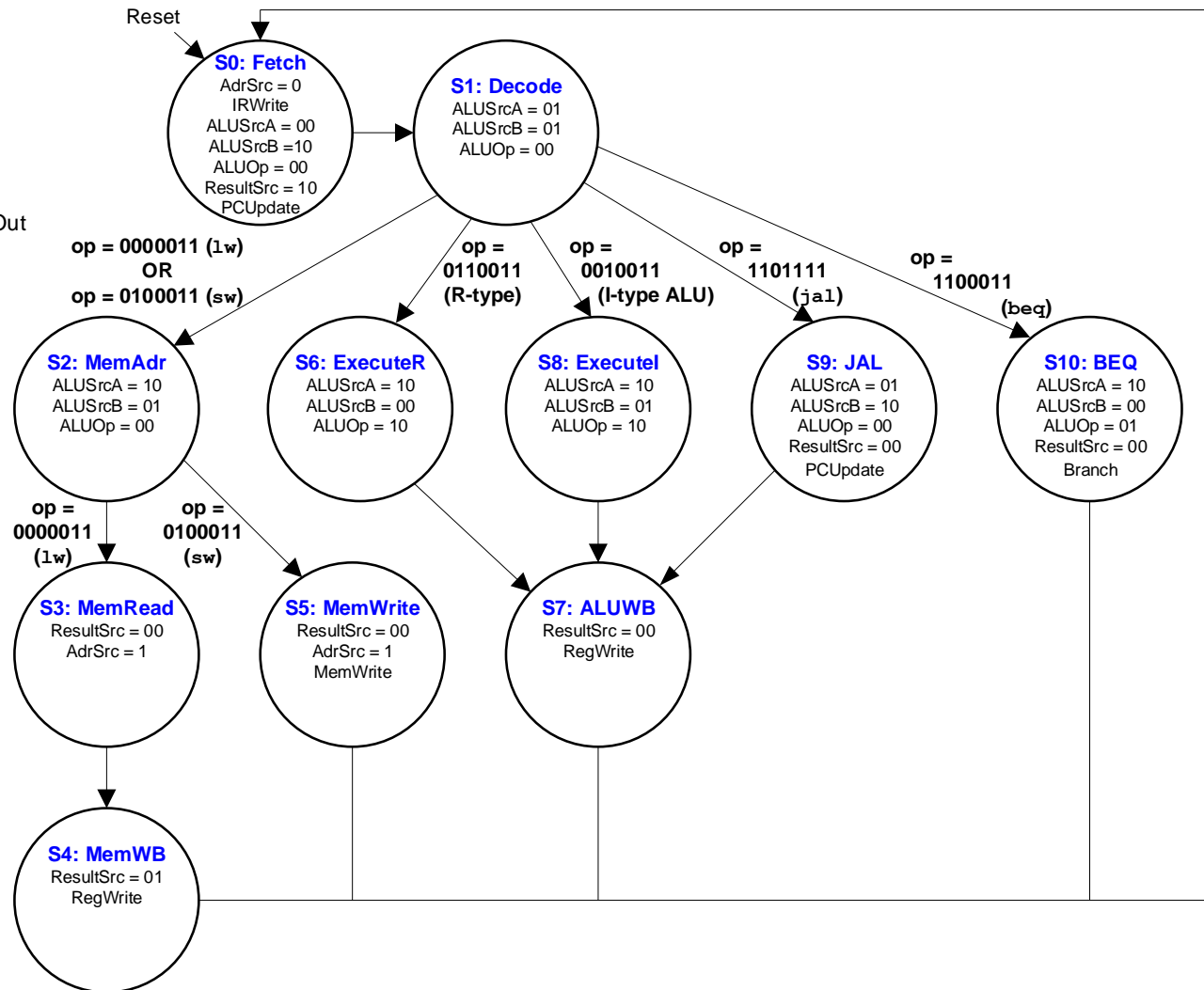






# Multicycle Processor Main FSM

State	Datapath $\mu$ Op
Fetch	Instr $\leftarrow$ Mem[PC]; PC $\leftarrow$ PC+4
Decode	ALUOut $\leftarrow$ PCTarget
MemAdr	ALUOut $\leftarrow$ rs1 + imm
MemRead	Data $\leftarrow$ Mem[ALUOut]
MemWB	rd $\leftarrow$ Data
MemWrite	Mem[ALUOut] $\leftarrow$ rd
ExecuteR	ALUOut $\leftarrow$ rs1 op rs2
ExecuteI	ALUOut $\leftarrow$ rs1 op imm
ALUWB	rd $\leftarrow$ ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC $\leftarrow$ ALUOut
JAL	PC $\leftarrow$ ALUOut; ALUOut $\leftarrow$ PC+4















# Chapter 7: Microarchitecture

## **Multicycle Performance**













# Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles: `beq`
  - 4 cycles: `R-type`, `addi`, `sw`, `jal`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type

$$\text{Average CPI} = (0.13)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12$$









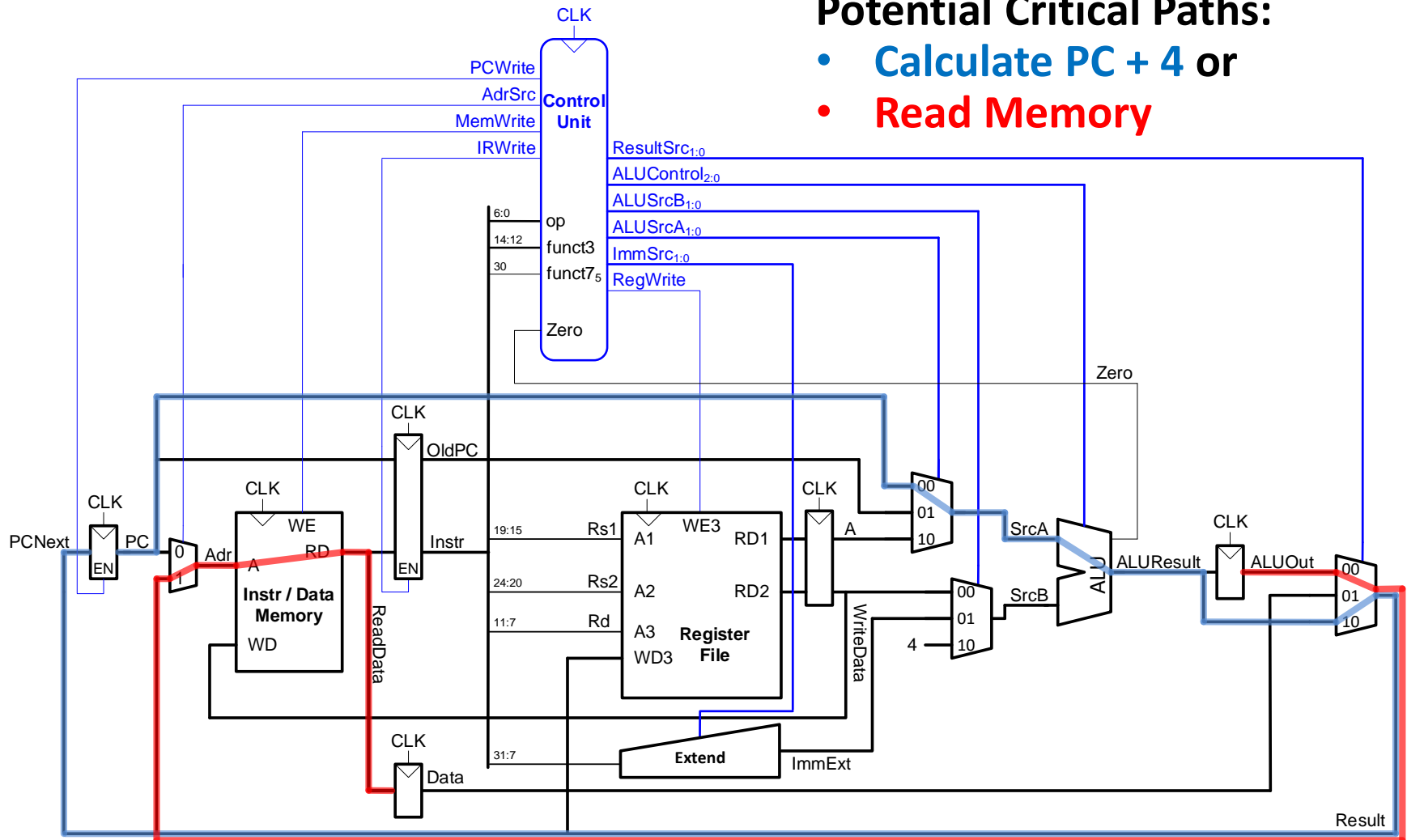




# Multicycle Critical Path

## Potential Critical Paths:

- Calculate PC + 4 or
- Read Memory













# Multicycle Processor Performance

Multicycle critical path:

- **Assumptions:**
  - RF is faster than memory
  - Writing memory is faster than reading memory

$$T_{c\_multi} = t_{pcq} + t_{dec} + 2t_{mux} + \max(t_{ALU}, t_{mem}) + t_{setup}$$













# Multicycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	$t_{pcq\_PC}$	40
Register setup	$t_{setup}$	50
Multiplexer	$t_{mux}$	30
AND-OR gate	$t_{AND-OR}$	20
ALU	$t_{ALU}$	120
Decoder (Control Unit)	$t_{dec}$	25
Extend unit	$t_{dec}$	35
Memory read	$t_{mem}$	200
Register file read	$t_{RFread}$	100
Register file setup	$t_{RFsetup}$	60

$$T_{c\_multi} = t_{pcq} + t_{dec} + 2t_{mux} + \max(t_{ALU}, t_{mem}) + t_{setup}$$

$$=$$













# Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** RISC-V processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:**  $T_{c\_multi} = 375 \text{ ps}$

**Execution Time** = (# instructions)  $\times$  CPI  $\times T_c$













## Chapter 7: Microarchitecture

# **Pipelined RISC-V Processor**











# Pipelined RISC-V Processor

- **Temporal parallelism**
- Divide single-cycle processor into **5 stages**:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add **pipeline registers** between stages







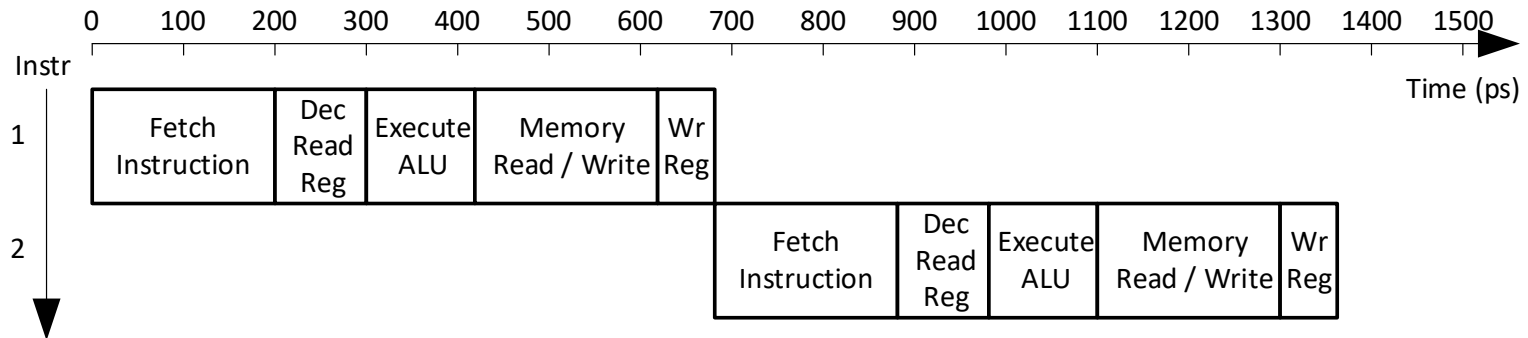




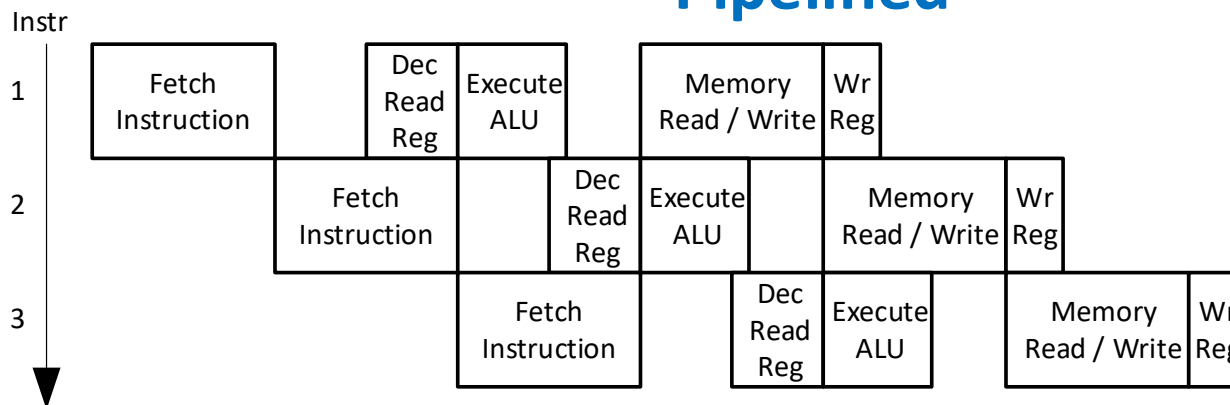


# Single-Cycle vs. Pipelined Processor

## Single-Cycle



## Pipelined







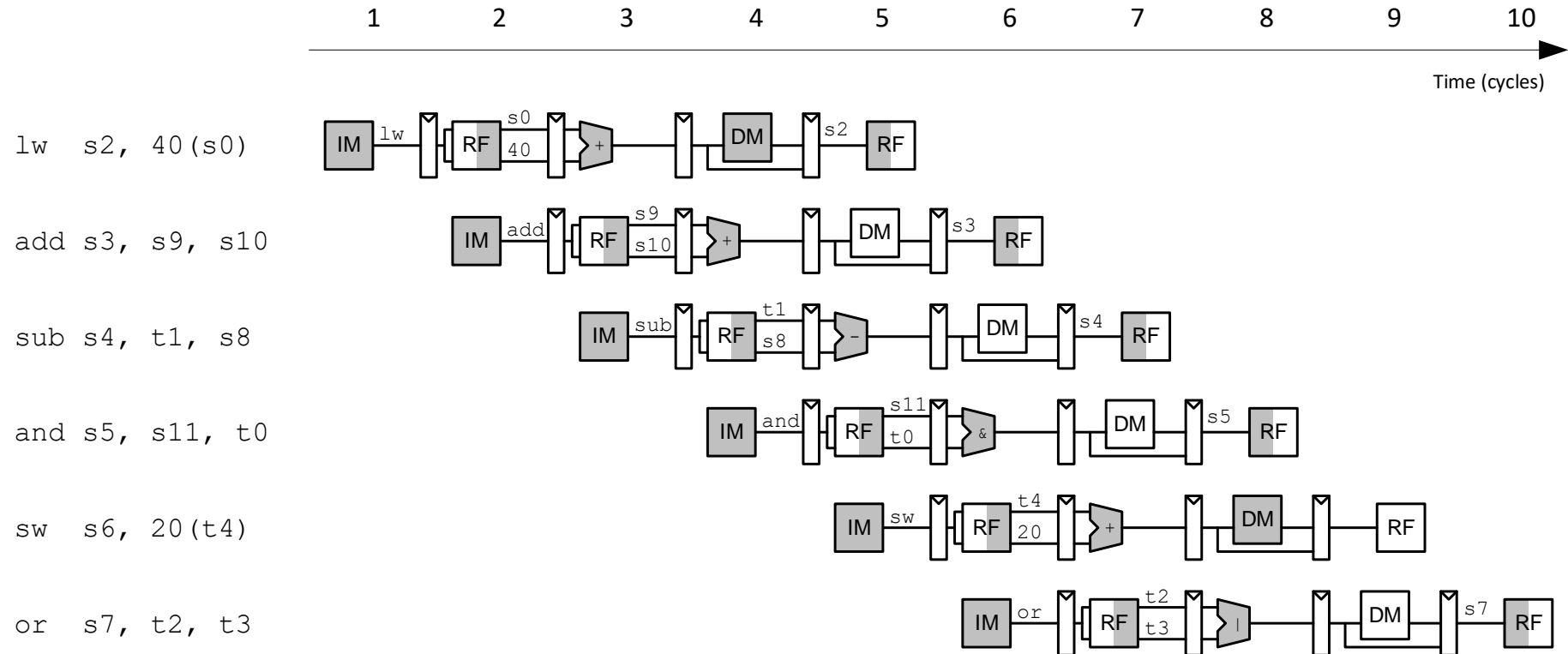








# Pipelined Processor Abstraction







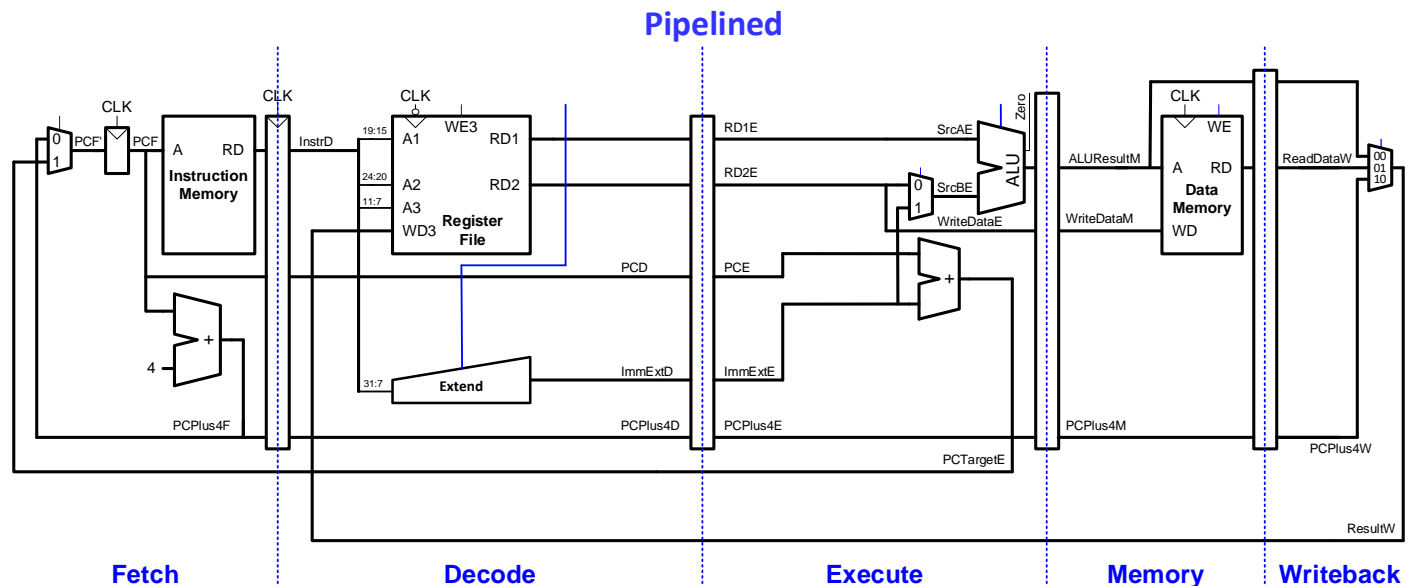
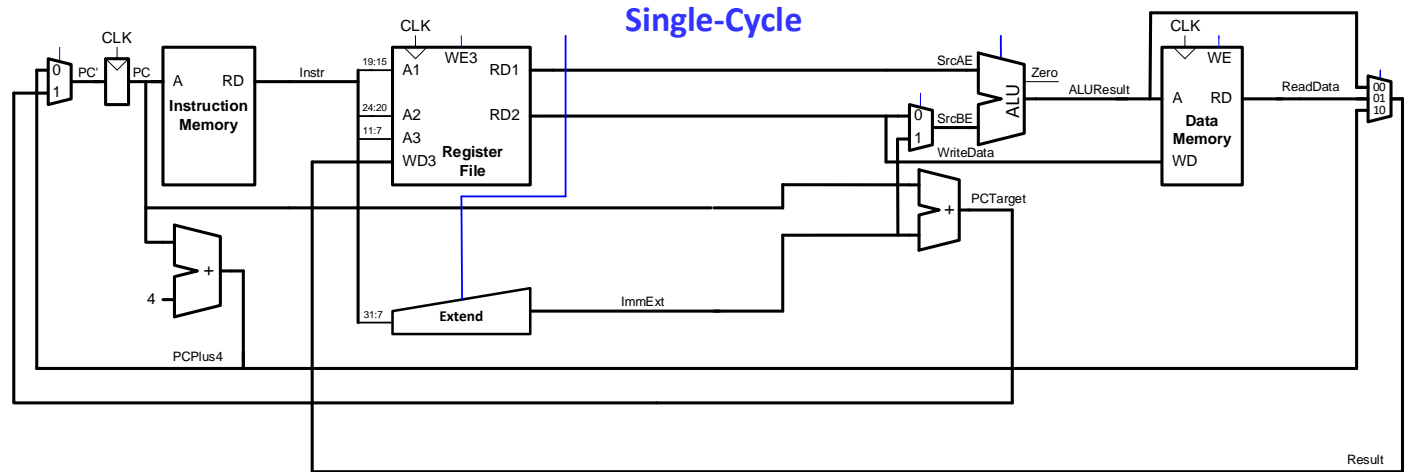








# Single-Cycle & Pipelined Datapaths



Signals in Pipelined Processor are appended with first letter of stage (i.e., PC**F**, PC**D**, PC**E**).



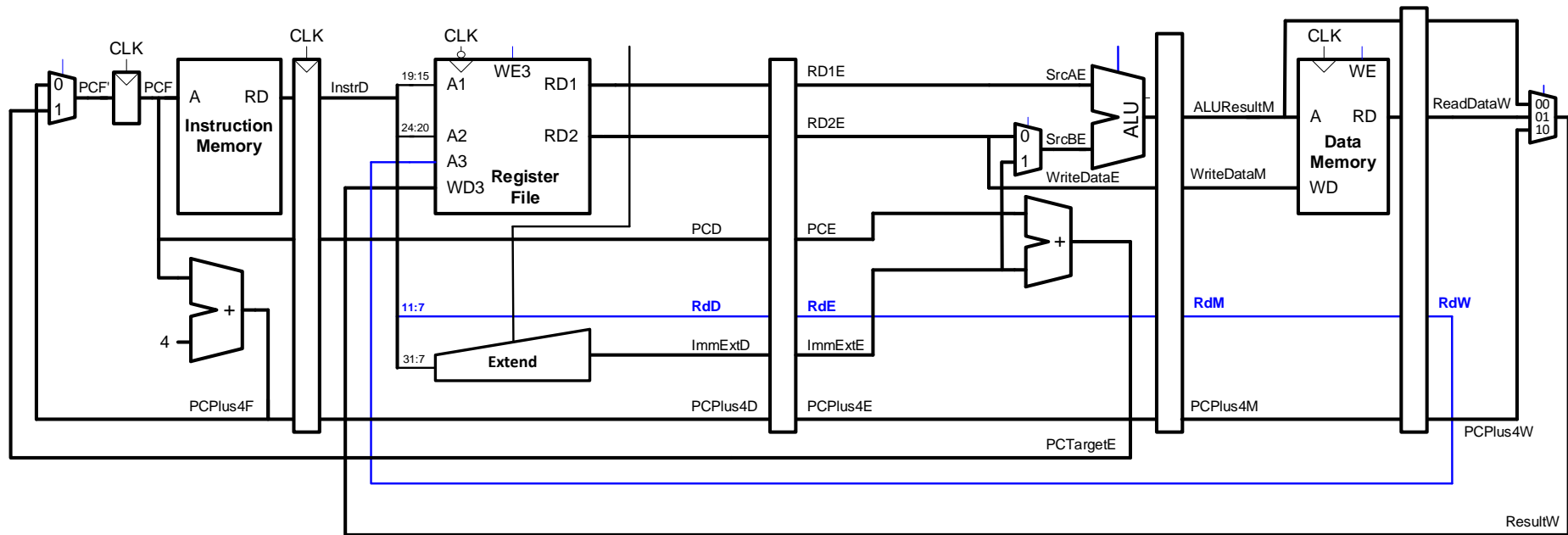








# Corrected Pipelined Datapath



- ***Rd*** must arrive at same time as ***Result***
- Register file written on **falling edge** of *CLK*





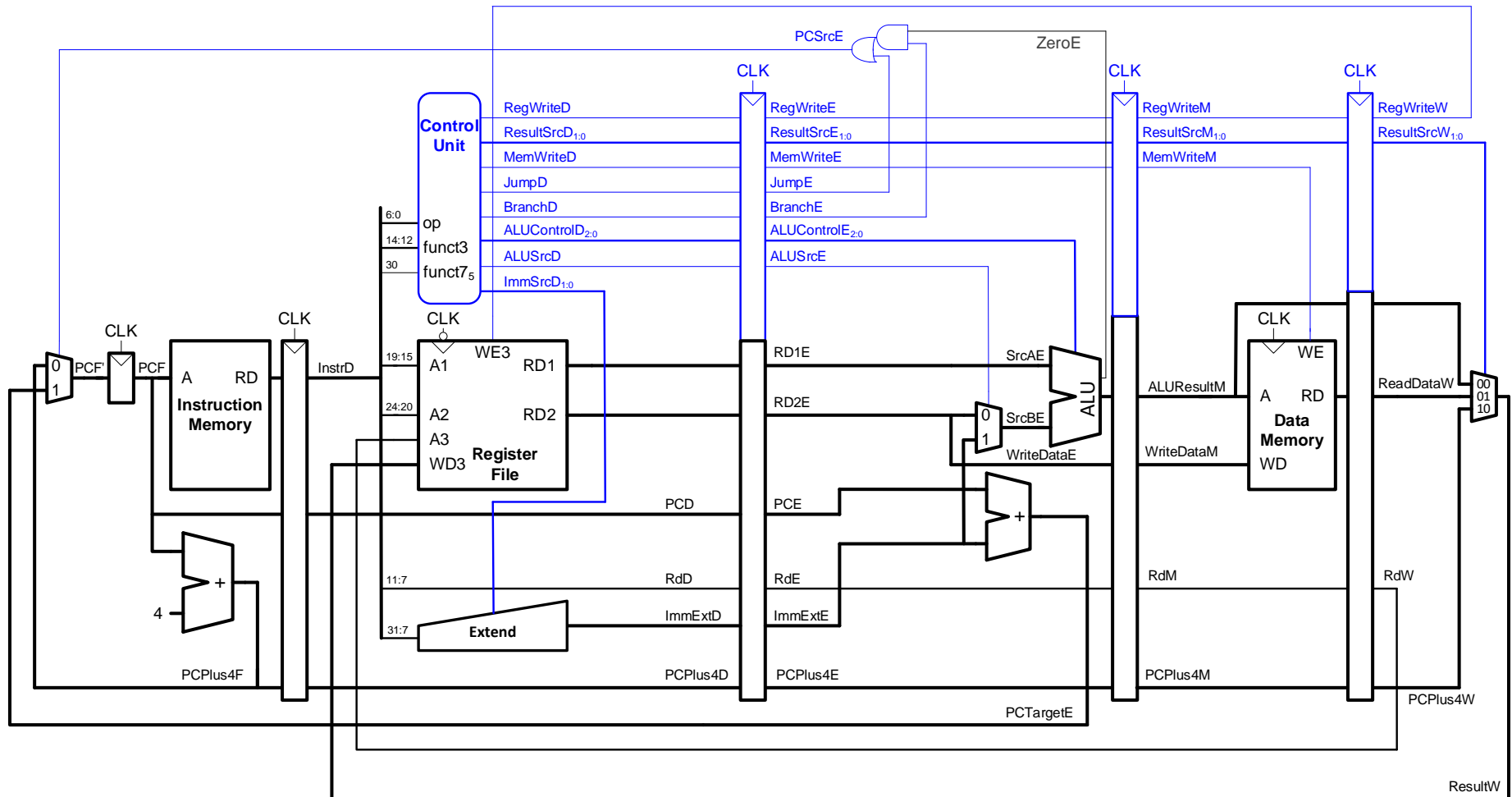








# Pipelined Processor with Control



- **Same control unit** as single-cycle processor
- **Control signals travel with** the instruction (drop off when used)













## Chapter 7: Microarchitecture

# **Pipelined Processor Hazards**













# Pipelined Hazards

- When an instruction depends on result from instruction that hasn't completed
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branch)



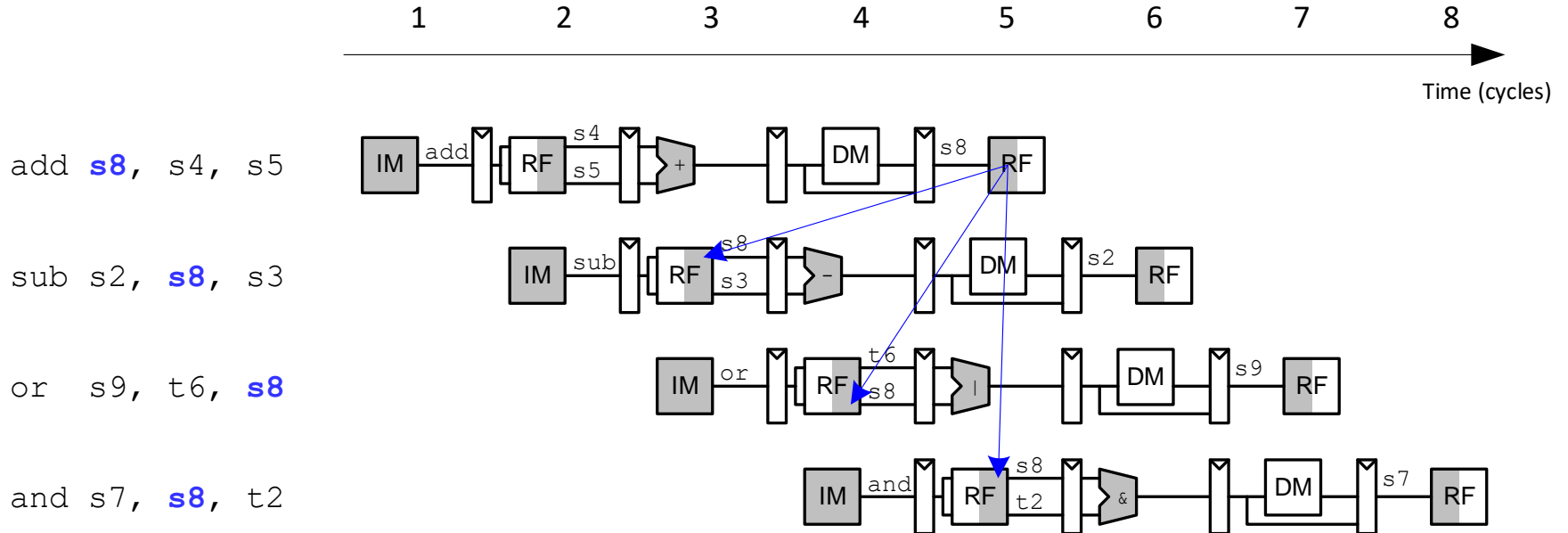








# Data Hazard















# Handling Data Hazards







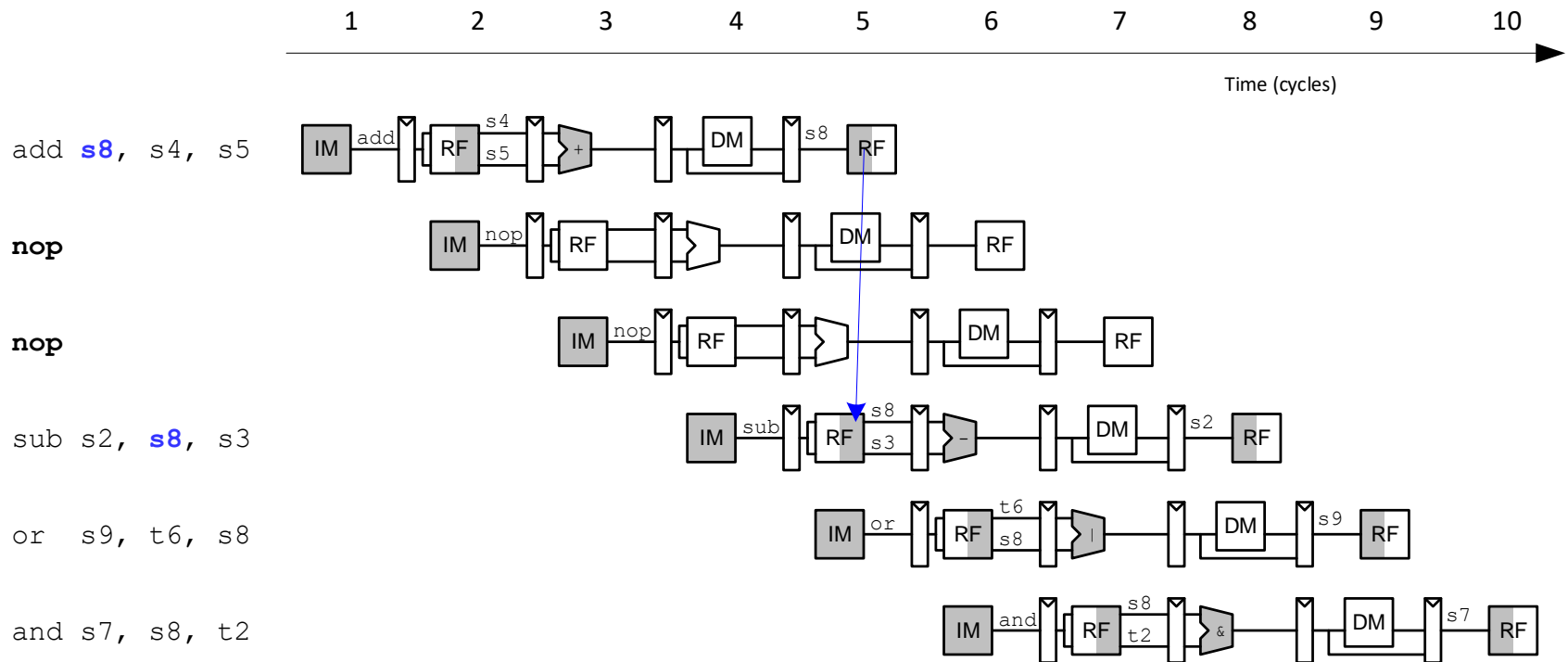






# Handling Data Hazards

- **Insert** enough **nops** for result to be ready
- Or move independent useful instructions forward









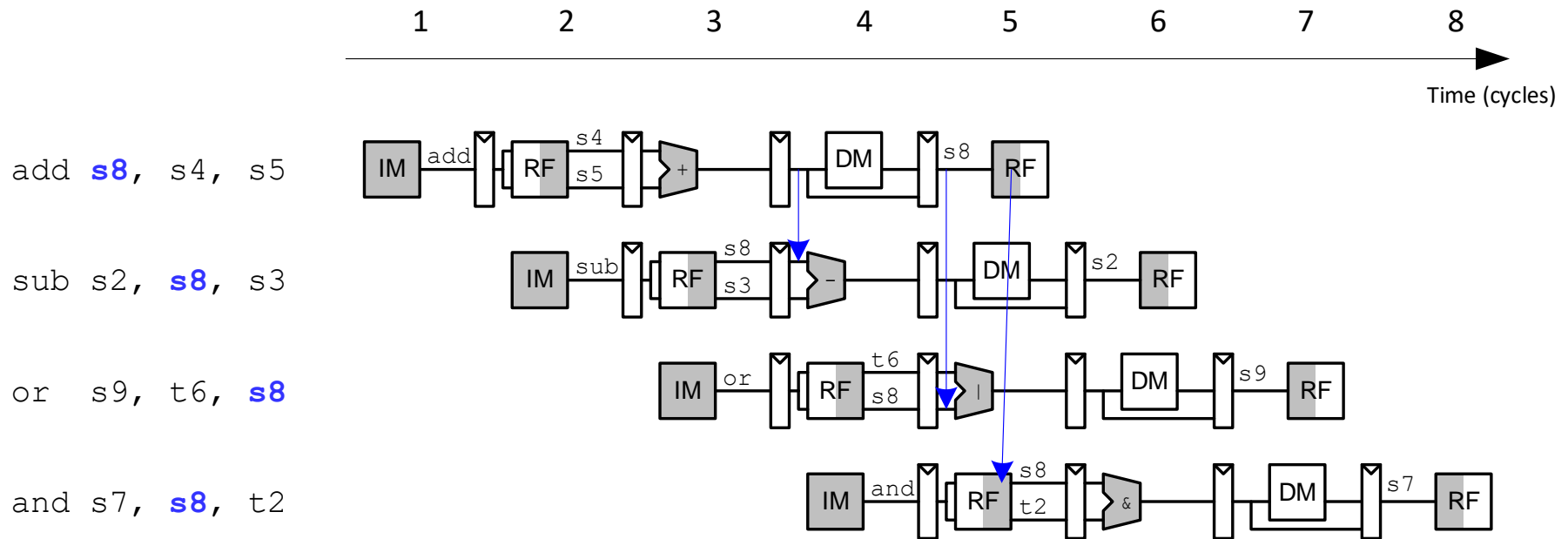






# Data Forwarding

- Data is **available on internal busses** before it is written back to the register file (RF).
- **Forward data** from internal busses **to Execute stage**.







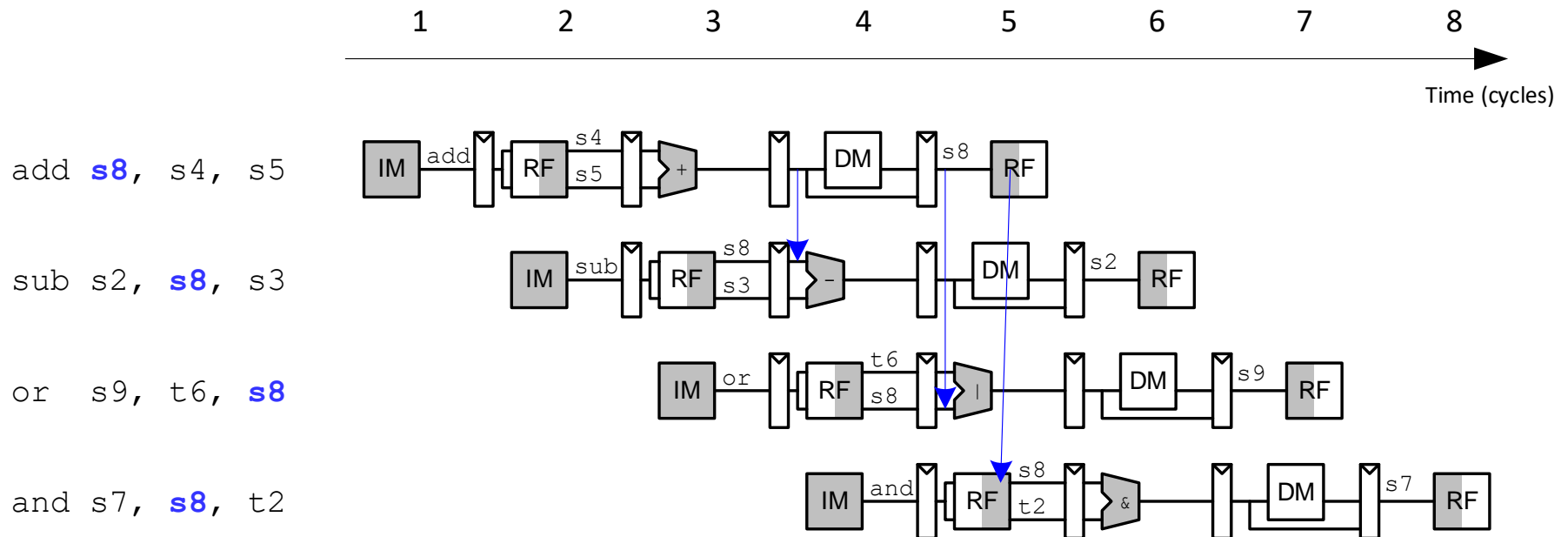






# Data Forwarding

- Check if source register **in Execute stage matches** destination register of instruction **in Memory or Writeback stage**.
- If so, forward result.







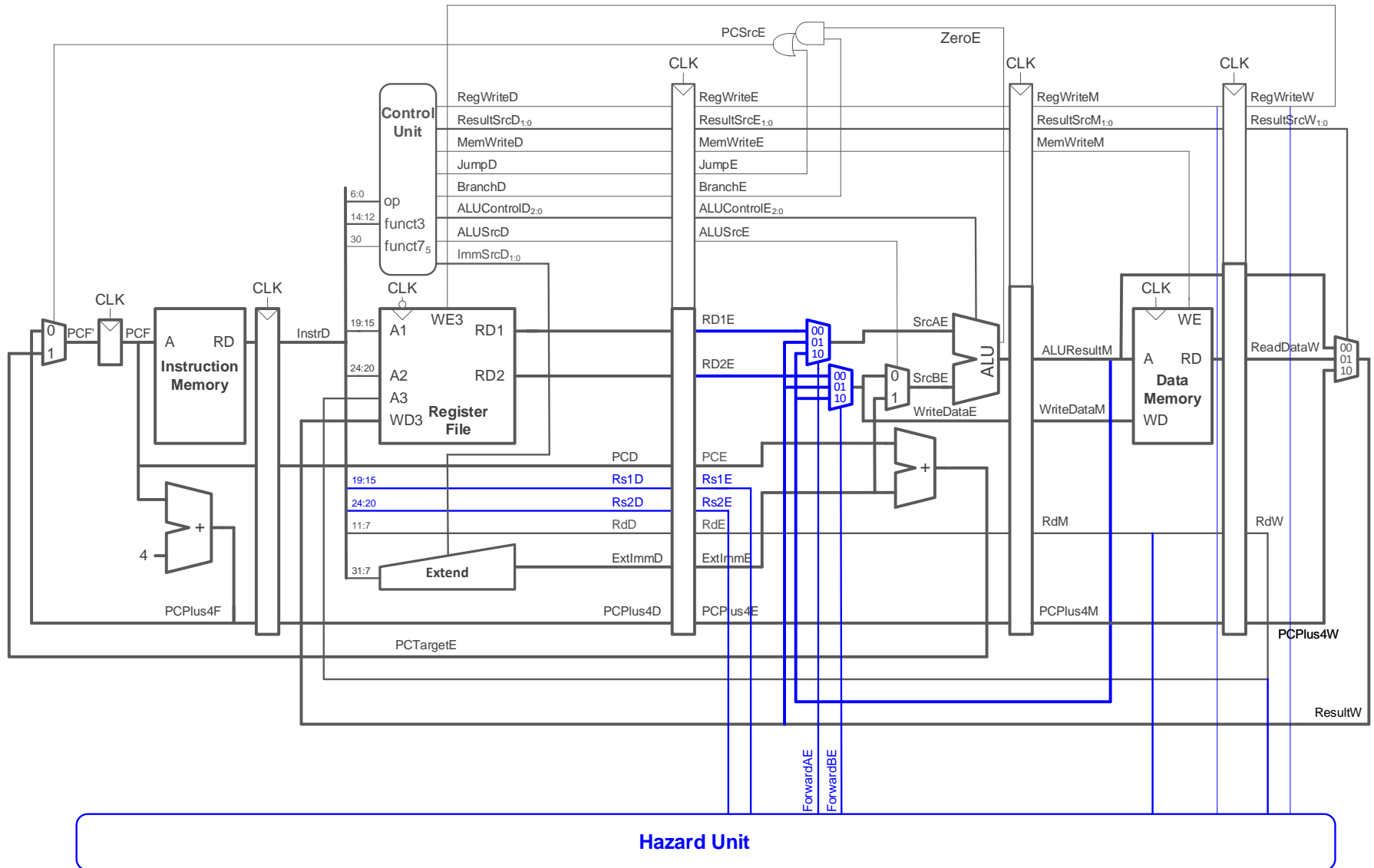








# Data Forwarding: Hazard Unit















# Data Forwarding

- **Case 1:** **Execute** stage  $Rs1$  or  $Rs2$  matches **Memory** stage  $Rd$ ?  
Forward from Memory stage
- **Case 2:** **Execute** stage  $Rs1$  or  $Rs2$  matches **Writeback** stage  $Rd$ ?  
Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for  $Rs1$ :

```
if      (( $Rs1E == RdM$ ) AND  $RegWriteM$ )           // Case 1
         $ForwardAE = 10$ 
else if (( $Rs1E == RdW$ ) AND  $RegWriteW$ )           // Case 2
         $ForwardAE = 01$ 
else     $ForwardAE = 00$                            // Case 3
```

***ForwardBE** equations are similar (replace  $Rs1E$  with  $Rs2E$ )*













# Data Forwarding

- **Case 1:** **Execute** stage  $Rs1$  or  $Rs2$  matches **Memory** stage  $Rd$ ?  
Forward from Memory stage
- **Case 2:** **Execute** stage  $Rs1$  or  $Rs2$  matches **Writeback** stage  $Rd$ ?  
Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for  $Rs1$ :

```
if      (( $Rs1E == RdM$ ) AND  $RegWriteM$ ) AND ( $Rs1E != 0$ ) // Case 1
        ForwardAE = 10
else if (( $Rs1E == RdW$ ) AND  $RegWriteW$ ) AND ( $Rs1E != 0$ ) // Case 2
        ForwardAE = 01
else    ForwardAE = 00                                     // Case 3
```

*ForwardBE equations are similar (replace  $Rs1E$  with  $Rs2E$ )*



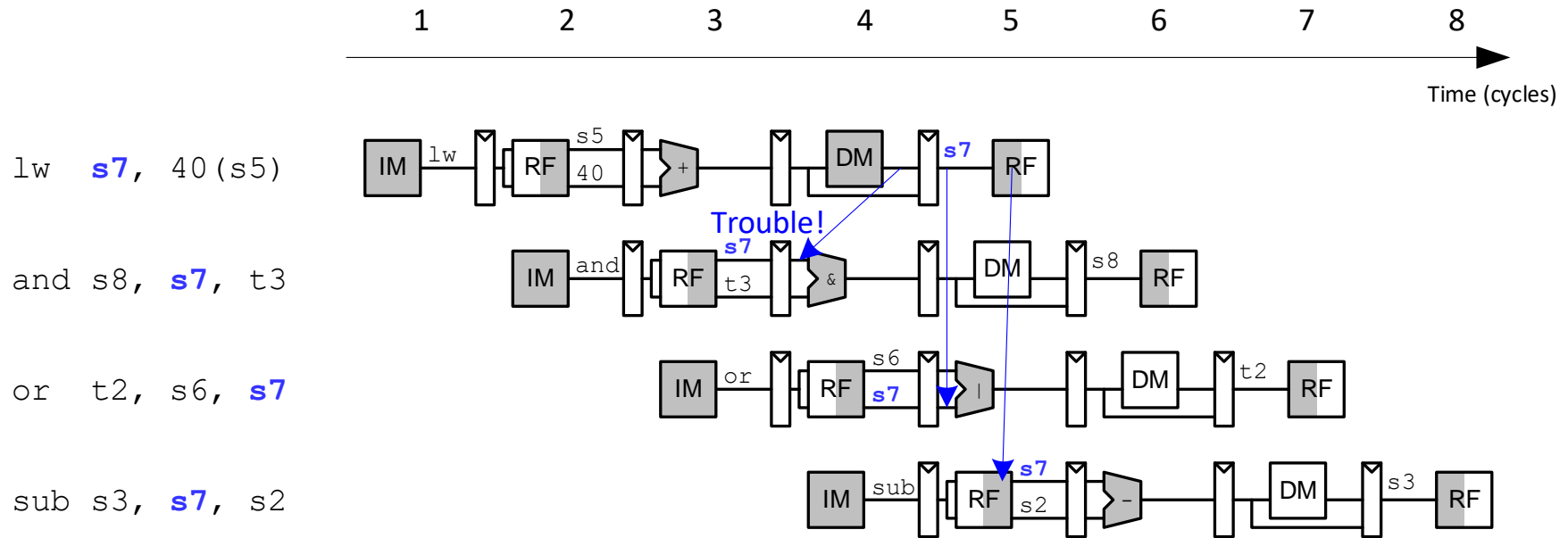








# Data Hazard due to $l_w$ Dependency







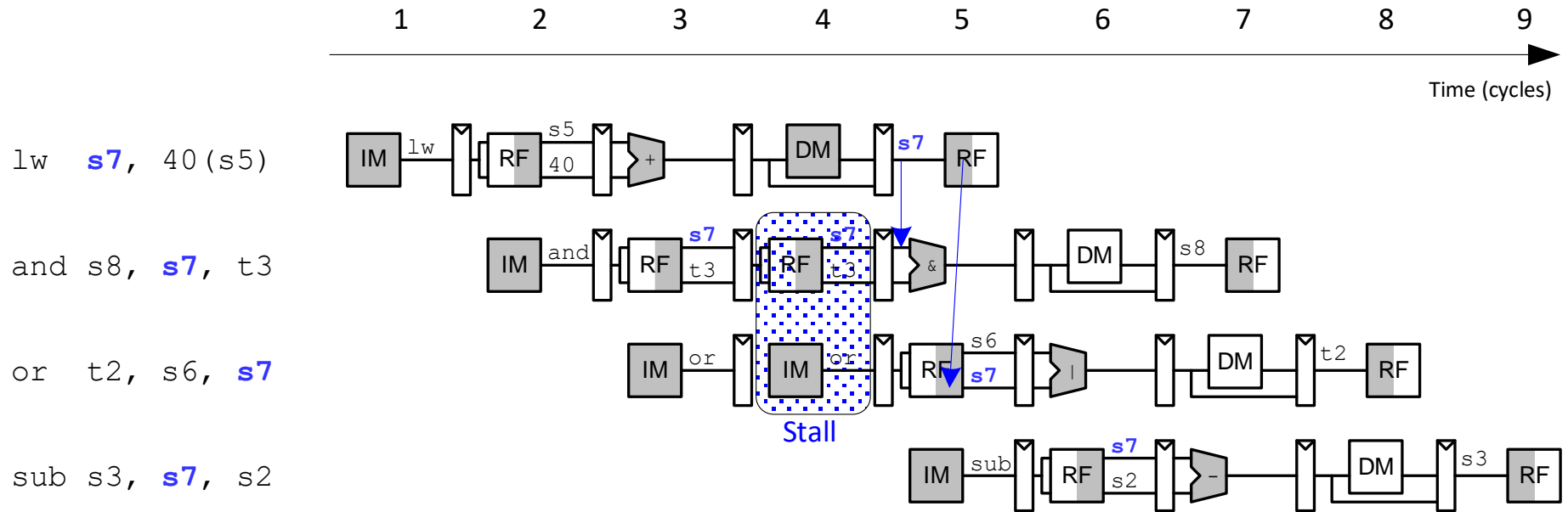








# Stalling to solve 1w Data Dependency















# Stalling Logic

- Is either **source register in the Decode stage** the same as the **destination register in the Execute stage**?

**AND**

- Is the instruction in the **Execute stage a lw**?

$lwStall = ((Rs1D == RdE) \text{ OR } (Rs2D == RdE)) \text{ AND } ResultSrcE_0$

$StallF = StallD = FlushE = lwStall$

(Stall the Fetch and Decode stages, and flush the Execute stage.)





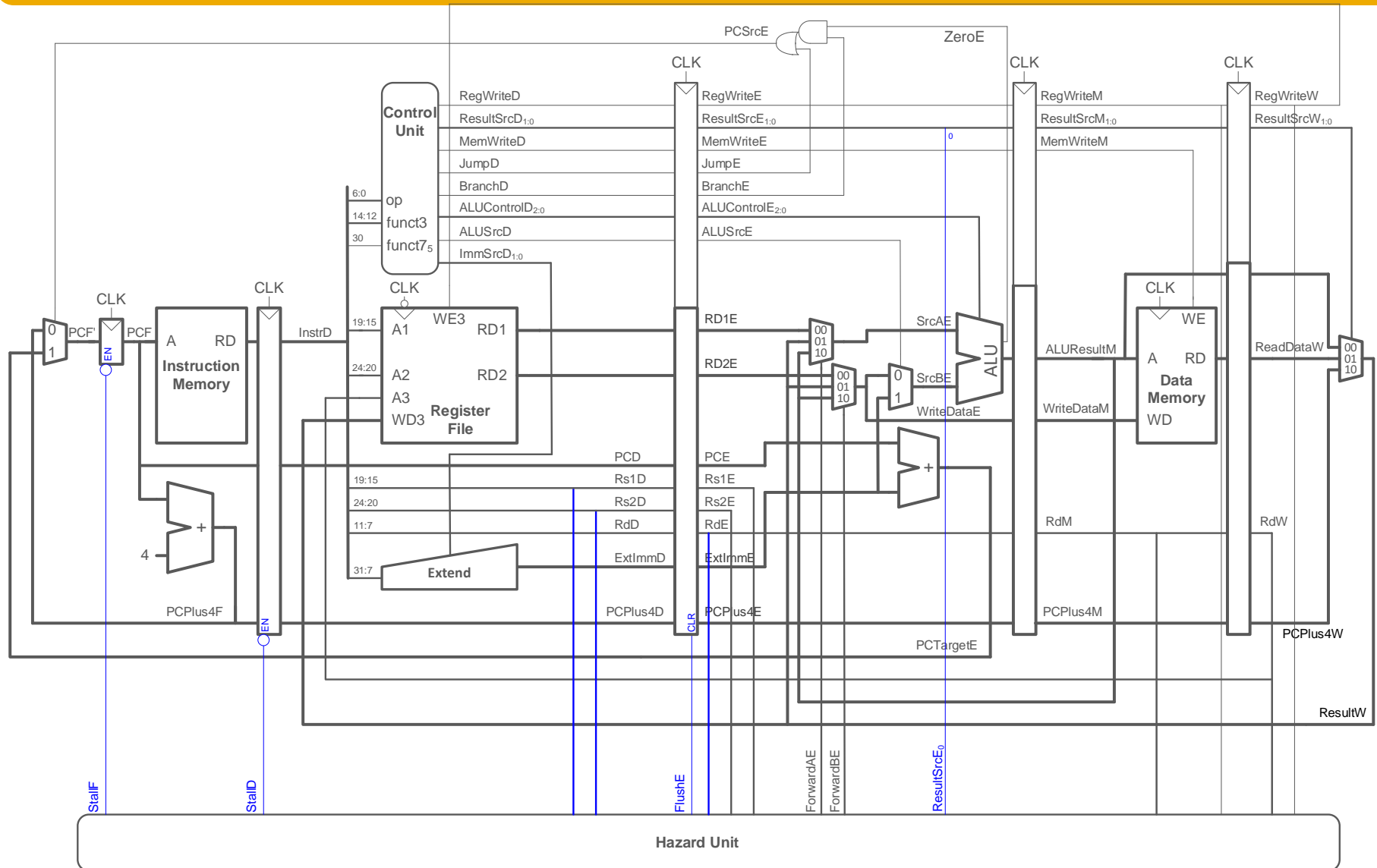








# Stalling Hardware













## Chapter 7: Microarchitecture

# **Pipelined Processor Control Hazards**













# Control Hazards

- **beq:**
  - Branch **not determined** until the **Execute stage** of pipeline
  - **Instructions** after branch **fetch**ed before branch occurs
  - These **2 instructions must be flushed** if branch happens





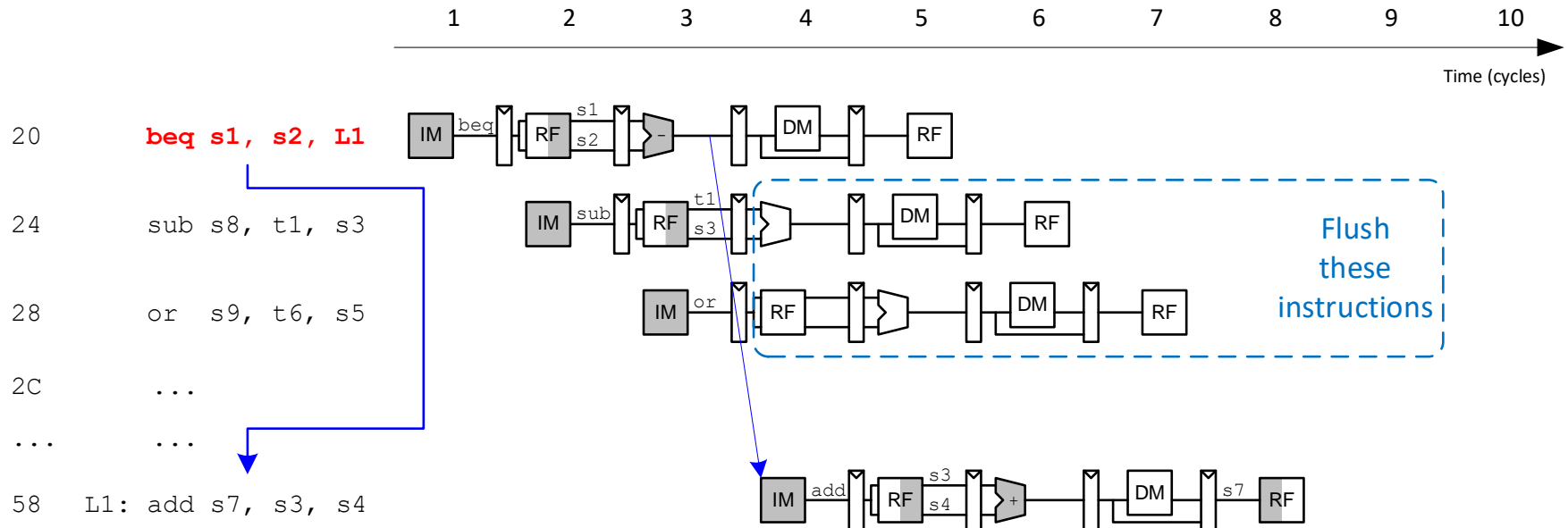








# Control Hazards



## Branch misprediction penalty:

The number of instructions flushed when a branch is taken (in this case, 2 instructions)













# Control Hazards: Flushing Logic

- If branch is taken in execute stage, need to flush the instructions in the Fetch and Decode stages
  - Do this by clearing Decode and Execute Pipeline registers using *FlushD* and *FlushE*
- **Equations:**

$$FlushD = PCSrcE$$

$$FlushE = lwStall \text{ OR } PCSrcE$$



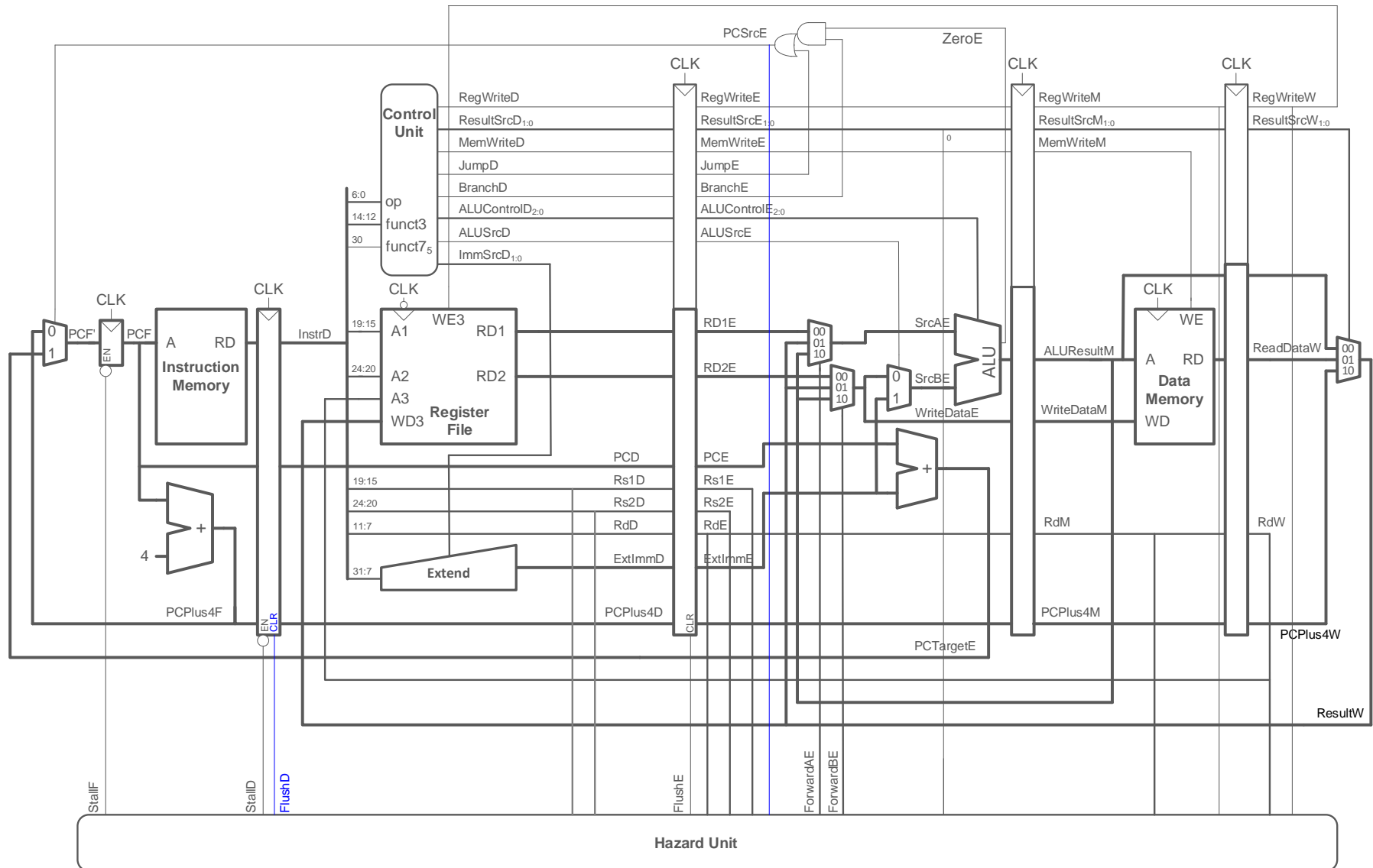








# Control Hazards: Flushing Hardware







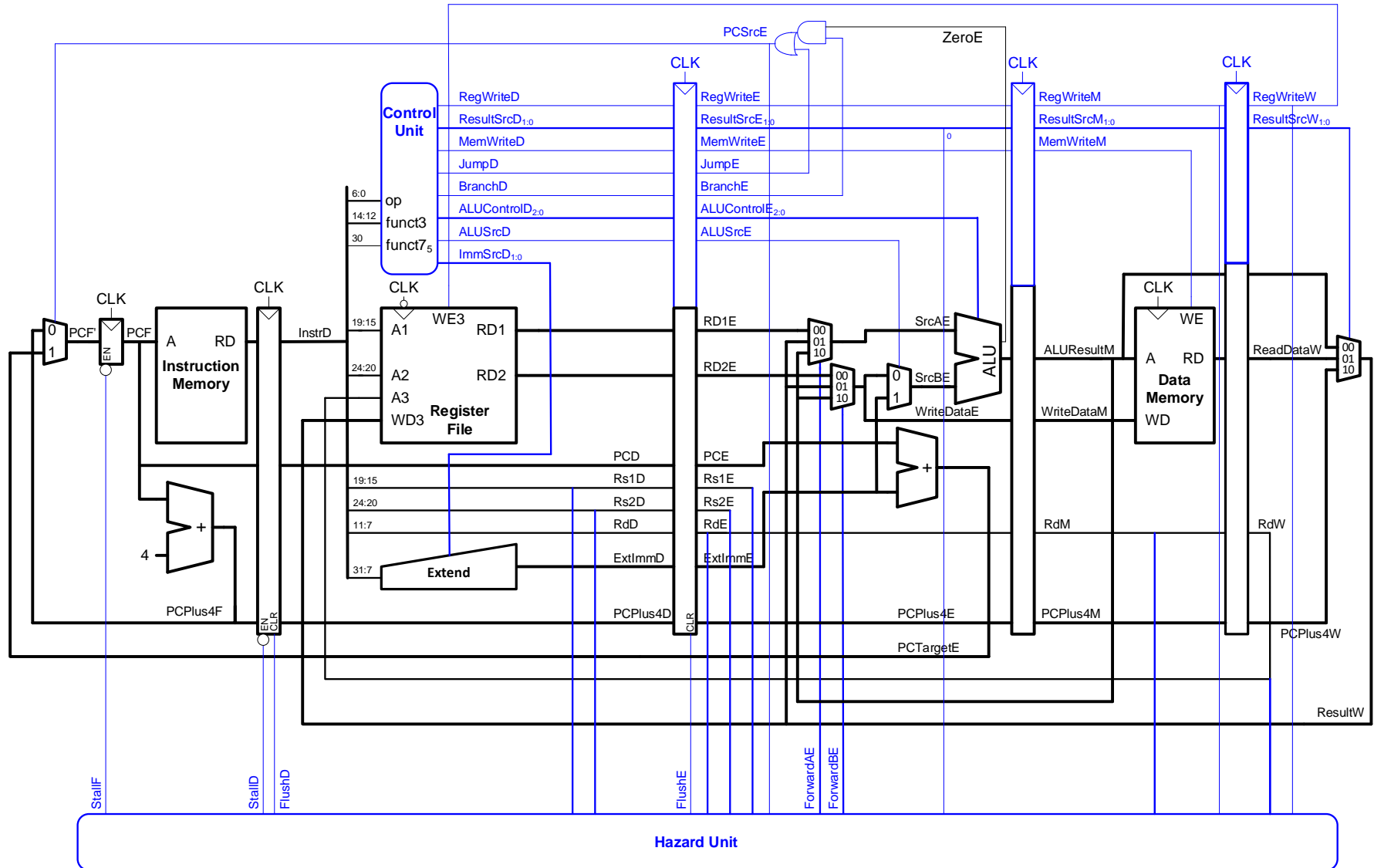








# RISC-V Pipelined Processor with Hazard Unit















# Summary of Hazard Logic

## Data hazard logic (shown for SrcA of ALU):

```
if      ((Rs1E == RdM) AND RegWriteM) AND (Rs1E != 0)    // Case 1
        ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW) AND (Rs1E != 0)    // Case 2
        ForwardAE = 01
else      ForwardAE = 00                                // Case 3
```

## Load word stall logic:

```
lwStall = ((Rs1D == RdE) OR (Rs2D == RdE)) AND ResultSrcE0
StallF = StallD = lwStall
```

## Control hazard flush:

```
FlushD = PCSrcE
FlushE = lwStall OR PCSrcE
```













# Chapter 7: Microarchitecture

## **Pipelined Performance**











# Pipelined Processor Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?** (Ideally it's 1, but...)













# Pipelined Processor Performance Example

Pipelined processor critical path:

$T_{c\_pipelined} = \max \text{ of}$

$$t_{pcq} + t_{mem} + t_{setup}$$

$$2(t_{RFread} + t_{setup})$$

$$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND-OR} + t_{setup}$$

$$t_{pcq} + t_{mem} + t_{setup}$$

$$2(t_{pcq} + t_{mux} + t_{RFwrite})$$

Fetch

Decode

Execute

Memory

Writeback

- Decode and Writeback stages **both use the register file** in each cycle
- So each stage gets half of the cycle time ( $T_c/2$ ) to do their work
- Or, stated a different way, **2x of their work** must fit in a cycle ( $T_c$ )





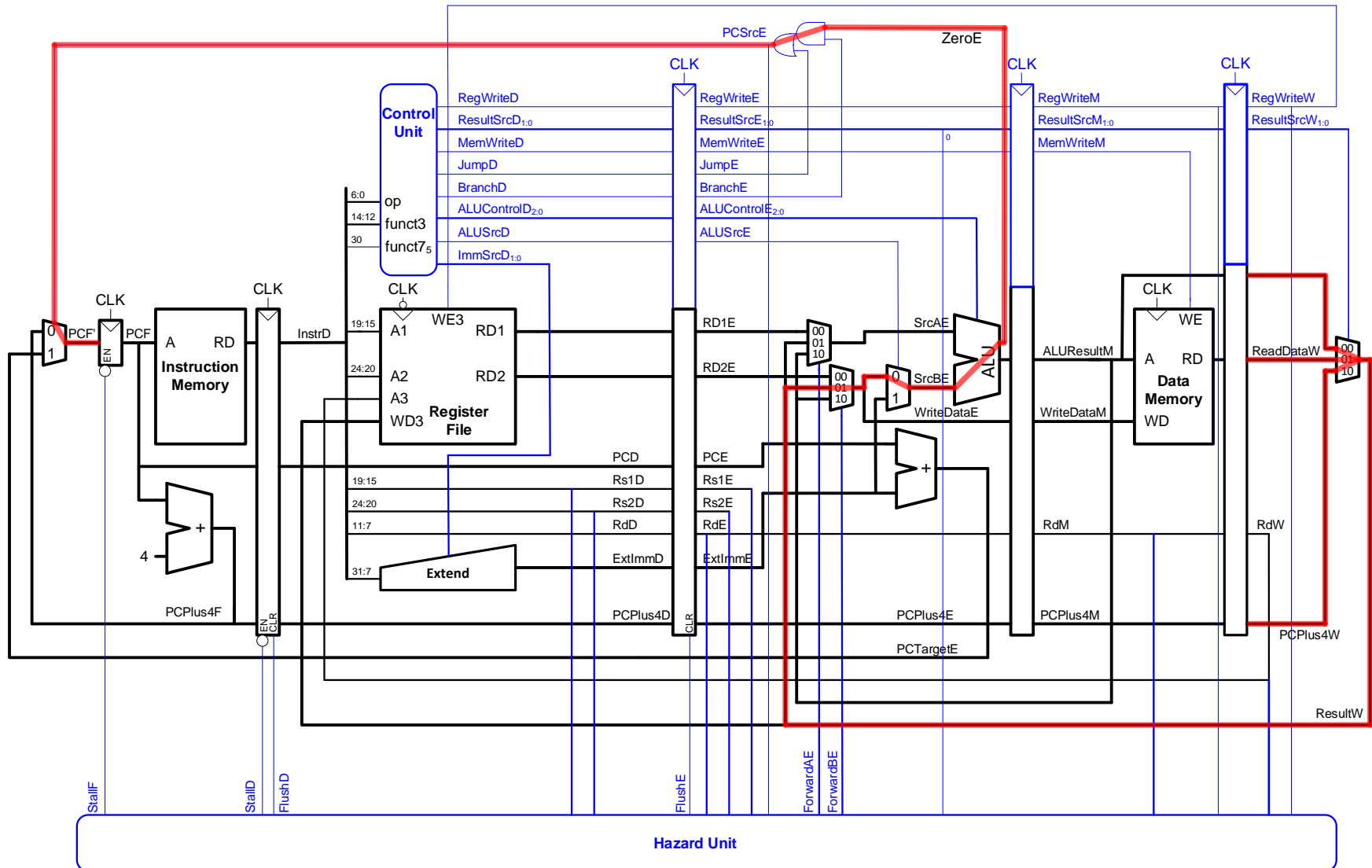








# Pipelined Critical Path: Execute Stage















# Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	$t_{pcq\_PC}$	40
Register setup	$t_{setup}$	50
Multiplexer	$t_{mux}$	30
AND-OR gate	$t_{AND-OR}$	20
ALU	$t_{ALU}$	120
Decoder (Control Unit)	$t_{dec}$	25
Extend unit	$t_{dec}$	35
Memory read	$t_{mem}$	200
Register file read	$t_{RFread}$	100
Register file setup	$t_{RFsetup}$	60

$$T_{c\_pipelined} = t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND-OR} + t_{setup}$$
$$=$$











# Pipelined Performance Example

Program with 100 billion instructions

$$\begin{aligned}\text{Execution Time} &= (\# \text{ instructions}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(1.23)(350 \times 10^{-12}) \\ &= \mathbf{43 \text{ seconds}}\end{aligned}$$













# Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	75	1
Multicycle	155	0.5
Pipelined	43	1.7













## Chapter 7: Microarchitecture

# **Advanced Microarchitecture**













# Advanced Microarchitecture

- Deep Pipelining
- Micro-operations
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors





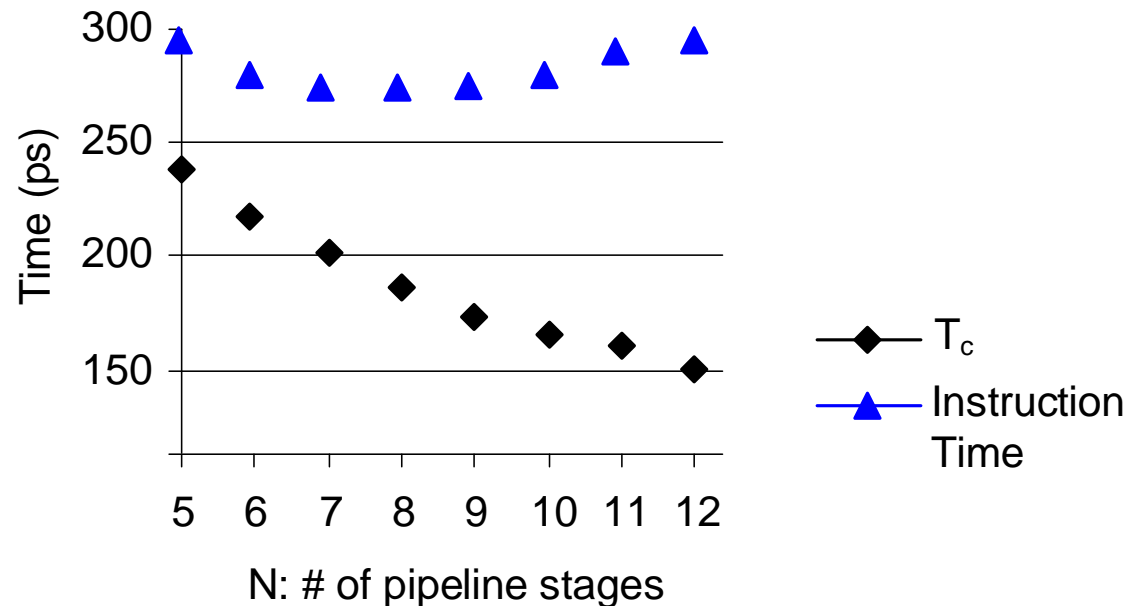






# Deep Pipelining

- **10-20 stages typical**
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost















# Micro-operations

- Decompose complex instructions into series of simple instructions called ***micro-operations*** (*micro-ops* or  $\mu$ -ops)
- **At run-time**, complex instructions are decoded into one or more micro-ops
- Used heavily in **CISC** (complex instruction set computer) architectures (e.g., x86)

## Complex Op

```
lw s1, 0(s2), postincr 4
```

## Micro-op Sequence

```
lw    s1, 0(s2)
addi  s2, s2, 4
```

**Without  $\mu$ -ops, would need 2nd write port on the register file**













# Branch Prediction

- **Guess** whether branch will be taken
  - Backward branches are usually taken (loops)
  - Consider history to improve guess
- Good prediction **reduces fraction of branches requiring a flush**













# Branch Prediction

- Ideal pipelined processor:  $CPI = 1$
- Branch misprediction increases CPI
- **Static branch prediction:**
  - Check direction of branch (forward or backward)
  - If backward, predict taken
  - Else, predict not taken
- **Dynamic branch prediction:**
  - Keep **history** of last several hundred (or thousand) branches in *branch target buffer*, record:
    - Branch destination
    - Whether branch was taken











# Dynamic Branch Prediction

- 1-bit branch predictor
- 2-bit branch predictor













# Branch Prediction Example

```
addi s1, zero, 0      # s1 = sum
addi s0, zero, 0      # s0 = i
addi t0, zero, 10     # t0 = 10
```

```
For:                  # for (i=0; i<10; i=i+1)
    bge s0, t0, Done
    add s1, s1, s0     # sum = sum + i
    addi s0, s0, 1     # i = i + 1
    j For
```

Done:













# 1-Bit Branch Predictor

- **Remembers** whether branch was taken the last time and **does the same thing**
- Mispredicts first and last branch of loop

```
addi s1, zero, 0      # s1 = sum
addi s0, zero, 0      # s0 = i
addi t0, zero, 10     # t0 = 10
```

```
For:                  # for (i=0; i<10; i=i+1)
    bge s0, t0, Done
    add s1, s1, s0     # sum = sum + i
    addi s0, s0, 1     # i = i + 1
    j For
```

Done:





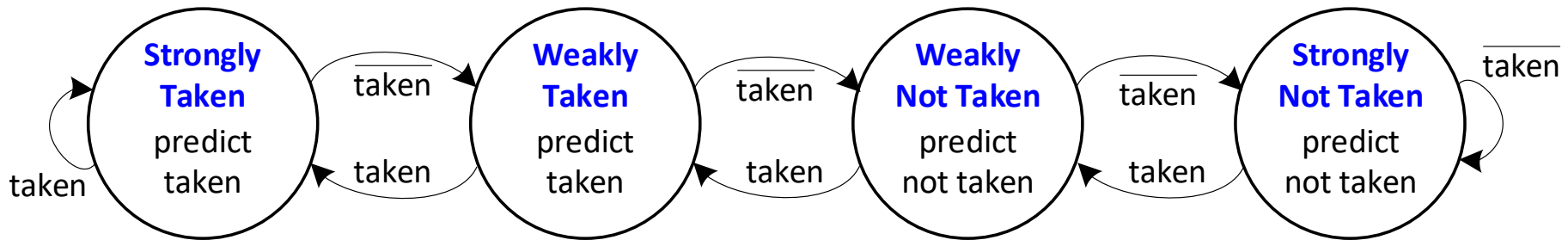








# 2-Bit Branch Predictor



```
addi s1, zero, 0      # s1 = sum
addi s0, zero, 0      # s0 = i
addi t0, zero, 10     # t0 = 10
```

```
For:                    # for (i=0; i<10; i=i+1)
    bge s0, t0, Done
    add s1, s1, s0      # sum = sum + i
    addi s0, s0, 1      # i = i + 1
j      For
```

Done:

Only mispredicts **last branch** of loop











## Chapter 7: Microarchitecture

# **Superscalar & Out of Order Processors**







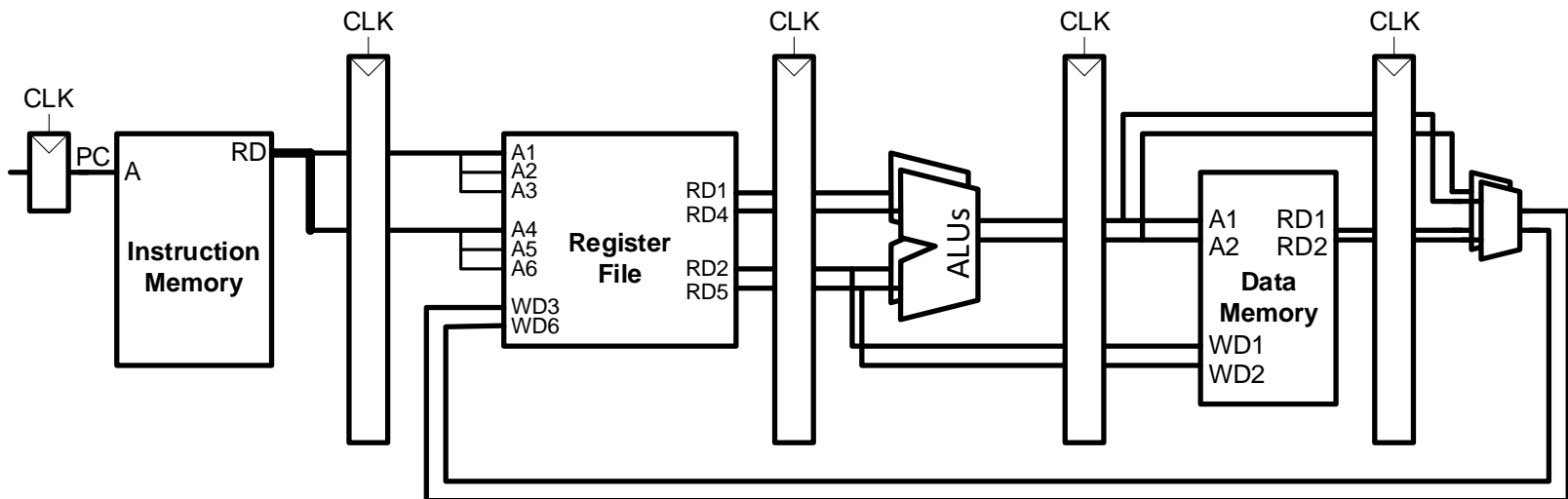






# Superscalar Processors

- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once











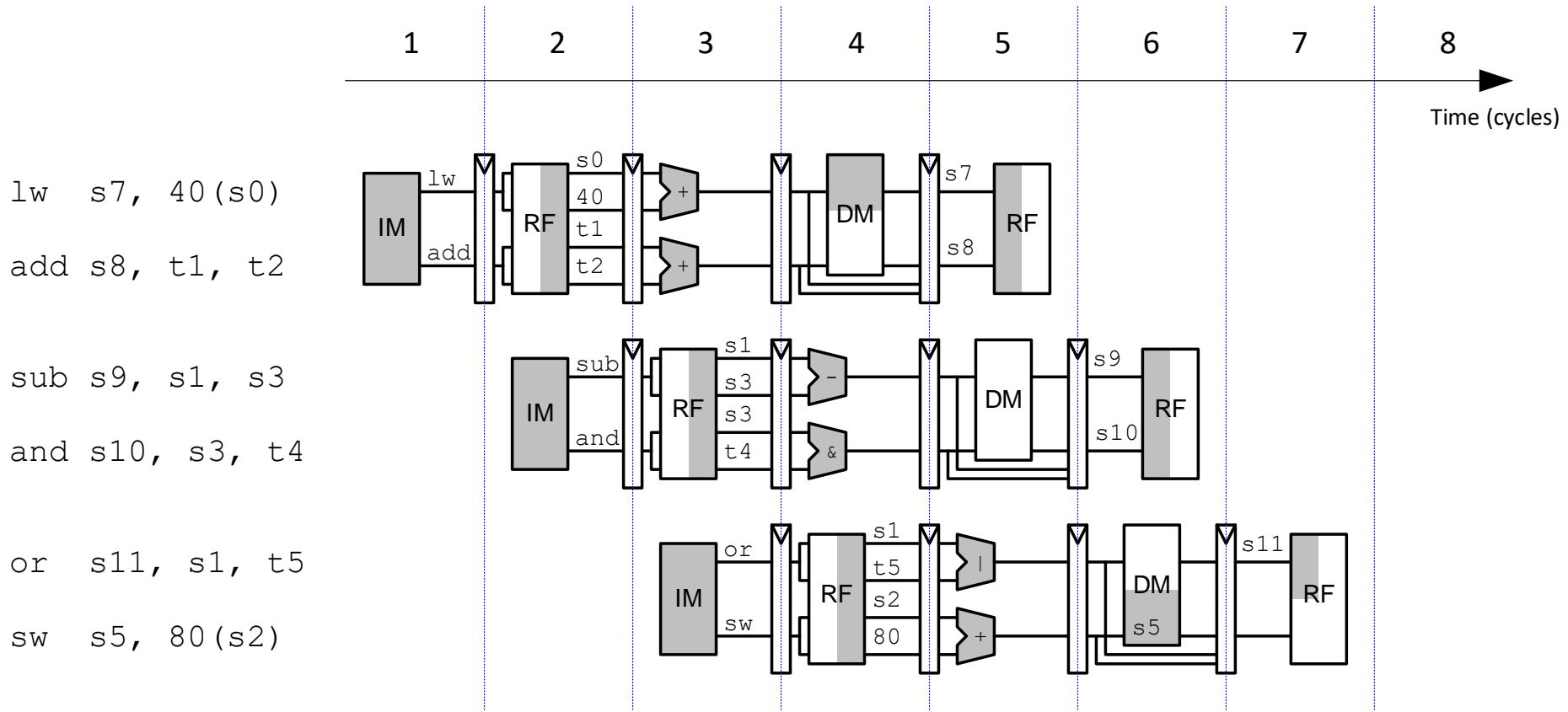




# Superscalar Example

**Ideal IPC: 2**

**Actual IPC: 2**











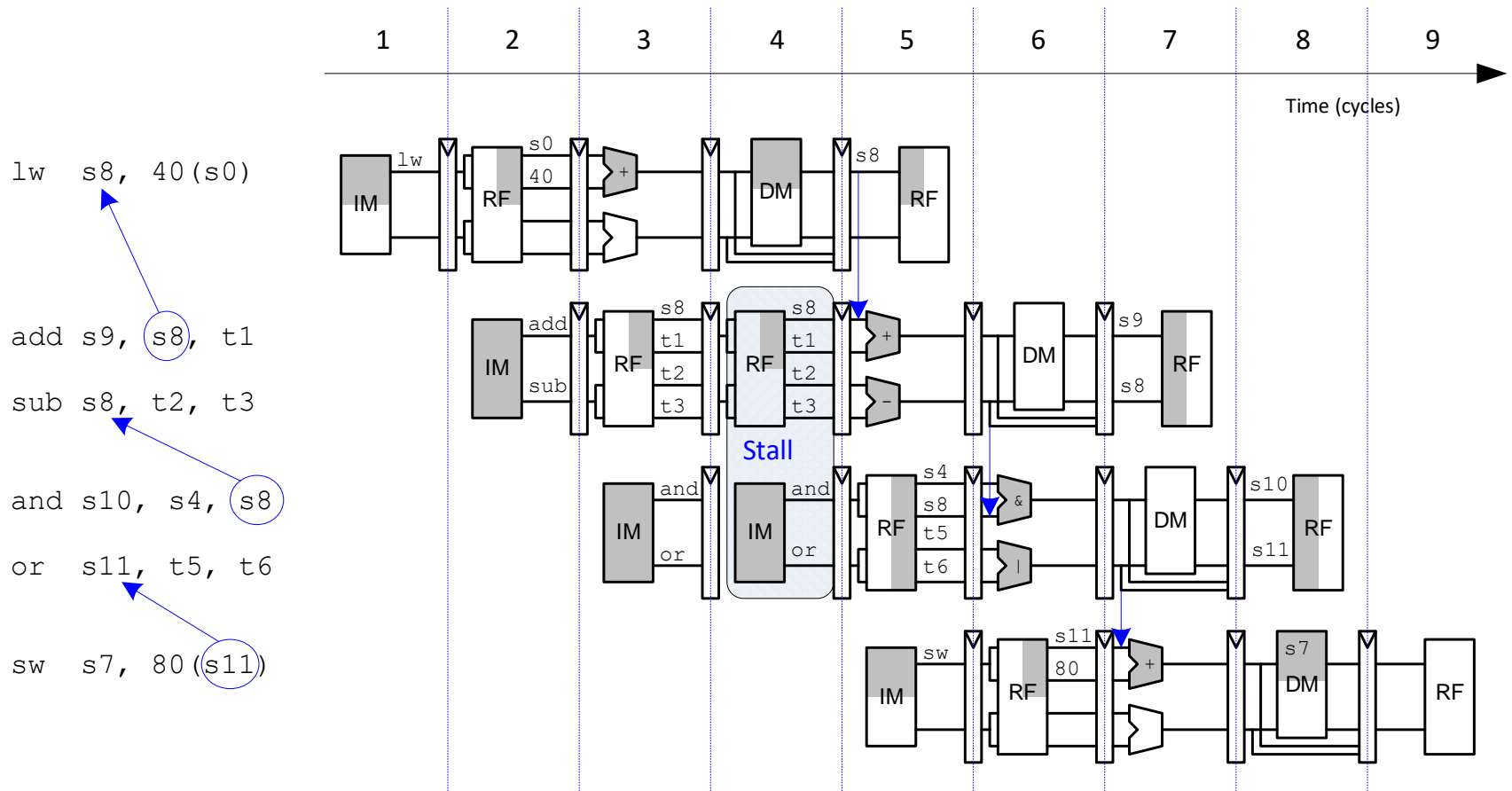




# Superscalar with Dependencies

**Ideal IPC: 2**

**Actual IPC:  $6/5 = 1.2$**













# Out of Order (OOO) Processor

- Looks ahead across multiple instructions
- Issues as many instructions as possible at once
- Issues instructions out of order (as long as no dependencies)
- **Dependencies:**
  - **RAW** (read after write): one instruction writes, later instruction reads a register
  - **WAR** (write after read): one instruction reads, later instruction writes a register
  - **WAW** (write after write): one instruction writes, later instruction writes a register













# Out of Order (OOO) Processor

- **Instruction level parallelism (ILP):** number of instruction that can be issued simultaneously (average  $< 3$ )
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies









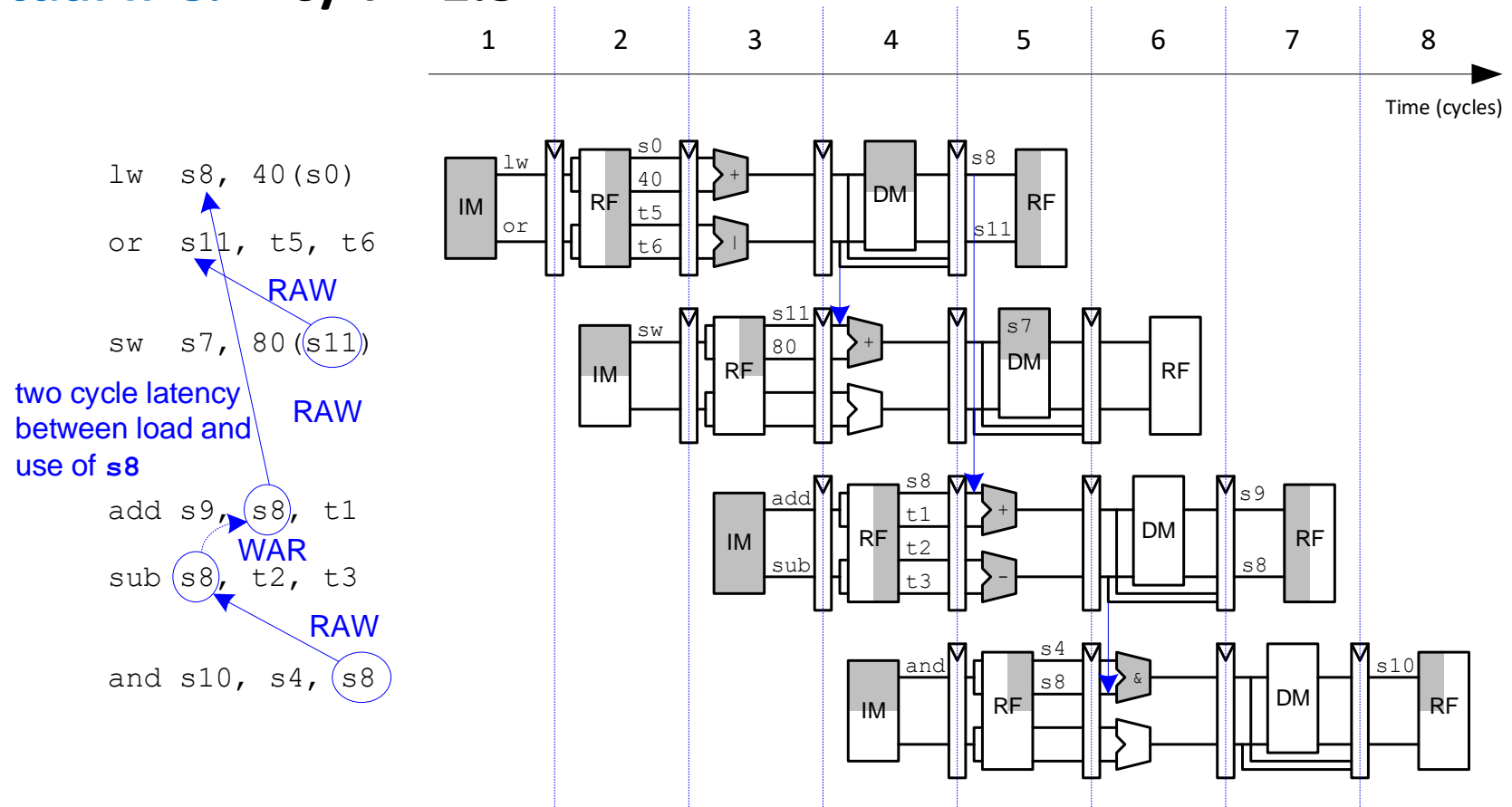




# Out of Order Processor Example

Ideal IPC: 2

Actual IPC:  $6/4 = 1.5$











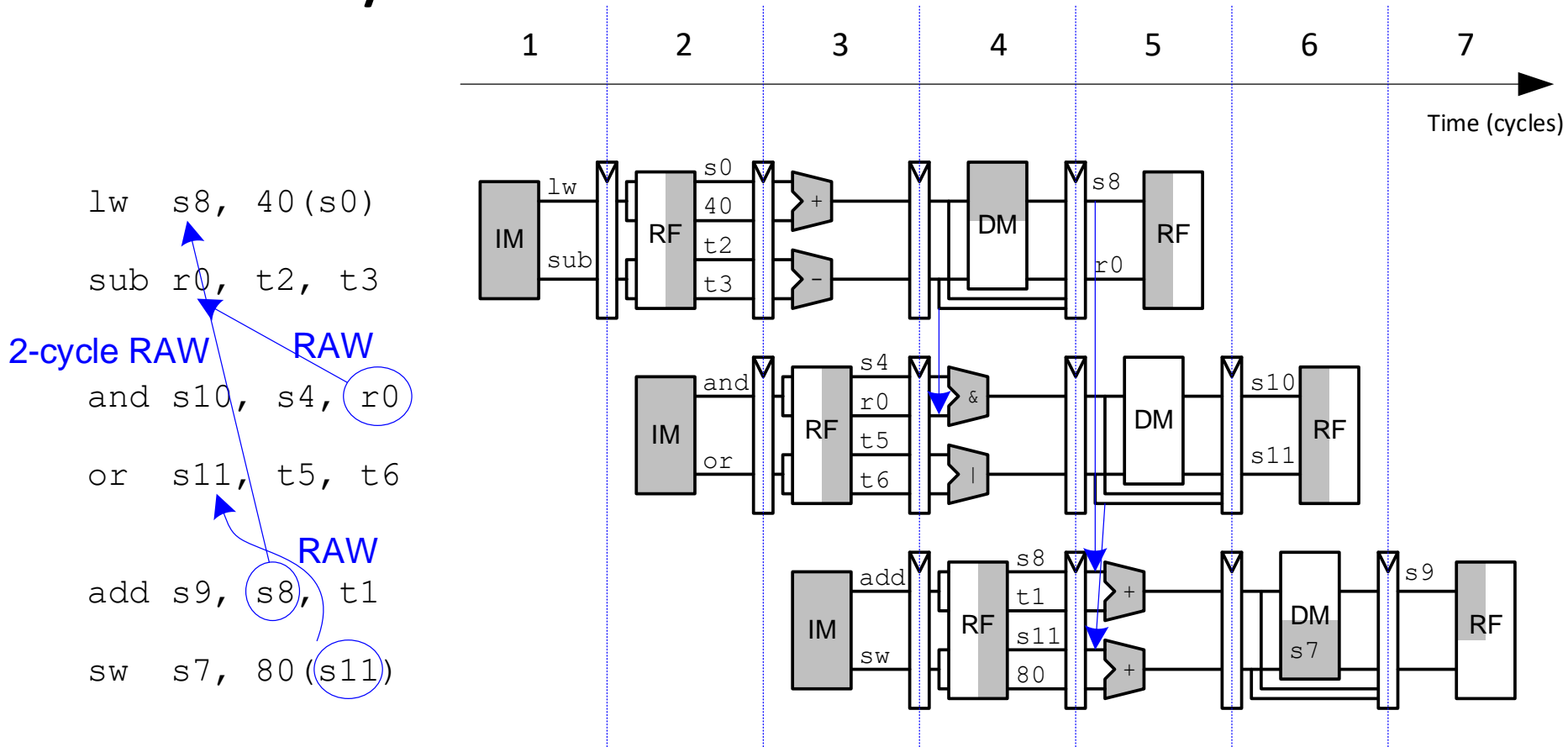




# Register Renaming

**Ideal IPC: 2**

**Actual IPC:  $6/3 = 2$**







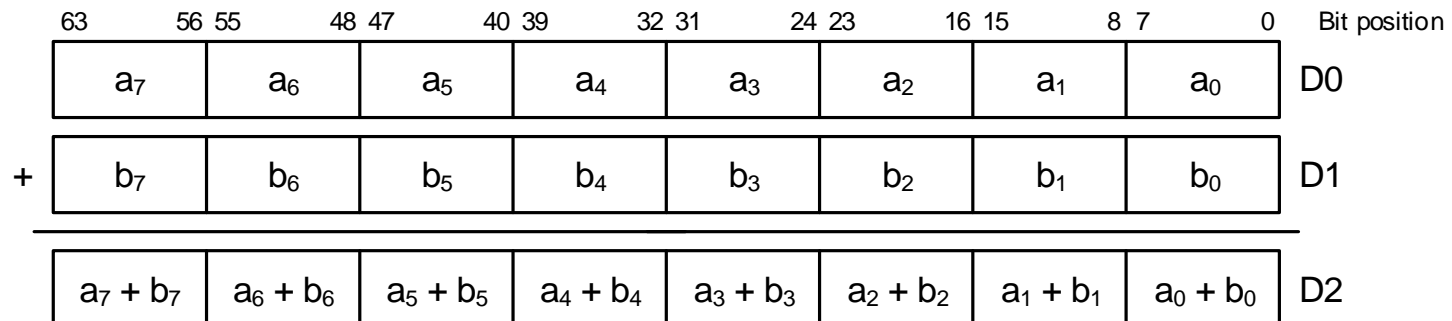






# SIMD

- **Single Instruction Multiple Data (SIMD)**
  - Single instruction **acts on multiple pieces of data at once**
  - Common application: **graphics**
  - Can apply to short arithmetic operations (also called *packed arithmetic*)
- For example, add eight 8-bit elements















## Chapter 7: Microarchitecture

# **Multithreading & Multiprocessors**













# Advanced Architecture Techniques

- **Multithreading**
  - Wordprocessor: thread for typing, spell checking, printing
- **Multiprocessors**
  - Multiple processors (cores) on a single chip













# Threading: Definitions

- **Process:** program running on a computer
  - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
  - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing











# Threads in a Conventional Processor

## Single-core system:

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called **context switching**
- Appears to user like all threads running simultaneously













# Multithreading

- Multiple copies of architectural state
- Multiple threads **active** at once:
  - When one thread stalls, another runs immediately
  - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

**Intel calls this “hyperthreading”**













# Multiprocessors

- Multiple processors (cores) with a method of communication between them
- Types:
  - **Homogeneous:** multiple cores with shared main memory
  - **Heterogeneous:** separate cores for different tasks (for example, DSP and CPU in cell phone)
  - **Clusters:** each core has own memory system













# About these Notes

**Digital Design and Computer Architecture Lecture Notes**

**© 2021 Sarah Harris and David Harris**

**These notes may be used and modified for educational and/or non-commercial purposes so long as the source is attributed.**









