Computersysteme

Microarchitecture

Markus Bader

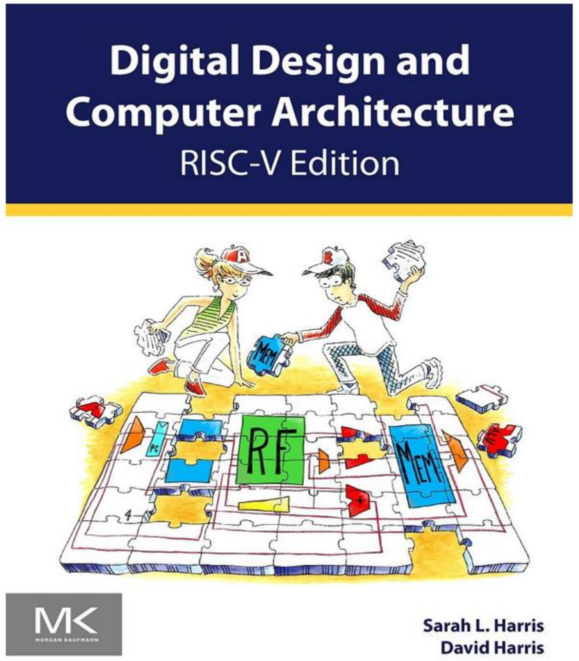
SS2025

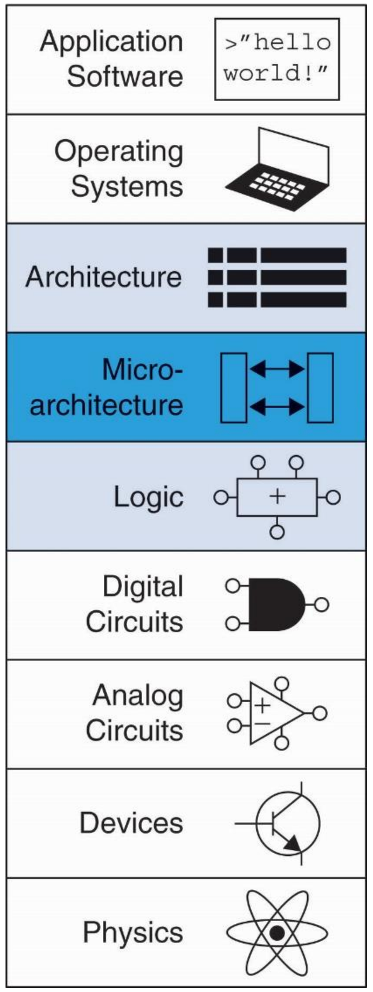
Introduction

DDCA Ch7 - Part 1: Microarchitecture Introduction https://youtu.be/lrN-uBKooRY?si=QEiy6eyr5c32m31n

Chapter 7 :: Topics

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





About these Notes

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

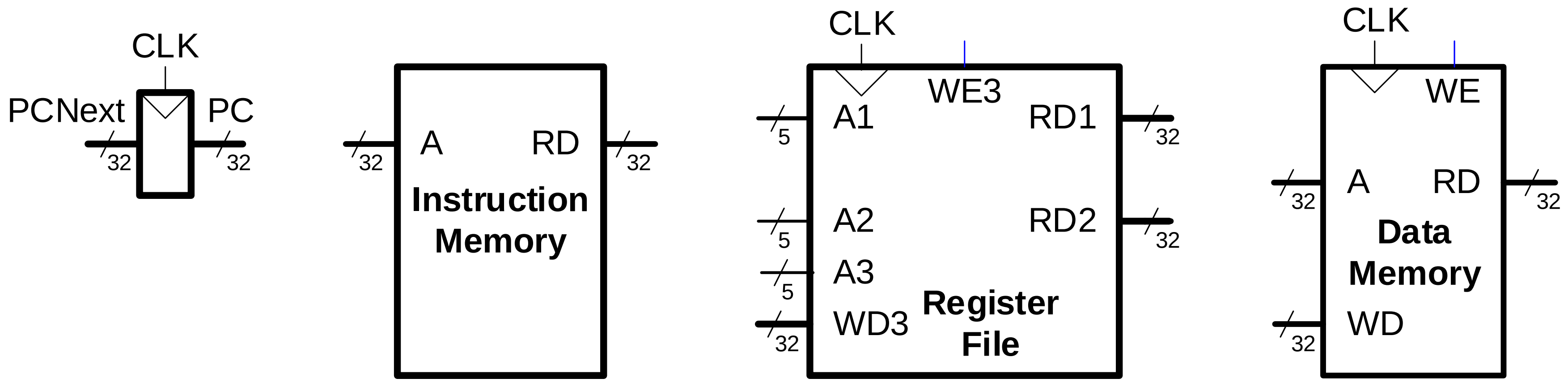
RISC-V Processor

- 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



Single-Cycle RISC-V Processor

DDCA Ch7 - Part 2: RISC-V Single-Cycle Processor Datapath: lw https://youtu.be/AoBkibslRBM?si=fgO1anrXwzMCdOrU

Single-Cycle RISC-V Processor

- Datapath
- Control

Example Program

- Design datapath
- View example program executing

Address	Instruction	Type	Fields						Machine Language
			$imm_{11:0}$		**rs1**	**f3**	**rd**	**op**	
0x1000	L7: lw x6, -4(x9)	I	111111111100		01001	010	00110	0000011	FFC4A303
			$imm_{11:5}$	**rs2**	**rs1**	**f3**	**$imm_{4:0}$**	**op**	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	**rs2**	**rs1**	**f3**	**rd**	**op**	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			$imm_{12,10:5}$	**rs2**	**rs1**	**f3**	**$imm_{4:1,11}$**	**op**	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Single-Cycle RISC-V Processor

- **Datapath**: start with `lw` instruction

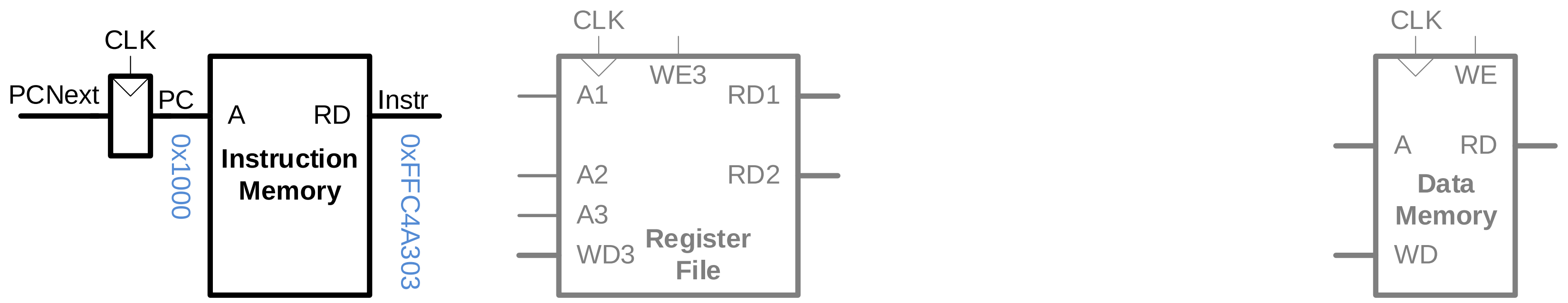
- Example:
```
lw x6, -4(x9)
lw rd, imm(rs1)
```

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

Single-Cycle Datapath: `lw` fetch

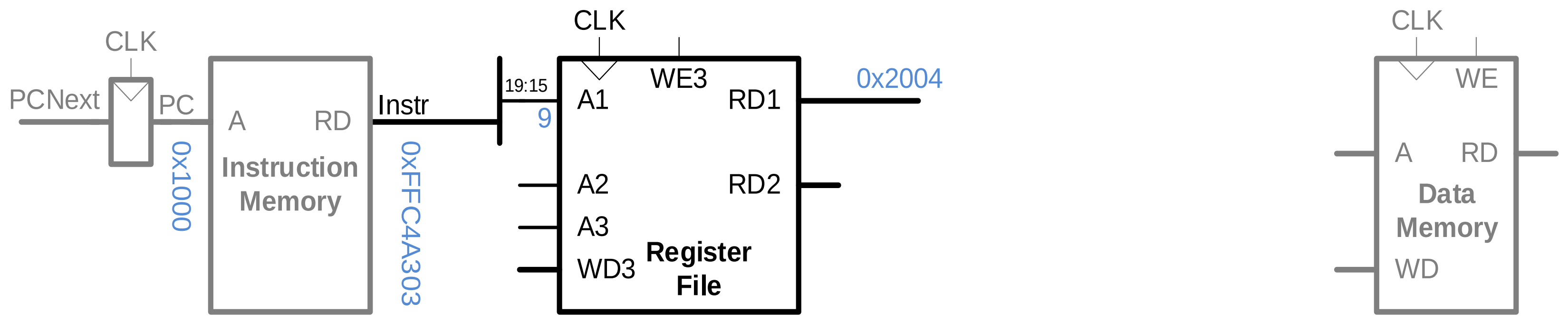
- **STEP 1:** Fetch instruction



Address	Instruction	Type	Fields				Machine Language	
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Reg Read

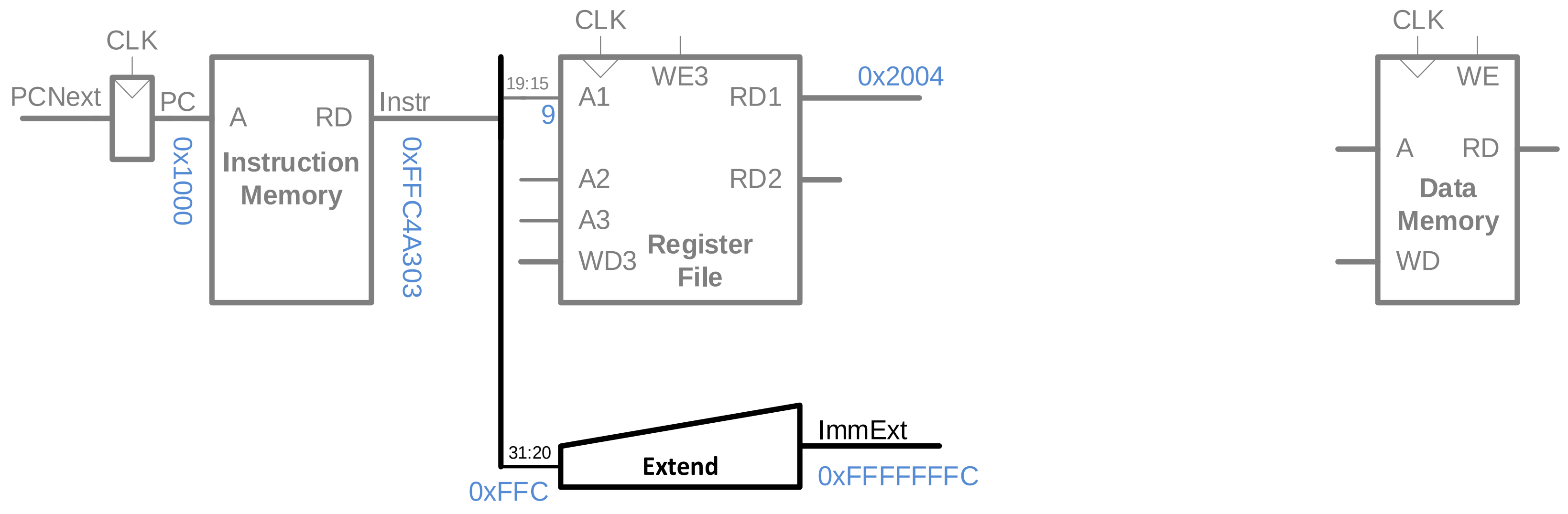
- **STEP 2:** Read source operand (**rs1**) from RF



Address	Instruction	Type	imm$_{11:0}$	rs1	f3	rd	op	Machine Language
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Immediate

- **STEP 3:** Extend the immediate

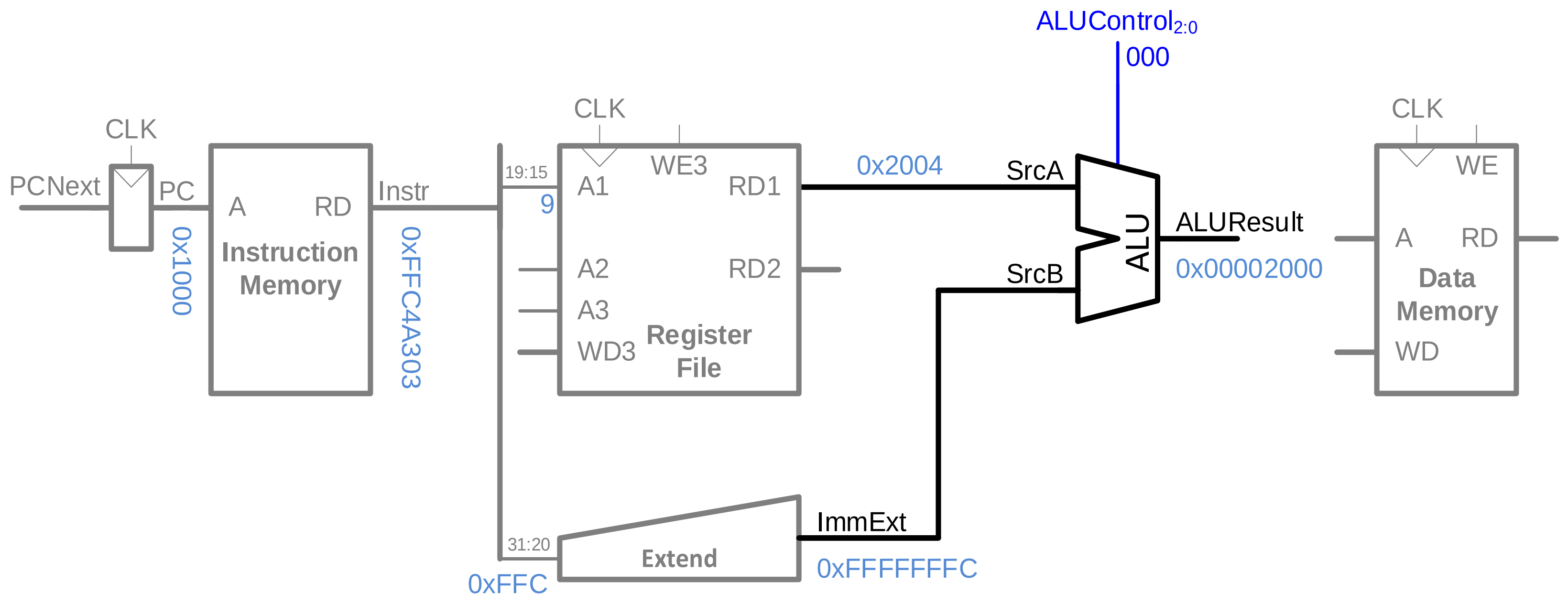


Address	Instruction	Type	Fields				Machine Language	
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Address

- **STEP 4:** Compute the memory address

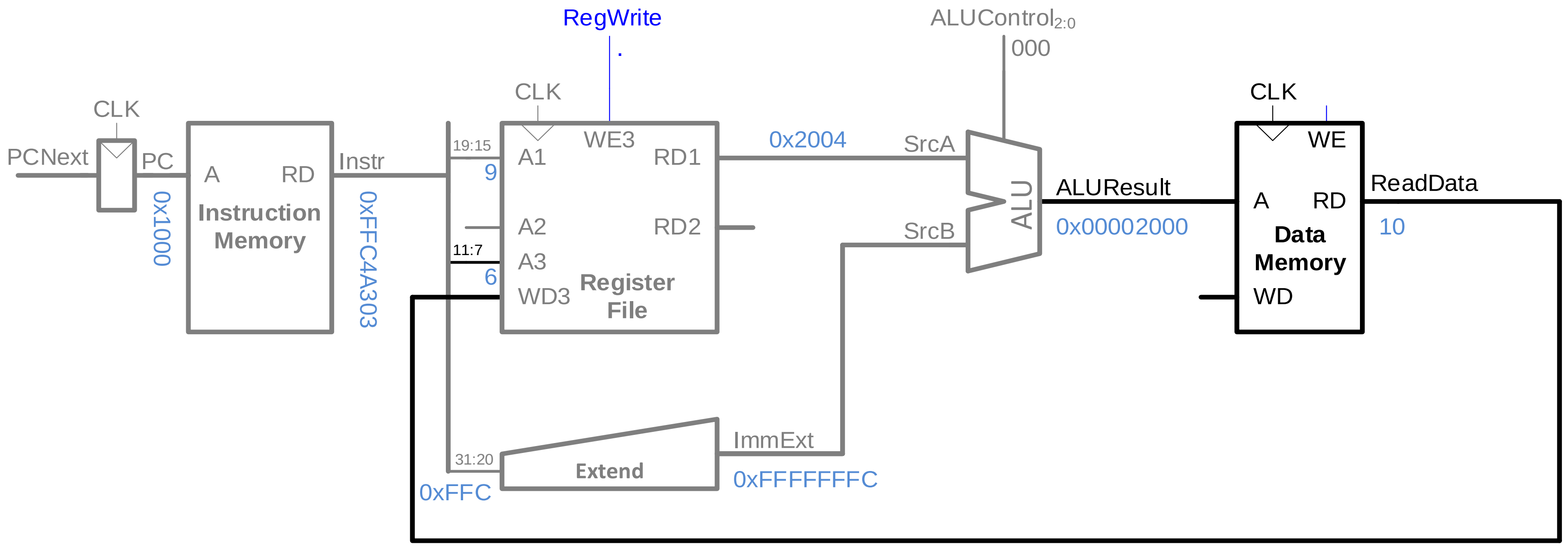
ALUControl$_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT



Address	Instruction	Type	Fields				Machine Language	
			imm$_{11:0}$	rs1	f3	rd	op	
`0x1000`	`L7: lw x6, -4(x9)`	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Mem Read

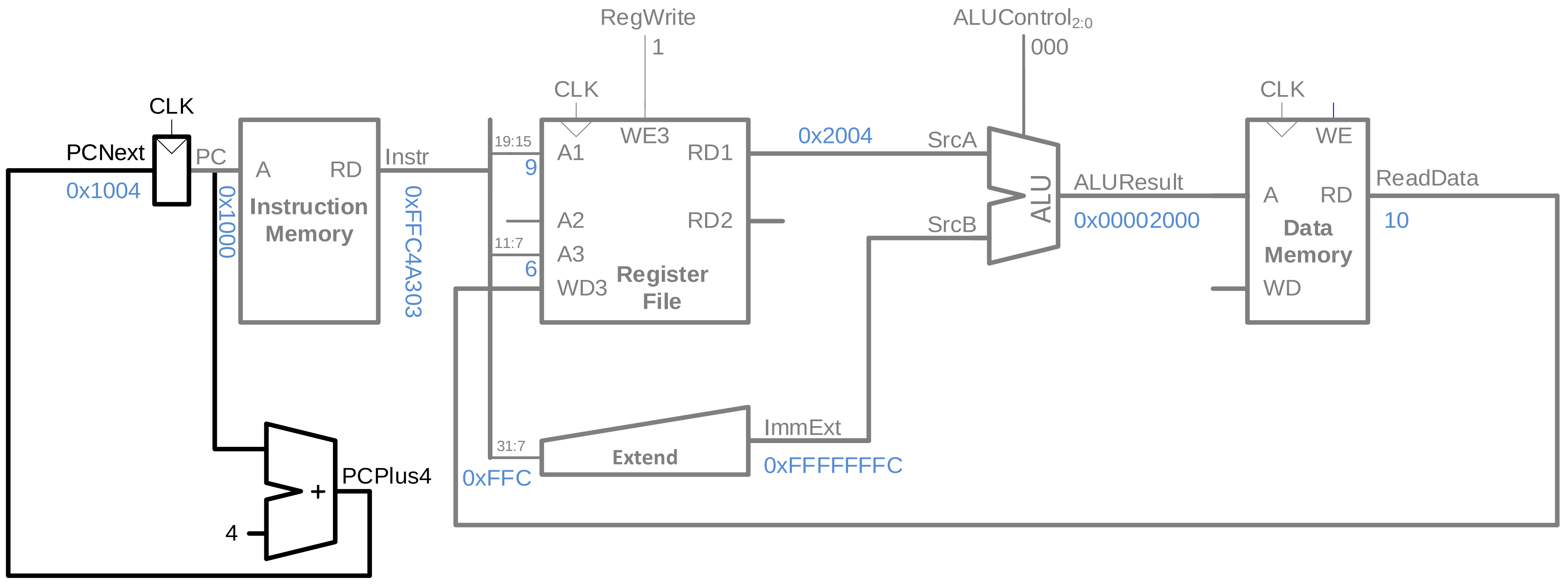
- **STEP 5:** Read data from memory and write it back to register file



Address	Instruction	Type	imm$_{11:0}$	rs1	f3	rd	op	Machine Language
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: PC Increment

- **STEP 6:** Determine address of next instruction



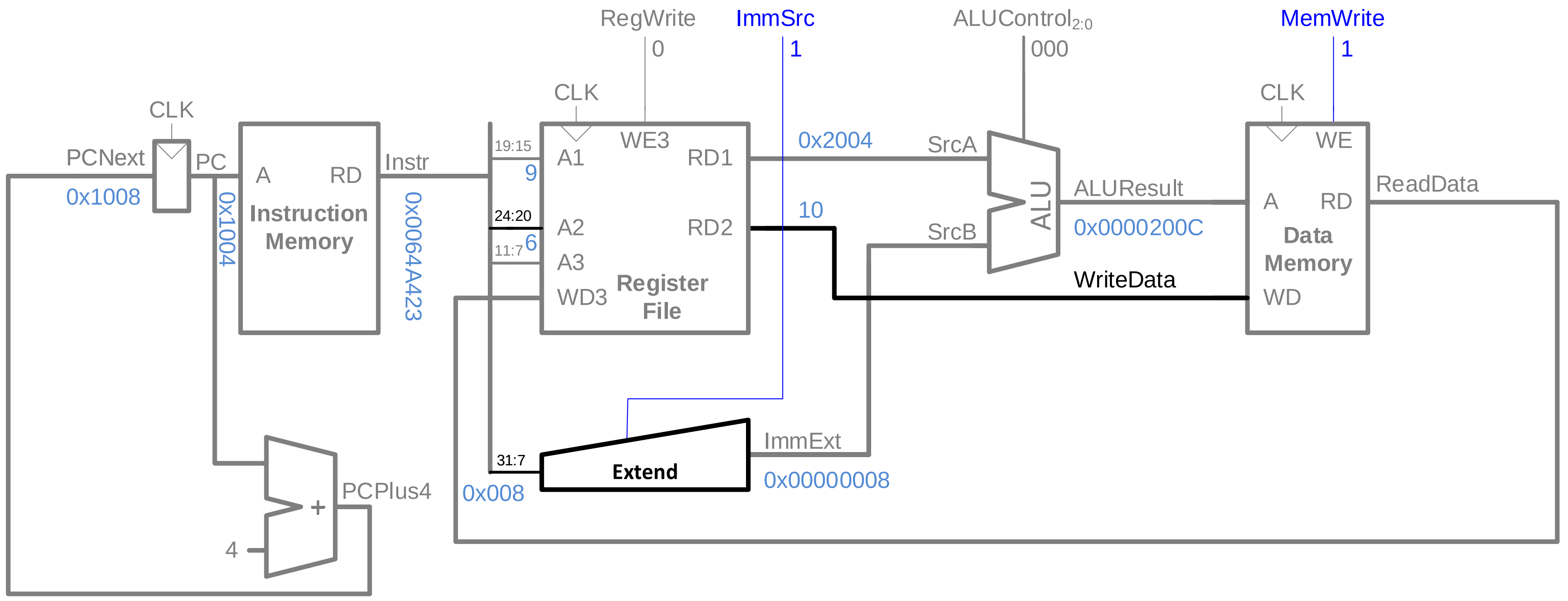
Address	Instruction	Type	Fields				Machine Language
			imm$_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: Other Instructions

DDCA Ch7 - Part 3: RISC-V Single-Cycle Processor Datapath https://youtu.be/sVZmqLRkbVk?si=SE_gnswCekVKNuwe

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$_{11:5}$	rs2	rs1	f3	imm$_{4:0}$	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]}}, **instr[31:20]**}	I-Type
1	{{20{instr[31]}}, **instr[31:25], instr[11:7]**}	S-Type

I-Type

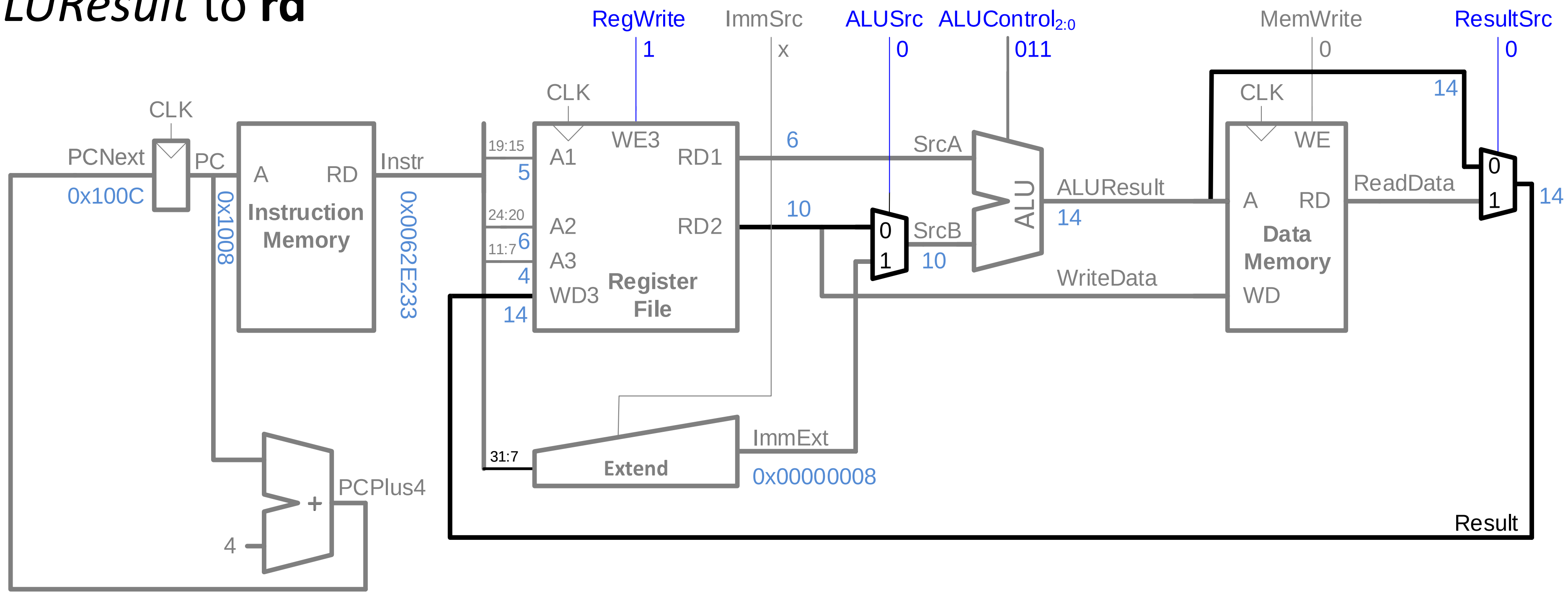
31:20	19:15	14:12	11:7	6:0
imm$_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

S-Type

31:25	24:20	19:15	14:12	11:7	6:0
imm$_{11:5}$	rs2	rs1	funct3	**imm$_{4:0}$**	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Single-Cycle Datapath: R-type

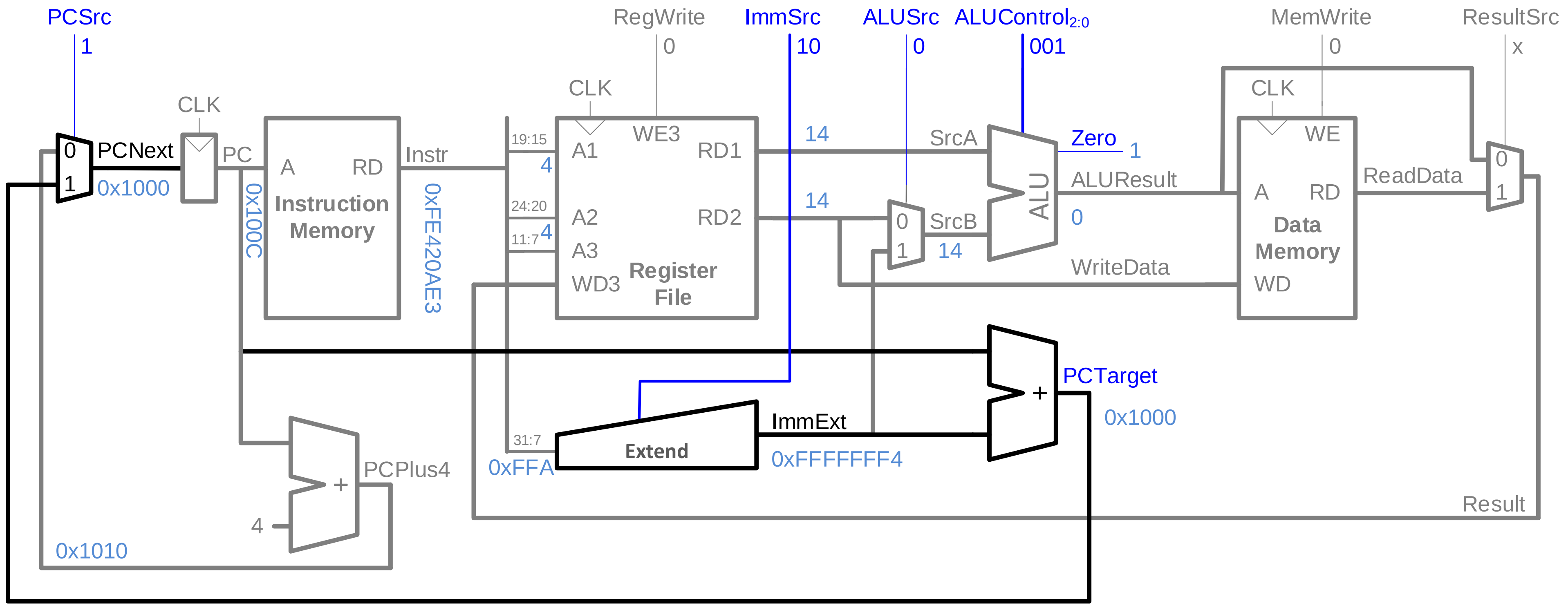
- Read from **rs1** and **rs2** (instead of **imm**)
- Write *ALUResult* to **rd**



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
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	`beq x4, x4, L7`	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Single-Cycle Datapath: ImmExt

$ImmSrc_{1:0}$	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

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

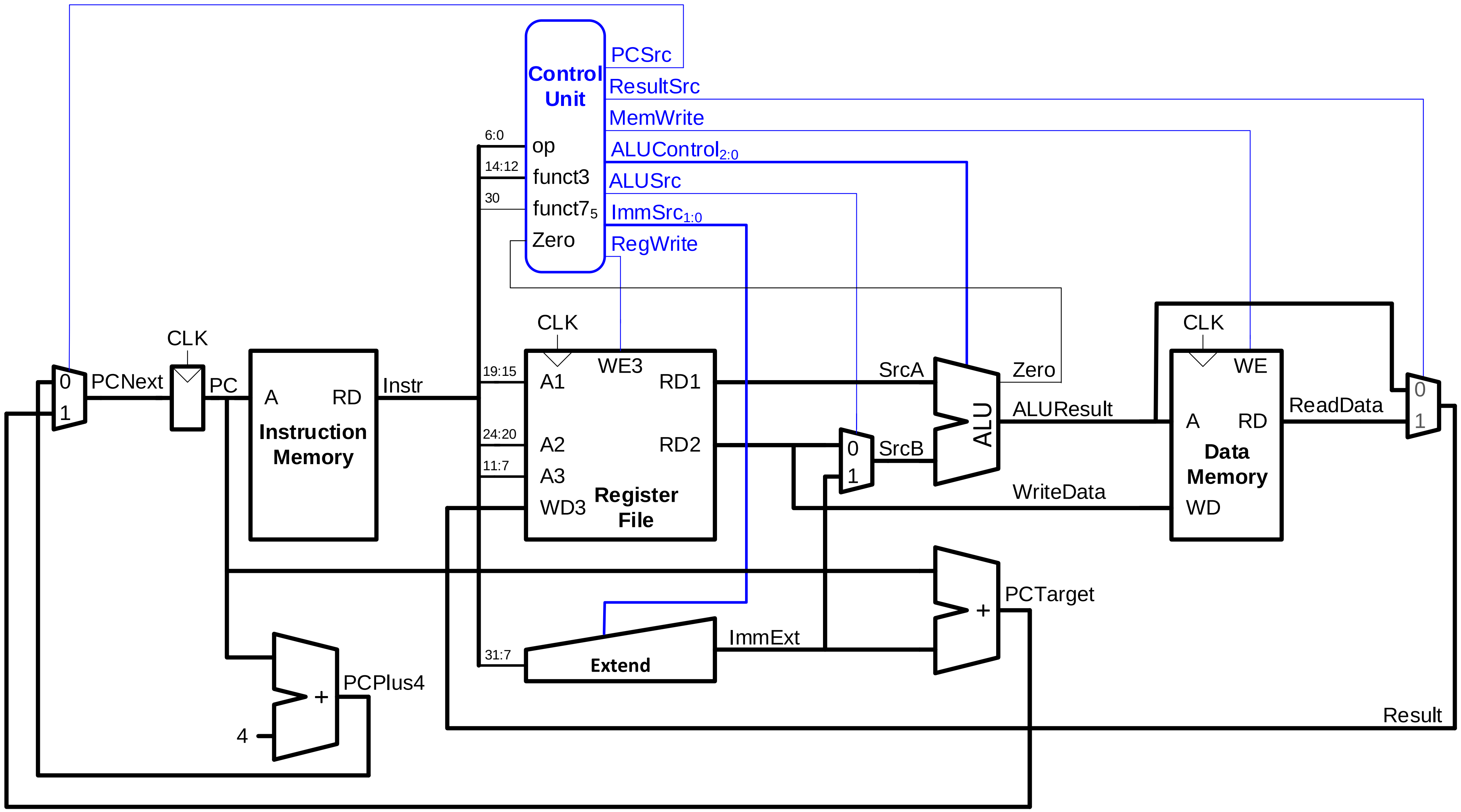
S-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Single-Cycle RISC-V Processor

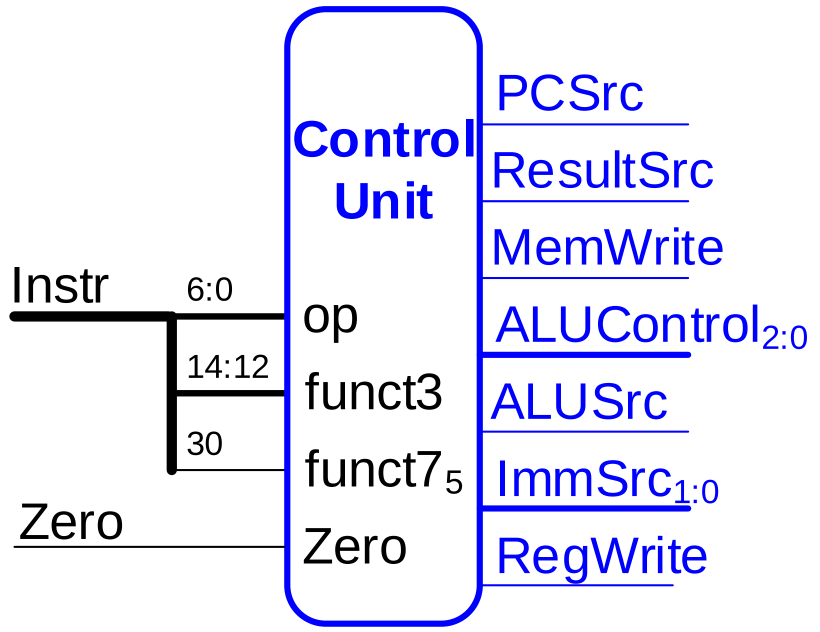


Single-Cycle Control

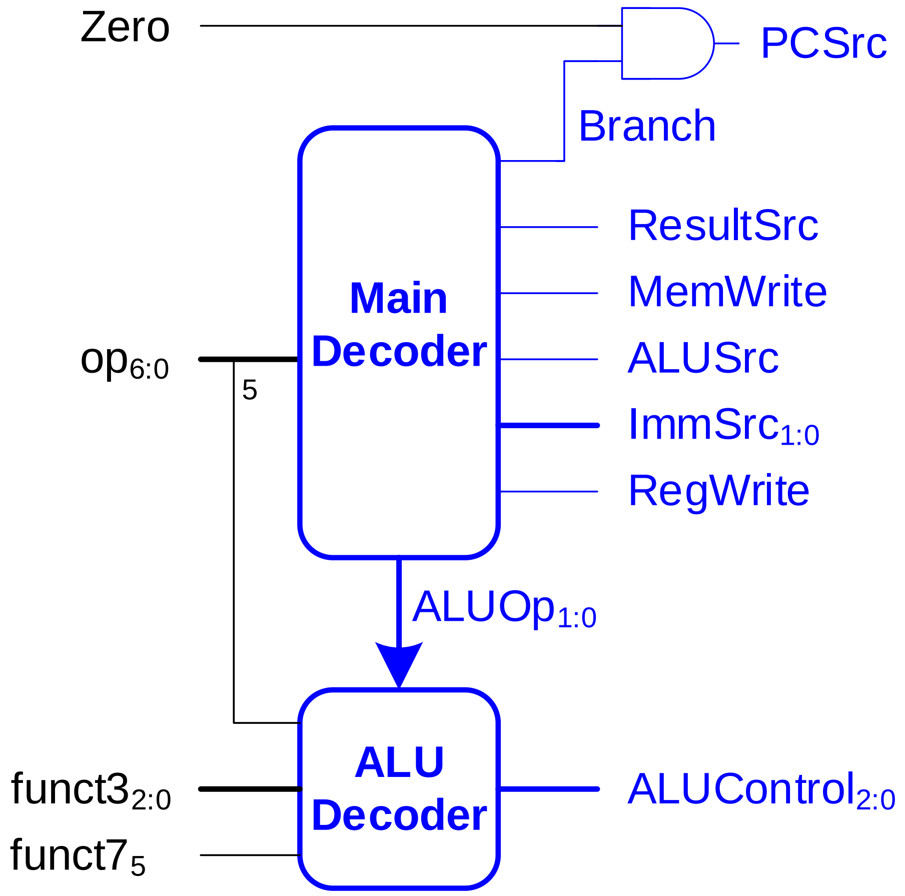
DDCA Ch7 - Part 4: RISC-V Single-Cycle Processor: Control https://www.youtube.com/watch?v=EZb1_VF-yMg

Single-Cycle Control

- High-Level View

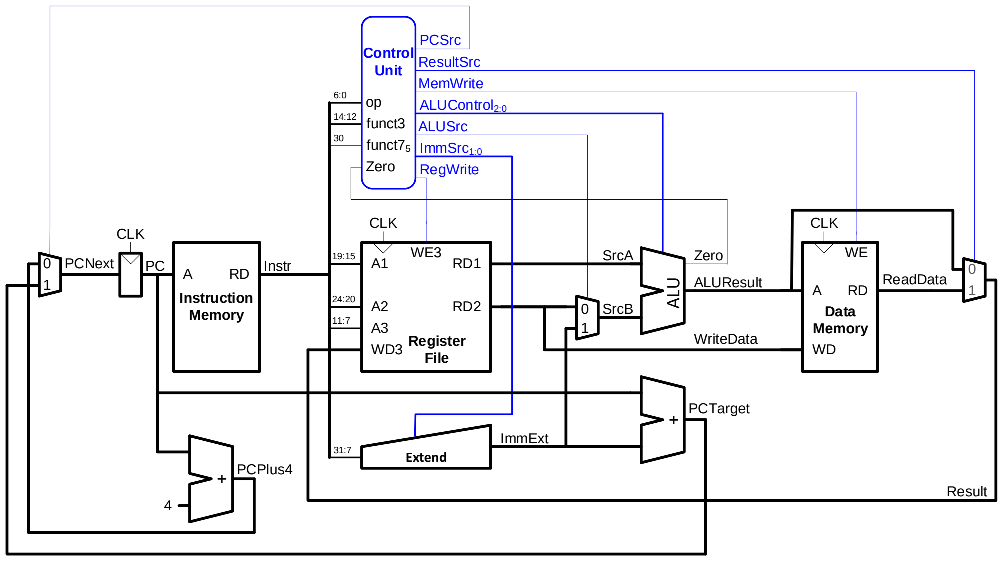


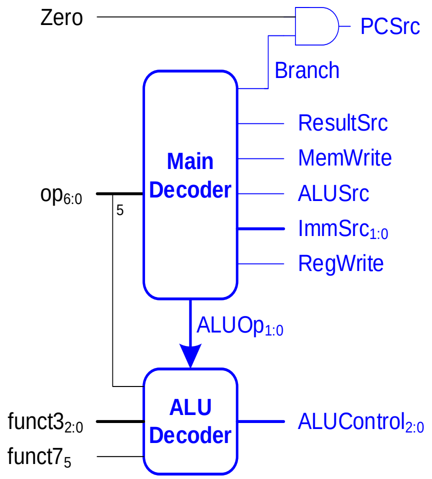
- Low-Level View



Single-Cycle Control: Main Decoder (Animation)

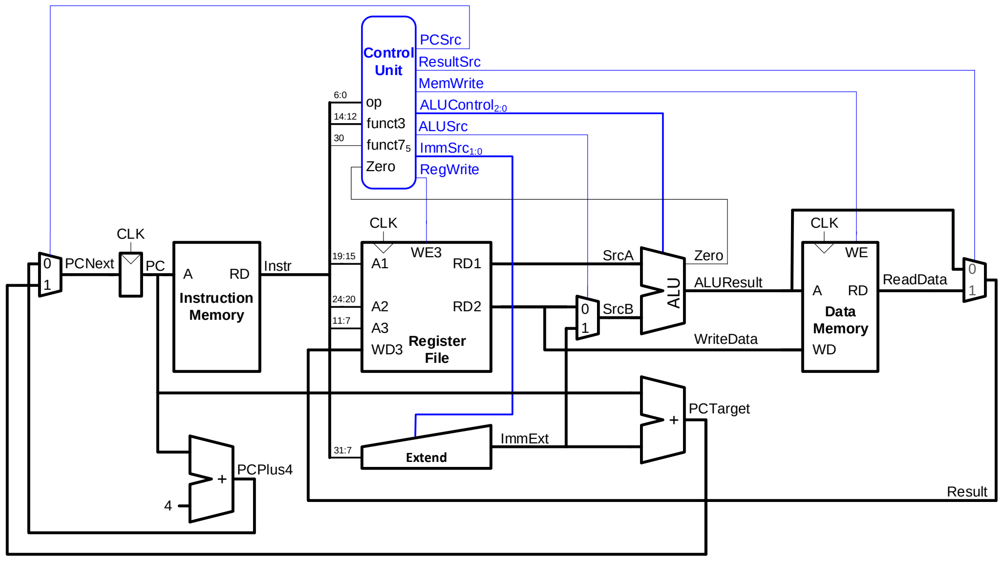
op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	`lw`							
35	`sw`							
51	R-type							
99	`beq`							

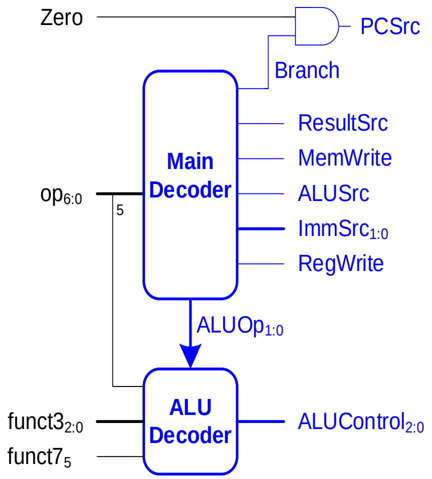




Single-Cycle Control: Main Decoder

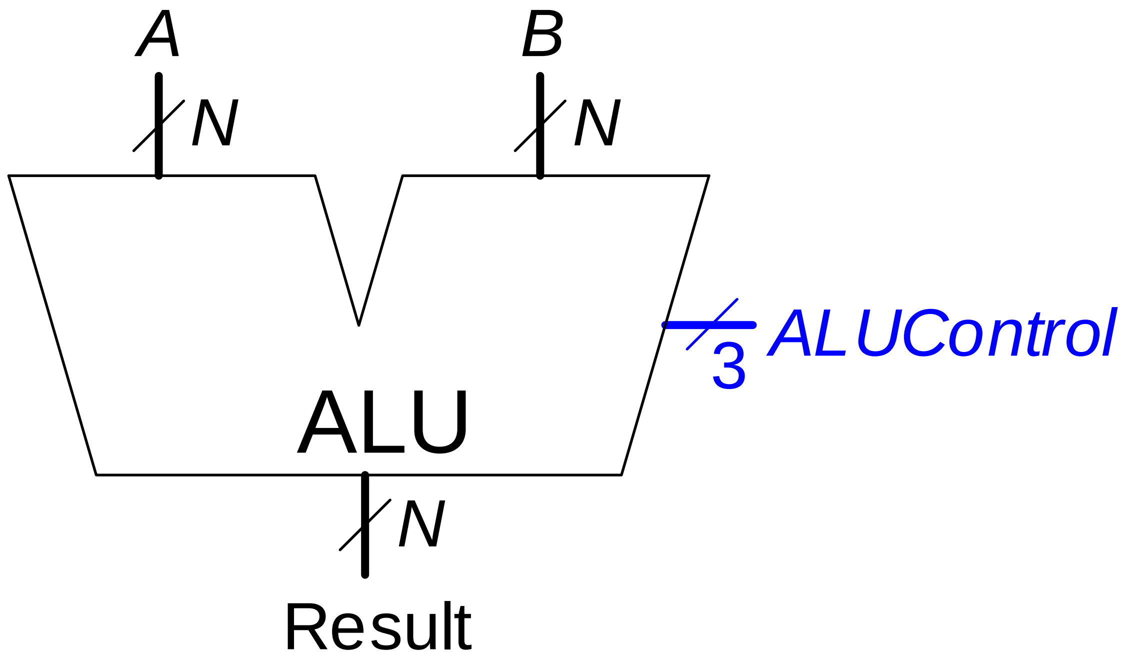
op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	`lw`	1	00	1	0	1	0	00
35	`sw`	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	`beq`	0	10	0	0	X	1	01





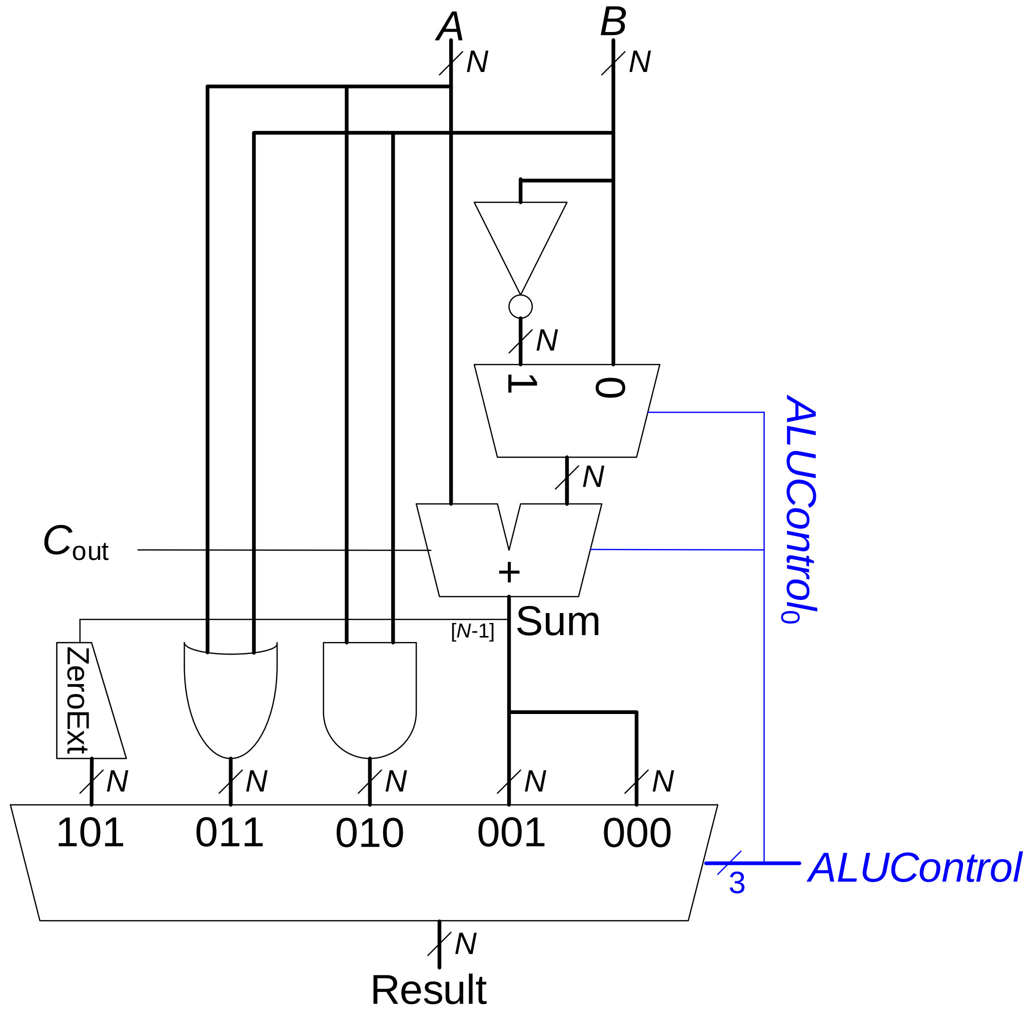
Review: ALU

$\text{ALUControl}_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT



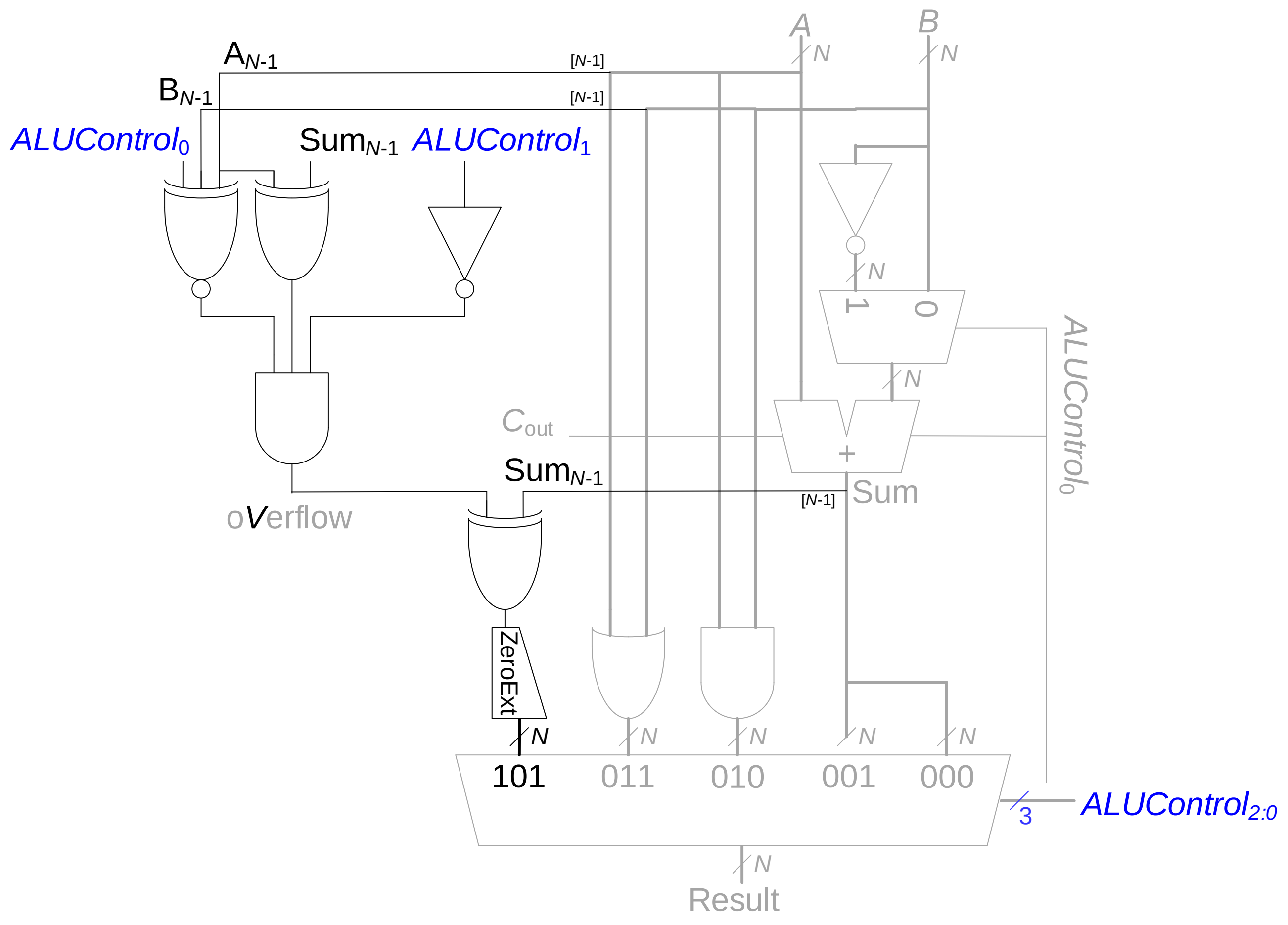
Review: ALU

$\text{ALUControl}_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT

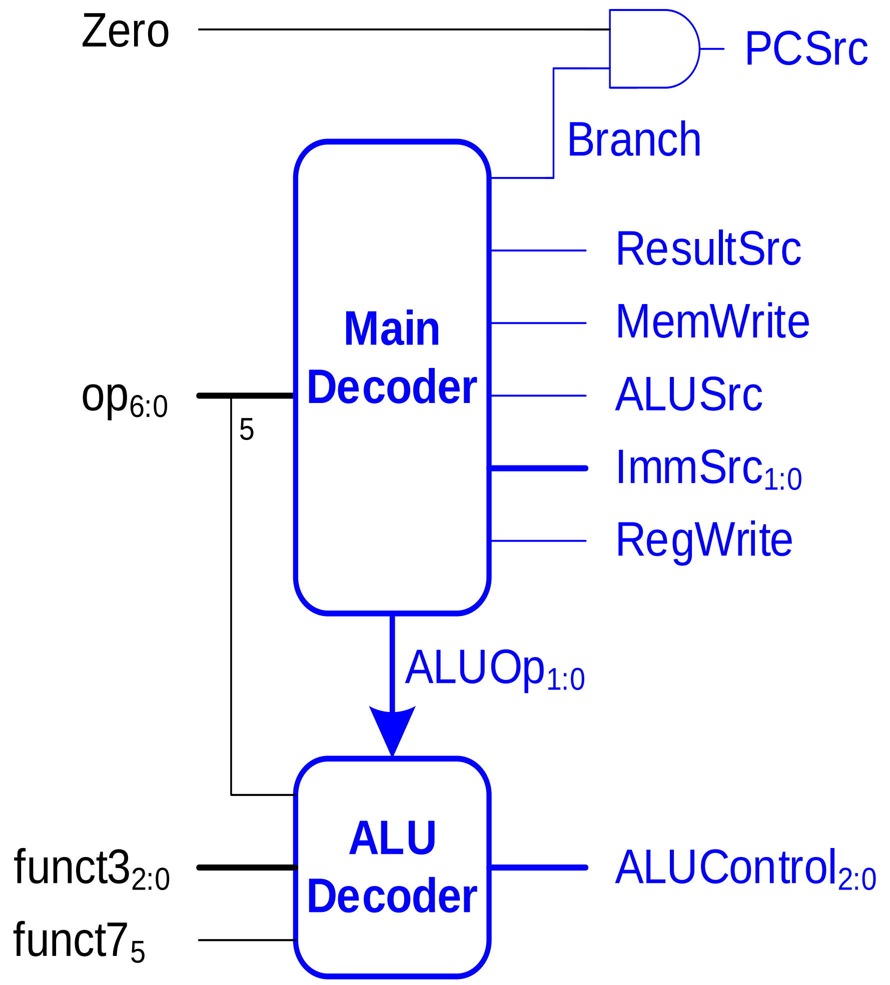


Review: ALU

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT

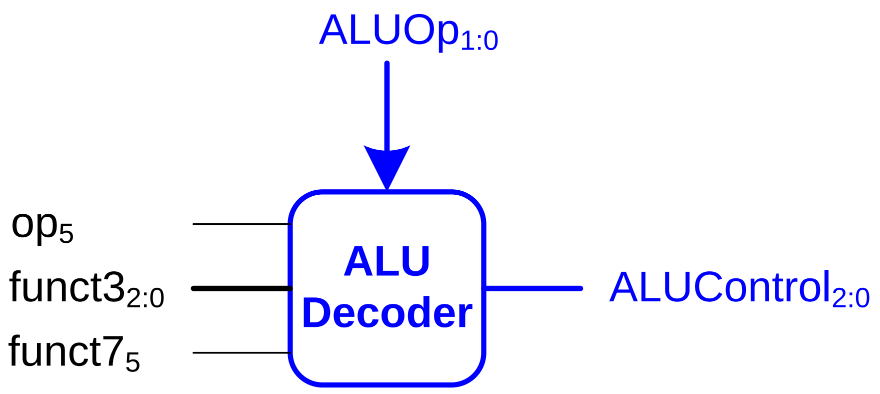


Single-Cycle Control: ALU Decoder



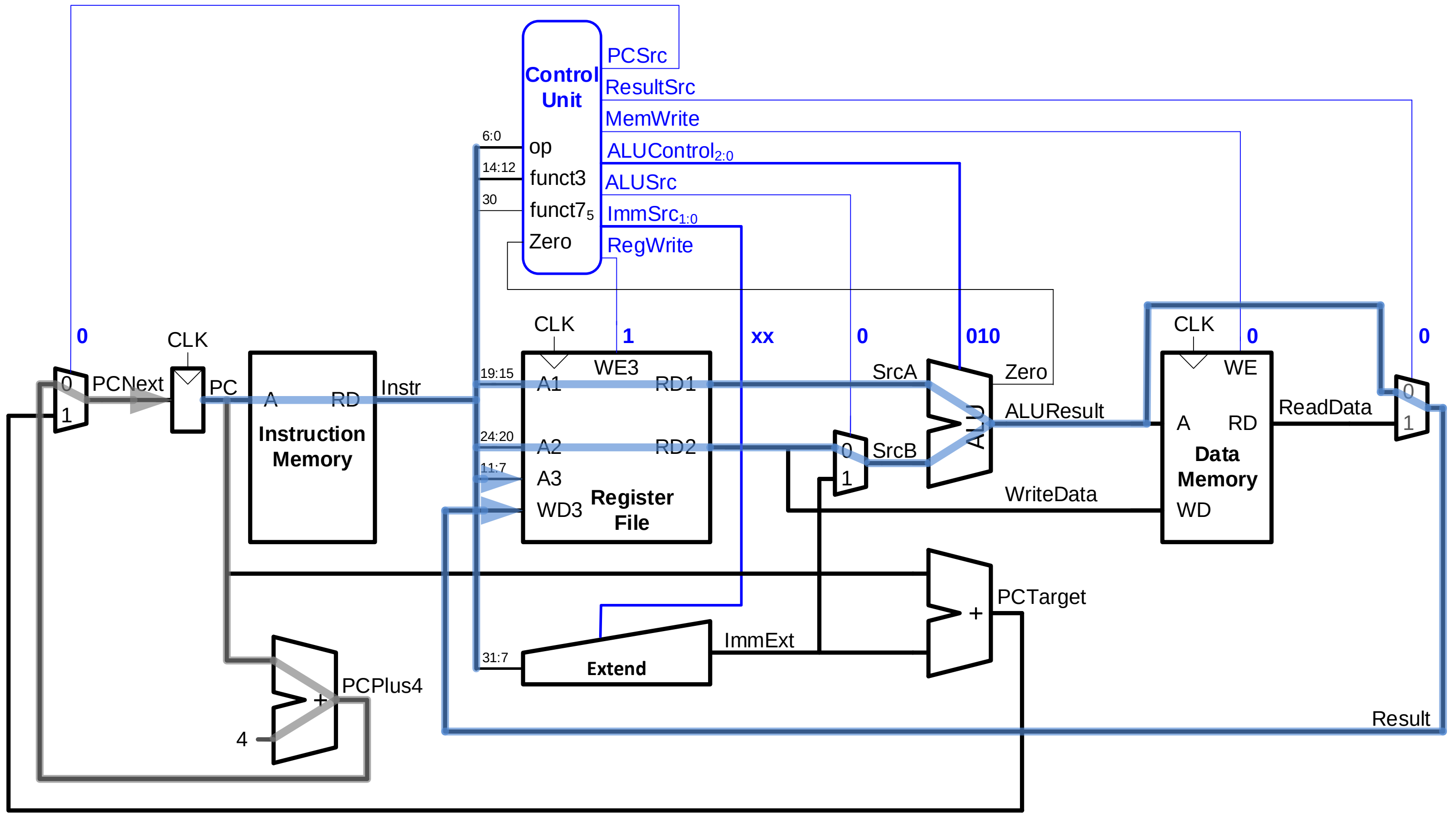
Single-Cycle Control: ALU Decoder

ALUOp	funct3	op_5, $funct7_5$	Instruction	$ALUControl_{2:0}$
00	x	x	**`lw, sw`**	000 (add)
01	x	x	**`beq`**	001 (subtract)
10	000	00, 01, 10	**`add`**	000 (add)
	000	11	**`sub`**	001 (subtract)
	010	x	**`slt`**	101 (set less than)
	110	x	**`or`**	011 (or)
	111	x	**`and`**	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
```

Extending the Single-Cycle Processor

DDCA Ch7 - Part 5: RISC-V Single-Cycle Processor: https://youtu.be/z6qxMFgNEM4?si=QTFcjiic_Hq3uRfi

Extended Functionality: I-Type ALU

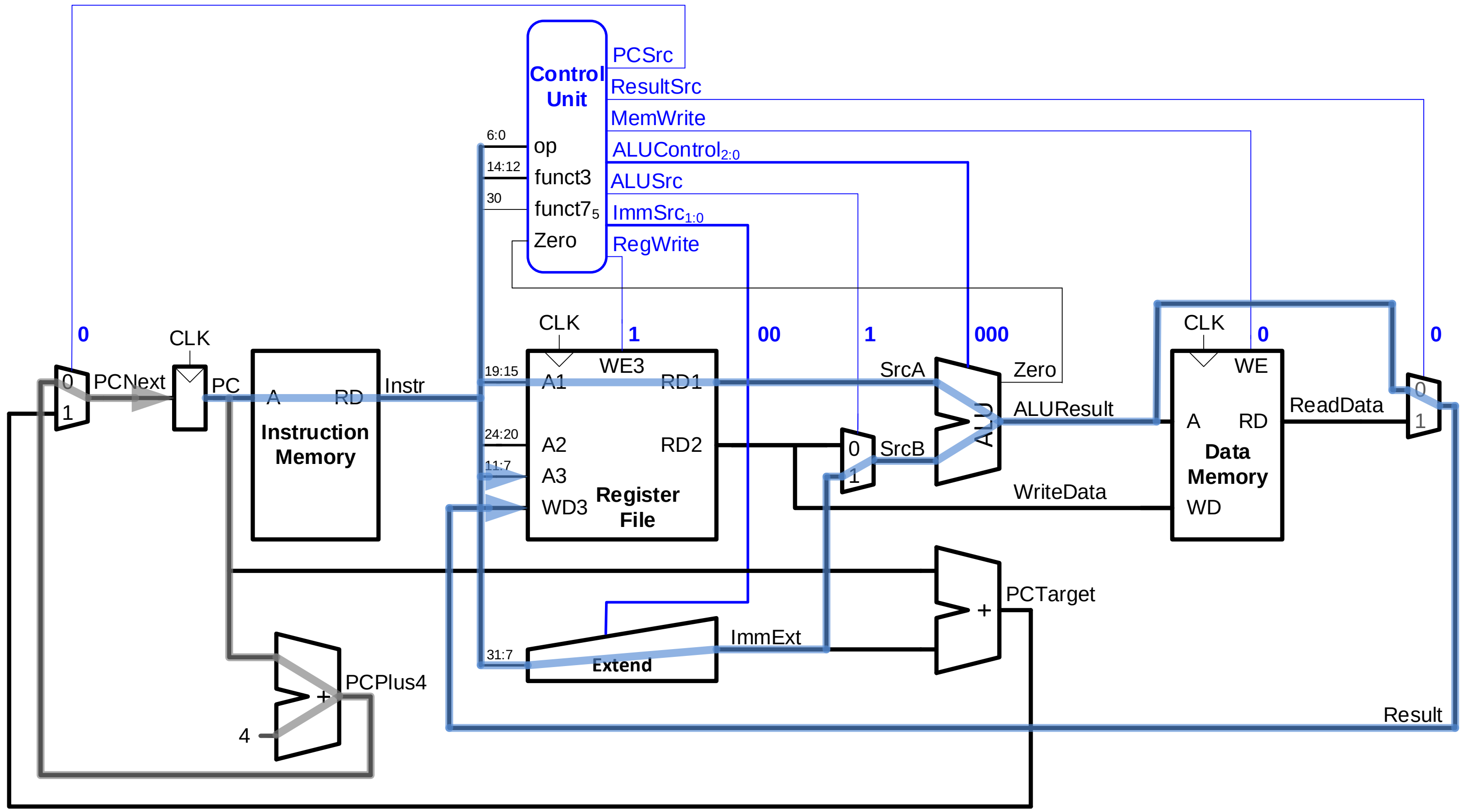
- Enhance the single-cycle processor to handle
  I-Type ALU instructions:
  `addi, andi, ori, and slti`
- **Similar to R-type** instructions
- But **second source** comes from **immediate**
- Change **ALUSrc** to select the immediate
- And **ImmSrc** to pick the correct immediate

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

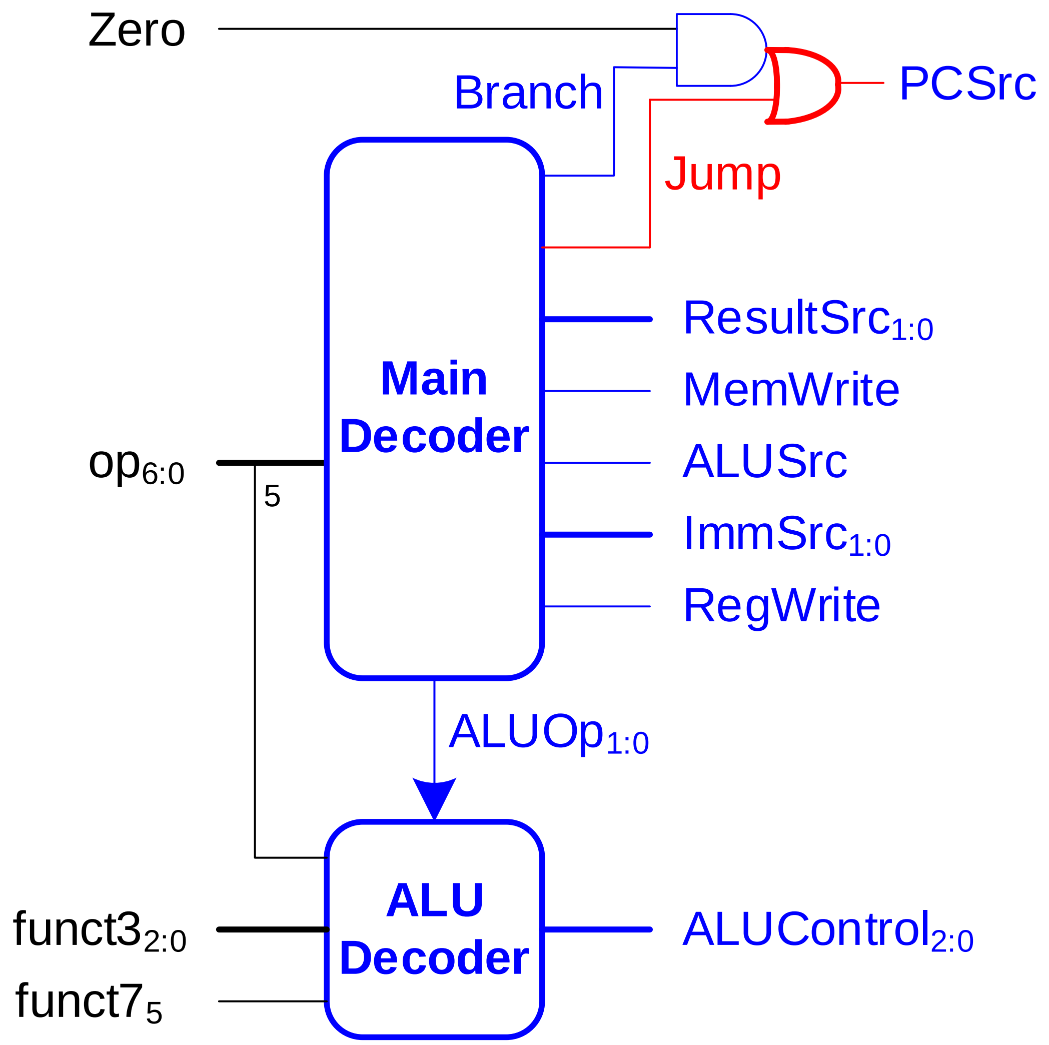
Extended Functionality: `addi`

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



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: ImmExt

$ImmSrc_{1:0}$	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
11	{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1’b0}	J-Type

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

S-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

J-Type

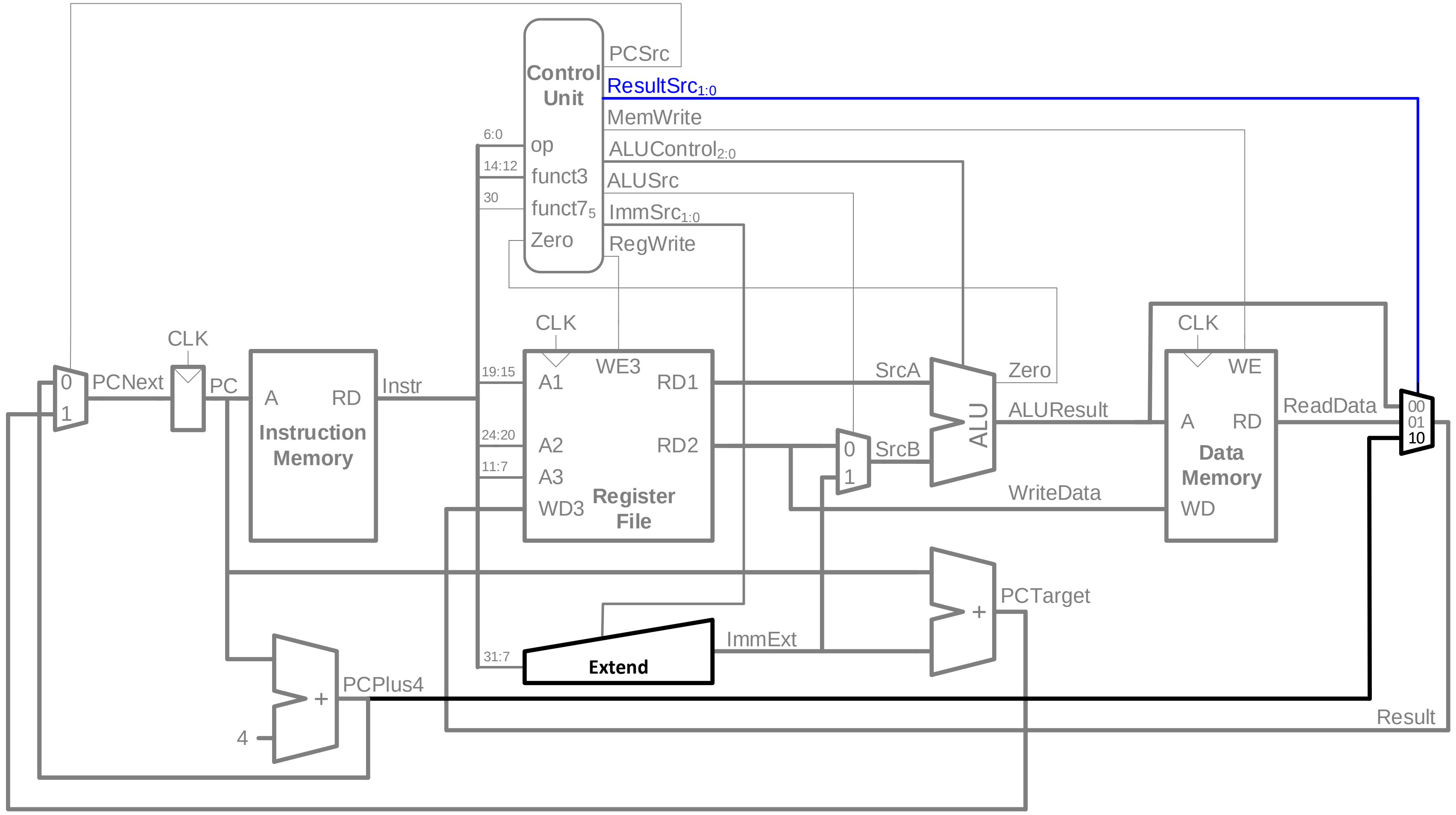
31:12	11:7	6:0
$imm_{20,10:1,11,19:12}$	rd	op
20 bits	5 bits	7 bits

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	**I-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



Single-Cycle Performance

DDCA Ch7 - Part 6: RISC-V Single-Cycle Performance https://www.youtube.com/watch?v=w82mNGranjA

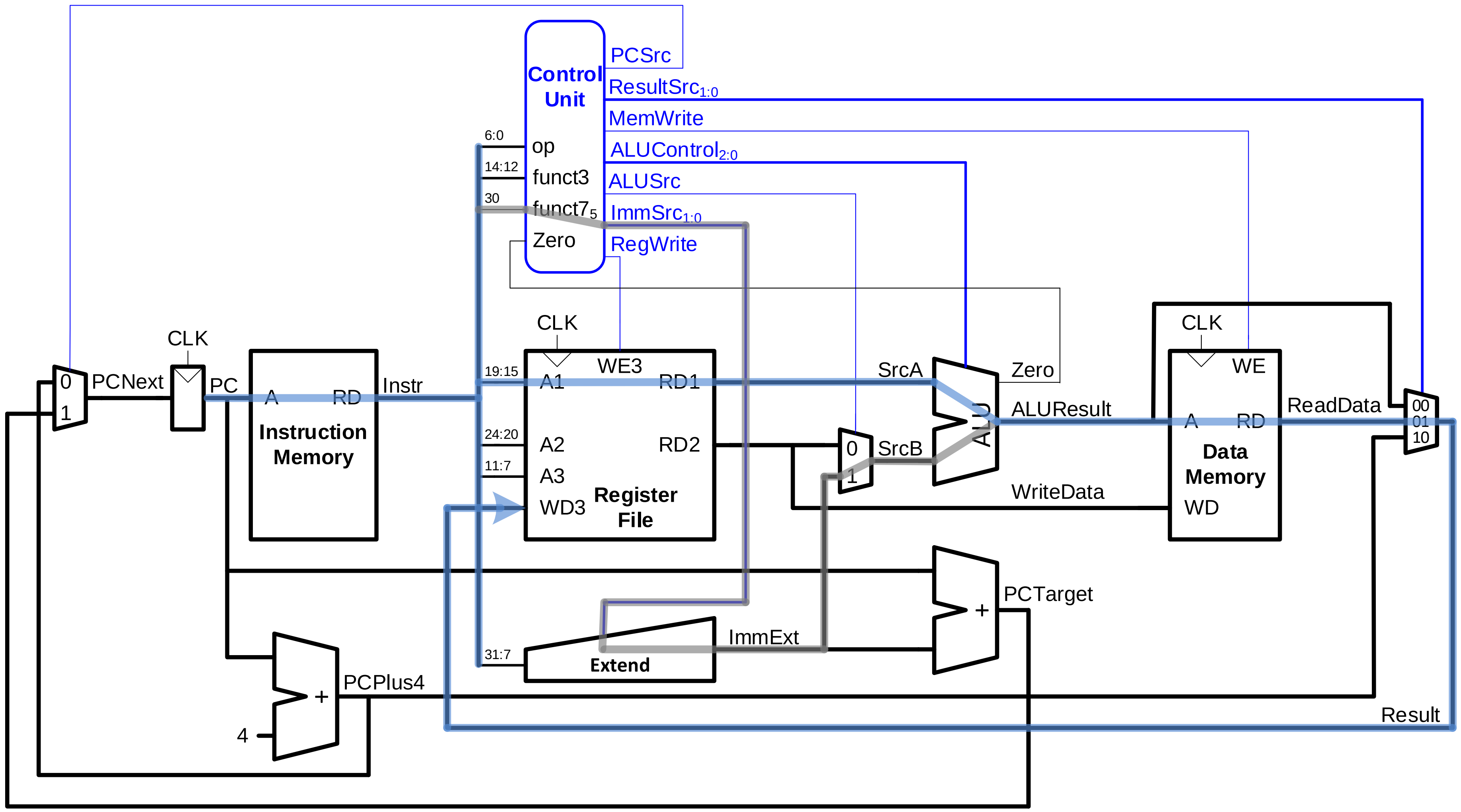
Processor Performance

Program Execution Time

$$= \#instructions \times \frac{cycles}{instruction} \times \frac{seconds}{cycle}$$

$$= \#instructions \times CPI \times T_C$$

Single-Cycle Processor Performance

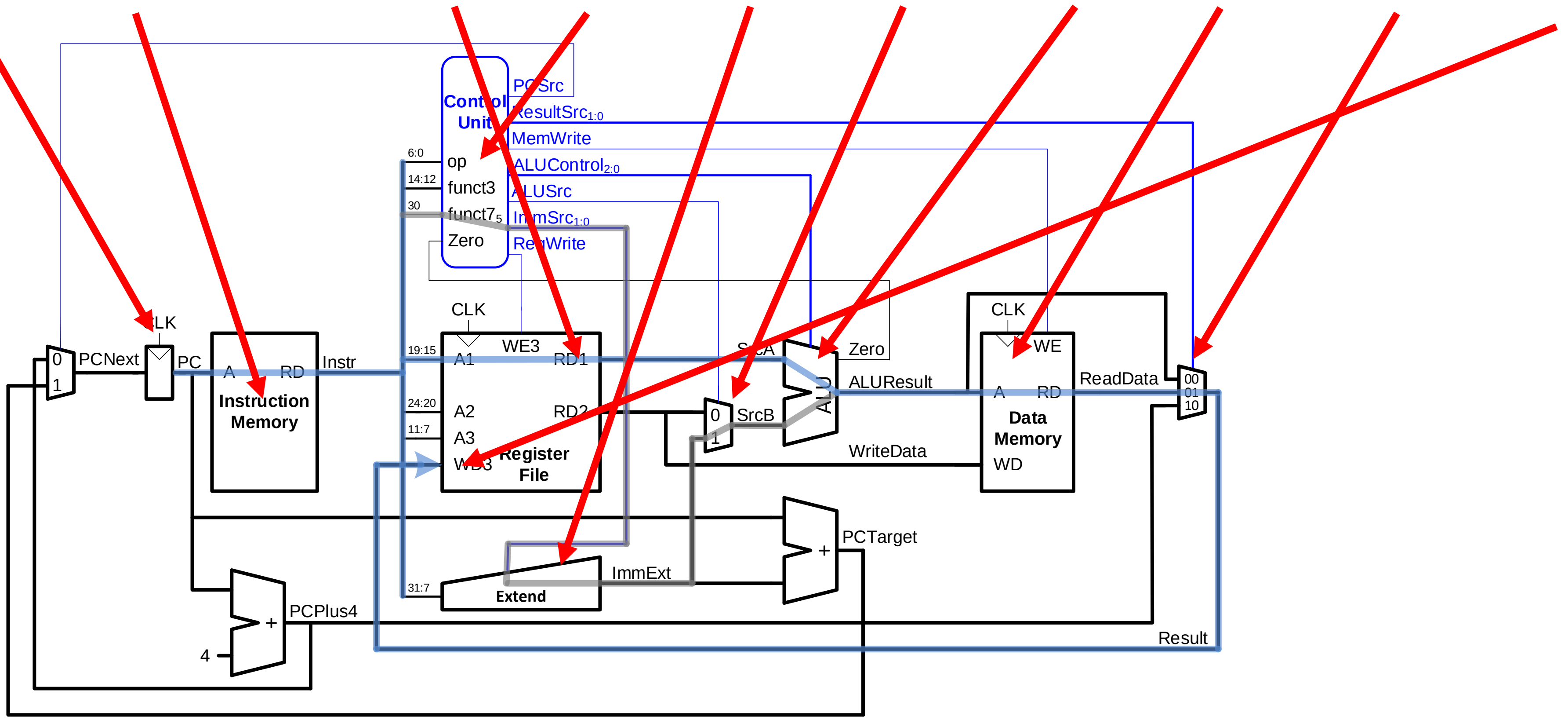


T_c limited by critical path (**lw**)

Single-Cycle Processor Performance

- **Single-cycle critical path:**

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



Single-Cycle Processor Performance

- **Single-cycle critical path:**

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

- **Typically, limiting paths are:**
  - memory, ALU, register file
  - So, $t_{dec} + t_{ext} + t_{mux}$ can be neglected

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

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

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	$t_{\mathrm{AND\text{-}OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\mathrm{read}}$	100
Register file setup	$t_{RF\mathrm{setup}}$	60

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

= (40 + 2*200 + 100 + 120 + 30 + 60) ps = **750 ps**

Single-Cycle Performance Example

Program with 100 billion instructions = 10^{11} instructions :

Execution Time = $\#instructions \times CPI \times T_C$

= $(100 \times 10^9)(1)(750 \times 10^{-12}\text{ s})$

= **75 seconds**

Multicycle RISC-V Processor

DDCA Ch7 - Part 7: Multicycle Processor https://www.youtube.com/watch?v=sATaQNCC0-g

Single- vs. Multicycle Processor

- **Single-cycle**:
  - **+ simple**
  - - cycle time limited by longest instruction (`lw`)
  - - 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

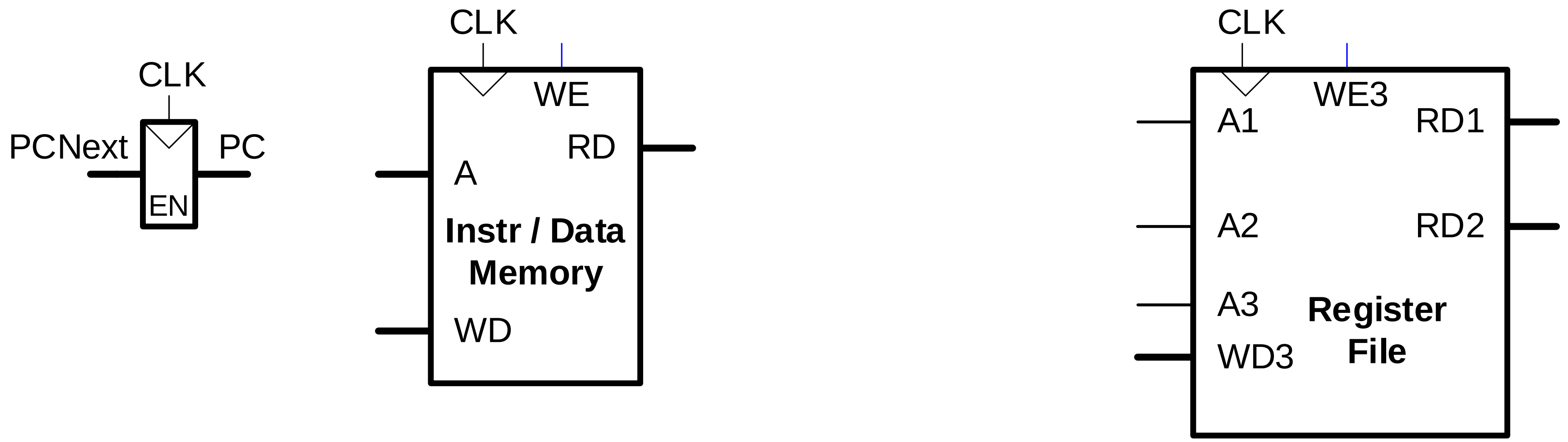
Single- vs. Multicycle Processor

- **Single-cycle**:
    - **+ simple**
    - - cycle time limited by longest instruction (`lw`)
    - - 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**

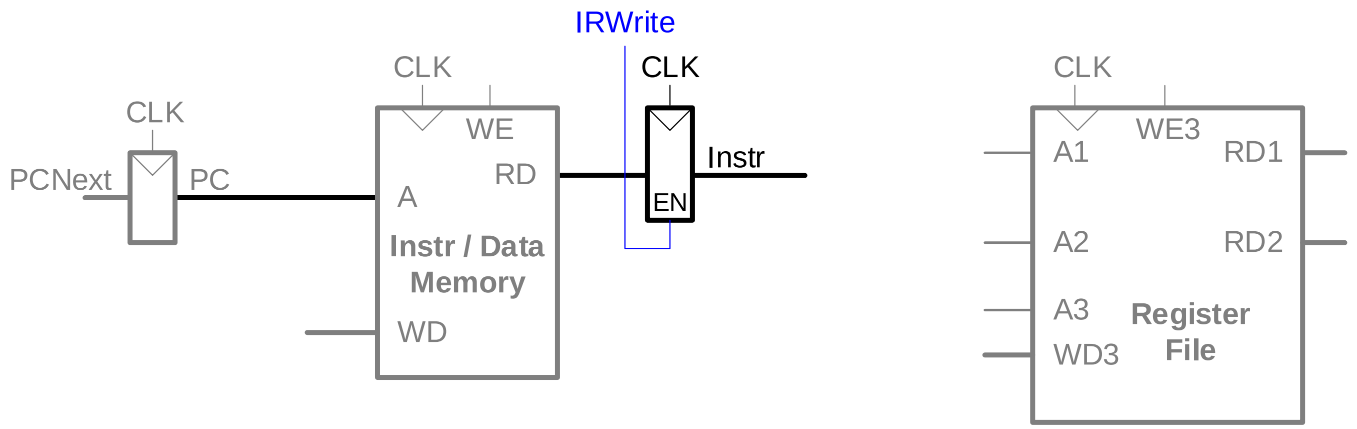
Multicycle State Elements

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



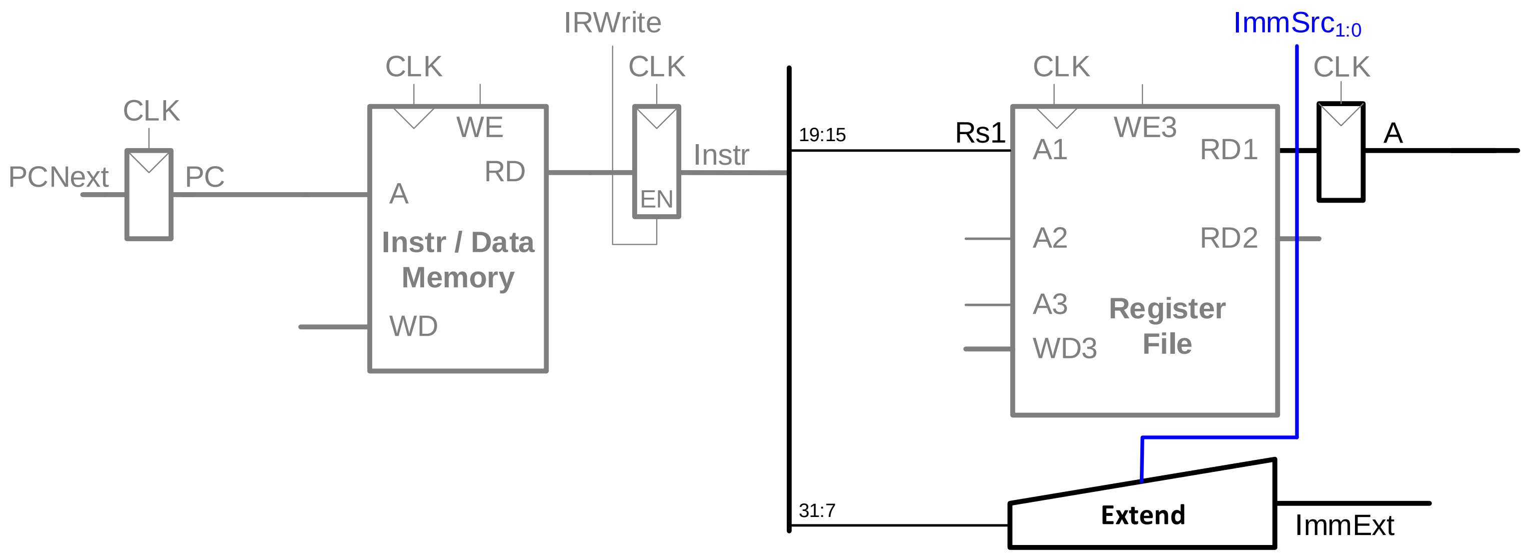
Multicycle Datapath: Instruction Fetch

STEP 1: Fetch instruction



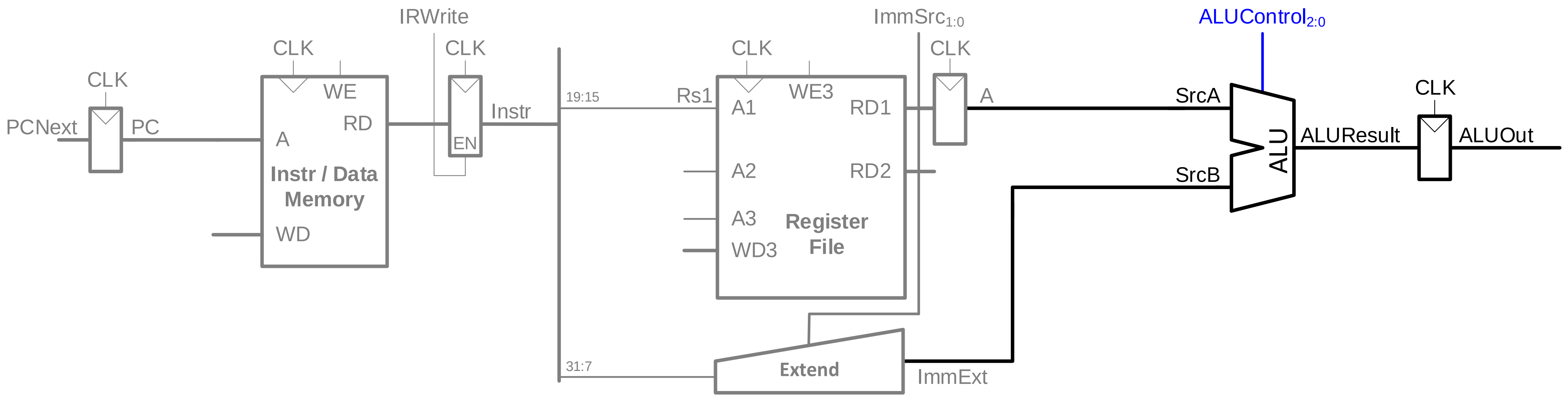
Multicycle Datapath: `lw` Get Sources

STEP 2: Read source operand from RF and extend immediate



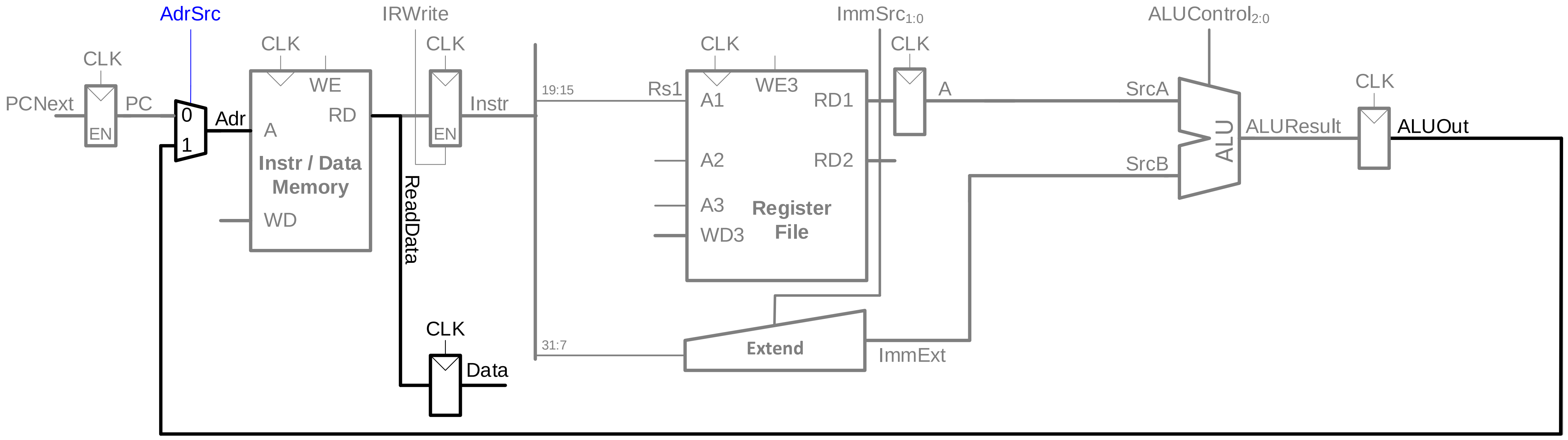
Multicycle Datapath: `lw` Address

STEP 3: Compute the memory address



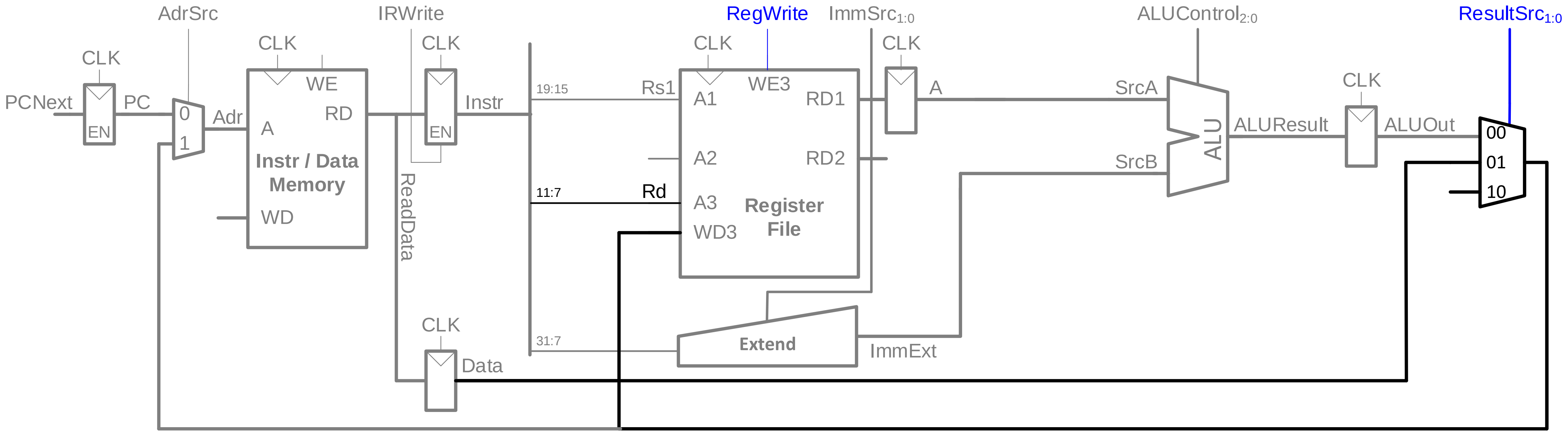
Multicycle Datapath: `lw` Memory Read

STEP 4: Read data from memory



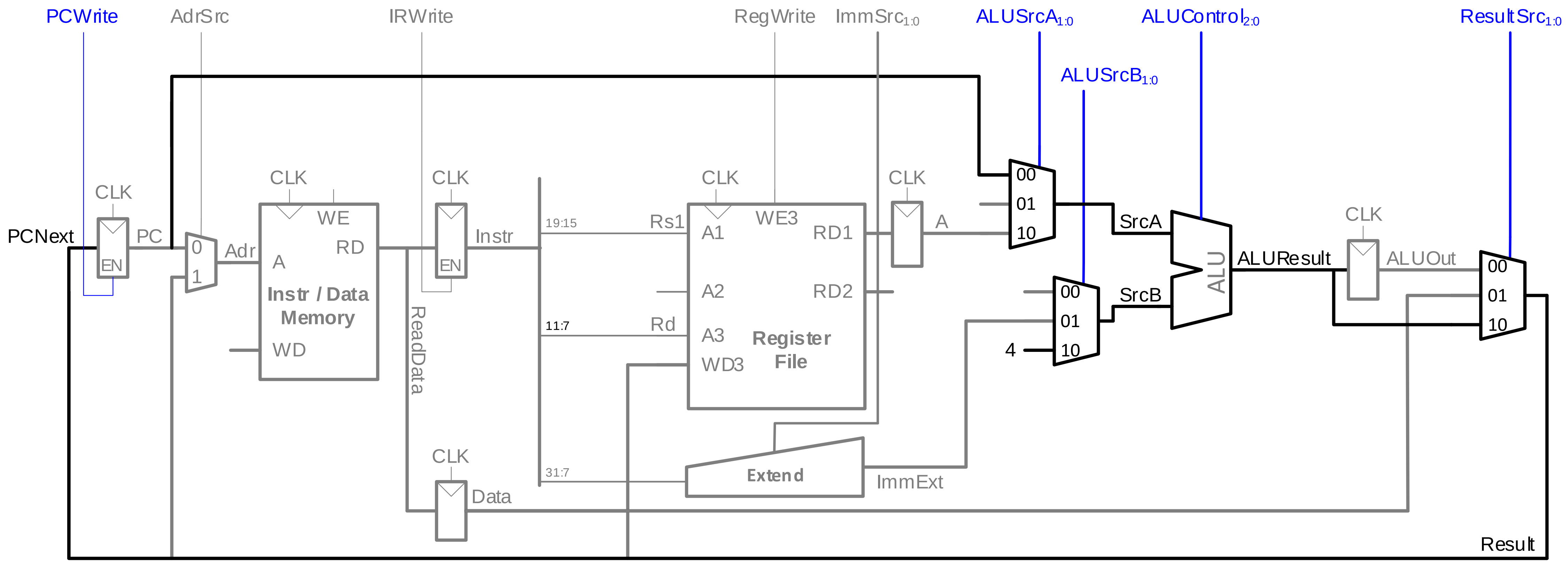
Multicycle Datapath: `lw` Write Register

STEP 5: Write data back to register file



Multicycle Datapath: Increment PC

STEP 6: Increment PC: PC = PC+4

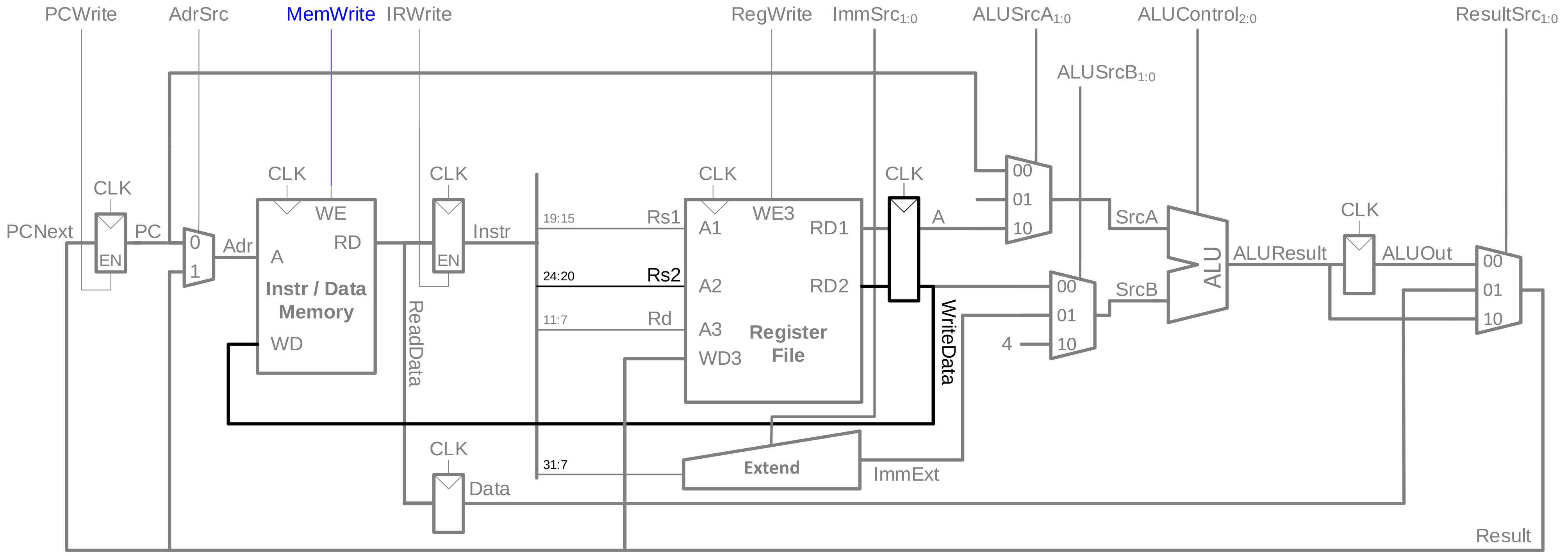


Multicycle Datapath: Other Instructions

DDCA Ch7 - part 8: RISC-V Multicycle Processor - Other Instructions https://www.youtube.com/watch?v=dnITBQQDmwU

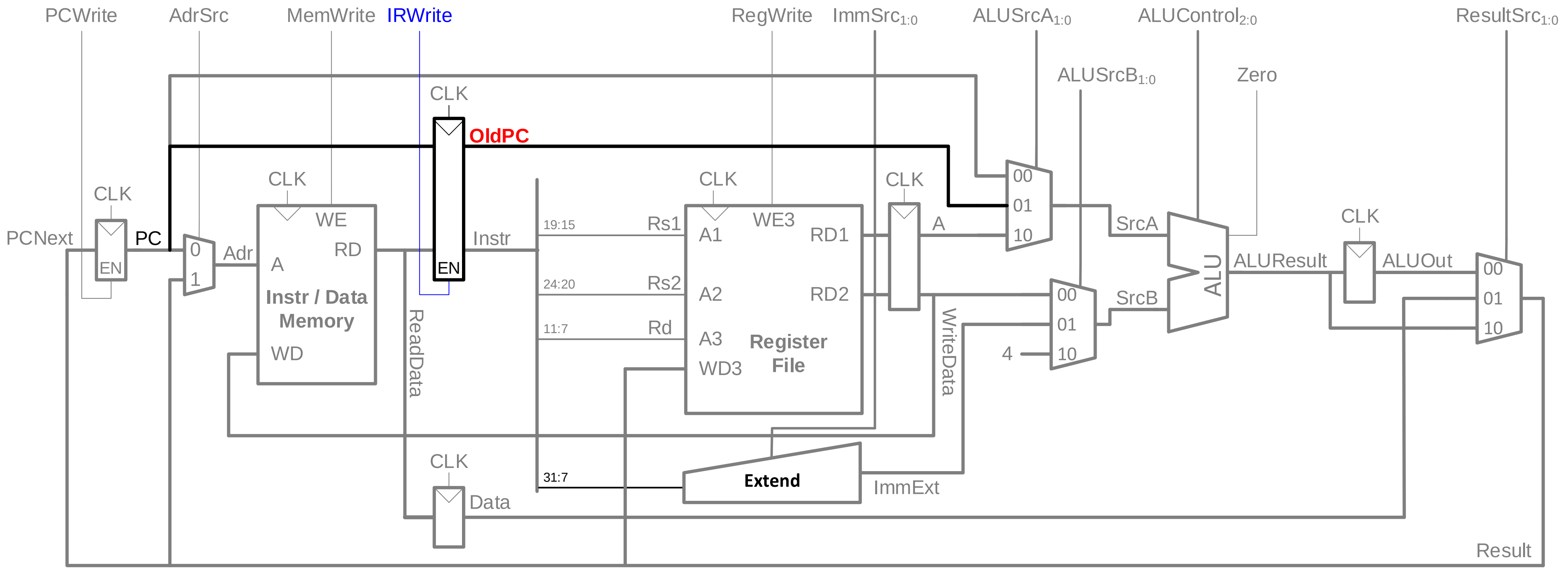
Multicycle Datapath: `sw`

Write data from Register File (**rs2**) to memory



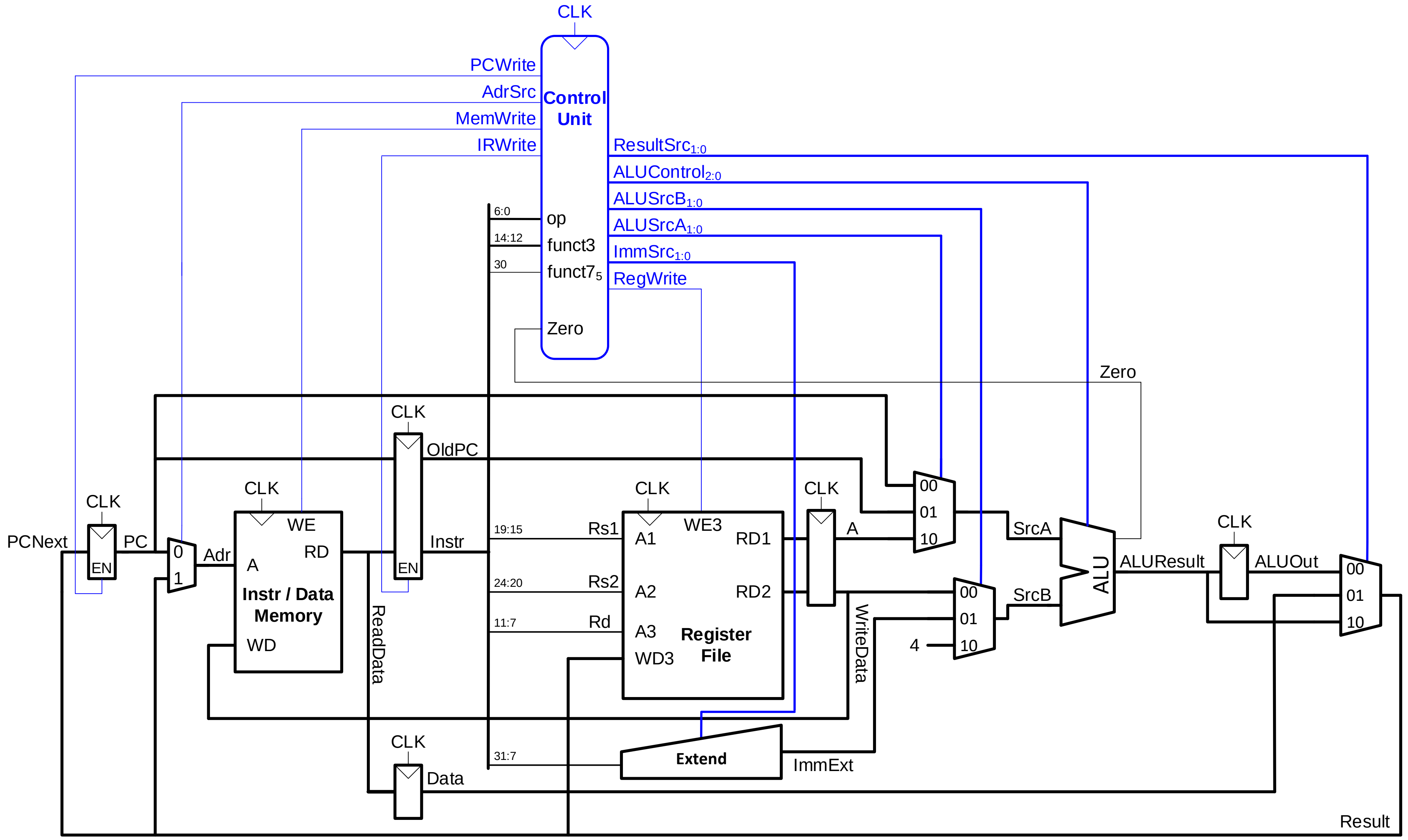
Multicycle Datapath: beq

- Calculate branch target address: BTA = PC + imm



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

Multicycle RISC-V Processor

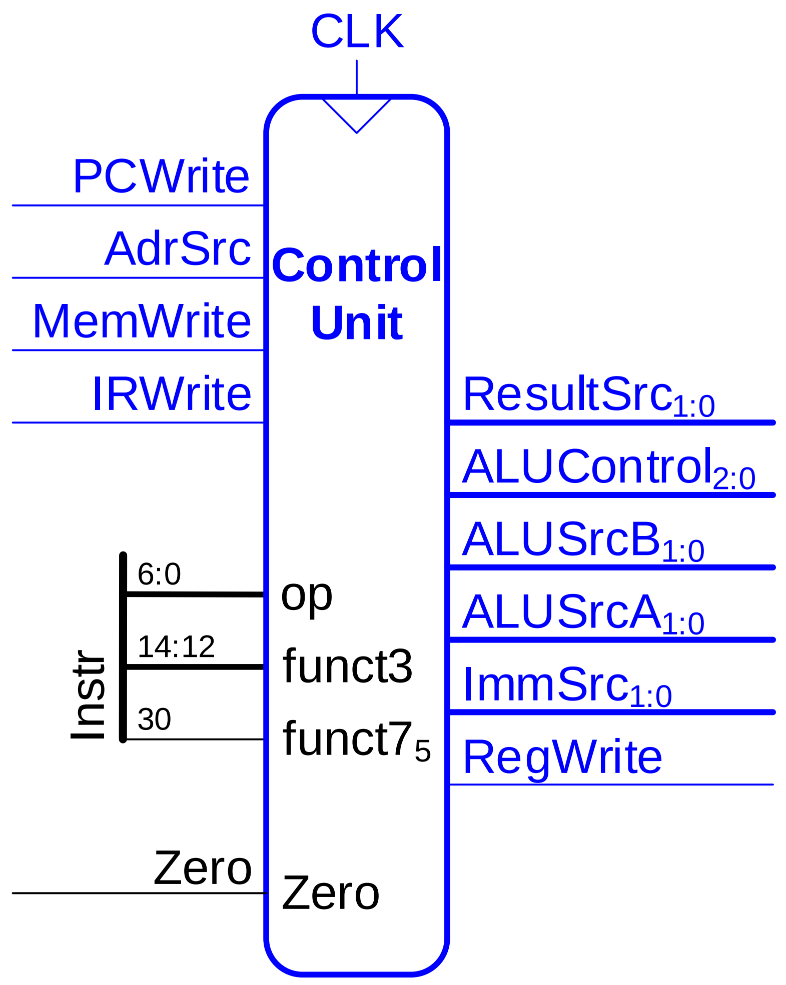


Multicycle Control

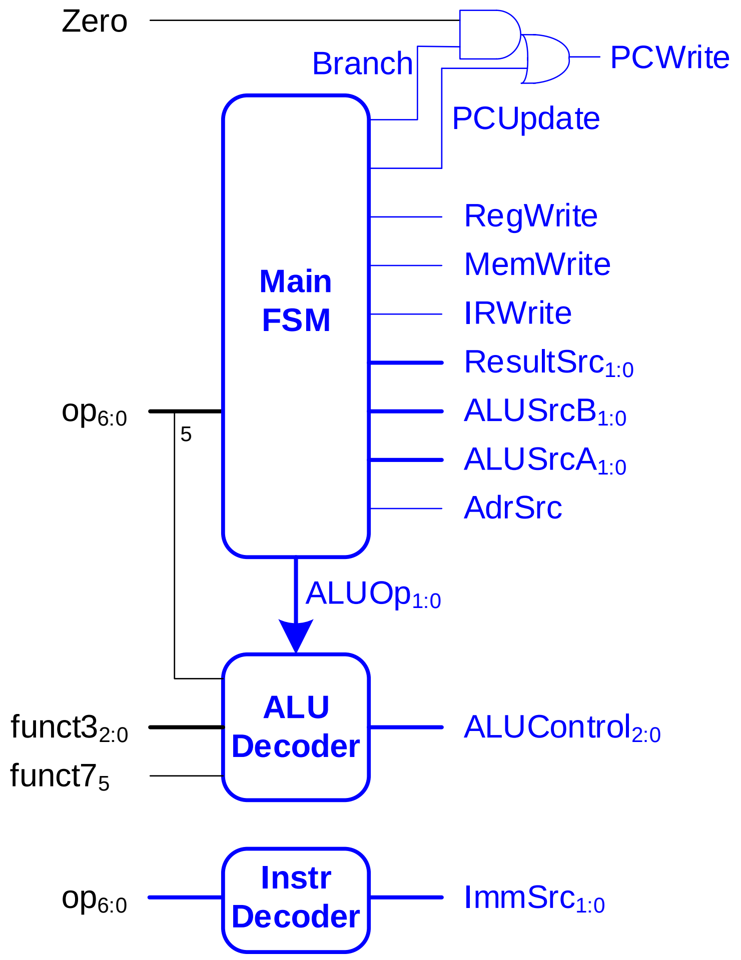
DDCA Ch7 - Part 9: RISC-V Multicycle Processor Control https://www.youtube.com/watch?v=YUJhNTpunqI

Multicycle Control

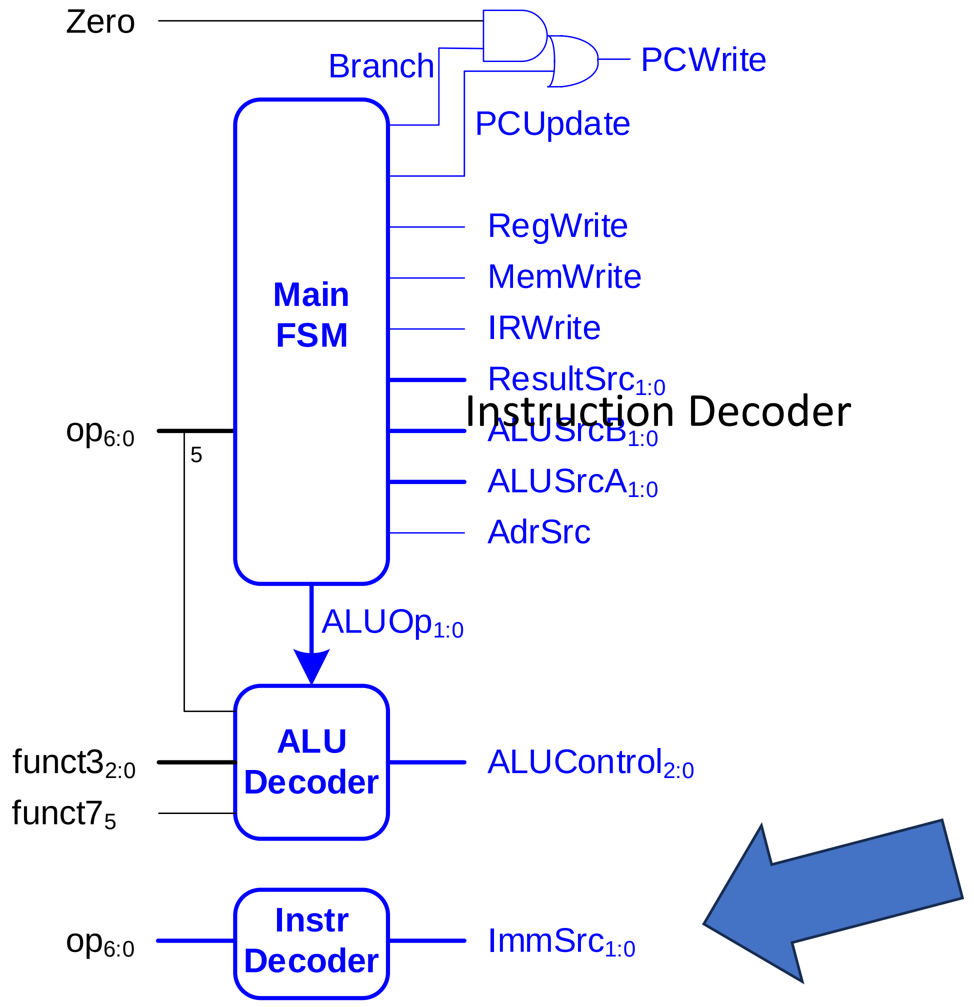
High-Level View



Low-Level View



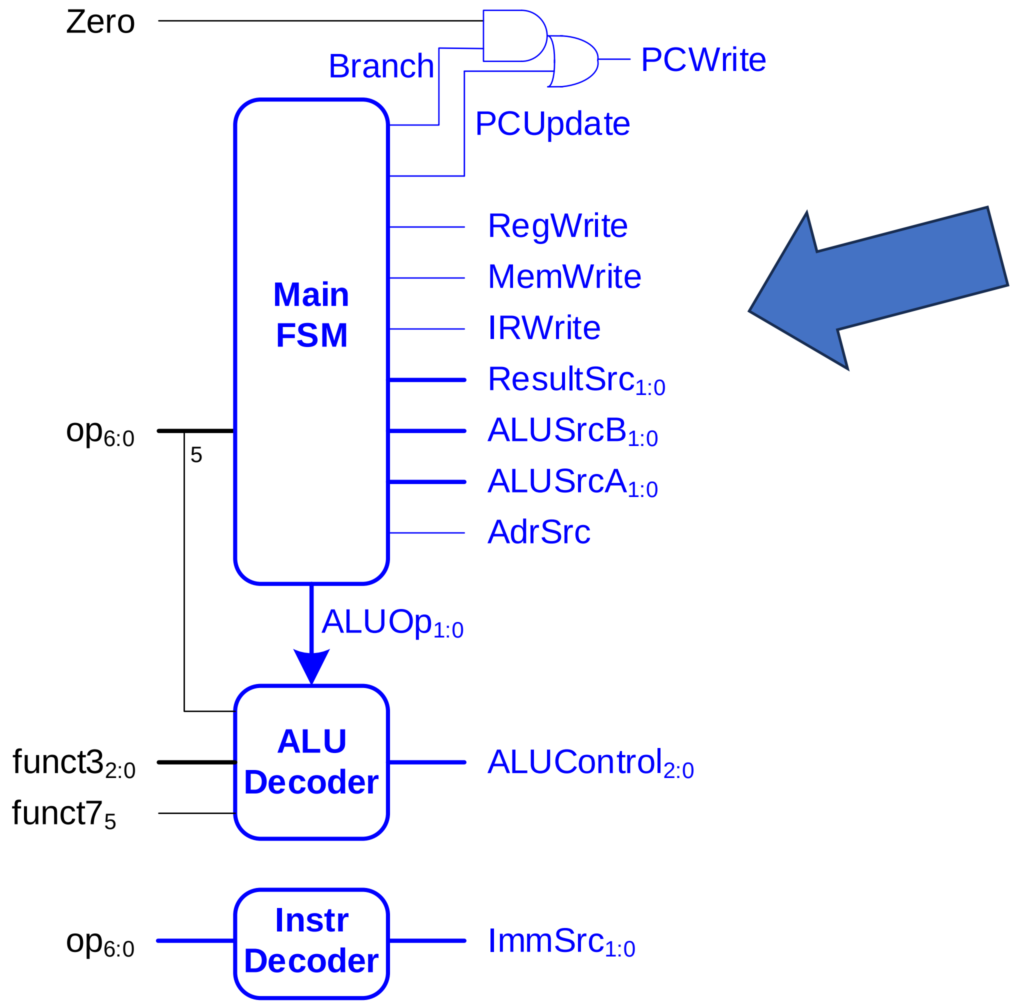
Multicycle Control: Instruction Decoder



Instruction Decoder

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

Multicycle Control: Main FSM

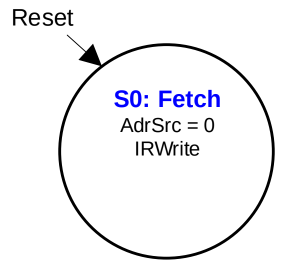


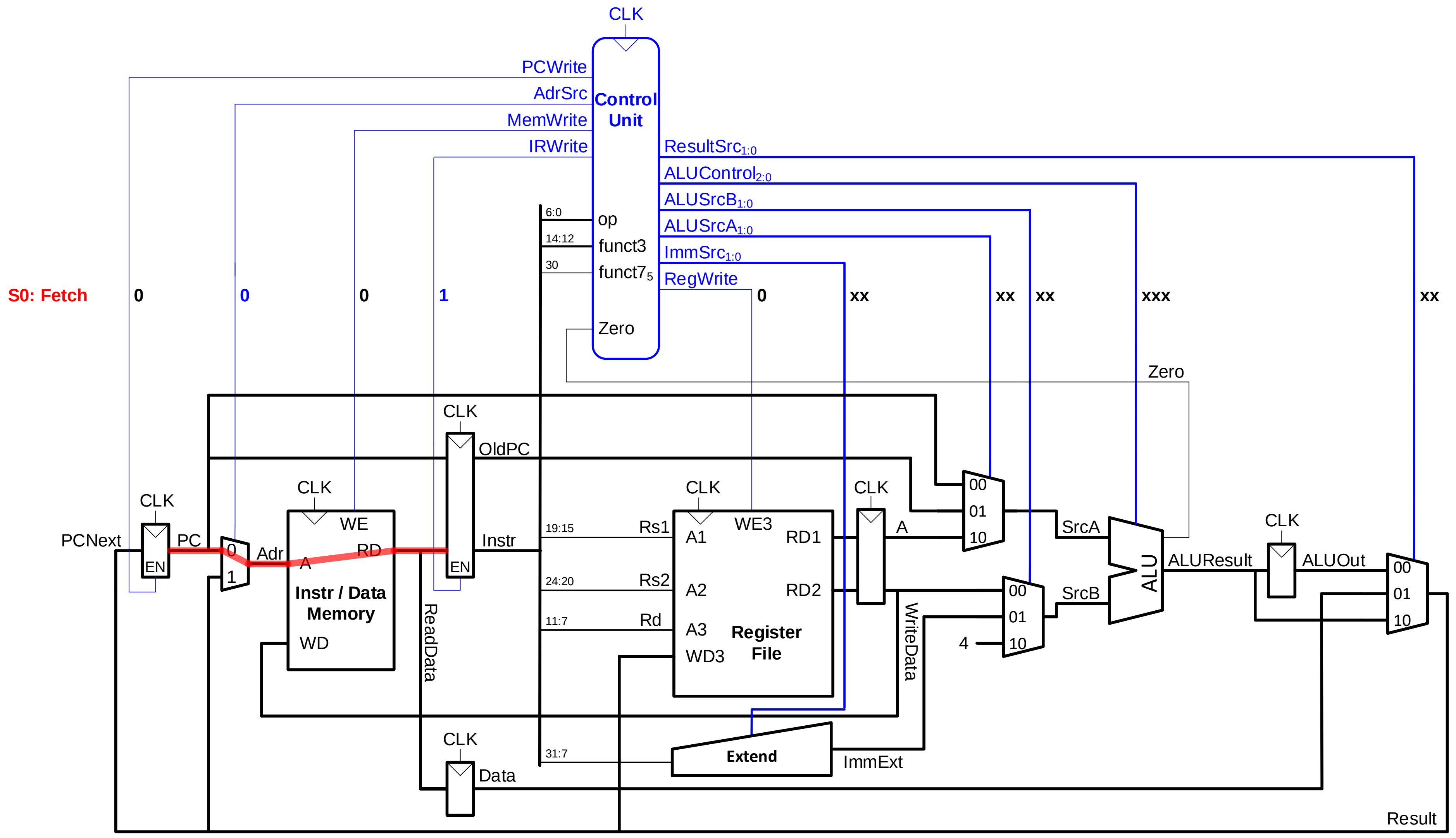
Main FSM

To declutter FSM:

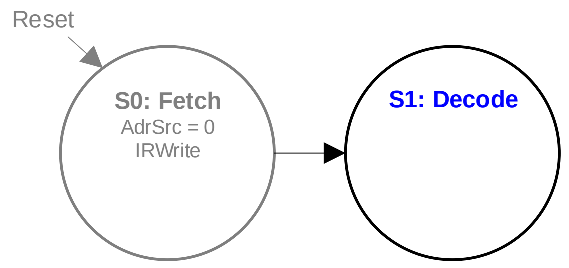
- **Write enable signals** (RegWrite, MemWrite, IRWrite, PCUpdate, and Branch) are **0** if not listed in a state.
- **Other signals are don't care** if not listed in a state

Main FSM: Fetch

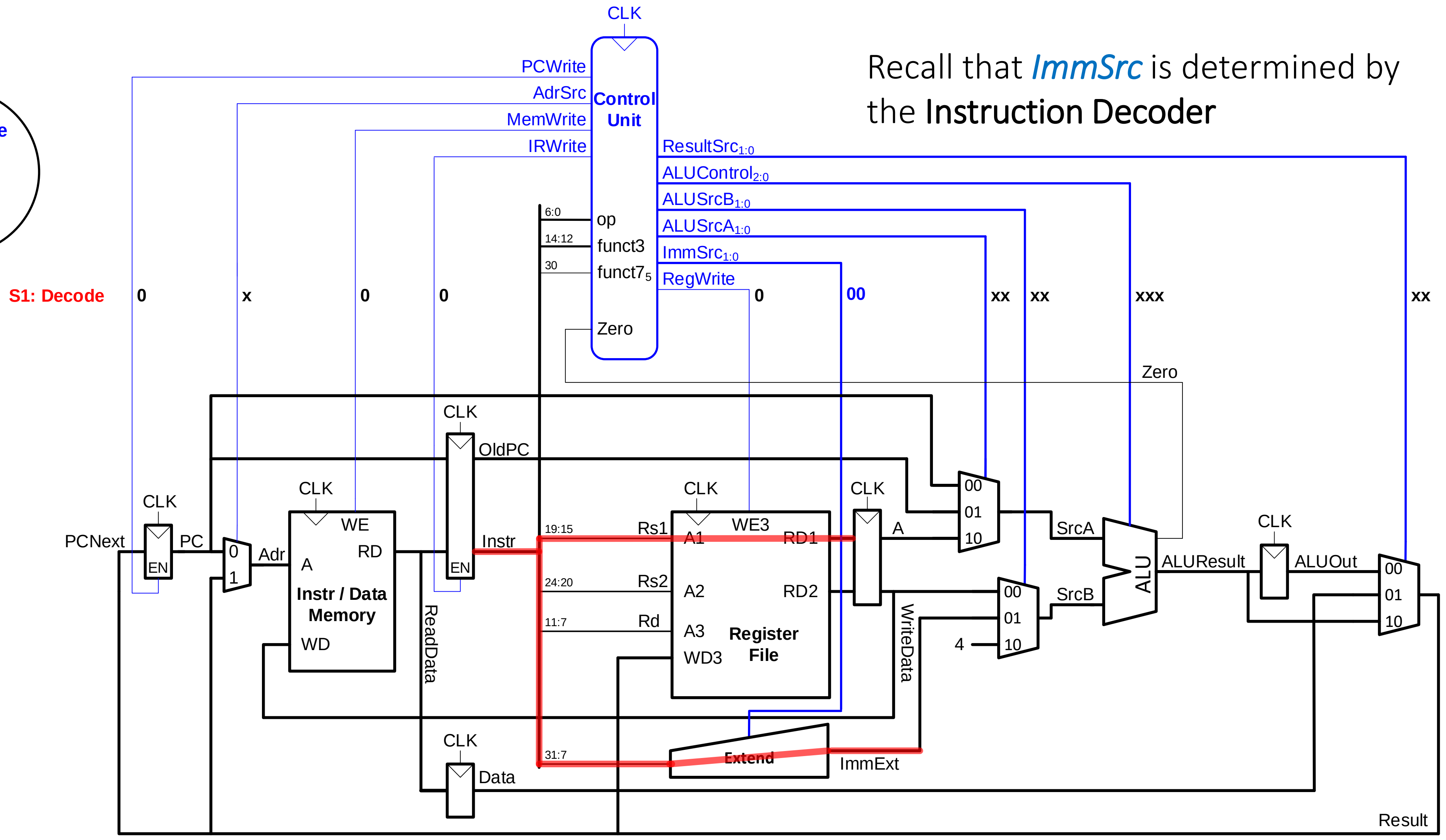




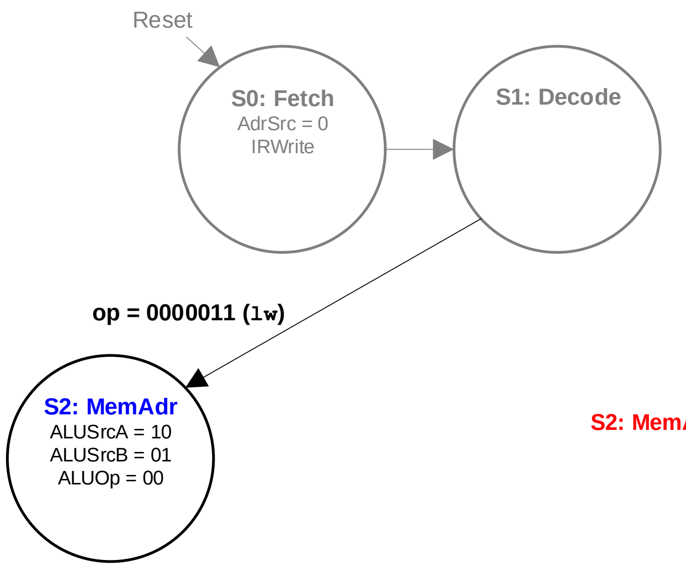
Main FSM: Decode

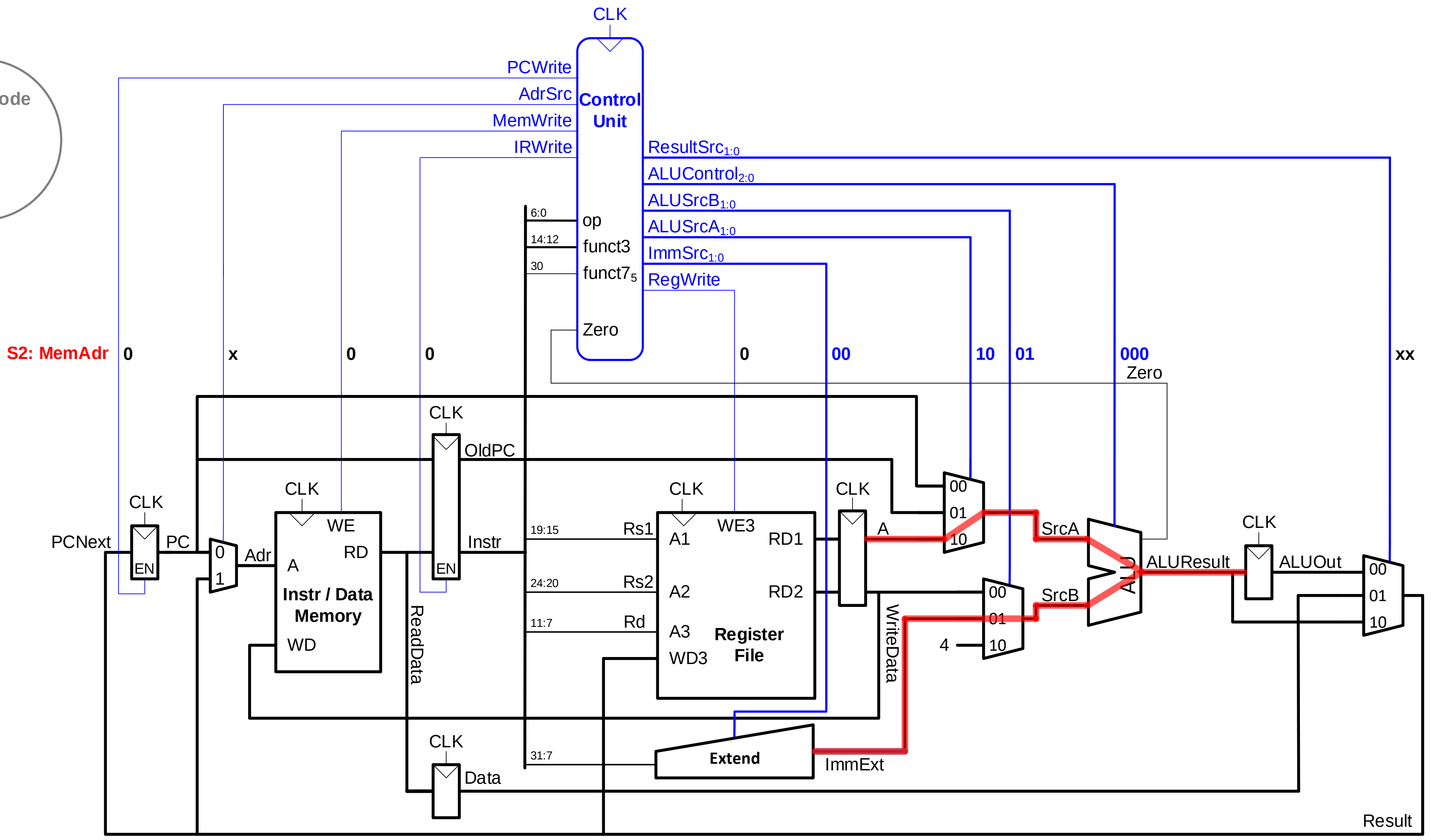


Recall that *ImmSrc* is determined by the **Instruction Decoder**

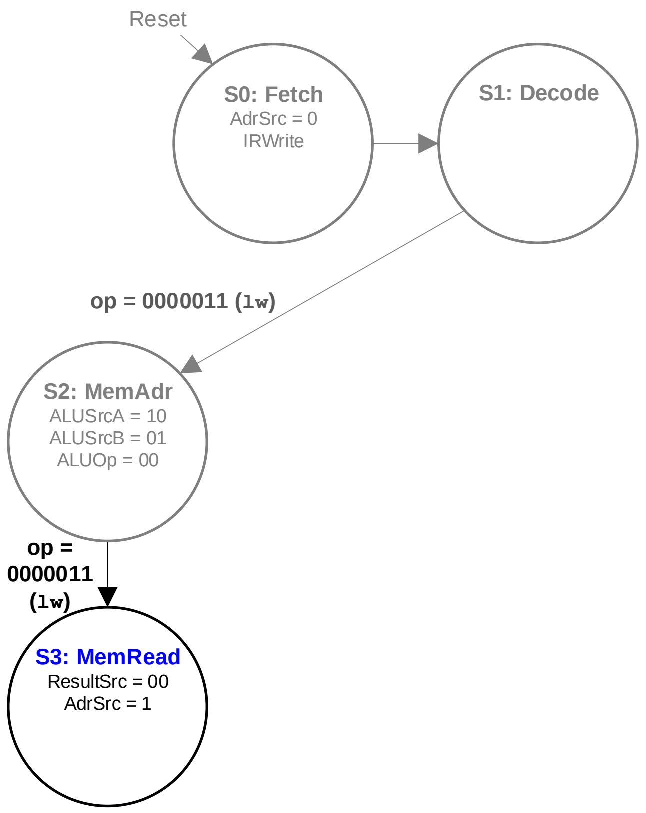


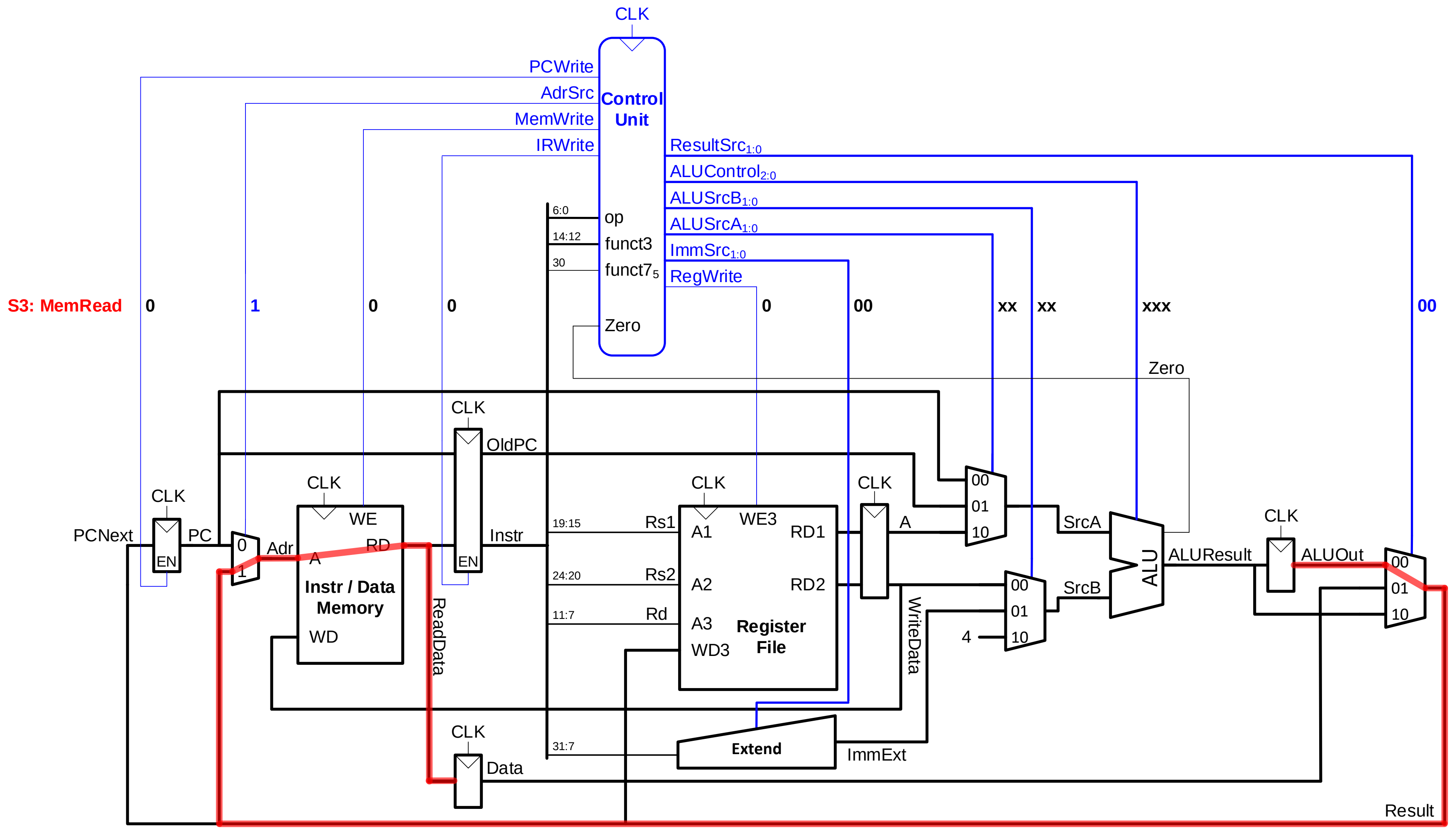
Main FSM: Address



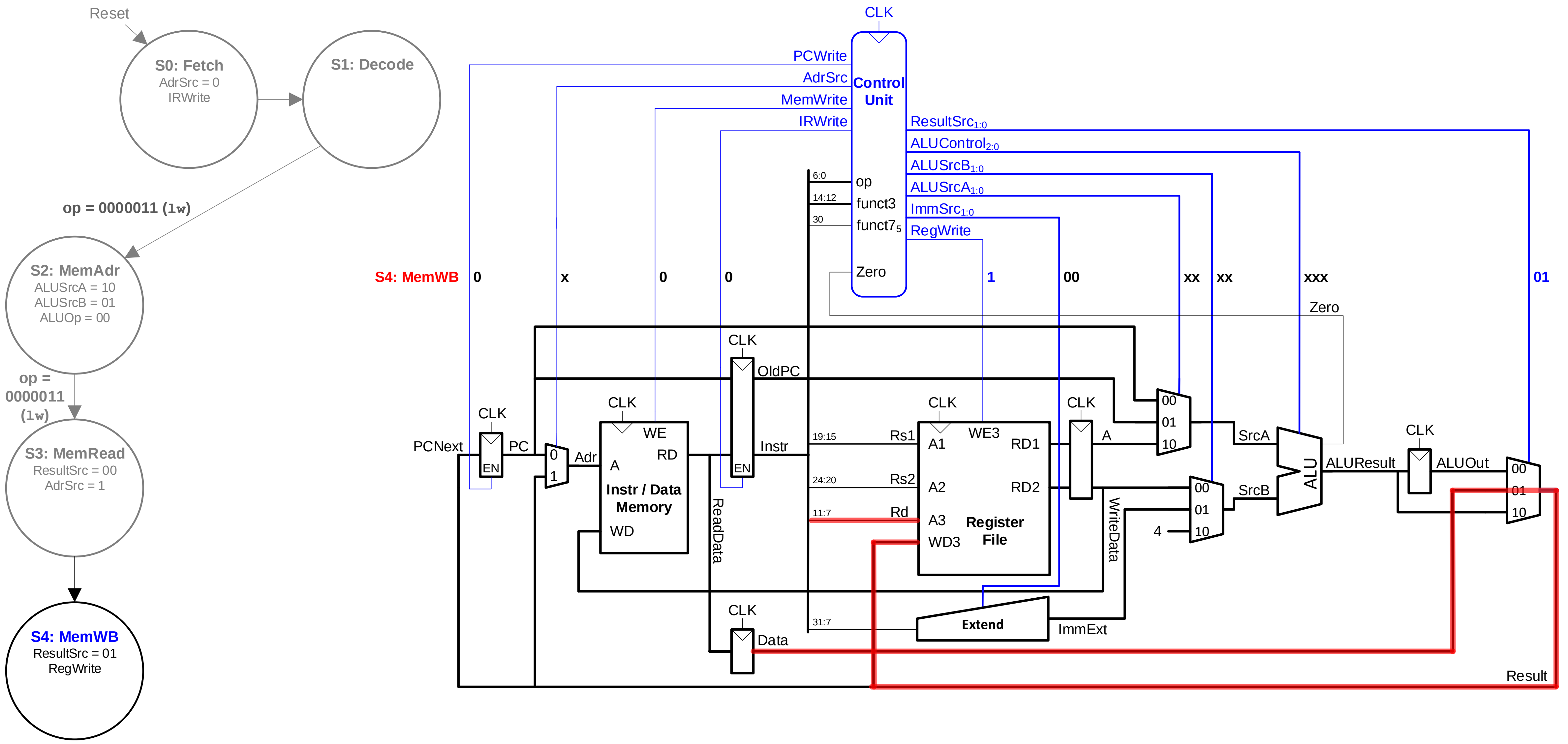


Main FSM: Read Memory

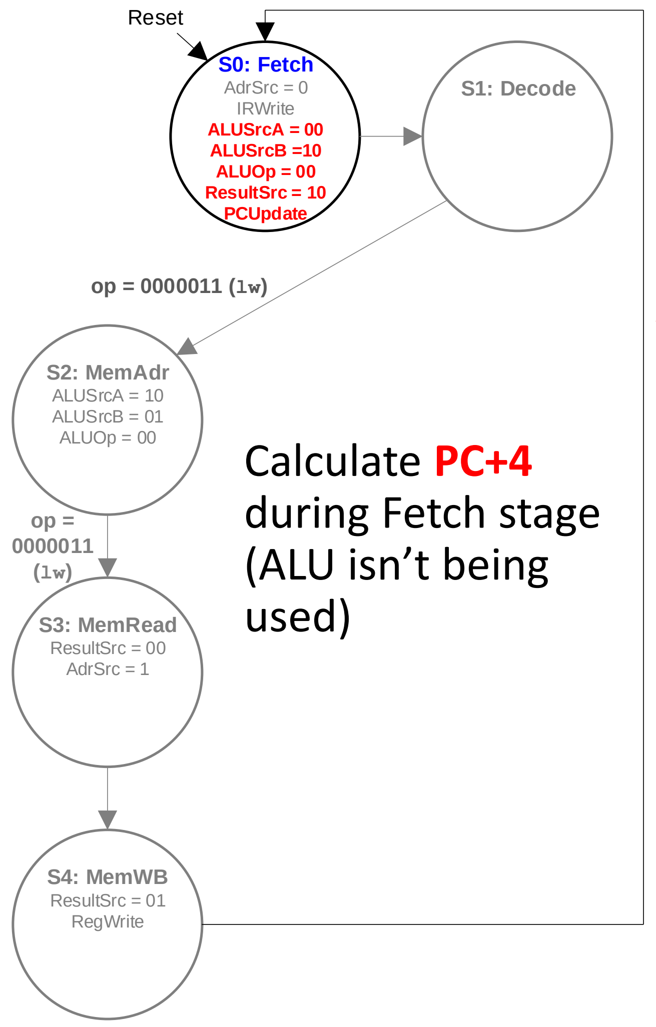




Main FSM: Write RF

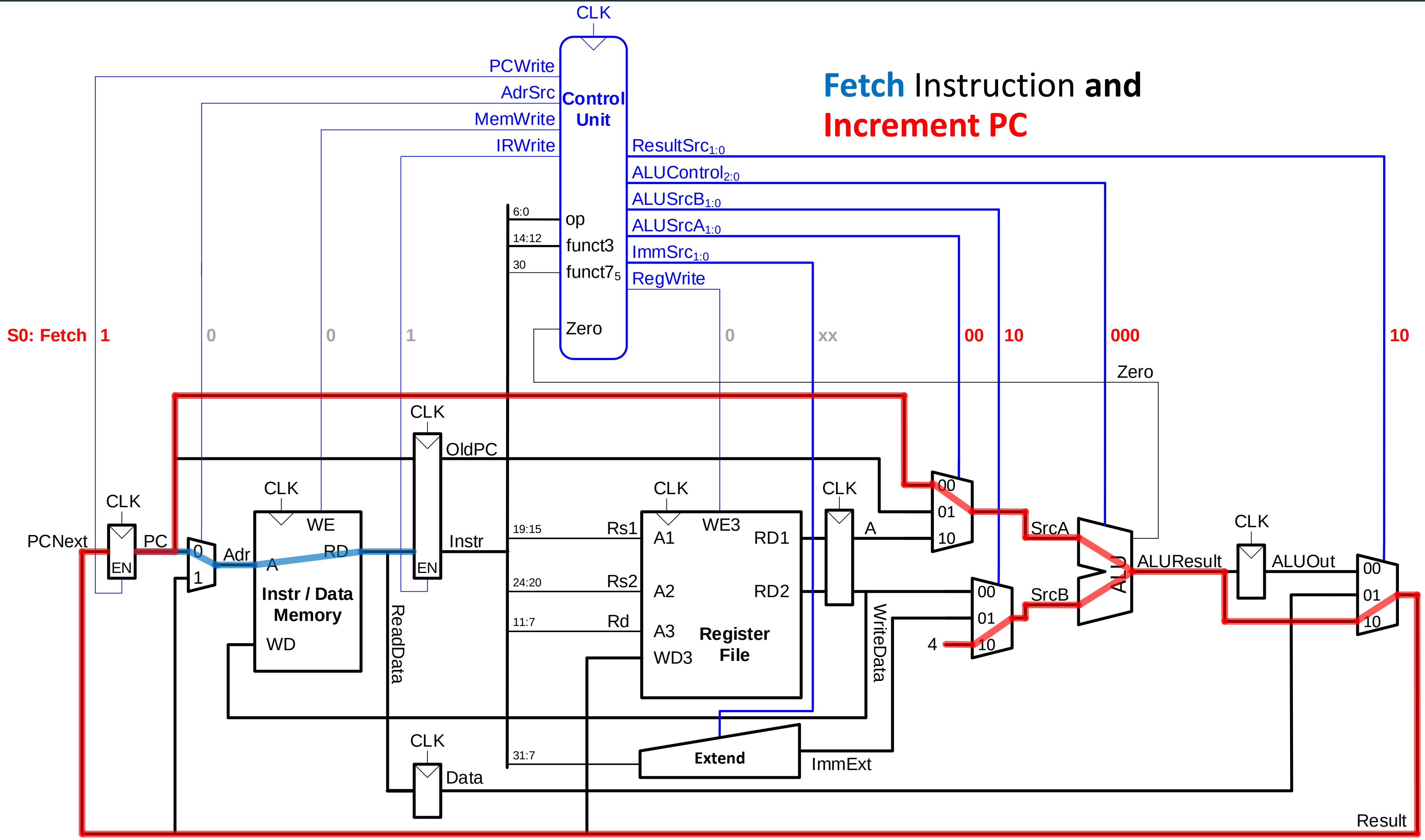


Main FSM: Fetch Revisited



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

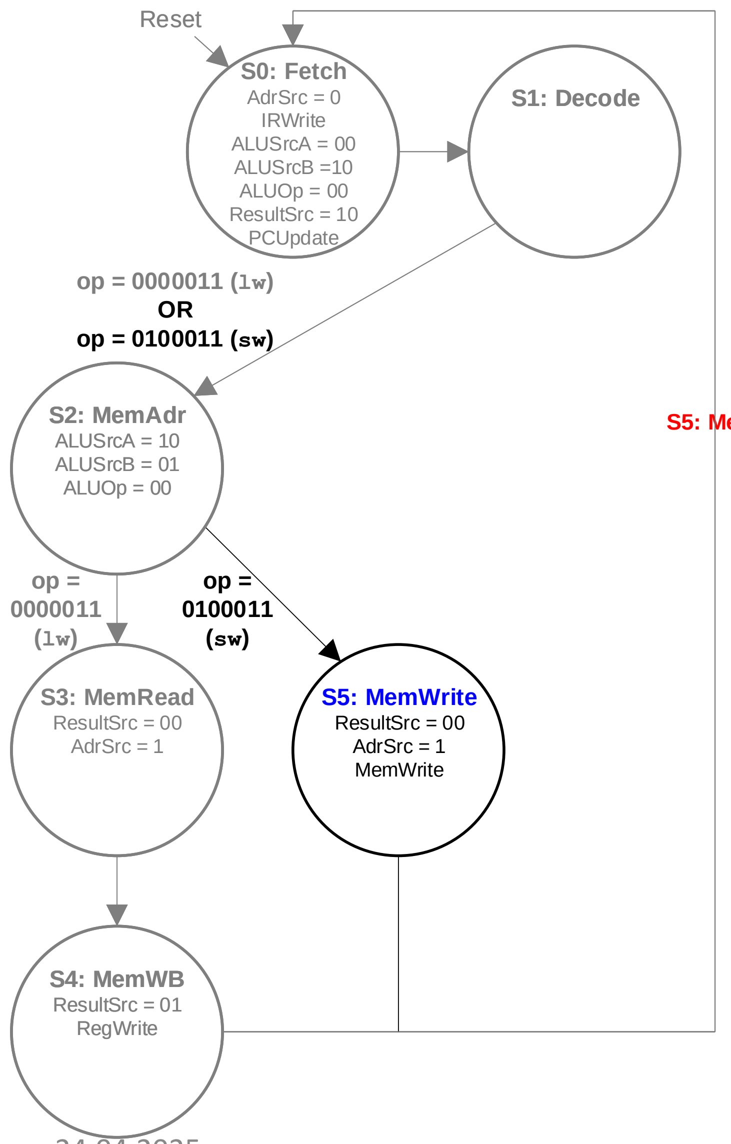
Fetch Instruction and **Increment PC**

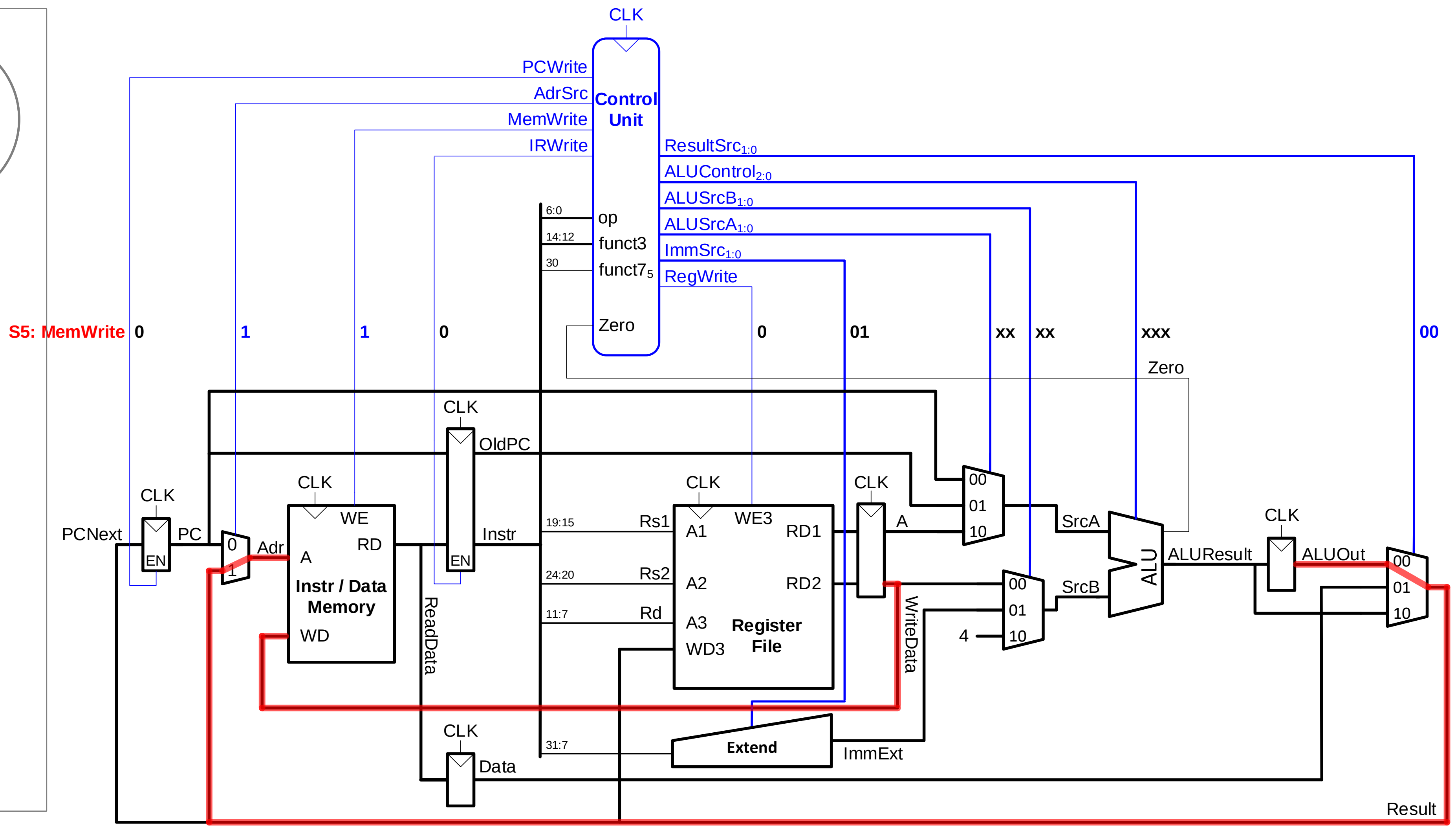


Multicycle Control: Other Instructions

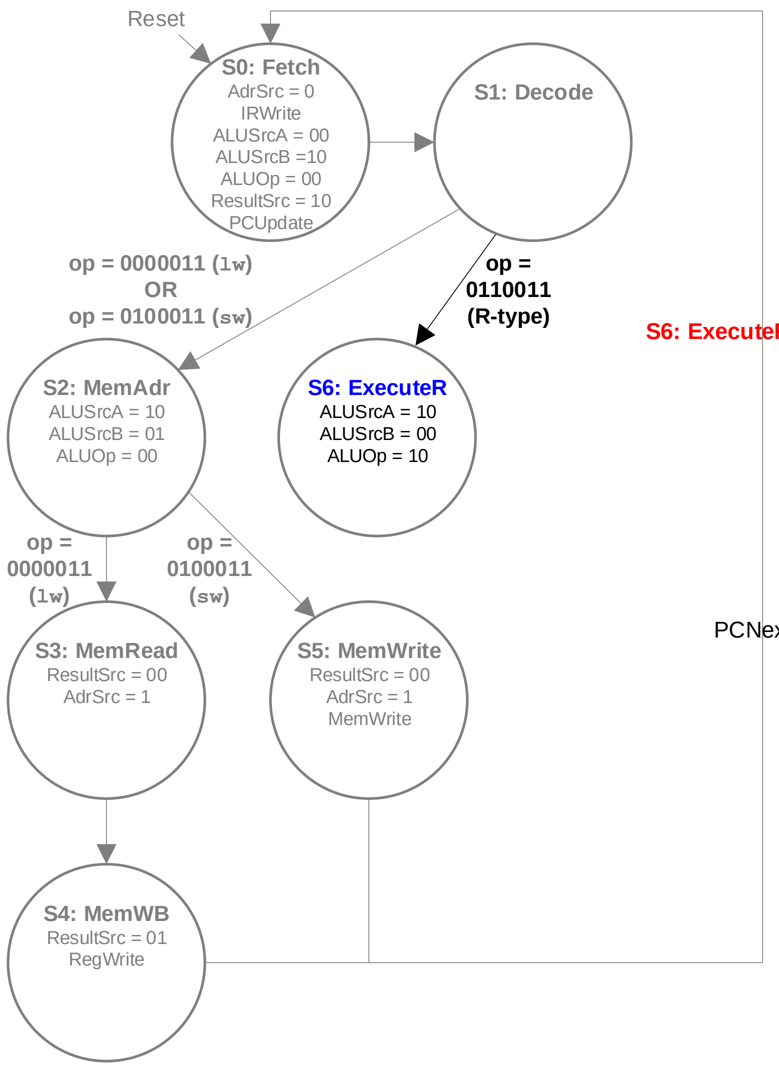
DDCA Ch7 - Part 10: RISC-V Multicycle Processor Control: Other Instructions
https://www.youtube.com/watch?v=rSodrnsYXQ8

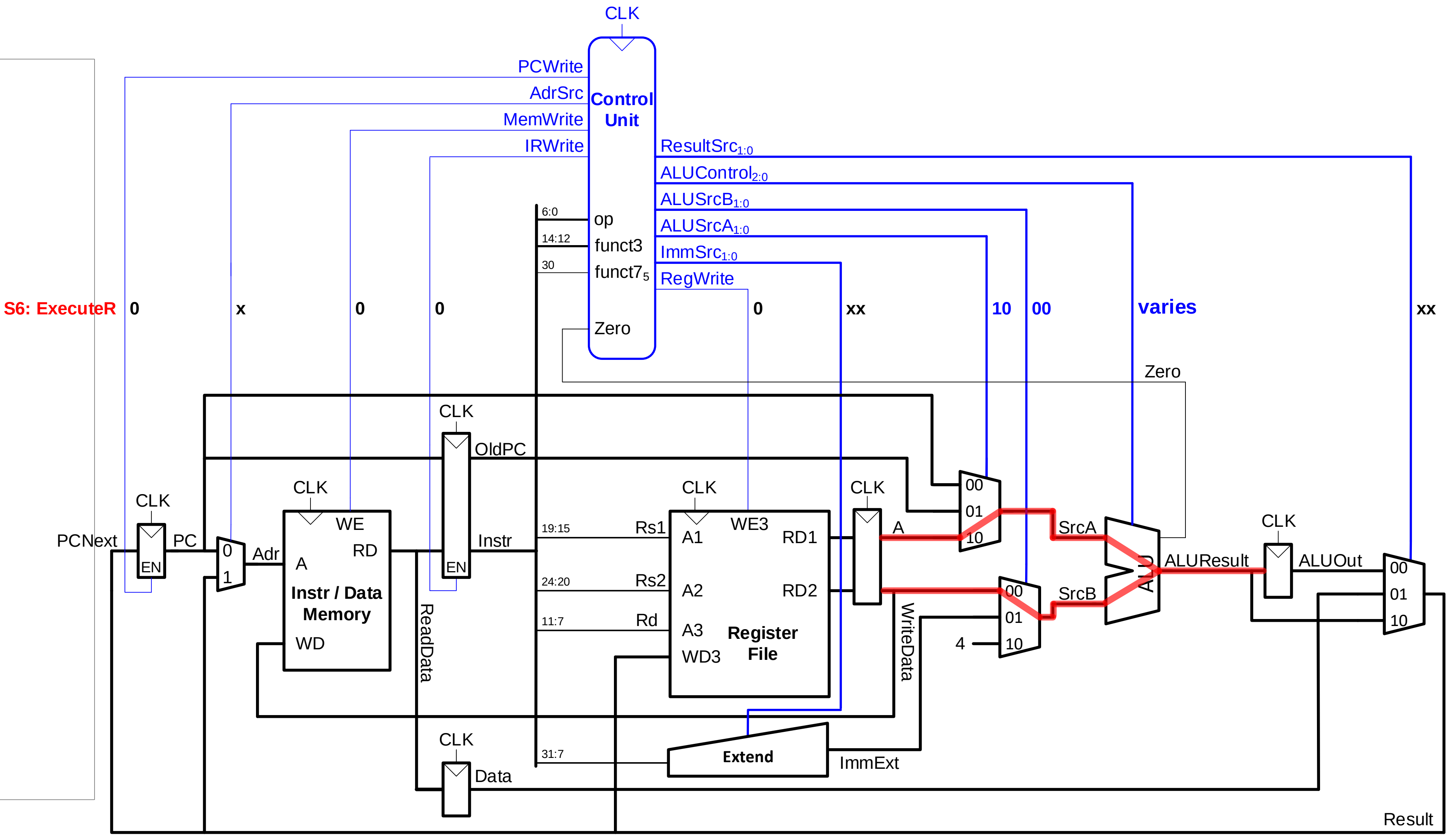
Main FSM: `sw`


Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
op = 0000011 (lw)
OR
op = 0100011 (sw)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S4: MemWB
ResultSrc = 01
RegWrite

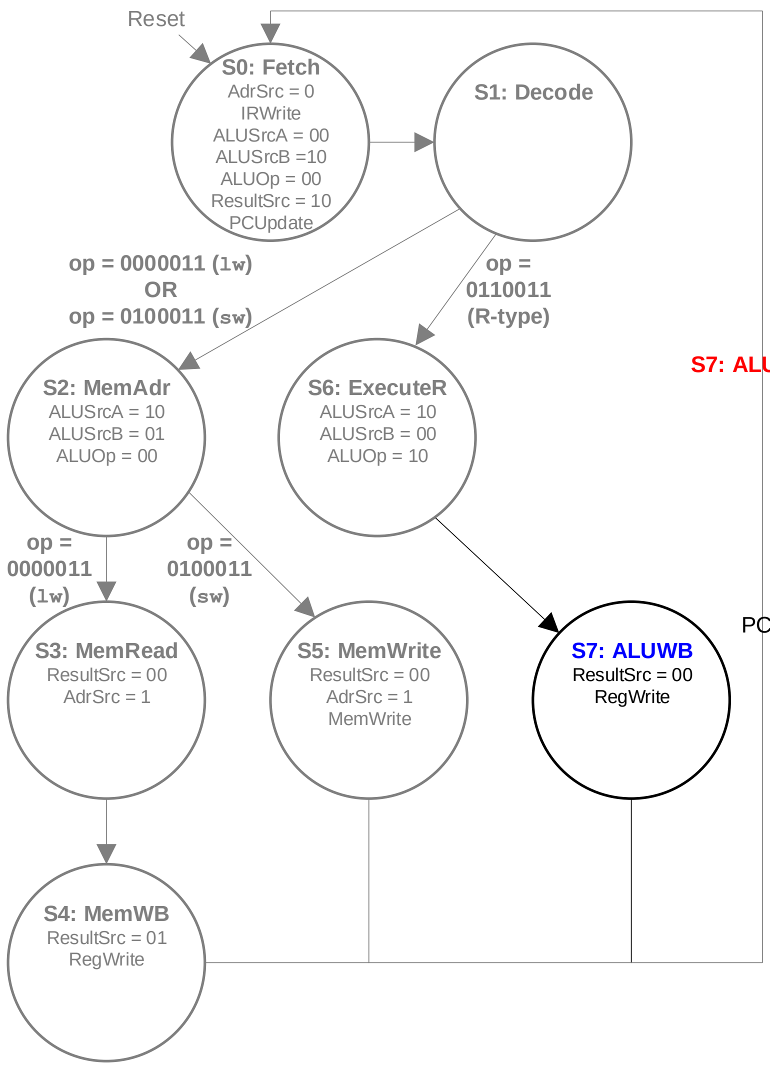

CLK
Control Unit
PCWrite
AdrSrc
MemWrite
IRWrite
ResultSrc1:0
ALUControl2:0
ALUSrcB1:0
ALUSrcA1:0
ImmSrc1:0
RegWrite
op
funct3
funct75
Zero
6:0
14:12
30
S5: MemWrite
0
1
1
0
0
01
xx
xx
xxx
00
Zero
CLK
OldPC
PCNext
CLK
EN
PC
0
1
Adr
CLK
WE
A
RD
Instr / Data Memory
WD
ReadData
EN
Instr
19:15
24:20
11:7
Rs1
Rs2
Rd
CLK
A1
WE3
RD1
A2
RD2
A3
WD3
Register File
CLK
A
WriteData
00
01
10
SrcA
00
01
10
4
SrcB
ALU
ALUResult
CLK
ALUOut
00
01
10
CLK
Data
31:7
Extend
ImmExt
Result

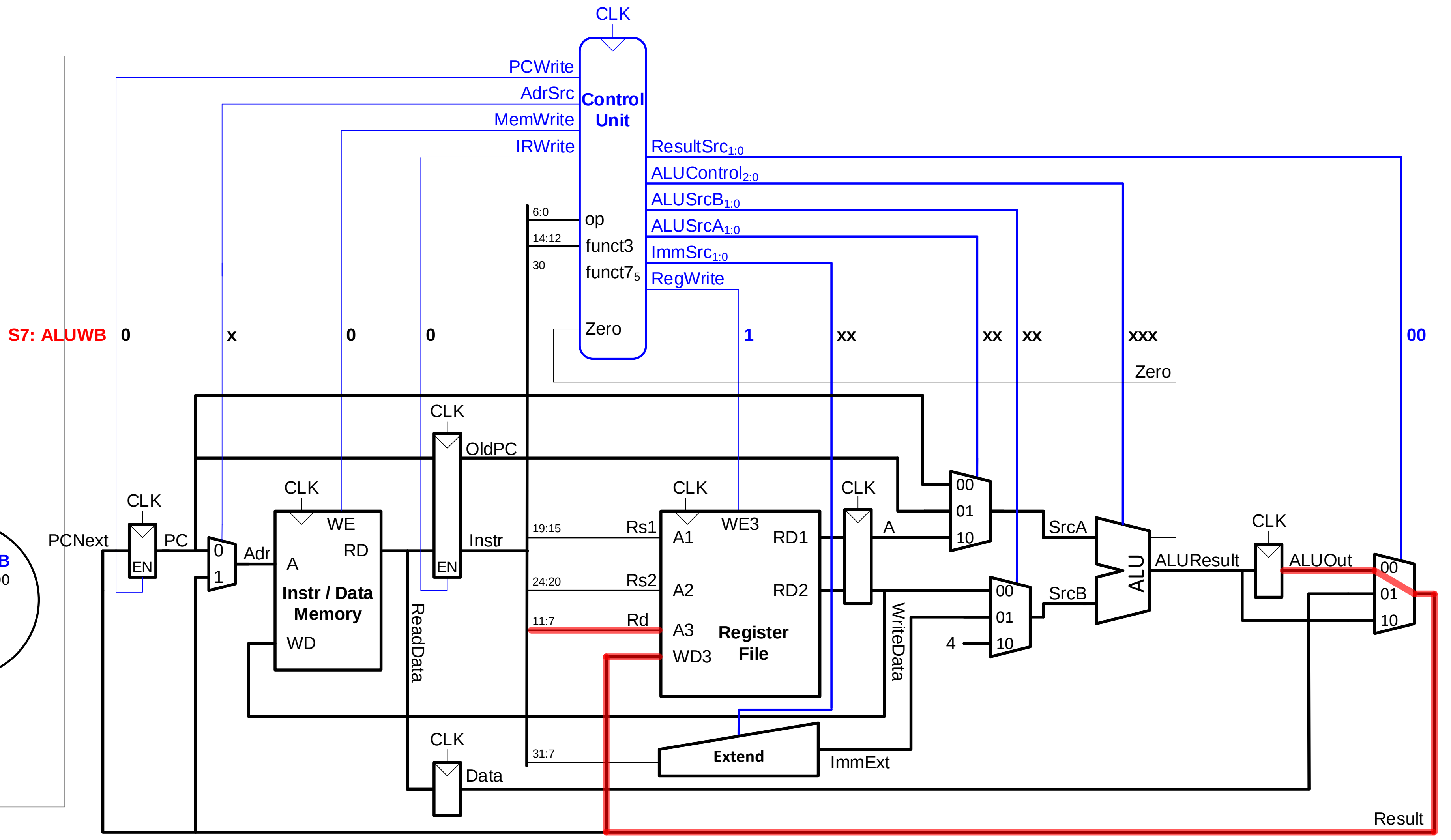
Main FSM: R-Type Execute





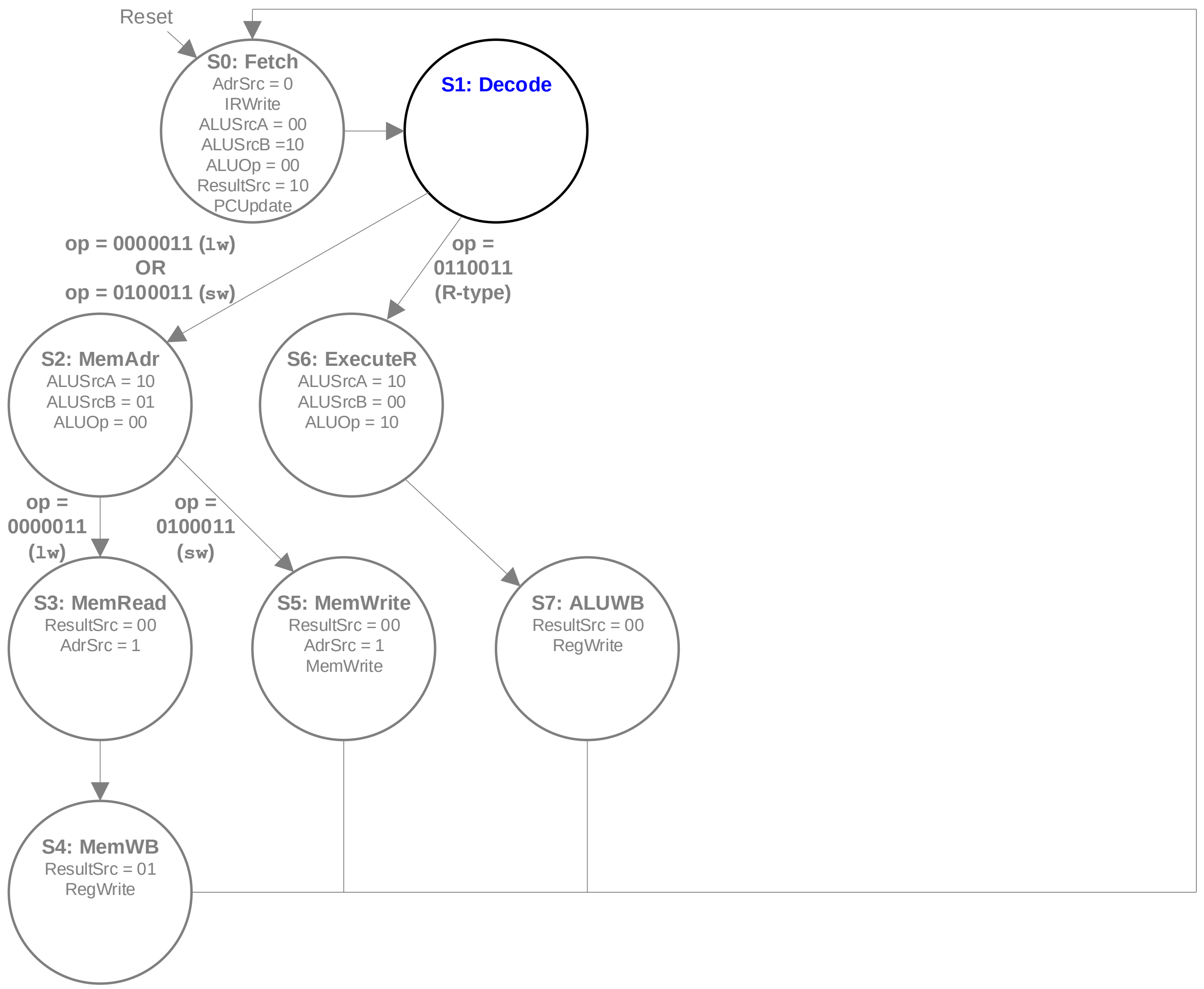
Main FSM: R-Type ALU Write Back





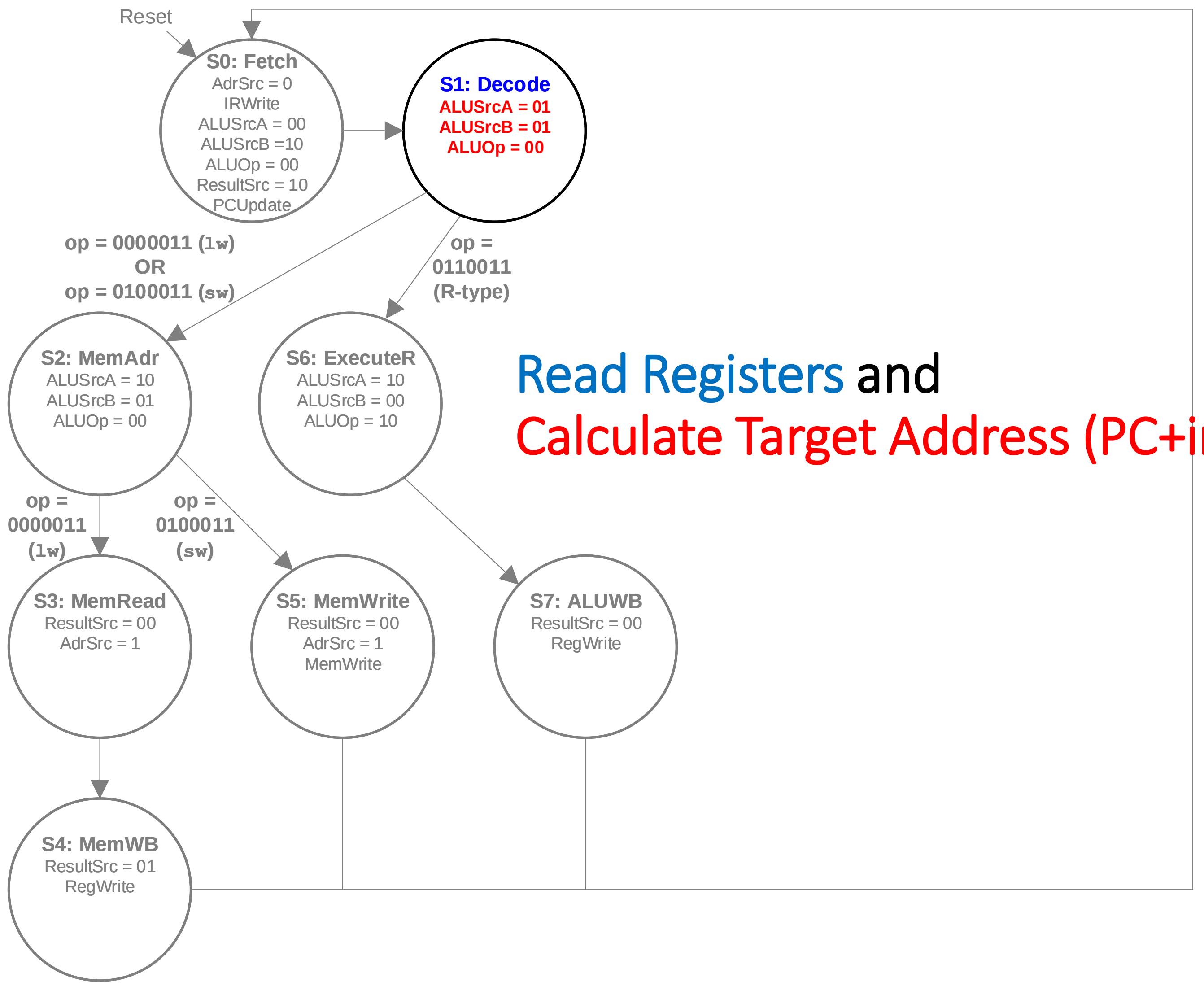
Main FSM: R-Type ALU Write Back

- **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)



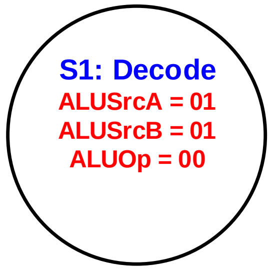
Main FSM: Decode Revisited

- **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)

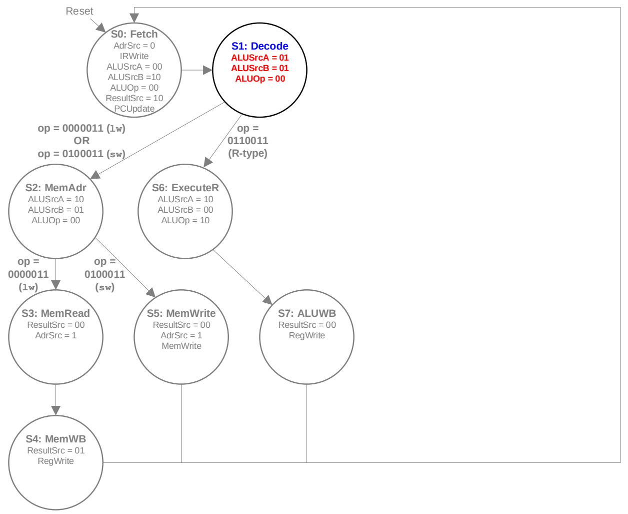


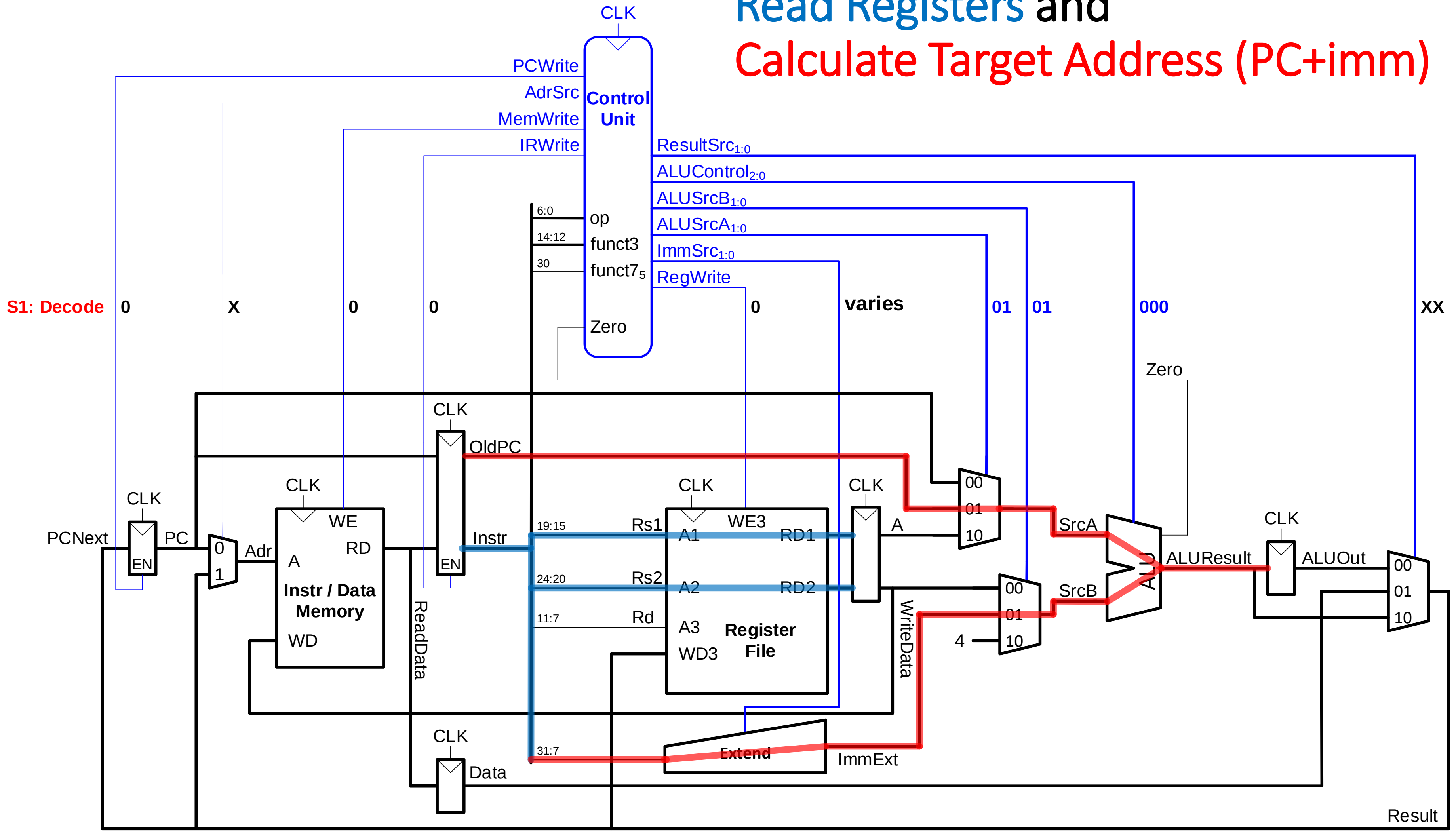
Read Registers and
Calculate Target Address (PC+imm)

Main FSM: Decode (Target Address)

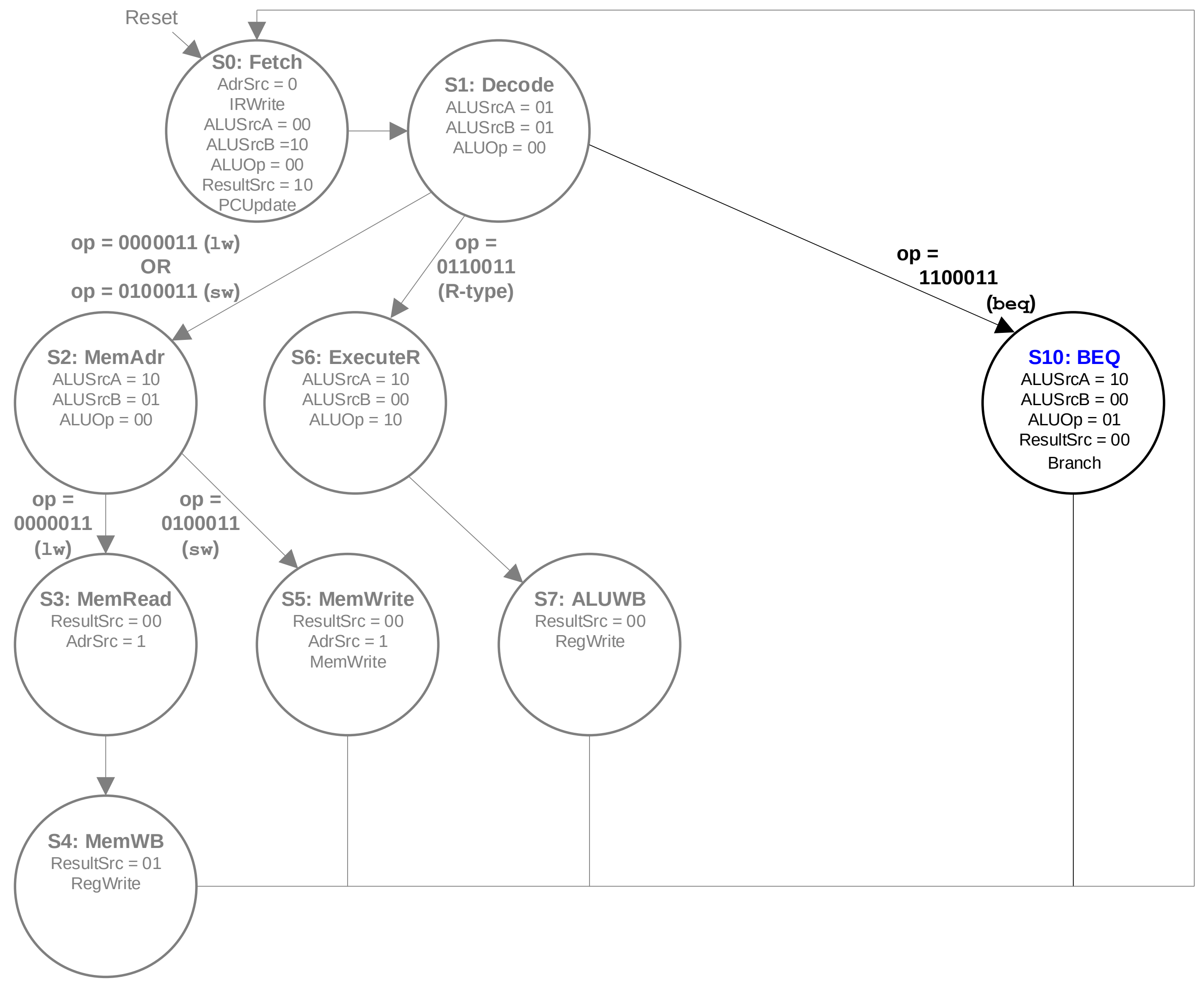


Read Registers and
Calculate Target Address (PC+imm)



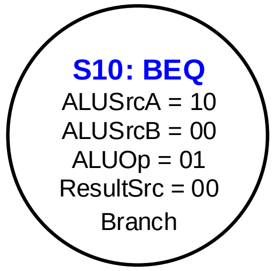


Main FSM: `beq`

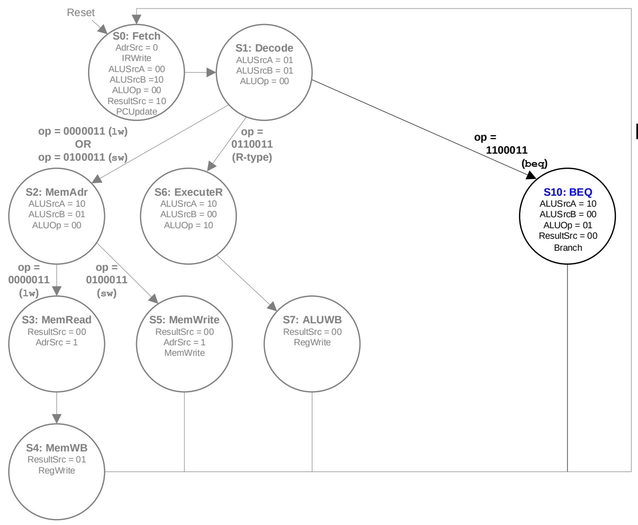


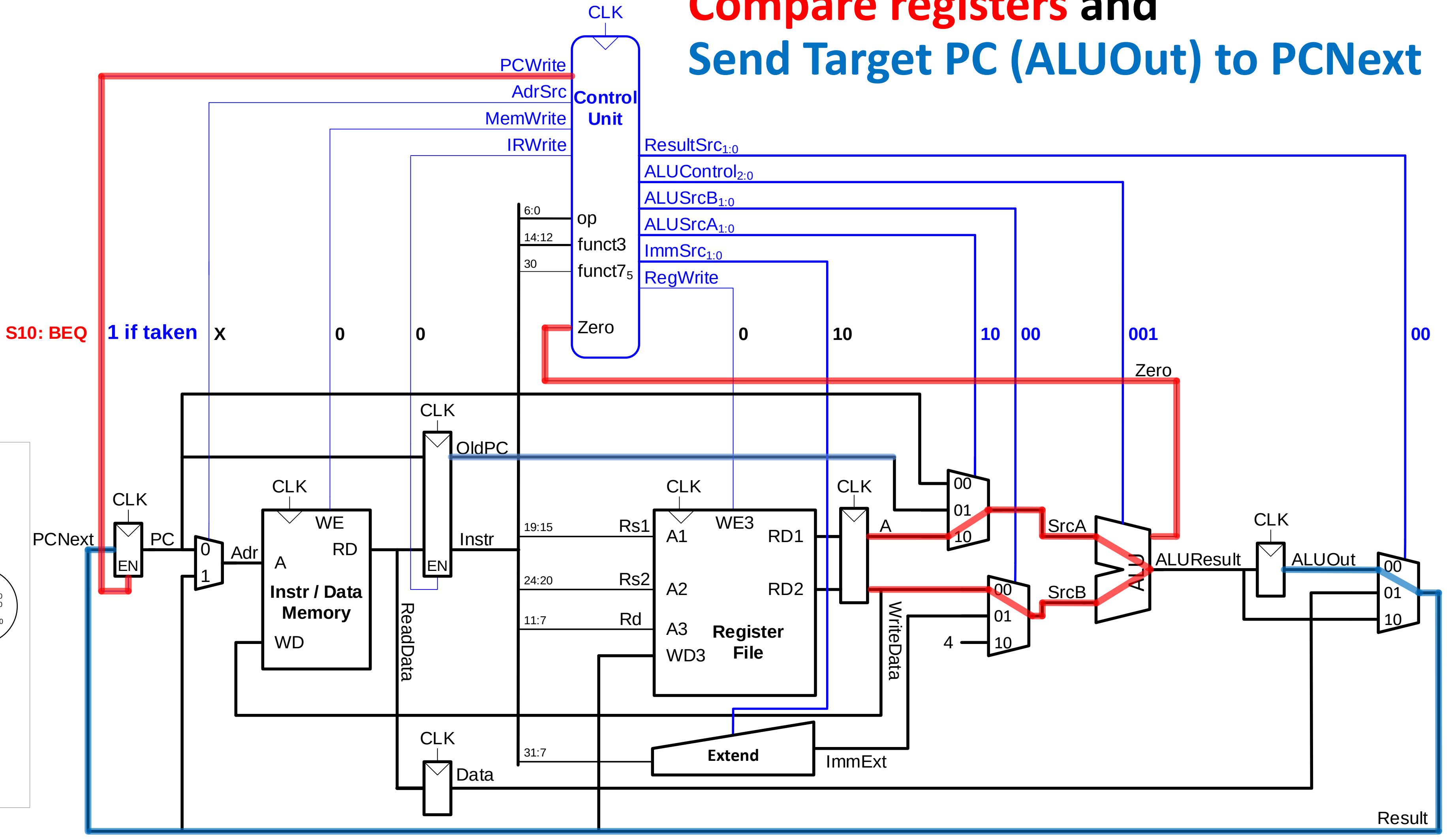
Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
ALUSrcA = 01
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw)
OR
op = 0100011 (sw)
op =
0110011
(R-type)
op =
1100011
(beq)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
S6: ExecuteR
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10
S10: BEQ
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 01
ResultSrc = 00
Branch
op =
0000011
(lw)
op =
0100011
(sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S7: ALUWB
ResultSrc = 00
RegWrite
S4: MemWB
ResultSrc = 01
RegWrite

Main FSM: `beq` Datapath



Compare registers and
Send Target PC (ALUOut) to PCNext

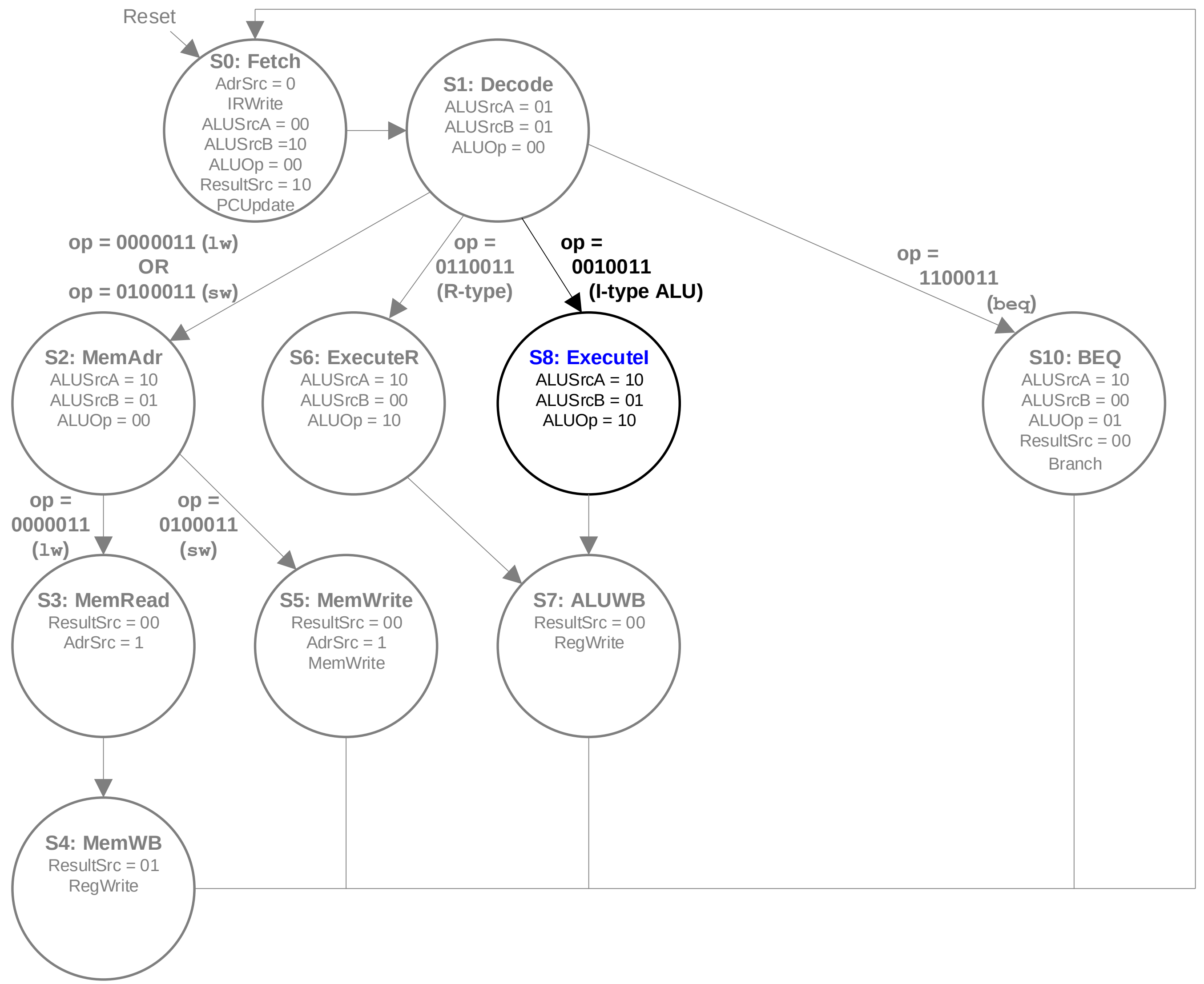




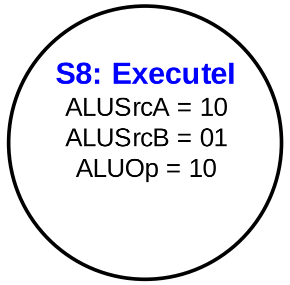
Extending the RISC-V Multicycle Processor

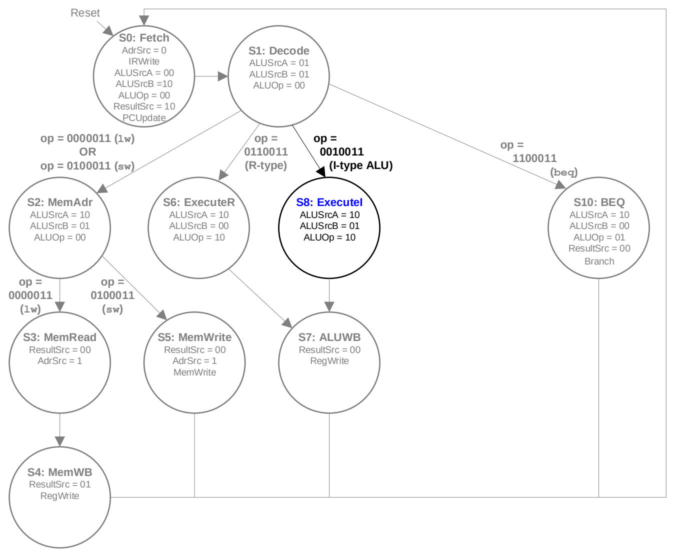
DDCA Ch7 - Part 11: Extending the RISC-V Multicycle Processor https://www.youtube.com/watch?v=8EhVN192FRU

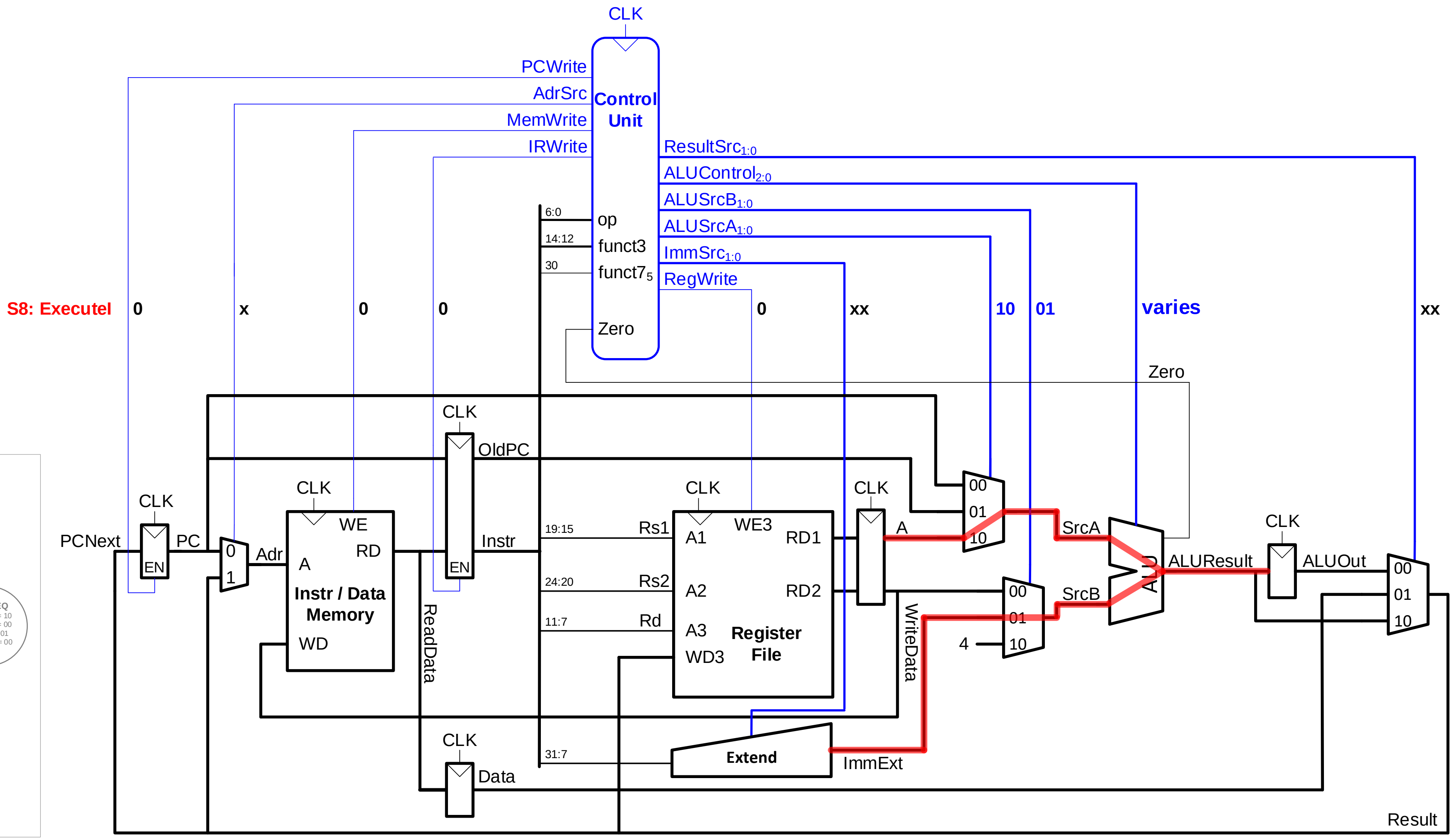
Main FSM: I-Type ALU Execute



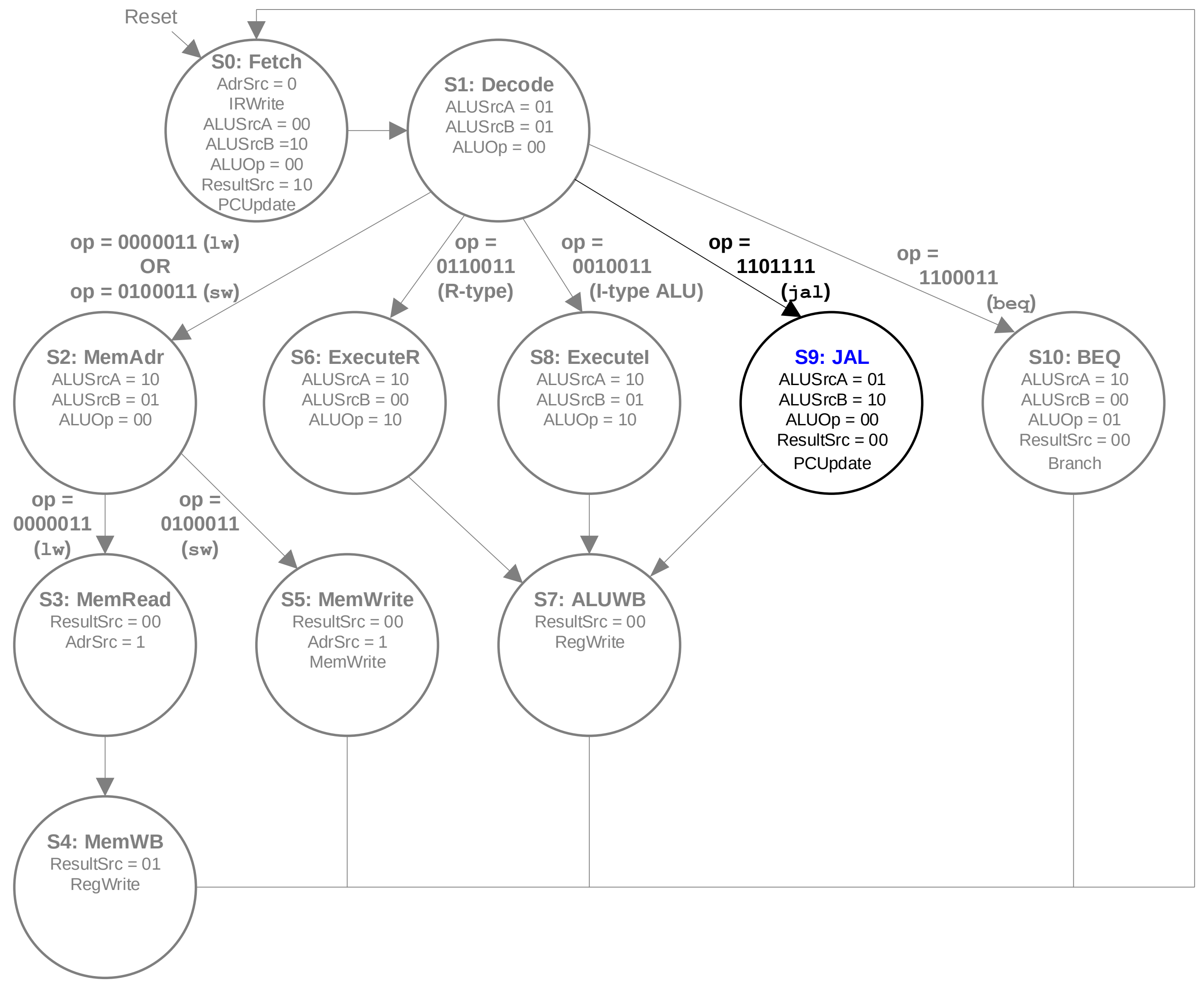
Main FSM: I-Type ALU Exec. Datapath







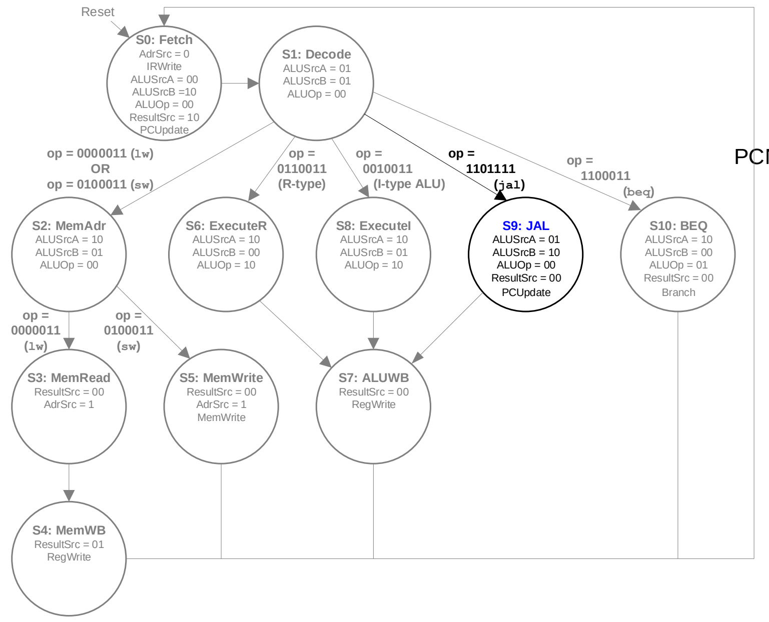
Main FSM: `jal`



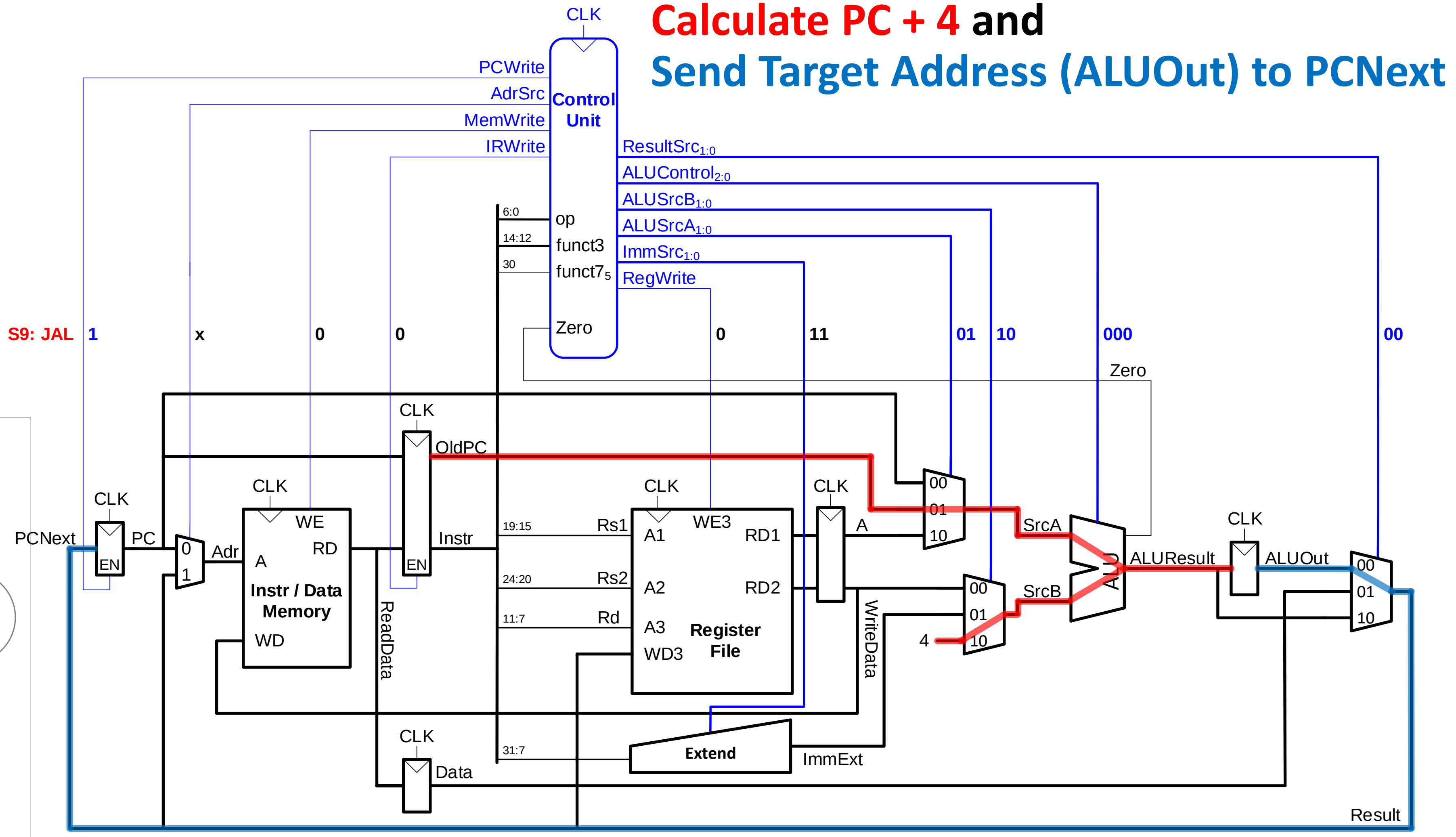
Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
ALUSrcA = 01
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw) OR op = 0100011 (sw)
op = 0110011 (R-type)
op = 0010011 (I-type ALU)
op = 1101111 (jal)
op = 1100011 (beq)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
S6: ExecuteR
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10
S8: ExecuteI
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 10
S9: JAL
ALUSrcA = 01
ALUSrcB = 10
ALUOp = 00
ResultSrc = 00
PCUpdate
S10: BEQ
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 01
ResultSrc = 00
Branch
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S7: ALUWB
ResultSrc = 00
RegWrite
S4: MemWB
ResultSrc = 01
RegWrite

Main FSM: `jal` Datapath

S9: JAL
ALUSrcA = 01
ALUSrcB = 10
ALUOp = 00
ResultSrc = 00
PCUpdate

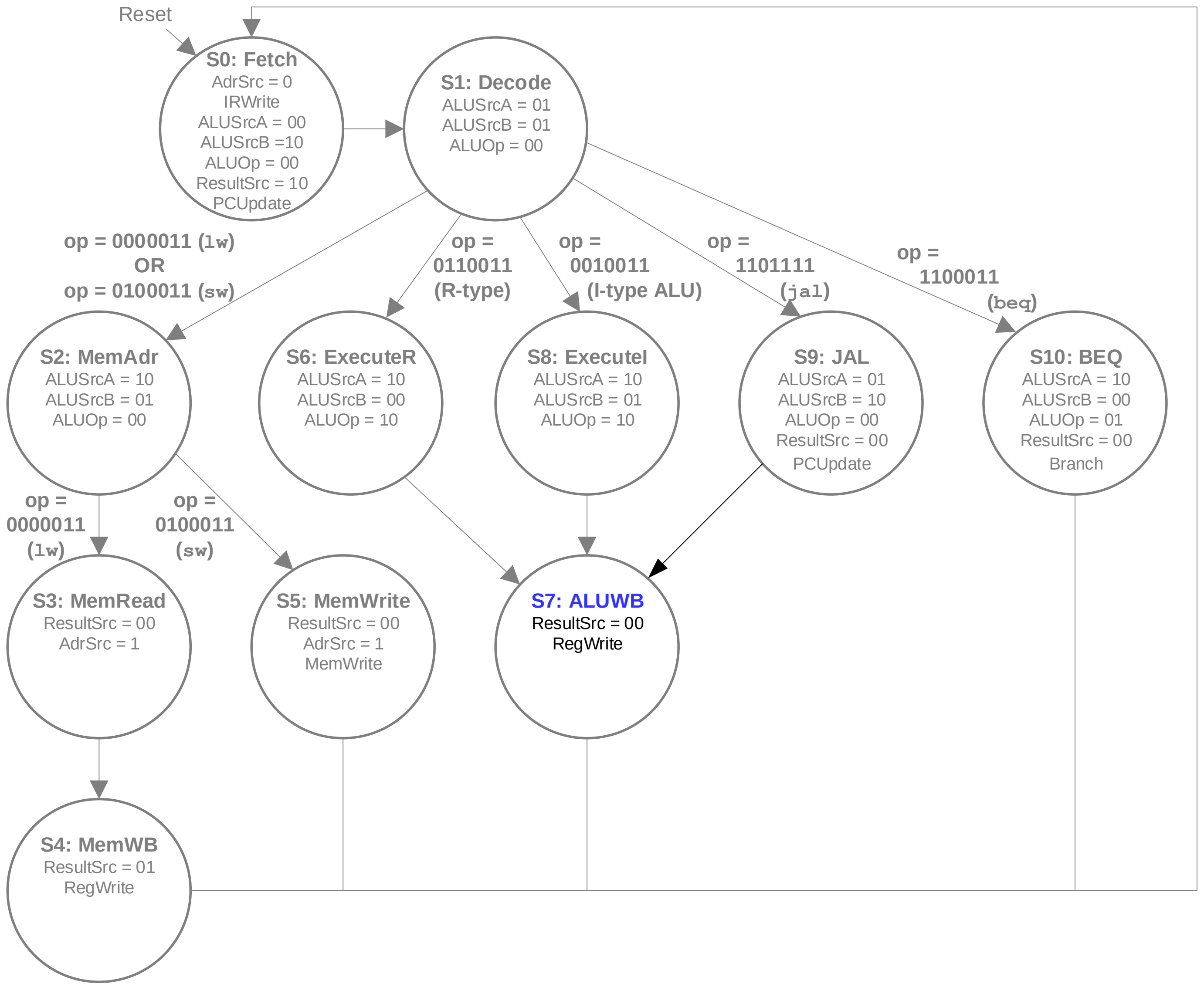


Calculate PC + 4 and
Send Target Address (ALUOut) to PCNext



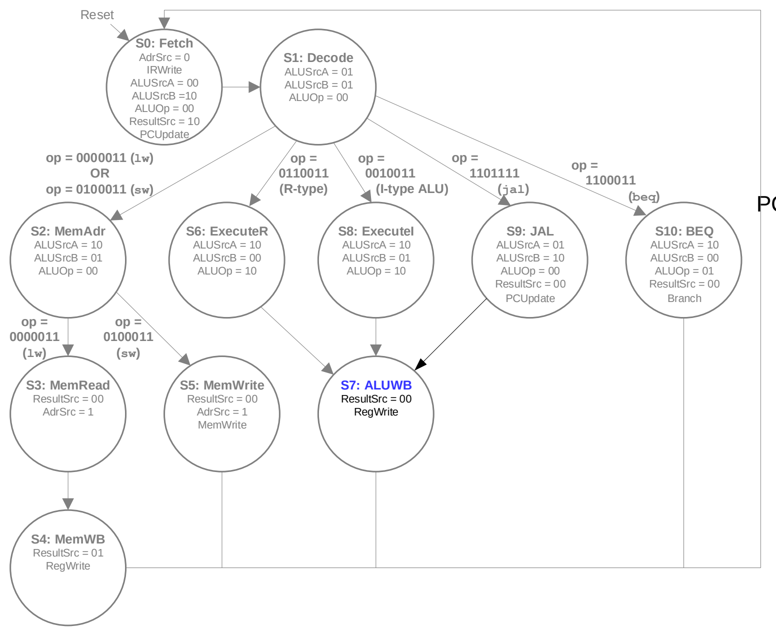
Main FSM: `jal`

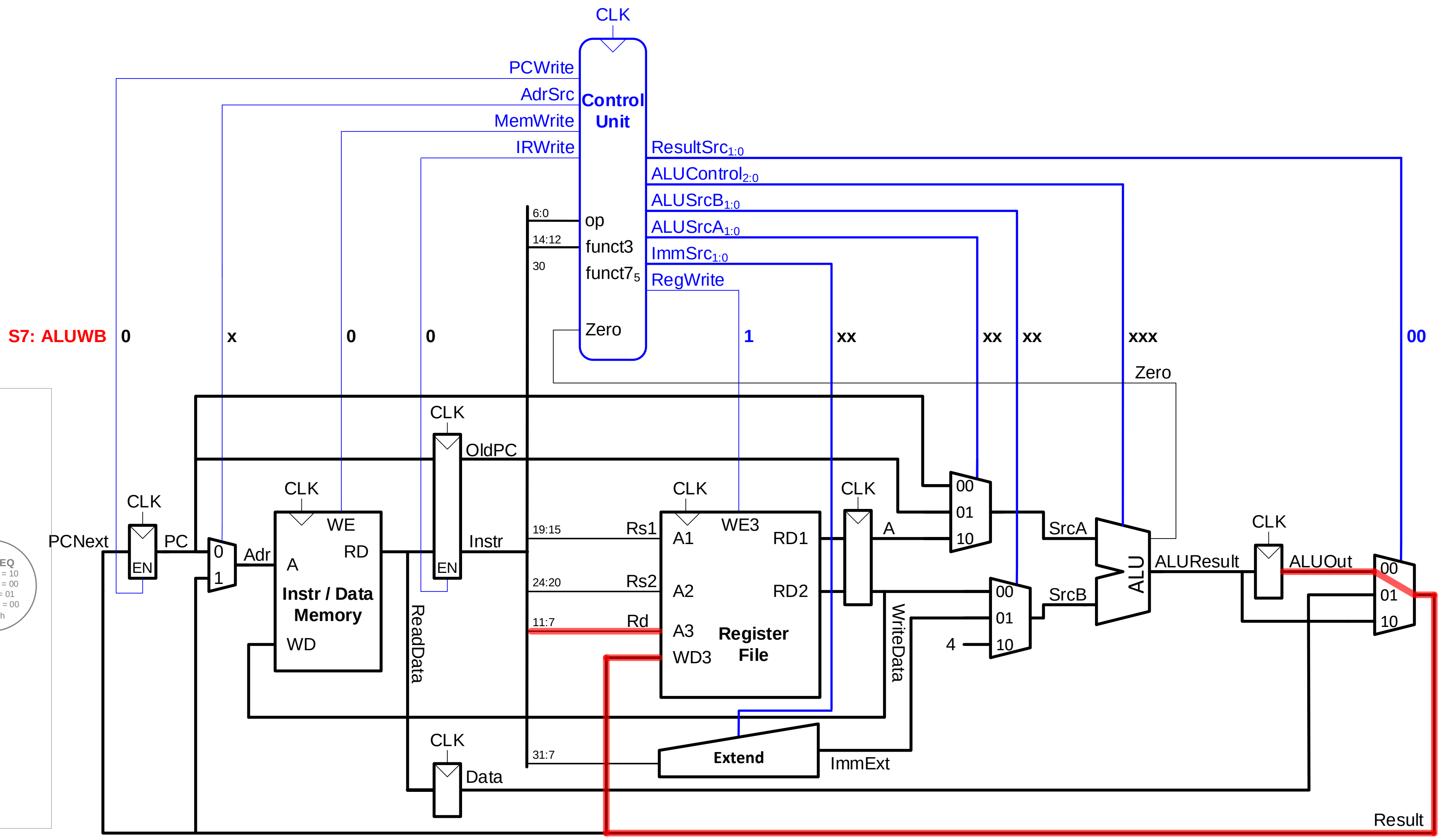
PC + 4 is written to **rd** in S7: ALUWB



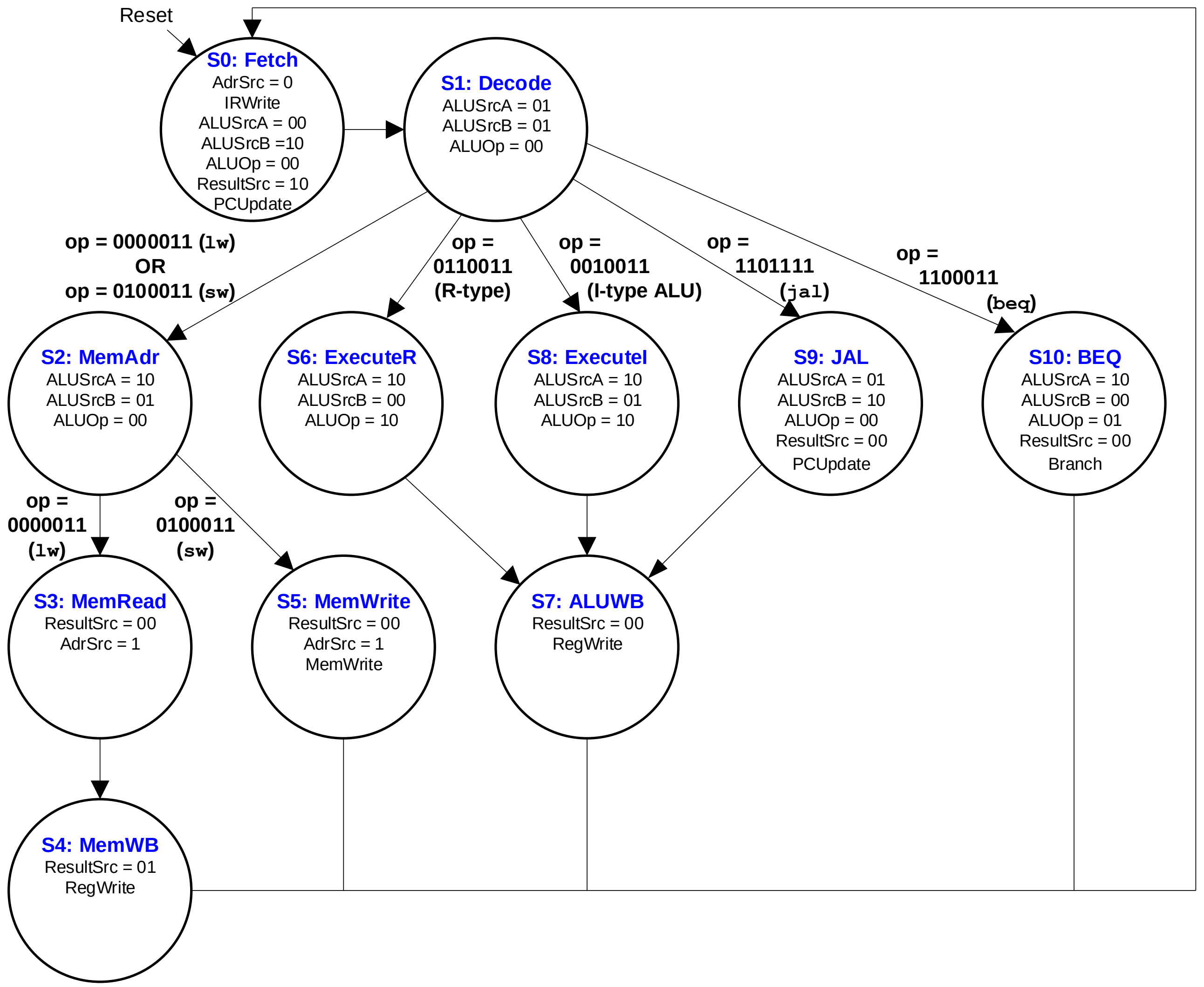
Main FSM: `jal` Datapath – WB same as before with PC+4

S7: ALUWB
ResultSrc = 00
RegWrite





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



Multicycle Performance

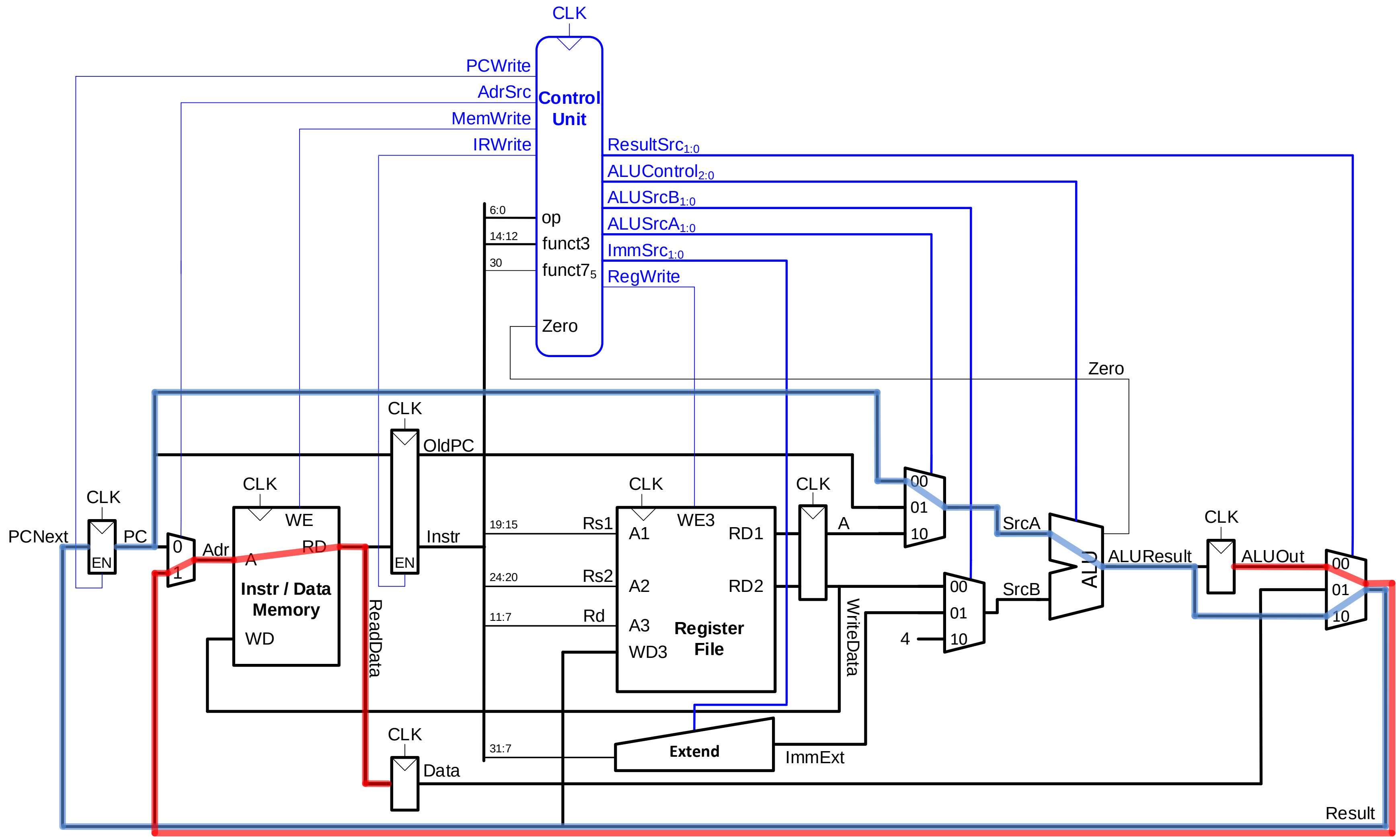
- Instructions take different number of cycles:
  - 3 cycles: `beq`
  - 4 cycles: R-type, `addi, sw , jal`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type

- **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

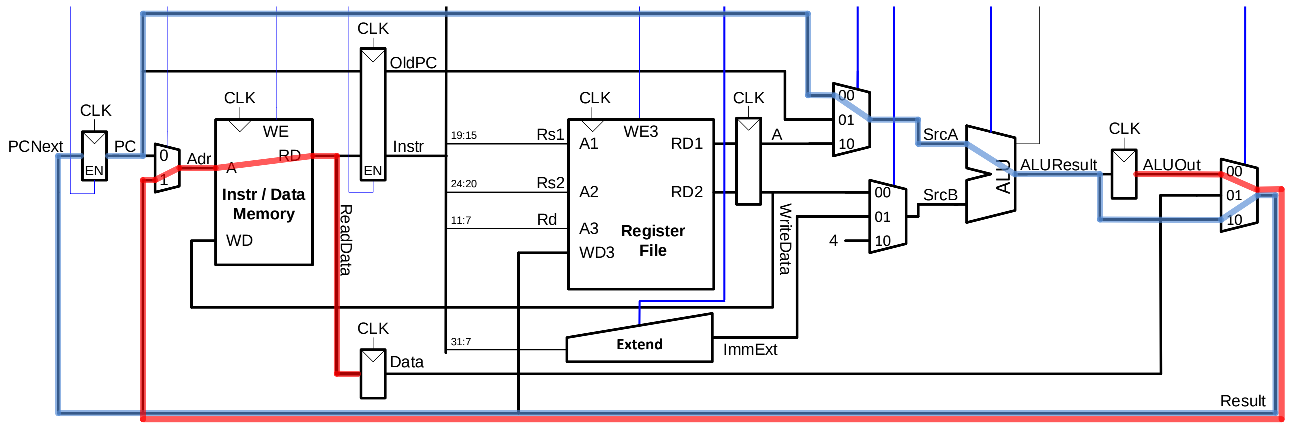


Multicycle Processor Performance

Multicycle critical path:

- **Assumptions**:
  - Registry File is faster than memory
  - Writing memory is faster than reading memory

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



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	$t_{\mathrm{AND\text{-}OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{dec}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\mathrm{read}}$	100
Register file setup	$t_{RF\mathrm{setup}}$	60

$$T_{c\_multi} = t_{pcq\_PC} + t_{dec} + 2t_{mux} + \max(t_{ALU}, t_{mem}) + t_{setup}$$
$$= (40 + 25 + 2 * 30 + 200 + 50)\, ps = \mathbf{375\, ps}$$

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 ps

- **Execution Time** = (# instructions) × CPI × T_c
  = $(100 \times 10^9)(4.12)(375 \times 10^{-12})$
  = **155 seconds**

- **This is slower than the single-cycle processor (75 sec.)**

Parallelism

DDCA Ch3 - Part 16: Parallelism https://www.youtube.com/watch?v=xX2Crru3xCg

Parallelism

Two types of parallelism:

- Spatial parallelism
  - duplicate hardware performs multiple tasks at once
- Temporal parallelism
  - task is broken into multiple stages
  - also called pipelining
  - for example, an assembly line

Parallelism

- Token: Group of inputs processed to produce group of outputs
- Latency: Time for one token to pass from start to end
- Throughput: Number of tokens produced per unit time

Parallelism increases throughput

Parallelism Example 1/3

- Ben Bitdiddle bakes cookies to celebrate traffic light controller installation
  - **5 minutes** to roll cookies
  - **15 minutes** to bake

- What is the latency and throughput without parallelism?
  - Latency (when is the first cooky finished?)
  - Throughput (how many cookies can Ben finish in an hour?)

Latency = 5 + 15 = 20 minutes = 1/3 hour

Throughput = 1 tray/ 1/3 hour = 3 trays/hour

Parallelism Example 2/3

What is the latency and throughput if Ben uses parallelism?

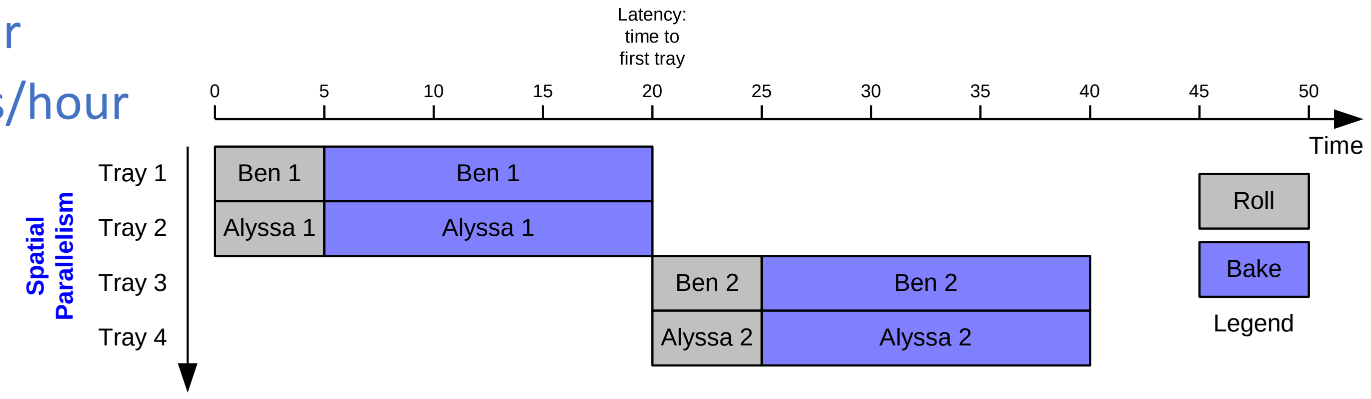
- Spatial parallelism:

  Ben asks Allysa P. Hacker to help, using her own oven

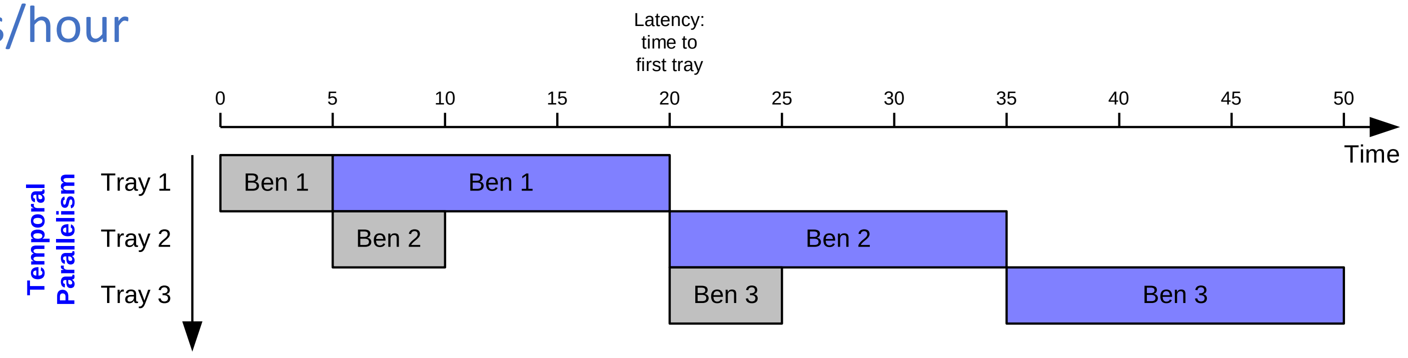
- Temporal parallelism:
  - two stages: rolling and baking
  - He uses two trays
  - While first batch is baking, he rolls the second batch, etc.

Spatial vs. Temporal Parallelism

- Spatial Parallelism
  - Latency = 5 + 15 = 20 minutes = 1/3 hour
  - Throughput = 2 trays/ 1/3 hour = 6 trays/hour



- Temporal Parallelism
  - Latency = 5 + 15 = 20 minutes = 1/3 hour
  - Throughput = 1 trays/ 1/4 hour = 4 trays/hour



Parallelism in Circuits

Pipelined RISC-V Processor

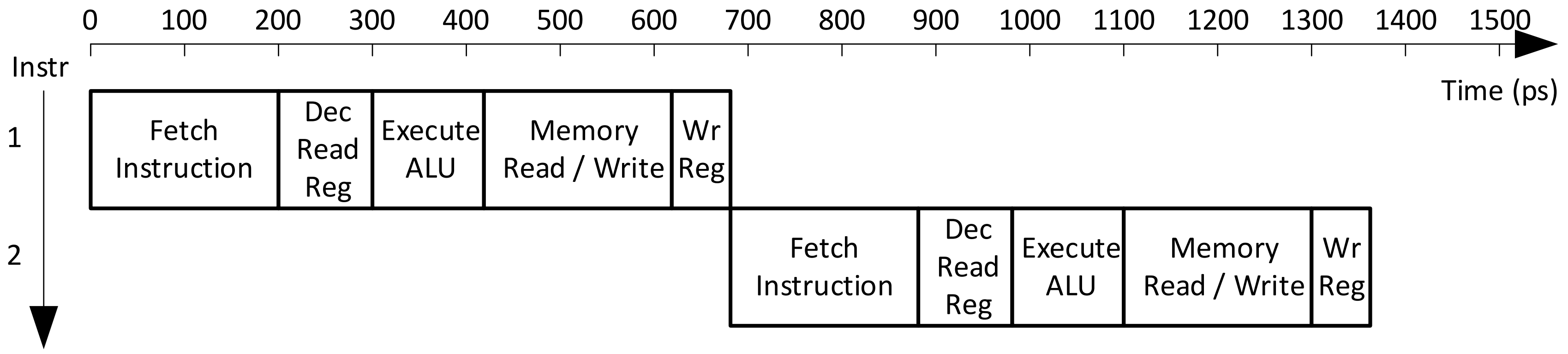
DDCA Ch7 - Part 13: Pipelined Processor https://www.youtube.com/watch?v=UZdURUwQMmk

Pipelined RISC-V Processor

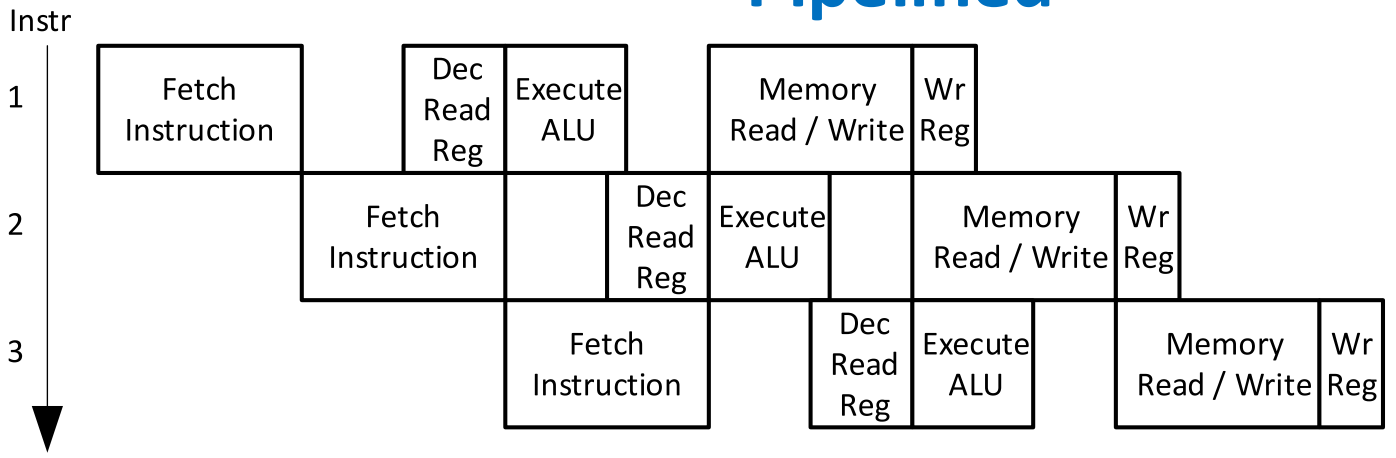
- **Temporal parallelism**
- Divide single-cycle processor into **5 stages:**
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add **pipeline registers** between stages

Single-Cycle vs. Pipelined Processor

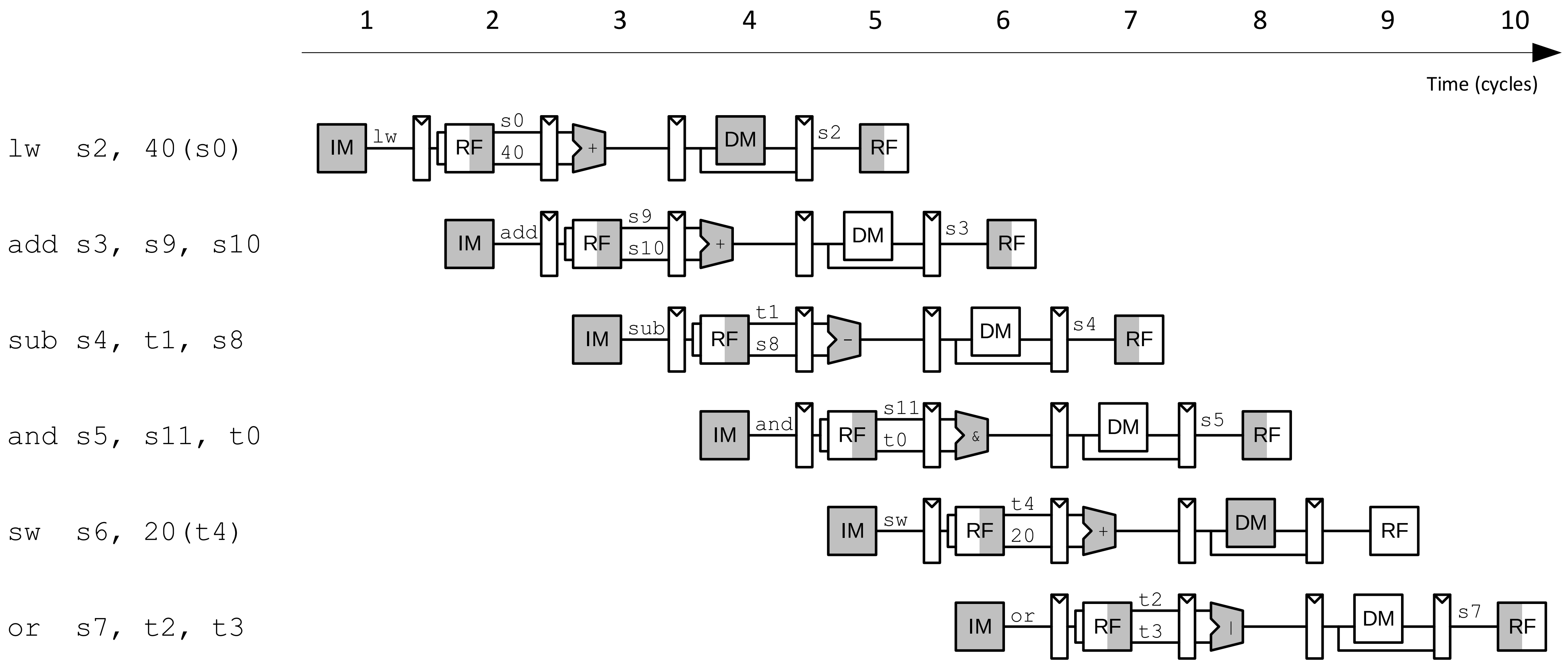
Single-Cycle



Pipelined

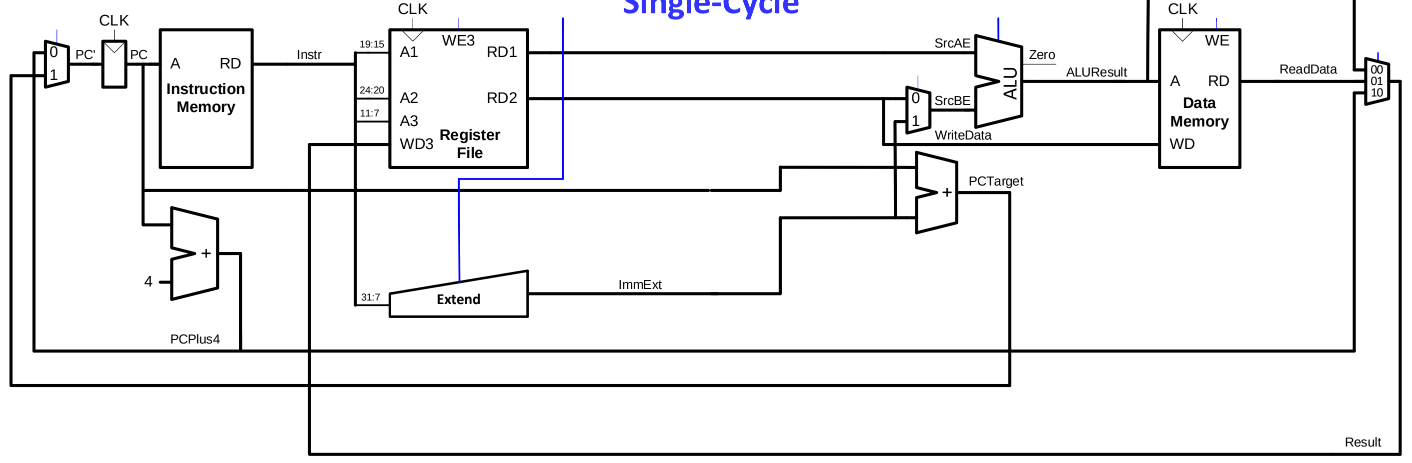


Pipelined Processor Abstraction

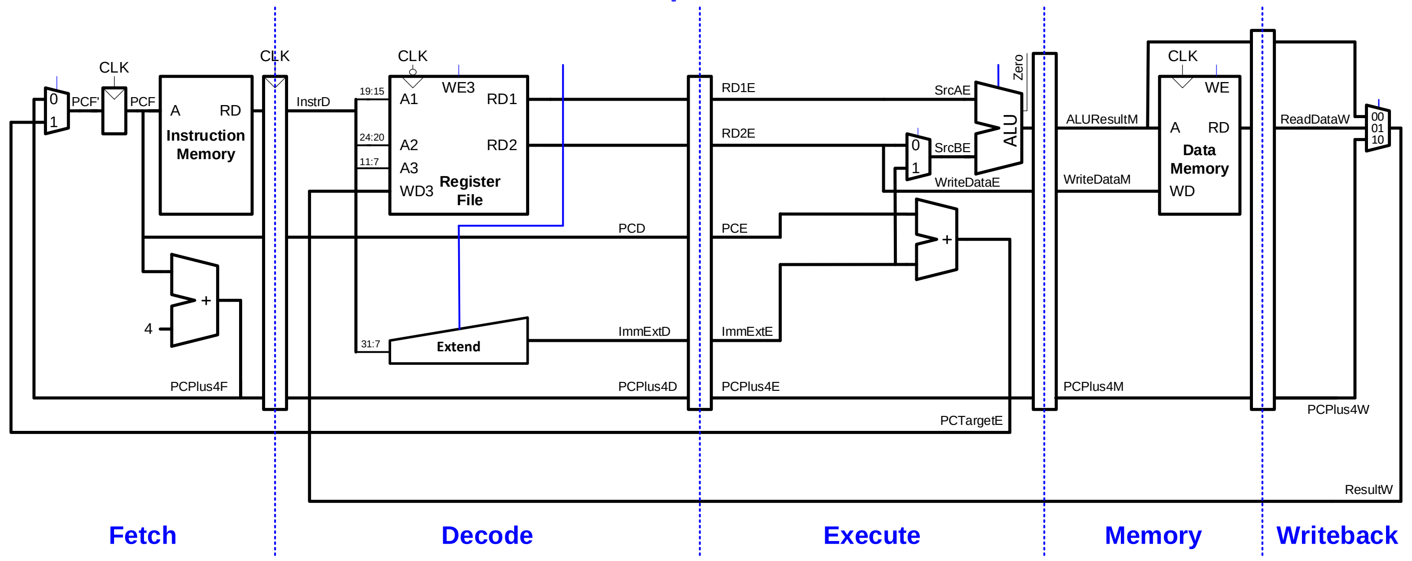


Single-Cycle & Pipelined Datapaths

Single-Cycle

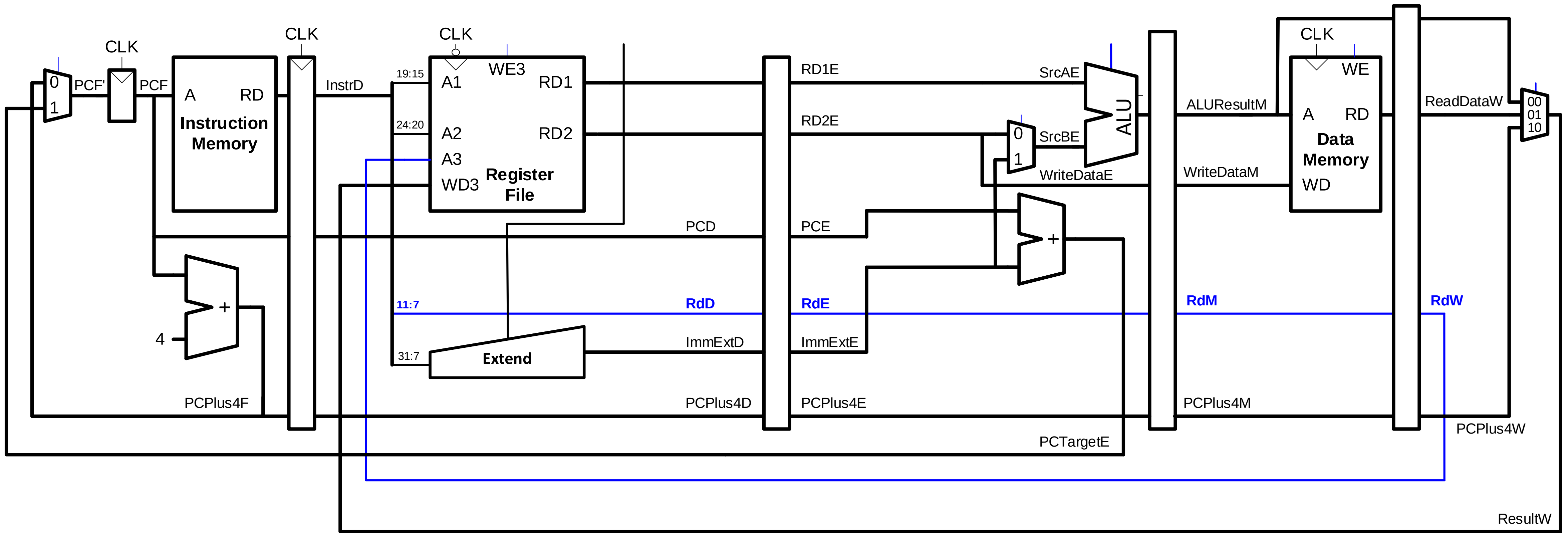


Pipelined



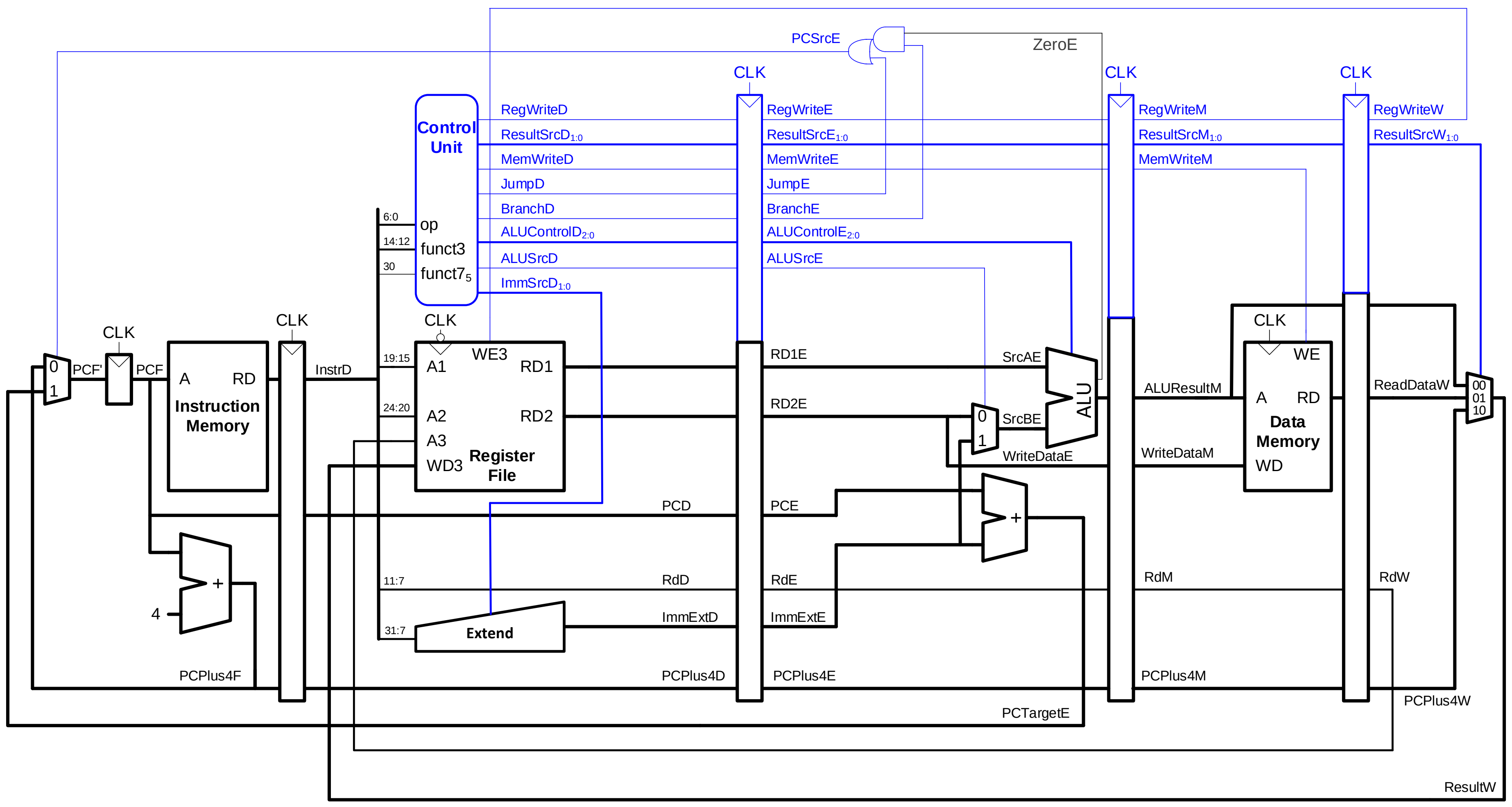
Signals in Pipelined Processor are appended with first letter of stage (i.e., PC**F**, PC**D** , PC**E** , PC**M** , PC**W**).

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)

Pipelined Processor Hazards

DDCA Ch7 - Part 14: Pipelined Processor Data Hazards https://www.youtube.com/watch?v=zuegcg6ZSFQ

Pipelined Hazards

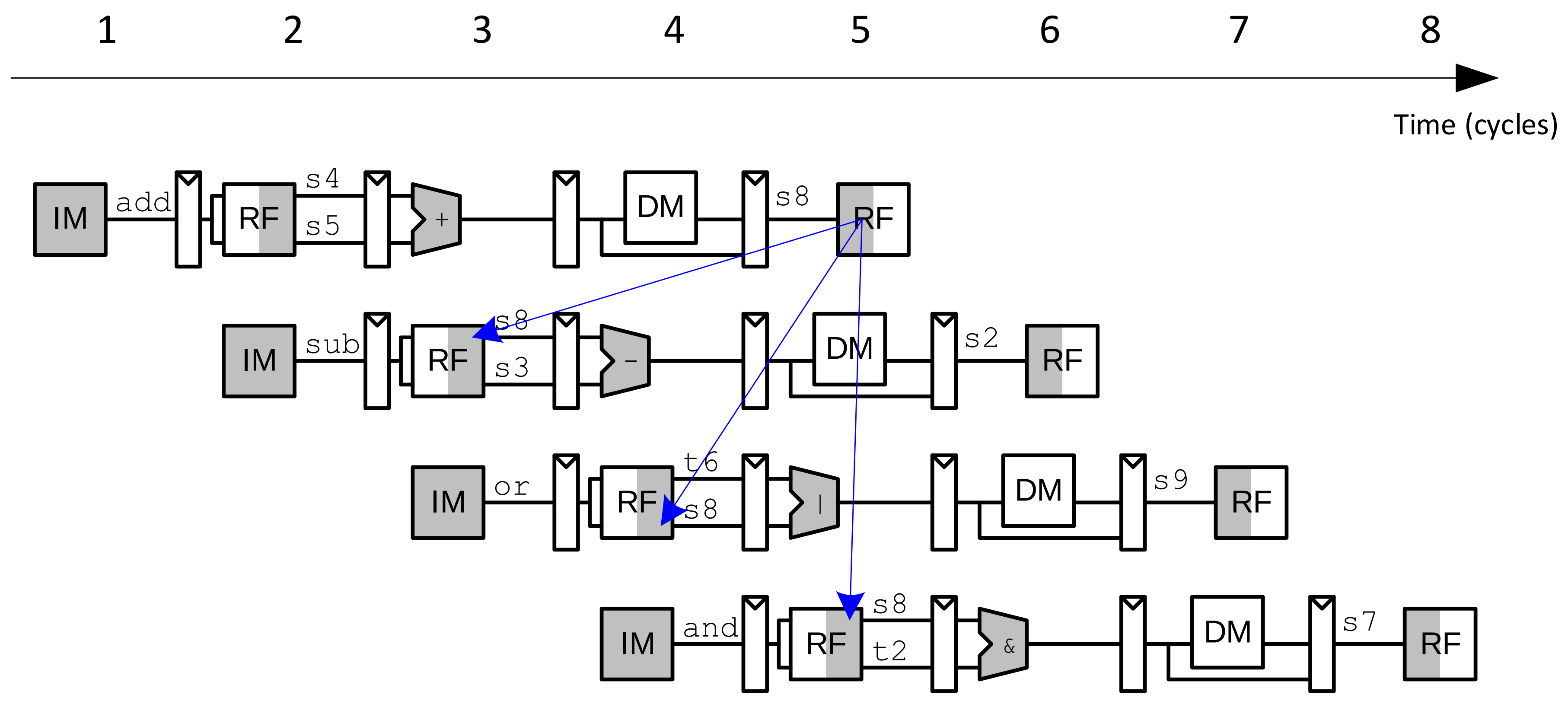
- When an instruction depends on result from instruction that hasn’t completed
- Types:
  - **Data hazard**:

    register value not yet written back to register file

  - **Control hazard**:

    next instruction not decided yet (caused by branch)

Data Hazard



```
add s8, s4, s5

sub s2, s8, s3

or  s9, t6, s8

and s7, s8, t2
```

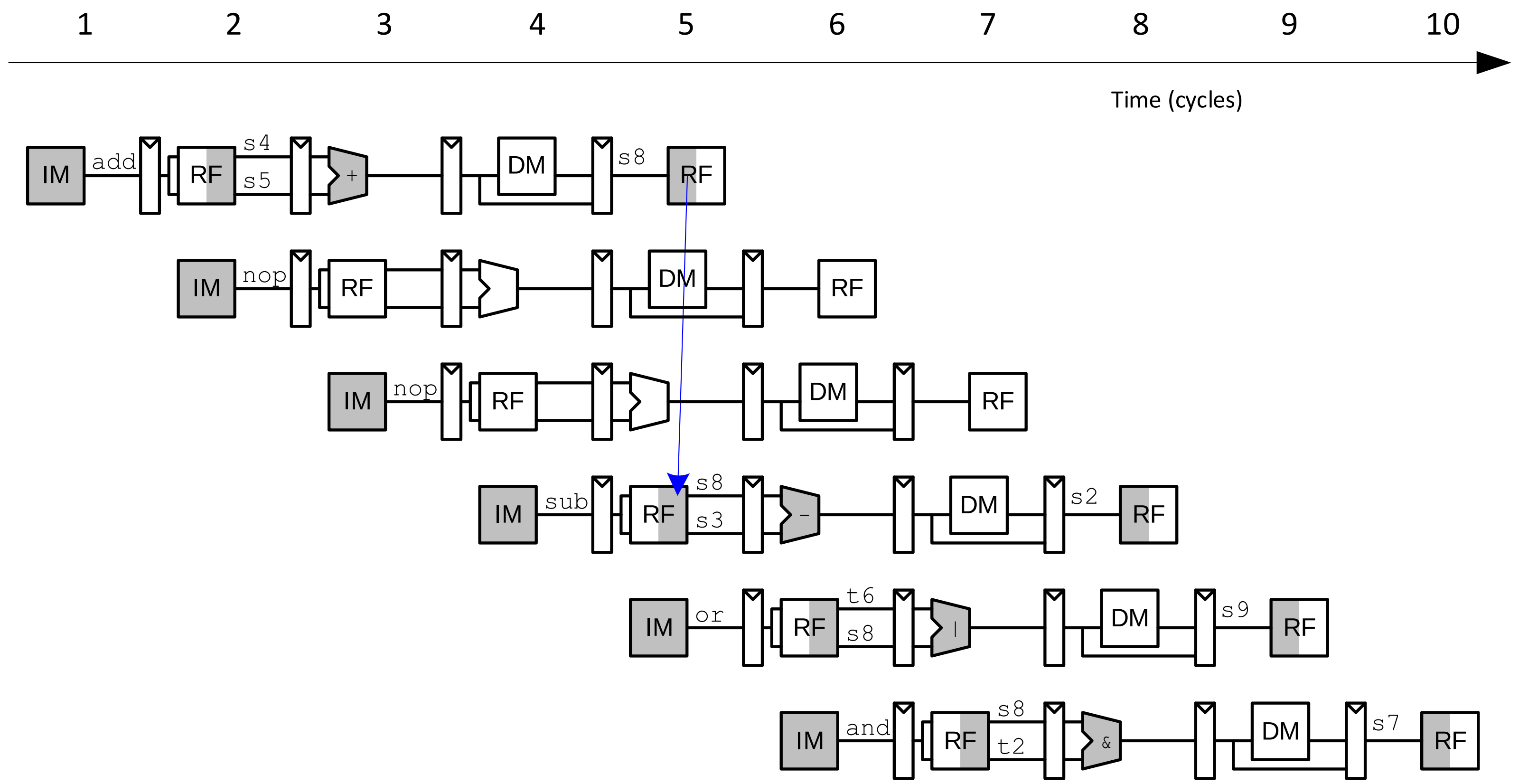
Handling Data Hazards

- Insert **nops** in code at compile time
- **Rearrange** code at compile time
- **Forward** data at run time
- **Stall** the processor at run time

Handling Data Hazards

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

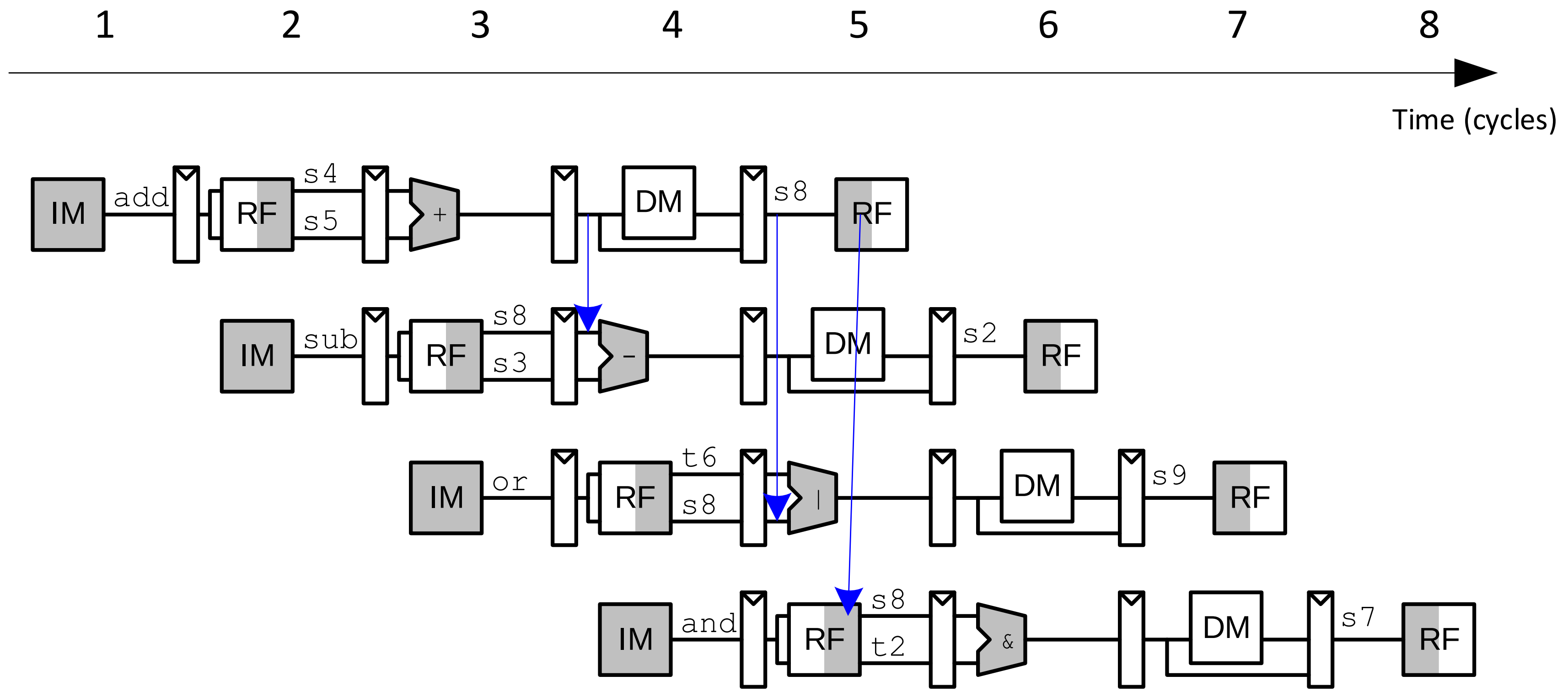
```
add s8, s4, s5

nop

nop

sub s2, s8, s3

or  s9, t6, s8

and s7, s8, t2
```



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**.

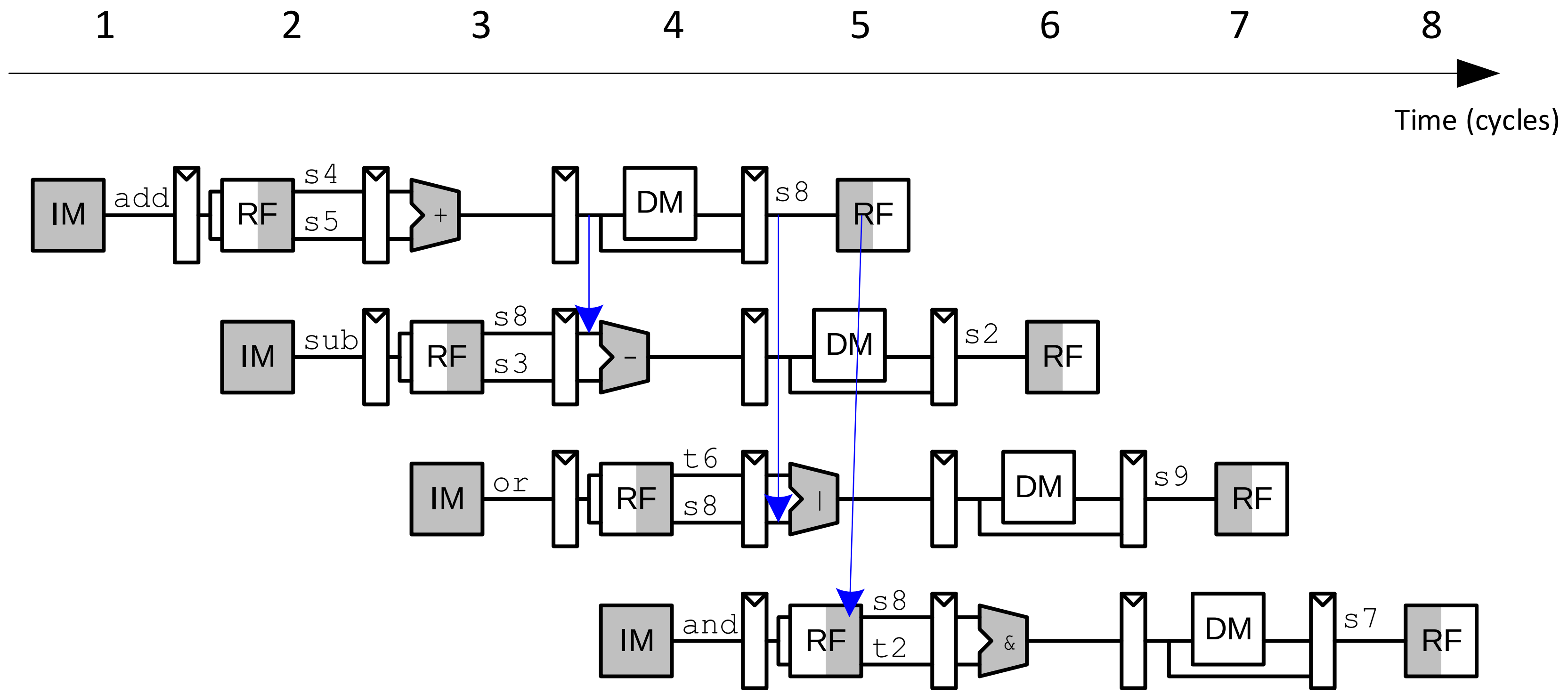
```
add s8, s4, s5

sub s2, s8, s3

or  s9, t6, s8

and s7, s8, t2
```



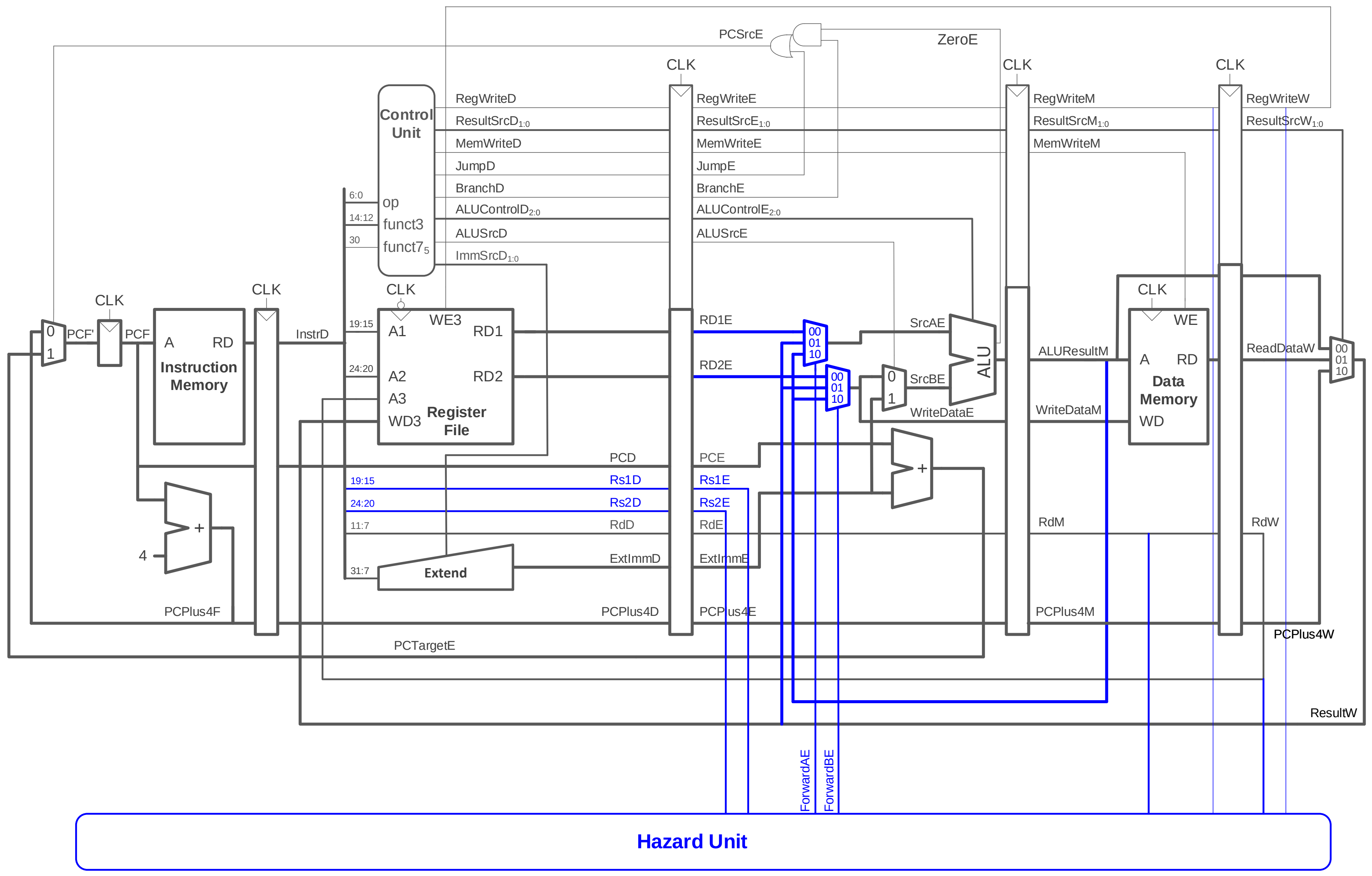
Data Forwarding

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

```
add s8, s4, s5

sub s2, s8, s3

or  s9, t6, s8

and s7, s8, t2
```



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)

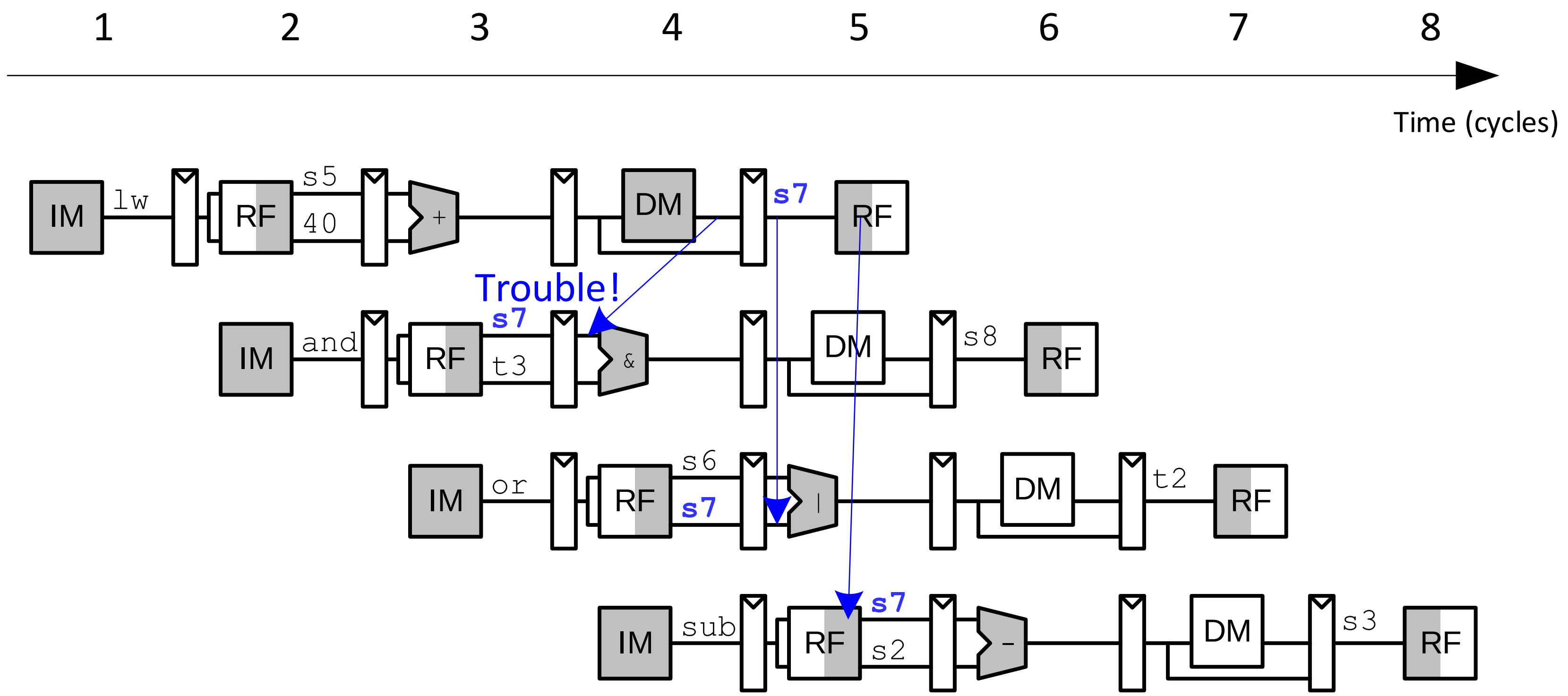
- **Equations for ForwardAE - *Rs1*:**

```
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**)

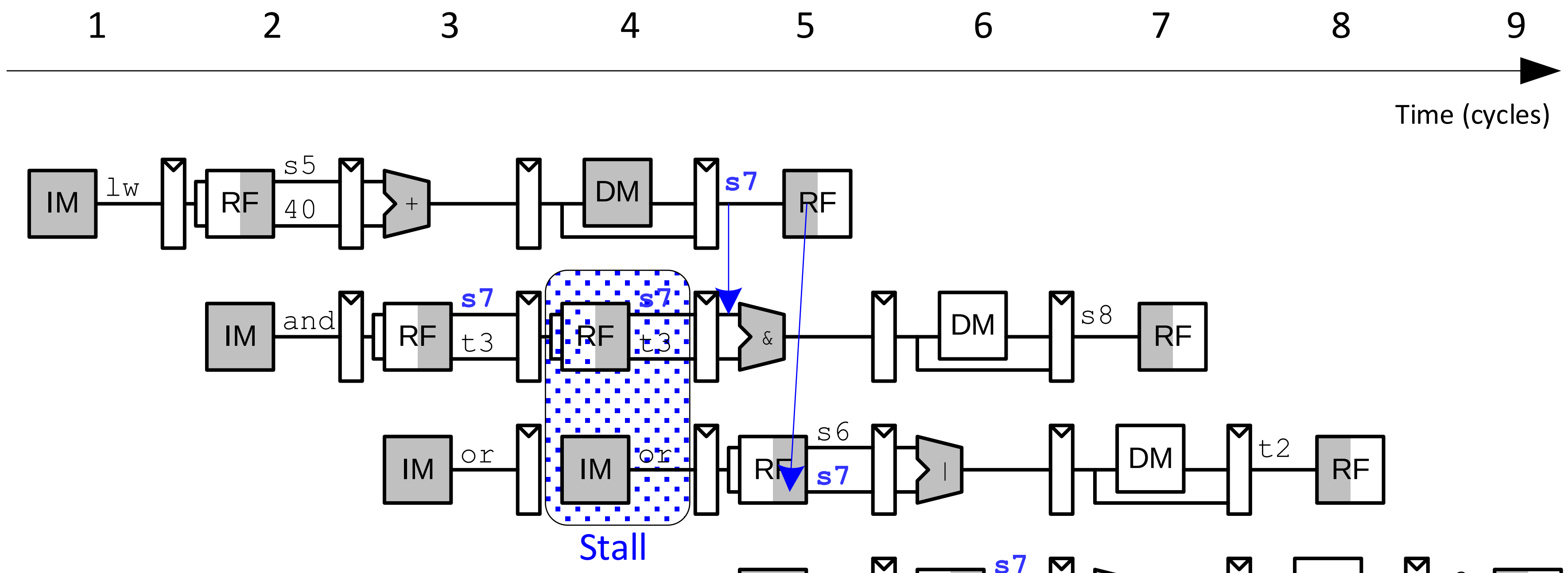
Data Hazard due to `lw` Dependency

```
lw   s7, 40(s5)

and s8, s7, t3

or   t2, s6, s7

sub s3, s7, s2
```



Data Hazard due to `lw` Dependency

```
lw   s7, 40(s5)
and  s8, s7, t3
or   t2, s6, s7
sub  s3, s7, s2
```

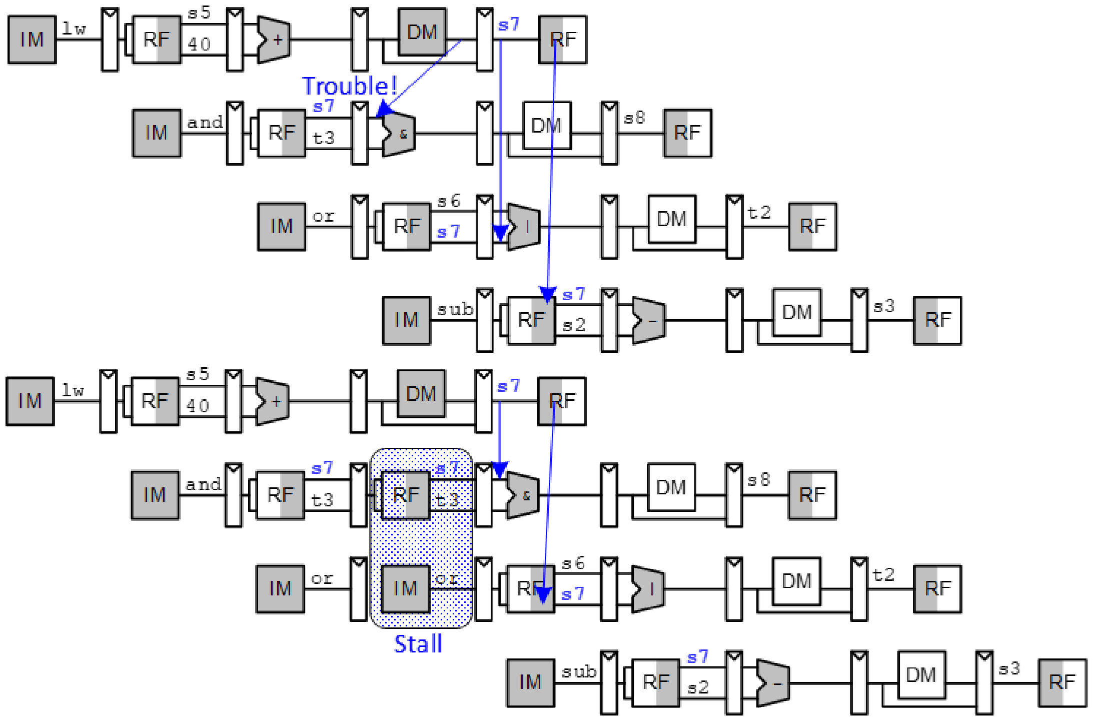


Stalling Logic

- 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**?

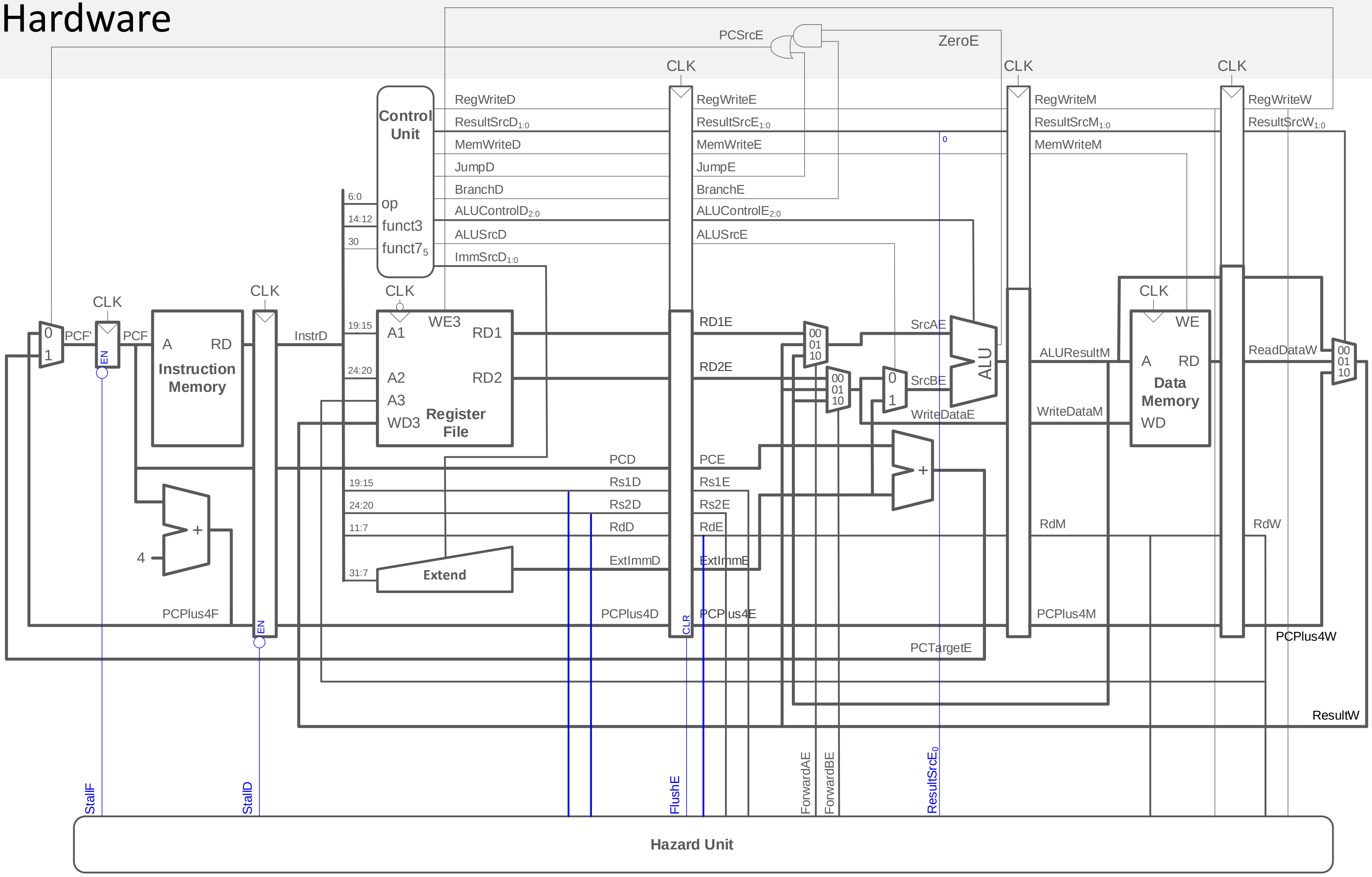


lwStall = ((*Rs1D* == *RdE*) OR (*Rs2D* == *RdE*)) **AND** ***ResultSrcE*$_0$**

StallF = ***StallD*** = ***FlushE*** = ***lwStall***

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

Stalling Hardware



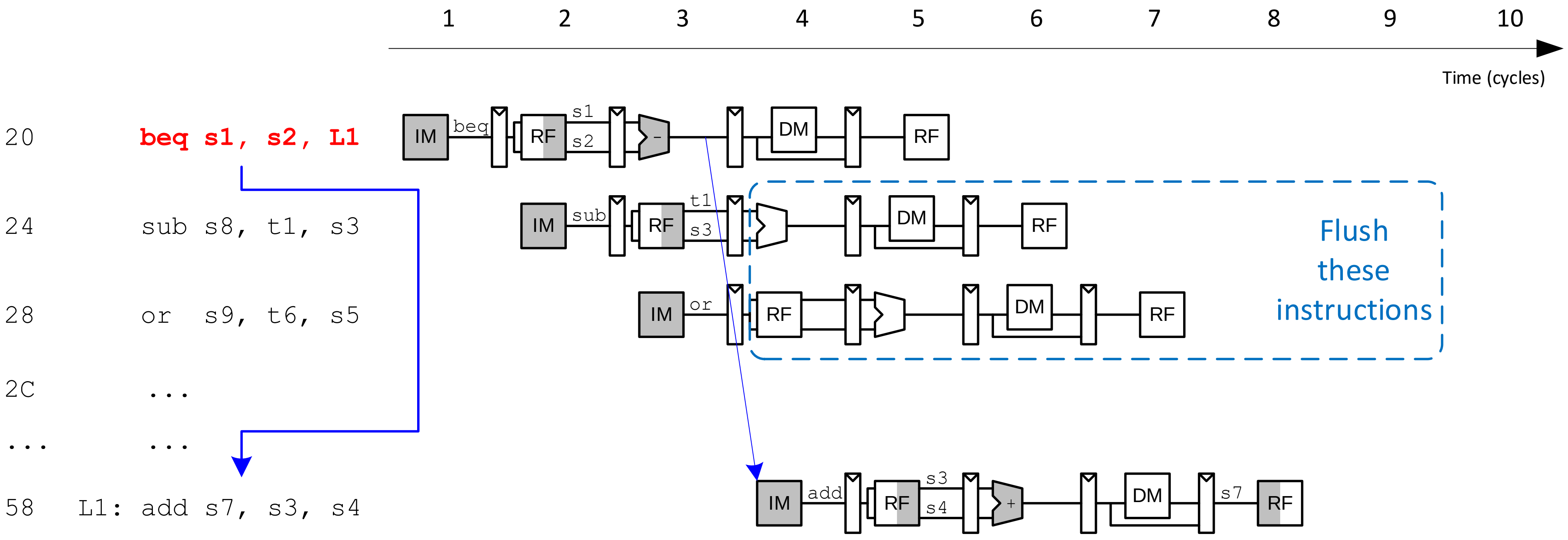
Pipelined Processor Control Hazards

DDCA Ch7 - Part 15: Pipelined Processor Control Hazards https://www.youtube.com/watch?v=VcnwVxD4LAc

Control Hazards

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

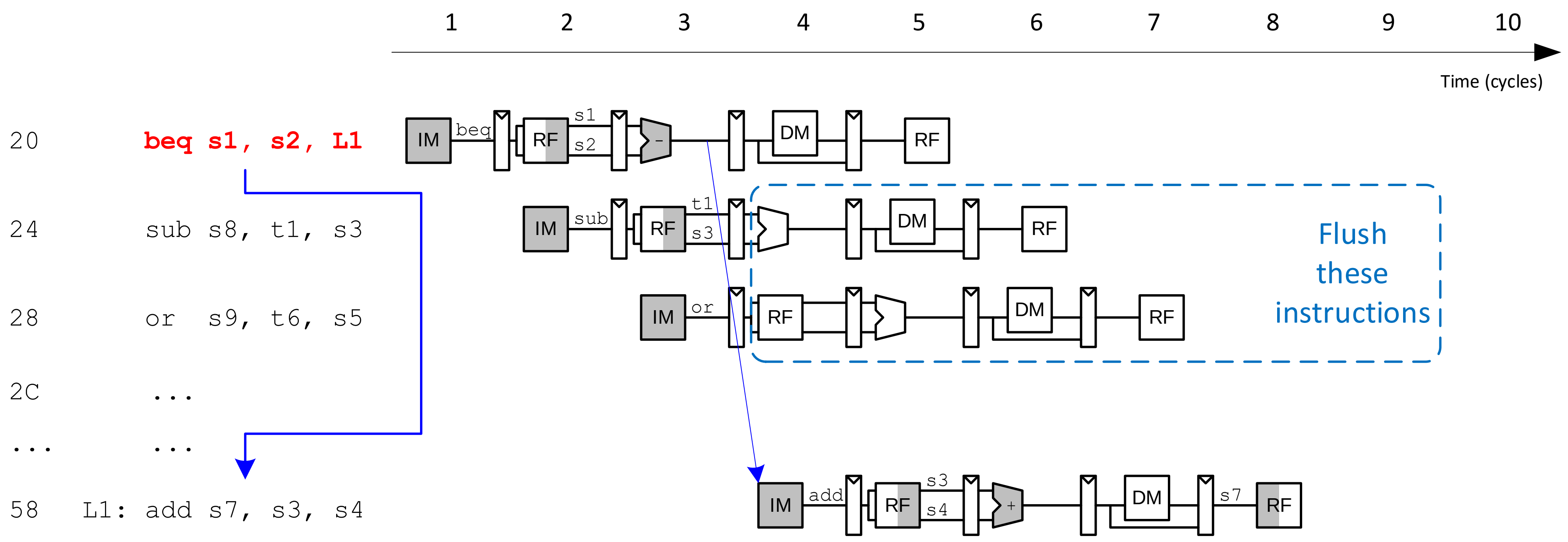
```
20        beq s1, s2, L1
24        sub s8, t1, s3
28        or  s9, t6, s5
2C        ...
...       ...
58    L1: add s7, s3, s4
```



Control Hazards

Branch misprediction penalty:

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



```
20        beq s1, s2, L1

24        sub s8, t1, s3

28        or  s9, t6, s5

2C        ...
...       ...
58   L1: add s7, s3, s4
```

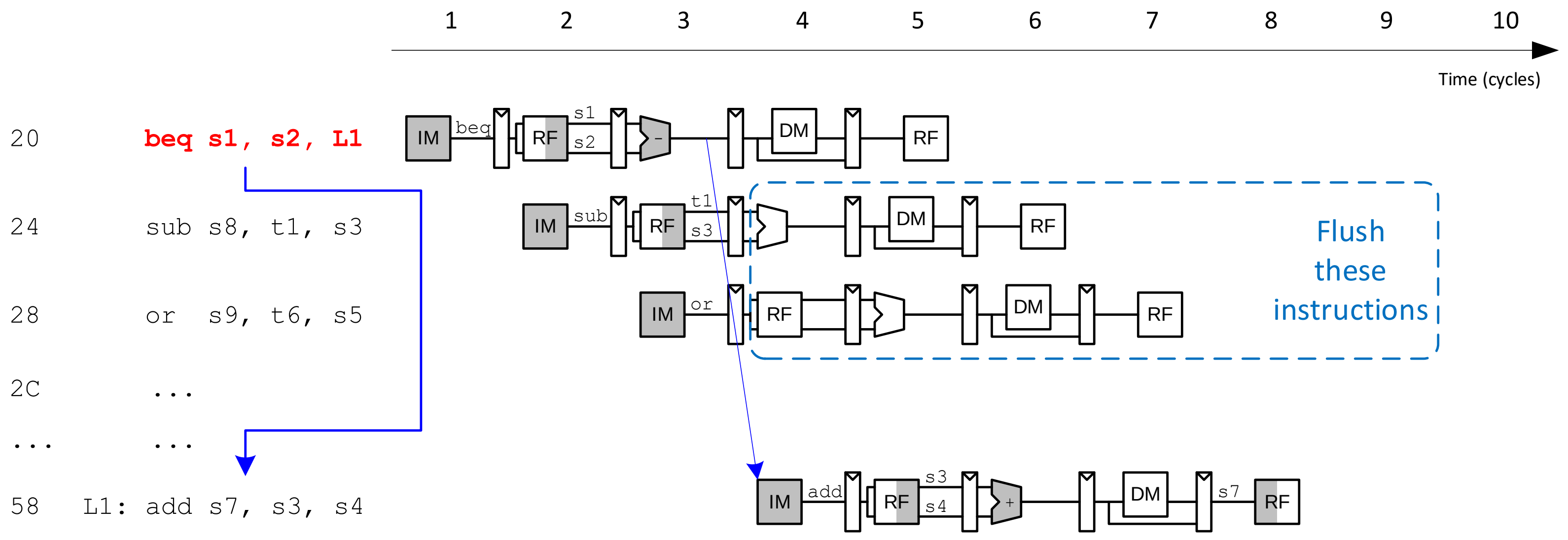
Control Hazards: Flushing Logic

- 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:**

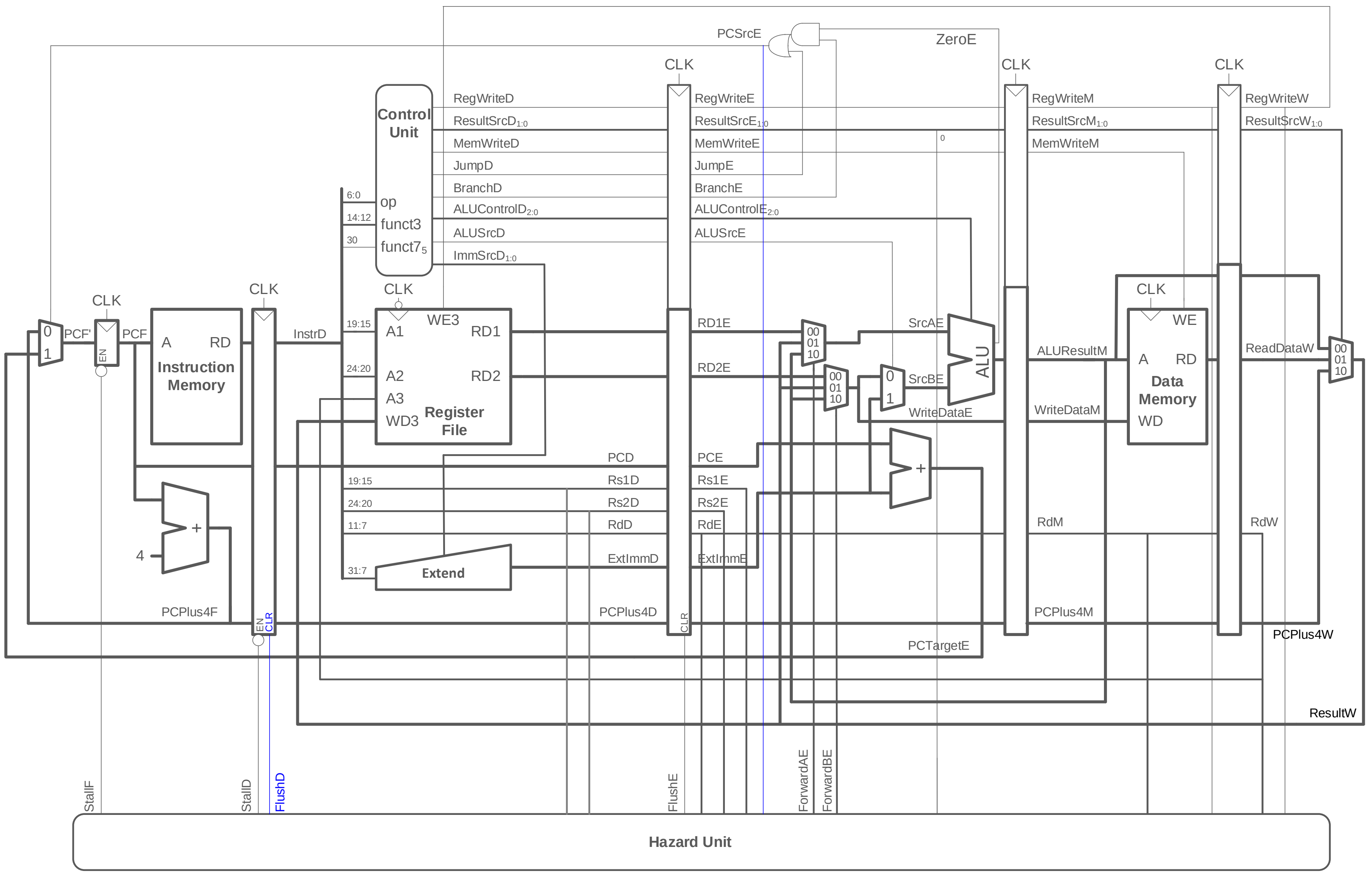
  FlushD = PCSrcE

  FlushE = lwStall OR *PCSrcE*

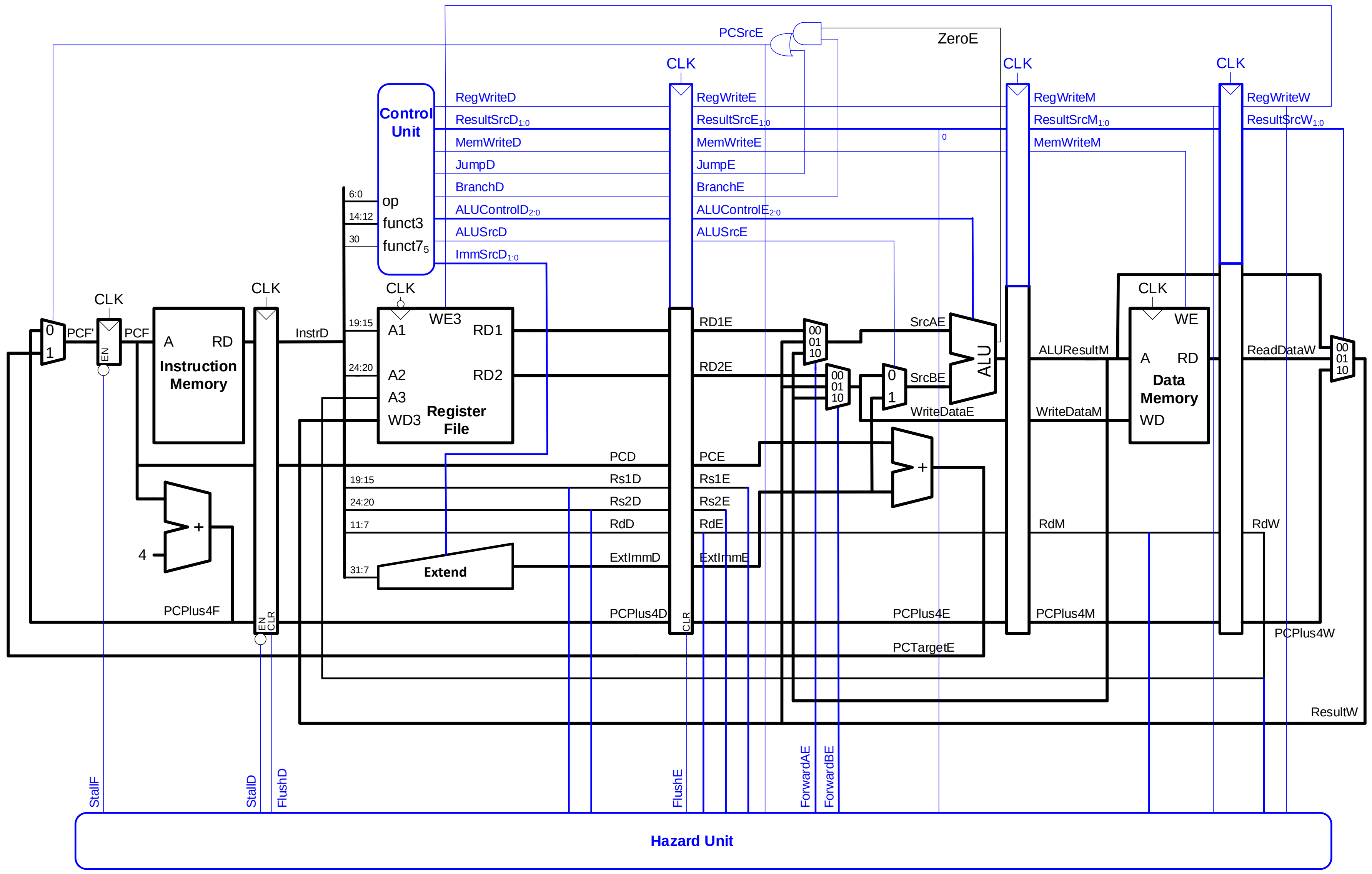


```
20       beq s1, s2, L1

24       sub s8, t1, s3

28       or  s9, t6, s5

2C       ...
...      ...

58   L1: add s7, s3, s4
```

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:

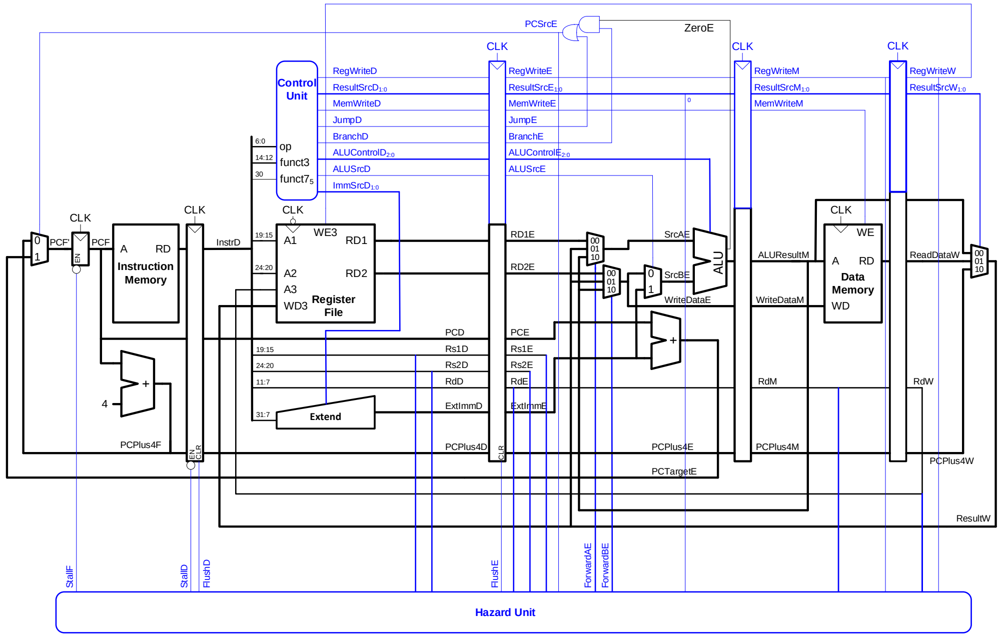
lwStall = ((*Rs1D* == *RdE*) OR (*Rs2D* == *RdE*)) AND $ResultSrcE_0$

StallF = ***StallD*** = *lwStall*

Control hazard flush:

FlushD = *PCSrcE*

FlushE = *lwStall* OR *PCSrcE*



Pipelined Performance

DDCA Ch7 - Part 14: Pipelined Processor Data Hazards https://www.youtube.com/watch?v=zuegcg6ZSFQ

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…)
  - Load CPI = 1 when not stalling, 2 when stalling → So, $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
  - Branch CPI = 1 when not stalling, 3 when stalling → So, $CPI_{beq} = 1(0.5) + 3(0.5) = 2$

  - **Average CPI =** (0.25)(1.4) + (0.1)(1) + (0.13)(2) + (0.52)(1) **= 1.23**

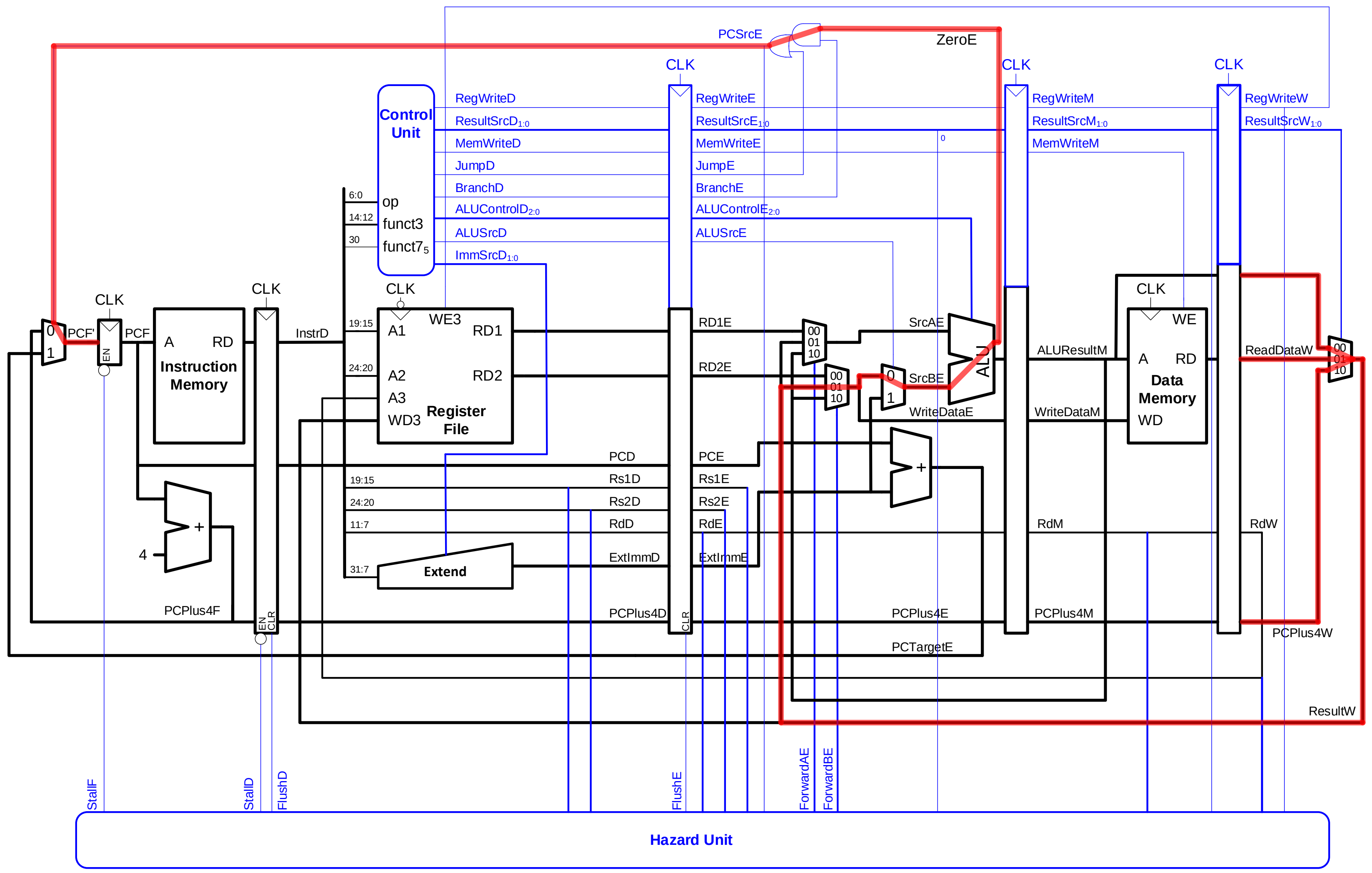
Pipelined Processor Performance Example

- Pipelined processor critical path:

$T_{c\_pipelined}$	$= \max \text{ of } [$	
	$t_{pcq} + t_{\text{mem}} + t_{\text{setup}}$	**Fetch**
	$2(t_{\text{RFread}} + t_{\text{setup}})$	**Decode**
	$t_{pcq} + 4t_{\text{mux}} + t_{\text{ALU}} + t_{\text{AND-OR}} + t_{\text{setup}}$	**Execute**
	$t_{pcq} + t_{\text{mem}} + t_{\text{setup}}$	**Memory**
	$2(t_{pcq} + t_{\text{mux}} + t_{\text{RFwrite}})\ ]$	**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	$t_{\mathrm{AND\text{-}OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{dec}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\mathrm{read}}$	100
Register file setup	$t_{RF\mathrm{setup}}$	60

$$\begin{aligned} T_{c\_pipelined} &= t_{pcq} + 4t_{\mathrm{mux}} + t_{\mathrm{ALU}} + t_{AND\text{-}OR} + t_{\mathrm{setup}} \\ &= (40 + 4*30 + 120 + 20 + 50)\ \mathrm{ps} = \mathbf{350\ ps} \end{aligned}$$

Pipelined Performance Example

Program with 100 billion instructions

Execution Time **= (# instructions) × CPI × Tc**

= (100 × 109)(1.23)(350 × 10-12)

= 43 seconds

Processor Performance Comparison

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