EECE 2322: Fundamentals of Digital Design and Computer Organization

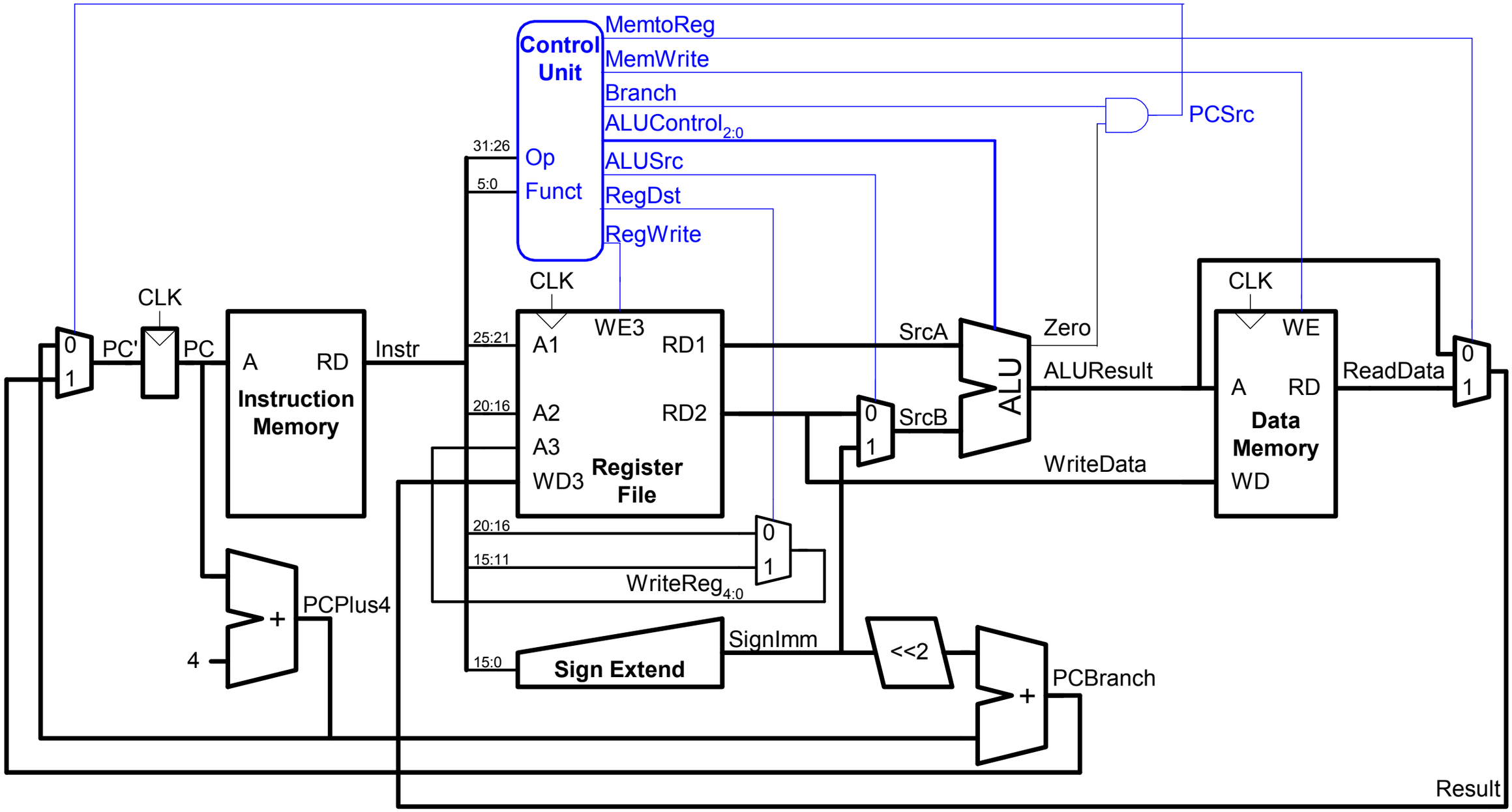
Lecture 13_2: Microarchitecture

Xiaolin Xu
Department of ECE
Northeastern University

Extended Functionality: addi

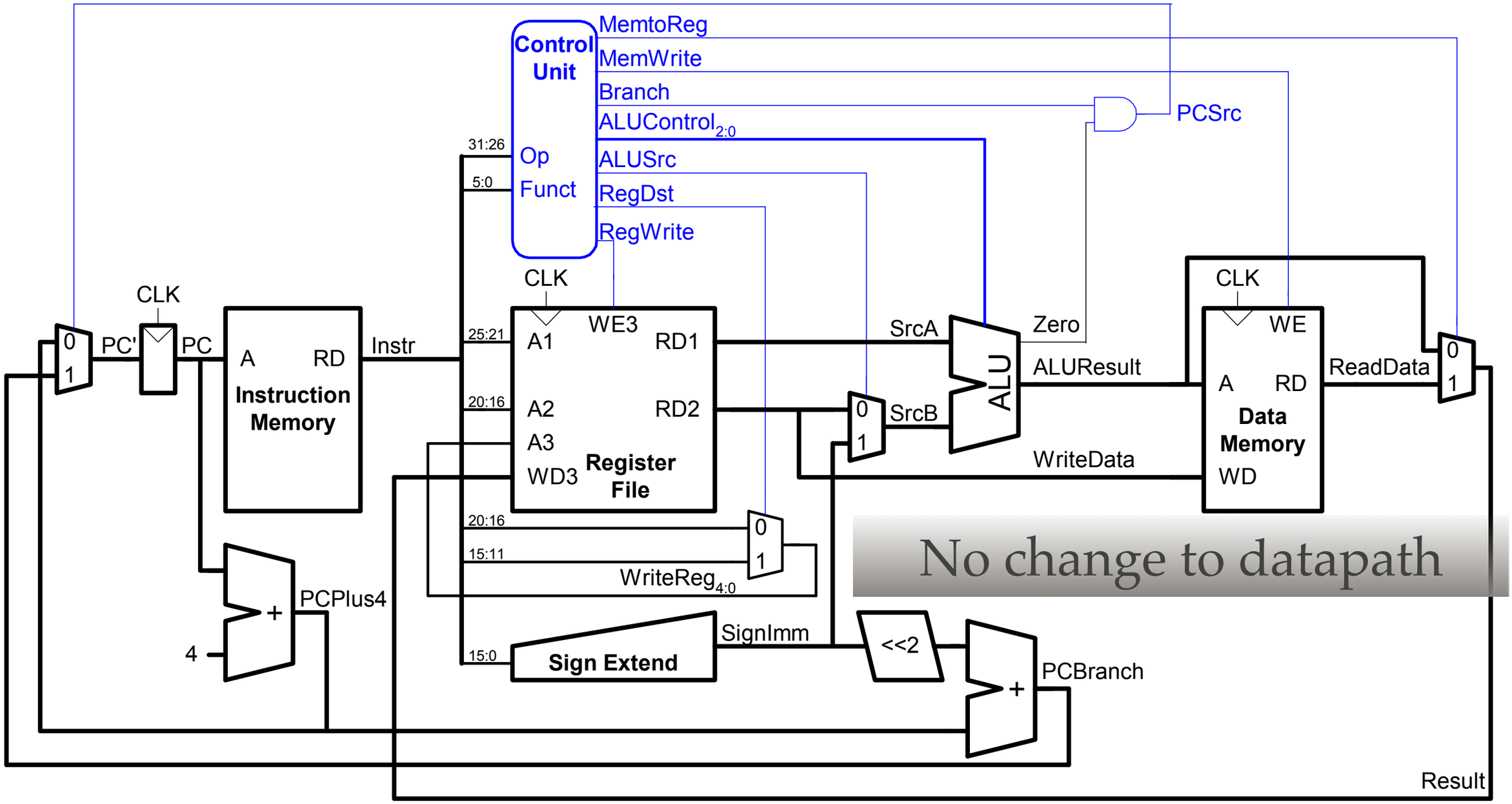
and immediate	`andi $1,$2,100`	$1=$2&100	Bitwise AND with immediate value

Extended Functionality: addi



and immediate	`andi $1,$2,100`	$1=$2&100	Bitwise AND with immediate value

Extended Functionality: addi



No change to datapath

and immediate	`andi $1,$2,100`	$1=$2&100	Bitwise AND with immediate value

Control Unit: addi

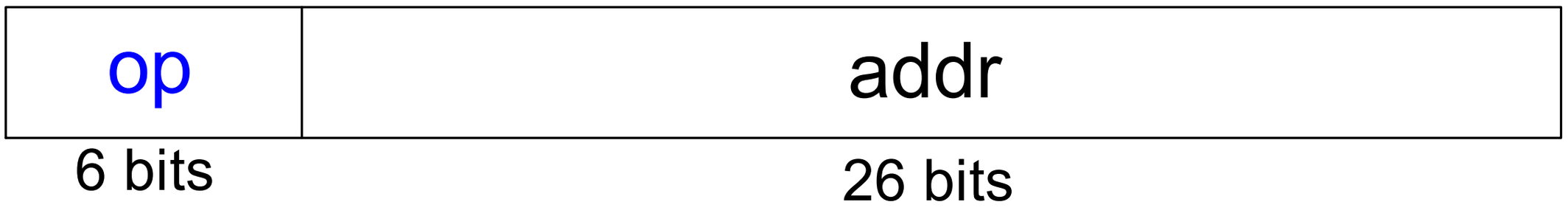
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01
`addi`	**001000**	**1**	**0**	**1**	**0**	**0**	**0**	**00**

Extended Functionality: j

Machine Language: J-Type

- *Jump-type:* `j/jal`
- Used for jump instructions (`j`)
  - 26-bit address operand (`addr`)
- Jump and link (`jal`)

J-Type



R-Type: Call Function: jump and link (`jal`**`r`**),

Return from function: jump register (`j`**`r`**)

Unconditional Branching: jump (j)

```
# MIPS assembly
  addi $s0, $0, 4          # $s0 = 4
  addi $s1, $0, 1          # $s1 = 1
  j    target              # jump to target
  sra  $s1, $s1, 2
  addi $s1, $s1, 1
  sub  $s1, $s1, $s0

target:
  add  $s1, $s1, $s0
```

Unconditional Branching: jump (j)

```
# MIPS assembly
  addi $s0, $0, 4          # $s0 = 4
  addi $s1, $0, 1          # $s1 = 1
  j    target              # jump to target
  sra  $s1, $s1, 2         # not executed
  addi $s1, $s1, 1         # not executed
  sub  $s1, $s1, $s0       # not executed

target:
  add  $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Revisit: Pseudo-direct Addressing

- Why pseudo-direct Addressing: no enough address bit!
  - Specifically used for J-type instructions , j and jal
- Jump target address (JTA) needs 32-bit, but only 26-bit available. How to achieve this?
  - The two least significant bits (1:0) of JTA are 0s and can be saved
  - The middle 27:2 are directly applied
  - The four most significant bits (31:28) are borrowed from PC+4

Revisit: Pseudo-direct Addressing

Pseudo-direct Addressing

```
0x0040005C          jal    sum
...
0x004000A0  sum:    add    $v0, $a0, $a1
```

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

Field Values

op	imm
3	0x0100028
6 bits	26 bits

Machine Code

op	addr	
000011	00 0001 0000 0000 0000 0010 1000	(0x0C100028)
6 bits	26 bits	

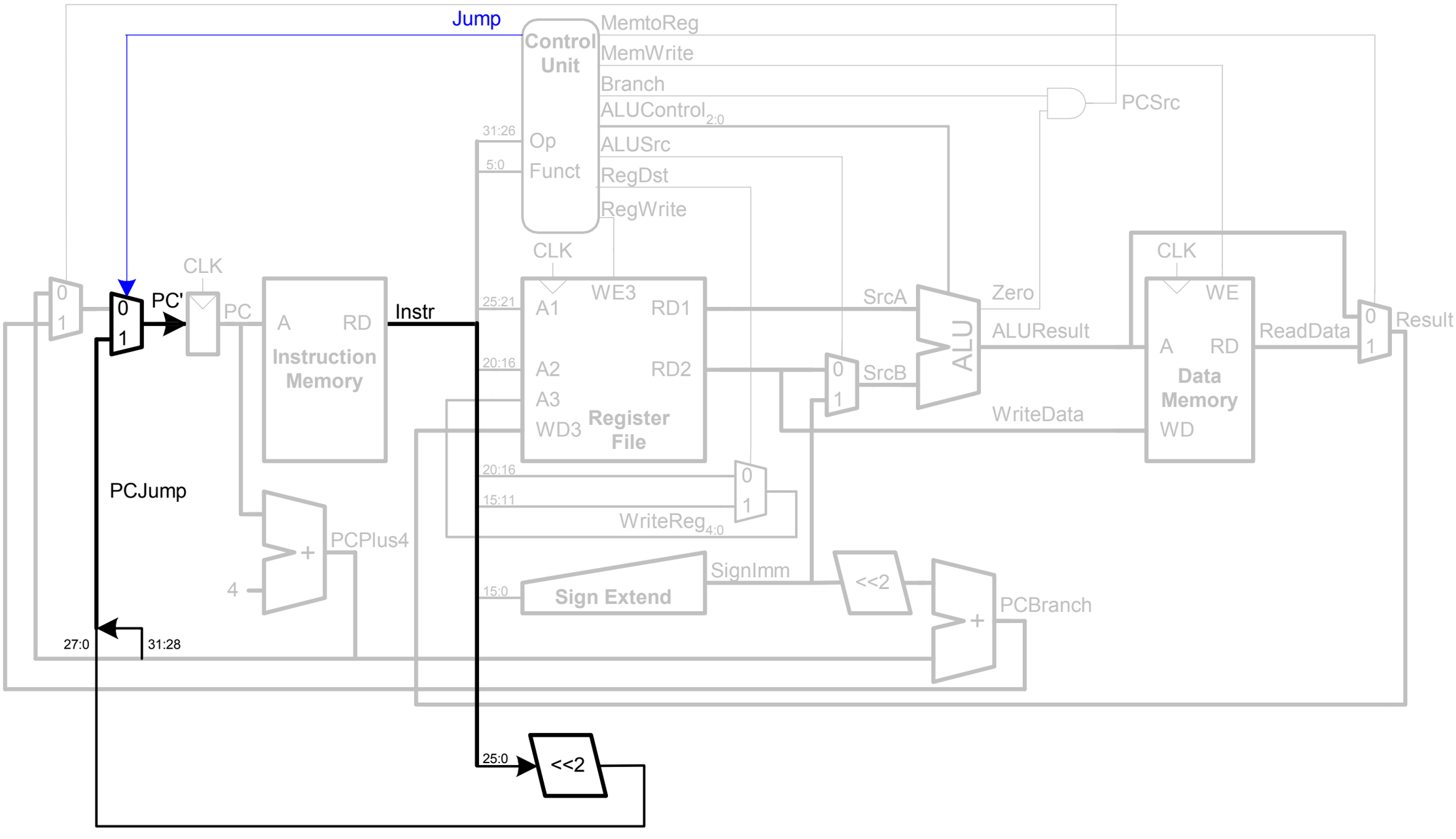
Revisit: : Pseudo-direct Addressing

- The effective address will always be word-aligned
  - The two least significant bits are 00
- The range of jump target is constrained
  - Anywhere within the current 256 MB block of code
  - Since the upper 4 bits of the PC are used
- What if to jump anywhere within the 4 GB space

Revisit: : Pseudo-direct Addressing

- The effective address will always be word-aligned
  - The two least significant bits are 00
- The range of jump target is constrained
  - Anywhere within the current 256 MB block of code
  - Since the upper 4 bits of the PC are used
- What if to jump anywhere within the 4 GB space
  - R-type instructions jr and jalr are used , where the complete 32 - bit target address is specified in a register

Extended Functionality: j



Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`j`	**000100**								

Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	**0**
`lw`	100011	1	0	1	0	0	1	00	**0**
`sw`	101011	0	X	1	0	1	X	00	**0**
`beq`	000100	0	X	0	1	0	X	01	**0**
`j`	**000100**	**0**	**X**	**X**	**X**	**0**	**X**	**XX**	**1**

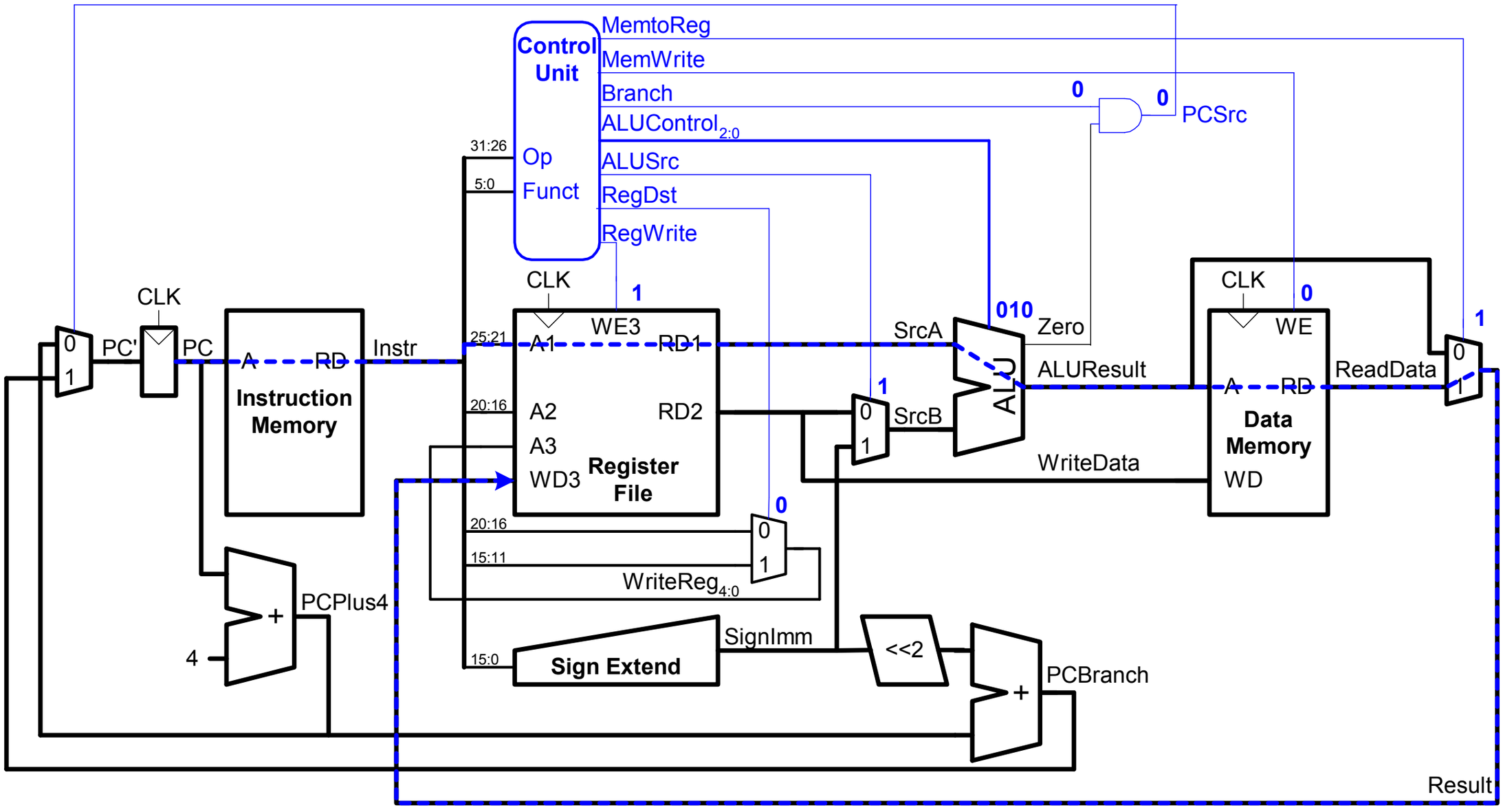
Processor Performance

Program Execution Time

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

= # instructions x CPI x T_C

Single-Cycle Performance: lw



Single-cycle critical path: T_C limited by critical path (`lw`)

$$T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

Single-Cycle Performance

- Single-cycle critical path:

$$T_c = t_{pcq\_PC} + t_{\text{mem}} + \max(t_{RF\text{read}}, t_{sext} + t_{\text{mux}}) + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

In most implementation technologies, the ALU, memory, and register file accesses are substantially slower than other operations

- Typically, limiting paths are:
  - memory, ALU, register file
  - $T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$$
$$= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}$$
$$= 925 \text{ ps}$$

Single-Cycle Performance Example

Program with 100 billion instructions:

Execution Time = # instructions x CPI x T_C

= $(100 \times 10^9)(1)(925 \times 10^{-12}\ \text{s})$

= **92.5 seconds**

Multicycle MIPS Processor

- **What we get from the Single-cycle:**

  + simple
  - cycle time limited by longest instruction (`lw`)
  - 2 adders/ALUs & 2 memories

Multicycle MIPS Processor

- **What we get from the Single-cycle:**

  + simple
  - cycle time limited by longest instruction (`lw`)
  - 2 adders/ALUs & 2 memories

- **Multicycle:**

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

Multicycle MIPS Processor

- **What we get from the Single-cycle:**

  + simple
  - cycle time limited by longest instruction (`lw`)
  - 2 adders/ALUs & 2 memories

- **Multicycle:**

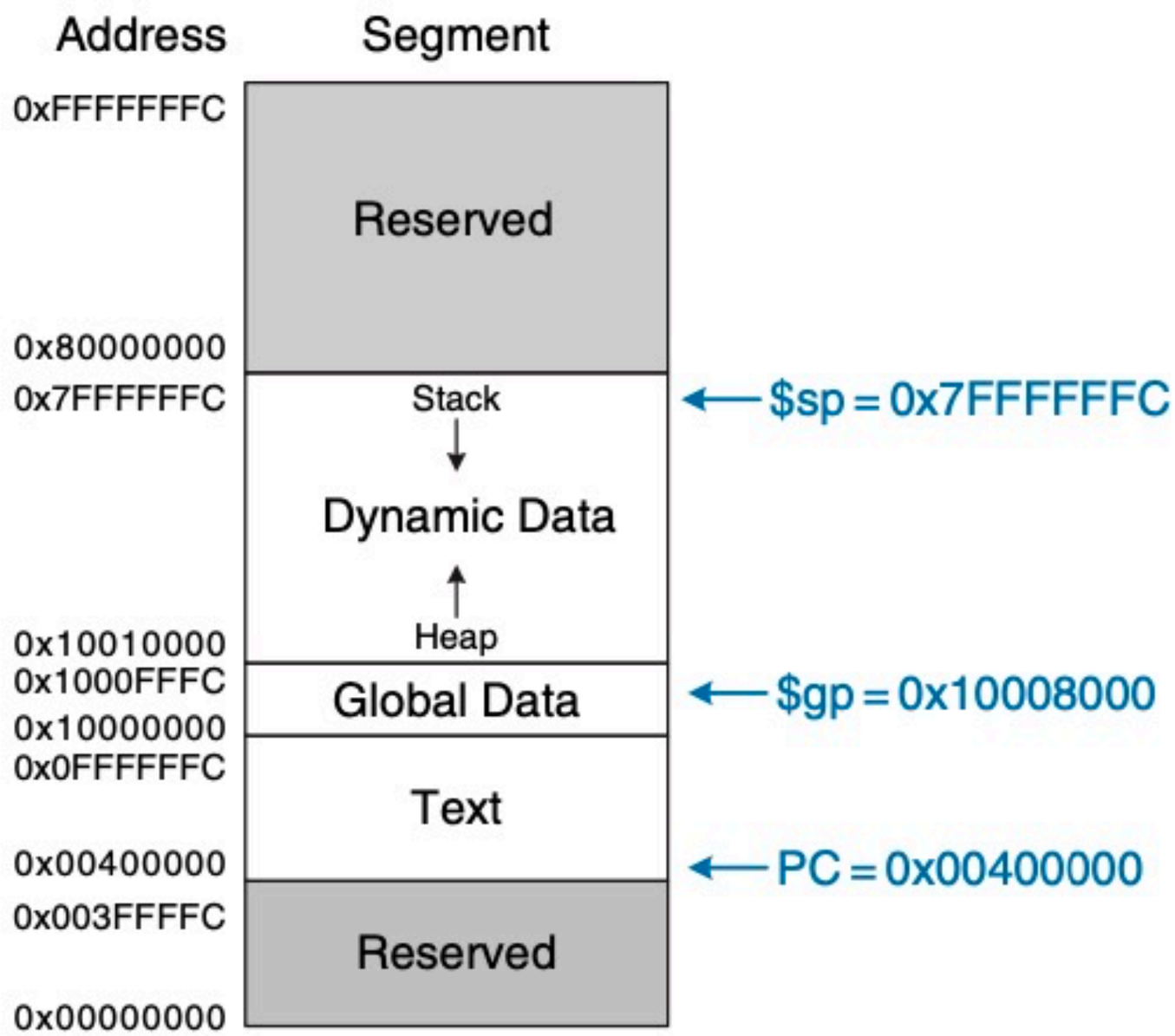
  + higher clock speed
  + simpler instructions run faster
  + reuse expensive hardware on multiple cycles
  - sequencing overhead paid many times

- **Same design steps: datapath & control**

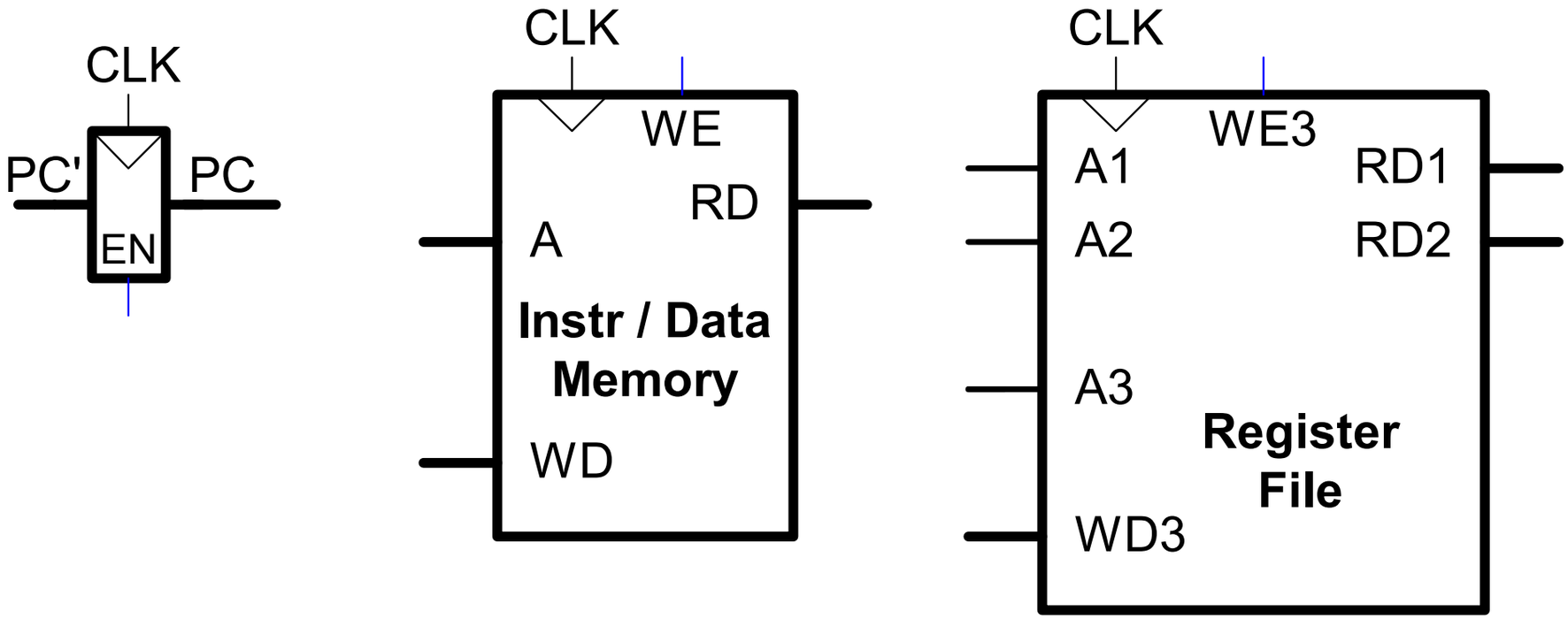
Memory Combination

What is Stored in Memory and Memory Map

- Instructions (also called *text*)
- Data
  - Global/static: allocated before program begins
  - Dynamic: allocated within program



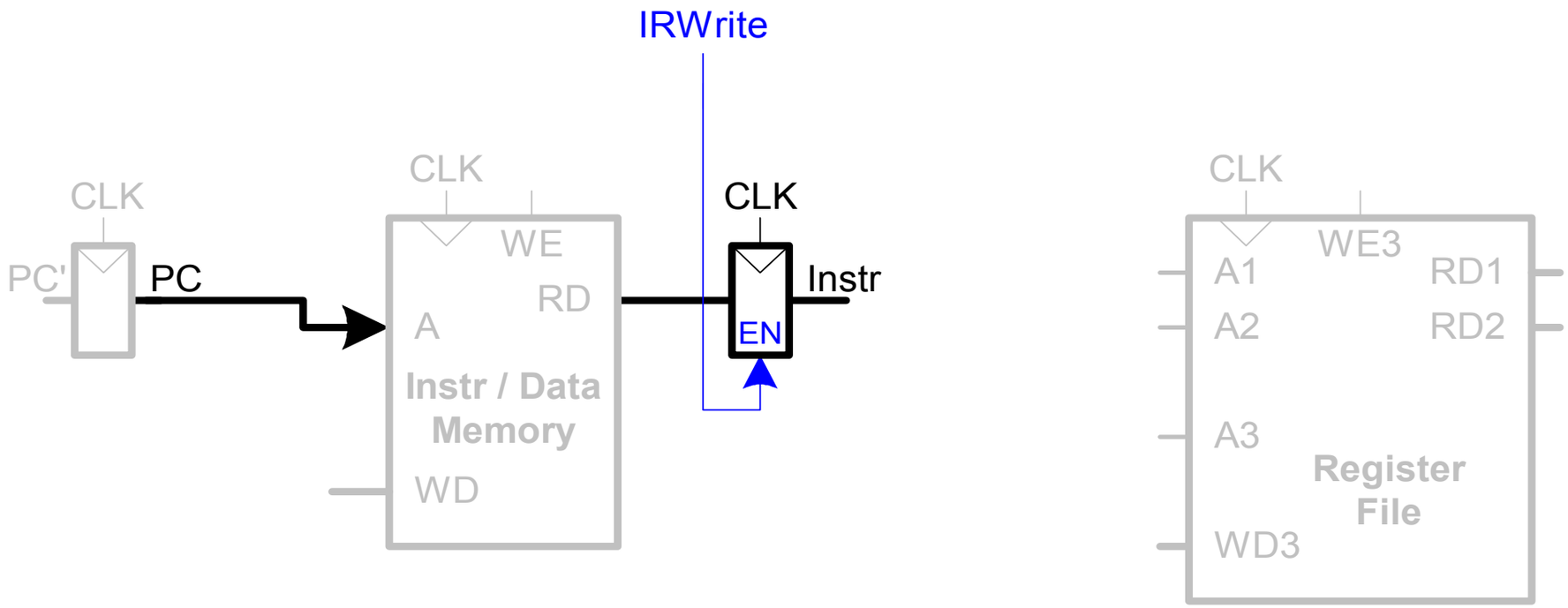
Multicycle State Elements



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

Multicycle Datapath: Instruction Fetch

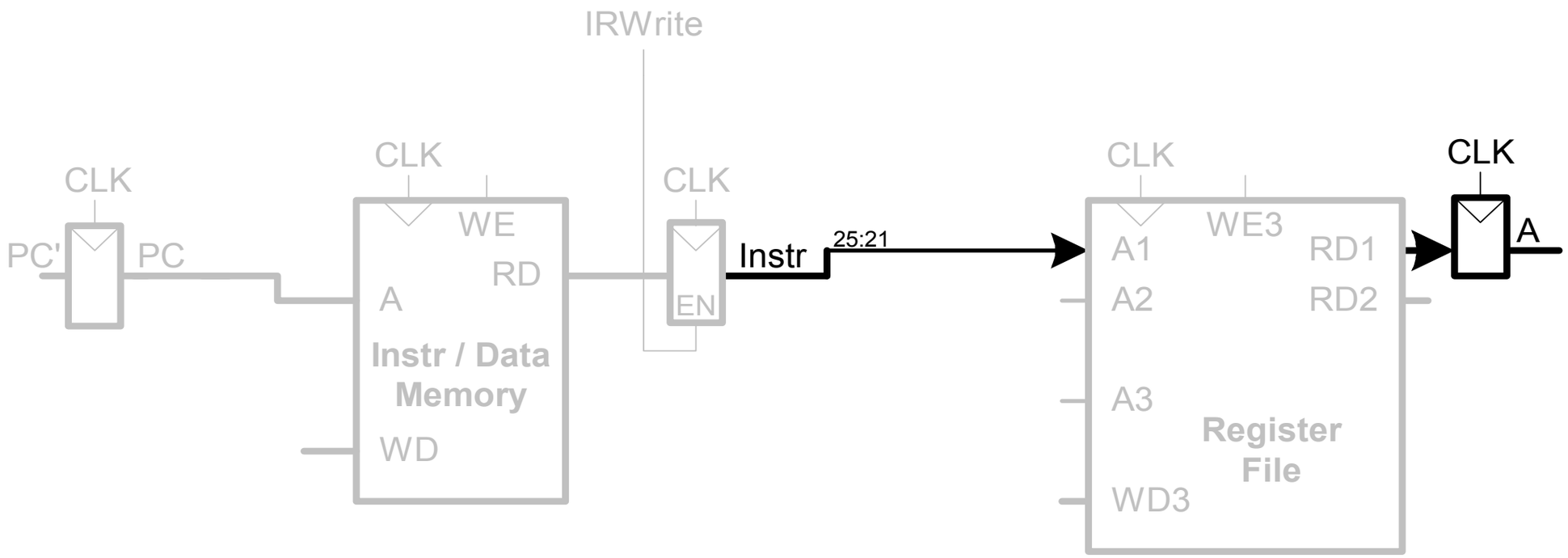
STEP 1: Fetch instruction



- IRWrite: Instruction Register enable signal

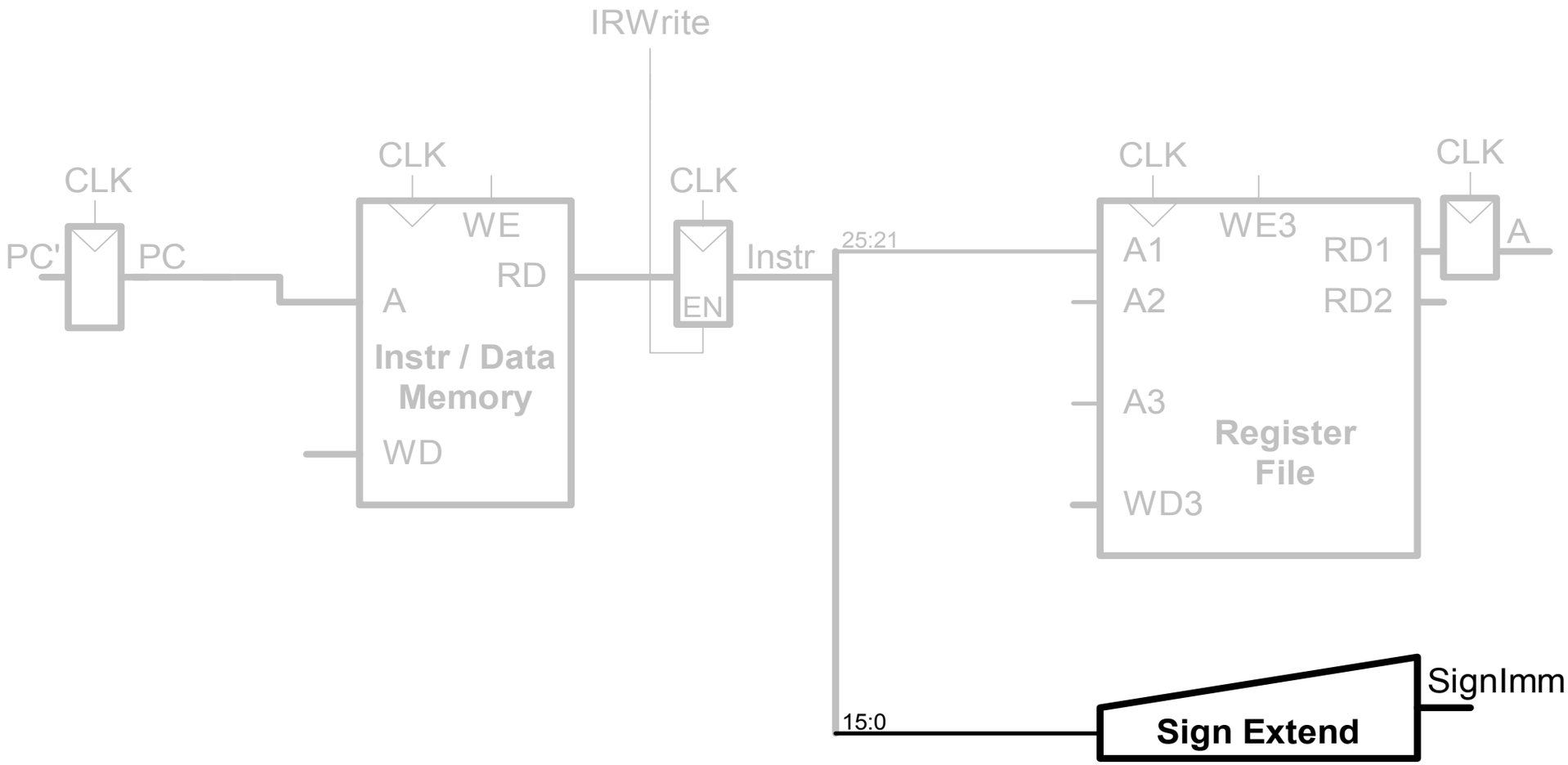
Multicycle Datapath: lw Register Read

STEP 2a: Read source operands from RF



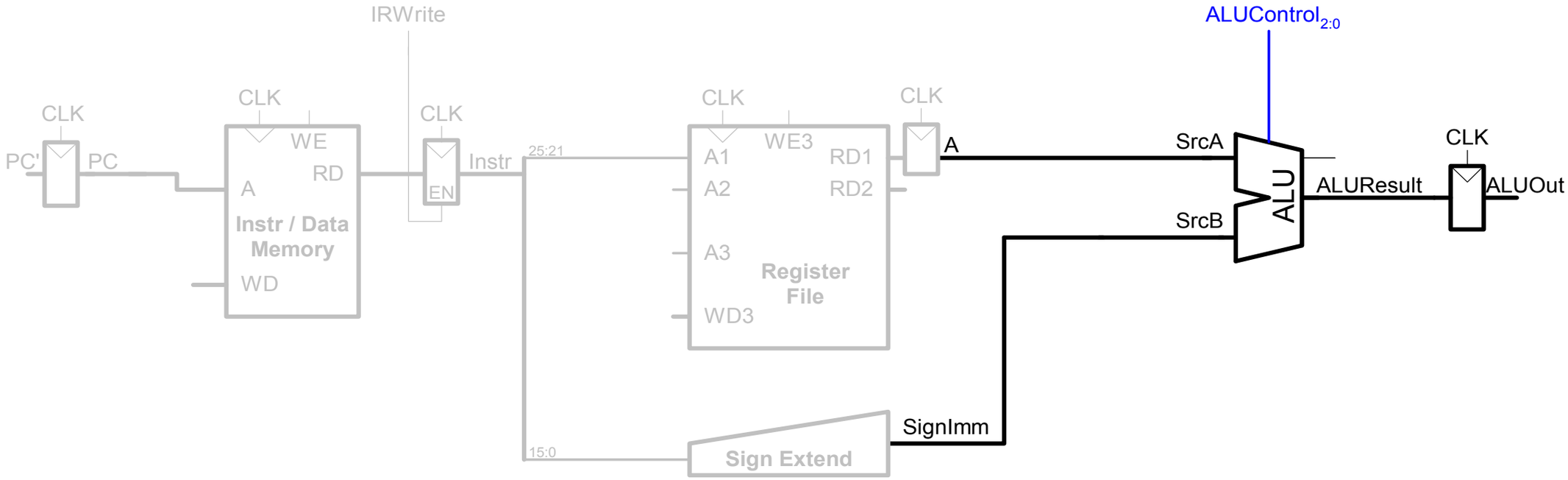
Multicycle Datapath: lw Immediate

STEP 2b: Sign-extend the immediate



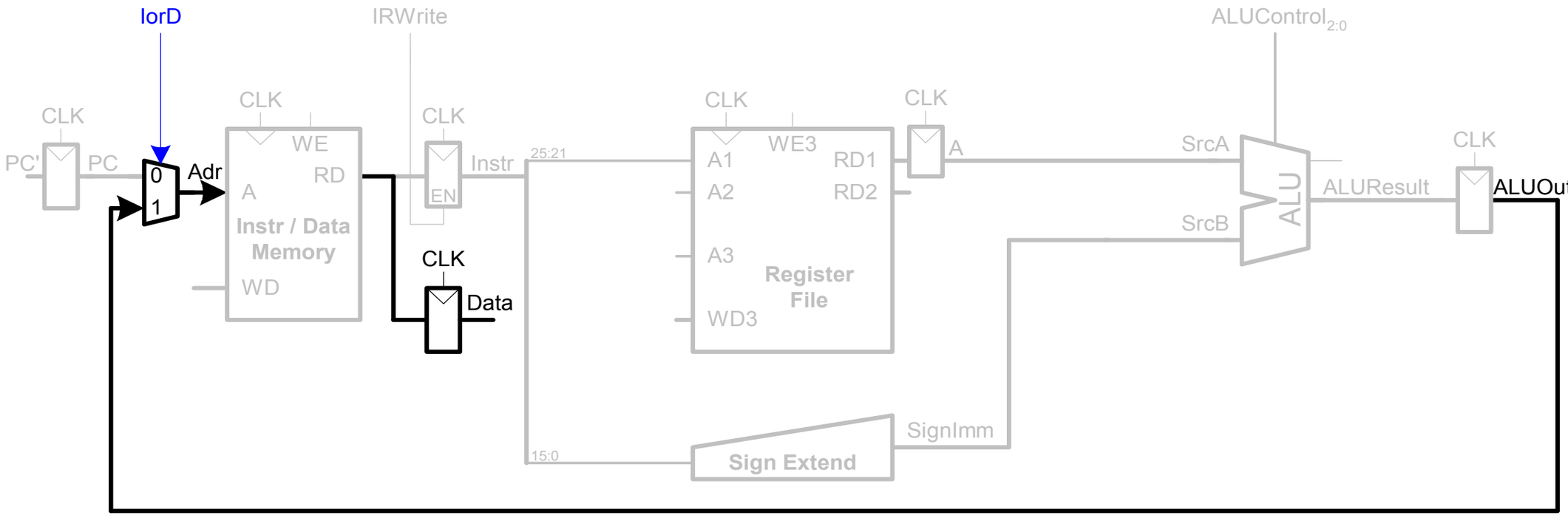
Multicycle Datapath: lw Address

STEP 3: Compute the memory address



Multicycle Datapath: lw Memory Read

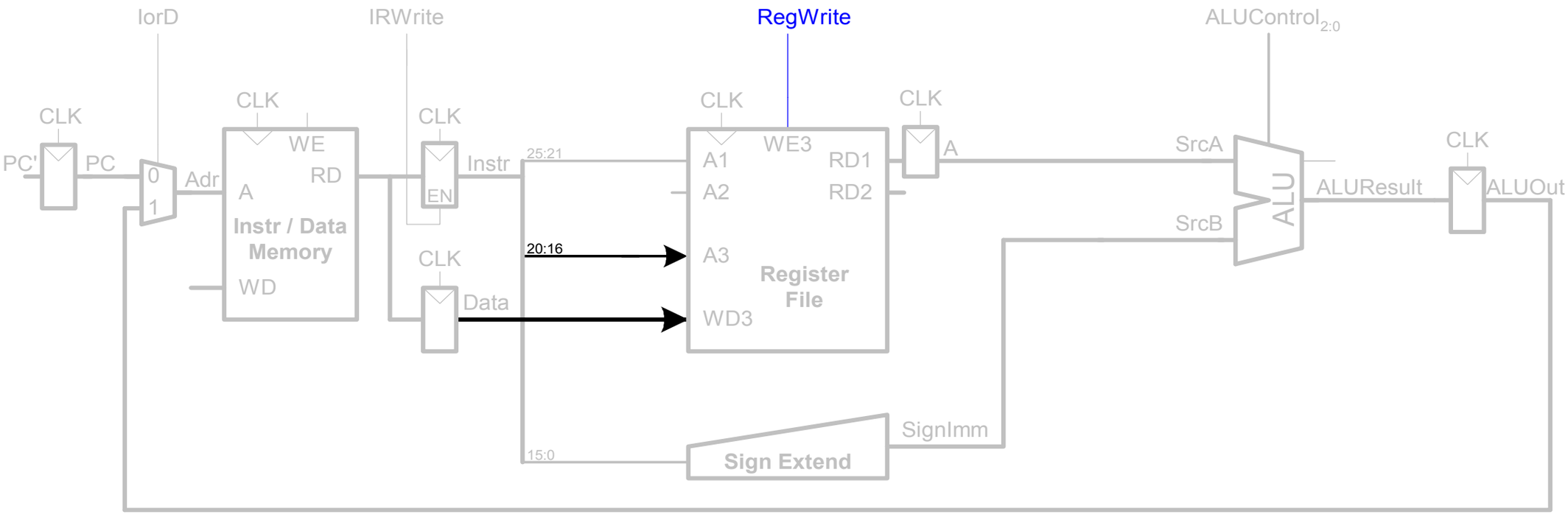
STEP 4: Read data from memory



- IorD: instruction or data address

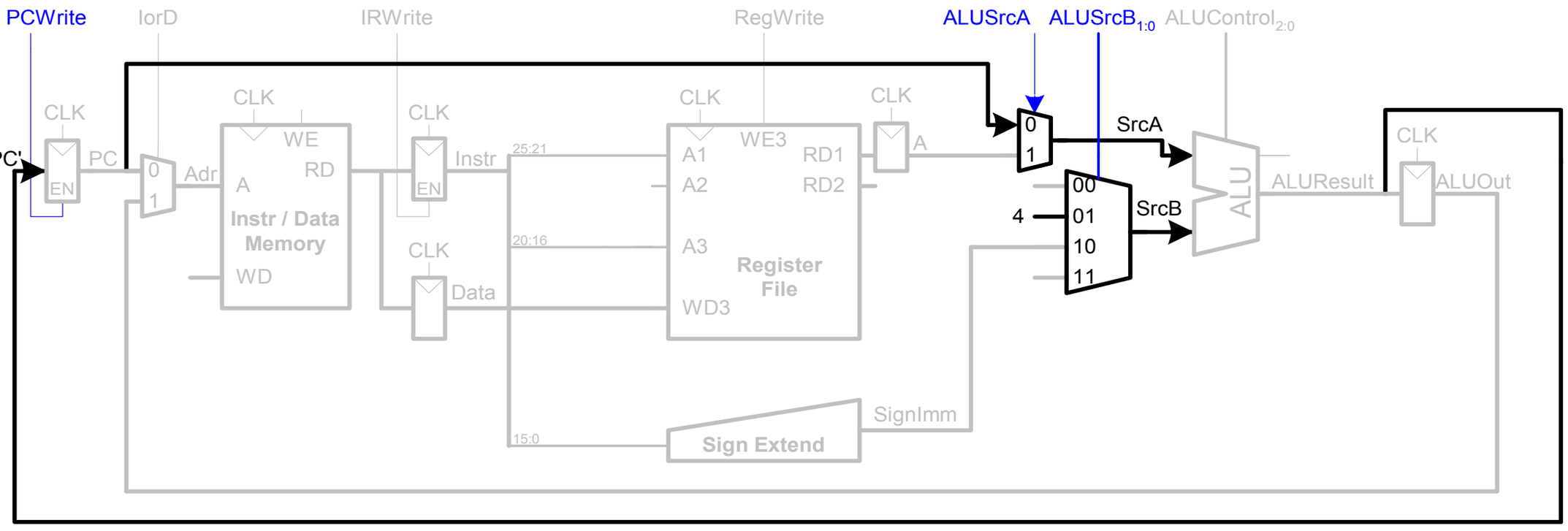
Multicycle Datapath: lw Write Register

STEP 5: Write data back to register file



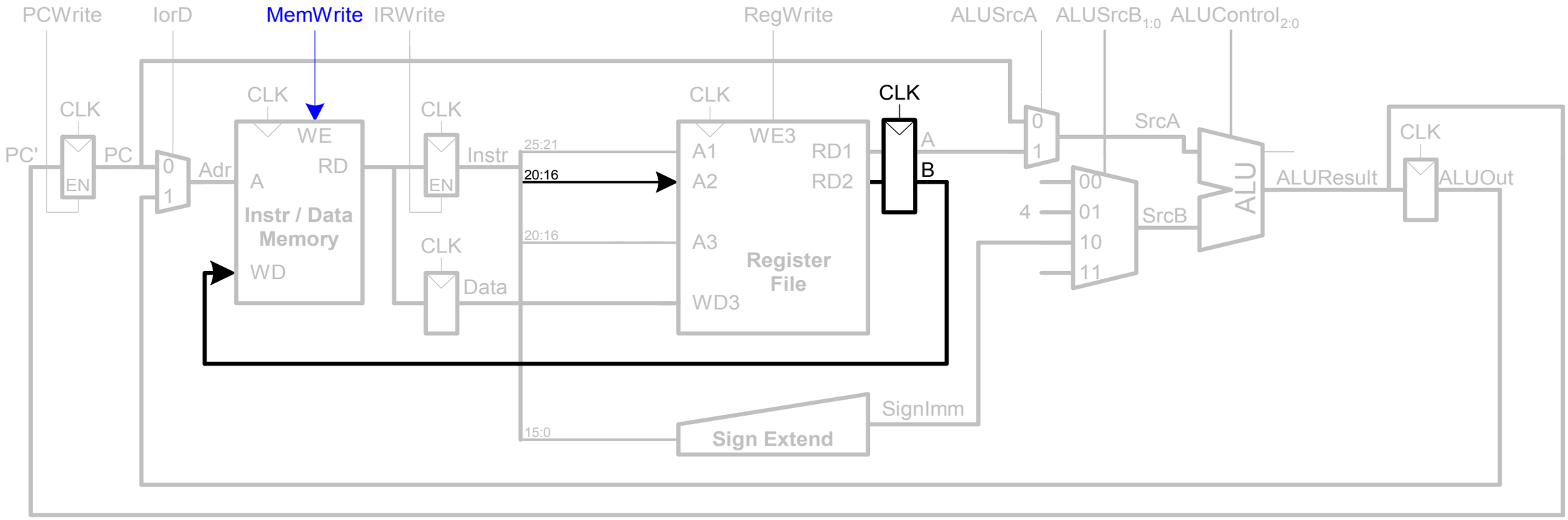
Multicycle Datapath: Increment PC

STEP 6: Increment PC



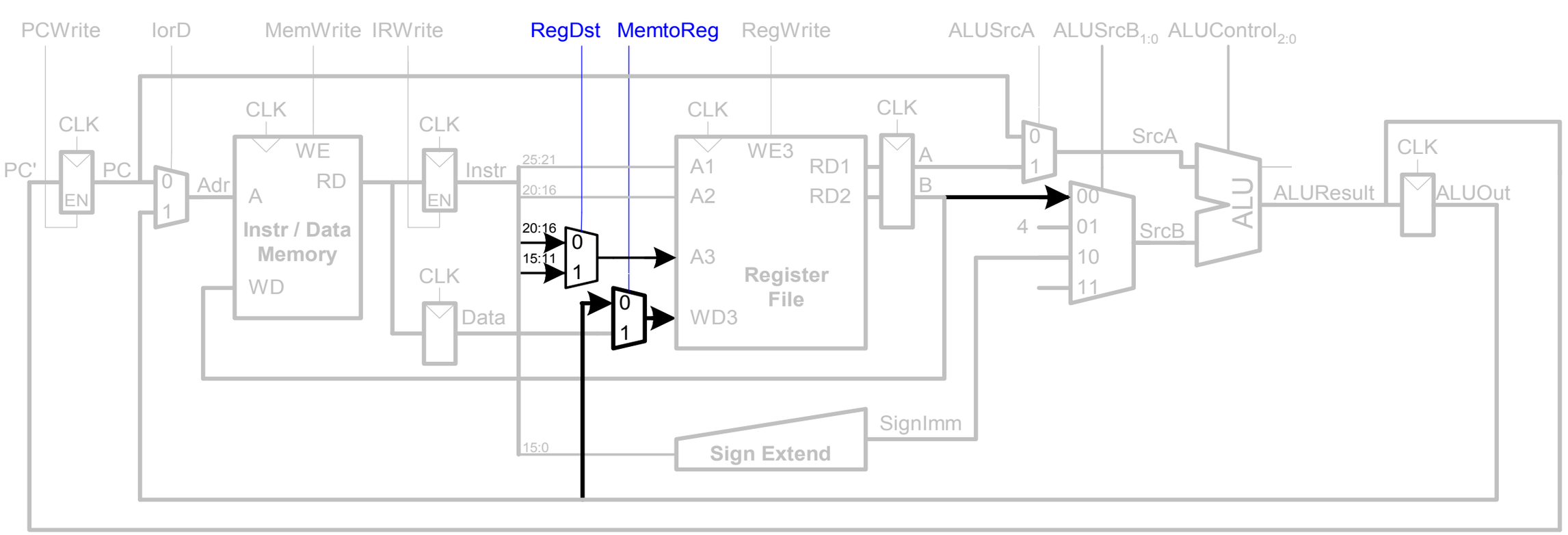
Multicycle Datapath: sw

Write data in `rt` to memory



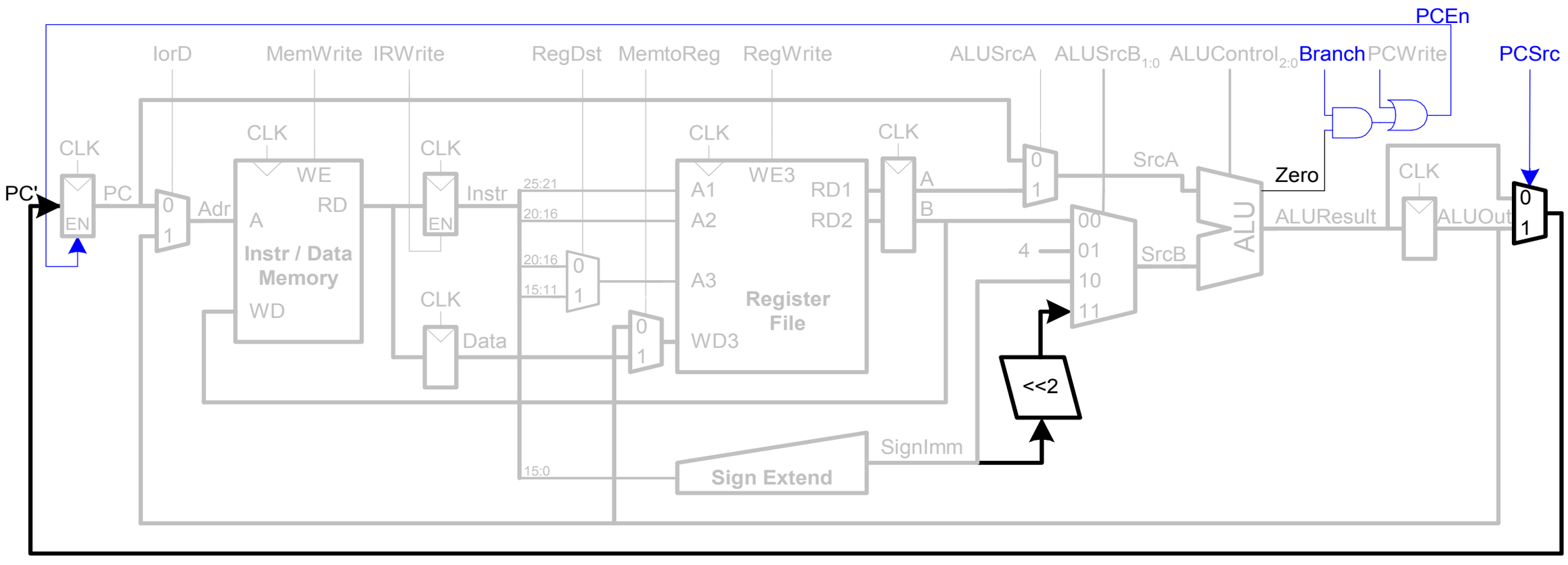
Multicycle Datapath: R-Type

- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)

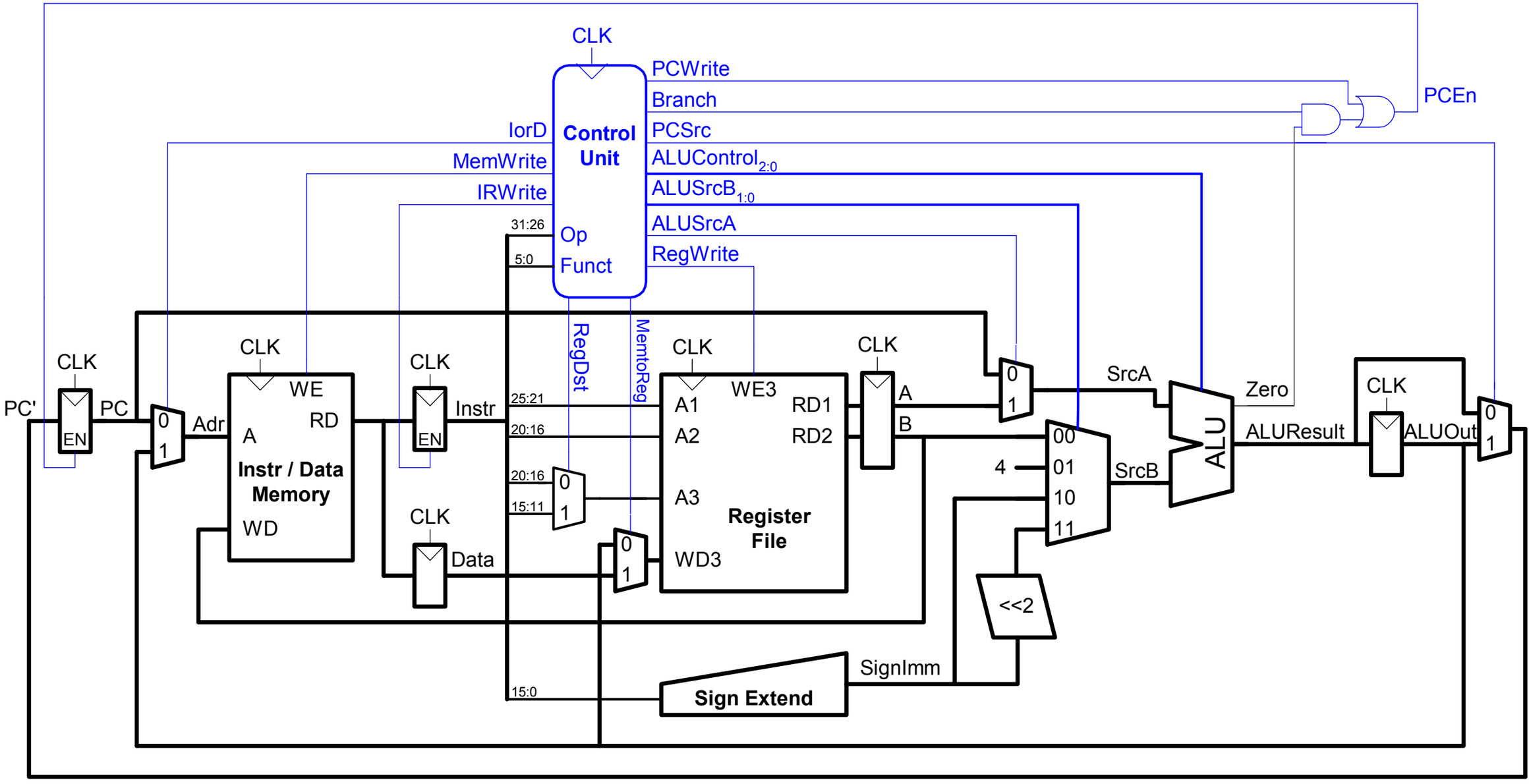


Multicycle Datapath: beq

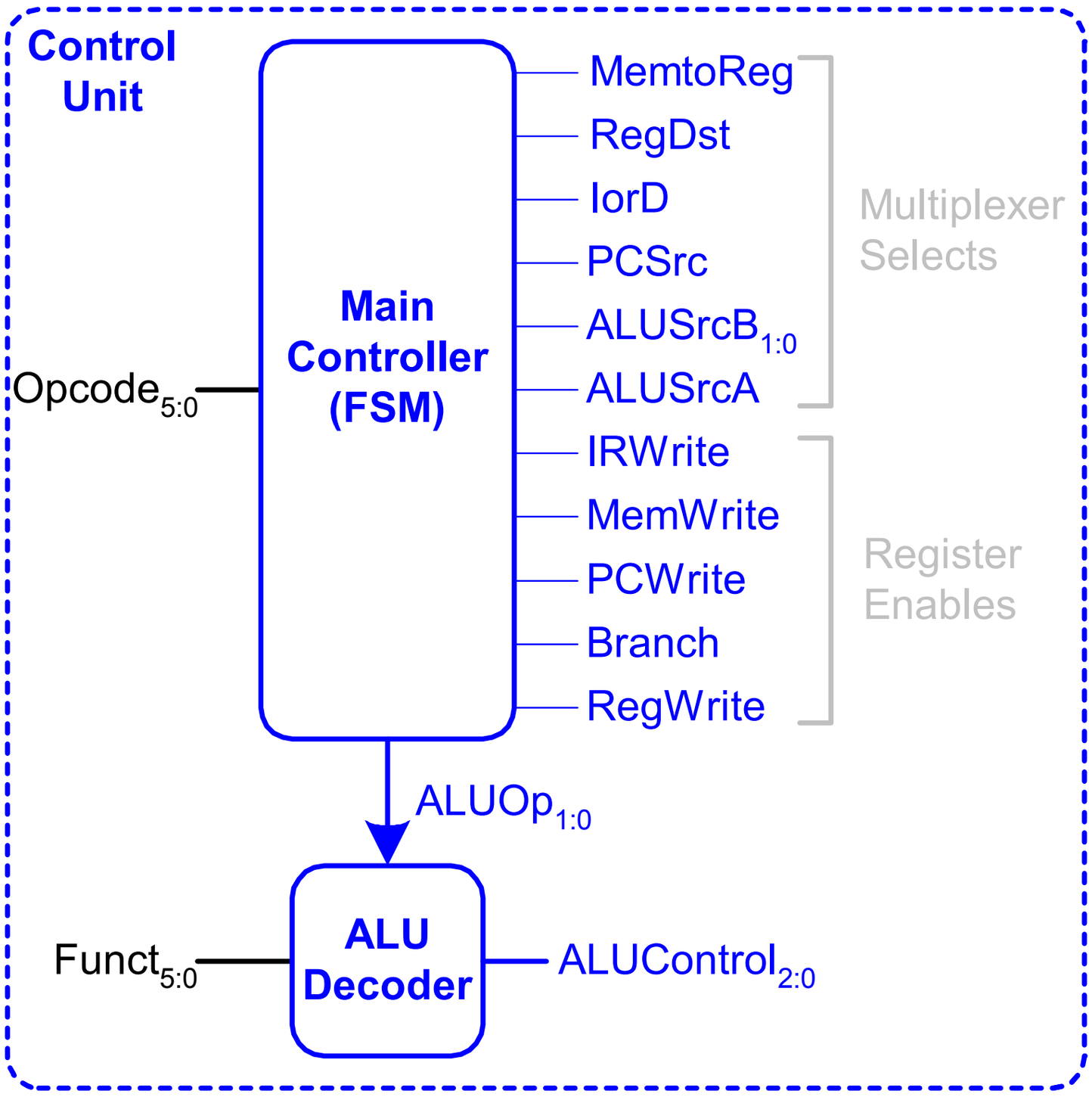
- `rs == rt`?
- BTA = (sign-extended immediate << 2) + (PC+4)



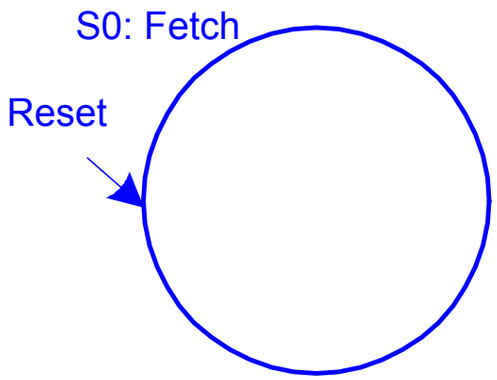
Multicycle Processor

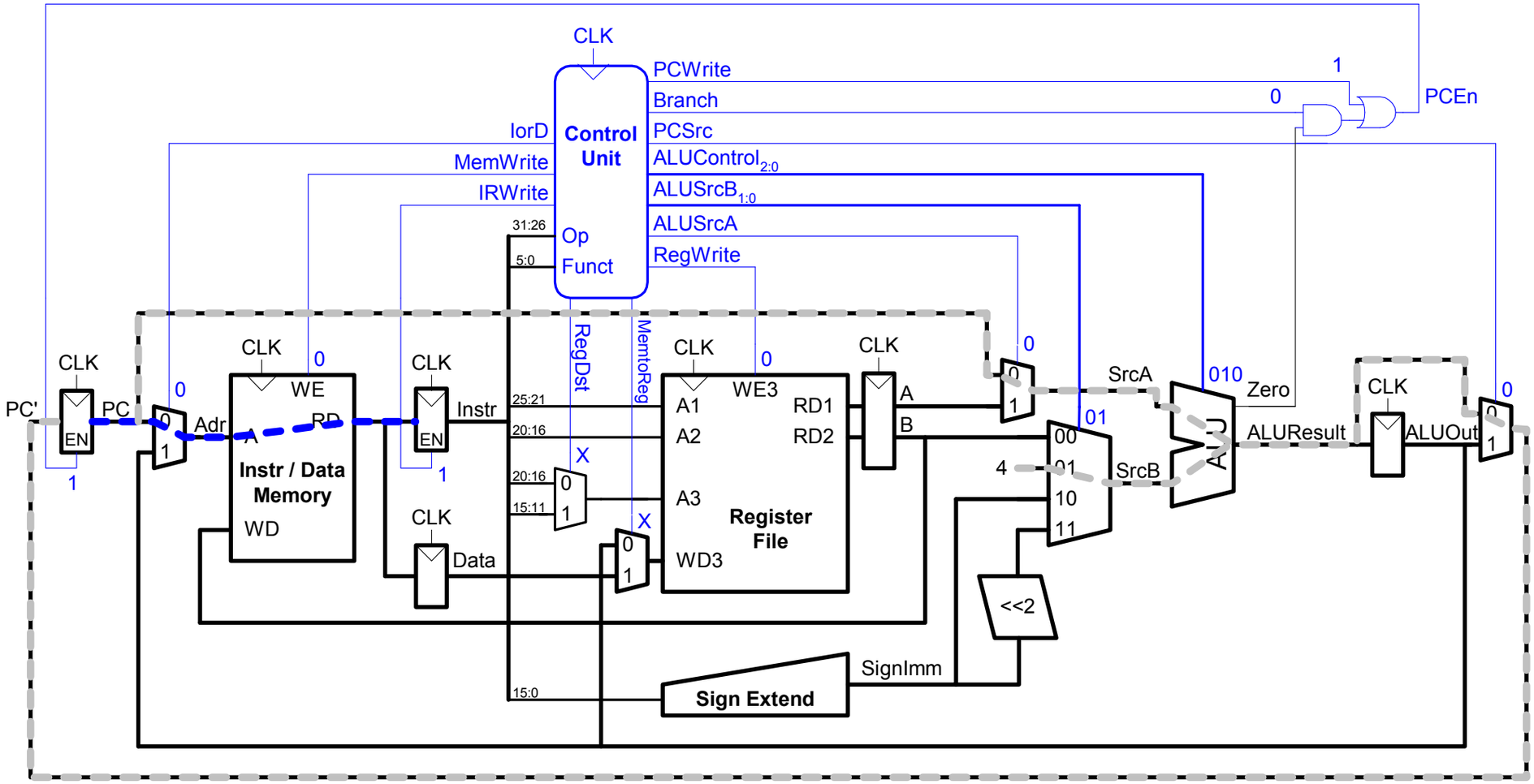


Multicycle Control

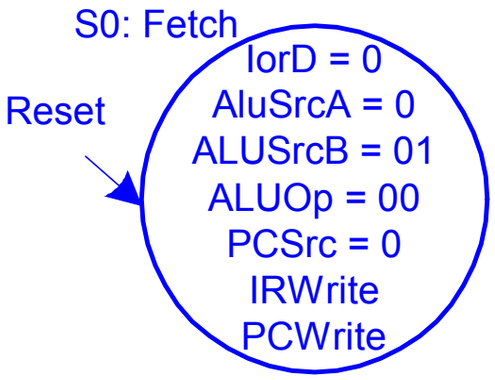


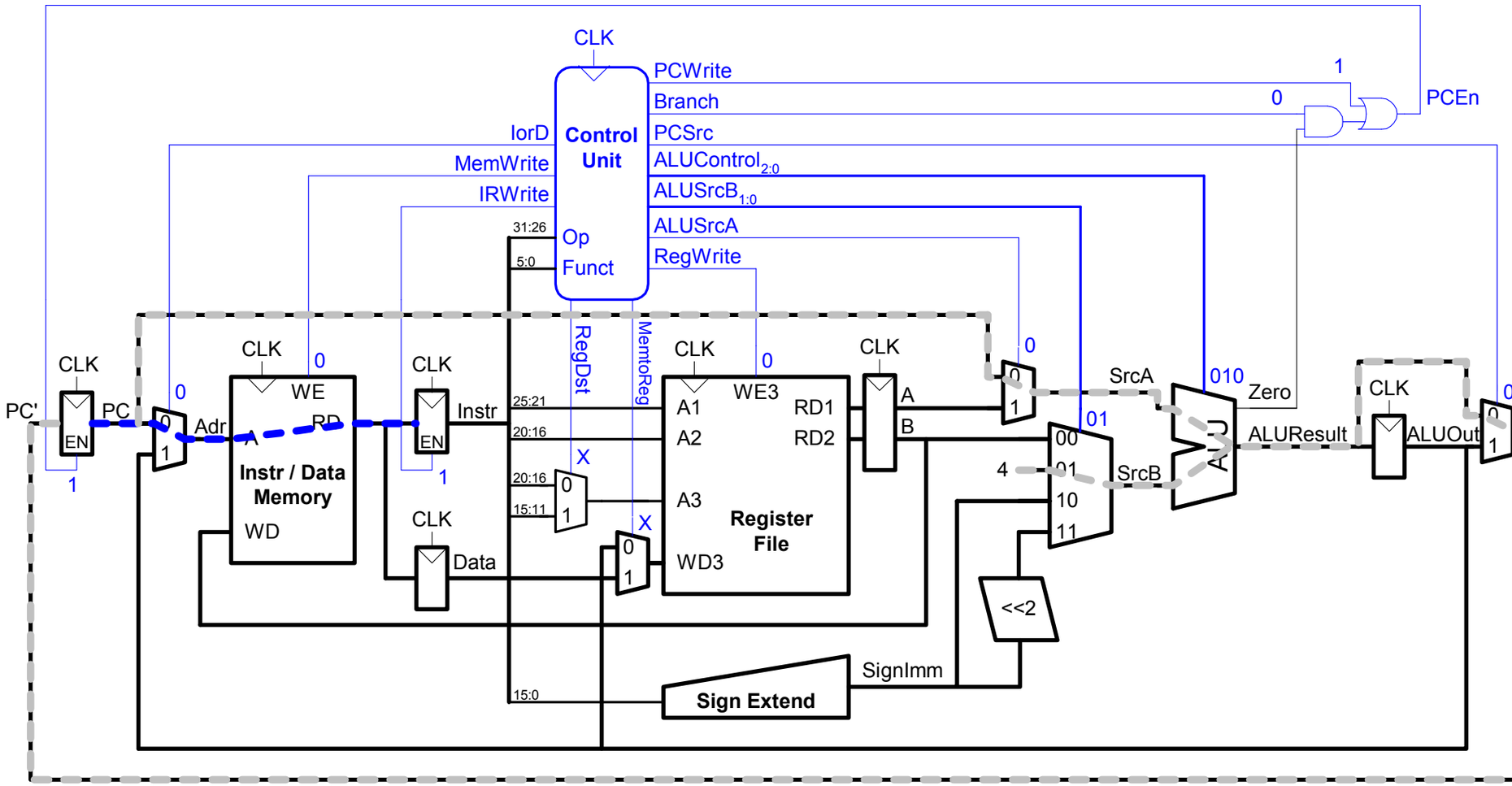
Main Controller FSM: Fetch



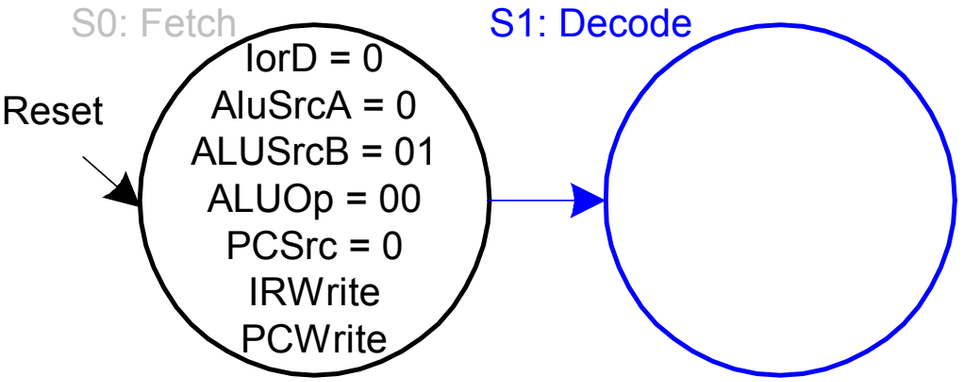


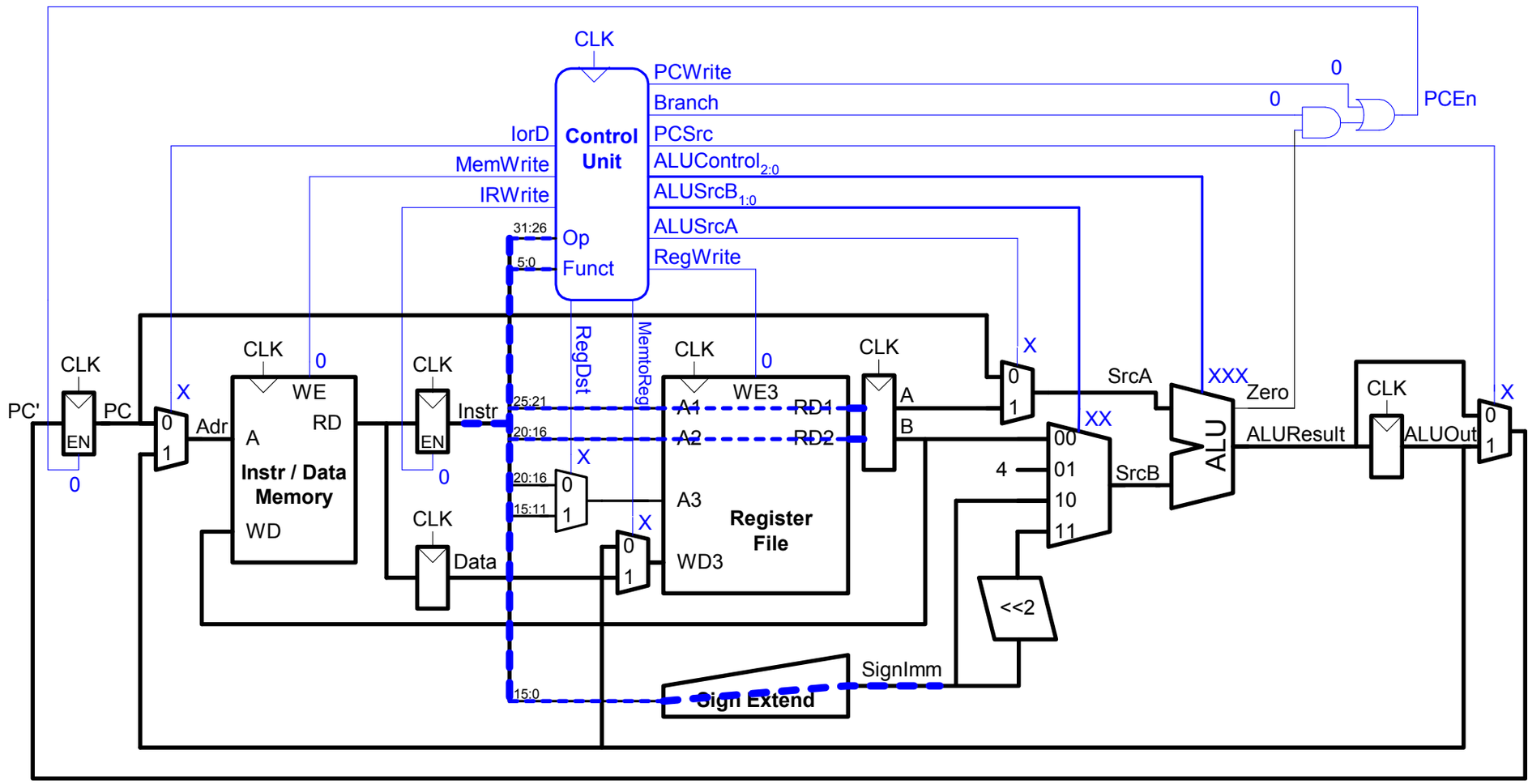
Main Controller FSM: Fetch



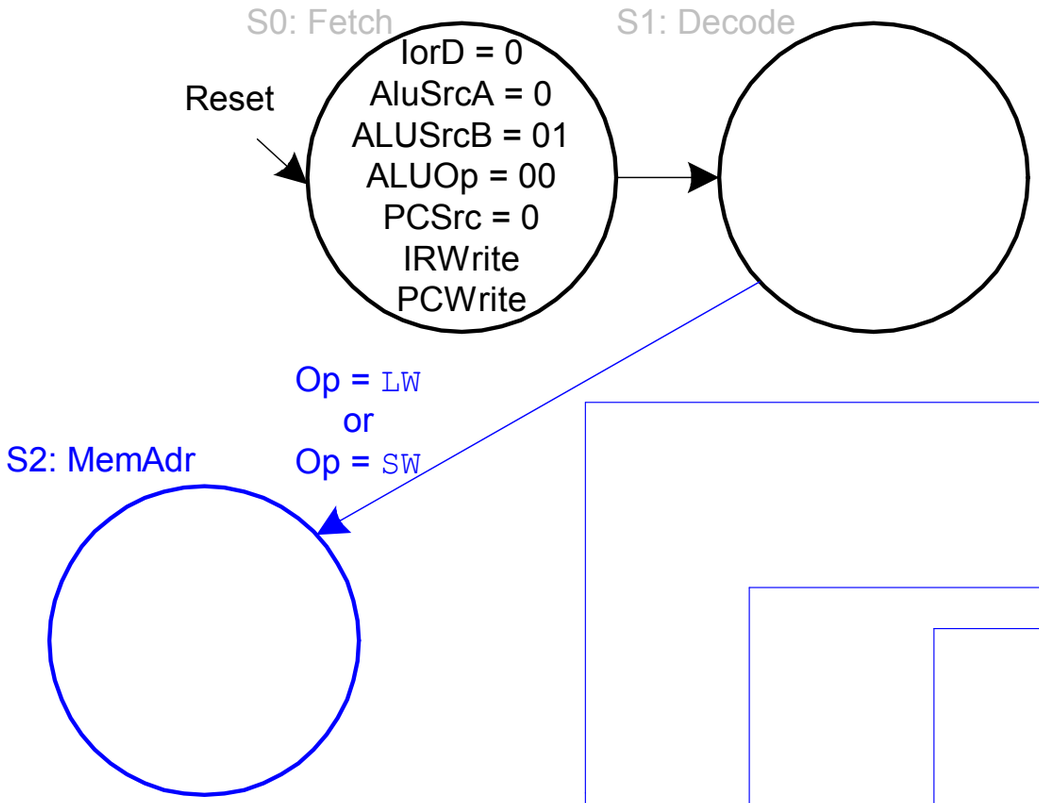


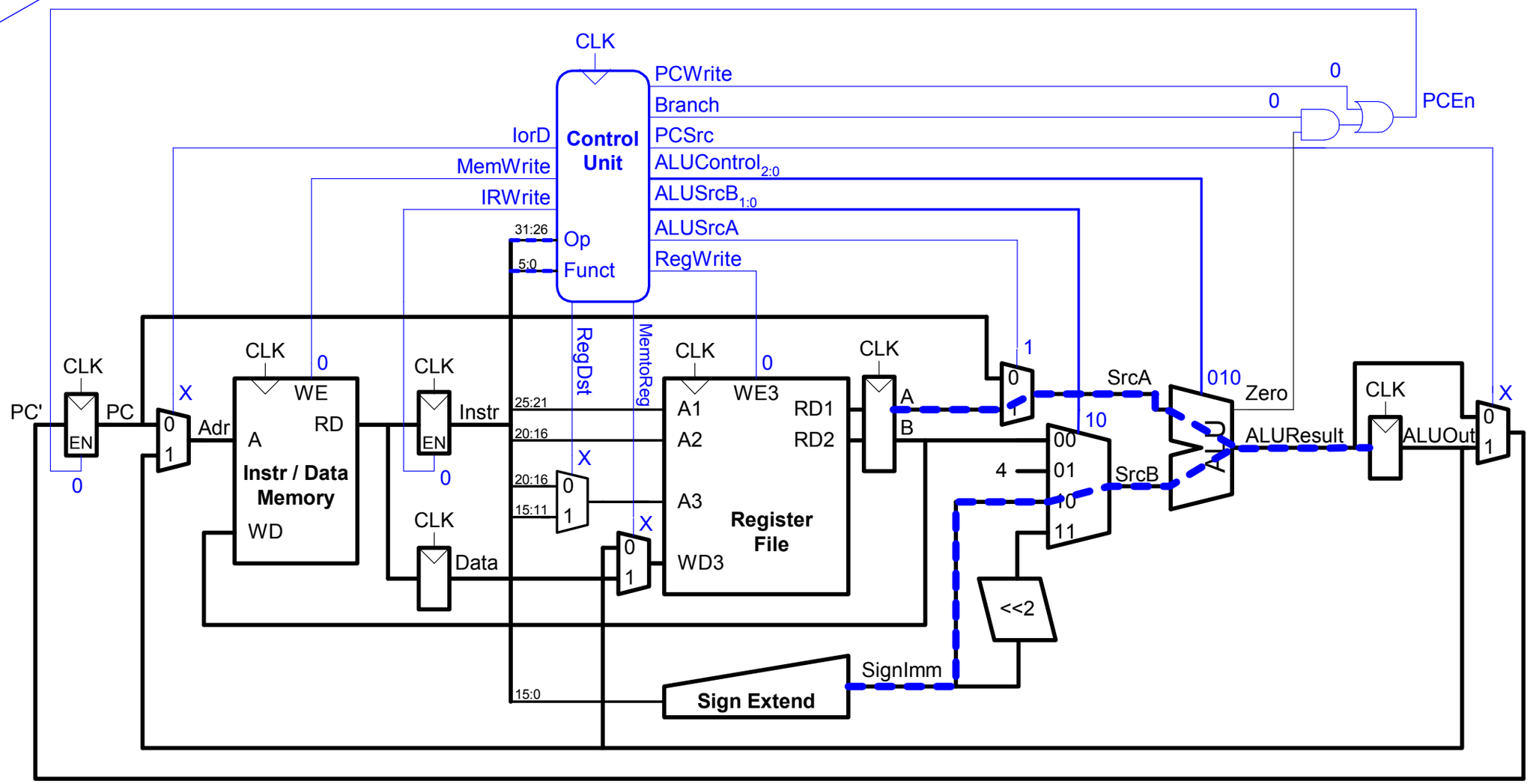
Main Controller FSM: Decode



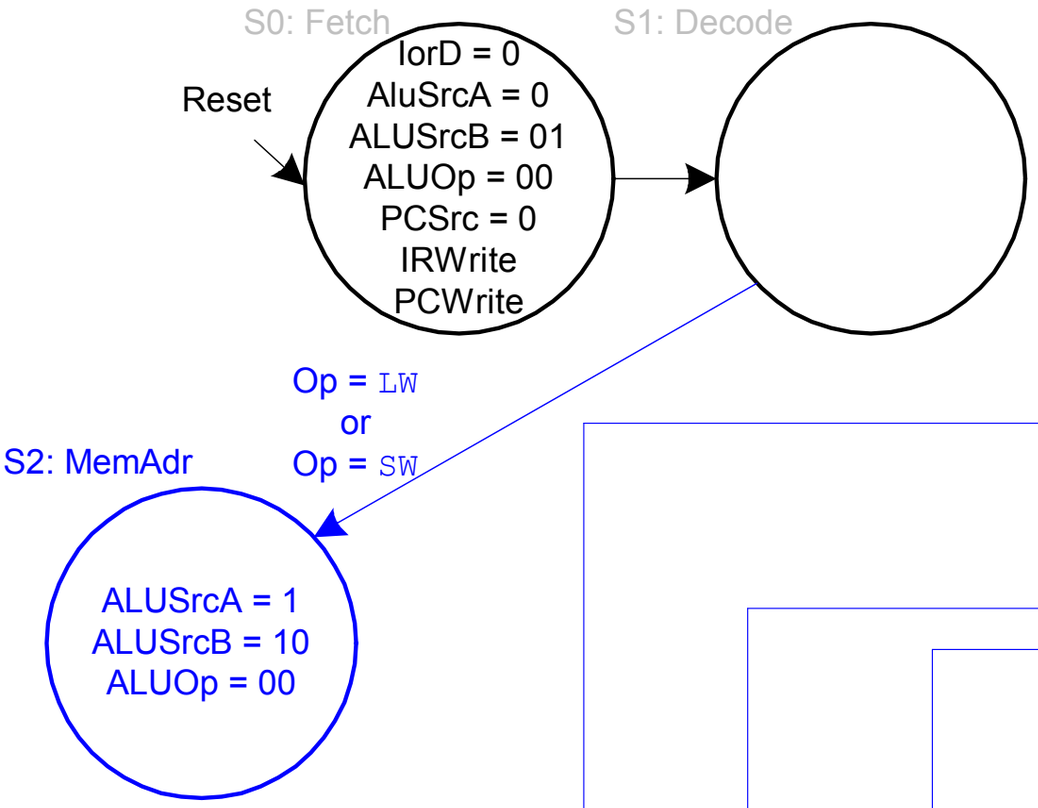


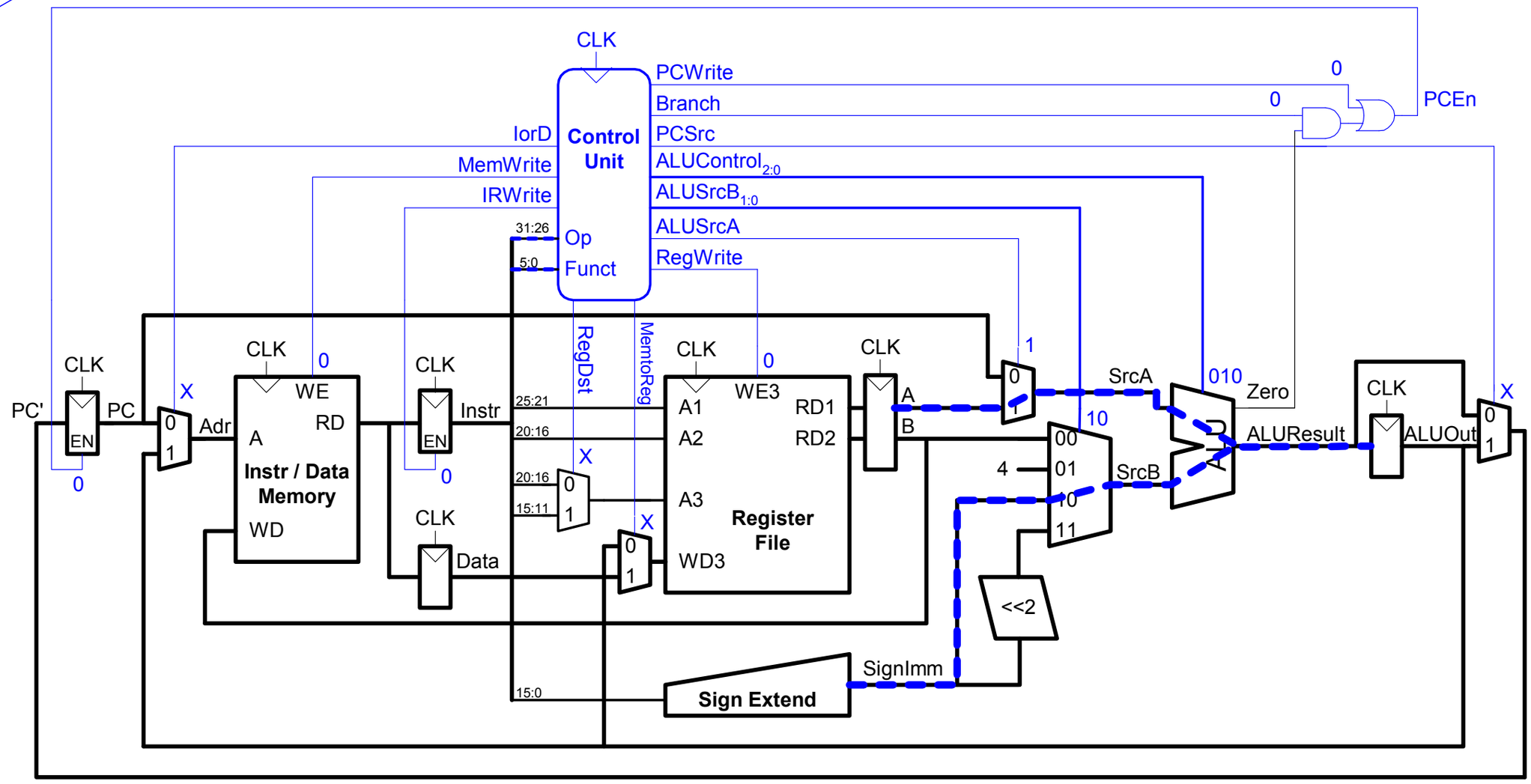
Main Controller FSM: Address



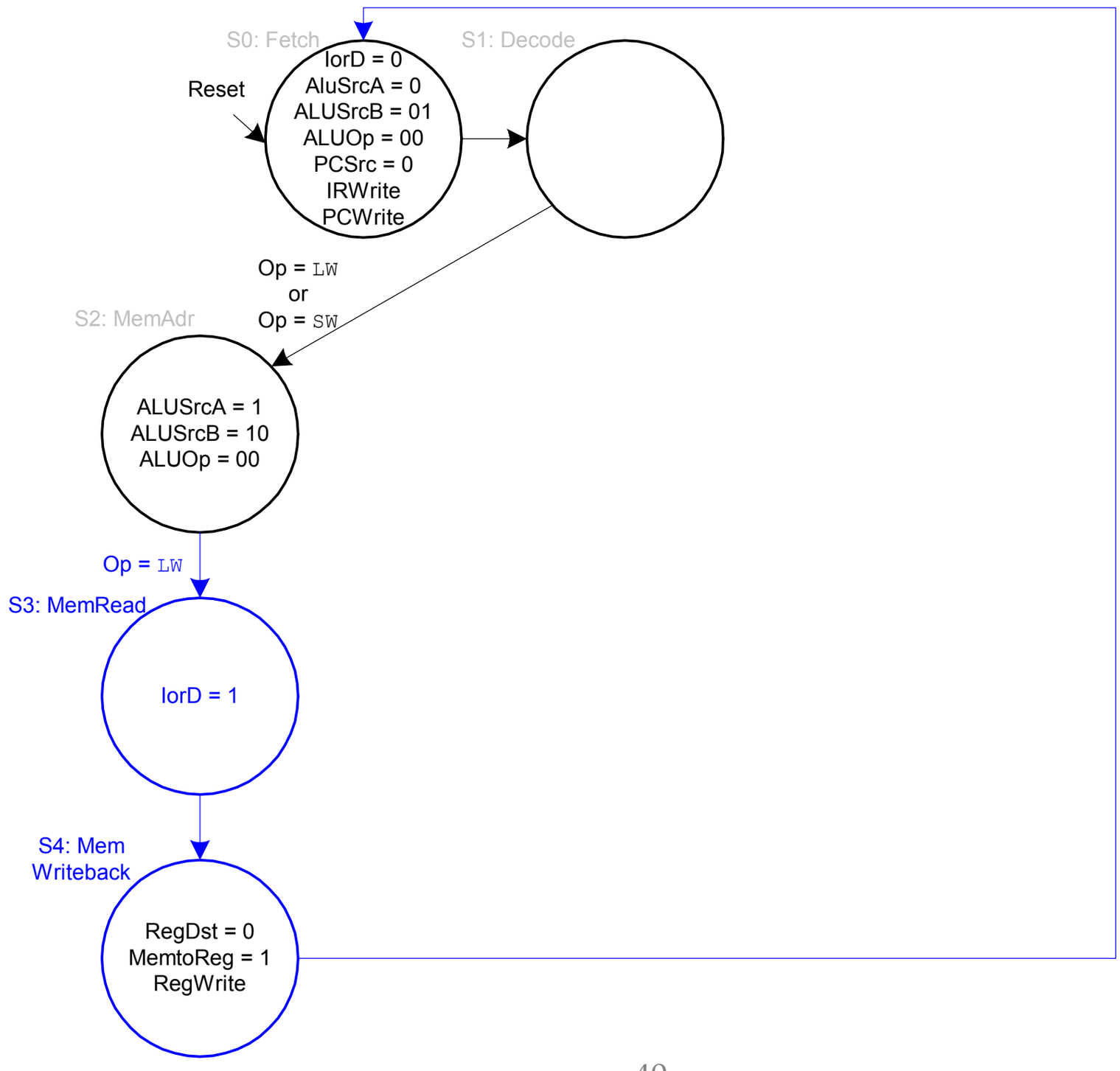


Main Controller FSM: Address

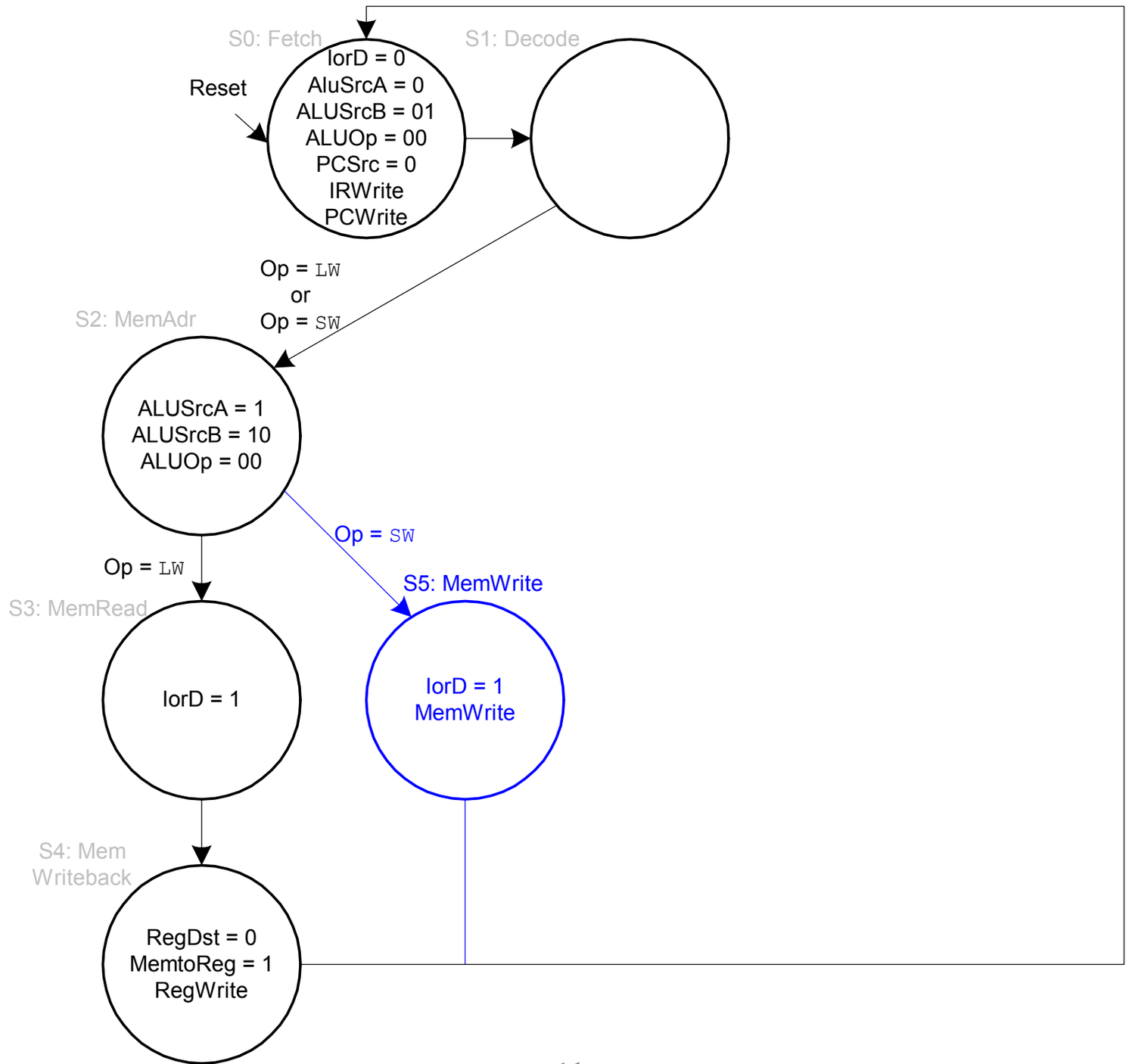




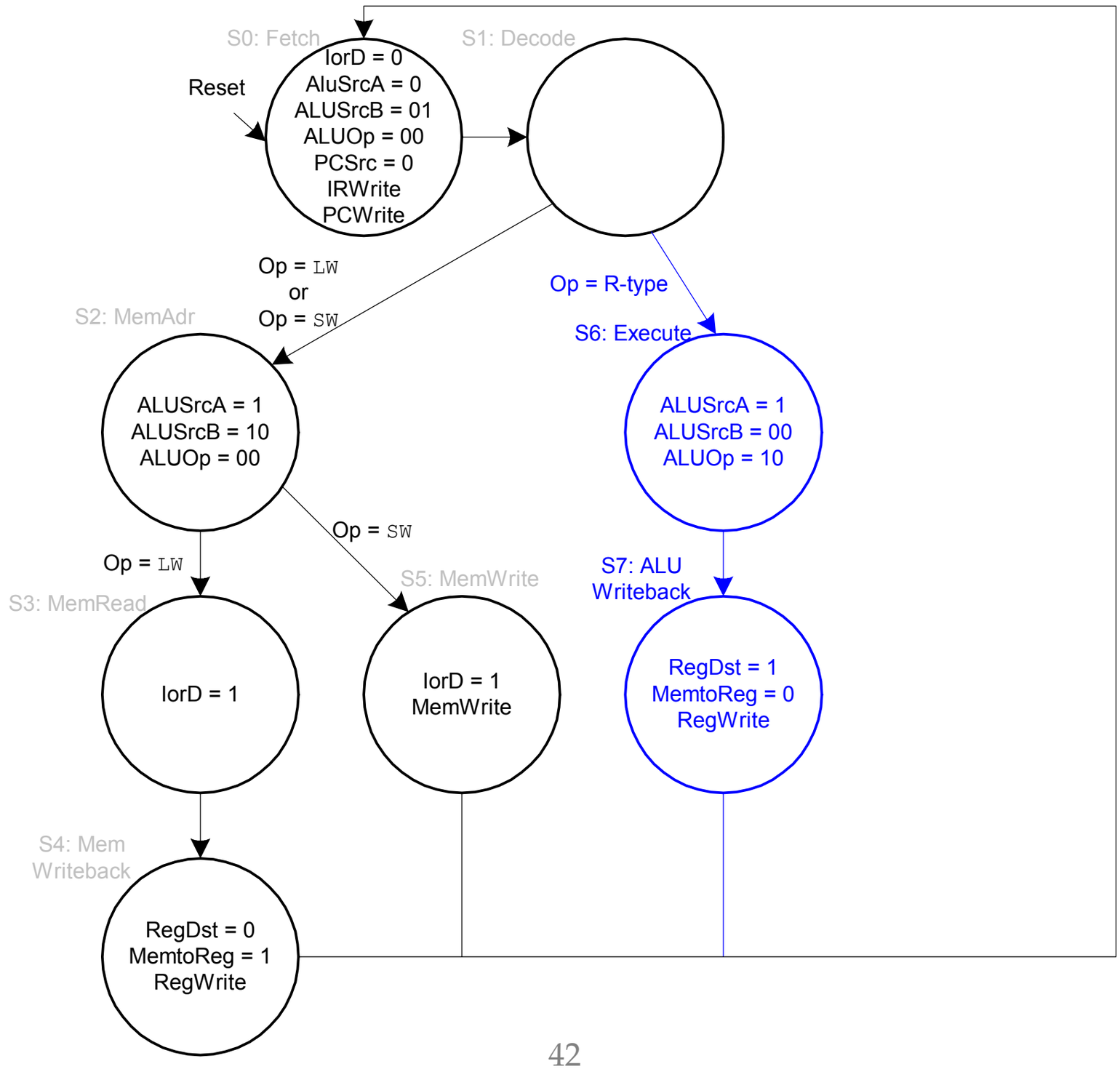
Main Controller FSM: lw



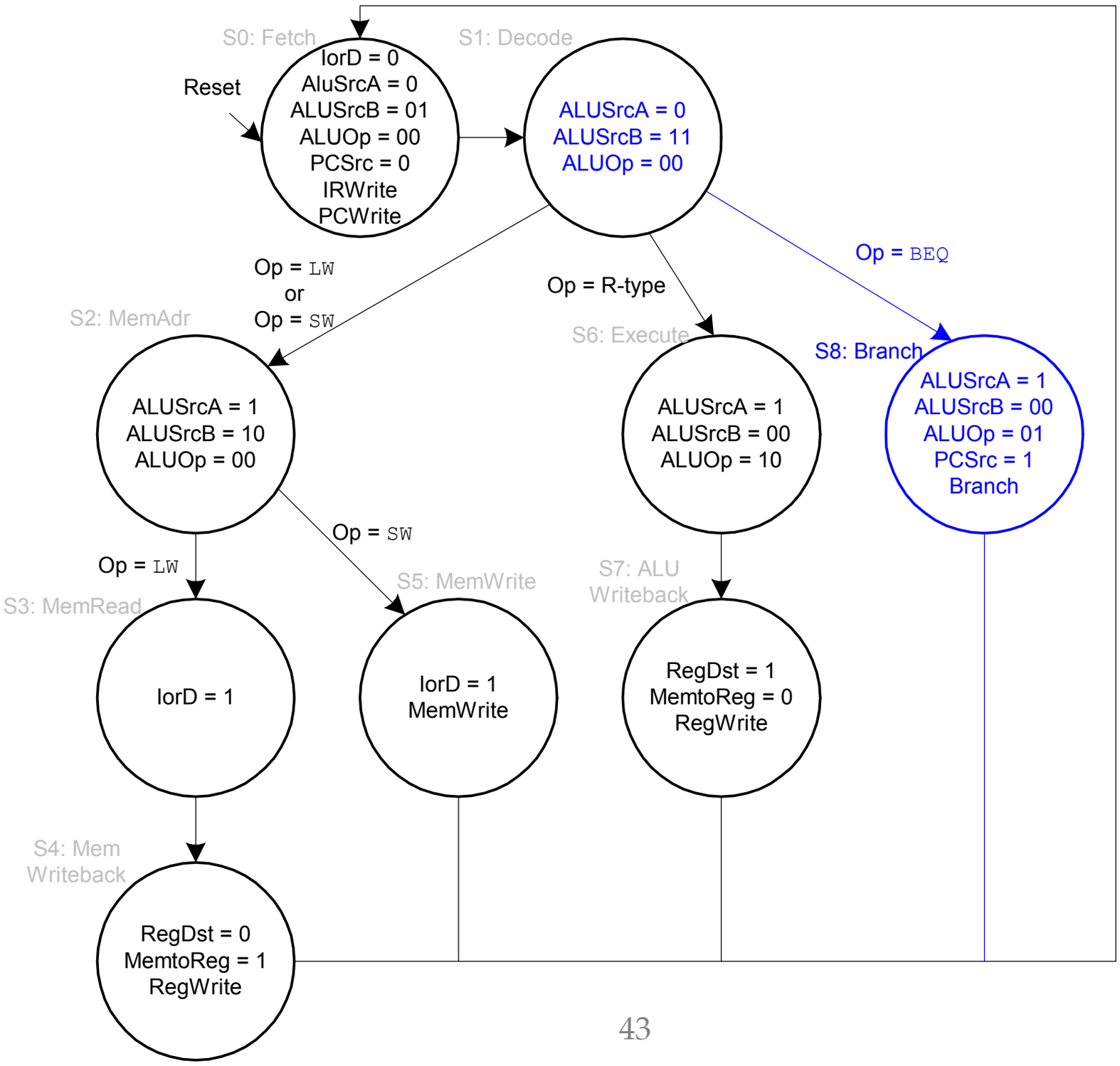
Main Controller FSM: sw



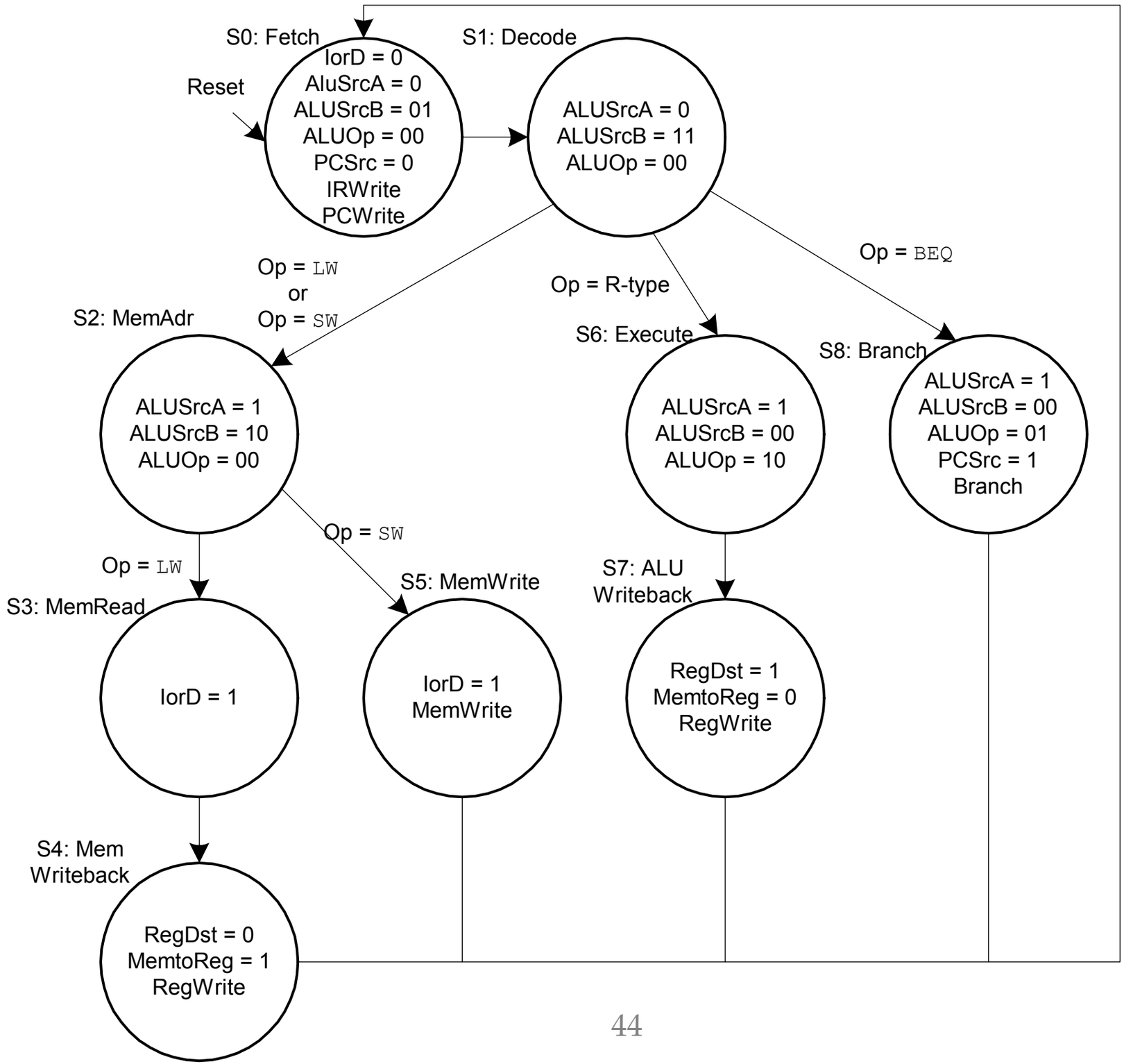
Main Controller FSM: R-Type



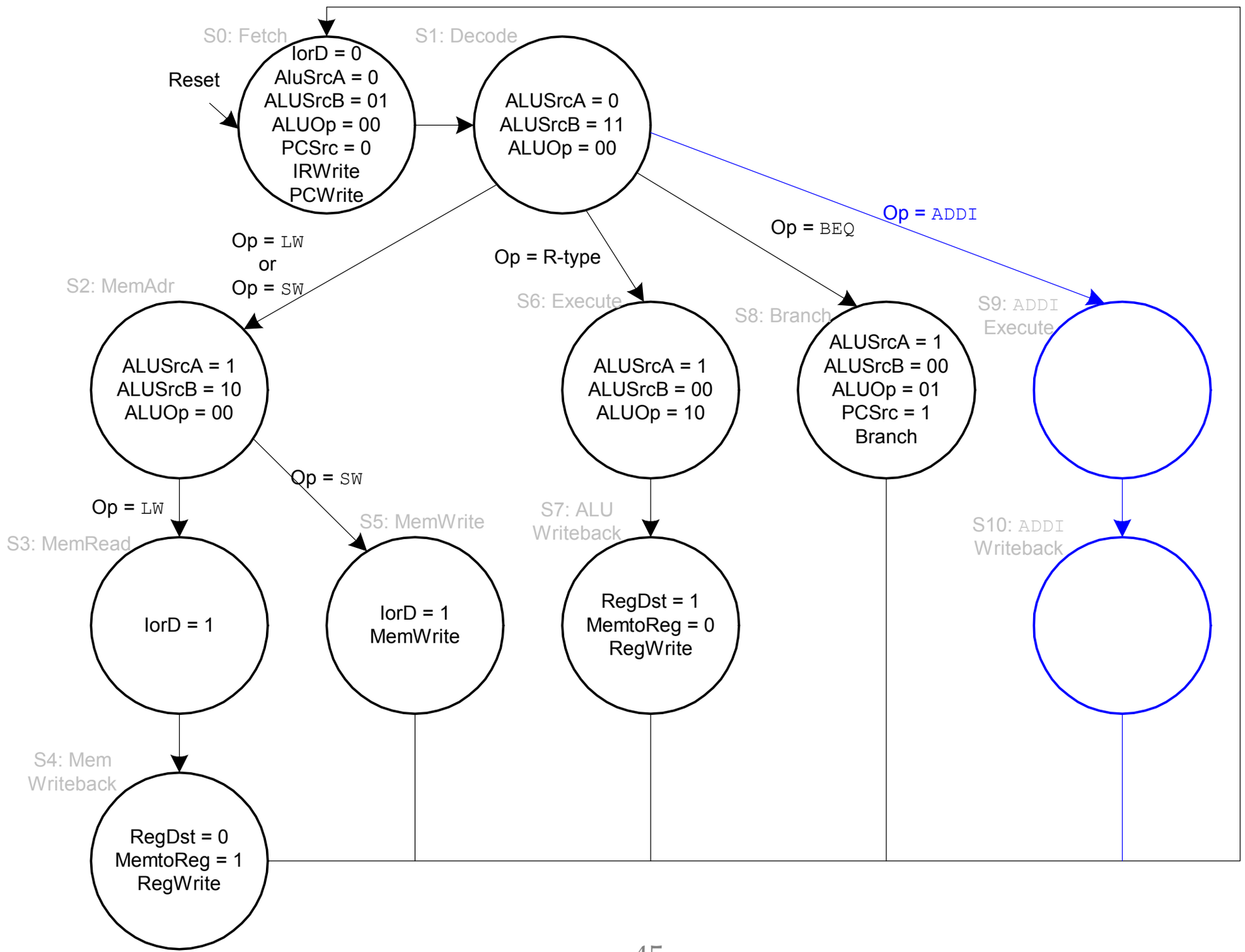
Main Controller FSM: beq



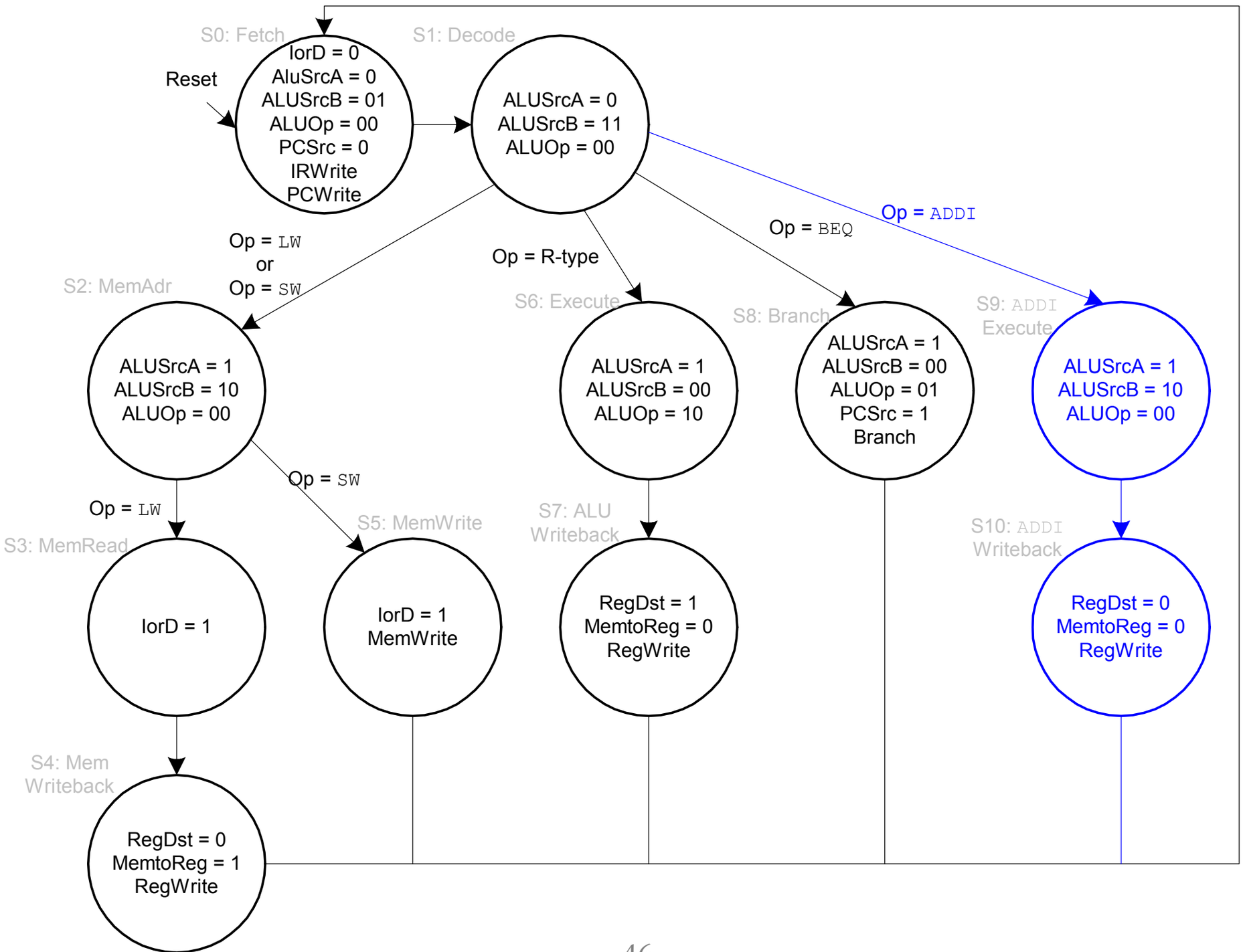
Multicycle Controller FSM



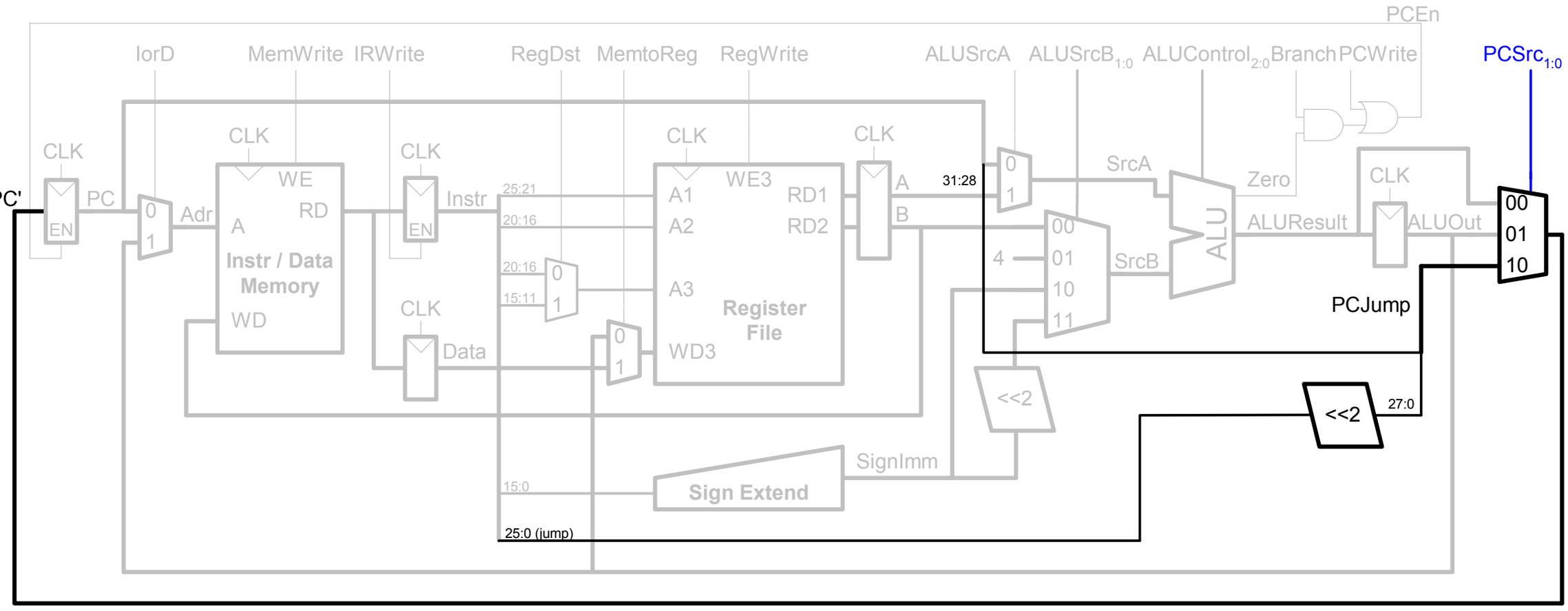
Extended Functionality: addi



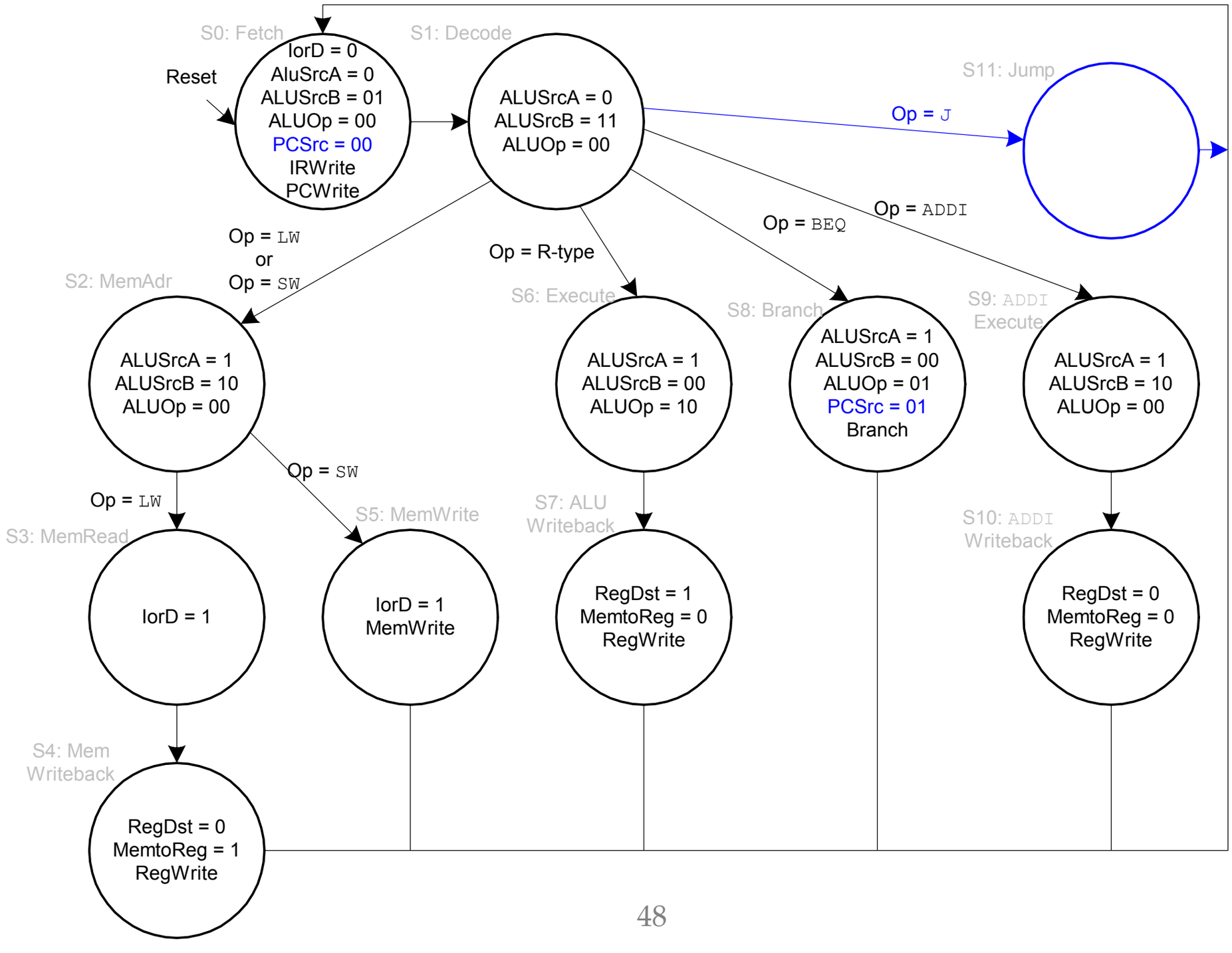
Main Controller FSM: addi



Extended Functionality: j



Main Controller FSM: j



Main Controller FSM: j