DIGITAL BUILDING BLOCKS

Chapter 5

***Digital Design and Computer Architecture*, 2nd Edition**

David Money Harris and Sarah L. Harris

ELSEVIER

Chapter 5 :: Topics

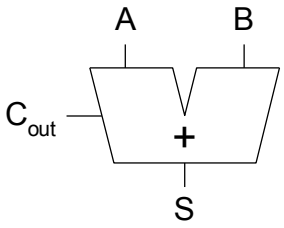
- **Introduction**
- **Arithmetic Circuits**
- **Number Systems (no)**
- **Sequential Building Blocks (later)**
- **Memory Arrays (later)**
- **Logic Arrays (maybe)**

Introduction

- **Digital building blocks:**
  - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- **Building blocks demonstrate hierarchy, modularity, and regularity:**
  - Hierarchy of simpler components
  - Well-defined interfaces and functions
  - Regular structure easily extends to different sizes
- **Will use these building blocks in Chapter 7 to build microprocessor**

1-Bit Adders

Half Adder

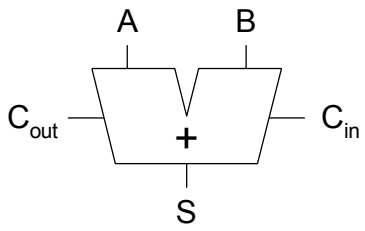


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

Full Adder



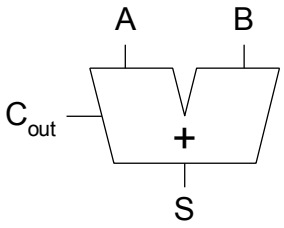
C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =

C_{out} =

1-Bit Adders

Half Adder

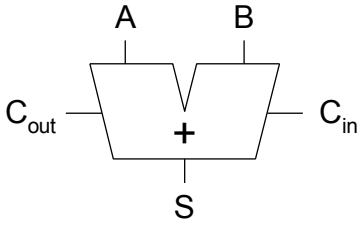


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S =

C_{out} =

Full Adder



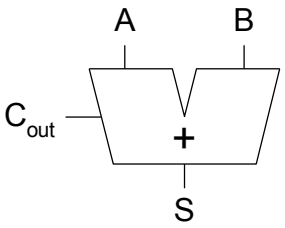
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S =

C_{out} =

1-Bit Adders

Half Adder

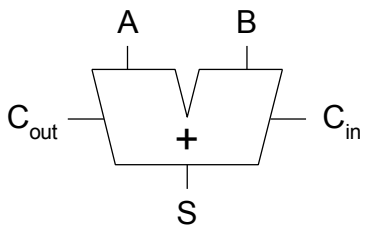


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

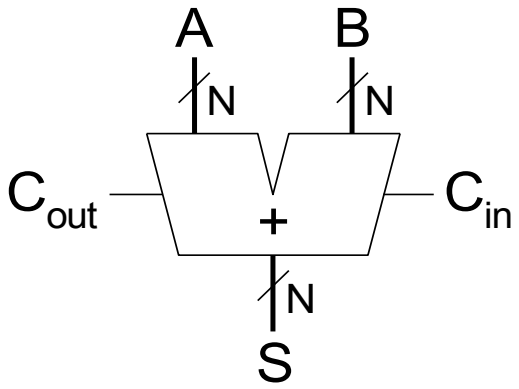
$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

Multibit Adders (CPAs)

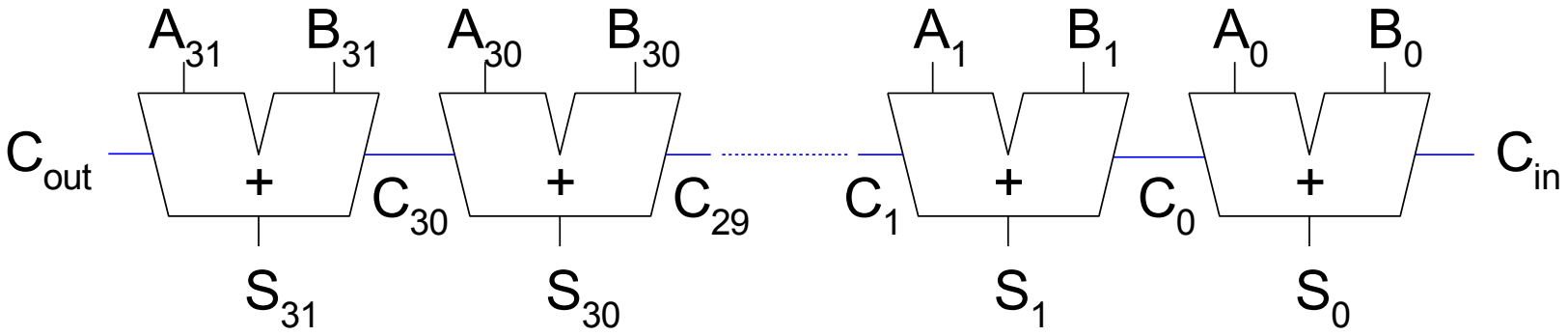
- Types of carry propagate adders (CPAs):
  - Ripple-carry (slow)
  - Carry-lookahead (fast)
  - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol



Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



Ripple-Carry Adder Delay

$$t_{ripple} = Nt_{FA}$$

where t_{FA} is the delay of a full adder

Carry-Lookahead Adder

- Compute carry out (C_{out}) for k-bit blocks using *generate* and *propagate* signals
- **Some definitions:**
  - Column i produces a carry out by either *generating* a carry out or *propagating* a carry in to the carry out
  - Generate (G_i) and propagate (P_i) signals for each column:
    - Column i will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

    - Column i will propagate a carry in to the carry out if A_i OR B_i is 1.

$$P_i = A_i + B_i$$

    - The carry out of column i (C_i) is:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks
- **Step 3:** C_{in} propagates through each k-bit propagate/generate block

Carry-Lookahead Adder

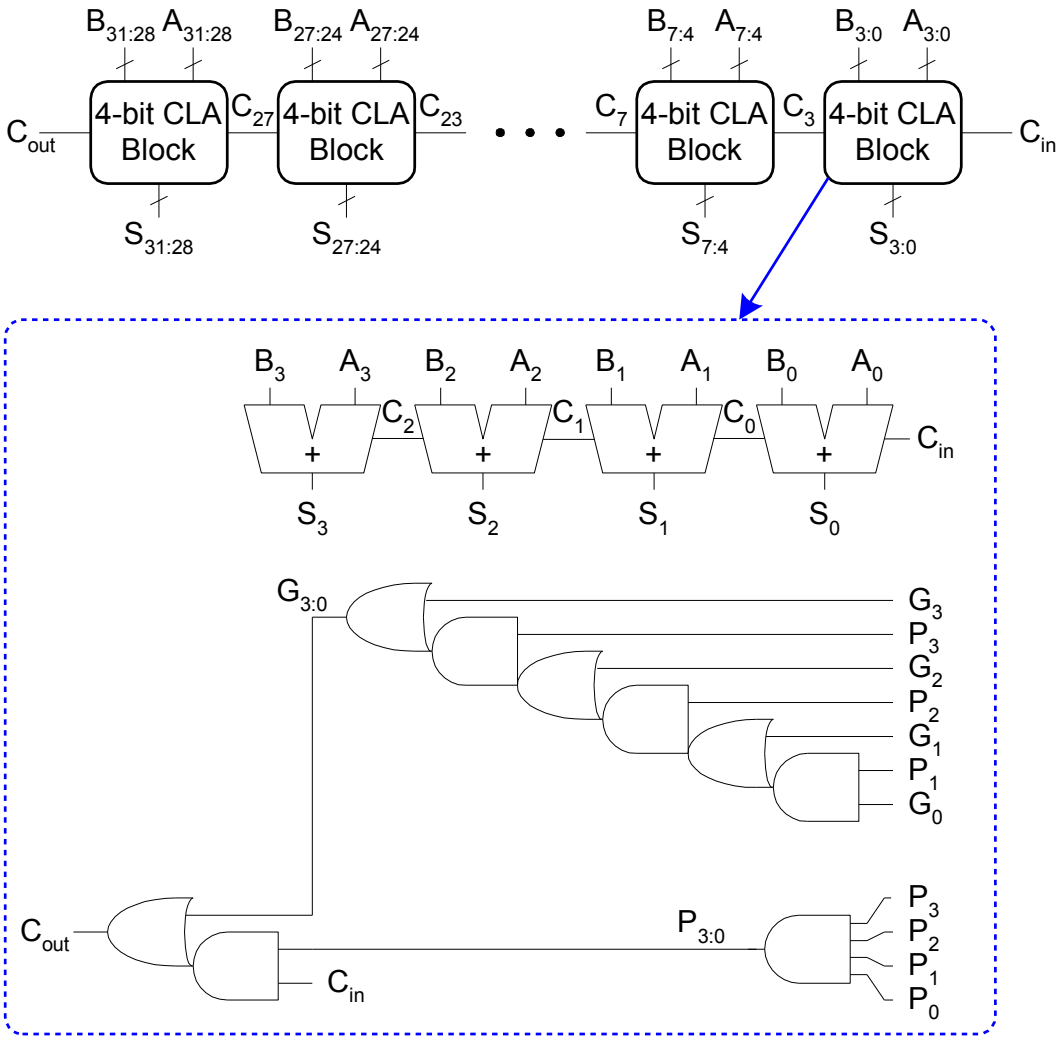
- **Example:** 4-bit blocks ($G_{3:0}$ and $P_{3:0}$) :

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$
$$P_{3:0} = P_3 P_2 P_1 P_0$$

- **Generally,**

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))$$
$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$
$$C_i = G_{i:j} + P_{i:j} C_{i-1}$$

32-bit CLA with 4-bit Blocks



Carry-Lookahead Adder Delay

For N-bit CLA with k-bit blocks:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in k-bit CLA block

An N-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$

Prefix Adder

- Computes carry in (C_{i-1}) for each column, then computes sum:

$$S_i = (A_i \oplus B_i) \oplus C_i$$

- Computes G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known
- $\log_2 N$ stages

Prefix Adder

- Carry in either *generated* in a column or *propagated* from a previous column.
- Column -1 holds C_{in}, so

  $$G_{-1} = C_{in}, P_{-1} = 0$$

- Carry in to column i = carry out of column i-1:

  $$C_{i-1} = G_{i-1:-1}$$

  $G_{i-1:-1}$: generate signal spanning columns i-1 to -1
- Sum equation:

  $$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

- **Goal:** Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (called ***prefixes***)

Prefix Adder

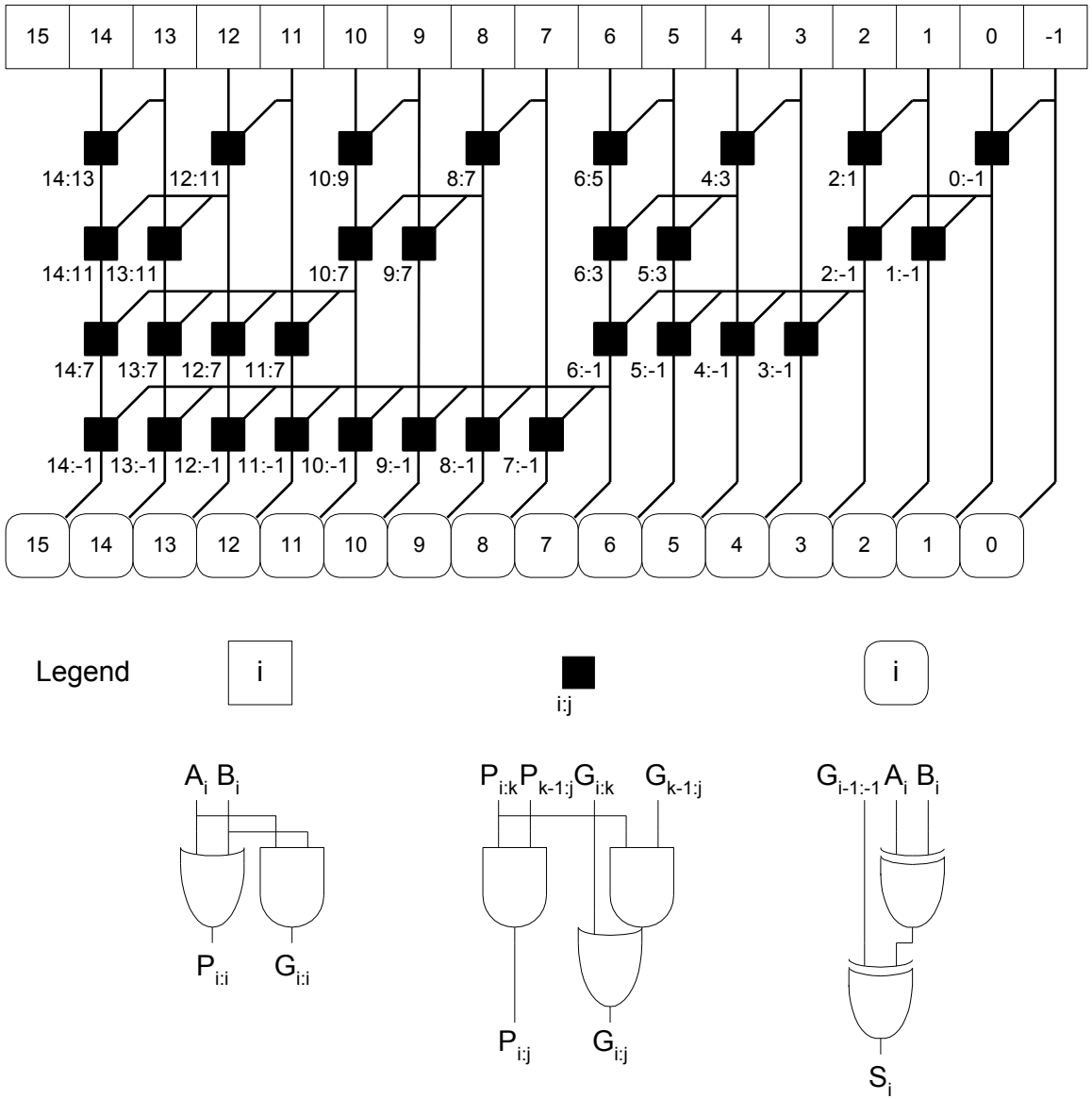
- Generate and propagate signals for a block spanning bits $i{:}j$:

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- In words:
  - **Generate:** block $i{:}j$ will generate a carry if:
    - upper part ($i{:}k$) generates a carry or
    - upper part propagates a carry generated in lower part (k-1$:j$)
  - **Propagate:** block $i{:}j$ will propagate a carry if *both* the upper and lower parts propagate the carry

Prefix Adder Schematic



Prefix Adder Delay

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

- t_{pg}: delay to produce P_i G_i (AND or OR gate)
- t_{pg_prefix}: delay of black prefix cell (AND-OR gate)

Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

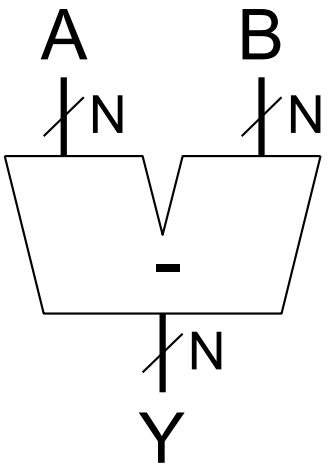
$$t_{\text{ripple}} = Nt_{FA} = 32(300\text{ ps}) = \mathbf{9.6\text{ ns}}$$

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA}$$
$$= [100 + 600 + (7)200 + 4(300)]\text{ ps} = \mathbf{3.3\text{ ns}}$$

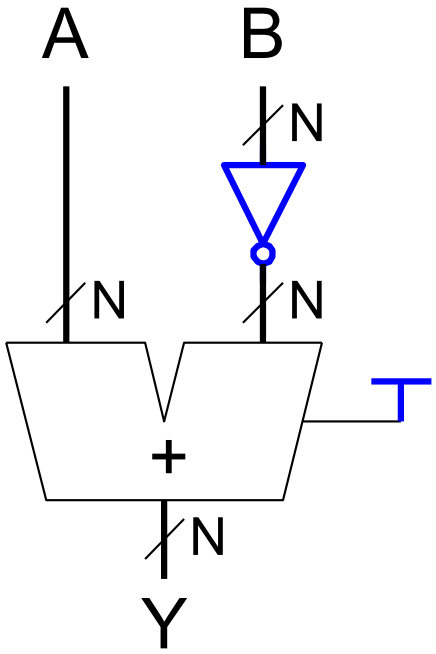
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$
$$= [100 + \log_2 32(200) + 100]\text{ ps} = \mathbf{1.2\text{ ns}}$$

Subtracter

Symbol

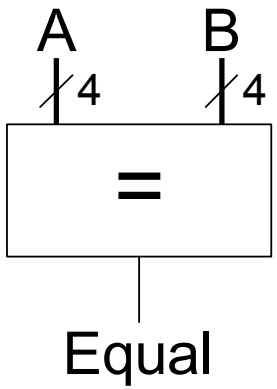


Implementation

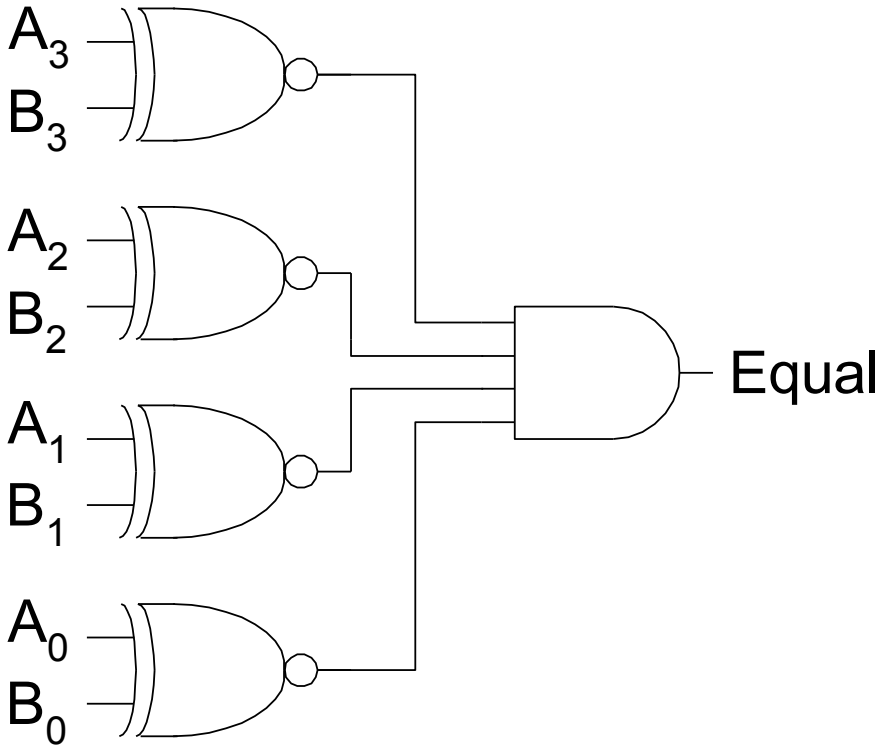


Comparator: Equality

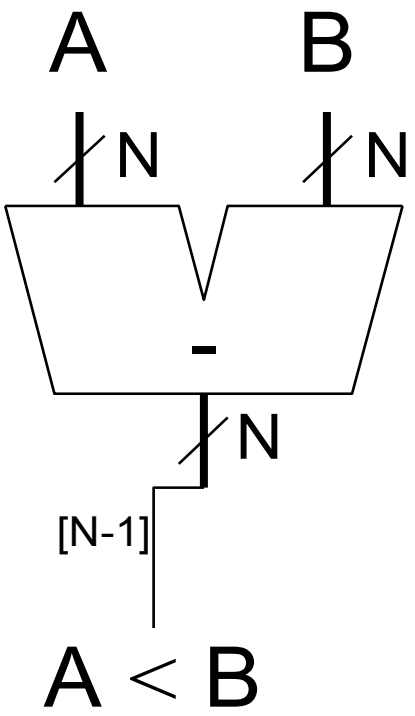
Symbol



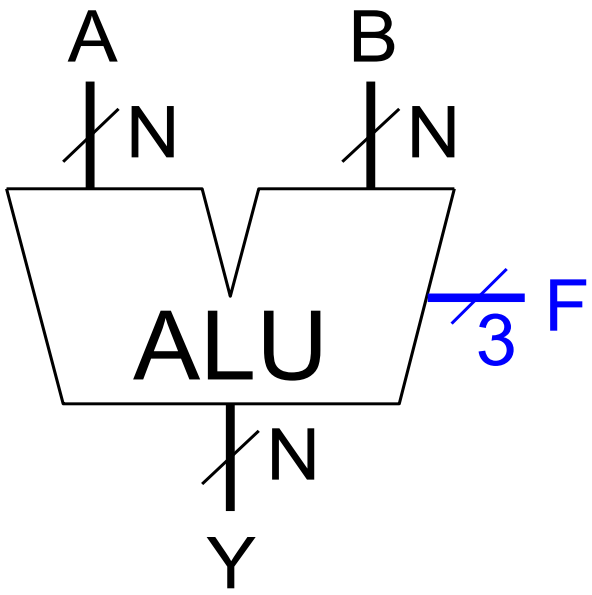
Implementation



Comparator: Less Than

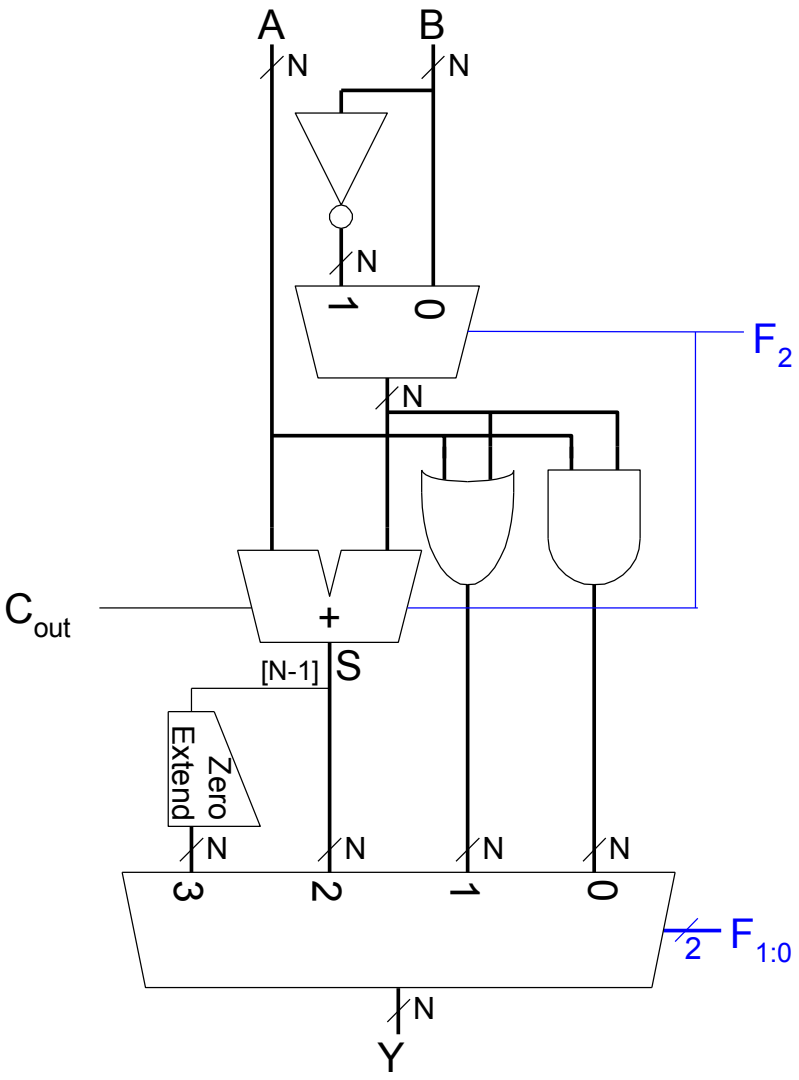


Arithmetic Logic Unit (ALU)



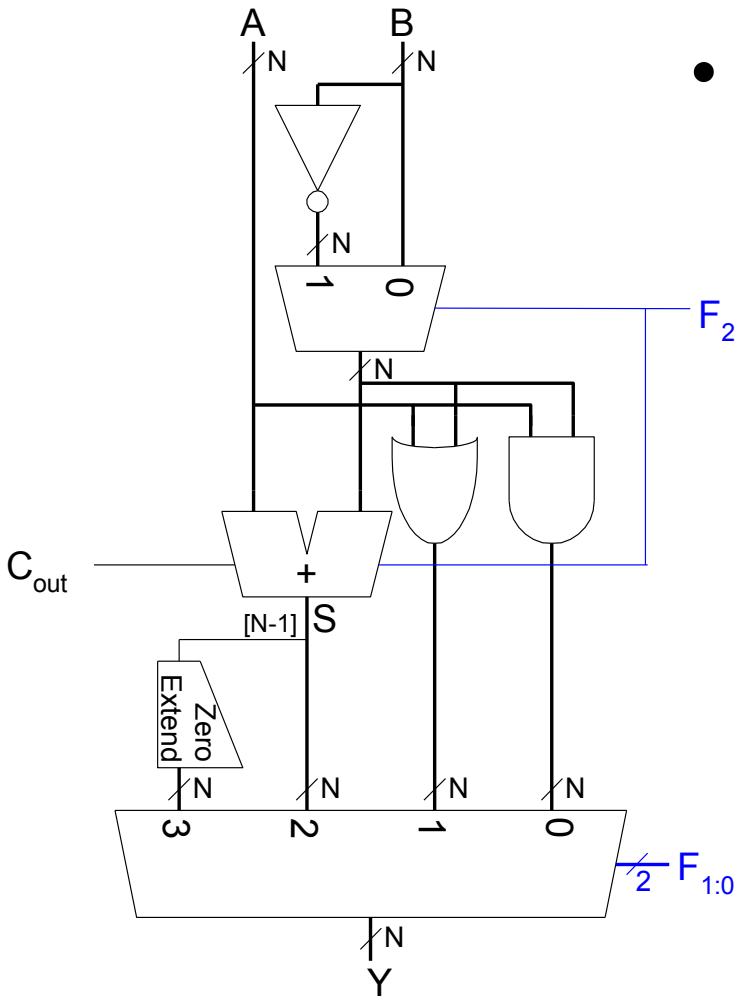
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

ALU Design



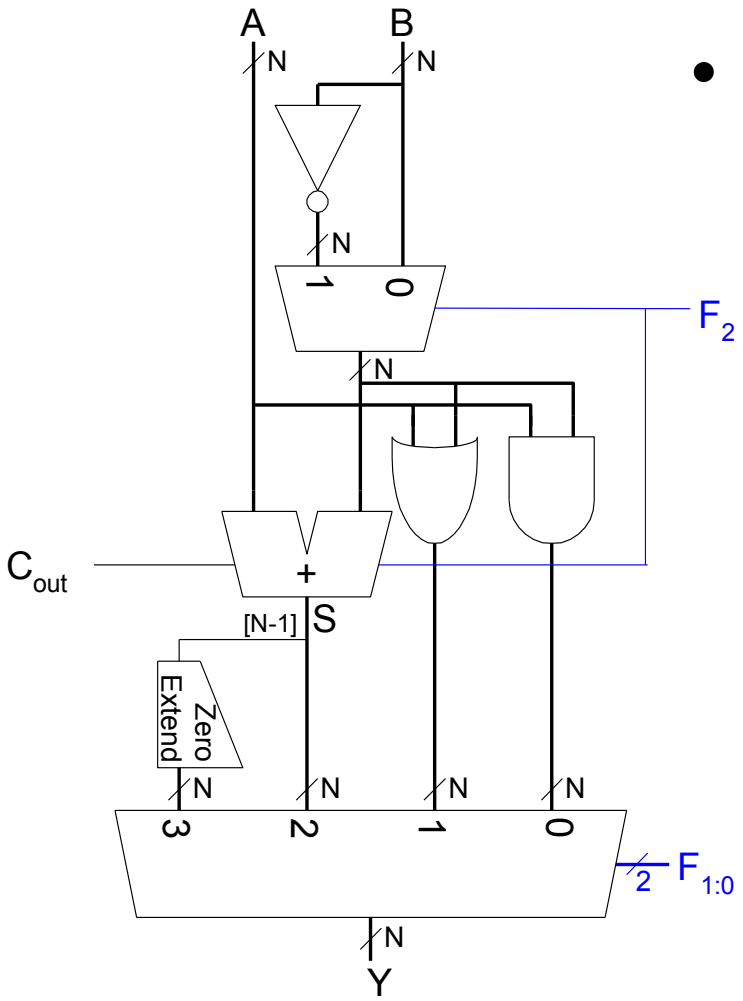
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Set Less Than (SLT) Example



- Configure 32-bit ALU for SLT operation: $A = 25$ and $B = 32$

Set Less Than (SLT) Example



- Configure 32-bit ALU for SLT operation: $A = 25$ and $B = 32$
  - $A < B$, so Y should be 32-bit representation of 1 (0x00000001)
  - $F_{2:0} = 111$
    - $F_2 = 1$ (adder acts as subtracter), so 25 - 32 = -7
    - -7 has 1 in the most significant bit ($S_{31} = 1$)
    - $F_{1:0} = 11$ multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

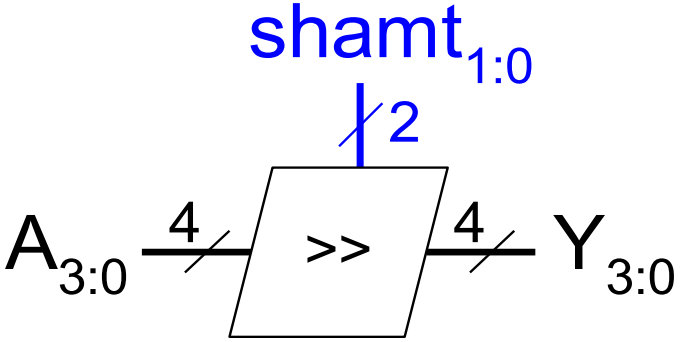
Shifters

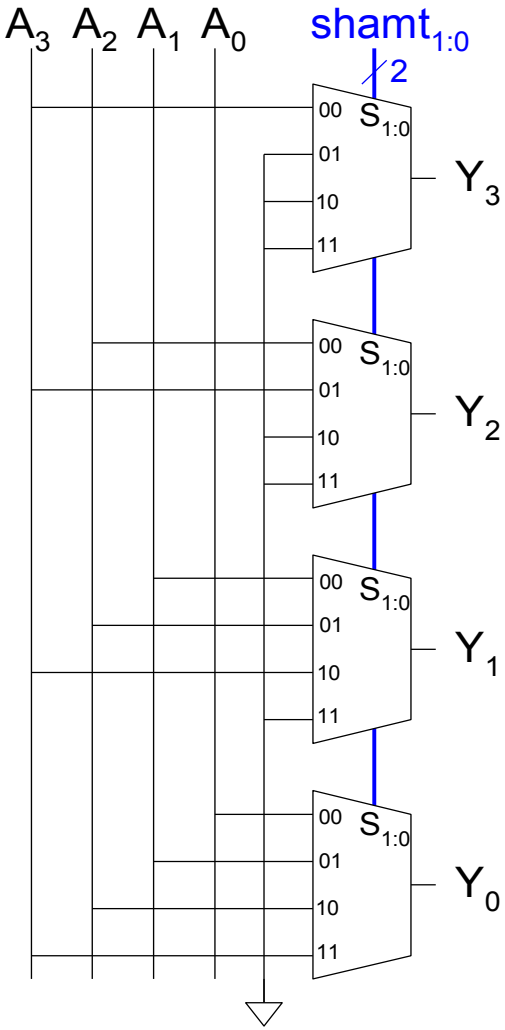
- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 =
  - Ex: 11001 << 2 =
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 =
  - Ex: 11001 <<< 2 =
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 =
  - Ex: 11001 ROL 2 =

Shifters

- **Logical shifter:**
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:**
  - Ex: 11001 >>> 2 = **11**110
  - Ex: 11001 <<< 2 = 00100
- **Rotator:**
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

Shifter Design





Shifters as Multipliers, Dividers

- $A << N = A \times 2^N$
  - **Example:** 00001 << 2 = 00100 $(1 \times 2^2 = 4)$
  - **Example:** 11101 << 2 = 10100 $(-3 \times 2^2 = -12)$
- $A >>> N = A \div 2^N$
  - **Example:** 01000 >>> 2 = 00010 $(8 \div 2^2 = 2)$
  - **Example:** 10000 >>> 2 = 11100 $(-16 \div 2^2 = -4)$

Multipliers

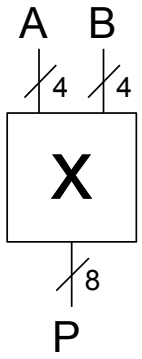
- **Partial products** formed by multiplying a single digit of the multiplier with multiplicand
- **Shifted** partial products **summed** to form Mlt

Decimal		Binary
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
9660		0101
		+ 0000
	result	0100011

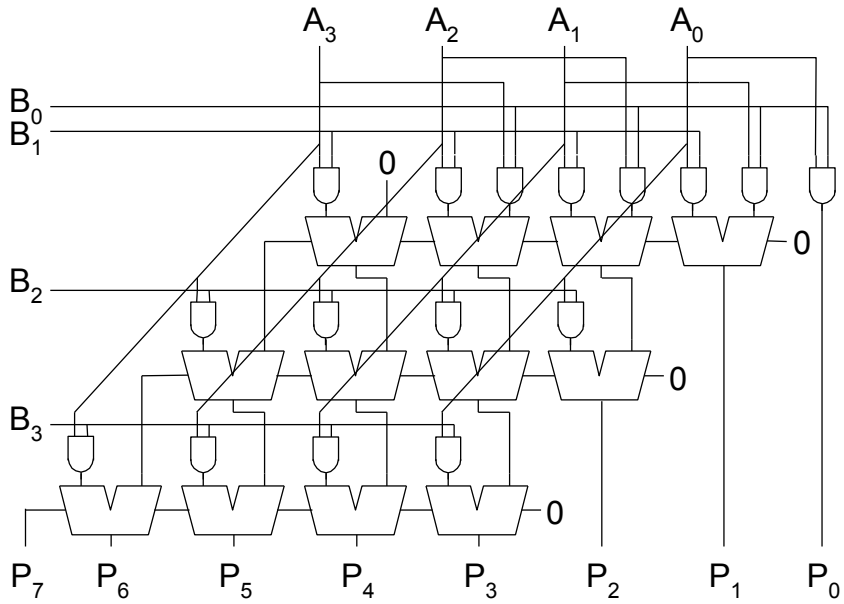
230 x 42 = 9660

5 x 7 = 35

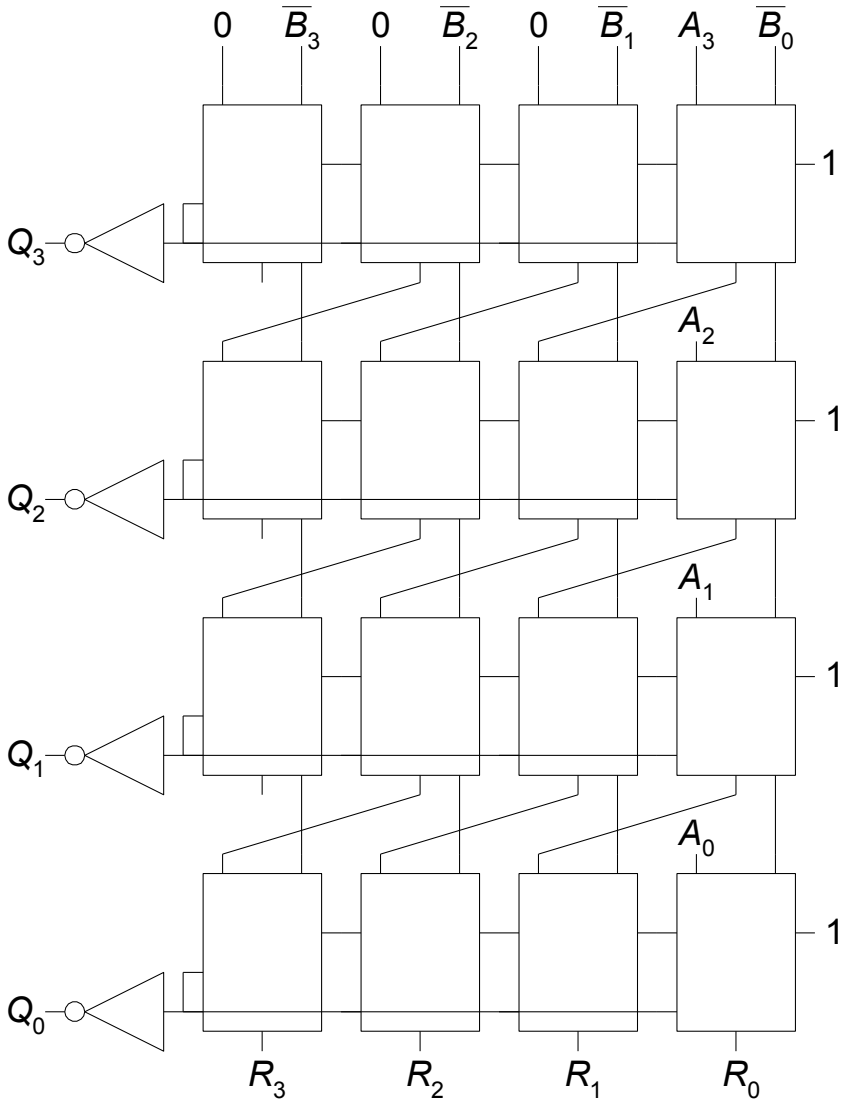
4 x 4 Multiplier

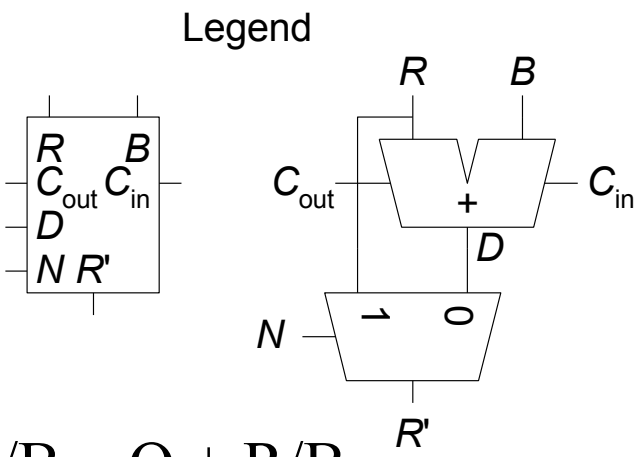


$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
\times & & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & & \\
 & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



4 x 4 Divider





A/B = Q + R/B

Algorithm:

R' = 0

for i = N-1 to 0

  R = {R' << 1. A_i}

  D = R - B

  if D < 0, Q_i=0, R'=R

  else Q_i=1, R'=D

R'=R

DIGITAL BUILDING BLOCKS

Chapter 7

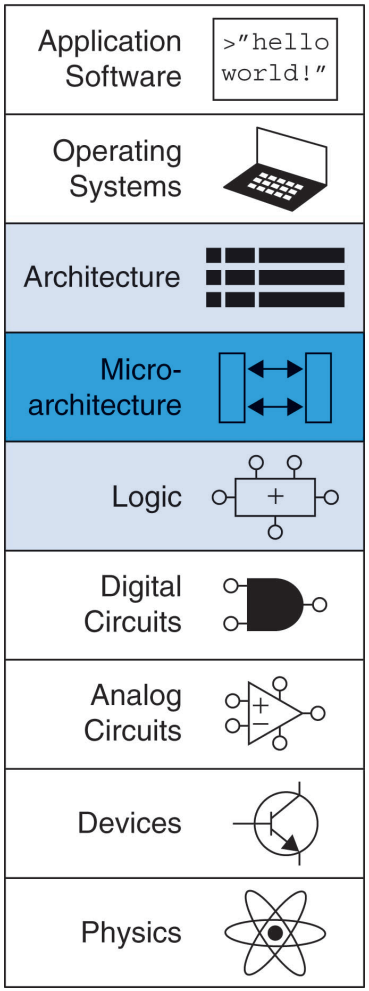
***Digital Design and Computer Architecture*, 2nd Edition**

David Money Harris and Sarah L. Harris

ELSEVIER

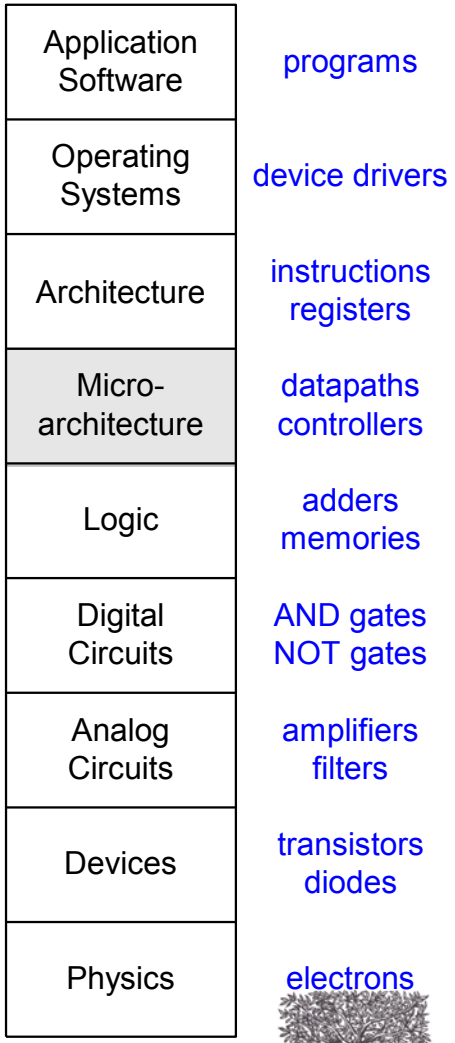
Chapter 7 :: Topics

- **Introduction**
- **Performance Analysis**
- **Single-Cycle Processor**
- **Multicycle Processor**
- **Pipelined Processor**
- **Exceptions**
- **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 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

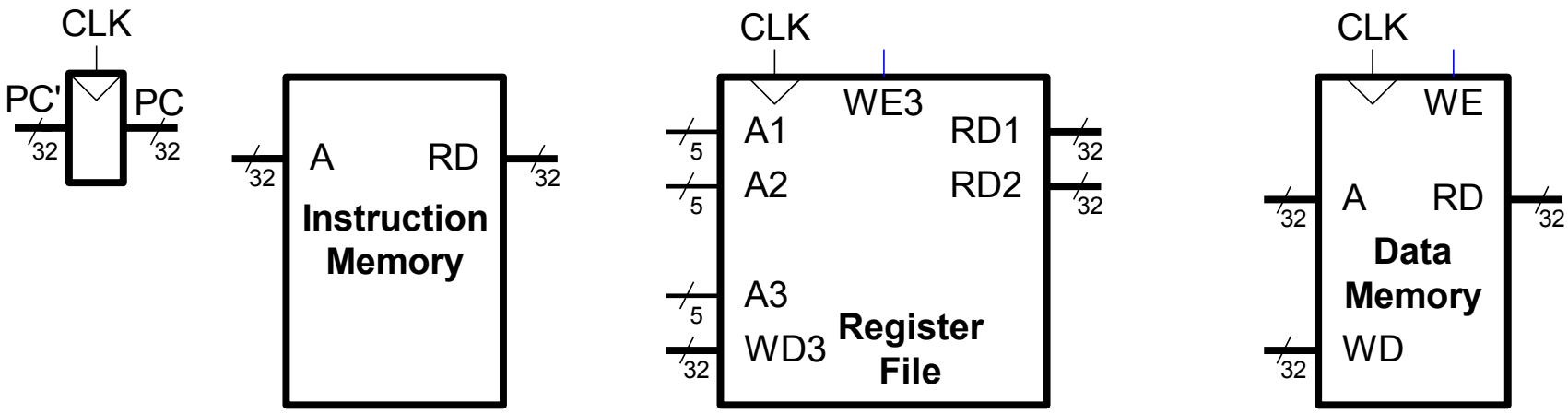
MIPS Processor

- Consider subset of MIPS instructions:
  - R-type instructions: `and`, `or`, `add`, `sub`, `slt`
  - Memory instructions: `lw`, `sw`
  - Branch instructions: `beq`

Architectural State

- Determines everything about a processor:
  - PC
  - 32 registers (and a few extra ones we'll ignore here)
  - Memory

MIPS State Elements

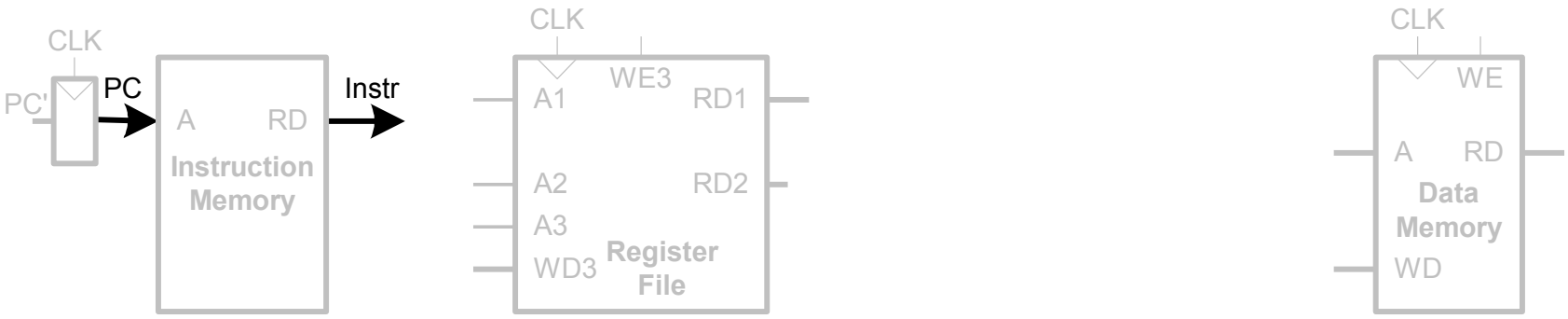


Single-Cycle MIPS Processor

- Datapath
- Control

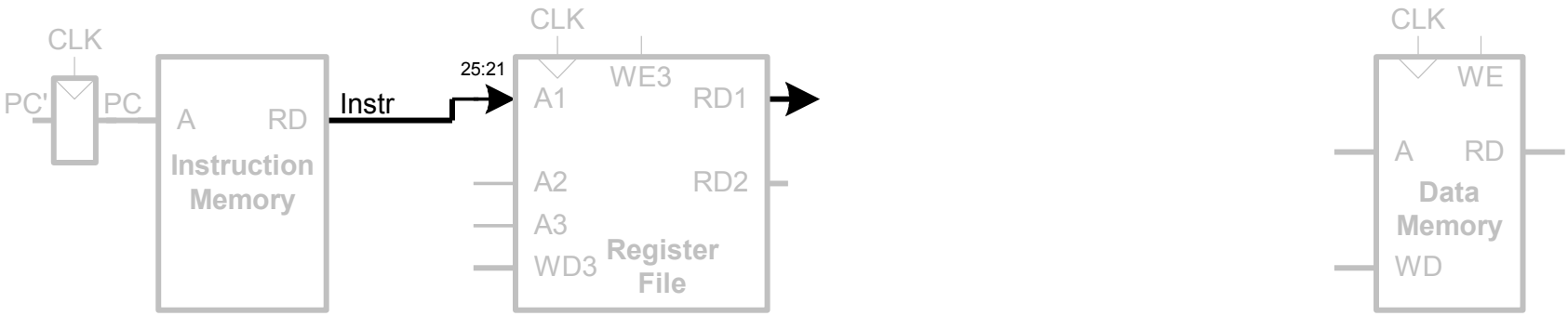
Single-Cycle Datapath: `lw` fetch

STEP 1: Fetch instruction



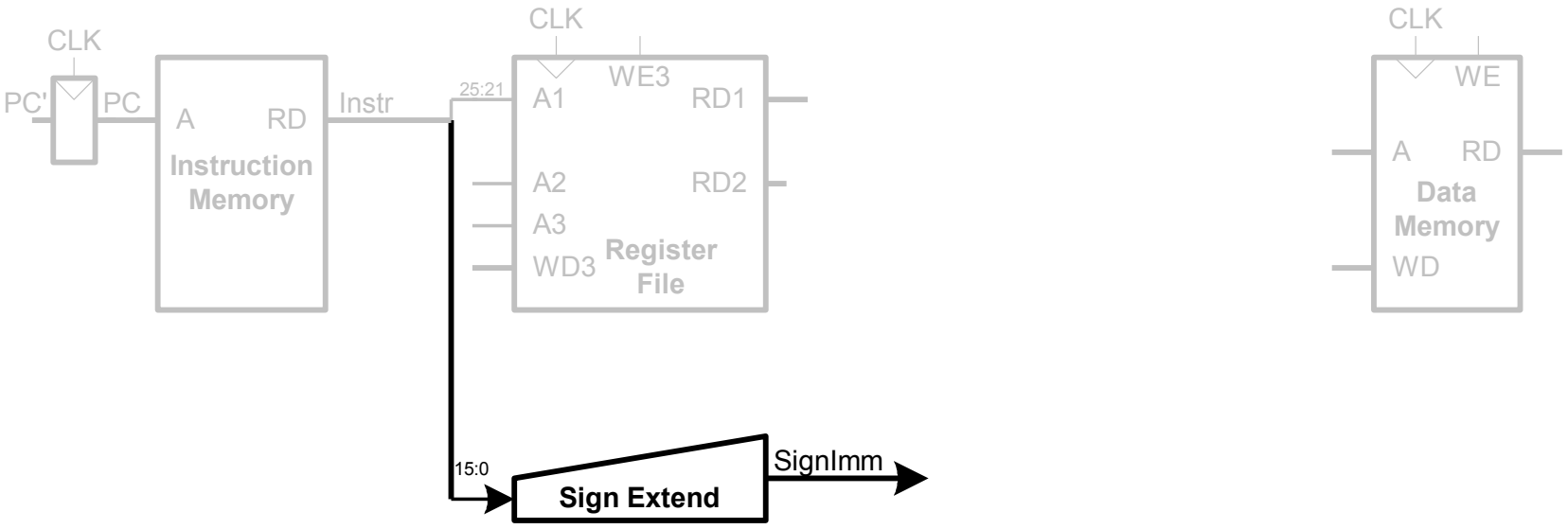
Single-Cycle Datapath: `lw` Register Read

STEP 2: Read source operands from RF



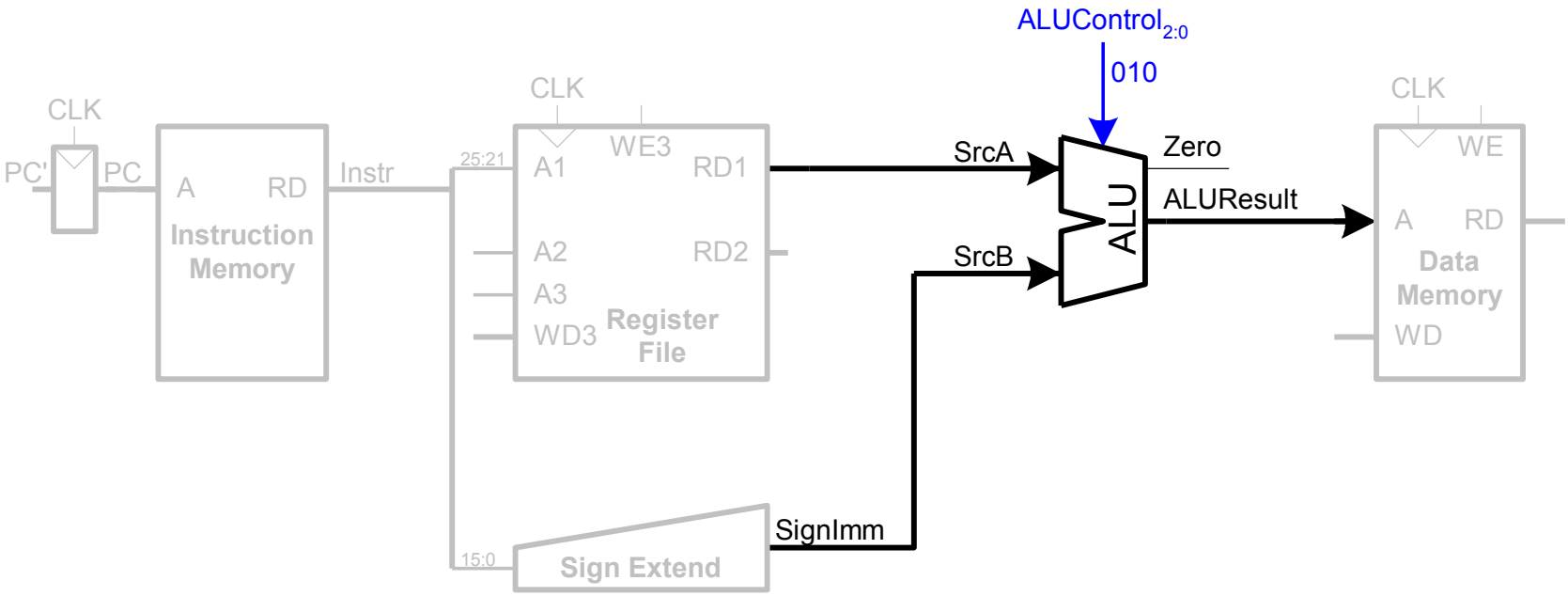
Single-Cycle Datapath: `lw` Immediate

STEP 3: Sign-extend the immediate



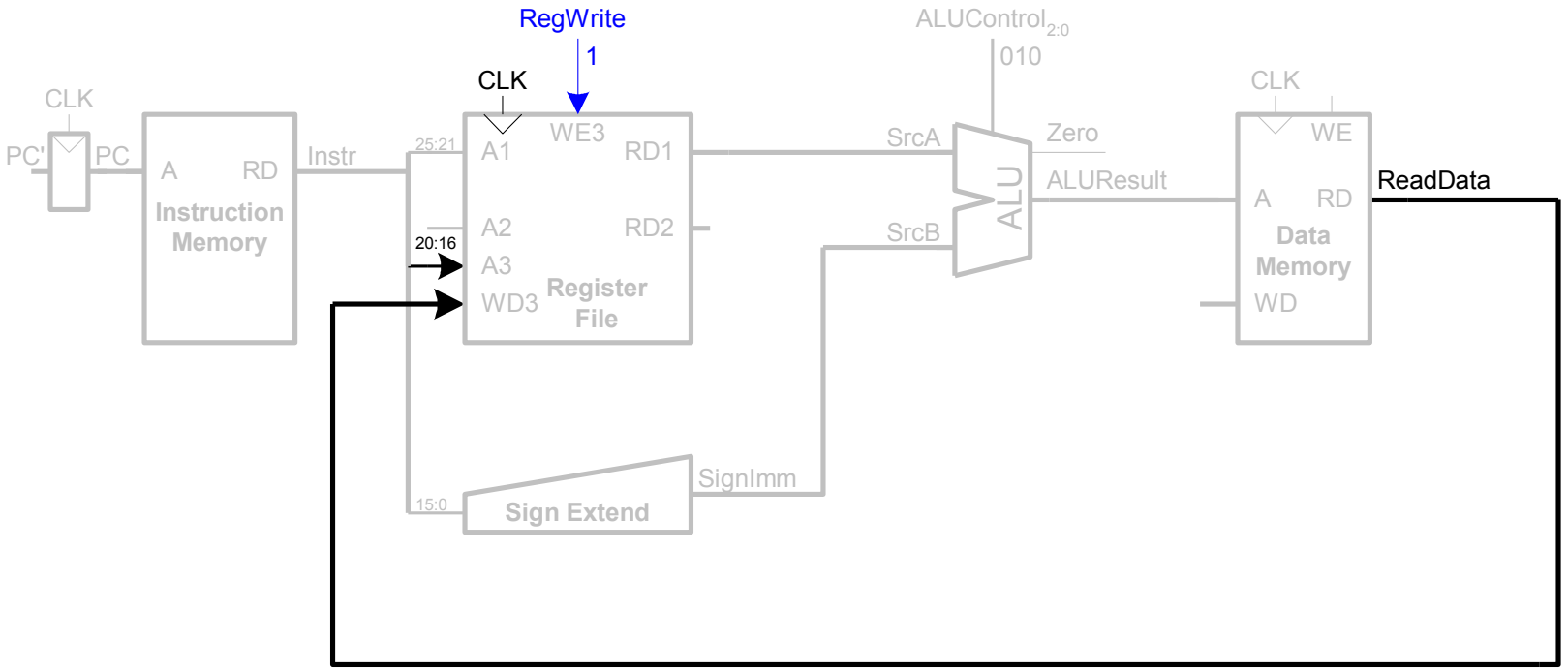
Single-Cycle Datapath: `lw` address

STEP 4: Compute the memory address



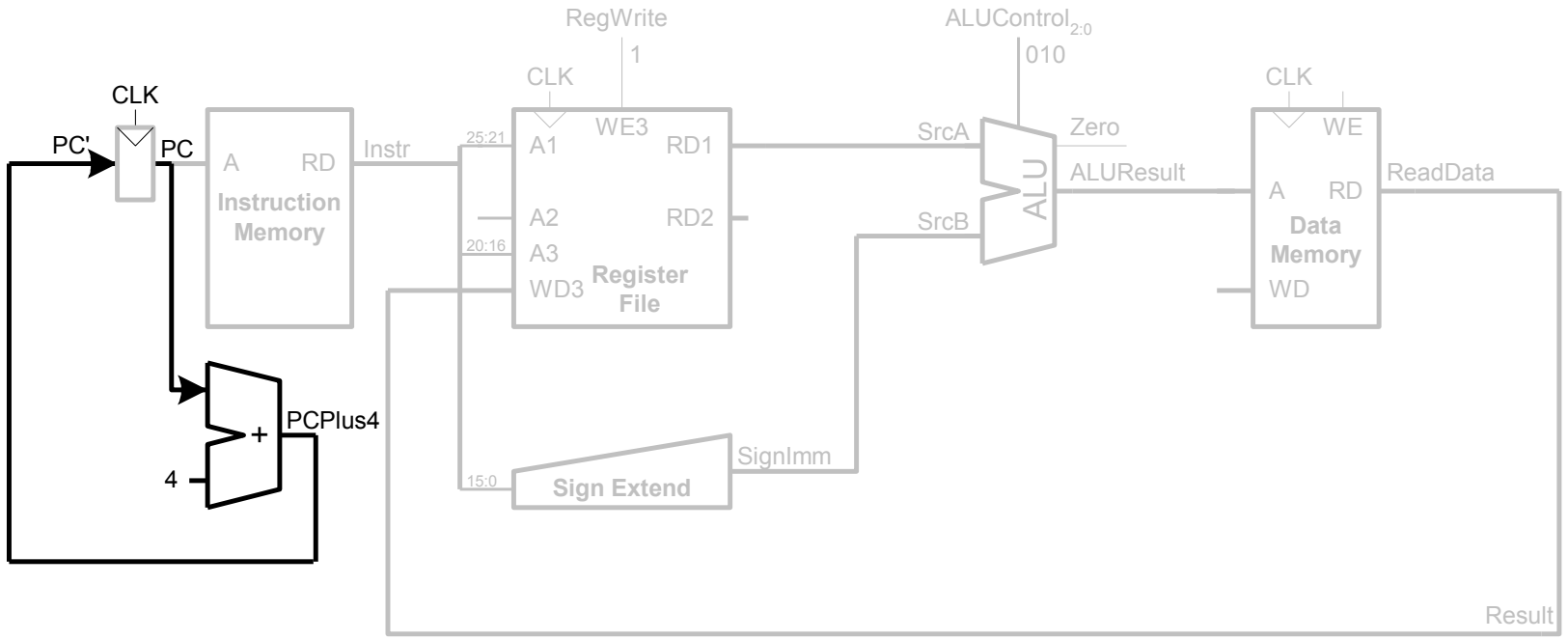
Single-Cycle Datapath: `lw` Memory Read

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



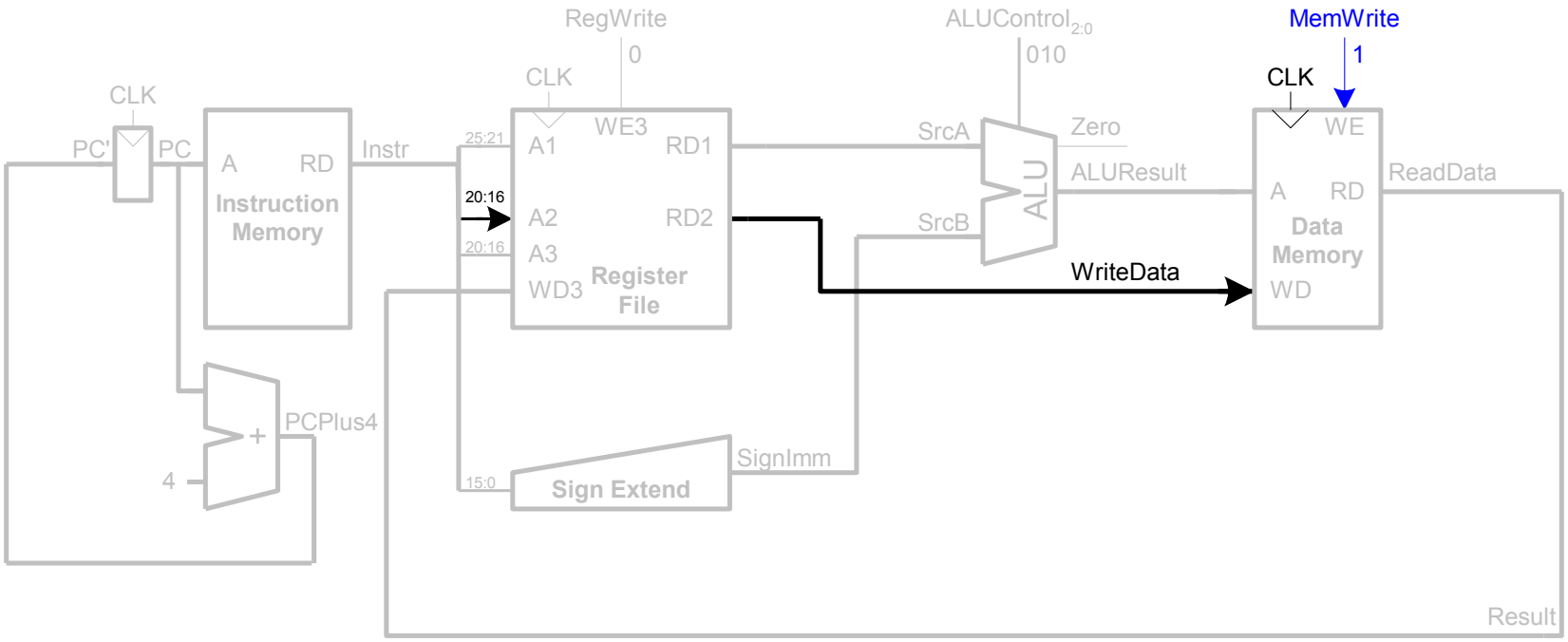
Single-Cycle Datapath: `lw` PC Increment

STEP 6: Determine address of next instruction



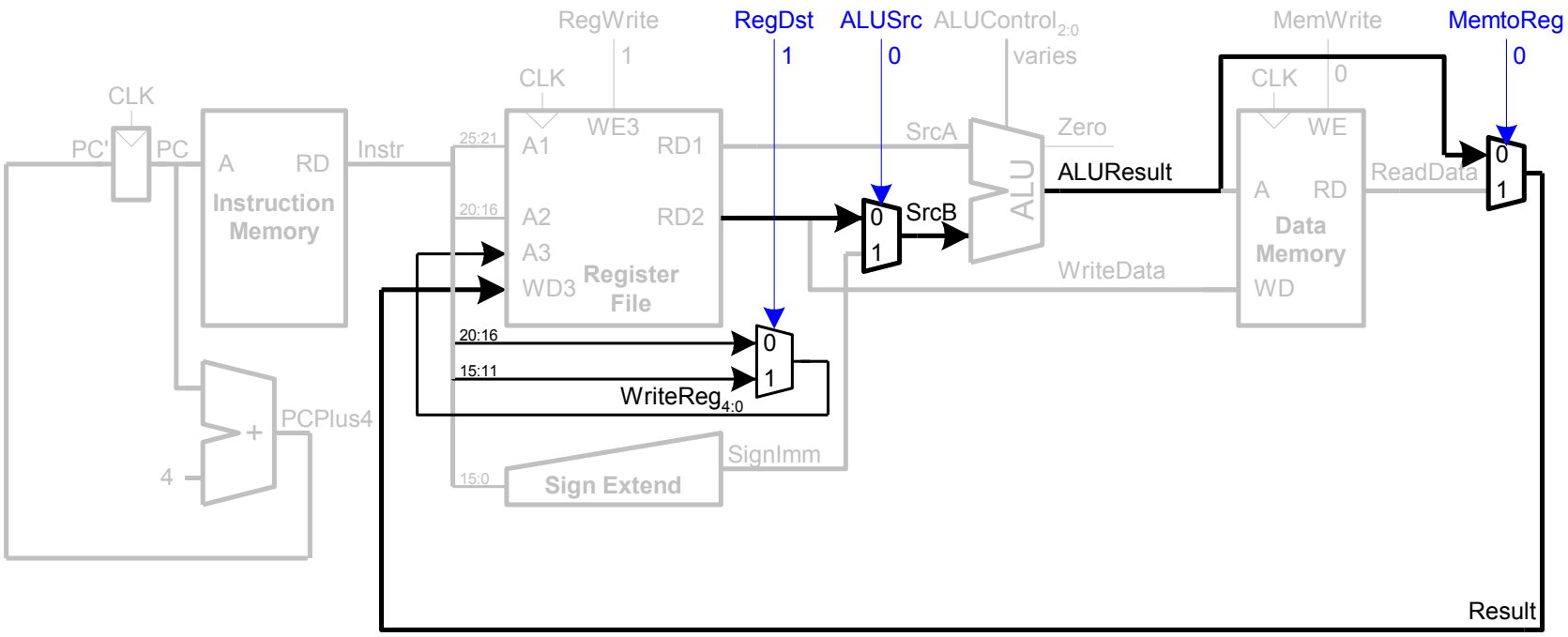
Single-Cycle Datapath: `sw`

Write data in `rt` to memory



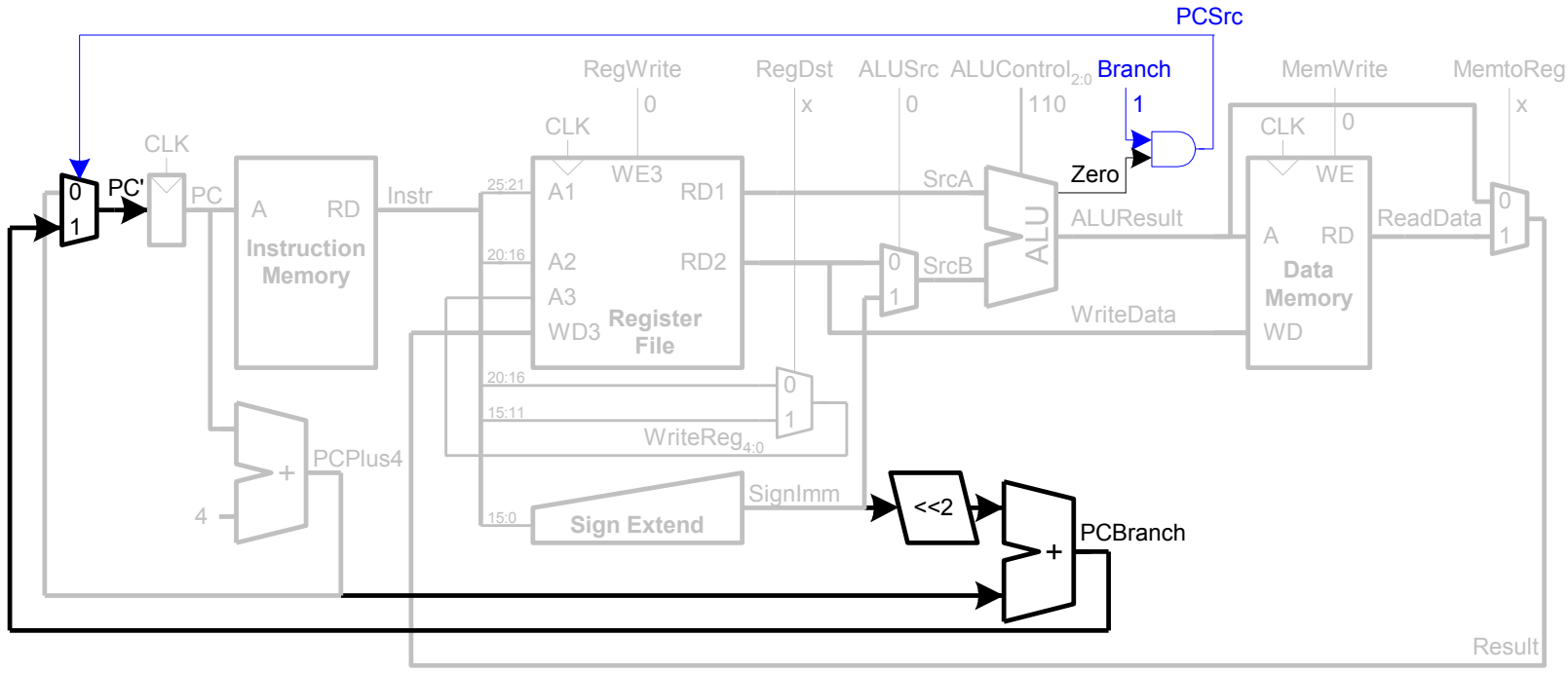
Single-Cycle Datapath: R-Type

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

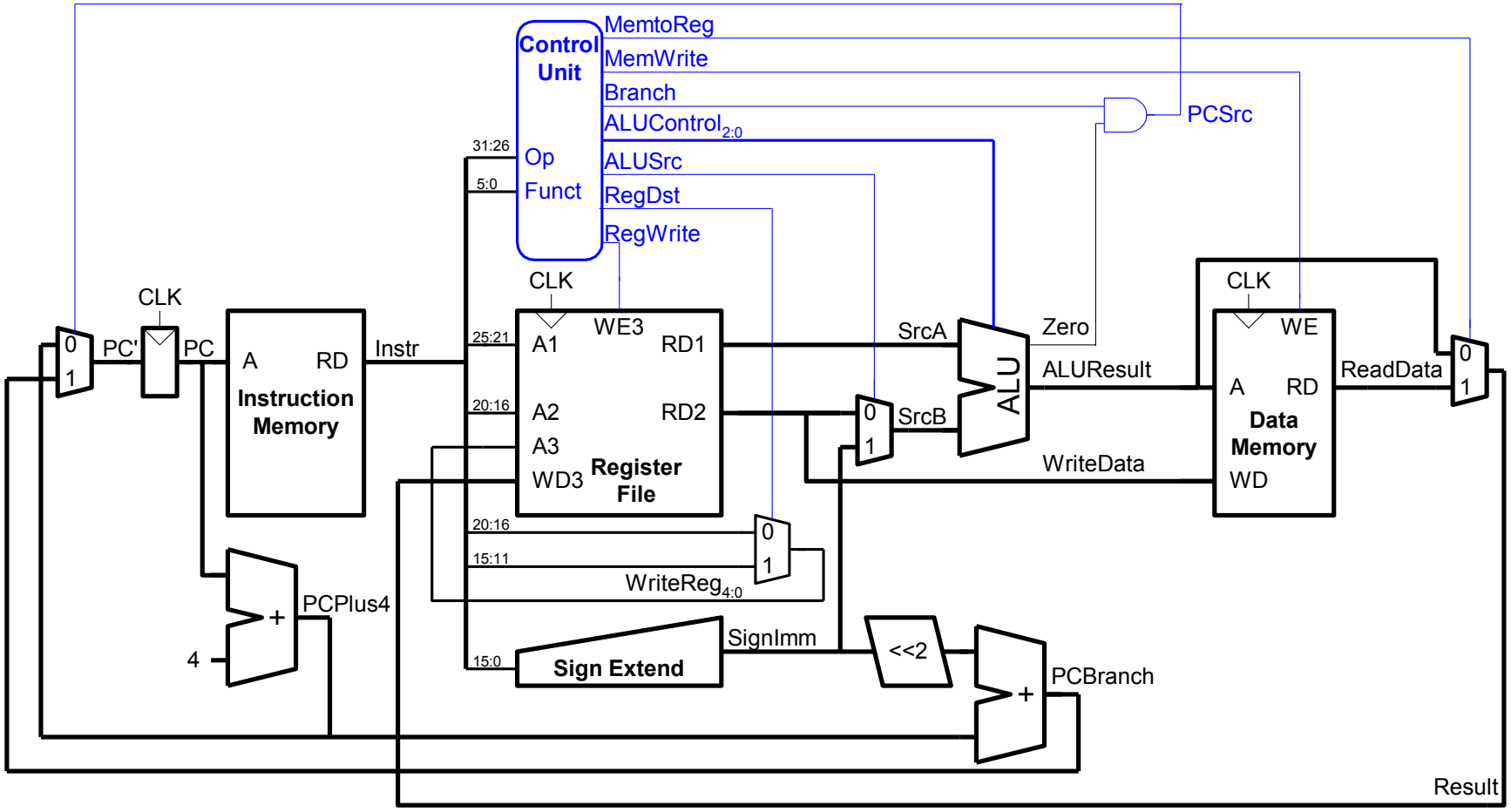


Single-Cycle Datapath: beq

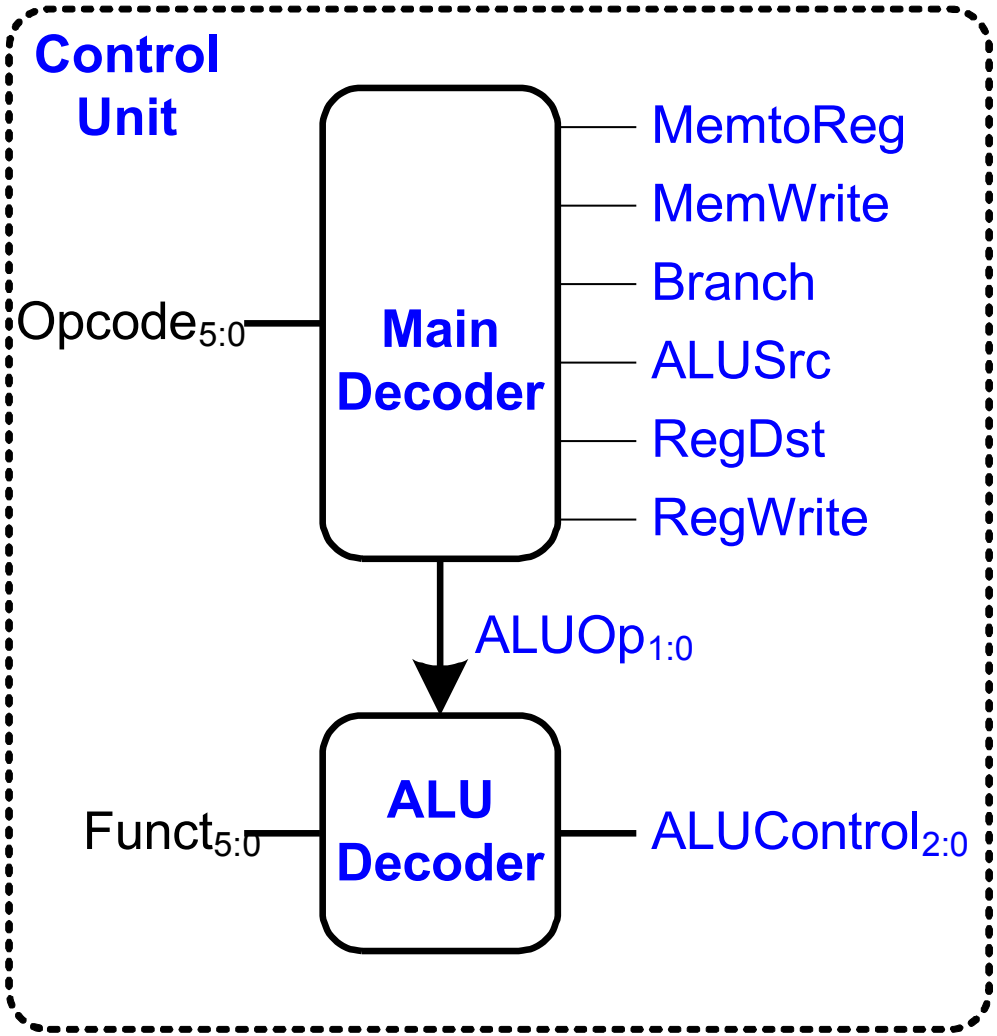
- Determine whether values in rs and rt are equal
- Calculate branch target address:
  BTA = (sign-extended immediate << 2) + (PC+4)



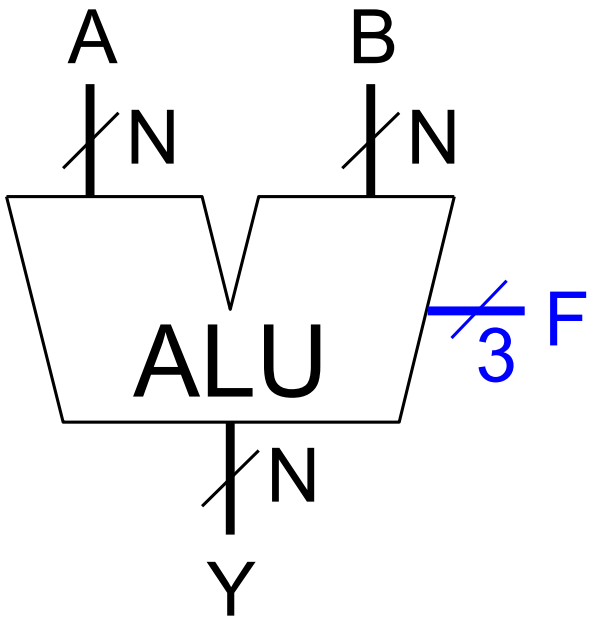
Single-Cycle Processor



Single-Cycle Control

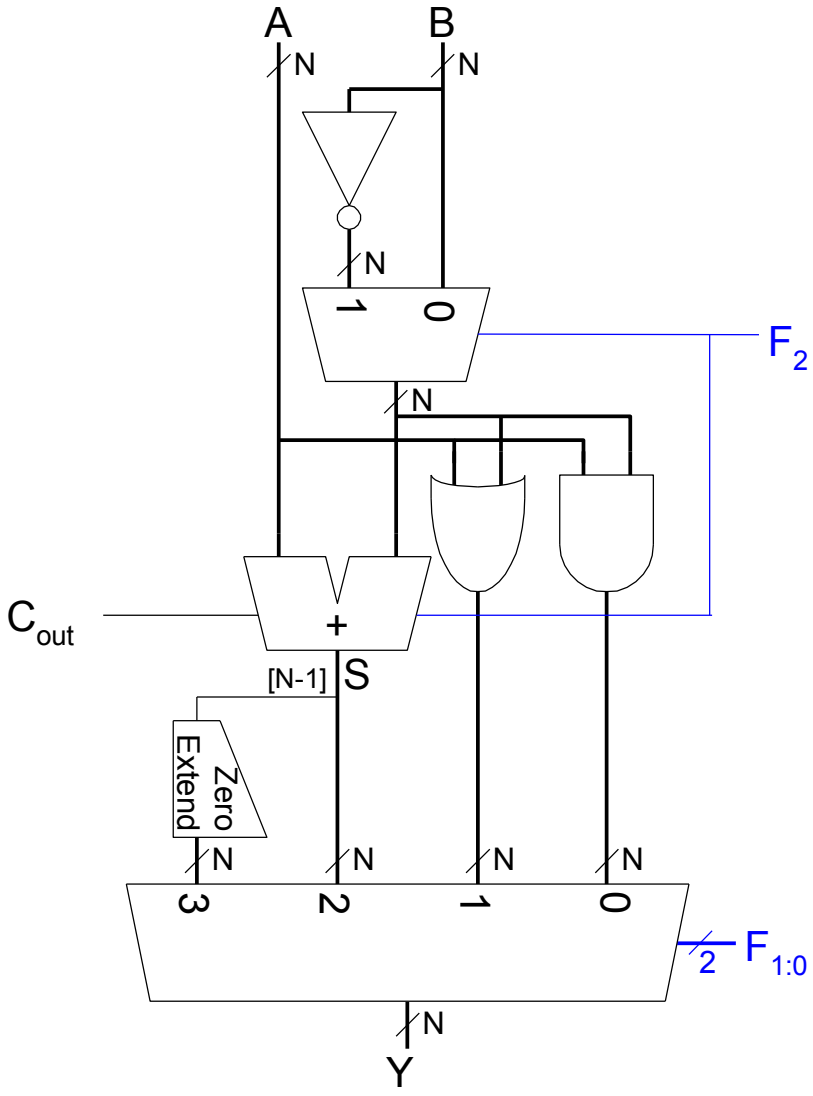


Review: ALU



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Review: ALU



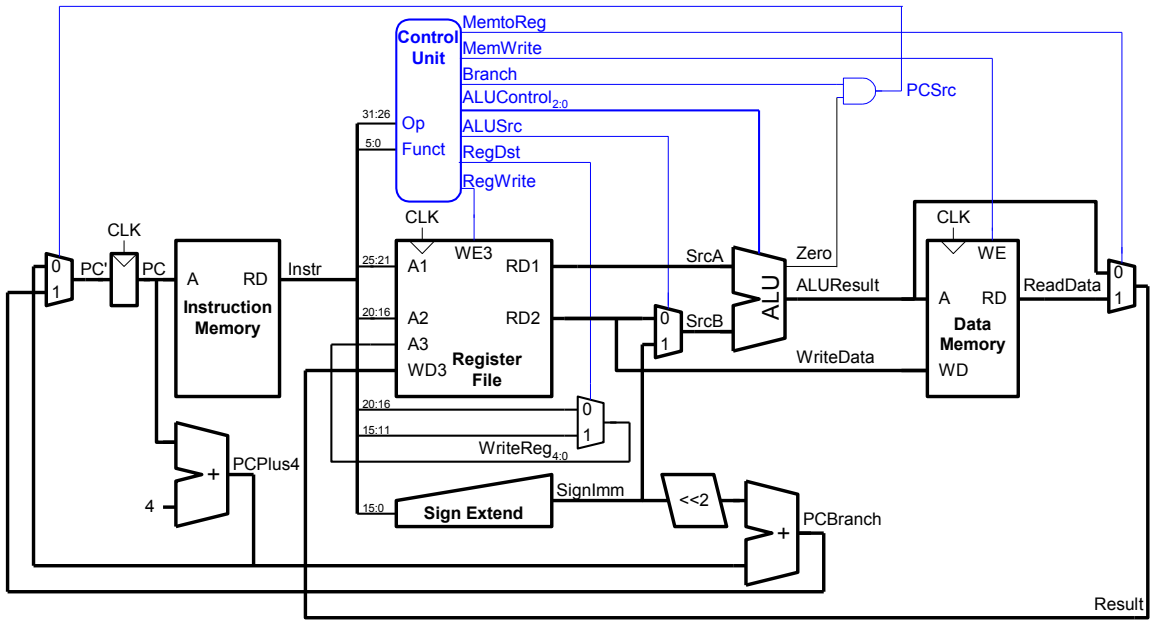
Control Unit: ALU Decoder

$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (`add`)	010 (Add)
1X	100010 (`sub`)	110 (Subtract)
1X	100100 (`and`)	000 (And)
1X	100101 (`or`)	001 (Or)
1X	101010 (`slt`)	111 (SLT)

Control Unit Main Decoder

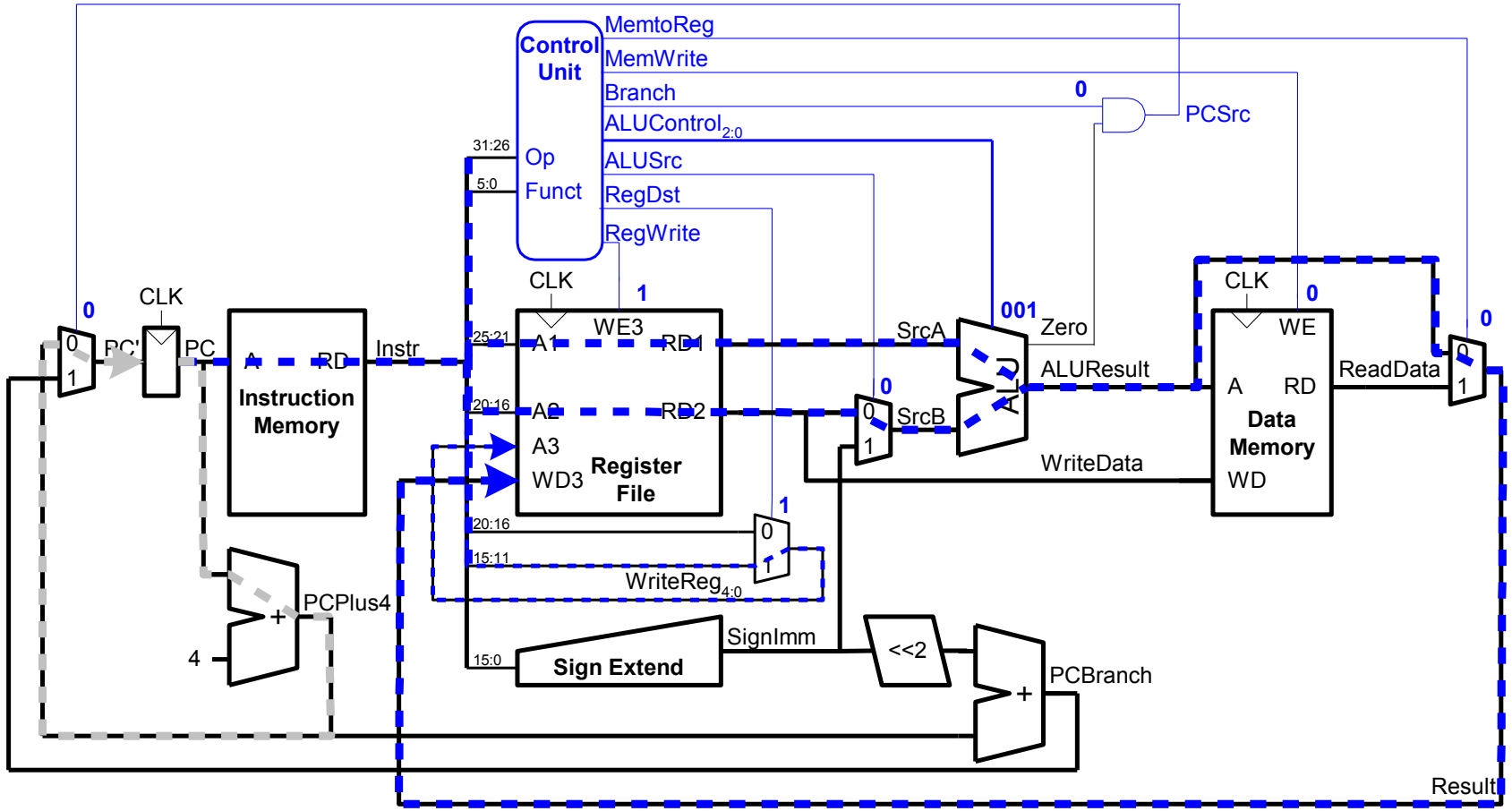
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000							
`lw`	100011							
`sw`	101011							
`beq`	000100							



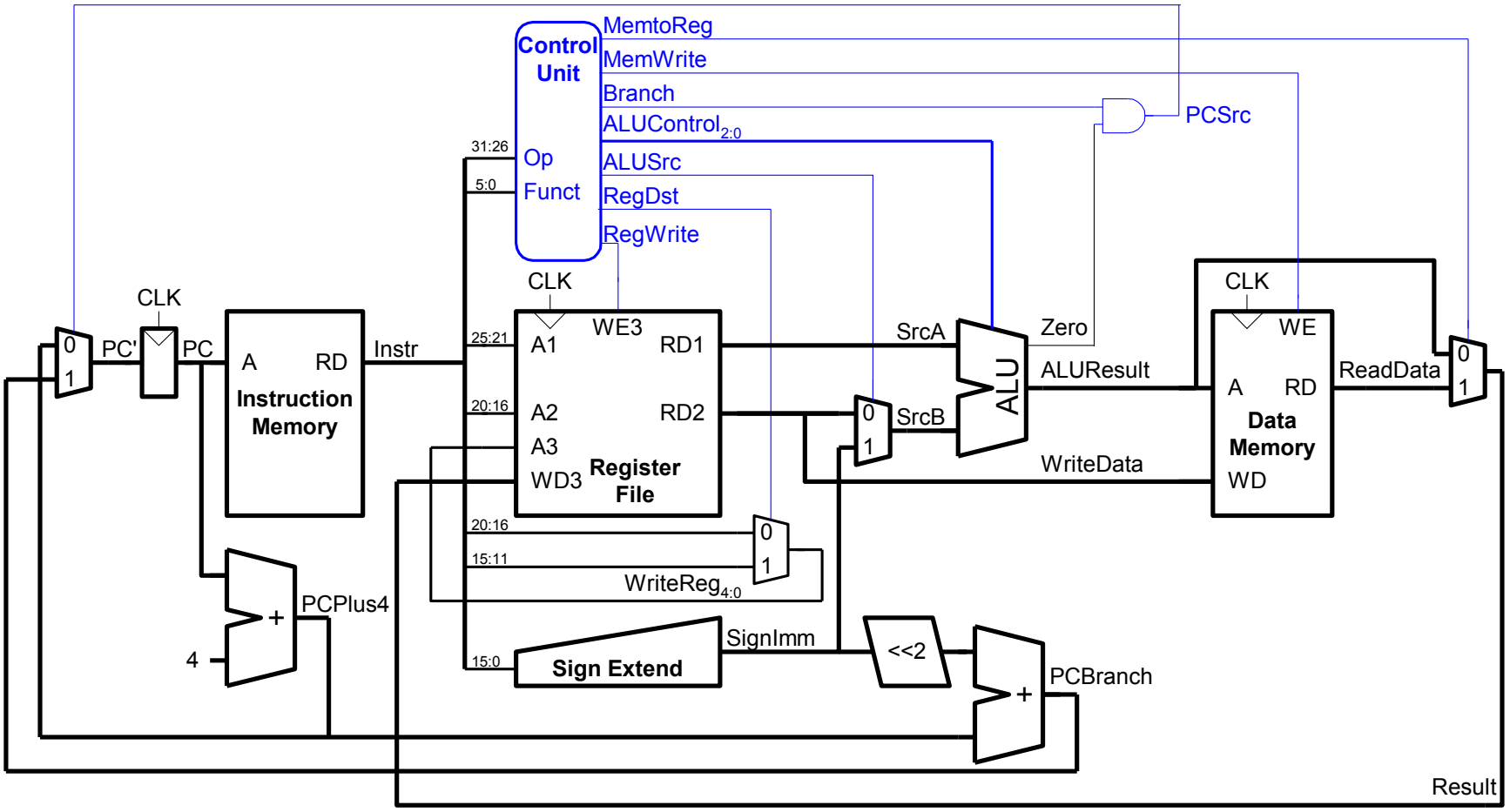
Control Unit: Main Decoder

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	0	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

Single-Cycle Datapath: `or`



Extended Functionality: addi



No change to datapath

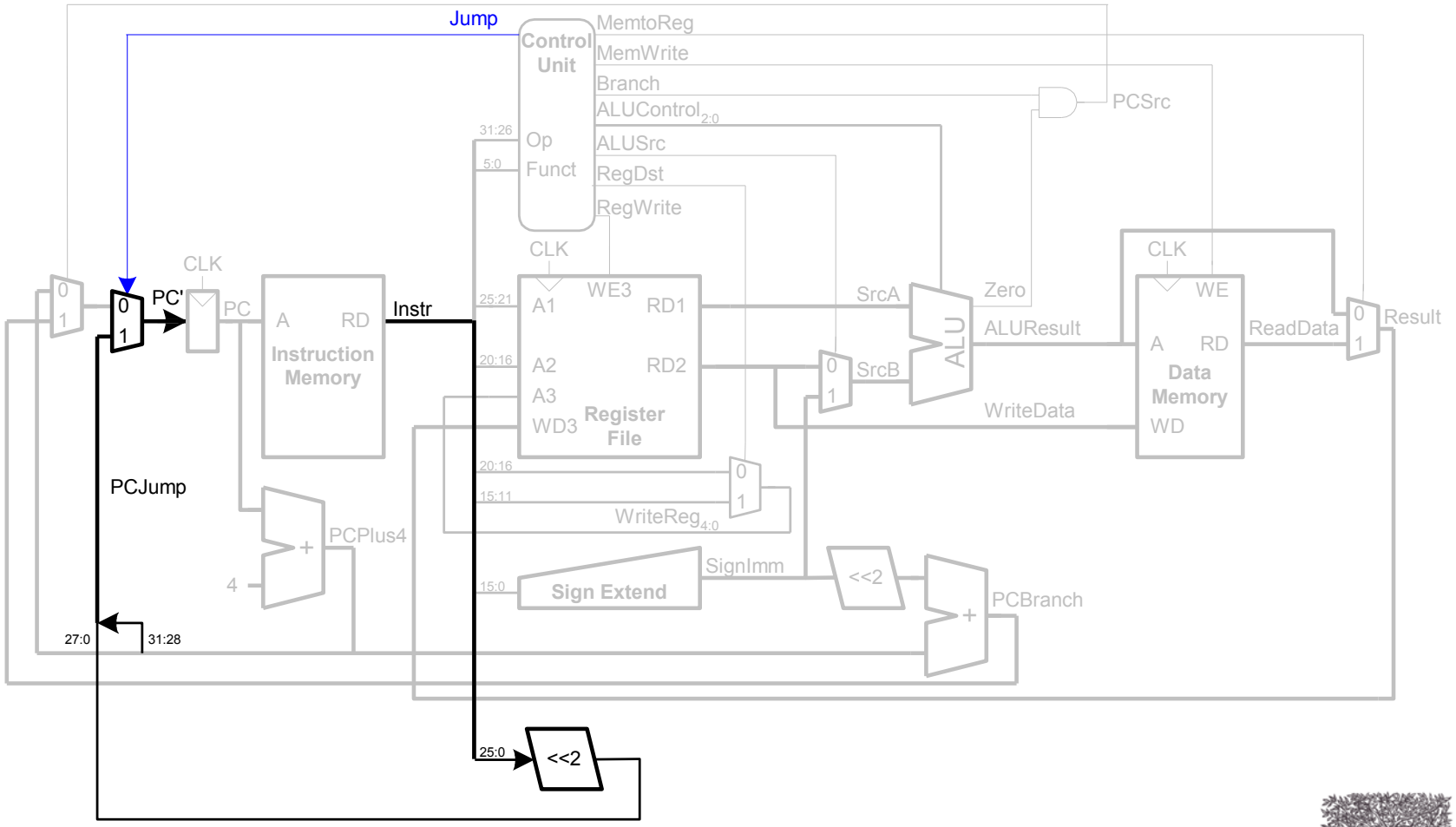
Control Unit: addi

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**							

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



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

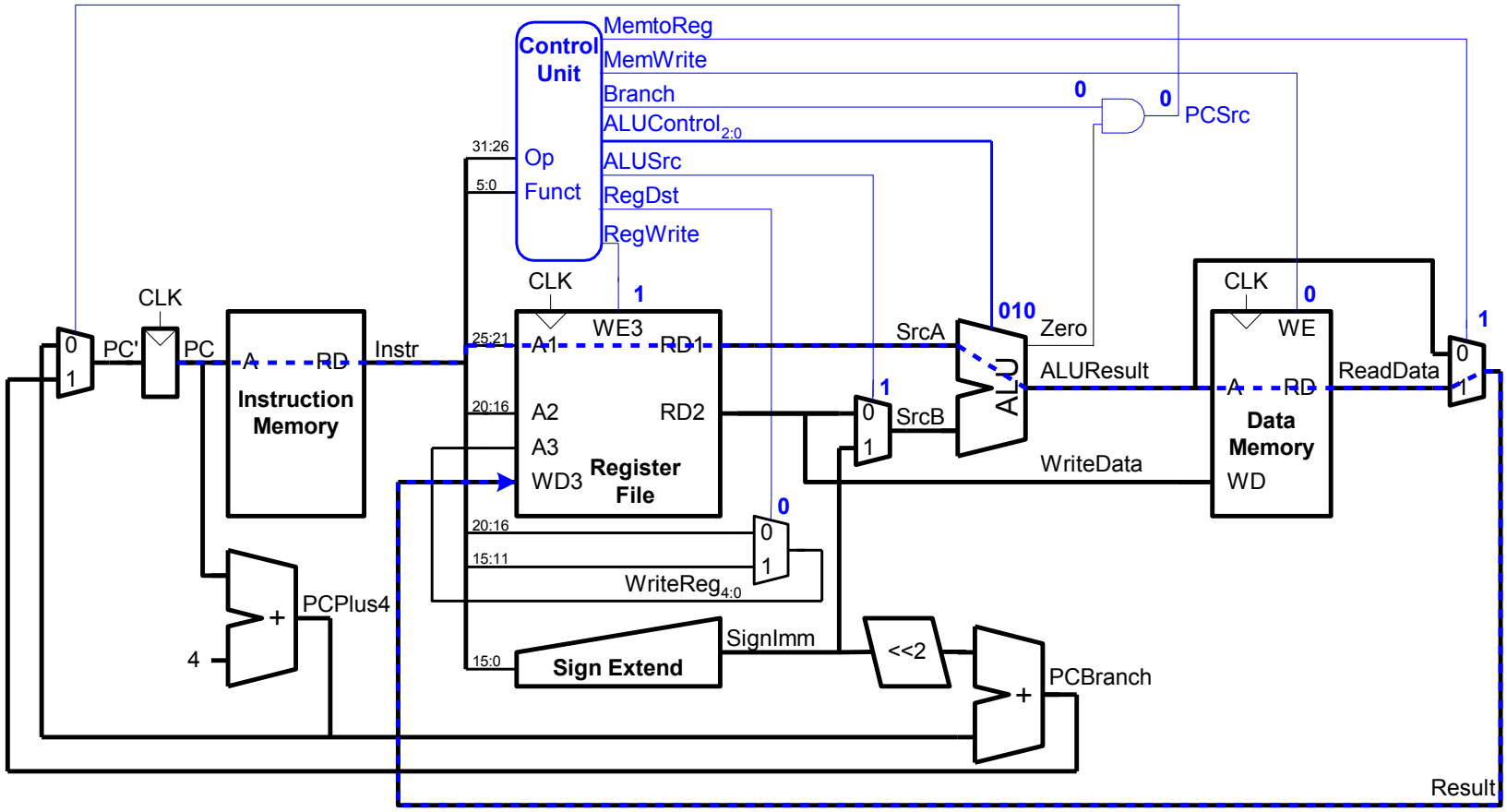
Review: Processor Performance

Program Execution Time

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

= # instructions x CPI x T_C

Single-Cycle Performance



T_C limited by critical path (`lw`)

Single-Cycle Performance

- Single-cycle critical path:
  $T_c = t_{pcq\_PC} + t_{\text{mem}} + \max(t_{RF\text{read}}, t_{sext} + t_{\text{mux}}) + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$
- 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

$$T_c = ?$$

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

- **Single-cycle:**

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

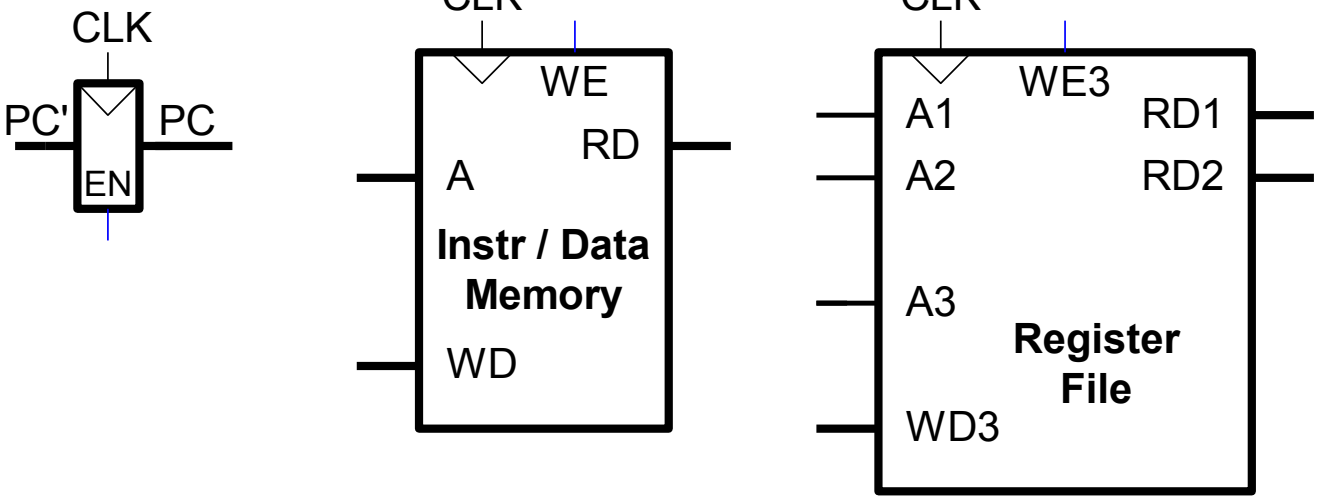
- **Multicycle:**

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

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

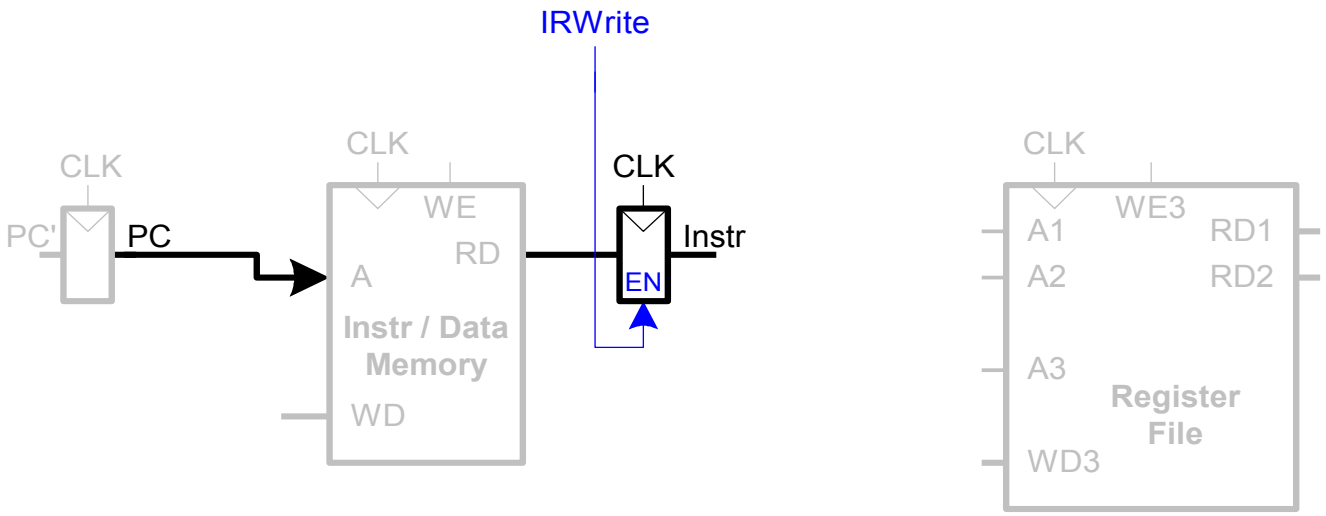
Multicycle State Elements

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



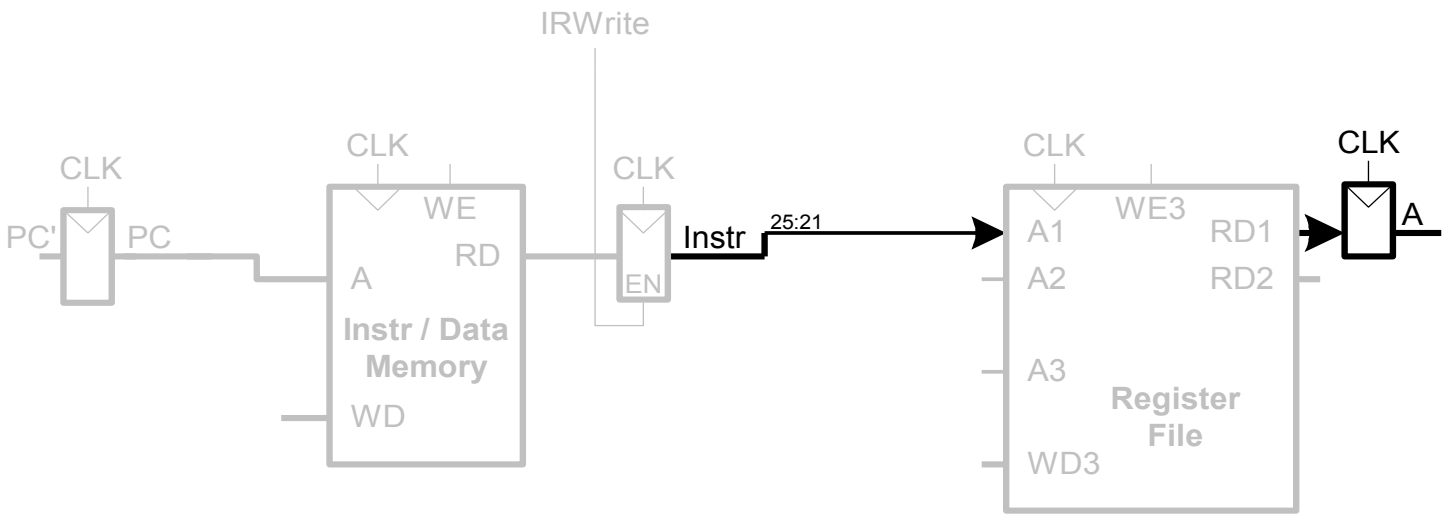
Multicycle Datapath: Instruction Fetch

STEP 1: Fetch instruction



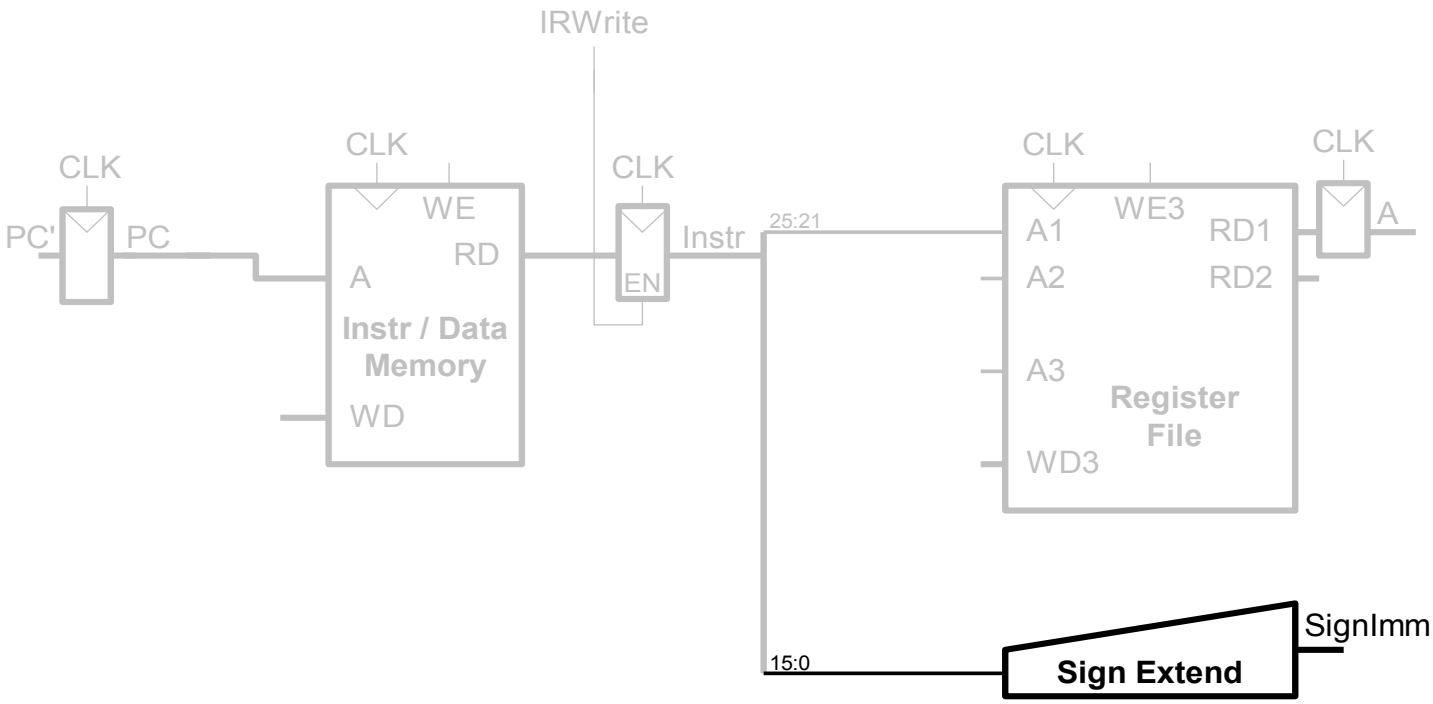
Multicycle Datapath: `lw` Register

STEP 2a: Read source operands from RF



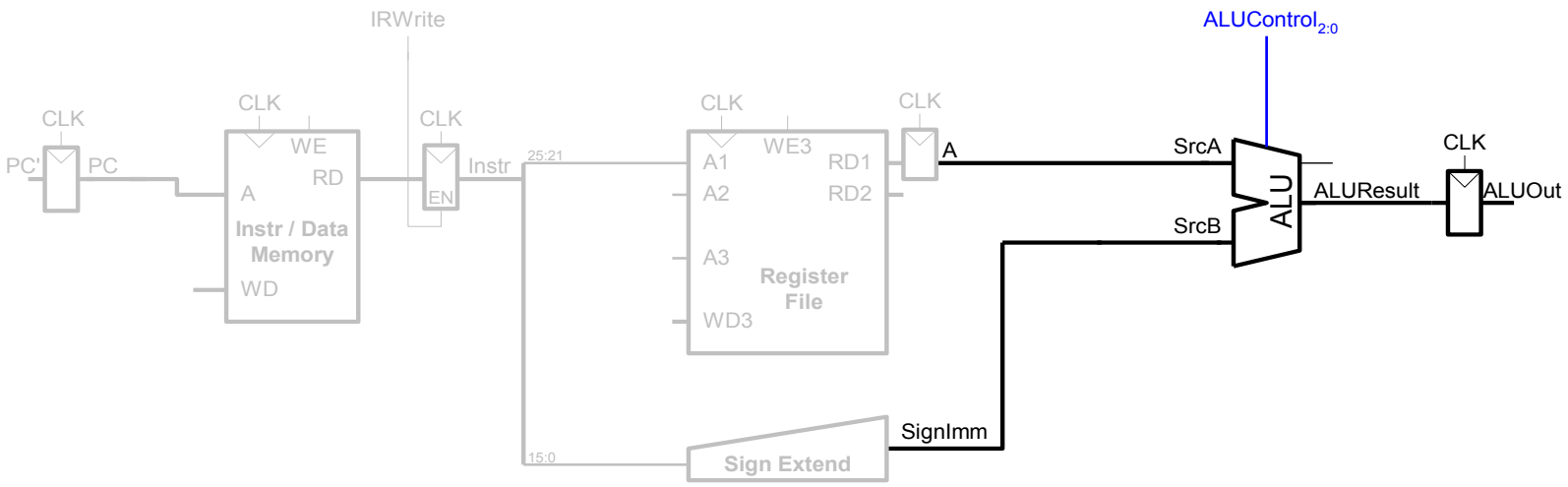
Multicycle Datapath: `lw` Immediate

STEP 2b: Sign-extend the immediate



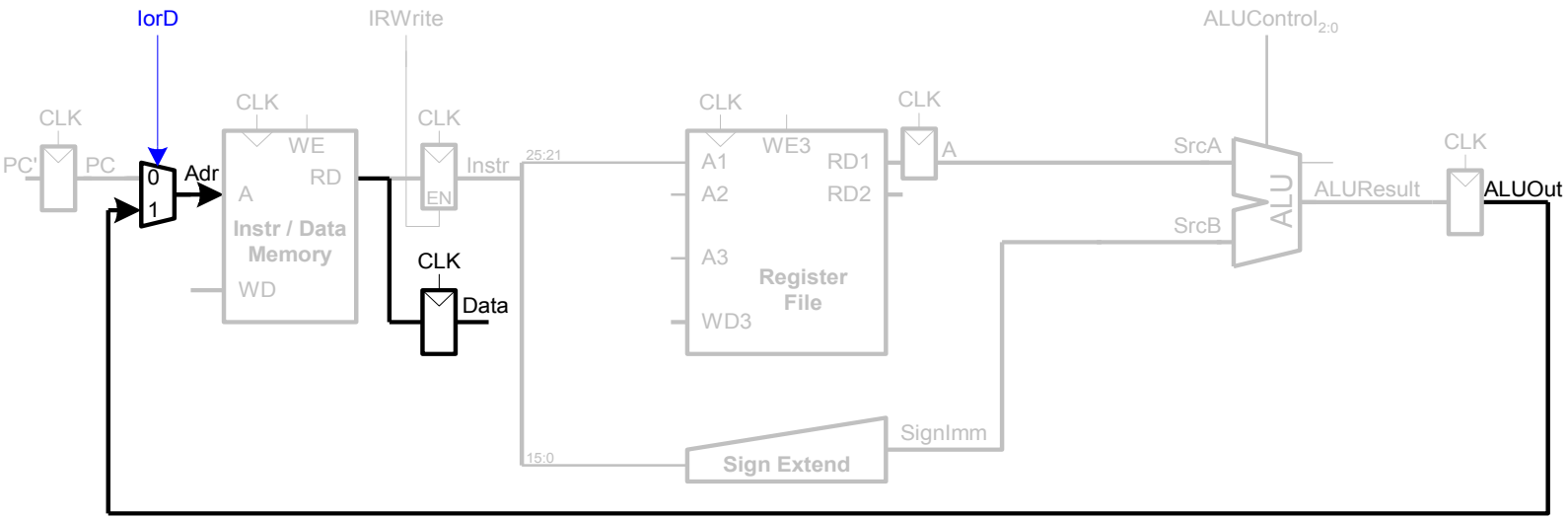
Multicycle Datapath: `lw` Address

STEP 3: Compute the memory address



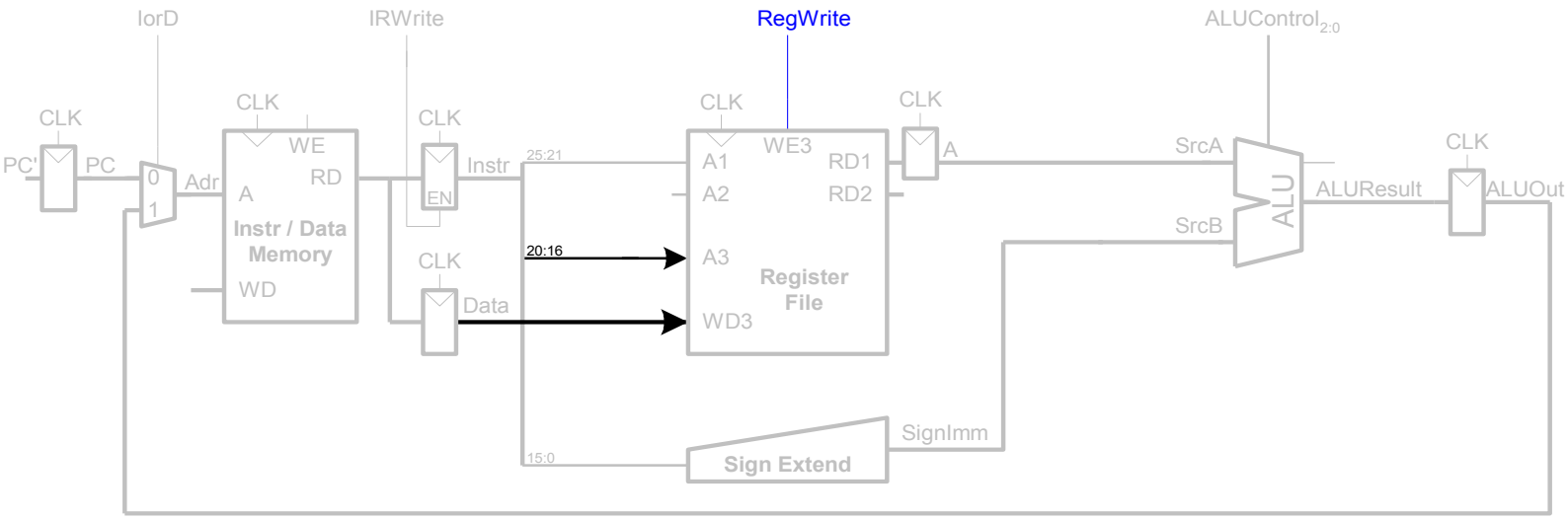
Multicycle Datapath: `lw` Memory Read

STEP 4: Read data from memory



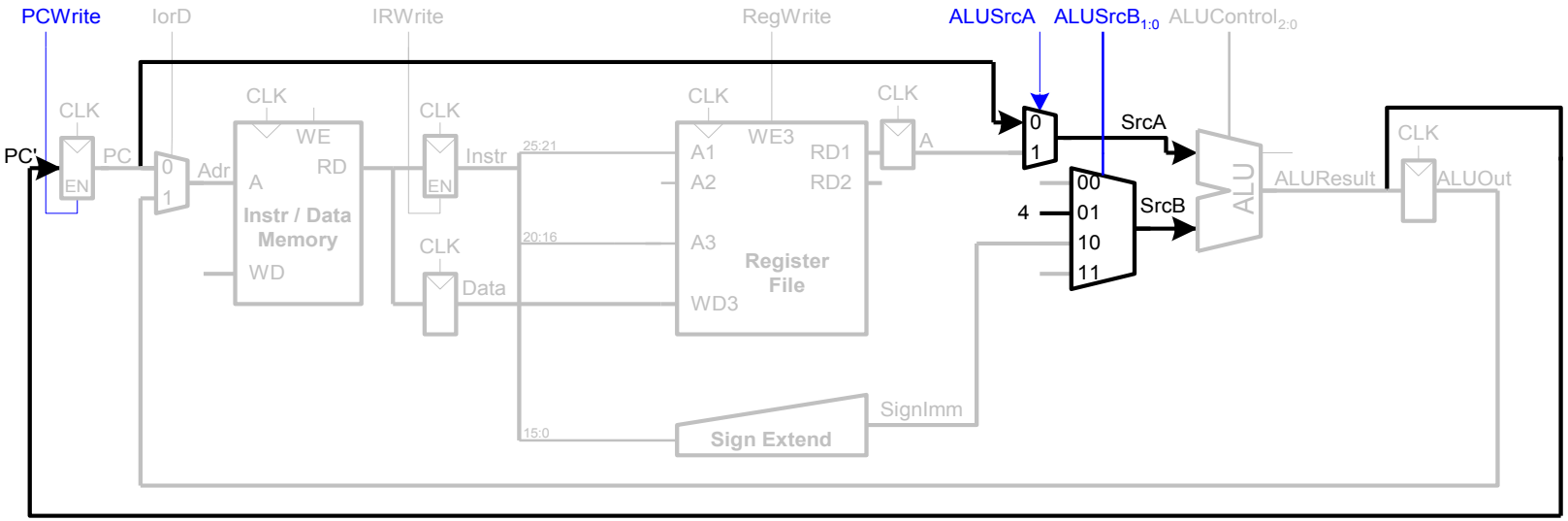
Multicycle Datapath: `lw` Write Register

STEP 5: Write data back to register file



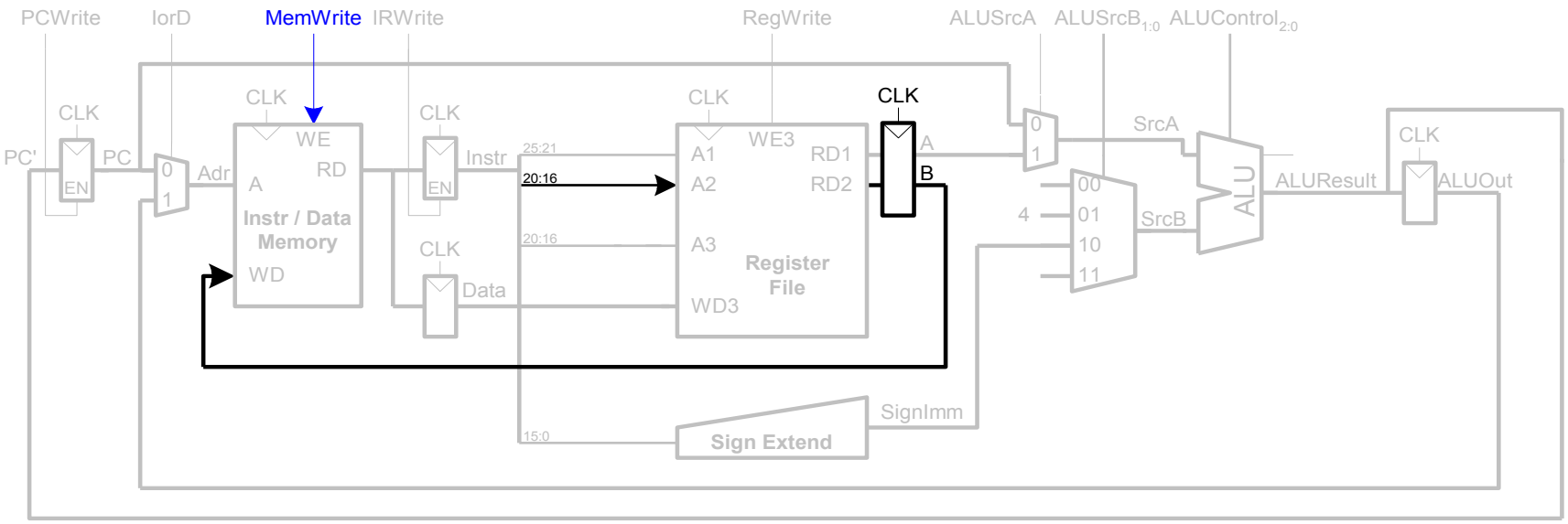
Multicycle Datapath: Increment PC

STEP 6: Increment PC



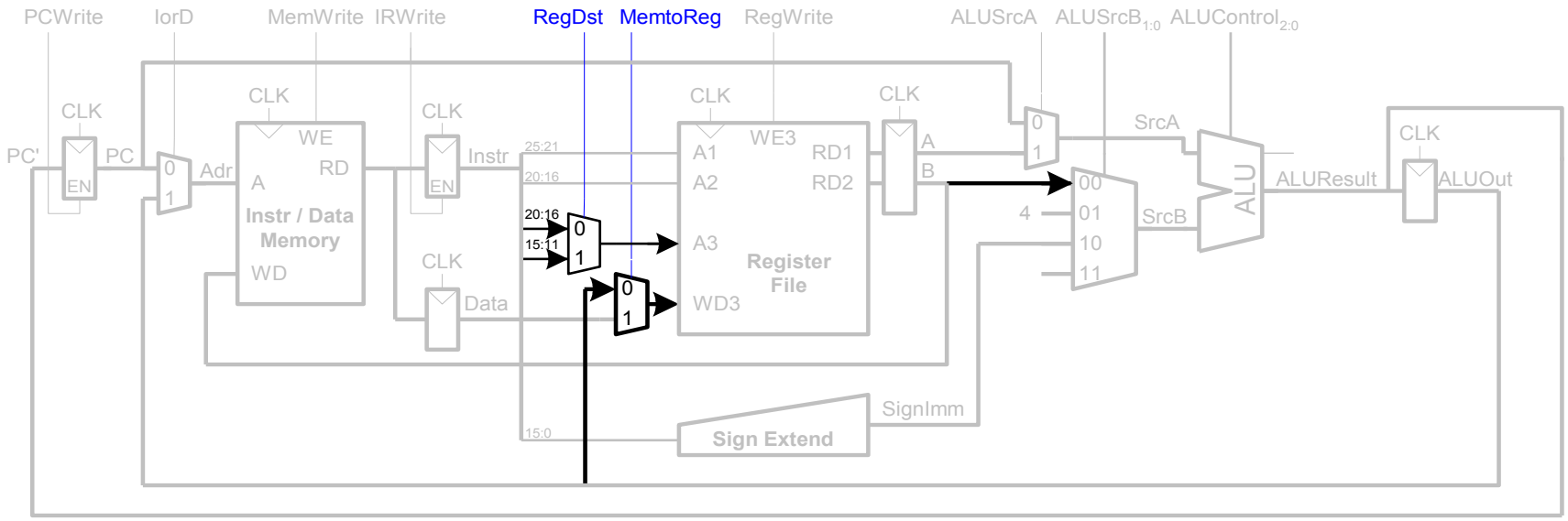
Multicycle Datapath: `sw`

Write data in `rt` to memory



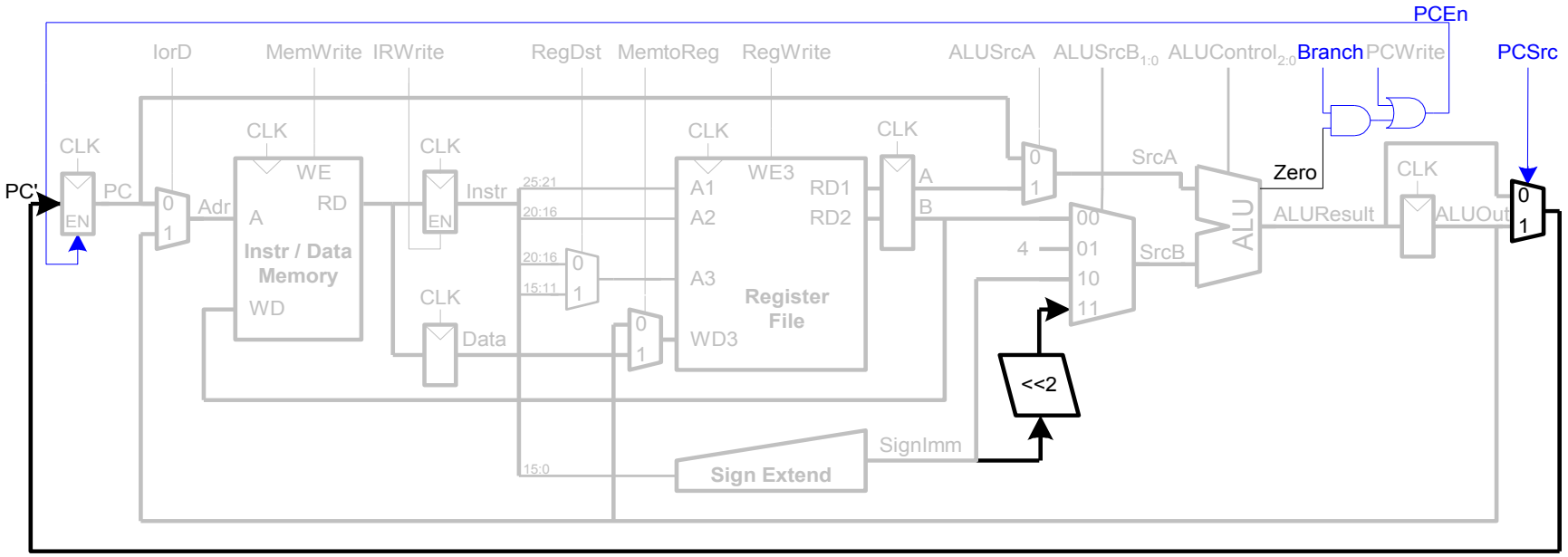
Multicycle Datapath: R-Type

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

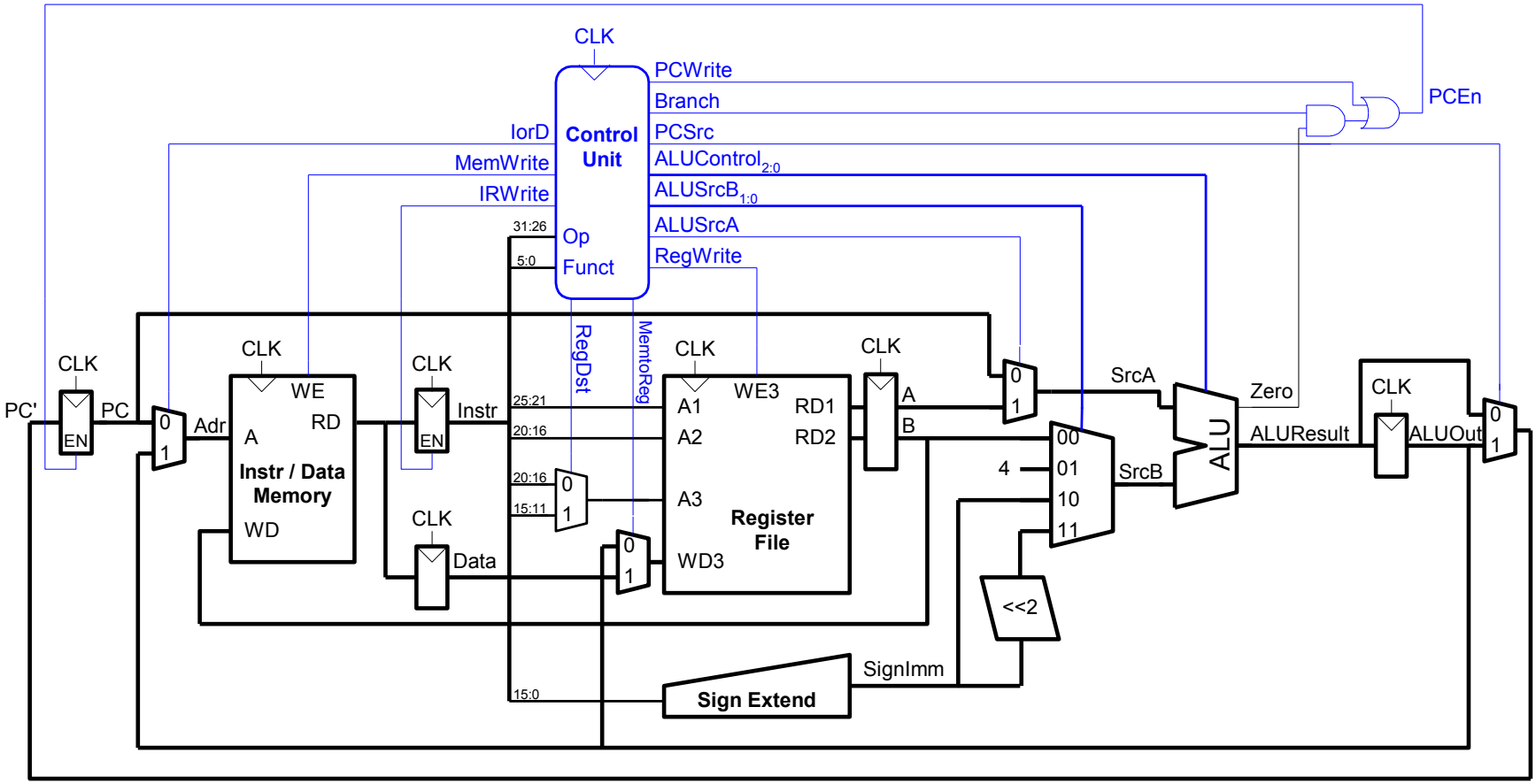


Multicycle Datapath: `beq`

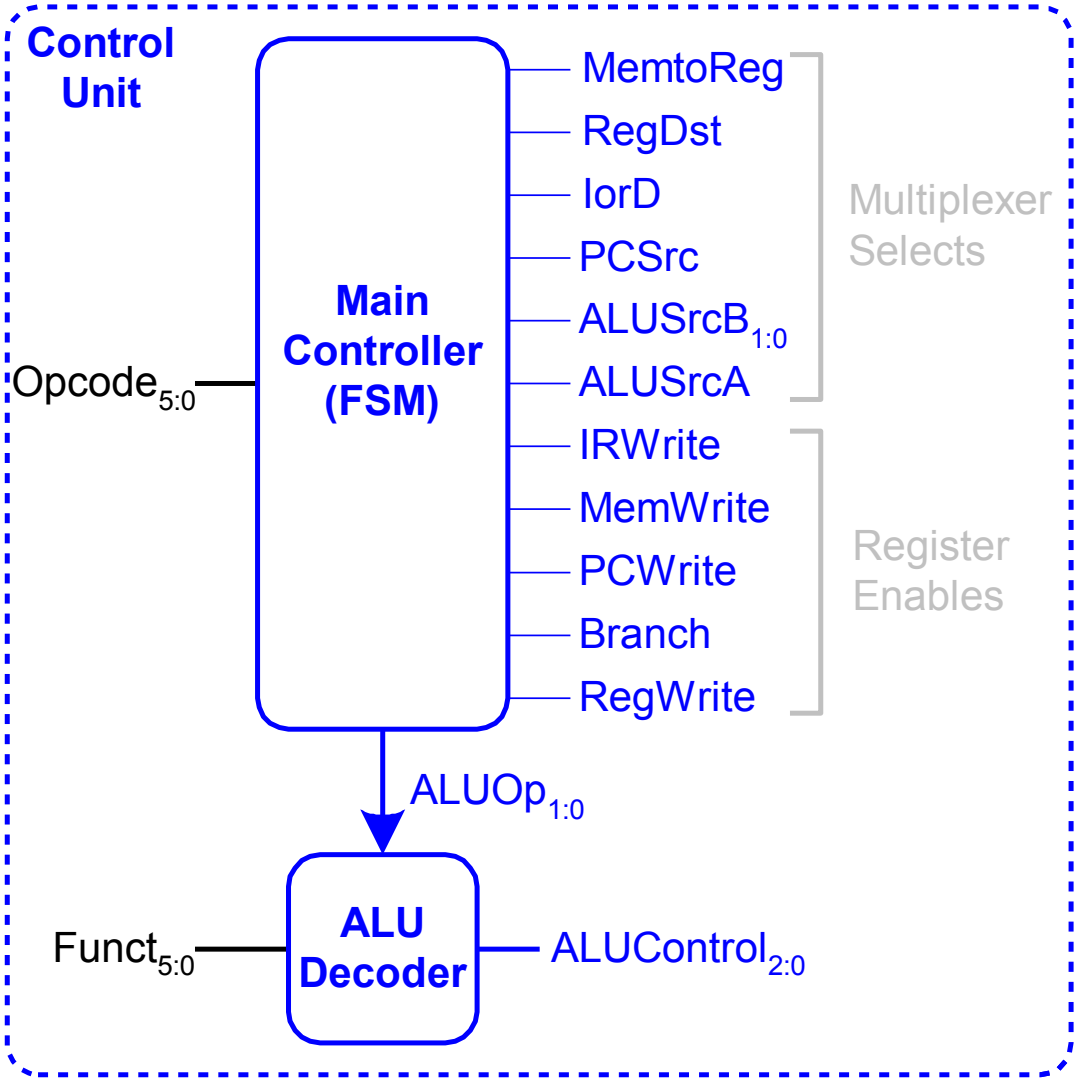
- `rs == rt`?
- BTA = (sign-extended immediate << 2) + (PC+4)



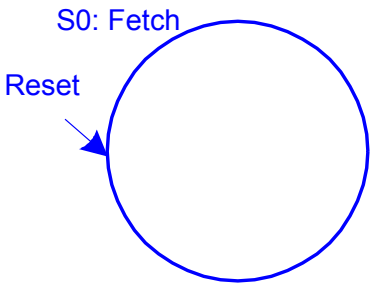
Multicycle Processor

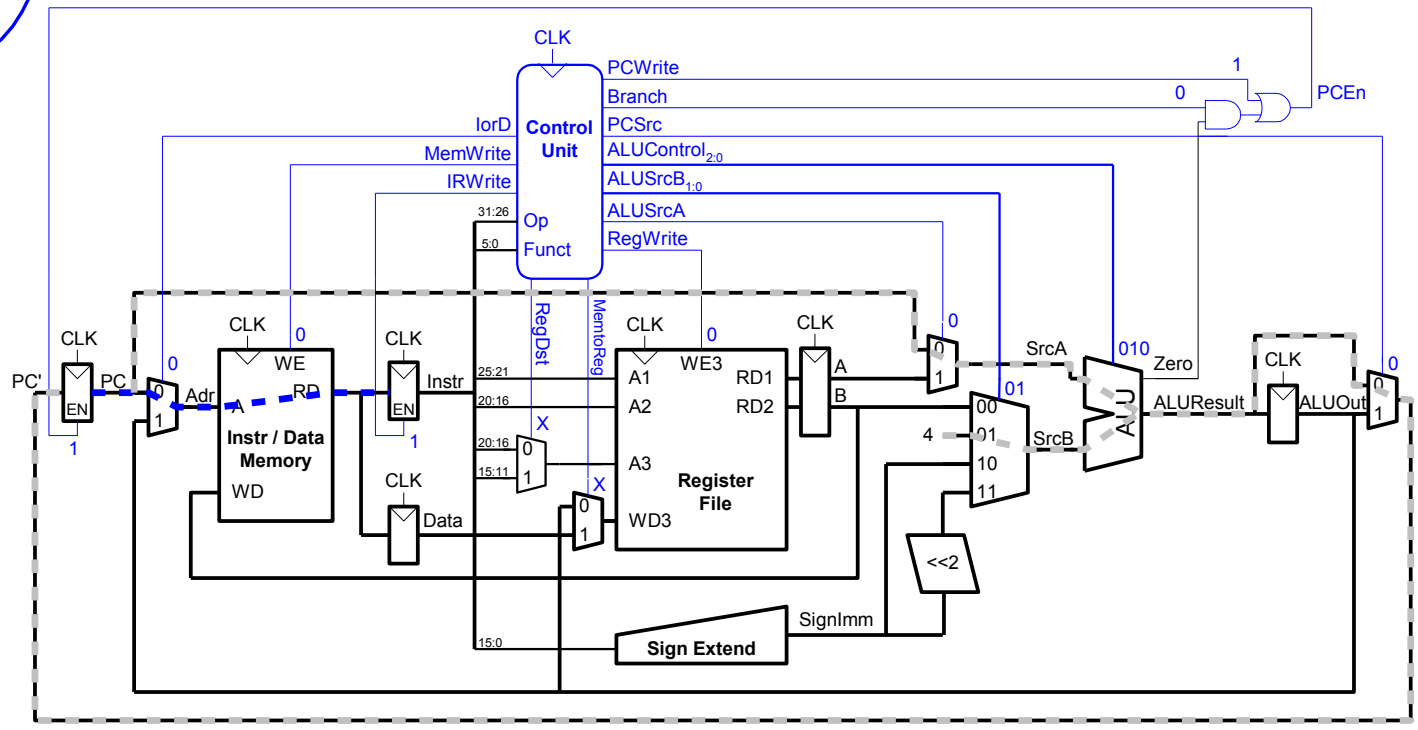


Multicycle Control

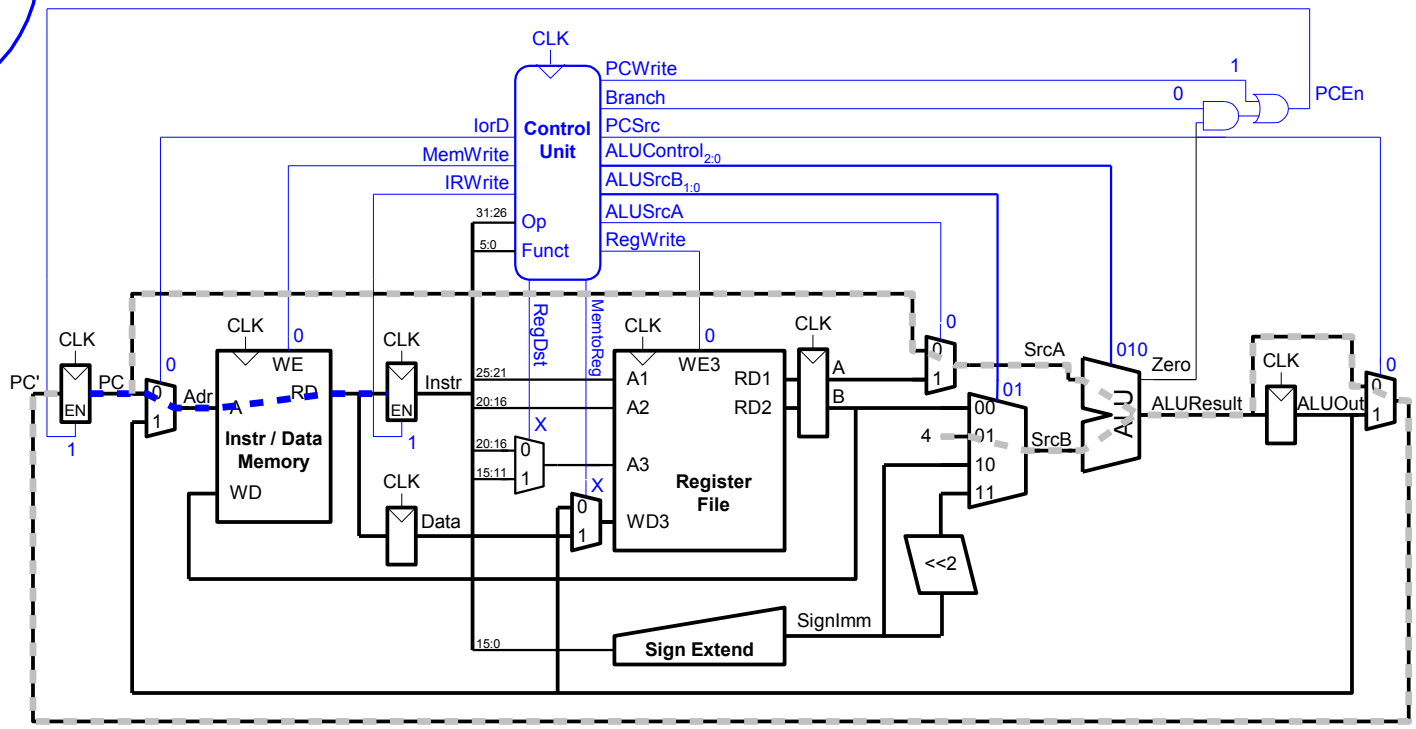


Main Controller FSM: Fetch

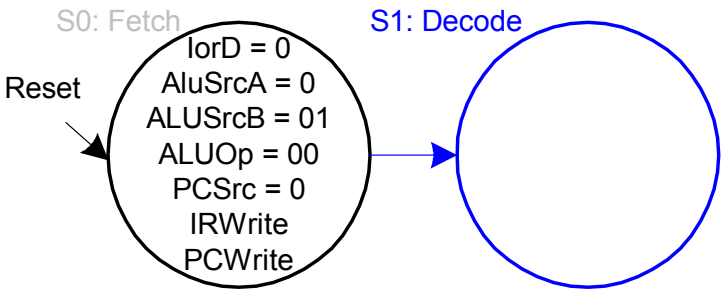


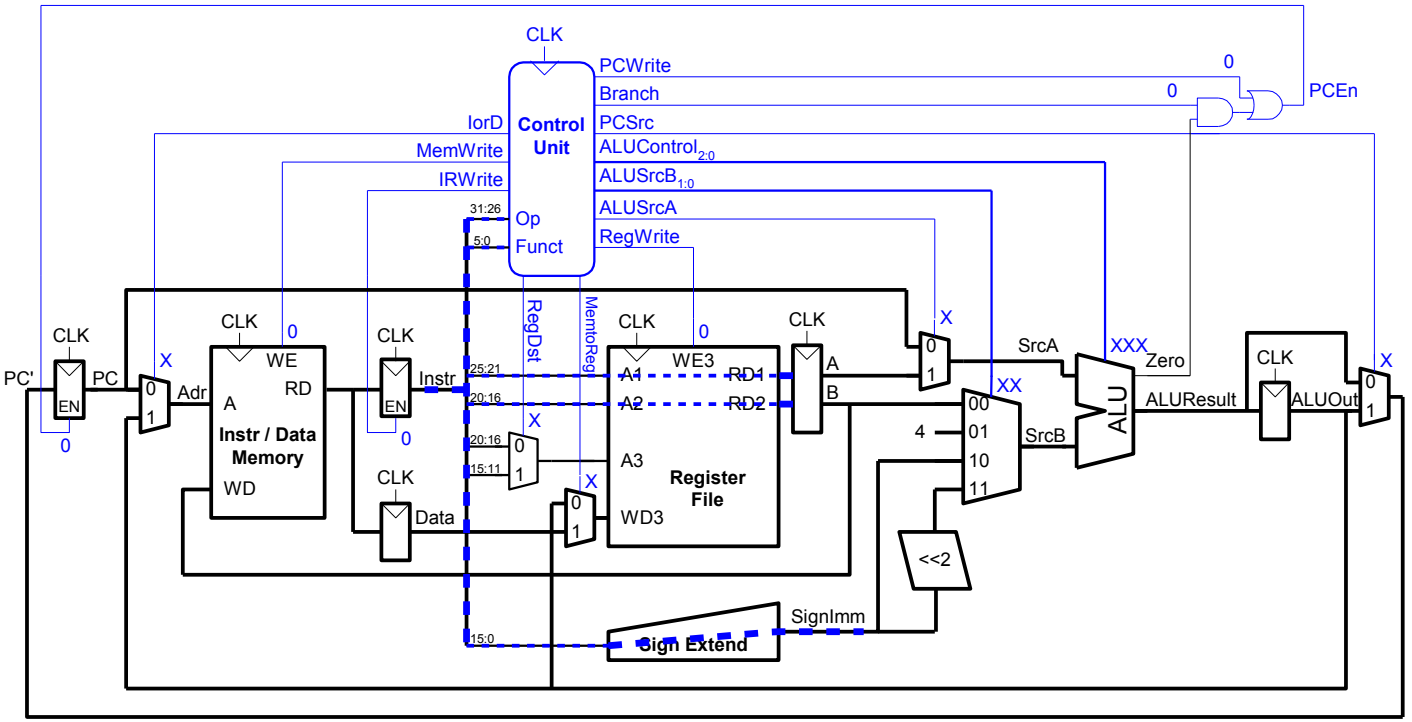


Main Controller FSM: Fetch

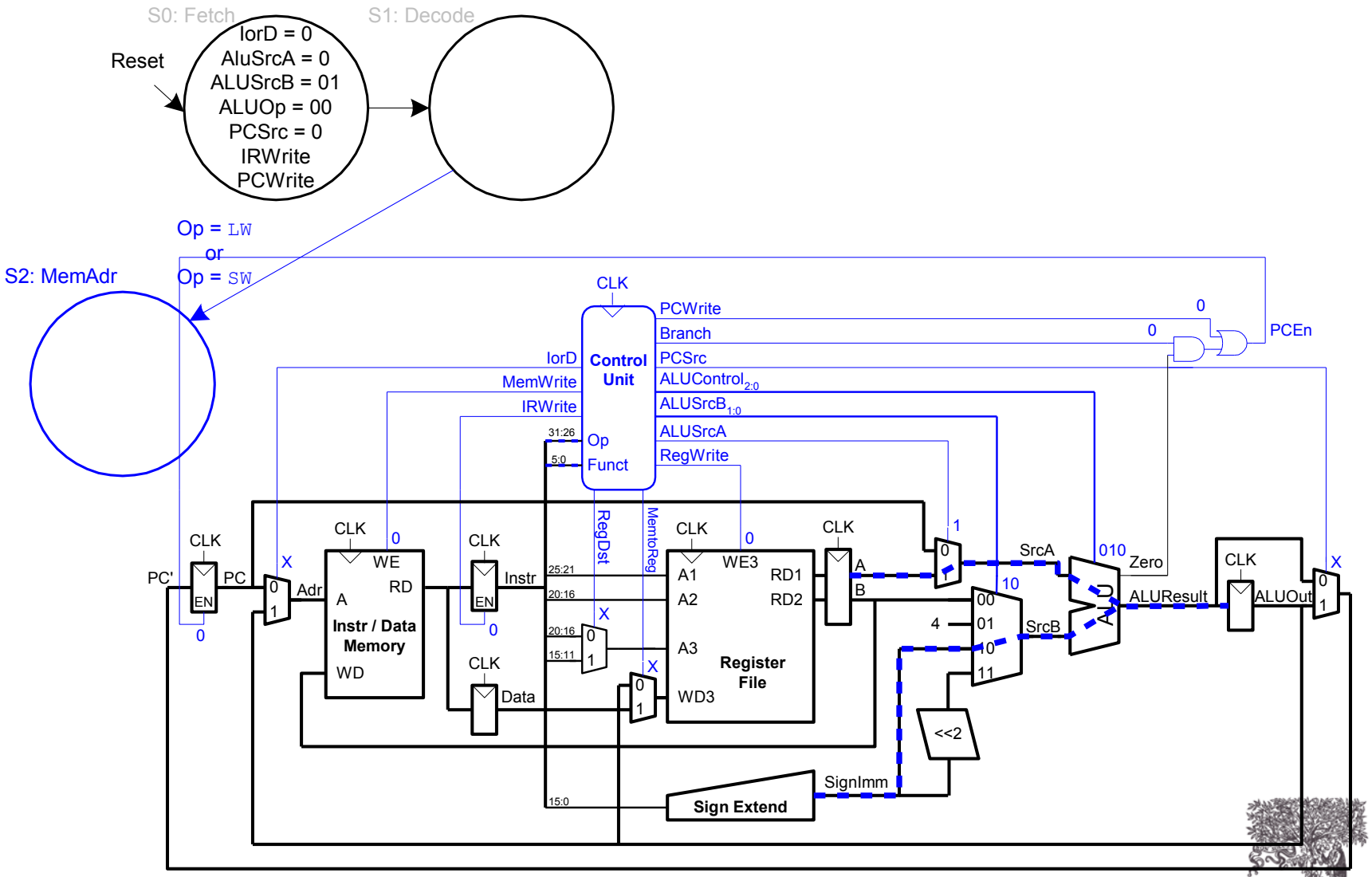


Main Controller FSM: Decode

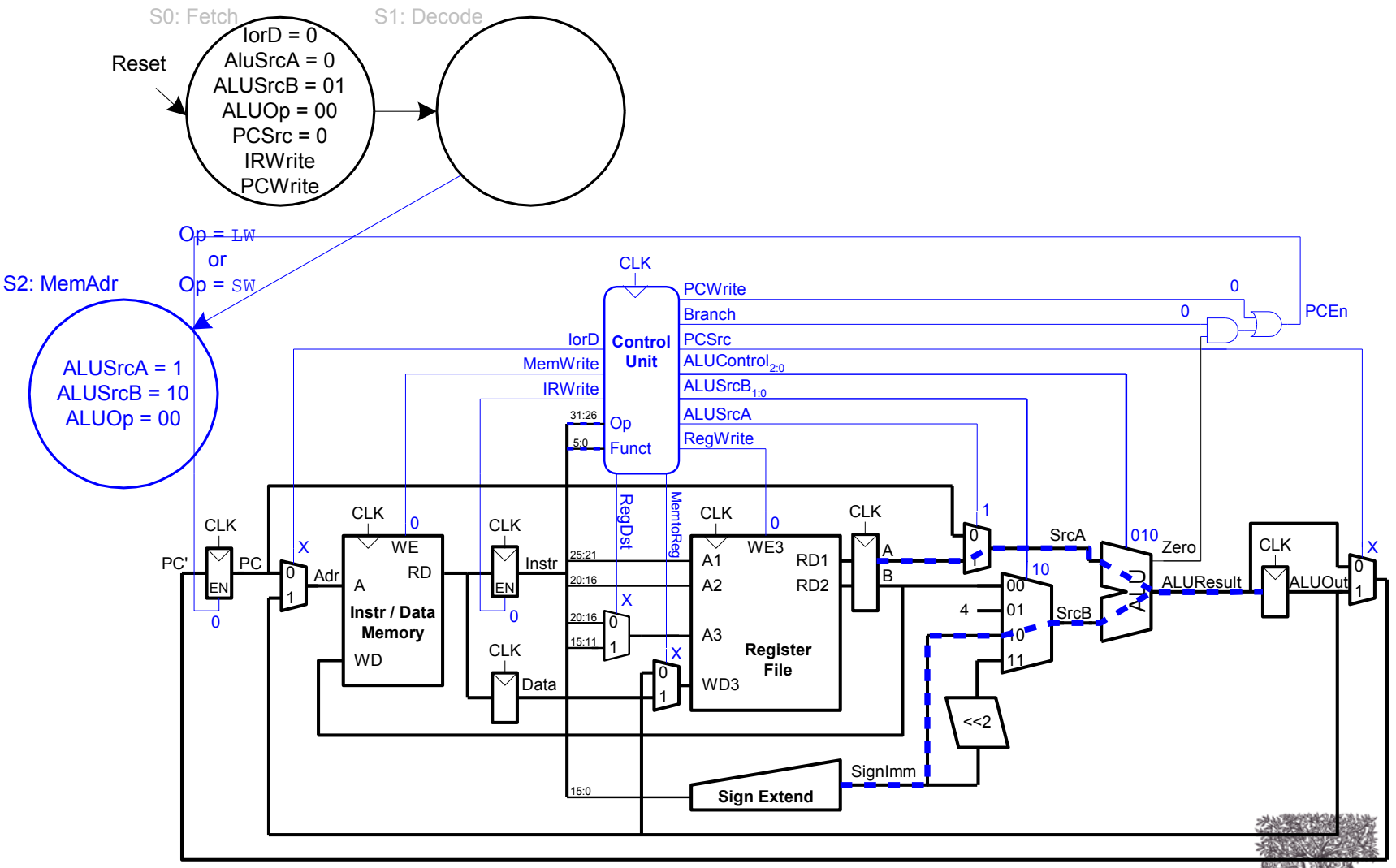




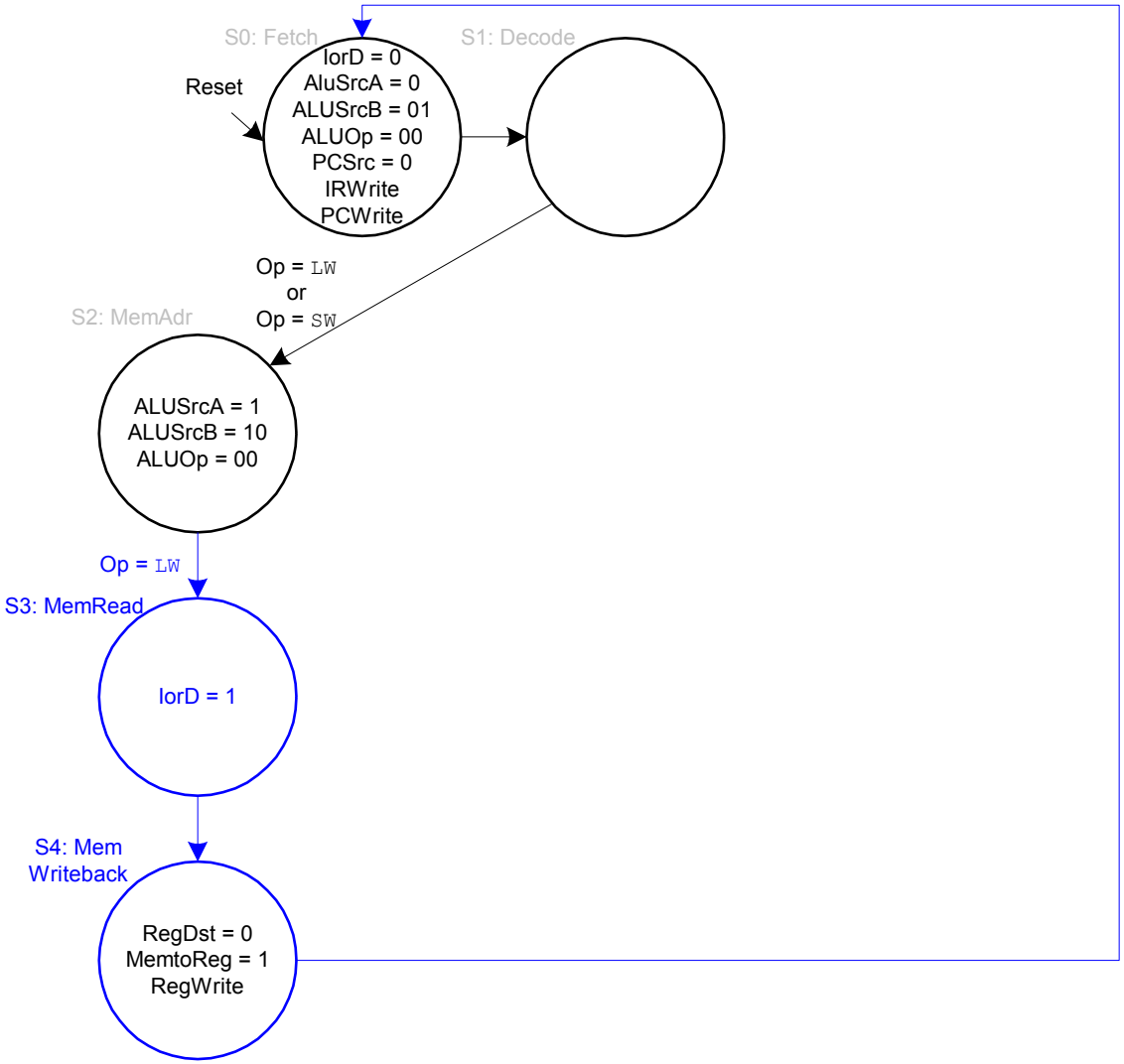
Main Controller FSM: Address



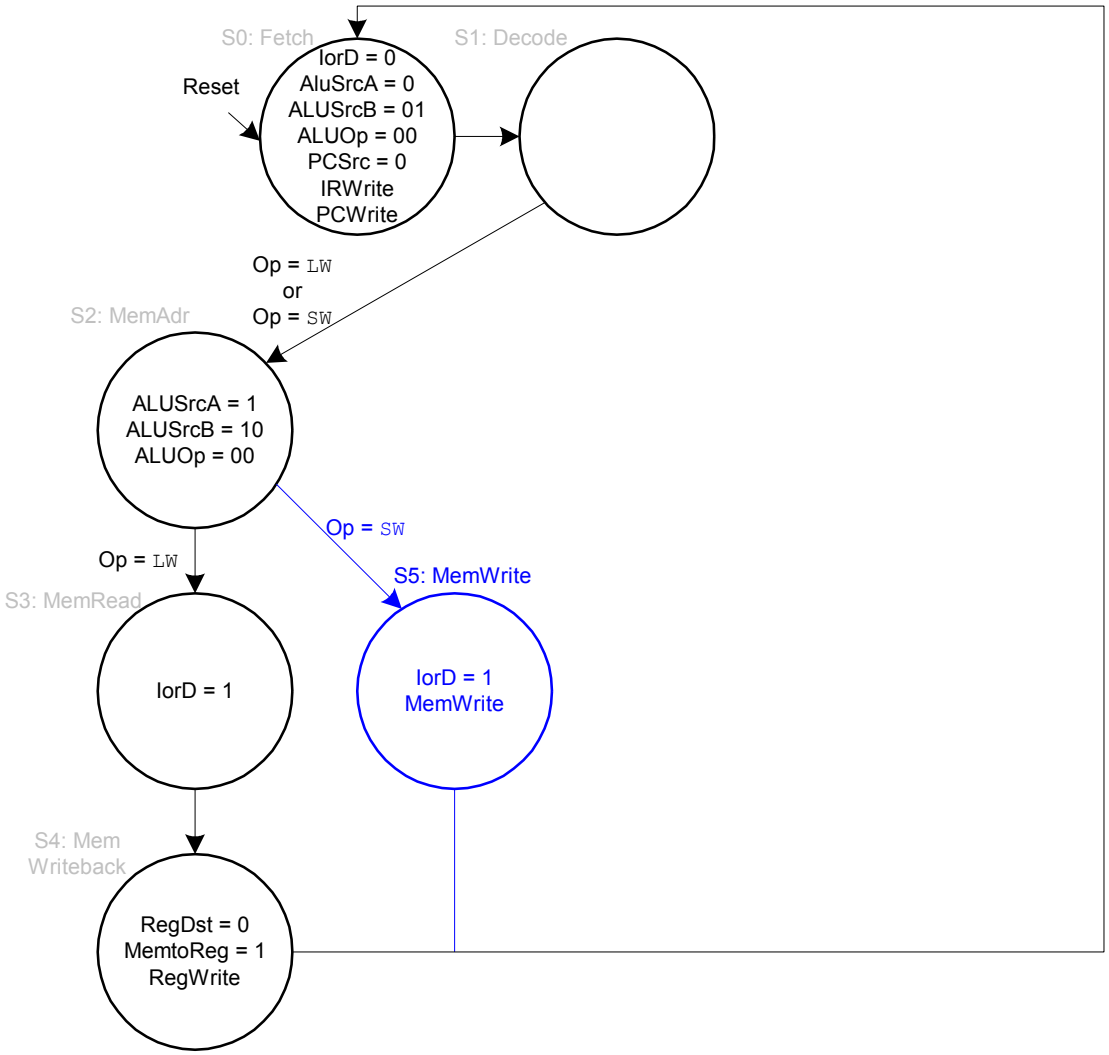
Main Controller FSM: Address



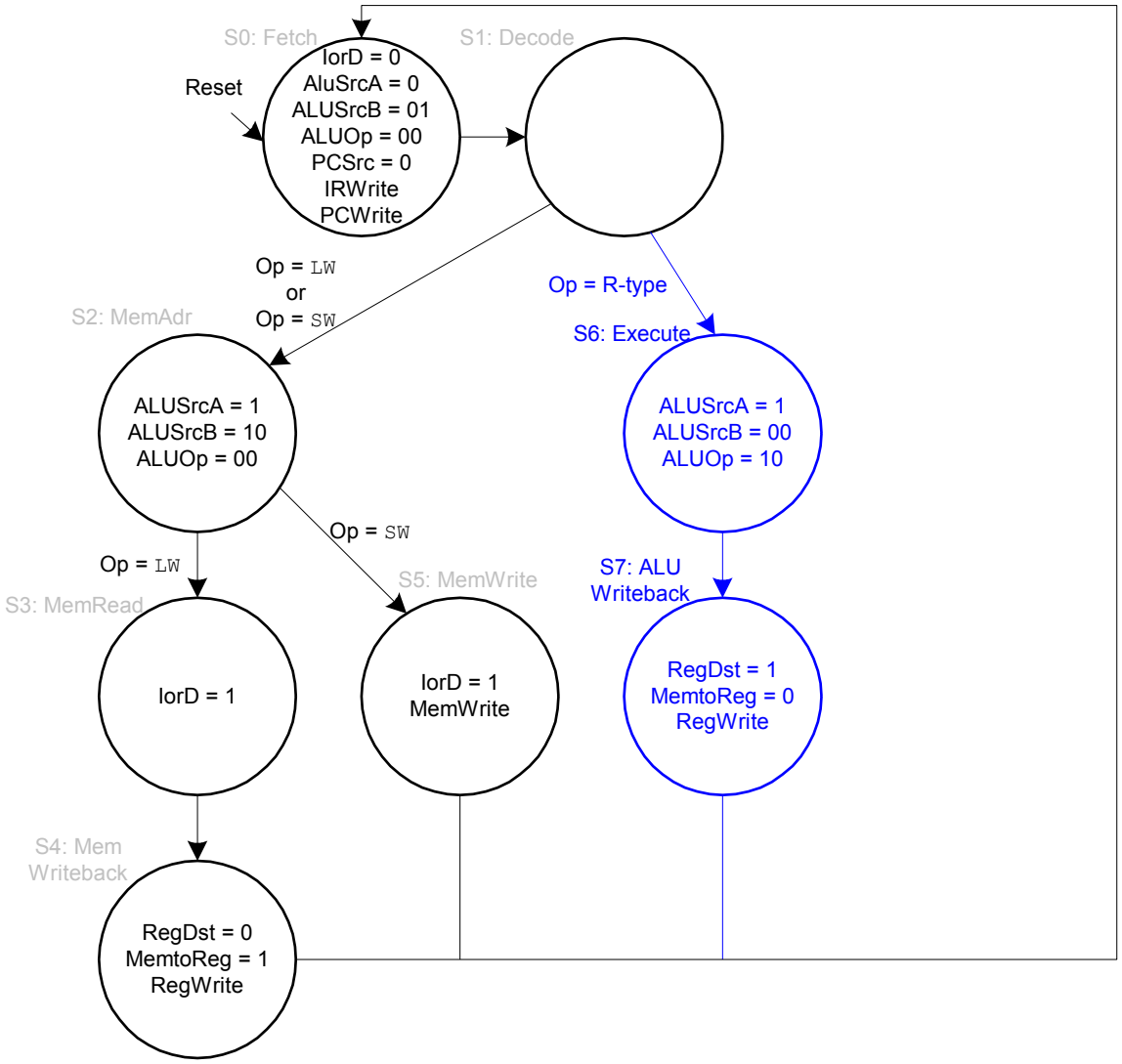
Main Controller FSM: `lw`



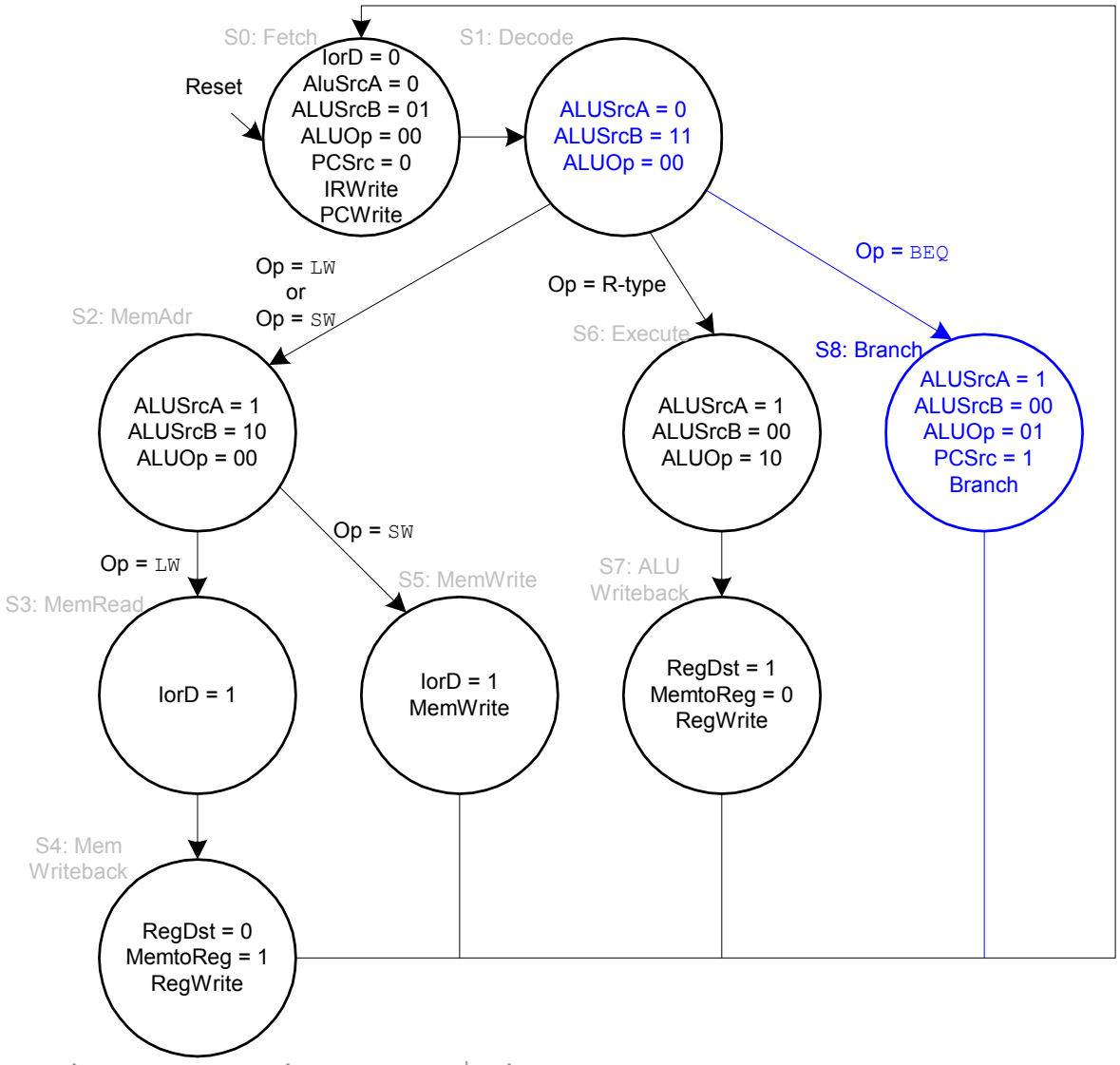
Main Controller FSM: SW



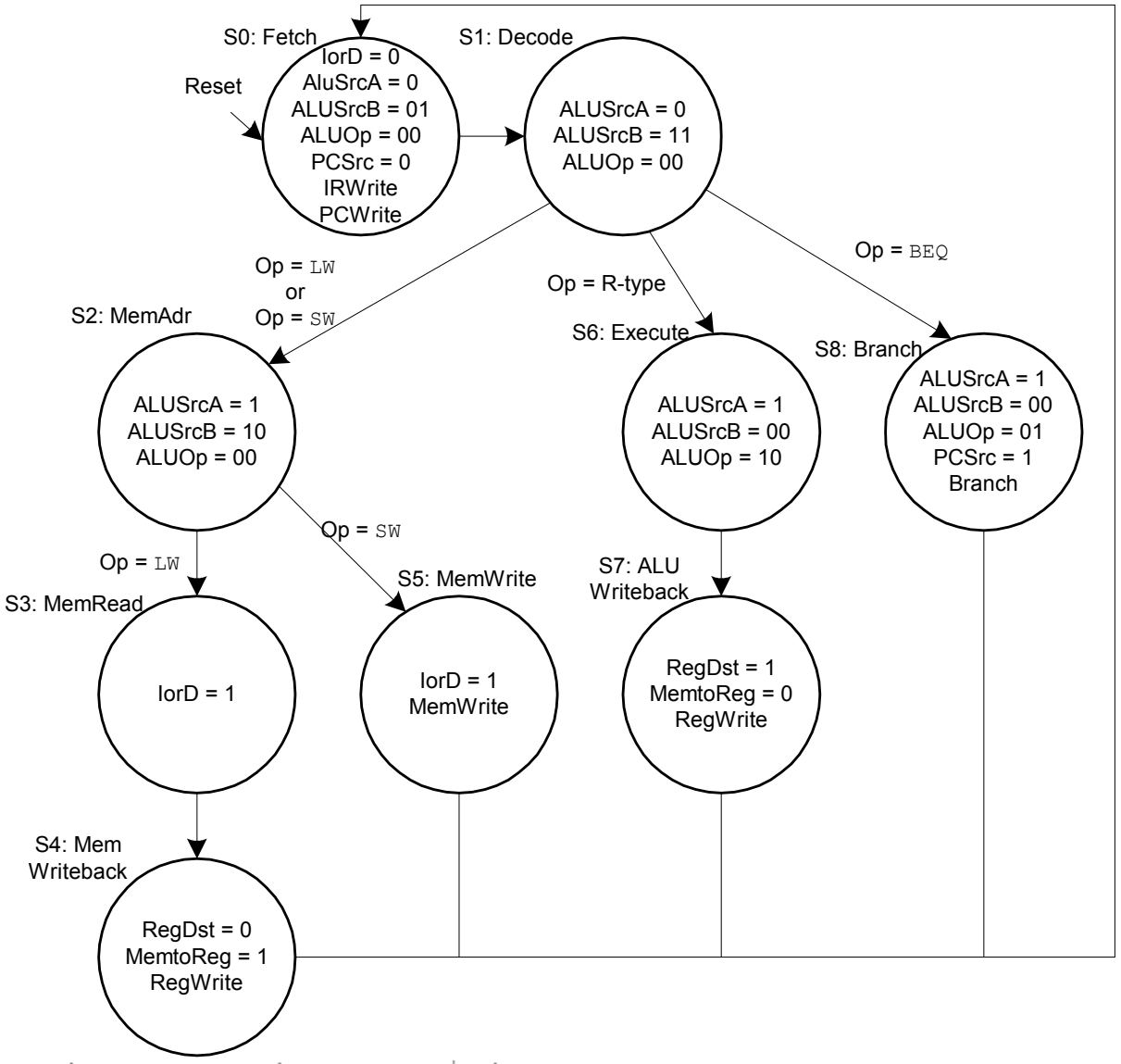
Main Controller FSM: R-Type



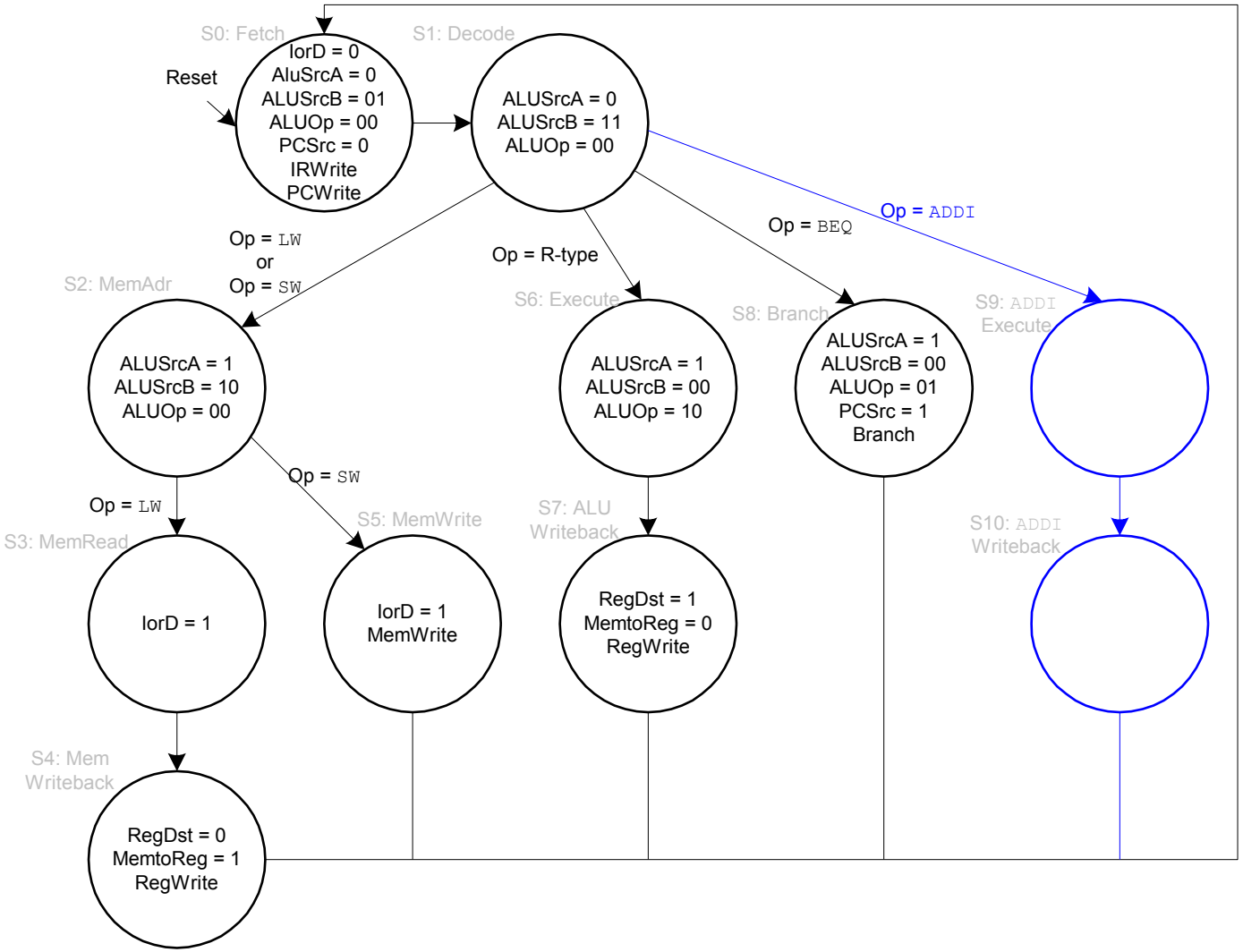
Main Controller FSM: beq



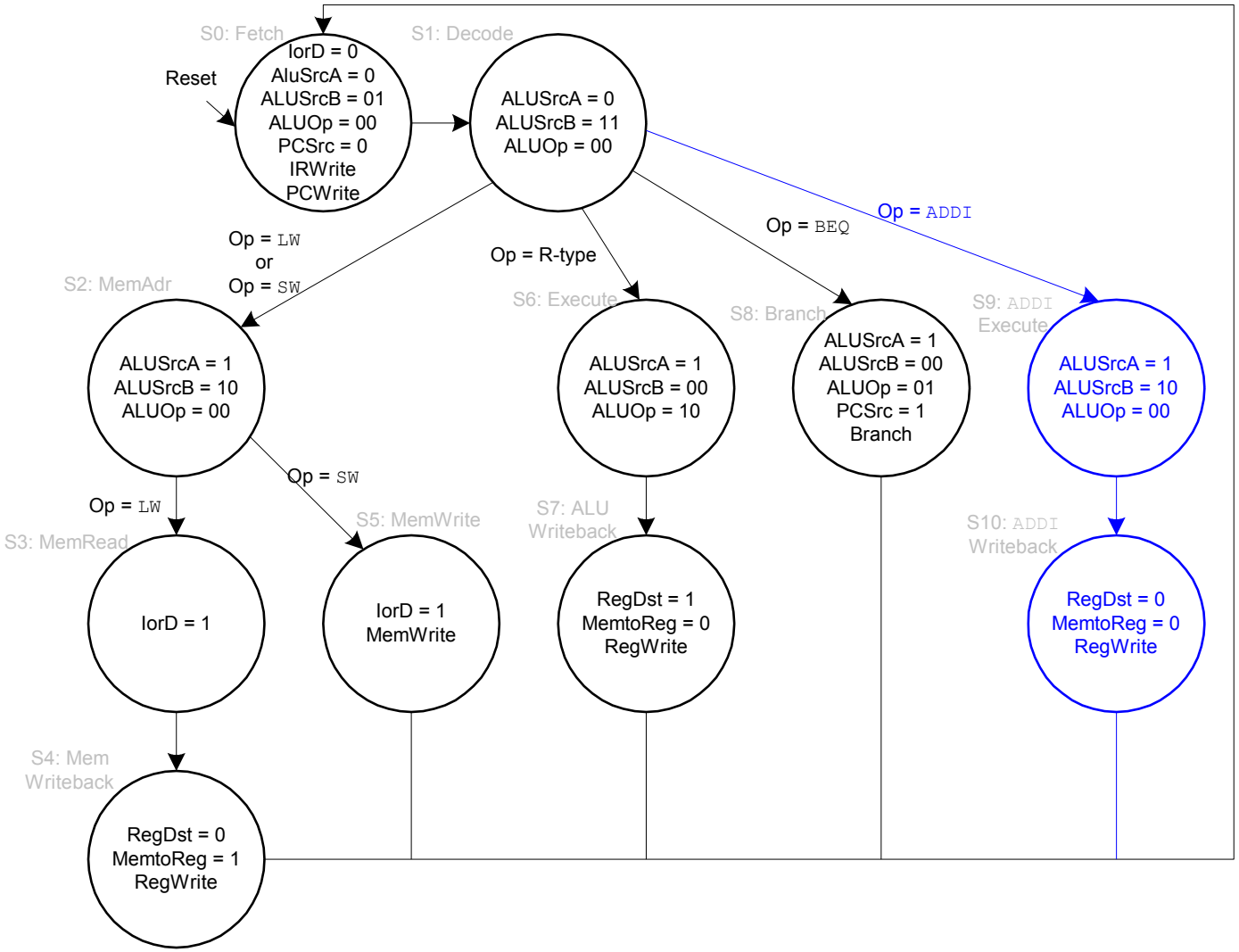
Multicycle Controller FSM



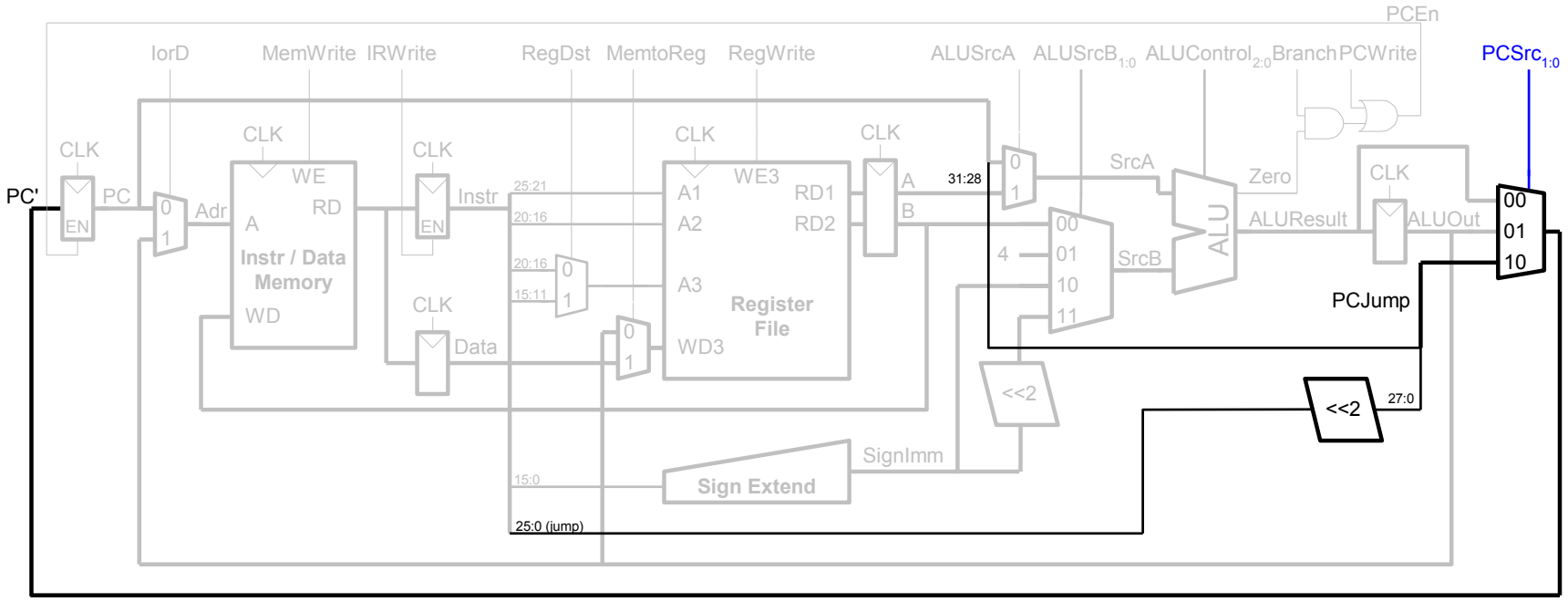
Extended Functionality: addi



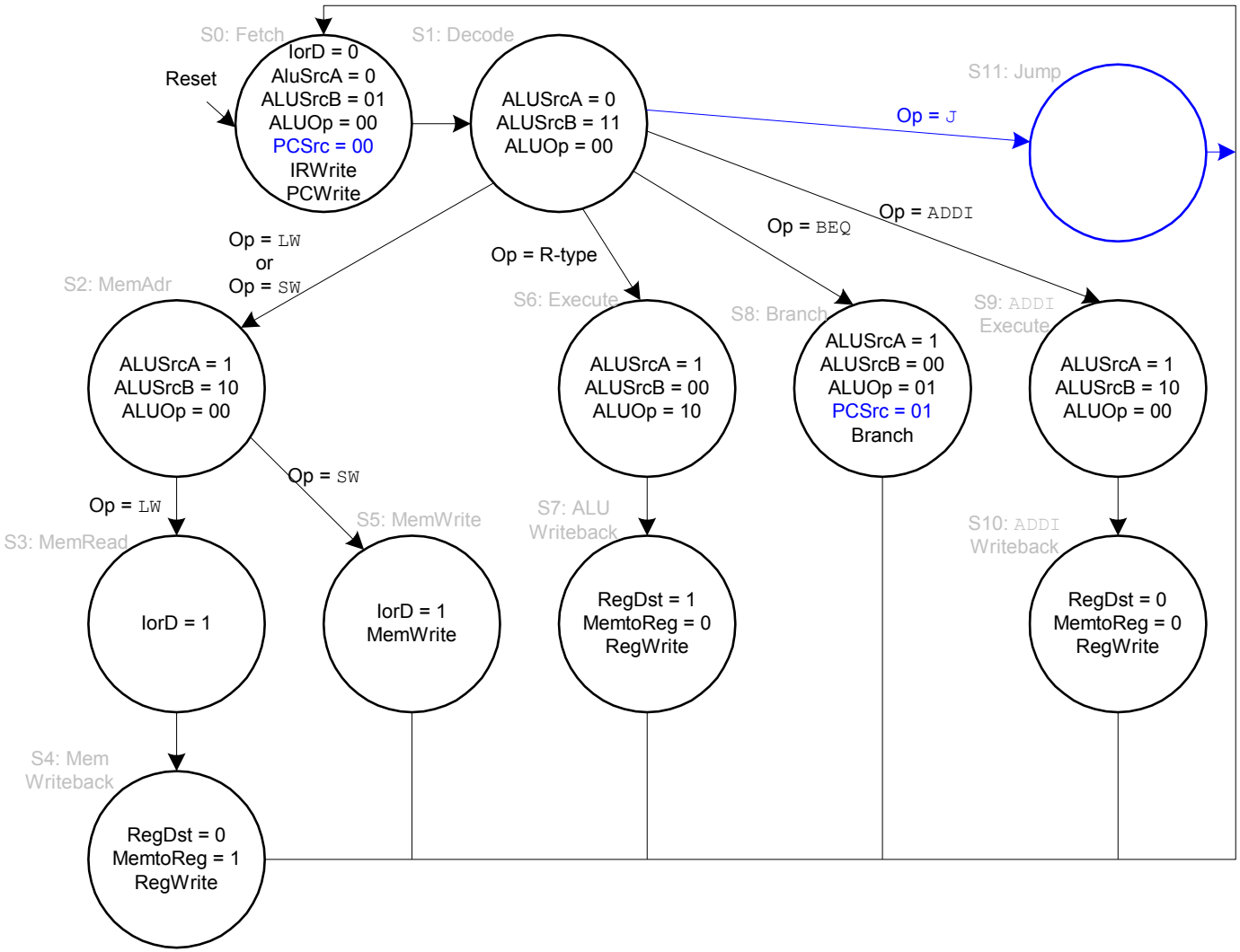
Main Controller FSM: addi



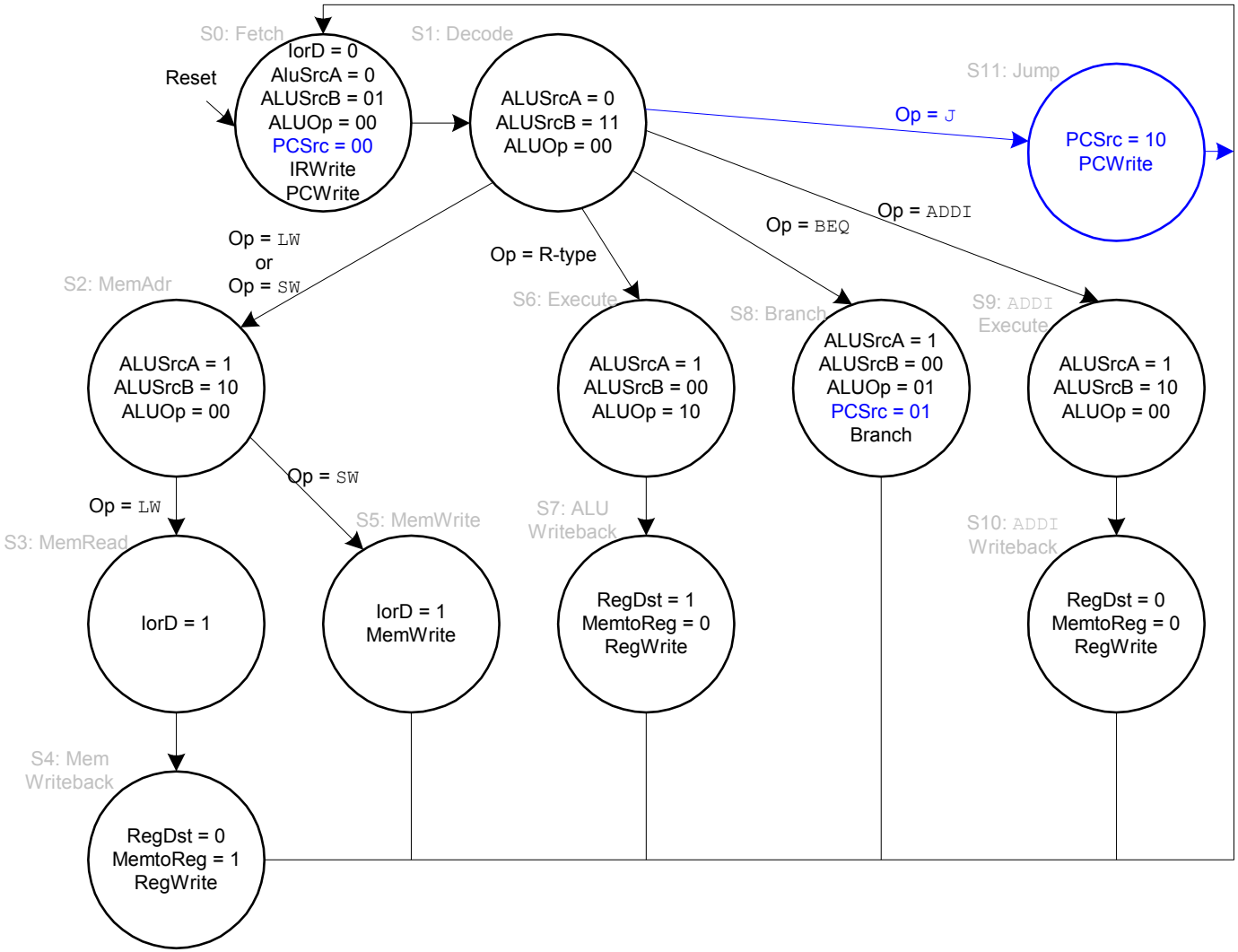
Extended Functionality: j



Main Controller FSM: j



Main Controller FSM: j



Multicycle Processor

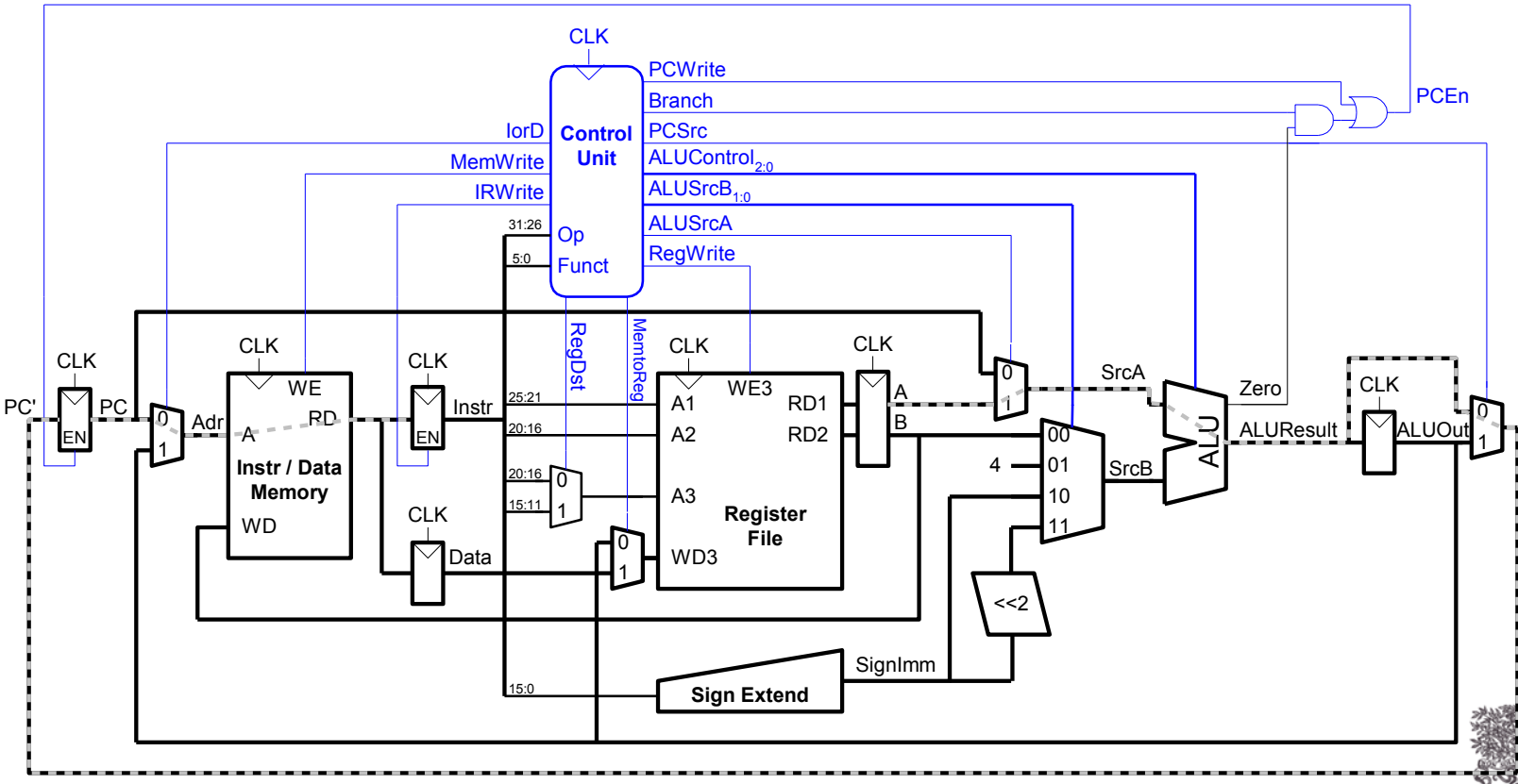
- Instructions take different number of cycles:
  - 3 cycles: `beq`, `j`
  - 4 cycles: R-Type, `sw`, `addi`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

Average CPI = (0.11 + 0.02)(3) + (0.52 + 0.10)(4) + (0.25)(5) = **4.12**

Multicycle Processor

Multicycle critical path:

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



Multicycle 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

$$T_c = ?$$

Multicycle 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

$$
\begin{aligned}
T_c &= t_{pcq\_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \\
&= t_{pcq\_PC} + t_{mux} + t_{mem} + t_{setup} \\
&= [30 + 25 + 250 + 20]\ \text{ps} \\
&= \mathbf{325\ ps}
\end{aligned}
$$

Multicycle Performance Example

Program with 100 billion instructions

Execution Time = (# instructions) × CPI × T_c

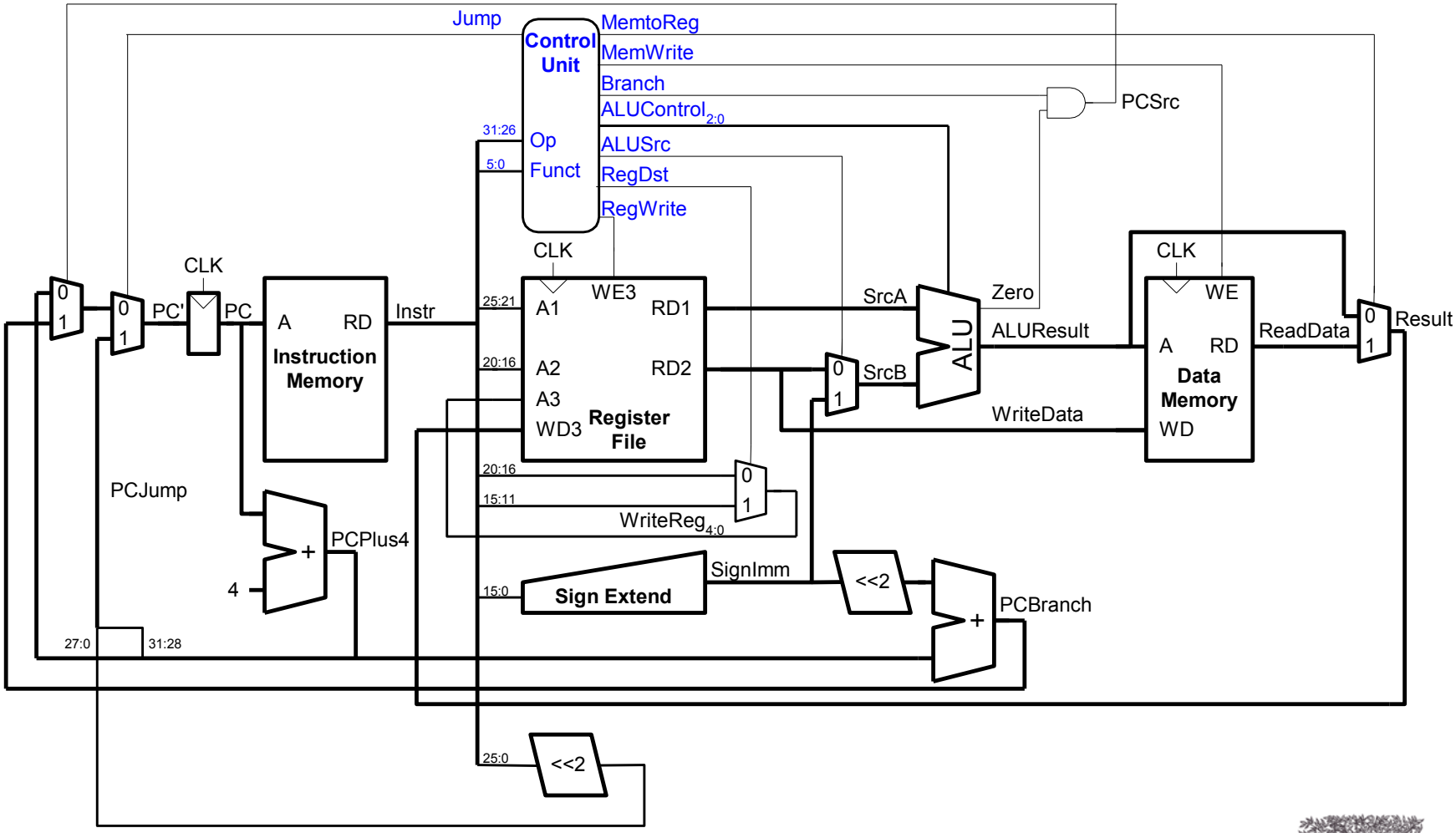
$= (100 \times 10^9)(4.12)(325 \times 10^{-12})$

$=$ **133.9 seconds**

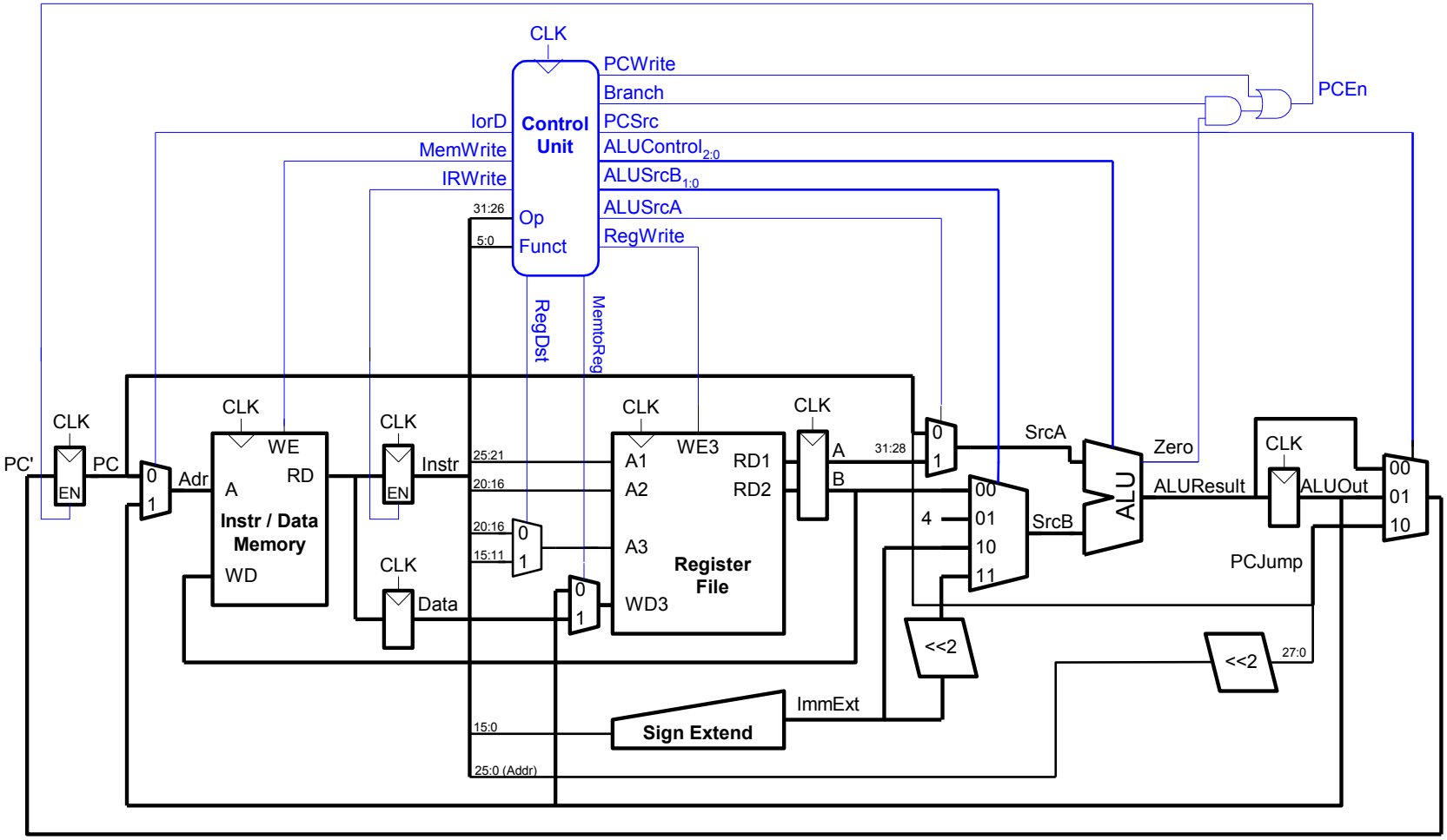
This is **slower** than the single-cycle processor (92.5 seconds). Why?

- Not all steps same length
- Sequencing overhead for each step ($t_{pcq} + t_{setup} = 50$ ps)

Review: Single-Cycle Processor



Review: Multicycle Processor

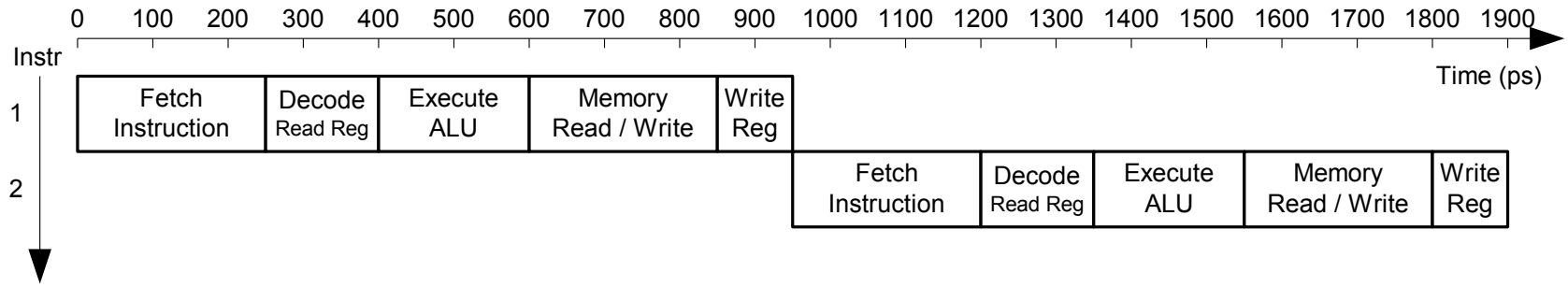


Pipelined MIPS Processor

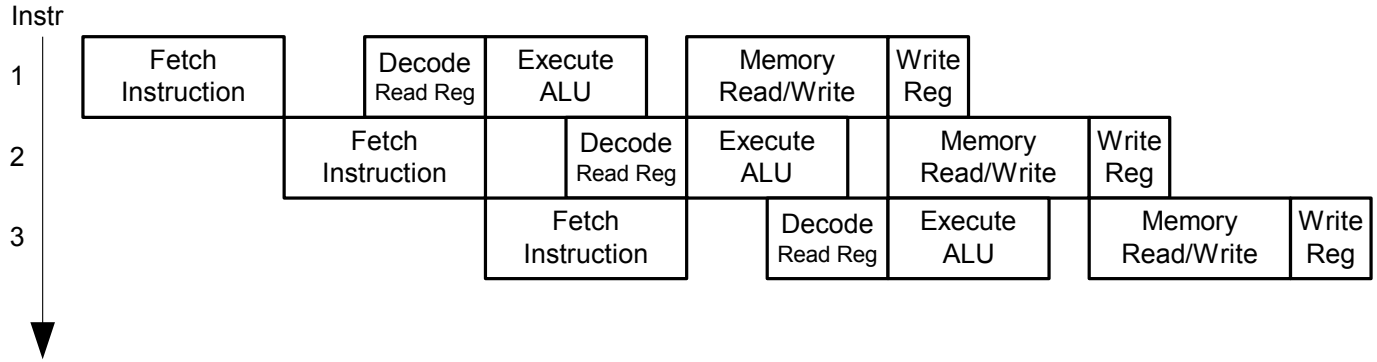
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Single-Cycle vs. Pipelined

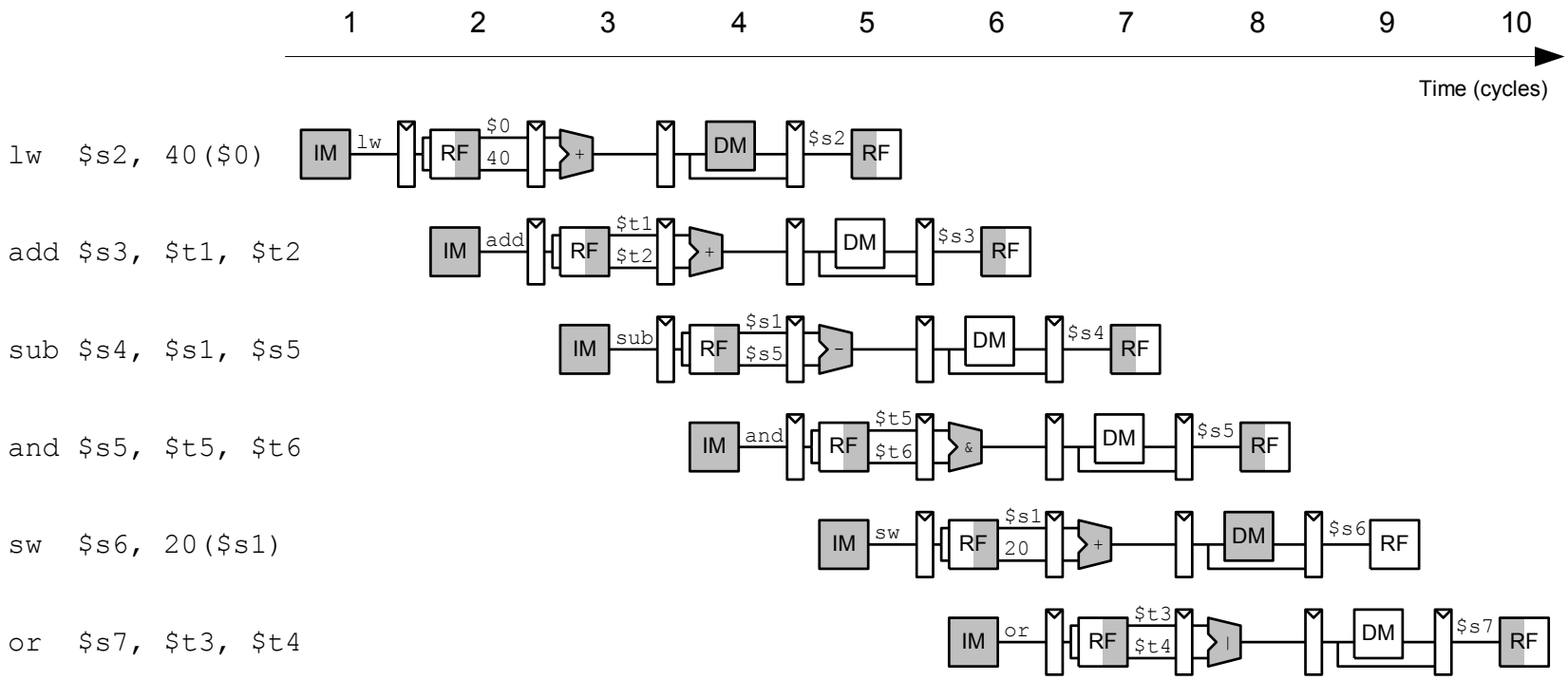
Single-Cycle



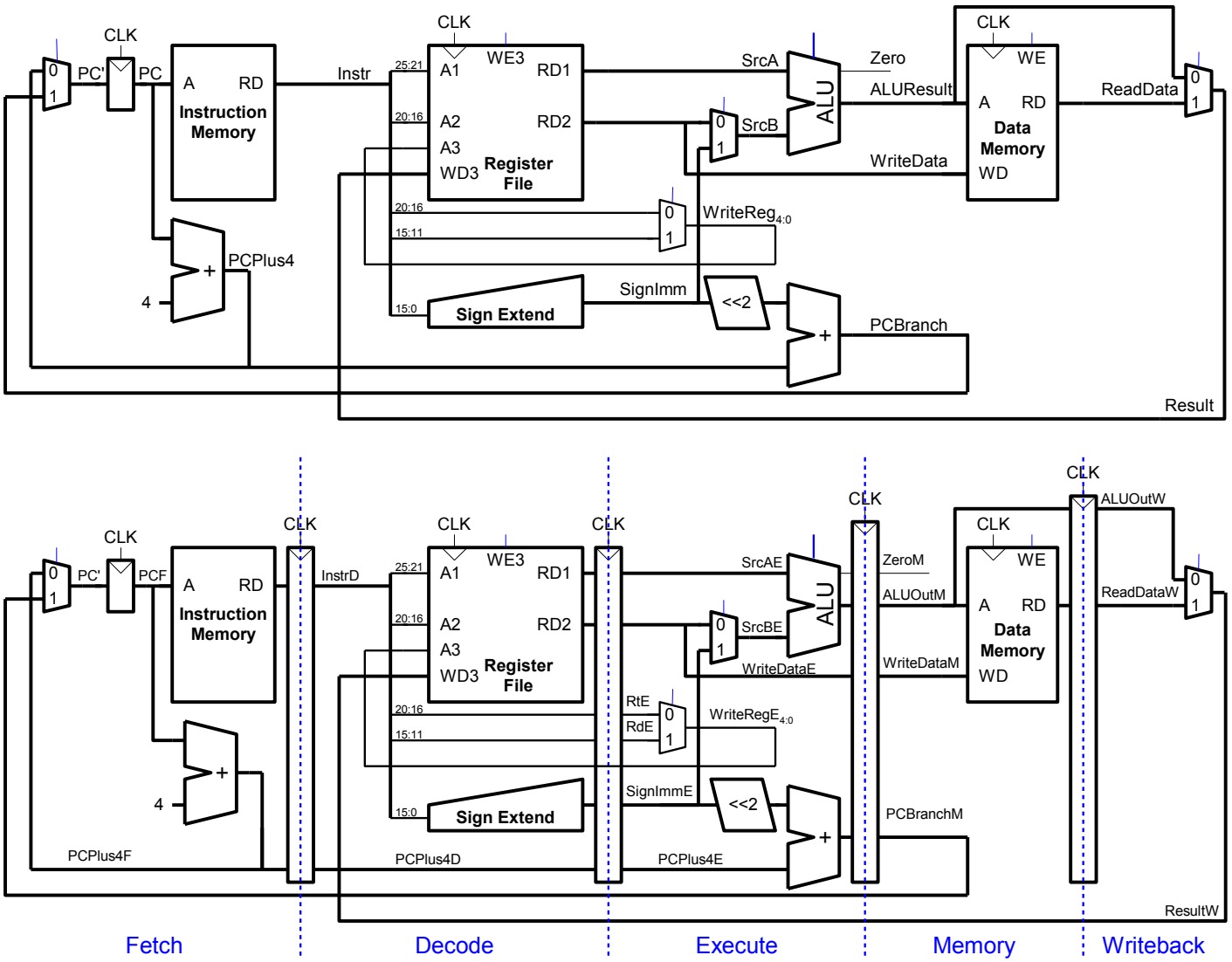
Pipelined



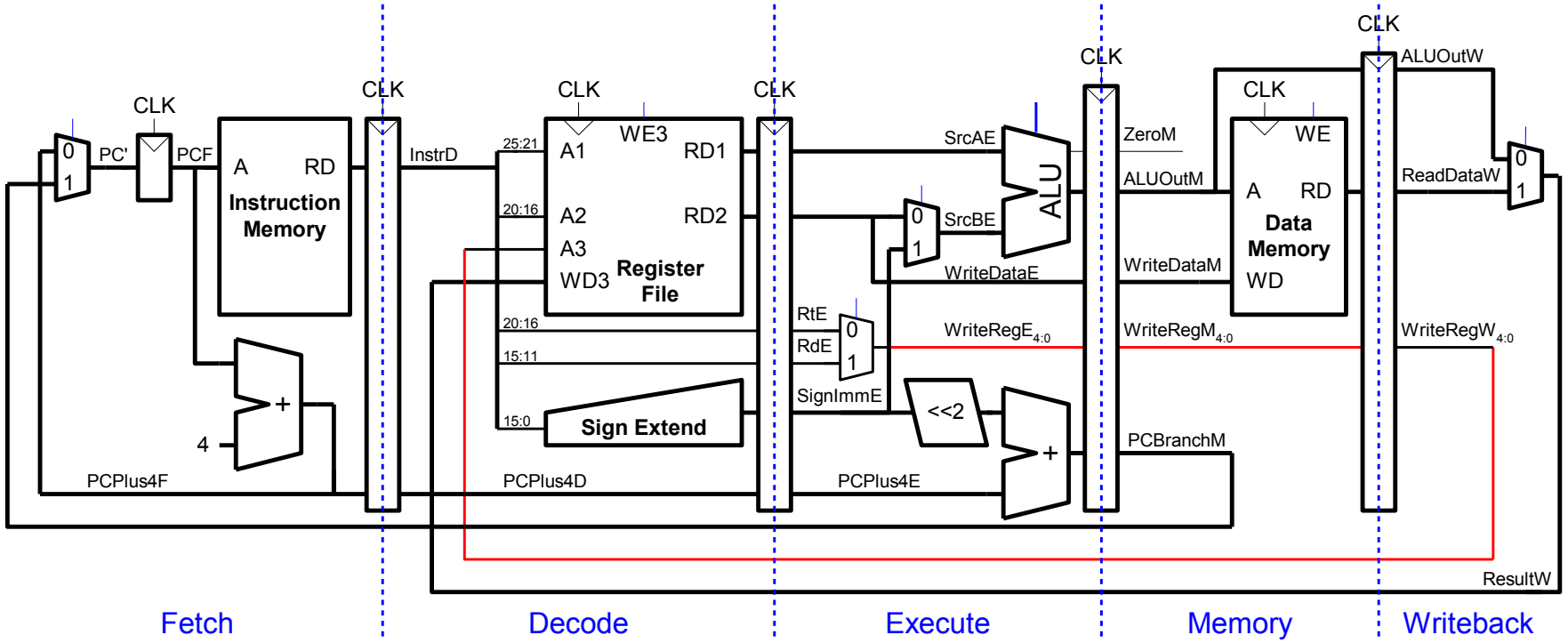
Pipelined Processor Abstraction



Single-Cycle & Pipelined Datapath

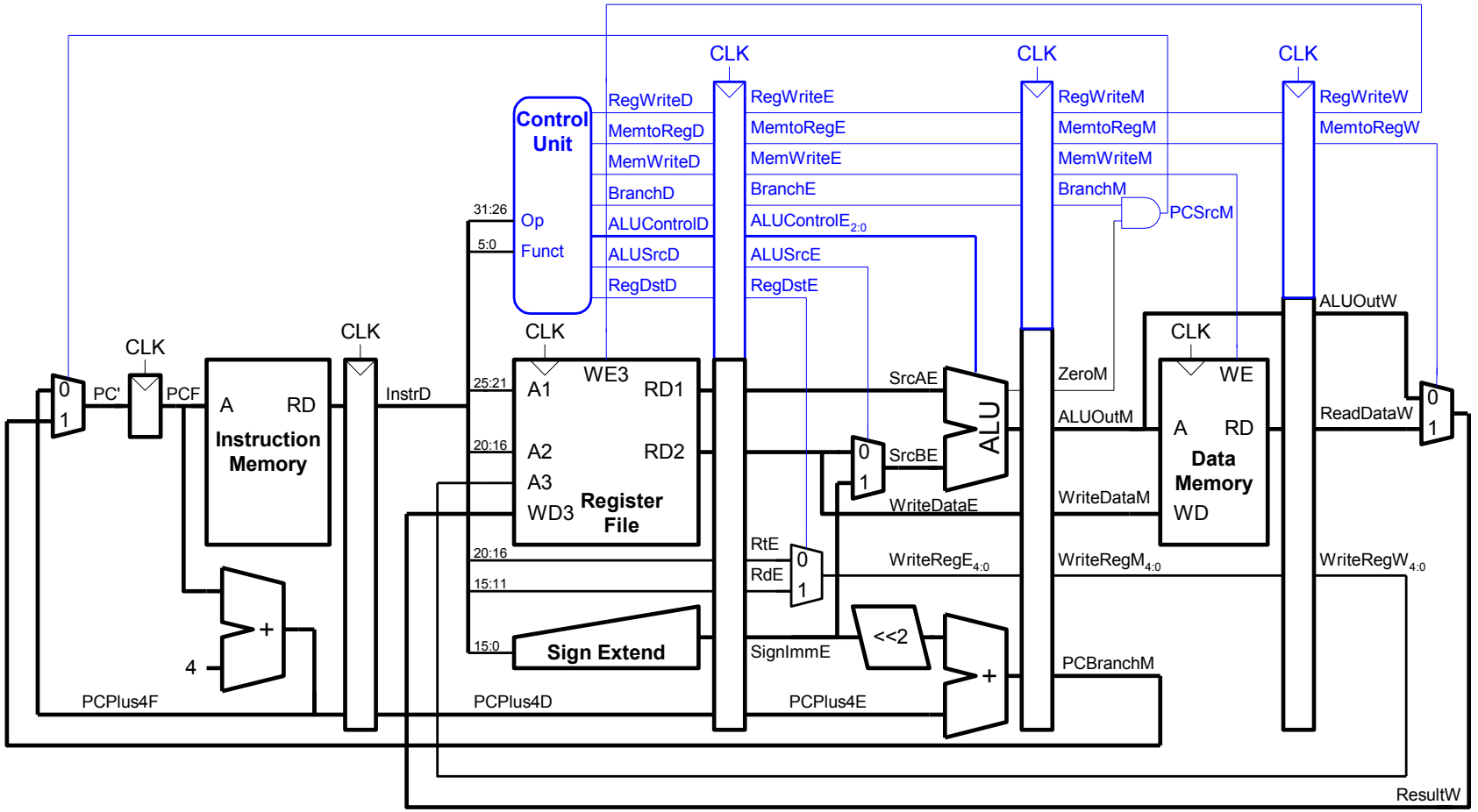


Corrected Pipelined Datapath



WriteReg* must arrive at same time as *Result

Pipelined Processor Control

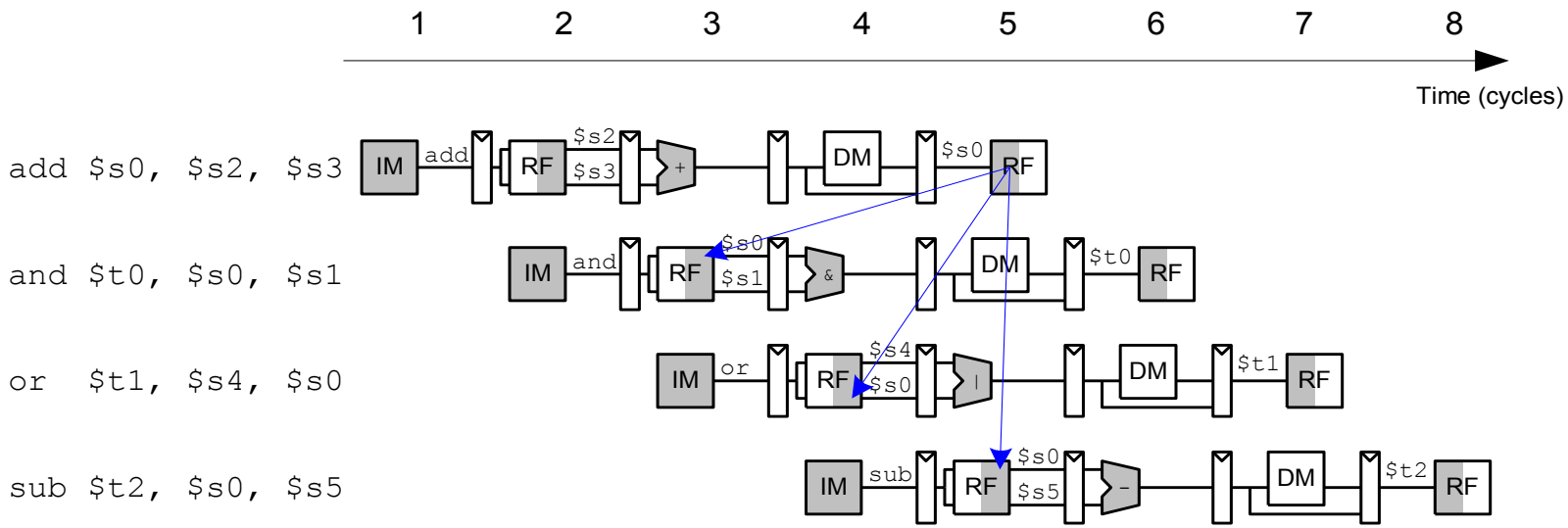


- **Same control unit as single-cycle processor**
- **Control delayed to proper pipeline stage**

Pipeline 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 branches)

Data Hazard

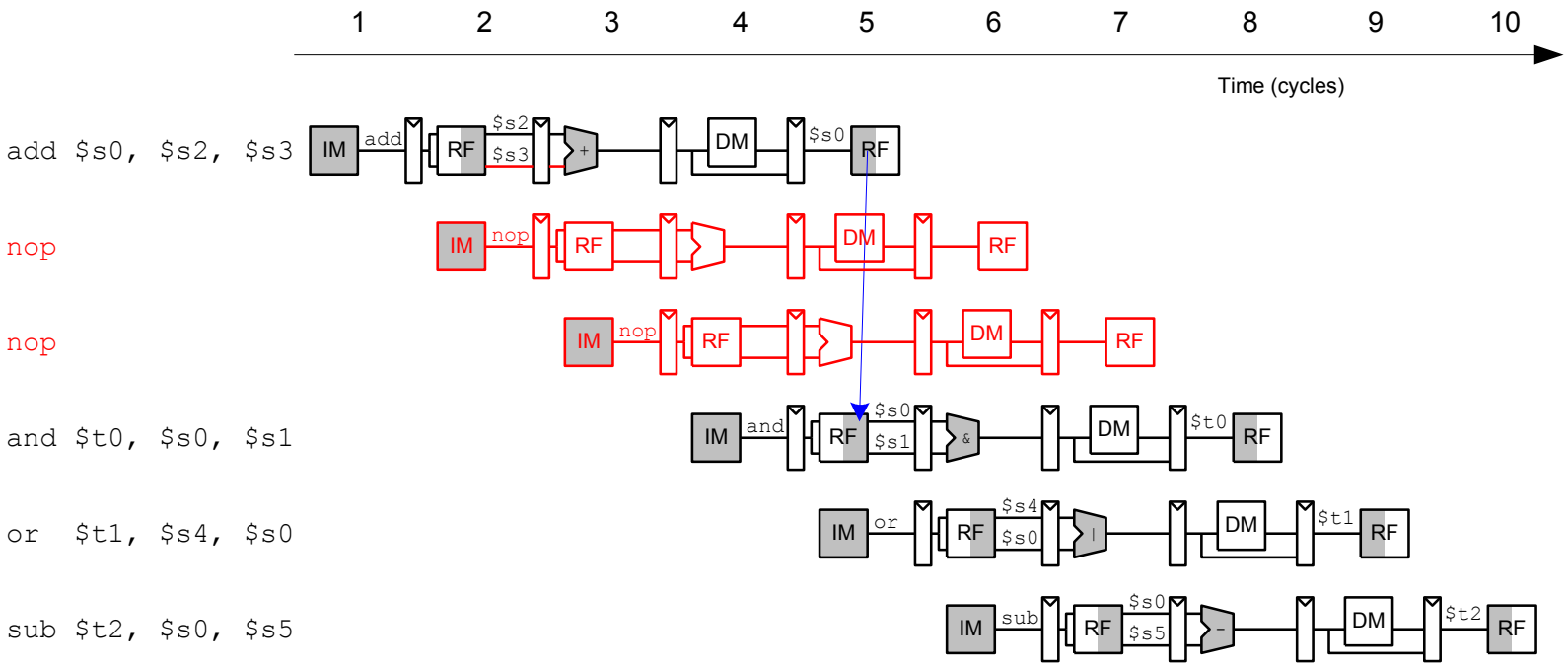


Handling Data Hazards

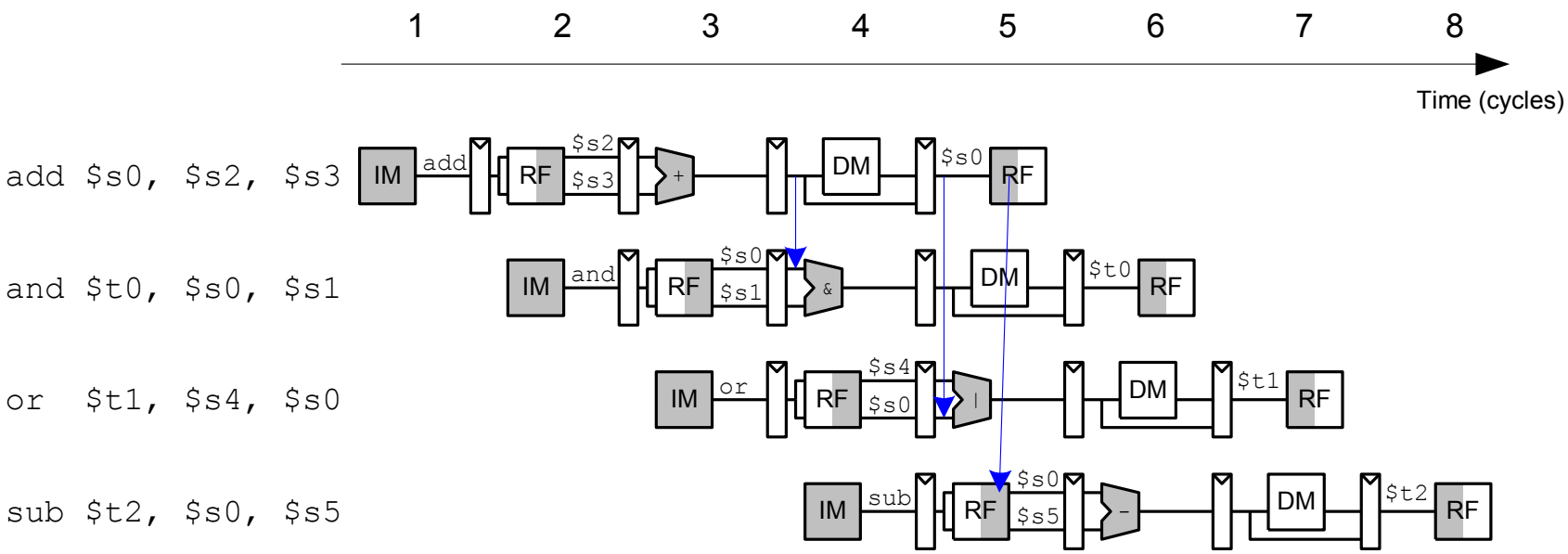
- Insert `nop`s in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Compile-Time Hazard Elimination

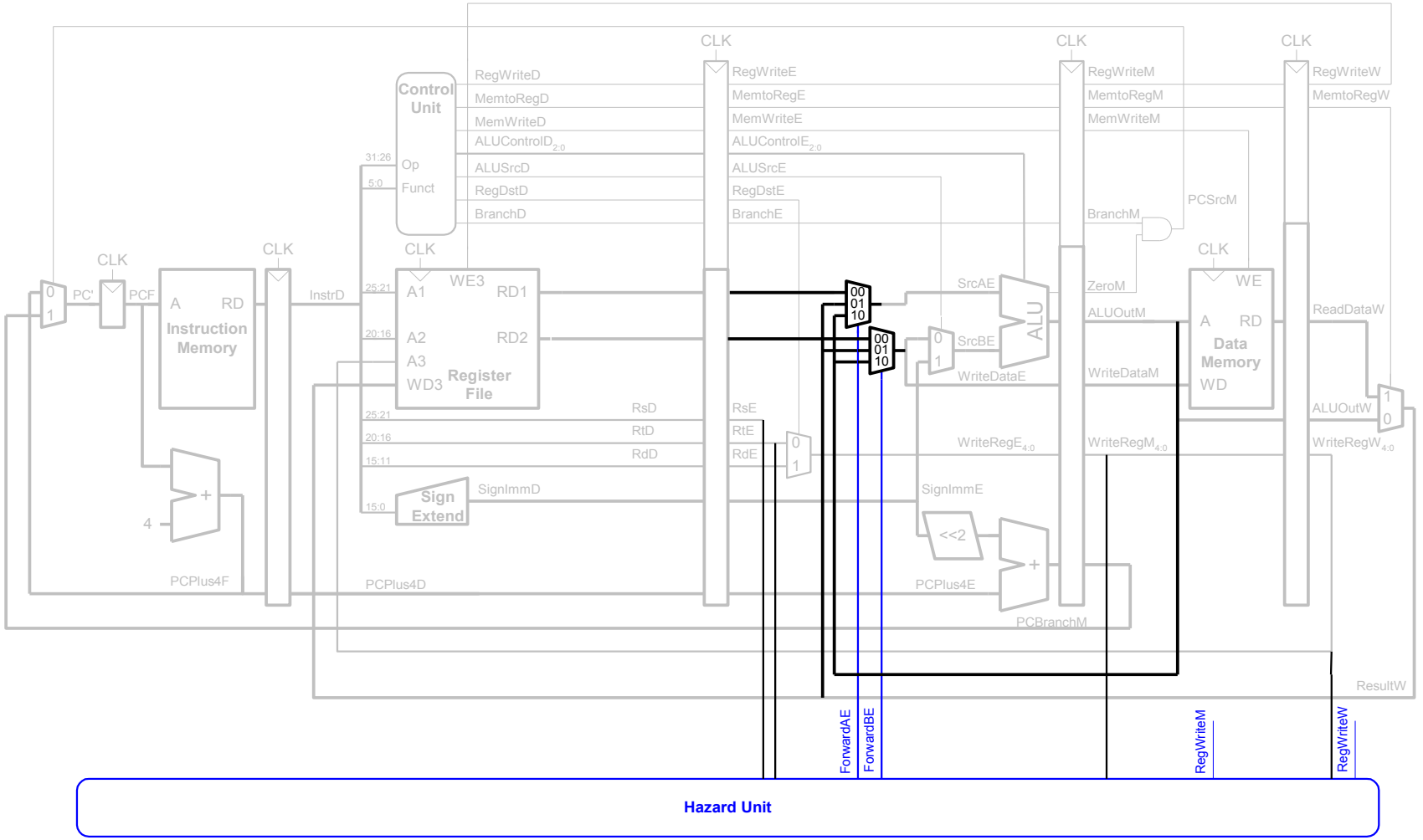
- Insert enough `nop`s for result to be ready
- Or move independent useful instructions forward



Data Forwarding



Data Forwarding



Data Forwarding

- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage
- Forwarding logic for *ForwardAE*:

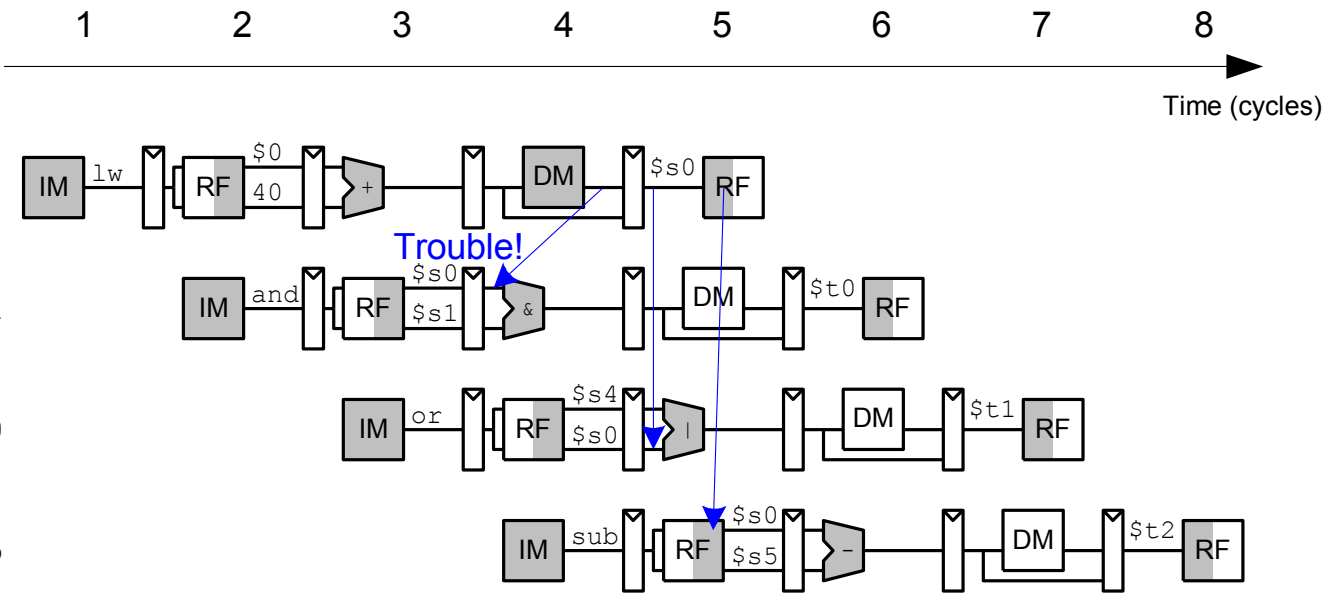
```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
 then   ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
 then   ForwardAE = 01

else        ForwardAE = 00
```

Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*

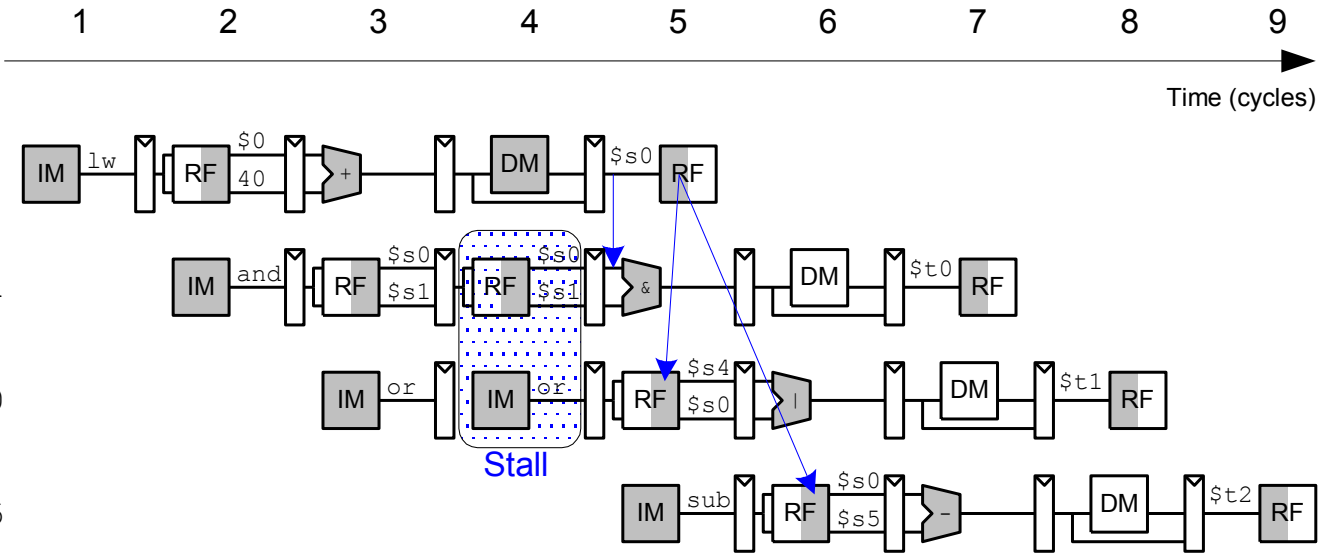
Stalling

```
lw $s0, 40($0)
and $t0, $s0, $s1
or  $t1, $s4, $s0
sub $t2, $s0, $s5
```

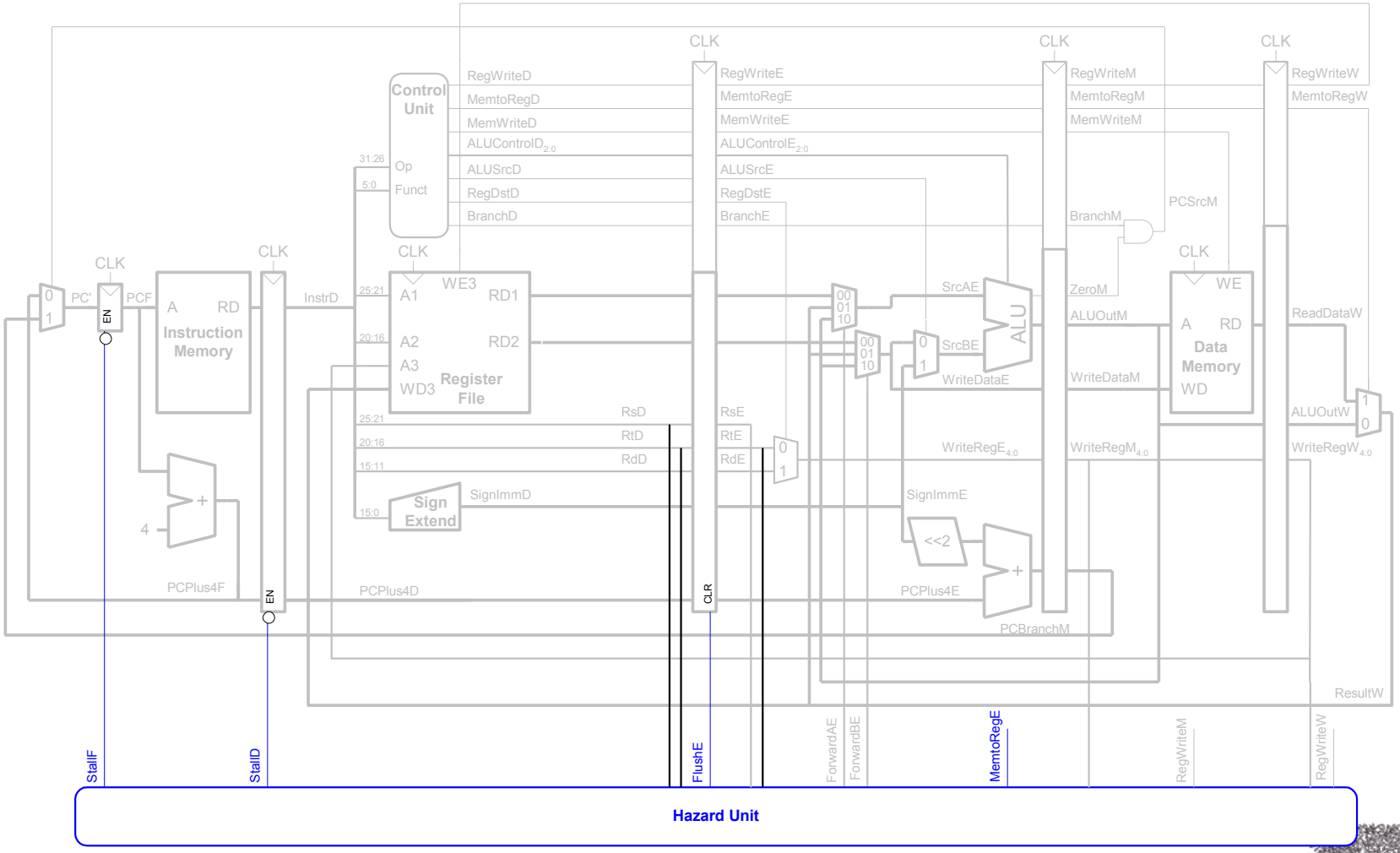


Stalling

```
lw $s0, 40($0)
and $t0, $s0, $s1
or  $t1, $s4, $s0
sub $t2, $s0, $s5
```



Stalling Hardware



Stalling Logic

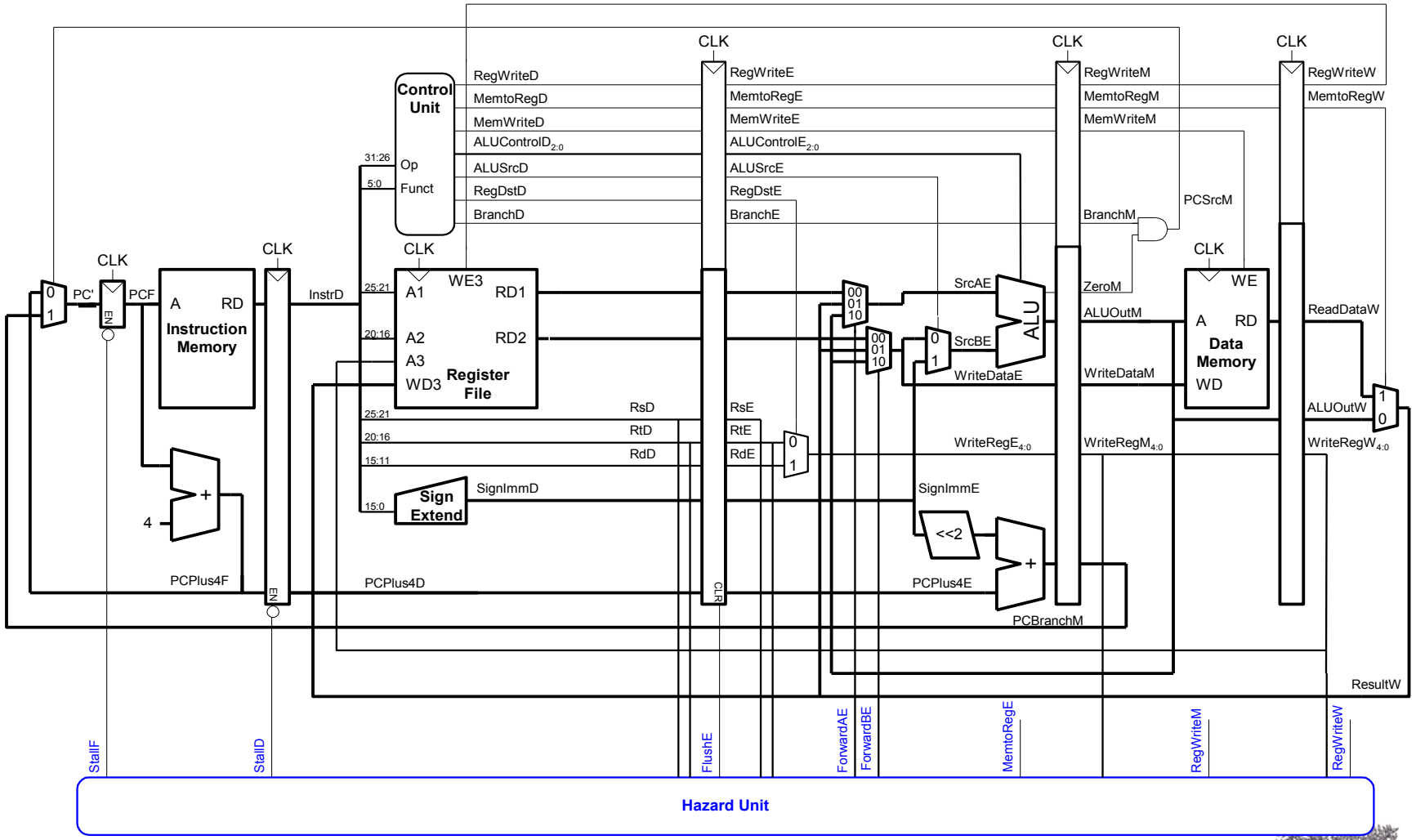
```
lwstall =
  ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall
```

Control Hazards

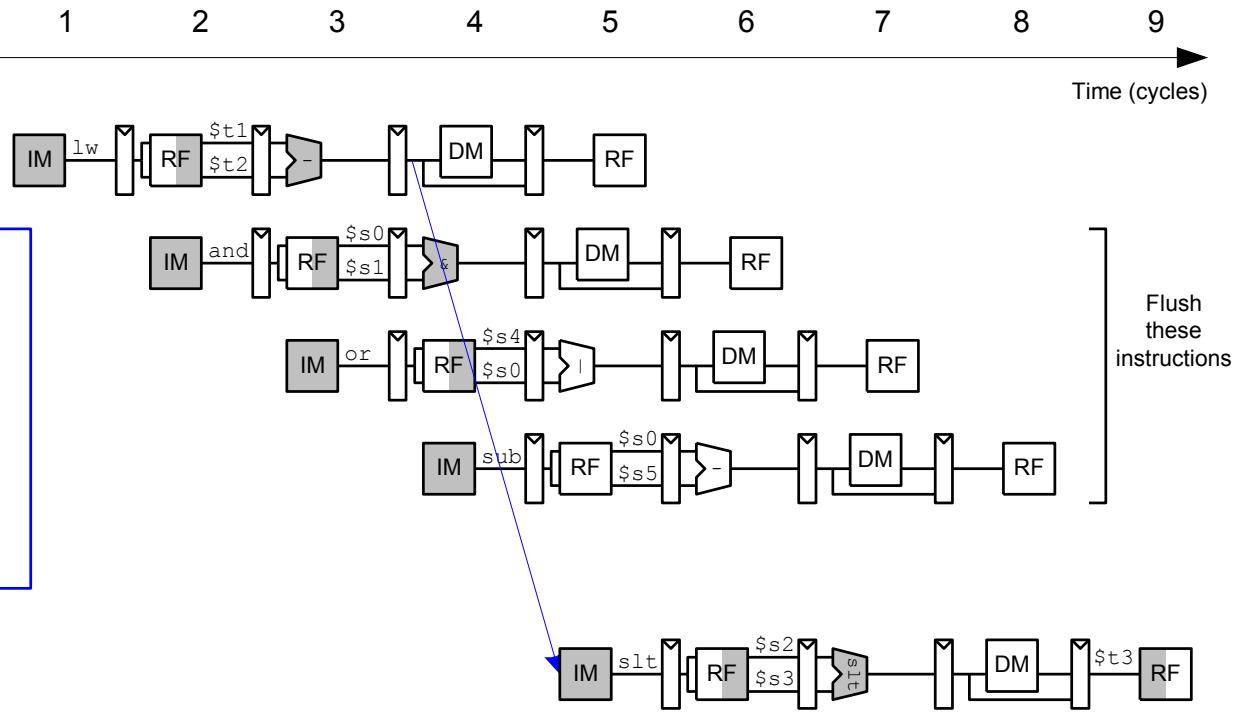
- **`beq`:**
  - branch not determined until 4th stage of pipeline
  - Instructions after branch fetched before branch occurs
  - These instructions must be flushed if branch happens
- **Branch misprediction penalty**
  - number of instruction flushed when branch is taken
  - May be reduced by determining branch earlier

Control Hazards: Original Pipeline

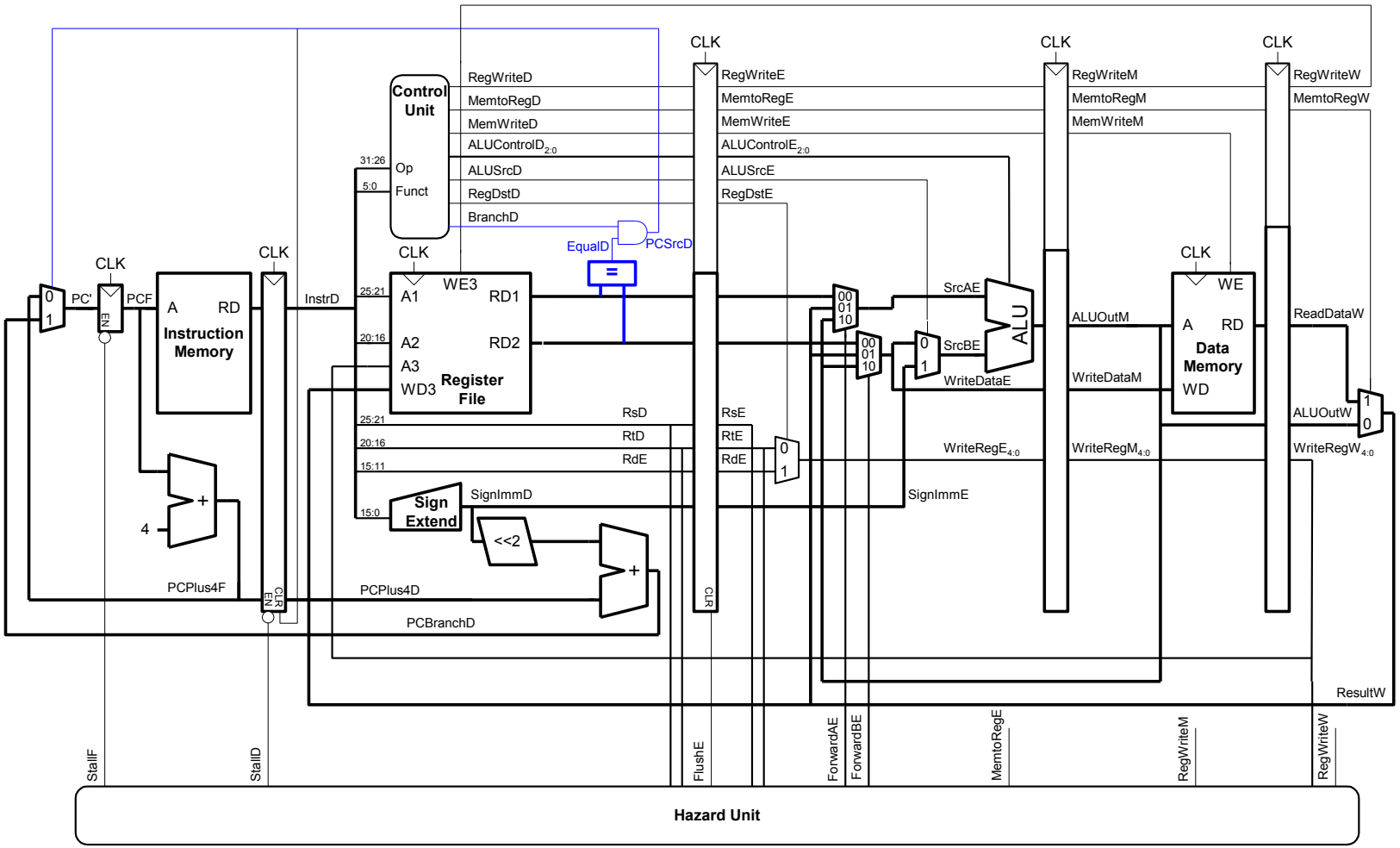


Control Hazards

```
20   beq $t1, $t2, 40
24   and $t0, $s0, $s1
28   or  $t1, $s4, $s0
2C   sub $t2, $s0, $s5
30   ...
...
64   slt $t3, $s2, $s3
```

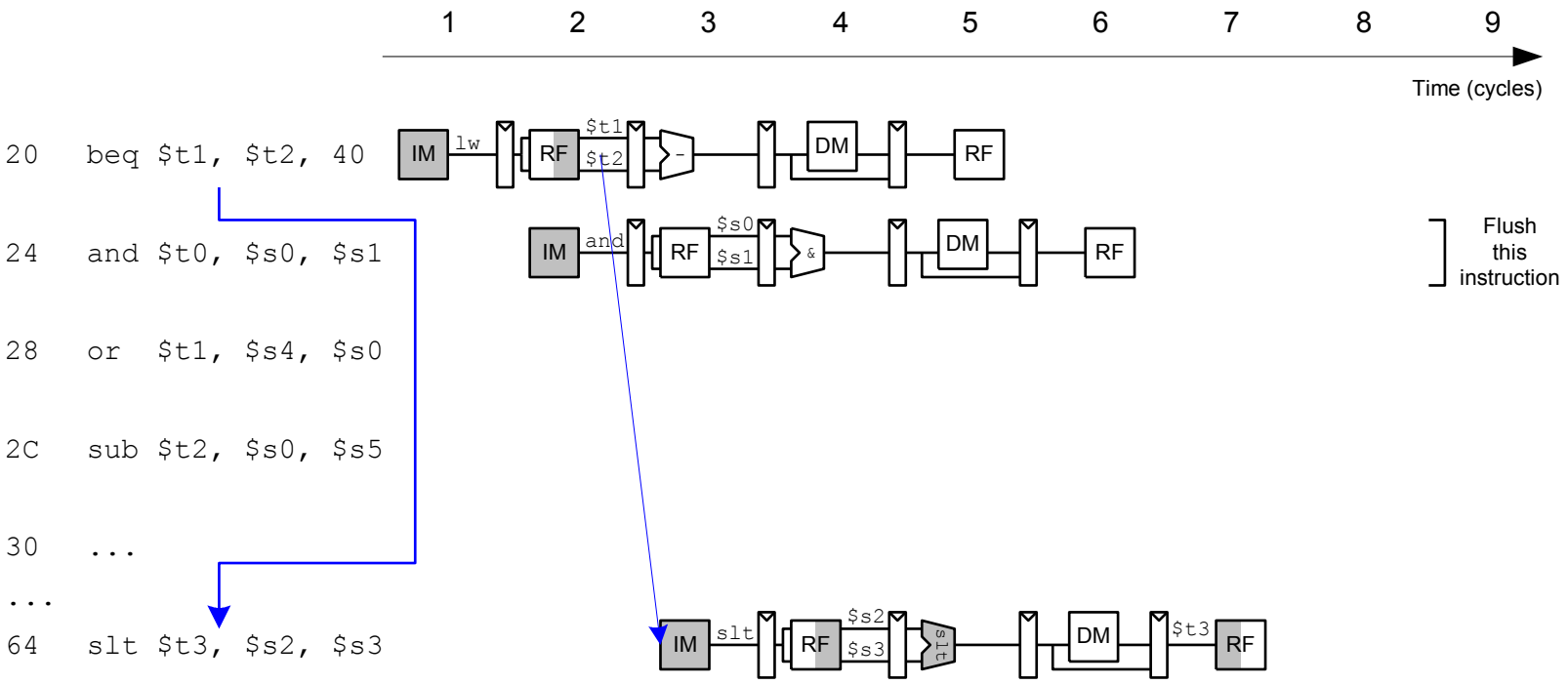


Early Branch Resolution

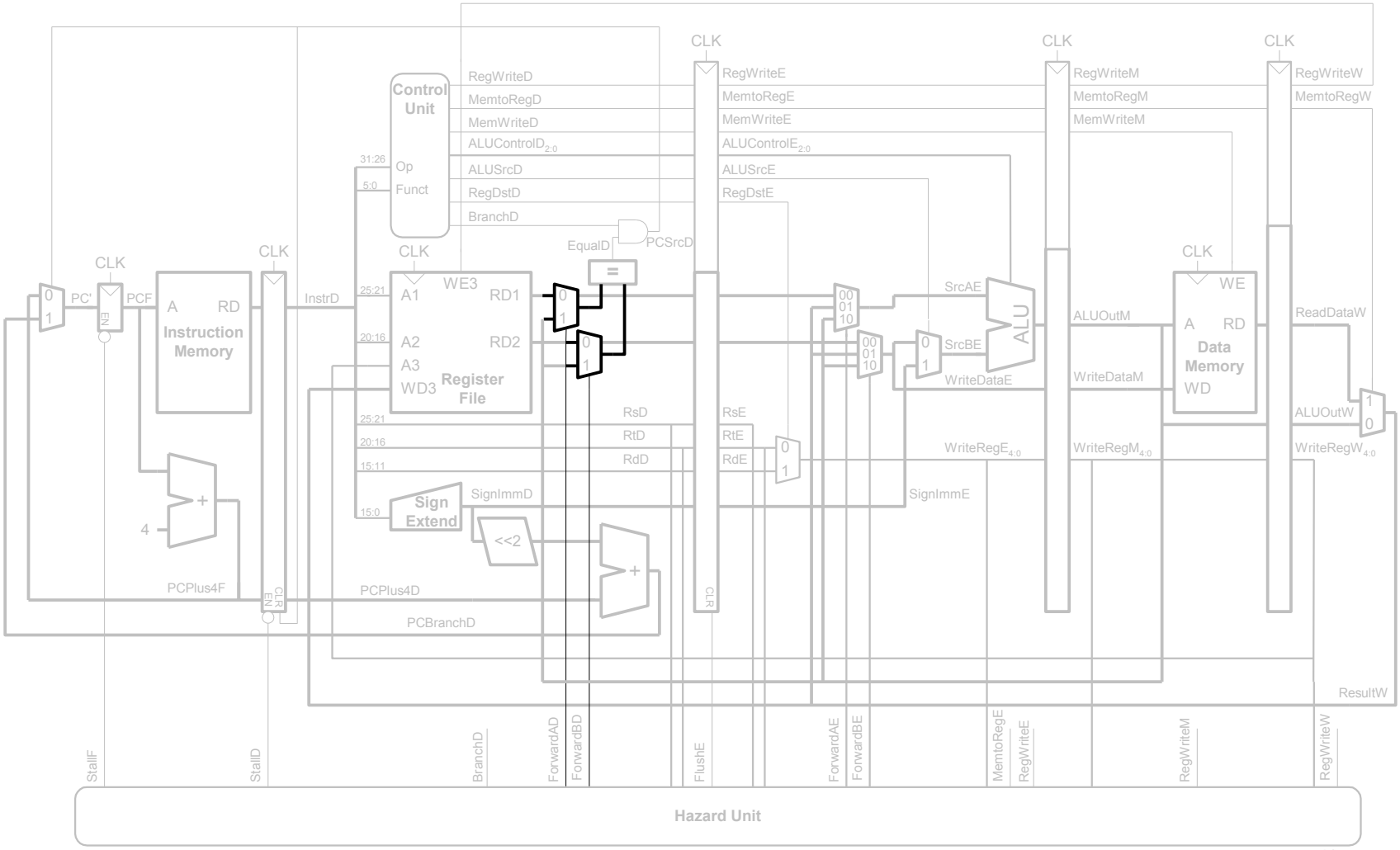


Introduced another data hazard in Decode stage

Early Branch Resolution



Handling Data & Control Hazards



Control Forwarding & Stalling Logic

- **Forwarding logic:**

```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM
```

- **Stalling logic:**

```
branchstall = BranchD AND RegWriteE AND
                (WriteRegE == rsD OR WriteRegE == rtD)
              OR
      BranchD AND MemtoRegM AND
                (WriteRegM == rsD OR WriteRegM == rtD)
```

```
StallF = StallD = FlushE = lwstall OR branchstall
```

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

Pipelined Performance Example

- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
  - All jumps flush next instruction
- **What is the average CPI?**

Pipelined Performance Example

- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
  - All jumps flush next instruction
- **What is the average CPI?**
  - Load/Branch CPI = 1 when no stalling, 2 when stalling
  - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
  - $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

Average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1)
= **1.15**

Pipelined Performance

- Pipelined processor critical path:

$$T_c = \max \{$$
$$t_{pcq} + t_{\text{mem}} + t_{\text{setup}}$$
$$2(t_{RFread} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}})$$
$$t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}}$$
$$t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}}$$
$$2(t_{pcq} + t_{\text{mux}} + t_{\text{RFwrite}}) \}$$

Pipelined 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
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	$T_{memwrite}$	220
Register file write	$t_{RFwrite}$	100 ps

$$T_c = 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$$
$$= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps} = \mathbf{550 \text{ ps}}$$

Pipelined Performance Example

Program with 100 billion instructions

Execution Time = (# instructions) × CPI × T_c

$= (100 \times 10^9)(1.15)(550 \times 10^{-12})$

= 63 seconds

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	92.5	1
Multicycle	133	0.70
Pipelined	63	1.47