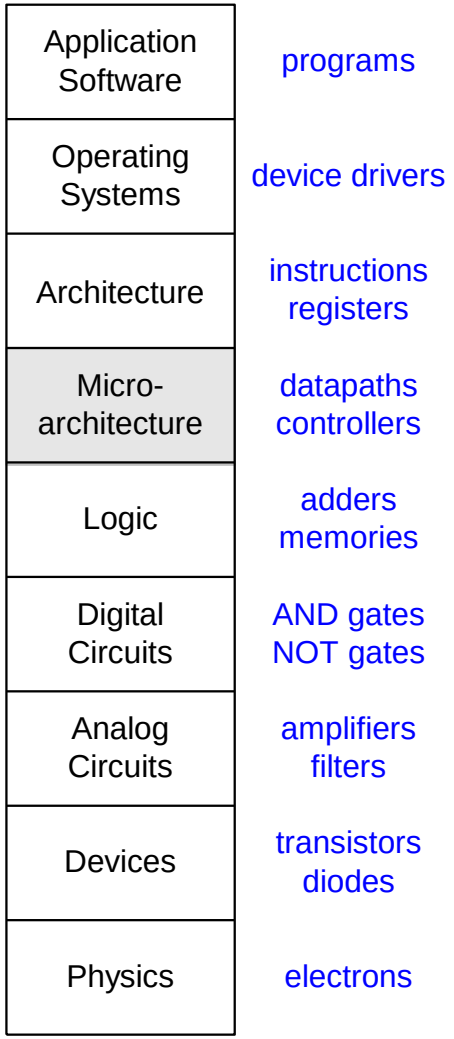
Chapter 7 :: Microarchitecture

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

Introduction

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



Microarchitecture

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

Processor Performance

- Program execution time

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

- Definitions:
  - Cycles/instruction = CPI
  - Seconds/cycle = clock period
  - 1/CPI = Instructions/cycle = IPC
- Challenge is to satisfy constraints of:
  - Cost
  - Power
  - Performance

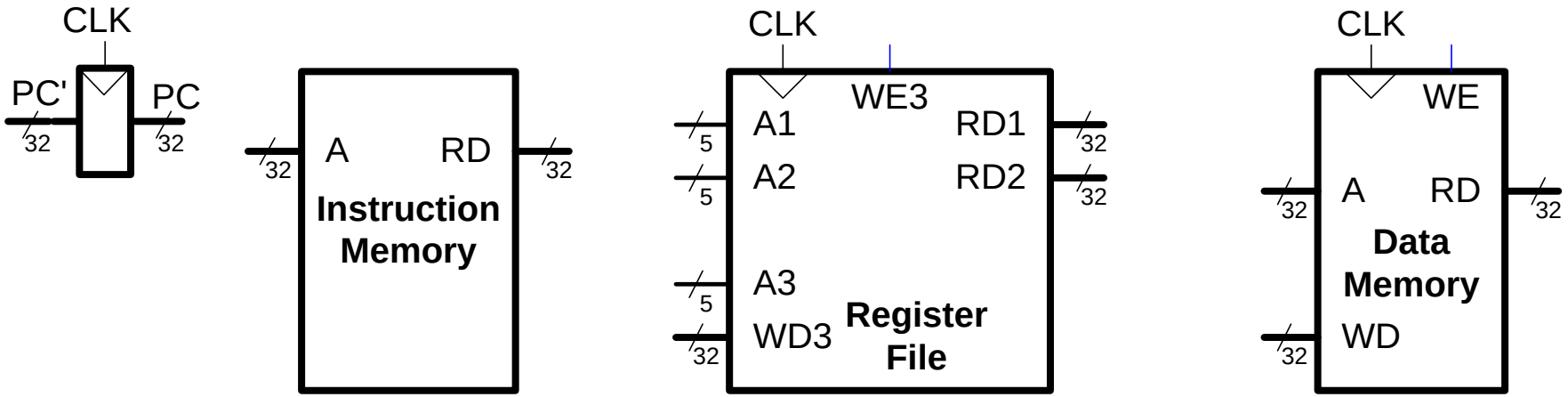
MIPS Processor

- We consider a subset of MIPS instructions:
  - R-type instructions: `and`, `or`, `add`, `sub`, `slt`
  - Memory instructions: `lw`, `sw`
  - Branch instructions: `beq`
- Later consider adding `addi` and `j`

Architectural State

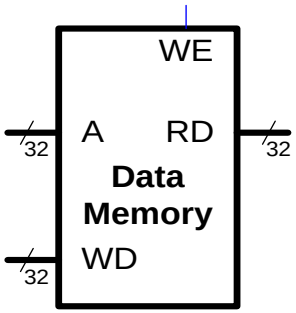
- Determines everything about a processor:
  - PC
  - 32 registers
  - Memory

MIPS State Elements



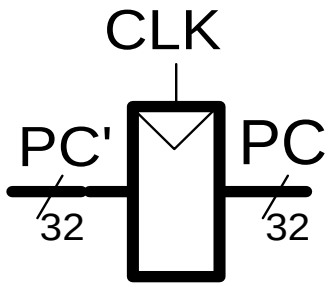
Data Memory

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all; use IEEE.STD_LOGIC_ARITH.all;
entity dmem is — — data memory
port (clk, we: in STD_LOGIC;
a, wd: in STD_LOGIC_VECTOR (31 downto 0);
rd: out STD_LOGIC_VECTOR (31 downto 0));
end;
architecture behave of dmem is
begin
process is
type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR (31 downto 0);
variable mem: ramtype;
begin
— — read or write memory
loop
if clk'event and clk = '1' then
          if (we = '1') then mem (CONV_INTEGER (a(7 downto 2))):= wd;
          end if;
end if;
rd <= mem (CONV_INTEGER (a (7 downto 2)));
wait on clk, a;
end loop;
end process;
end;
```



RESETTABLE FLIP-FLOP

```
library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity flopr is – – flip-flop with synchronous reset
generic (width: integer);
port (   clk, reset: in STD_LOGIC;
         d:          in STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;
architecture asynchronous of flopr is
begin
process (clk, reset) begin
if reset = '1' then q <= CONV_STD_LOGIC_VECTOR(0, width);
elsif clk'event and clk = '1' then
         q <= d;
end if;
end process;
end;
```

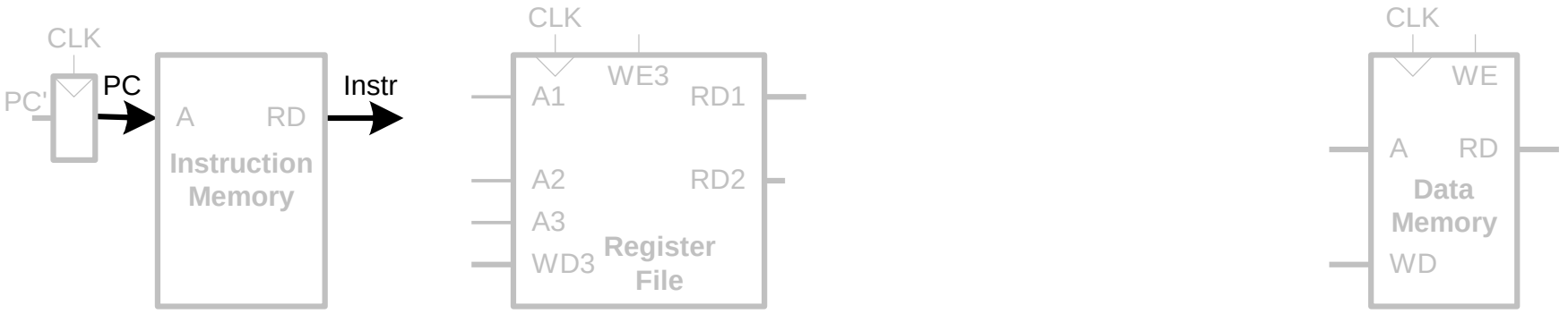


Single-Cycle MIPS Processor

- Datapath
- Control

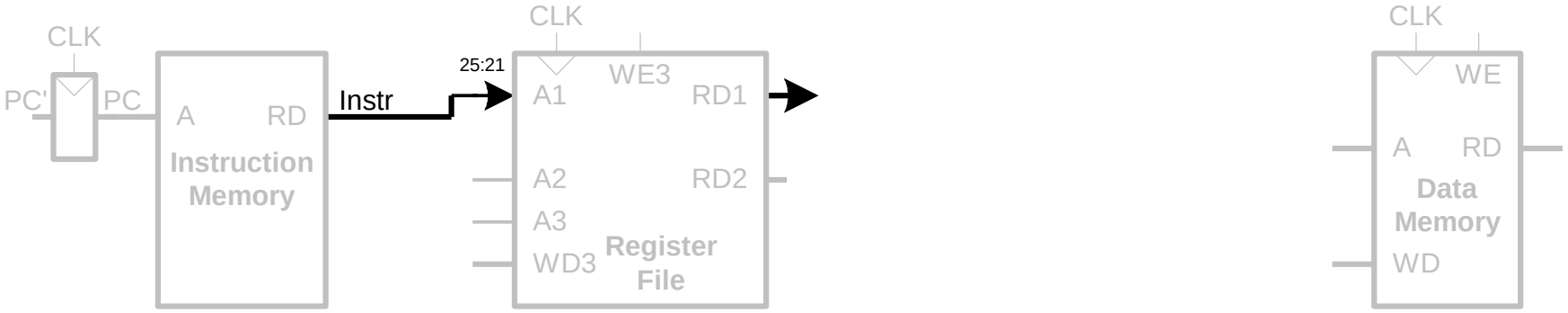
Single-Cycle Datapath: `lw` fetch

- First consider executing `lw`
- **STEP 1:** Fetch instruction



Single-Cycle Datapath: `lw` register read

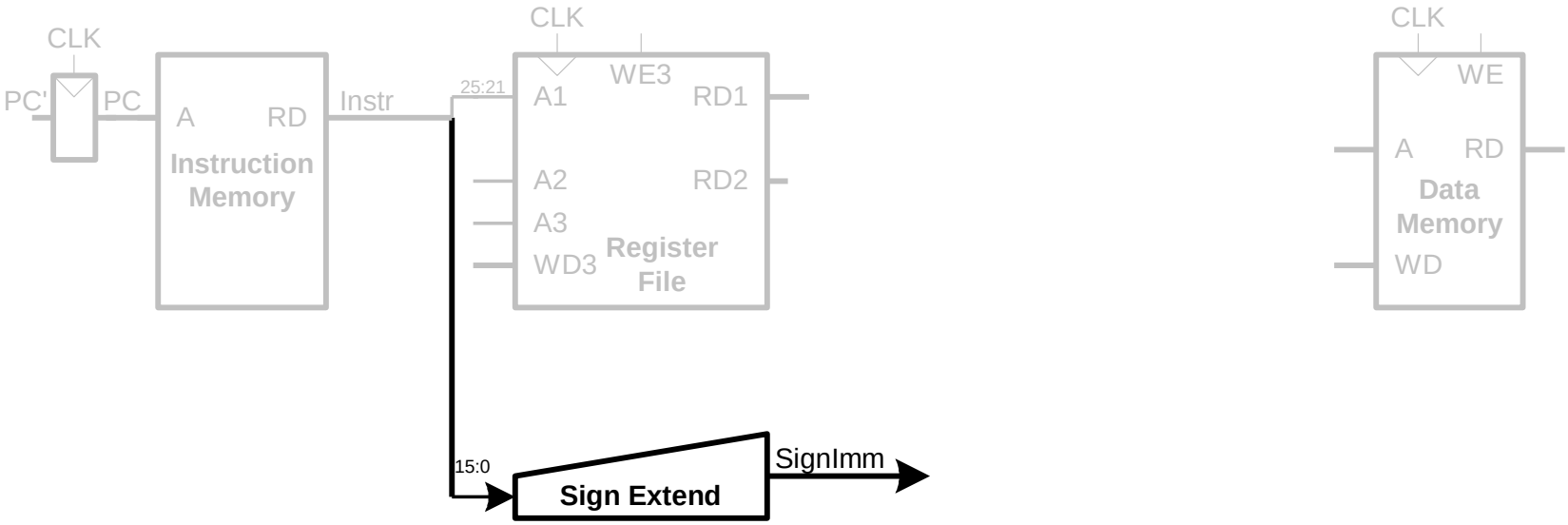
- **STEP 2:** Read source operands from register file



I-Type

Single-Cycle Datapath: `lw` immediate

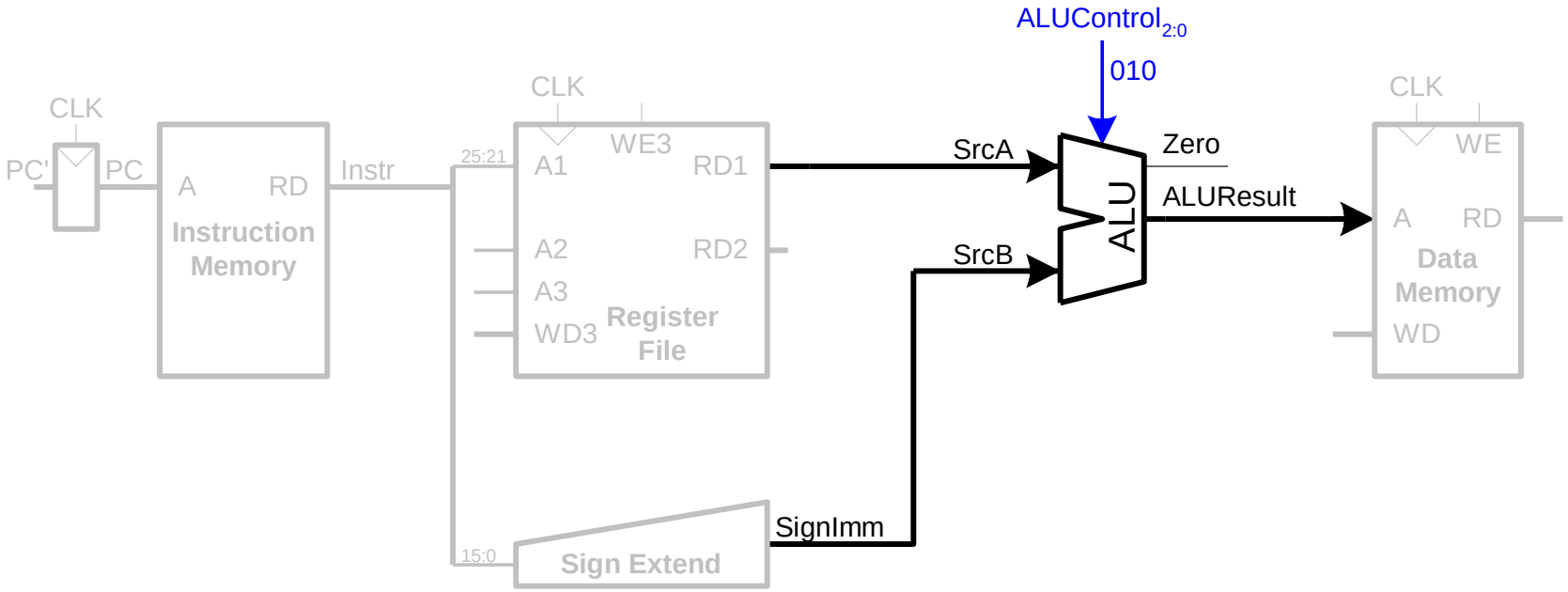
- **STEP 3:** Sign-extend the immediate



I-Type

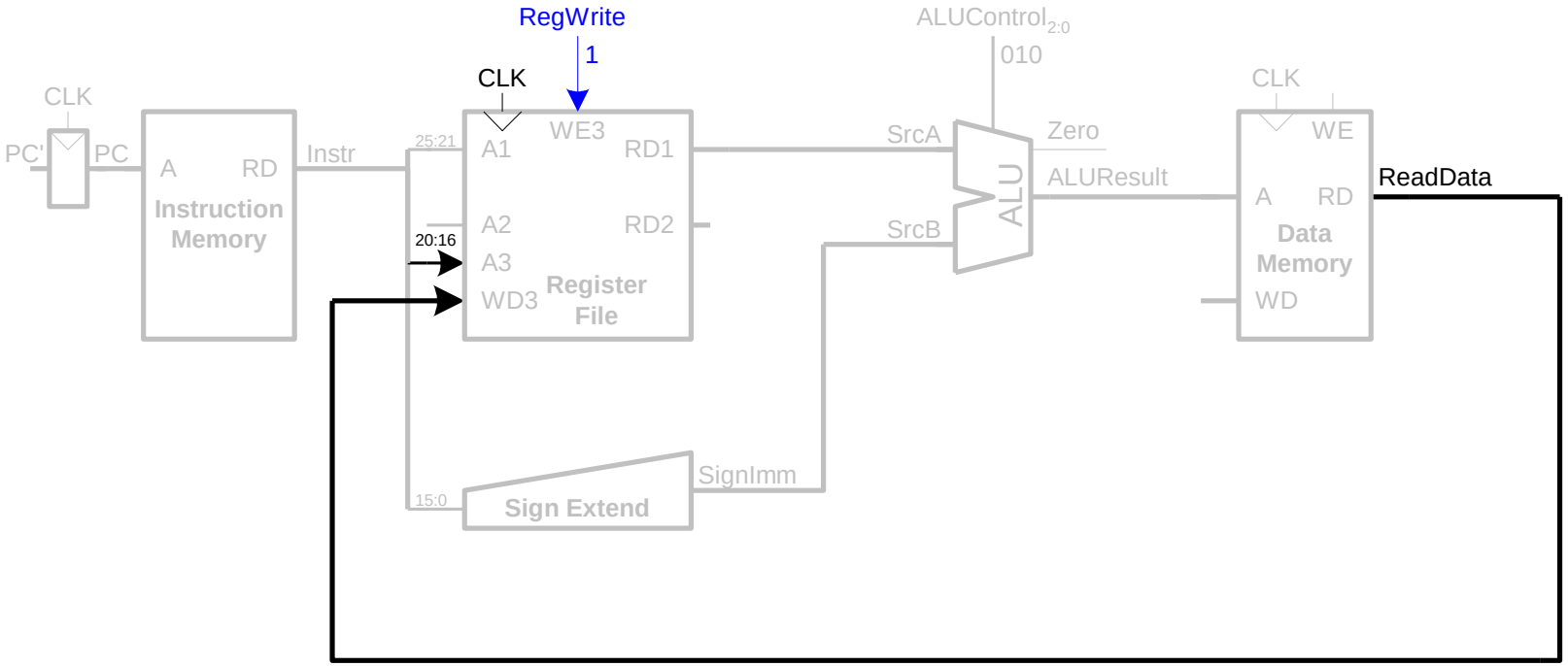
Single-Cycle Datapath: `lw` address

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



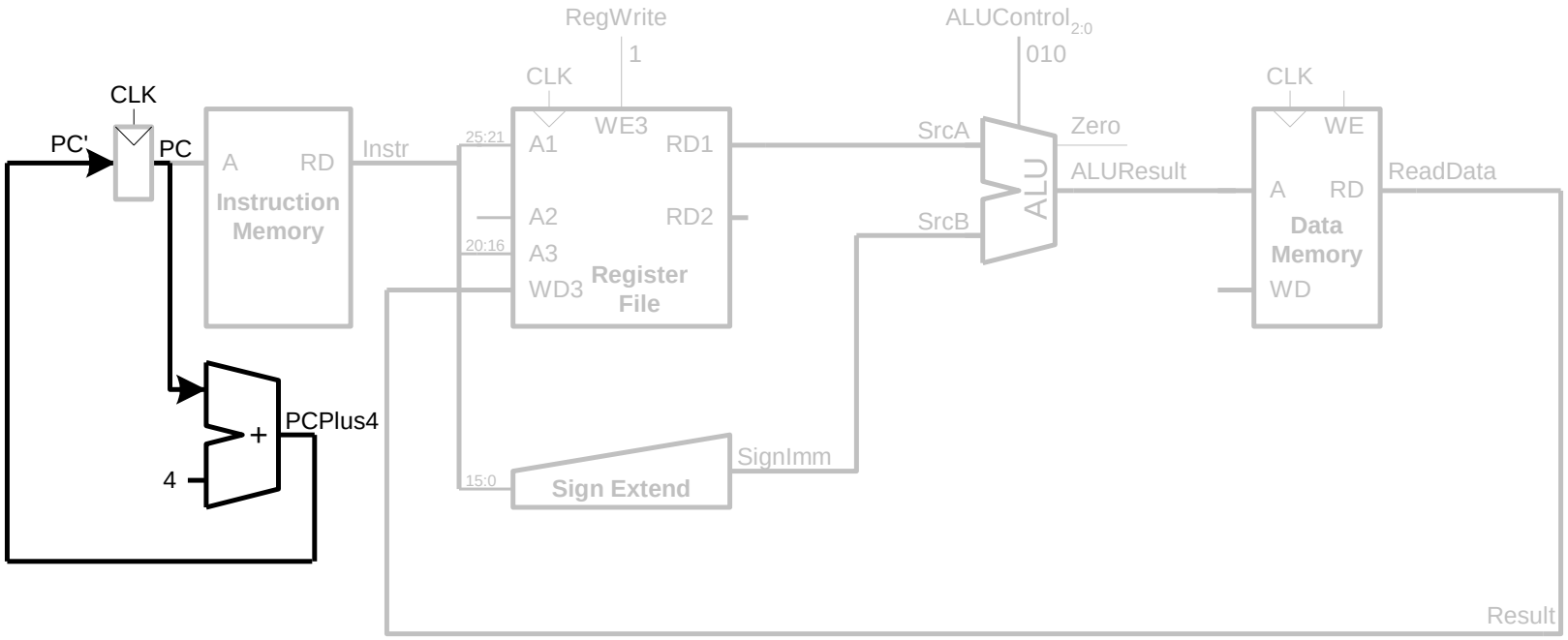
Single-Cycle Datapath: `lw` memory read

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



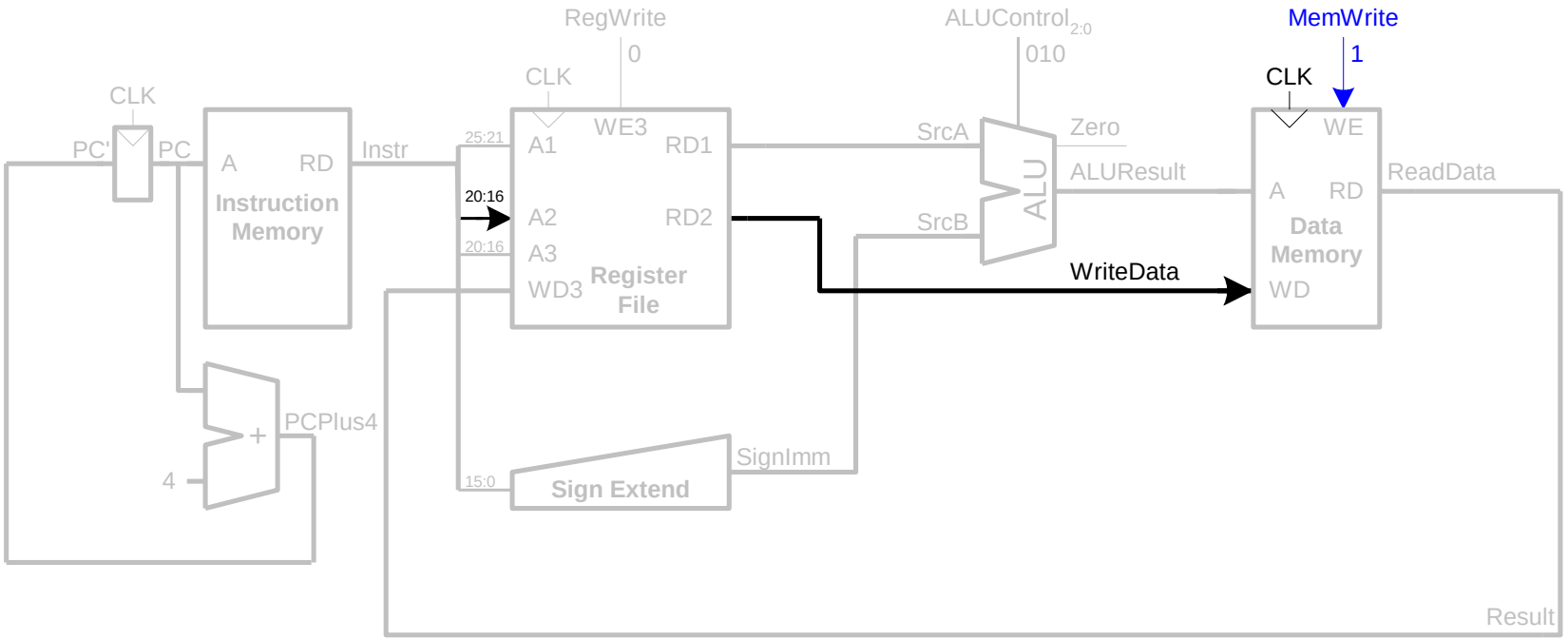
Single-Cycle Datapath: `lw` PC increment

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



Single-Cycle Datapath: `SW`

- Write data in `rt` to memory

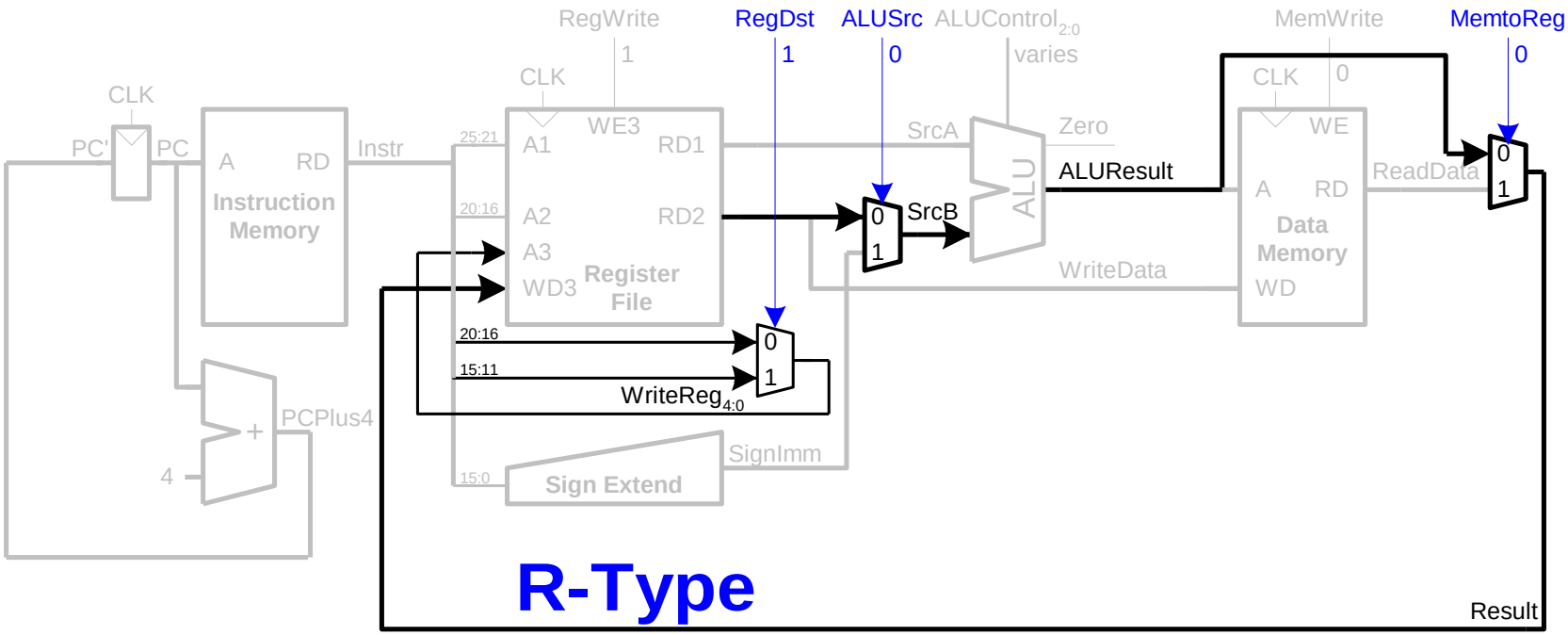


I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: R-type instructions

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

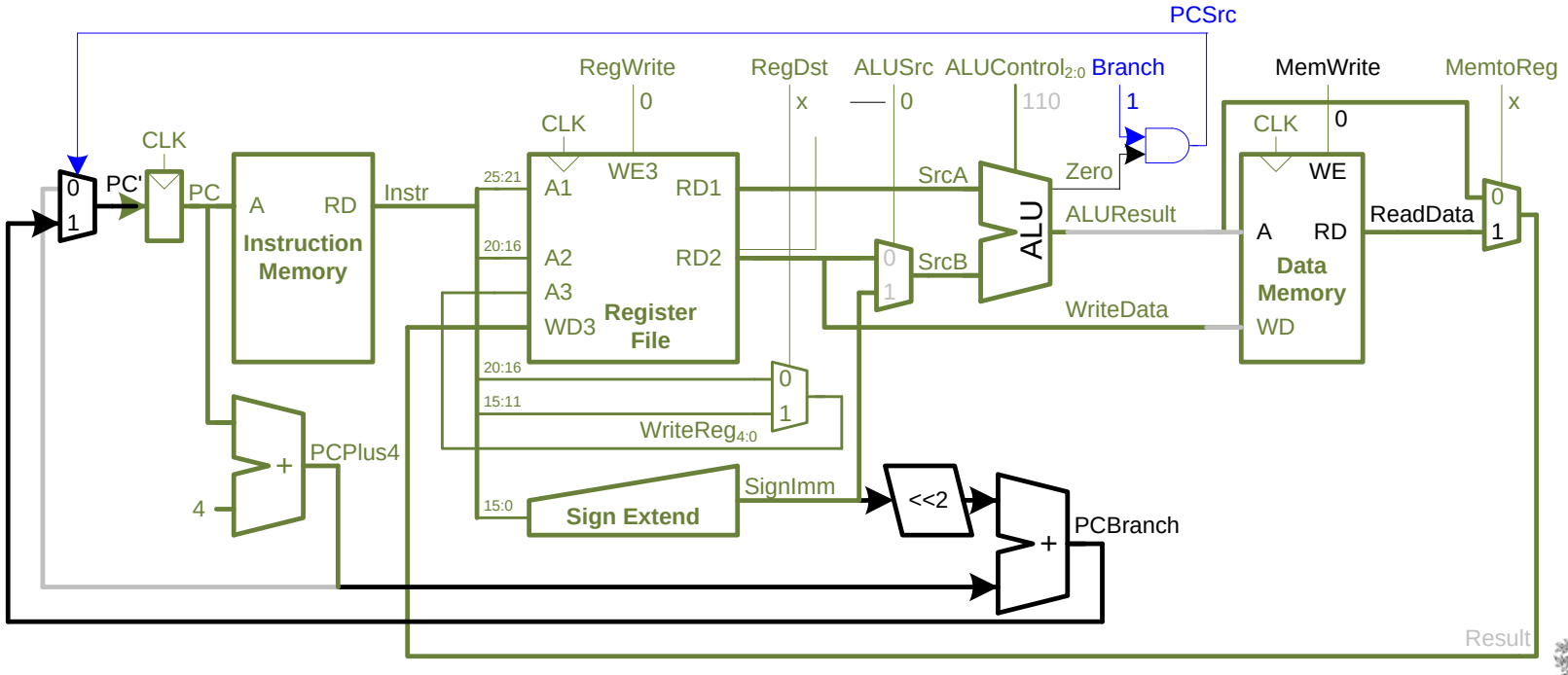


op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Single-Cycle Datapath: `beq`

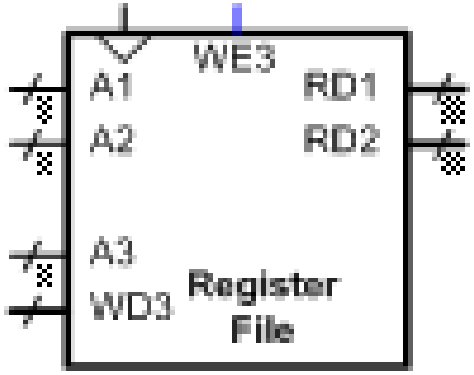
- Determine whether values in `rs` and `rt` are equal
- Calculate branch target address:

  BTA = (sign-extended immediate << 2) + (PC+4)



Register File in VHDL (1)

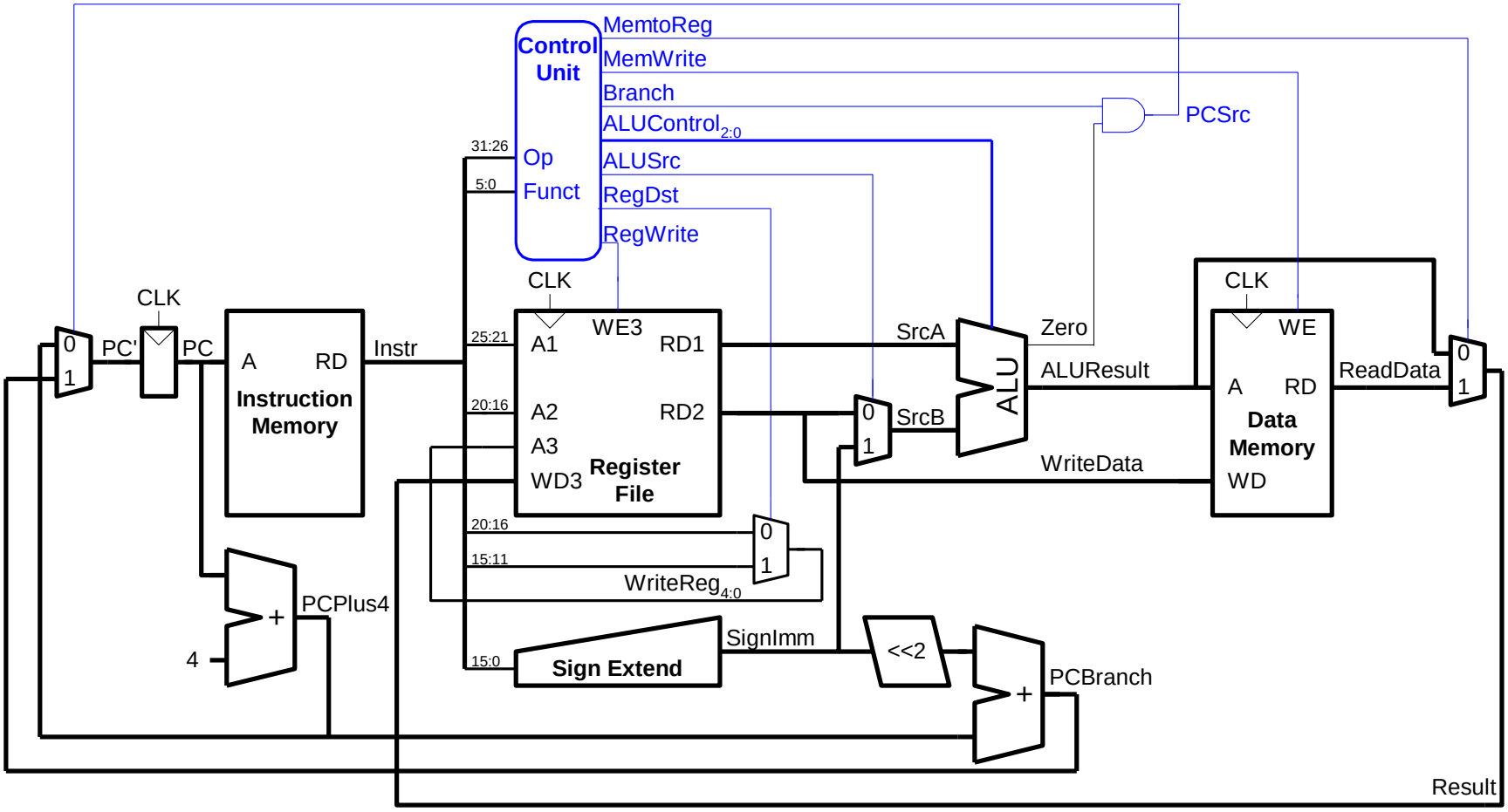
```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity regfile is — — three-port register file
port(   clk: in STD_LOGIC;
        we3: in STD_LOGIC;
        ra1, ra2, wa3: in STD_LOGIC_VECTOR(4 downto 0);
        wd3: in STD_LOGIC_VECTOR(31 downto 0);
        rd1, rd2: out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of regfile is
type ramtype is array (31 downto 0) of STD_LOGIC_VECTOR (31 downto
   0);
signal mem: ramtype;
begin
```



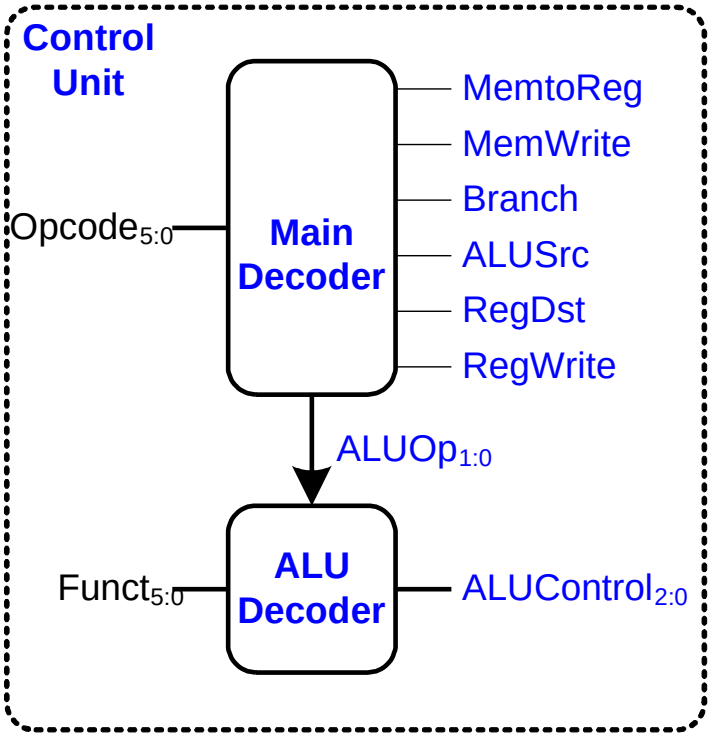
Register File in VHDL (2)

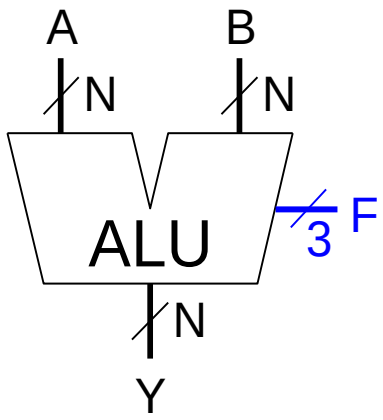
```
begin
- - three-ported register file
- - read two ports combinationally
- - write third port on rising edge of clock
process(clk) begin
    if clk'event and clk = '1' then
         if we3 = '1' then mem(CONV_INTEGER(wa3)) <= wd3;
         end if;
    end if;
end process;
process (ra1, ra2) begin
    if (conv_integer (ra1) = 0) then rd1 <= X"00000000";
- - register 0 holds 0
    else rd1 <= mem(CONV_INTEGER (ra1));
    end if;
    if (conv_integer(ra2) = 0) then rd2 <= X"00000000";
    else rd2 <= mem(CONV_INTEGER(ra2));
    end if;
end process;
end;
```

Complete Single-Cycle Processor



Control Unit



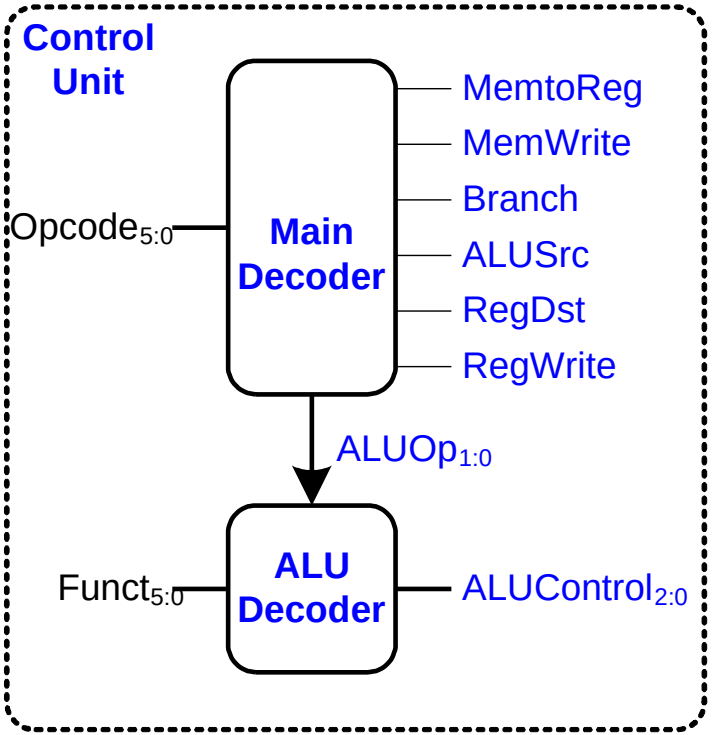


$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Control Unit: ALU Decoder

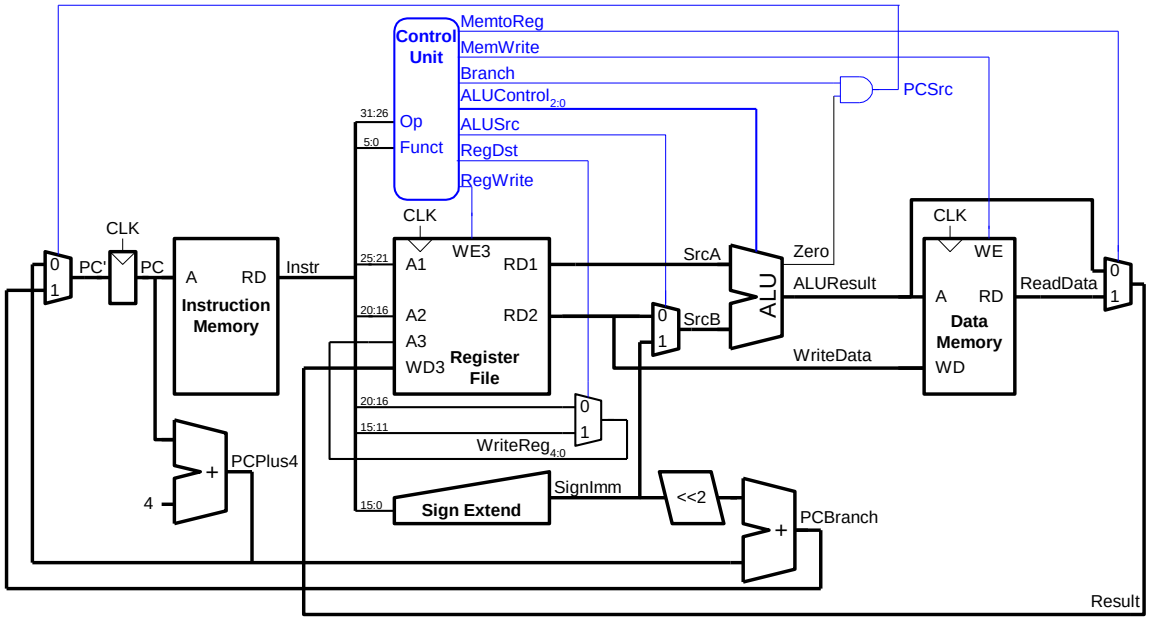
$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

$ALUOp_{1:0}$	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

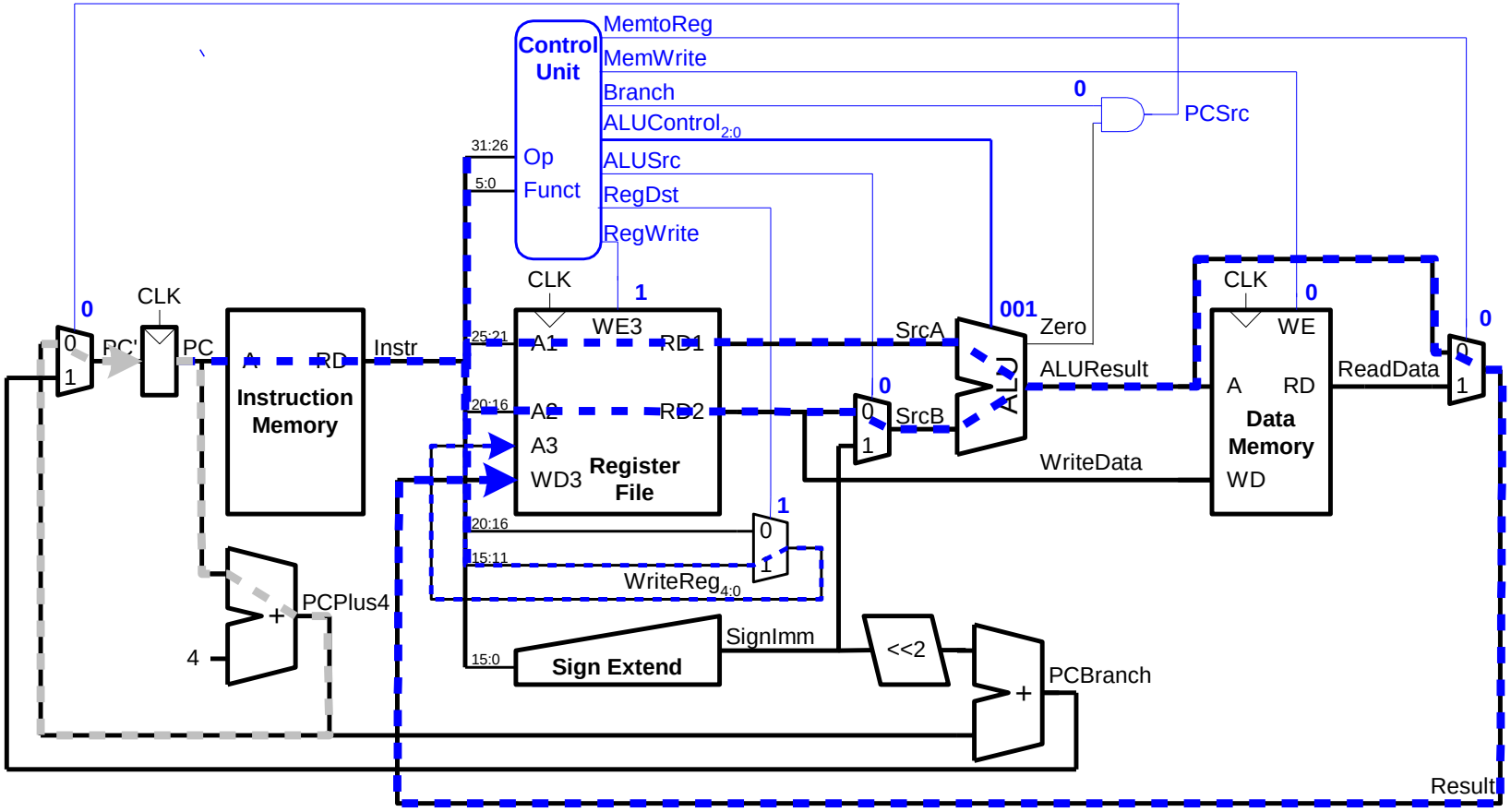
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000							
`lw`	100011							
`sw`	101011							
`beq`	000100							



Control Unit: Main Decoder

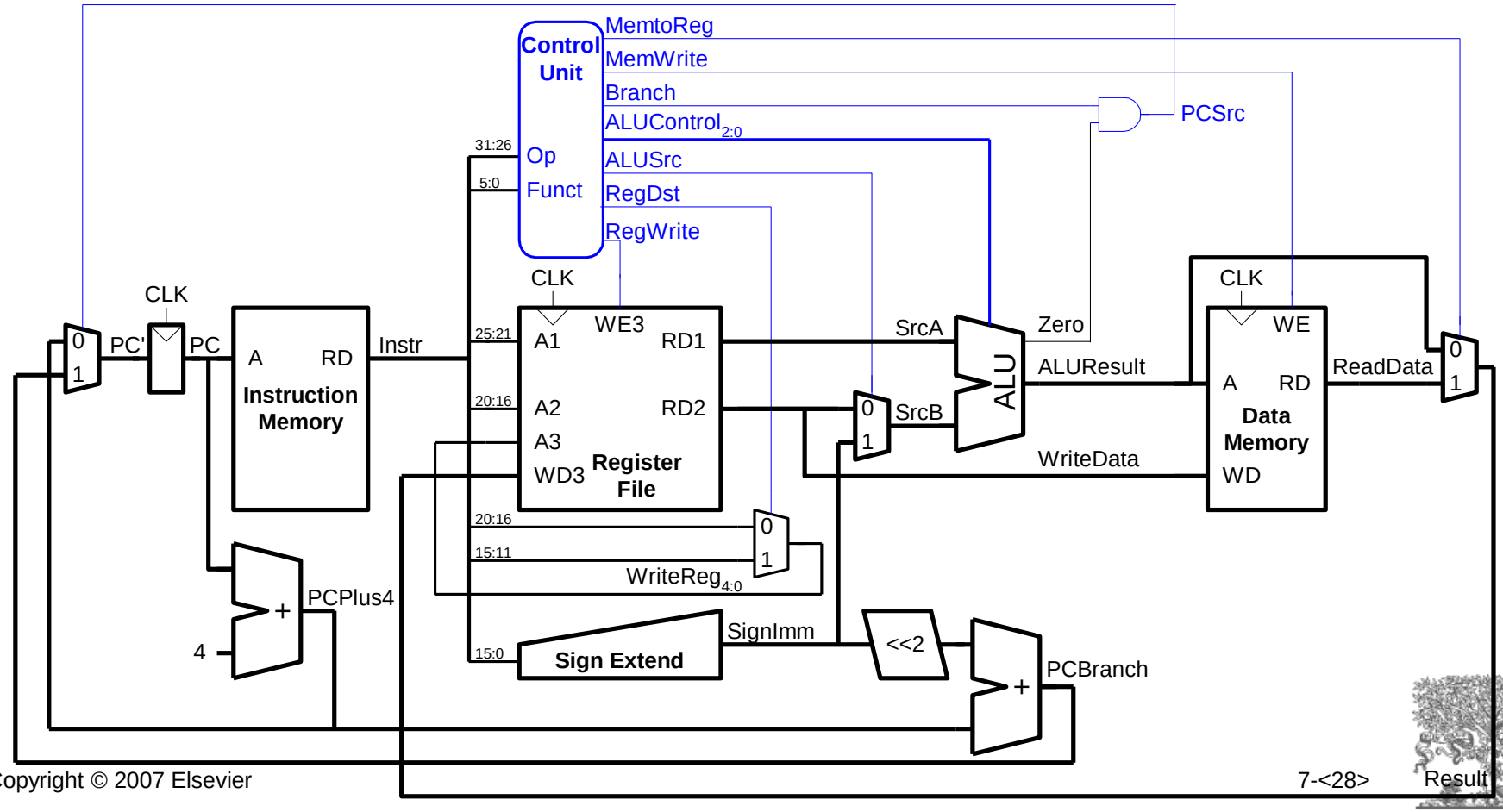
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

Single-Cycle Datapath Example: `or`



Extended Functionality: addi

- No change to datapath



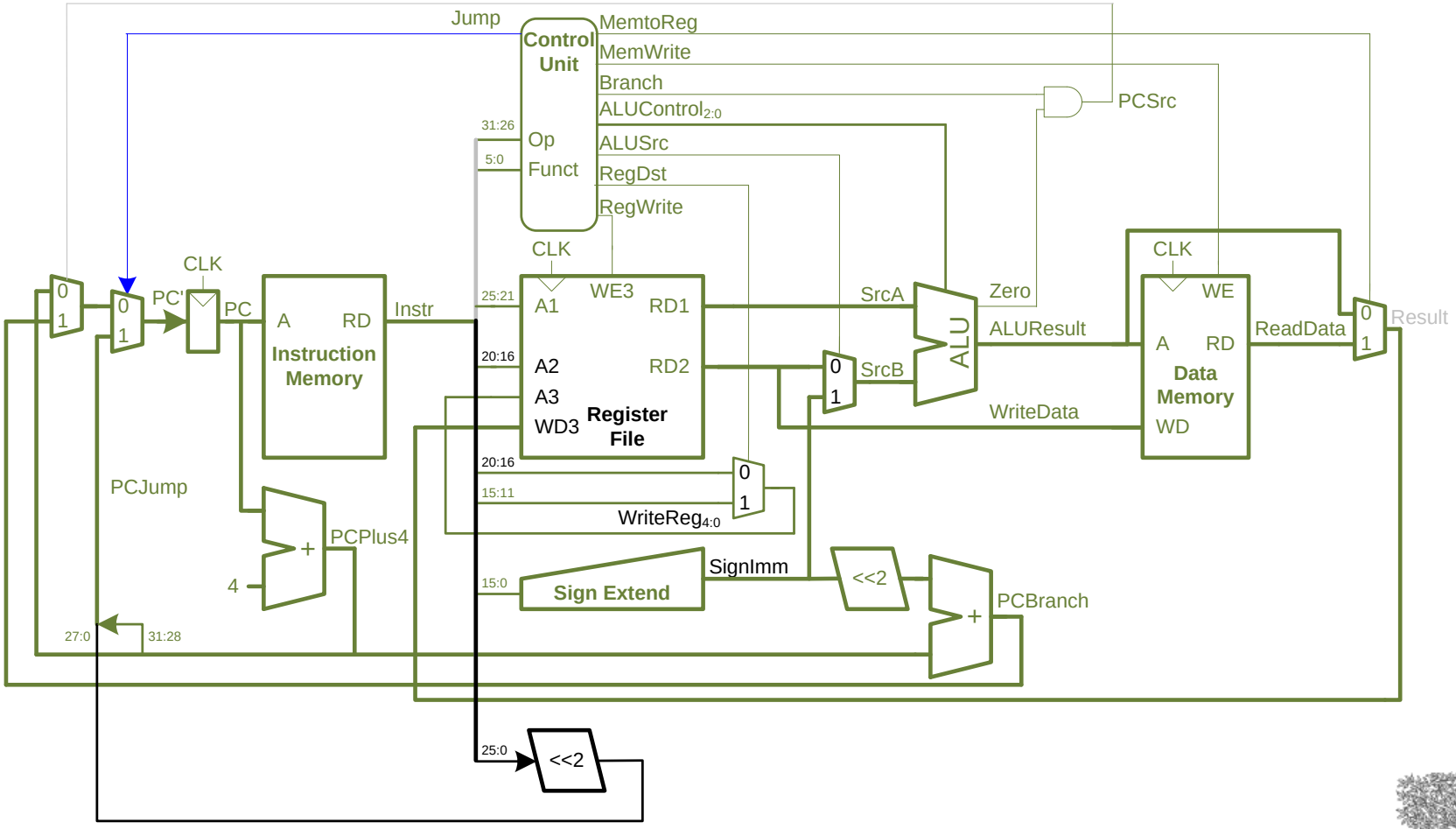
Control Unit: addi

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**	**0**	**0**	**X**	**XX**	**1**

Review: Processor Performance

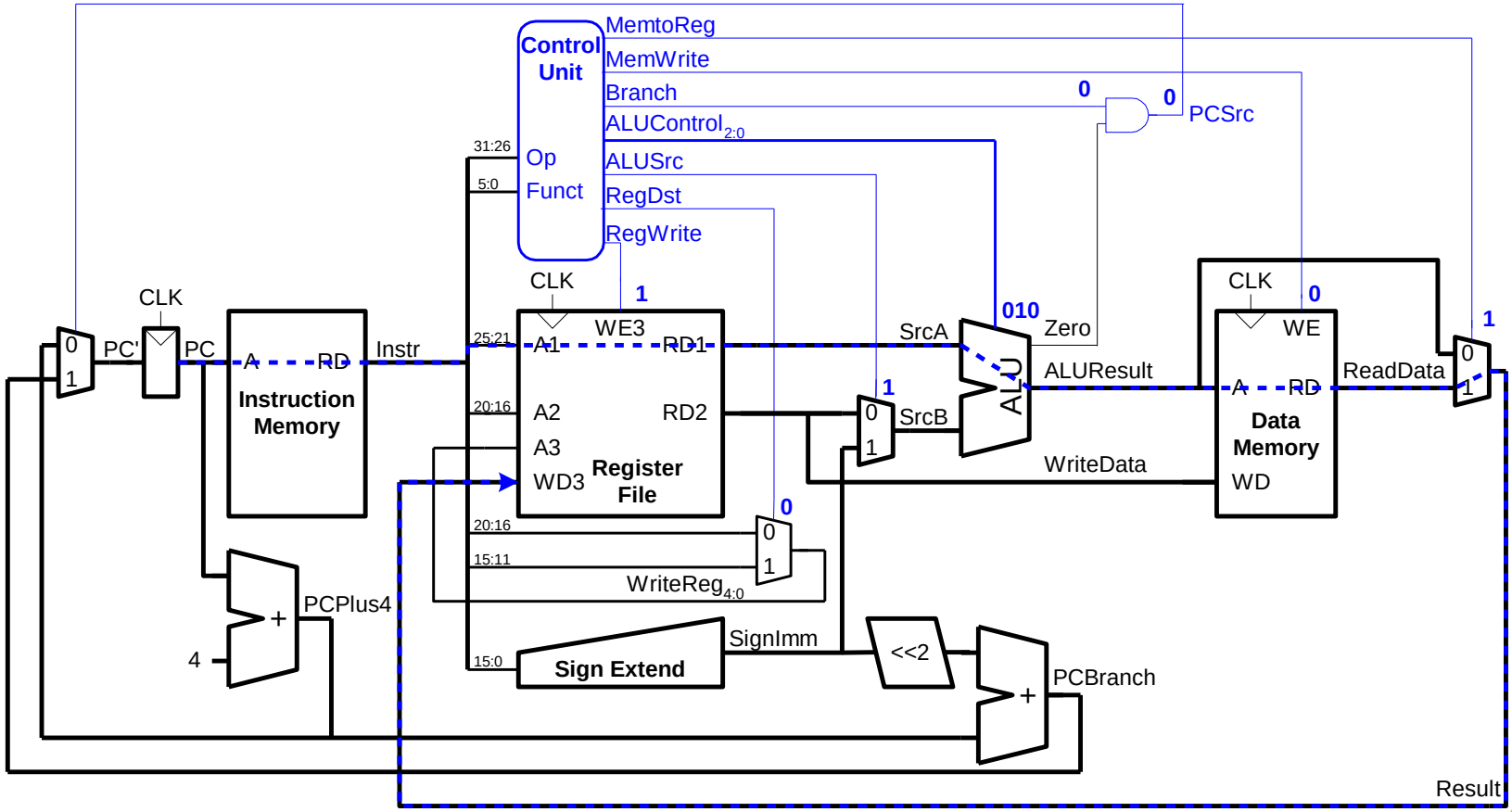
Program Execution Time

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

= # instructions x CPI x T_C

Single-Cycle Performance

- T_C is limited by the critical path (`lw`)



Single-Cycle Performance

- Single-cycle critical path:

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

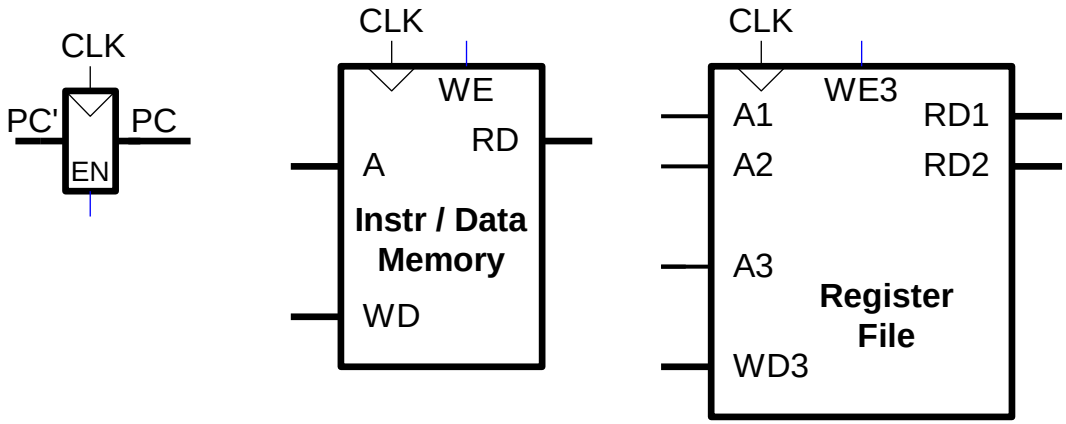
- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$

Multicycle MIPS Processor

- Single-cycle microarchitecture:
  + simple
  - cycle time limited by longest instruction (`lw`)
  - two adders/ALUs and two memories
- Multicycle microarchitecture:
  + higher clock speed
  + simpler instructions run faster
  + reuse expensive hardware on multiple cycles
  - sequencing overhead paid many times
- Same design steps: datapath & control

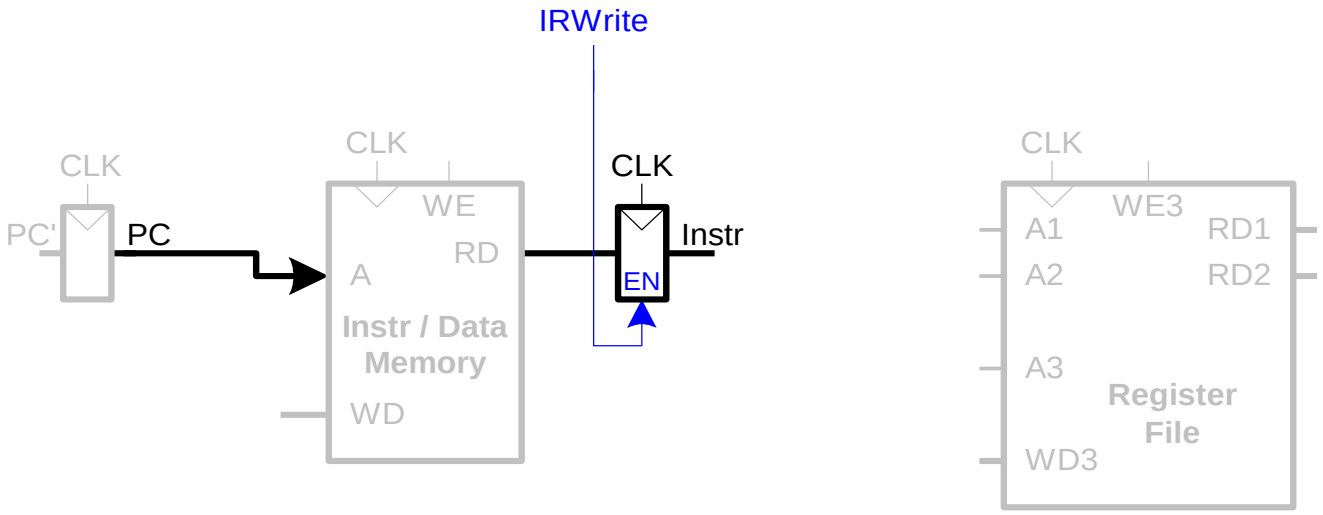
Multicycle State Elements

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

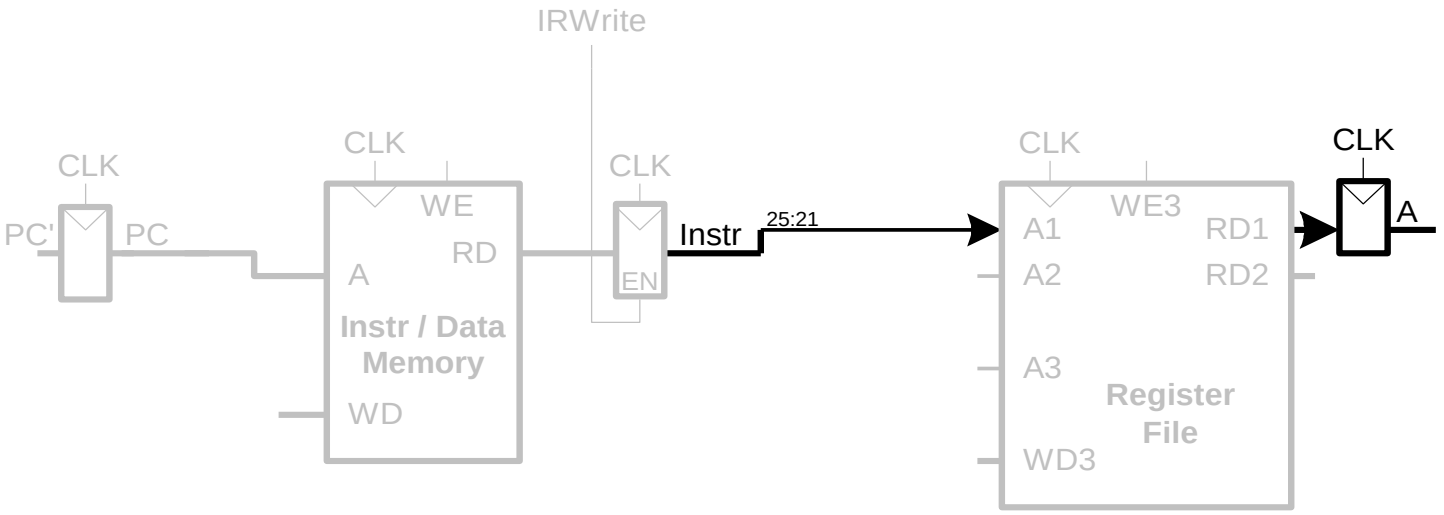


Multicycle Datapath: instruction fetch

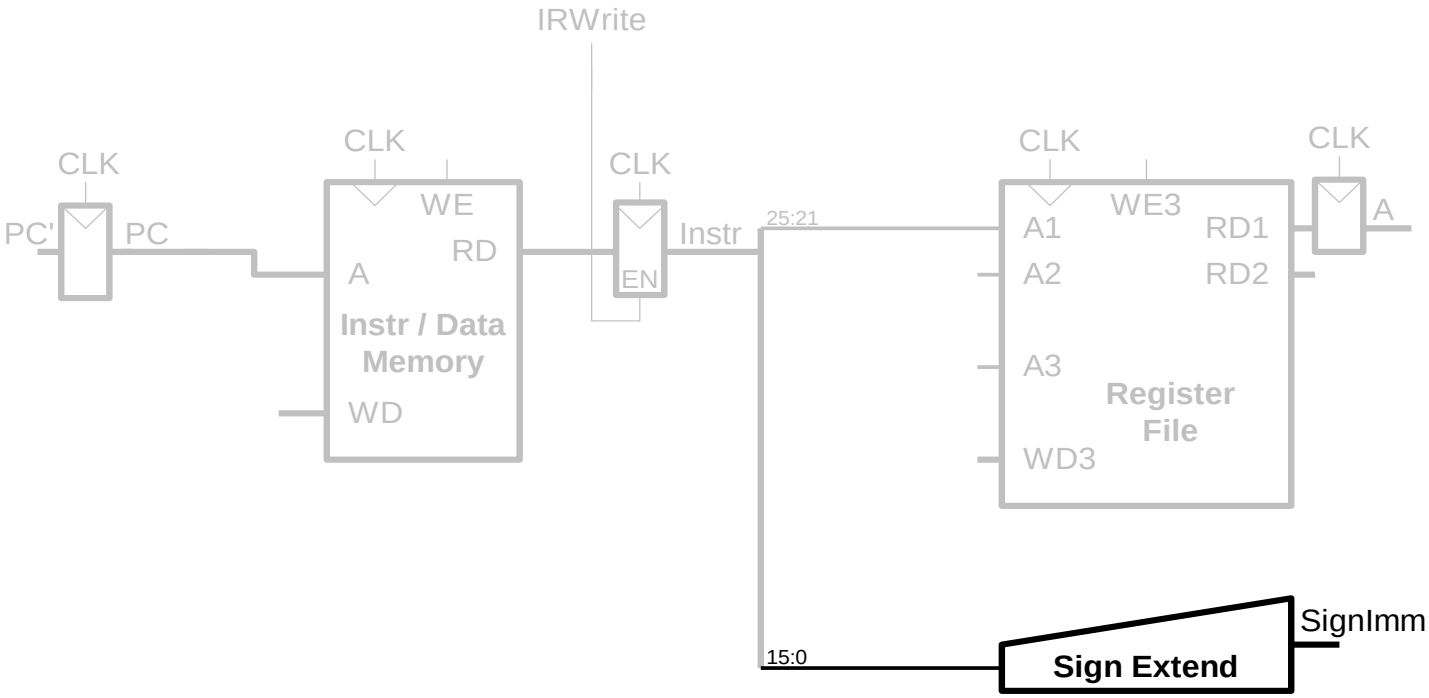
- First consider executing `lw`
- **STEP 1:** Fetch instruction



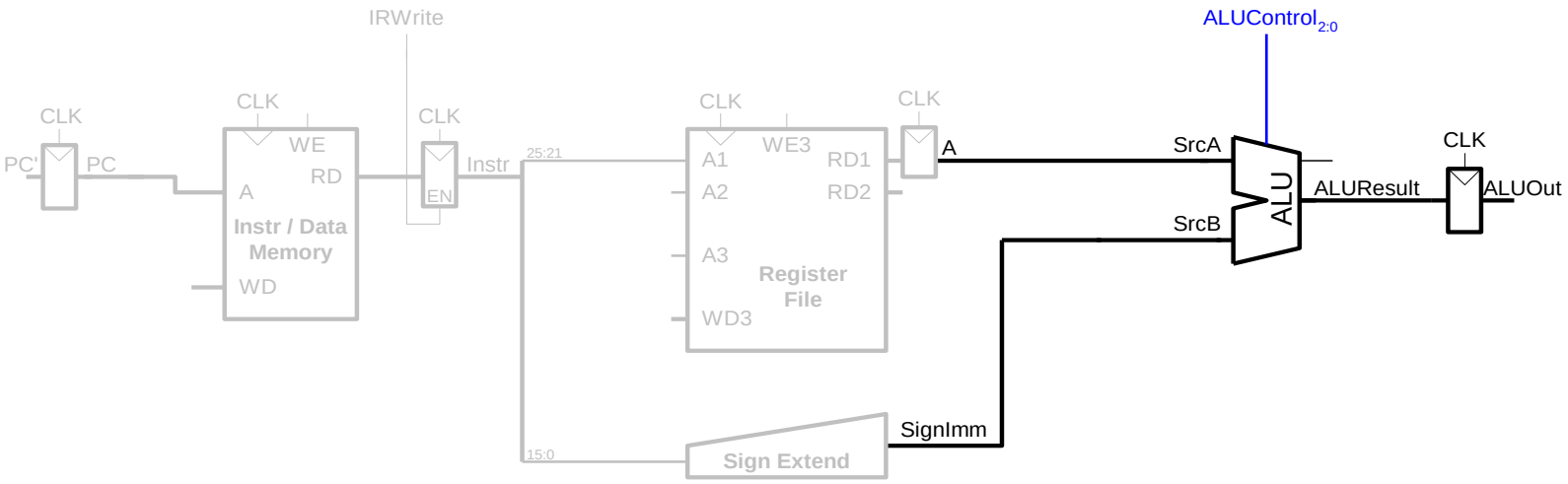
Multicycle Datapath: `lw` register read



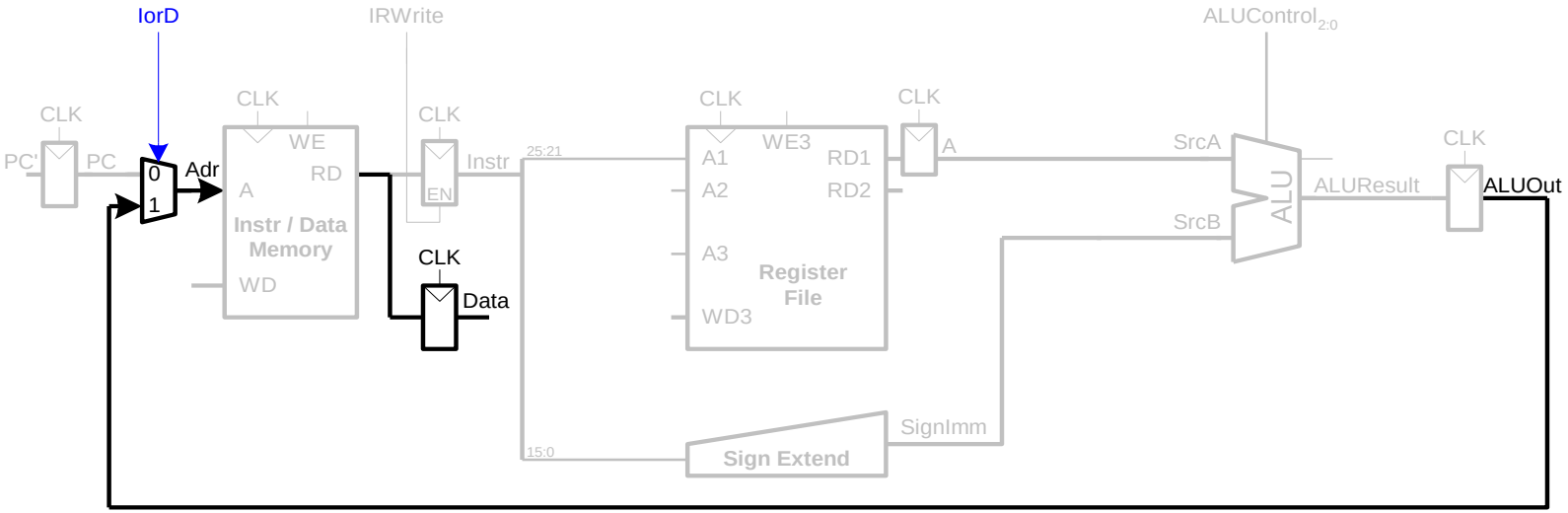
Multicycle Datapath: `lw` immediate



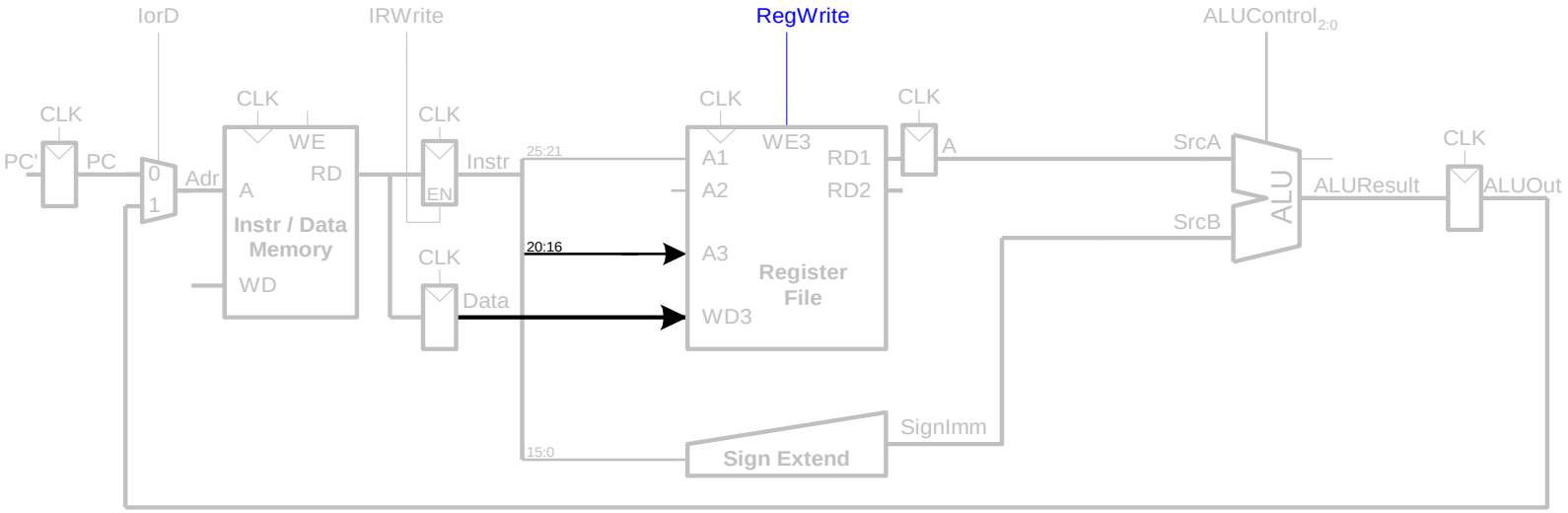
Multicycle Datapath: `lw` address



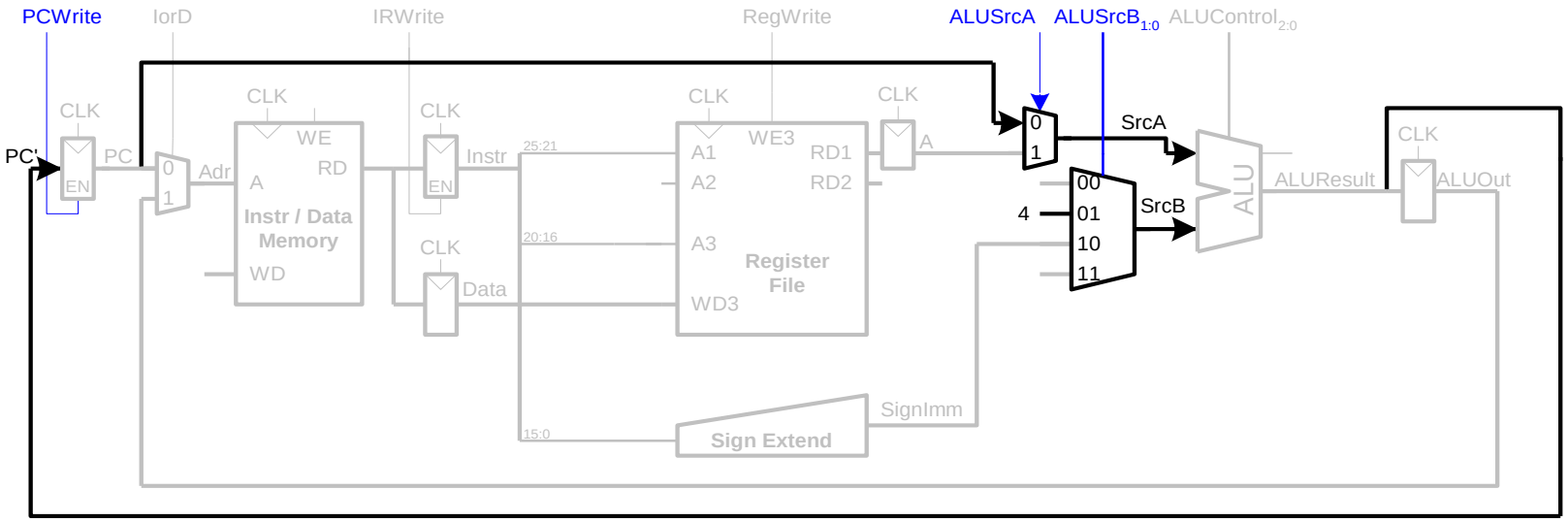
Multicycle Datapath: `lw` memory read



Multicycle Datapath: `lw` write register

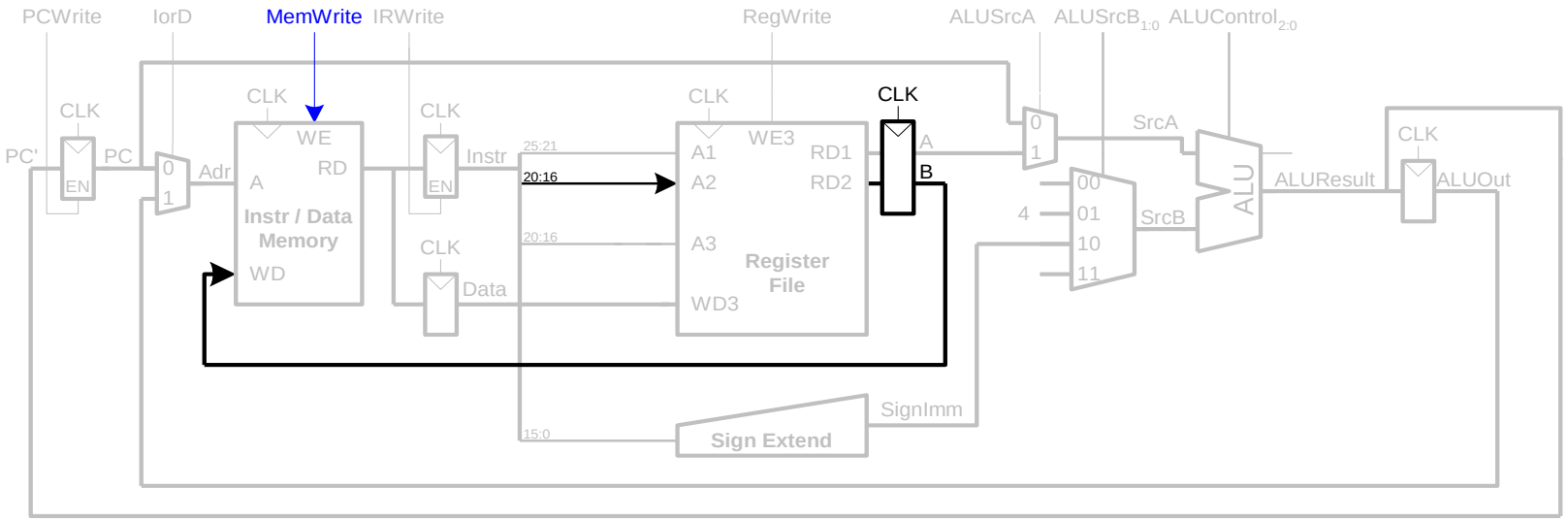


Multicycle Datapath: increment PC



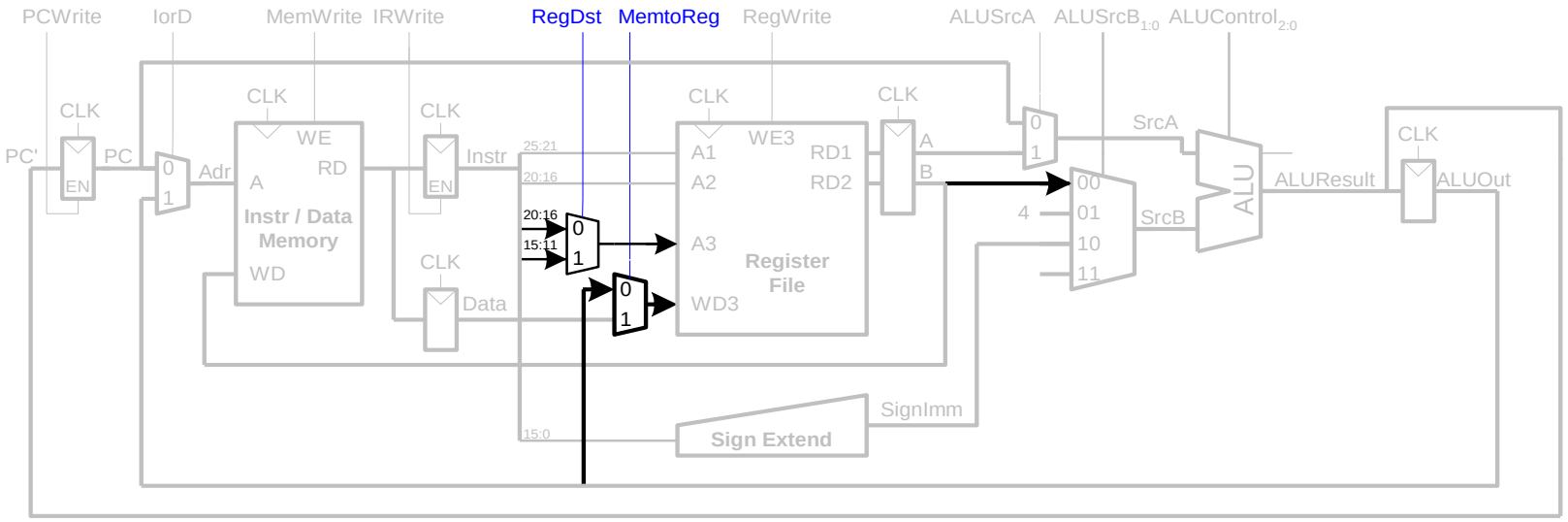
Multicycle Datapath: `sw`

- Write data in `rt` to memory



Multicycle Datapath: R-type Instructions

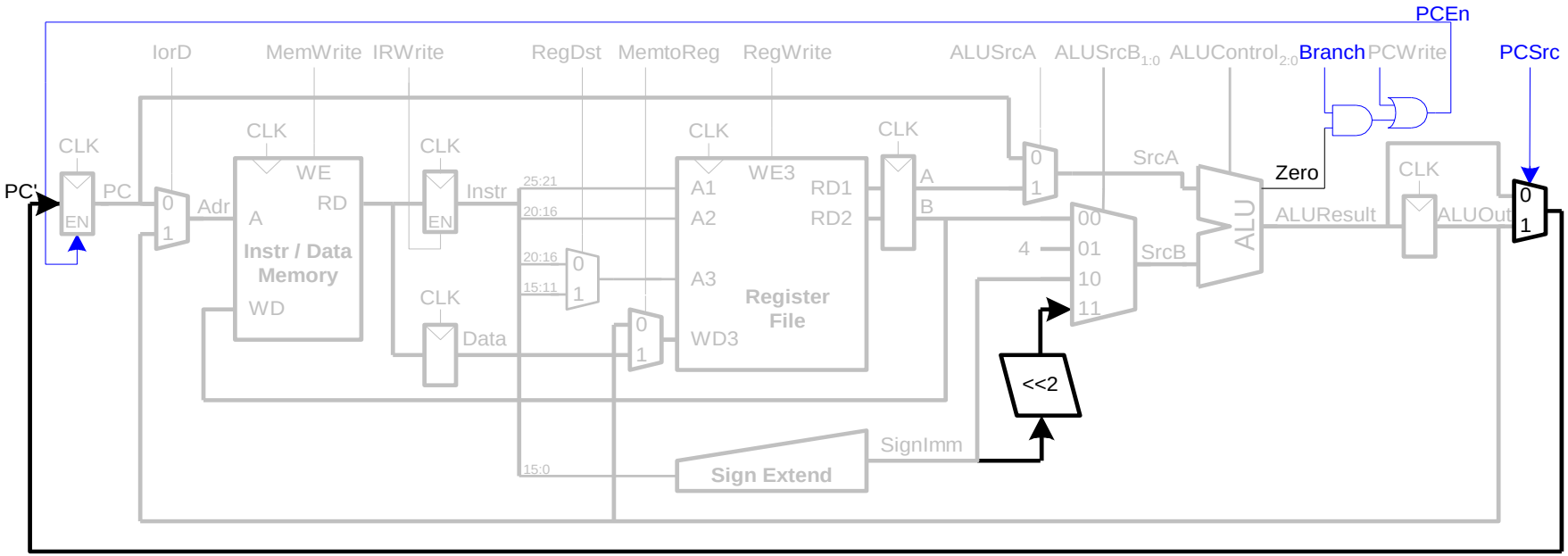
- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)



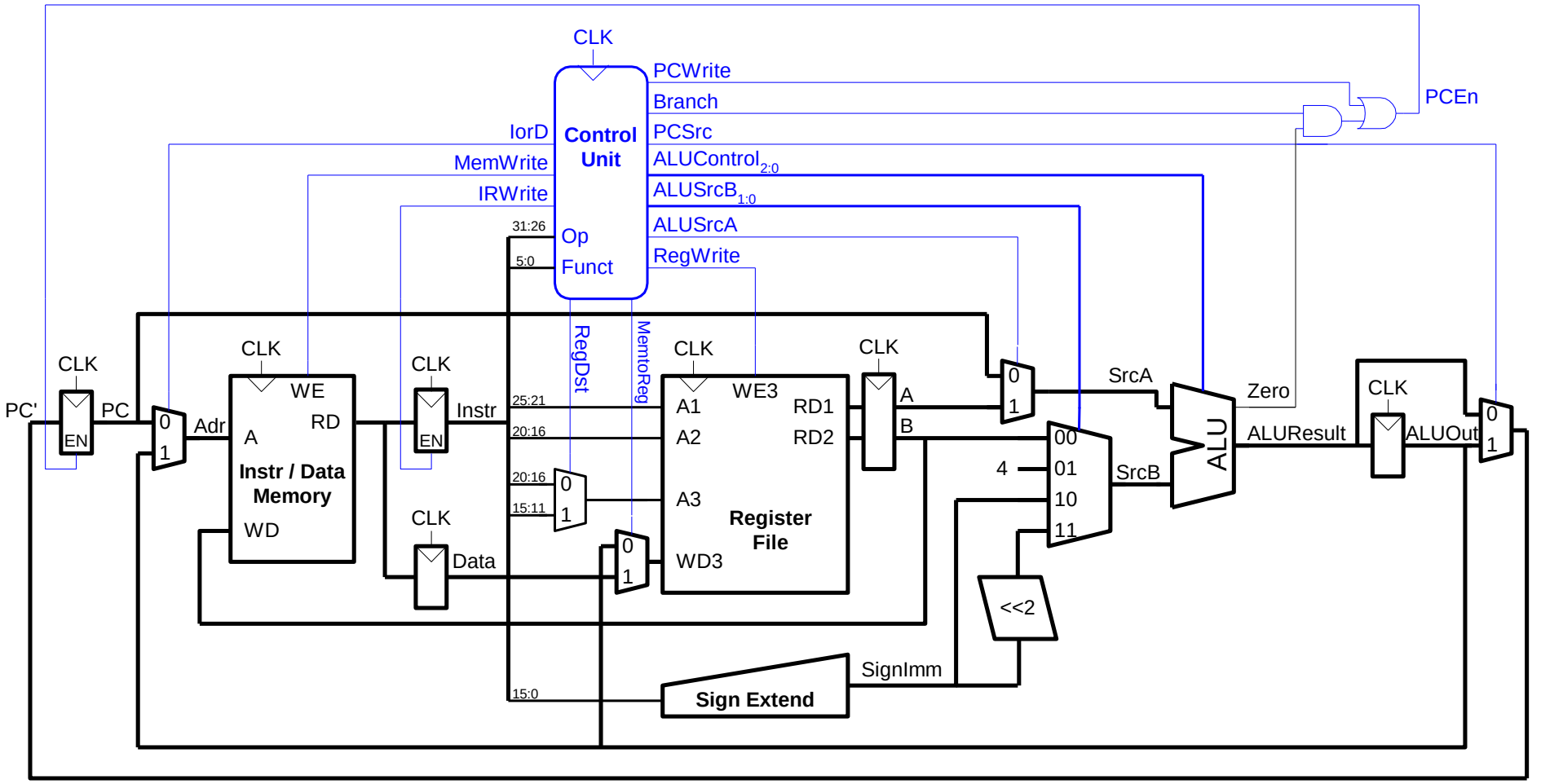
Multicycle Datapath: `beq`

- Determine whether values in `rs` and `rt` are equal
- Calculate branch target address:

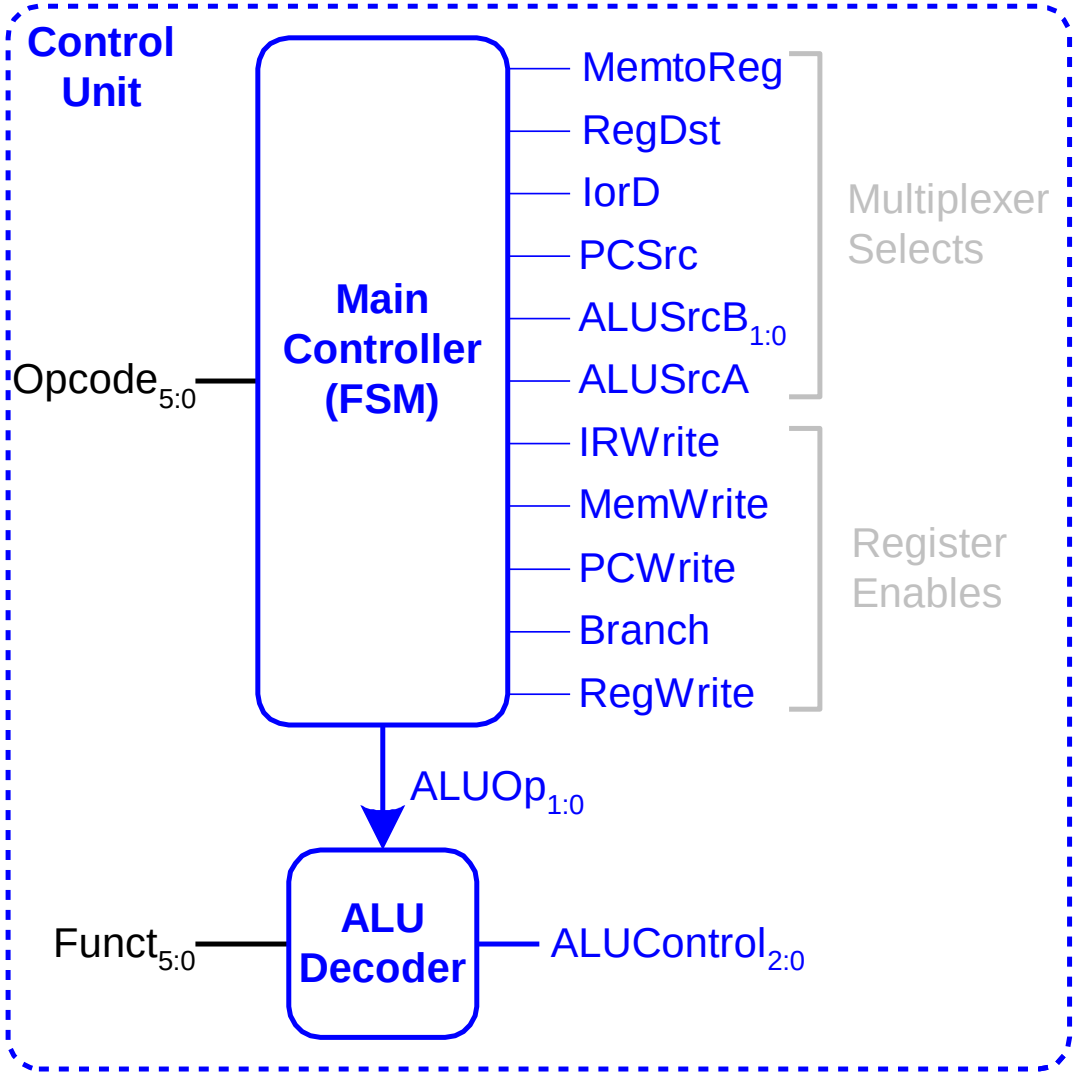
  BTA = (sign-extended immediate << 2) + (PC+4)



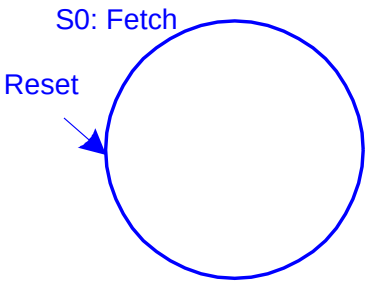
Complete Multicycle Processor

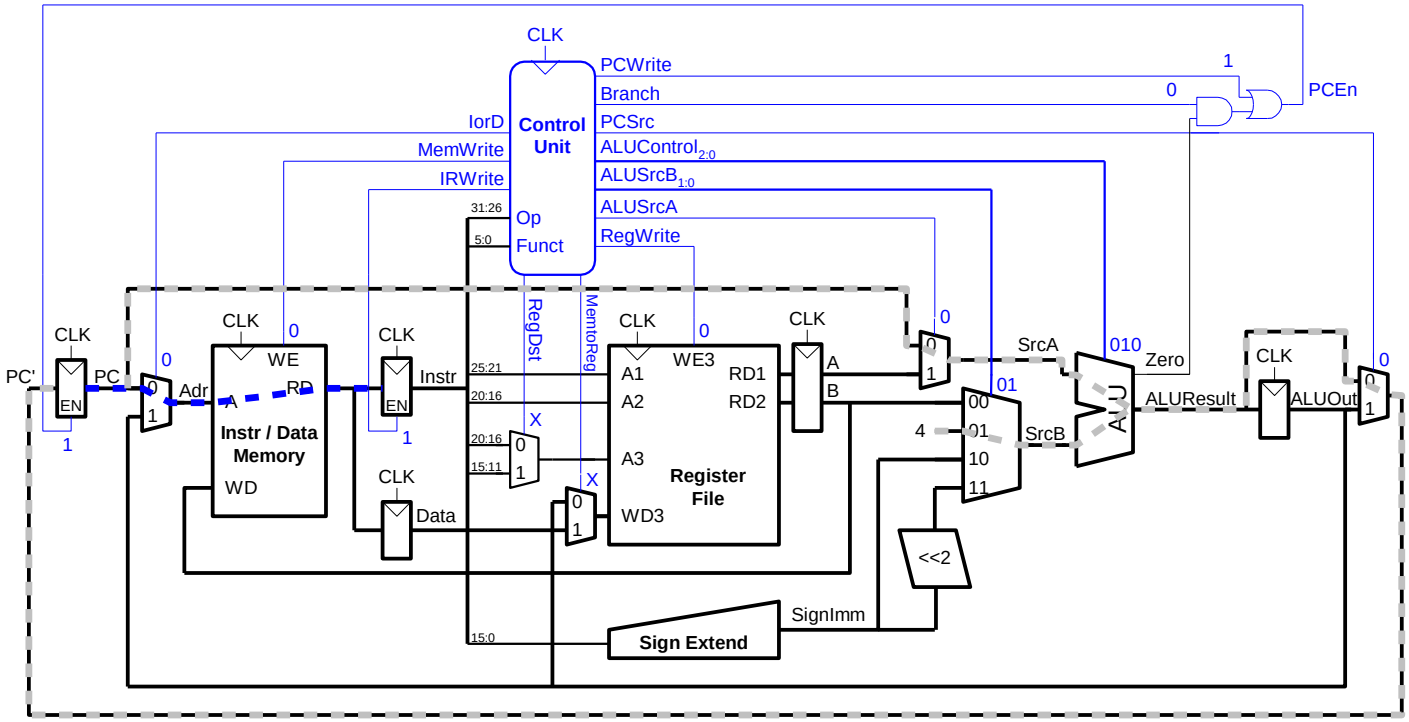


Control Unit

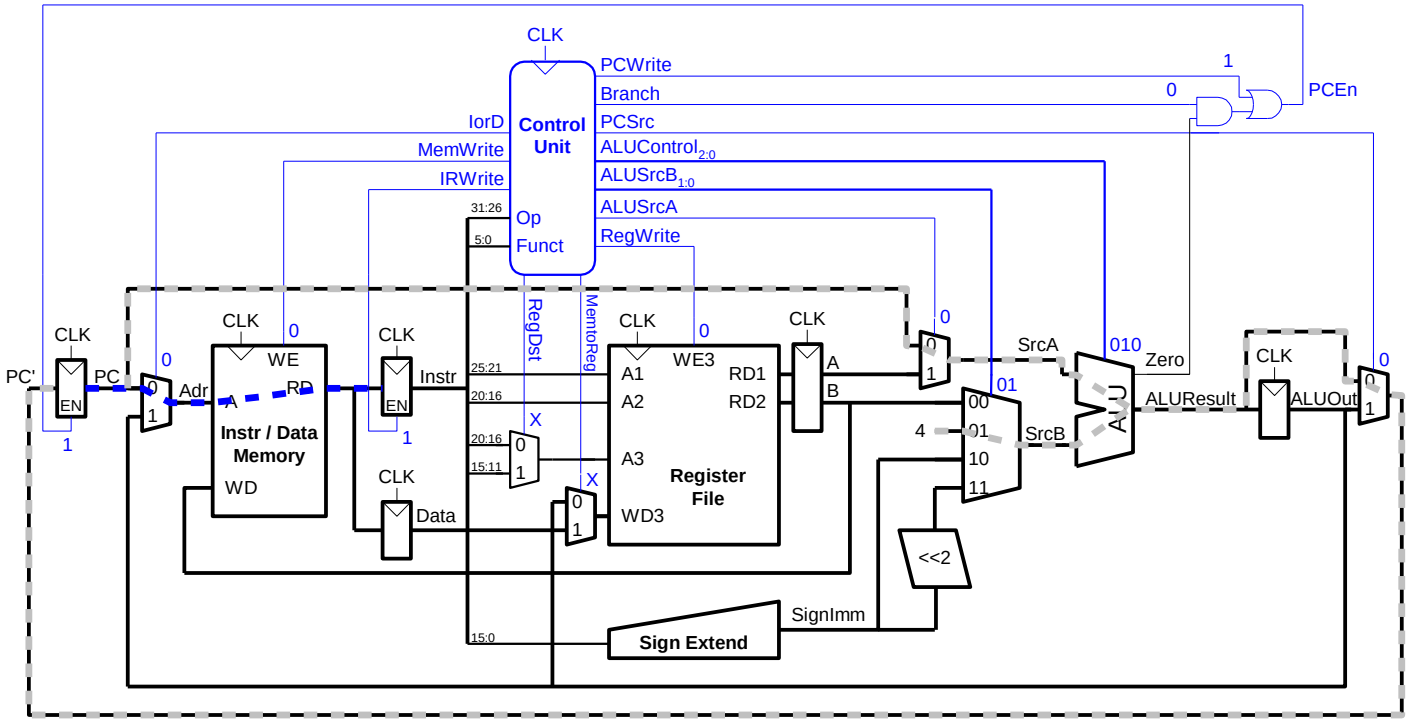


Main Controller FSM: Fetch

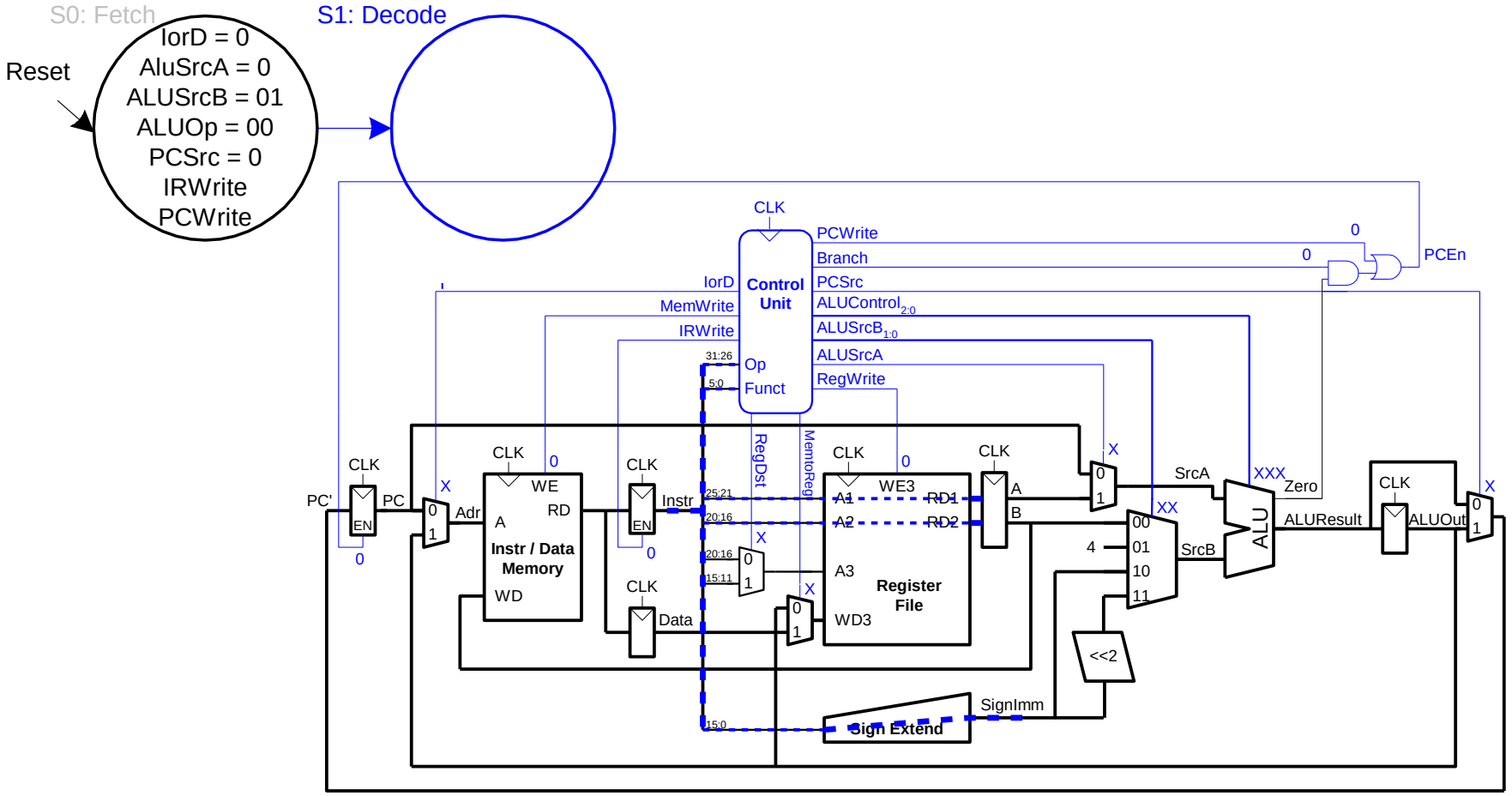




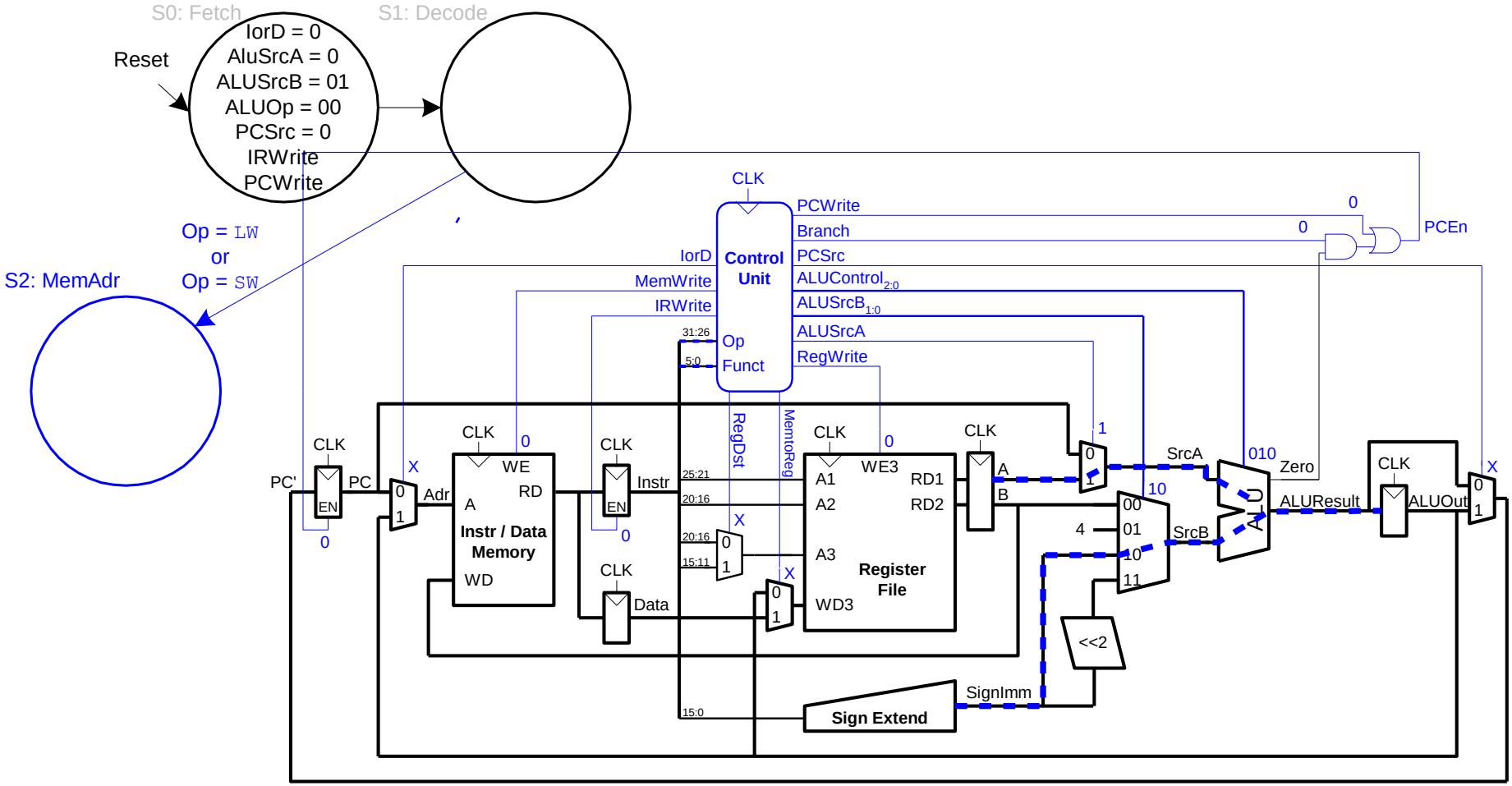
Main Controller FSM: Fetch



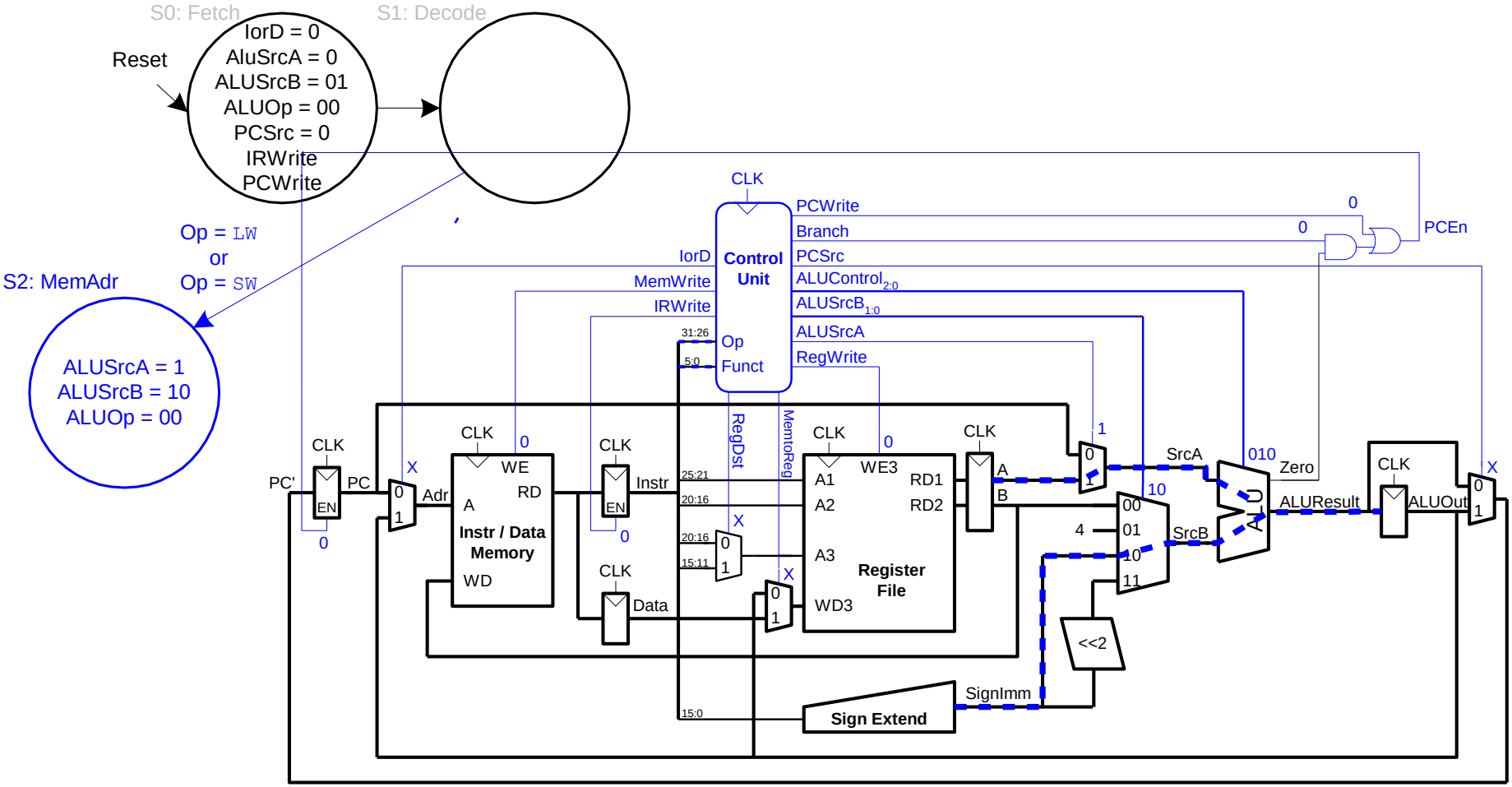
Main Controller FSM: Decode



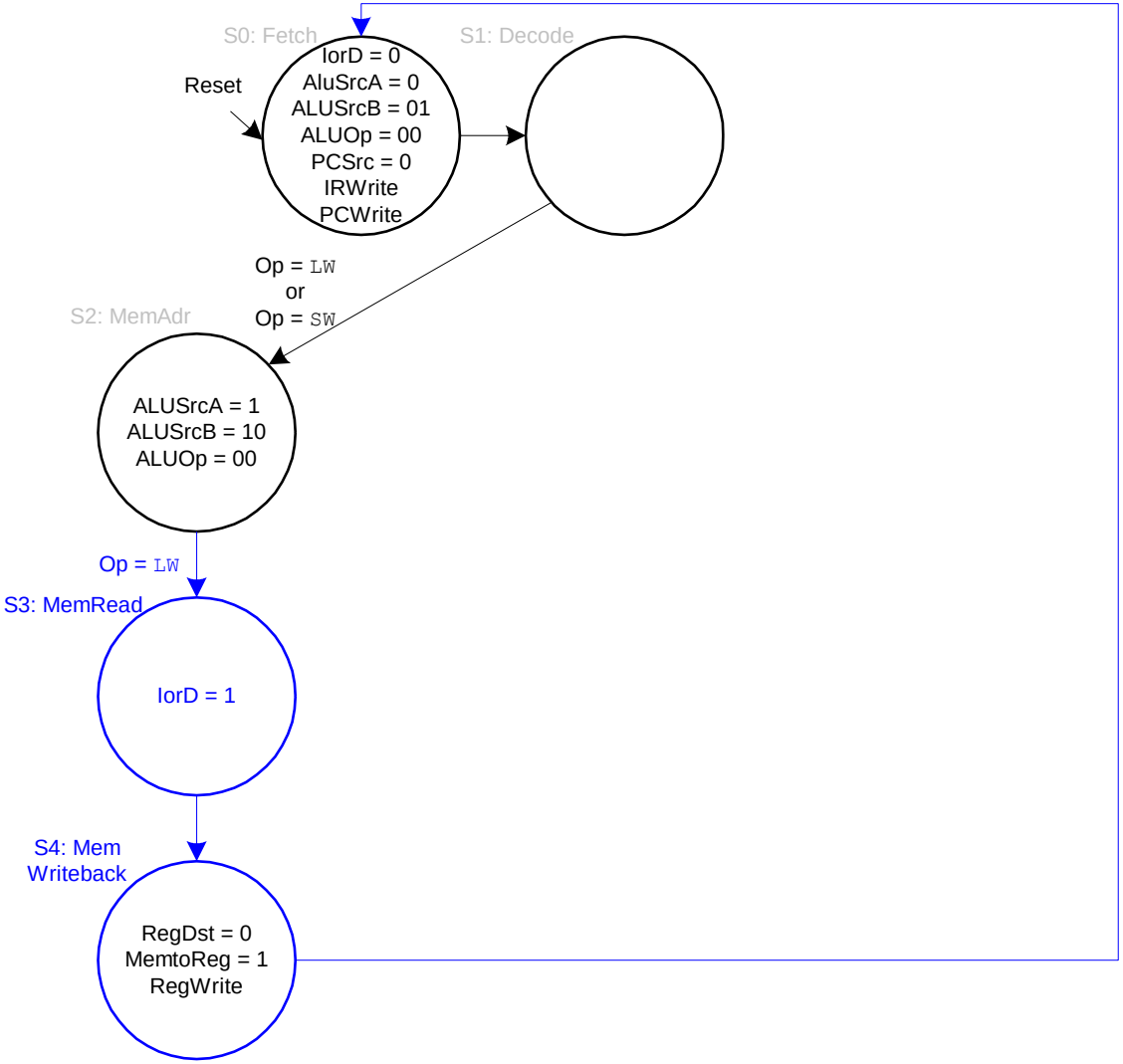
Main Controller FSM: Address Calculation



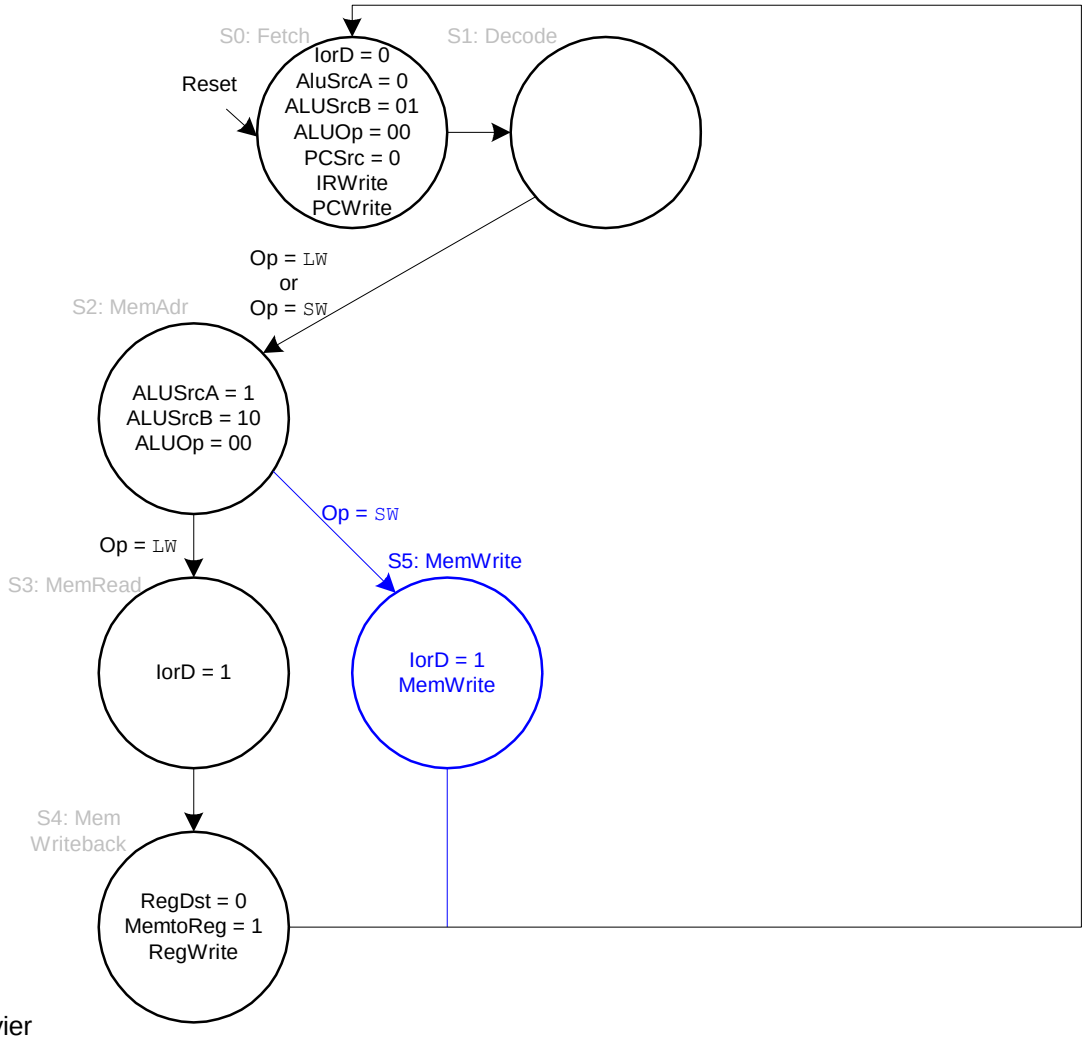
Main Controller FSM: Address Calculation



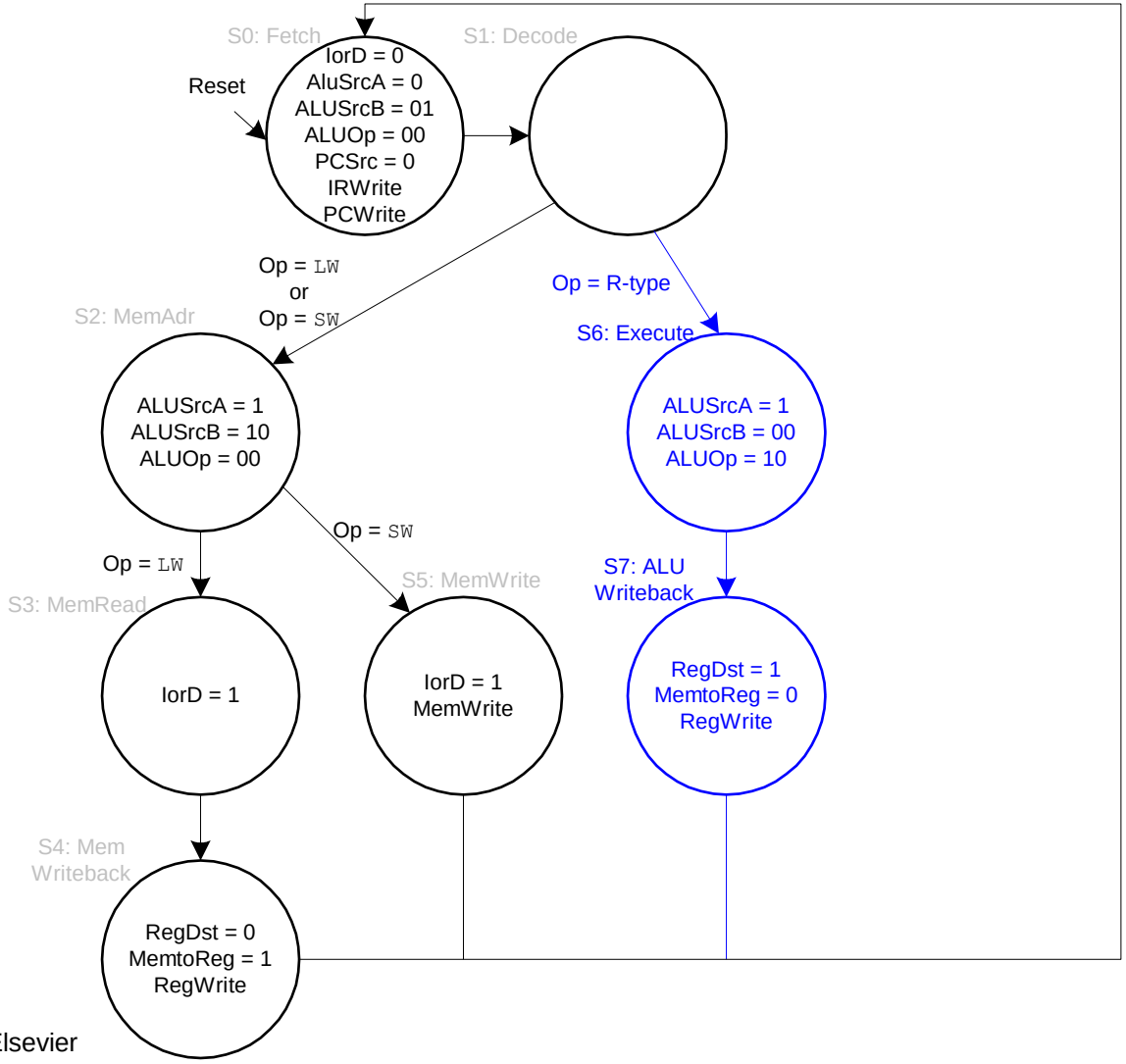
Main Controller FSM: lw



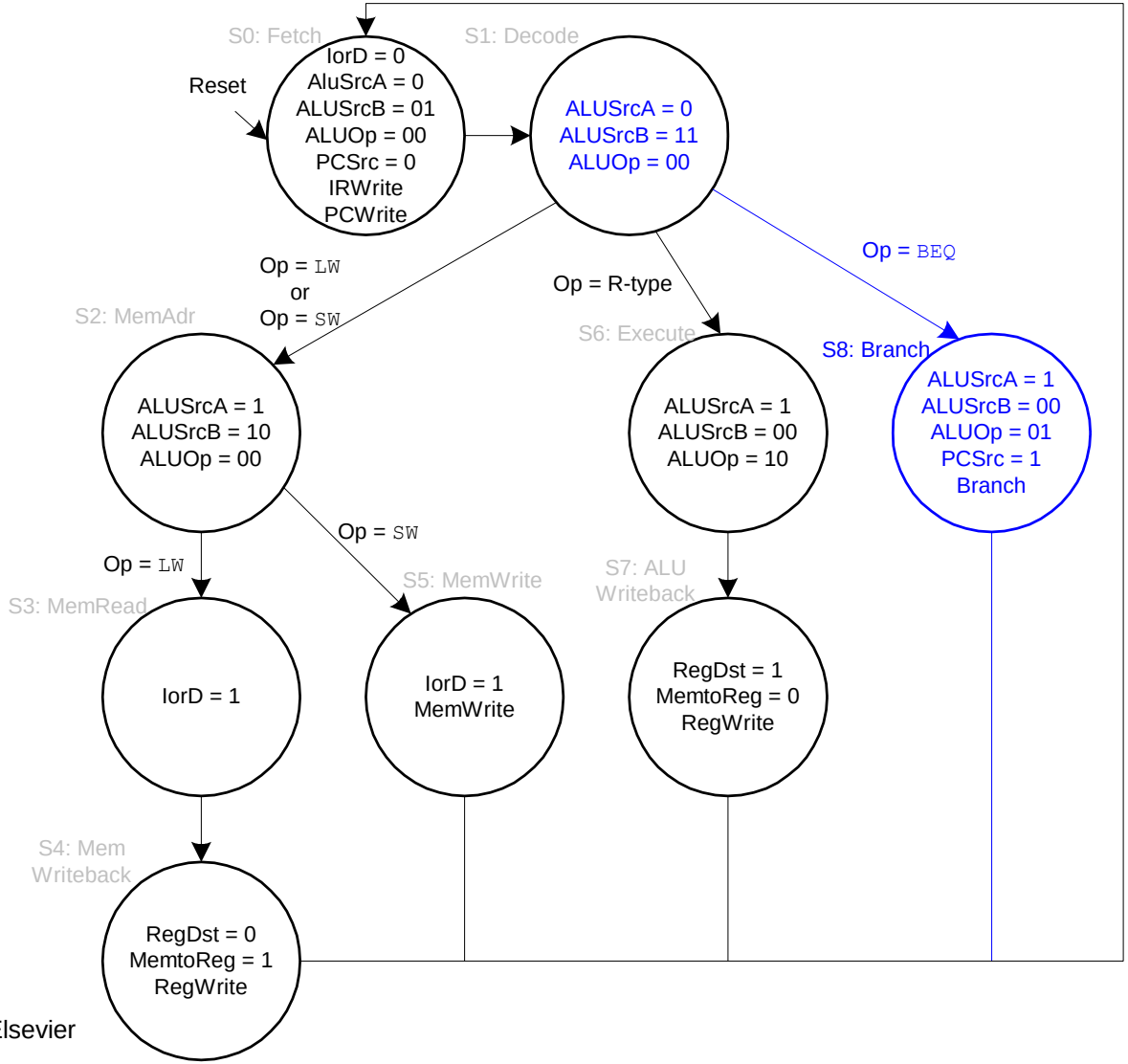
Main Controller FSM: SW



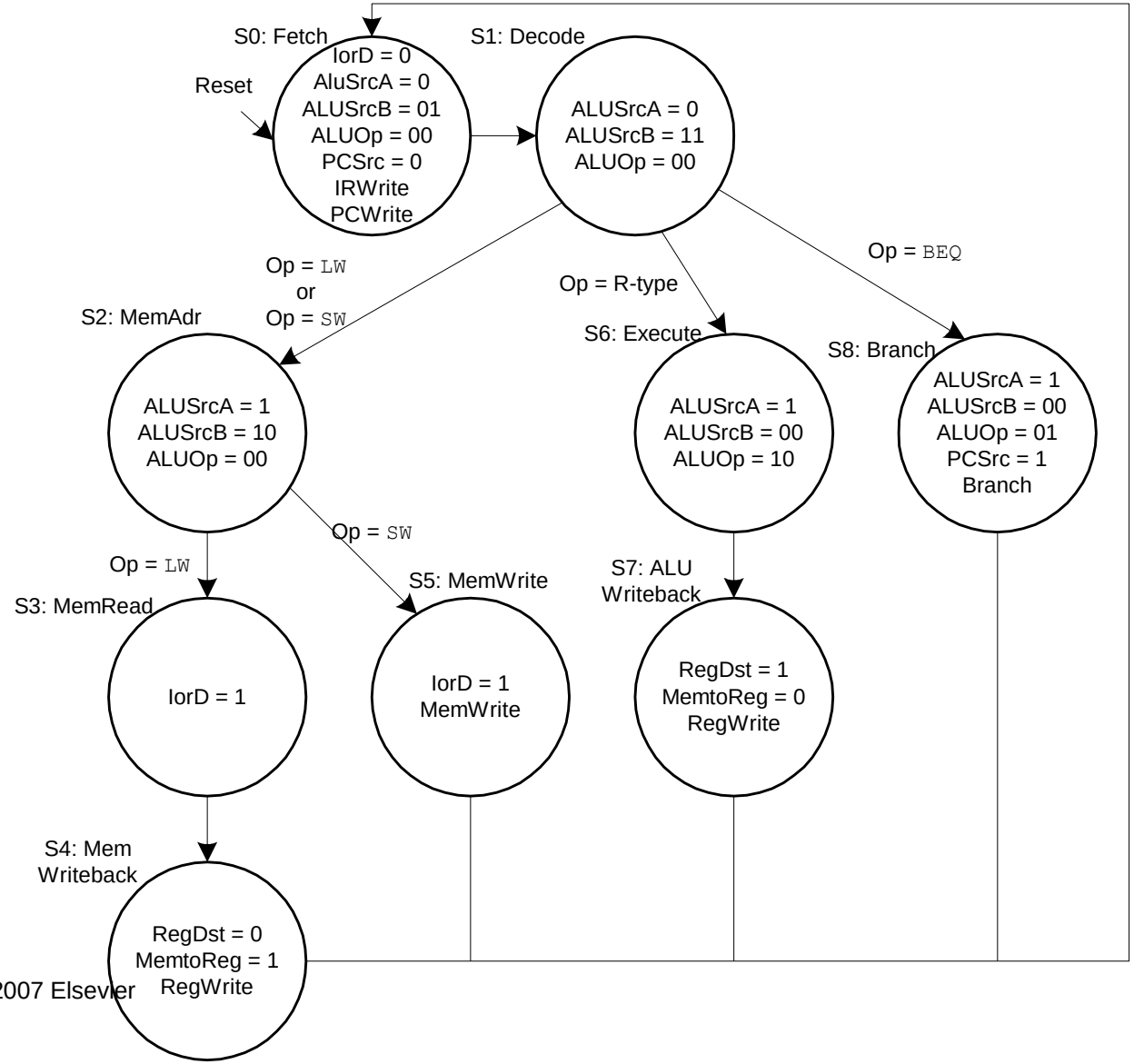
Main Controller FSM: R-Type



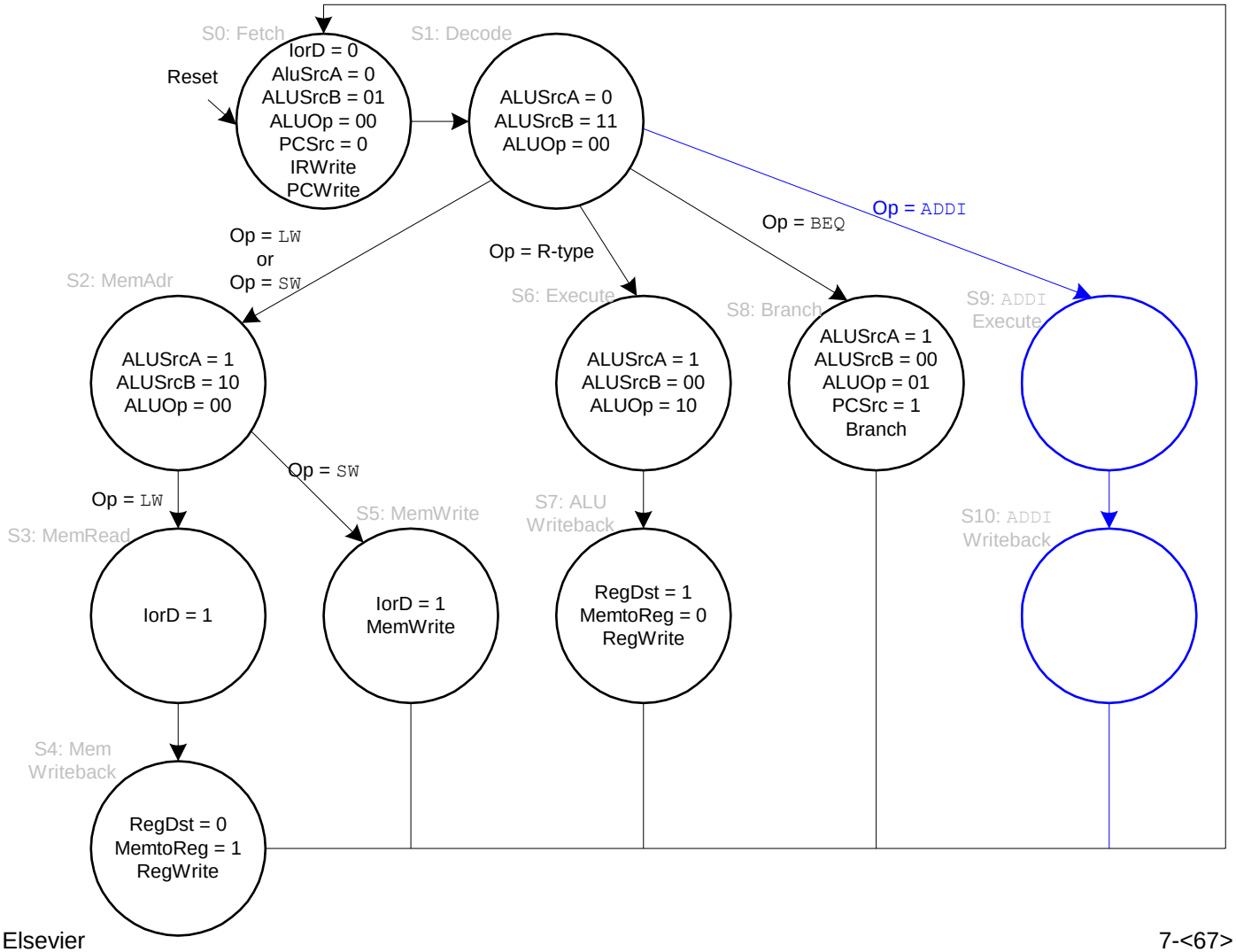
Main Controller FSM: beq



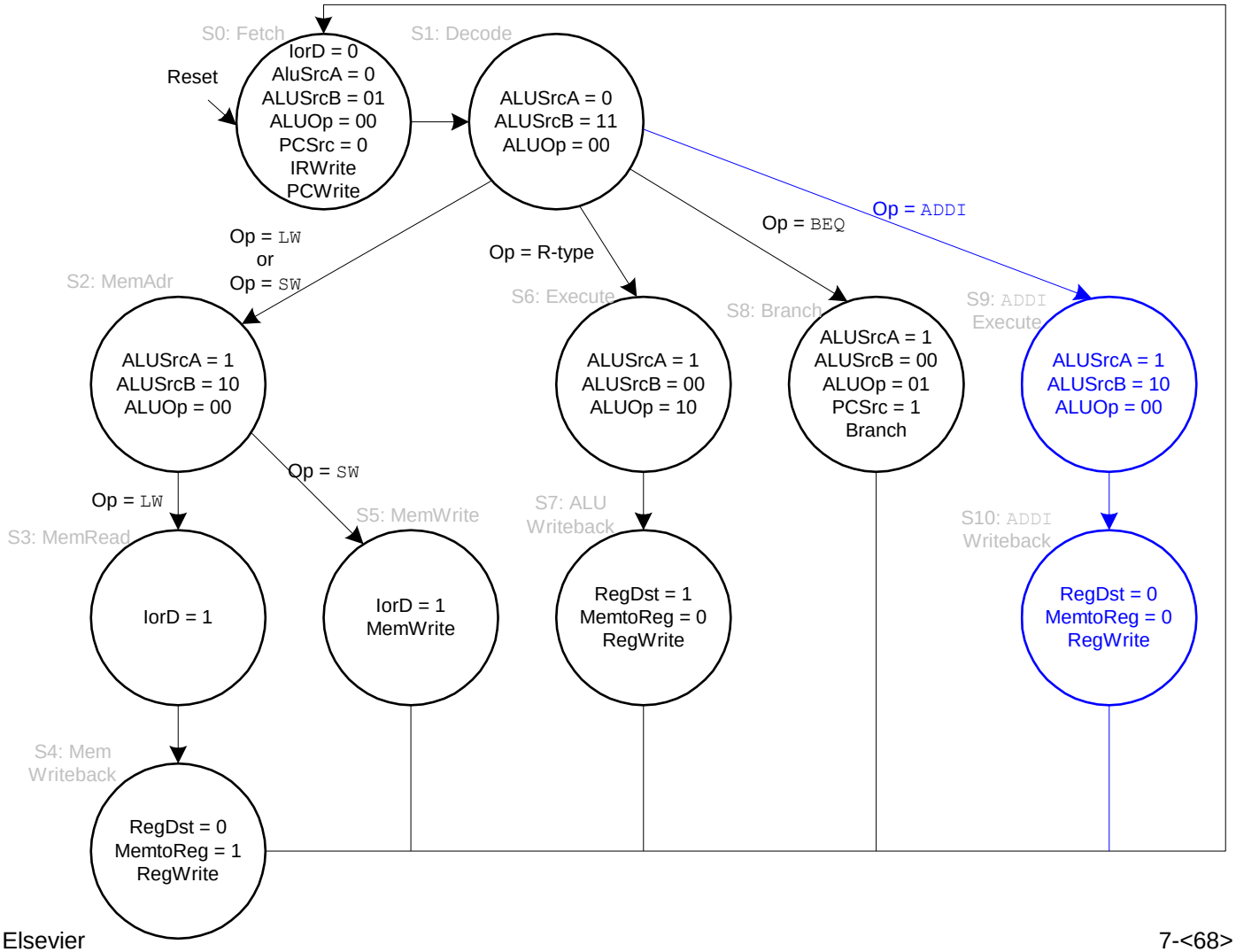
Complete Multicycle Controller FSM



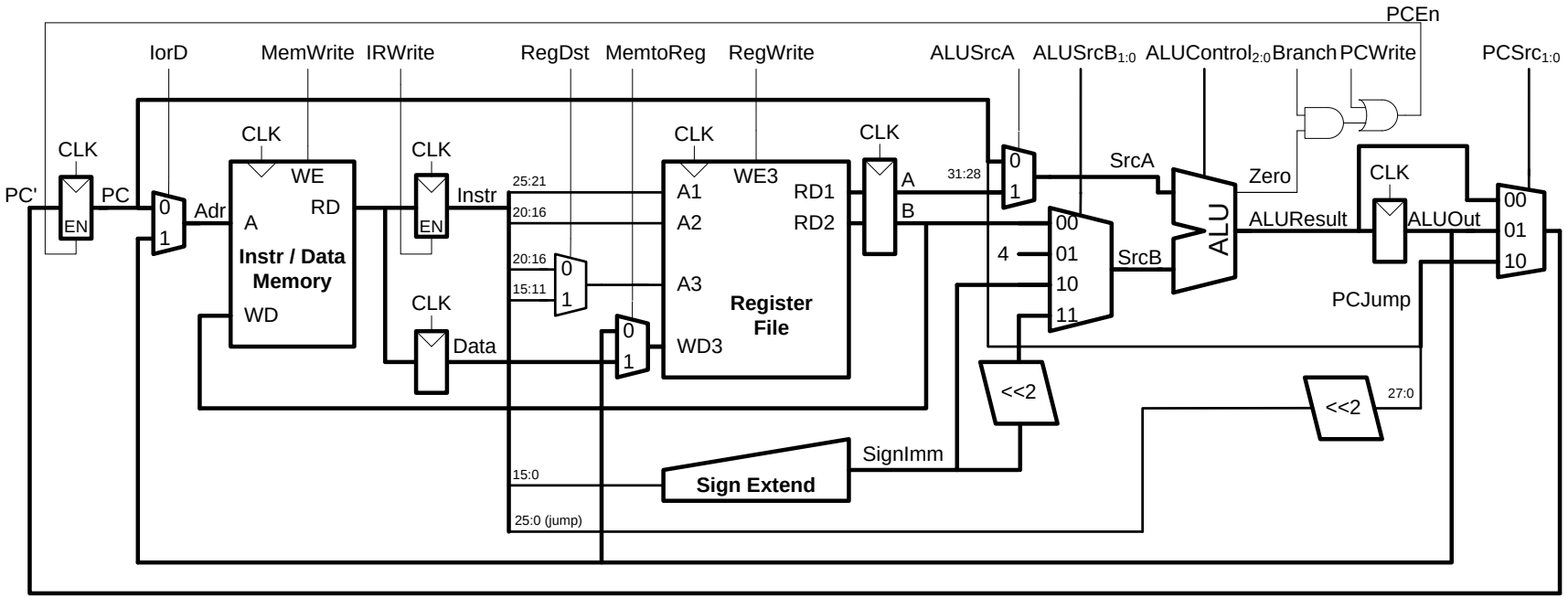
Main Controller FSM: addi



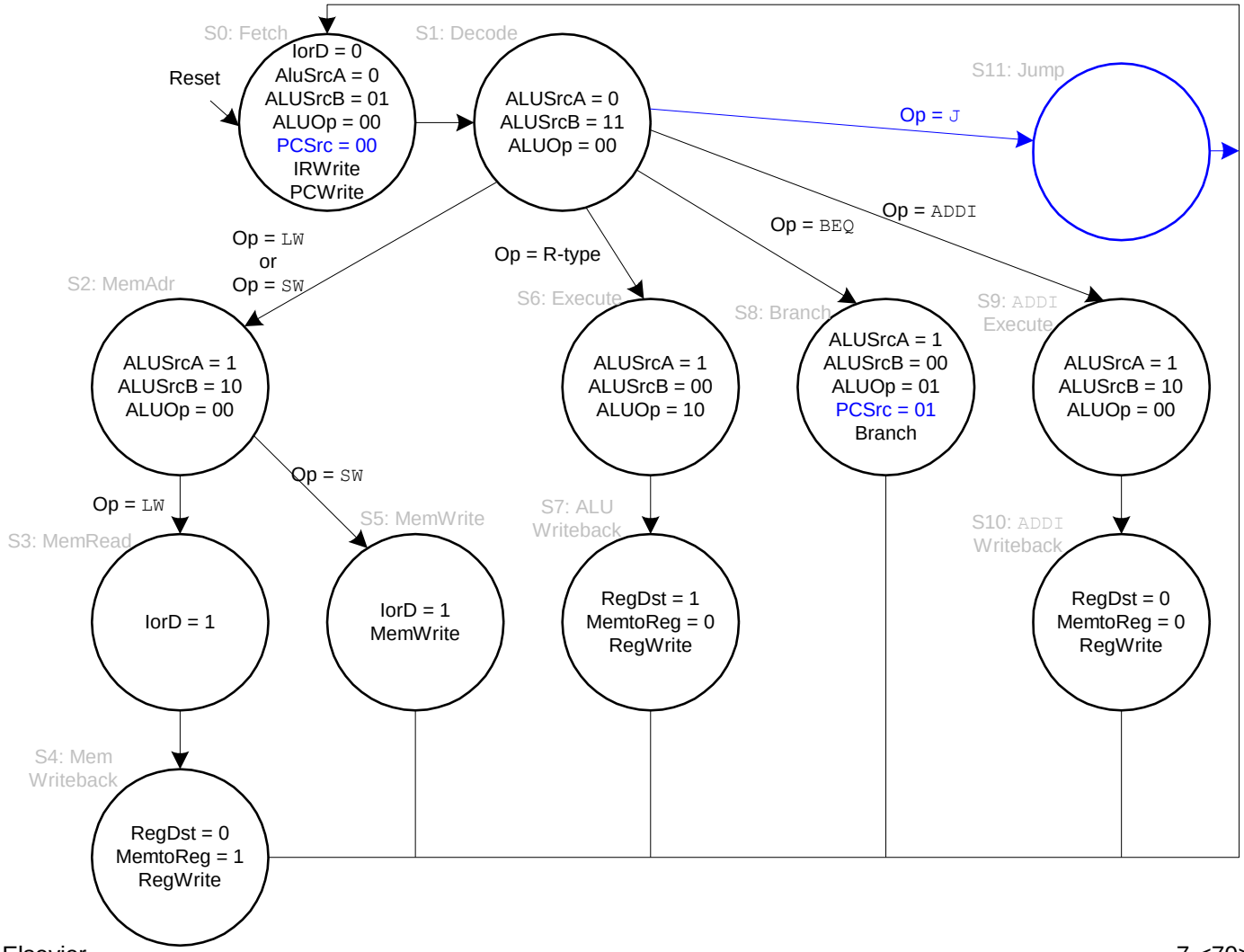
Main Controller FSM: addi



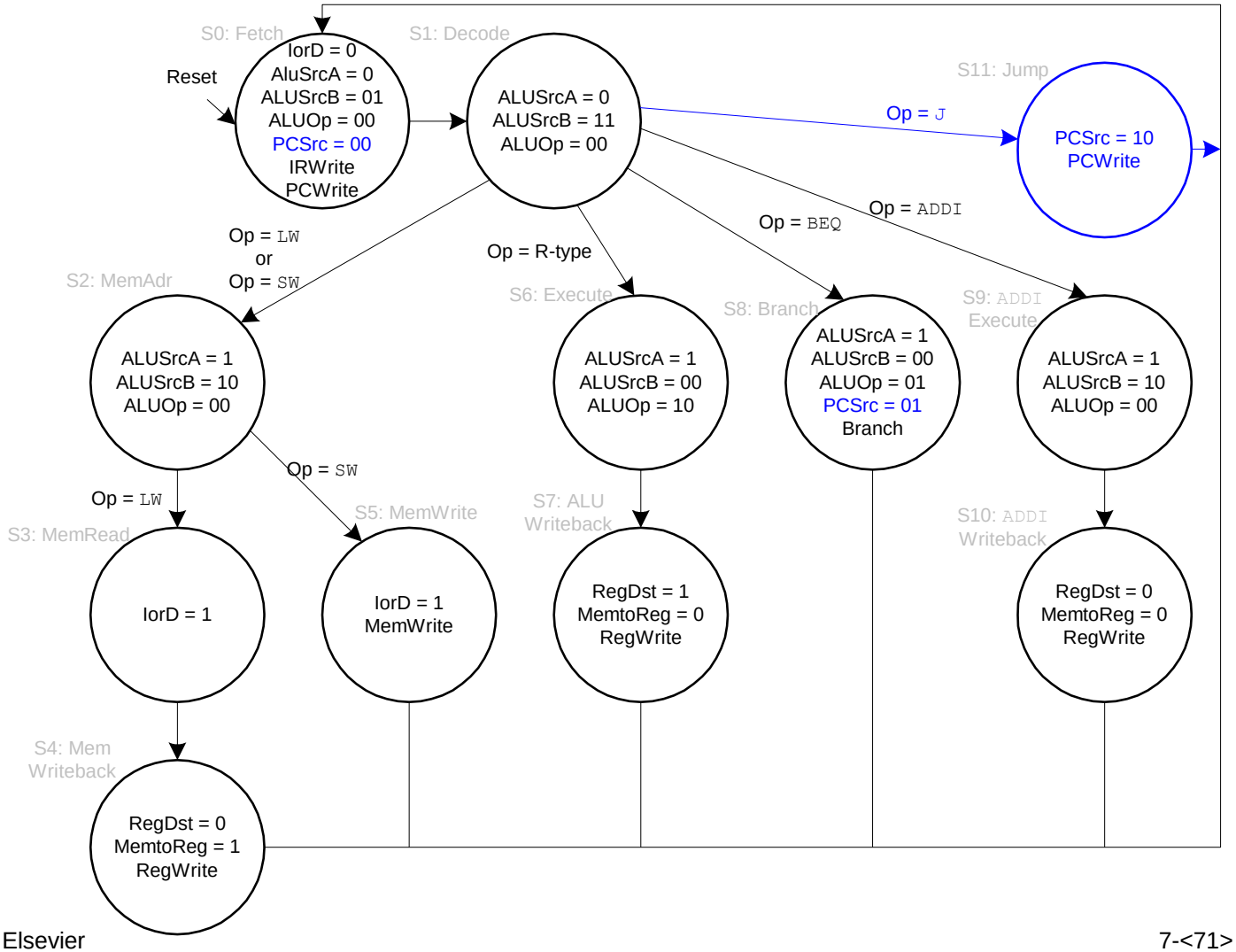
Extended Functionality: j



Control FSM: j



Control FSM: j



Multicycle Performance

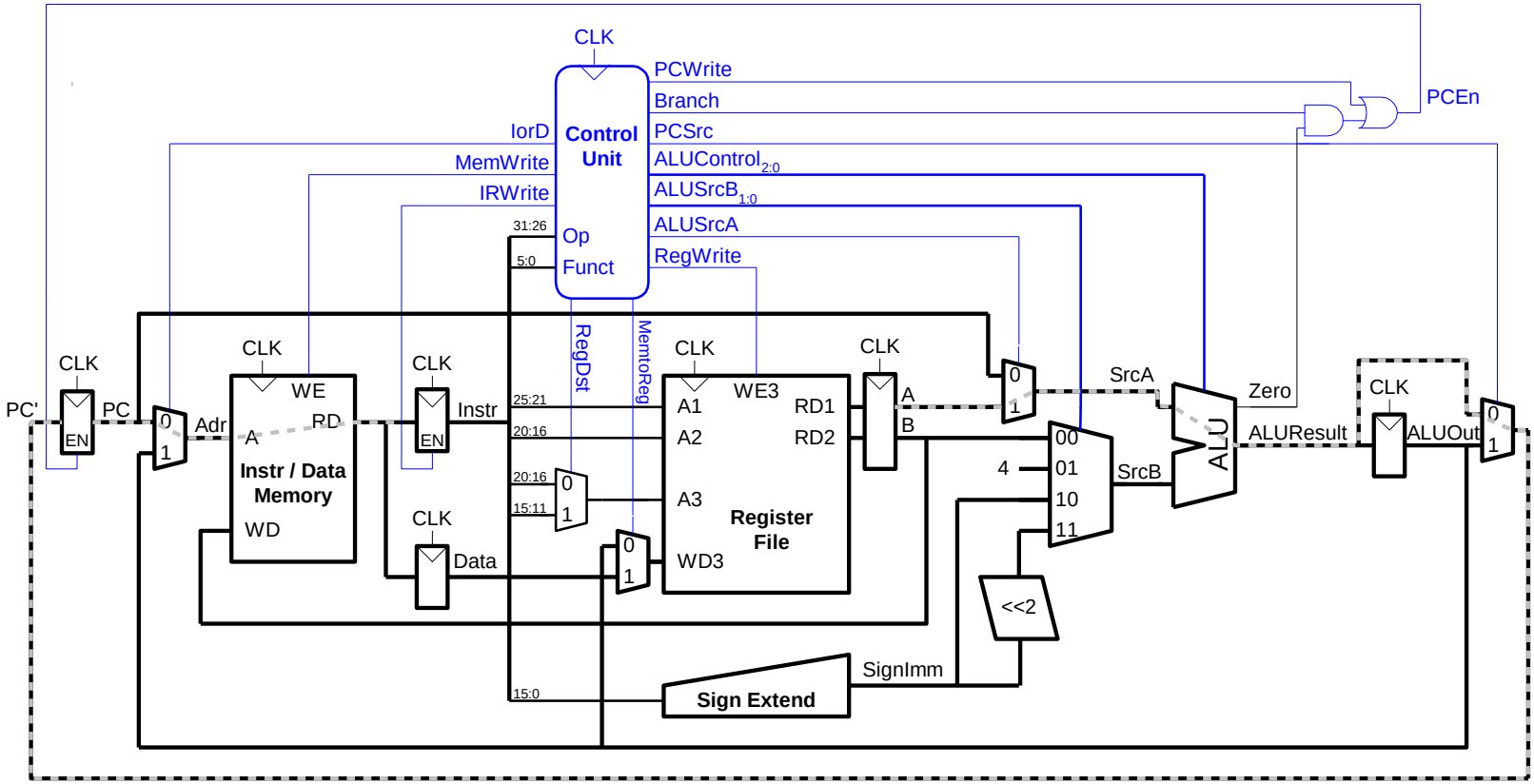
- Instructions take different number of cycles:
  - 3 cycles: `beq`, `j`
  - 4 cycles: R-Type, `sw`, `addi`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

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

Multicycle Performance

- Multicycle critical path:

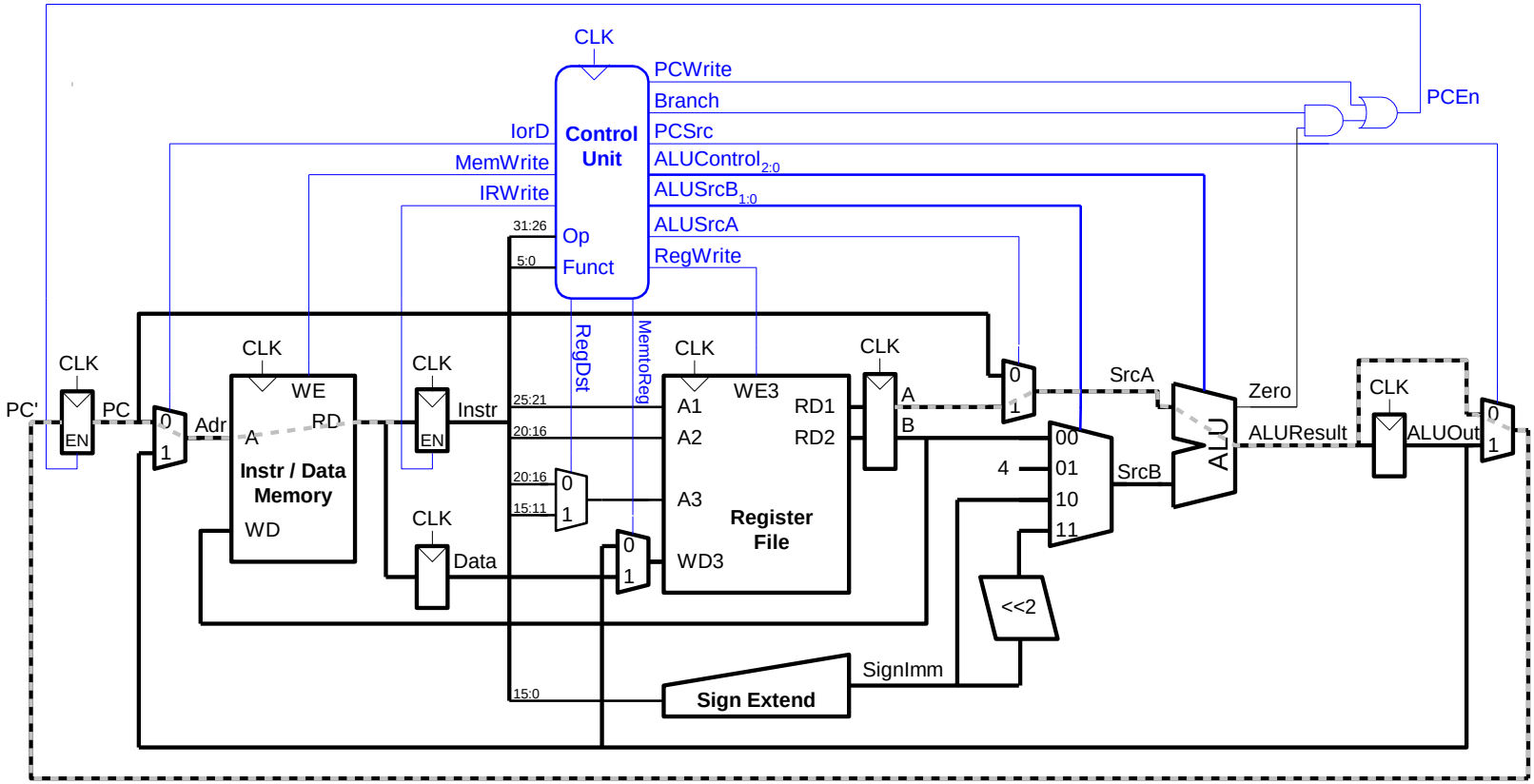
  T_c =



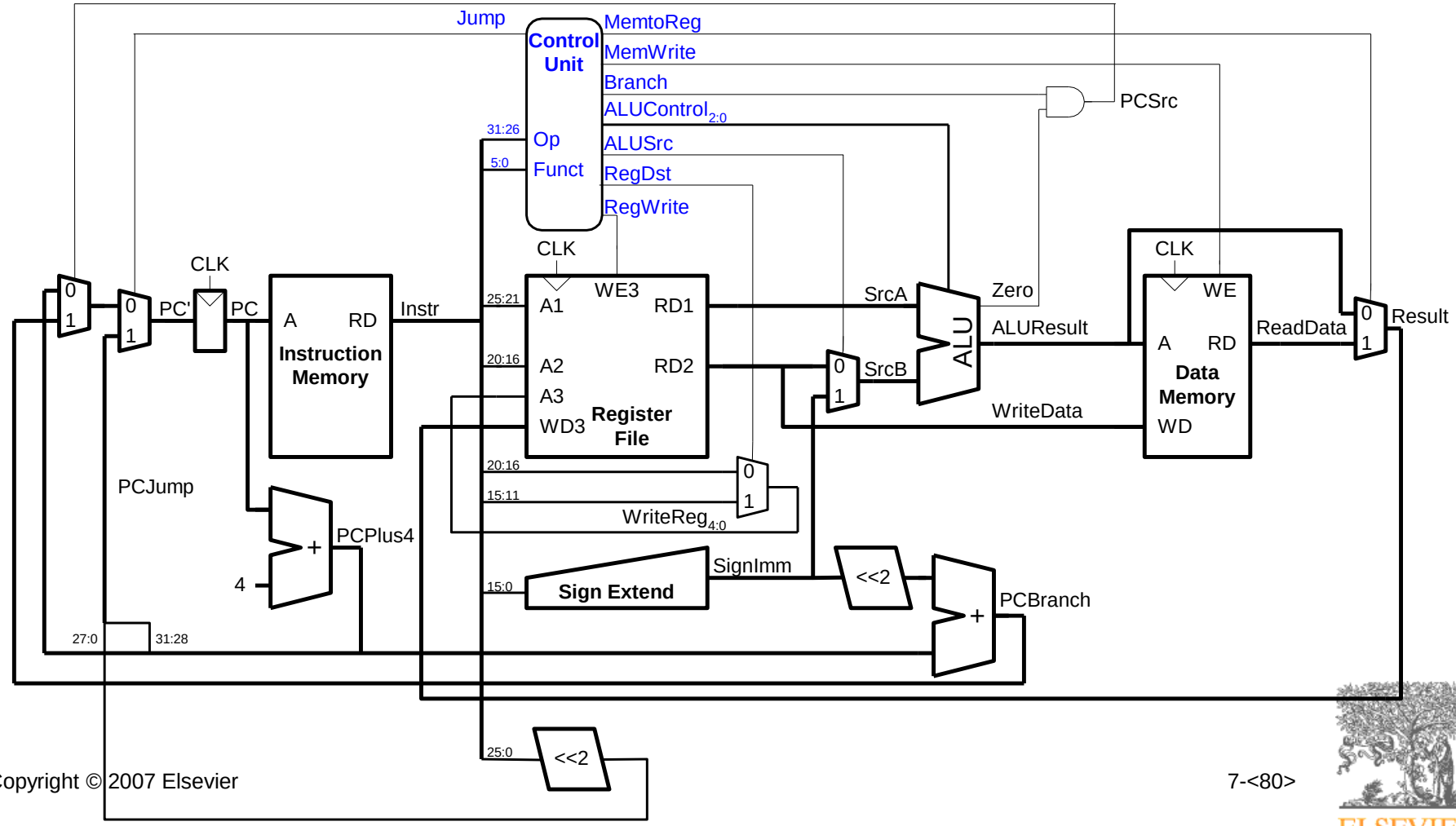
Multicycle Performance

- Multicycle critical path:

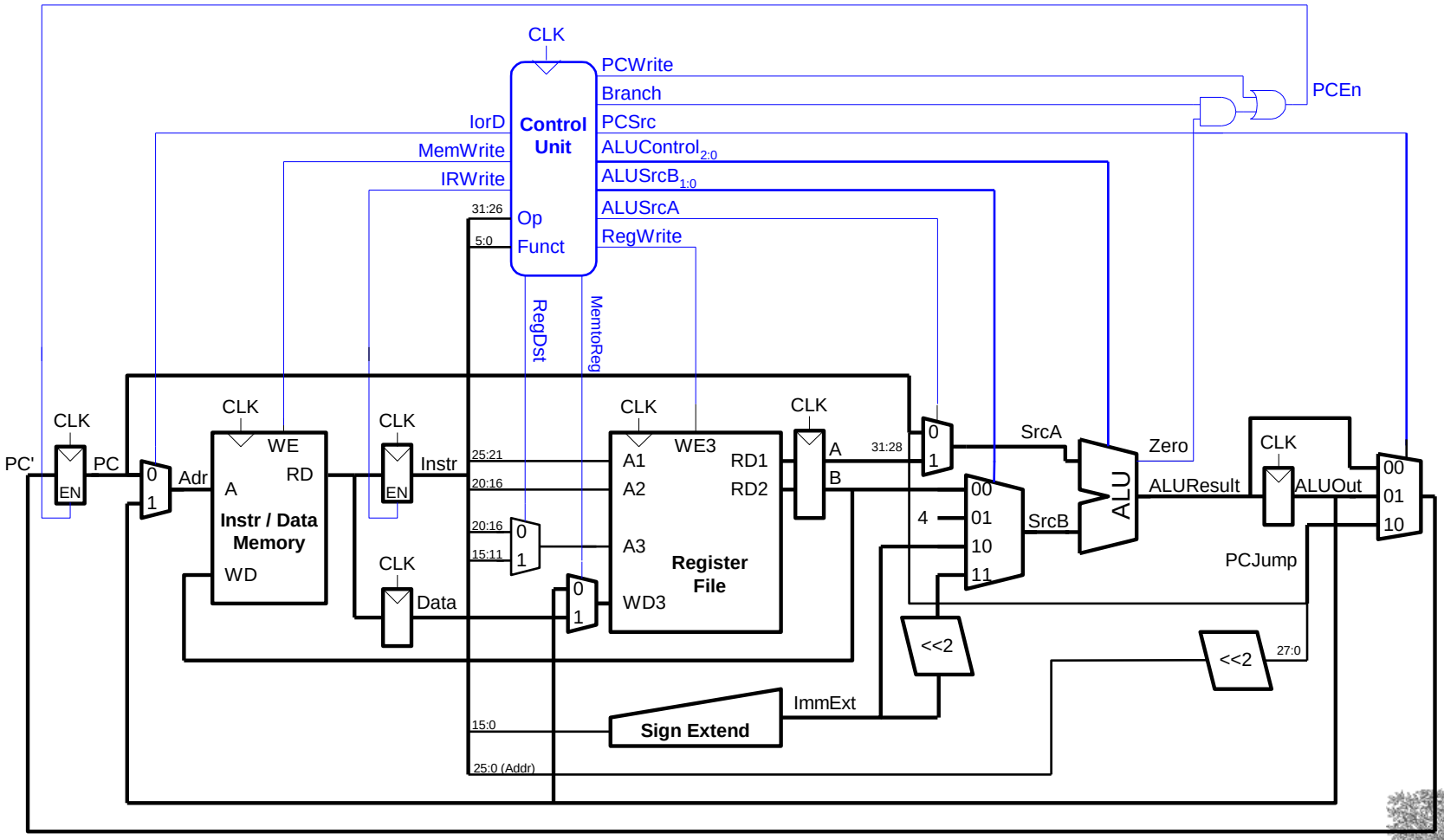
$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



Review: Single-Cycle MIPS Processor



Review: Multicycle MIPS Processor

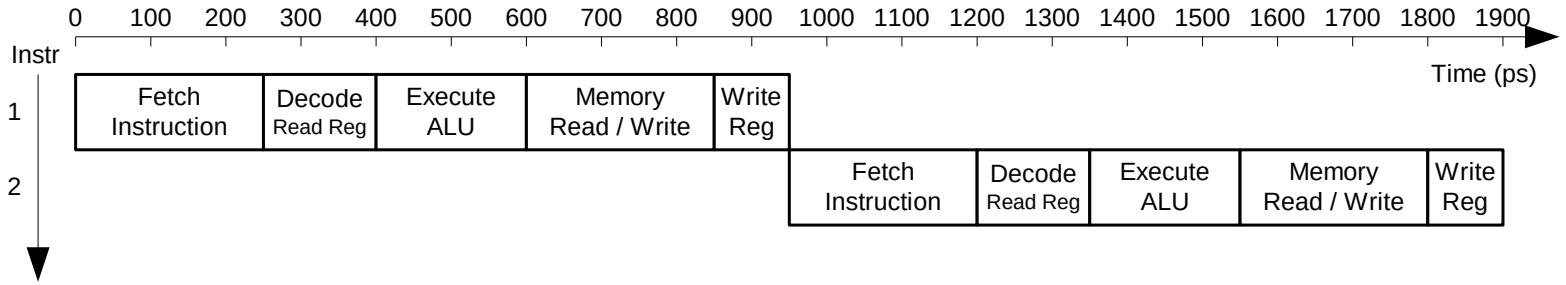


Pipelined MIPS Processor

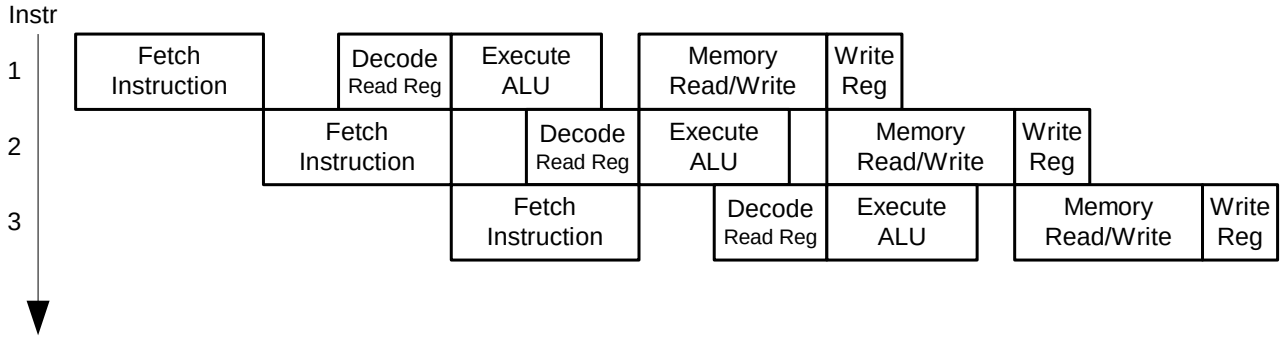
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Single-Cycle vs. Pipelined Performance

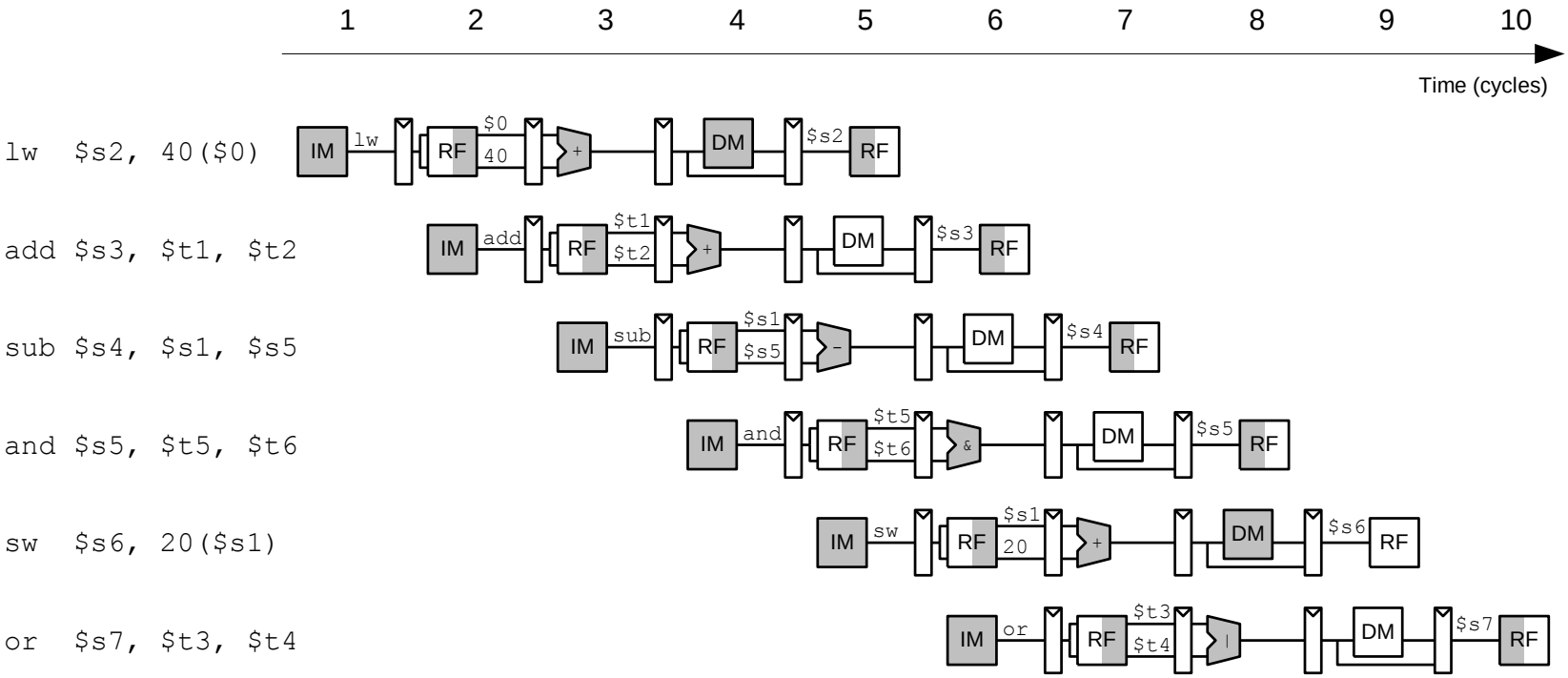
Single-Cycle



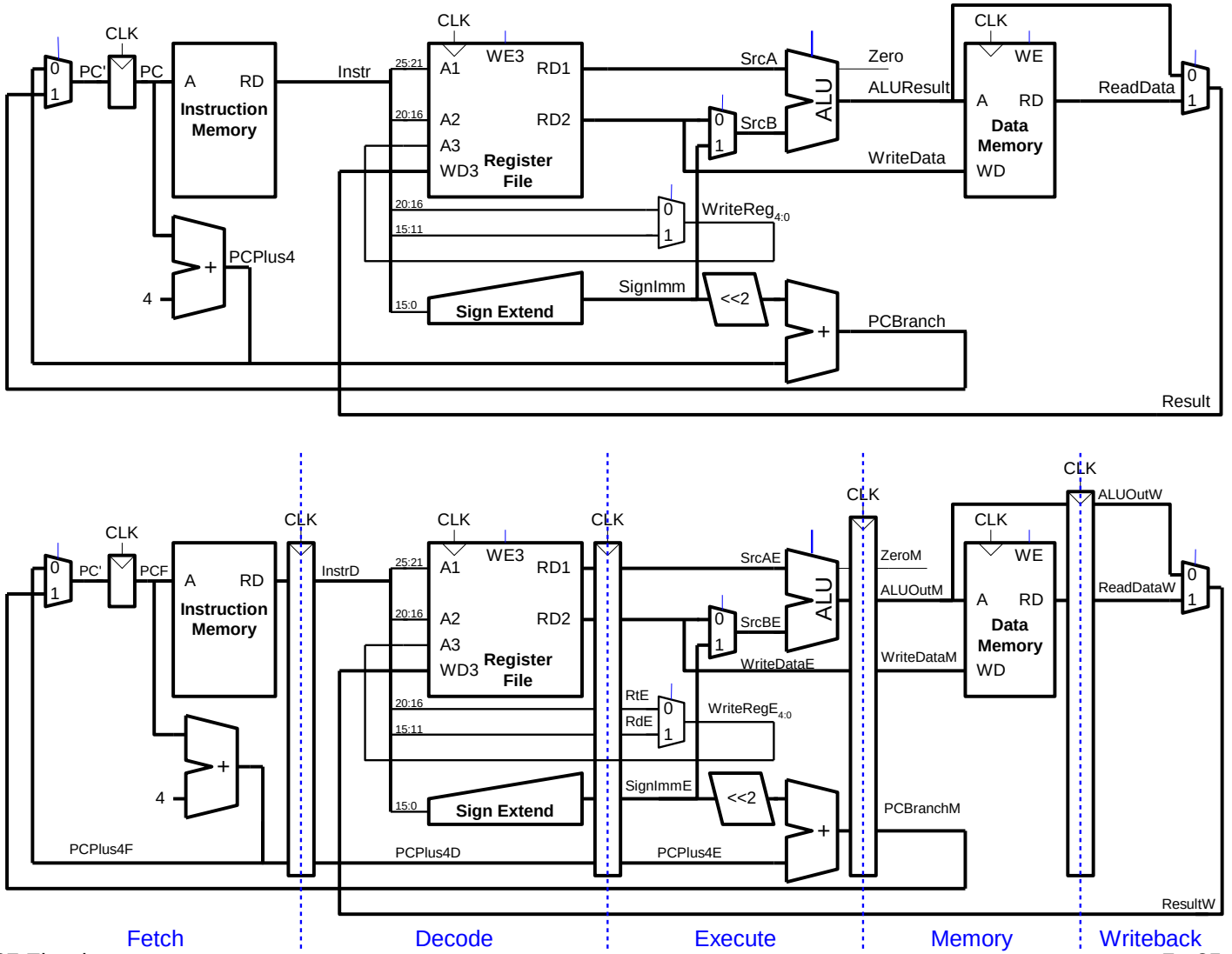
Pipelined



Pipelining Abstraction

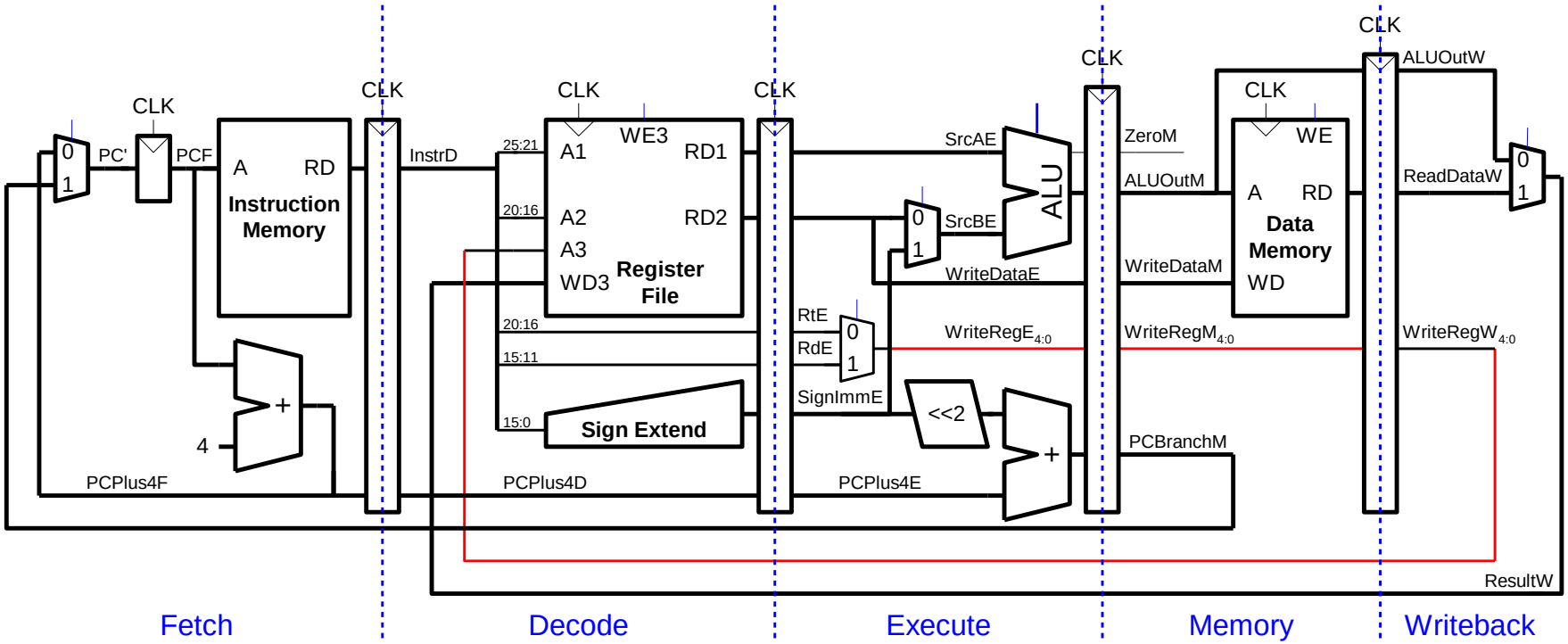


Single-Cycle and Pipelined Datapath

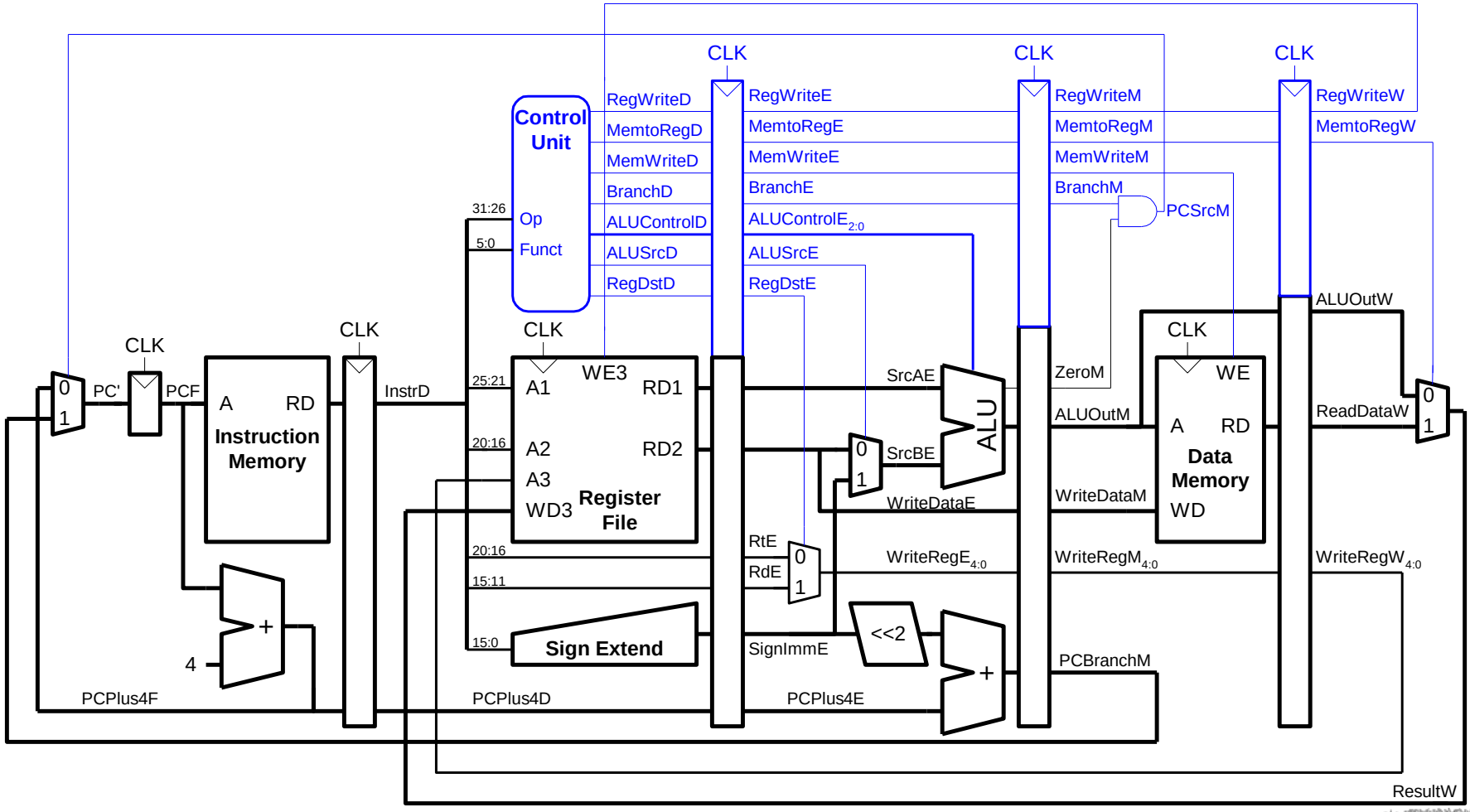


Corrected Pipelined Datapath

- WriteReg must arrive at the same time as Result



Pipelined Control



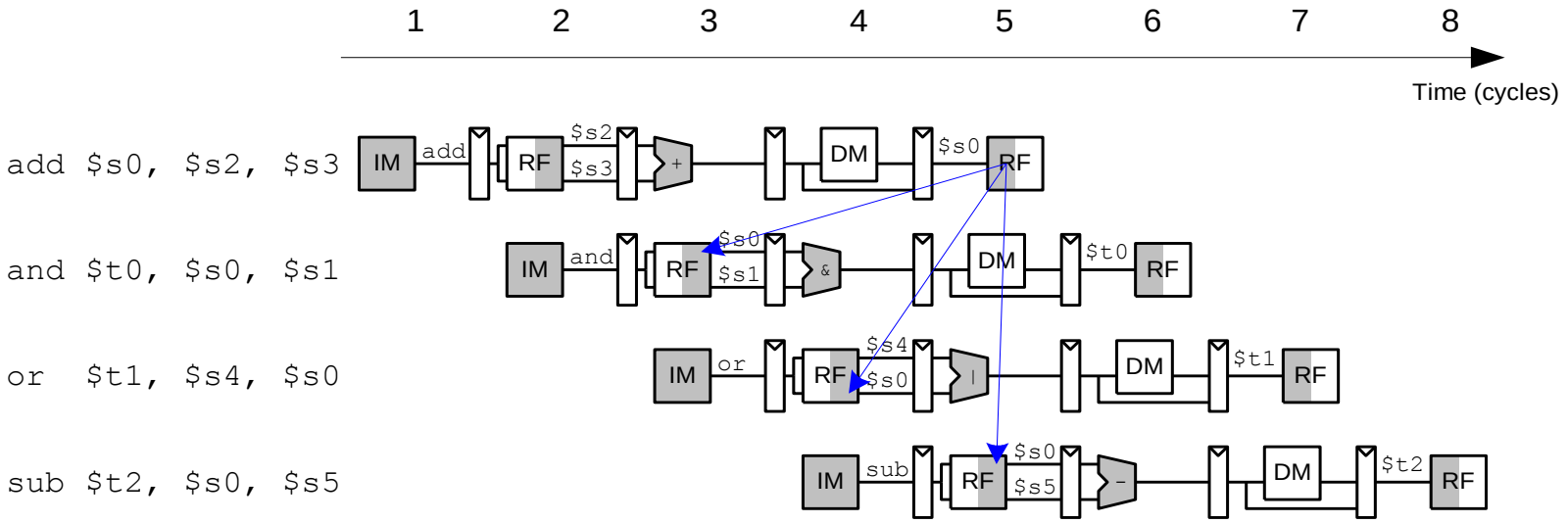
Same control unit as single-cycle processor

Control delayed to proper pipeline stage

Pipeline Hazard

- Occurs when an instruction depends on results from previous instruction that hasn't completed.
- Types of hazards:
  - **Data hazard:** register value not written back to register file yet
  - **Control hazard:** next instruction not decided yet (caused by branches)

Data Hazard

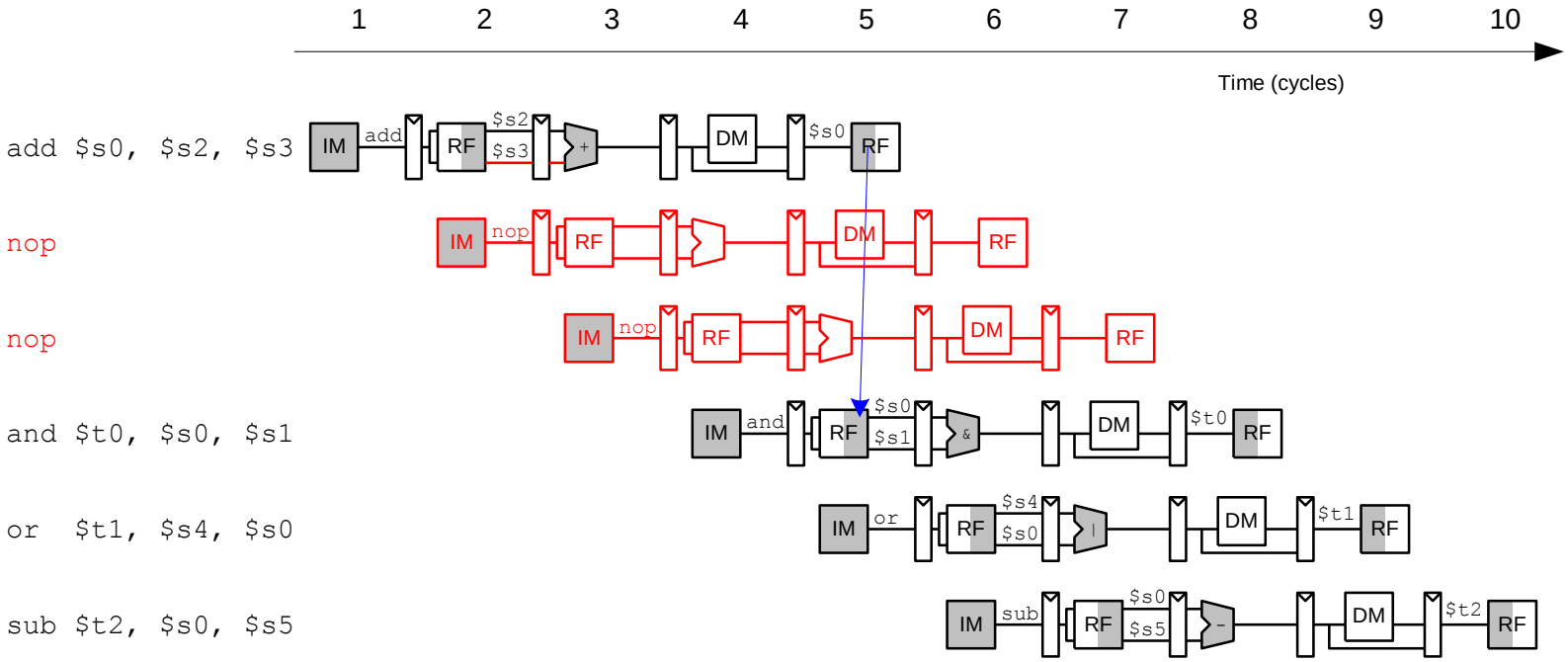


Handling Data Hazards

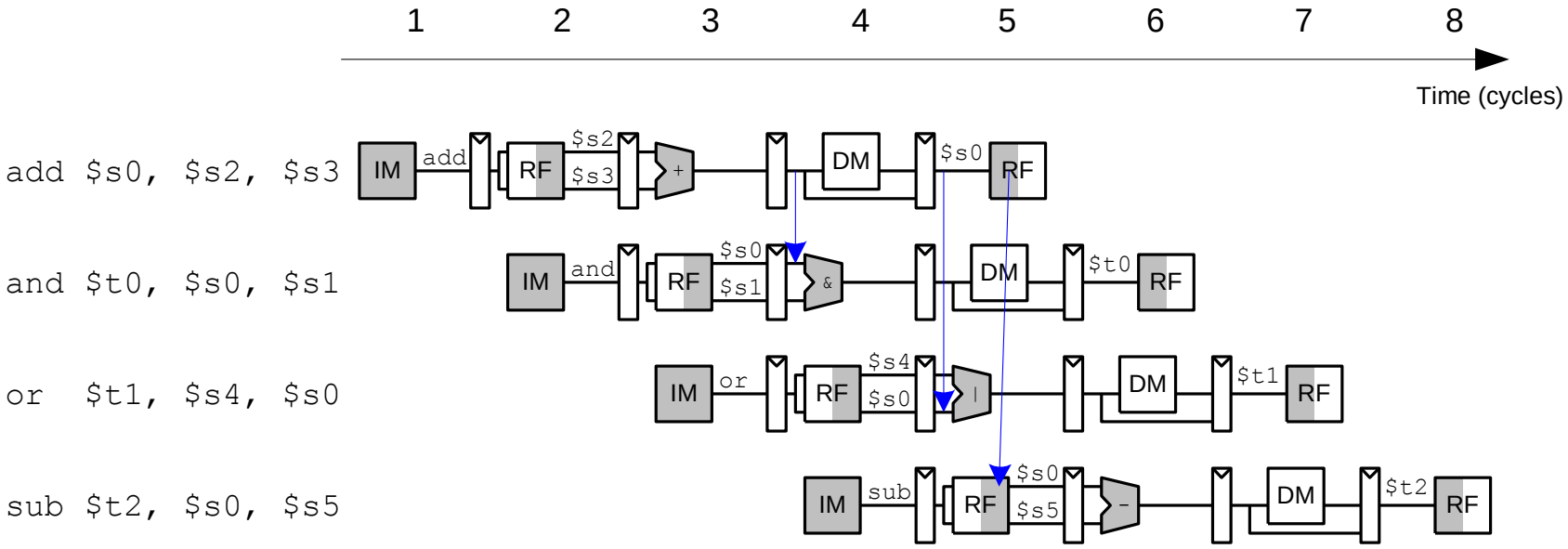
- Insert `nop`s in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Compile-Time Hazard Elimination

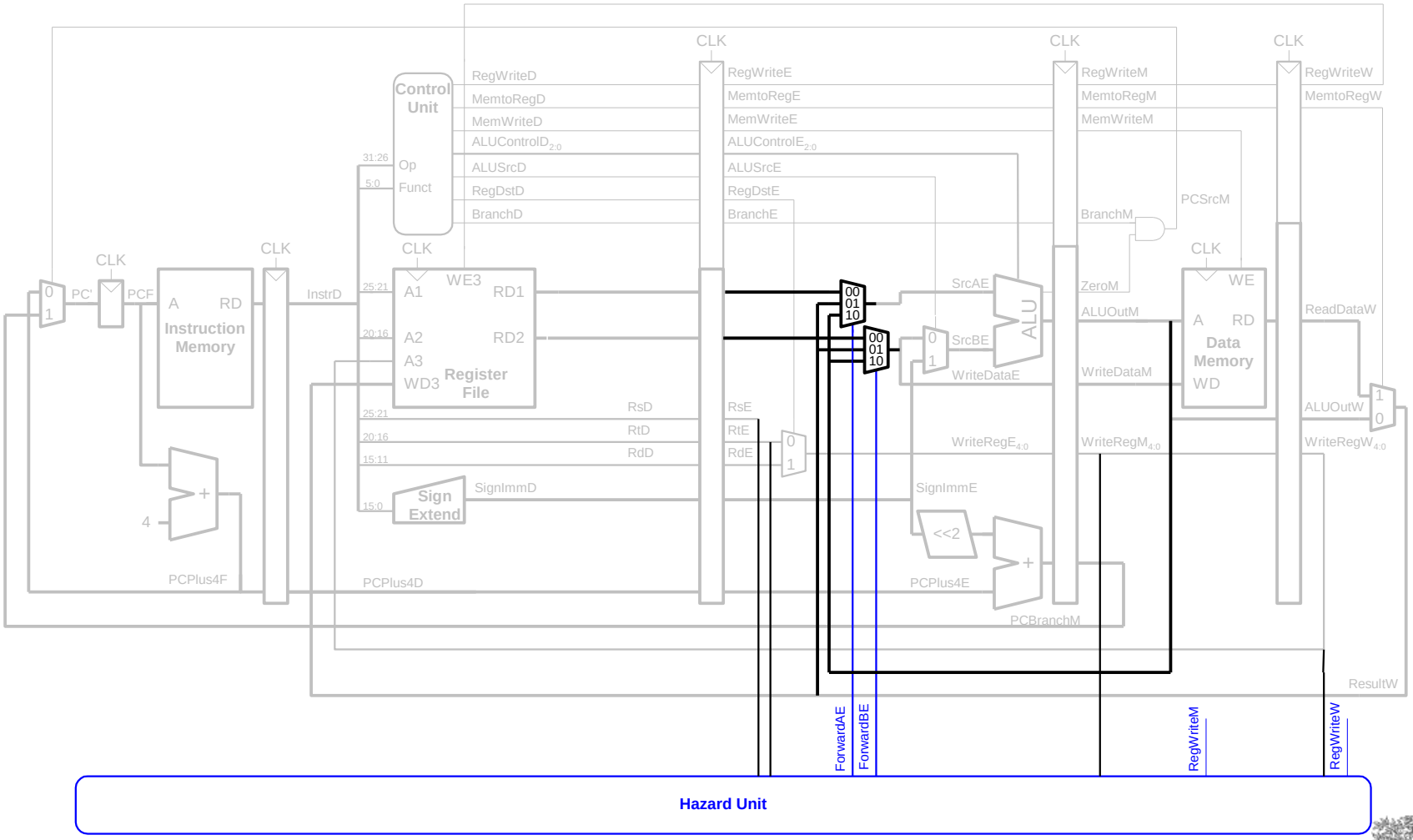
- Insert enough `nop`s for result to be ready
- Or move independent useful instructions forward



Data Forwarding



Data Forwarding



Data Forwarding

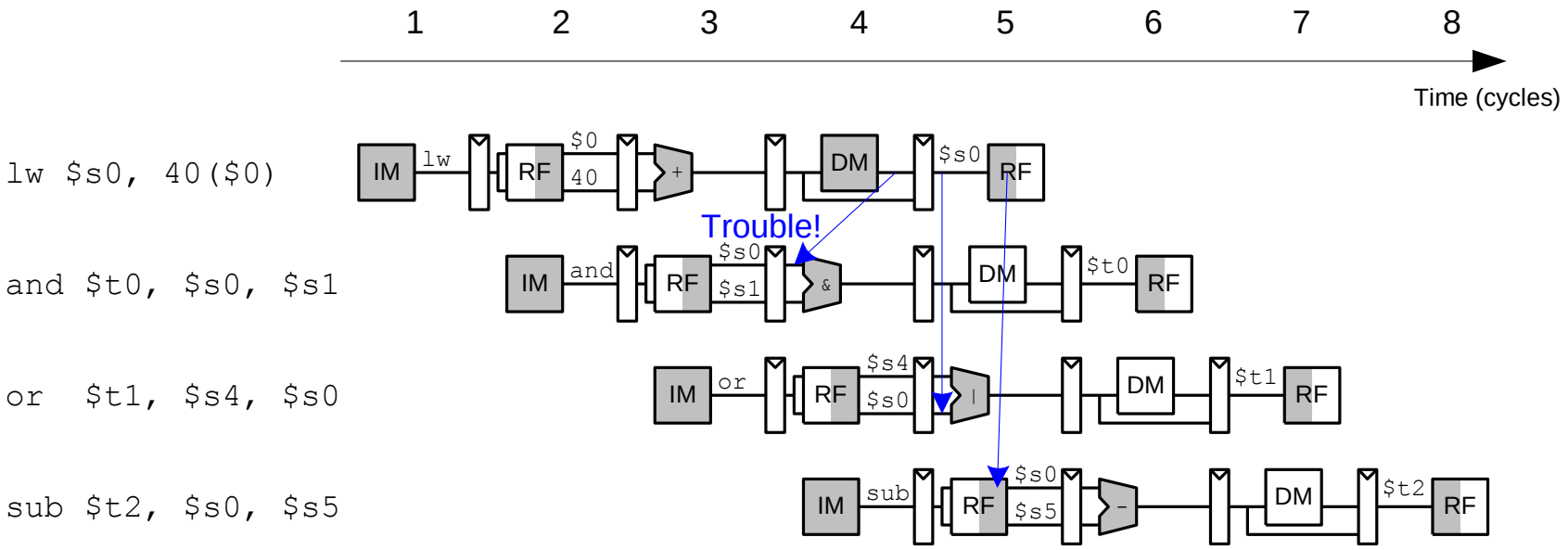
- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage

- Forwarding logic for *ForwardAE*:

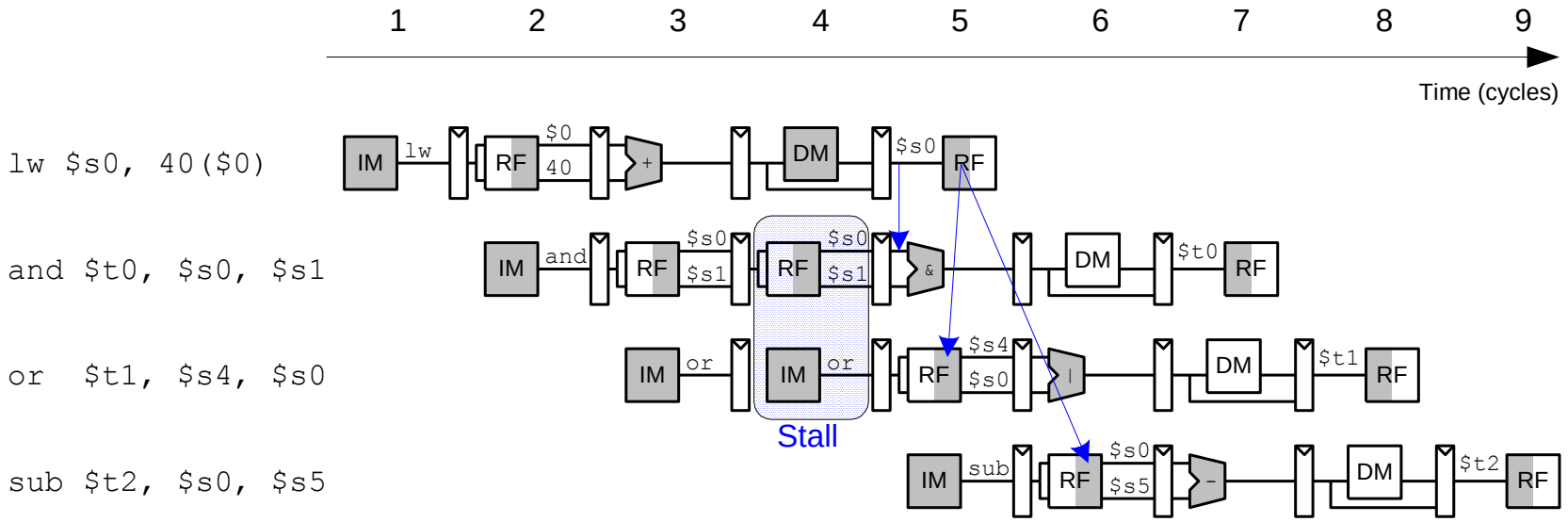
```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
then    ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
then     ForwardAE = 01
else     ForwardAE = 00
```

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

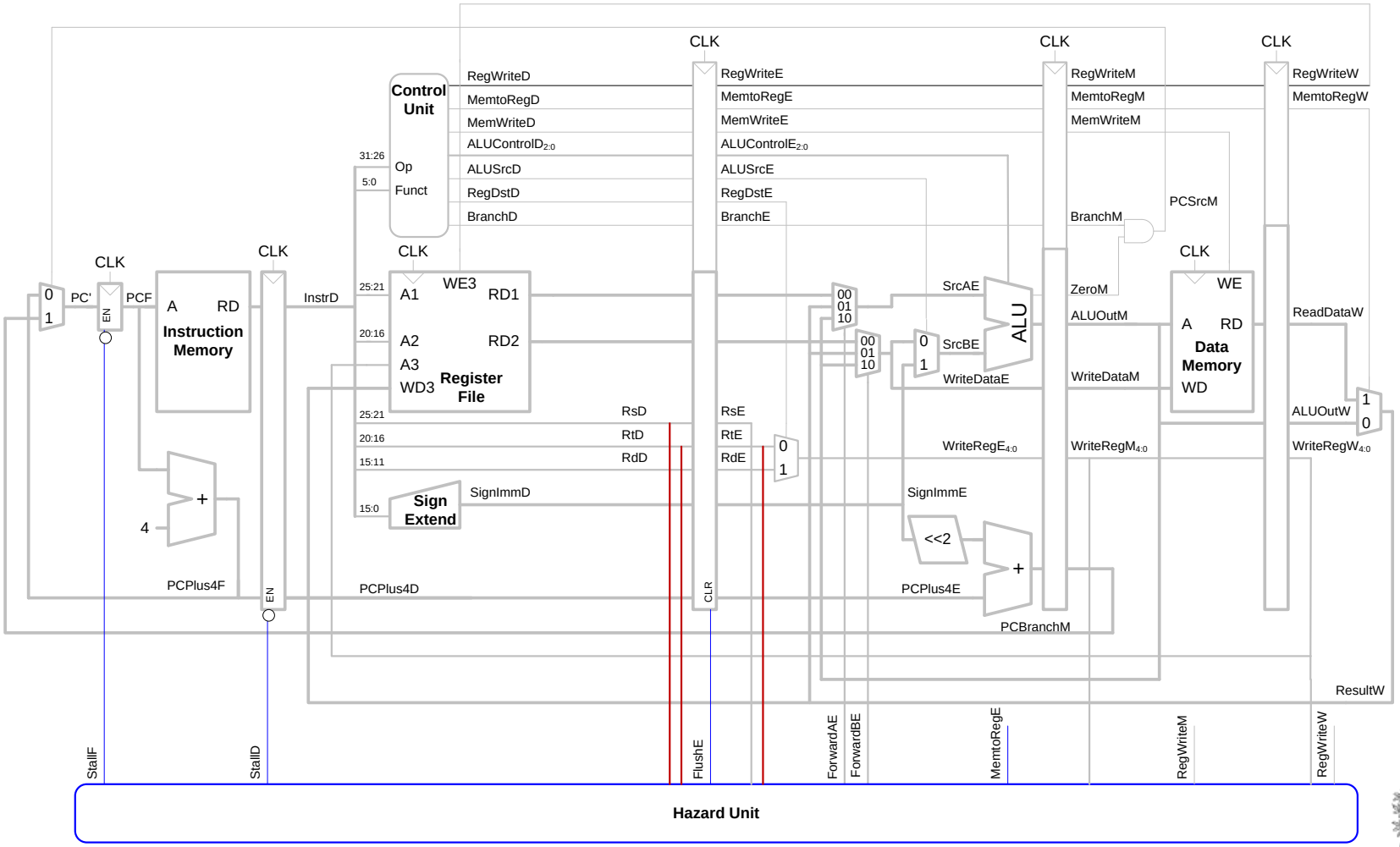
Stalling



Stalling



Stalling Hardware



Stalling Hardware

- Stalling logic:

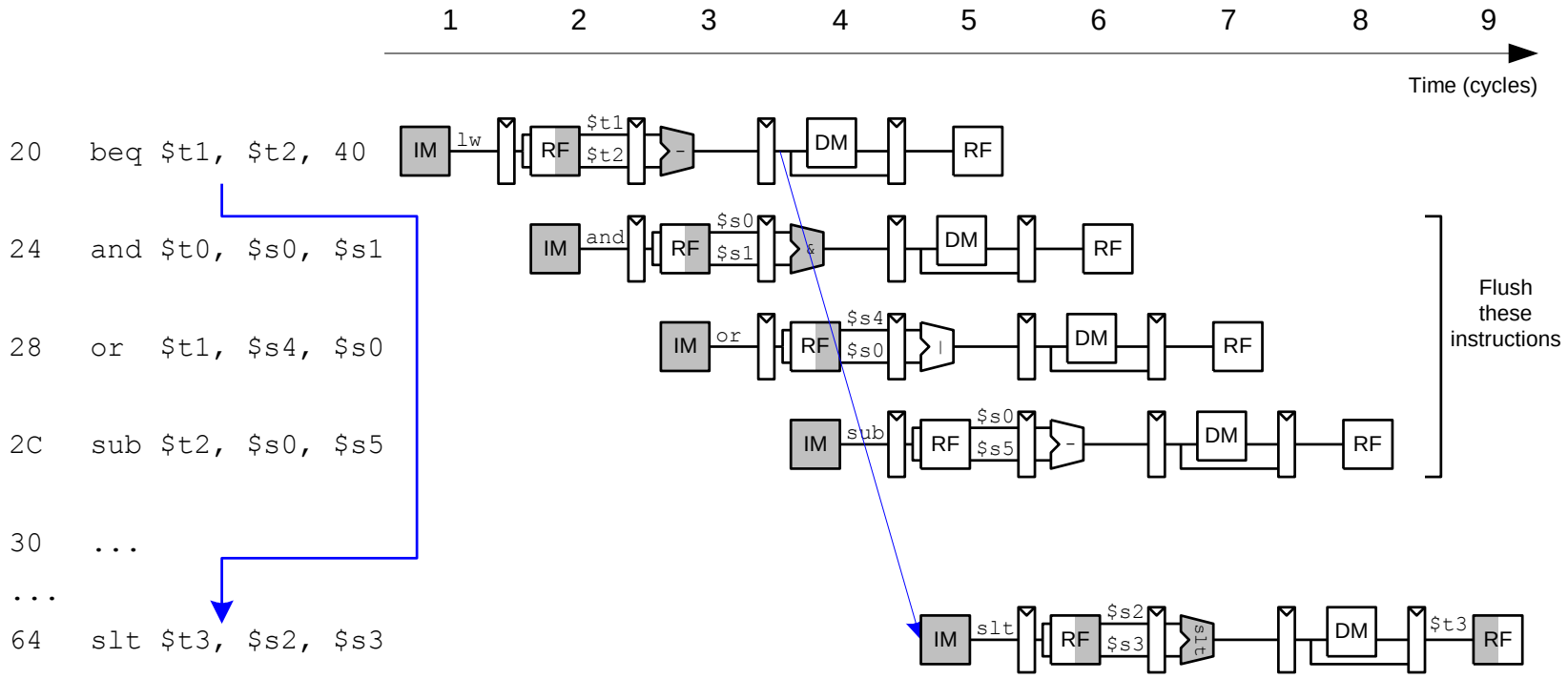
```
lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall
```

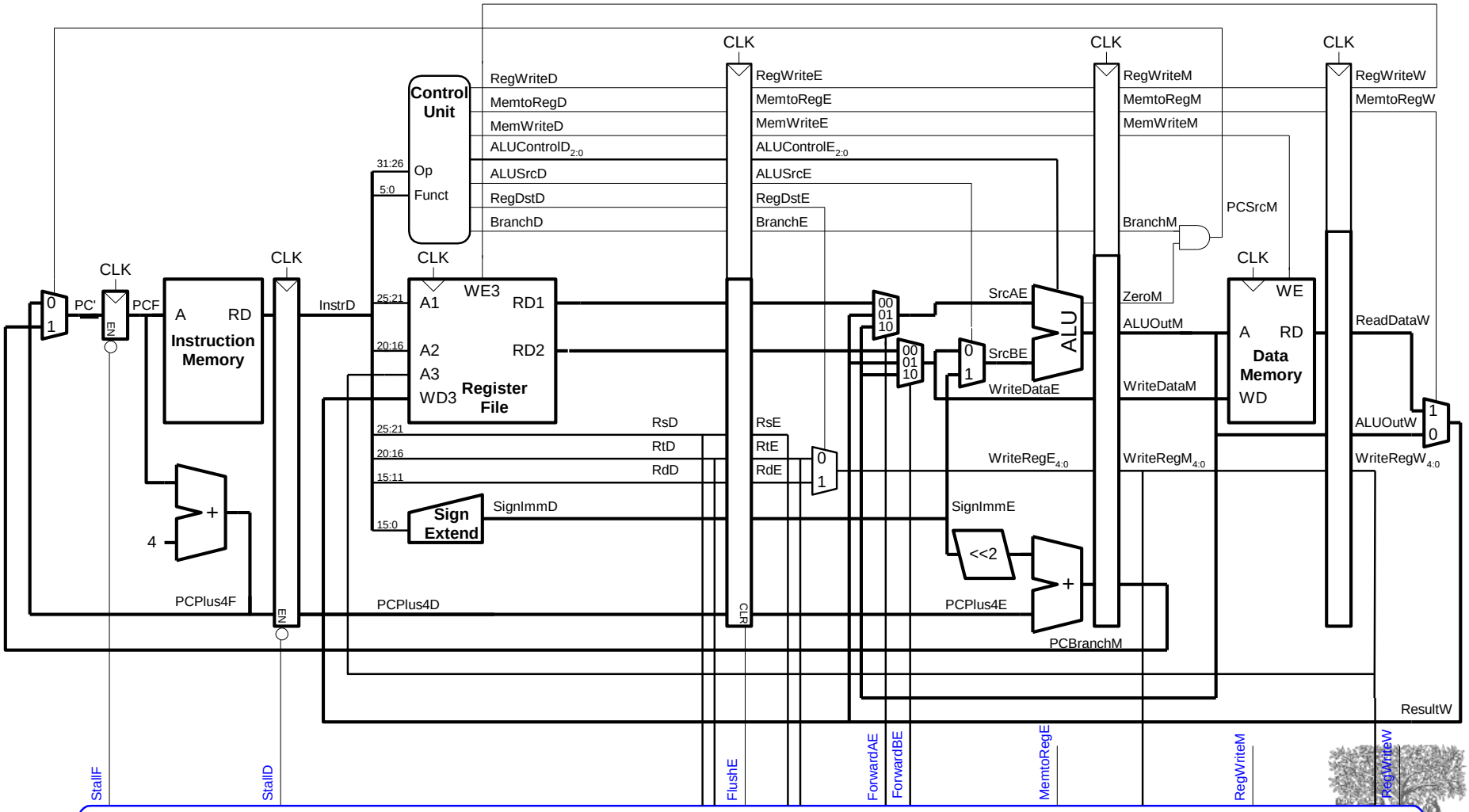
Control Hazards

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

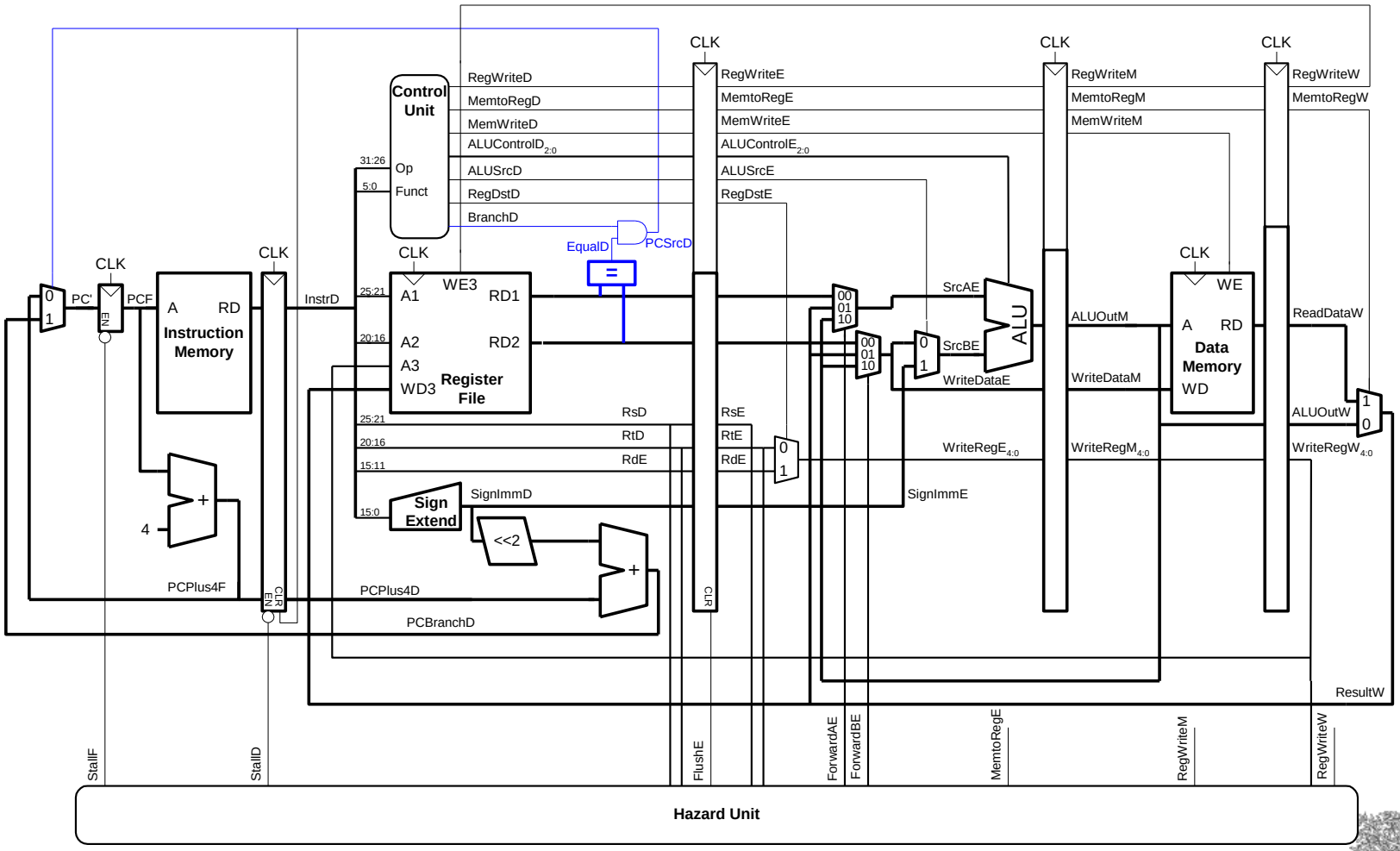
Control Hazards



Control Hazards: Original Pipeline

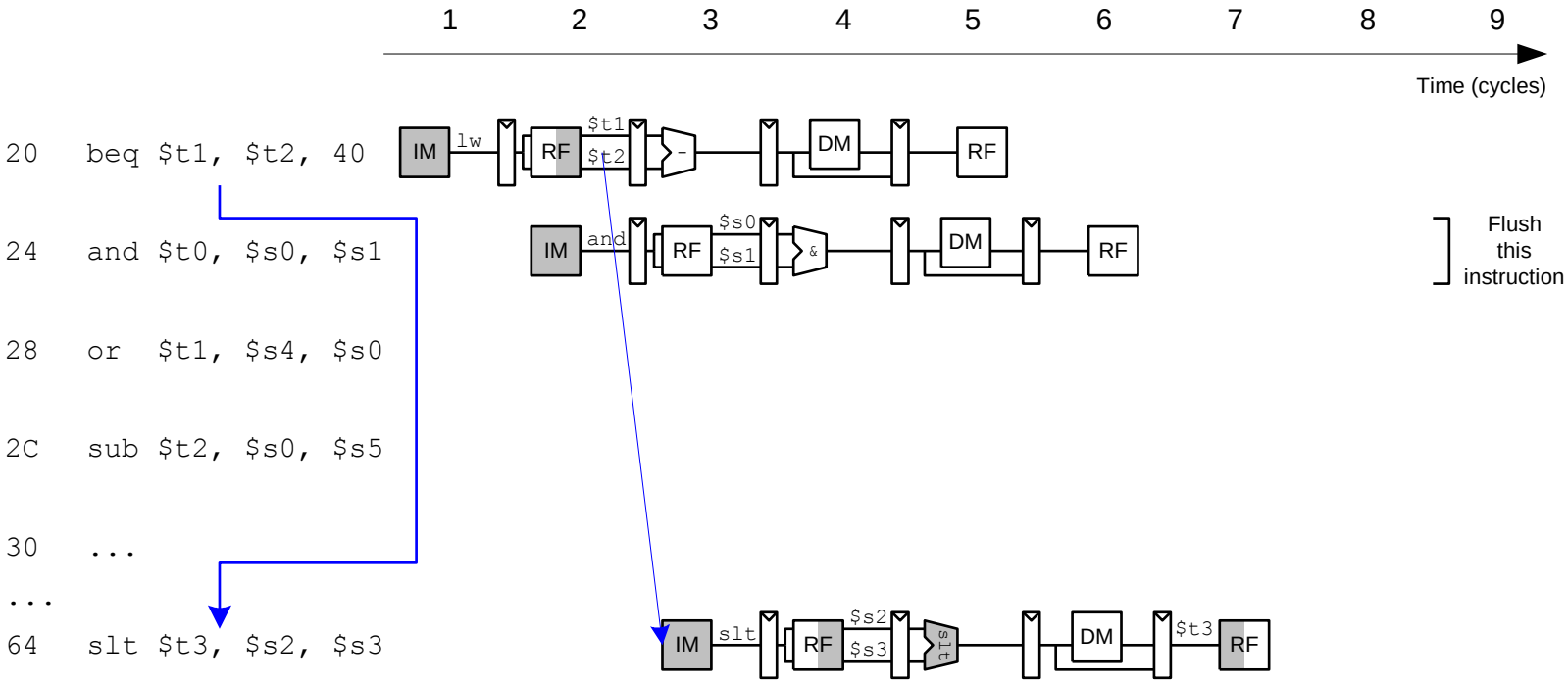


Control Hazards: Early Branch Resolution



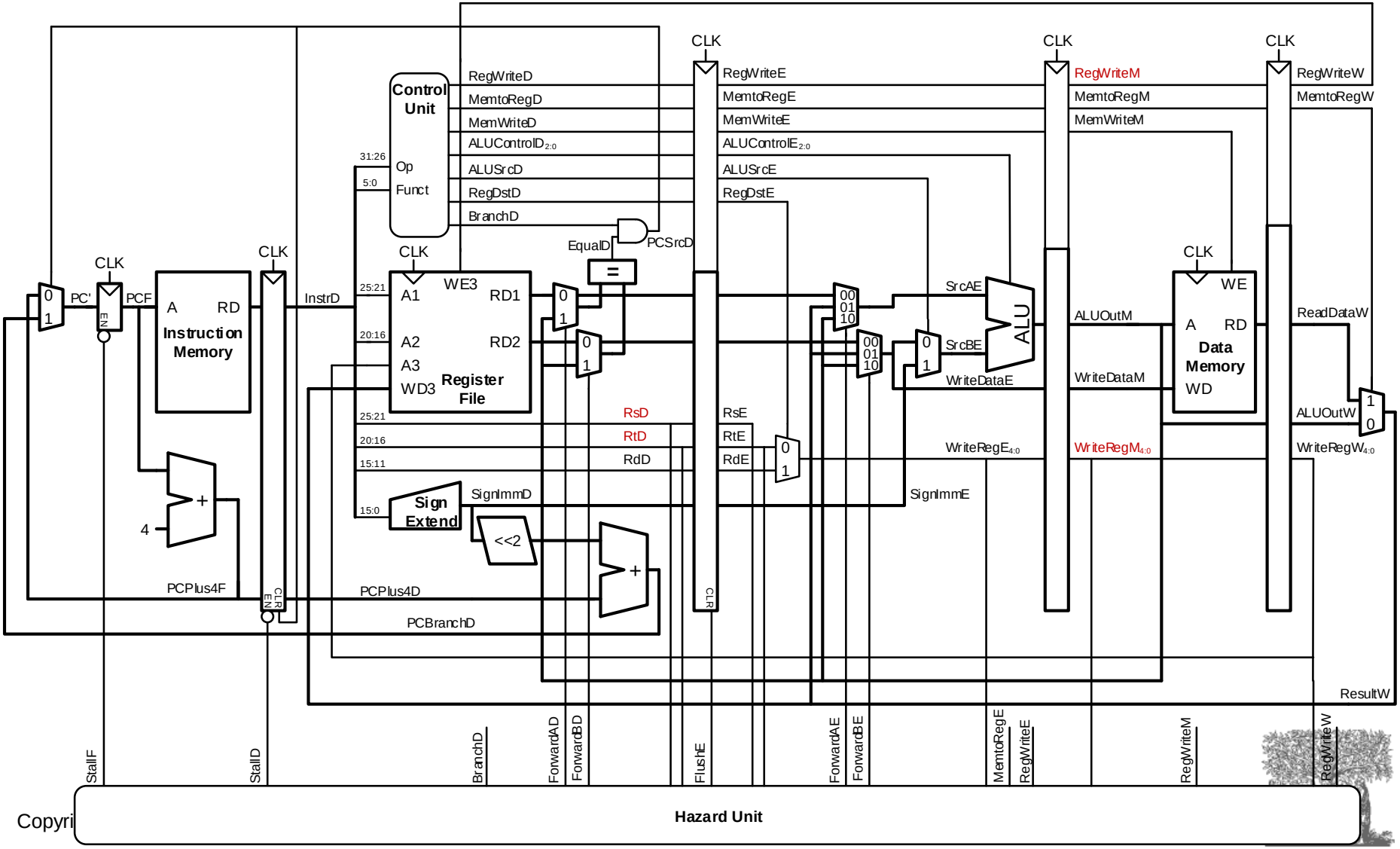
Introduced another data hazard in Decode stage

Control Hazards with Early Branch Resolution



ELSEVIER

$$ForwardAD = (rsD \neq 0) \text{ AND } (rsD == WriteRegM) \text{ AND } RegWriteM$$
$$ForwardBD = (rtD \neq 0) \text{ AND } (rtD == WriteRegM) \text{ AND } RegWriteM$$



Control Forwarding and Stalling Hardware

- 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)
  - Perhaps consider history of whether branch was previously taken to improve the guess
- Good prediction reduces the fraction of branches requiring a flush

Pipelined Performance Example

- Ideally CPI = 1
- But need to handle stalling (caused by loads and branches)
- 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
- **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. Thus,
  - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
  - $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$
  - Thus,

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:

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

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

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

Pipelined Performance Example

- For a program with 100 billion instructions executing on a pipelined MIPS processor,
- CPI = 1.15
- T_c = 550 ps

Execution Time = (# instructions) × CPI × T_c

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

= 63 seconds

Processor	Execution Time (seconds)	Speedup (single-cycle is baseline)
Single-cycle	95	1
Multicycle	133	0.71
Pipelined	63	1.51

Review: Exceptions

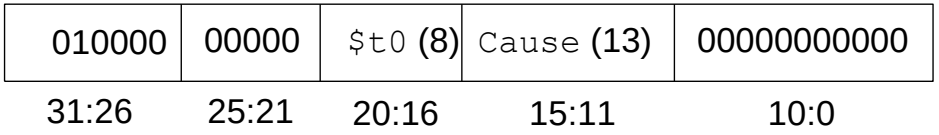
- Unscheduled procedure call to the *exception handler*
- Casued by:
  - Hardware, also called an *interrupt*, e.g. keyboard
  - Software, also called *traps*, e.g. undefined instruction
- When exception occurs, the processor:
  - Records the cause of the exception (`Cause` register)
  - Jumps to the exception handler at instruction address 0x80000180
  - Returns to program (`EPC` register)

Example Exception

Exception Registers

- Not part of the register file.
  - `Cause`
    - Records the cause of the exception
    - Coprocessor 0 register 13
  - `EPC` (Exception PC)
    - Records the PC where the exception occurred
    - Coprocessor 0 register 14
- Move from Coprocessor 0
  - `mfc0 $t0, Cause`
  - Moves the contents of `Cause` into `$t0`

mfc0

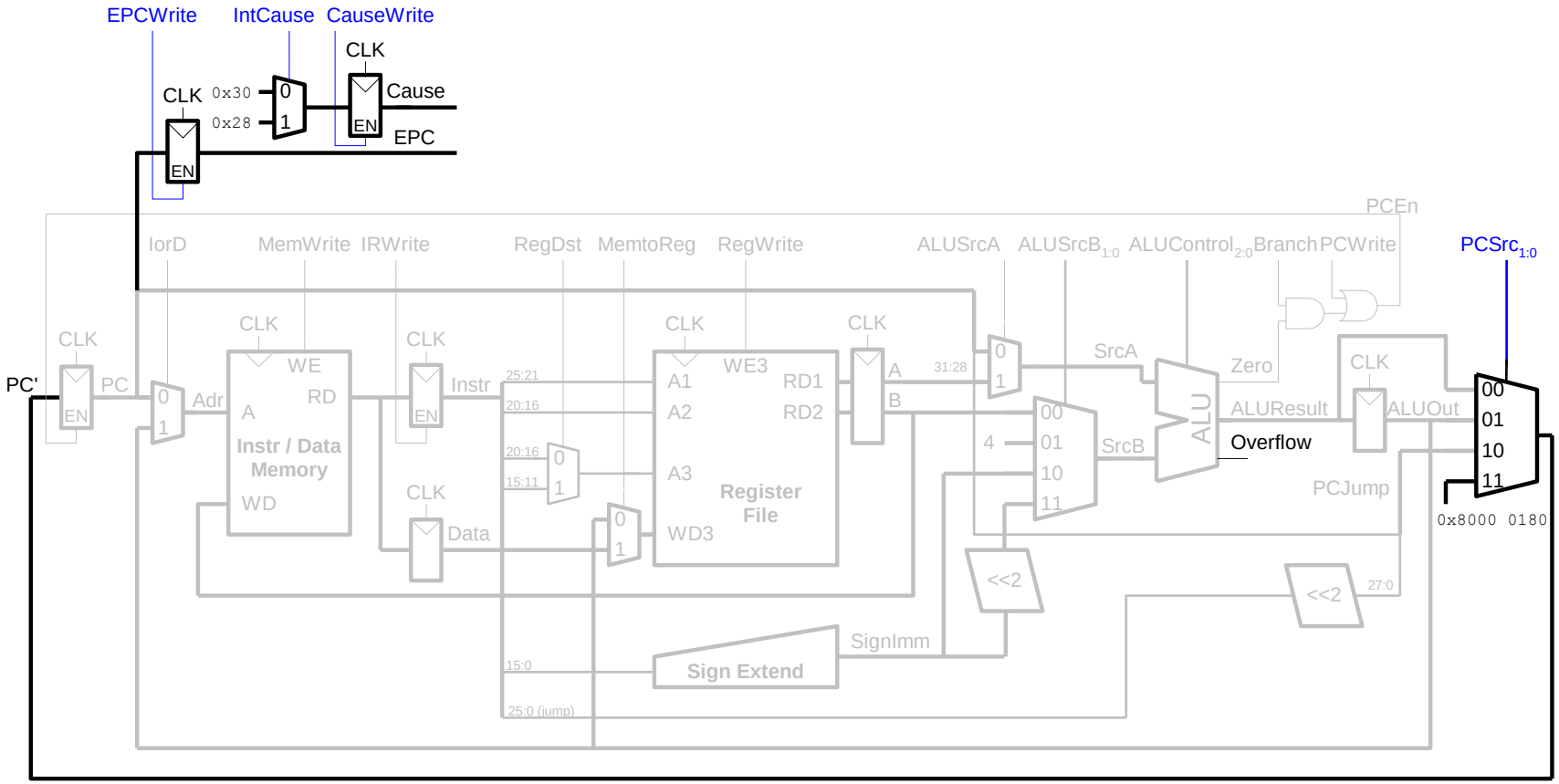


Exception Causes

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

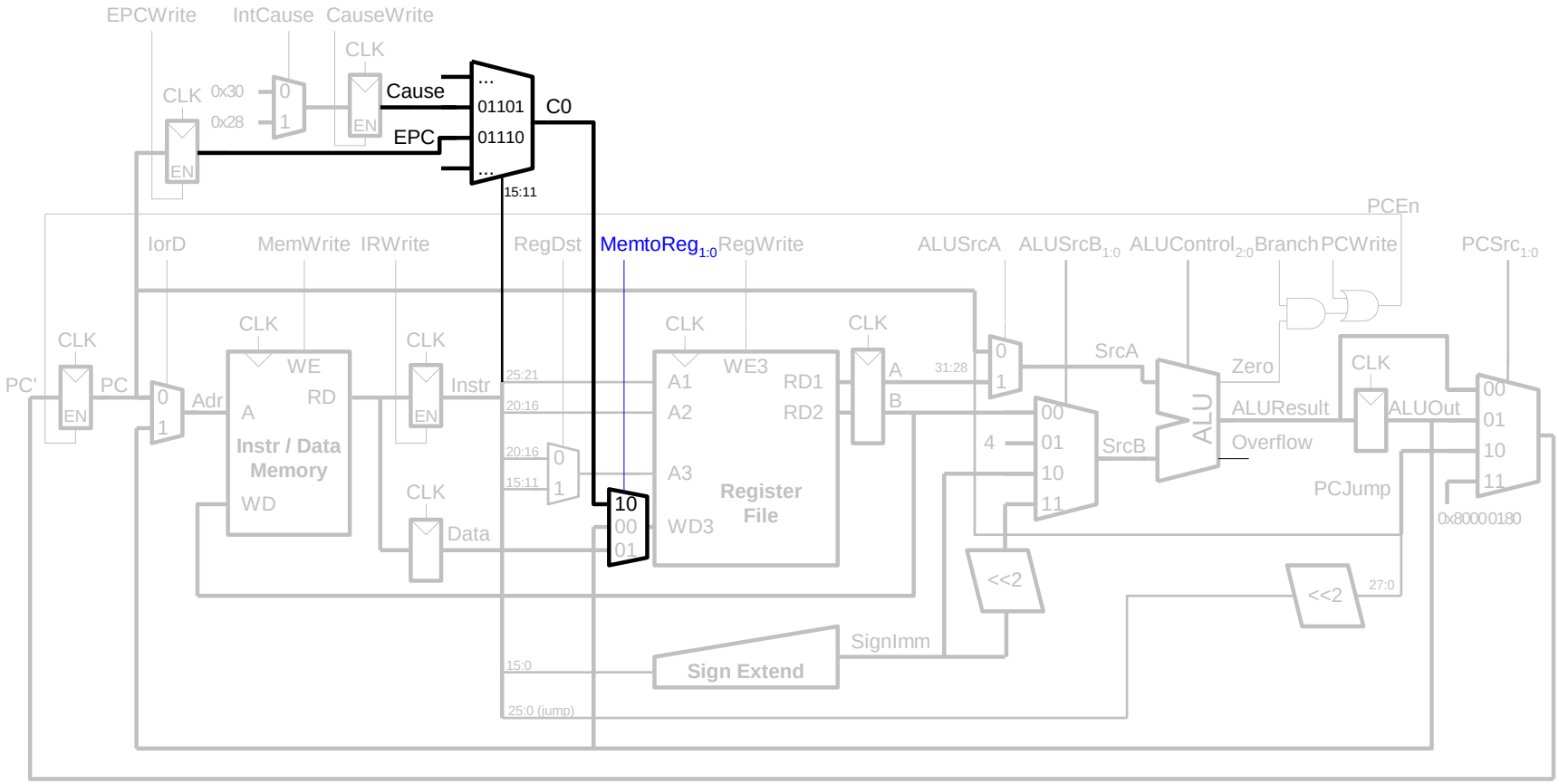
We extend the multicycle MIPS processor to handle the last two types of exceptions.

Exception Hardware: EPC & Cause



ELSEVIER

Exception Hardware: `mfc0`



Control FSM with Exceptions