Computer Architecture

Lec 4b: Multi-Cycle Datapath

Multi-Cycle Microarchitectures

Multi-Cycle Microarchitectures

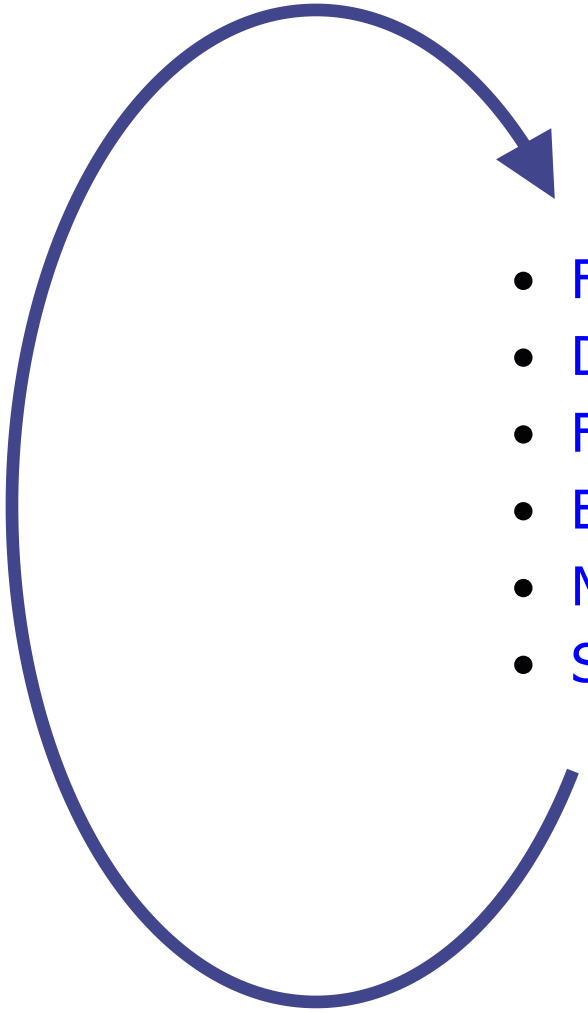
- Goal: Let each instruction take (close to) only as much time it really needs

- Idea
  - Determine clock cycle time independently of instruction processing time
  - Each instruction takes as many clock cycles as it needs to take
    - Multiple state transitions per instruction
    - The states followed by each instruction is different

Multi-Cycle Microarchitectures

- Key Idea for Realization
  - One can implement the "process instruction" step as a finite state machine that sequences between states and eventually returns back to the "fetch instruction" state
  - A state is defined by the control signals asserted in it
  - Control signals for the next state are determined in current state

The Instruction Processing Cycle

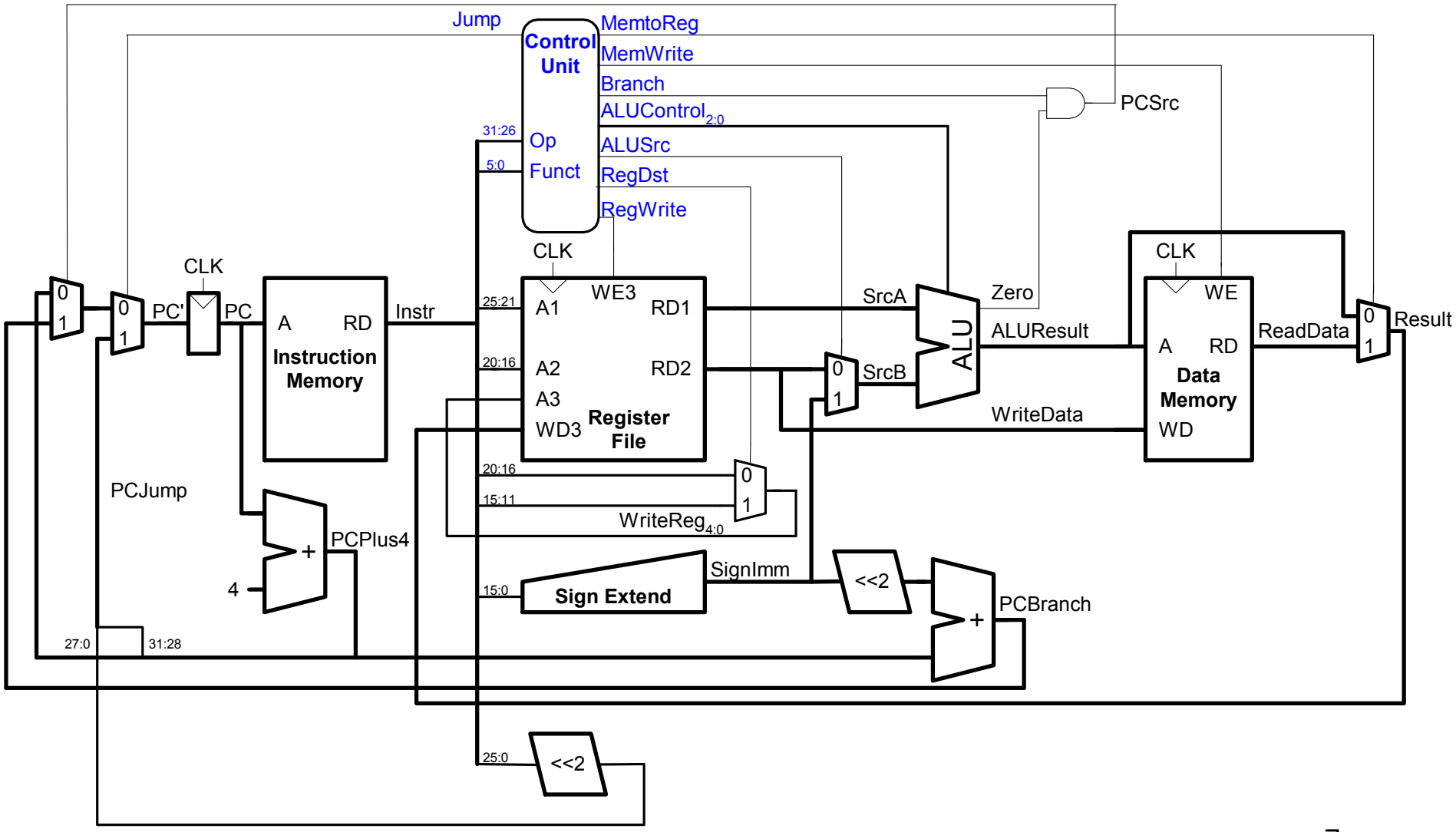


- Fetch
- Decode
- Fetch Operands
- Execute
- Memory Access
- Store Result

A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into "states"
  - A stage in the instruction processing cycle can take multiple states
- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state
- The behavior of the entire processor is specified fully by a *finite state machine*
- In a state (clock cycle), control signals control two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle

Remember: Single-Cycle MIPS Processor



Multi-cycle MIPS Processor

- Single-cycle microarchitecture:
  - cycle time limited by longest instruction (`lw`) → low clock frequency
  - three adders/ALUs and two memories → high hardware cost
- Multi-cycle microarchitecture:
  + higher clock frequency
  + simpler instructions run faster
  + reuse expensive hardware across multiple cycles
  - sequencing overhead paid many times
  - hardware overhead for storing intermediate results
- Same design steps: datapath & control

What Do We Want To Optimize

- Single Cycle Architecture uses two memories
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)

What Do We Want To Optimize

- Single Cycle Architecture uses two memories
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)
- Single Cycle Architecture needs three adders
  - ALU, PC, Branch address calculation
  - We want to use the ALU for all operations (smaller size)

What Do We Want To Optimize

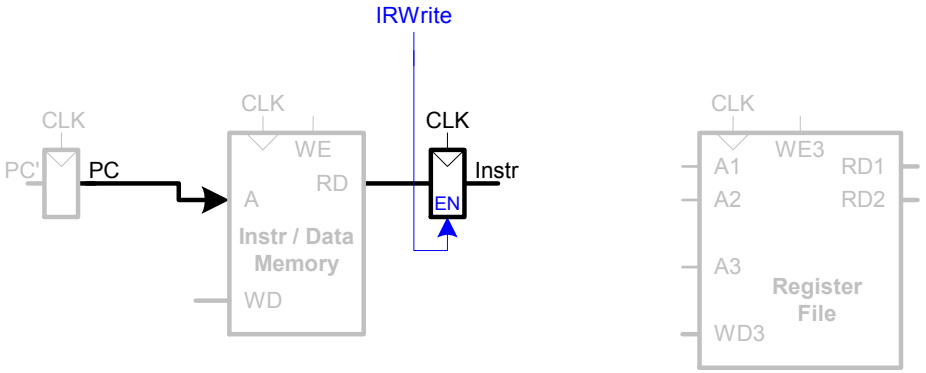
- Single Cycle Architecture uses two memories
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)
- Single Cycle Architecture needs three adders
  - ALU, PC, Branch address calculation
  - We want to use the ALU for all operations (smaller size)
- In Single Cycle Architecture all instructions take one cycle
  - The most complex operation slows down everything!
  - Divide all instructions into multiple steps
  - Simpler instructions can take fewer cycles (average case may be faster)

Consider the `lw` instruction

- For an instruction such as: `lw $t0, 0x20($t1)`
- We need to:
  - Read the instruction from memory
  - Then read **`$t1`** from register array
  - Add the immediate value (**`0x20`**) to calculate the memory address
  - Read the content of this address
  - Write to the register **`$t0`** this content

Multi-cycle Datapath: instruction fetch

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

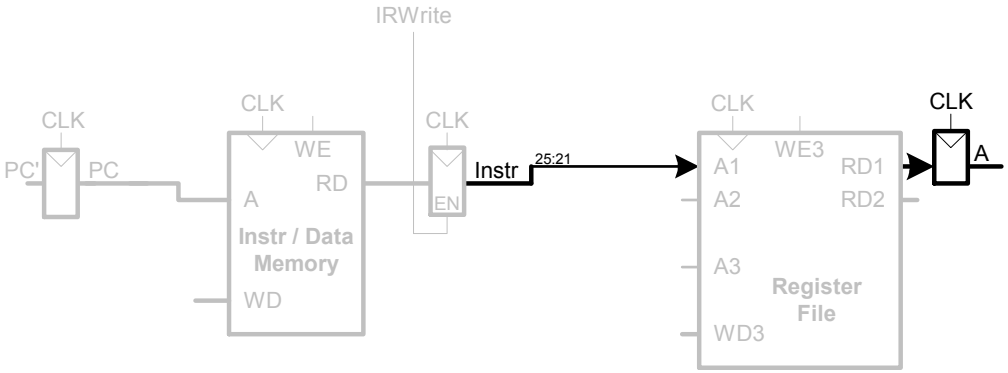


`read from the memory location [rs]+imm to location [rt]`

I-Type

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

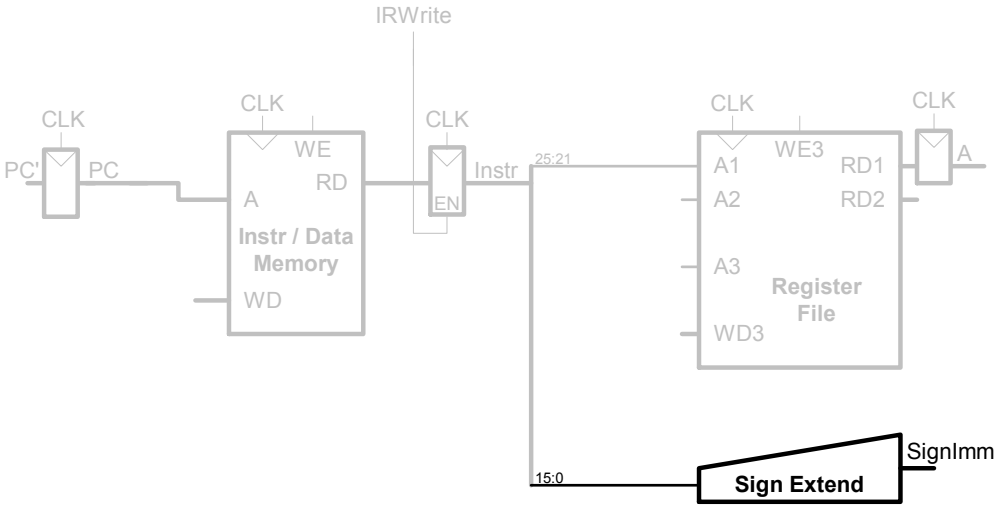
Multi-cycle Datapath: `lw` register read



I-Type

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

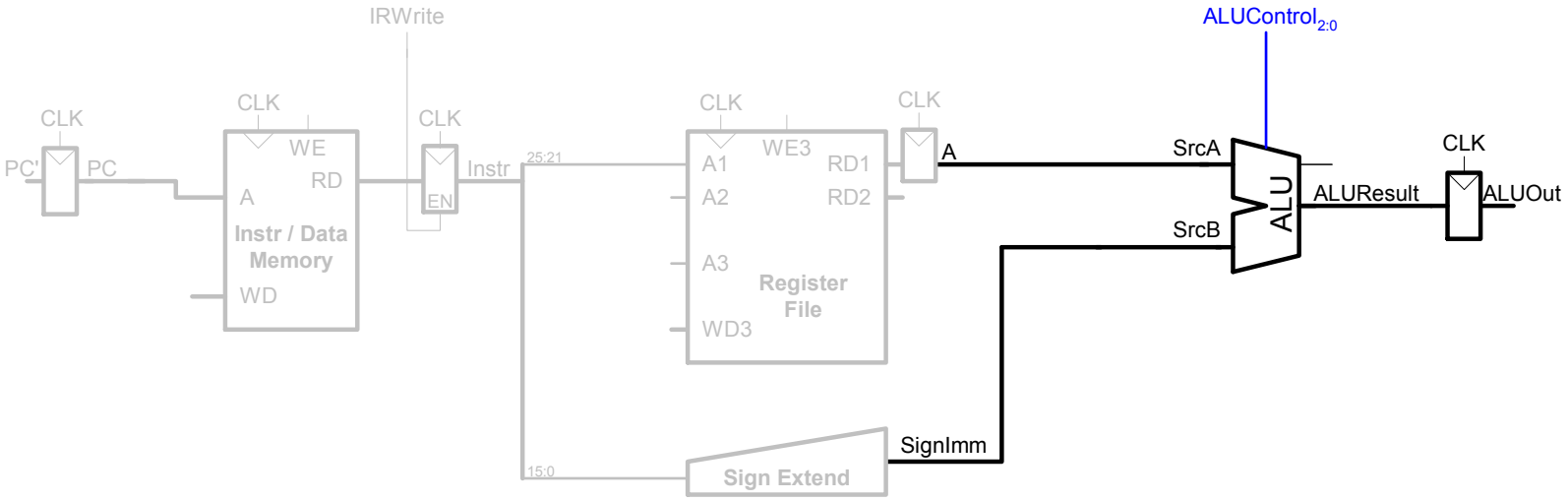
Multi-cycle Datapath: `lw` immediate



I-Type

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

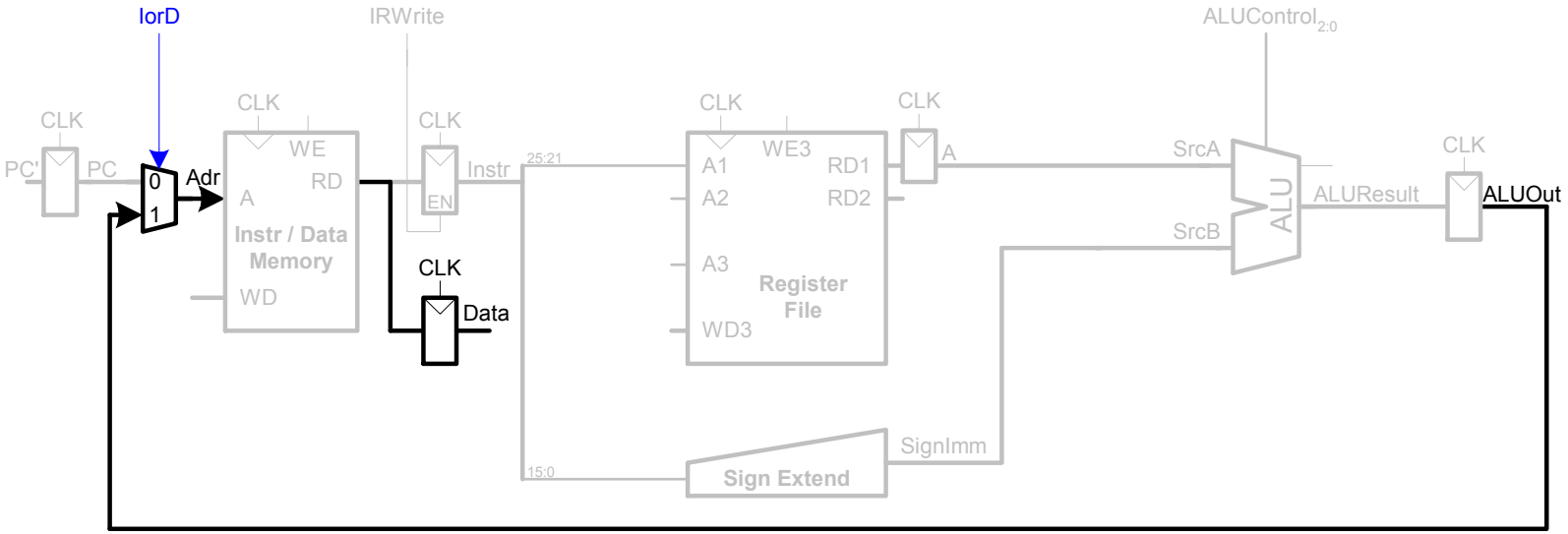
Multi-cycle Datapath: `lw` address



I-Type

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

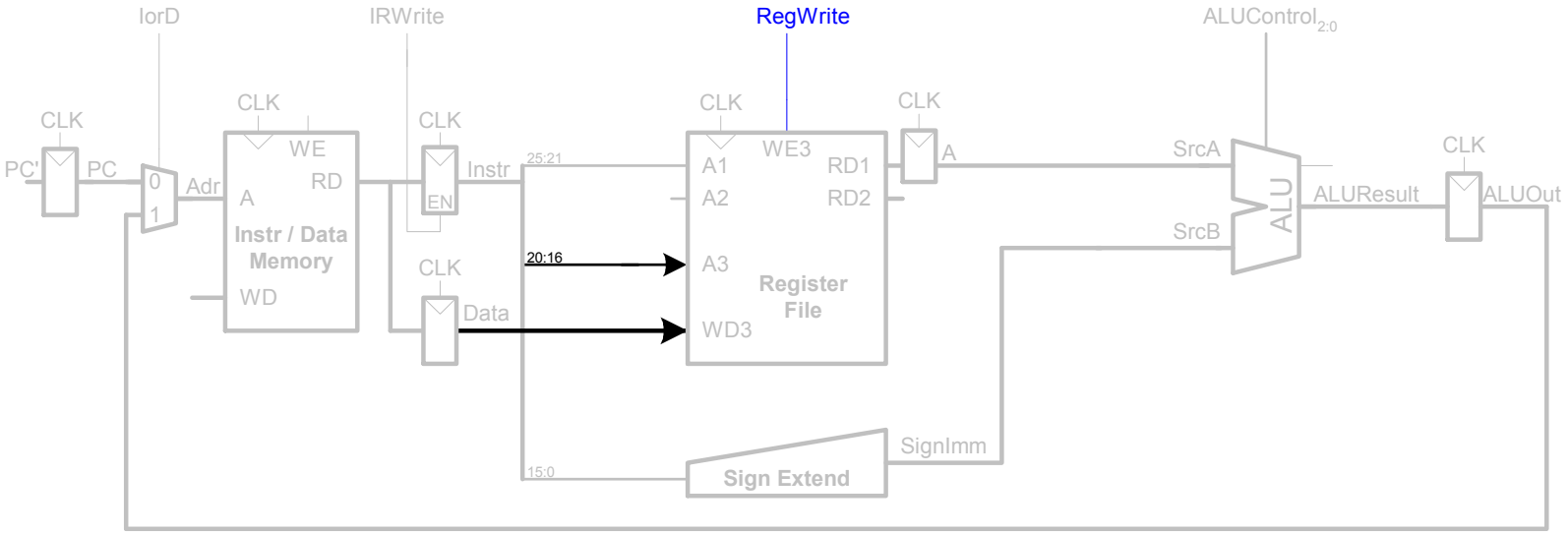
Multi-cycle Datapath: `lw` memory read



I-Type

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

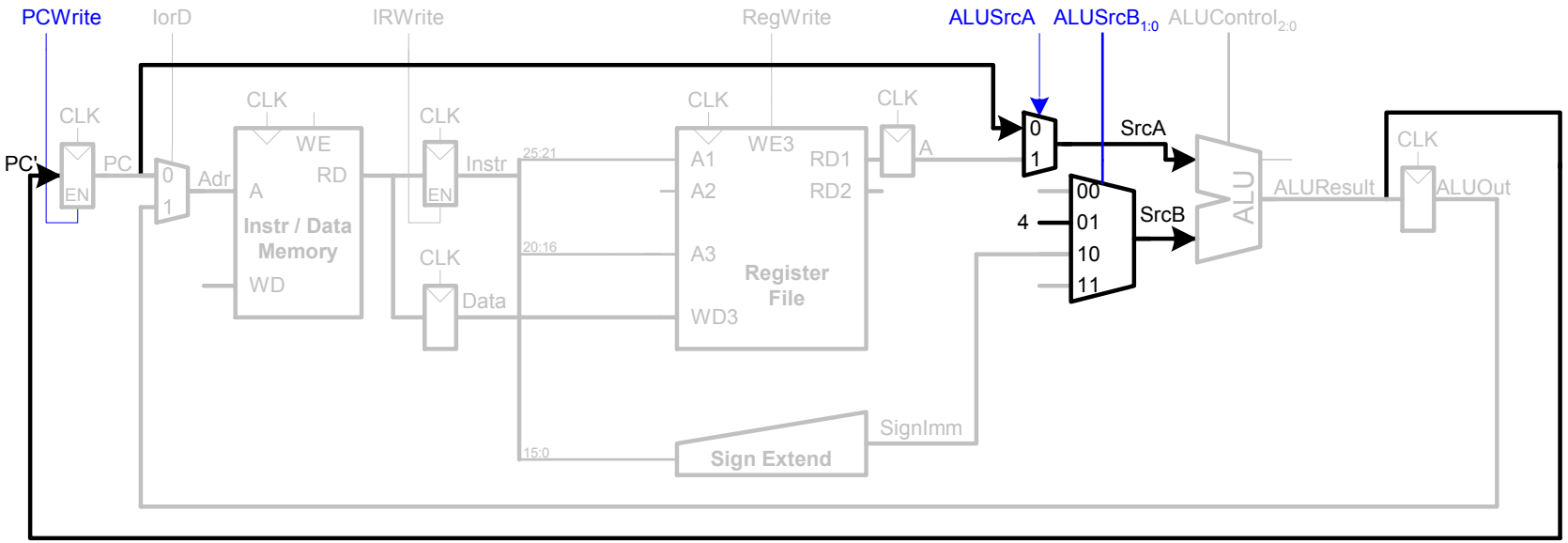
Multi-cycle Datapath: `lw` write register



I-Type

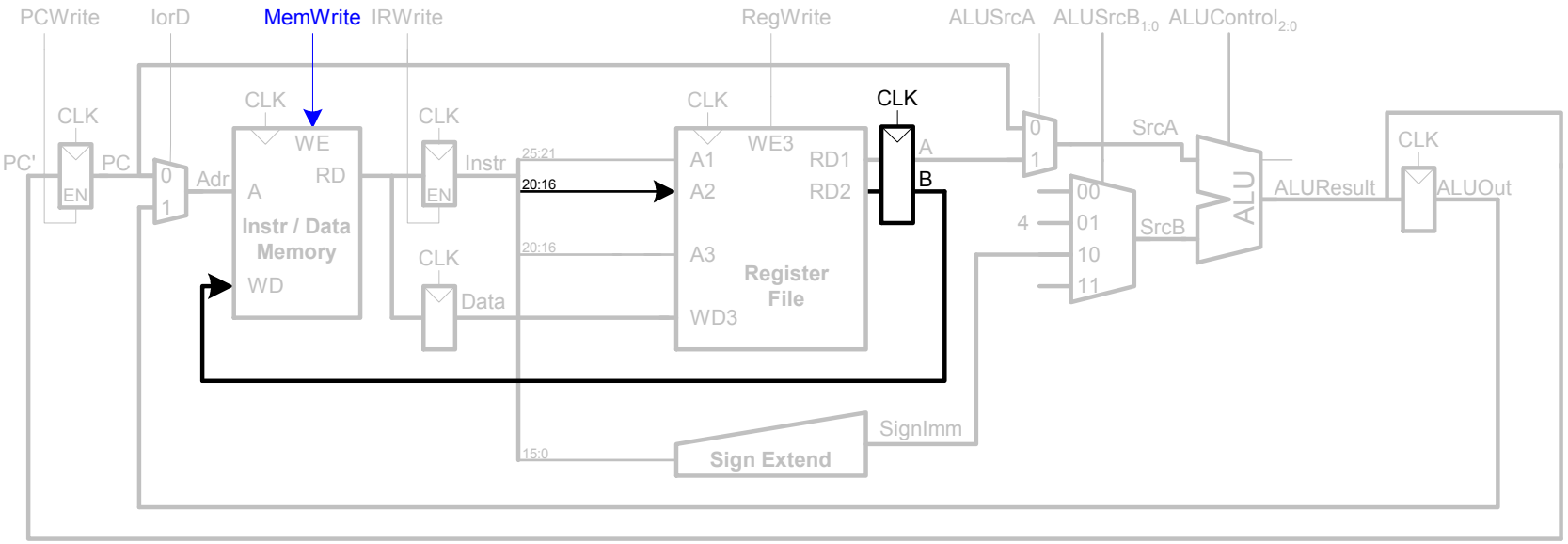
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-cycle Datapath: increment PC



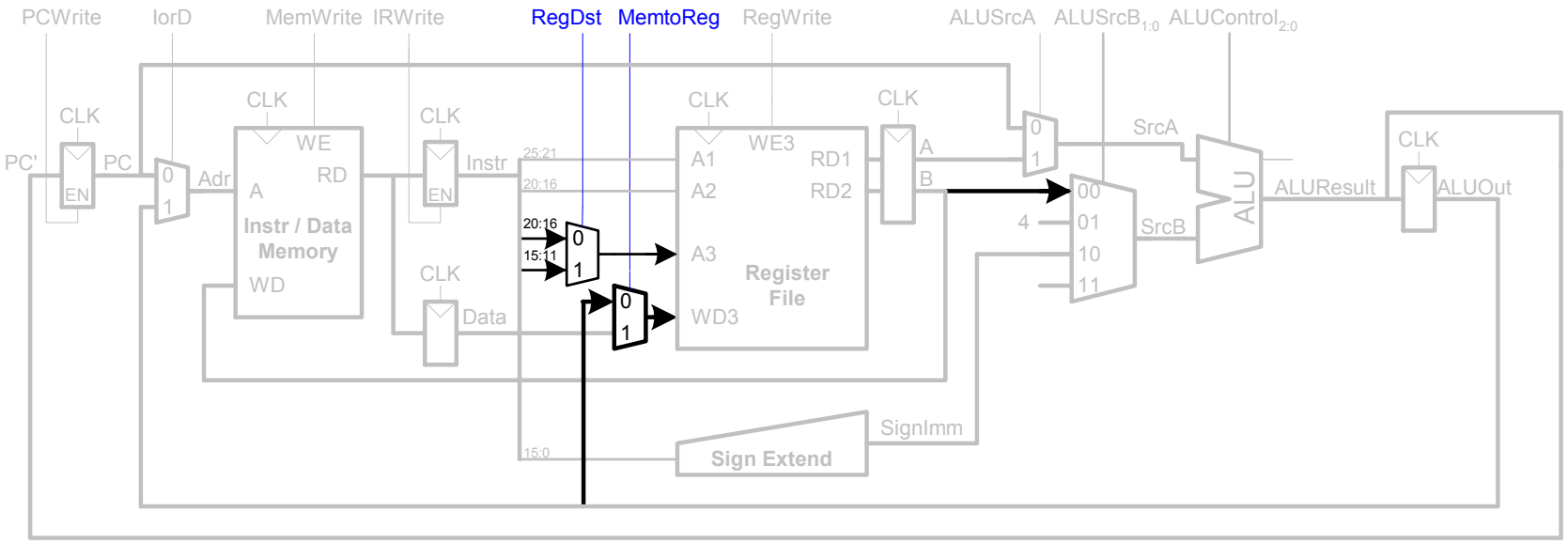
Multi-cycle Datapath: `sw`

- **Write data in rt to memory**



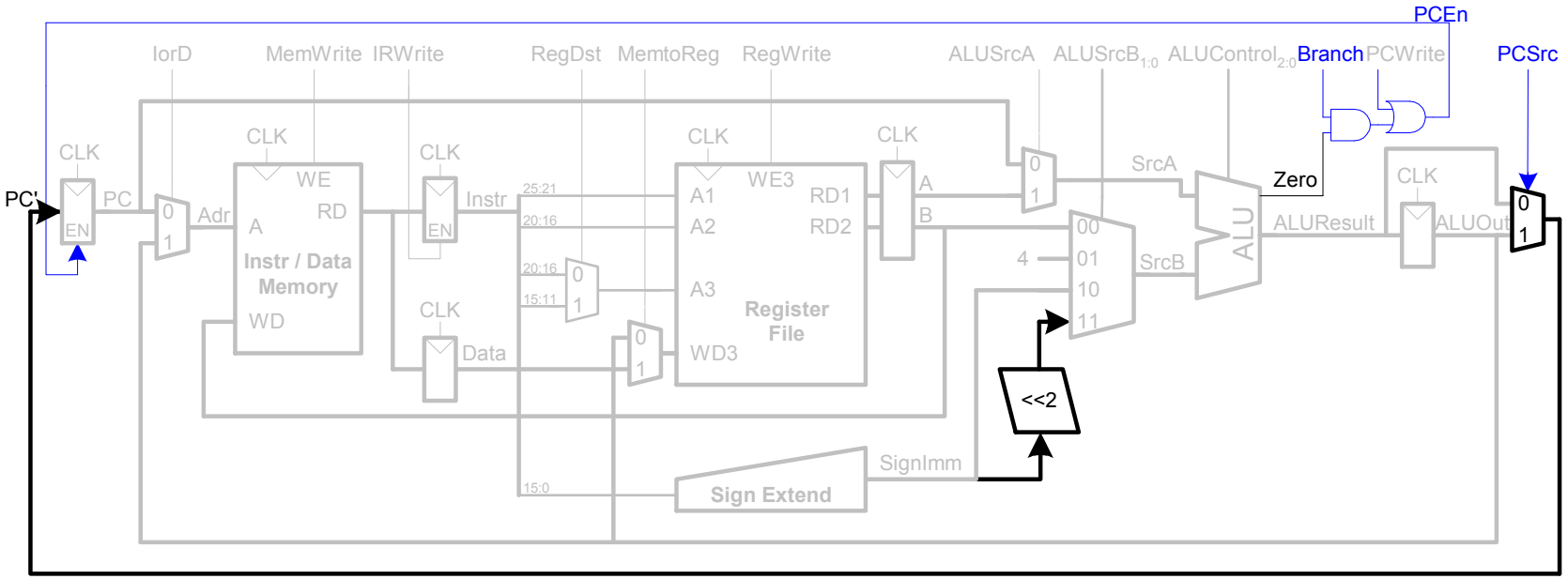
Multi-cycle Datapath: R-type Instructions

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

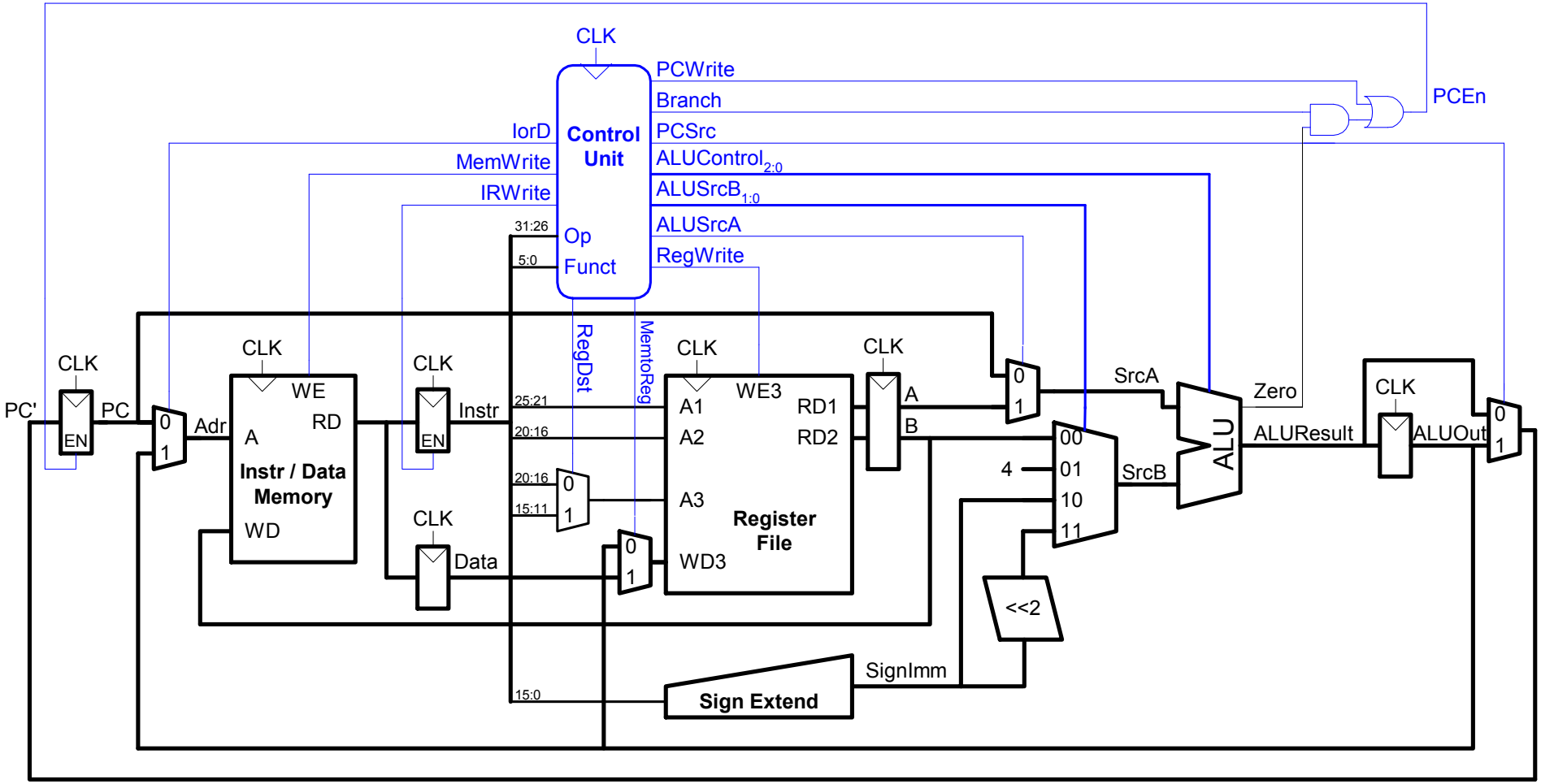


Multi-cycle Datapath: **beq**

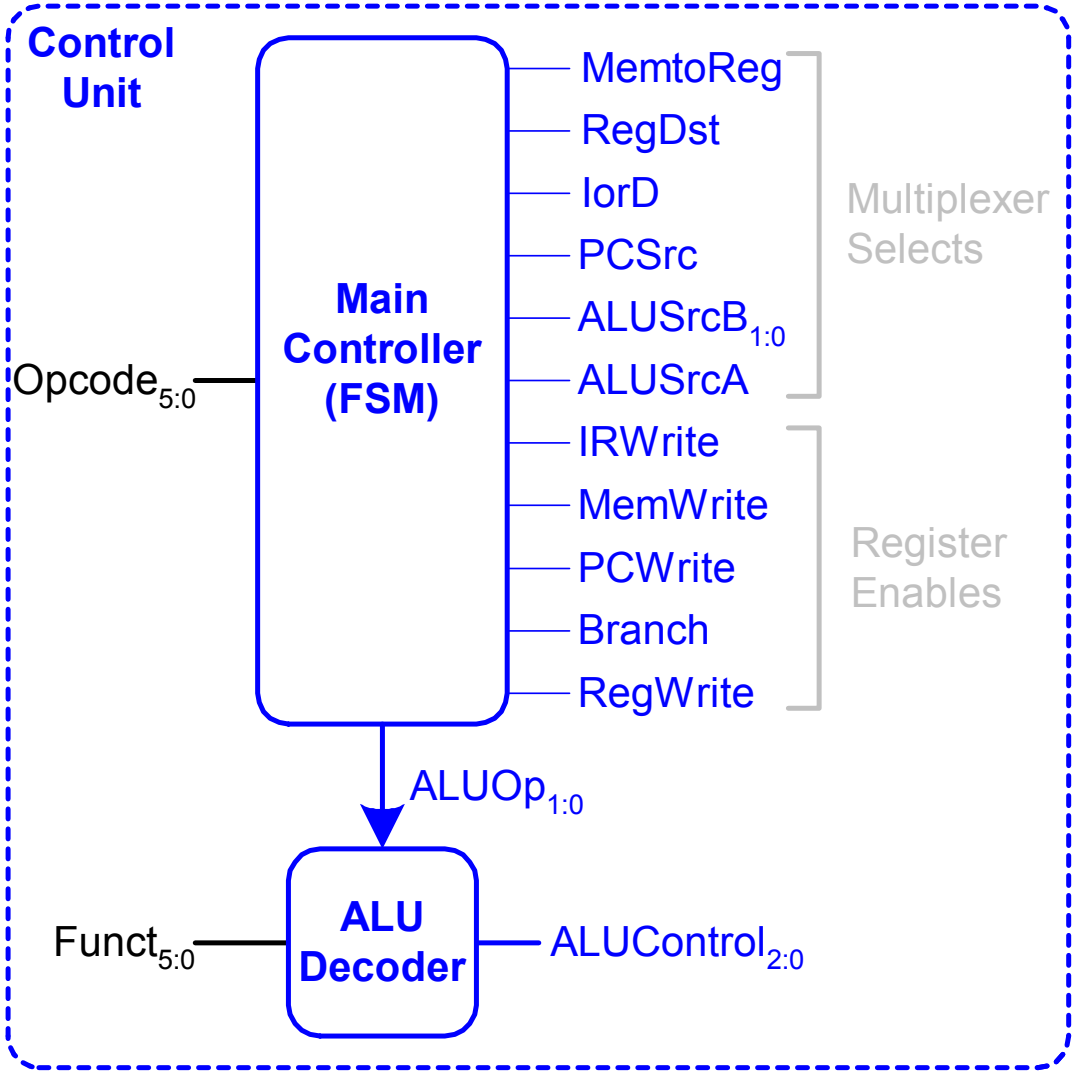
- **Determine whether values in rs and rt are equal**
  - Calculate branch target address:
    BTA = (sign-extended immediate << 2) + (PC+4)



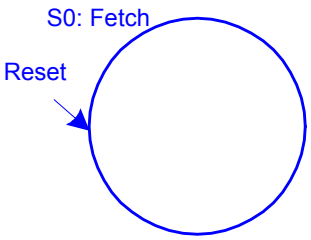
Complete Multi-cycle Processor

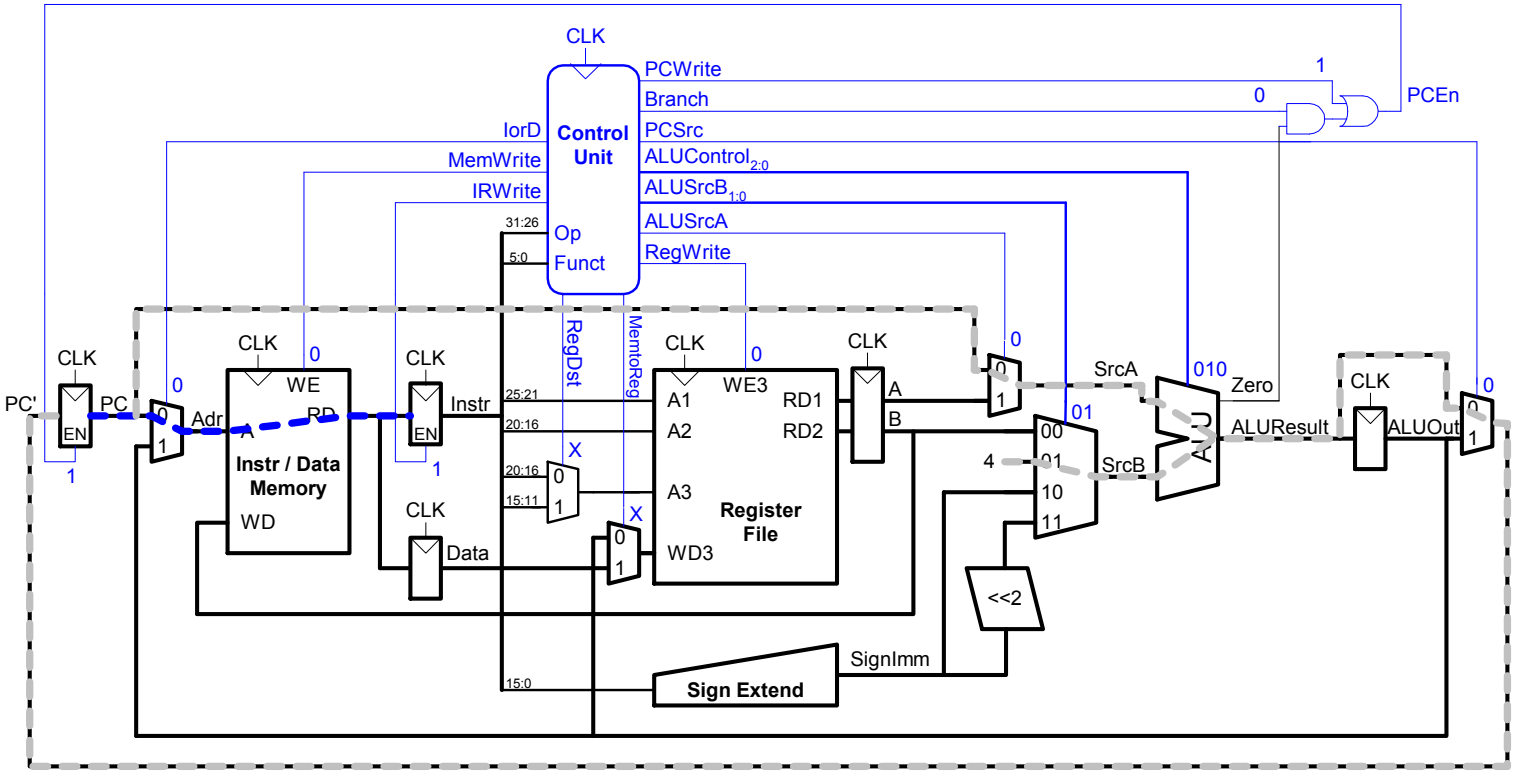


Control Unit

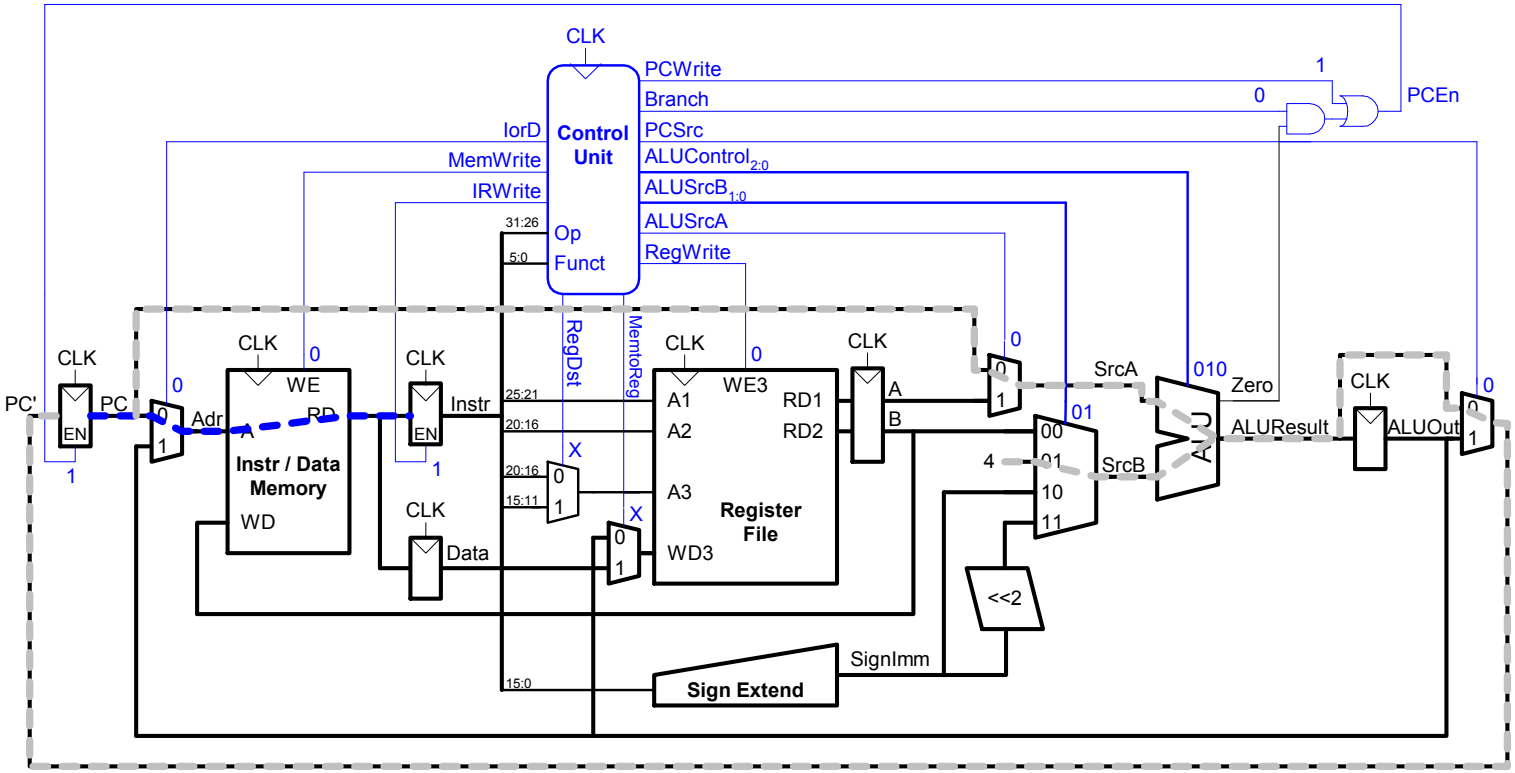


Main Controller FSM: Fetch

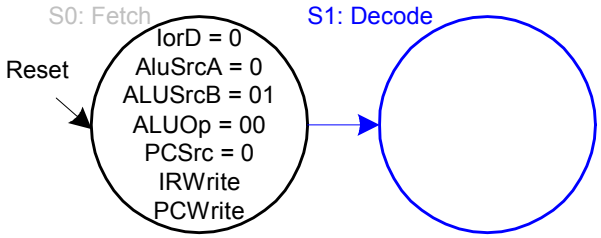


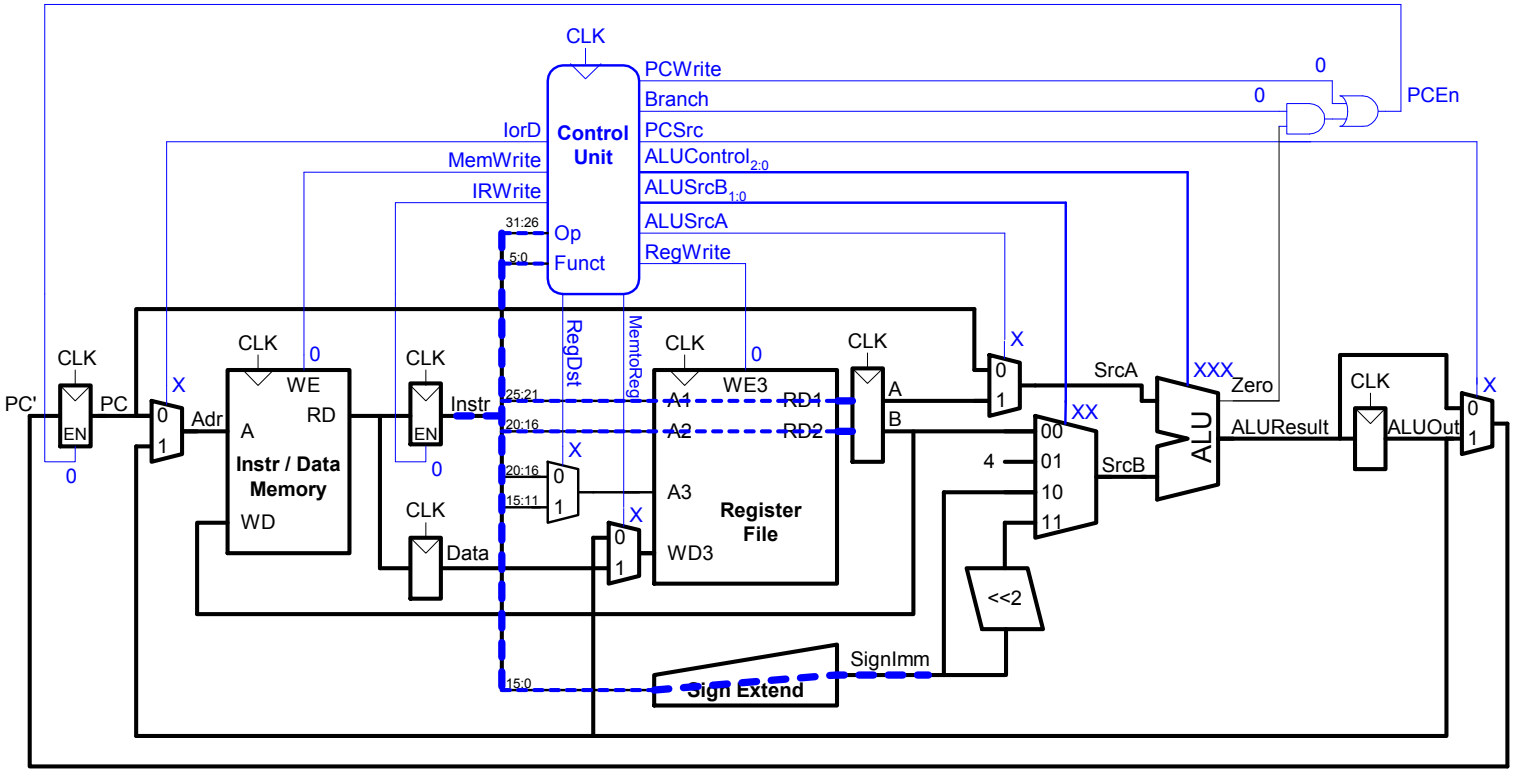


Main Controller FSM: Fetch

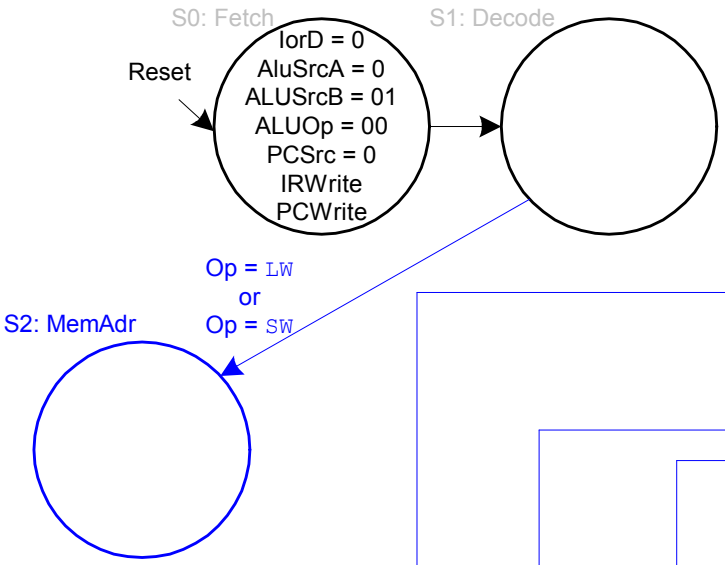


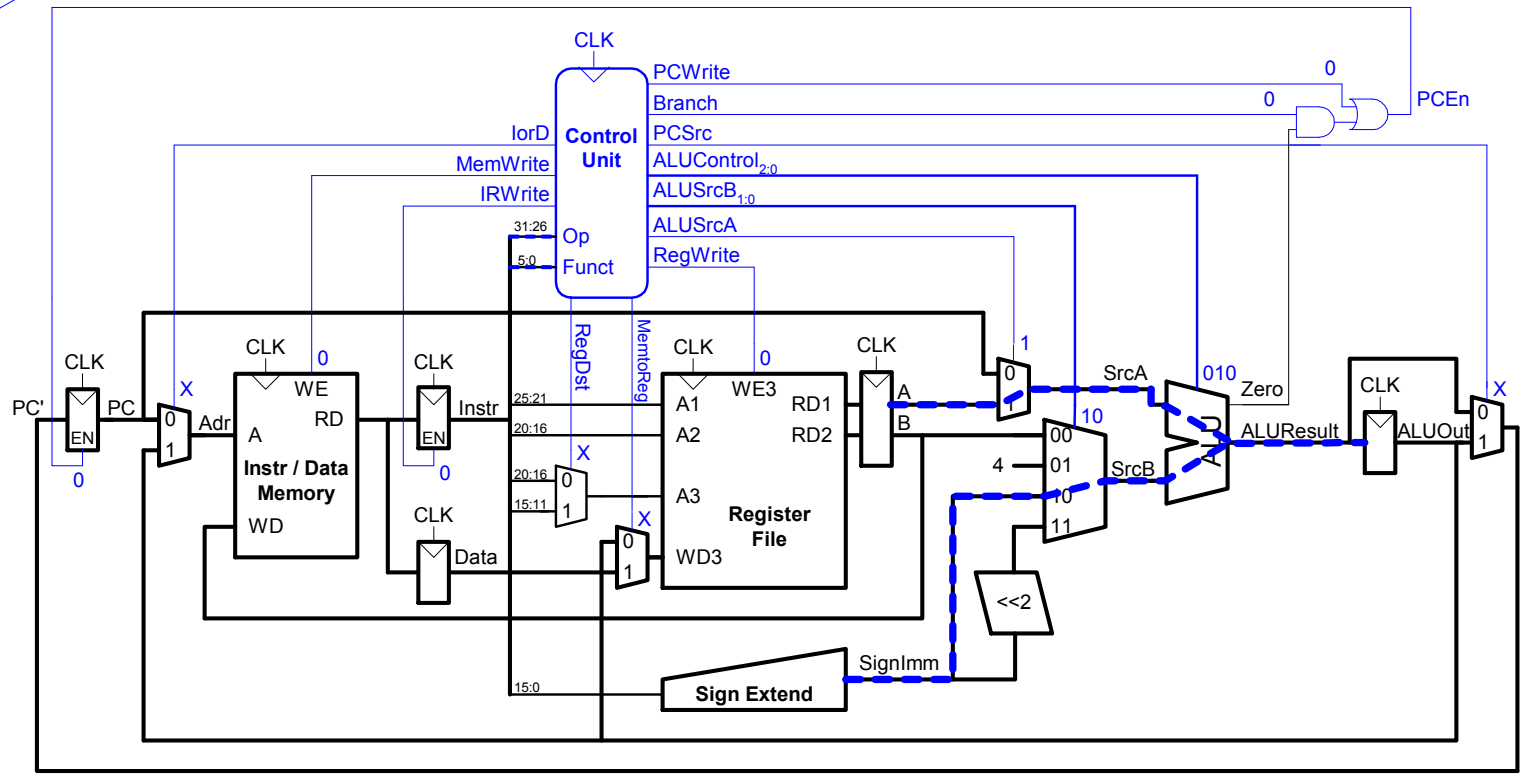
Main Controller FSM: Decode



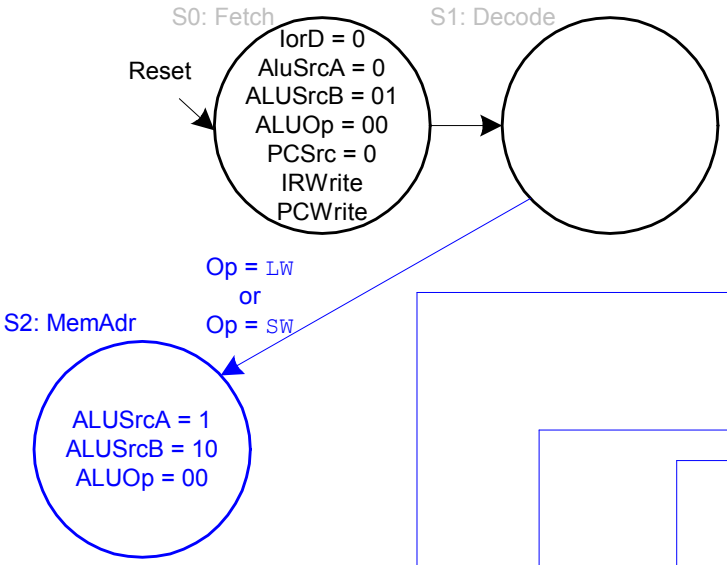


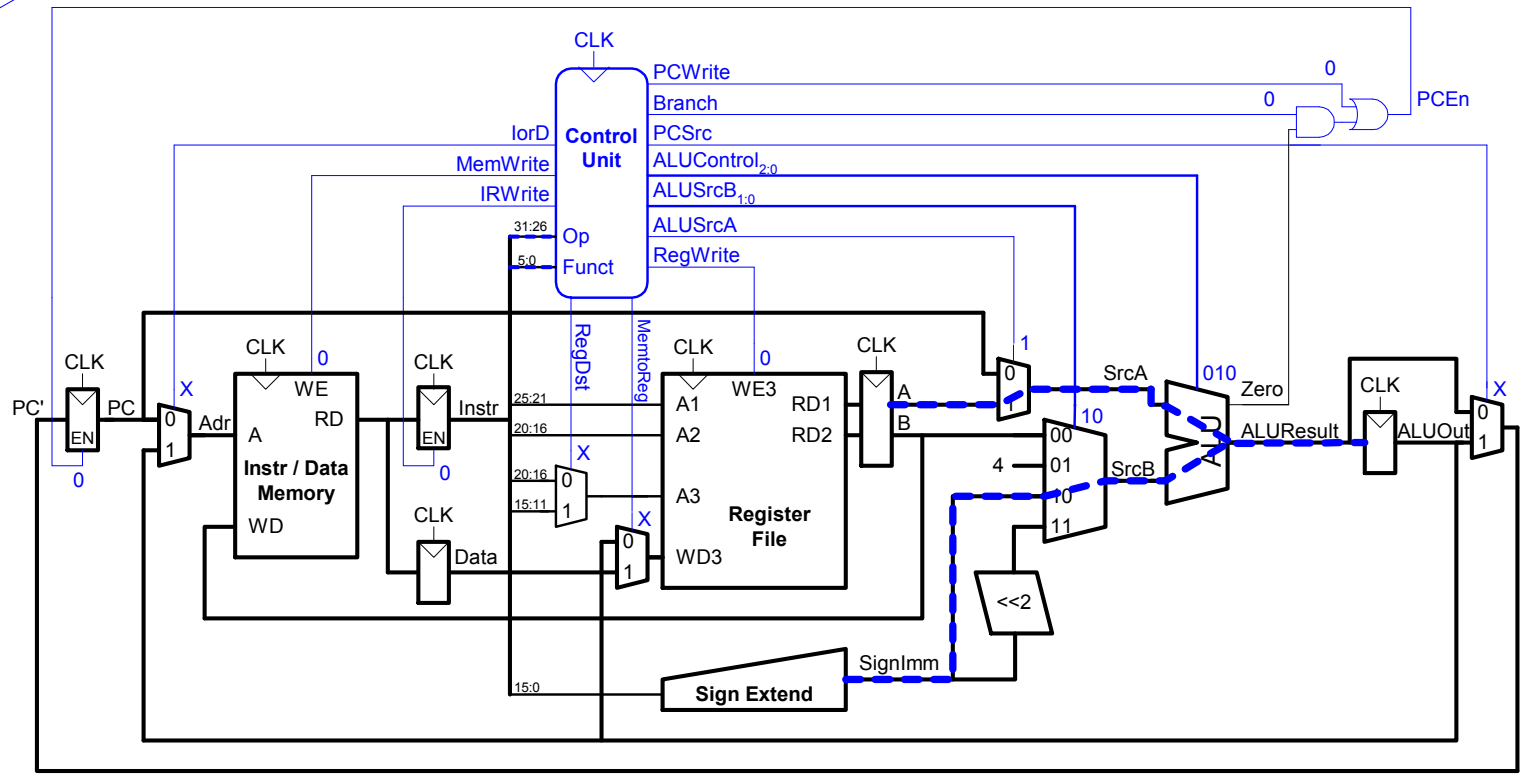
Main Controller FSM: Address Calculation



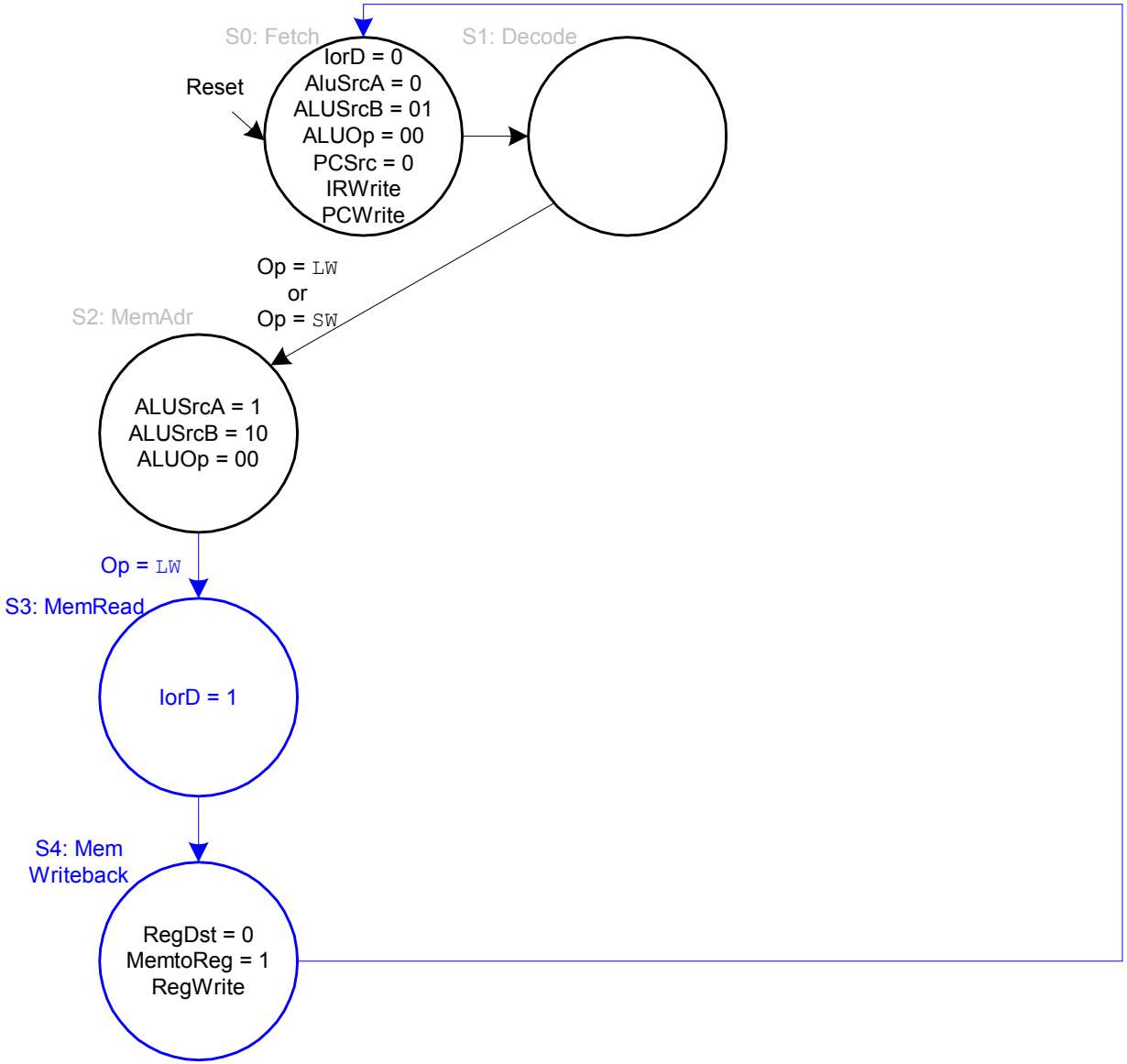


Main Controller FSM: Address Calculation

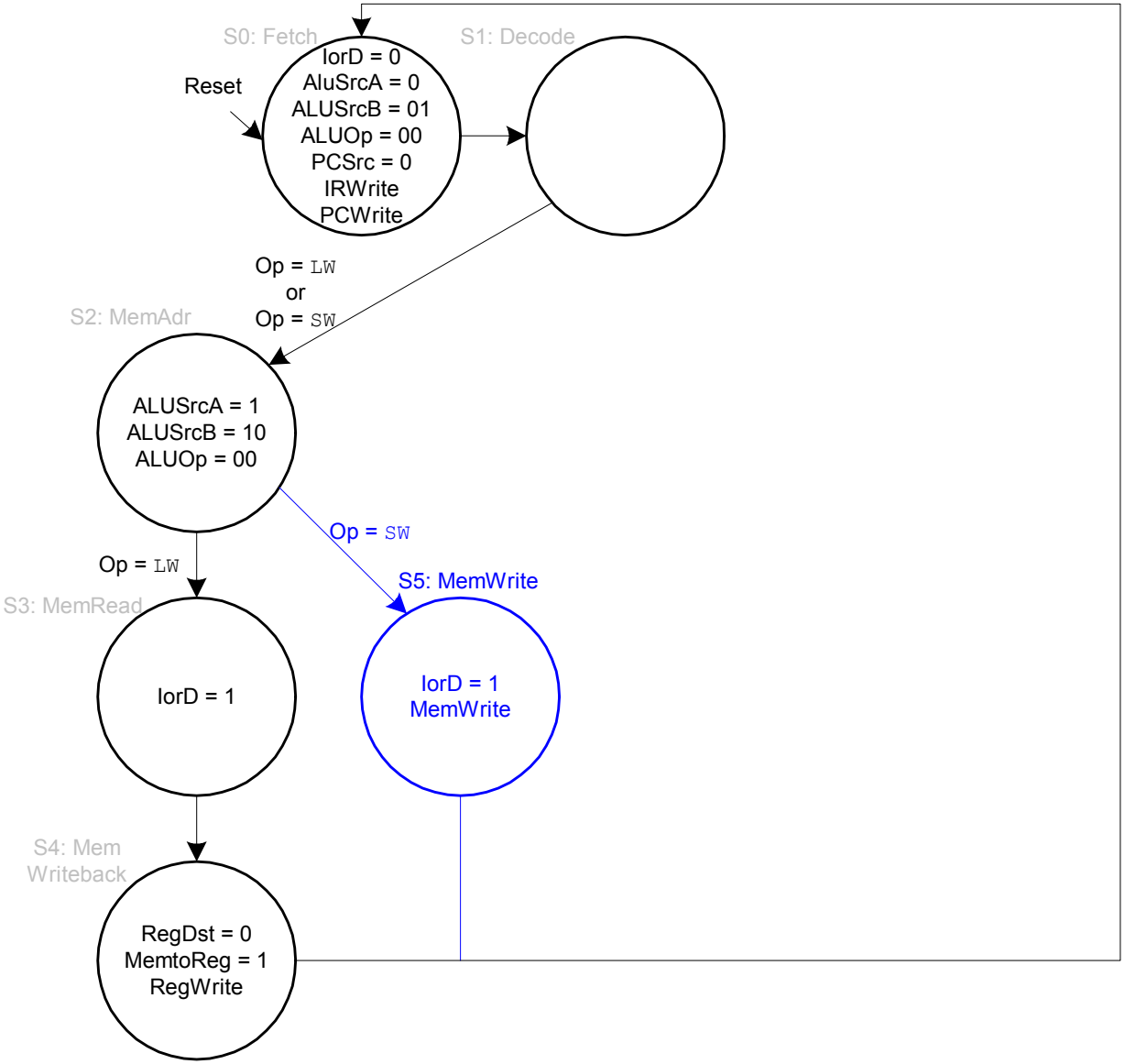




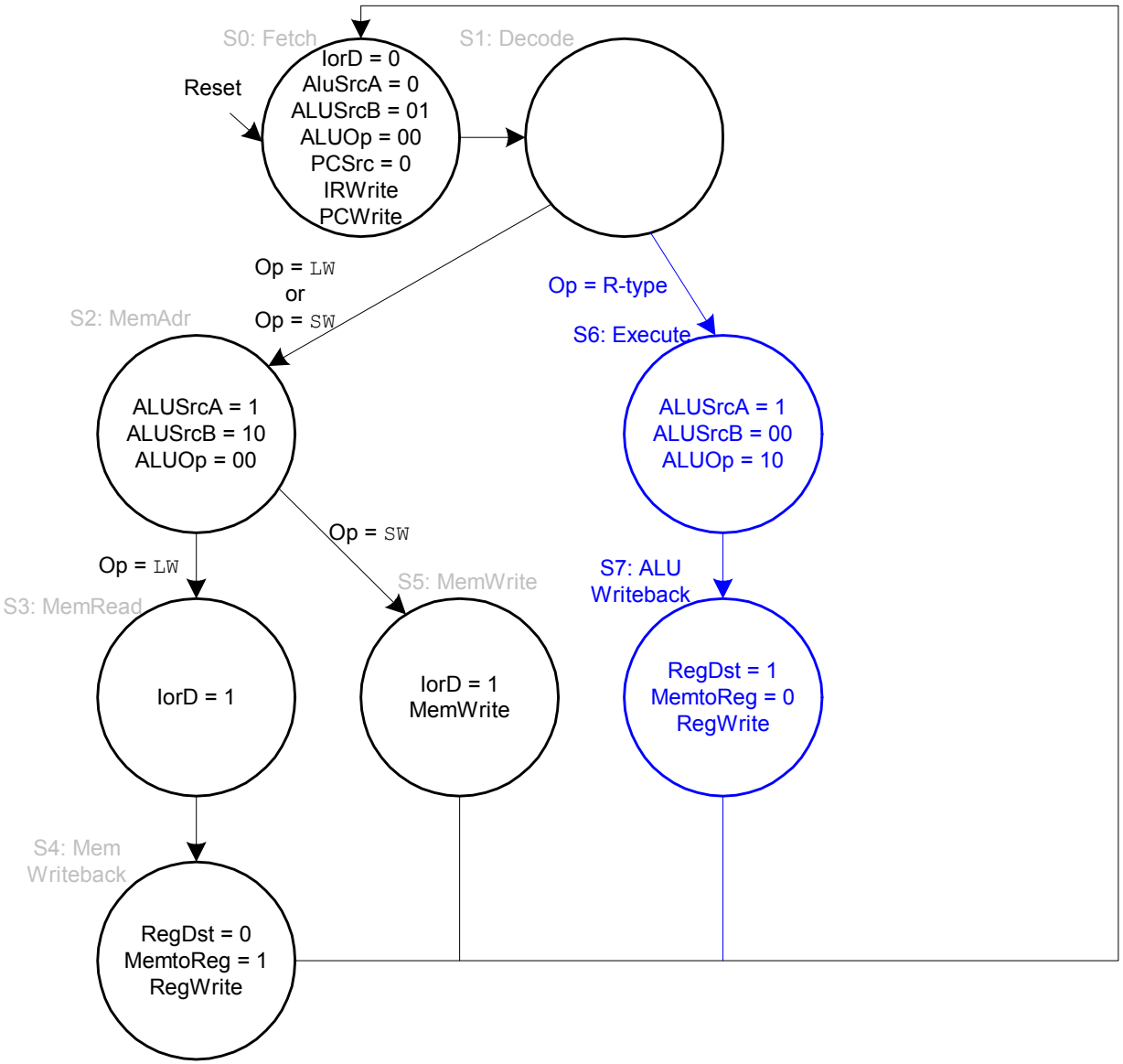
Main Controller FSM: `lw`



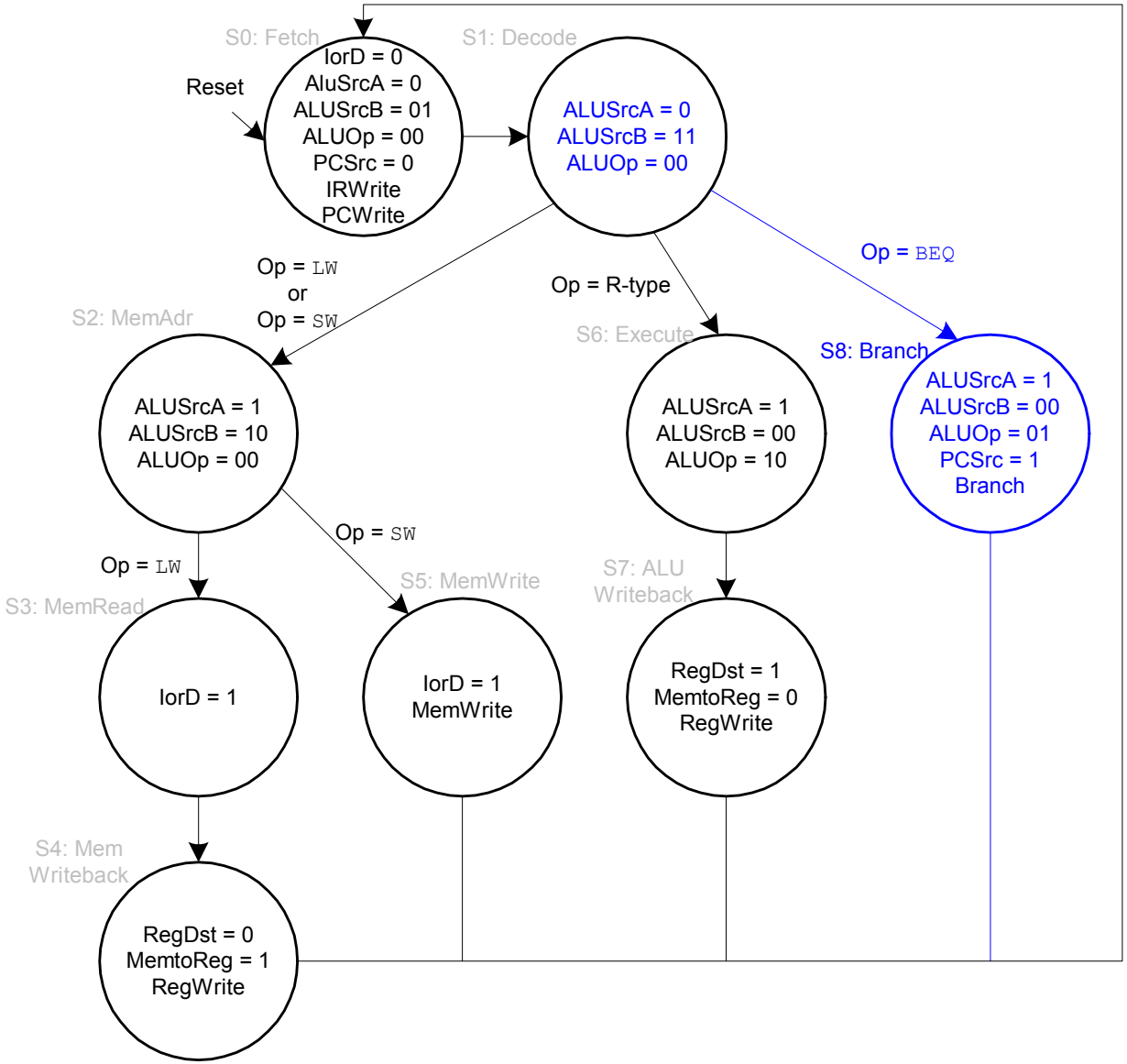
Main Controller FSM: `sw`



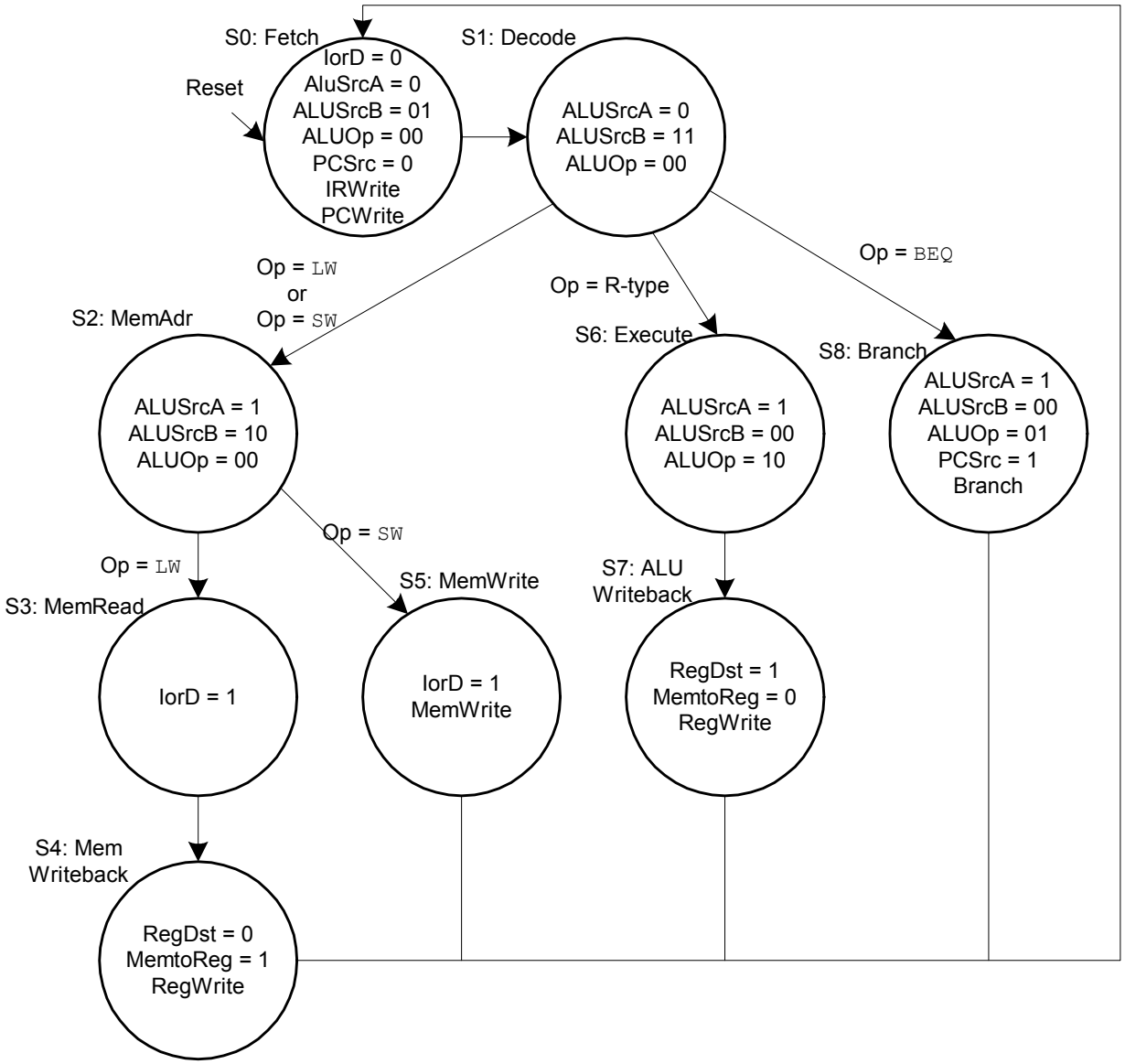
Main Controller FSM: R-Type



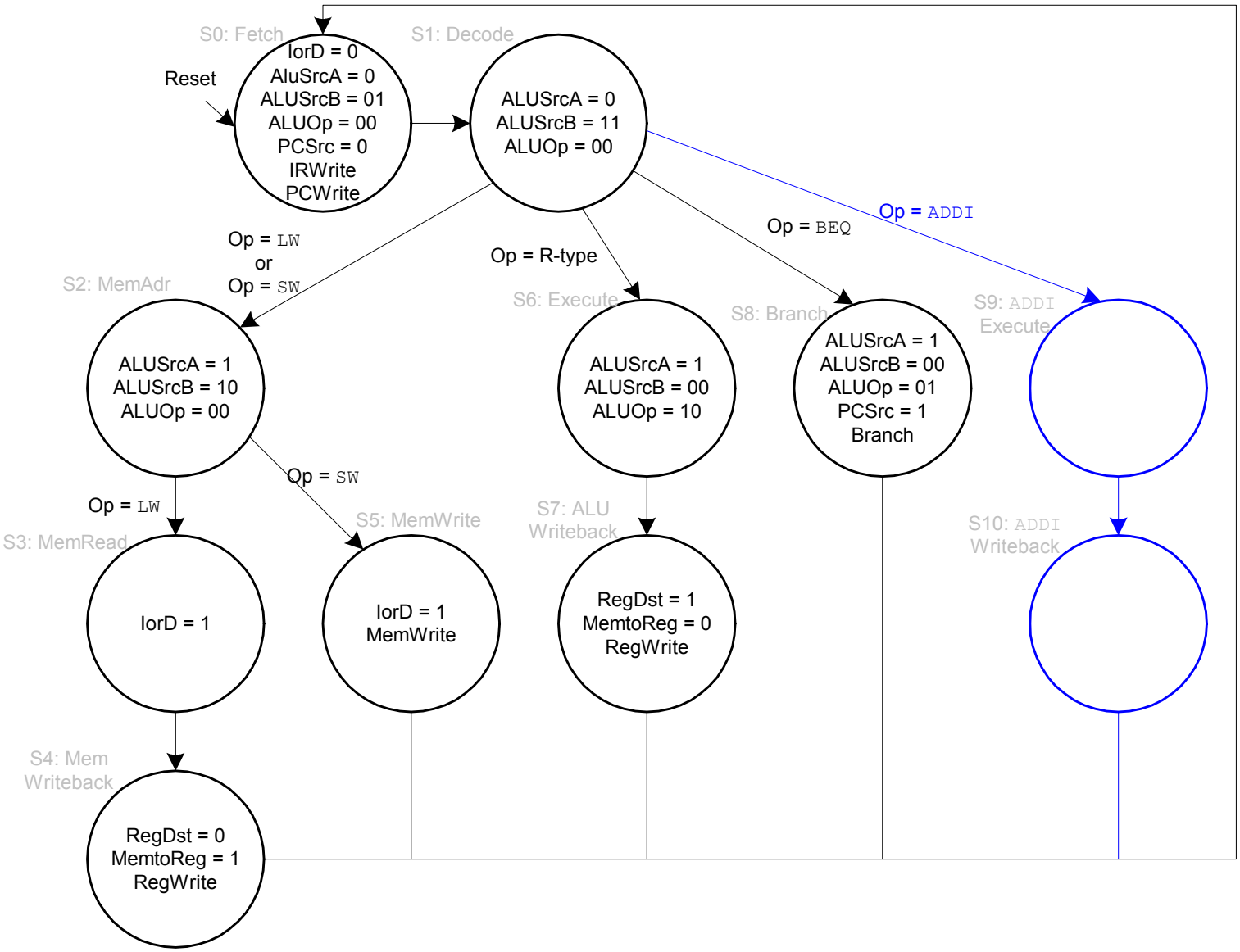
Main Controller FSM: **beq**



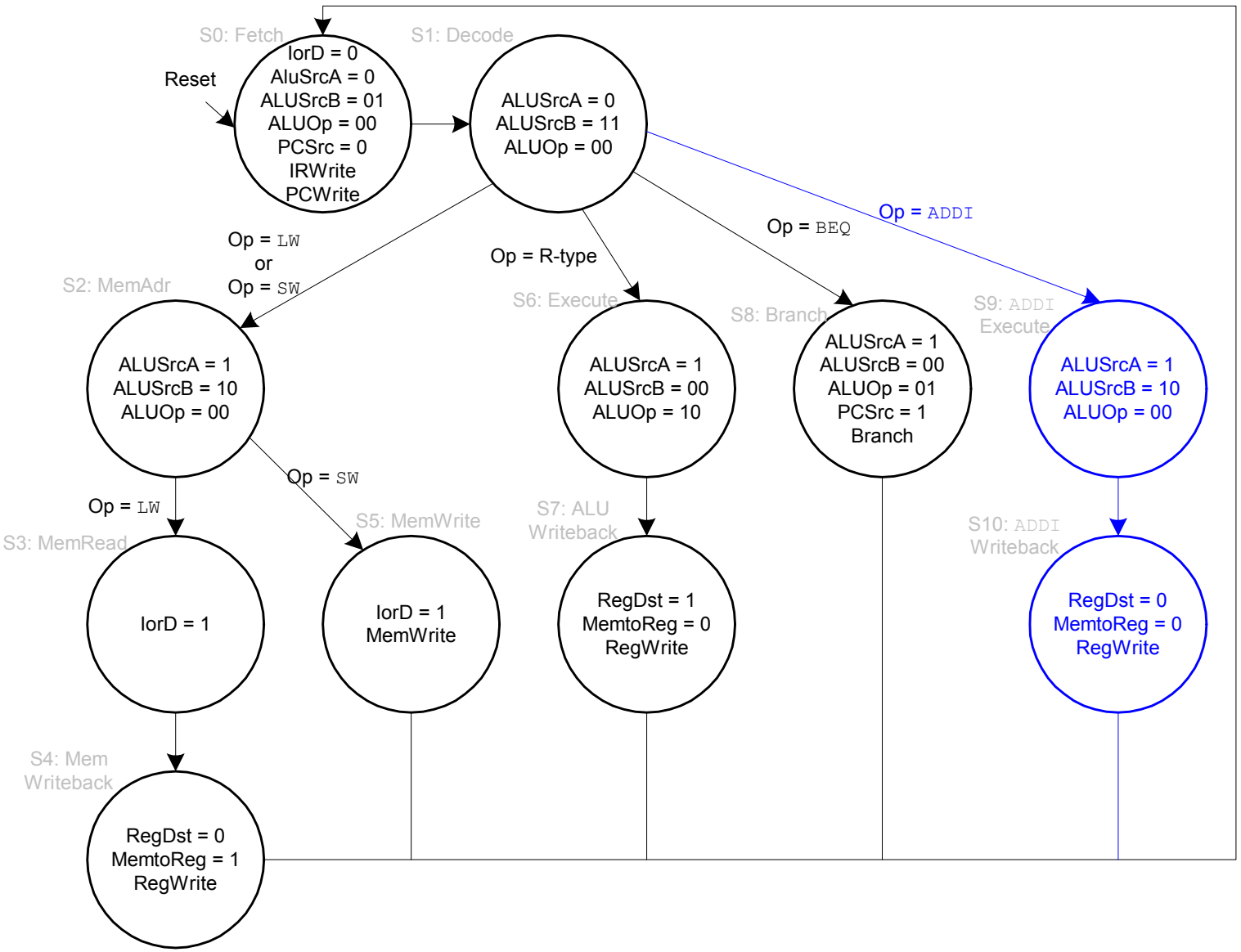
Complete Multi-cycle Controller FSM



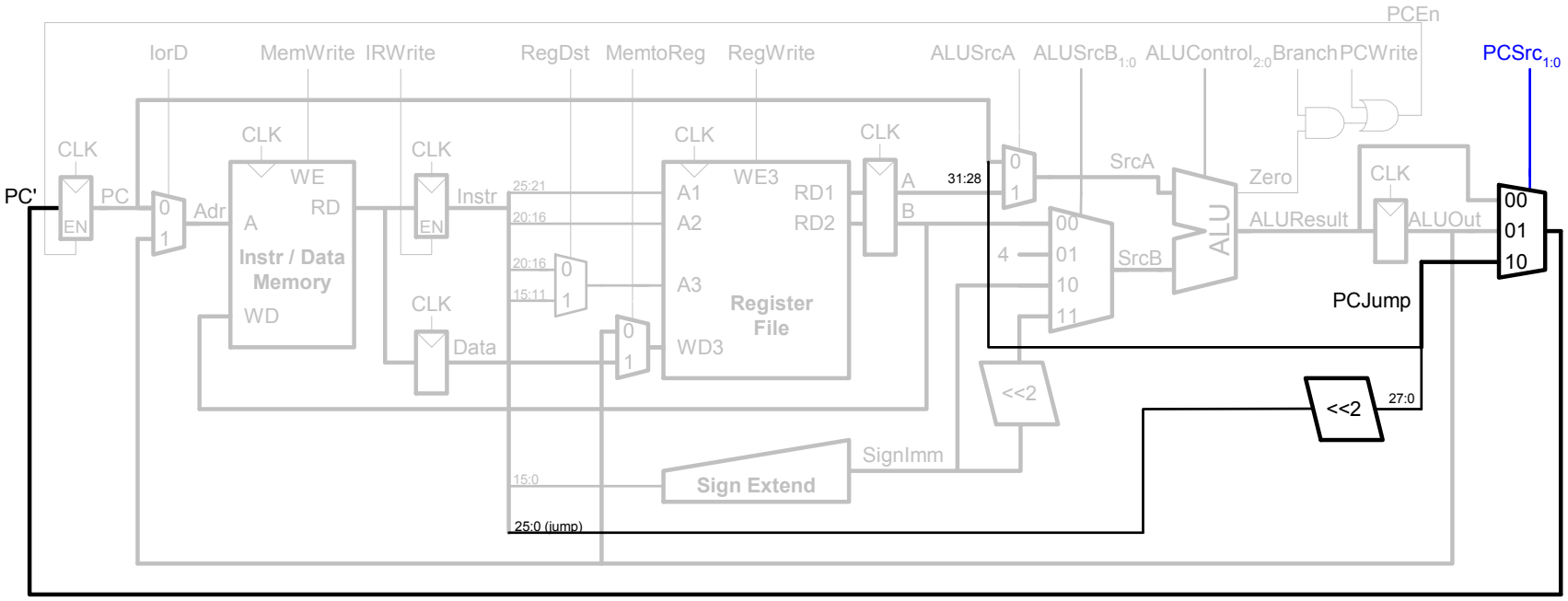
Main Controller FSM: **addi**



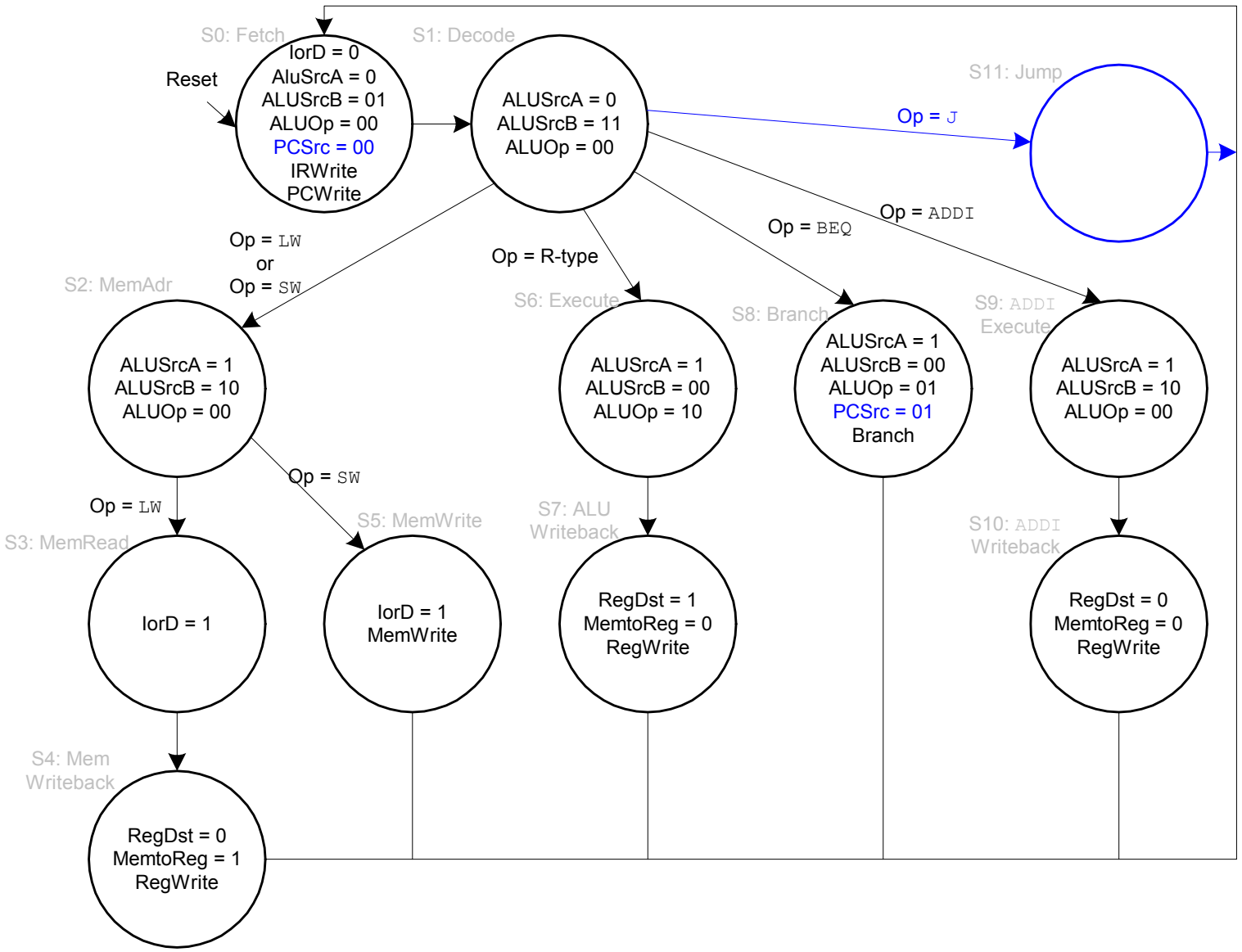
Main Controller FSM: **addi**



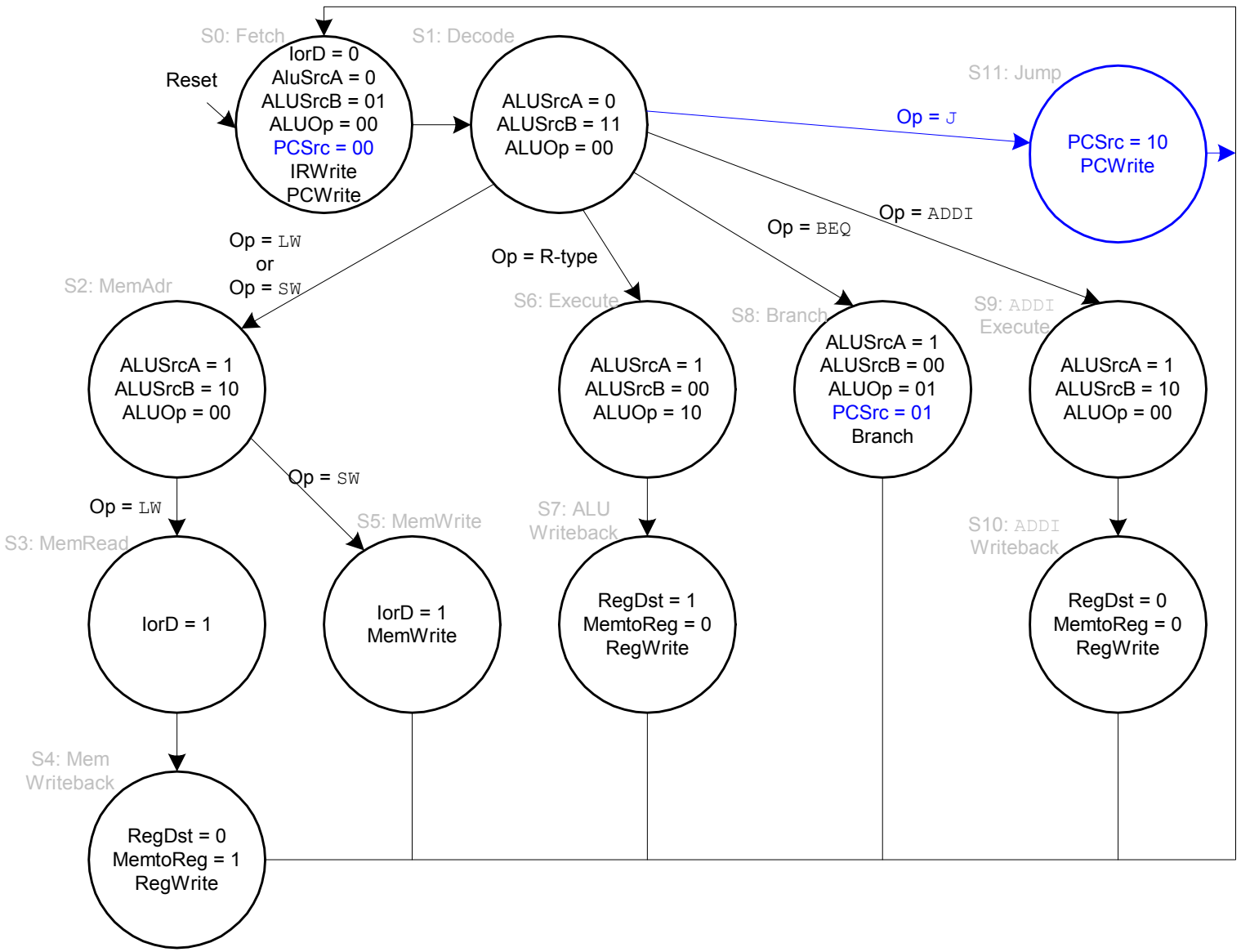
Extended Functionality: j



Control FSM: j



Control FSM: j



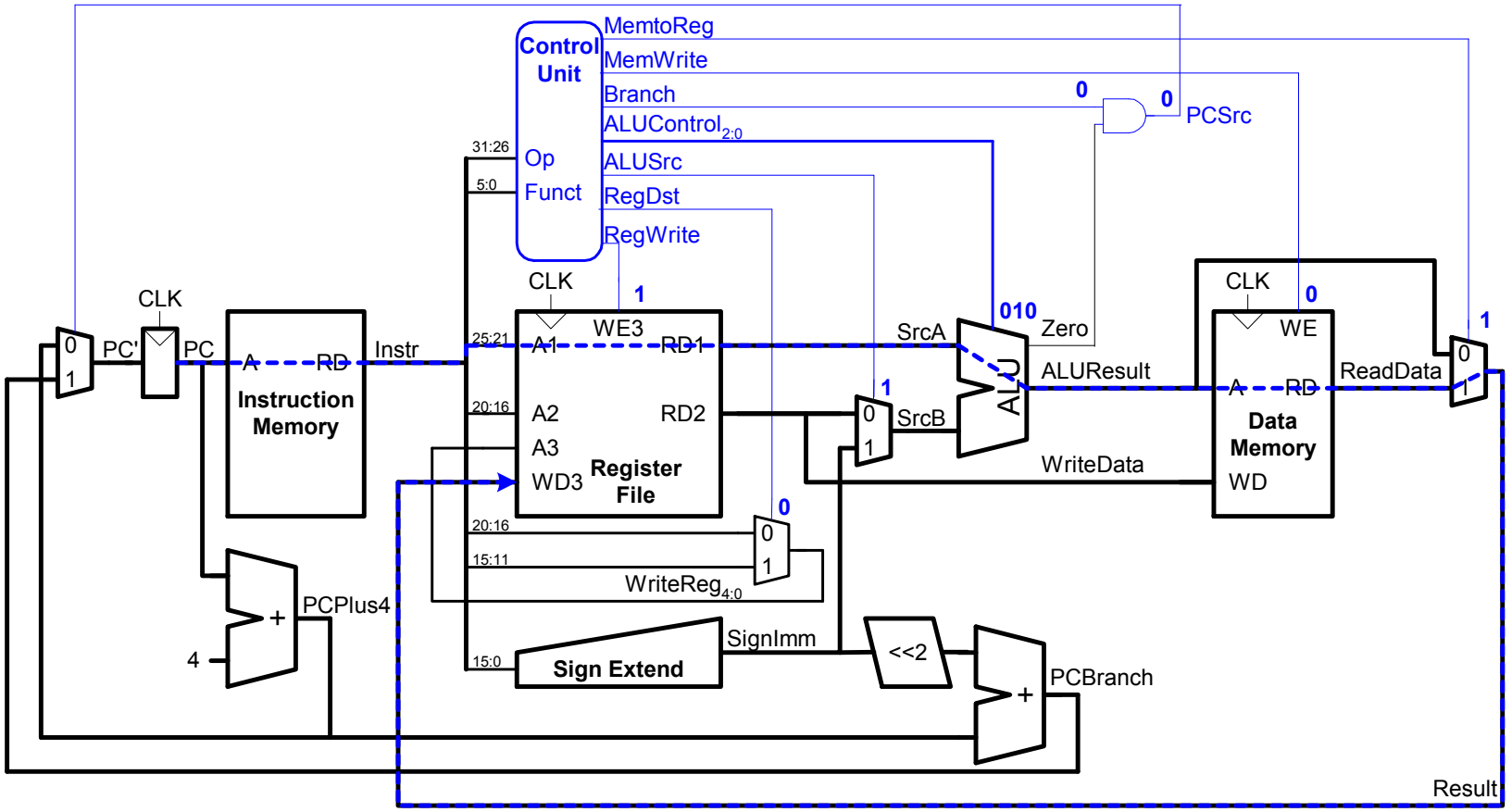
Summary of Multi-cycle Design

- Multi-cycle microarchitecture:
  + higher clock frequency
  + simpler instructions run faster
  + reuse expensive hardware across multiple cycles
- Need to store the intermediate results at the end of each clock cycle
  - Hardware overhead for registers
  - Register setup/hold overhead paid multiple times for an instruction

Performance Analysis

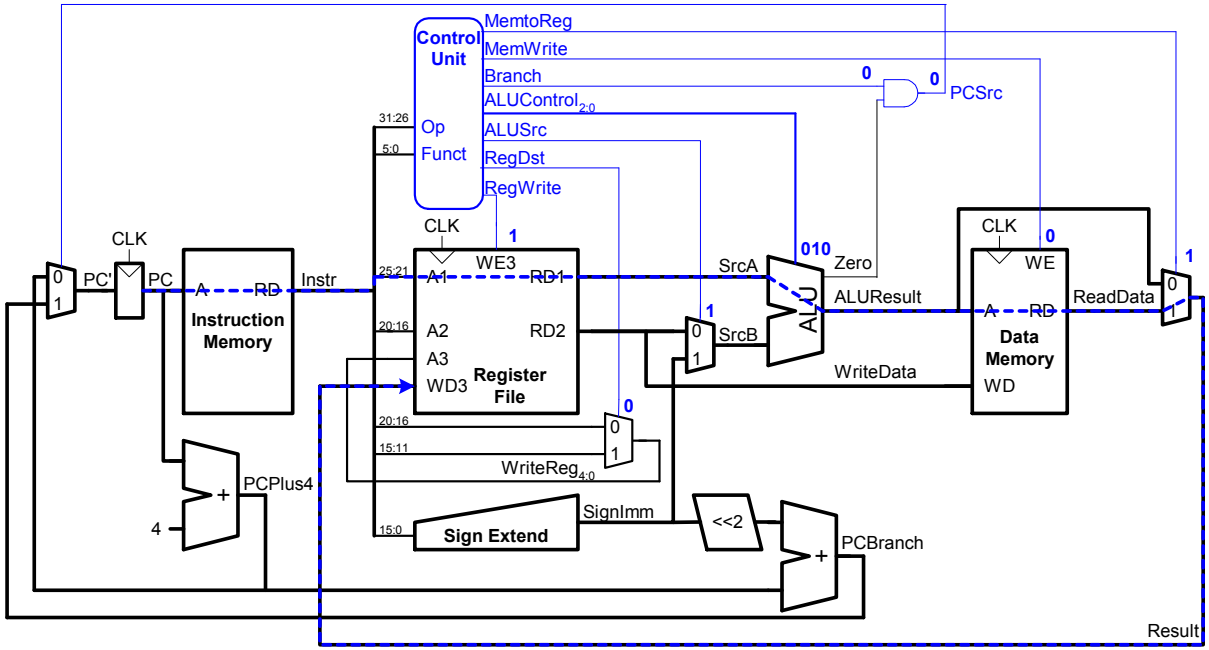
Single-Cycle Performance

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



Single-Cycle Performance

- Single-cycle critical path:
  - $T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$
- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$



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

- Example:

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

Single-Cycle Performance Example

- Example:

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

Execution Time = # instructions x CPI x T_c

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

= 92.5 seconds

Multi-Cycle Performance: CPI

- Instructions take different number of cycles:
  - 3 cycles: `beq, j`
  - 4 cycles: `R-Type, sw, addi`
  - 5 cycles: `lw`

Realistic?

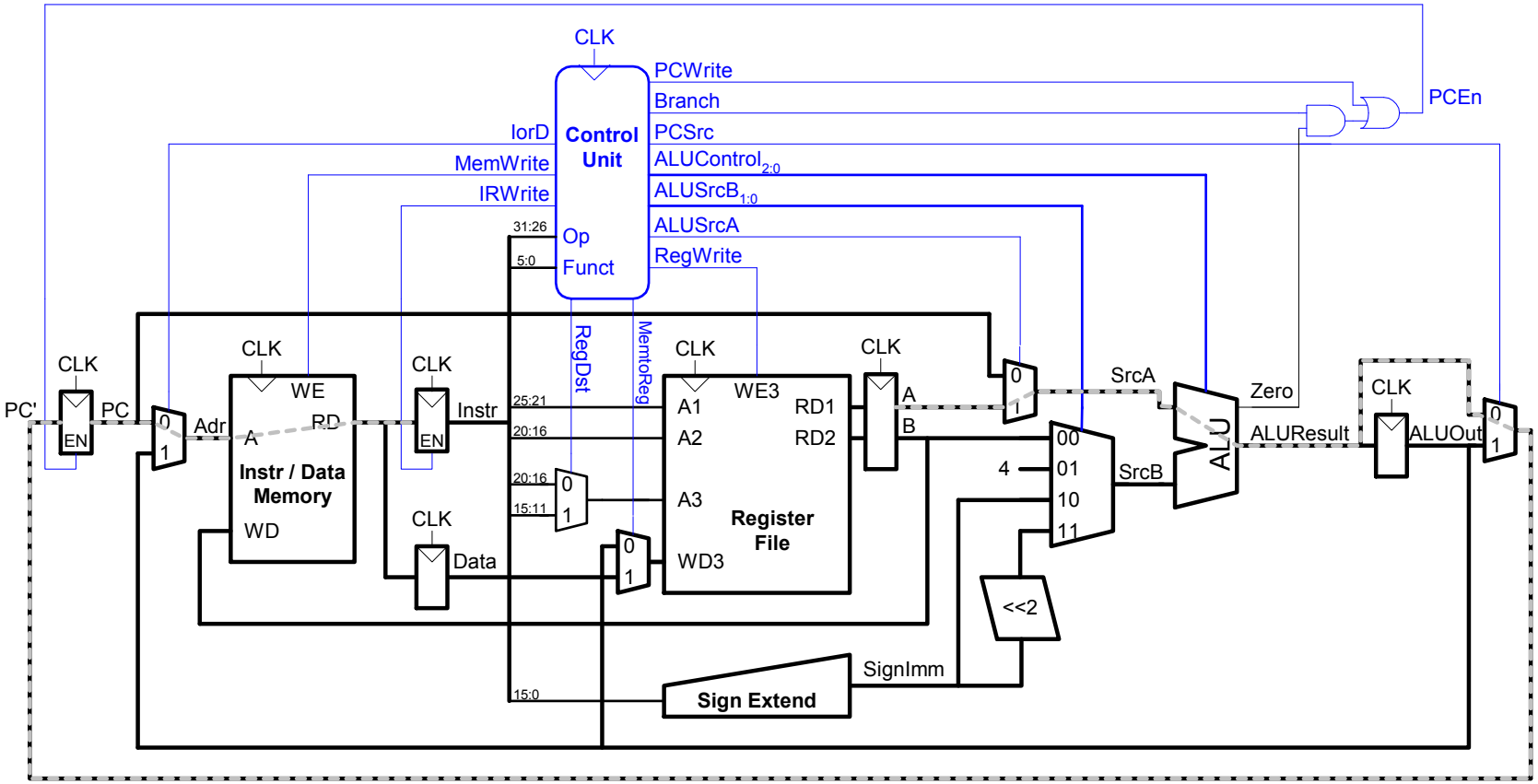
- CPI is weighted average, e.g. SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

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

Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

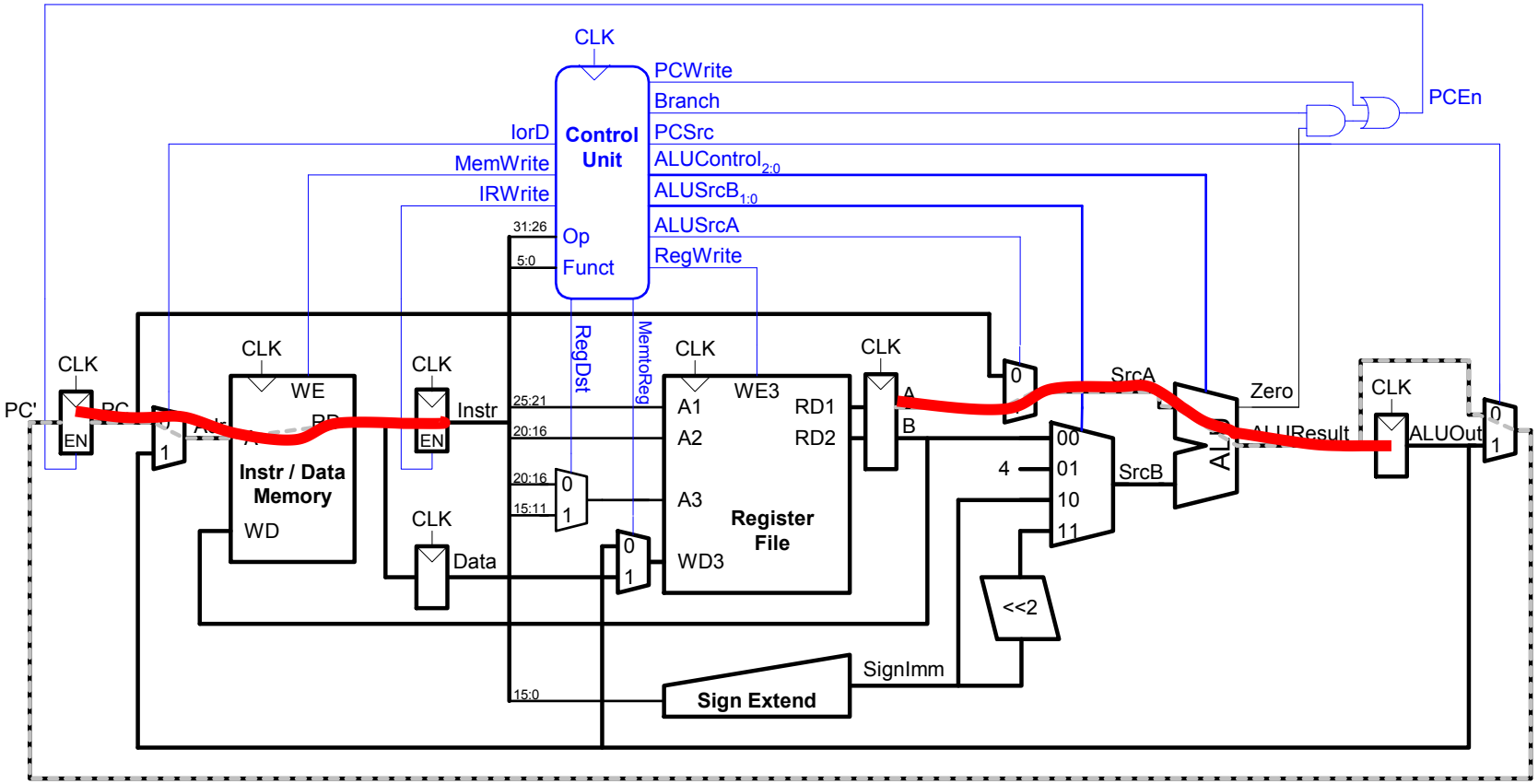
  T_c =



Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

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



Multi-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

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

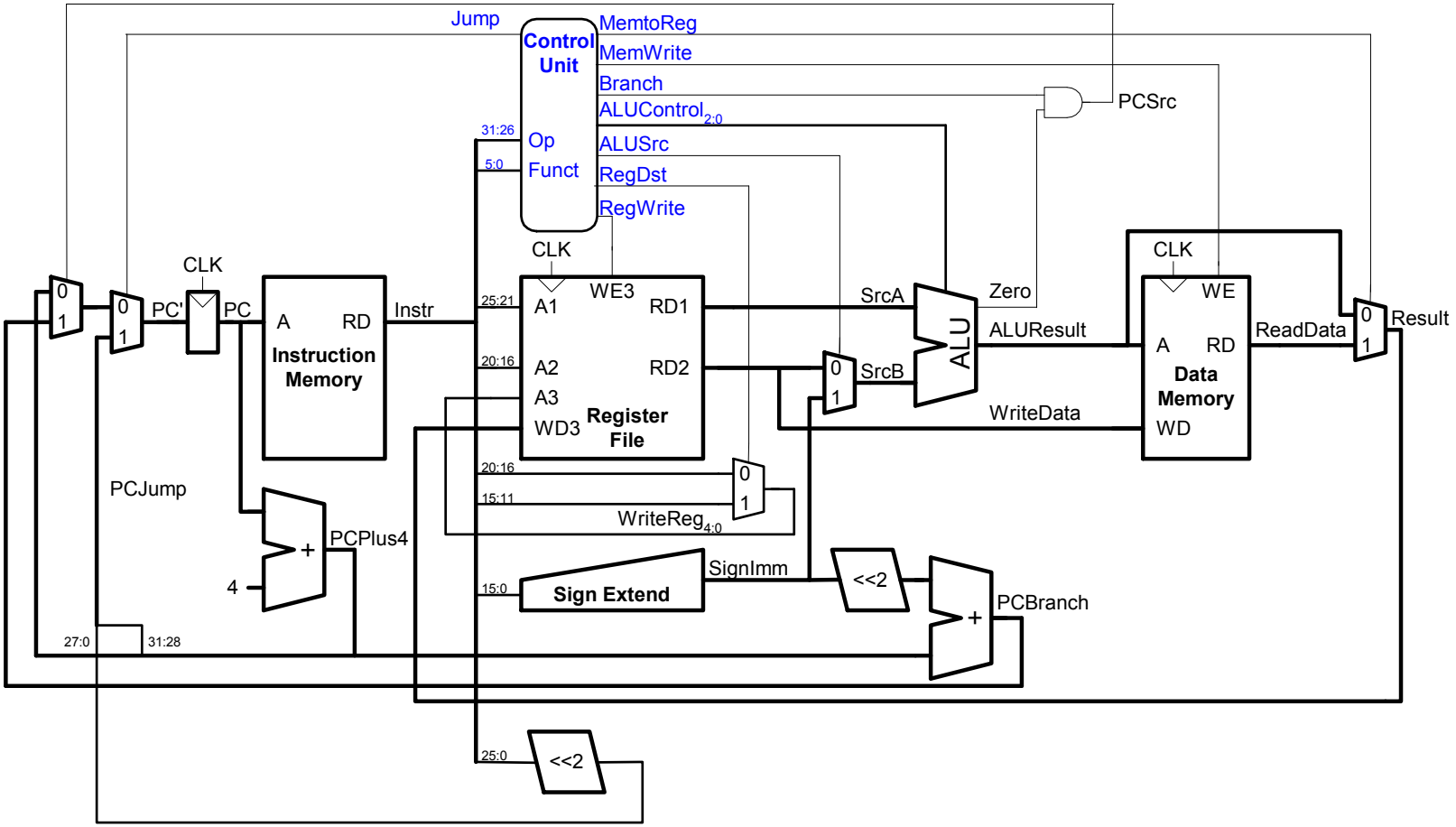
$$= [30 + 25 + 250 + 20] \text{ ps}$$

$$= 325 \text{ ps}$$

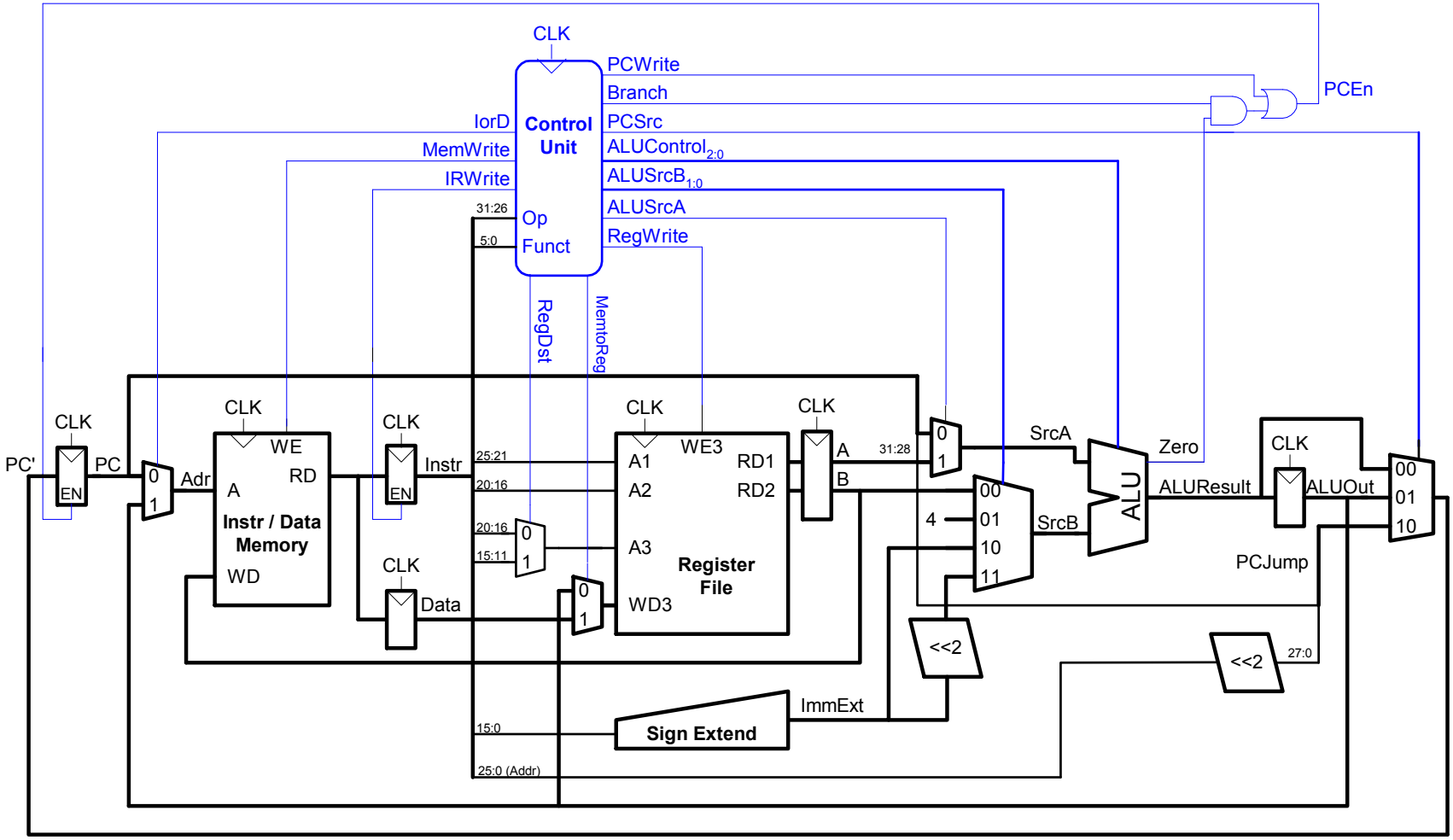
Multi-Cycle Performance Example

- For a program with 100 billion instructions executing on a multi-cycle MIPS processor
  - CPI = 4.12
  - T_c = 325 ps
- *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?
- Did we break the stages in a balanced manner?
- Overhead of register setup/hold paid many times
- How would the results change with different assumptions on memory latency and instruction mix?

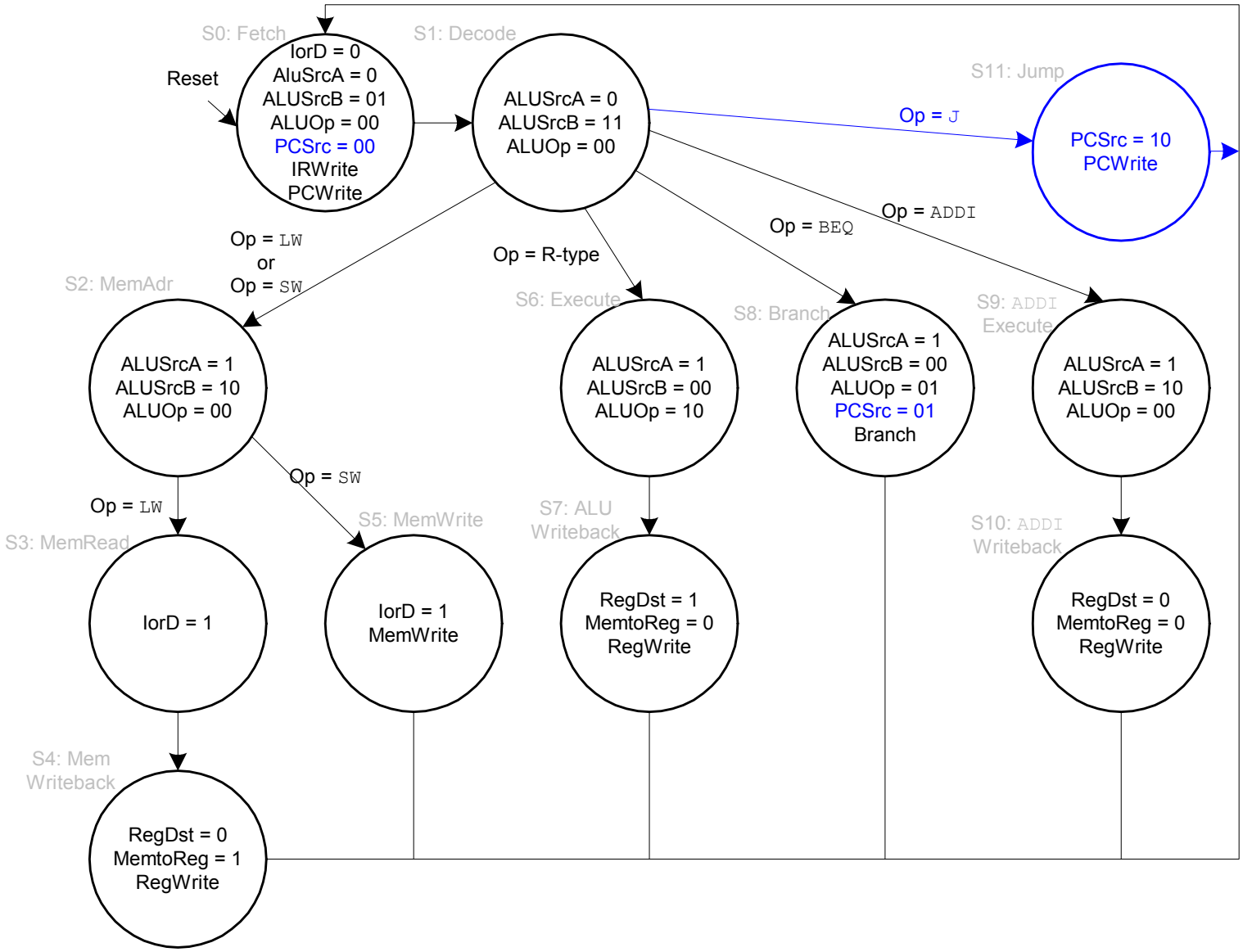
Review: Single-Cycle MIPS Processor



Review: Multi-Cycle MIPS Processor



Review: Multi-Cycle MIPS FSM



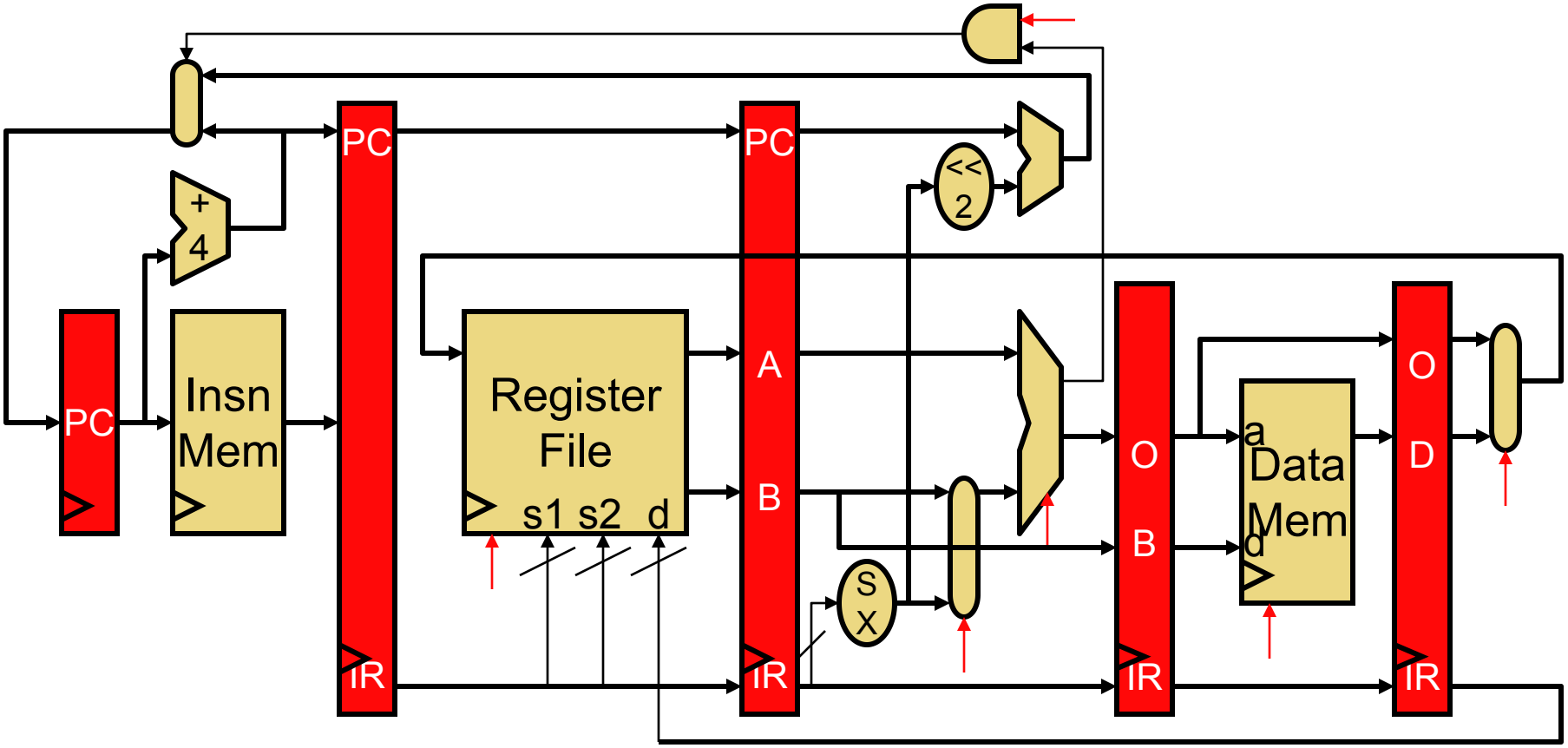
What is the shortcoming of this design?

What does this design assume about memory?

What If Memory Takes > One Cycle?

- Stay in the same "memory access" state until memory returns the data
- "Memory Ready?" bit is an input to the control logic that determines the next state

Foreshadowing: Pipelined Datapath



- Split datapath into multiple stages
  - Assembly line analogy
  - 5 stages results in up to 5x clock & performance improvement