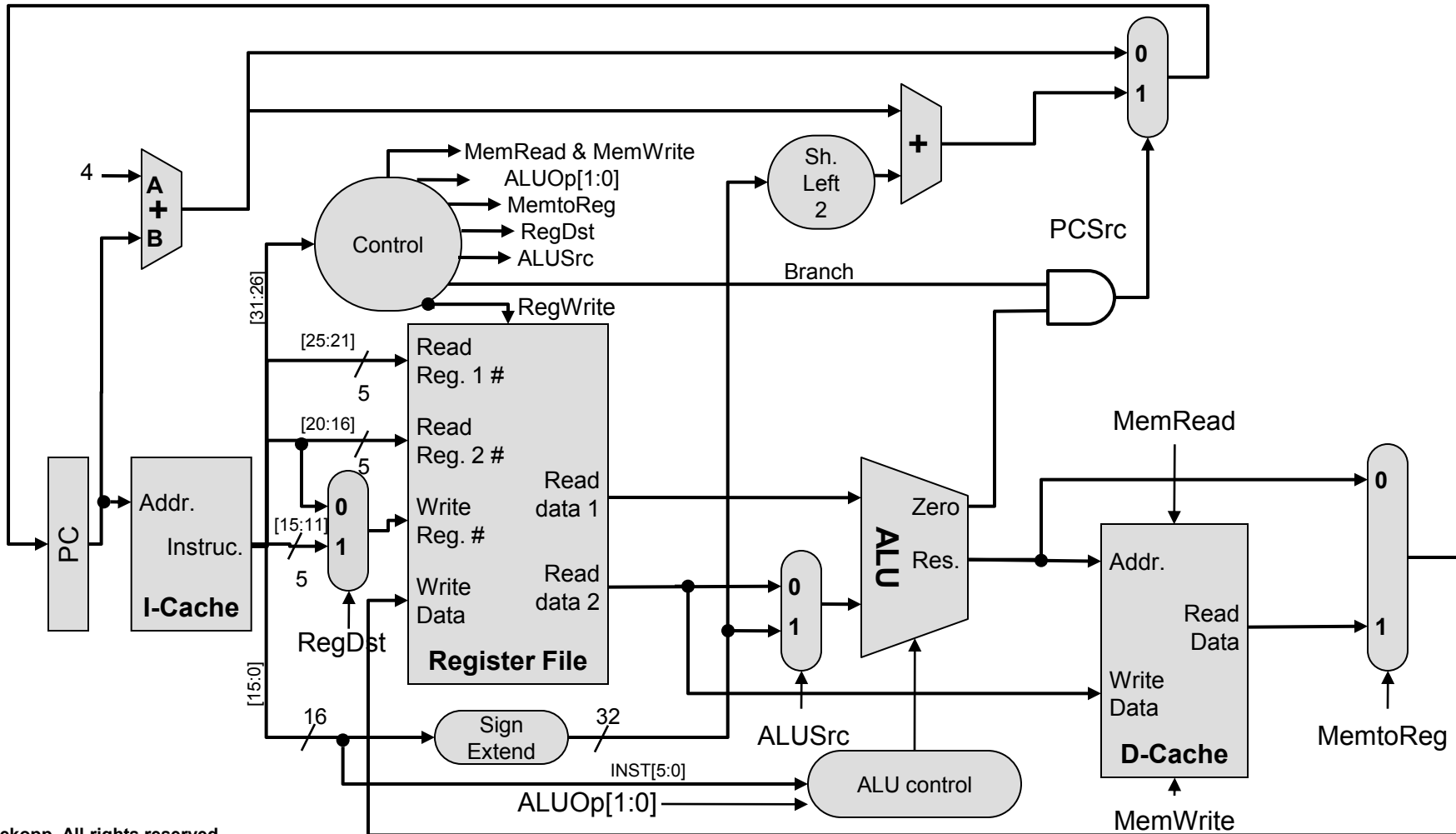


# Multi-Cycle CPU Organization

## Datapath and Control

# Single-Cycle CPU Datapath

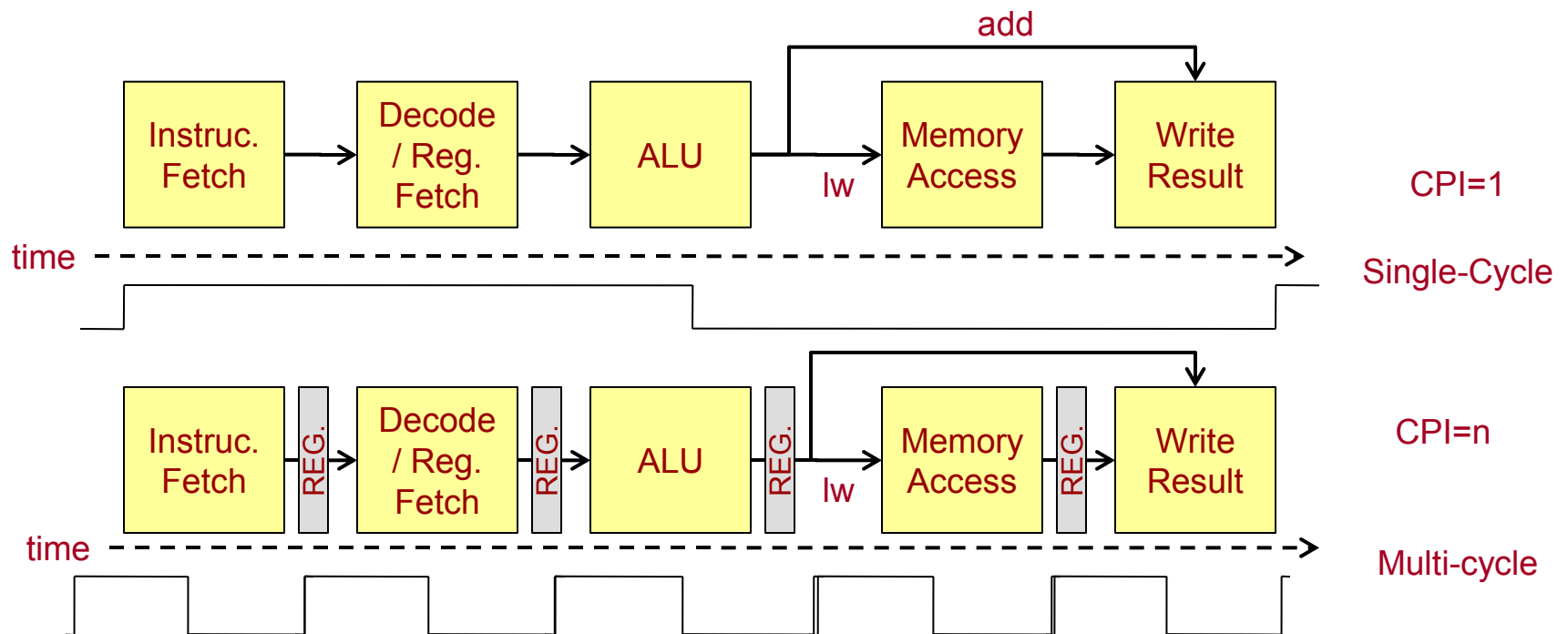


# Multicycle CPU Implementation

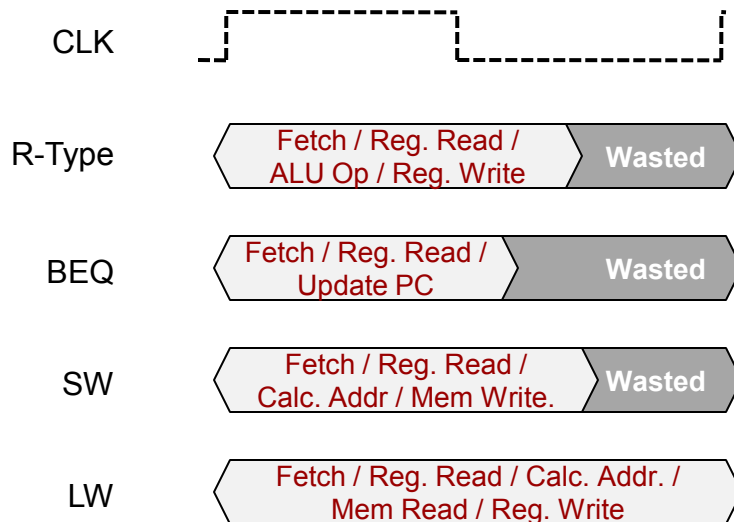
- Single cycle CPU sets the clock period according to the longest instruction execution time
- Rather than making every instruction “pay” the worst case time, why not make each instruction “pay” just for what it uses
  - Example: Pay Parking
    - Parking meters: Cost proportional to time spent
    - Flat fee parking lot: One price no matter the time
- Multicycle CPU implementation breaks instructions into smaller, shorter sub-operations
  - Clock period according to the longest sub-operation
- Instructions like ADD or Jump with few sub-operations will take fewer cycles while more involved instructions like LW will take more cycles

# Single vs. Multi-Cycle CPU

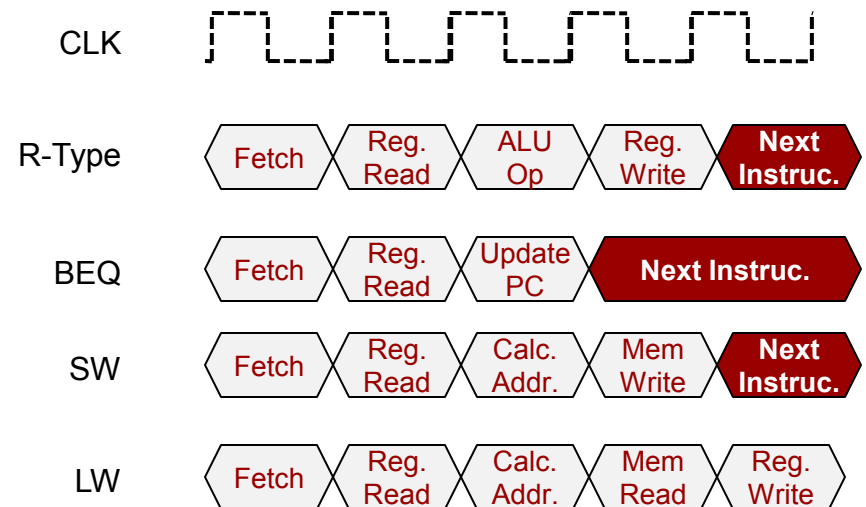
- Single Cycle CPU design makes all instructions wait for the full clock cycle and the cycle time is based on the SLOWEST instruction
- Multi-cycle CPU will break datapath into sub-operations with the cycle time set by the longest sub-operation. Now instructions only take the number of clock cycles they need to perform their sub-ops.



# Single-/Multi-Cycle Comparison



In single-cycle implementations, the clock cycle time must be set for the longest instruction. Thus, shorter instructions waste time if they require a shorter delay.

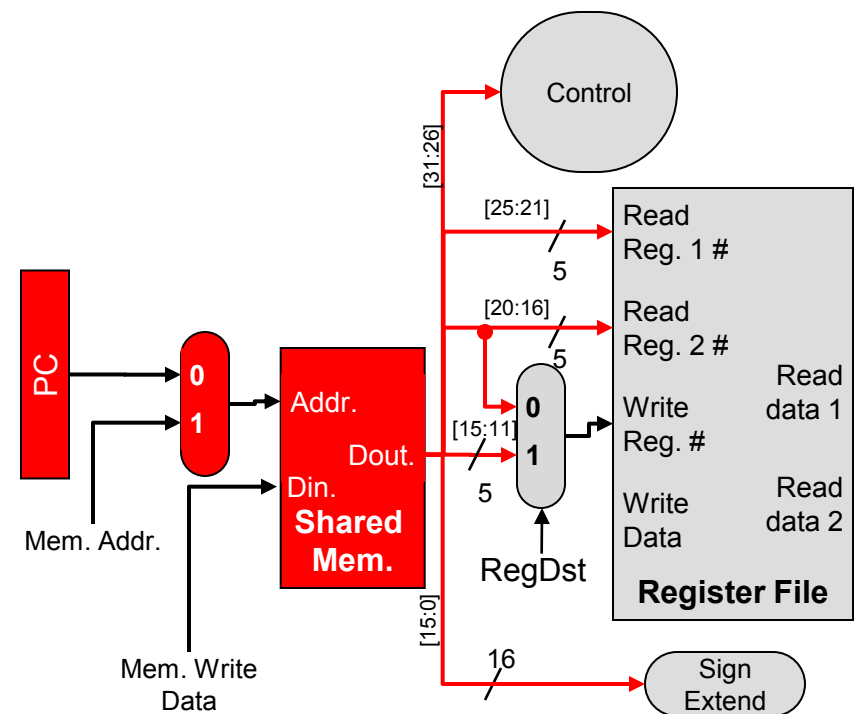


In multi-cycle CPU, each instruction is broken into separate short (and hopefully time-balanced) sub-operations. Each instruction takes only the clock cycles needed, allowing shorter instructions to finish earlier and have the next instruction start.

# Sharing Resources in Single-Cycle

- Single-cycle CPU required multiple:
  - Adders/ALU
  - Memories (instruc. & data)

because all operations occurred during a single clock cycle which limited our control of the flow of data signals



# Sharing Resources in Multicycle CPU

- Any resource needed in different clock cycles (time steps) can be shared/re-used
  - 1 ALU and 2 adders in single-cycle CPU can be replaced by just the one ALU (& some muxes)
  - Separate instruction and data memories can be replaced with a single memory

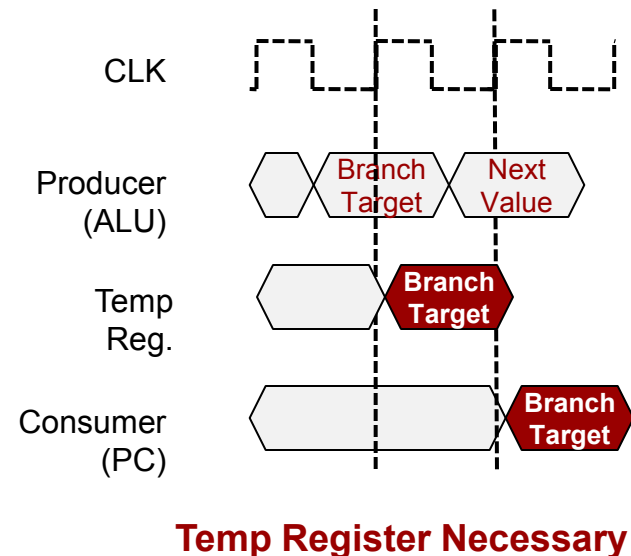
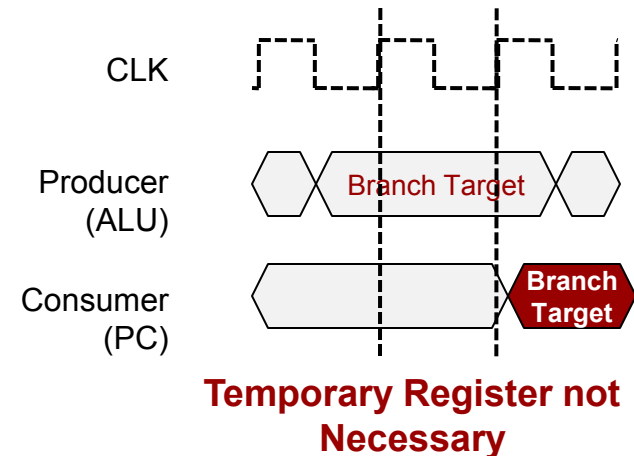
# Temporary Registers

- Another implication of a multi-cycle implementation is that data may be produced in one cycle (step) but consumed in a later cycle
- This may necessitate saving/storing that value in a temporary register
  - If the producer can keep producing across multiple cycles (i.e. is not needed for another subsequent operation) then we can do without the temporary register
  - If the producer is needed for another operation in a subsequent cycle, then we must save the value it produced in a temporary register



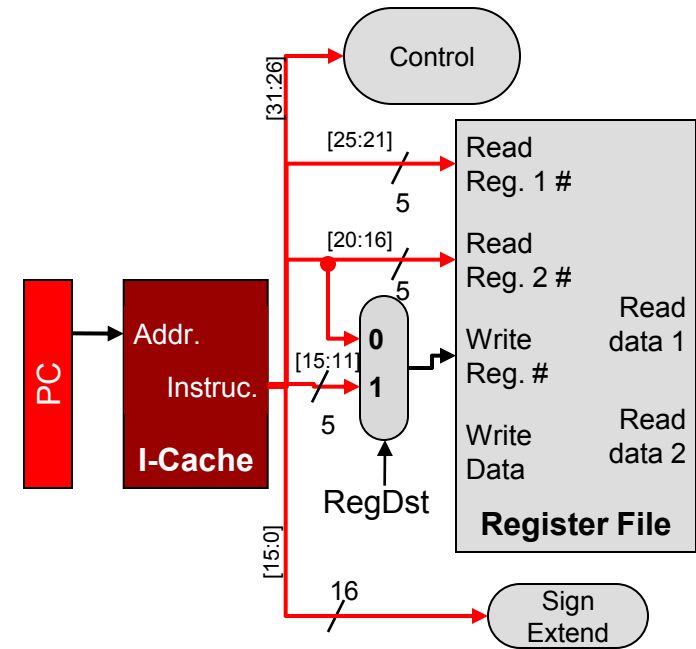
# Temporary Registers

- If the producer can keep producing across multiple cycles (i.e. is not needed for another subsequent operation) then we can do without the temporary register
- If the producer is needed for another operation in a subsequent cycle, then we must save the value it produced in a temporary register

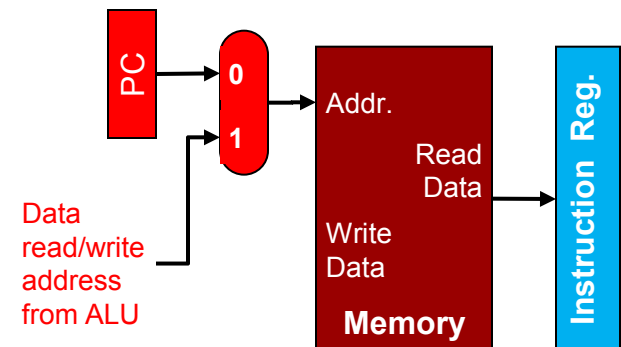


# Instruction Register

- Do we need a register to store instruction
  - In single-cycle CPU: NO
    - Separate instruction memory + stable PC value during entire cycle = Stable instruction value for entire execution of instruction
  - In multi-cycle CPU: YES
    - Single memory may need to be used to read or write data after reading instruction. We must buffer / save the instruction somewhere (i.e. IR)



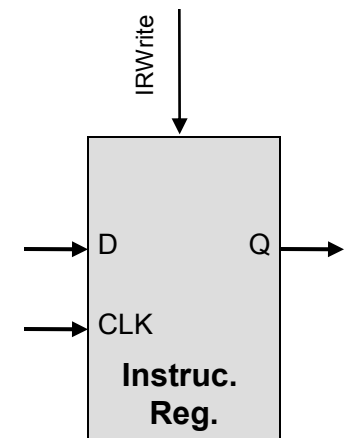
Single-Cycle CPU Datapath



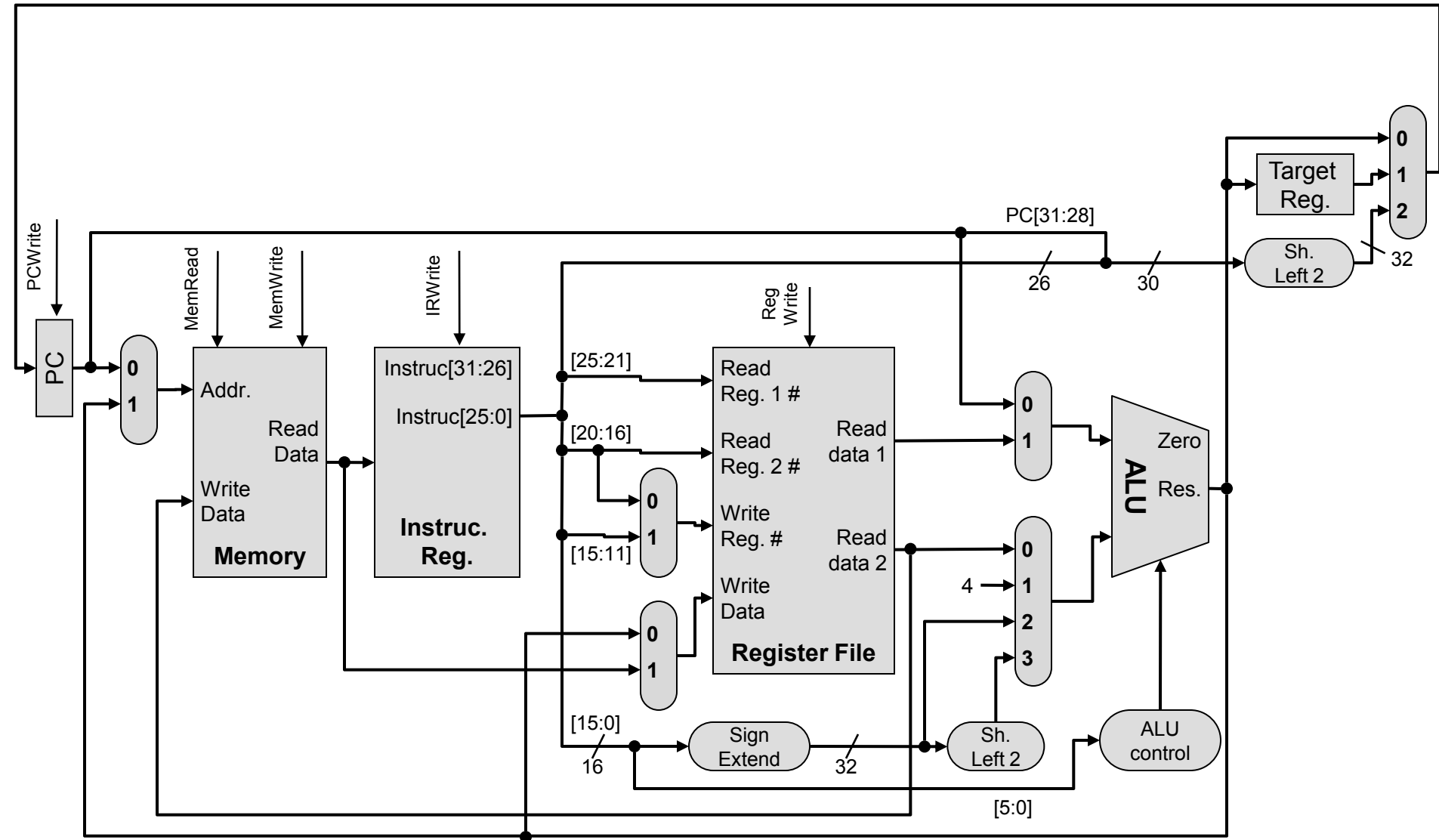
Multicycle CPU Datapath

# More on Temporary Registers

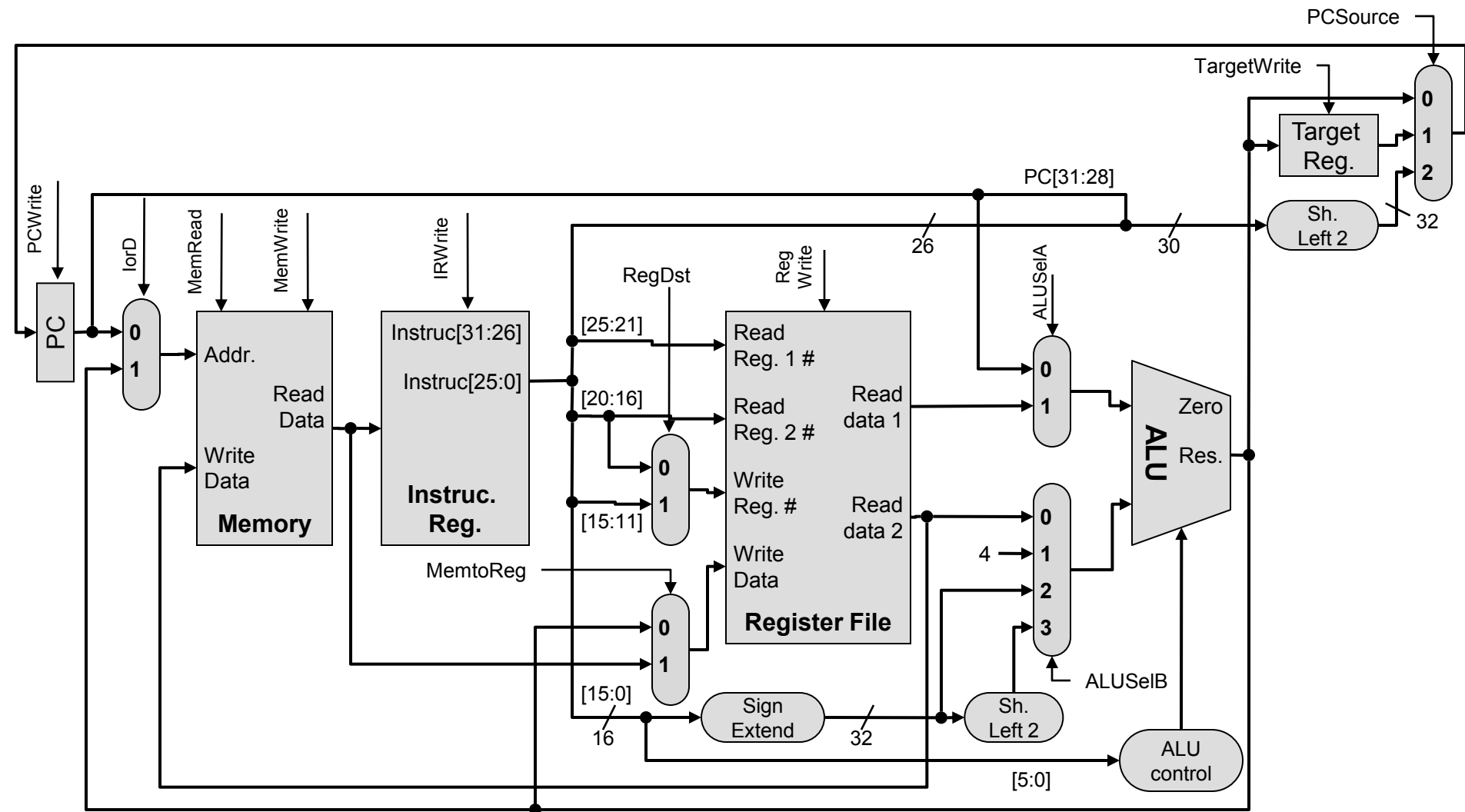
- Do temporary registers need a write enable (i.e. do we need IRwrite signal?)
- Unless it is acceptable for the register to be written on every clock cycle, then we do need a write enable
  - Based on our design, we only write the IR after the cycle we read the instruction and not on other cycles...thus we need an IRwrite



# Multi-Cycle CPU Datapath



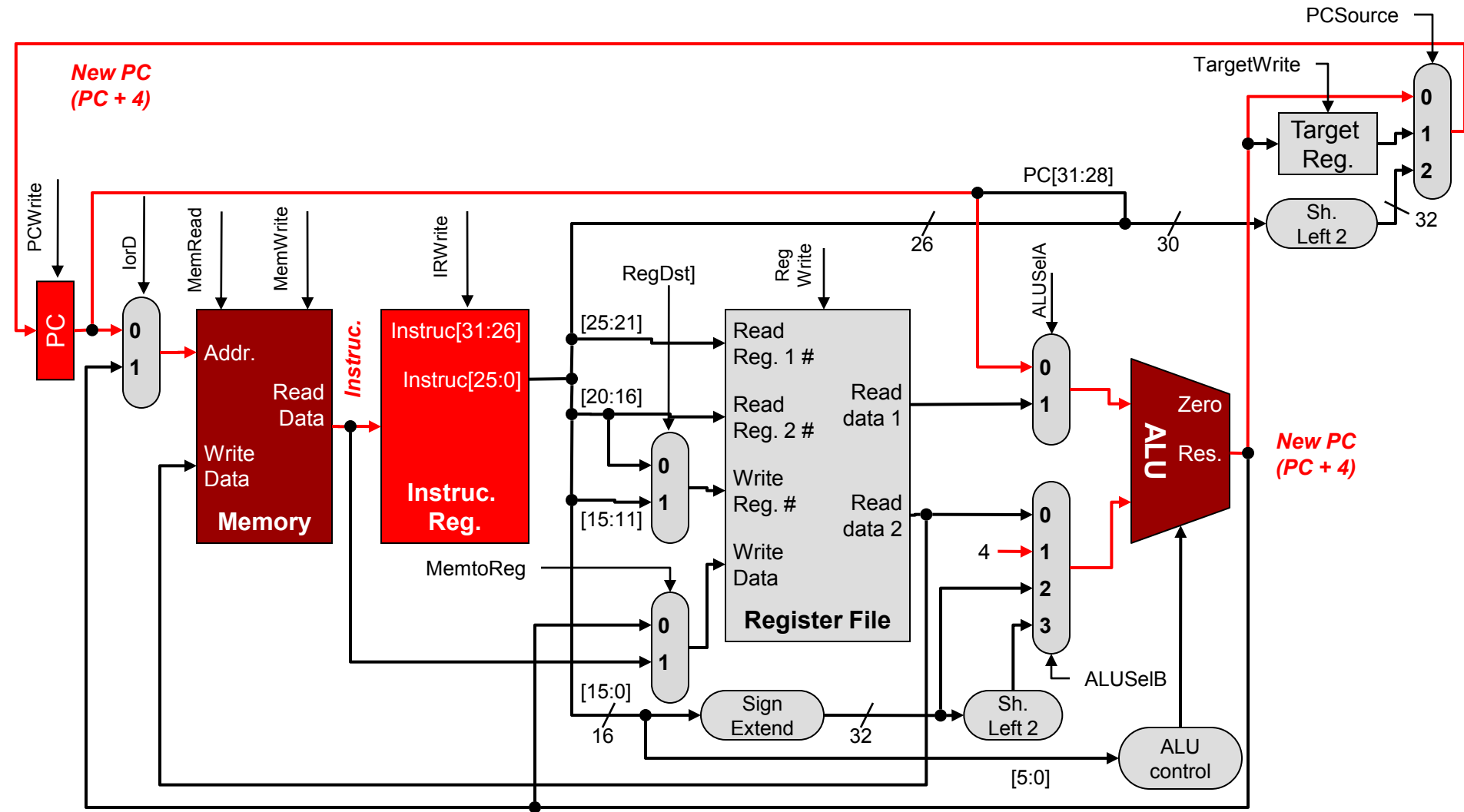
# Multi-Cycle CPU Datapath



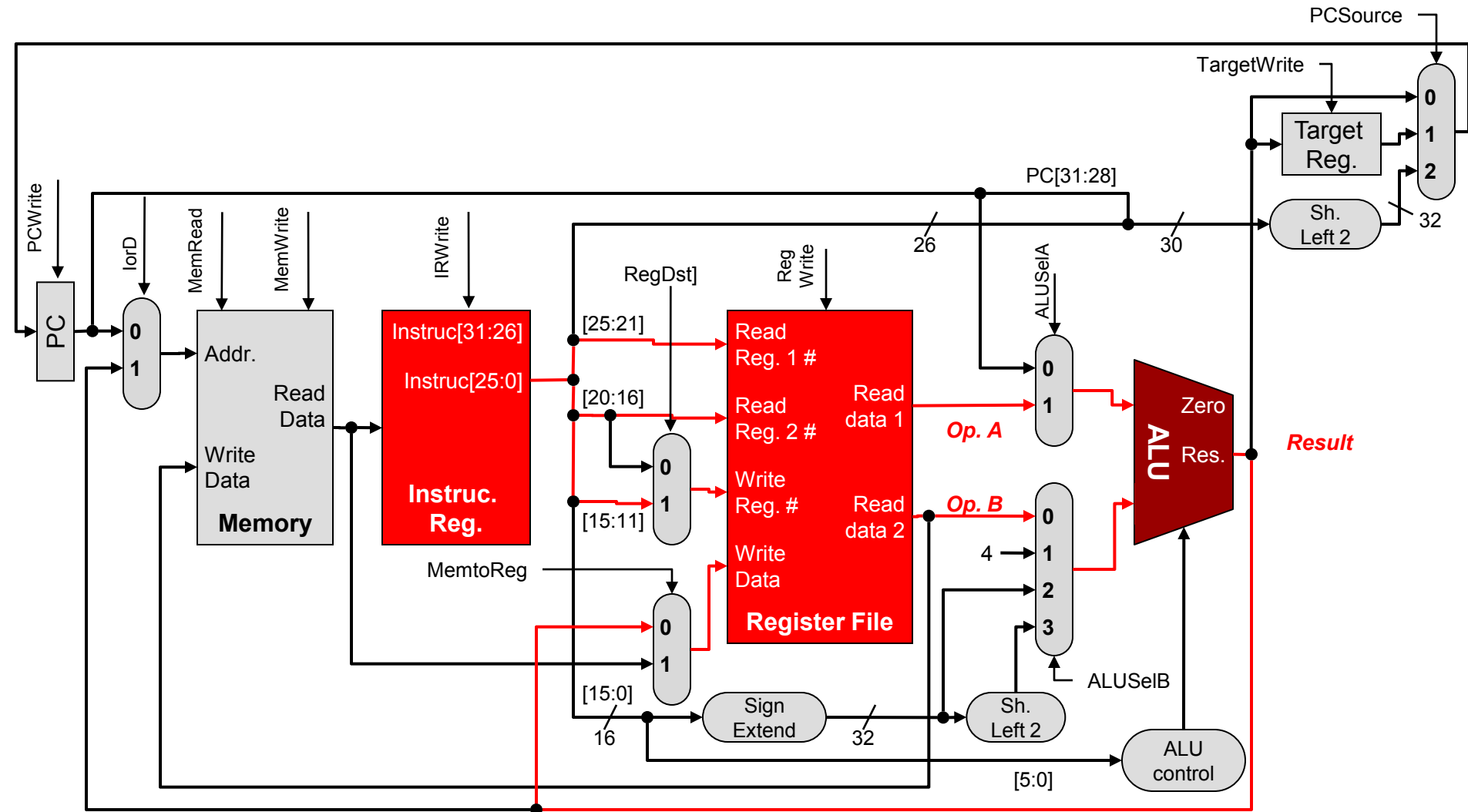
# Single vs. Multi-Cycle CPU

Single-Cycle CPU	Multi-Cycle CPU
Single LONG clock	Several SHORT clocks
No sharing of resources	Sharing resources possible
ALU & 2 separate adders	Single ALU does all three jobs
Separate instruction & data memory	Single unified memory
No need for any temp. register	Need for temp. registers like IR
PCWrite Unneeded	PCWrite Needed
Control unit not an FSM	Control Unit is an FSM

# Instruction Fetch + PC Increment

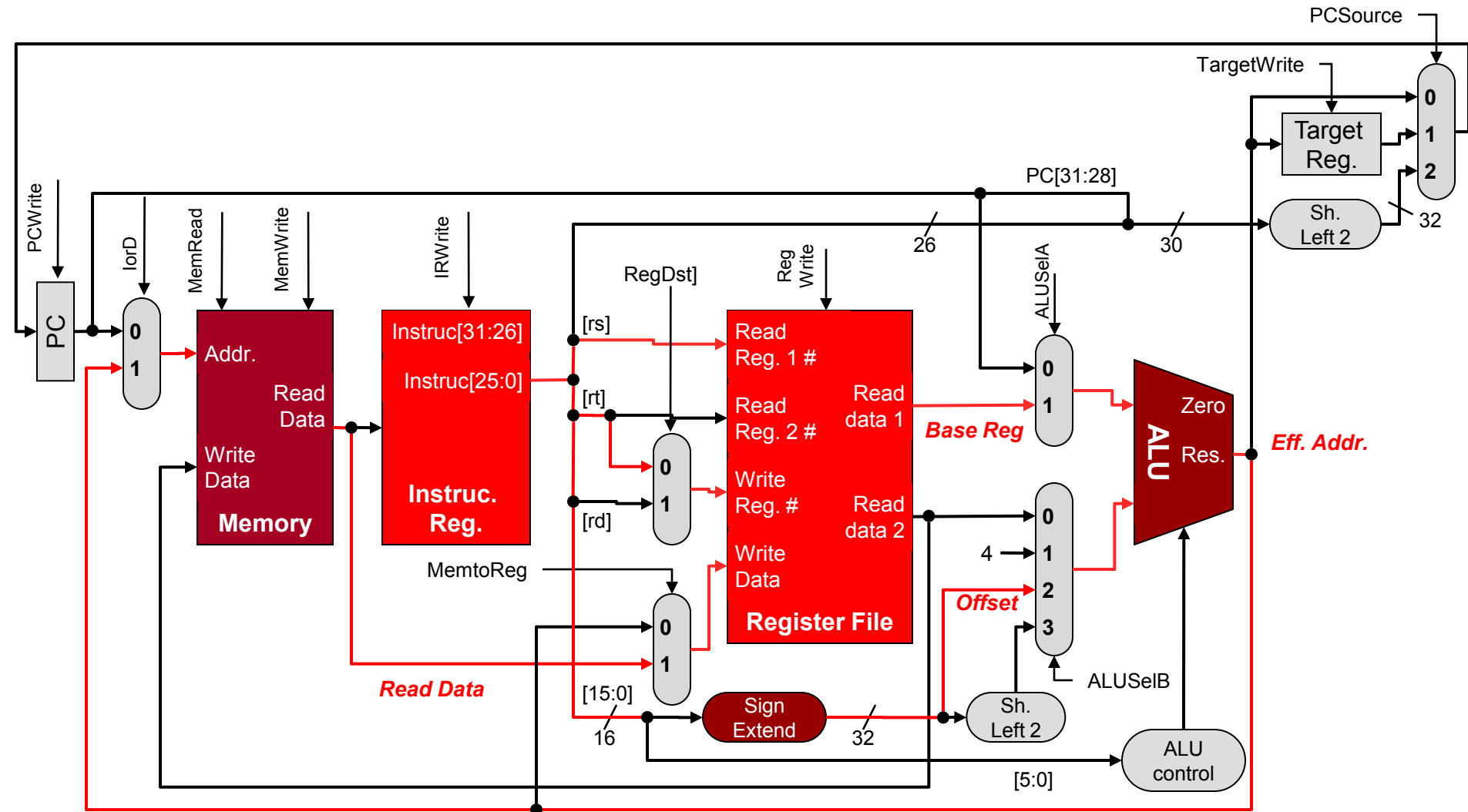


# R-Type Execution

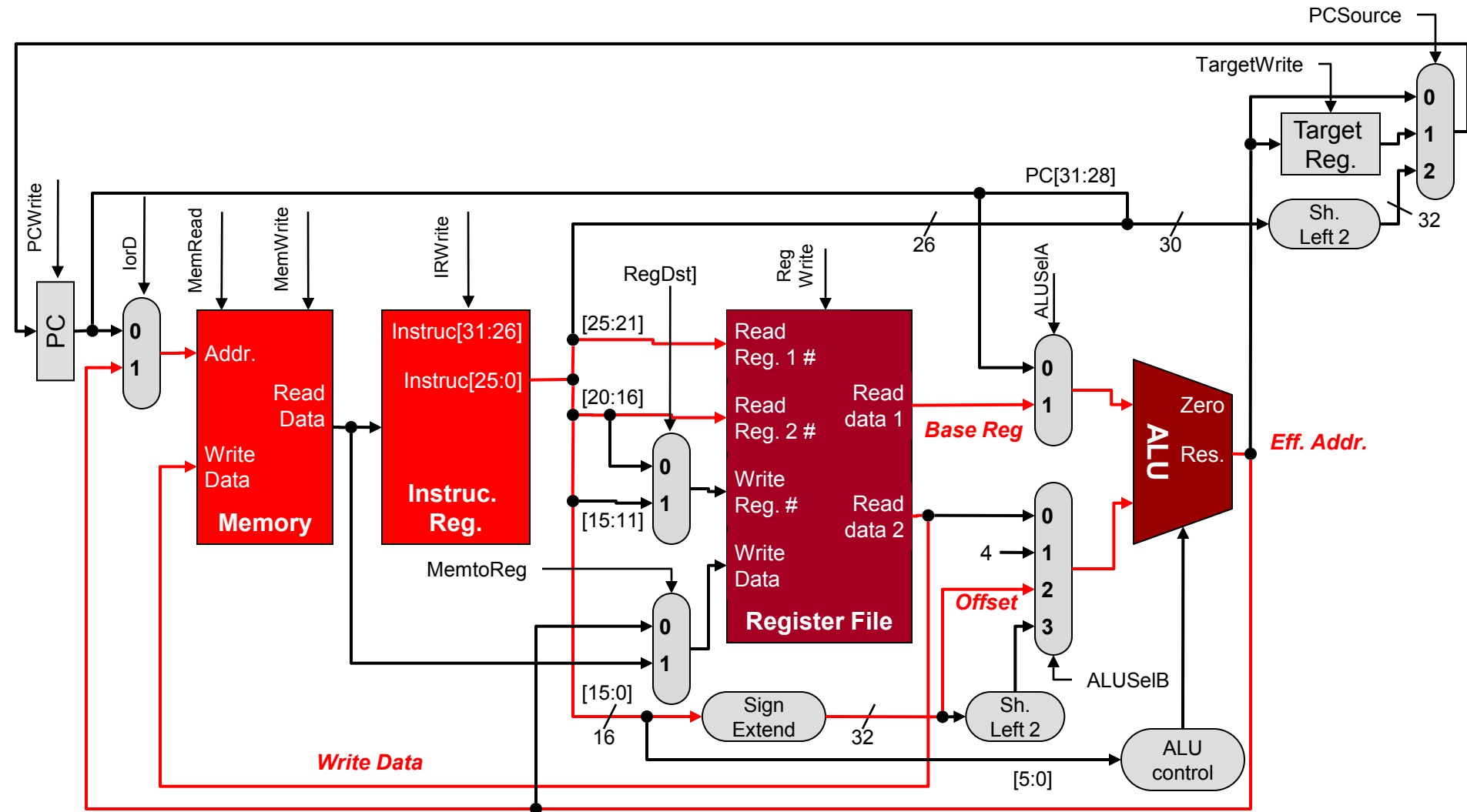




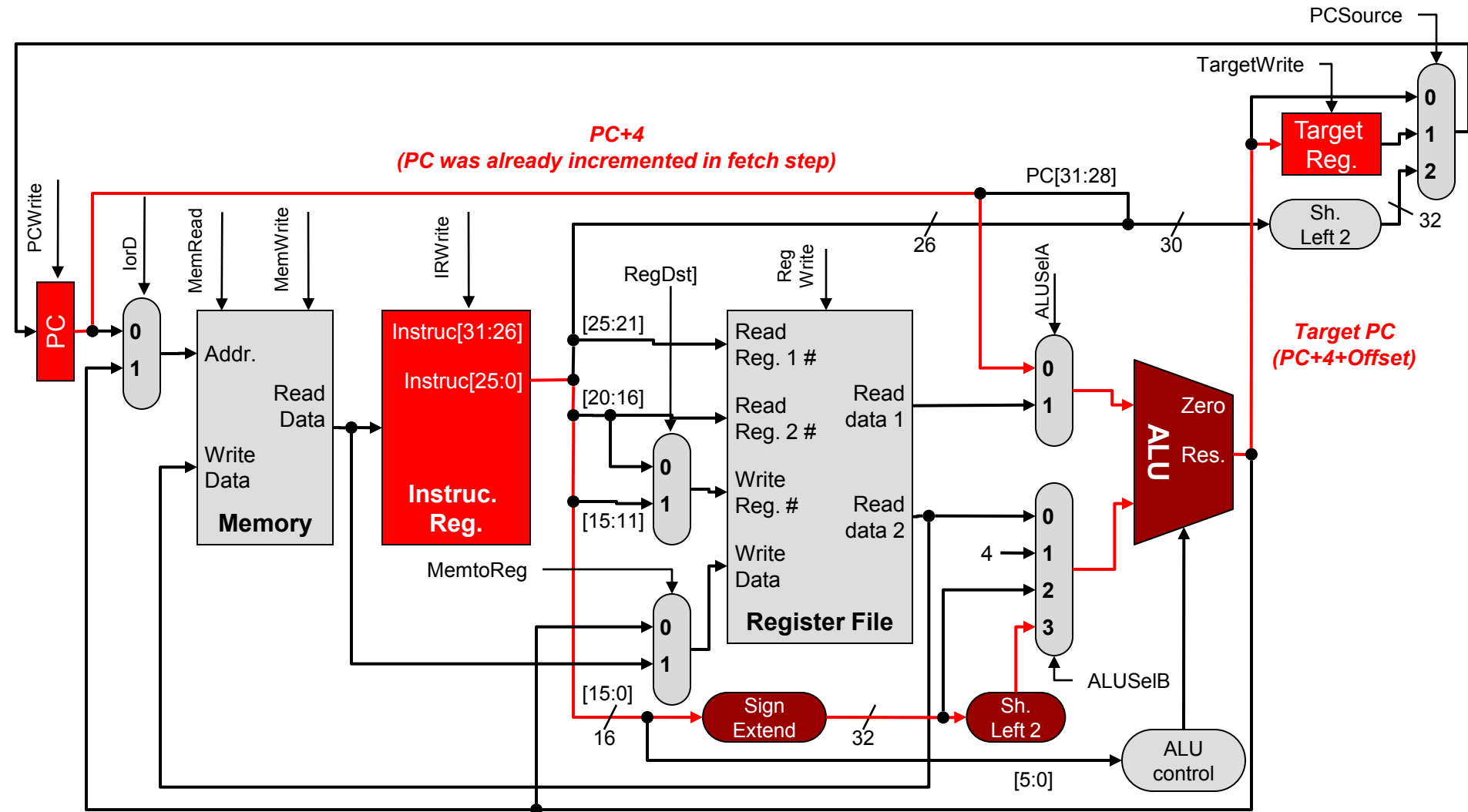
# LW Execution



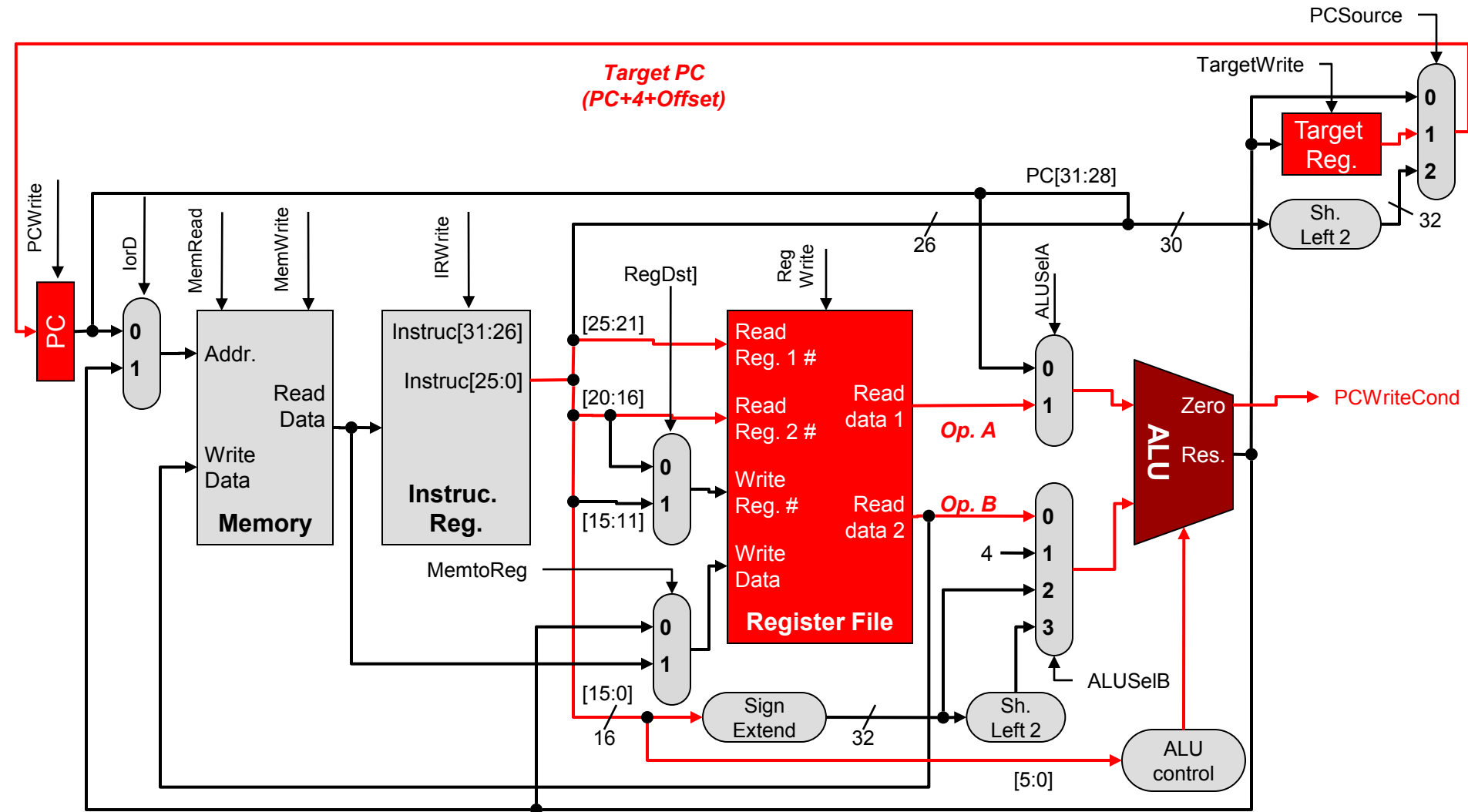
# SW Execution



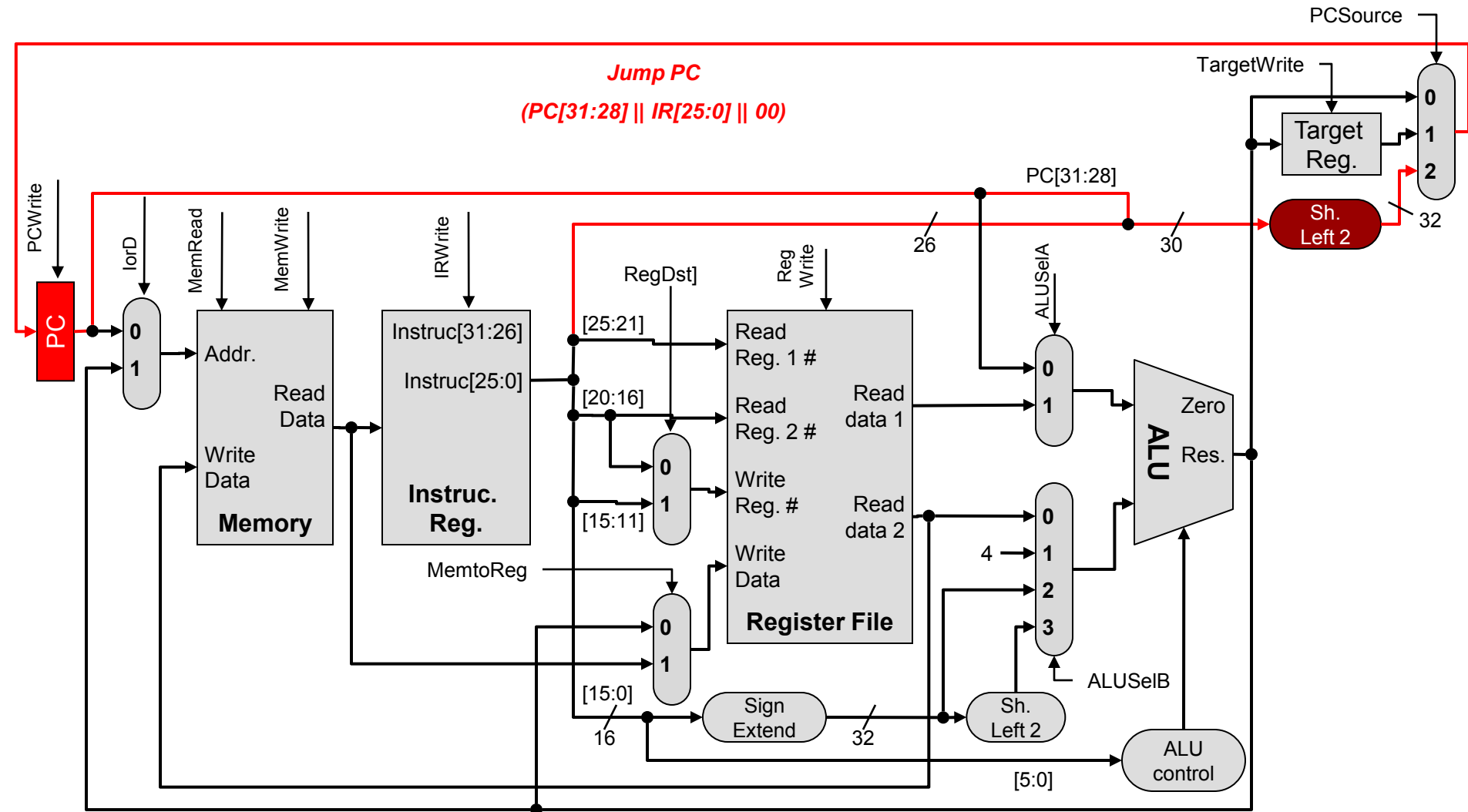
# BEQ Execution Step 1



# BEQ Execution Step 2



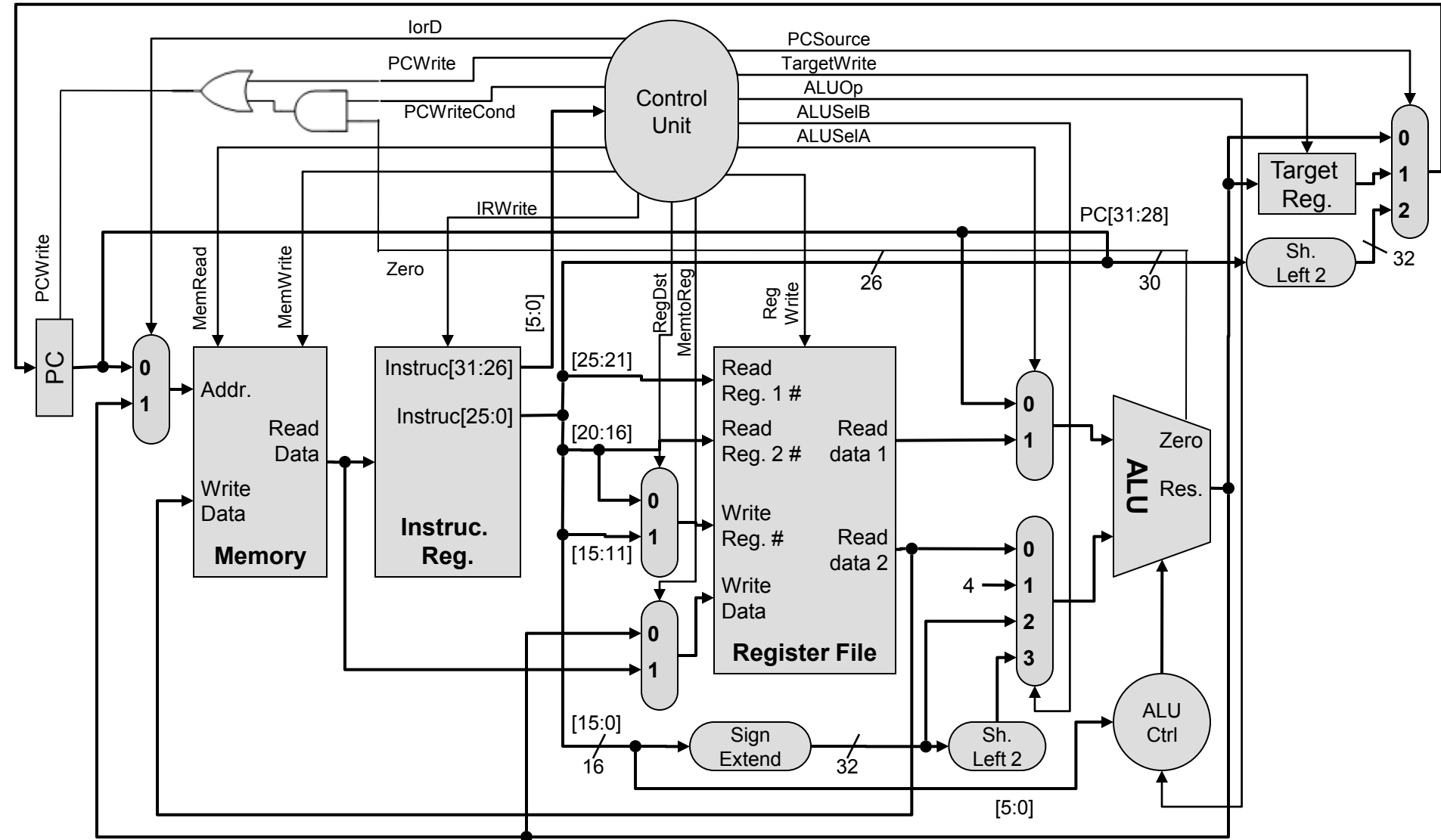
# Jump Execution



# Controlling the Datapath

- Now we need to implement the logic for the control signals
- This will require an FSM for our multi-cycle CPU (since we will have sub-operations or steps to execute each instruction)

# Multi-Cycle CPU



# Control Signal Explanation

Signal Name	Effect when Deasserted	Effect when Asserted
MemRead	None	Read data from memory
MemWrite	None	Write to data memory
ALUSelA	Select the PC value	Selects the rs register value
RegDst	Register to write is specified by rt field	Register to write is specified by rd field
RegWrite	None	Register file will write the specified register
MemtoReg	Reg. file write data comes from ALU	Reg. file write data comes from memory read data
lorD	PC is used as address to memory	ALU output is used as address to memory
IRWrite	None	Memory read data is written to IR



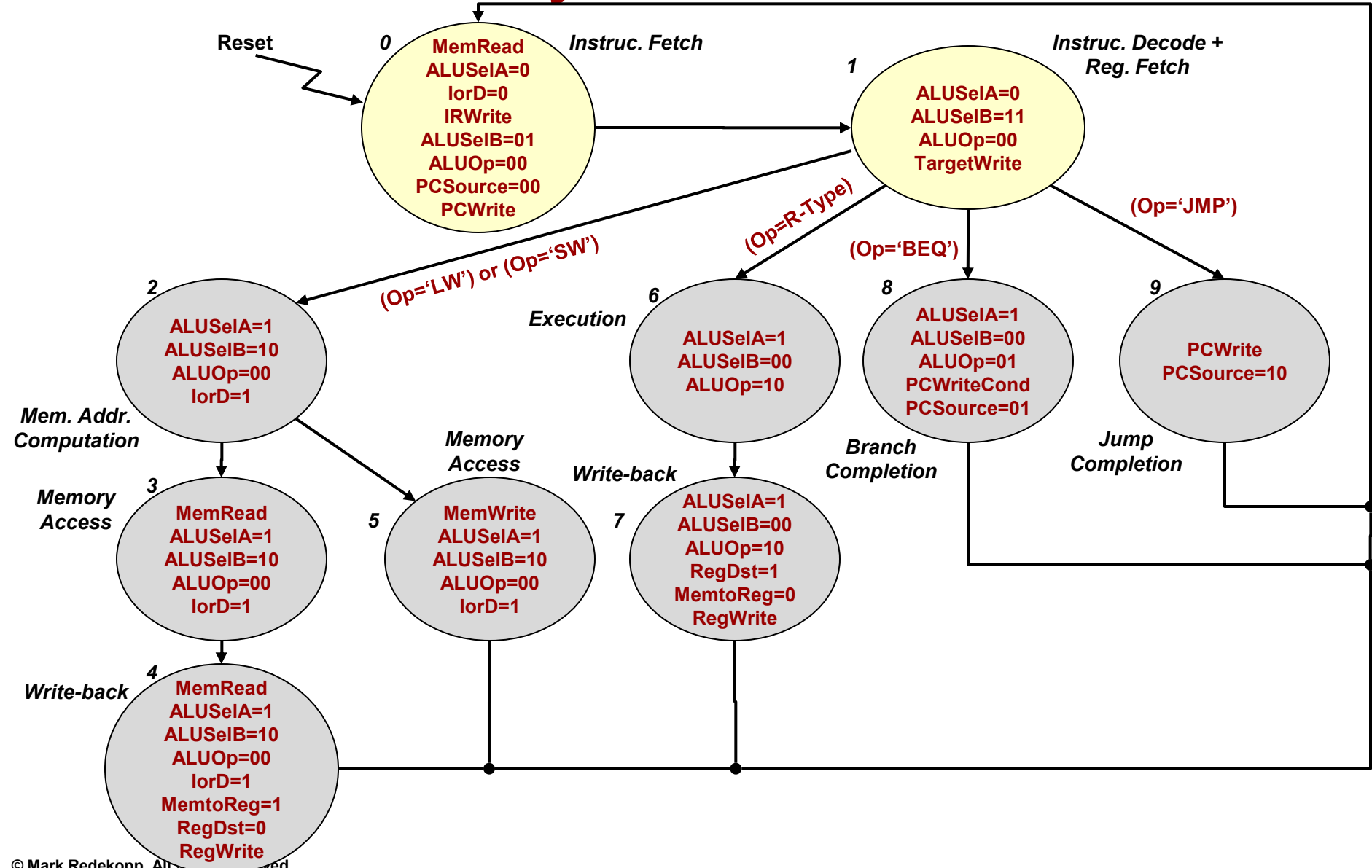
# Control Signal Explanation

Signal Name	Value	Effect
ALUSelB	00	Selects the rt register value
	01	Selects the constant 4
	10	Selects the sign extended lower 16-bits of IR
	11	Selects the sign extended and shifted lower 16-bits of IR
ALUOp	00	ALU performs an ADD operation
	01	ALU performs a SUB operation
	10	The function code field of instruction will determine ALU op.
PCSource	00	Selects the ALU output to pass back to the PC input
	01	Selects the target register value to pass back to the PC input
	10	Selects the jump address value to pass back to the PC input

# Generating a State Diagram

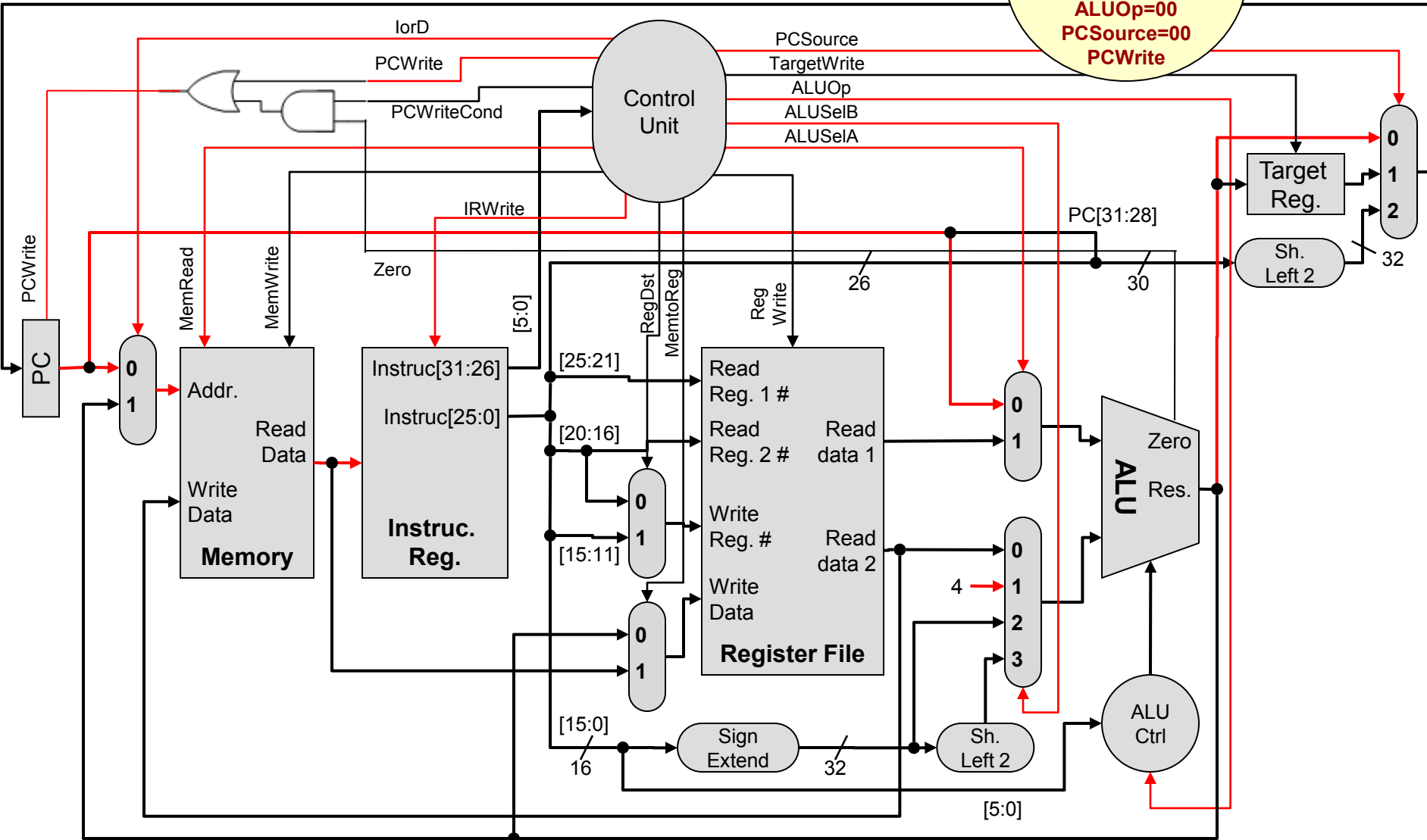
- Start with states to fetch instruction, increment PC, & decode it
  - These are common to any instruction because at this point we don't know what instruction it is
- Once decoded use a separate sequence of states for each instruction
  - One state for each sub-operation of each instruction
- Goal is to find state breakdown that leads to short, equal timed steps
  - **Short**: Shorter the time delay of the step => Faster clock rate
  - **Equal-timed**: Clock cycle is set by the slowest state; if the delays in states are poorly balanced, some states will have to pay a longer delay even though they don't need it

# Multi-cycle CPU FSM



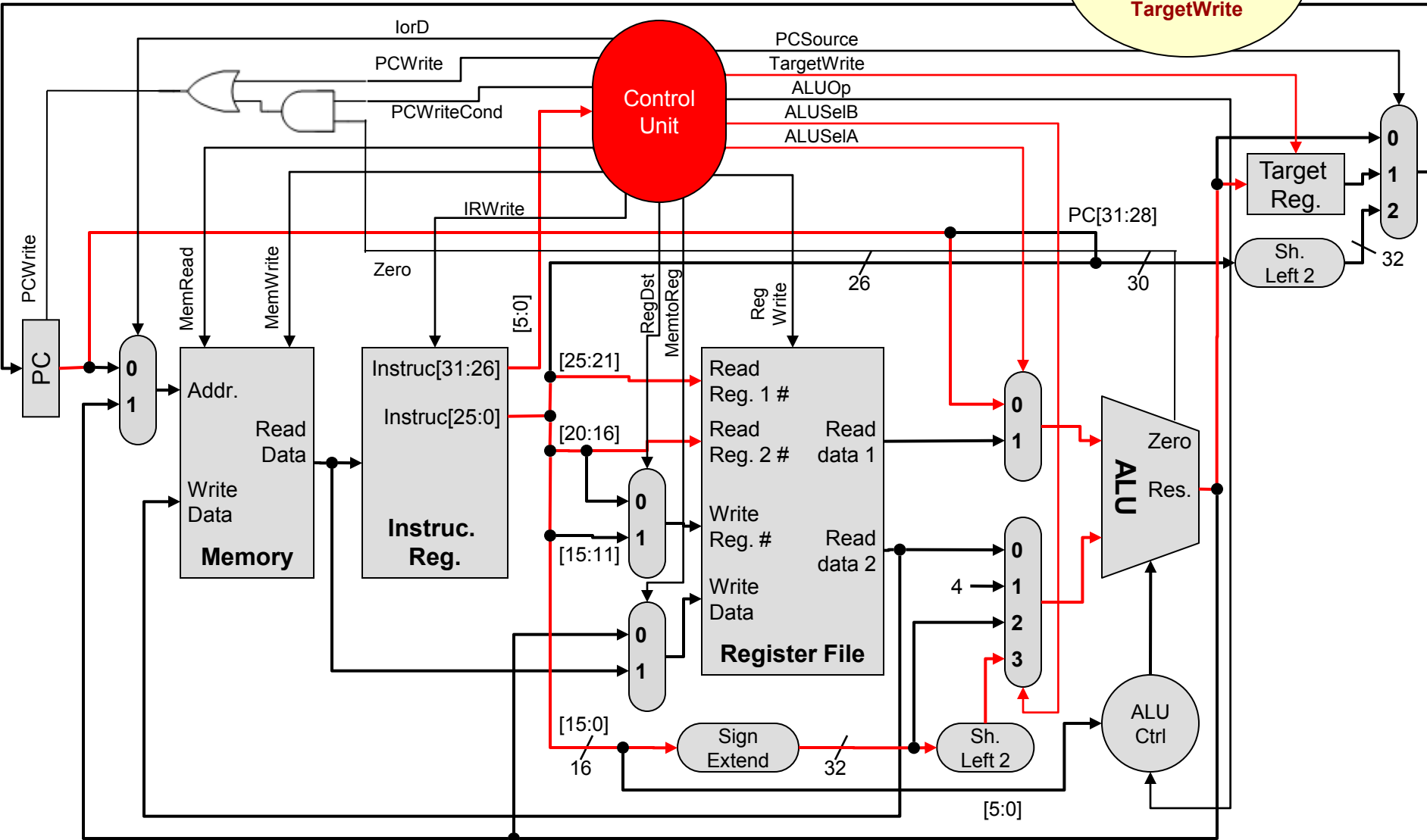
# State 0 = Fetch

MemRead  
ALUSelA=0  
lorD=0  
IRWrite  
ALUSelB=01  
ALUOp=00  
PCSource=00  
PCWrite



# State 1 = Decode / Reg. Fetch

ALUSelA=0  
ALUSelB=11  
ALUOp=00  
TargetWrite

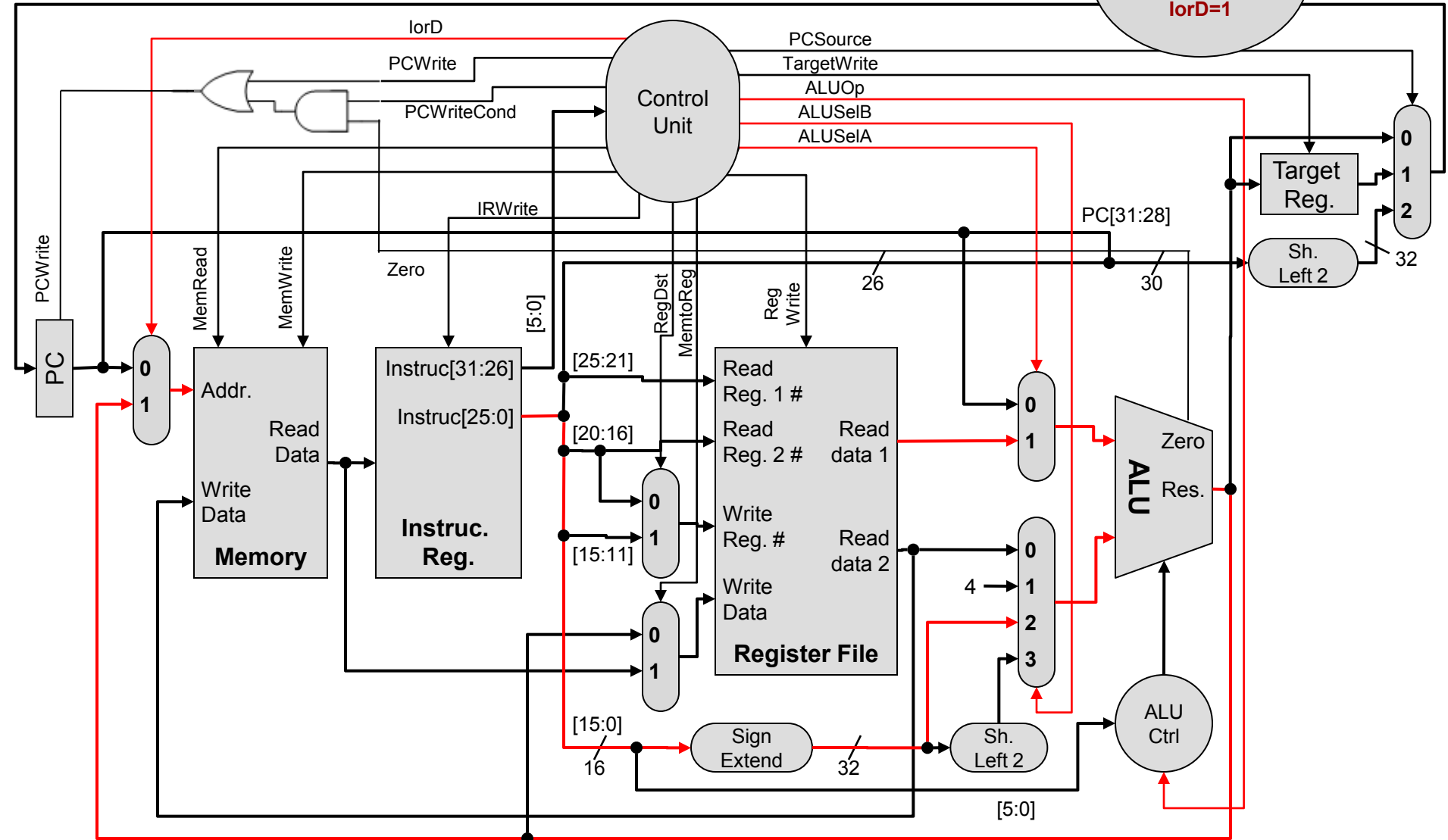


# Questions

- After state 0 (fetch) we store the instruction in the IR, after state 1 when we fetch register operands do we need to store operands in temp reg's (e.g. AReg, BReg)?
- Do we need RegReadA, RegReadB control signals?

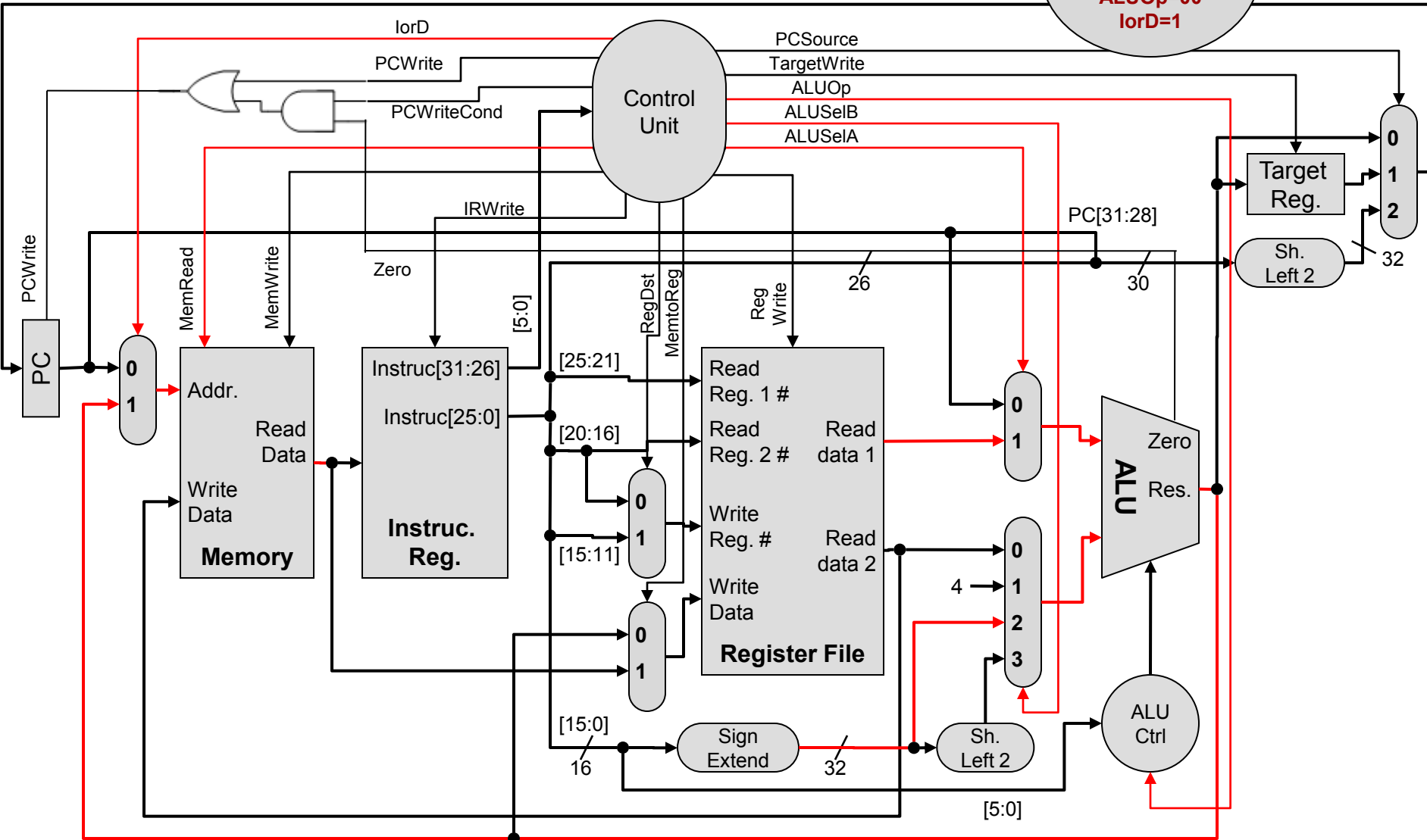
# LW/SW State 2

ALUSelA=1  
ALUSelB=10  
ALUOp=00  
lorD=1



# LW State 3

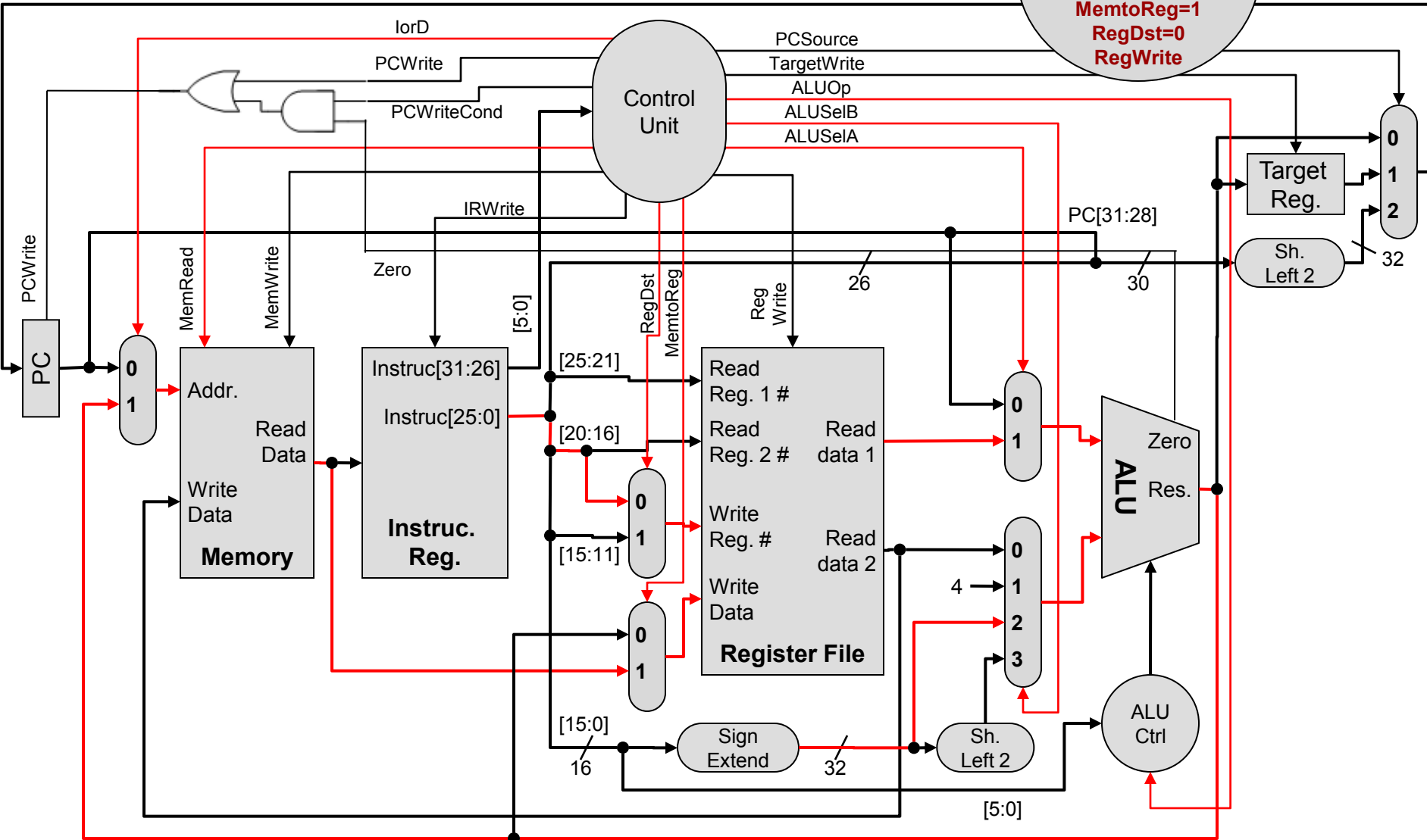
MemRead  
ALUSelA=1  
ALUSelB=10  
ALUOp=00  
IorD=1





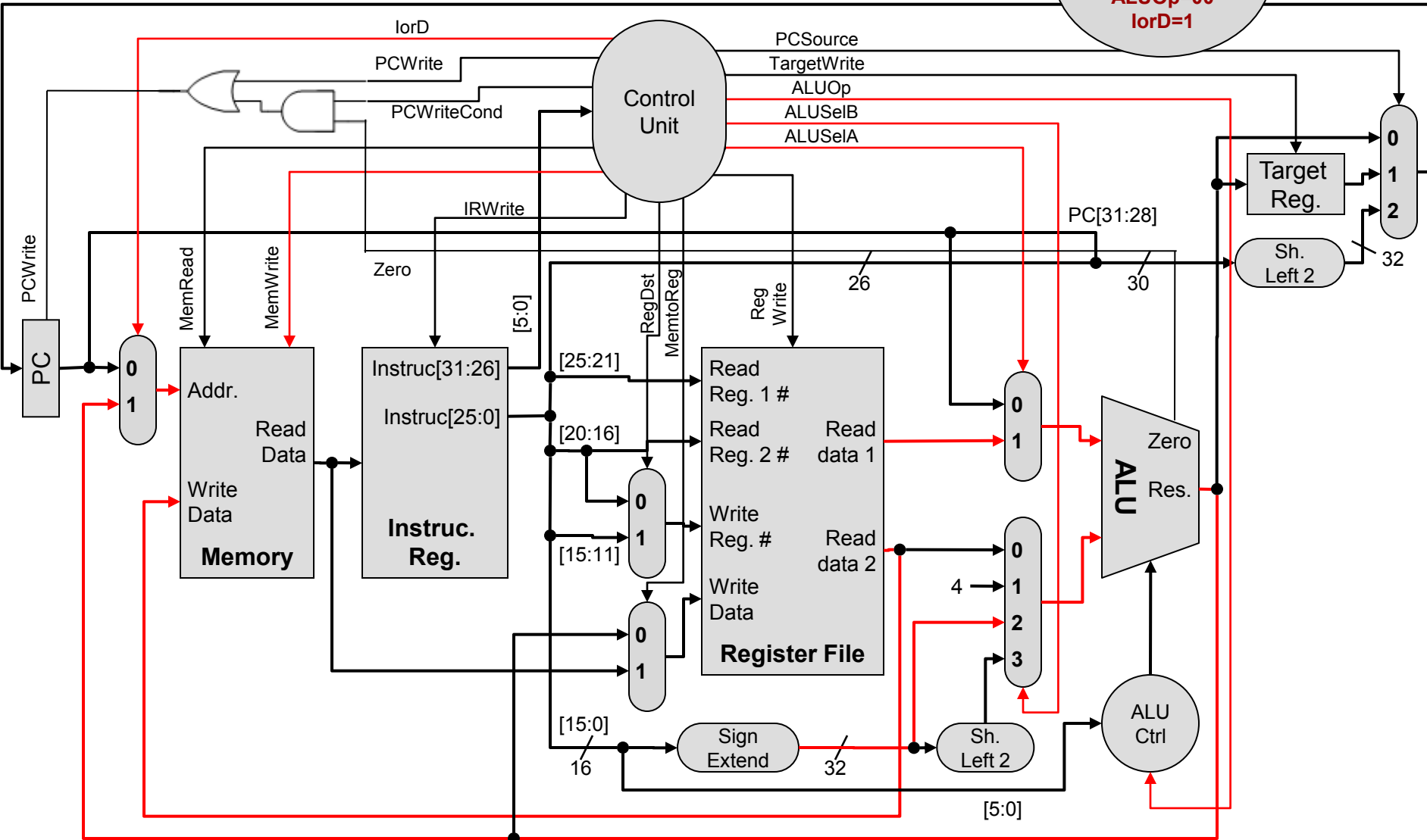
# LW State 4

MemRead  
ALUSelA=1  
ALUSelB=10  
ALUOp=00  
lorD=1  
MemtoReg=1  
RegDst=0  
RegWrite



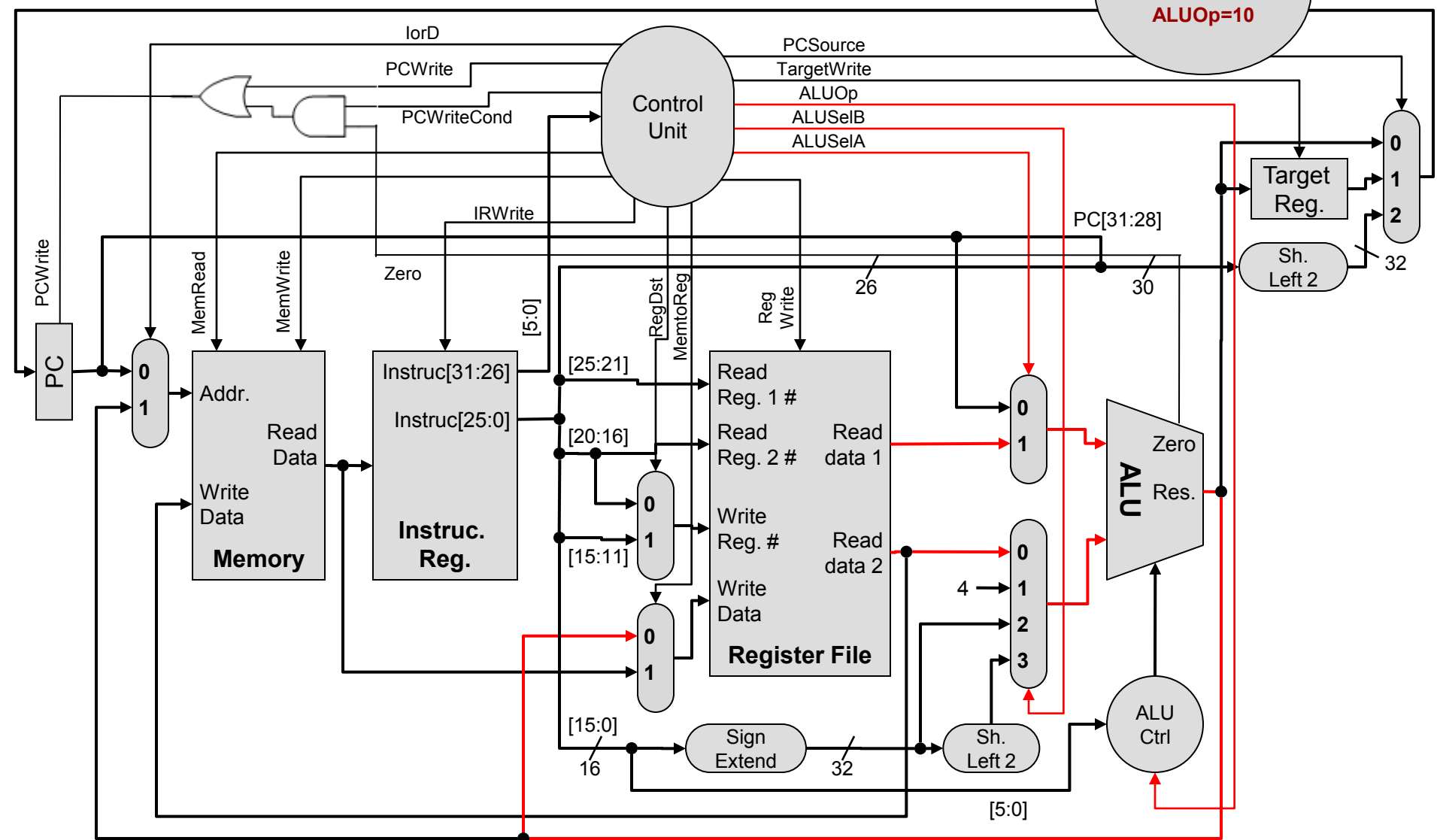
# SW State 5

MemWrite  
ALUSelA=1  
ALUSelB=10  
ALUOp=00  
lorD=1



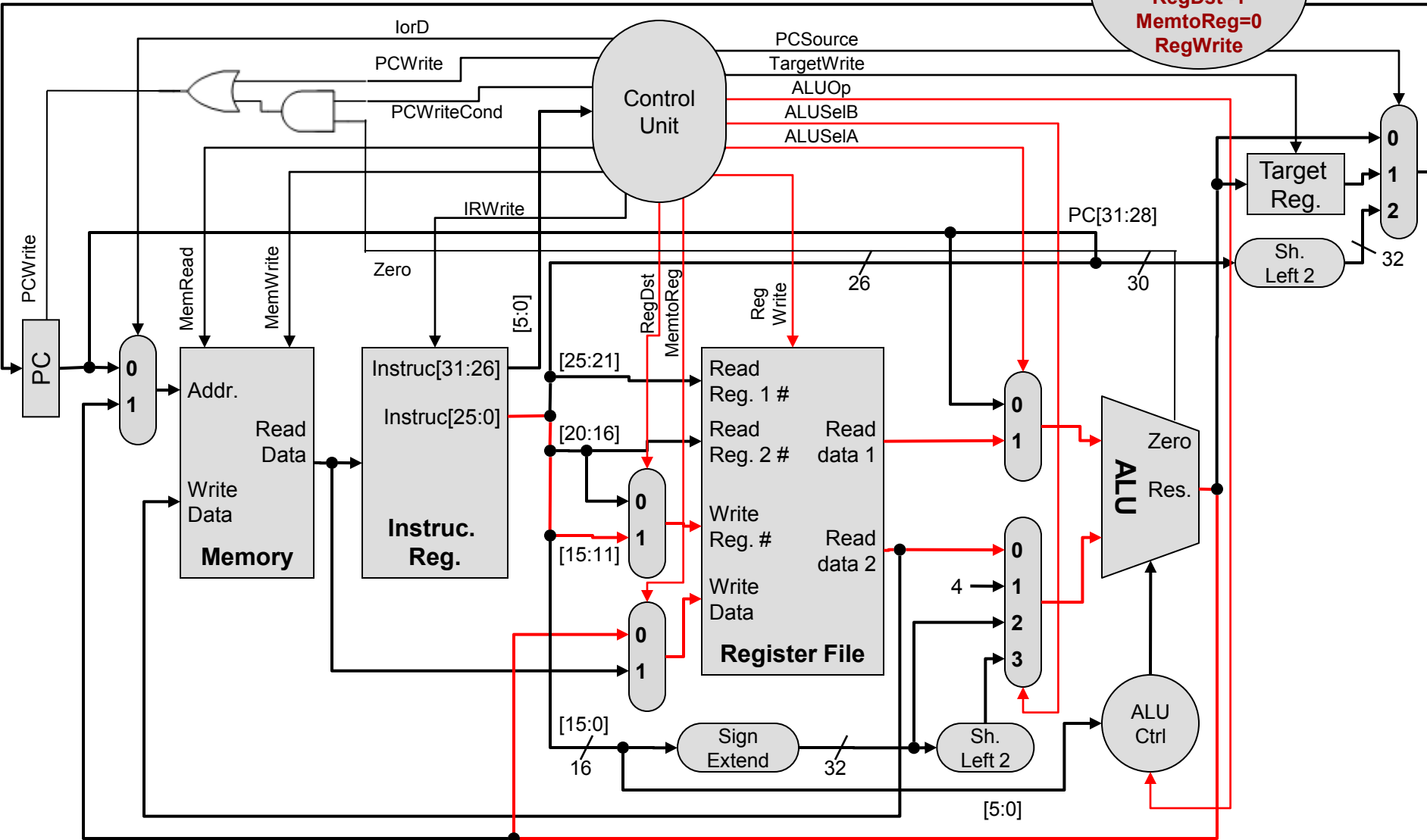
# R-Type State 6

ALUSelA=1  
ALUSelB=00  
ALUOp=10



# R-Type State 7

ALUSelA=1  
ALUSelB=00  
ALUOp=10  
RegDst=1  
MemtoReg=0  
RegWrite

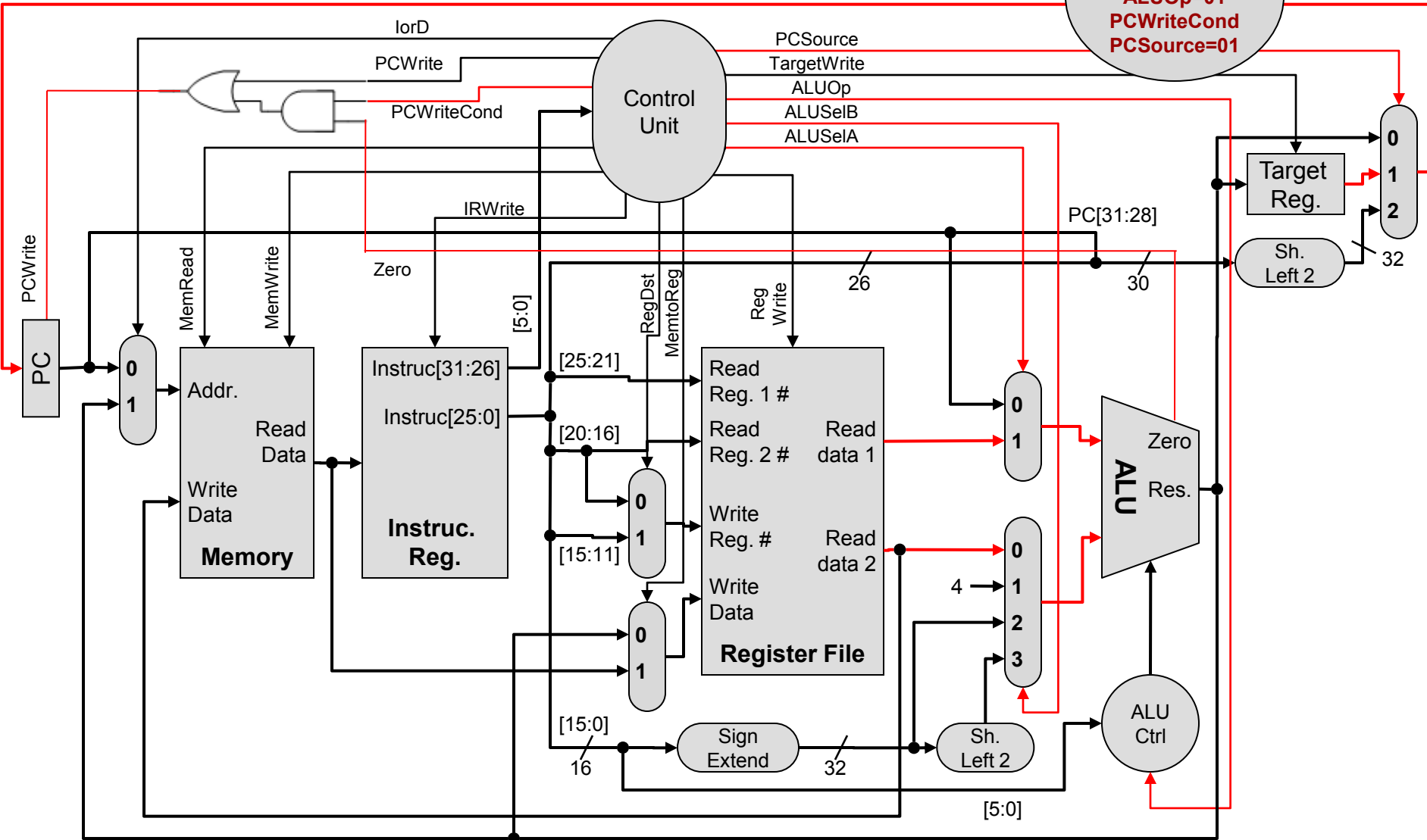


# Questions

- For R-Type or LW...
  - Can we turn on RegWrite one state earlier?
  
  
  
  
  
  
  
  
  
  
  - Can we set the RegDst signal earlier?

# BEQ State 8

ALUSelA=1  
ALUSelB=00  
ALUOp=01  
PCWriteCond  
PCSource=01



# Jump State 9

