MultiCycle CPU

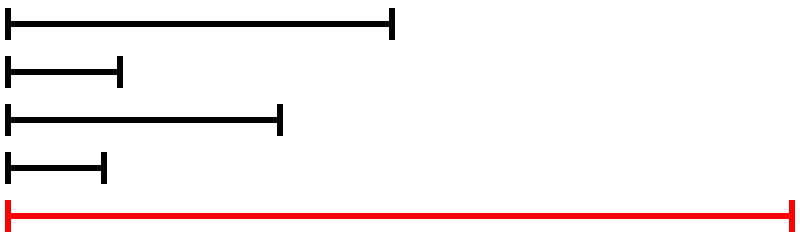
中科院计算所
INSTITUTE OF COMPUTING TECHNOLOGY

University of Chinese Academy of Sciences (UCAS)

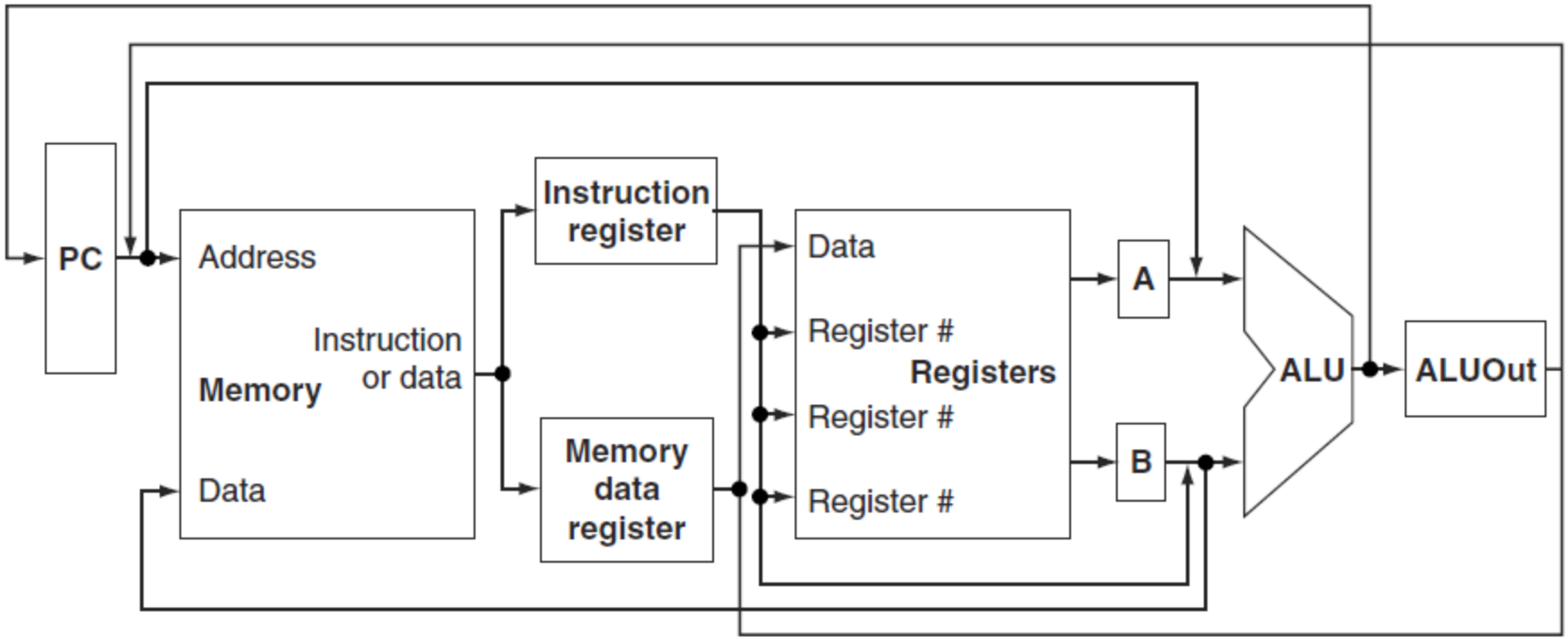
Why a Multiple Cycle CPU?

- The problem => single-cycle cpu has a cycle time long enough to complete the longest instruction in the machine
- The solution => break up execution into smaller tasks, each task taking a cycle, different instructions requiring different numbers of cycles or tasks
- Other advantages => reuse of functional units (e.g., alu, memory)

- ET = IC * CPI * CT



High Level View



- A single memory unit is used for both instructions and data.
- There is a single ALU, rather than an ALU and two adders.
- One or more registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle

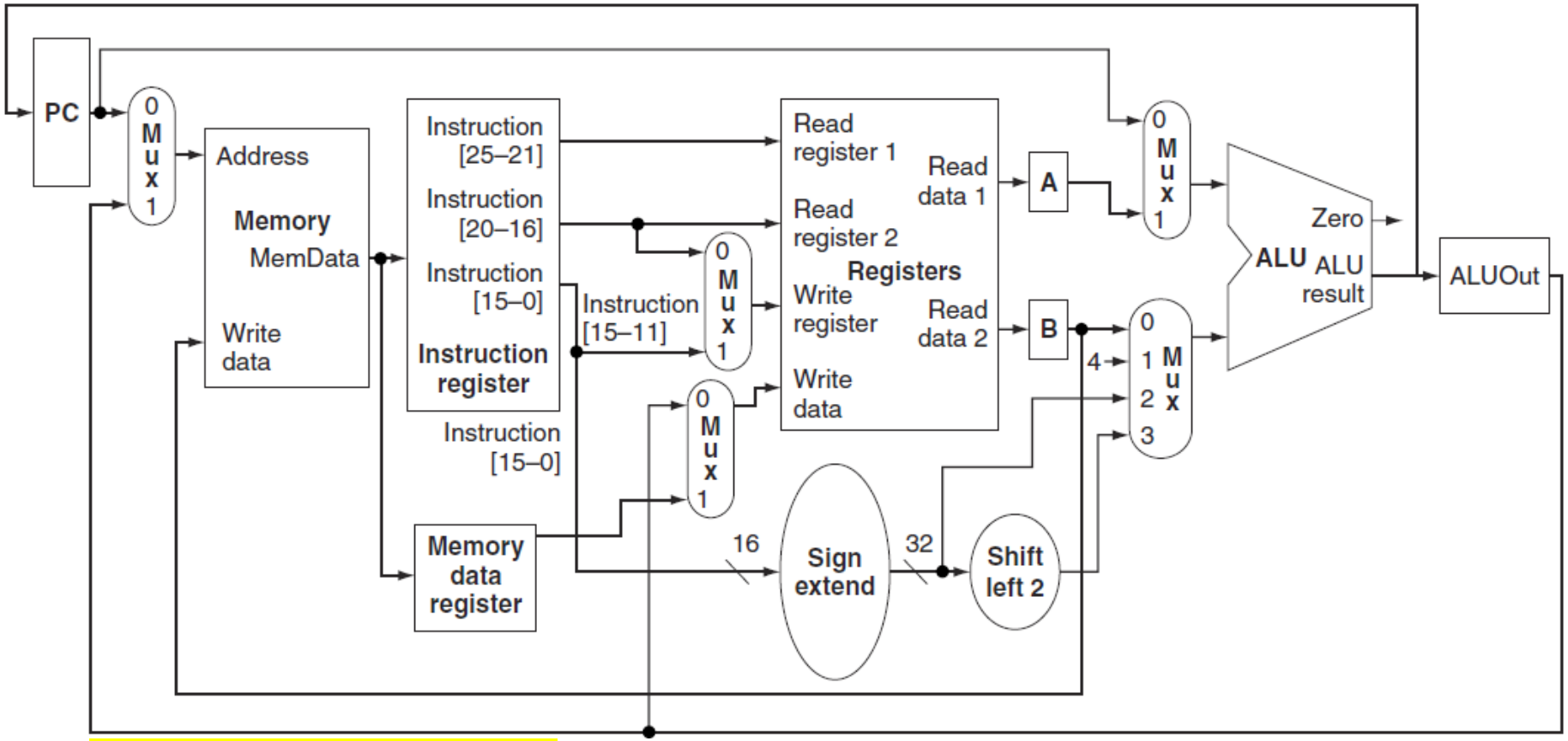
Breaking Execution into Clock Cycles

- We will have five execution steps (not all instructions use all five)
  - Fetch
  - Decode & register fetch
  - Execute
  - Memory access
  - Write-back
- We will use Register-Transfer-Language (RTL) to describe these steps

Breaking Execution into Clock Cycles

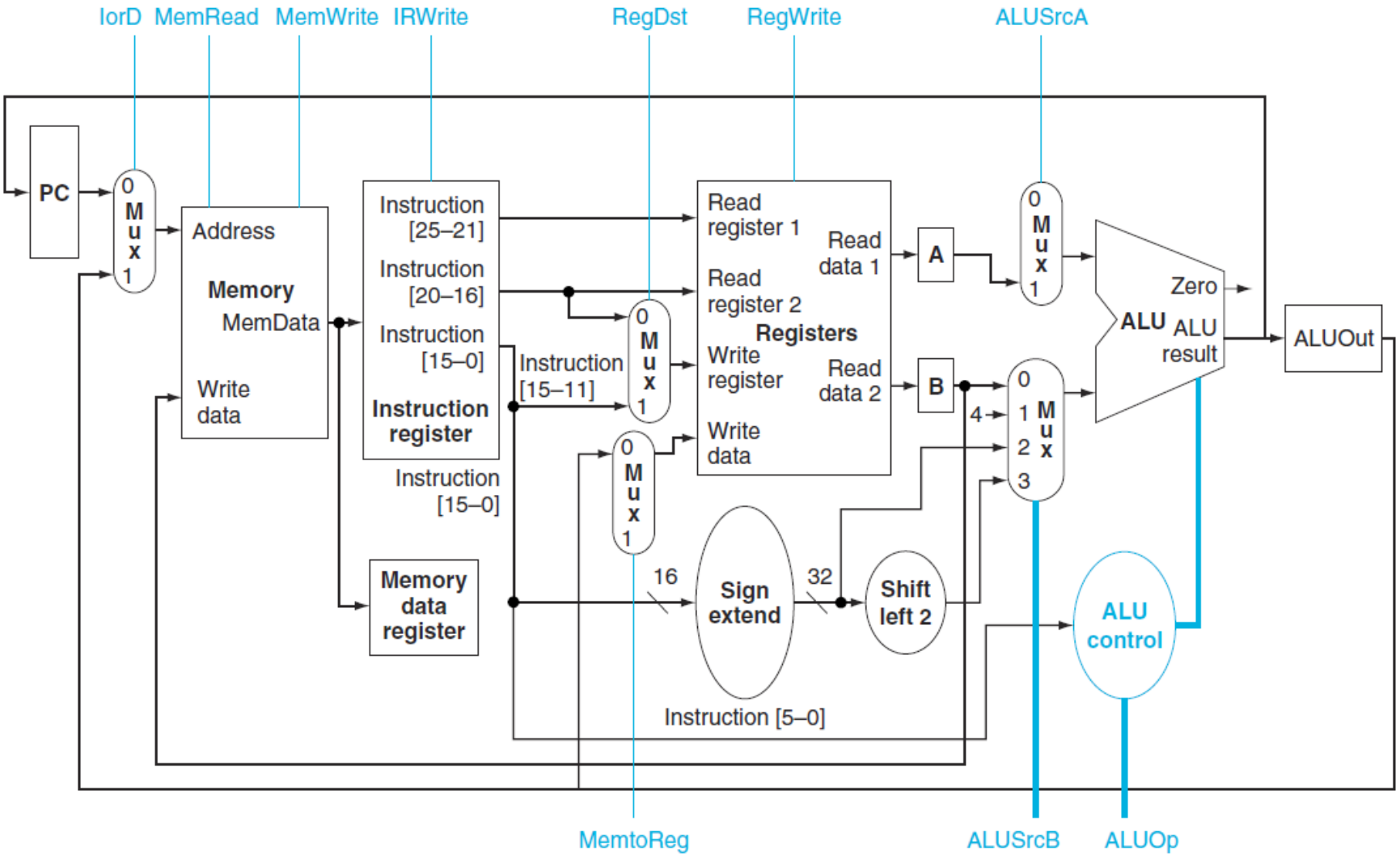
- Introduces extra registers when:
  - Signal is computed in one clock cycle and used in another, AND
  - The inputs to the functional block that outputs this signal can change before the signal is written into a state element.
- Significantly complicates control. Why?
- The goal is to balance the amount of work done each cycle.

Multi-Cycle Datapath

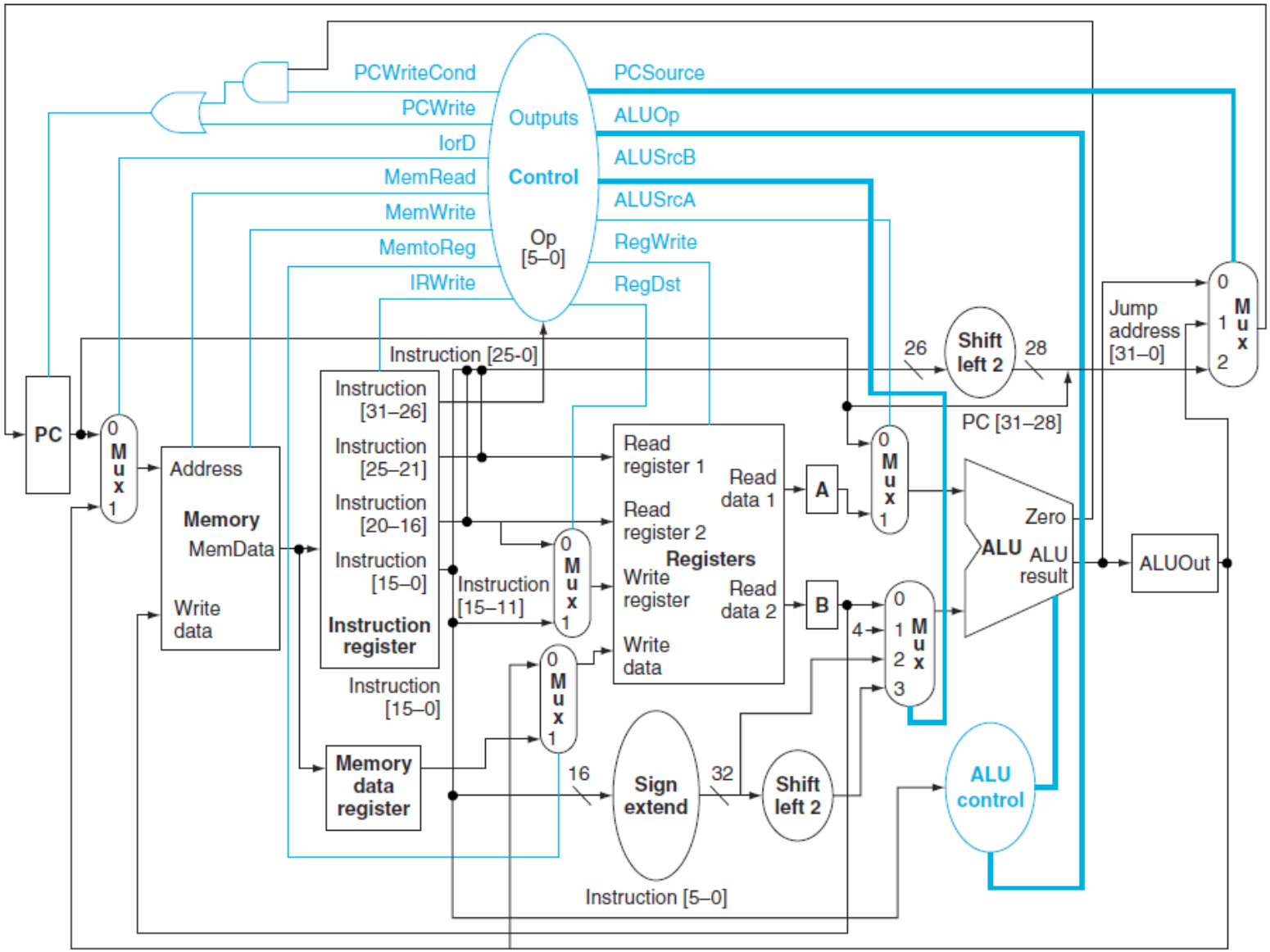


- Intermediate latches
- One ALU
- One memory

Multicycle Datapath with Control Signals



Control Unit



Actions of Control Signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

Breaking the Instruction Execution into Clock Cycles

1. Fetch

- IR = Mem[PC]
- PC = PC + 4

- *May not be final value of PC*

2. Instruction Decode and Register Fetch

- A = Reg[IR[25-21]]
- B = Reg[IR[20-16]]
- ALUOut = PC + (sign-extend (IR[15-0]) << 2)

- Compute target before we know if it will be used (may not be branch, branch may not be taken)
- ALUOut is a new state element (temp register)
- Everything up to this point must be Instruction-independent, because we still haven't decoded the instruction.
- Everything instruction (opcode)-dependent from here on.

3. Execution, Memory Address Computation, or Branch Completion

- Memory reference (load or store)
  - ALUOut = A + sign-extend(IR[15-0])
- R-type
  - ALUout = A op B
- Branch
  - if (A == B) PC = ALUOut

- *At this point, Branch is complete, and we start over; others require more cycles.*

4. Memory access or R-type completion

- Memory reference (load or store)
  - Load
    - MDR = Mem[ALUout]
  - Store
    - Mem[ALUout] = B
- R-type
  - Reg[IR[15-11]] = ALUout

- *R-type is complete, store is complete.*

5. Memory Write-Back (Memory Read Completion)

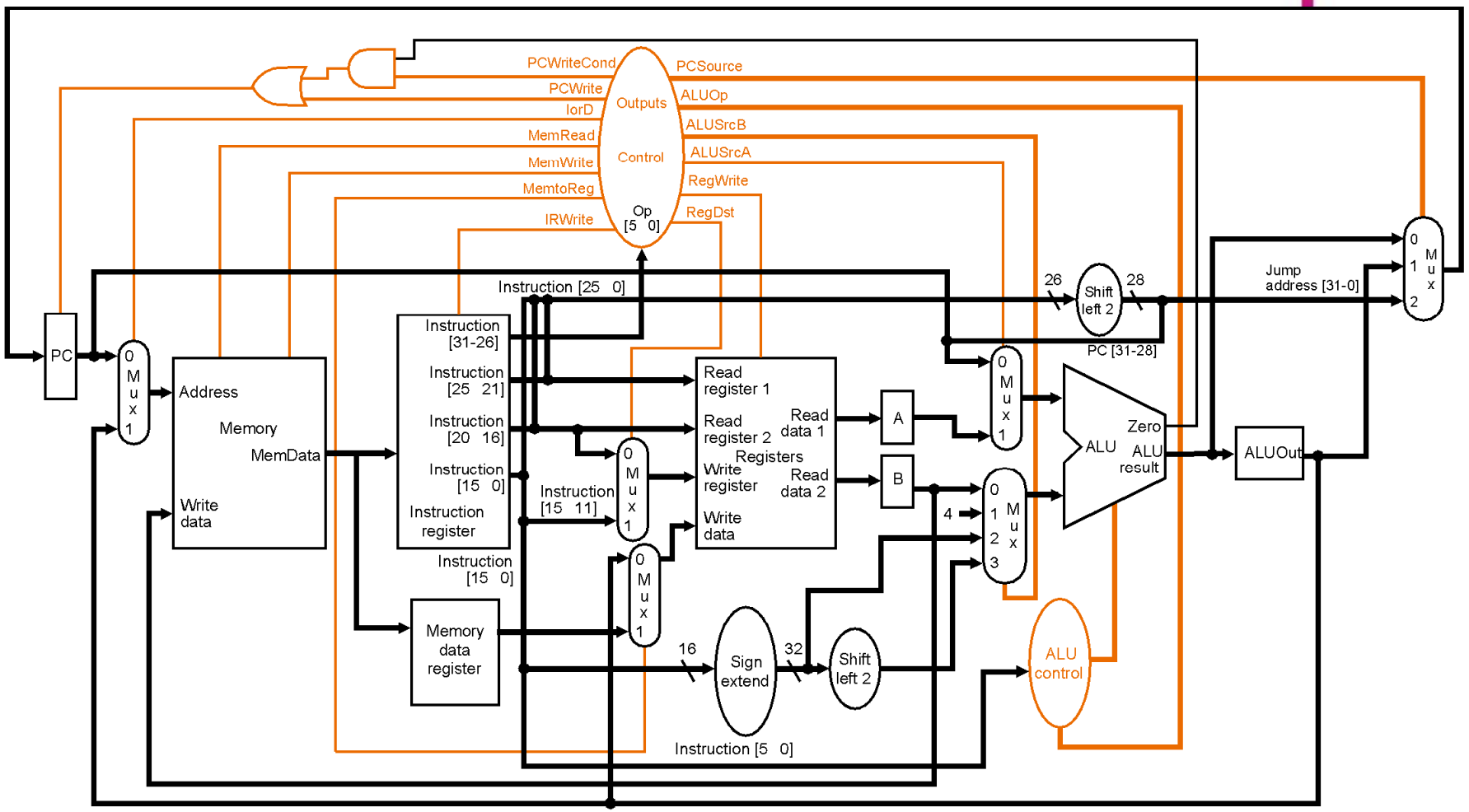
- Reg[IR[20-16]] = MDR

- *Load is complete*

Summary of Execution Steps

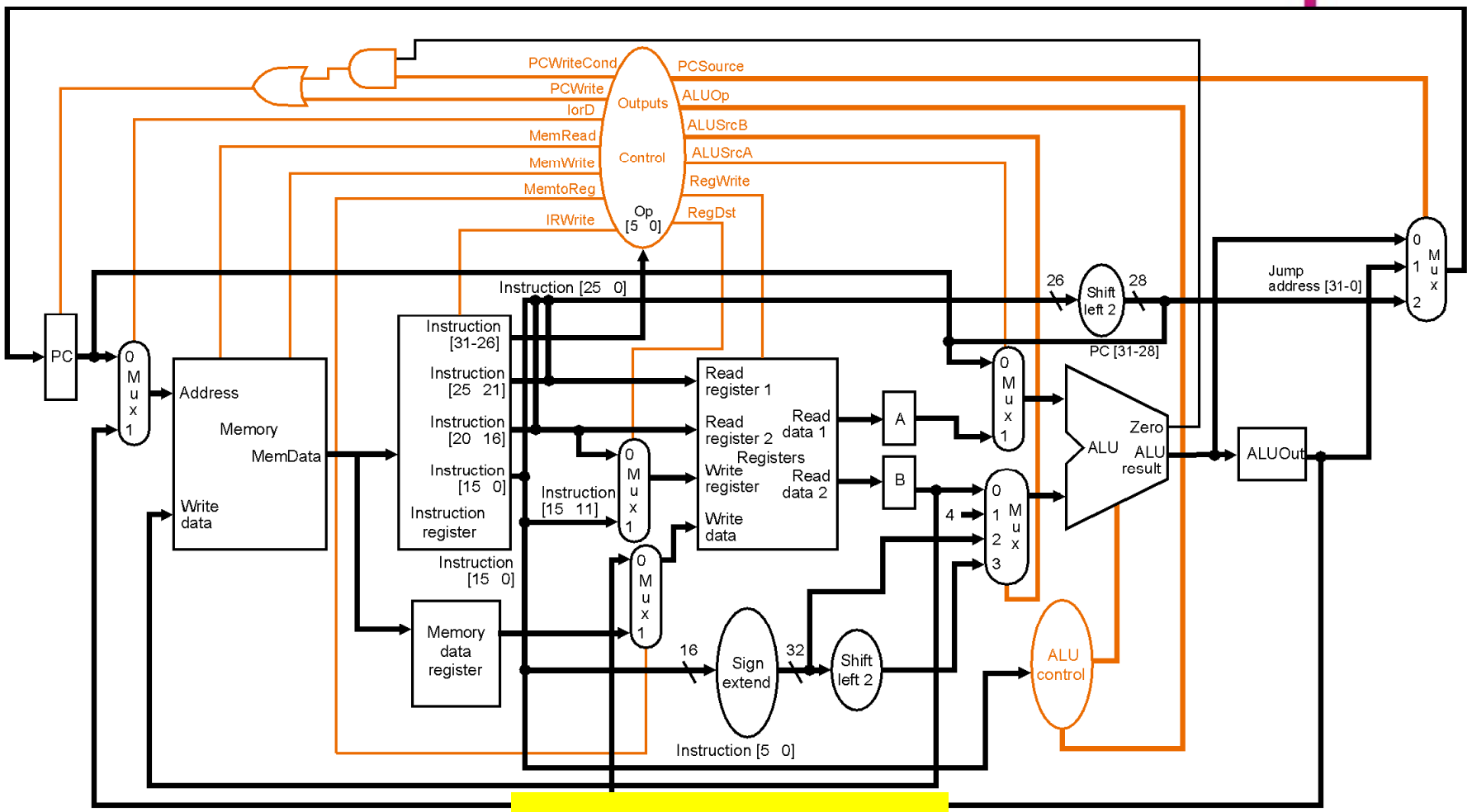
Step	R-type	Memory	Branch
Instruction Fetch	IR = Mem[PC] PC = PC + 4		
Instruction Decode/ register fetch	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUout = PC + (sign-extend(IR[15-0]) << 2)		
Execution, address computation, branch completion	ALUout = A op B	ALUout = A + sign-extend(IR[15-0])	if (A==B) then PC=ALUout
Memory access or R-type completion	Reg[IR[15-11]] = ALUout	memory-data = Mem[ALUout] *or* Mem[ALUout]= B	
Write-back		Reg[IR[20-16]] = memory-data	

Complete Multi-Cycle Datapath



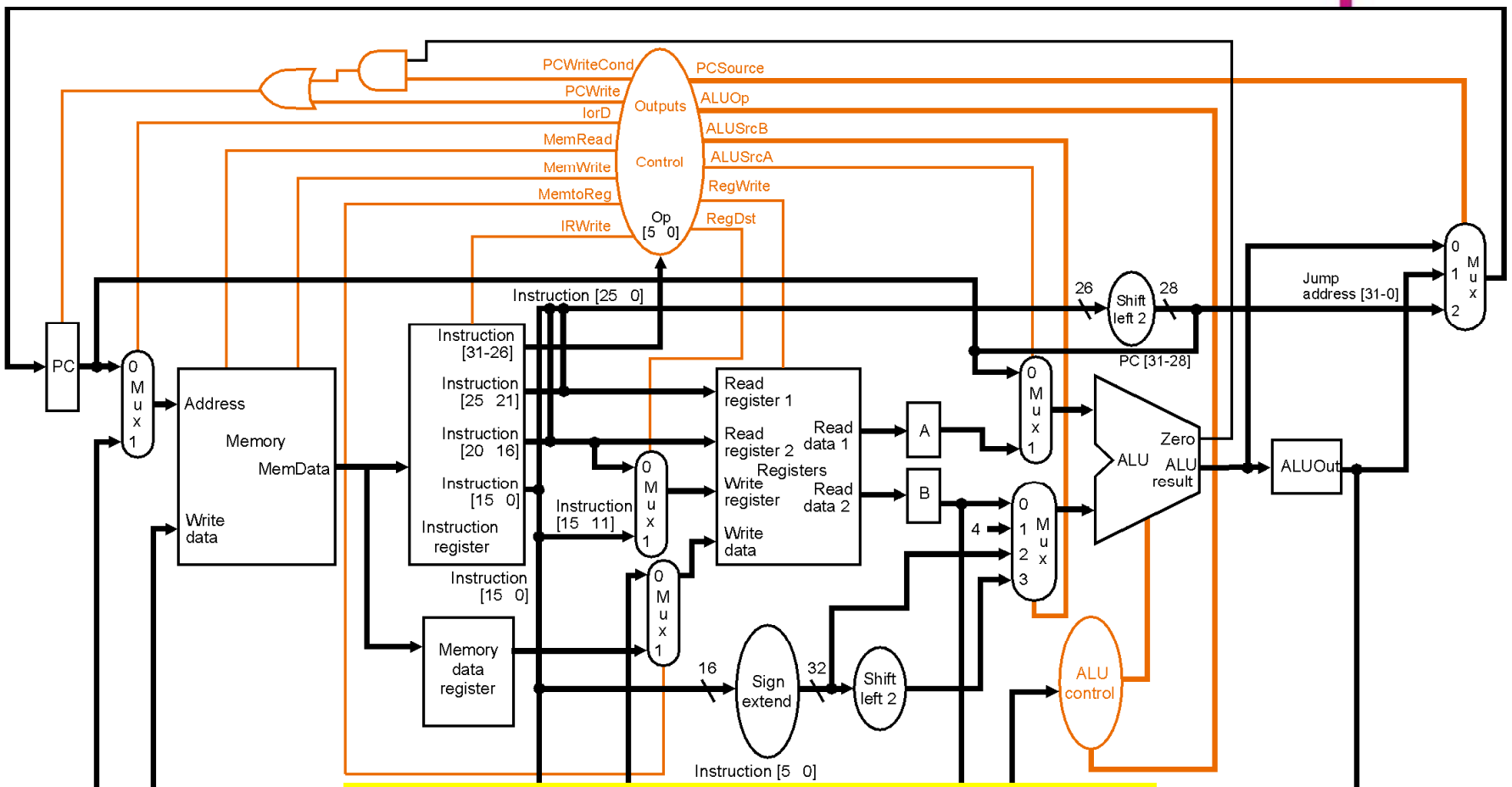
New Instruction Appears Out of Nowhere? Which One?

1. Instruction Fetch



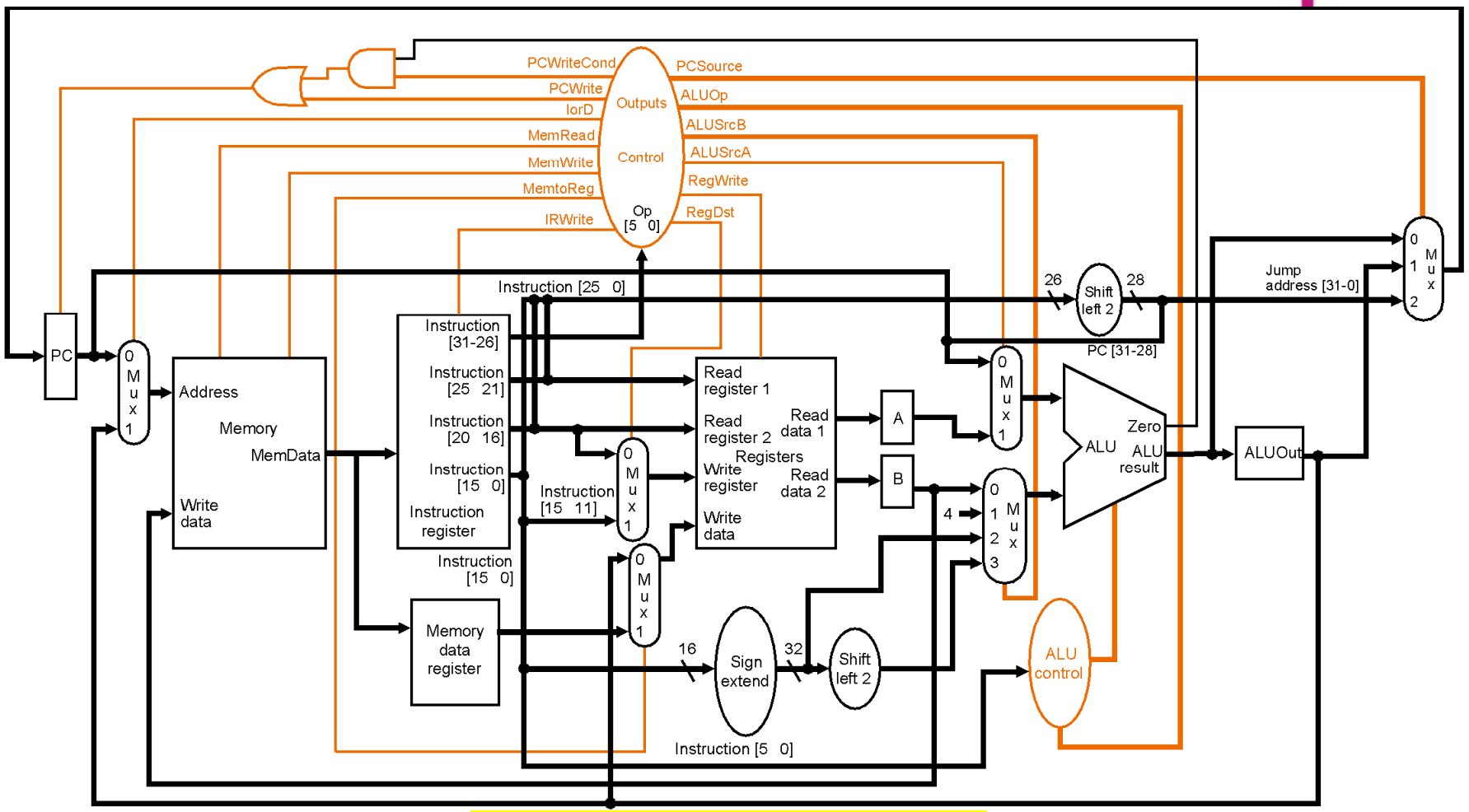
IR = Memory[PC]
PC = PC + 4

2. Instruction Decode and Register Fetch



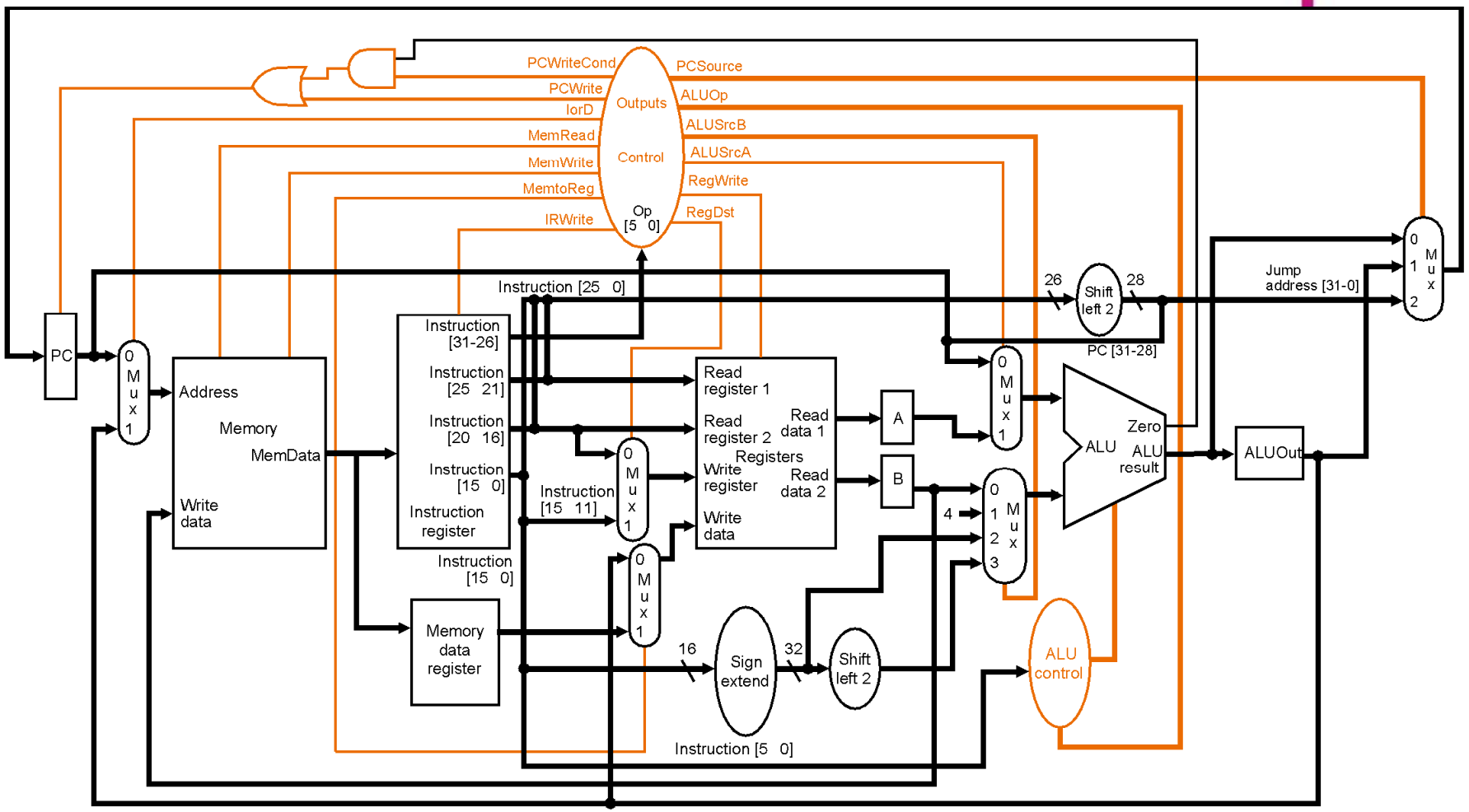
A = Register[IR[25-21]]
B = Register[IR[20-16]]
ALUOut = PC + (sign-extend (IR[15-0]) << 2)

3. Branch Completion



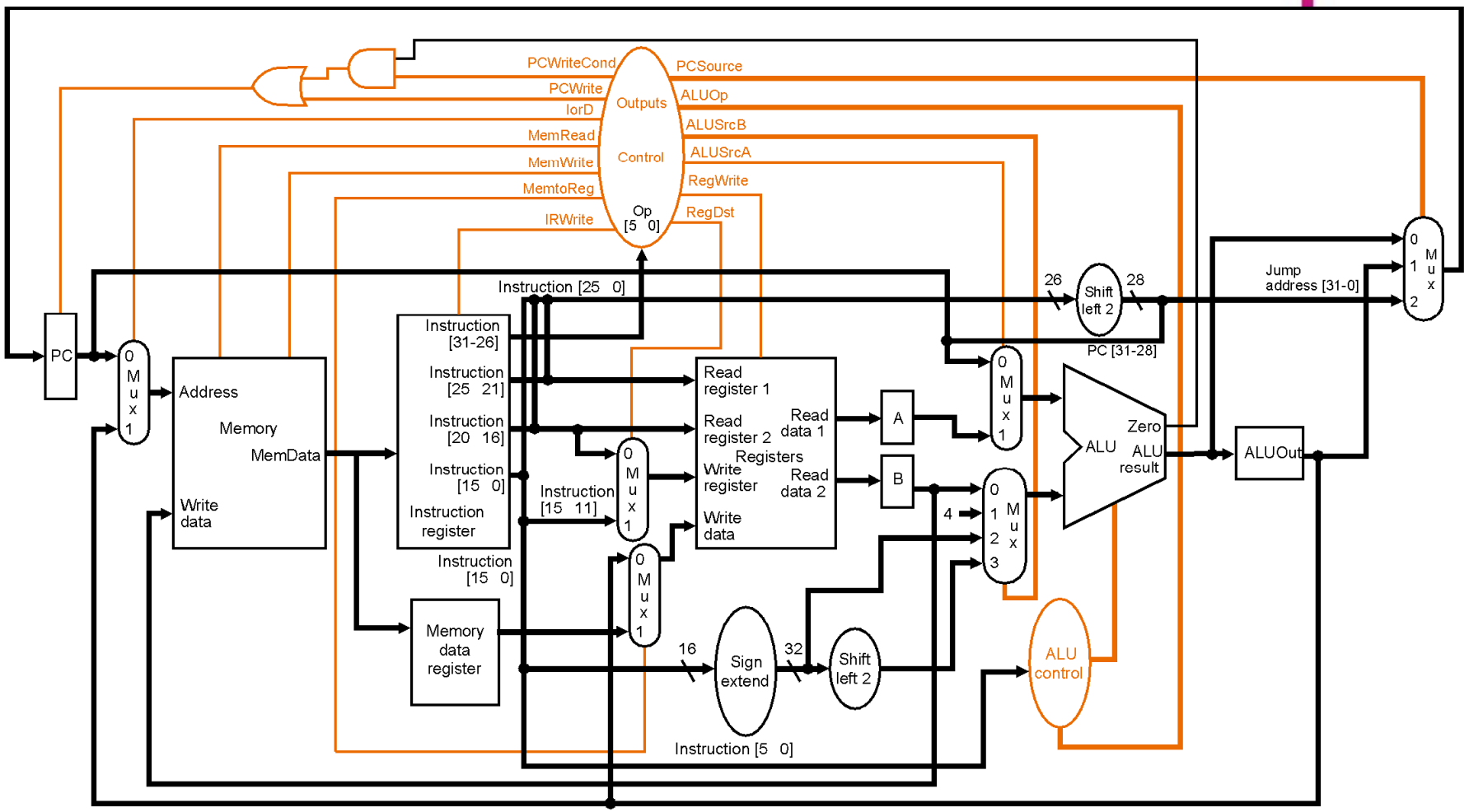
if (A == B) PC = ALUOut

3. Execution (R-Type)



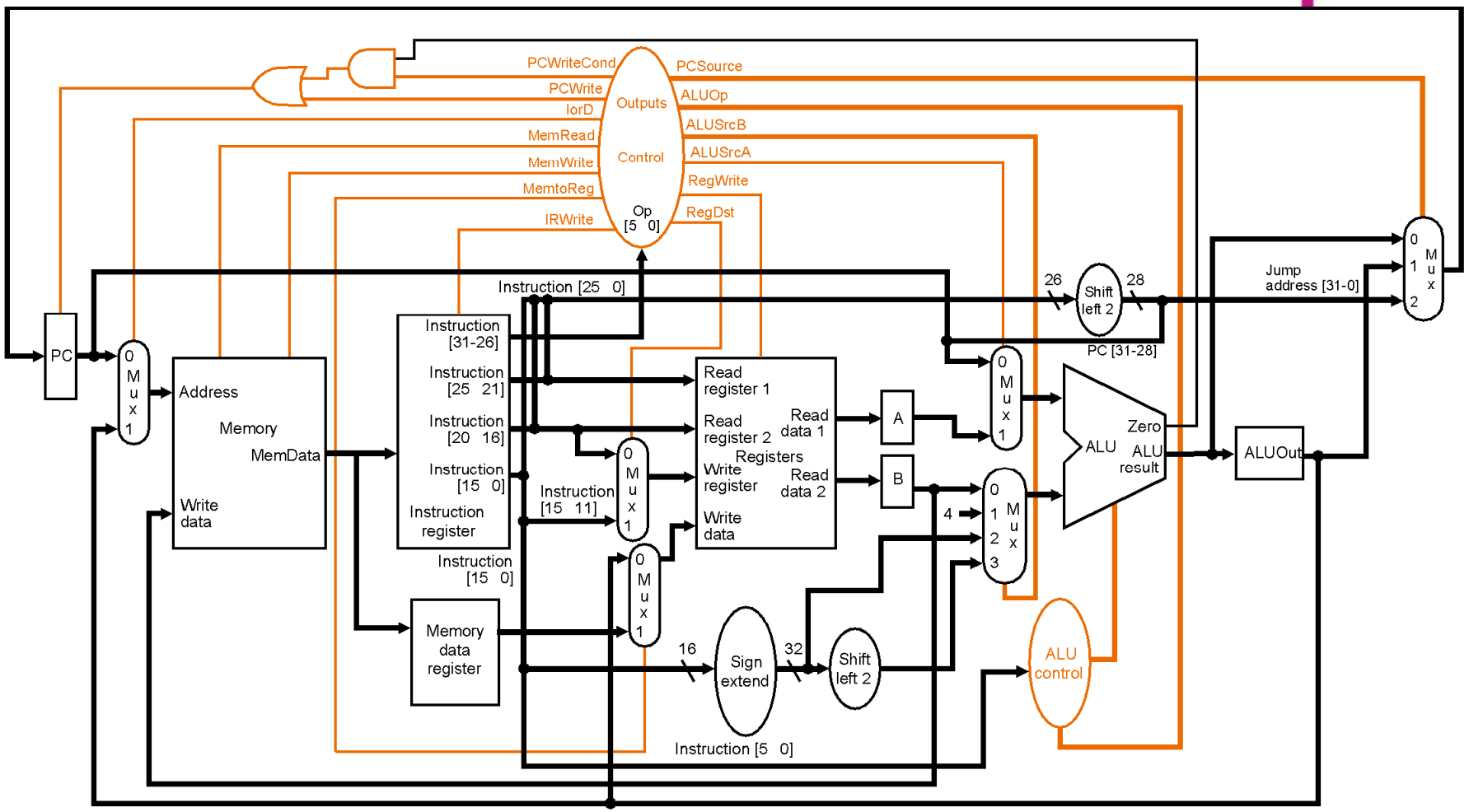
ALUout = A op B

4. R-Type Completion



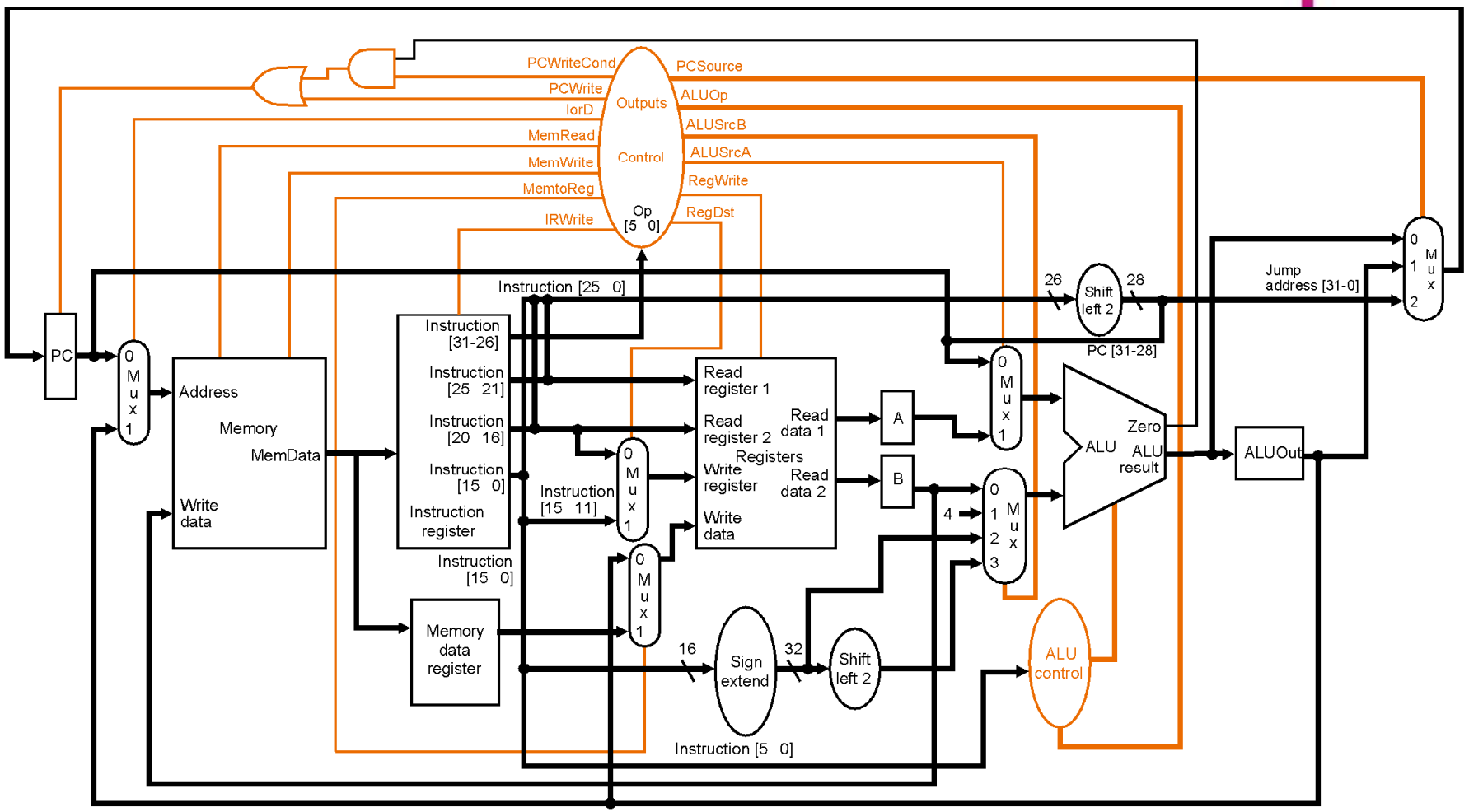
Reg[IR[15-11]] = ALUout

3. Memory Address Computation



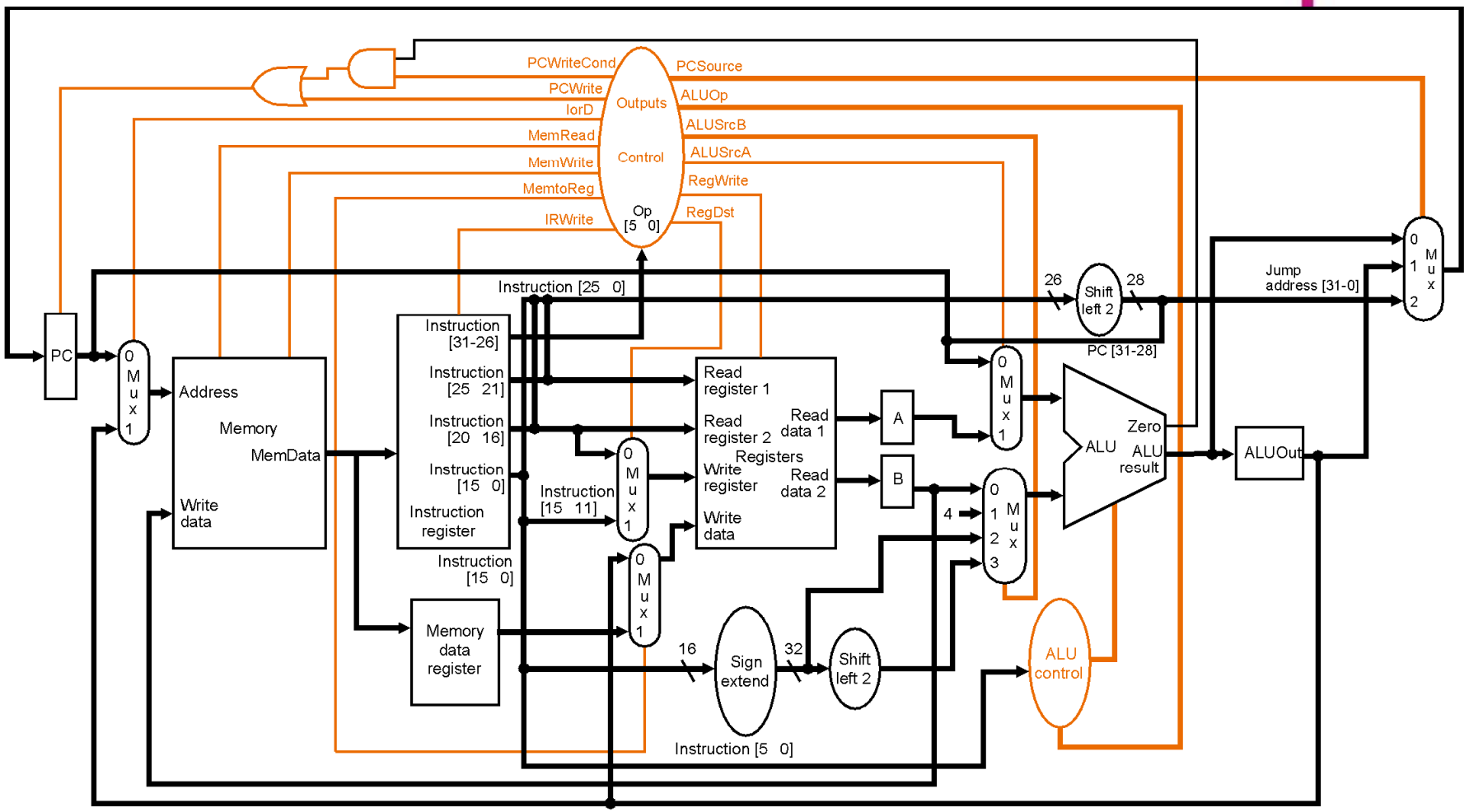
ALUout = A + sign-extend(IR[15-0])

4. Memory Access Load



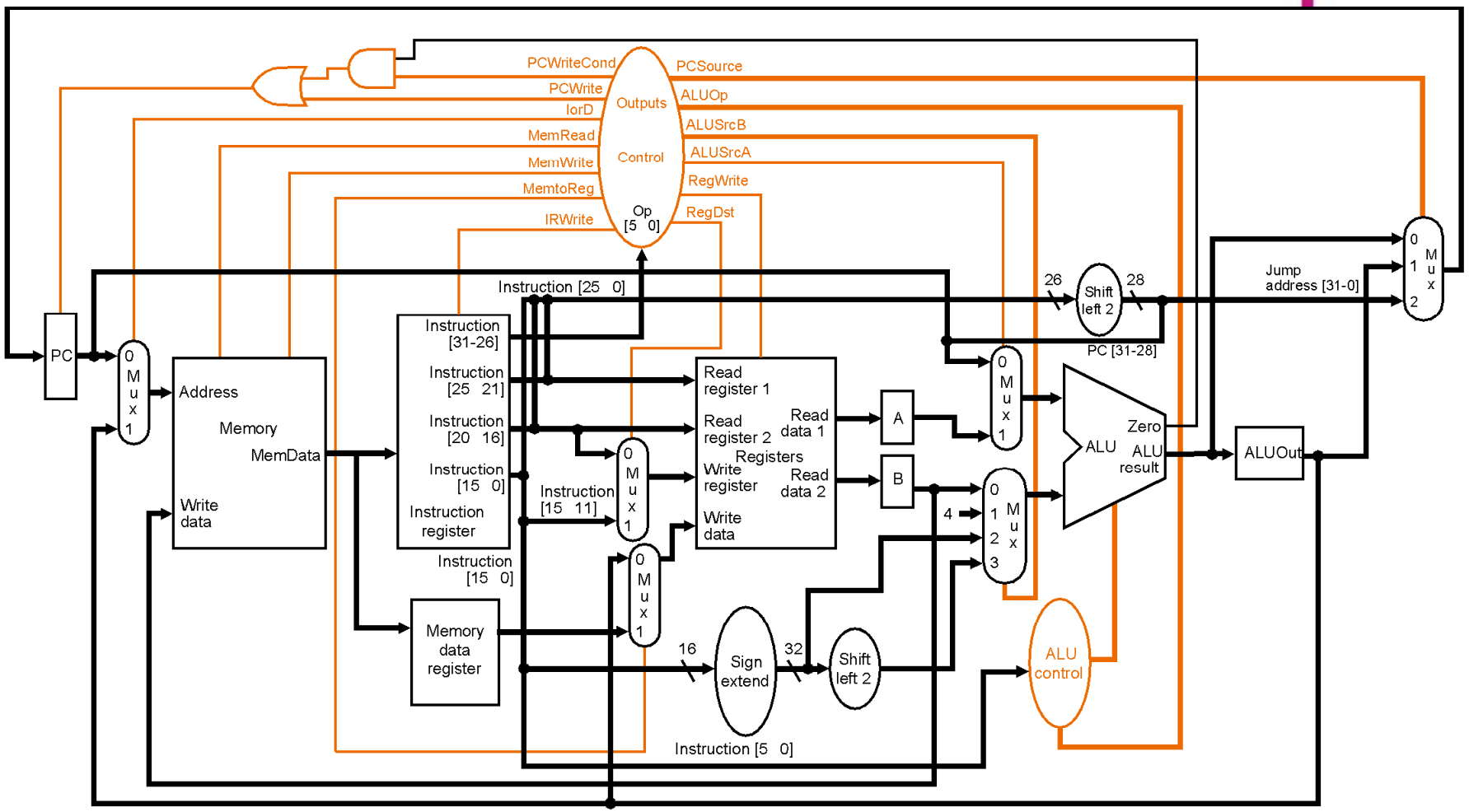
memory-data = Memory[ALUout]

4. Memory Access Store



Memory[ALUout] = B

5. Load Write-Back

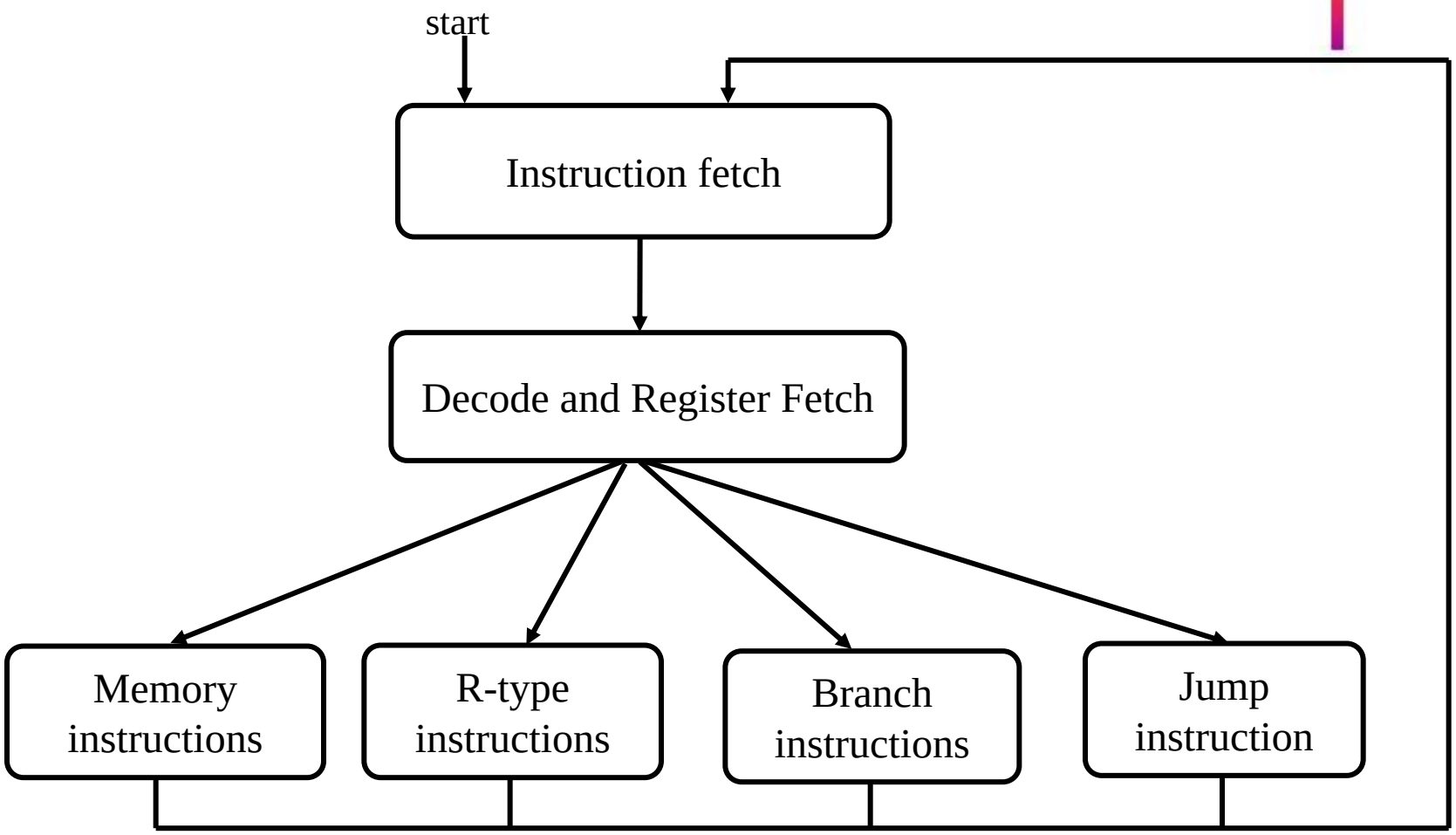


Reg[IR[20-16]] = memory-data

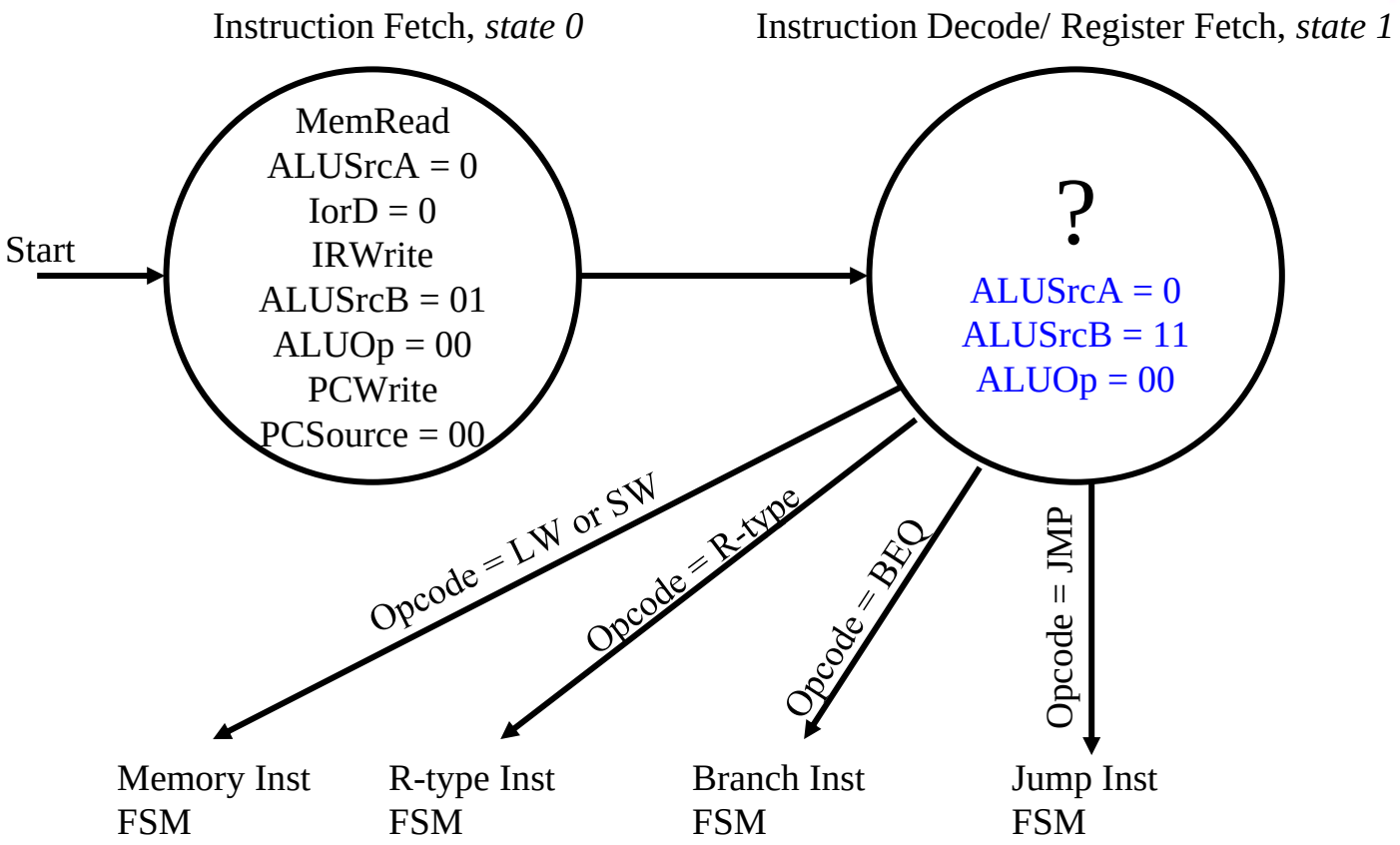
What About the Control?

- Single-cycle control used combinational logic
- What does Multi-cycle control use?
  - FSM defines a succession of states, transitions between states (based on inputs), and outputs (based on state)
  - First two states same for every instruction, next state depends on opcode

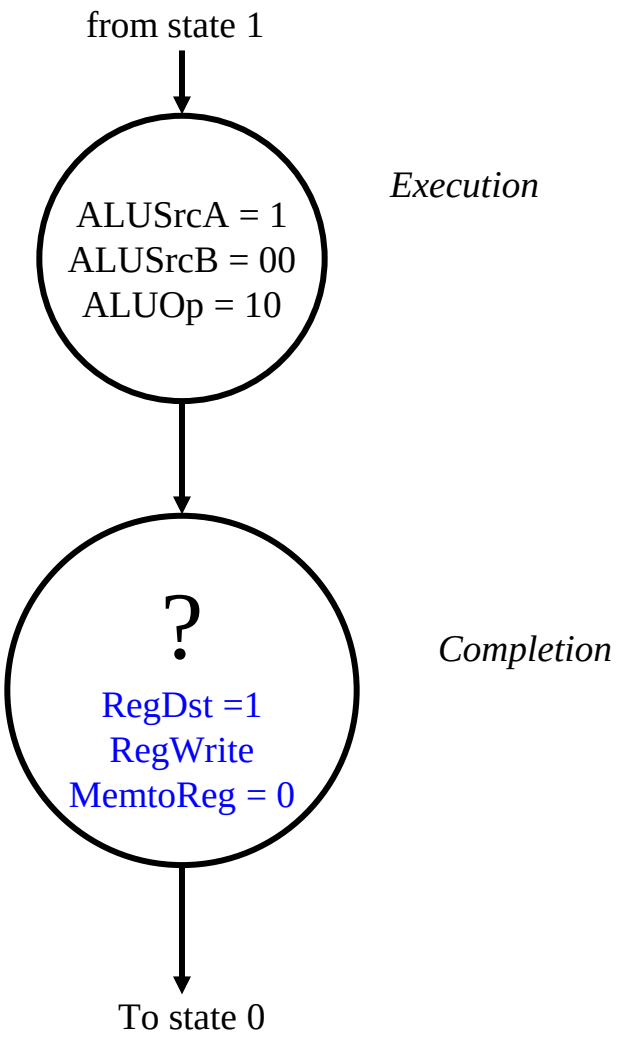
Multi-Cycle Control FSM



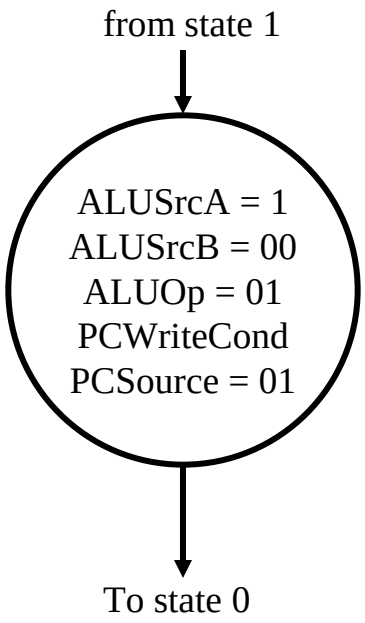
First two states of the FSM



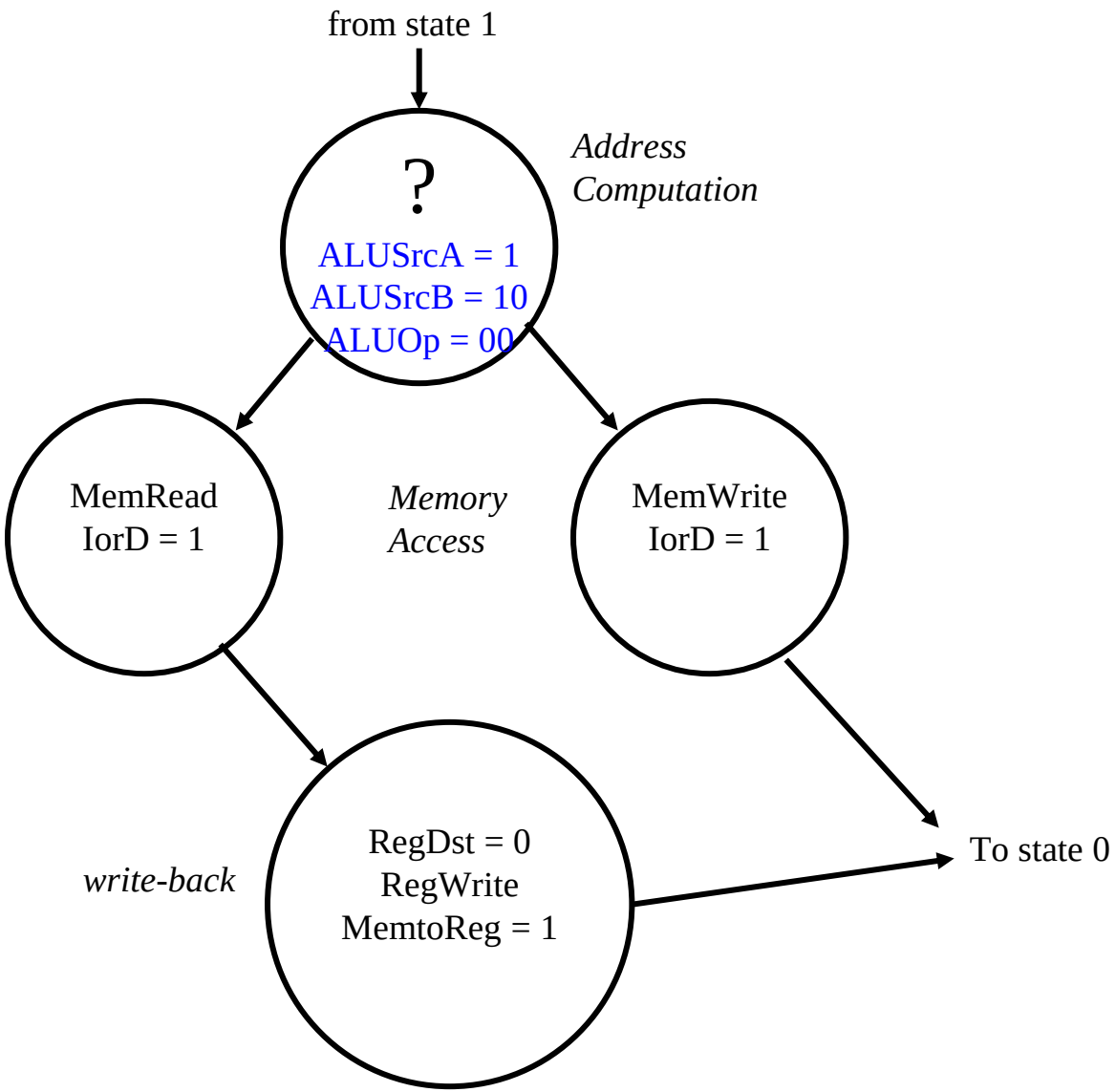
R-type Instructions



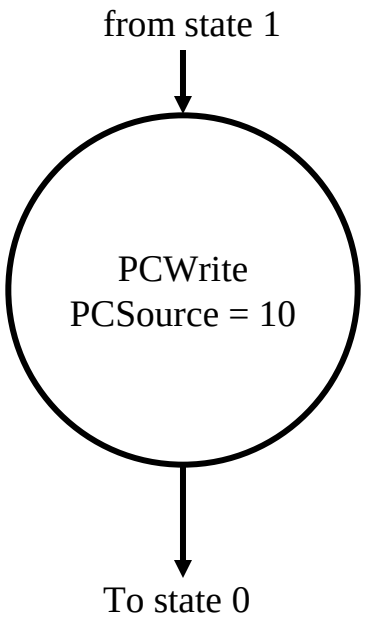
BEQ Instruction



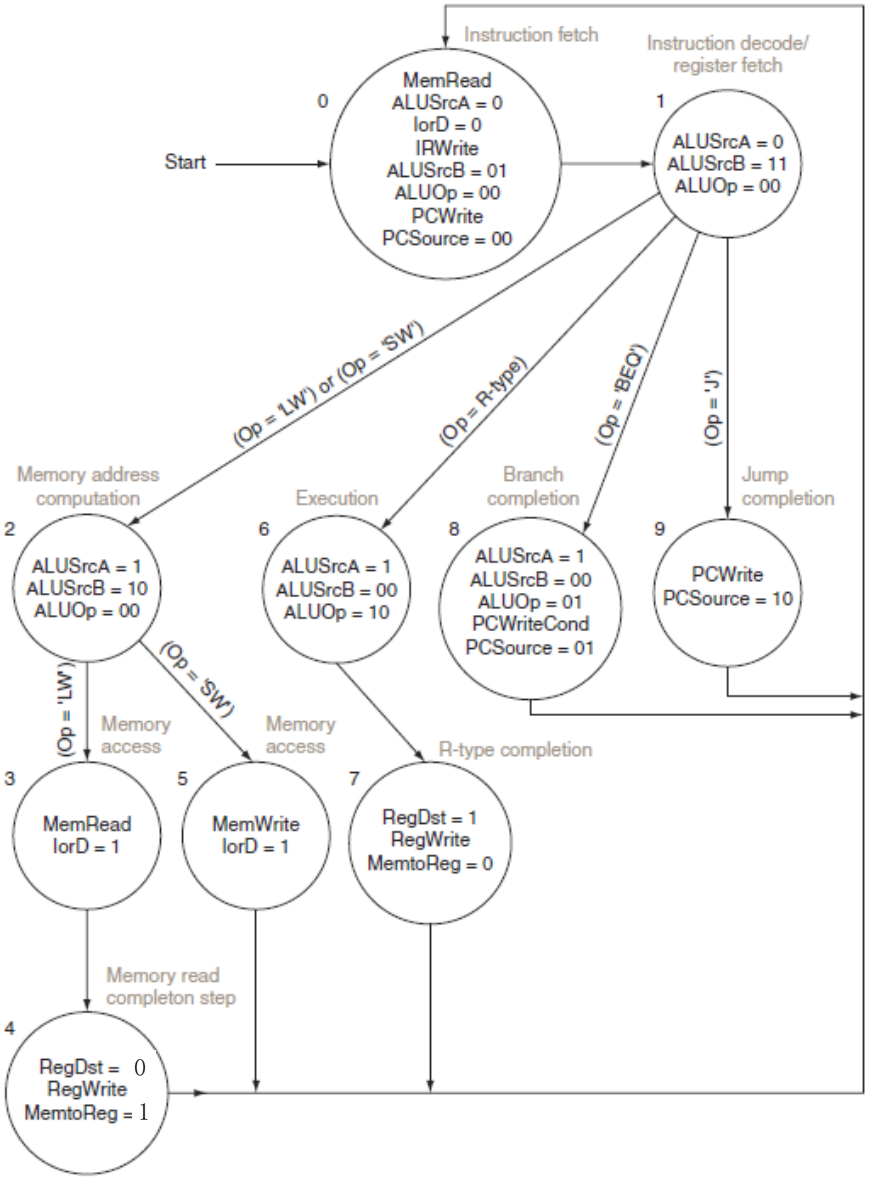
Memory Instructions



JMP Instruction

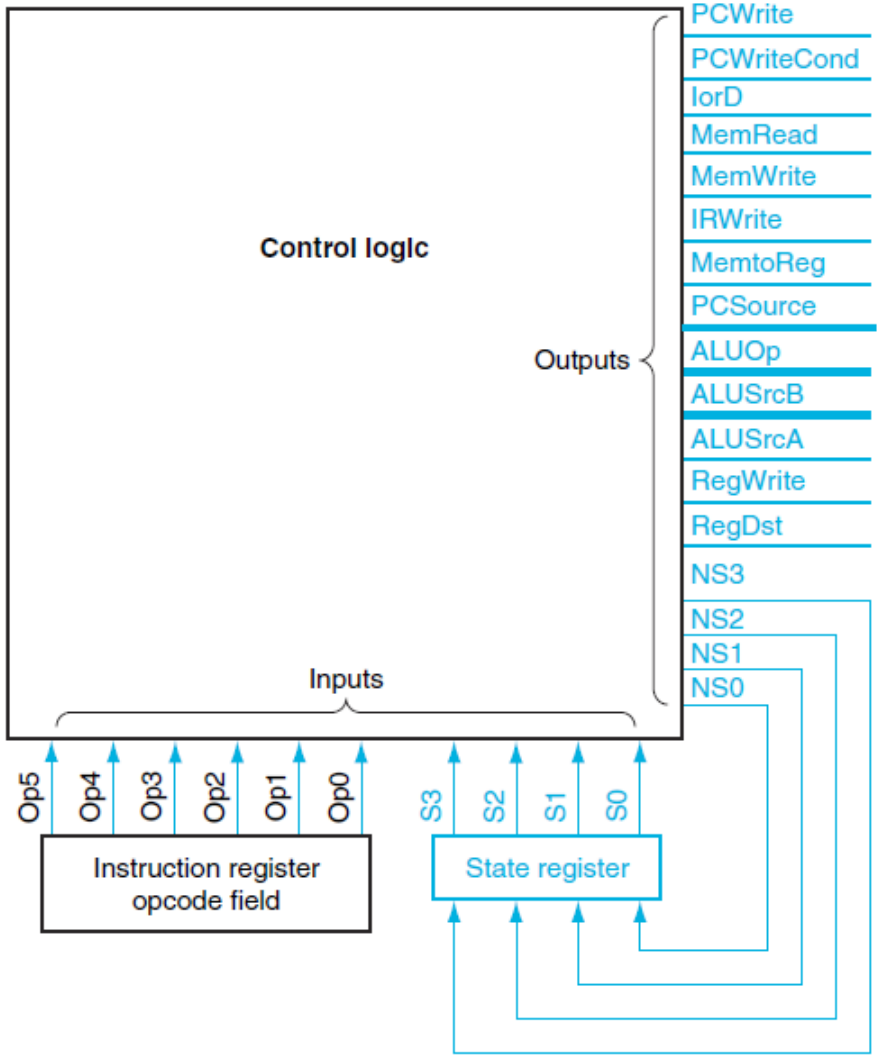


The Whole FSM



Implementing Finite-State Machine Control

- State register that holds the current state
  - S3, S2, S1, and S0
- Combinational logic block to compute the next state and output functions.



Control Truth Table

Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

Multi-Cycle Key Points

- Performance gain achieved from variable-length instructions
- ET = IC * CPI * cycle time
- Required very few new state elements
- More, and more complex, control signals
- Control requires FSM