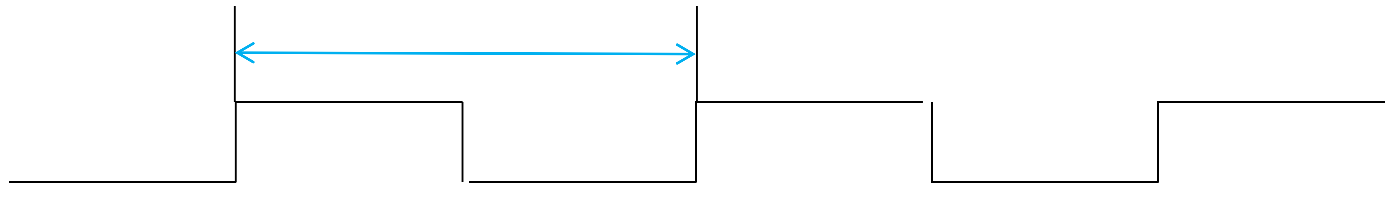
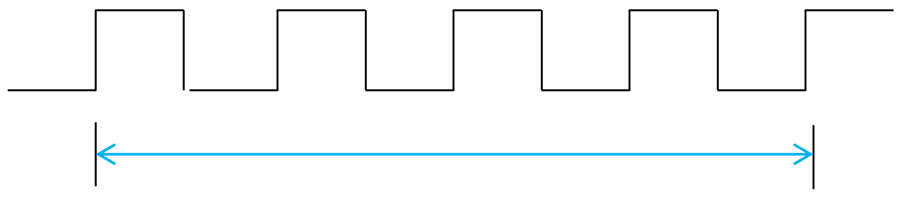
Computer Organization

Lecture 16 - Multi-Cycle Processor Design





Multicycle Execution - Key Idea

- Break instruction execution into multiple cycles
- One clock cycle for each major task
  1. Instruction Fetch
  2. Instruction Decode and Register Fetch
  3. Execution, memory address computation, or branch computation
  4. Memory access / R-type instruction completion
  5. Memory read completion
- Share hardware to simplify datapath

Characteristics of Multicycle Design

- **Instructions take more than one cycle**
  - **Some instructions take more cycles than others**
  - **Clock cycle is shorter than single-cycle clock**
- **Reuse of major components simplifies datapath**
  - **Single ALU for all calculations**
  - **Single memory for instructions and data**
  - **But, added registers needed to store values across cycles**
- **Control Unit Implemented w/ State Machine**
  - **Control signals are no longer a function of just the instruction**

Review: Register Transfers

- **Instruction Fetch**

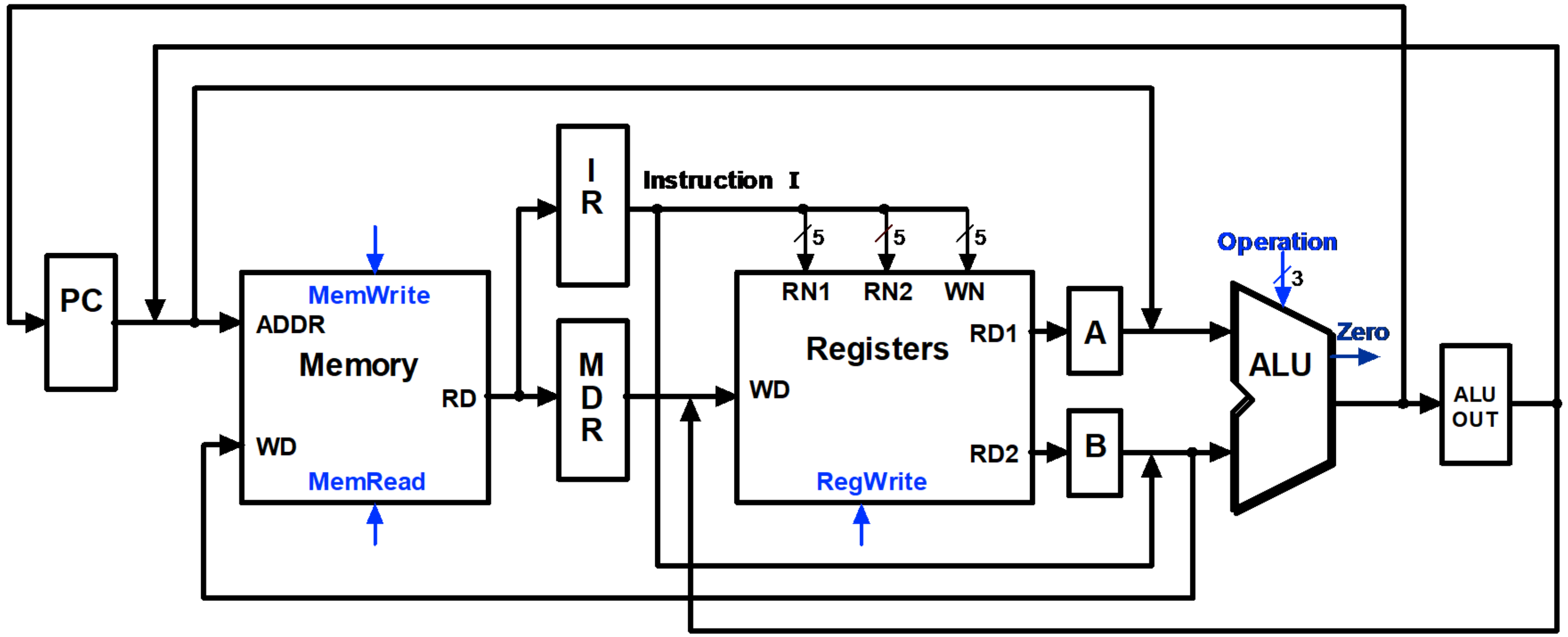
  Instruction <- MEM[PC];   PC = PC + 4;

- **Instruction Execution**

Key idea: break into multiple cycles!

Instr.	Registration Transfers
add	R[rd] ← R[rs] + R[rt];
sub	R[rd] ← R[rs] - R[rt];
and	R[rd] ← R[rs] & R[rt];
or	R[rd] ← R[rs] | R[rt];
lw	R[rt] ← MEM[R[rs] + s_extend(offset)];
sw	MEM[R[rs] + sign_extend(offset)] ← R[rt]; PC ← PC + 4
j	PC ← upper(PC)@(address << 2)

Multicycle Datapath - High-Level View



(Equivalent to 3rd. Ed. Fig. 5.25, p. 318)

Summary - Multicycle Execution

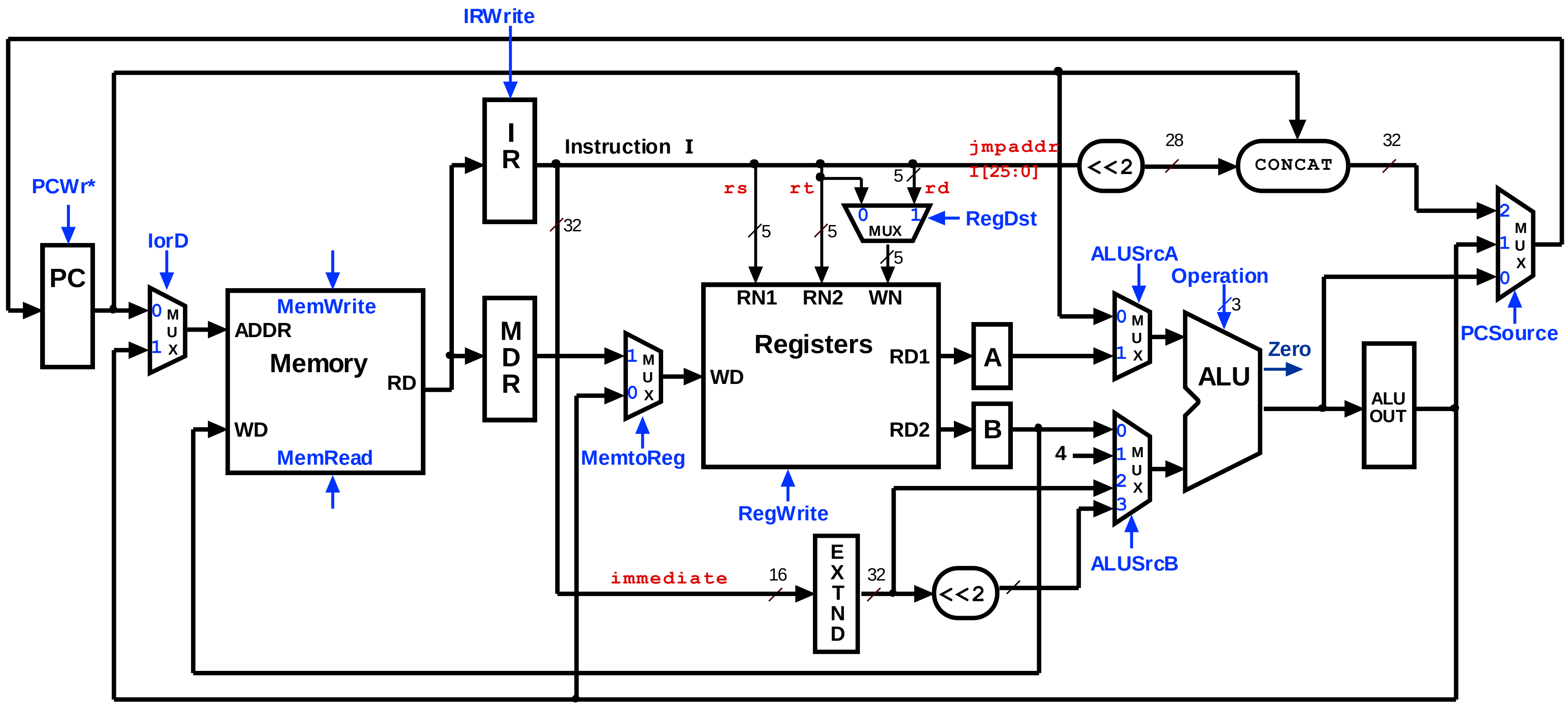
- Instructions take between 3 and 5 clock cycles

	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
(1)	Instruction fetch	IR = Memory[PC] PC = PC + 4			
(2)	Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
(3)	Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
(4)	Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
(5)	Memory read completion		Load: Reg[IR[20-16]] = MDR		

(3rd. ed. Fig. 5.30, p. 329)

New registers needed to store values across clock steps!

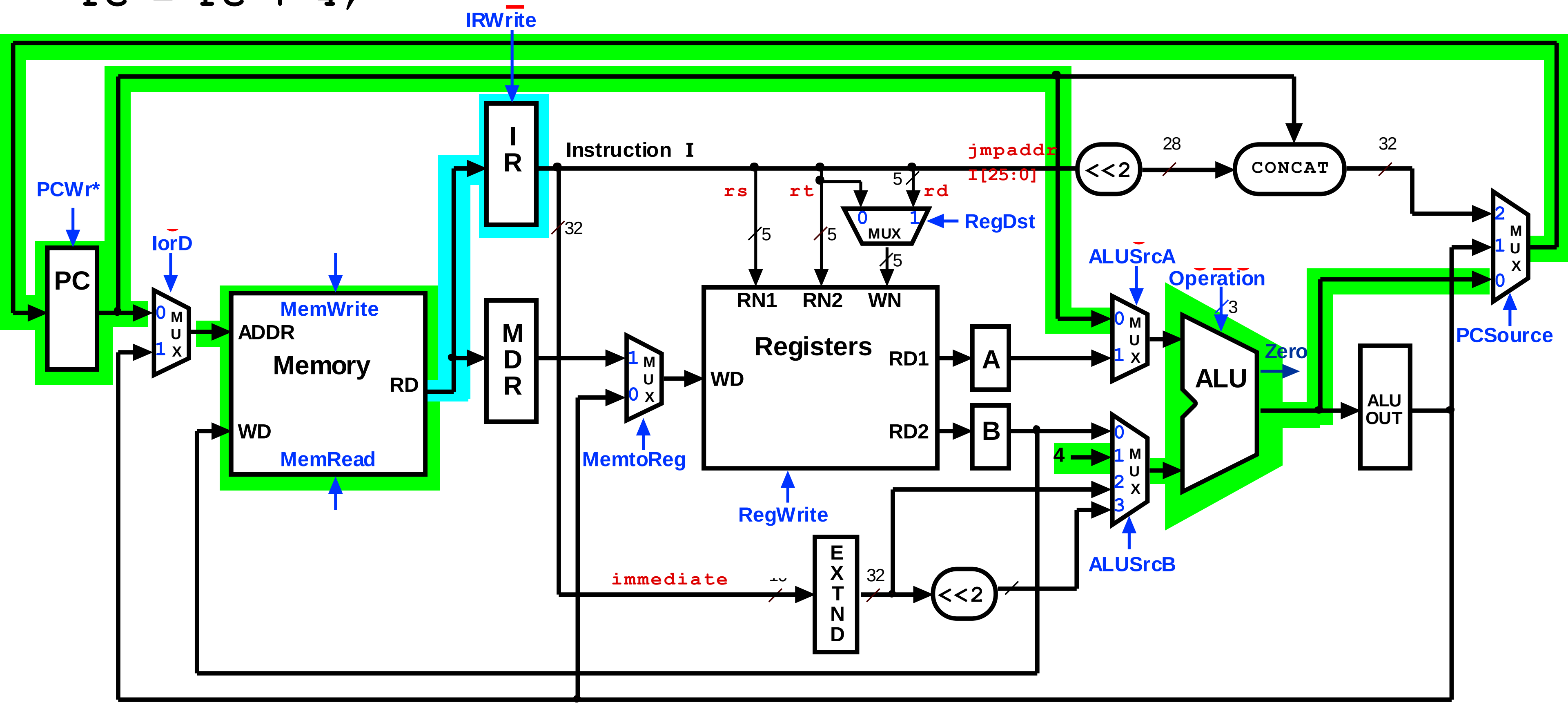
Full Multicycle Datapath



(Equivalent to 3rd Edition Book Fig. 5.27, p. 322 without ALU control

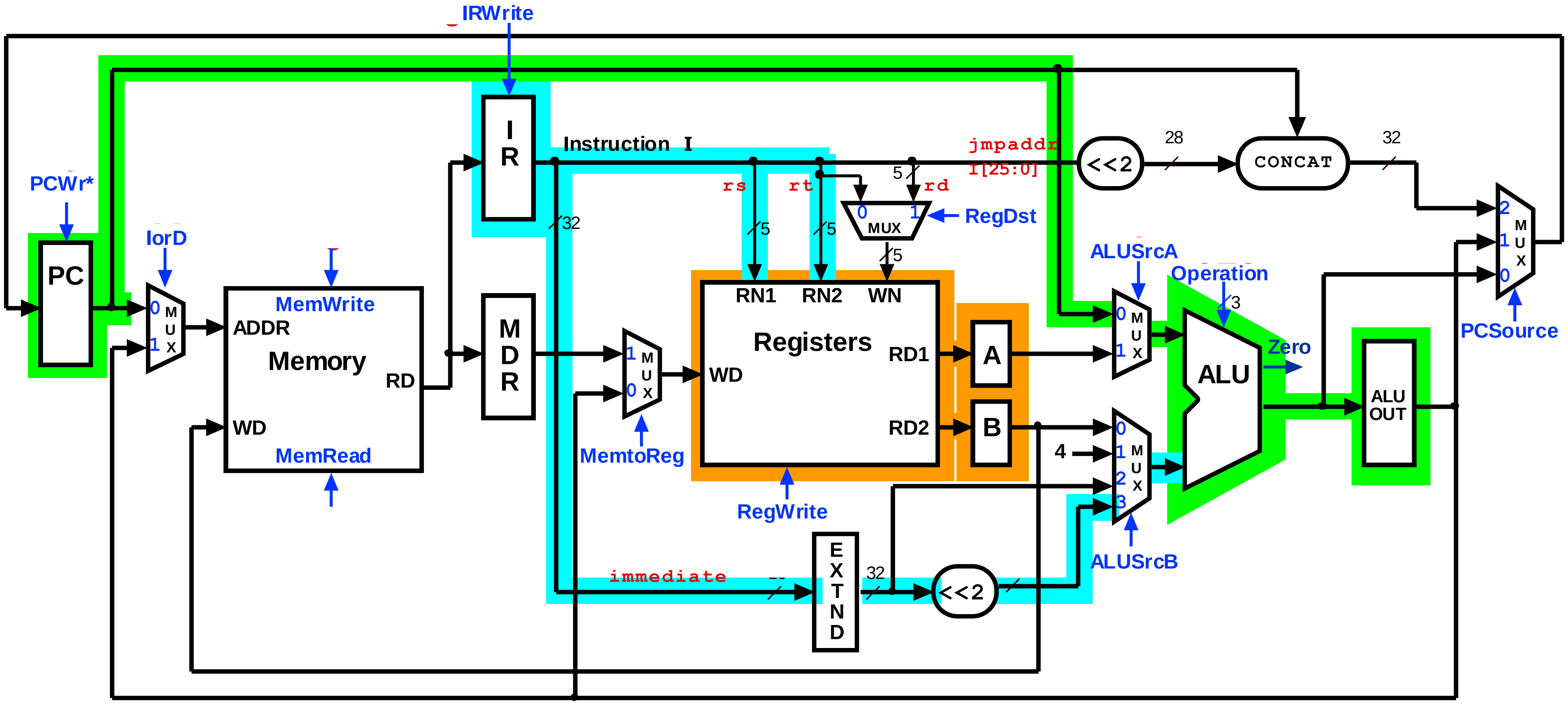
Multicycle Execution Step (1) Fetch

```
IR = Memory[PC];
PC = PC + 4;
```



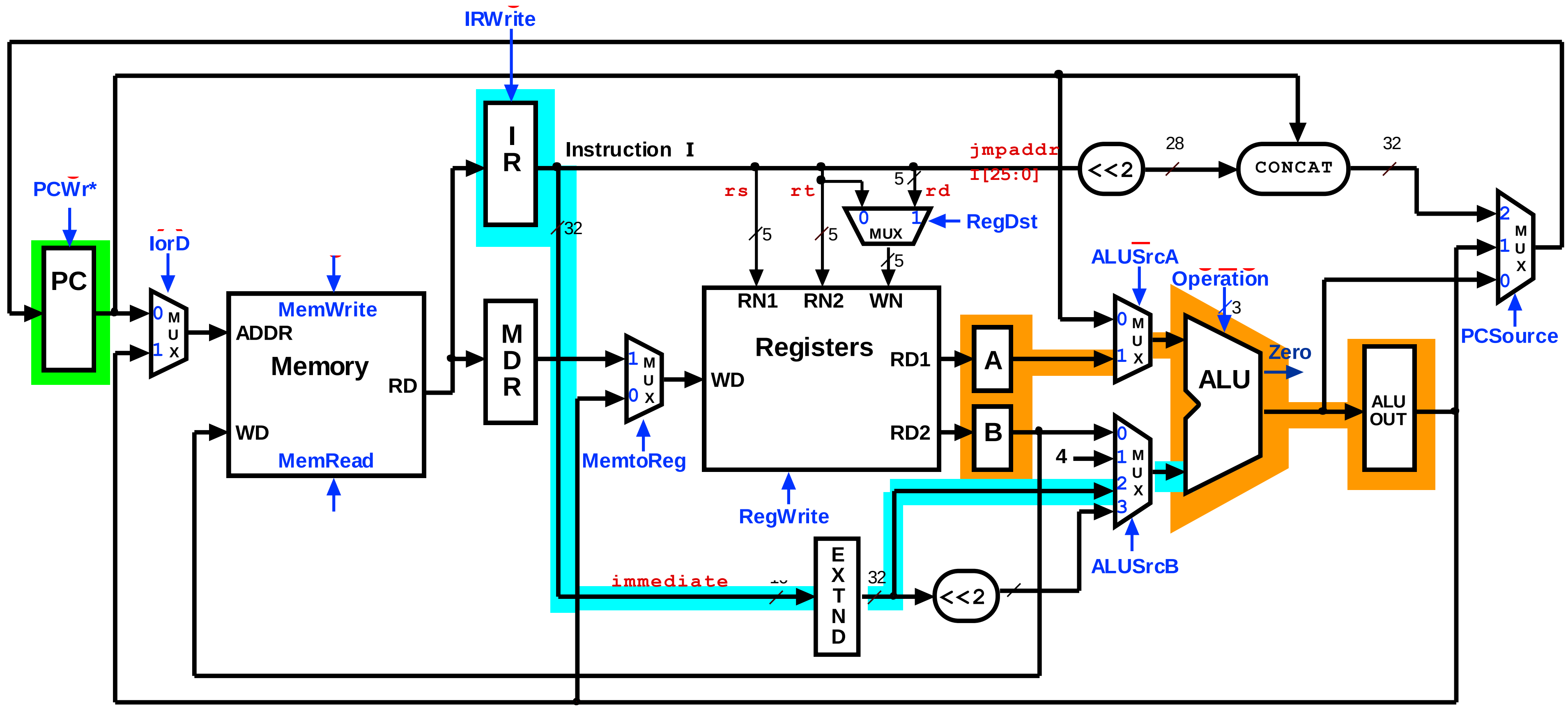
Multicycle Execution Step (2) Instruction Decode and Register Fetch

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



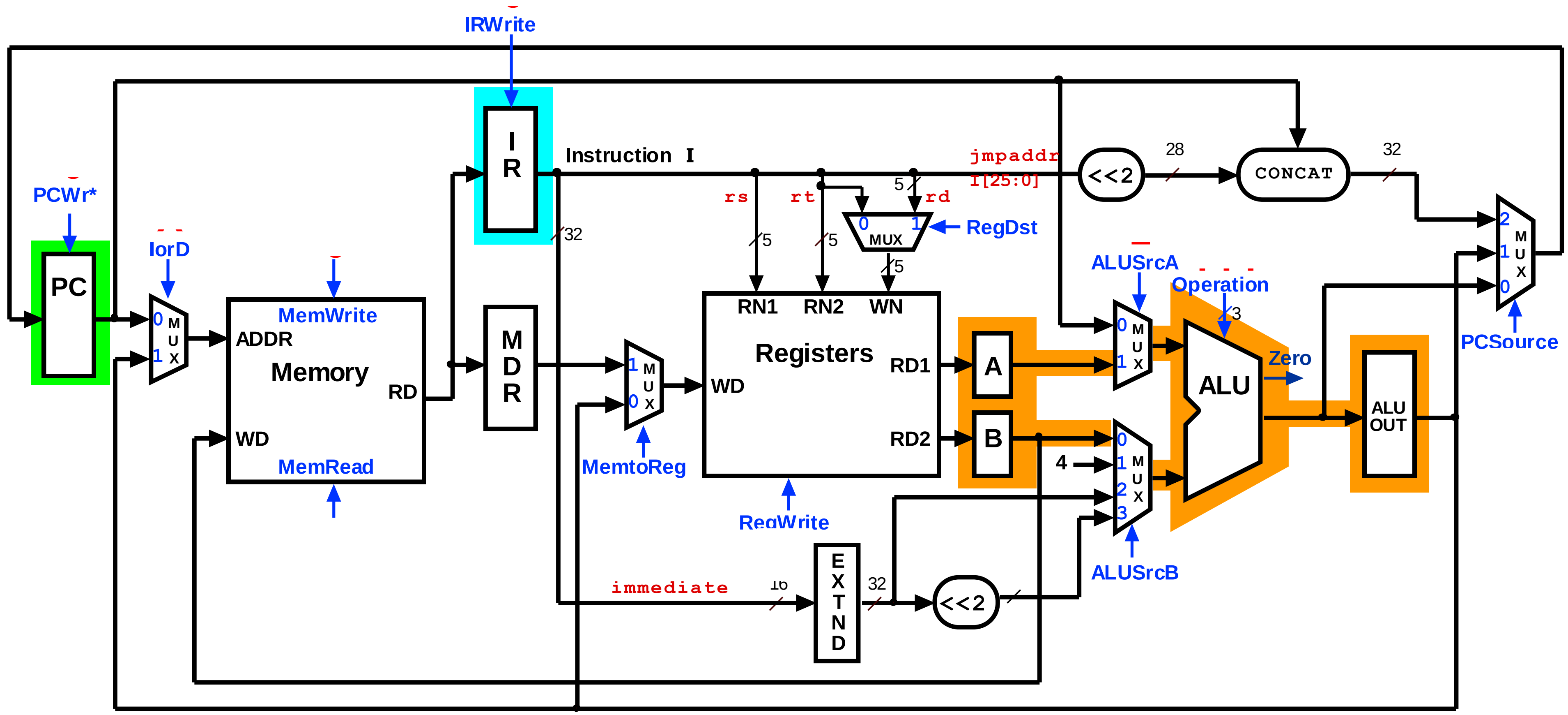
Multicycle Execution Steps (3) Memory Reference Instructions

```
ALUOut = A + sign-extend(IR[15-0]);
```



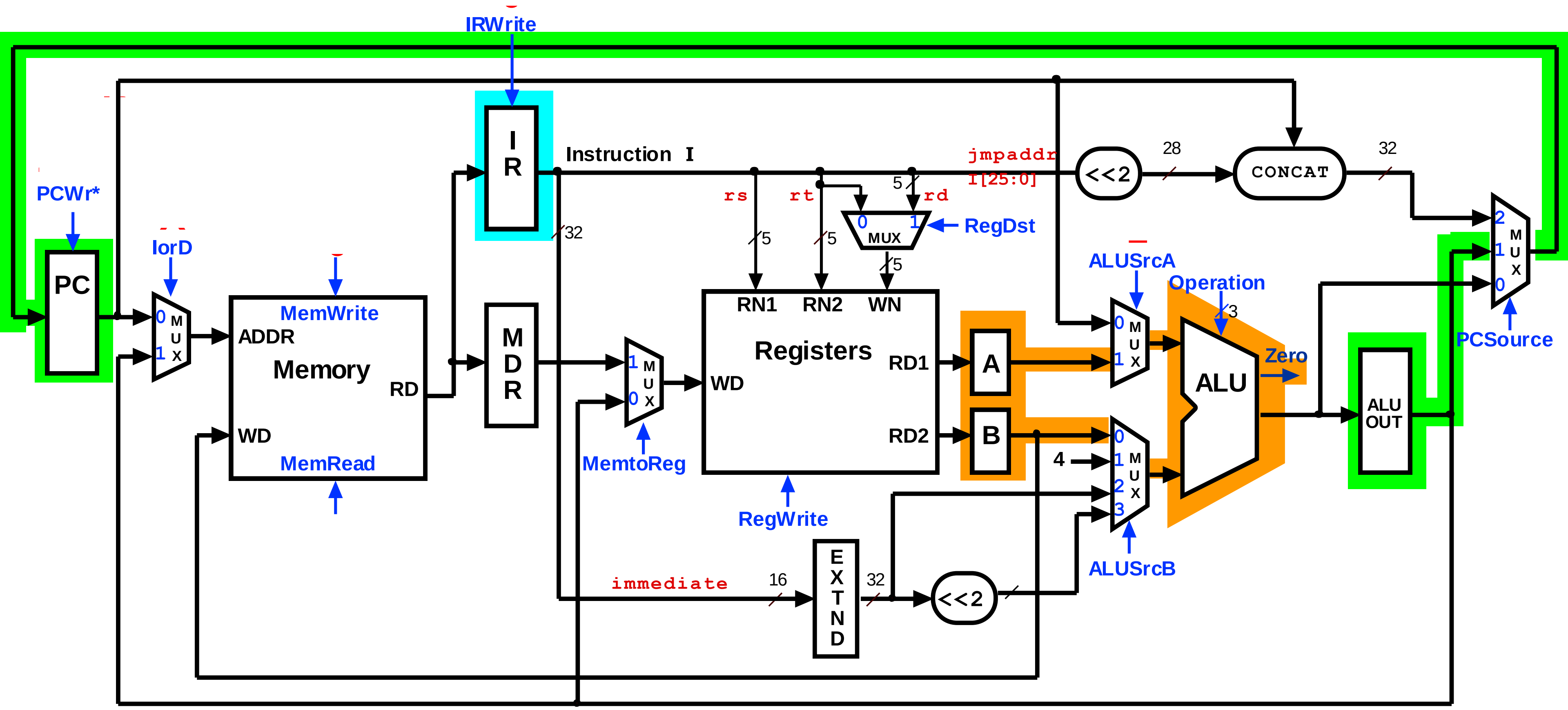
Multicycle Execution Steps (3)
ALU Instruction (R-Type)

`ALUOut = A op B`



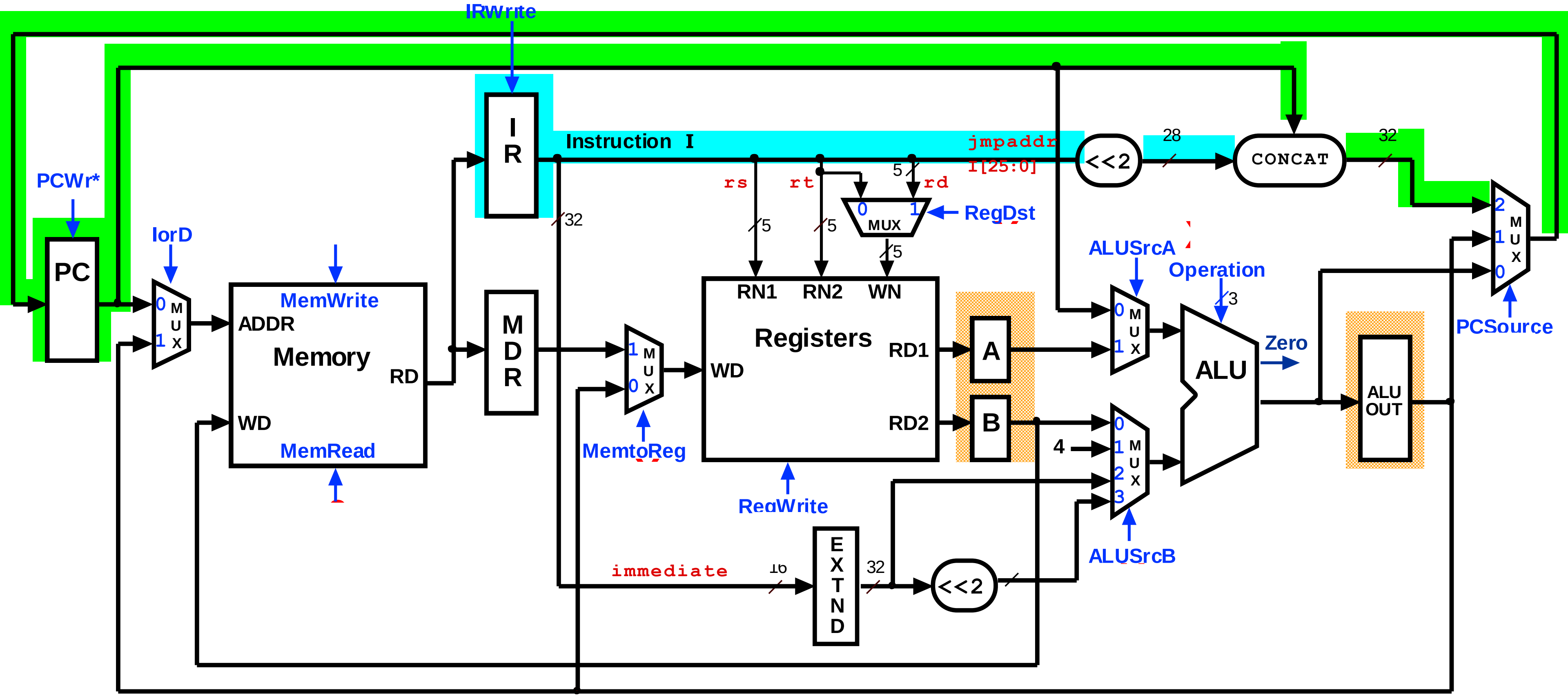
Multicycle Execution Steps (3)
Branch Instructions

```
if (A == B) PC = ALUOut;
```



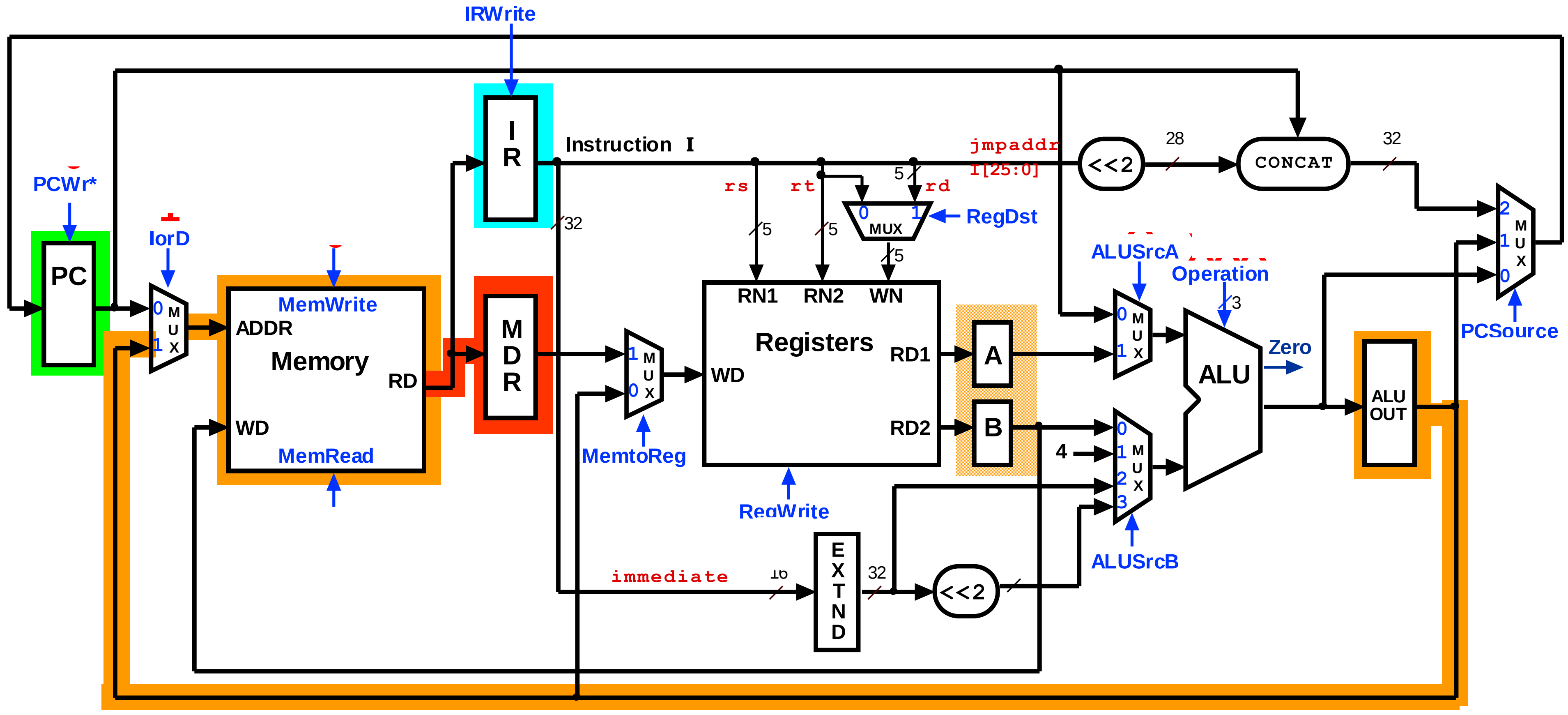
Multicycle Execution Step (3)
Jump Instruction

```
PC = PC[21-28] concat (IR[25-0] << 2)
```



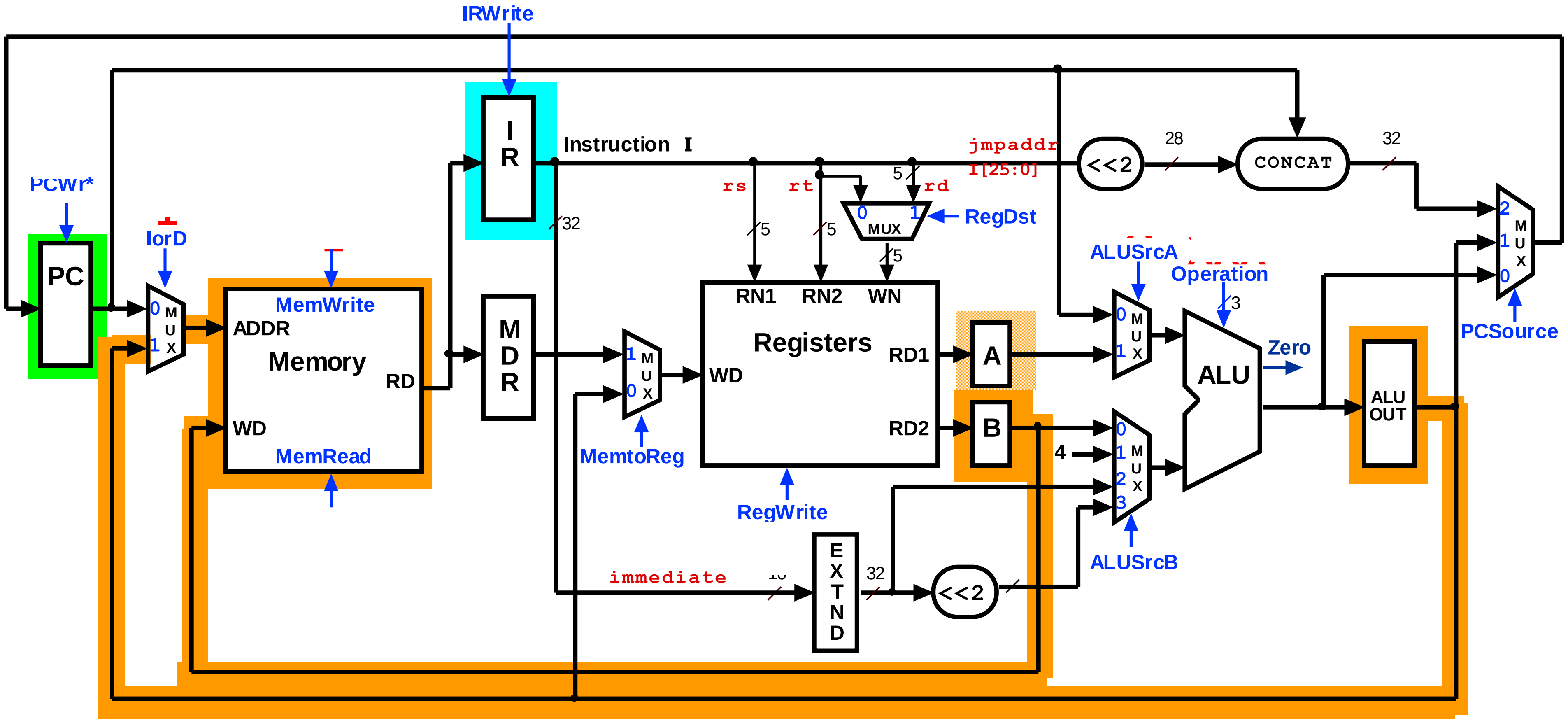
Multicycle Execution Steps (4)
Memory Access - Read (lw)

```
MDR = Memory[ALUOut];
```



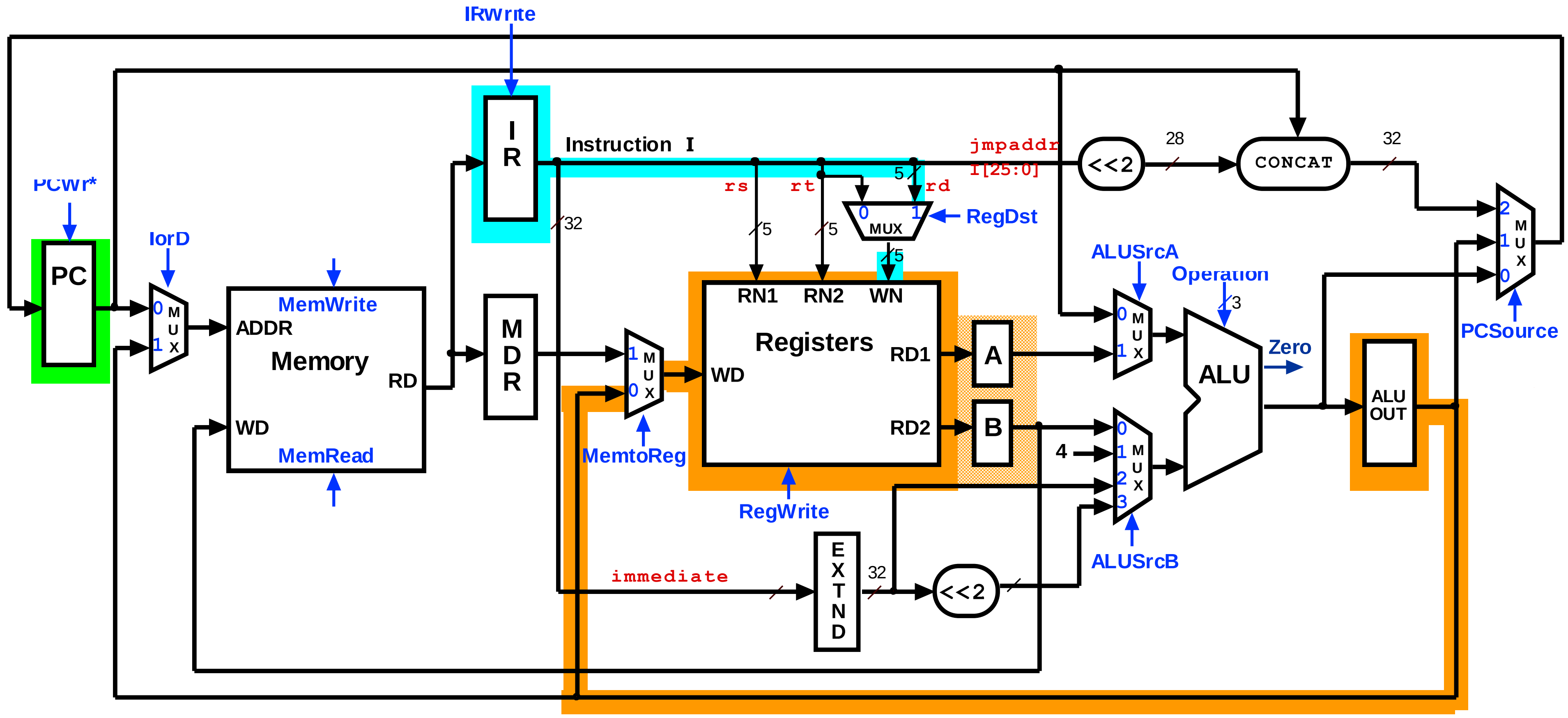
Multicycle Execution Steps (4)
Memory Access - Write (sw)

```
Memory[ALUOut] = B;
```



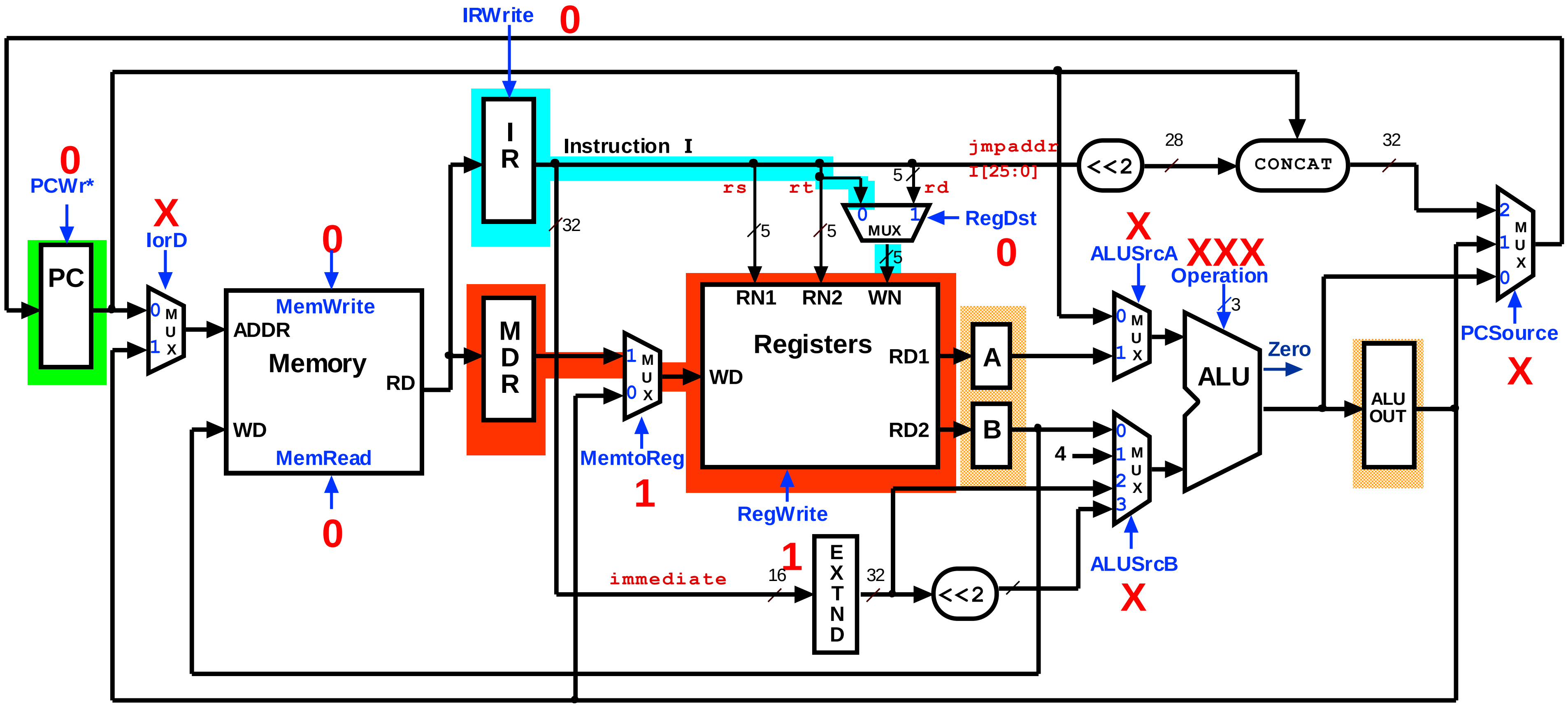
Multicycle Execution Step (4)
ALU Instruction (R-Type)

`Reg[IR[15:11]] = ALUOut;` `(Reg[Rd] = ALUOut)`

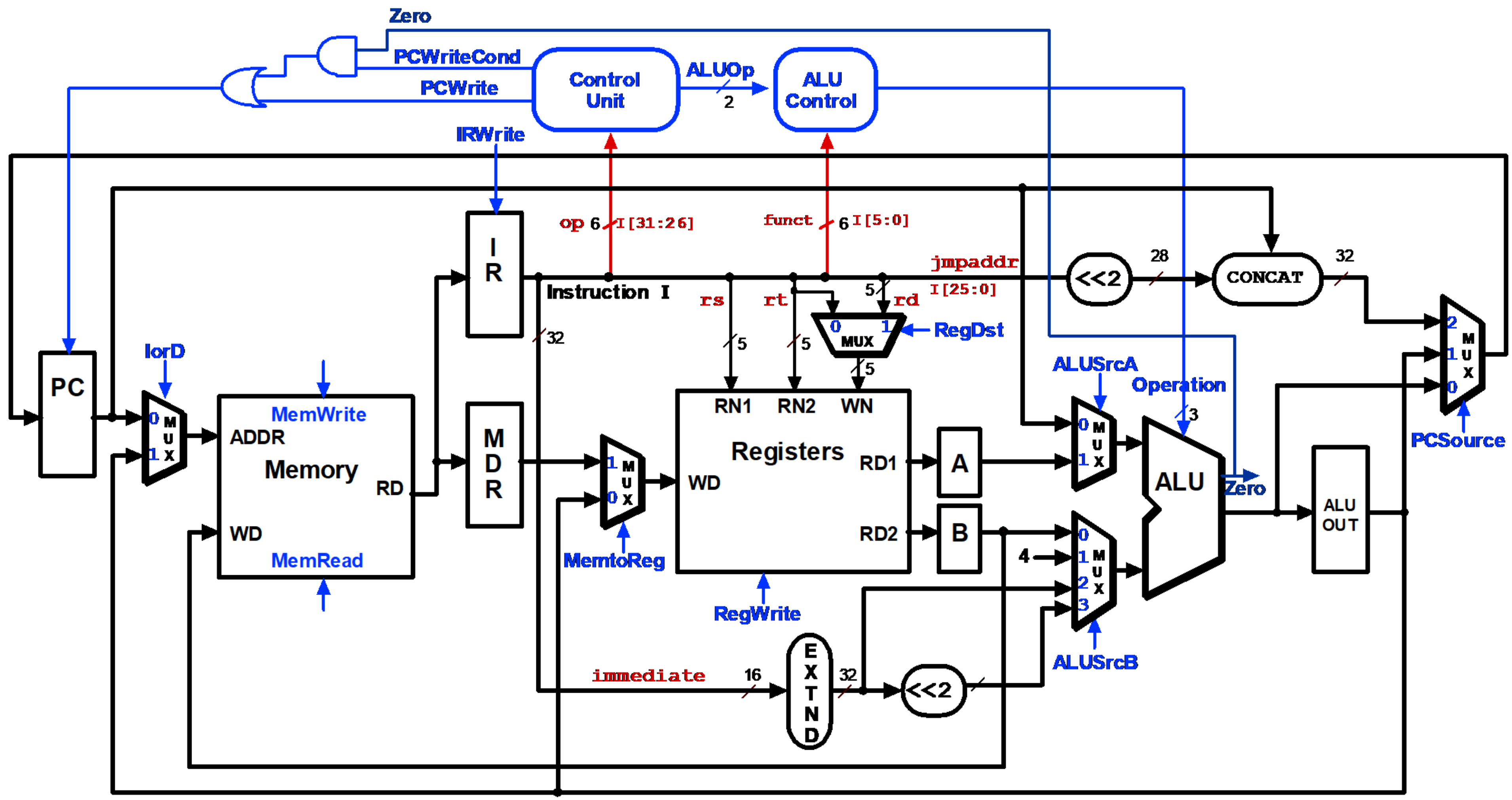


Multicycle Execution Steps (5) Memory Read Completion (lw)

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



Full Multicycle Implementation

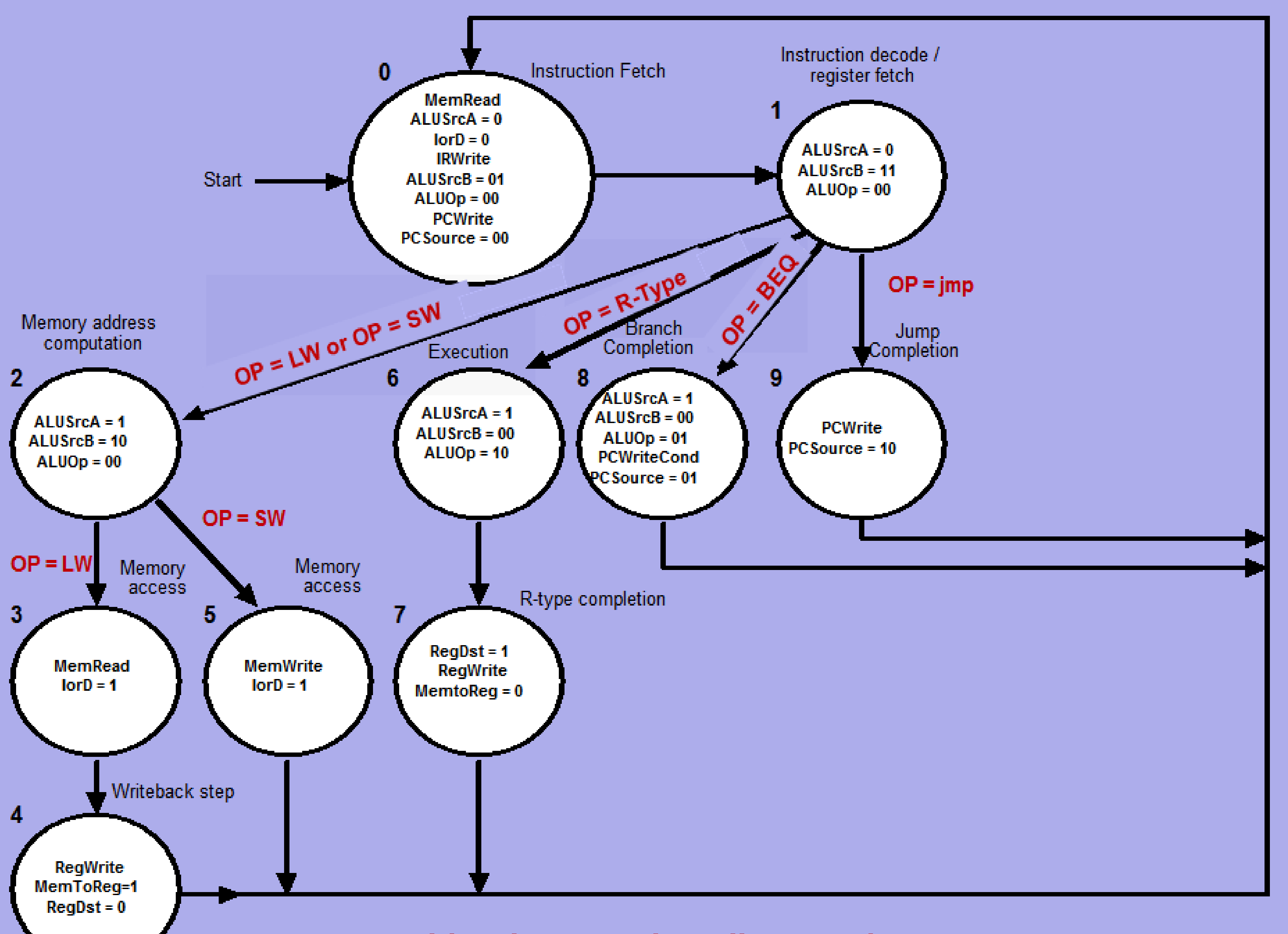


Multicycle Control Unit

- Review: single-cycle implementation
  - All control signals can be determined by current instruction + condition
  - Implemented in combinational logic
- Multicycle implementation is different
  - Control signals depend on
    - current instruction
    - which clock cycle is currently being executed
  - Implemented as
    - Finite State Machine (FSM) OR
    - Microprogrammed Implementation - “stylized FSM”

FSM Control Unit Implementation

- FSM Inputs:
  - Clock
  - Instruction register op field
- FSM Outputs: control signals for datapath
  - Enable signals: Register file, Memory, PC, MDR, and IR
  - Multiplexer signals: ALUSrcA, ALUSrcB, etc.
  - ALUOp signal - used for ALU Control as in single-cycle implementation



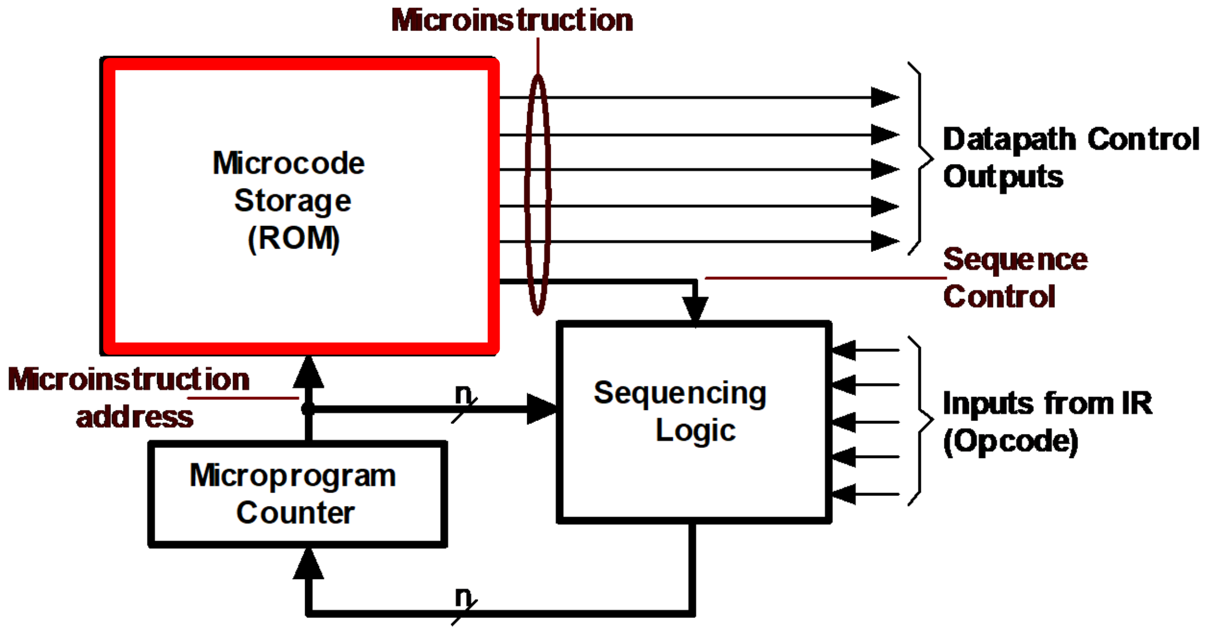
Multicycle Control - Full State Diagram

Microprogrammed Control

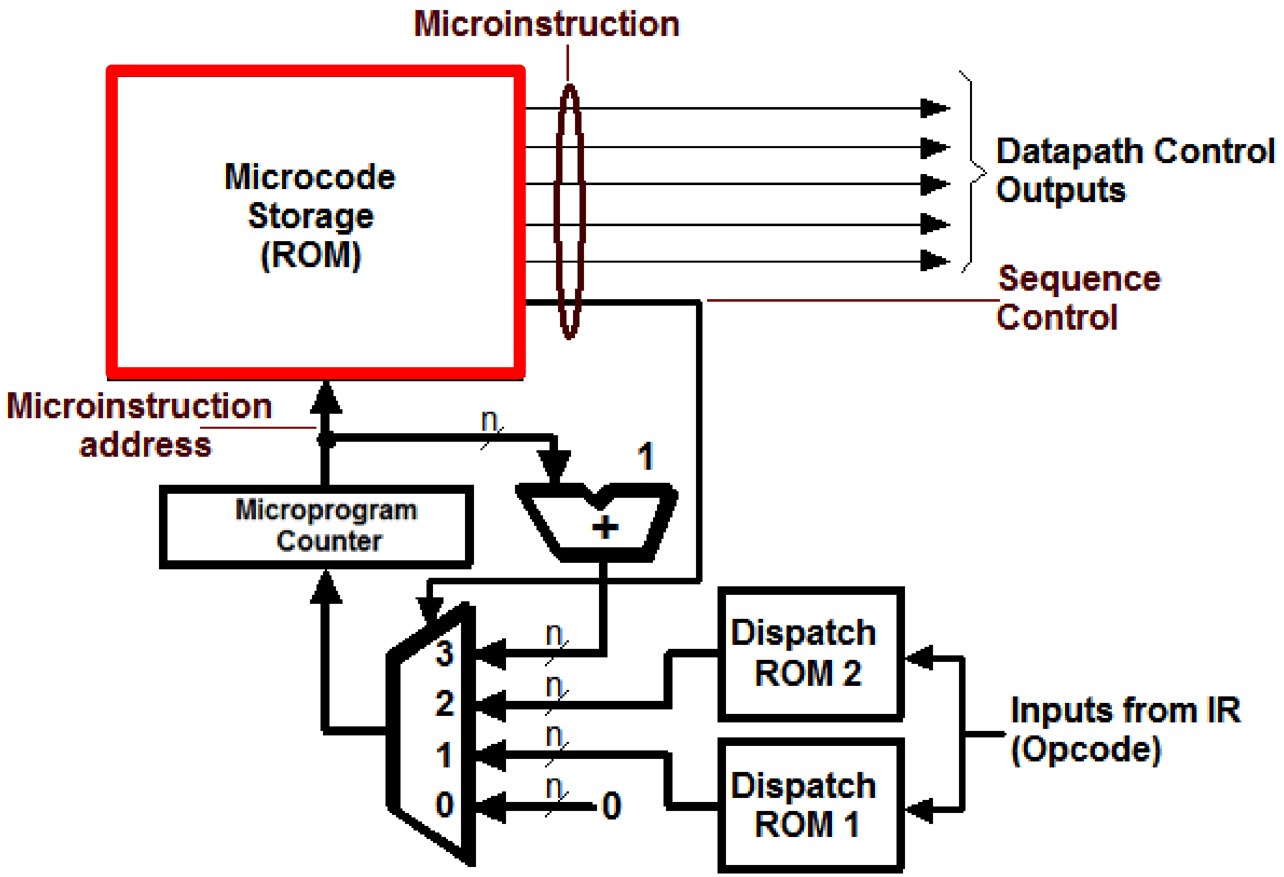
- A Control Unit FSM in a CISC processor can have thousands of states
- Microprogramming structures control signals for states by storing them in a ROM
- Very popular in 1980s-1990s microprocessors
- Still used as a “fallback” to implement complex instructions in x86 processors
- Also used for patches in some x86 processors

Microprogramming - Basic Idea

- Idea: expand on ROM control implementation
  - One state = one ROM word = one microinstruction
  - State sequences form a microprogram or microcode
  - Each state code becomes a microinstruction address



Microprogramming - Sequencer Design



Performance of a Multicycle Implementation

- What is the CPI of the Multicycle Implementation?
- Using measured instruction mix from SPECINT2000

lw	5 cycles	25%
sw	4 cycles	10%
R-type	4 cycles	53%
branch	3 cycles	11%
jump	3 cycles	2%

- What is the CPI?
  - CPI = (5 cycles * 0.25) + (4 cycles * 0.10) + (4 cycles * 0.53) + (3 cycles * 0.11) + (3 cycles * 0.02)
  - CPI = 4.12 cycles per instruction

Performance Continued

- **Assuming a 200ps clock, what is average execution time/instruction?**
  - **Sec/Instr = 4.12 CPI * 200ps/cycle = 824ps/instr**

- **How does this compare to the Single-Cycle Case?**
  - **Sec/Instr = 1 CPI * 600ps/cycle = 600ps/instr**
  - **Single-Cycle is 1.38 times faster than Multicycle**

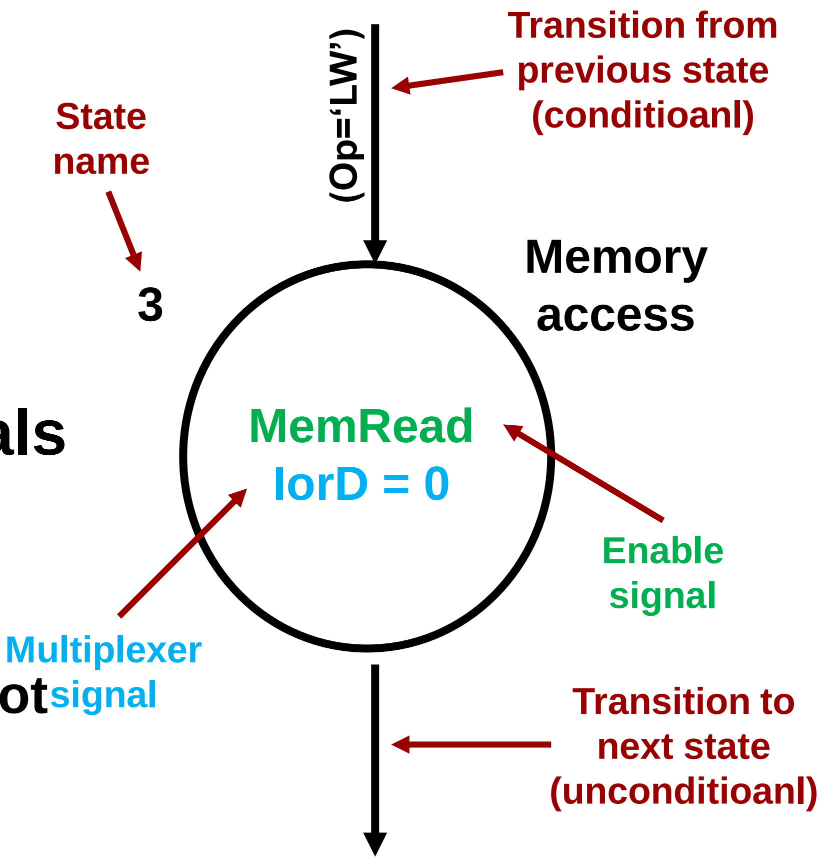
- **Why is Single-Cycle faster than Multicycle?**
  - **Branch & jump are the same speed (600ps vs 600ps)**
  - **R-type & store are faster (600ps vs 800ps)**
  - **Load word is faster (600ps vs 1000ps)**

Multicycle Design - Summary

- Advantages
  - Uses less hardware - modules can be shared!
  - Different instructions can take different times
  - Can implement more complex instructions
  - Microcode-based control gives flexibility
- Disadvantages
  - Not necessarily faster than single-cycle
- Bottom line (결론/요점)
  - Used in older processors due to resource constraints
  - Not widely used in modern processor design
  - Microcoded control still used as a “fallback” for complex x86 instructions

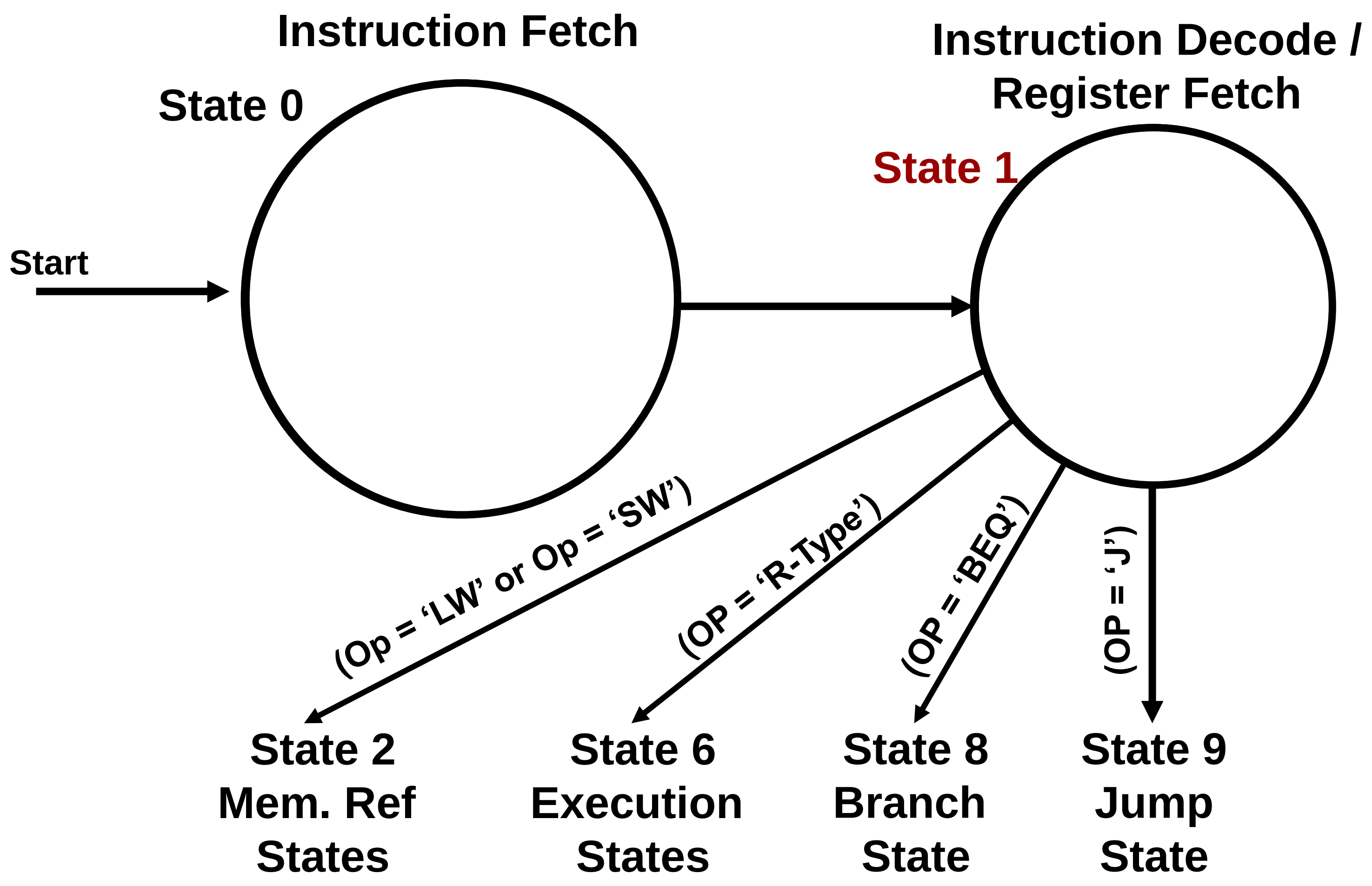
Appendix - Control Unit Details

- **Enable Control Signals**
  - **Asserted (true) in a state if they appear in the state bubble**
  - **Assumed Deasserted (false) in a state if they do not appear in the state bubble**
- **Multiplexer/ALU Control Signals**
  - **Asserted with a given value in a state if they appear in a state bubble**
  - **Assumed Don't Care if they do not appear in the state bubble**
- **See Figures 5.32 - 5.36**

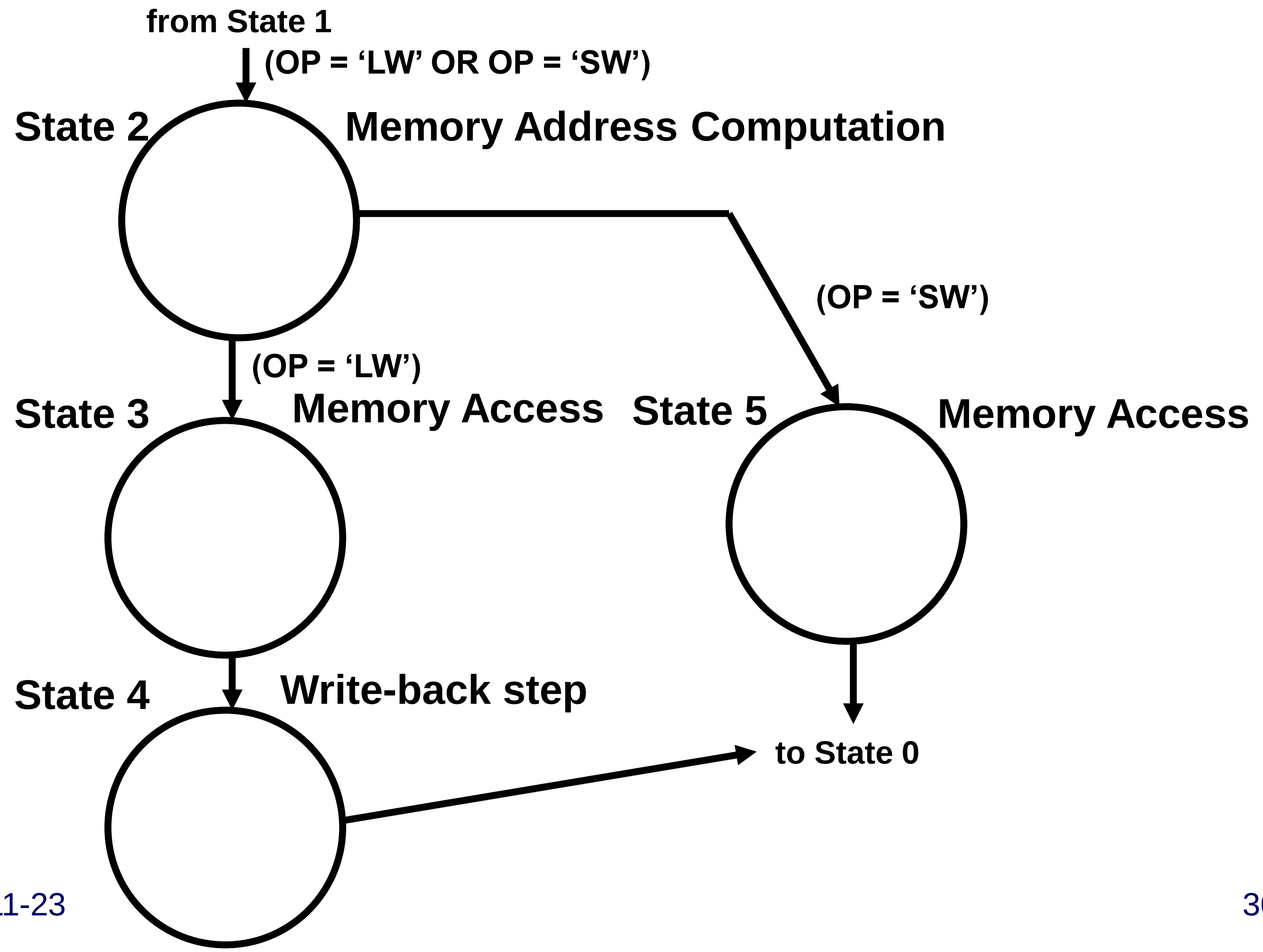


Transition from previous state (conditioanl)
(Op='LW')
State name
3
Memory access
MemRead
IorD = 0
Enable signal
Multiplexer signal
Transition to next state (unconditioanl)

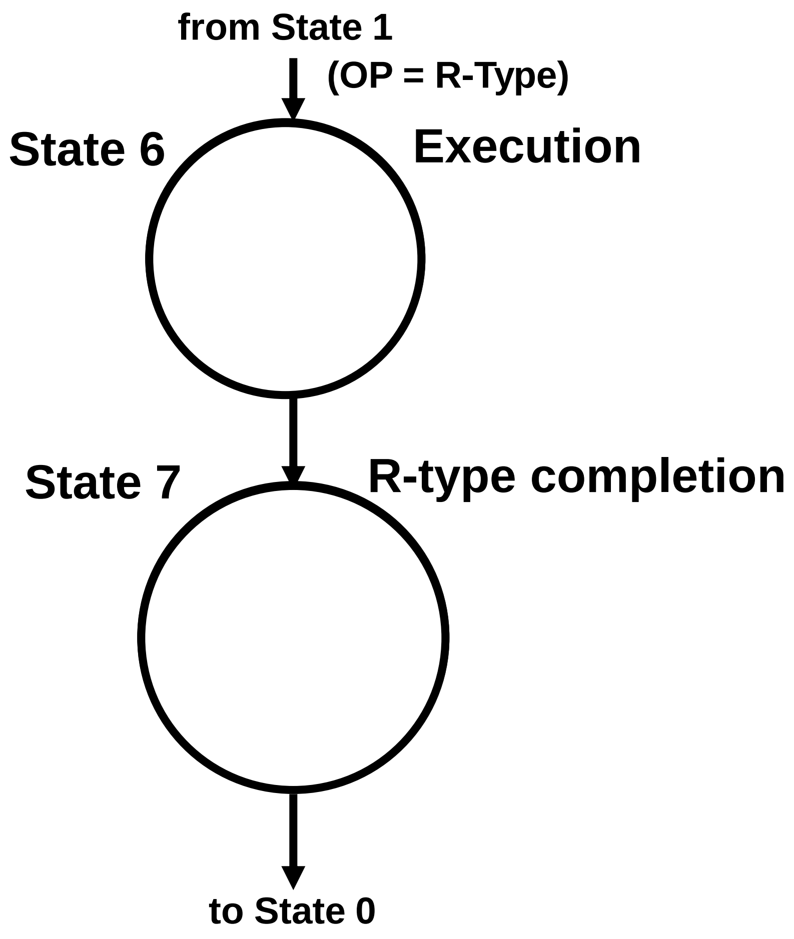
Control FSM - Instruction Fetch / Decode States



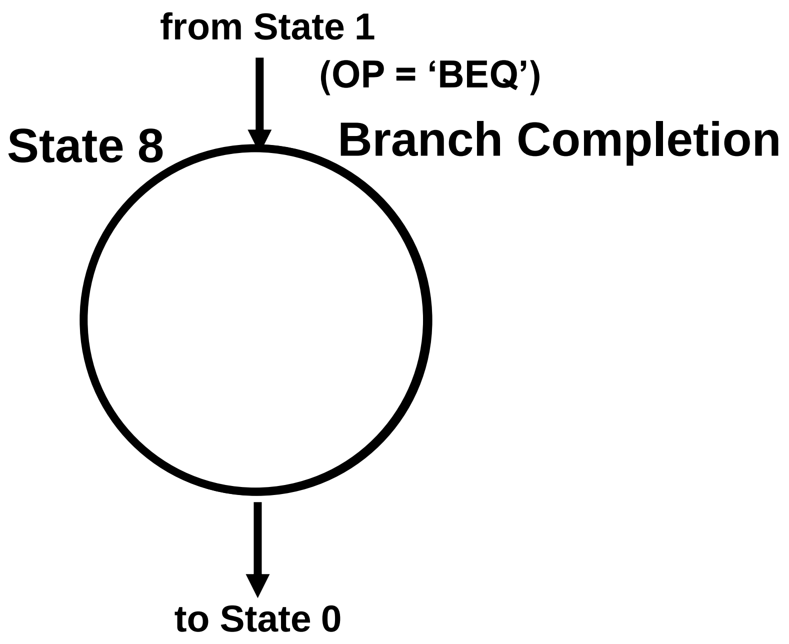
Control FSM - Memory Reference States



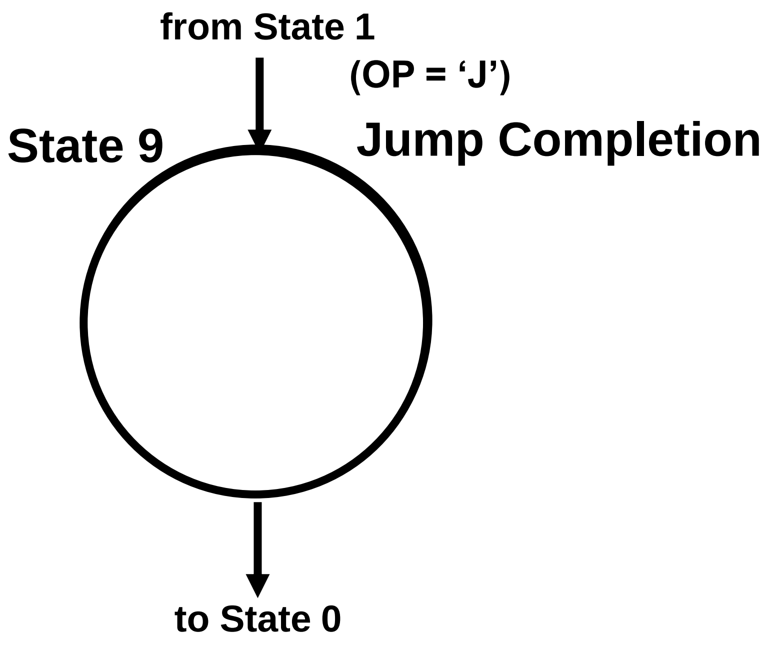
Control FSM - R-Type States



Control FSM - Branch State

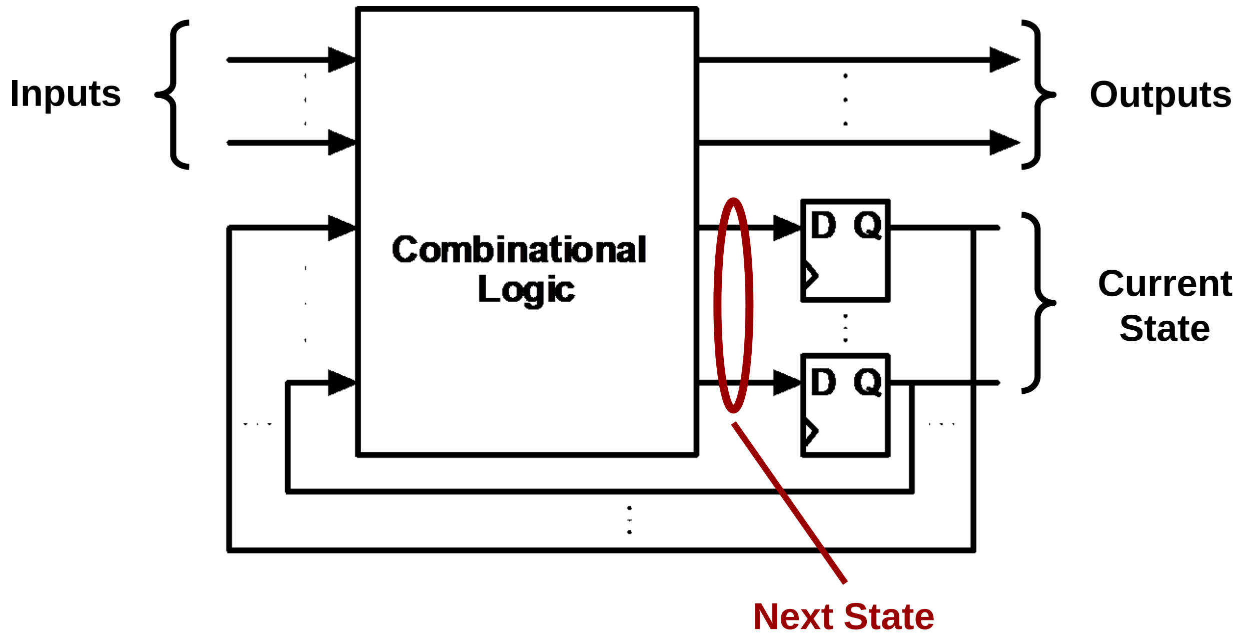


Control FSM - Jump State



Appendix - FSM Review

- What's in the Finite State Machine
  - Combinational Logic
  - Storage Elements (Flip-flops)

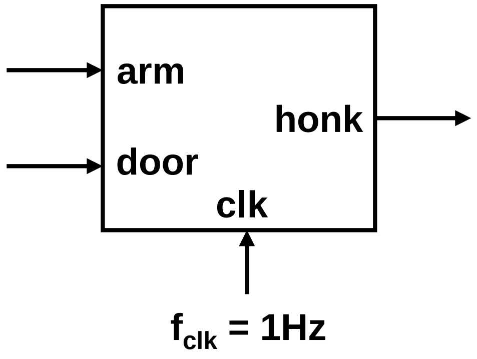


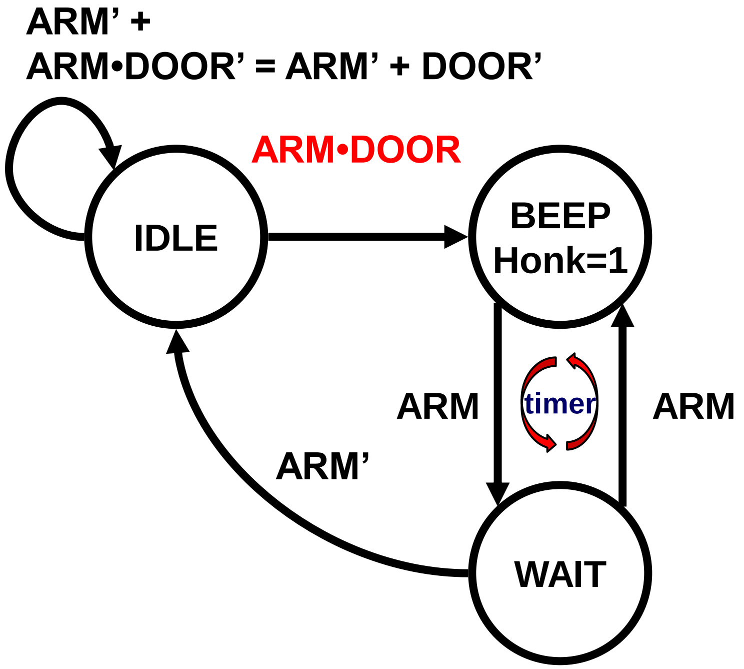
Review - Finite State Machines (cont'd)

- Behavior characterized by
  - States - unique values of flip-flops e.g., "101"
  - Transitions between states, depending on
    - current state
    - inputs
  - Output values, depending on
    - current state
    - inputs
- Describing FSMs:
  - State Diagrams
  - State Transition Tables

Review - State Diagrams

- "Bubbles" - states
- Arrows - transition edges labeled with condition expressions
- Example: Car Alarm



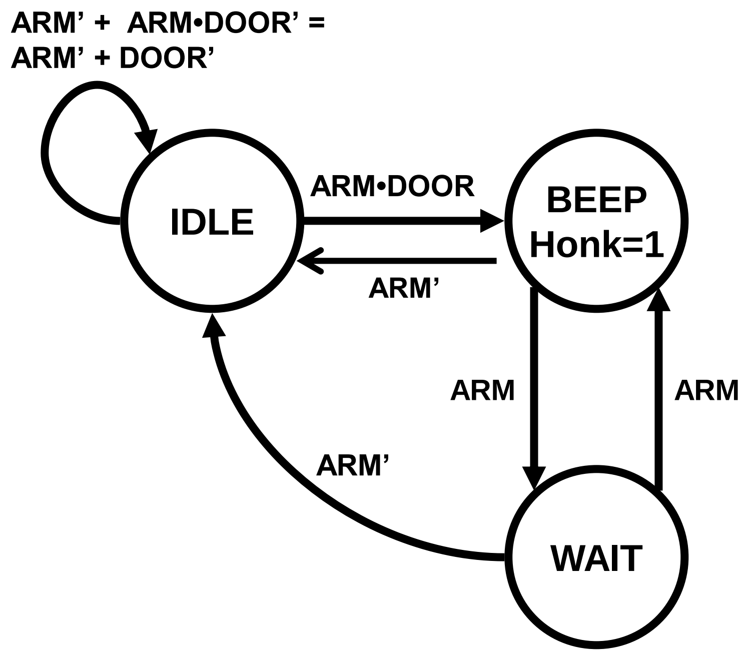


Honk if door is open and arm is touched

State Transition Table

- Transition List - lists edges in State Diagram

PS	Condition	NS	Output
IDLE	ARM' + DOOR'	IDLE	0
IDLE	ARM*DOOR	BEEP	0
BEEP	ARM	WAIT	1
BEEP	ARM'	IDLE	1
WAIT	ARM	BEEP	0
WAIT	ARM'	IDLE	0



State Machine Design

- Traditional Approach:
  - Create State Diagram
  - Create State Transition Table
  - Assign State Codes
  - Write Excitation Equations & Minimize
- Modern Approach
  - Enter FSM Description into CAD program
  - Synthesize implementation of FSM