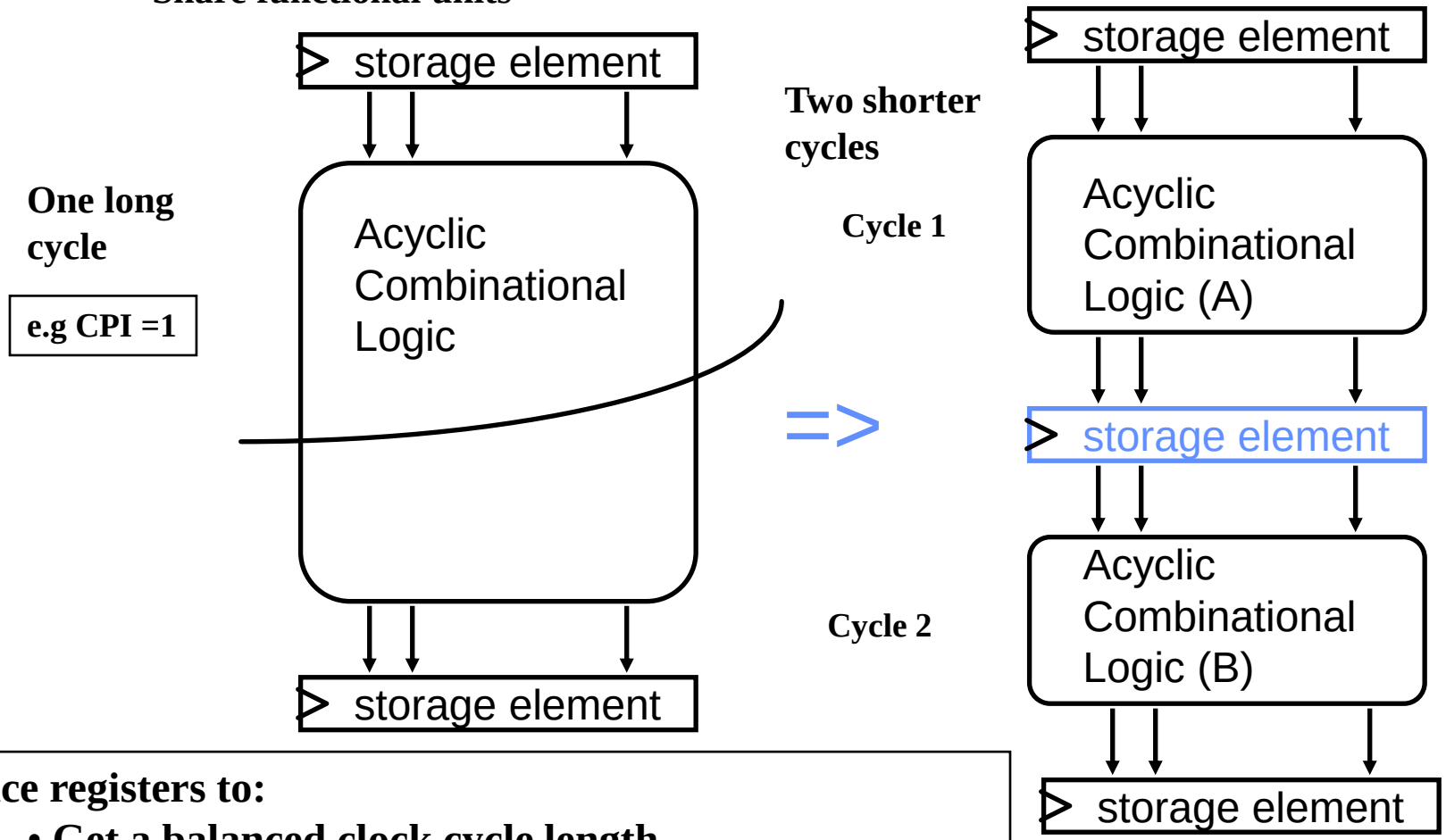
MIPS Multi-Cycle CPU

Datapath & Control

Reducing Cycle Time: Multi-Cycle Design

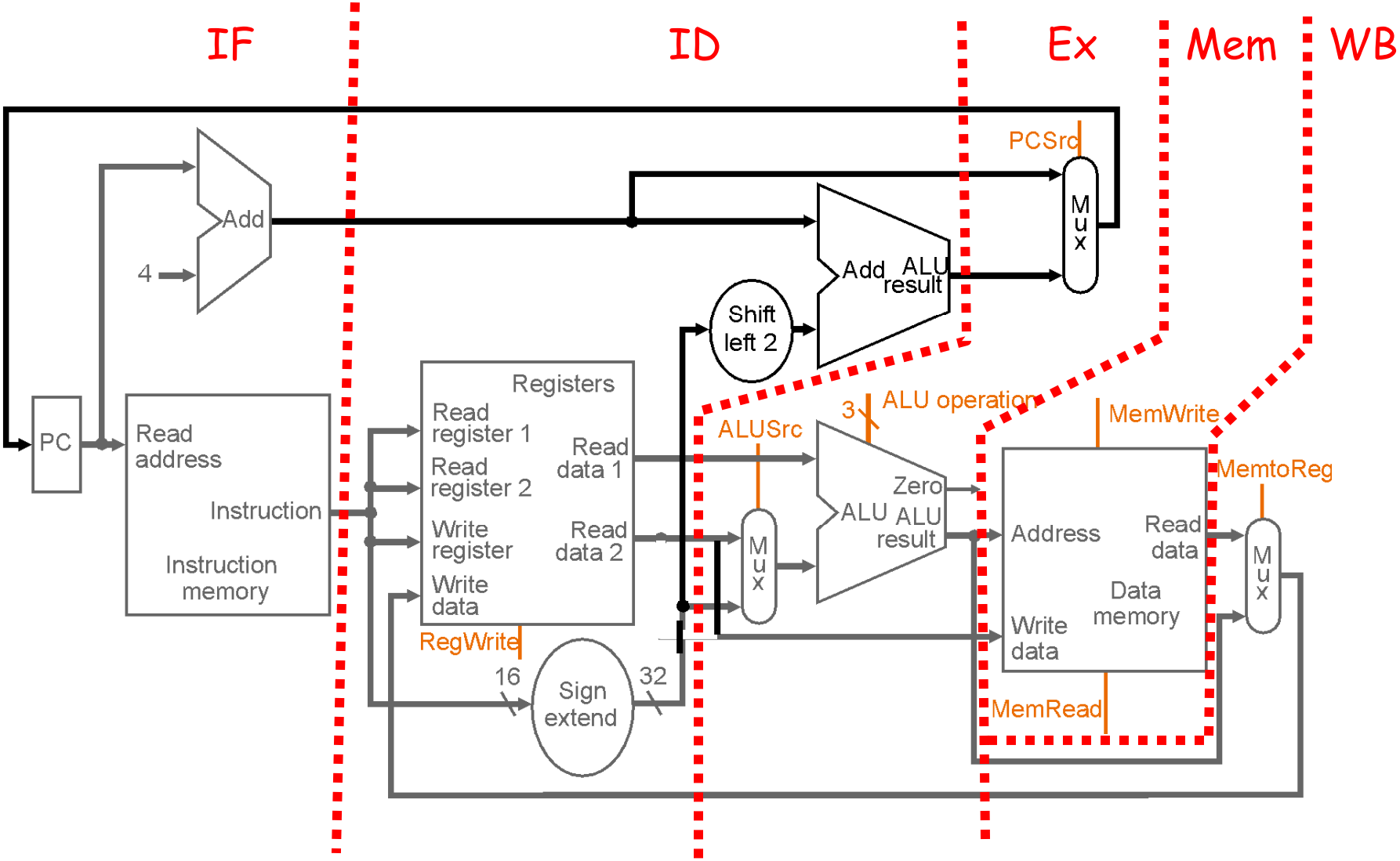
- **Cut combinational dependency graph by <u>inserting registers / latches.</u>**
- **The same work is done in two or more shorter cycles, rather than one long cycle.**
  - **Different CPI**
  - **Share functional units**



Place registers to:

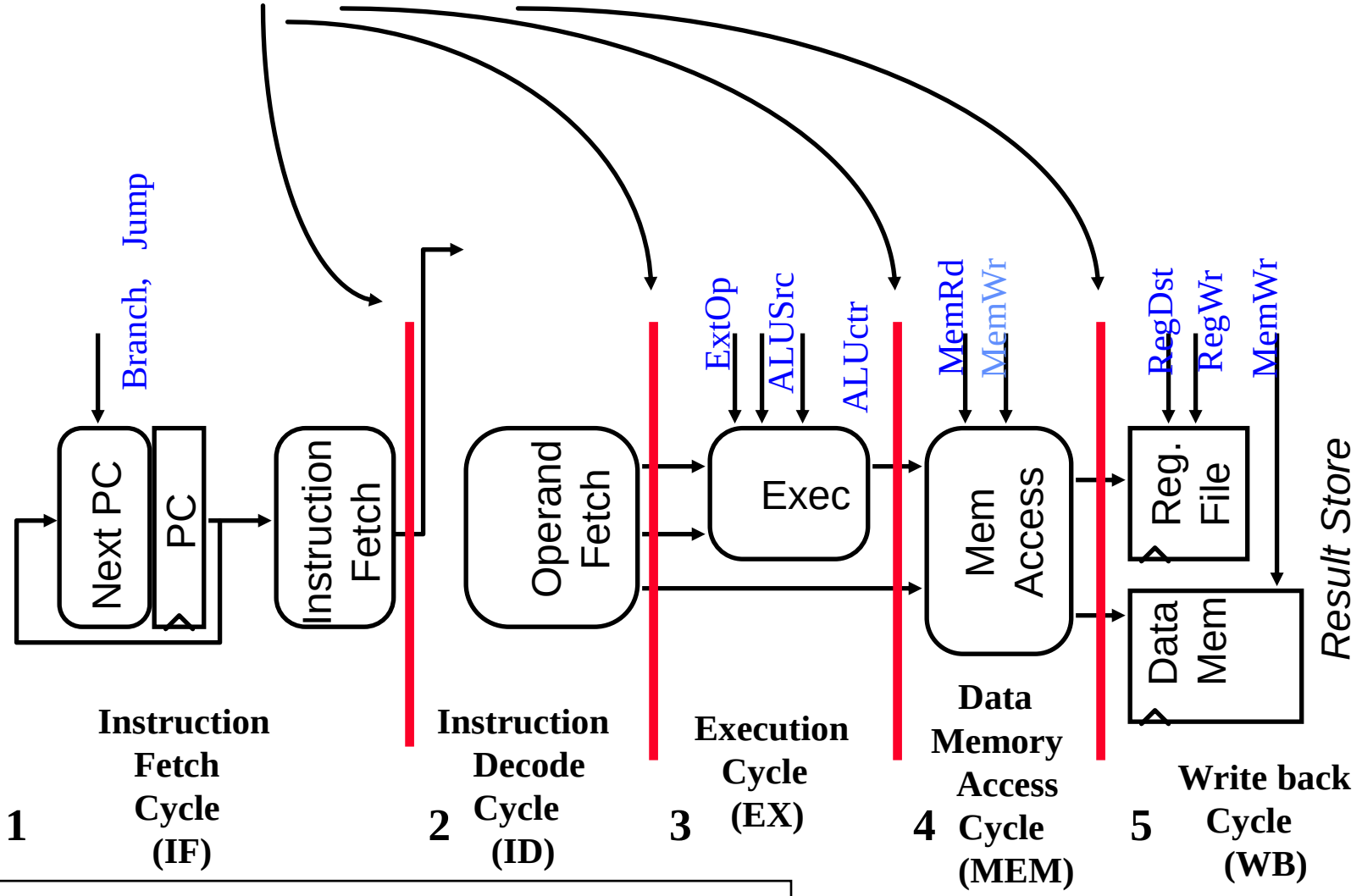
- **Get a balanced clock cycle length**
- **Save any results needed for the remaining cycles**

Partitioning the Single-Cycle Design



Partitioning The Single Cycle Datapath

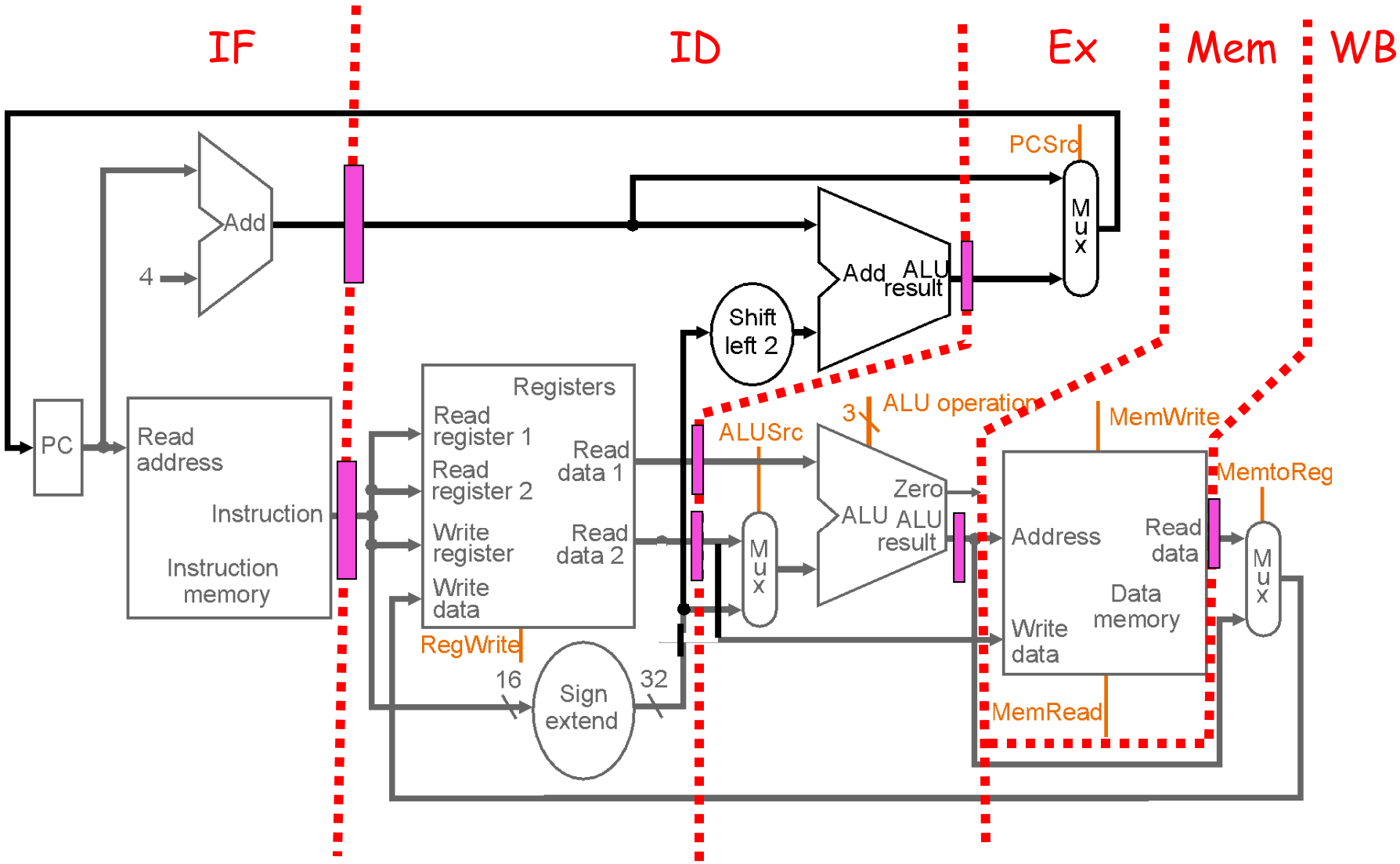
Add registers between steps to break into cycles



Place registers to:

- **Get a balanced clock cycle length**
- **Save any results needed for the remaining cycles**

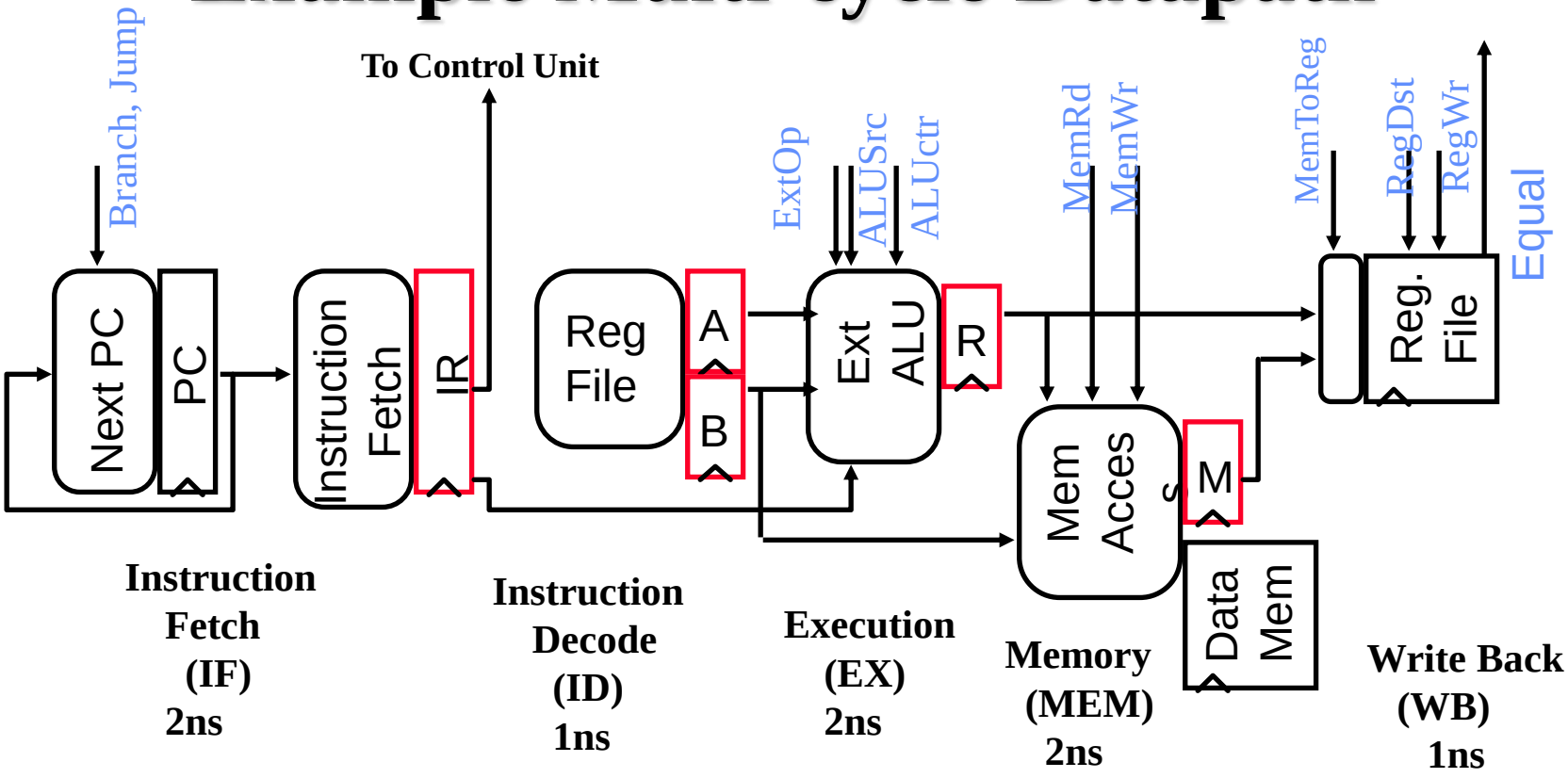
Where to add registers



Place registers to:

- **Get a balanced clock cycle length**
- **Save any results needed for the remaining cycles**

Example Multi-cycle Datapath



Registers added: All clock-edge triggered (not shown register write enable control lines)

IR: Instruction register

A, B: Two registers to hold operands read from register file.

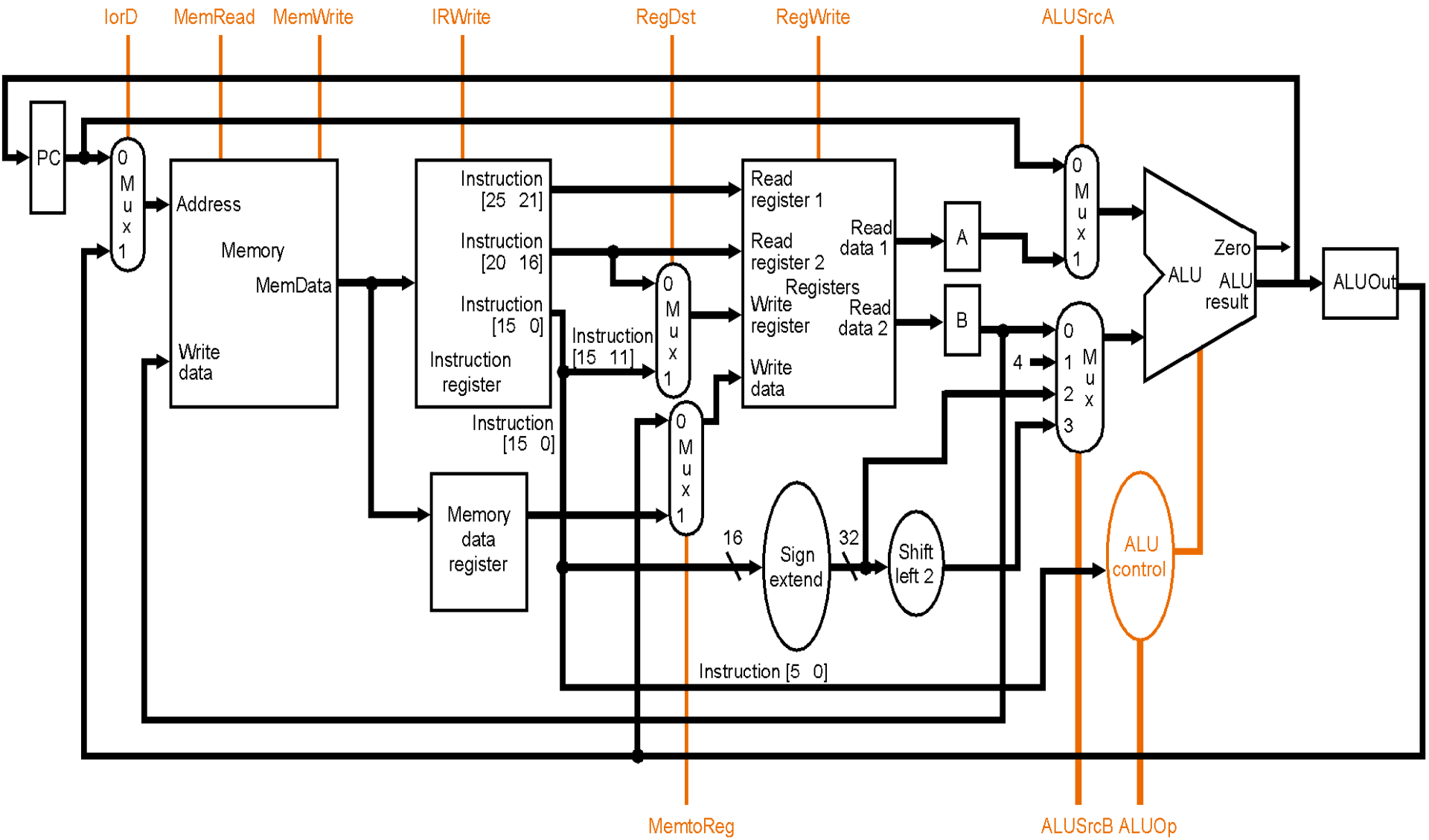
R: or ALUOut, holds the output of the main ALU

M: or Memory data register (MDR) to hold data read from data memory

CPU Clock Cycle Time: Worst cycle delay = C = 2ns (ignoring MUX, CLK-Q delays)

Assuming the following datapath/control hardware components delays:
Memory Units: 2 ns ALU and adders: 2 ns
Register File: 1 ns Control Unit < 1 ns

MIPS Multi-cycle Datapath



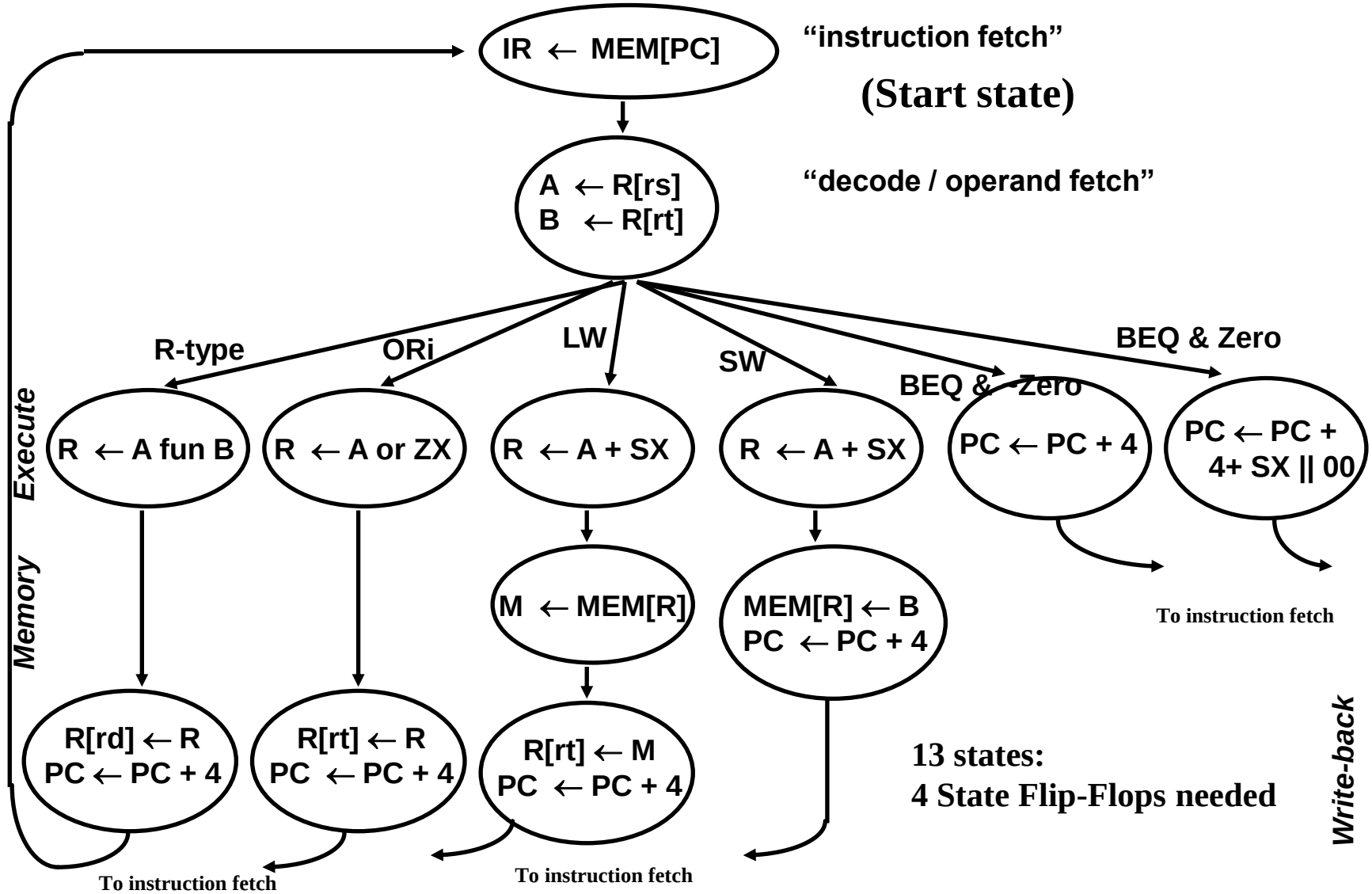
Operations (Dependant RTN) for Each Cycle

		R-Type	Logic Immediate	Load	Store	Branch
IF	Instruction Fetch	IR ← Mem[PC]	IR ← Mem[PC]	IR ← Mem[PC]	IR ← Mem[PC]	IR ← Mem[PC]
ID	Instruction Decode	A ← R[rs] B ← R[rt]	A ← R[rs] B ← R[rt]	A ← R[rs] B ← R[rt]	A ← R[rs] B ← R[rt]	A ← R[rs] B ← R[rt]
EX	Execution	R ← A funct B	R ← A OR ZeroExt[imm16]	R ← A + SignEx(Im16)	R ← A + SignEx(Im16)	Zero ← A - B If Zero = 1: PC ← PC + 4 + (SignExt(imm16) x4) else (i.e Zero =0): PC ← PC + 4
MEM	Memory			M ← Mem[R]	Mem[R] ← B PC ← PC + 4	
WB	Write Back	R[rd] ← R PC ← PC + 4	R[rt] ← R PC ← PC + 4	R[rt] ← M PC ← PC + 4		

Instruction Fetch (IF) & Instruction Decode cycles are common for all instructions

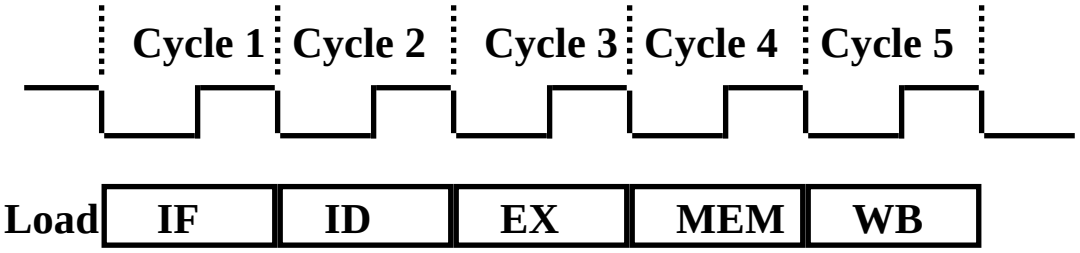
Control Specification For Multi-cycle CPU

Finite State Machine (FSM) - State Transition Diagram



MIPS Multi-Cycle Datapath:

Five Cycles of Load



1- **Instruction Fetch (IF):**

Fetch the instruction from instruction Memory.

2- **Instruction Decode (ID):**

Operand Register Fetch and Instruction Decode.

3- **Execute (EX):** Calculate the effective memory address.

4- **Memory (MEM):** Read the data from the Data Memory.

5- **Write Back (WB):**

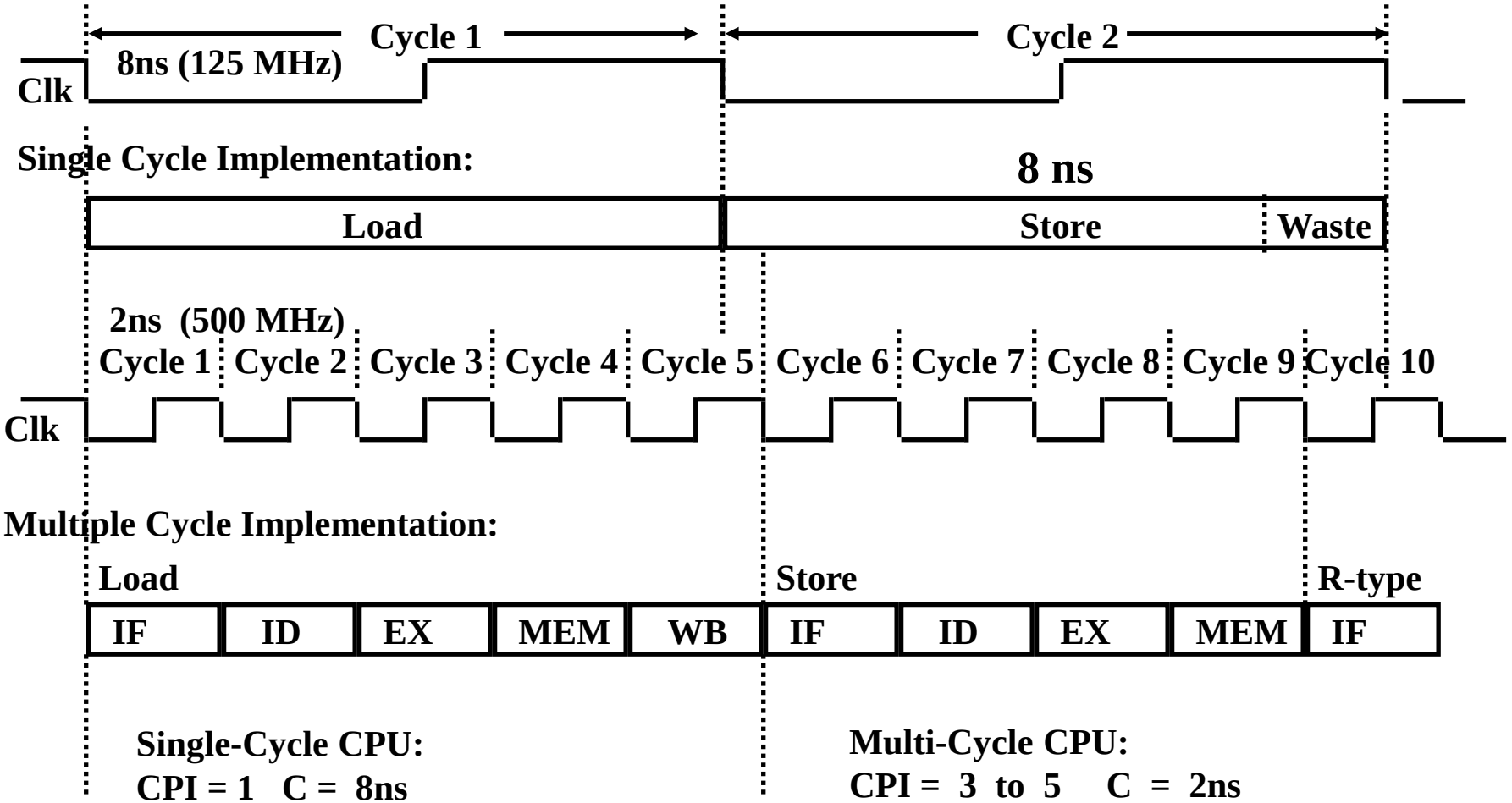
Write the loaded data to the register file. Update PC.

Multi-cycle Datapath Instruction CPI

- **R-Type/Immediate: Require four cycles, CPI = 4**
  - **IF, ID, EX, WB**
- **Loads: Require five cycles, CPI = 5**
  - **IF, ID, EX, MEM, WB**
- **Stores: Require four cycles, CPI = 4**
  - **IF, ID, EX, MEM**
- **Branches/Jumps: Require three cycles, CPI = 3**
  - **IF, ID, EX**

- **Average or effective program CPI: $3 \leq \text{CPI} \leq 5$ depending on program profile (instruction mix).**

Single Cycle Vs. Multi-Cycle CPU



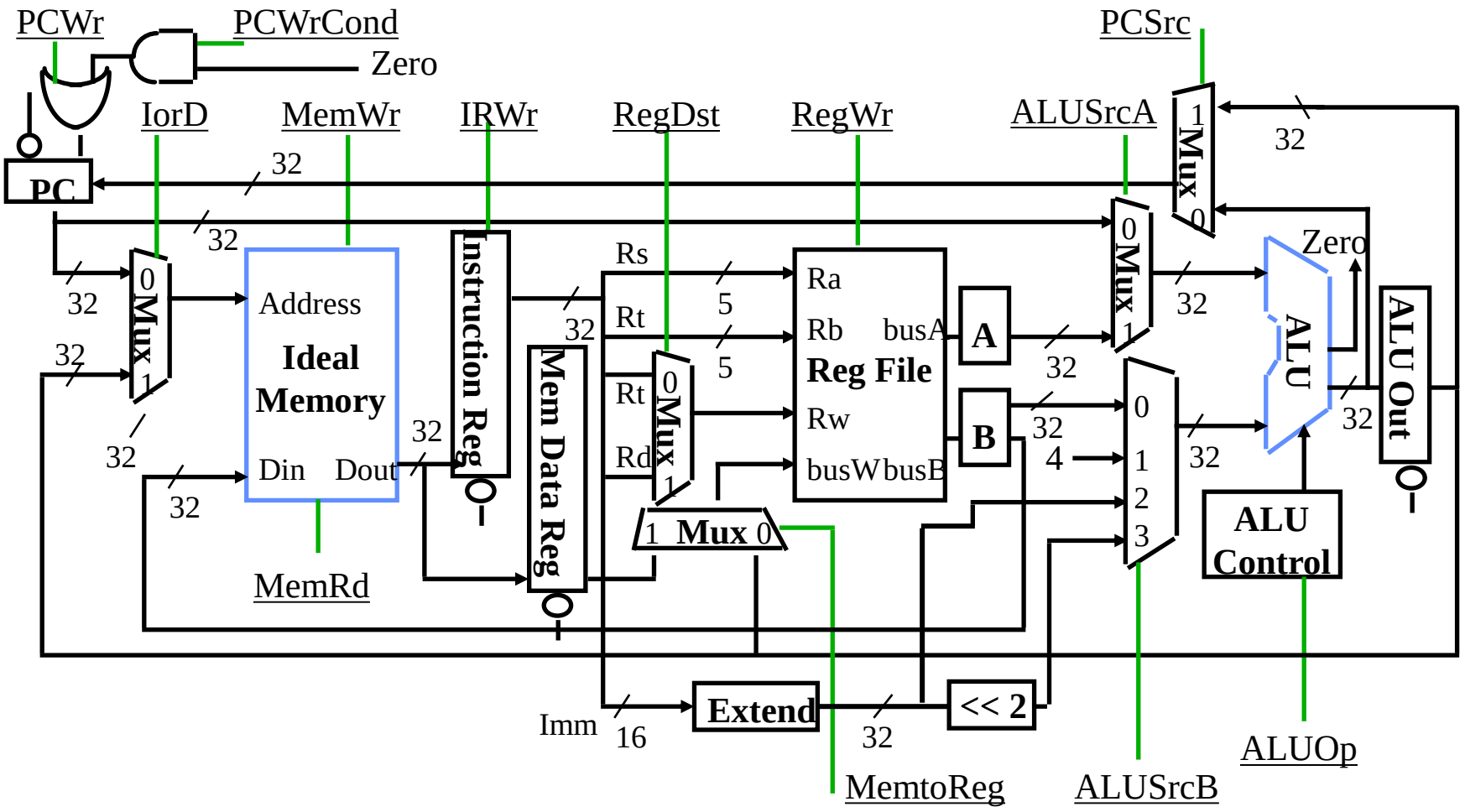
Single-Cycle CPU:
CPI = 1 C = 8ns
One million instructions take =
I x CPI x C = 10^6 x 1 x 8x10^{-9} = 8 msec

Multi-Cycle CPU:
CPI = 3 to 5 C = 2ns
One million instructions take from
10^6 x 3 x 2x10^{-9} = 6 msec
to 10^6 x 5 x 2x10^{-9} = 10 msec
depending on instruction mix used.

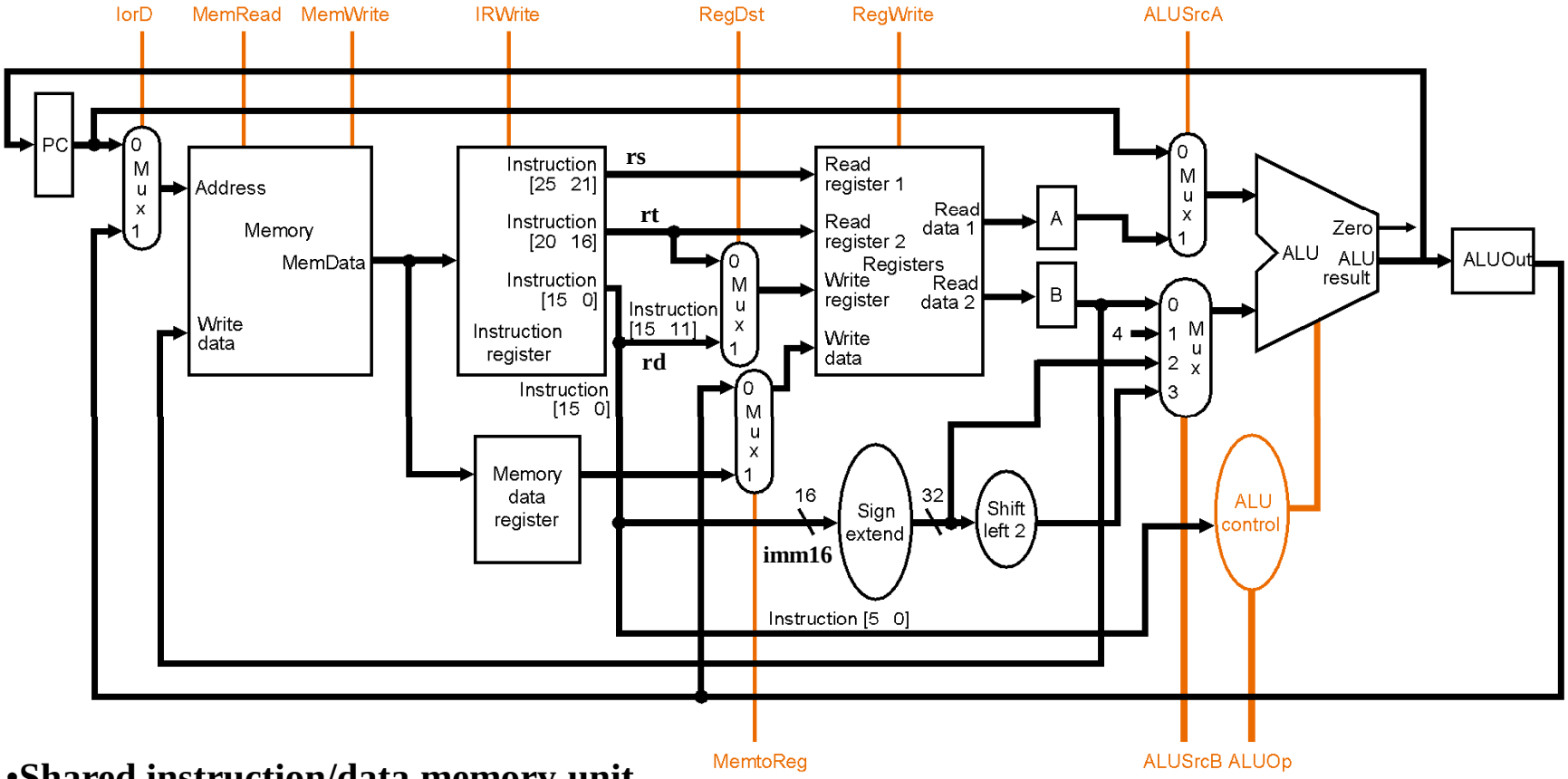
Assuming the following datapath/control hardware components delays:
Memory Units: 2 ns ALU and adders: 2 ns
Register File: 1 ns Control Unit < 1 ns

Alternative Multiple Cycle Datapath (In Textbook)

- **Miminizes Hardware: 1 memory, 1 ALU**



Alternative Multiple Cycle Datapath (In Textbook)

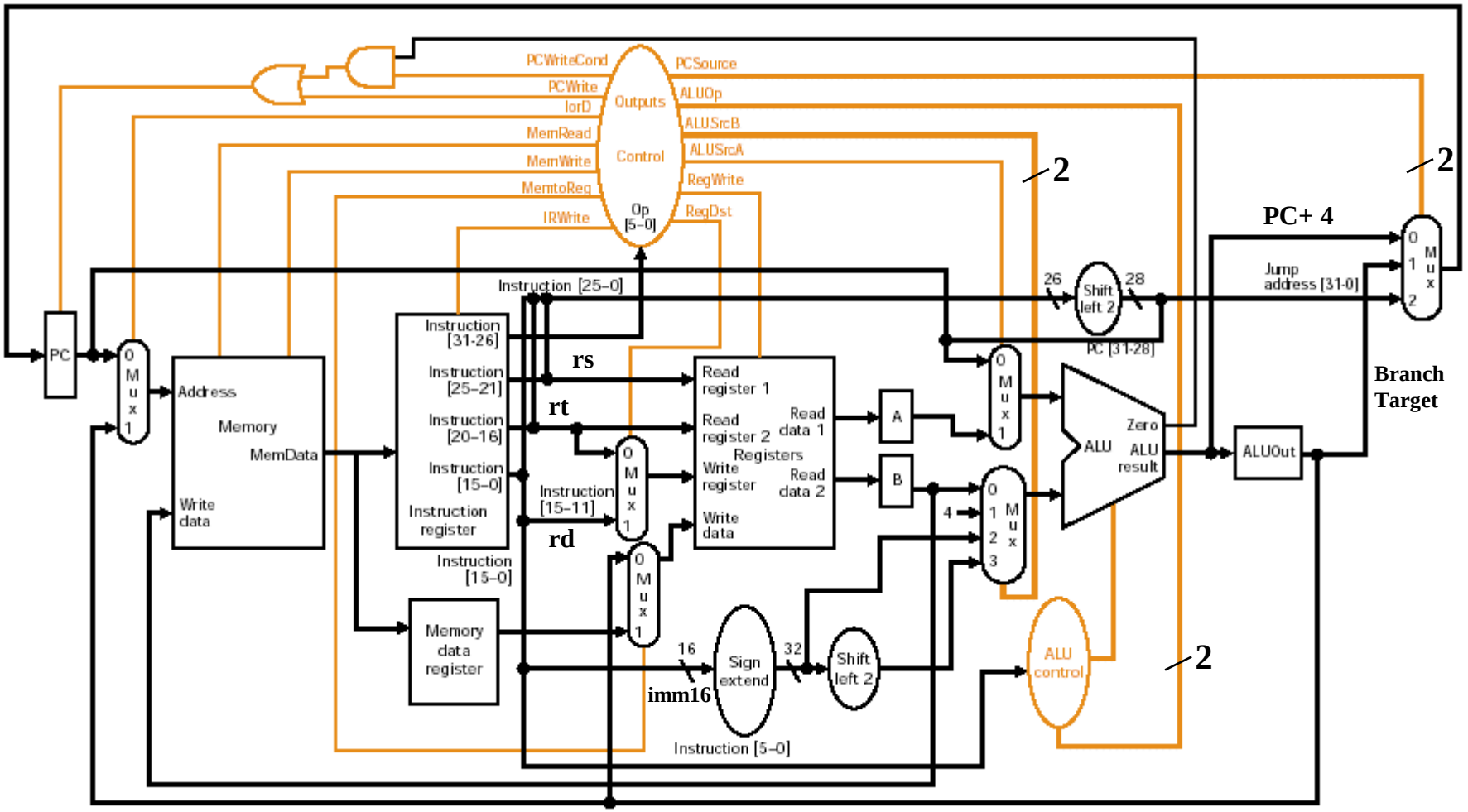


- **Shared instruction/data memory unit**
- **A single ALU shared among instructions**
- **Shared units require additional or widened multiplexors**
- **Temporary registers to hold data between clock cycles of the instruction:**
  - **Additional registers:**

    Instruction Register (IR), Memory Data Register (MDR), A, B, ALUOut

(Figure 5.27 page 322)

Alternative Multiple Cycle Datapath With Control Lines (Fig 5.28 In Textbook)



(ORI not supported, Jump supported)

(Figure 5.28 page 323)

The Effect of The 1-bit Control Signals

Signal Name	Effect when deasserted (=0)	Effect when asserted (=1)
RegDst	The register destination number for the write register comes from the rt field (instruction bits 20:16).	The register destination number for the write register comes from the rd field (instruction bits 15:11).
RegWrite	None	The register on the write register input is written with the value on the Write data input.
ALUSrcA	The first ALU operand is the PC	The First ALU operand is register A (I.e R[rs])
MemRead	None	Content of memory specified by the address input are put on the memory data output.
MemWrite	None	Memory contents specified by the address input is replaced by the value on the Write data input.
MemtoReg	The value fed to the register write data input comes from ALUOut register.	The value fed to the register write data input comes from data memory register (MDR).
IorD	The PC is used to supply the address to the memory unit.	The ALUOut register is used to supply the the address to the memory unit.
IRWrite	None	The output of the memory is written into Instruction Register (IR)
PCWrite	None	The PC is written; the source is controlled by PCSource
PCWriteCond	None	The PC is written if the Zero output of the ALU is also active.

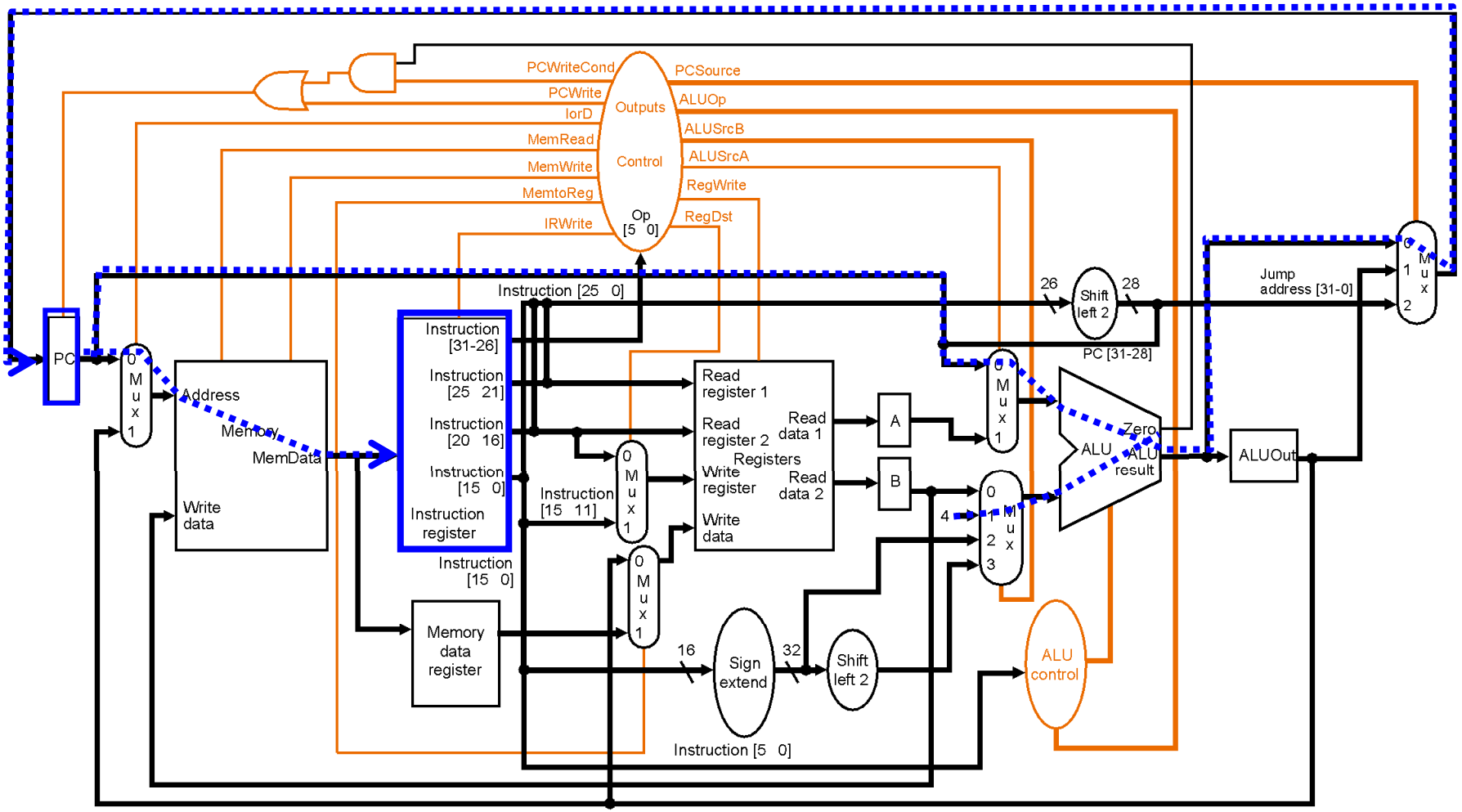
(Figure 5.29 page 324)

The Effect of The 2-bit Control Signals

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 (R-Type)
ALUSrcB	00	The second input of the ALU comes from register B
	01	The second input of the ALU is the constant 4
	10	The second input of the ALU is the sign-extended 16-bit immediate field of the instruction in IR
	11	The second input of the ALU is is the sign-extended 16-bit immediate field of IR shifted left 2 bits
PCSource	00	Output of the ALU (PC+4) is sent to the PC for writing
	01	The content of ALUOut (the branch target address) is 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

(Figure 5.29 page 324)

Cycle 1: Instruction Fetch



Datapath: IR = Memory[PC], PC = PC + 4 (may be revised later)
Control: IorD=0, MemRead=1, MemWr=0, IRwrite=1, ALUsrcA=0, etc

Cycle 2: Instruction Decode (and register fetch)

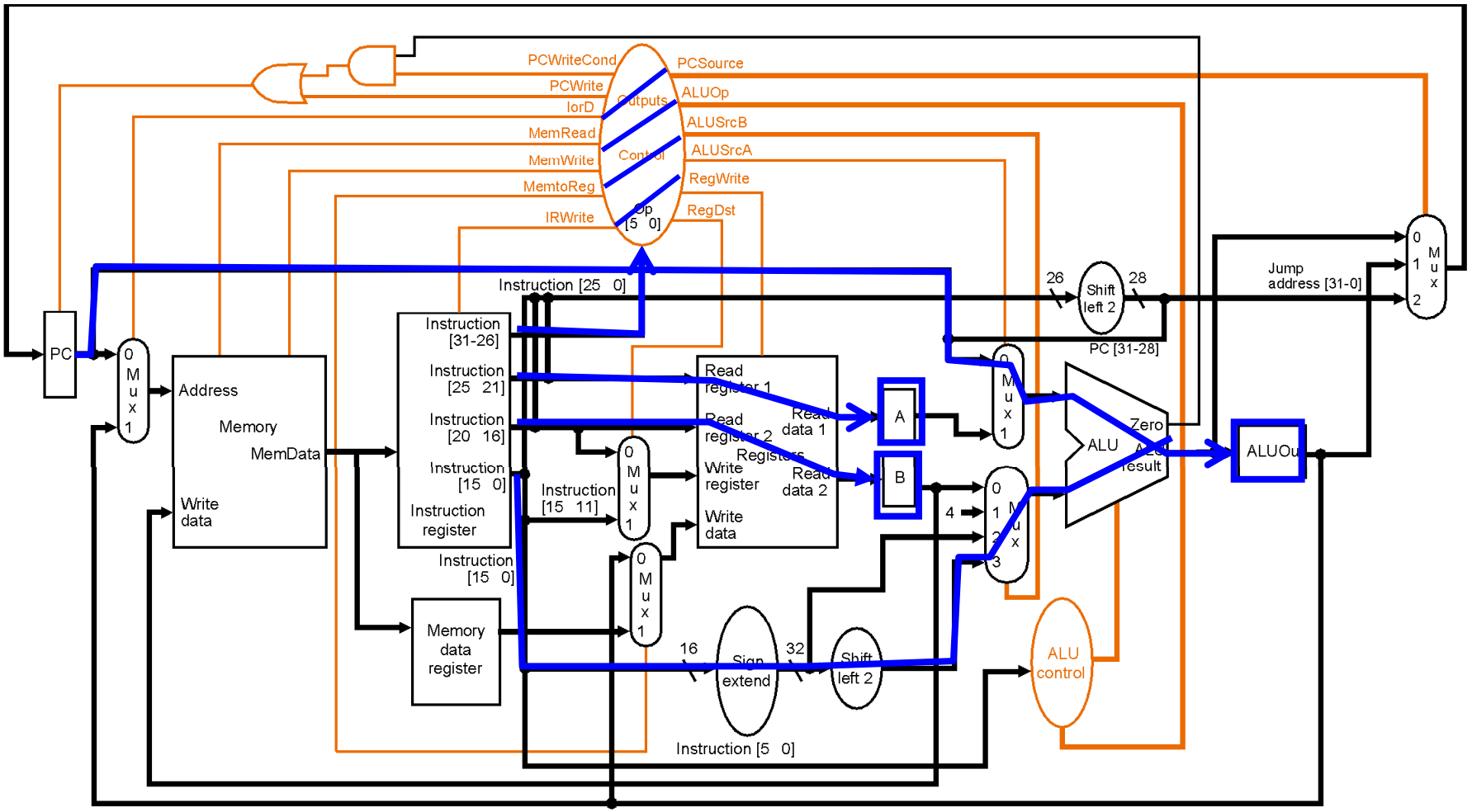
A = Reg[IR[25-21]]

B = Reg[IR[20-16]]

ALUout = PC + (sign-extend (IR[15-0]) << 2)

- **We compute target address even though we don't know if it will be used**
  - **Operation may not be branch**
  - **Even if it is, branch may not be taken**
- **Why?**
  - **Everything up to this point must be instruction-independent, because we haven't decoded the instruction.**
  - **The ALU, the (incremented) PC, and the immed field are now all available**

Cycle 2: Instruction Decode cycle

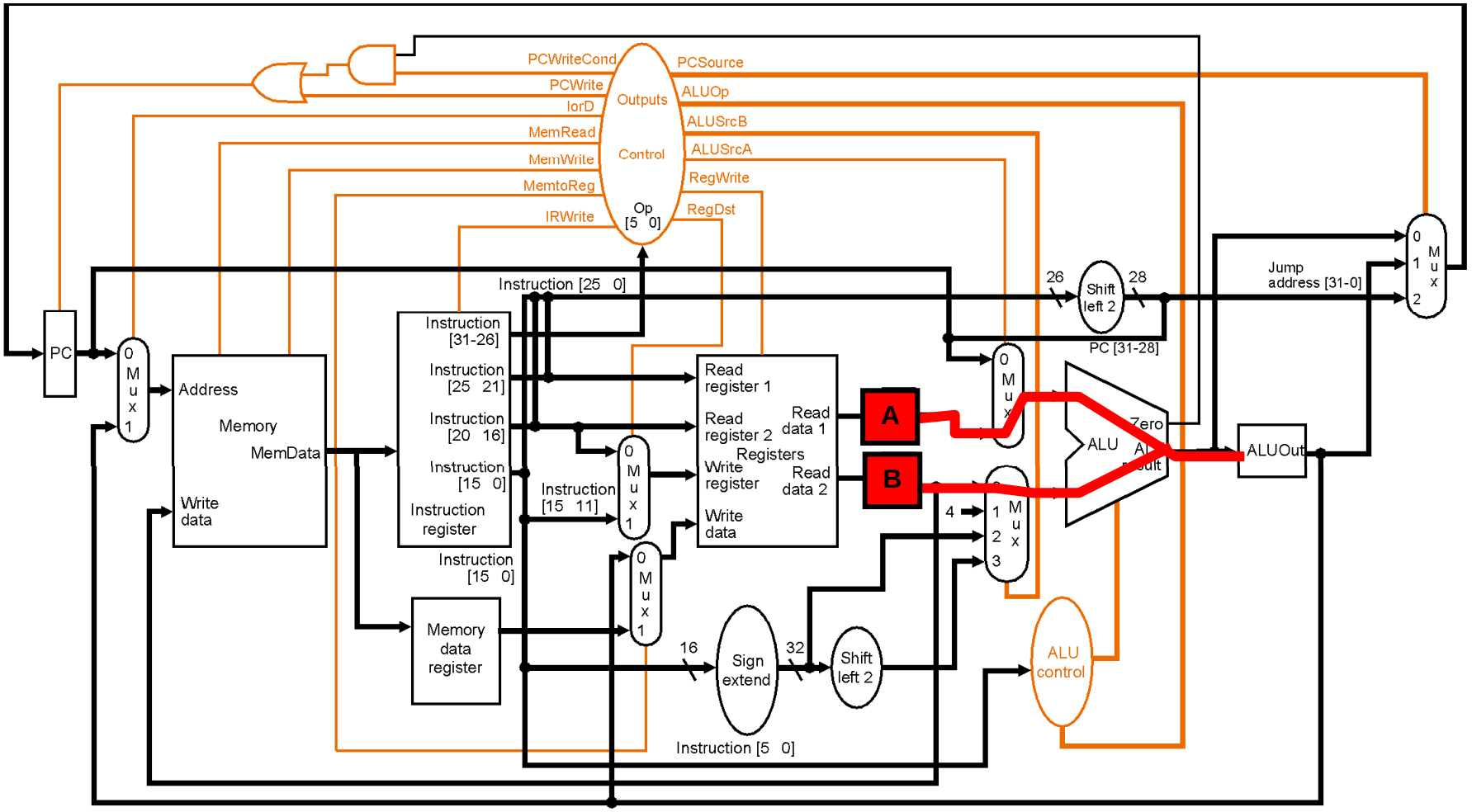


A = Register[IR[25-21]]

B = Register[IR[20-16]]

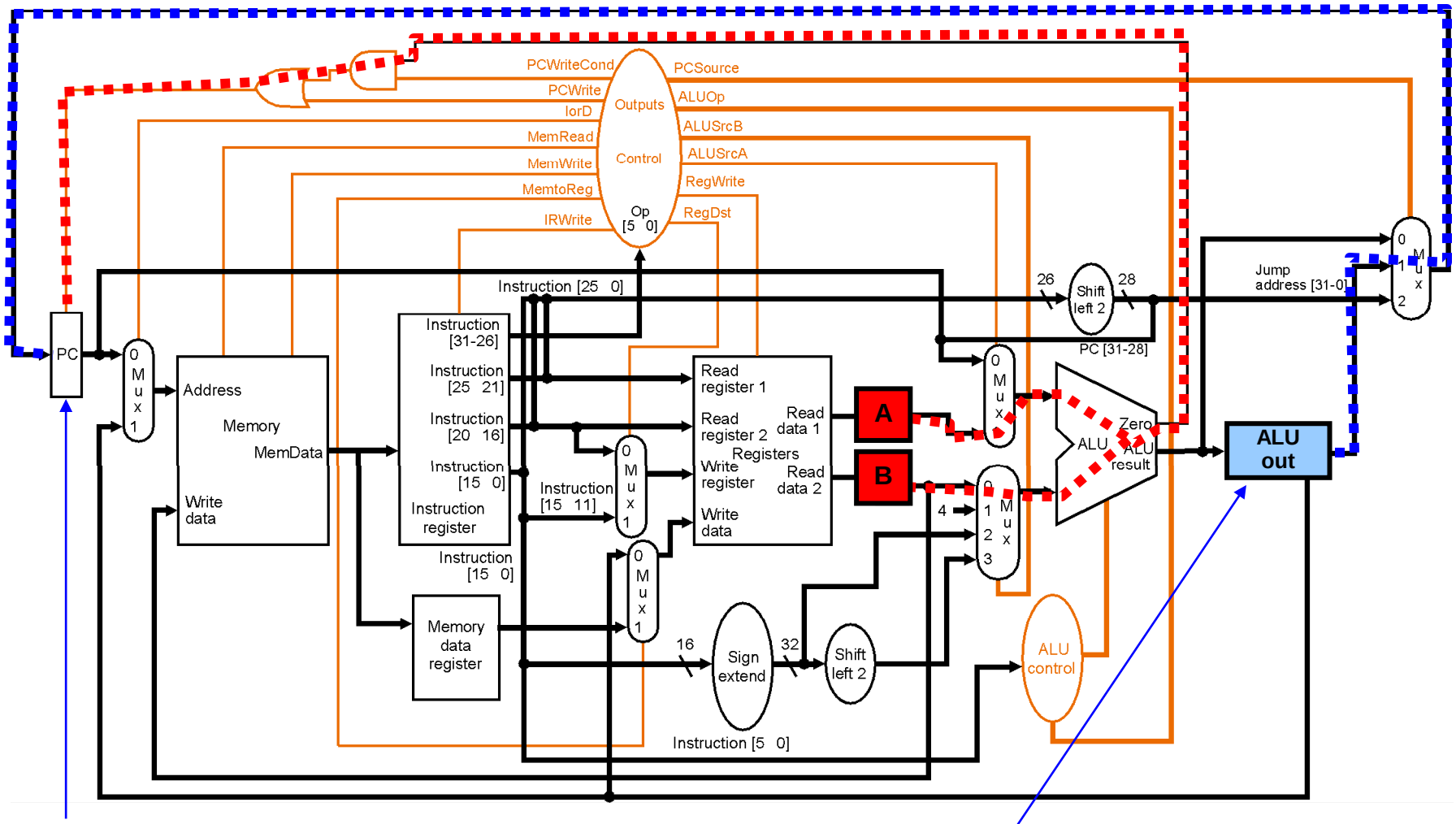
ALUout = PC + (sign-extend (IR[15-0]) << 2)

R-type EXecution



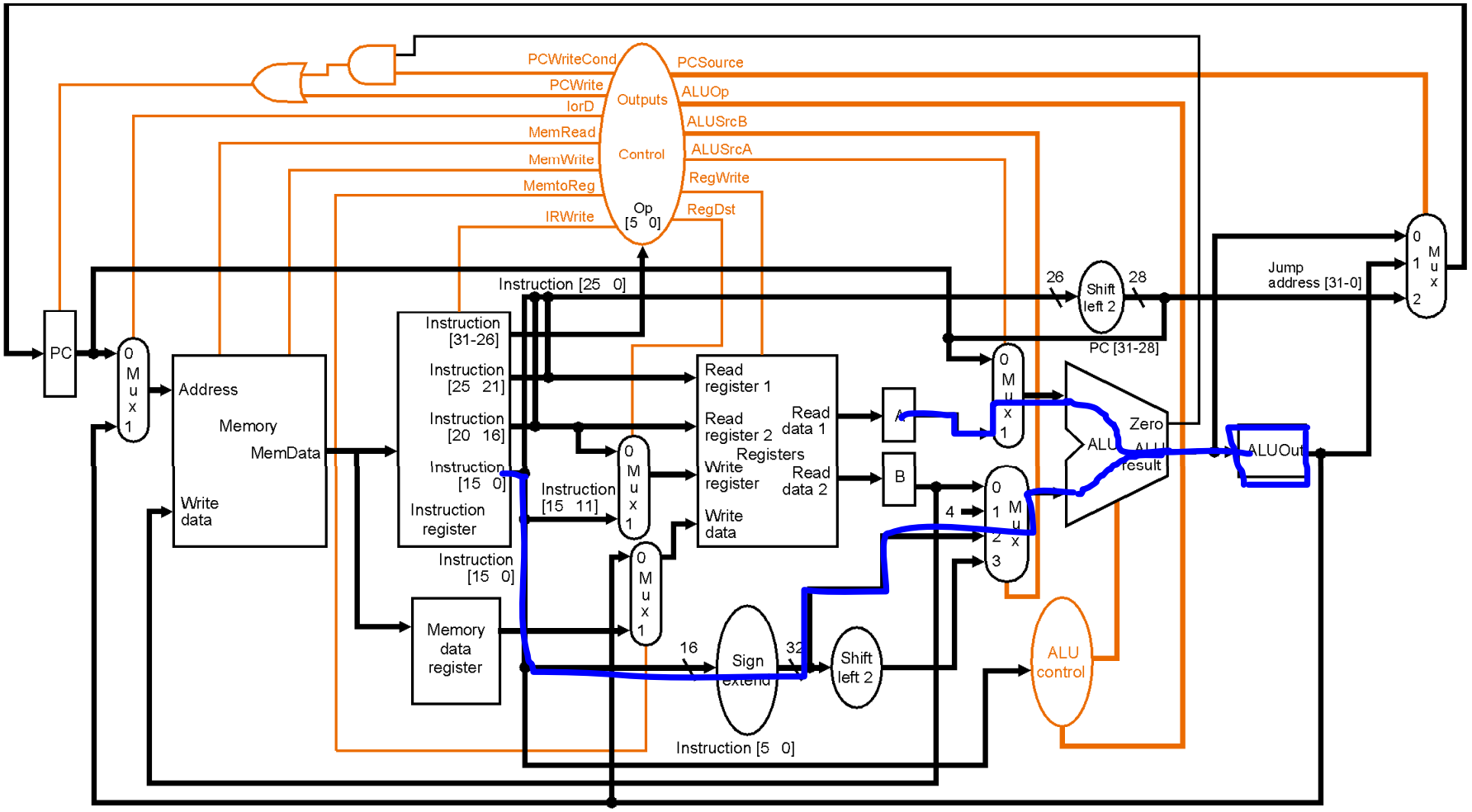
- **Cycle 3:** ALUout = A op B

Cycle 3 for beq: EXecute



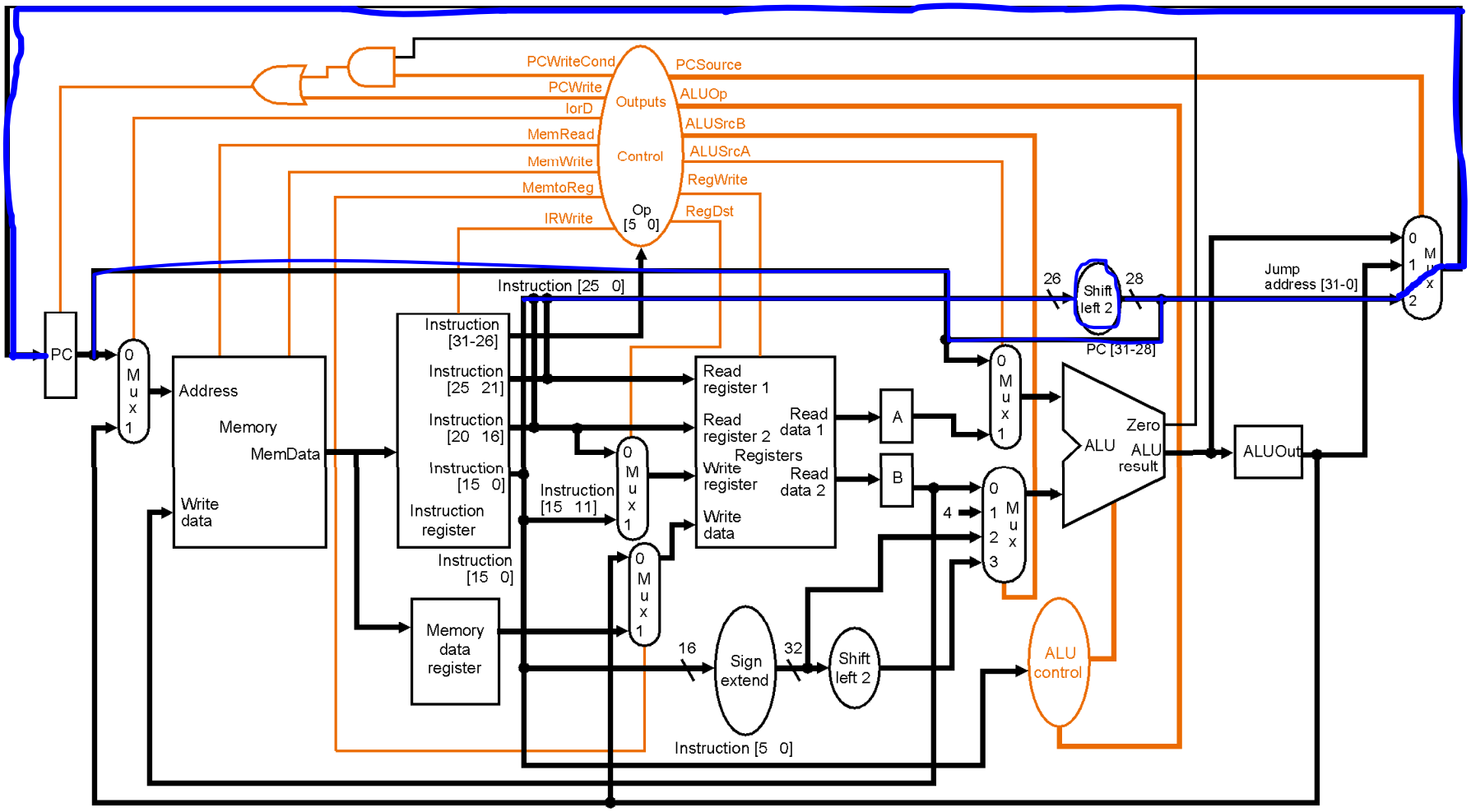
- In cycle 1, PC was incremented by 4
- In cycle 2, ALUout was set to branch target
- This cycle, we conditionally reset PC: if (A==B) PC=ALUout

Cycle 3 for lw and sw: Address Computation



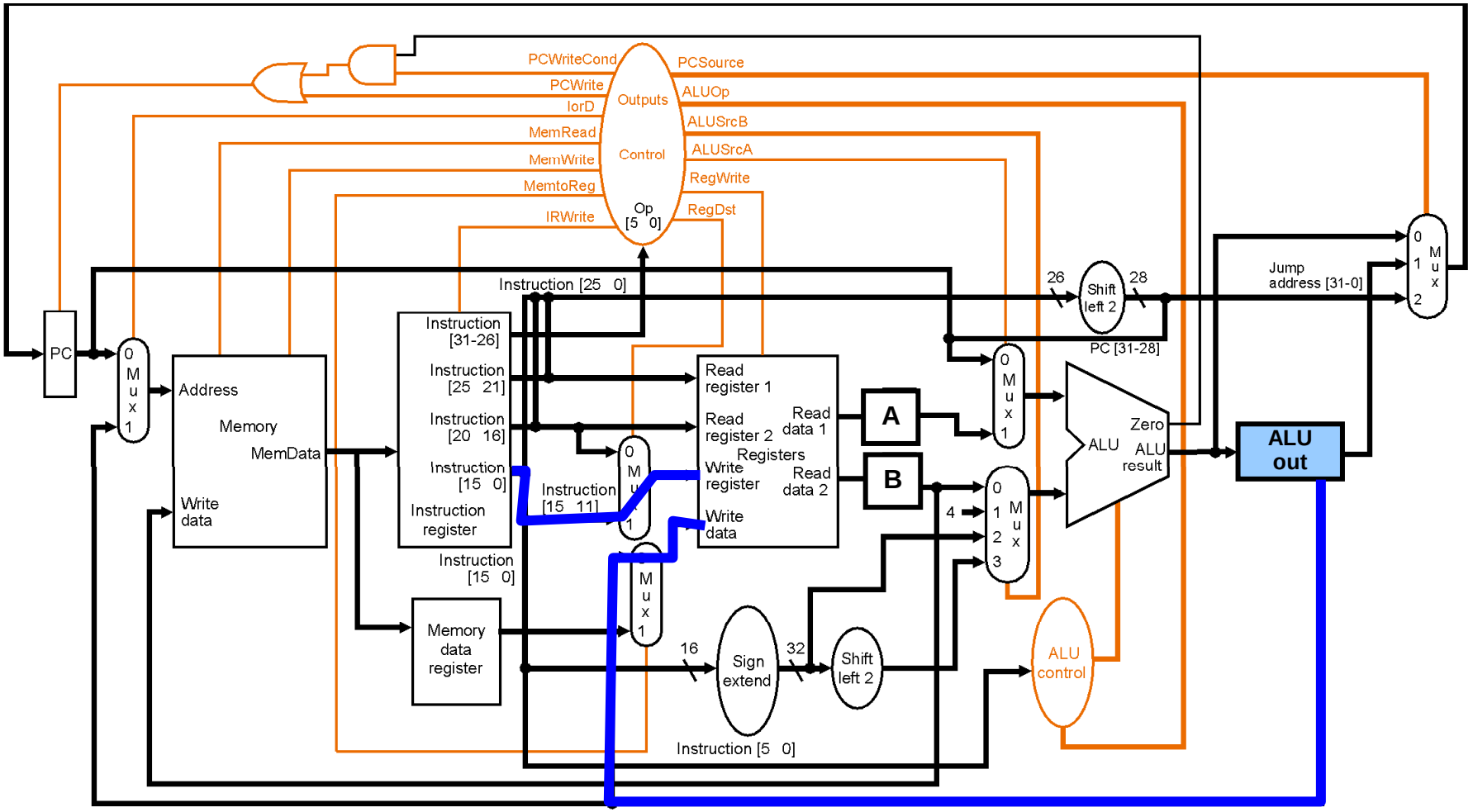
ALUout = A + sign-extend(IR[15-0])

Cycle 3 for Jump



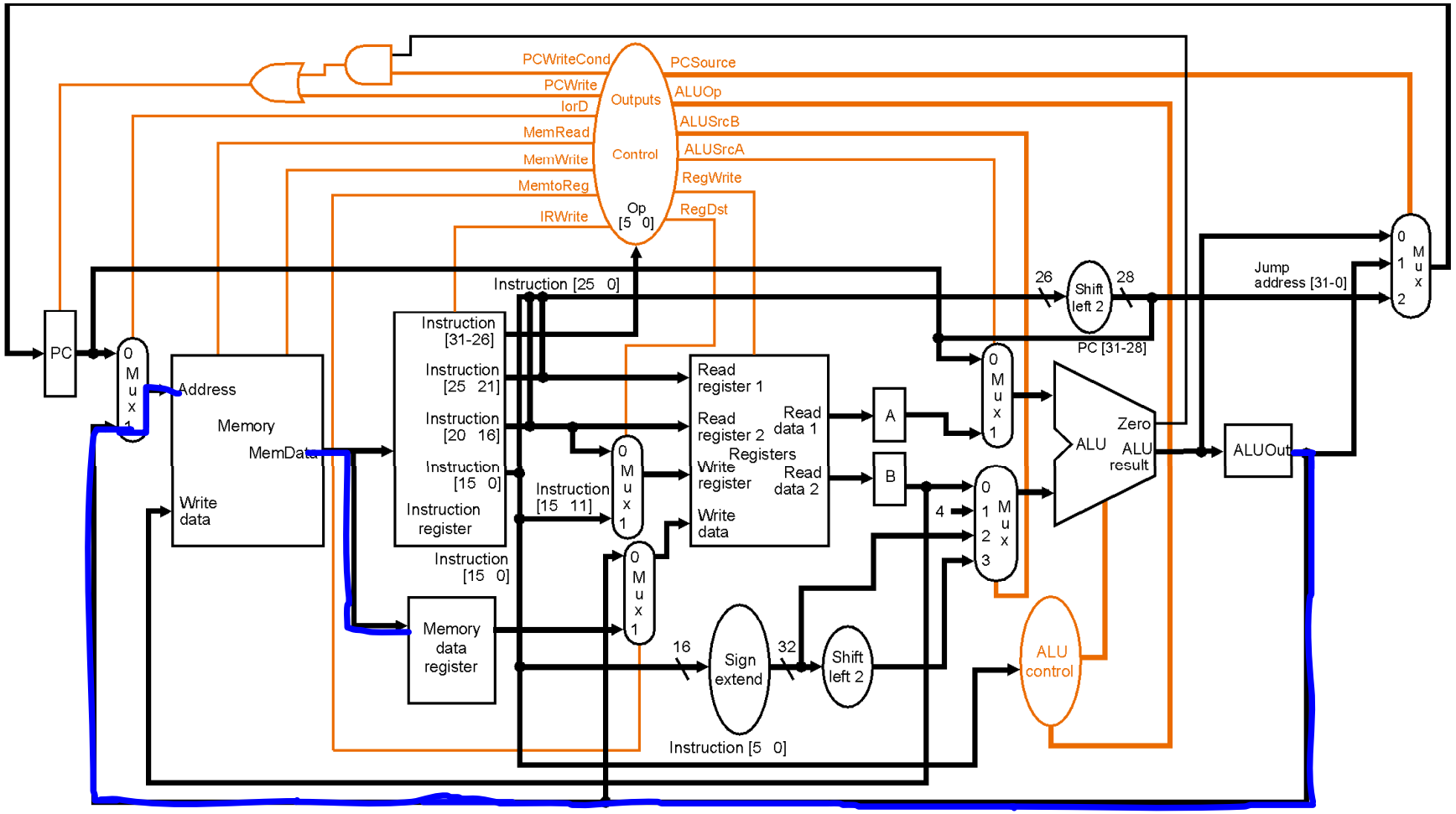

```
PC = PC[31-28] | (IR[25-0] <<2)
```

R-type WriteBack



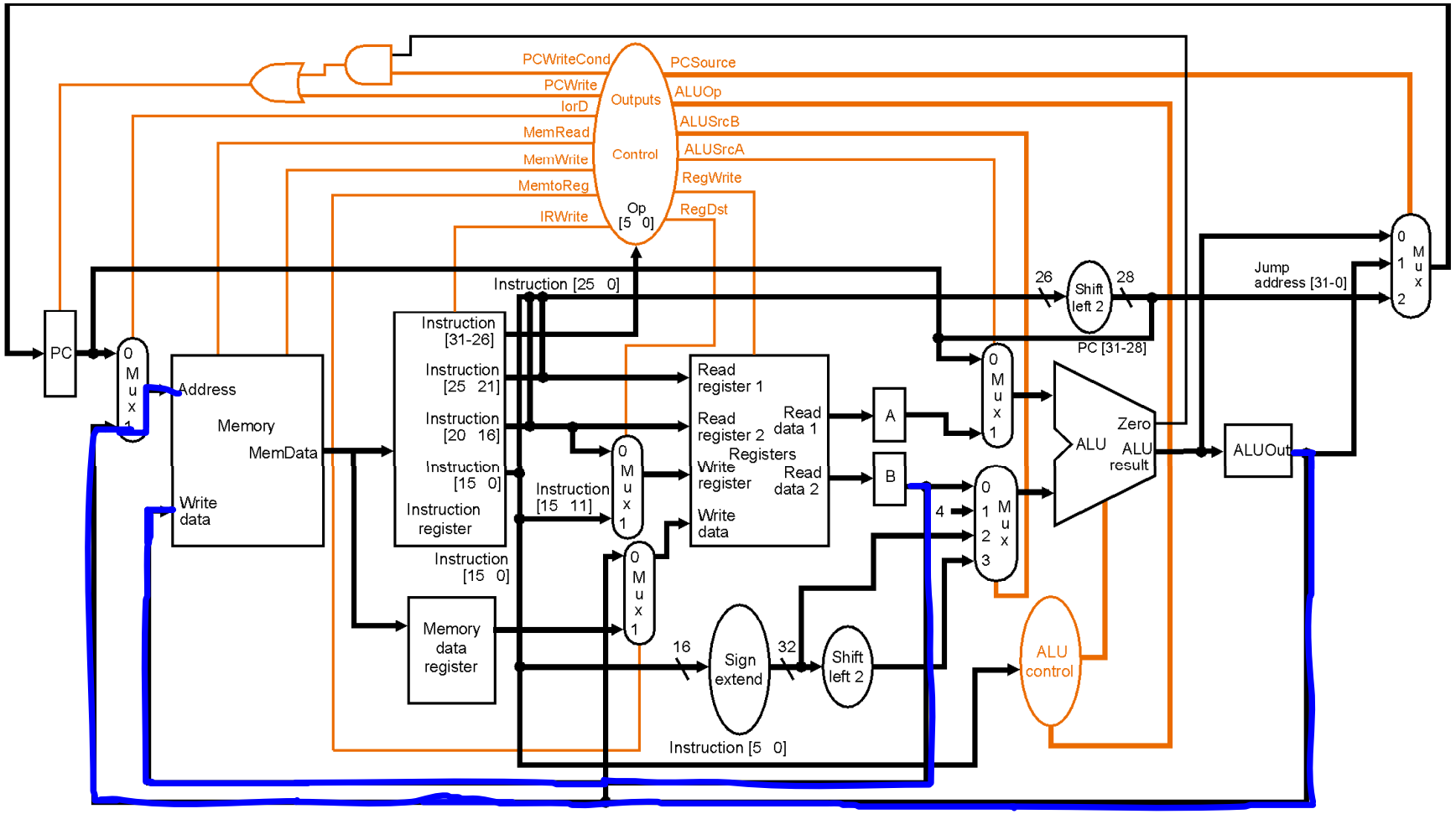
- **Cycle 4:** Reg[IR[15-11]] = ALUout

Cycle 4 for Lw: Memory Access



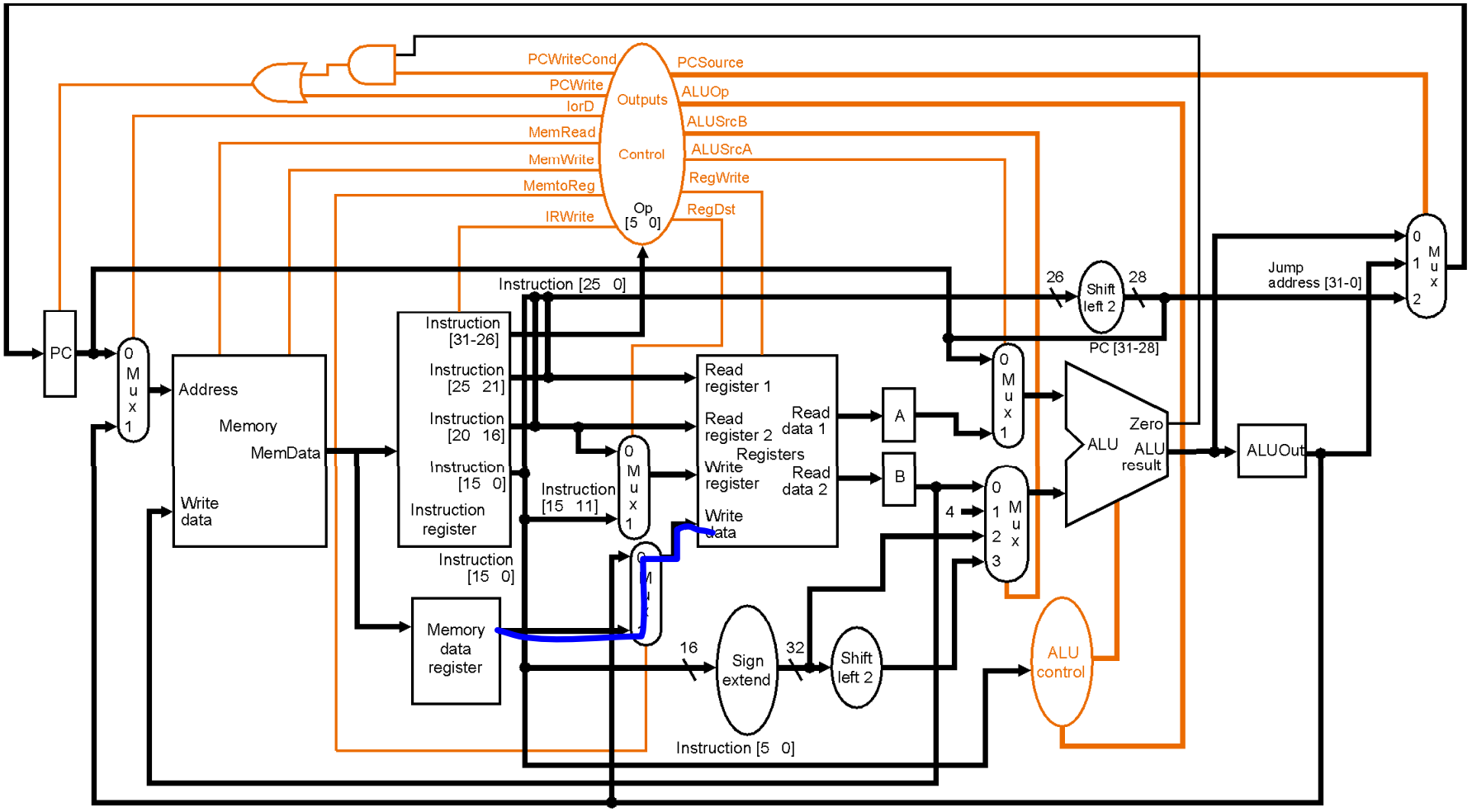
Memory Data Register = Memory[ALUout]

Cycle 4 for Sw: Memory Access



`Memory[ALUout] = B`

Cycle 5 for load: WriteBack



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

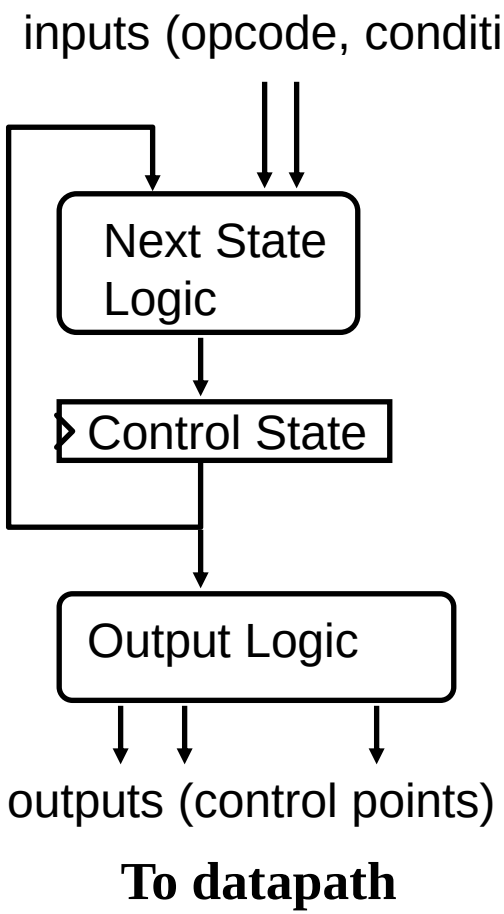
Operations (Dependant RTN) for Each Cycle

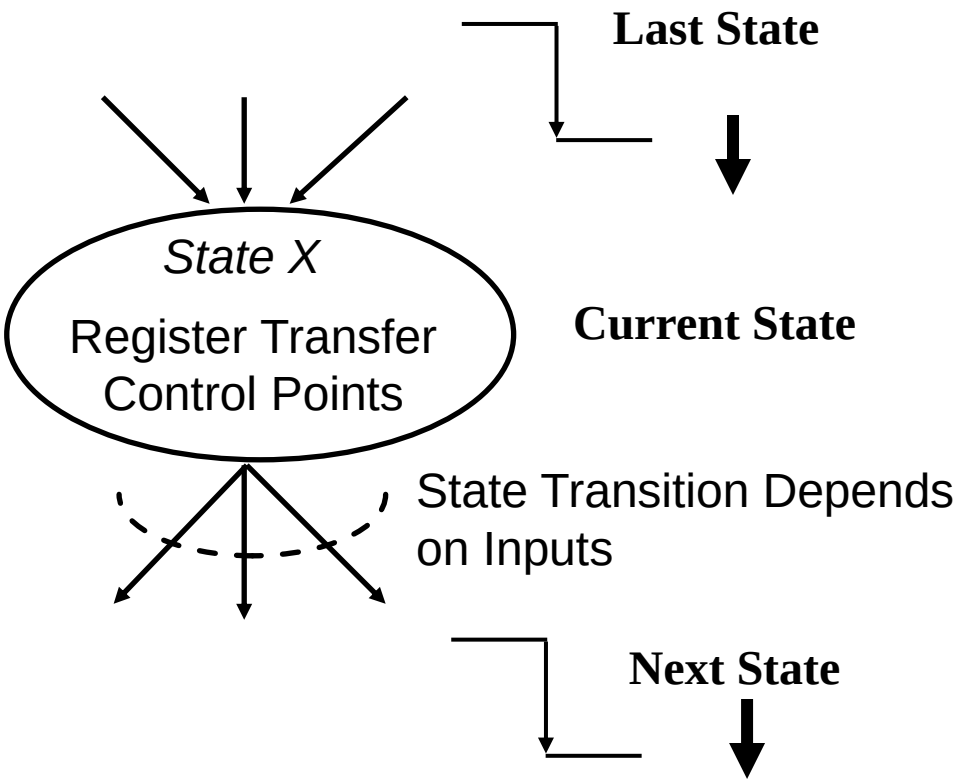
		R-Type	Load	Store	Branch	Jump
IF	Instruction Fetch	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4
ID	Instruction Decode	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)
EX	Execution	ALUout ← A funct B	ALUout ← A + SignEx(Im16)	ALUout ← A + SignEx(Im16)	Zero ← A - B Zero: PC ← ALUout	PC ← Jump Address
MEM	Memory		M ← Mem[ALUout]	Mem[ALUout] ← B		
WB	Write Back	R[rd] ← ALUout	R[rt] ← M			

Instruction Fetch (IF) & Instruction Decode cycles are common for all instructions

Finite State Machine (FSM) Control Model

- **State specifies control points (outputs) for Register Transfer.**
- **Control points (outputs) are assumed to depend only on the current state and not inputs (i.e. Moore finite state machine)**
- **Transfer (register/memory writes) and state transition occur upon exiting the state on the falling edge of the clock.**

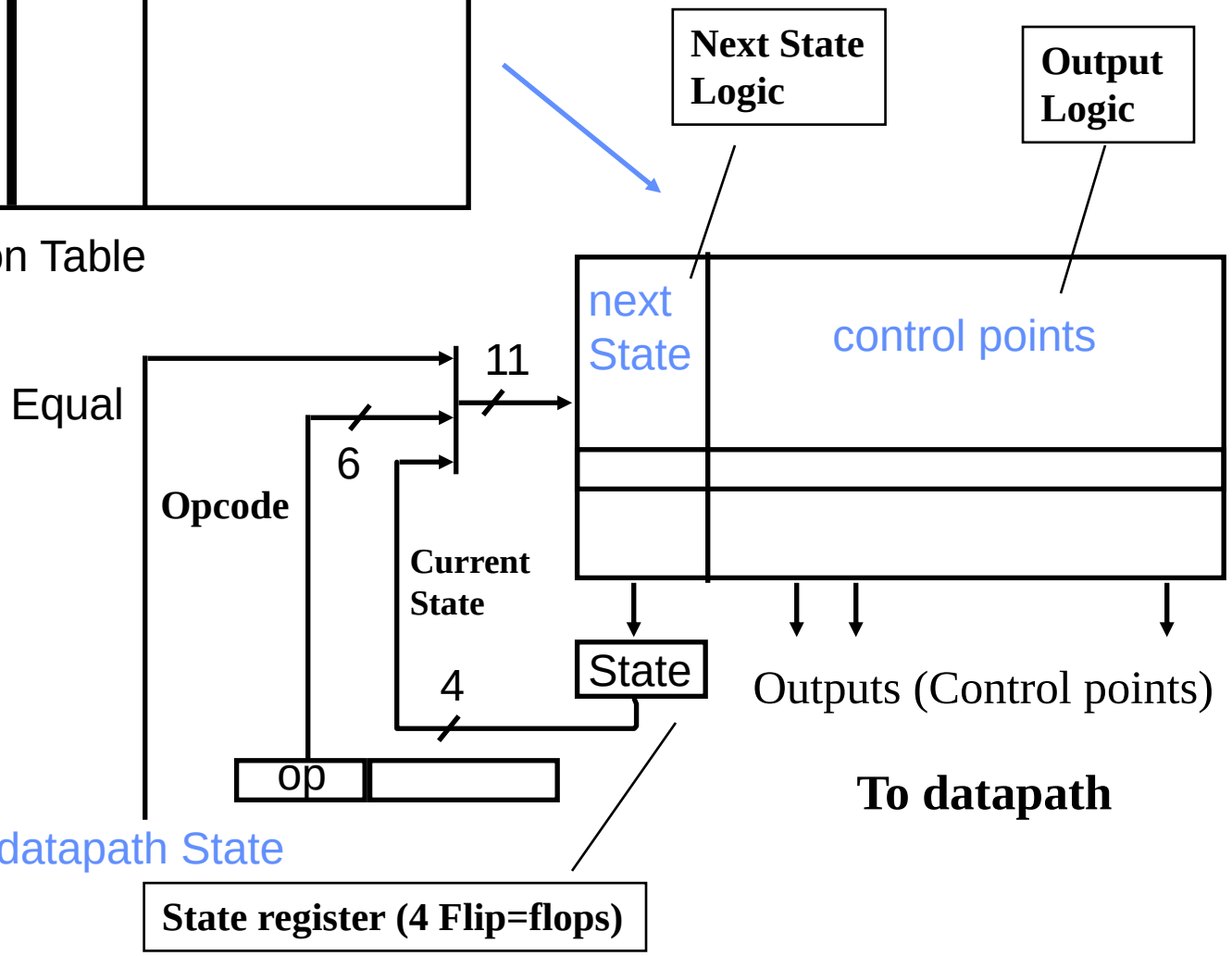




Traditional FSM Controller

state	op	cond	next state	control points

Truth or Transition Table

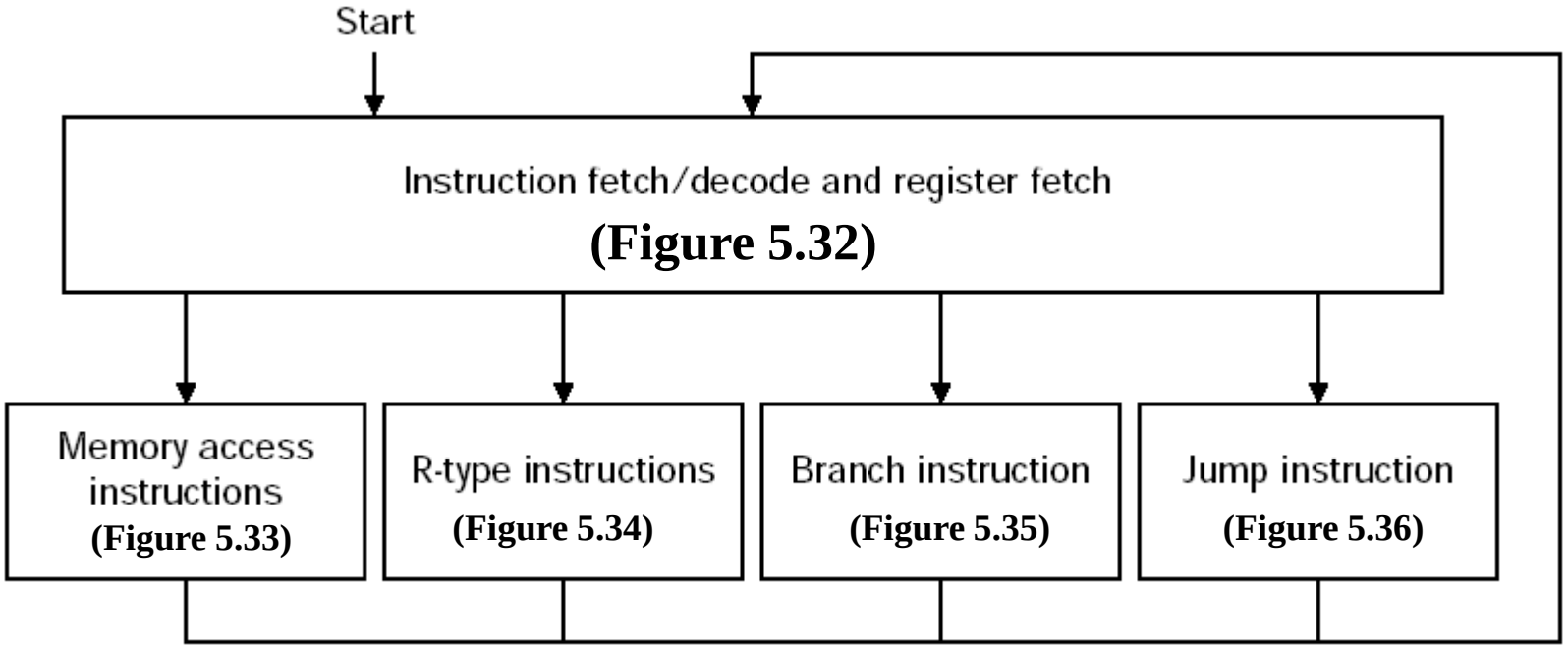


Traditional FSM Controller

datapath + state diagram => control

- **Translate RTN statements into control points.**
- **Assign states.**
- **Implement the controller.**

High-Level View of Finite State Machine Control

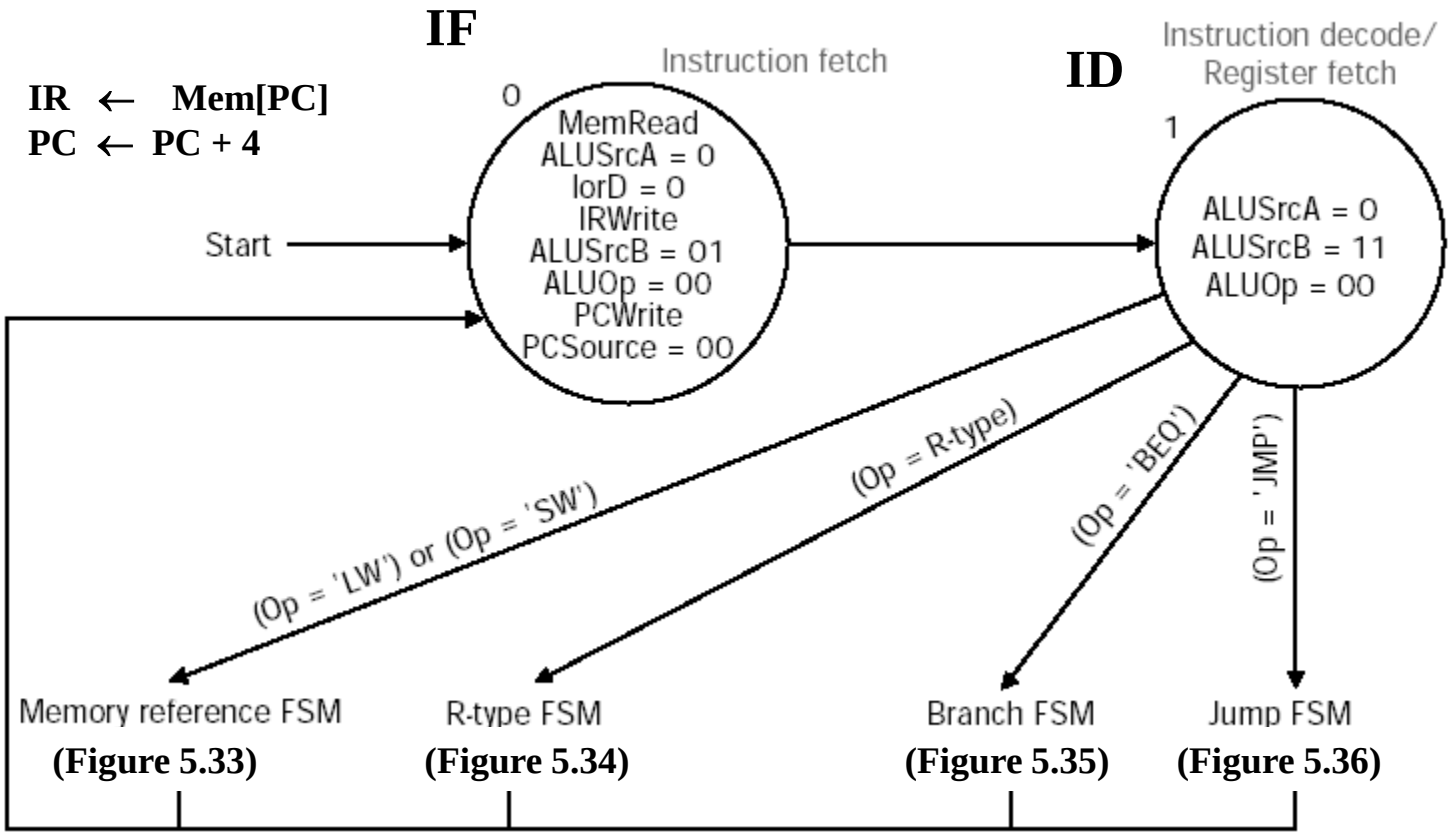


- **First steps are independent of the instruction class**
- **Then a series of sequences that depend on the instruction opcode**
- **Then the control returns to fetch a new instruction.**
- **Each box above represents one or several state.**

(Figure 5.31 page 332)

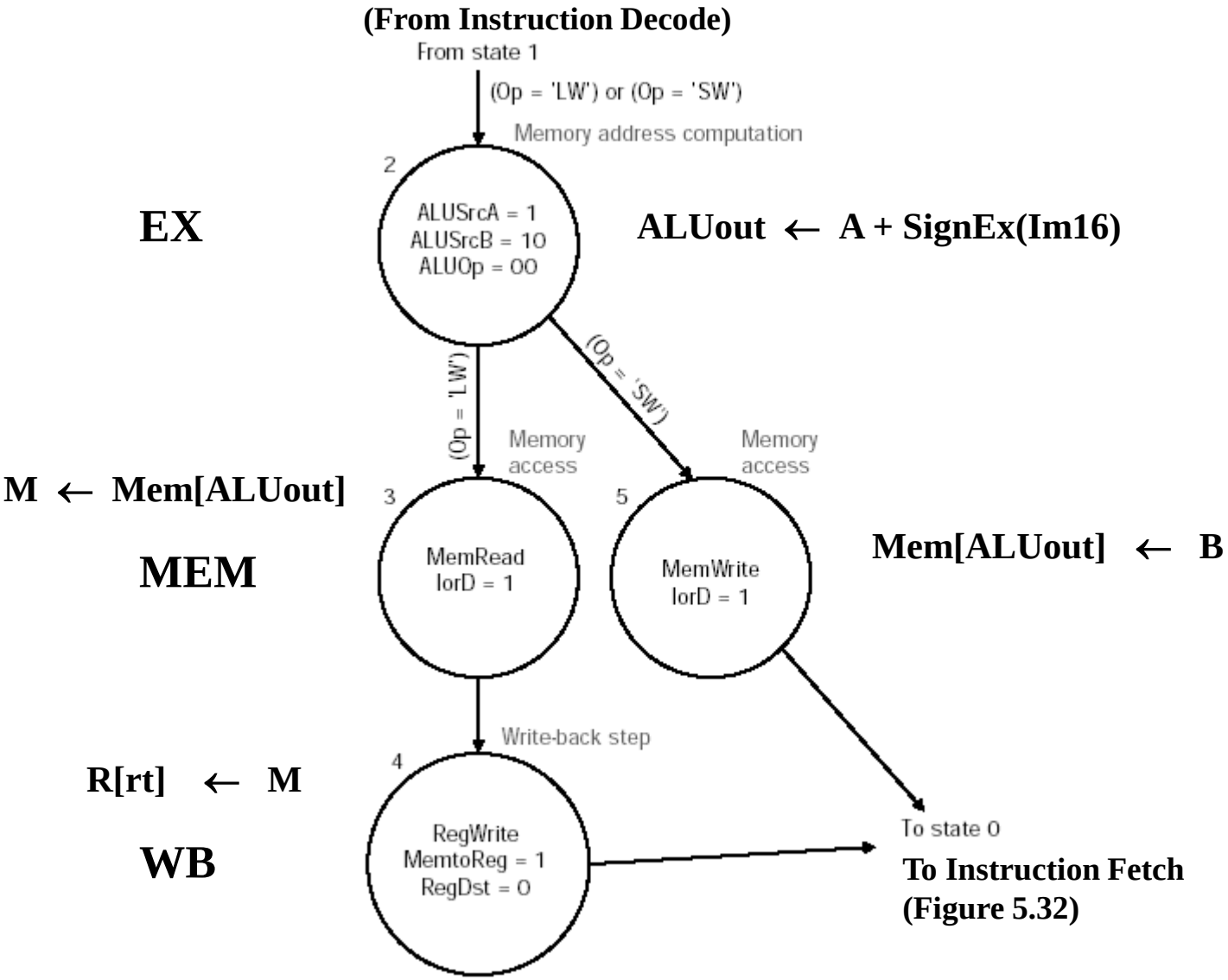
Instruction Fetch (IF) and Decode (ID) FSM States

A ← R[rs]
B ← R[rt]
ALUout ← PC + (SignExt(imm16) x4)



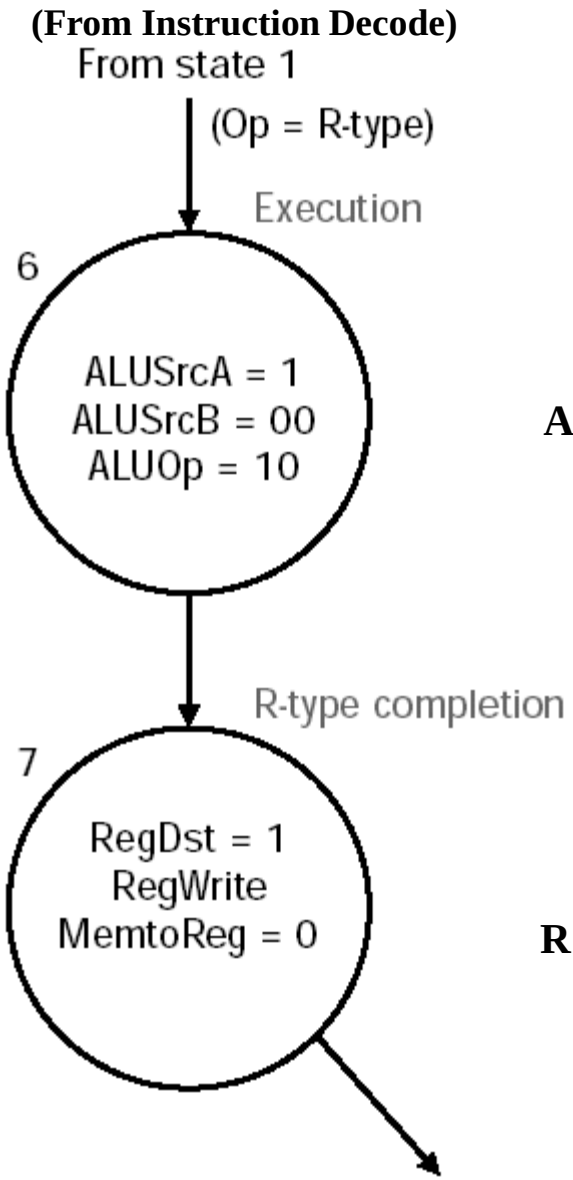
(Figure 5.32 page 333)

Load/Store Instructions FSM States



(Figure 5.33 page 334)

R-Type Instructions FSM States



EX

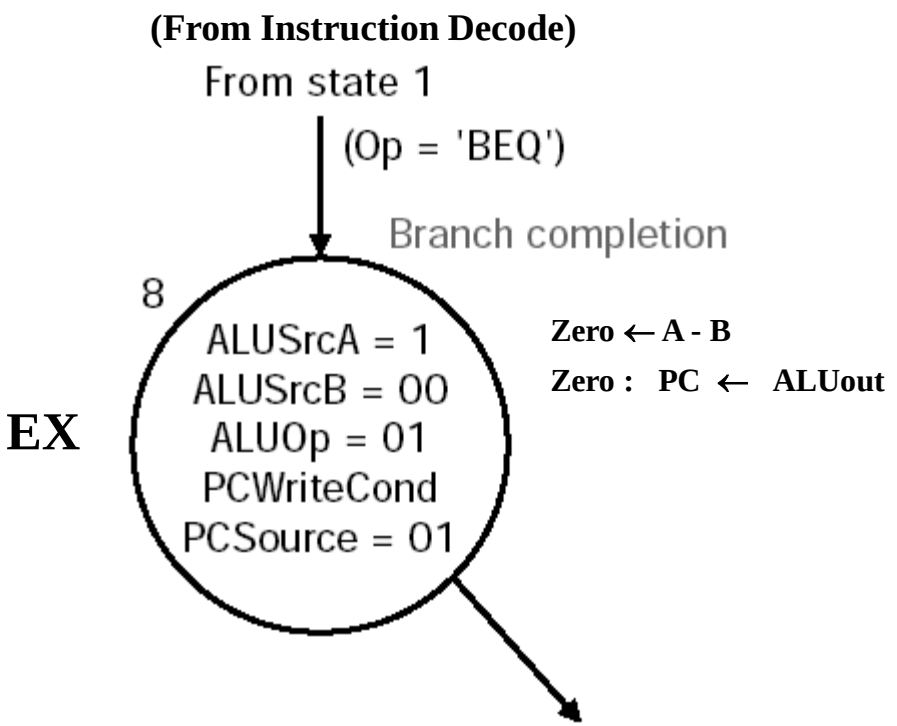
ALUout ← A funct B

WB

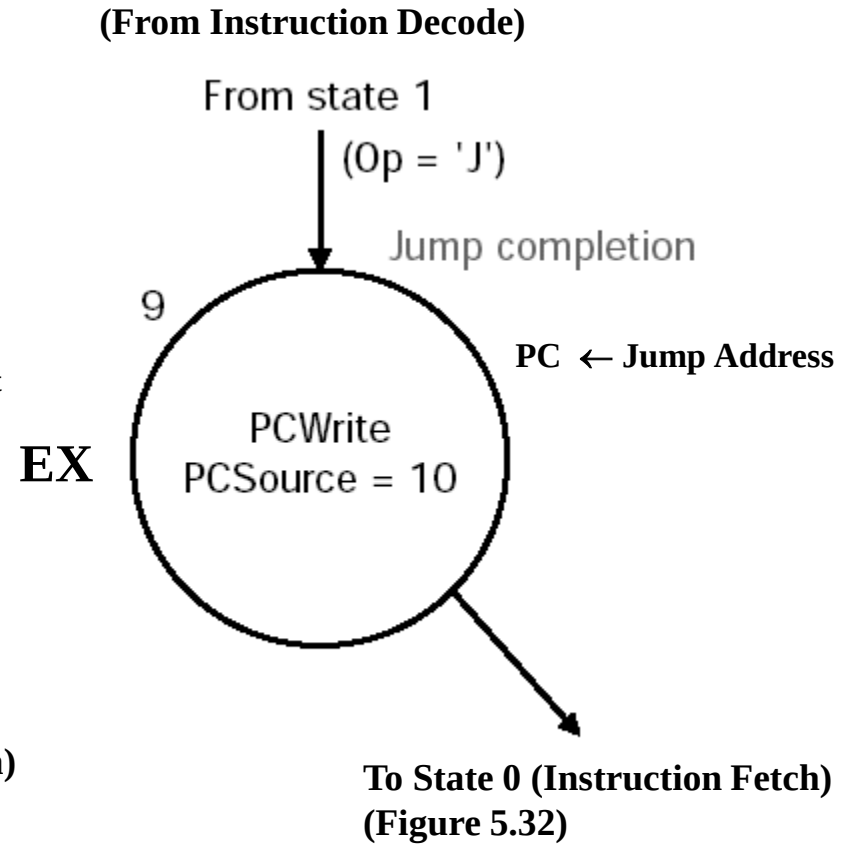
R[rd] ← ALUout

(Figure 5.34 page 335)

Branch Instruction Single EX State

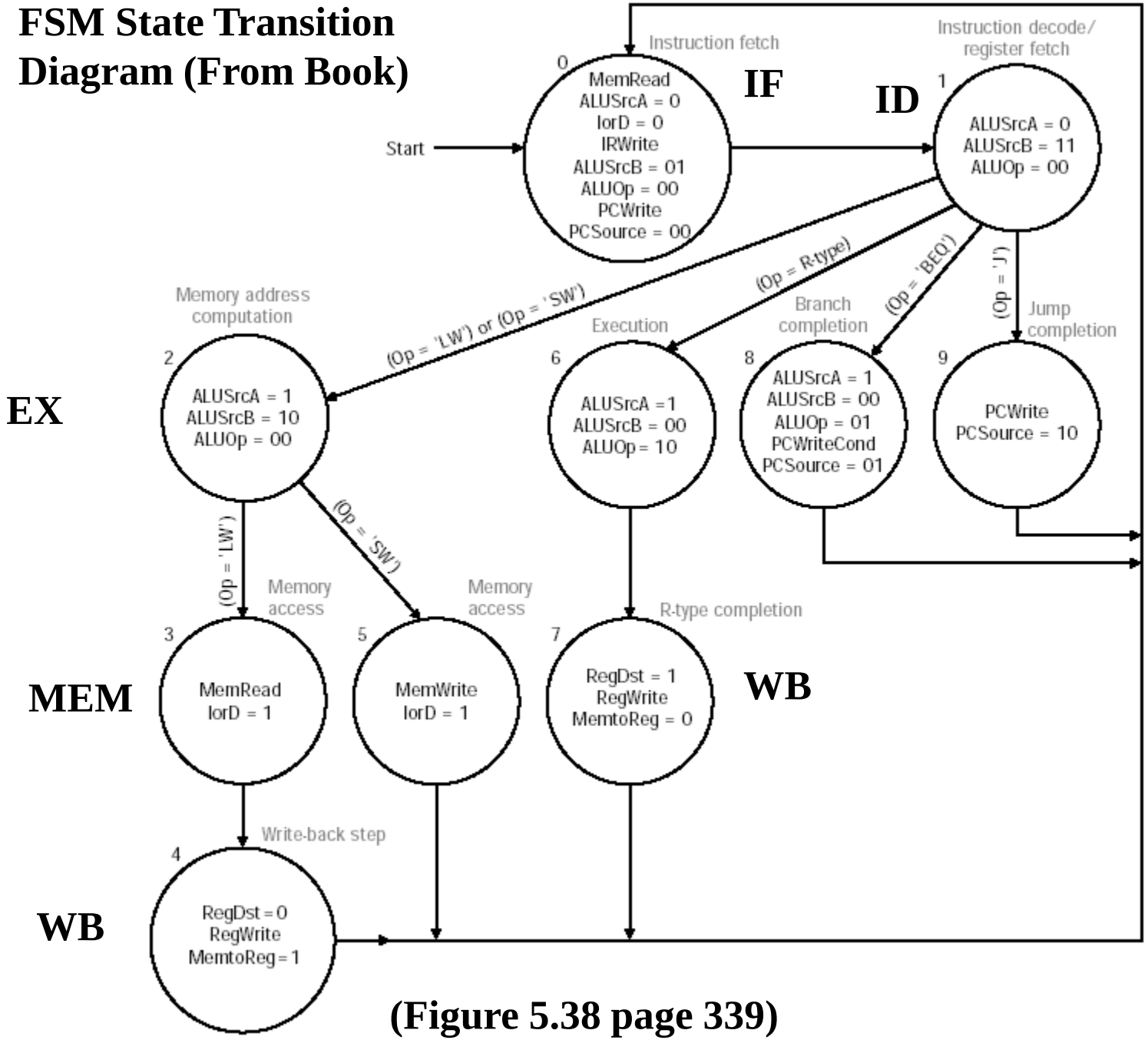


Jump Instruction Single EX State



(Figures 5.35, 5.36 page 337)

FSM State Transition Diagram (From Book)



(Figure 5.38 page 339)

MIPS Multi-cycle Datapath Performance Evaluation

- **What is the average CPI?**
  - **State diagram gives CPI for each instruction type.**
  - **Workload (program) below gives frequency of each type.**

Type	CPI_i for type	Frequency	CPI_i x $freqI_i$
Arith/Logic	4	40%	1.6
Load	5	30%	1.5
Store	4	10%	0.4
branch	3	20%	0.6
		Average CPI:	4.1

Better than CPI = 5 if all instructions took the same number of clock cycles (5).

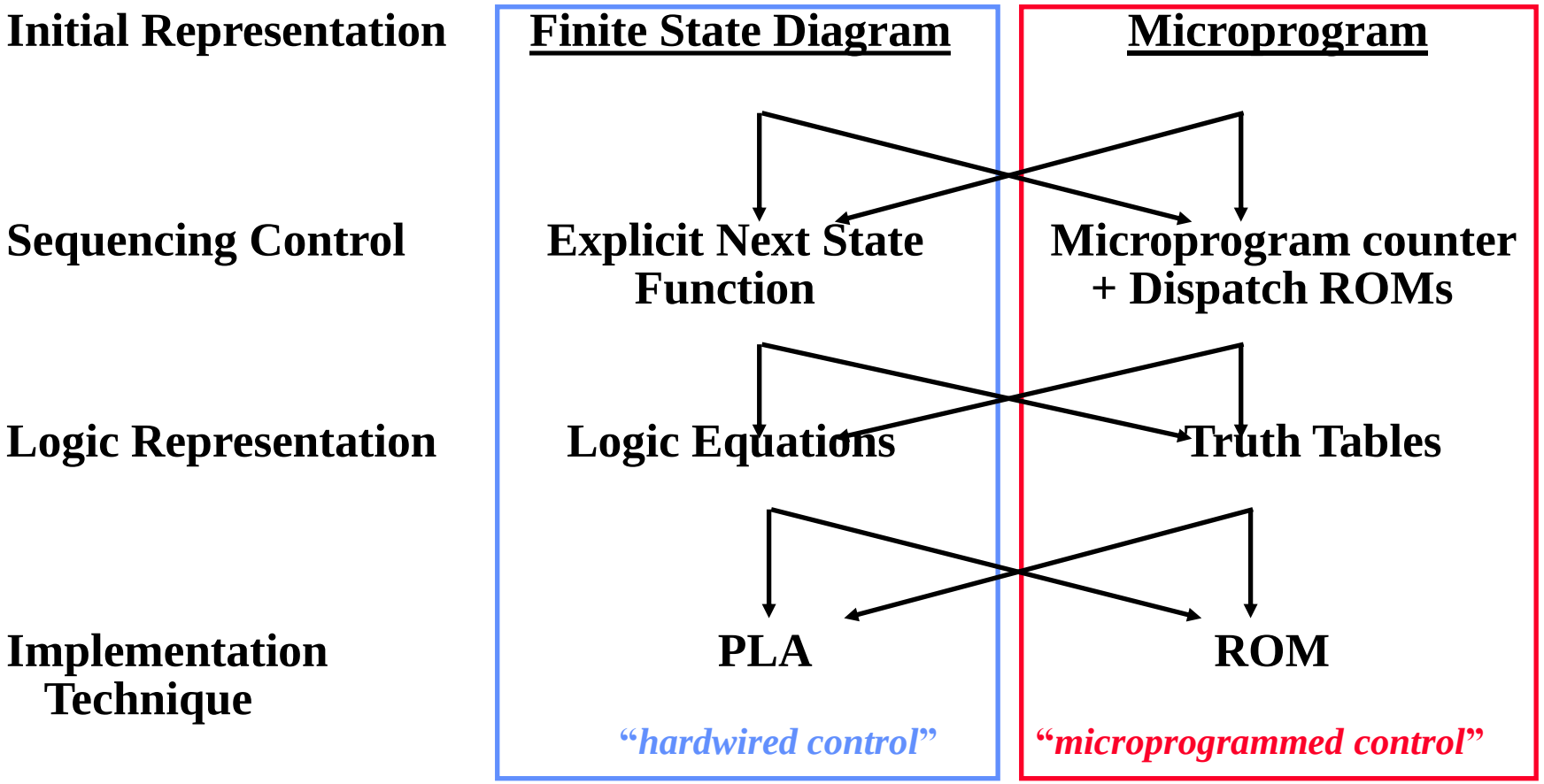
Control Implementation Alternatives

- **Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.**

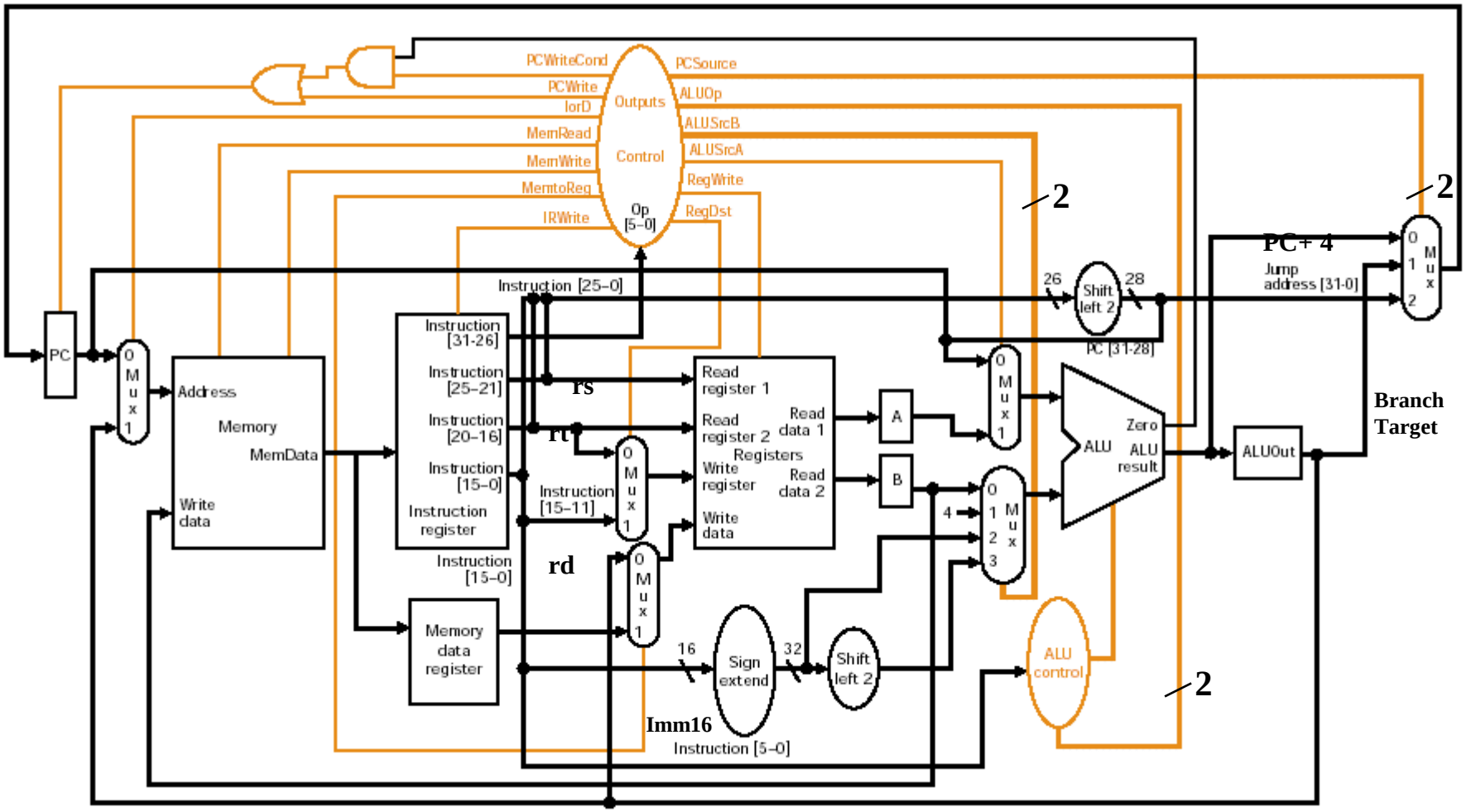
- **Two general approaches**
  - **Hardwired**
    - **In a hardwired state machine, we build a circuit that implements the state machine directly**
  - **Microprogrammed**
    - **In a microprogrammed state machine, we use a memory to hold the state transition tables.**

Control Implementation Alternatives

- Two general approaches
  - Hardwired
  - Microprogrammed



Multiple Cycle Datapath With Control Lines



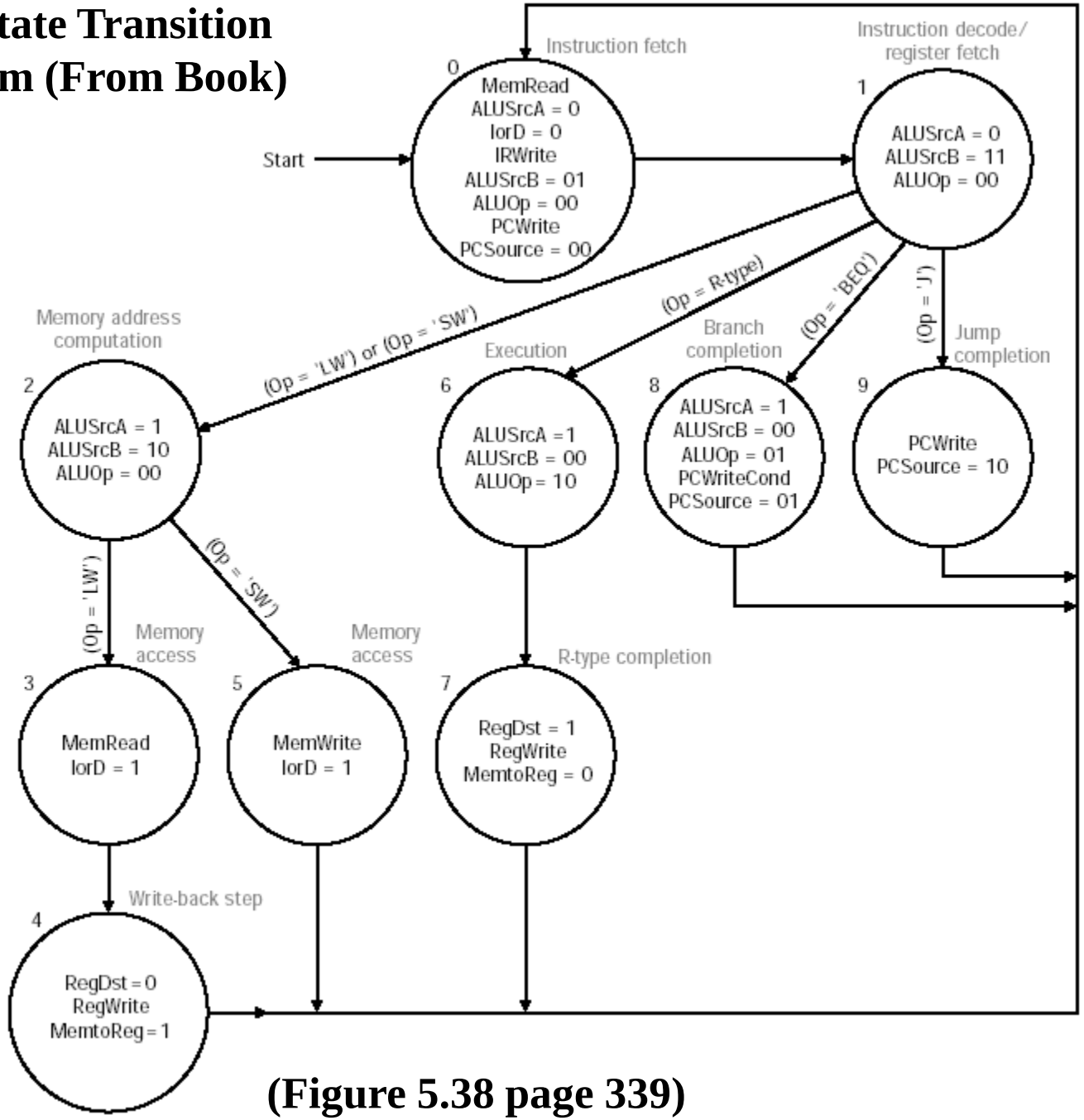
(Book version: ORI not supported, Jump supported)

Operations (Dependant RTN) for Each Cycle

		R-Type	Load	Store	Branch	Jump
IF	Instruction Fetch	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4
ID	Instruction Decode	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)
EX	Execution	ALUout ← A funct B	ALUout ← A + SignEx(Im16)	ALUout ← A + SignEx(Im16)	Zero ← A - B Zero: PC ← ALUout	PC ← Jump Address
MEM	Memory		M ← Mem[ALUout]	Mem[ALUout] ← B		
WB	Write Back	R[rd] ← ALUout	R[rt] ← M			

Instruction Fetch (IF) & Instruction Decode cycles are common for all instructions

FSM State Transition Diagram (From Book)



(Figure 5.38 page 339)