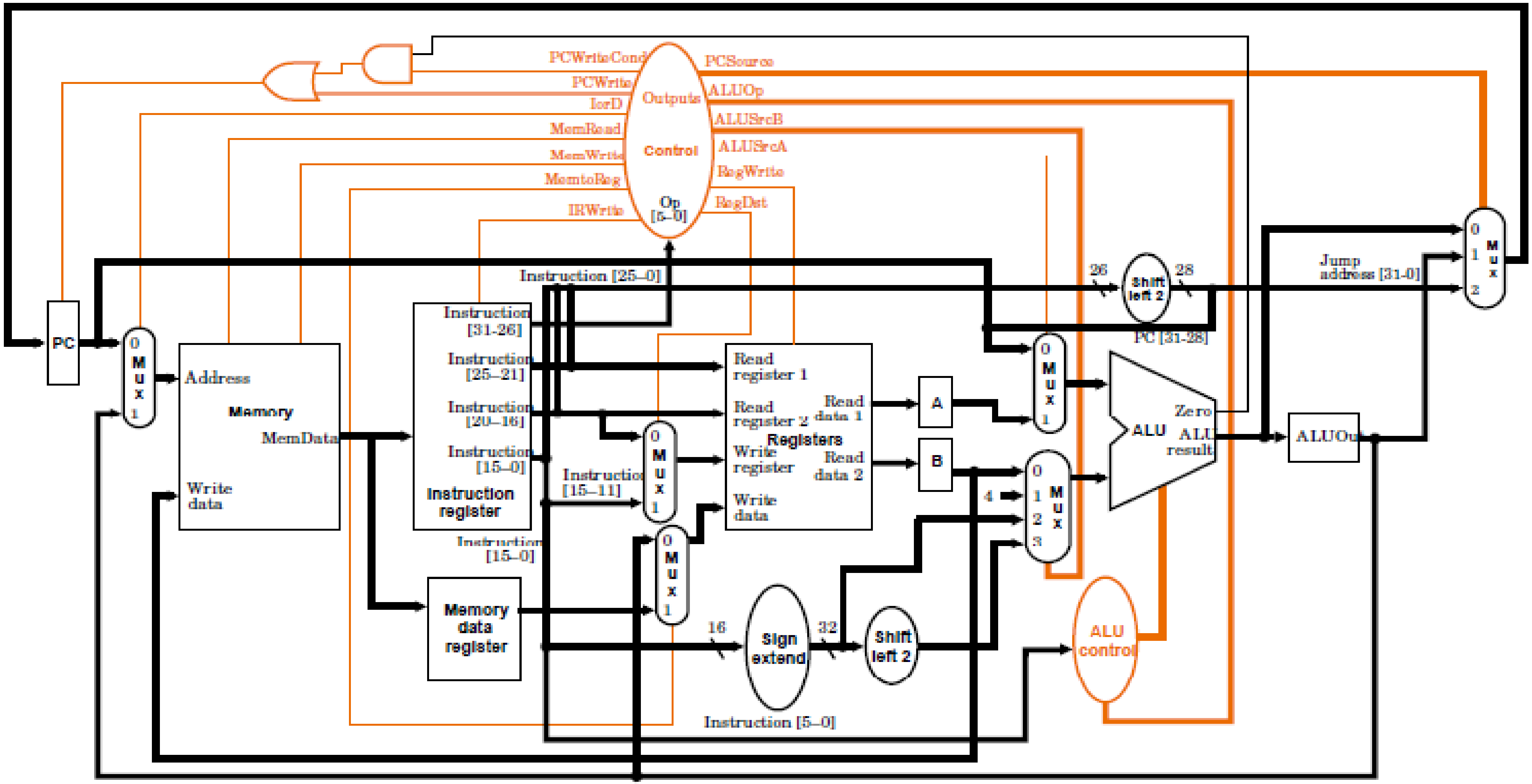
Multi-Cycle Architecture

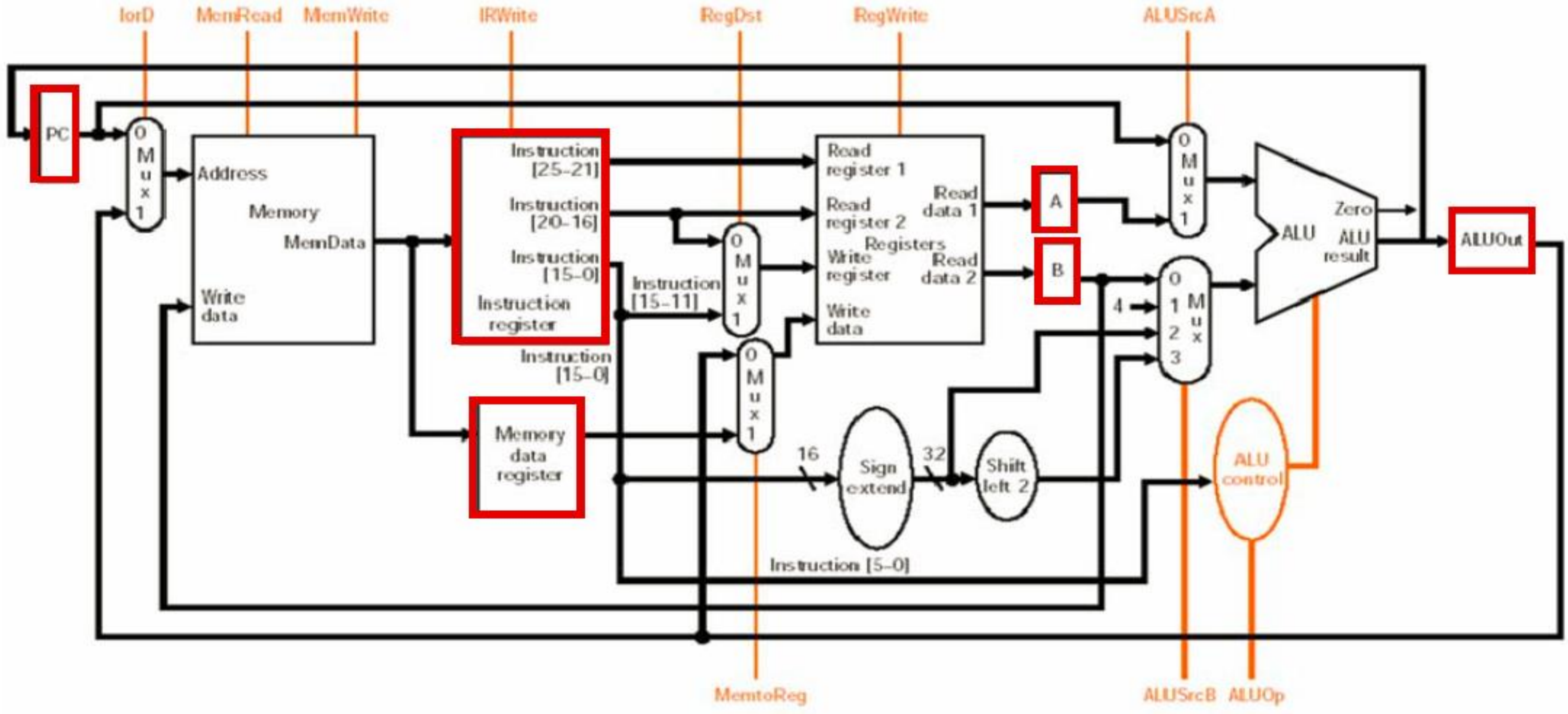
Part 2

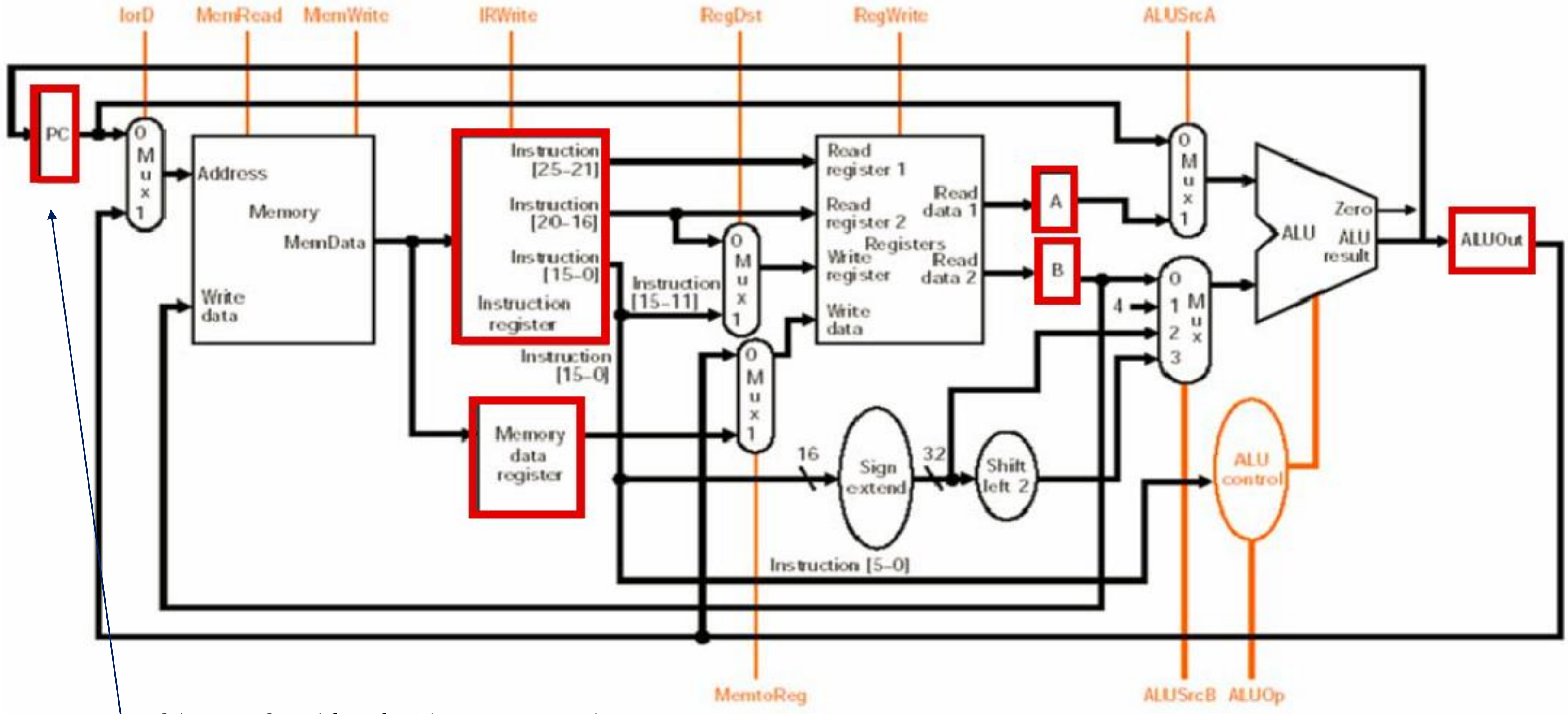
Please note change in Office Hours

- ***Rosina will have the following office hours until A4 is due:**
- On Wednesday July 08 1pm-2pm
- Thursday July 09 12:30-1:30pm
- Monday July 13th 1pm-2pm



MULTI CYCLE DATA PATH


IorD
MemRead
MemWrite
IRWrite
RegDst
RegWrite
ALUSrcA
PC
0 Mux 1
Address
Memory
MemData
Write data
Instruction [25–21]
Instruction [20–16]
Instruction [15–0]
Instruction register
Instruction [15–0]
Instruction [15–11]
0 Mux 1
0 Mux 1
Memory data register
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
0 Mux 1
4
0 1 2 3 Mux
ALU
Zero
ALU result
ALUOut
16
Sign extend
32
Shift left 2
ALU control
Instruction [5–0]
MemtoReg
ALUSrcB
ALUOp



PC is Not Considered a Temporary Register:

Its Writes are controlled

Are all Temporary Registers written to at the **End** of every clock cycle ?

A) No : Temporary Registers are Not written to.

B) **IR** is not written every clock cycle but the other temporary registers are written to.

C) Yes

D) **IR** and **ALUOut** are not written to every clock cycle but others are

E) Registers **A** and **B** are the only registers written to at the end of each clock cycle

If temporary registers (other than IR) are All written to at the **End** of every clock cycle…

- What also Must Be True :

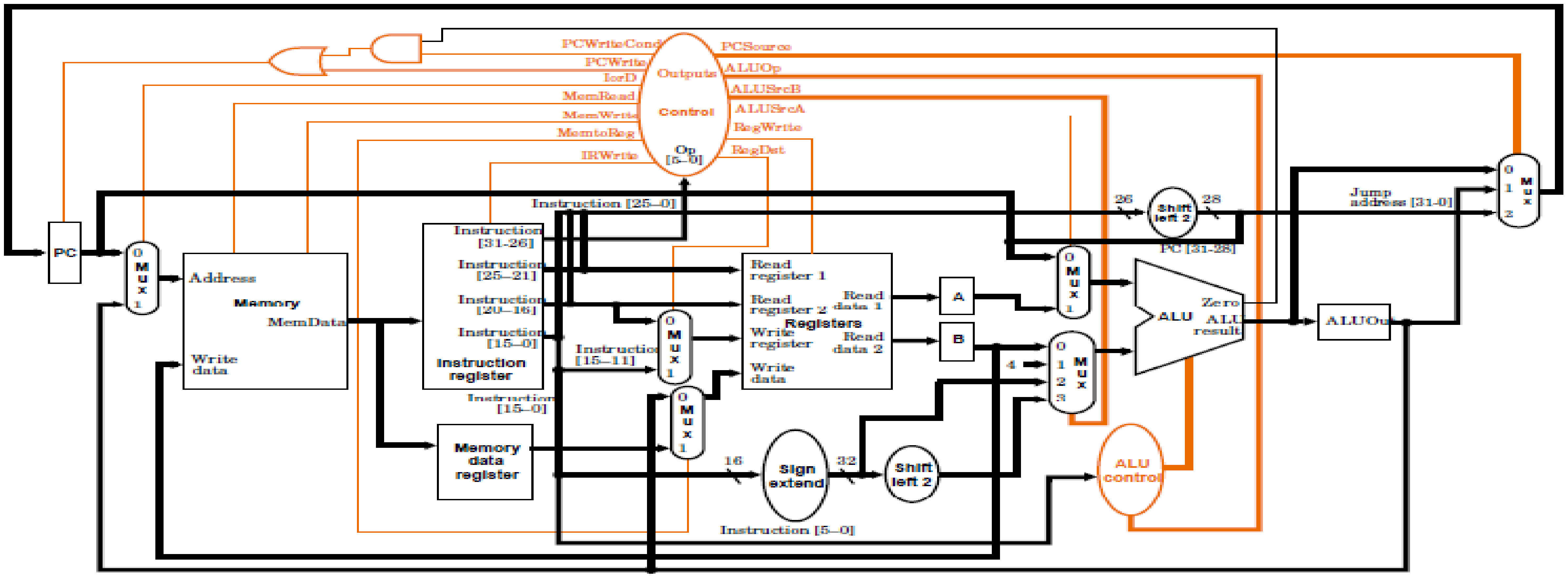
A) Temporary Registers are Not Read from

B) Temporary Registers are only Read from when they are needed

C) Temporary Registers are controlled for reading and writing

D) Temporary Registers are Not Read from except Once

E) Temporary Registers are also Read from at the beginning of Every Clock Cycle



Step	R-Type	Memory Ref	Branch	Jumps

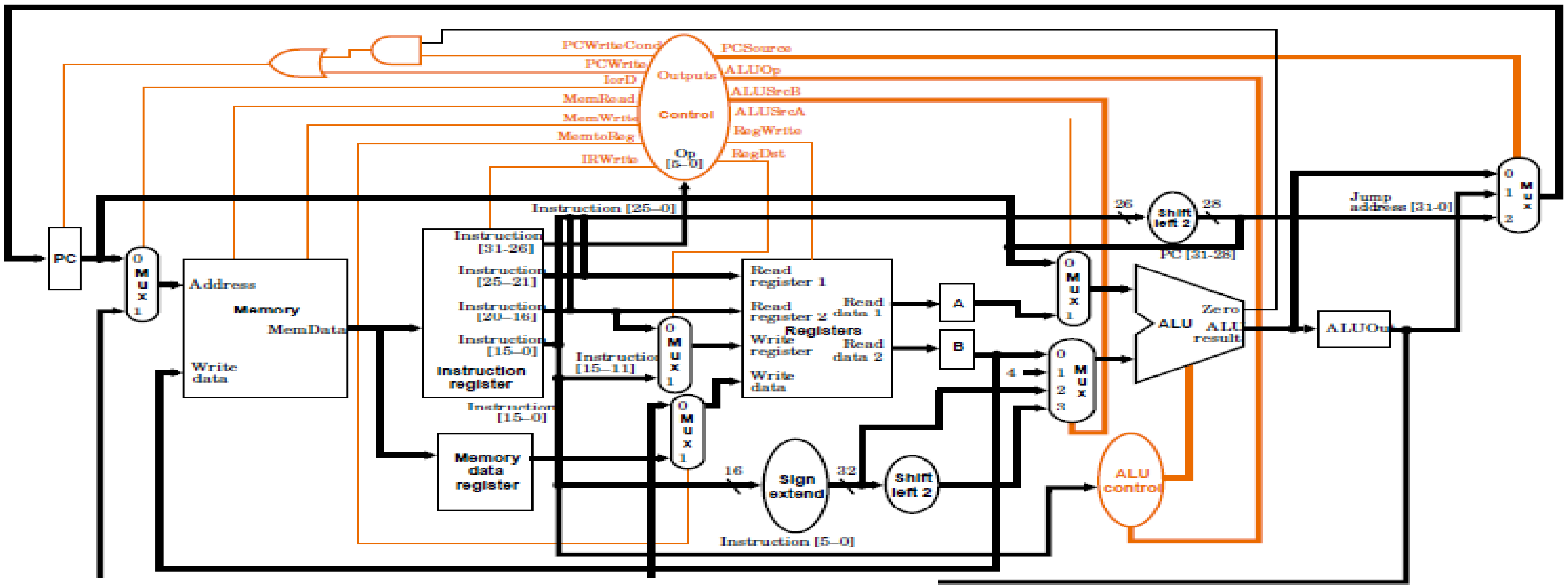
Instruction Fetch	IR = Mem[PC]
	PC = PC+4			

Decode, Register Fetch	A = Reg[IR[25-21]], B = Reg[IR[20-16]]
	ALUOut = PC + se(IR[15-0]) << 2			

Execute, etc	ALUOut =
	A op B
		ALUOut =
		A + se(IR[15-0])
			if (A==B) then
			PC = ALUout
				PC = PC[31-28] | |
				IR[25-0]<< 2

Memory, R-type	Reg[IR[15-11]] = ALUout	LD: MDR = Mem[ALUOut] ST: Mem[ALUOut] = B		
Memory Read		LD: Reg[IR[20-16]] = MDR		

Different Steps In the execution Of Each Instruction



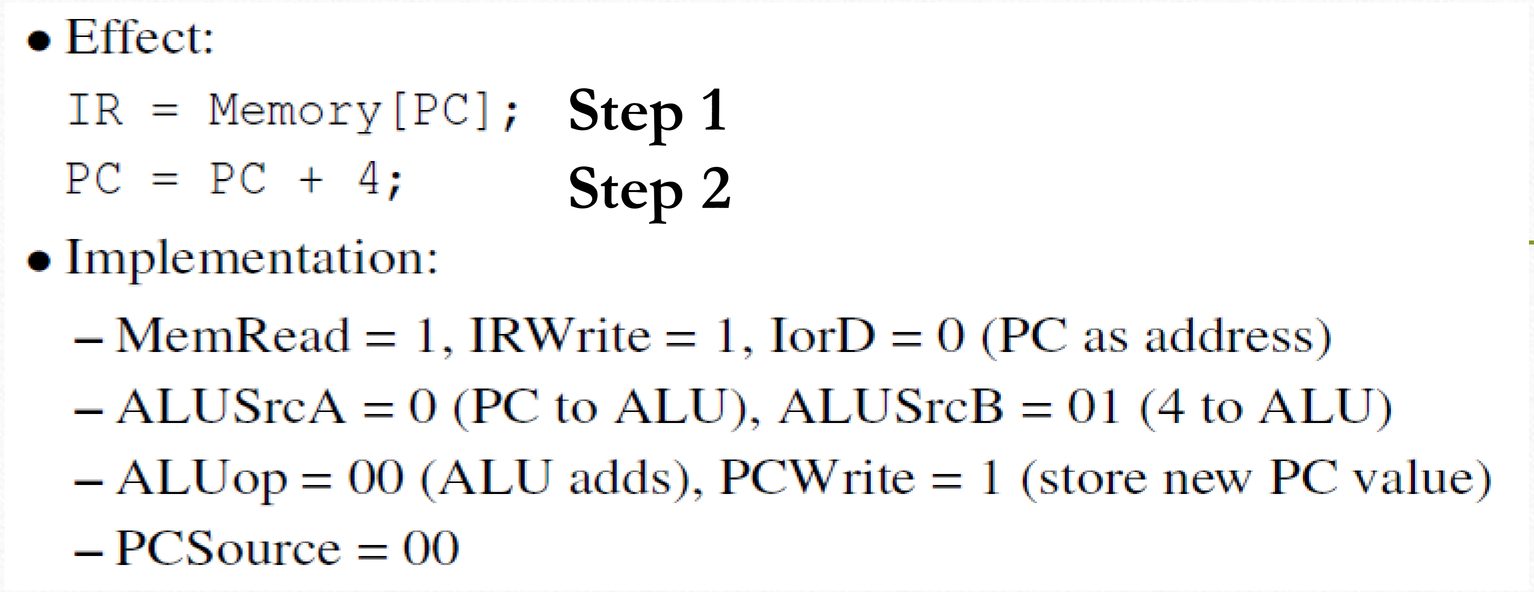
- Effect:

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

STEP 01: Instruction Fetch

- Implementation:
  - – MemRead = 1, IRWrite = 1, IorD = 0 (PC as address)
  - – ALUSrcA = 0 (PC to ALU), ALUSrcB = 01 (4 to ALU)
  - – ALUop = 00 (ALU adds), PCWrite = 1 (store new PC value)
  - – PCSource = 00

Instruction Fetch Stage : How long does it take?



- Effect:

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

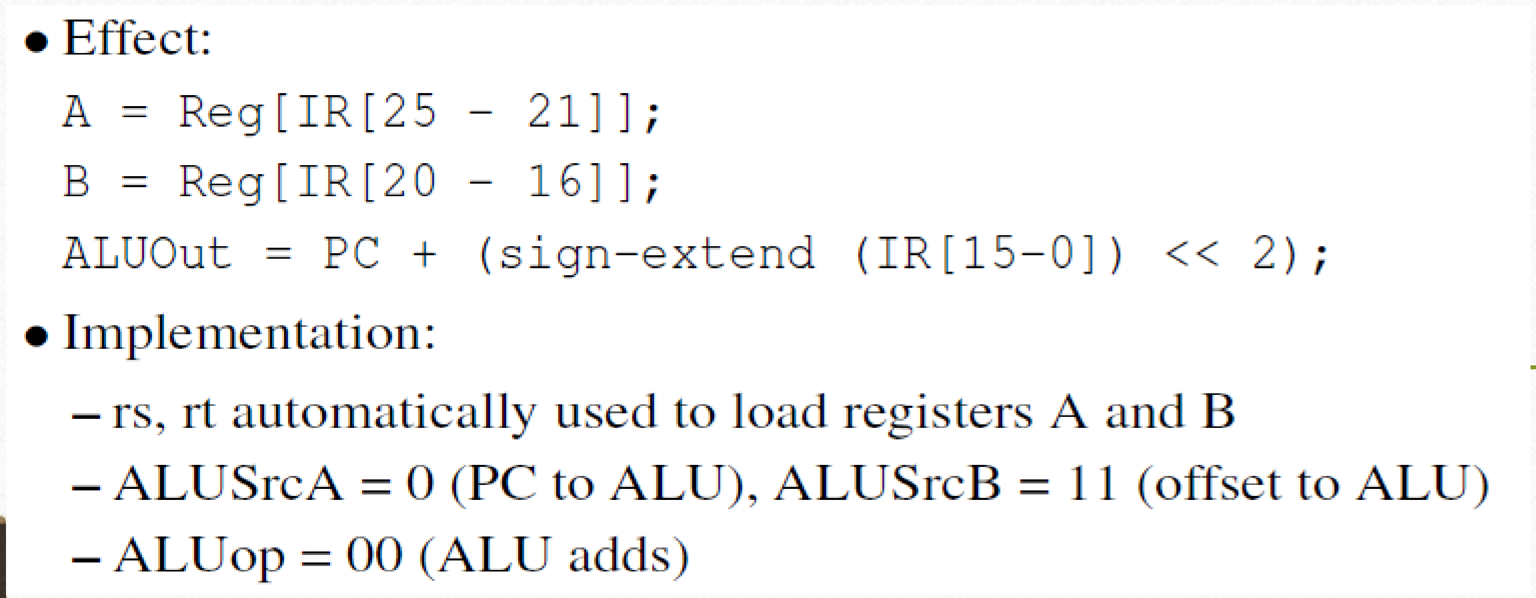
Step 1

Step 2

- Implementation:
  - MemRead = 1, IRWrite = 1, IorD = 0 (PC as address)
  - ALUSrcA = 0 (PC to ALU), ALUSrcB = 01 (4 to ALU)
  - ALUop = 00 (ALU adds), PCWrite = 1 (store new PC value)
  - PCSource = 00

MEMORY ACCESS : 200ps
ALU Operation: 100ps
Writing to Temp Register: 25ps
Slowest part of this Stage: ________________________________

STEP 02: DECODE/Register Fetch



- Effect:

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

- Implementation:
  - rs, rt automatically used to load registers A and B
  - ALUSrcA = 0 (PC to ALU), ALUSrcB = 11 (offset to ALU)
  - ALUop = 00 (ALU adds)

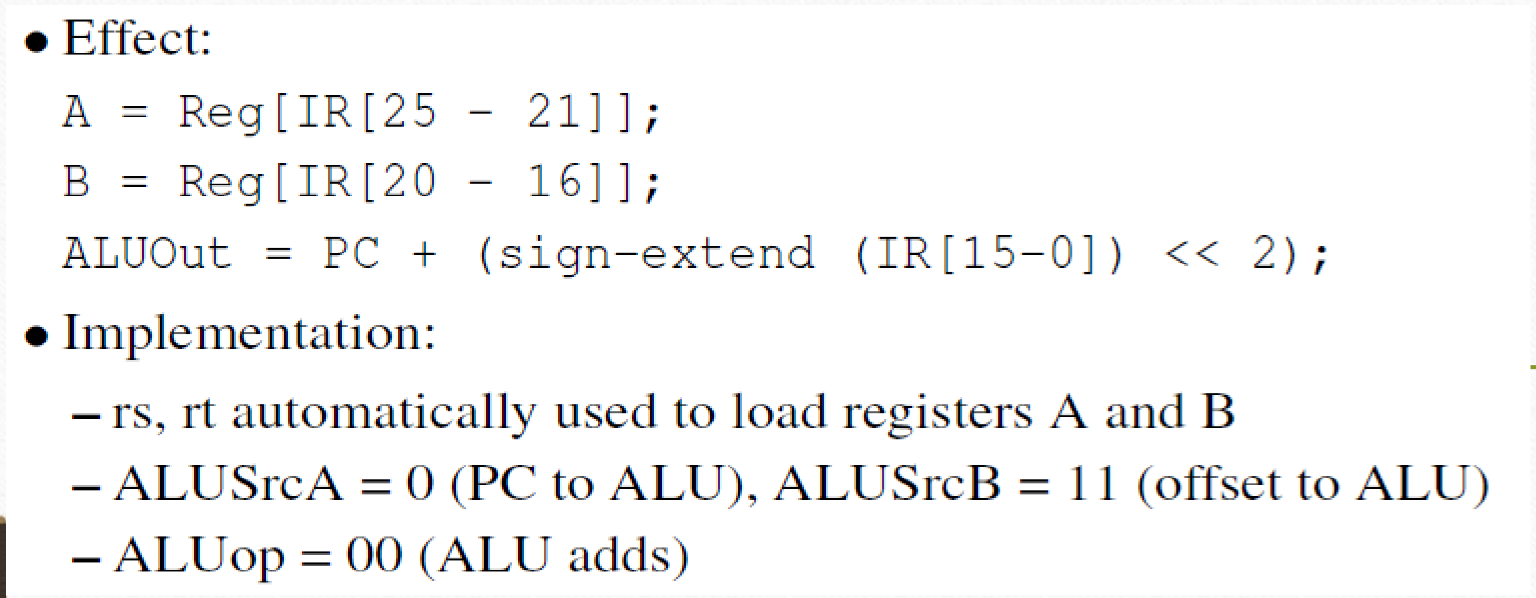
***WORK in This CYCLE: Reading from Register File.**

***Instruction Bits are sent to be "Decoded" in Control Unit**

***Write to Temporary Registers : A and B**

*What else:

STEP 02: DECODE/Register Fetch



- Effect:

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

- Implementation:
  – rs, rt automatically used to load registers A and B
  – ALUSrcA = 0 (PC to ALU), ALUSrcB = 11 (offset to ALU)
  – ALUop = 00 (ALU adds)

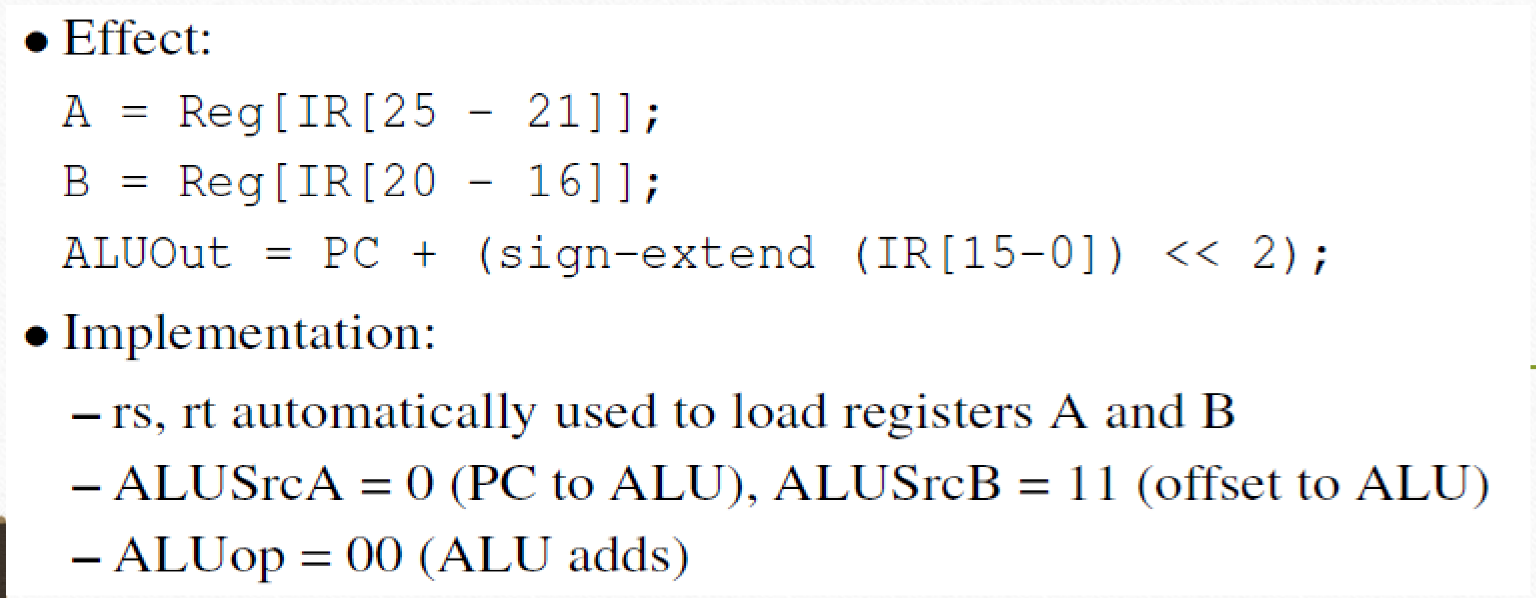
***WORK in This CYCLE: Reading from Register File.**

***Instruction Bits are sent to be "Decoded" in Control Unit**

***Write to Temporary Registers : A and B**

*USING ALU: Compute Branch Target Address
*Writing to Temp register ALUOut at the end of clock cycle.

STEP 02: DECODE/Register Fetch



- Effect:

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

- Implementation:
  - rs, rt automatically used to load registers A and B
  - ALUSrcA = 0 (PC to ALU), ALUSrcB = 11 (offset to ALU)
  - ALUop = 00 (ALU adds)

*WORK in This CYCLE: Reading from Register File.

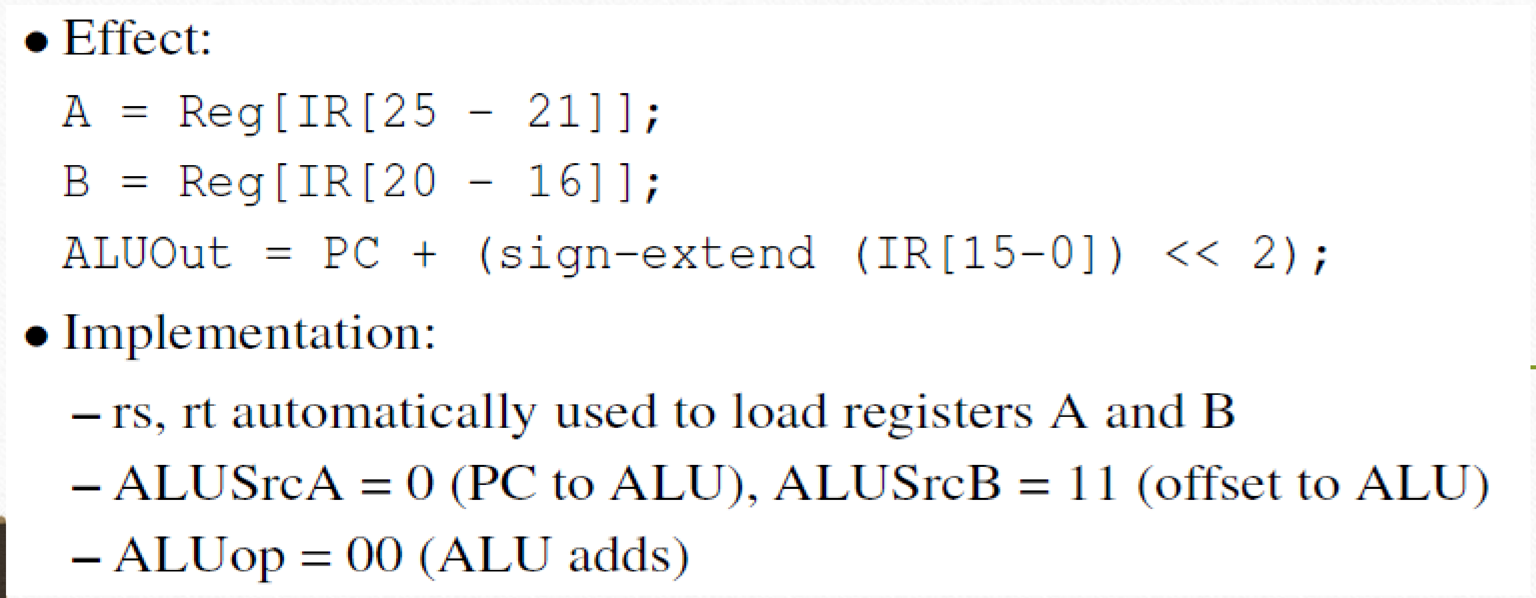
***Instruction Bits are sent to be "Decoded" in Control Unit**

***Write to Temporary Registers : A and B**

Which one takes longer?

*USING ALU: Compute Branch Target Address
*Writing to Temp register ALUOut at the end of clock cycle.

STEP 02: DECODE/Register Fetch



- Effect:

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

- Implementation:
  - rs, rt automatically used to load registers A and B
  - ALUSrcA = 0 (PC to ALU), ALUSrcB = 11 (offset to ALU)
  - ALUop = 00 (ALU adds)

*WORK in This CYCLE: Reading from Register File.

*Instruction Bits are sent to be "Decoded" in Control Unit

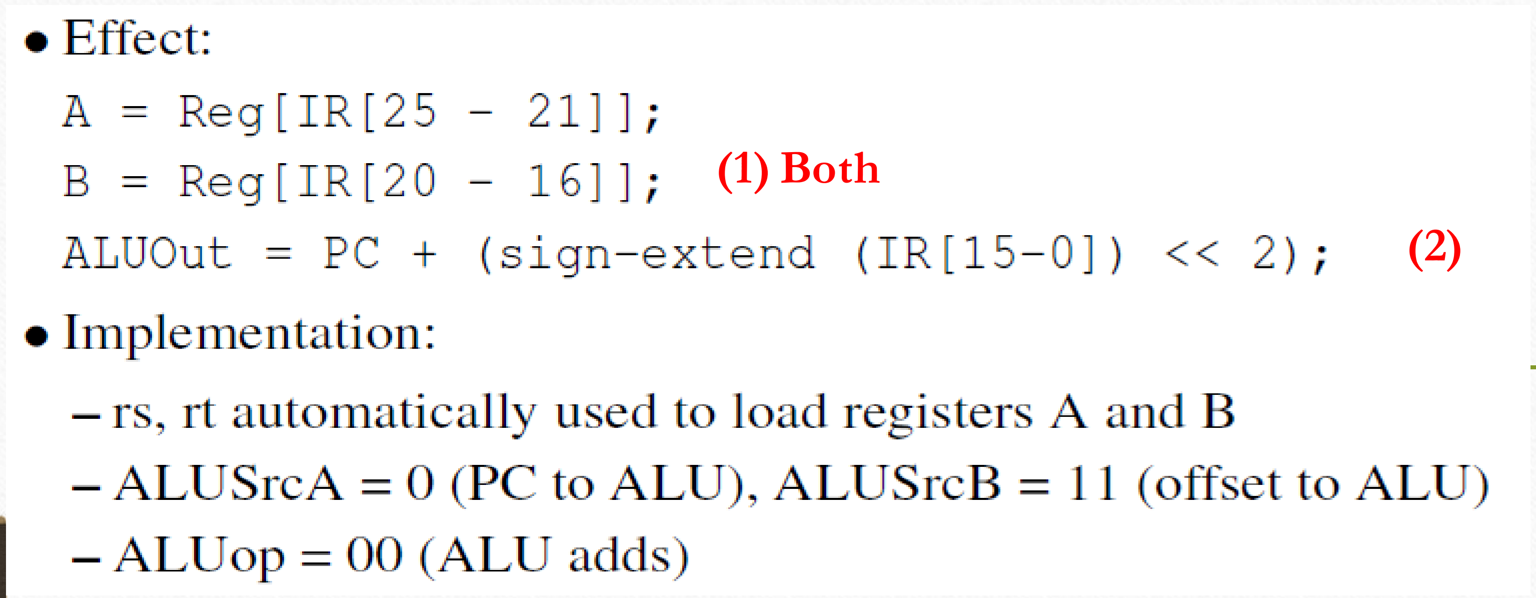
*Write to Temporary Registers : A and B

*USING ALU: Compute Branch Target Address
*Writing to Temp register ALUOut at the end of clock cycle.

Which one takes longer?

Again :ALU 100ps
Write to temp Reg: 25ps
Read from Reg File 50ps

STEP 02: DECODE/Register Fetch



- Effect:

```
A = Reg[IR[25 - 21]];
B = Reg[IR[20 - 16]];   (1) Both
ALUOut = PC + (sign-extend (IR[15-0]) << 2);   (2)
```

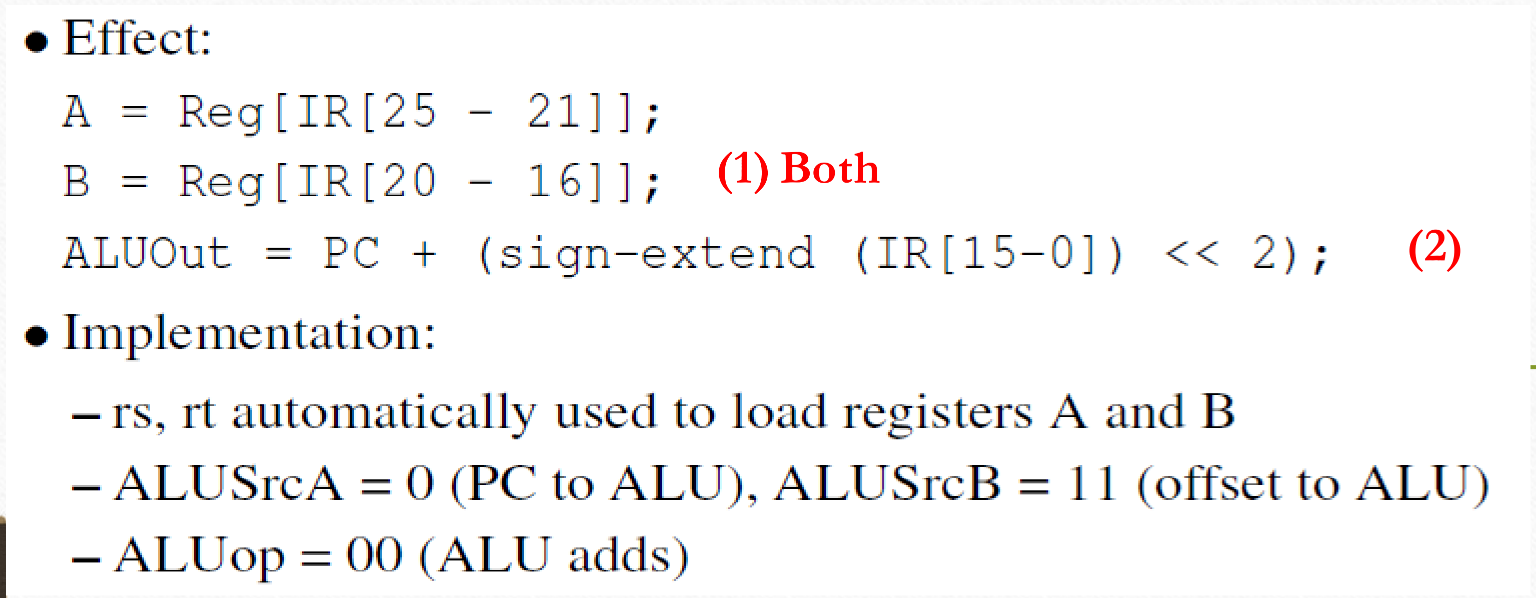
- Implementation:
  - rs, rt automatically used to load registers A and B
  - ALUSrcA = 0 (PC to ALU), ALUSrcB = 11 (offset to ALU)
  - ALUop = 00 (ALU adds)

Which one takes longer? 1 or 2

***ALU 100ps**
***Write to Temp Registers: 25ps**
***Read from Register File 50ps**

A) 1 B) 2 C) SAME TIME

STEP 02: DECODE/Register Fetch



- Effect:

```
A = Reg[IR[25 - 21]];
B = Reg[IR[20 - 16]];   (1) Both
ALUOut = PC + (sign-extend (IR[15-0]) << 2);   (2)
```

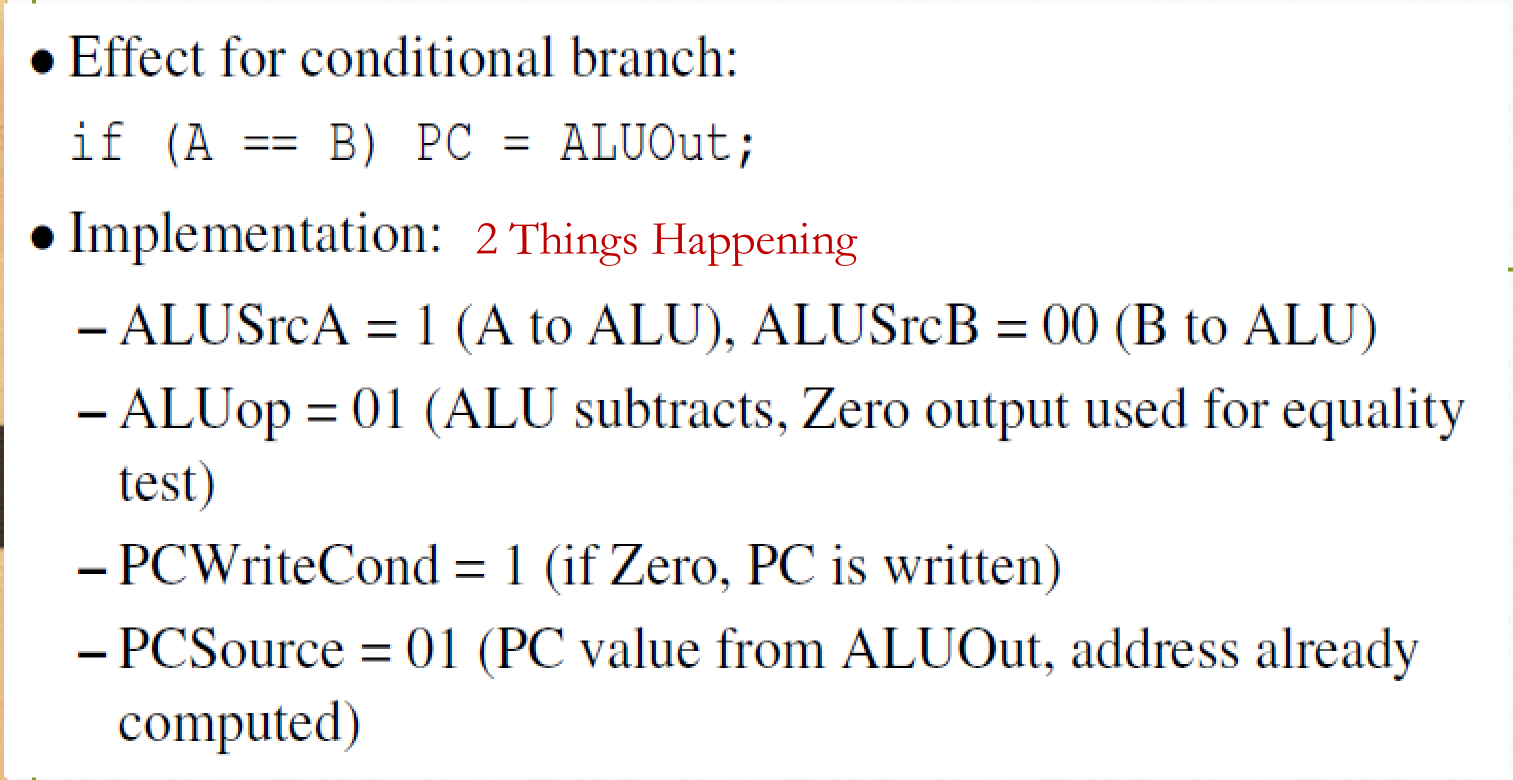
- Implementation:
  - rs, rt automatically used to load registers A and B
  - ALUSrcA = 0 (PC to ALU), ALUSrcB = 11 (offset to ALU)
  - ALUop = 00 (ALU adds)

Which one takes longer? 1 or 2

*ALU 100ps
*Write to Temp Registers: 25ps
*Read from Register File 100ps
Sign Extend 10ps , Shift Left 10ps

A) 1 B) 2 C) SAME TIME

STEP 03: BRANCH



- Effect for conditional branch:

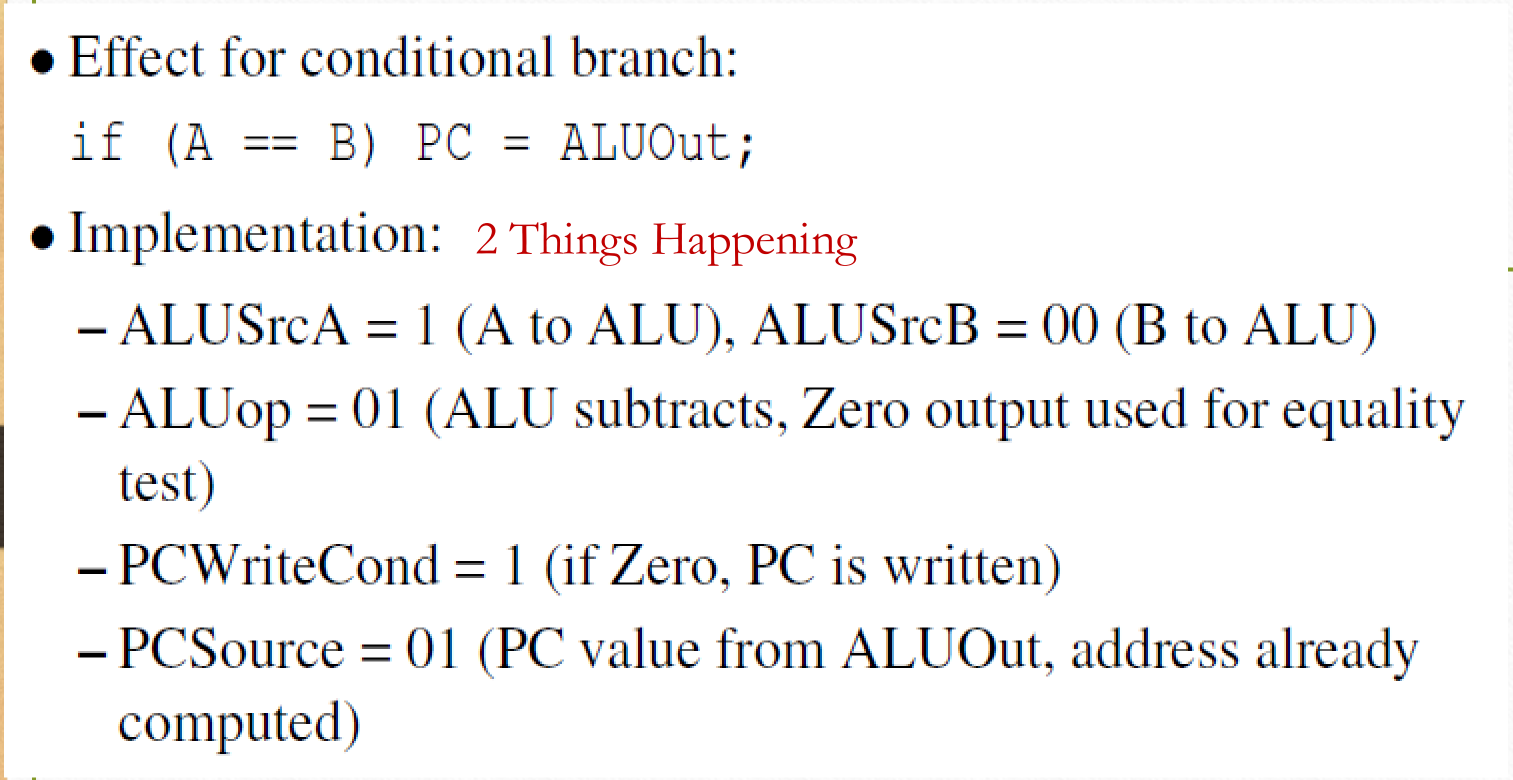
```
if (A == B) PC = ALUOut;
```

- Implementation: 2 Things Happening
  - ALUSrcA = 1 (A to ALU), ALUSrcB = 00 (B to ALU)
  - ALUop = 01 (ALU subtracts, Zero output used for equality test)
  - PCWriteCond = 1 (if Zero, PC is written)
  - PCSource = 01 (PC value from ALUOut, address already computed)

How can this work.

ALU is being used to compare $rs, $rt
But at the same time ALUOut is
Being used for its Data.

STEP 03: BRANCH



- Effect for conditional branch:

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

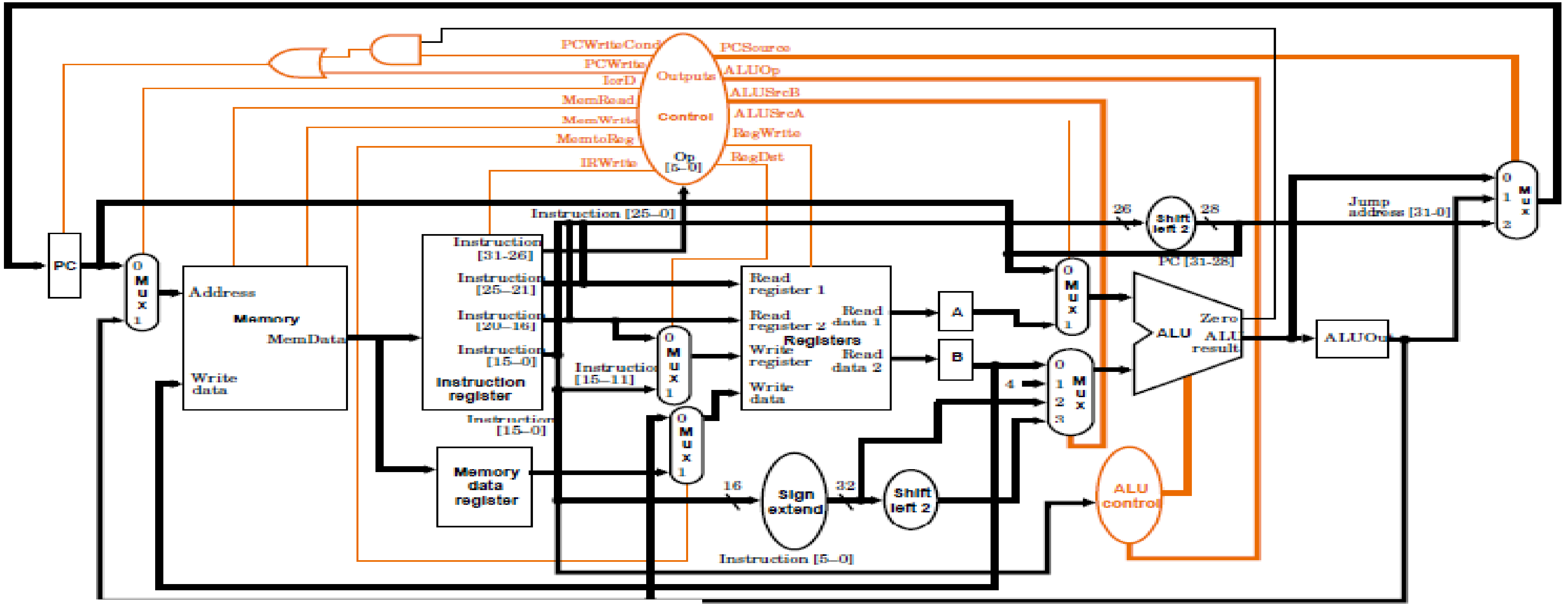
- Implementation: 2 Things Happening
  - ALUSrcA = 1 (A to ALU), ALUSrcB = 00 (B to ALU)
  - ALUop = 01 (ALU subtracts, Zero output used for equality test)
  - PCWriteCond = 1 (if Zero, PC is written)
  - PCSource = 01 (PC value from ALUOut, address already computed)

How can this work.

ALU is being used to compare $rs, $rt
But at the same time ALUOut is
Being used for its Data.

Because ALUOut is written to at the end of the clock cycle

AND Read from at the beginning of Every Clock Cycle



- Effect for unconditional jump:

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

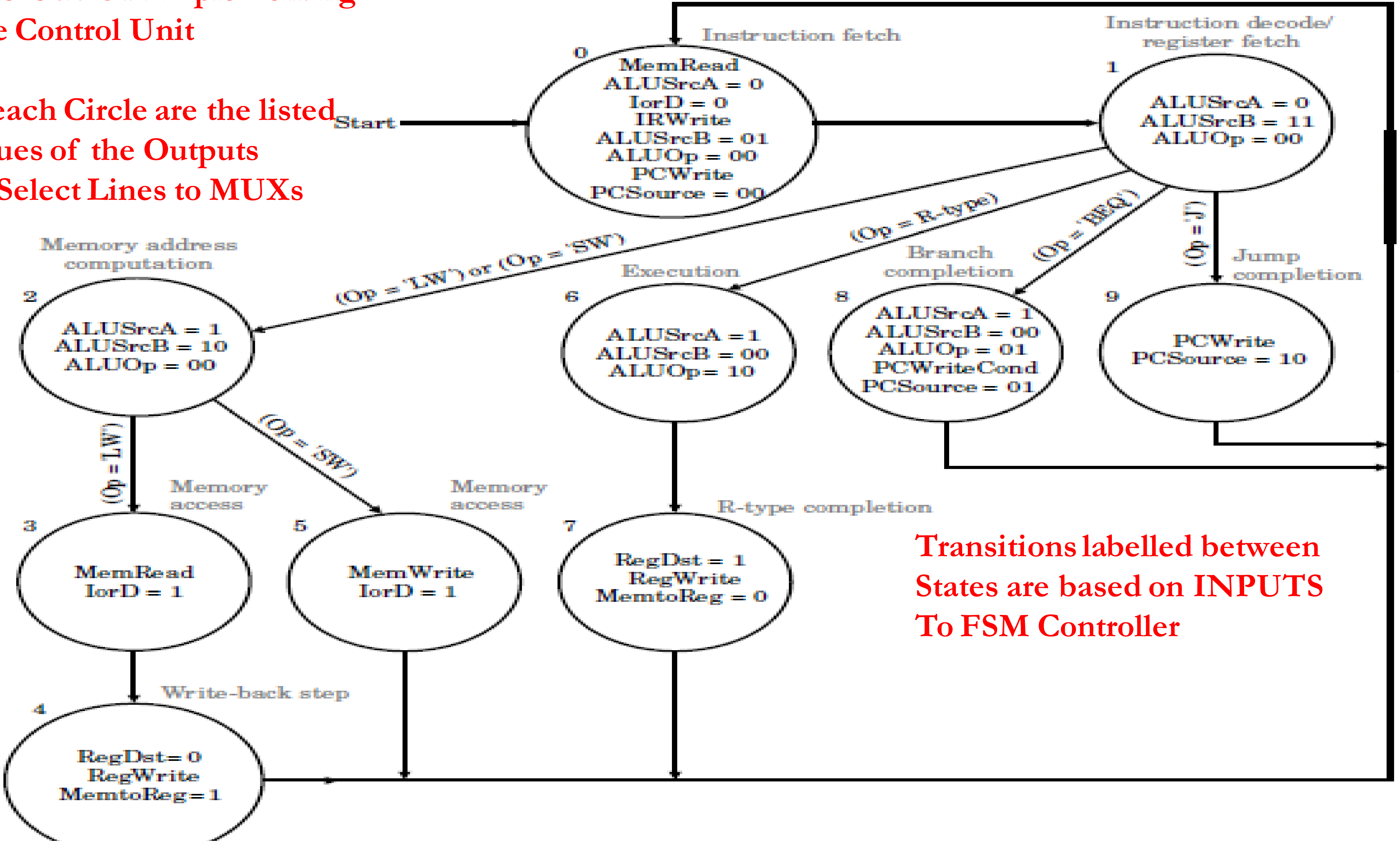
- Implementation:
  - PCWrite = 1 (PC is written)
  - PCSource = 10 (PC value as specified above)

STEP 03: Jump

Jump Instruction Takes 3 steps Because Instruction Decode Occurs in 2nd step

This is a FSM implementing
The Control Unit

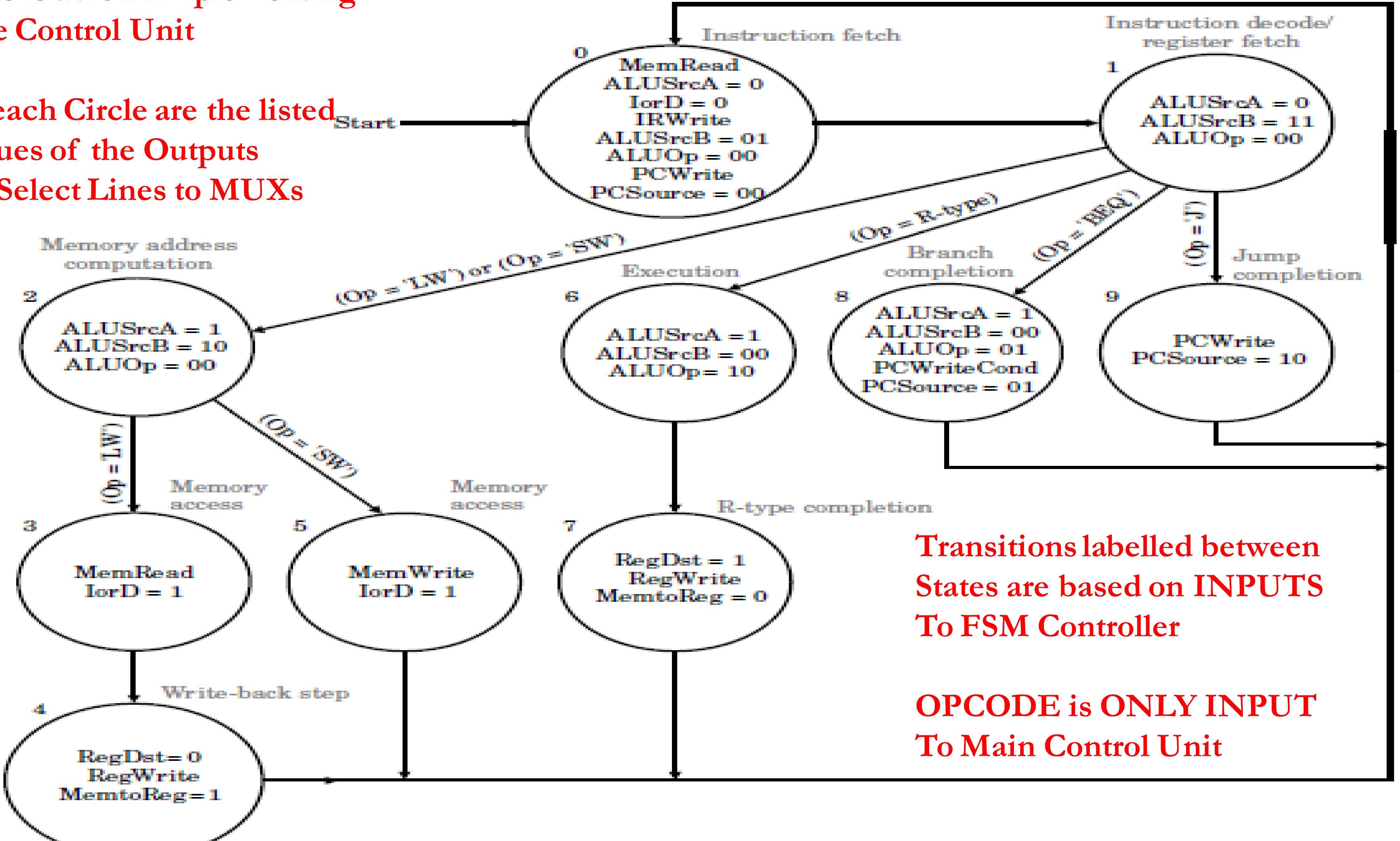
In each Circle are the listed
Values of the Outputs
Or Select Lines to MUXs



Transitions labelled between
States are based on INPUTS
To FSM Controller

This is a FSM implementing
The Control Unit

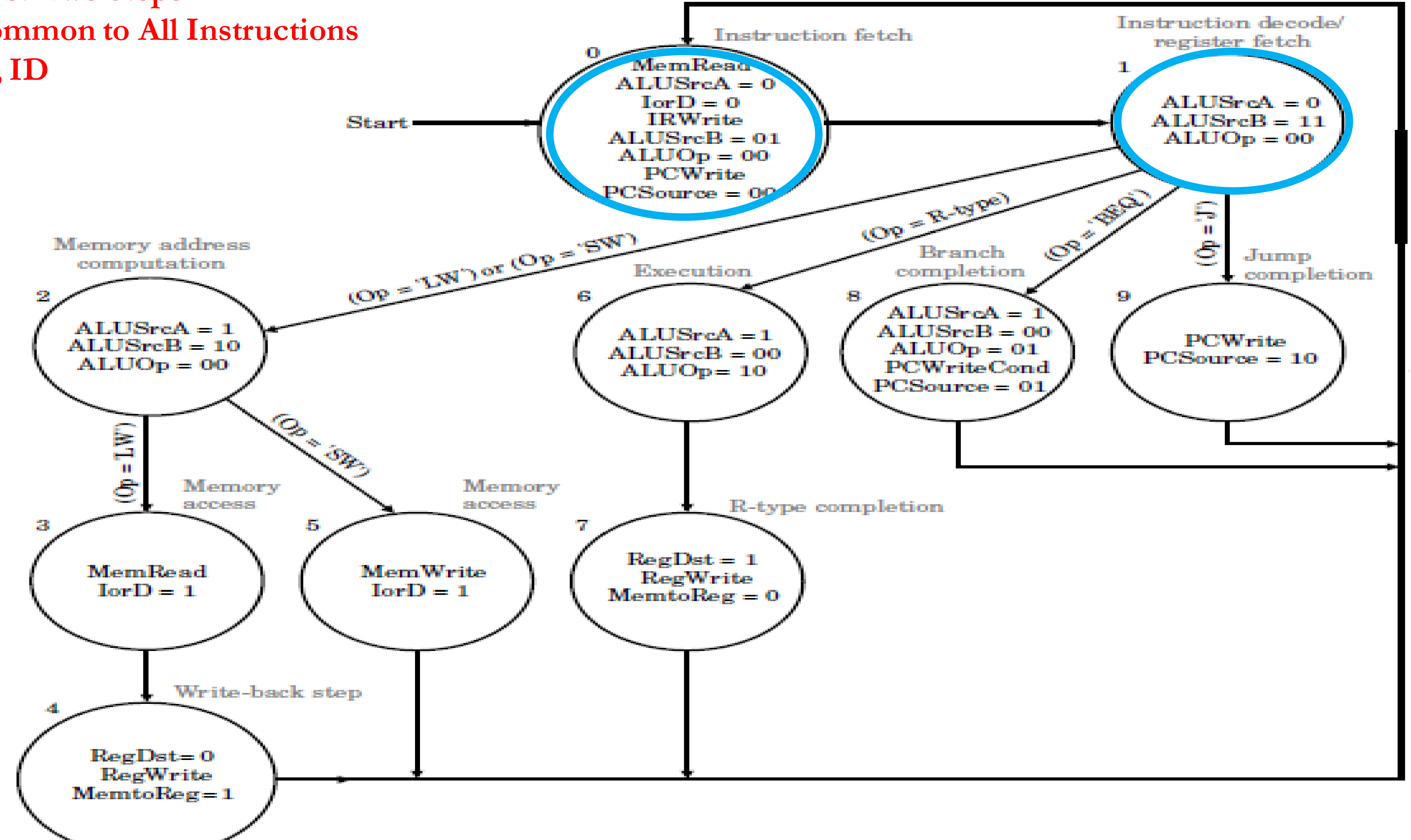
In each Circle are the listed
Values of the Outputs
Or Select Lines to MUXs



Transitions labelled between
States are based on INPUTS
To FSM Controller

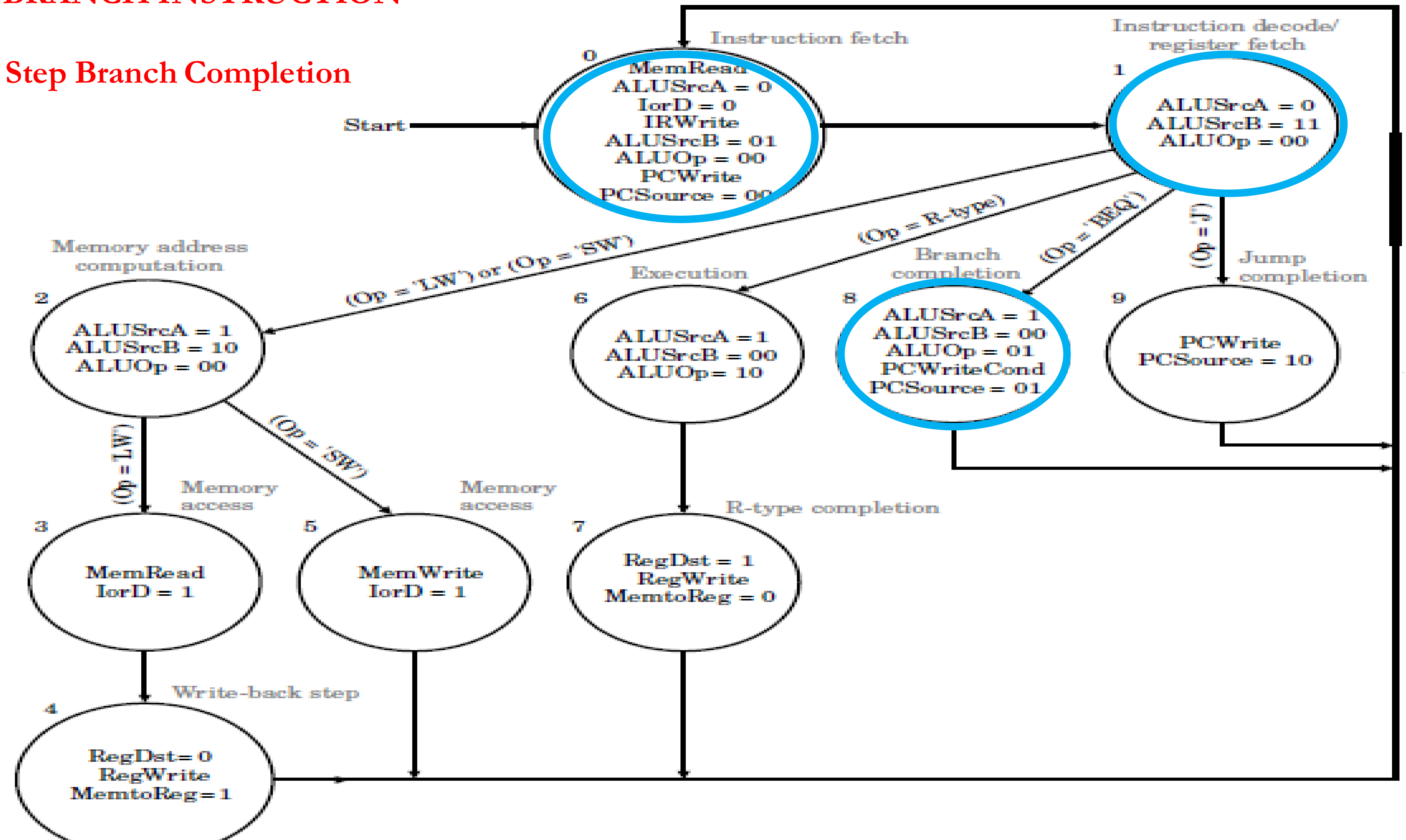
OPCODE is ONLY INPUT
To Main Control Unit

First Two Steps
Common to All Instructions
IF, ID



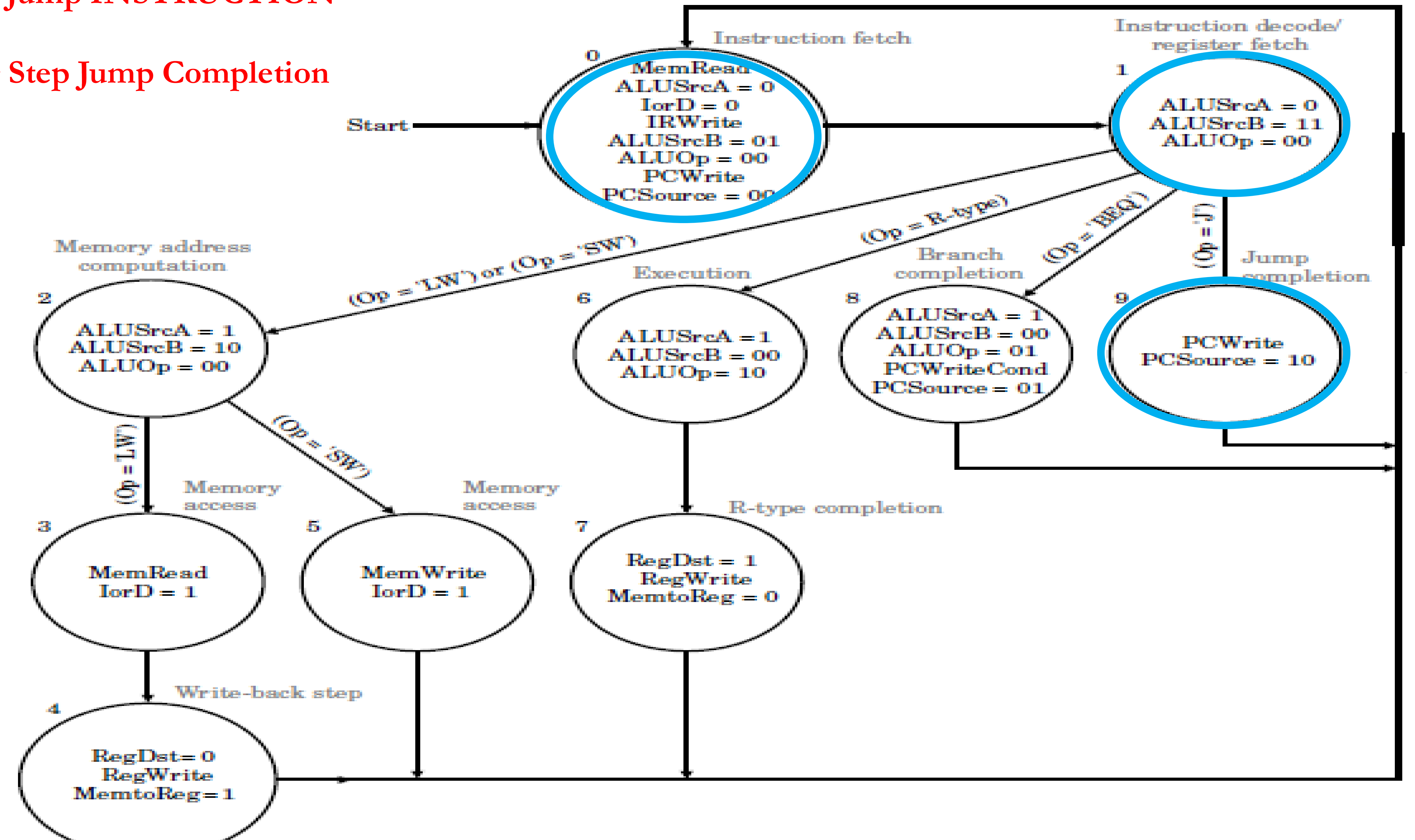
IF BRANCH INSTRUCTION

3rd Step Branch Completion



IF Jump INSTRUCTION

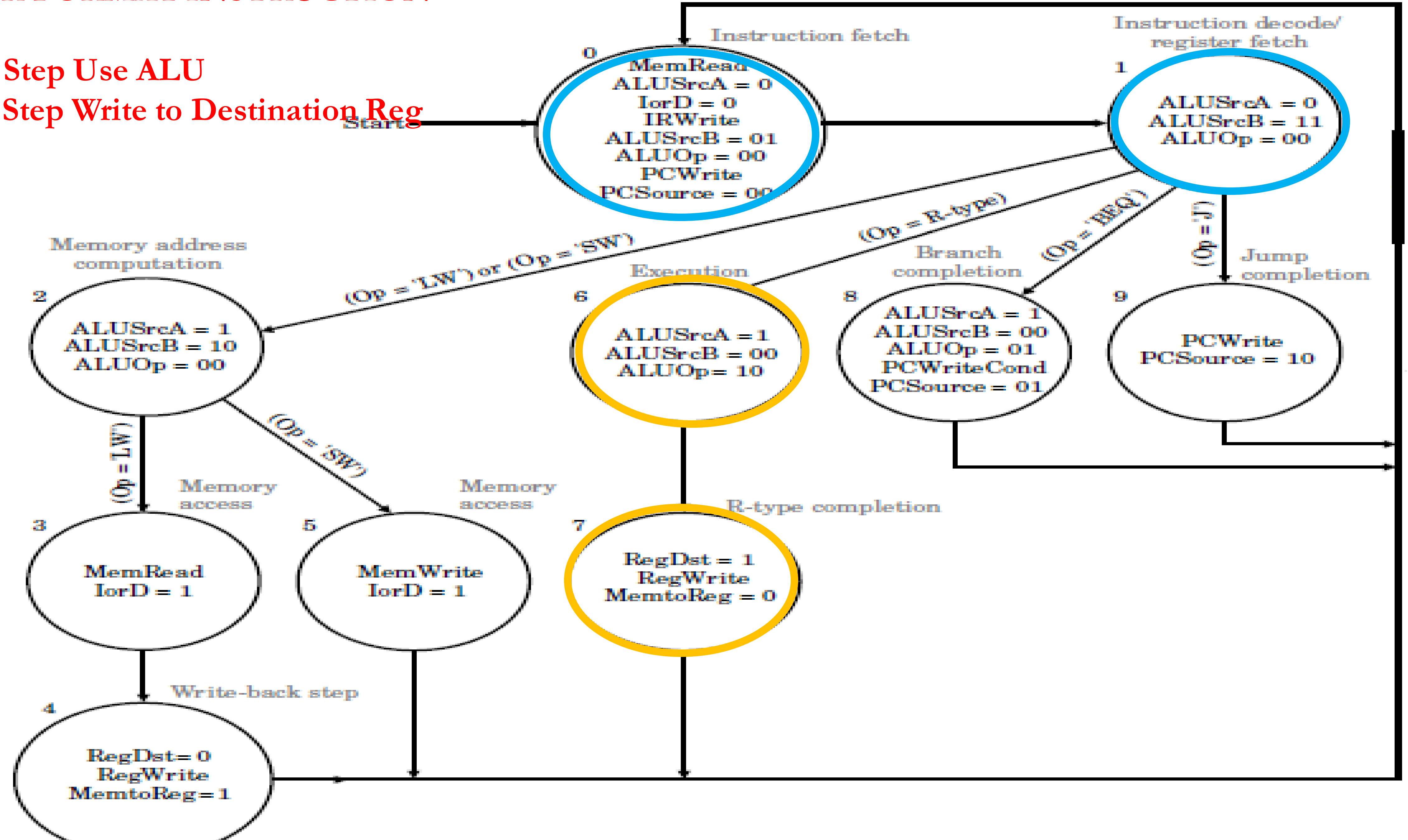
3rd Step Jump Completion



IF R FORMAT INSTRUCTION

3rd Step Use ALU

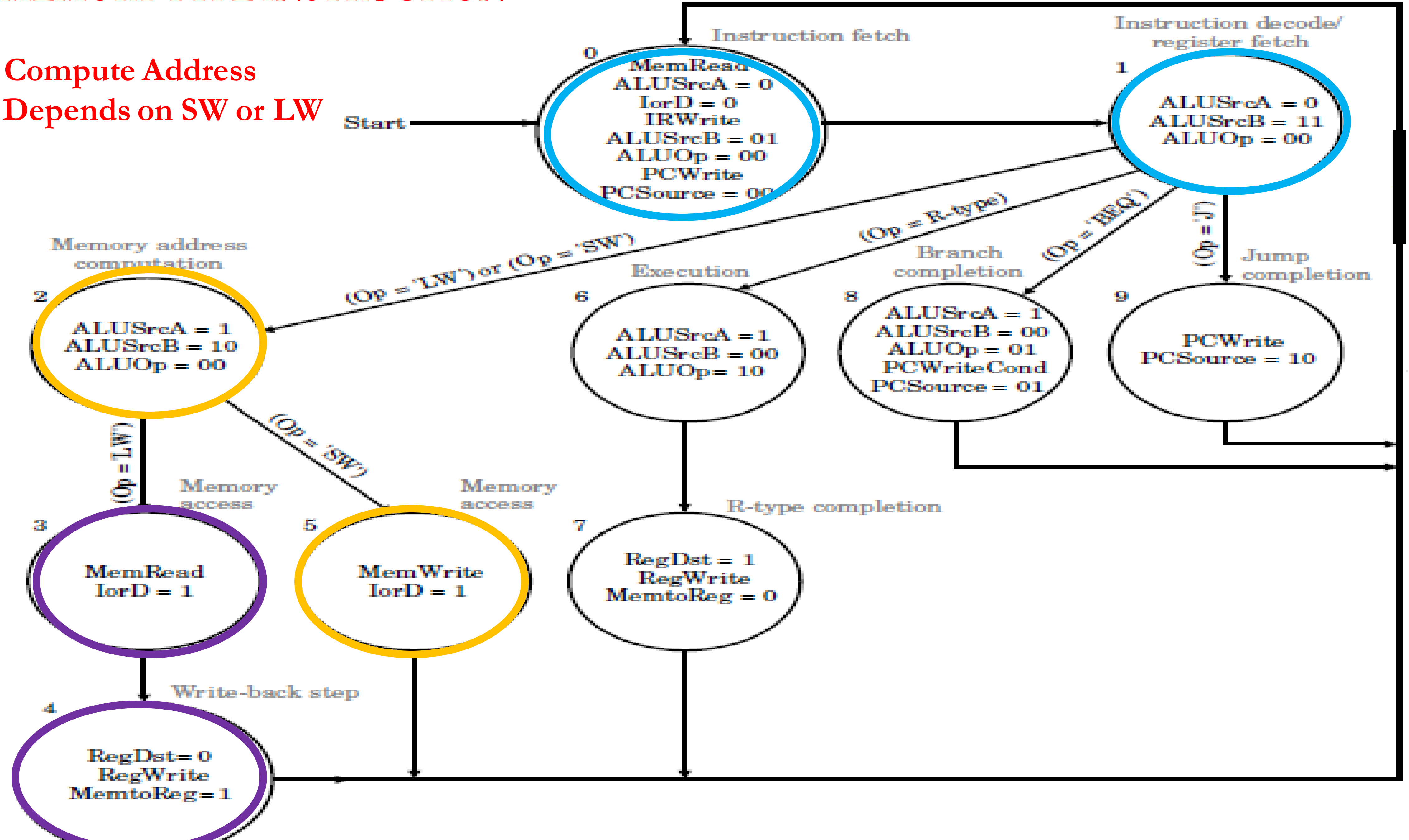
4th Step Write to Destination Reg



IF MEMORY TYPE INSTRUCTION

3rd Compute Address

4th Depends on SW or LW



Clock Cycle of Multi-cycle Datapath

- Clock Rate: 1/time of one clock cycle : 200ps : 5GHz
- Instruction Memory / Data Memory : 200ps
- Register File (read) 50ps, (write) 50ps
- ALU: 100ps
- Sign Extension: 15ps
- Shift Left (multiple by 2) : 10ps
- Writing to temporary registers at end of clock cycle: 25ps

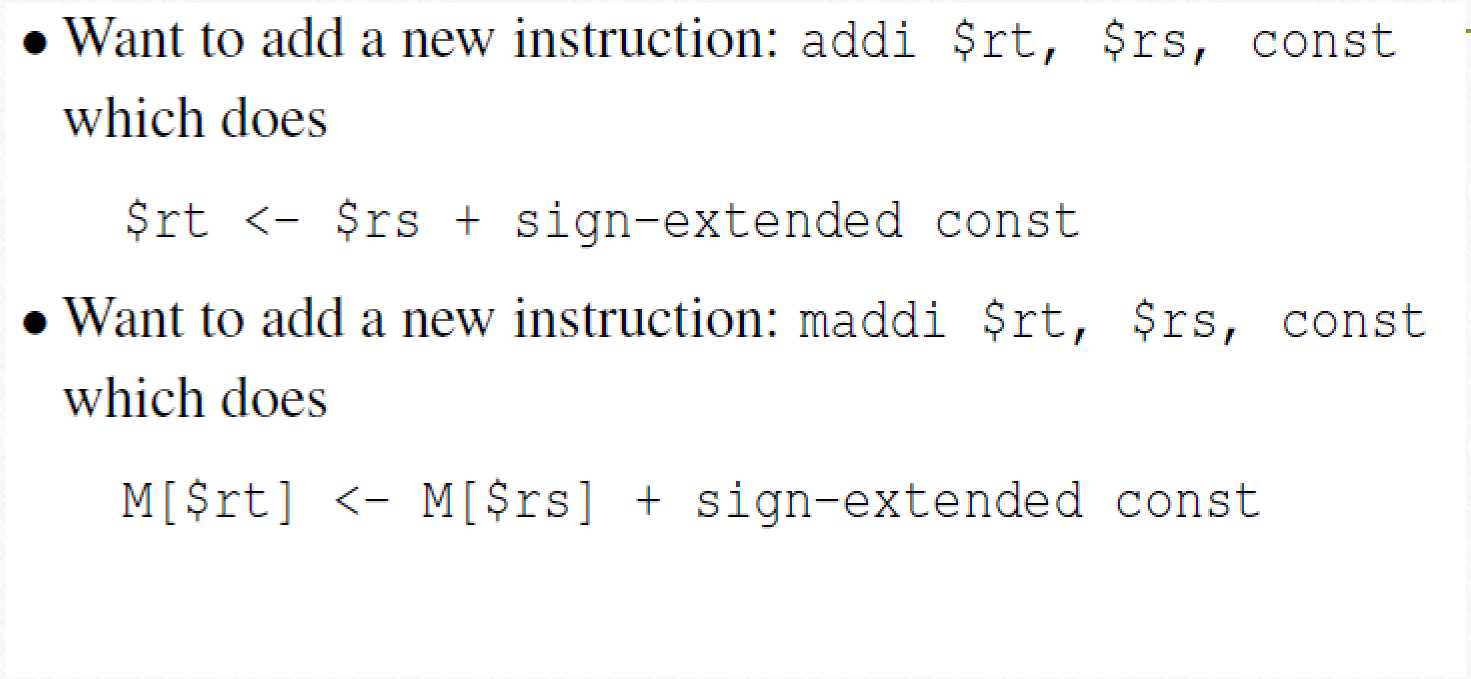
- Instruction Memory / Data Memory : 200ps
- Register File (read) 50ps, (write) 50ps
- ALU: 100ps
- Sign Extension: 15ps
- Shift Left (multiple by 2) : 10ps
- Writing to temporary registers at end of clock cycle: 25ps

What is the shortest time possible for one clock cycle Multicycle Datapath ?
A) 250ps B) 325ps C) 225ps D) 180ps E)275ps

- Instruction Memory 150ps/ Data Memory : 200ps
- Register File (read) 30ps, (write) 50ps
- ALU: 100ps
- Sign Extension: 15ps
- Shift Left (multiple by 2) : 10ps
- Writing to temporary registers at end of clock cycle: 0

What is the shortest time possible for one clock cycle Multicycle Datapath ?
A) 250ps B) 150ps C) 200ps D) 180ps E)275ps

Adding Instructions to Datapath



- Want to add a new instruction: `addi $rt, $rs, const` which does

```
$rt <- $rs + sign-extended const
```

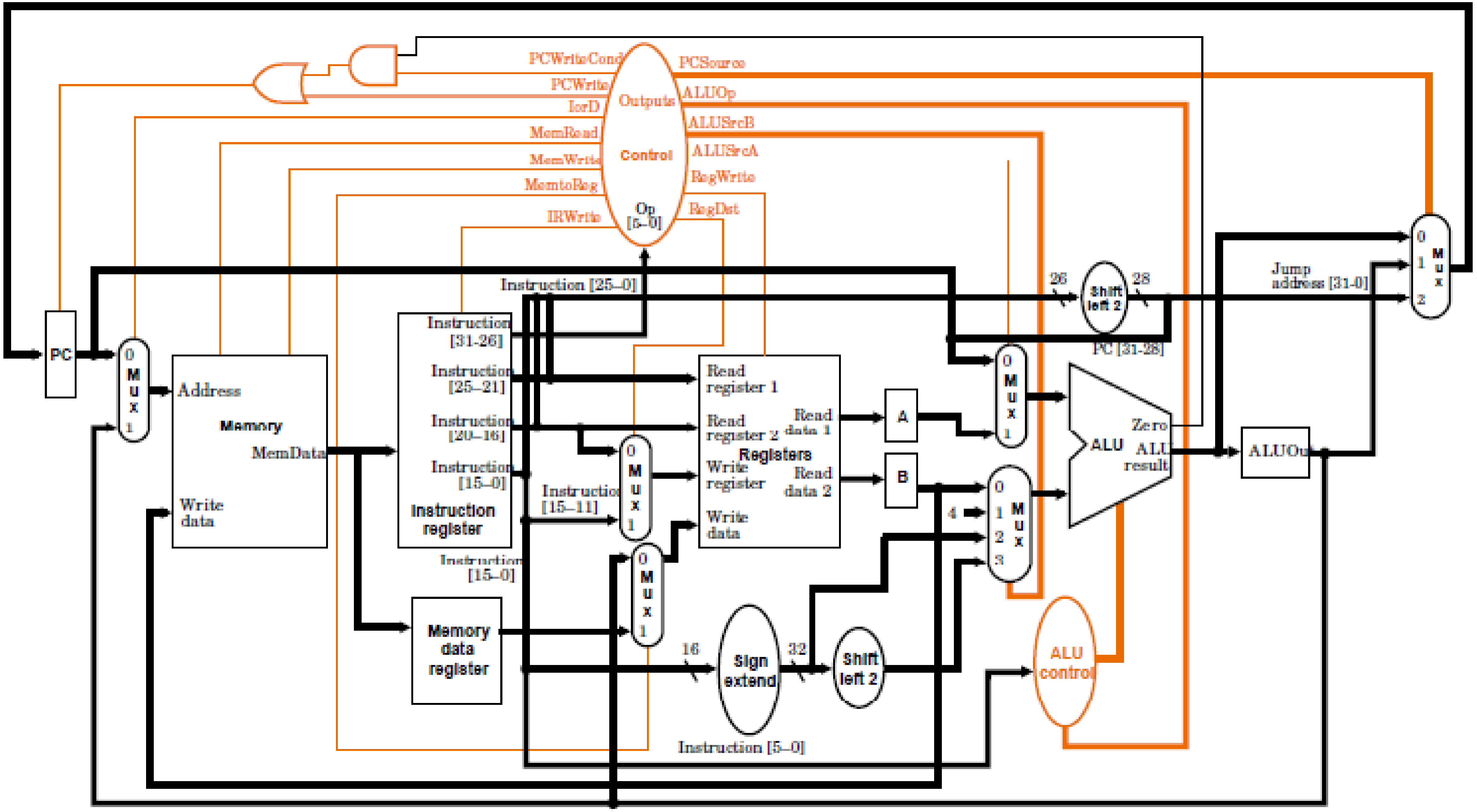
- Want to add a new instruction: `maddi $rt, $rs, const` which does

```
M[$rt] <- M[$rs] + sign-extended const
```

Adding Instructions to Datapath

- **subi $rt, $rs, 100** : For the **addi**, and **subi** instructions do I need Any Physical Modifications to existing Multicycle Datapath
- A)YES
- B) NO

MULTICYCLE DATAPATH



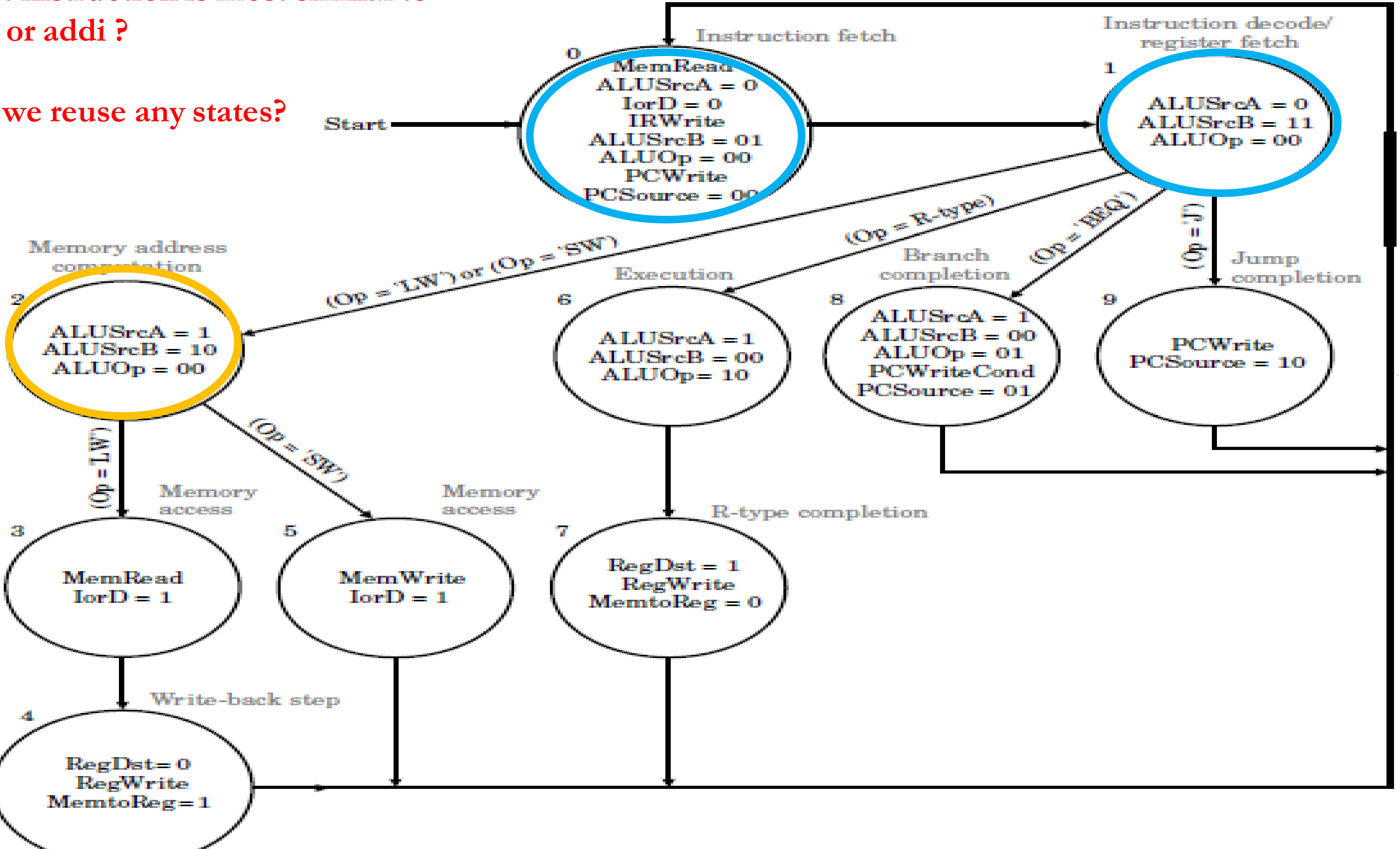
Adding Instructions to Datapath

- Subi $rt, $rs, 100 : For the Addi, and Subi instructions do I need Any Physical Modifications to existing Multi-Cycle Datapath
- A)YES
- B) NO

***FINITE STATE CONTROL: Need Changes**

What instruction is most similar to subi or addi ?

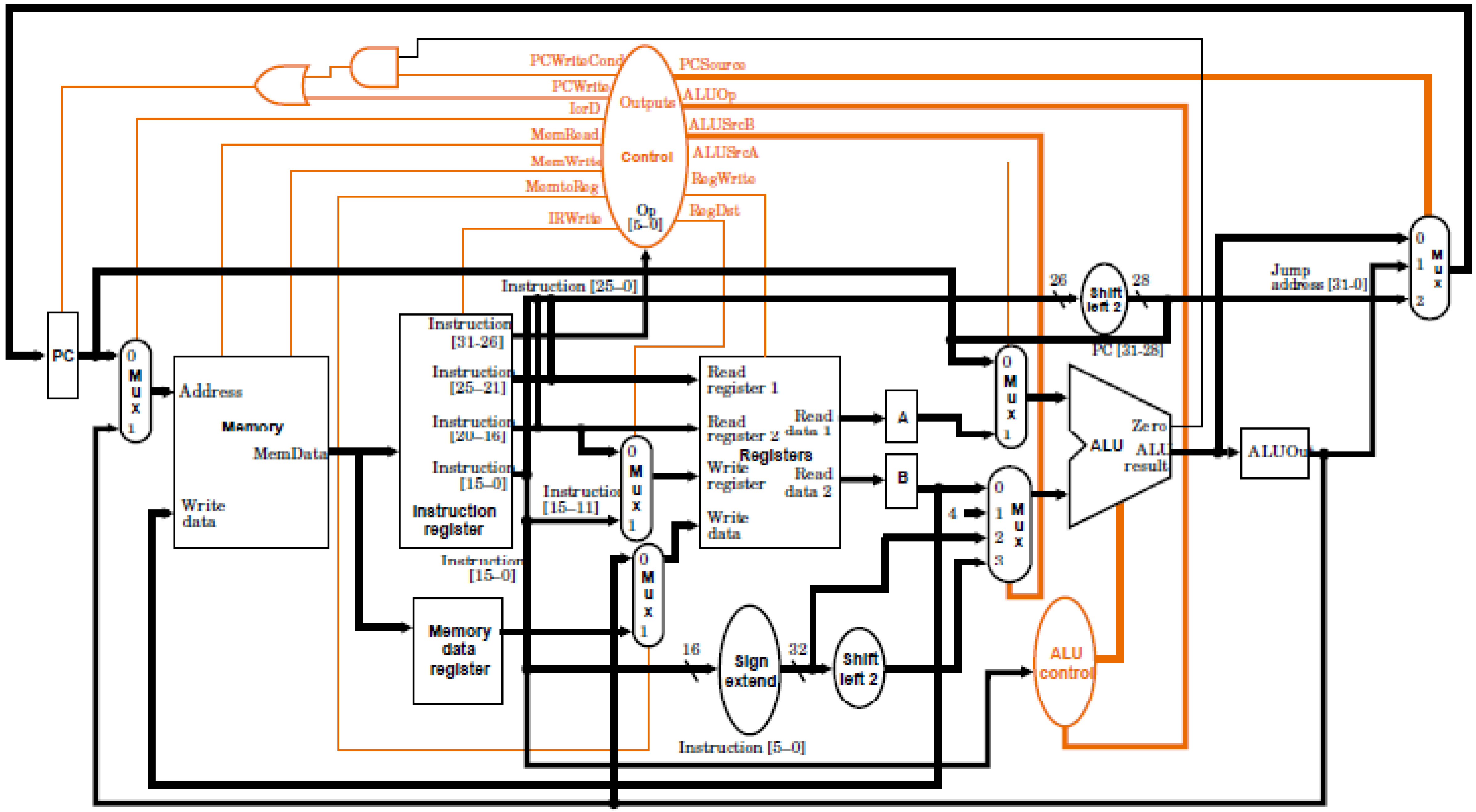
Can we reuse any states?



Adding Instructions to Datapath

- **Jump Register**: Jump to an address specified by $rs (exactly that register)
- **PC ← Address in $rs**
- Do I need to Modify Multicycle datapath
- A)YES
- B) NO

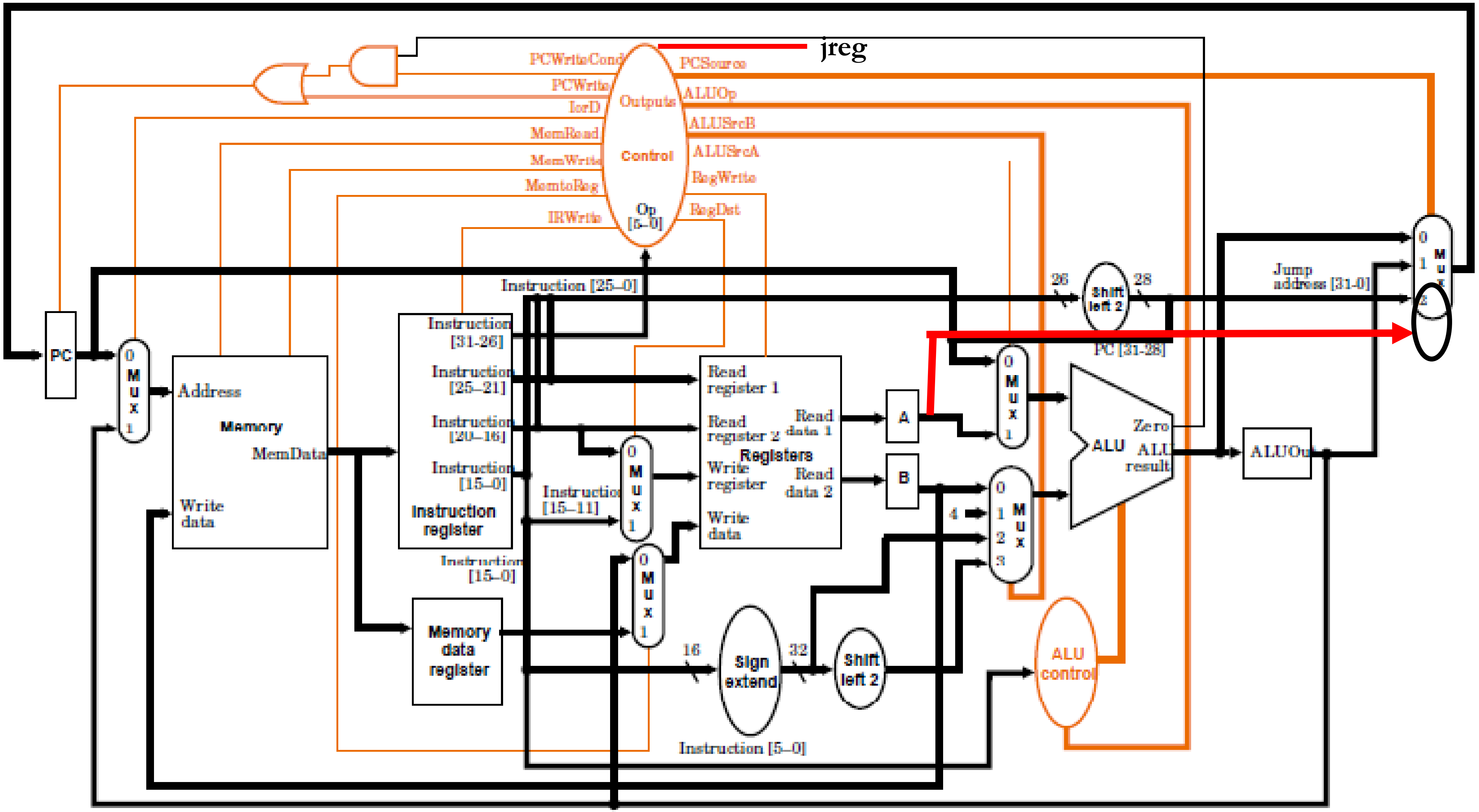
MULTICYCLE DATAPATH

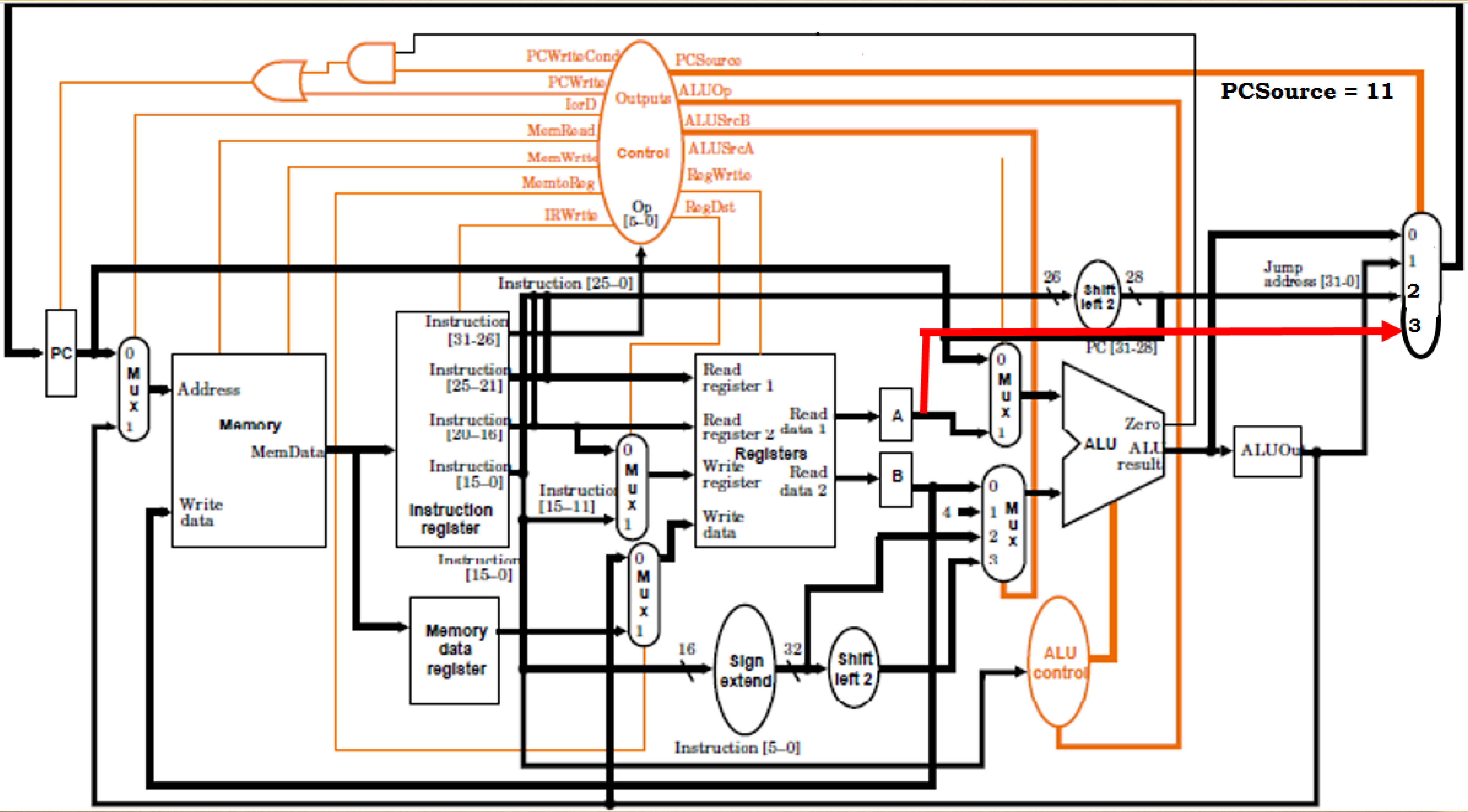


Adding Instructions to Datapath

- **Jump Register**: Jump to an address specified by $rs (exactly that register)
- **PC ← Address in $rs**
- Do I need to Modify Multicycle datapath
- A)YES
- B) NO

MULTICYCLE DATAPATH



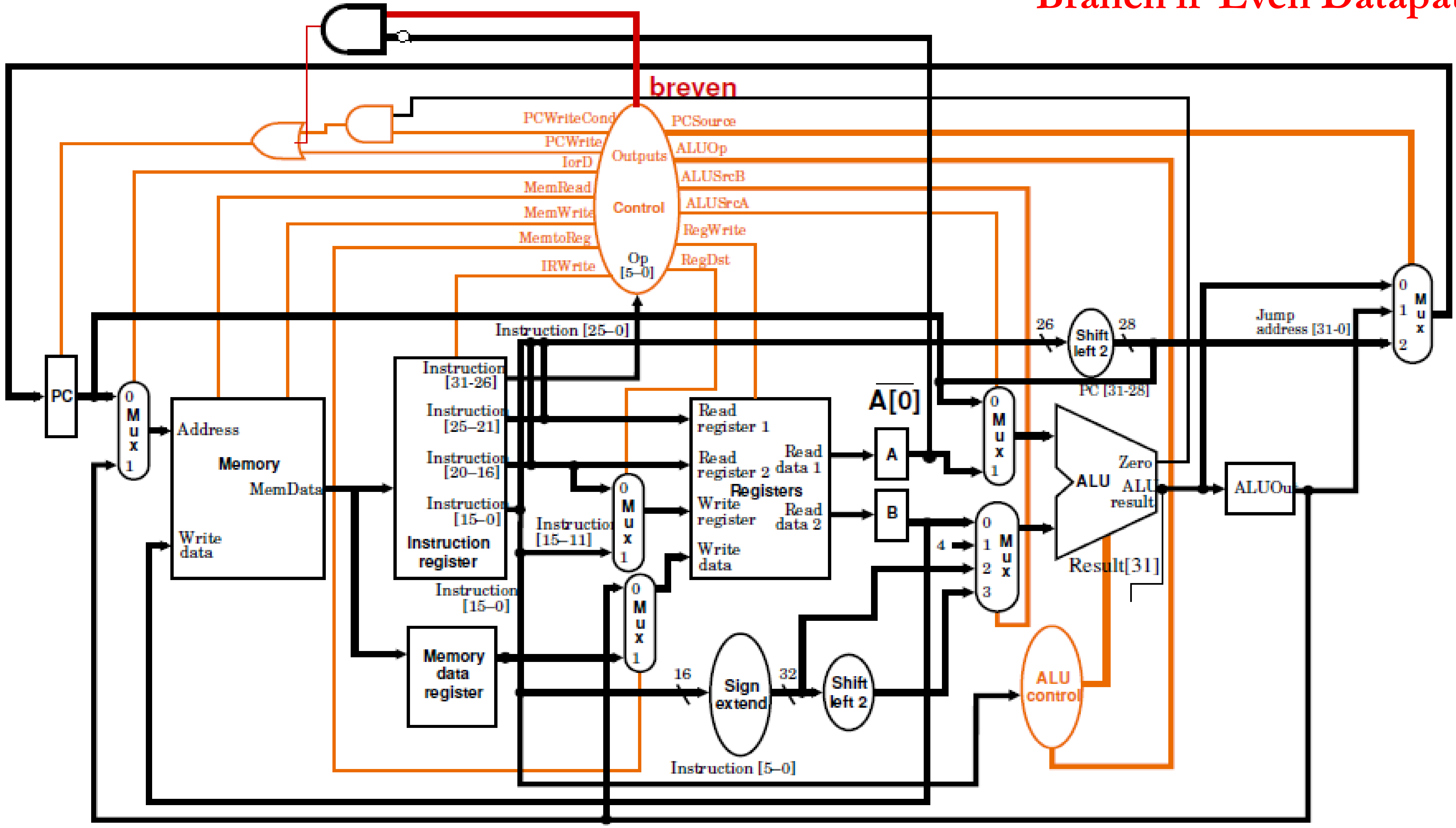


PCSource = 11
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op [5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
Address
Memory
MemData
Write data
Instruction [31-26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–0]
Instruction register
Instruction [25–0]
Instruction [15–11]
Instruction [15–0]
Memory data register
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
A
B
26
Shift left 2
28
PC [31-28]
Jump address [31-0]
ALU
Zero
ALU result
ALUOut
4
16
Sign extend
32
Shift left 2
ALU control
Instruction [5–0]
0 1 2 3

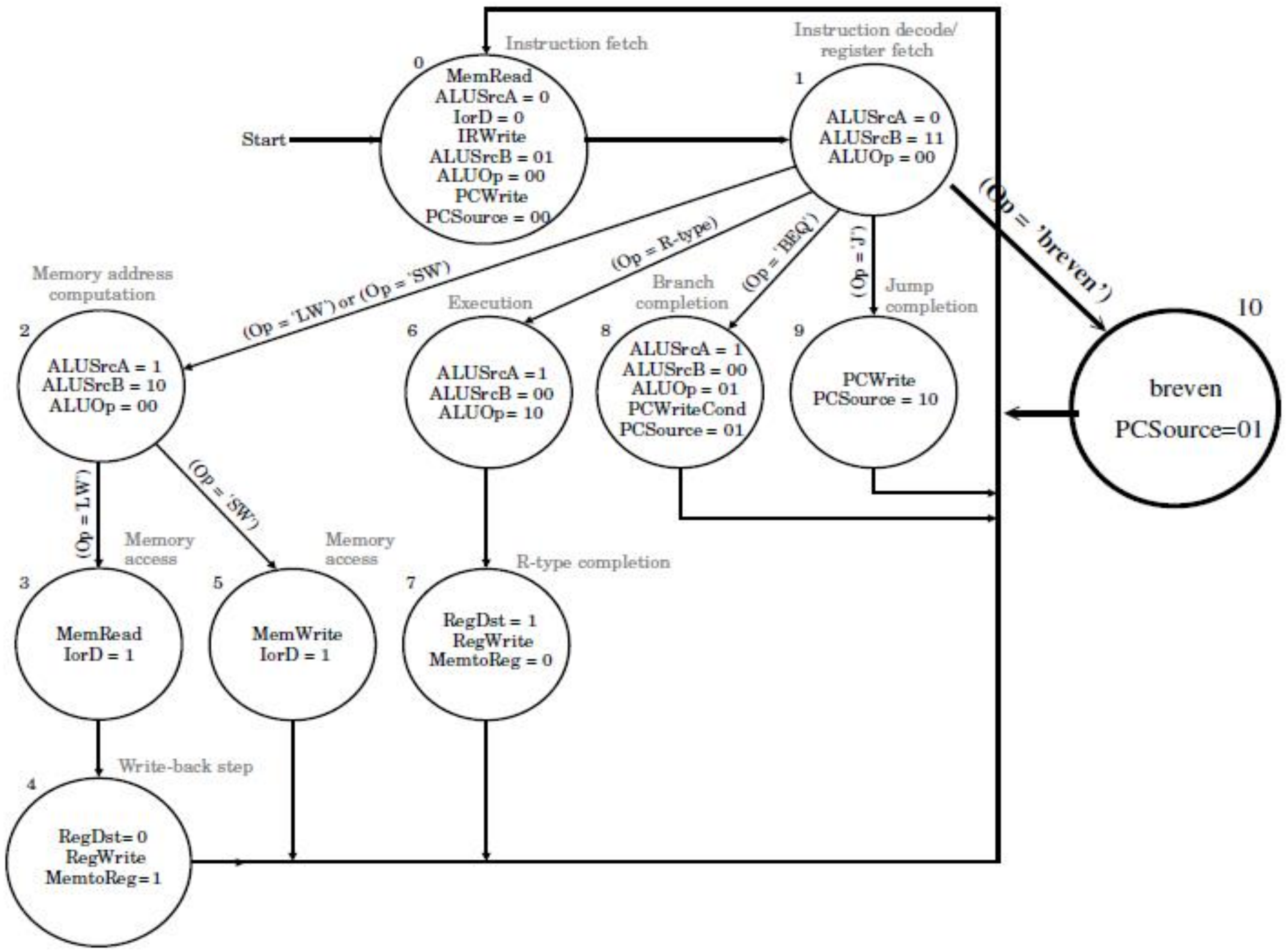
Branch if Even: **Breven** ☺

- We want to add a new I-Format Instruction **breven** (branch even) to the multicycle computer.
- The command sets the PC to: PC+4+ four times the immediate field (only if $s1 is an even number)
- **1000: breven $s1, 100 : breven $rs, Offset**
- effect: PC: PC + 4 + 400 = 1000 + 4 + 400 = 1404 if $s1 is an even number.
- $rt is not used in this I-format Instruction

Branch if Even Datapath



Branch If Even Modify the Control Unit FSM



Last Example: ***maddi***

M[$rt] ← M[$rs] + Sign Extended Offset

- **Write to Memory** at $rt address:
  - **Data**: (Value From Memory[$rs] + Offset)

Last Example: *maddi*

M[$rt] 🡐 M[$rs] + Sign Extended Offset

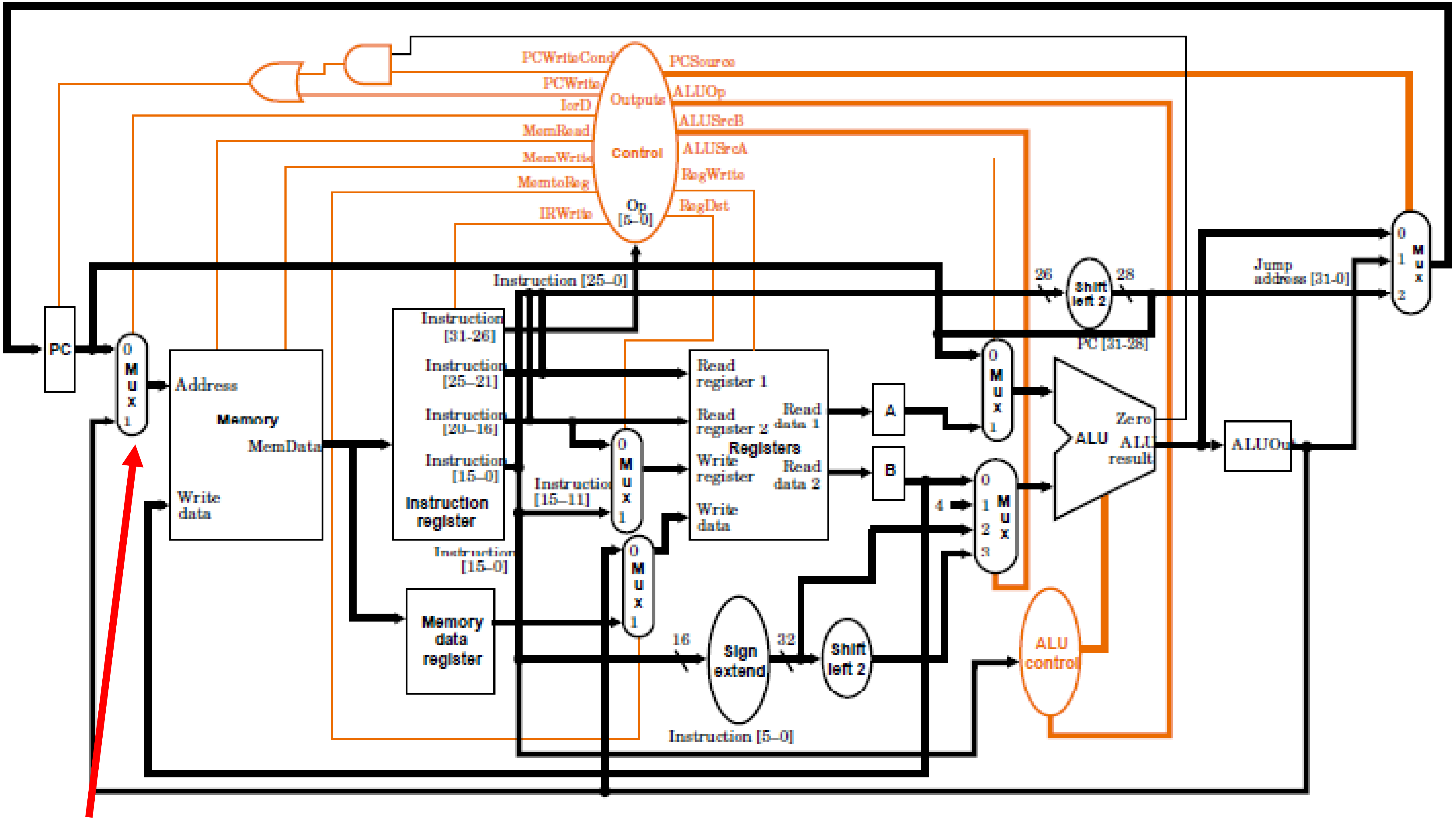
- **Write to Memory** at $rt address:
  - **Data**: (Value From Memory[$rs] + Offset)
- We Begin at Third Step? (after IF, and Decode)
  - Go To Memory: With what address?

Last Example: ***maddi***

M[$rt] ← M[$rs] + Sign Extended Offset

- Third Step? (after IF, and Decode)
  - **Use Registers A and B to provide Address into Memory: Extend MUX**

MULTICYCLE DATAPATH

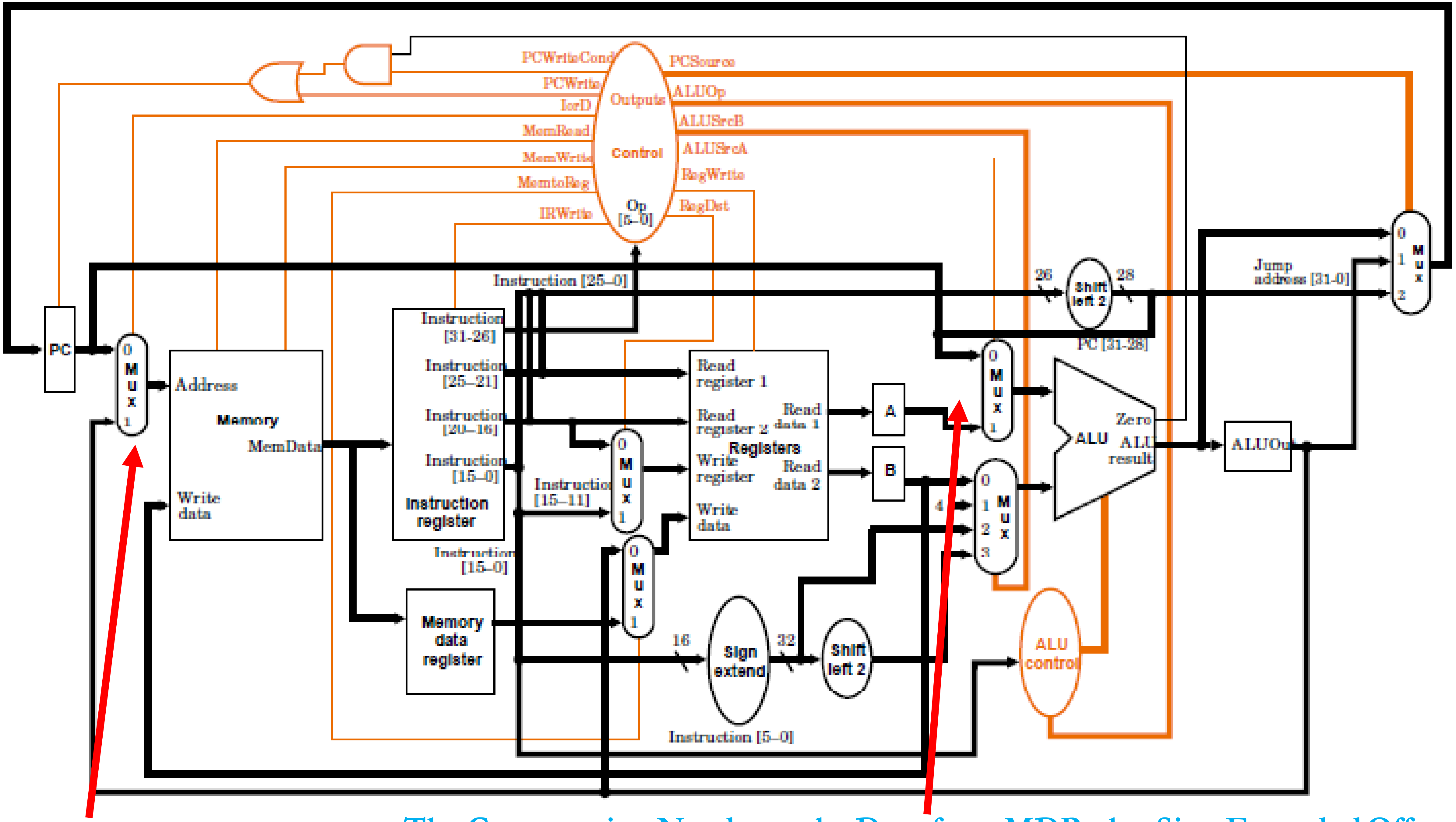


Extend this MUX, add address options coming from Registers A and B

Last Example: ***maddi***

M[$rt] ← M[$rs] + Sign Extended Offset

- Third Step? (after IF, and Decode)
  - **Use Registers A and B to provide Address into Memory: Extend MUX**
- Next:
  - **Store M[$rs] into MDR**
  - **MDR + Sign Extended Offset → ALUOut * Therefore ALU input A must allow MDR as data**



The Computation Needs to take Data from MDR plus Sign Extended Offset

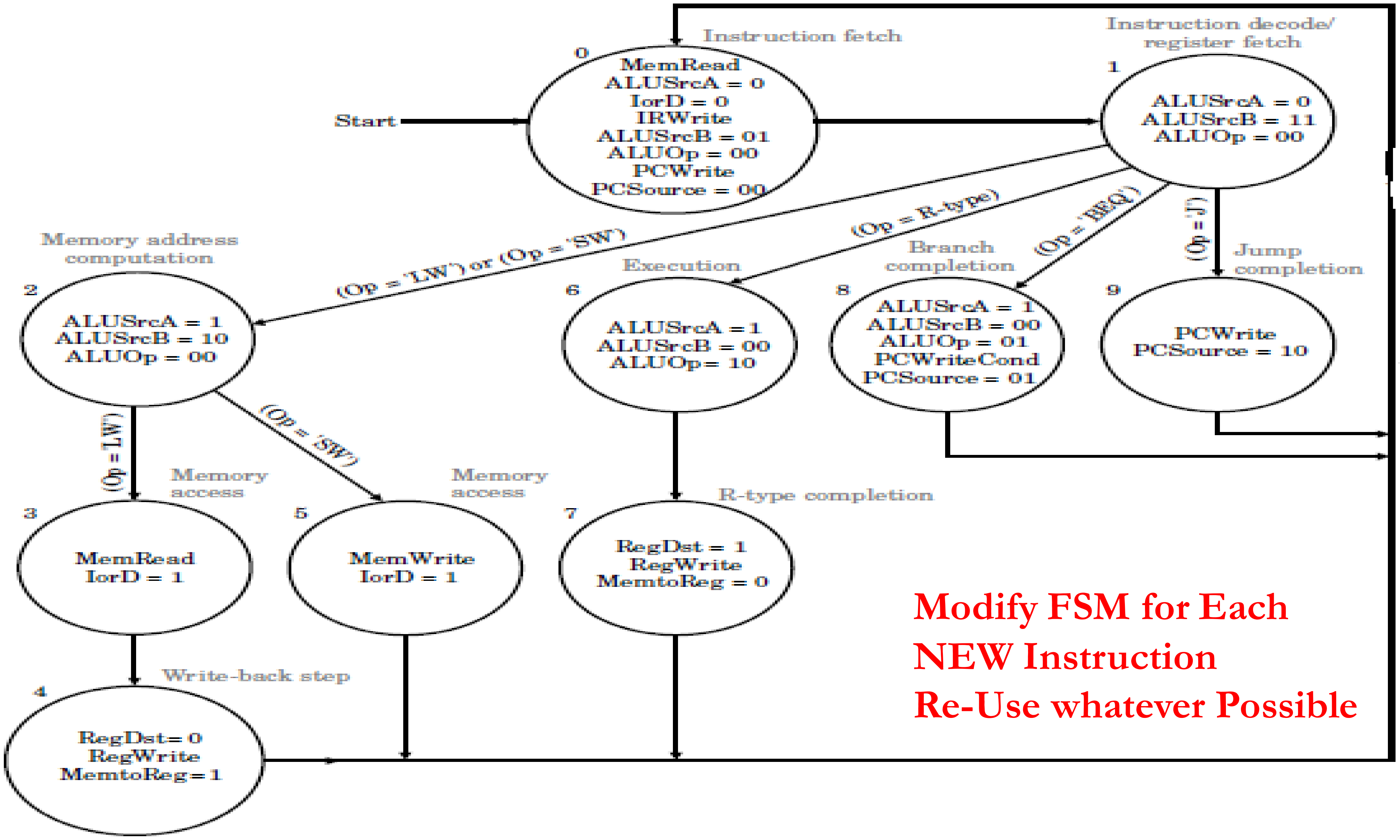
Last Example: *maddi*
M[$rt] ← M[$rs] + Sign Extended Offset

- Third Step? (after IF, and Decode)
  - **Use Registers A and B to provide Address into Memory: Extend MUX**
- Next:
  - **Store M[$rs] into MDR**
  - **MDR + Sign Extended Offset → ALUOut**
  - **Write Back to Memory : M[$rt]**

Last Example: *maddi*

M[$rt] ← M[$rs] + Sign Extended Offset

- Third Step? (after IF, and Decode)
  - **Use Registers A and B to provide Address into Memory: Extend MUX**
- Next:
  - **Store M[$rs] into MDR**
  - **MDR + Sign Extended Offset → ALUOut**
  - **Write Back to Memory : M[$rt] *Where is the Write Data Source : ALUOut**
  - **Also MUST UPDATE FSM: For this new Instruction *maddi***



Modify FSM for Each NEW Instruction
Re-Use whatever Possible

Multicycle Control Performance

- Loads take 5 cycles, stores and R-format take 4 cycles, branches and jumps take 3 cycles

  How to measure performance?

- Experiments show that gcc (GNU C compiler) uses on average 22% loads, 11% stores, 49% R-format, 16% branches, 2% jumps

- Average number of cycles per instruction (CPI) is:

$$0.22 \times 5 + 0.11 \times 4 + 0.49 \times 4 + 0.16 \times 3 + 0.02 \times 3 = 4.04$$

IF we had (at best) a 200ps clock cycle time:
Approximately 800ps CPI

Multicycle Control Performance

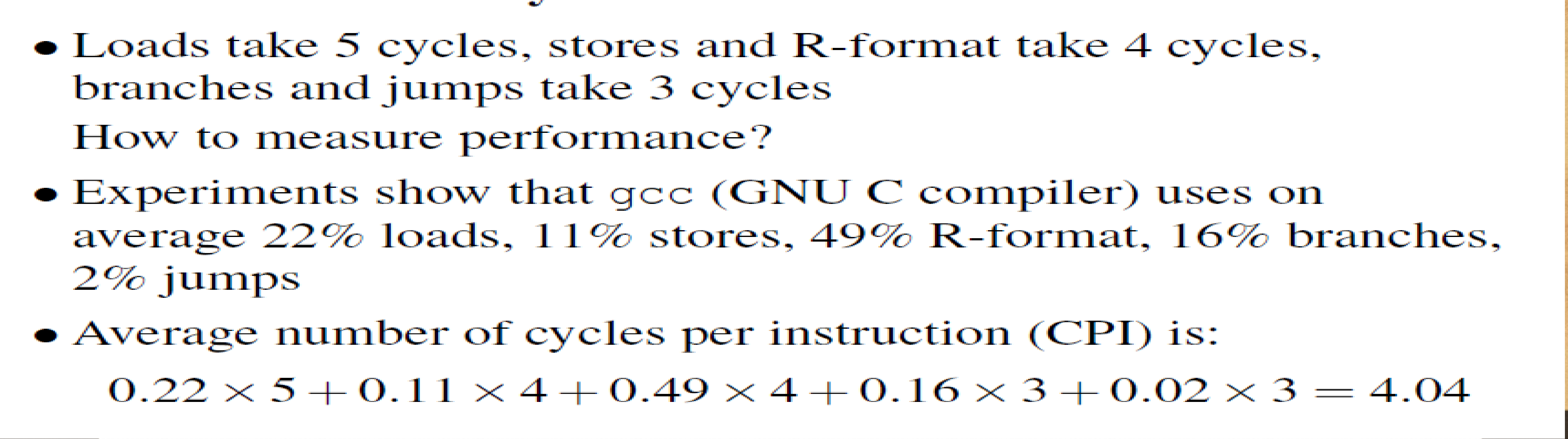
- Loads take 5 cycles, stores and R-format take 4 cycles, branches and jumps take 3 cycles

  How to measure performance?
- Experiments show that gcc (GNU C compiler) uses on average 22% loads, 11% stores, 49% R-format, 16% branches, 2% jumps
- Average number of cycles per instruction (CPI) is:

$$0.22 \times 5 + 0.11 \times 4 + 0.49 \times 4 + 0.16 \times 3 + 0.02 \times 3 = 4.04$$

800ps CPI

Compare this to Single Cycle: 600ps for longest instruction/ Some instructions sitting Around waiting…But Single Cycle was still better



- Loads take 5 cycles, stores and R-format take 4 cycles, branches and jumps take 3 cycles

  How to measure performance?
- Experiments show that gcc (GNU C compiler) uses on average 22% loads, 11% stores, 49% R-format, 16% branches, 2% jumps
- Average number of cycles per instruction (CPI) is:

$$0.22 \times 5 + 0.11 \times 4 + 0.49 \times 4 + 0.16 \times 3 + 0.02 \times 3 = 4.04$$

At a clock cycle time of 200ps (at best) :

The following are total execution times for different instructions:

A) 800ps for LW : 600ps for SW

B) 1000ps for LW: 600ps for SW

C) 800ps for R-Format : 400ps Jumps

D) 1000ps for LW : 800ps R-format and SW

E) All of the Above answers are wrong ☺

Comparing Single Cycle/ Multi-Cycle

- Yes, clearly Single cycle was faster per instruction
- 600ps vs average 800ps CPI on Multi-Cycle
- However, Multi-Cycle will prove it is useful with Pipelining the instructions.
- Trying to achieve ONE completed instruction Per Clock Cycle.
- Let's See…..