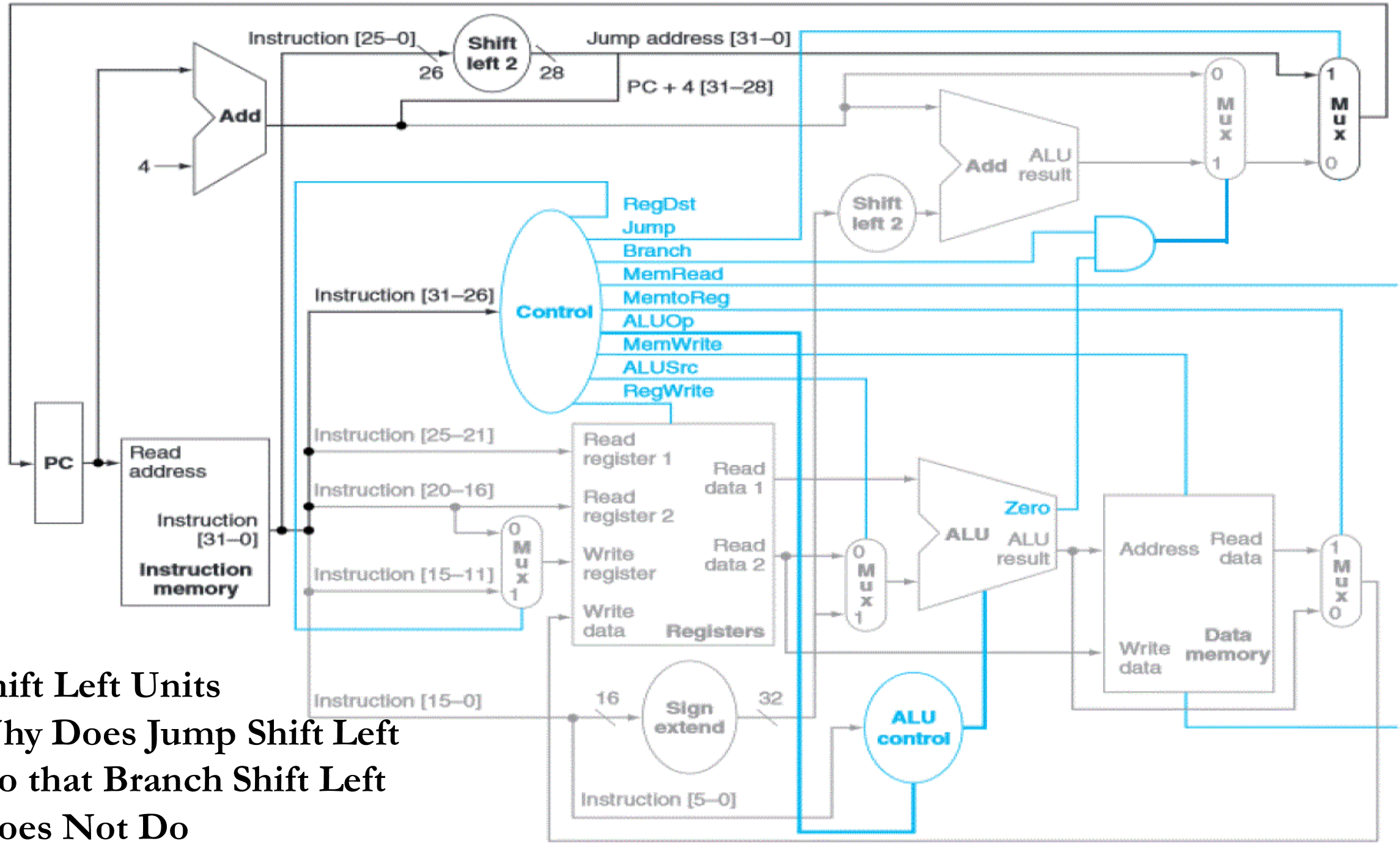
Multi-Cycle Architecture

Part 1



Shift Left Units

Why Does Jump Shift Left

Do that Branch Shift Left

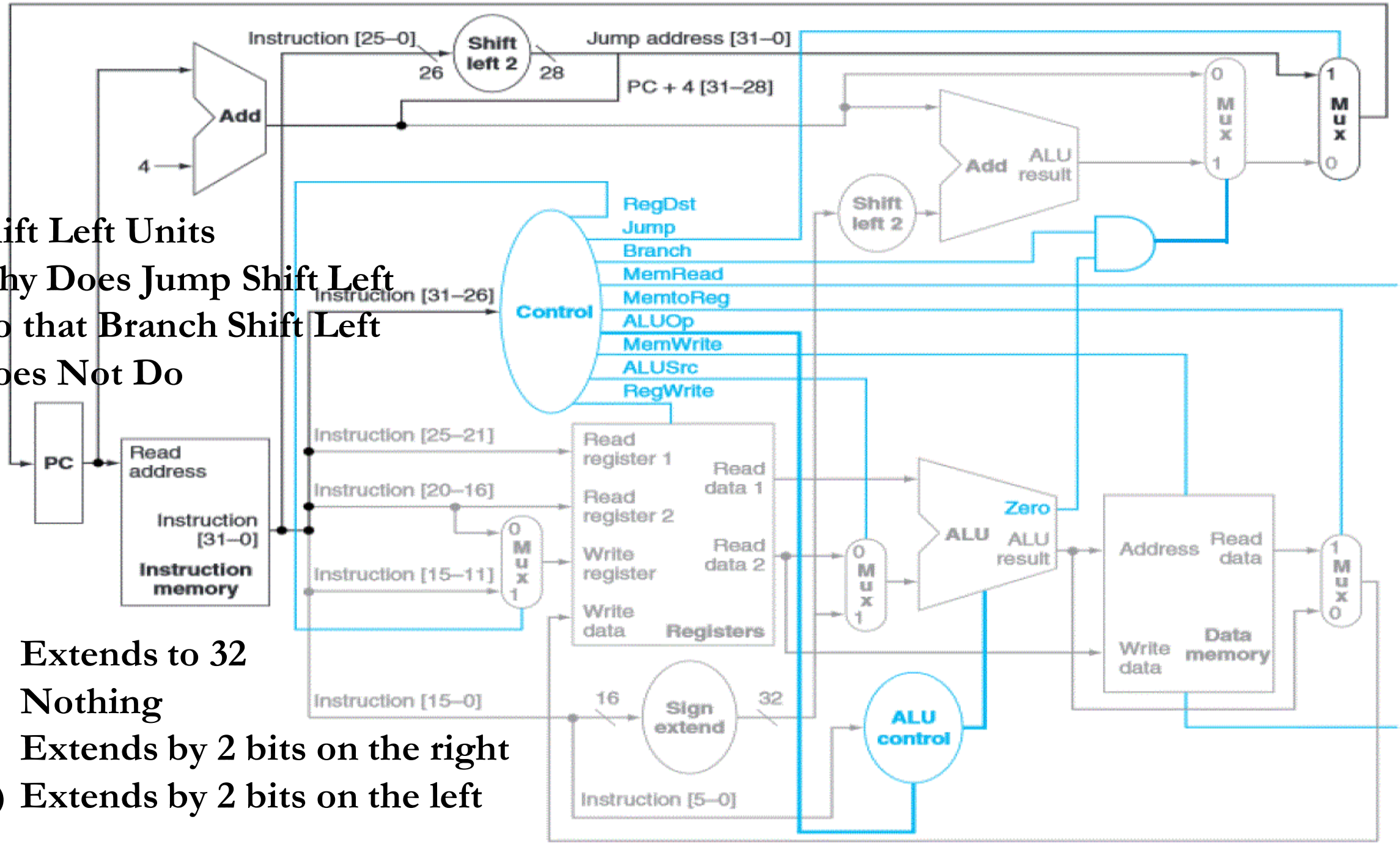
Does Not Do

Shift Left Units

Why Does Jump Shift Left

Do that Branch Shift Left

Does Not Do



A) Extends to 32

B) Nothing

C) Extends by 2 bits on the right

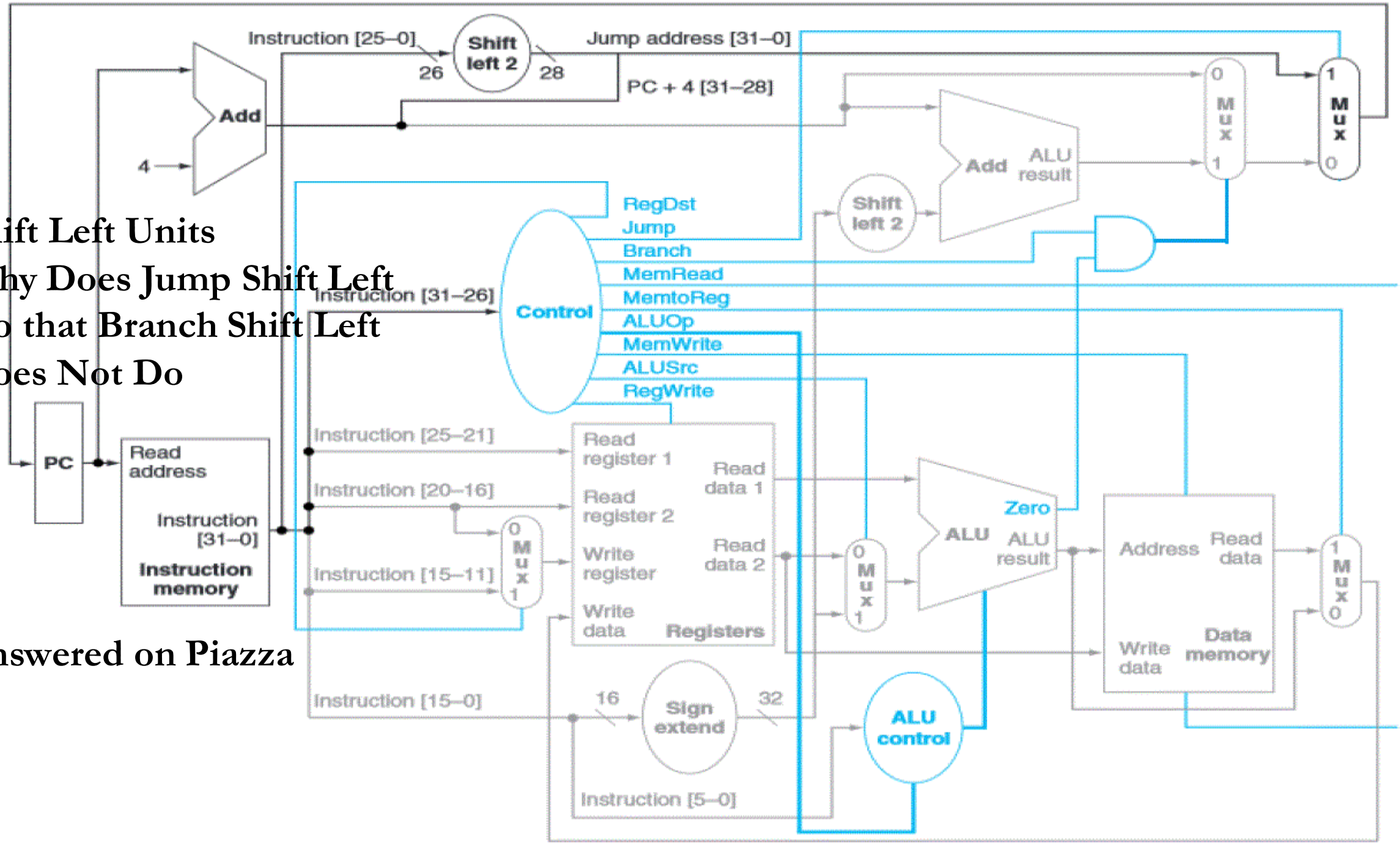
D) Extends by 2 bits on the left

Shift Left Units

Why Does Jump Shift Left

Do that Branch Shift Left

Does Not Do

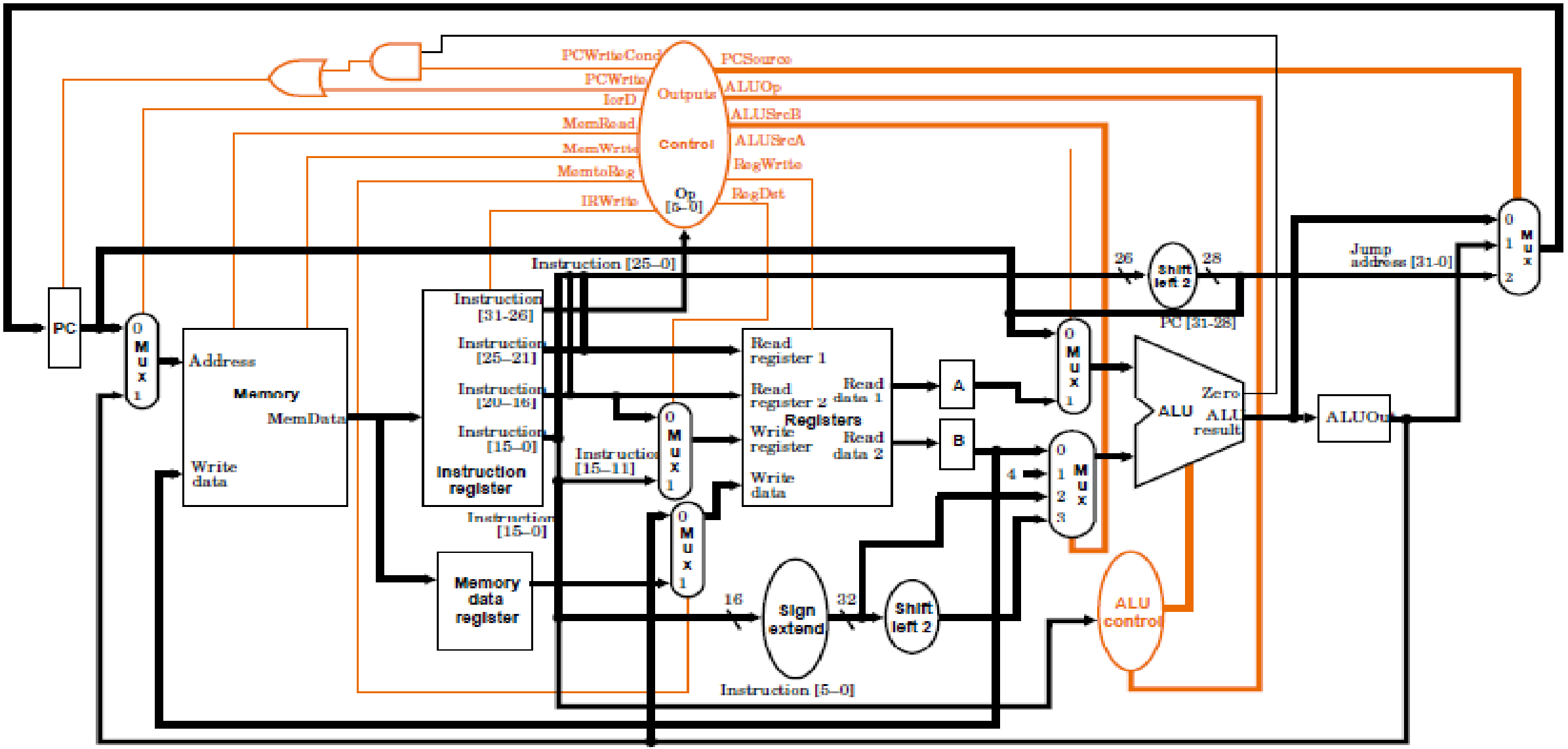


Answered on Piazza

Multi-Cycle Datapath

- One Memory for accessing Instructions or Data: Address source matters
- One ALU does all computations
- Instructions are broken into steps on this datapath
- Intermediate Temporary Registers to store intermediate results
- Some new Control Lines needed. Number of MUXs increased

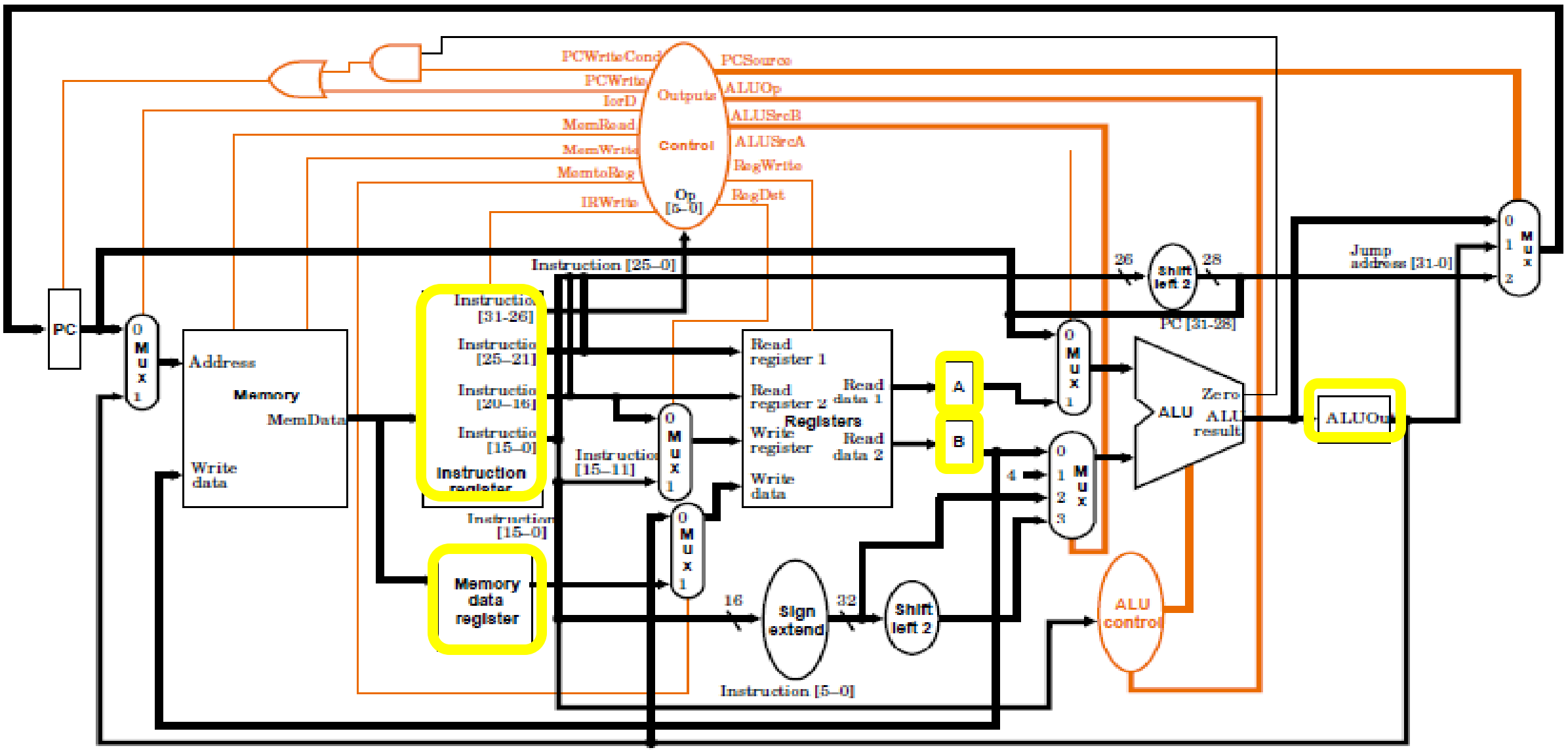
MULTI CYCLE DATA PATH



Breaking Every instruction up into a series of steps. Each step is a new clock cycle

Allow every instruction to have exclusive use of the datapath until it completes

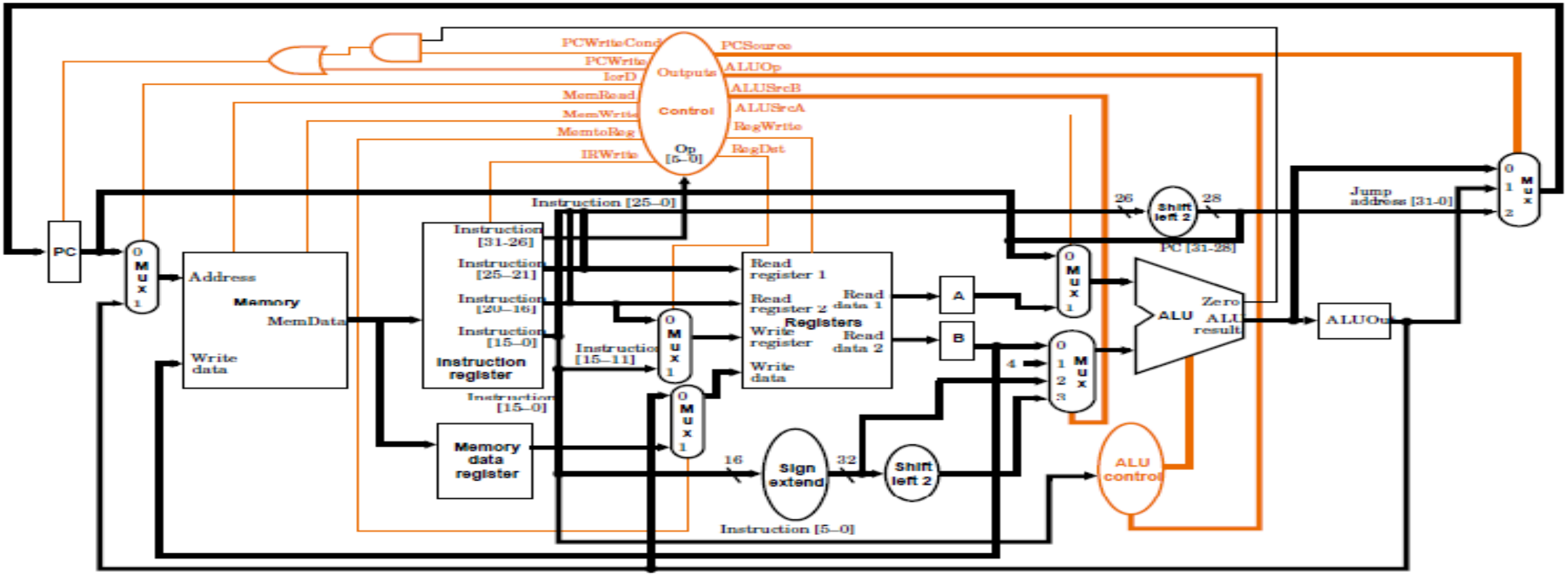
MULTI CYCLE DATA PATH



Breaking Every instruction up into a series of steps. Each step is a new clock cycle

Allow every instruction to have exclusive use of the datapath until it completes

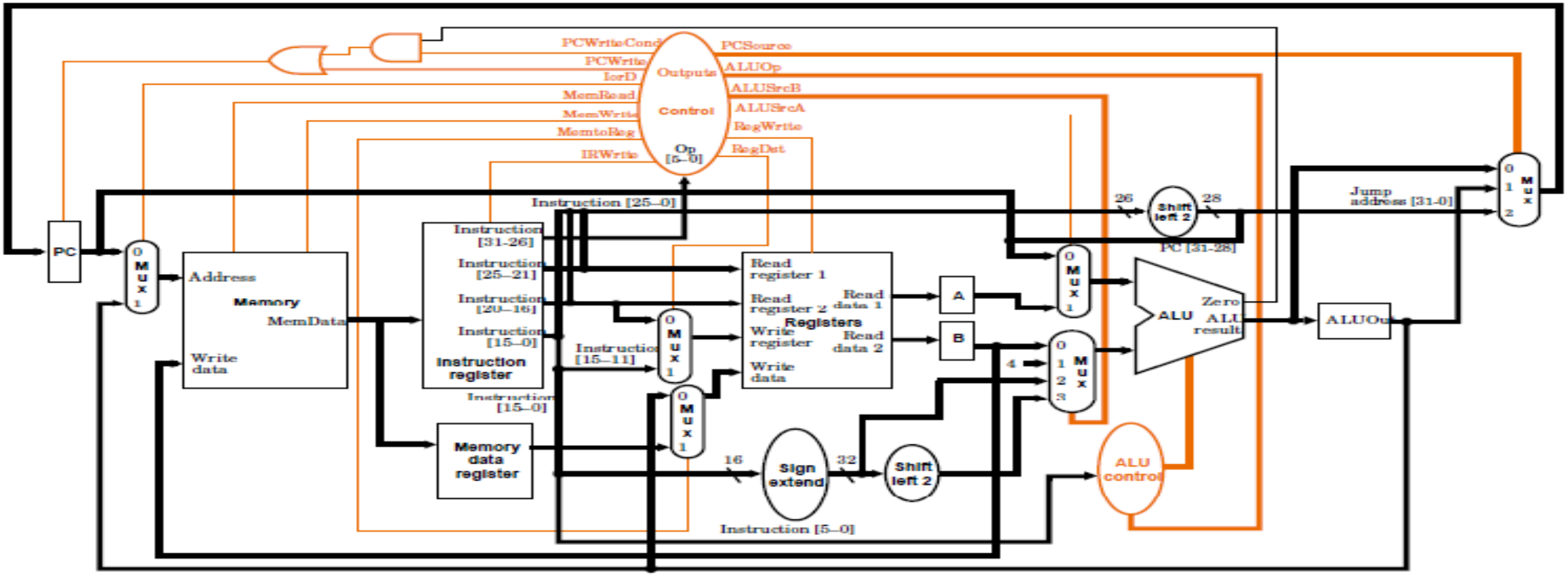
Need intermediate registers to store values between clock cycles.



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]]			
Execute, etc	ALUOut = A op B	ALUOut = A + se(IR[15-0])	**ALUOut = PC + se(IR[15-0]) << 2**	PC = PC[31-28] || IR[25-0]<< 2
Memory, R-type	Reg[IR[15-11]] = ALUout	LD: MDR = Mem[ALUOut] ST: Mem[ALUOut] = B	**if (A==B) then PC = ALUout**	
Memory Read		LD: Reg[IR[20-16]] = MDR		

Each step is in a new clock cycle

Notice: First two steps are common To every instruction.



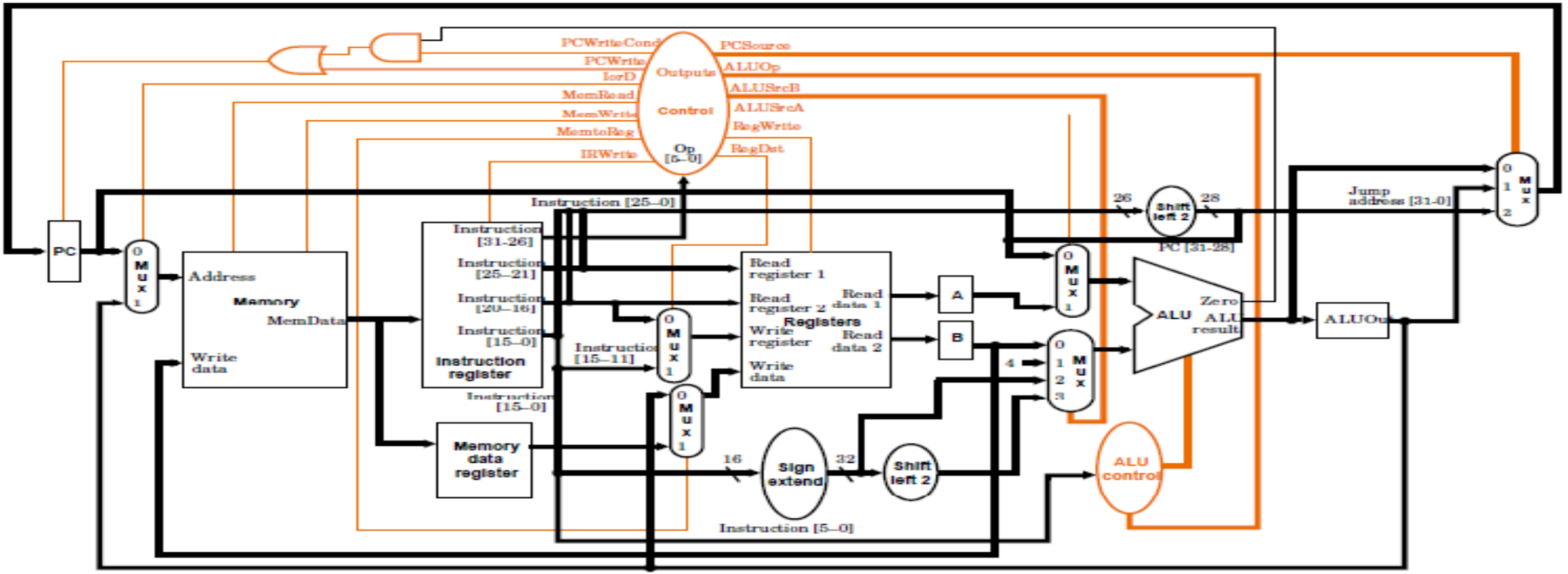
Step	R-Type	Memory Ref	Branch	Jumps
Instruction Fetch	IR = Mem[PC] PC = PC+4			
Decode, Register Fetch	A = Reg[IR[25-21]] B = Reg[IR[20-16]]			
Execute, etc	ALUOut = A op B	ALUOut = A + se(IR[15-0])	**ALUOut = PC + se(IR[15-0]) << 2**	PC = PC[31-28] || IR[25-0]<< 2
Memory, R-type	Reg[IR[15-11]] = ALUout	LD: MDR = Mem[ALUOut] ST: Mem[ALUOut] = B	**if (A==B) then PC = ALUout**	
Memory Read		LD: Reg[IR[20-16]] = MDR		

Each step is in a new clock cycle

Notice: First two steps are common To every instruction.

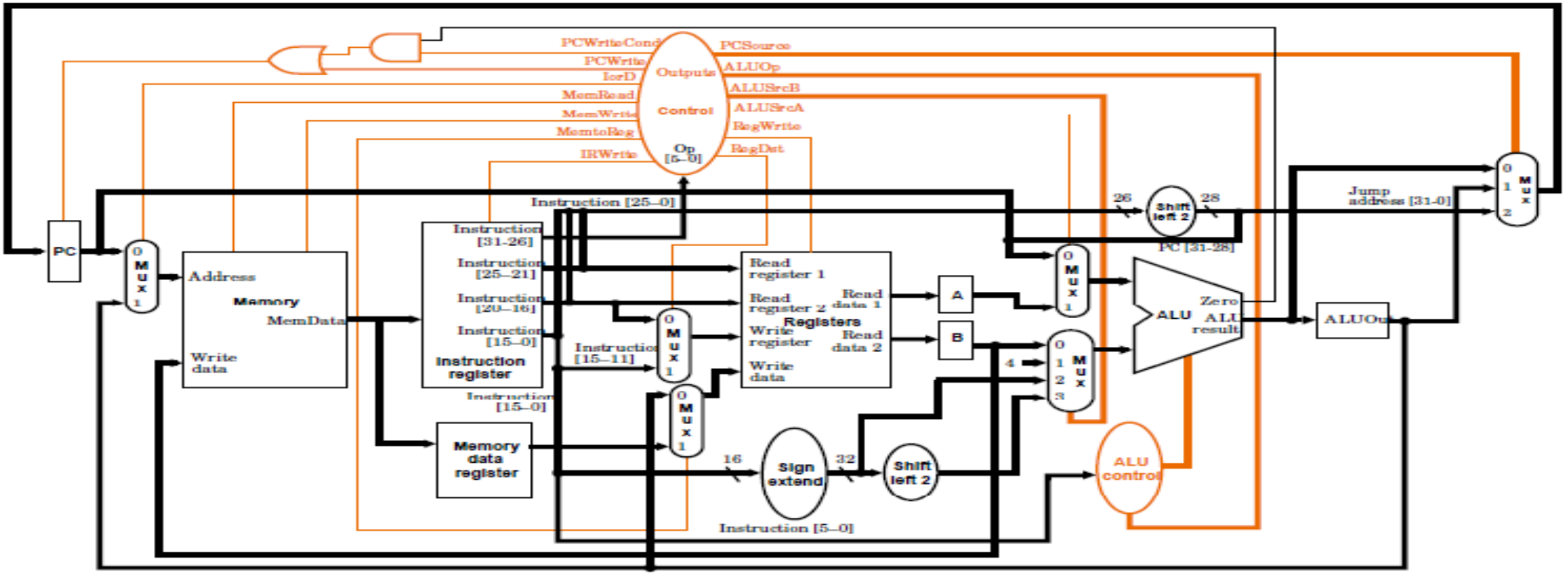
~~Instruction Fetch (1)~~

Instruction Decode (2)



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]]			
Execute, etc	ALUOut = A op B	ALUOut = A + se(IR[15-0])	**ALUOut = PC + se(IR[15-0]) << 2**	PC = PC[31-28] || IR[25-0]<< 2
Memory, R-type	Reg[IR[15-11]] = ALUout	LD: MDR = Mem[ALUOut] ST: Mem[ALUOut] = B	**if (A==B) then PC = ALUout**	
Memory Read		LD: Reg[IR[20-16]] = MDR		

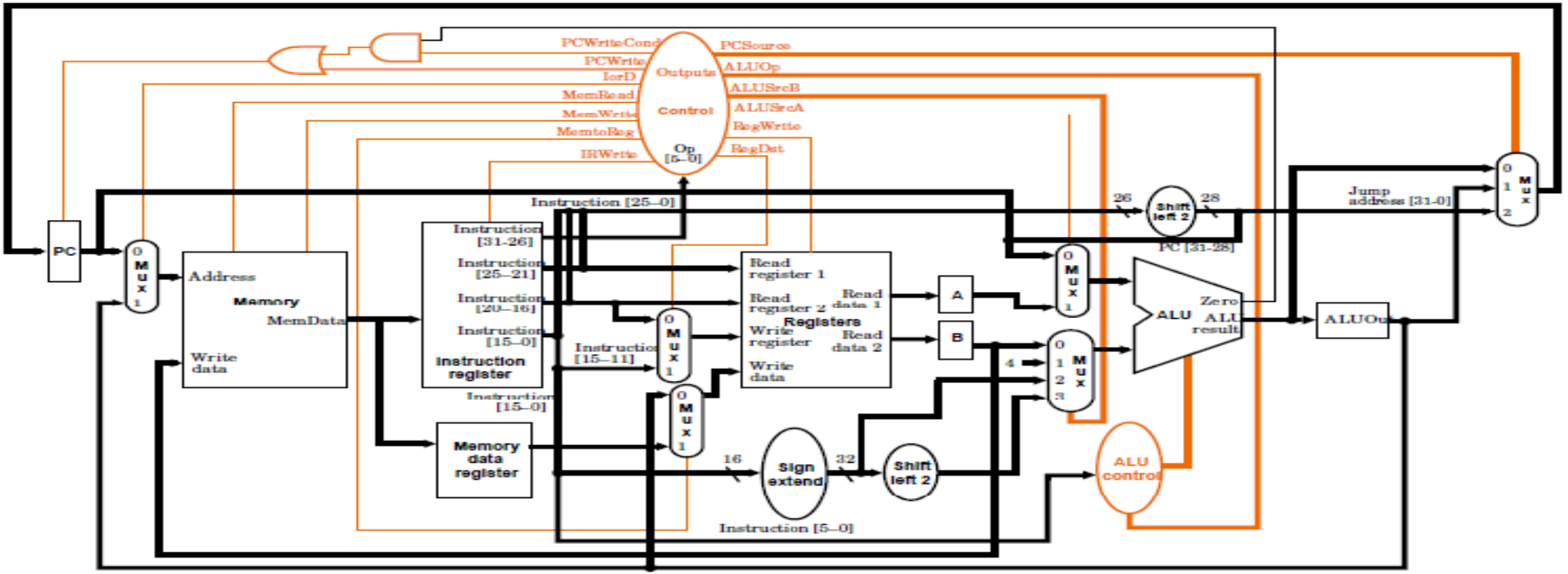
Only One ALU to do any computations Therefore try to optimize the usage of the ALU in each clock cycle.



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]]			
Execute, etc	ALUOut = A op B	ALUOut = A + se(IR[15-0])	**ALUOut = PC + se(IR[15-0]) << 2**	PC = PC[31-28] || IR[25-0]<< 2
Memory, R-type	Reg[IR[15-11]] = ALUout	LD: MDR = Mem[ALUOut] ST: Mem[ALUOut] = B	**if (A==B) then PC = ALUout**	
Memory Read		LD: Reg[IR[20-16]] = MDR		

Only One ALU to do any computations Therefore try to optimize the usage of the ALU in each clock cycle.

Is the ALU being used in each step?



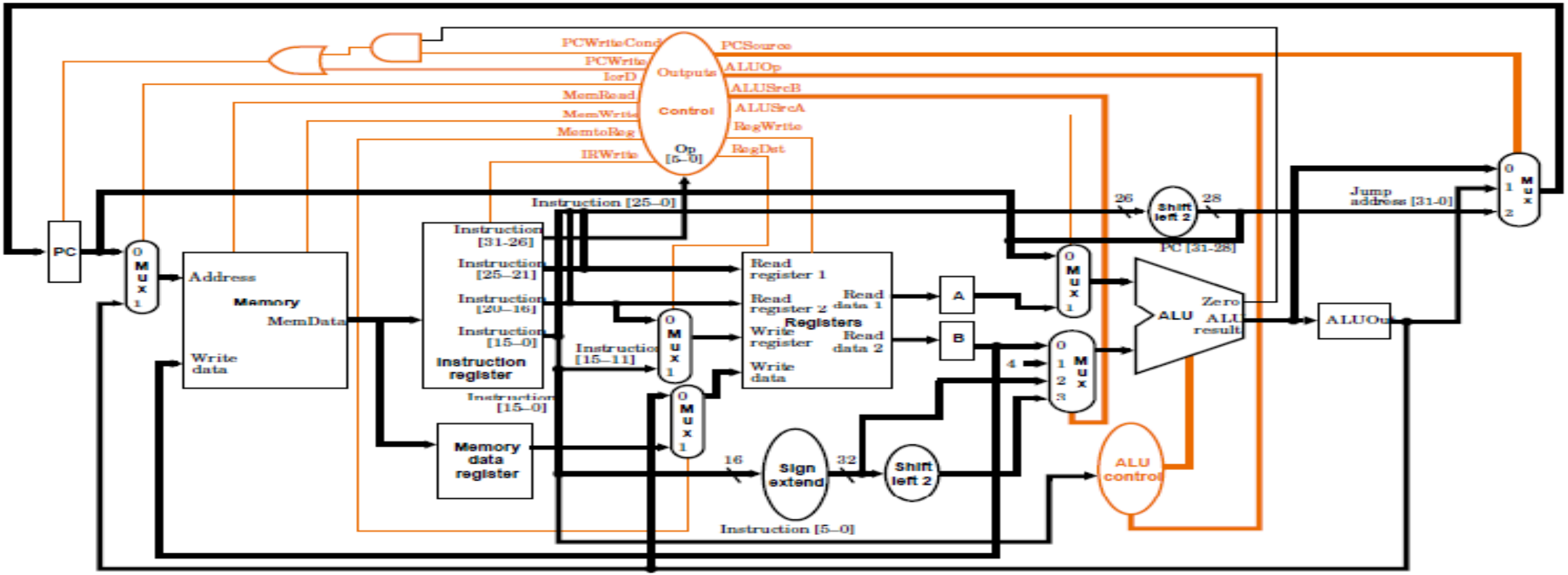
Step	R-Type	Memory Ref	Branch	Jumps
Instruction Fetch	IR = Mem[PC] PC = PC+4 ← ALU			
Decode, Register Fetch	A = Reg[IR[25-21]] B = Reg[IR[20-16]]			
Execute, etc	ALUOut = A op B	ALUOut = A + se(IR[15-0])	**ALUOut = PC + se(IR[15-0]) << 2**	PC = PC[31-28] || IR[25-0]<< 2
Memory, R-type	Reg[IR[15-11]] = ALUout	LD: MDR = Mem[ALUOut] ST: Mem[ALUOut] = B	**if (A==B) then PC = ALUout**	
Memory Read		LD: Reg[IR[20-16]] = MDR		

ALU

ALU

Only One ALU to do any computations Therefore try to optimize the usage of the ALU in each clock cycle.

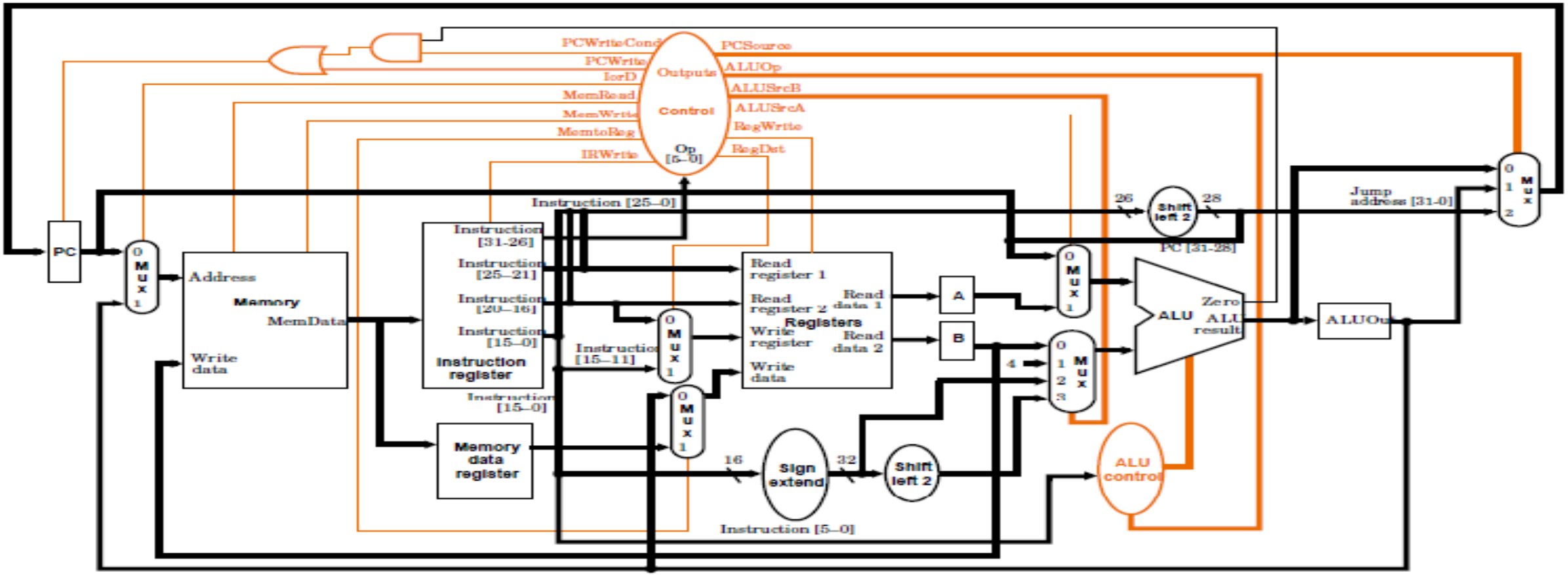
Is the ALU being used in each step?



Step	R-Type	Memory Ref	Branch	Jumps
Instruction Fetch	IR = Mem[PC] PC = PC+4 ← ALU			
Decode, Register Fetch	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALU?			
Execute, etc	ALUOut = A op B	ALUOut = A + se(IR[15-0])	**ALUOut = PC + se(IR[15-0]) << 2**	PC = PC[31-28] || IR[25-0]<< 2
Memory, R-type	Reg[IR[15-11]] = ALUout	LD: MDR = Mem[ALUOut] ST: Mem[ALUOut] = B	**if (A==B) then PC = ALUout**	
Memory Read		LD: Reg[IR[20-16]] = MDR		

Only One ALU to do any computations
Therefore try to optimize the usage of the ALU in each clock cycle.

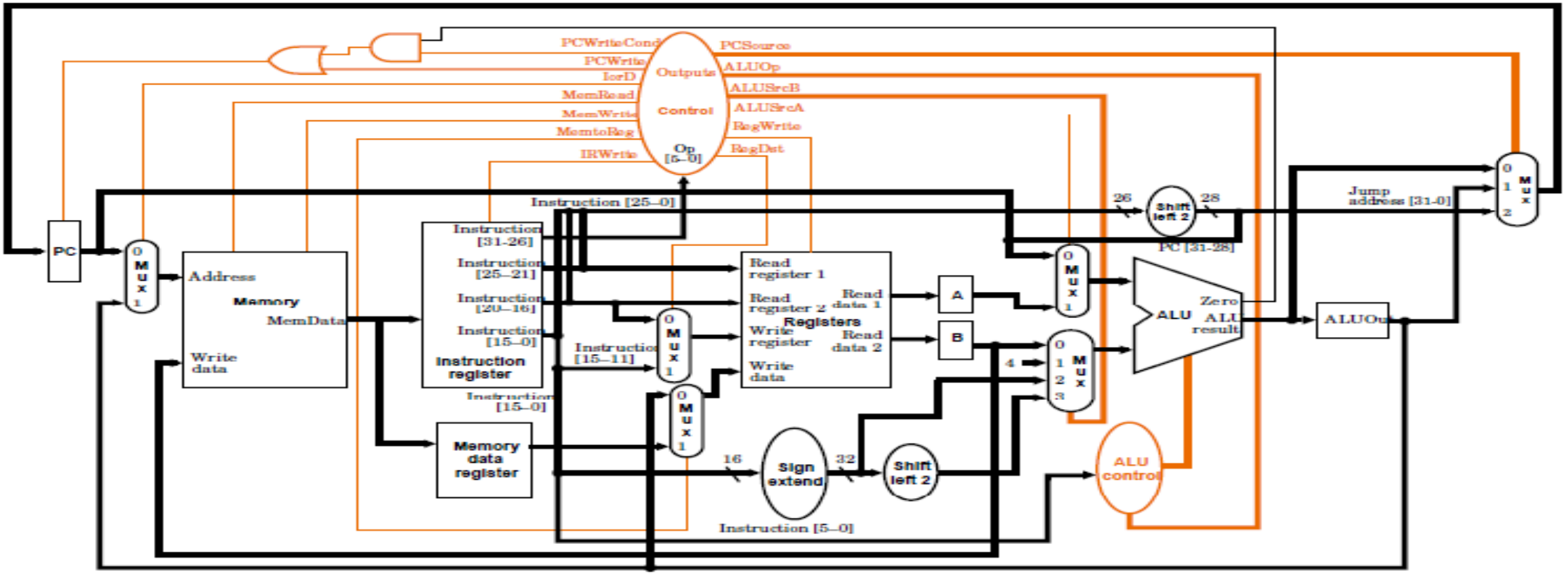
Is the ALU being used in each step?



Step	R-Type	Memory Ref	Branch	Jumps
Instruction Fetch	IR = Mem[PC] PC = PC+4 ← ALU			
Decode, Register Fetch	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALU?			
Execute, etc	ALUOut = A op B	ALUOut = A + se(IR[15-0])	**ALUOut = PC + se(IR[15-0]) << 2**	PC = PC[31-28] || IR[25-0]<< 2
Memory, R-type	Reg[IR[15-11]] = ALUout	LD: MDR = Mem[ALUOut] ST: Mem[ALUOut] = B	**if (A==B) then PC = ALUout**	
Memory Read		LD: Reg[IR[20-16]] = MDR		

Examine the Branch Instruction:

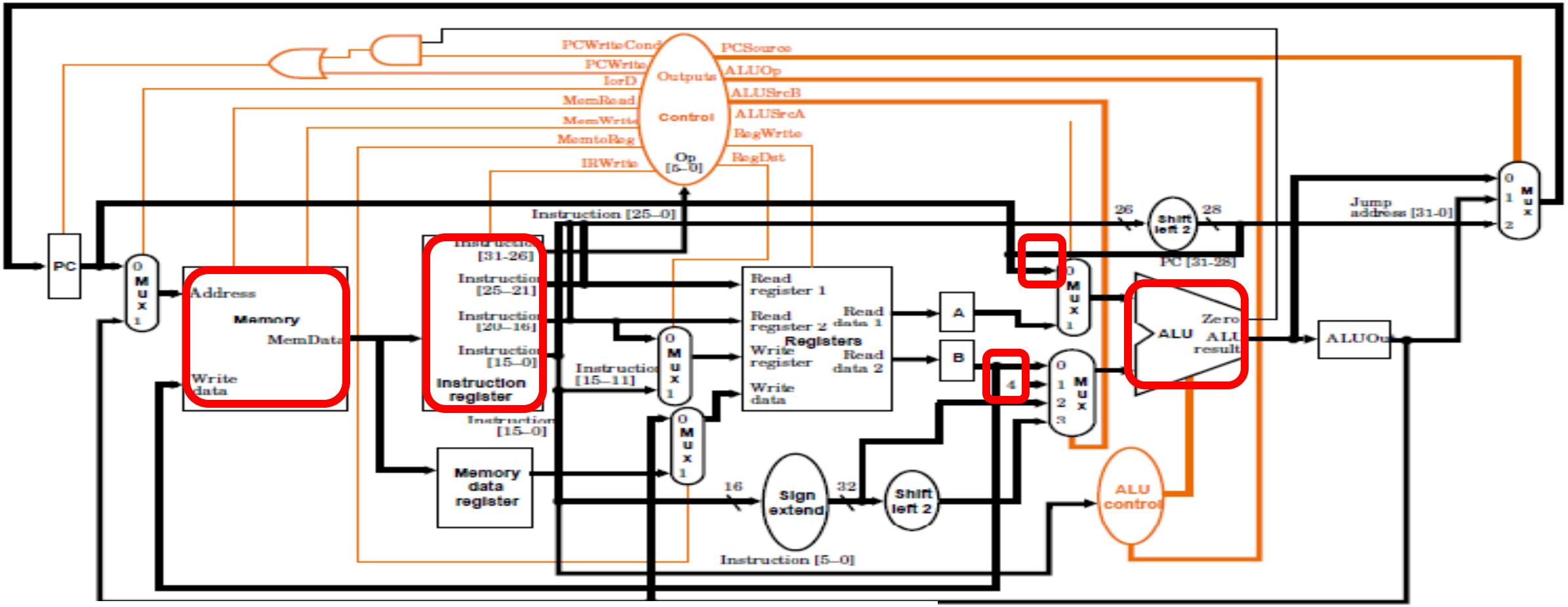
Currently takes 4 clock cycles To complete.
Can some of the branch computation Complete earlier



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		

Pull Up Branch Target Address computation Into 2nd stage:

Therefore, If instruction was a branch: Target Address is ready and computed.

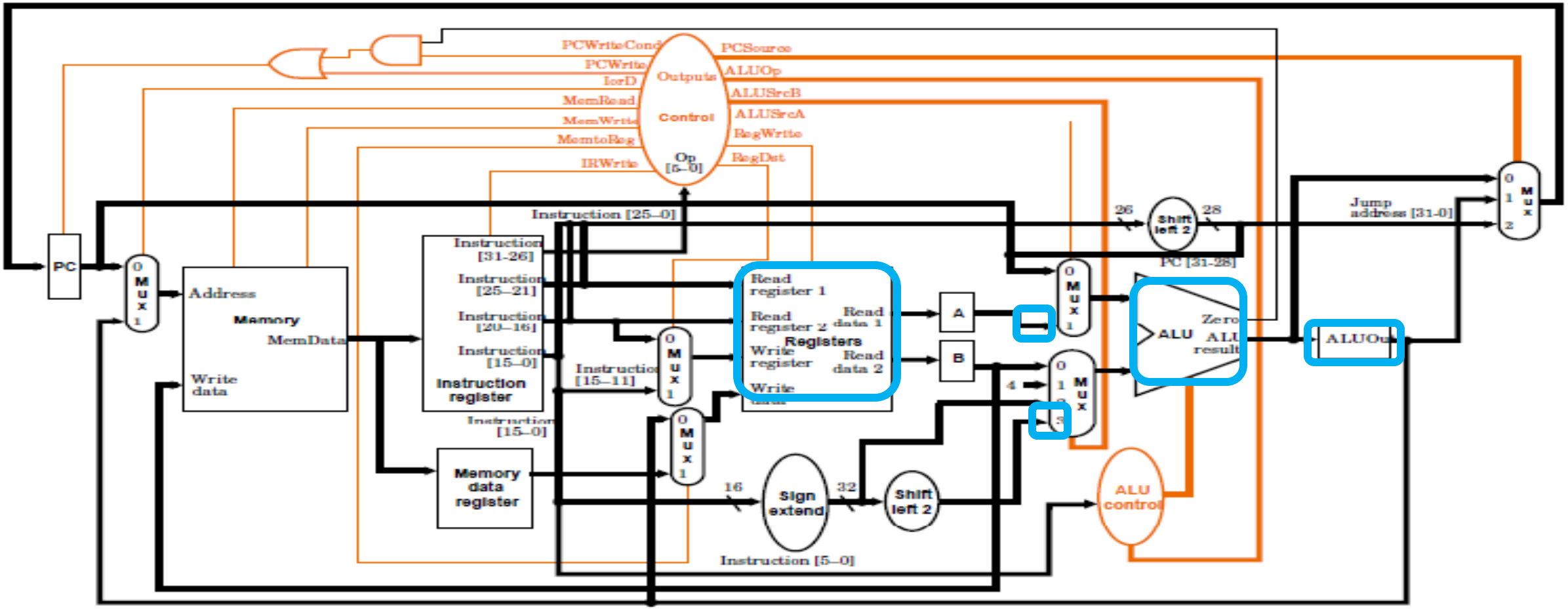


STEP 01: Instruction Fetch

- Effect:

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

- 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



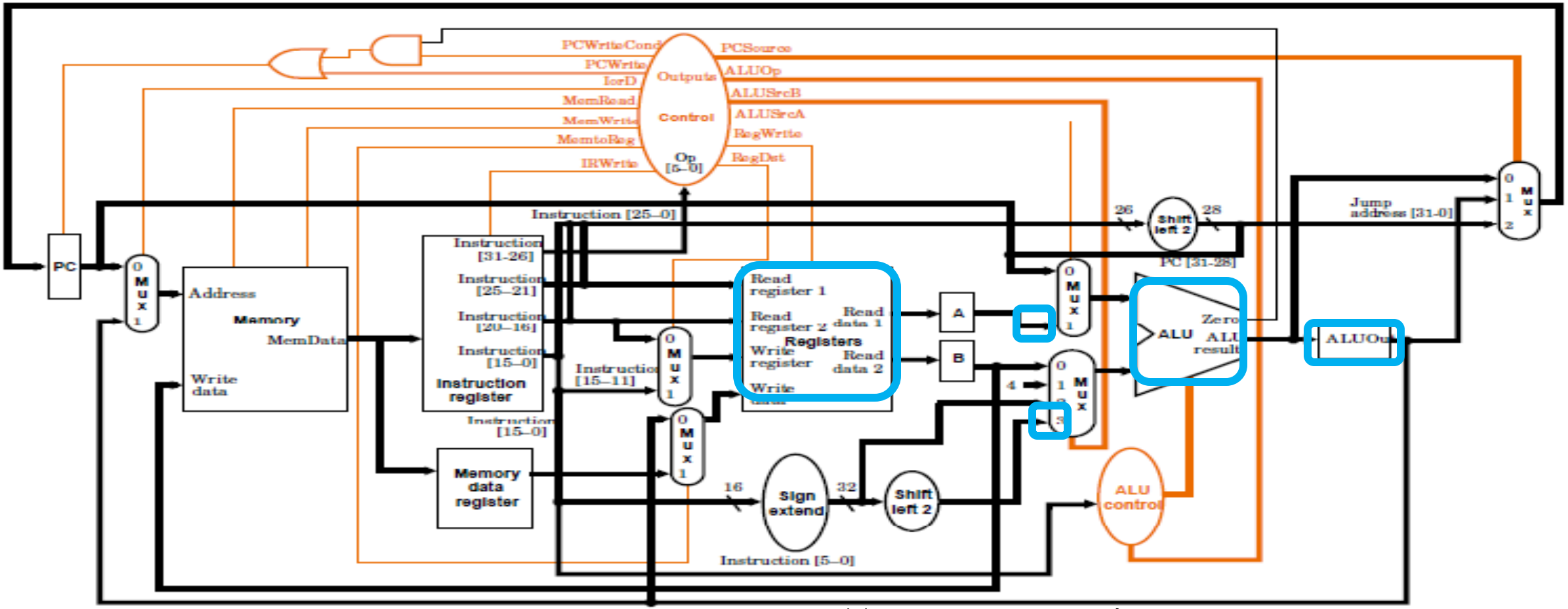
- Effect:

```
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)

STEP 02: DECODE/Register Fetch

Compute the Branch Target Address
Instruction bits are all available,
Just incase it is the Branch Instruction
Lets compute the Address and store it in ALUO



STEP 02: DECODE/Register Fetch

- Effect:

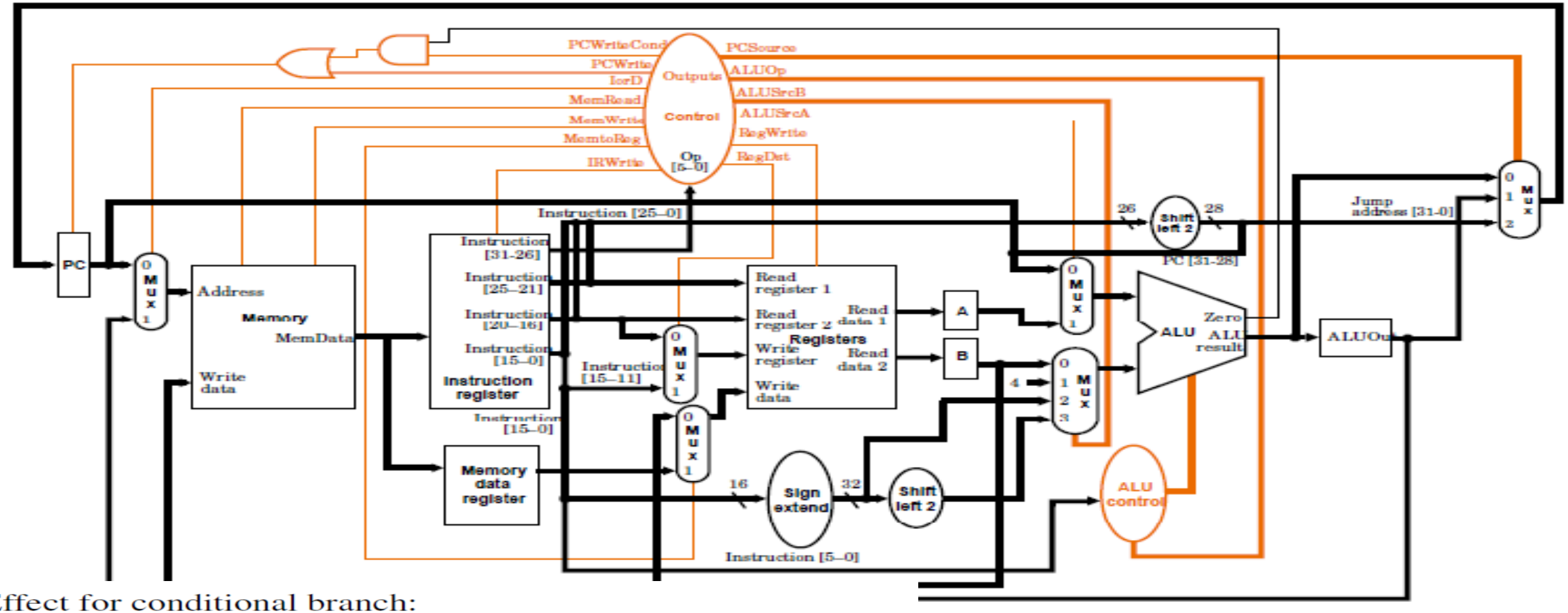
```
A = Reg[IR[25 – 21]];
B = Reg[IR[20 – 16]];
ALUOut = PC + (sign-extend (IR[15-0]) << 2);
```

- 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)

Also in this stage: Control Unit gets
The Opcode Bits and decipher them.
By the next clock cycle- the control lines can be set differently depending on what instruction it is

Third Step in Multi-Cycle Datapath

- In Step 3 we now differentiate the steps as needed for each instruction.
- It is in the 3rd clock cycle that we now know what the opcode was (Control has set the control lines differently for each instruction)
- Therefore we will do a different step three depending on what instruction it is
- Keep In Mind : Temporary Registers are read from in the beginning of the clock cycle and written to at the end of every clock cycle.

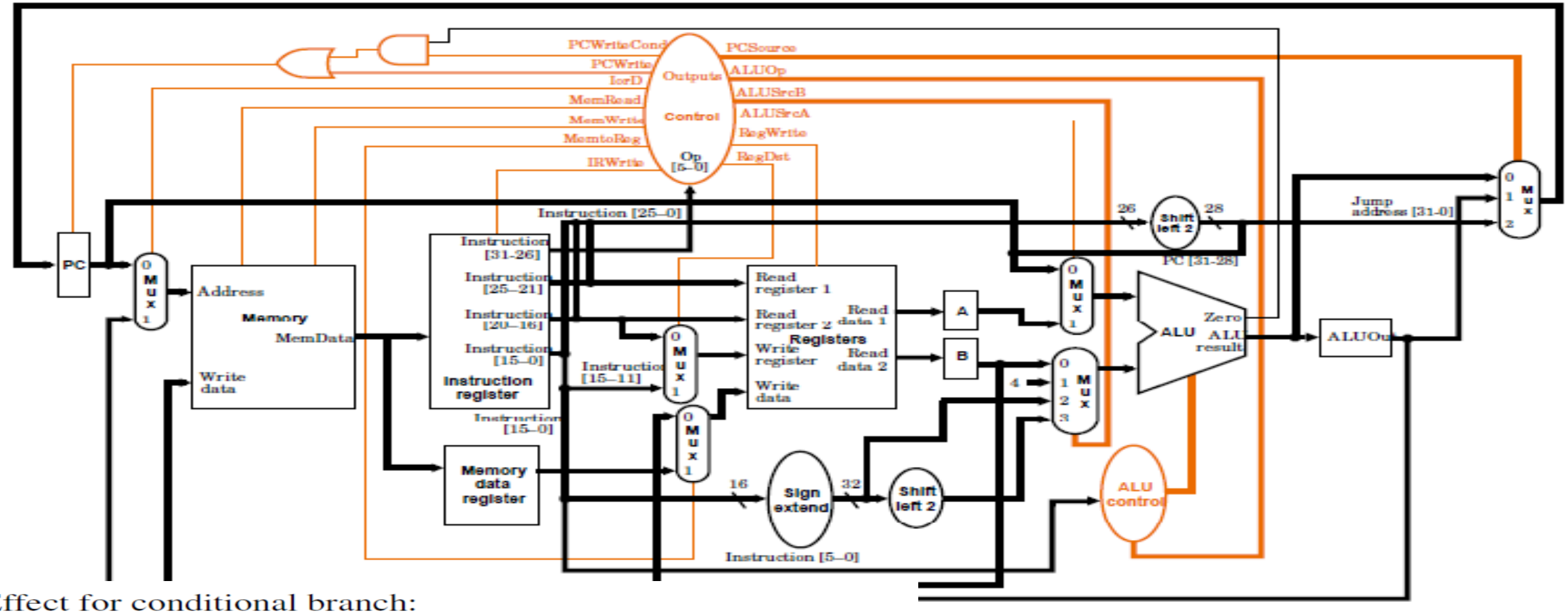


- Effect for conditional branch:
  `if (A == B) PC = ALUOut;`
- Implementation:
  - ALUSrcA = 1 (A to ALU), ALUSrcB = 00 (B to ALU)
  - ALUop = 01 (ALU subtracts, Zero output used for equality test)
  - PCWriteCond = 1 (if Zero, PC is written)
  - PCSource = 01 (PC value from ALUOut, address already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

STEP 03: BRANCH

What Else needs to be done for Branch? We have computed Branch Target Address

Now do we set PC to this address?

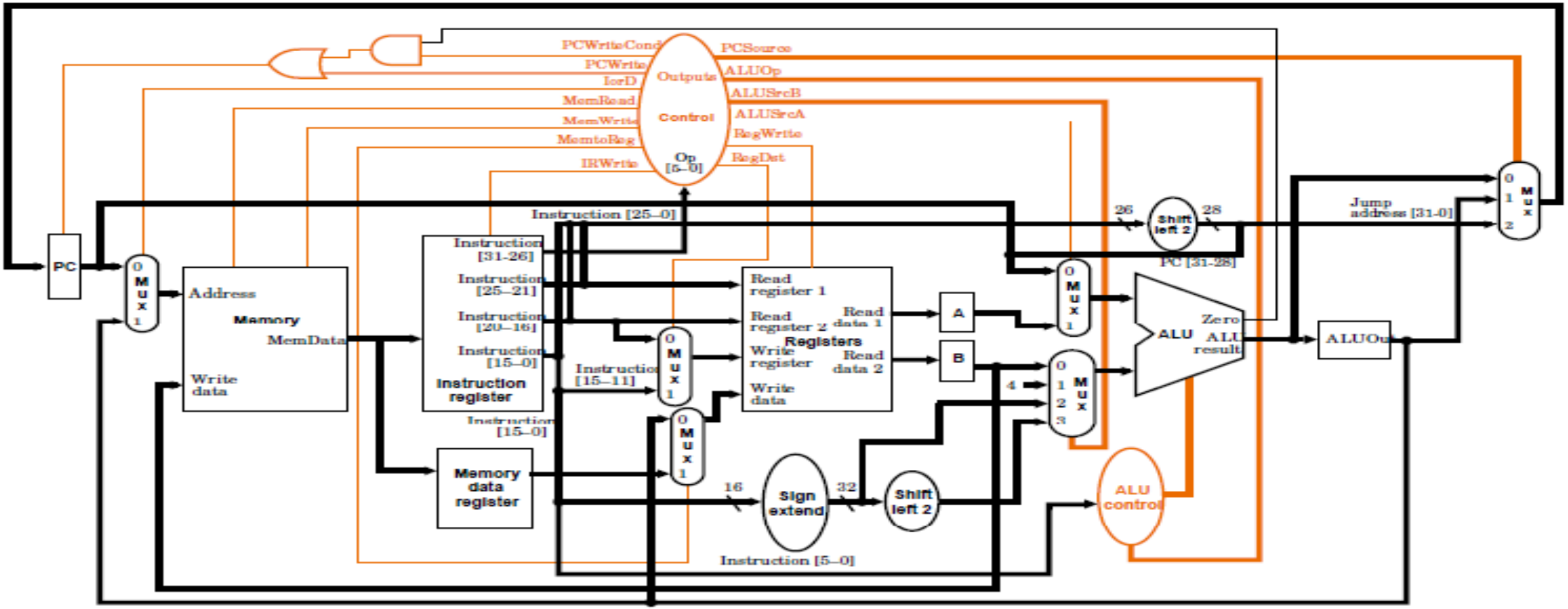


- Effect for conditional branch:
  `if (A == B) PC = ALUOut;`
- Implementation:
  - ALUSrcA = 1 (A to ALU), ALUSrcB = 00 (B to ALU)
  - ALUop = 01 (ALU subtracts, Zero output used for equality test)
  - PCWriteCond = 1 (if Zero, PC is written)
  - PCSource = 01 (PC value from ALUOut, address already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

STEP 03: BRANCH

Compare Two Registers Using ALU.

ALUOut Had Something in it already



In this Third Step For Branch Instruction.

I use the ALU to compare two Source Registers.

What does ALUOut Already contain at beginning of this clock cycle?

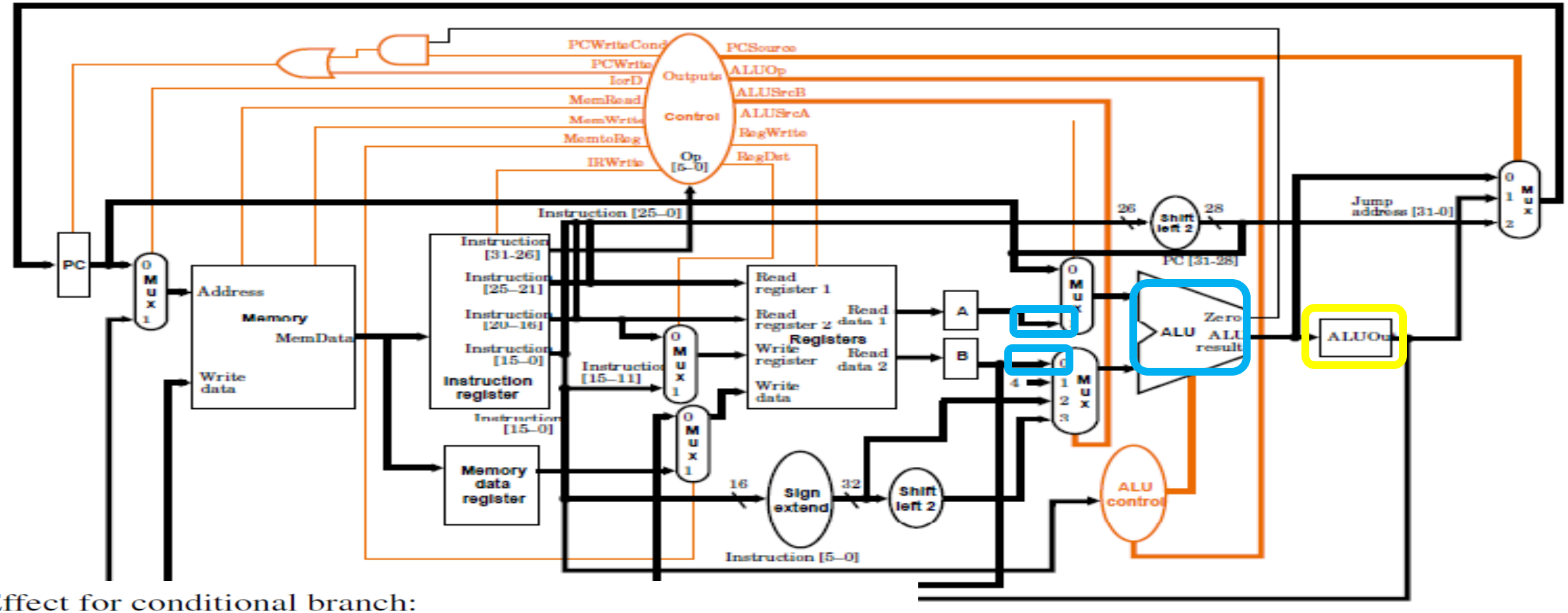
A) Difference between Rs and Rt

B) ~~Branch offset~~

C) Branch Target Address

D) Garbage Data

E) Jump Target Address

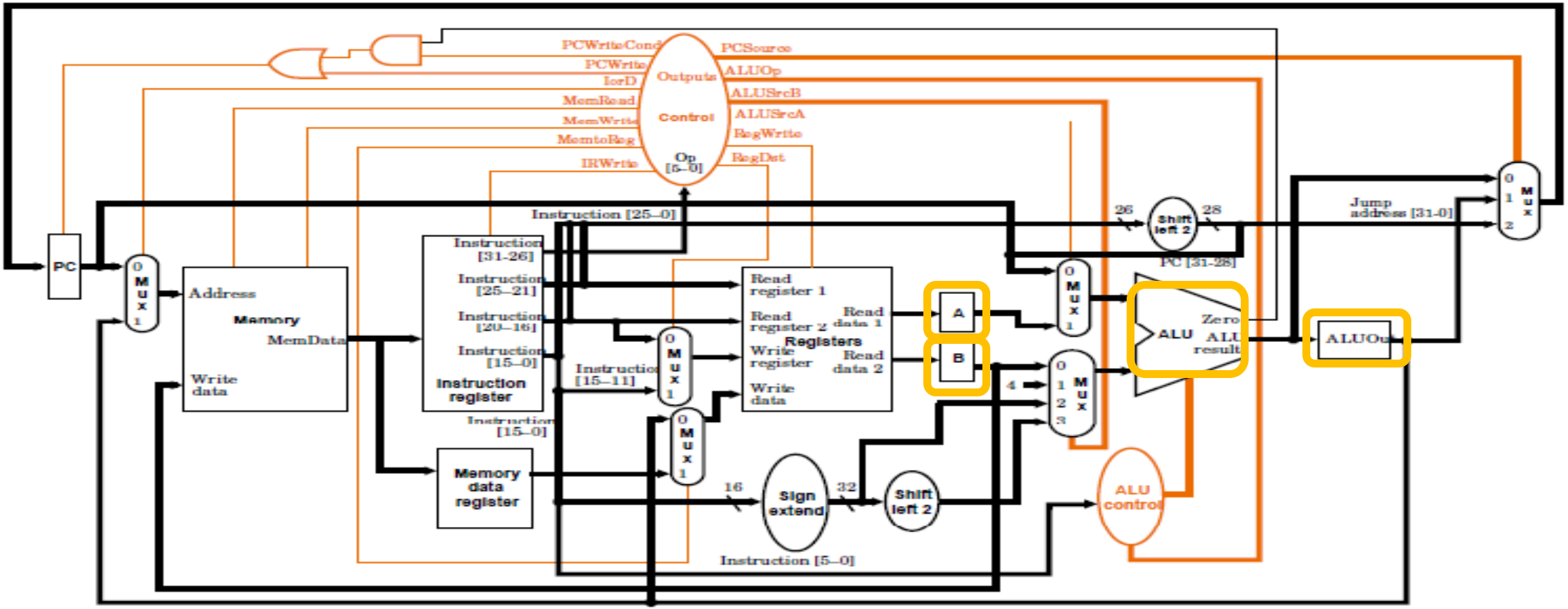


- Effect for conditional branch:
  `if (A == B) PC = ALUOut;`
- Implementation:
  - ALUSrcA = 1 (A to ALU), ALUSrcB = 00 (B to ALU)
  - ALUop = 01 (ALU subtracts, Zero output used for equality test)
  - PCWriteCond = 1 (if Zero, PC is written)
  - PCSource = 01 (PC value from ALUOut, address already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

STEP 03: BRANCH

Compare Two Registers

Use Branch Target Address from ALUOut,
If two registers were equal



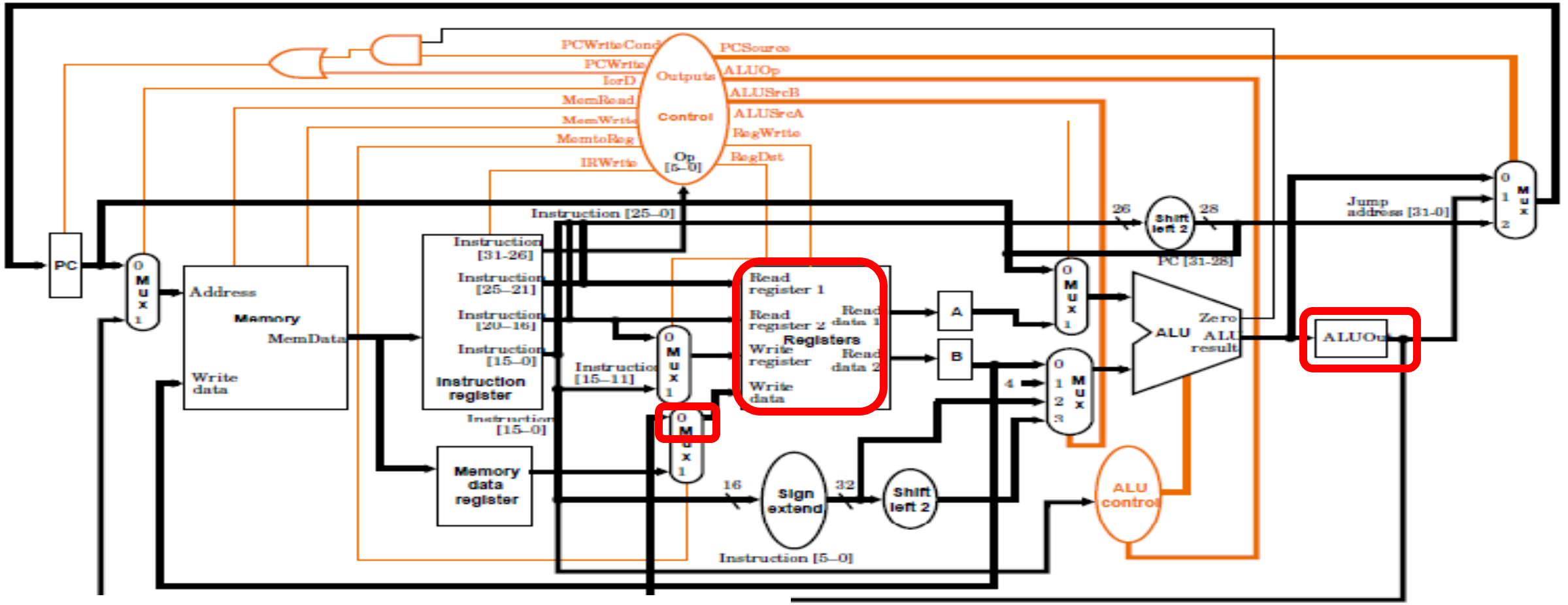
- Effect for R-type instructions:

```
ALUOut = A op B;
```

- Implementation:
  - ALUSrcA = 1, ALUSrcB = 00 (A and B to ALU)
  - ALUop = 10 (ALU function determined by funct field)

STEP 03: Execution → If this was an R-Format Instruction

Use ALU to perform arithmetic operation Specified by R-Format.
Already have Register data stored



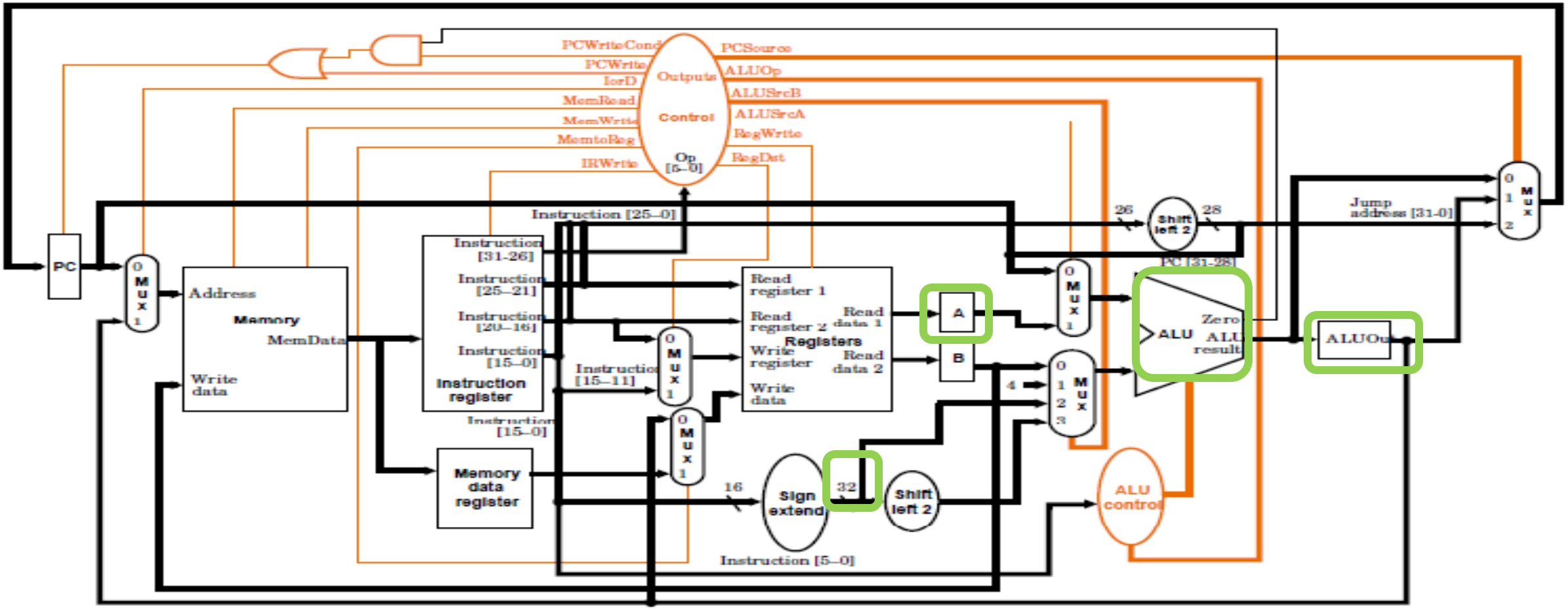
- Effect for R-type instruction:

  `Reg[IR[15-11]] = ALUOut;`

- Implementation:
  - RegDst = 1 (rd field specifies write register)
  - RegWrite = 1 (register file written)
  - MemToReg = 0 (ALUOut value used, not MDR)

STEP 04: R TYPE COMPLETION

Last step for R-Format
Write to Register File From computation ALU Did in Last clock cycle (stored in ALUOut)



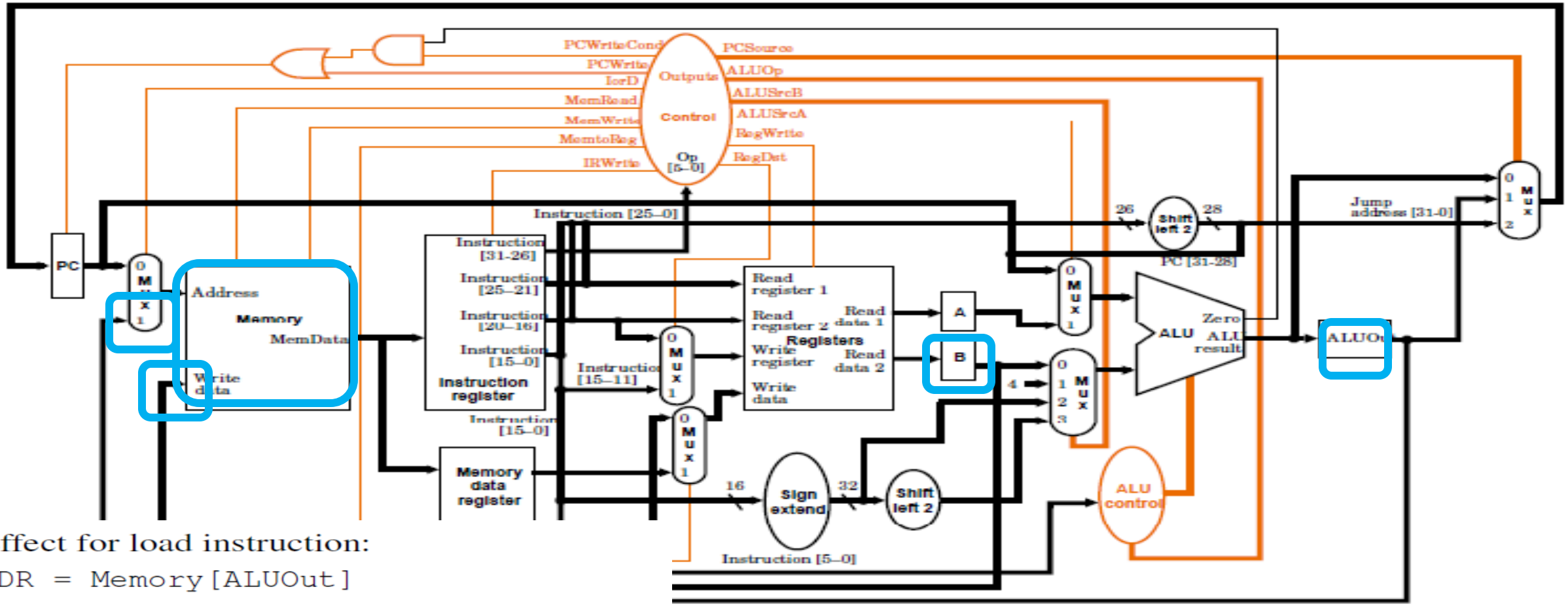
- Effect for load/store instructions:

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

- Implementation:
  - ALUSrcA = 1 (A to ALU), ALUSrcB = 10 (base to ALU)
  - ALUop = 00 (ALU adds)

STEP 03: MEMORY Instruction

Step 03 is compute the Address



STEP 04: SW

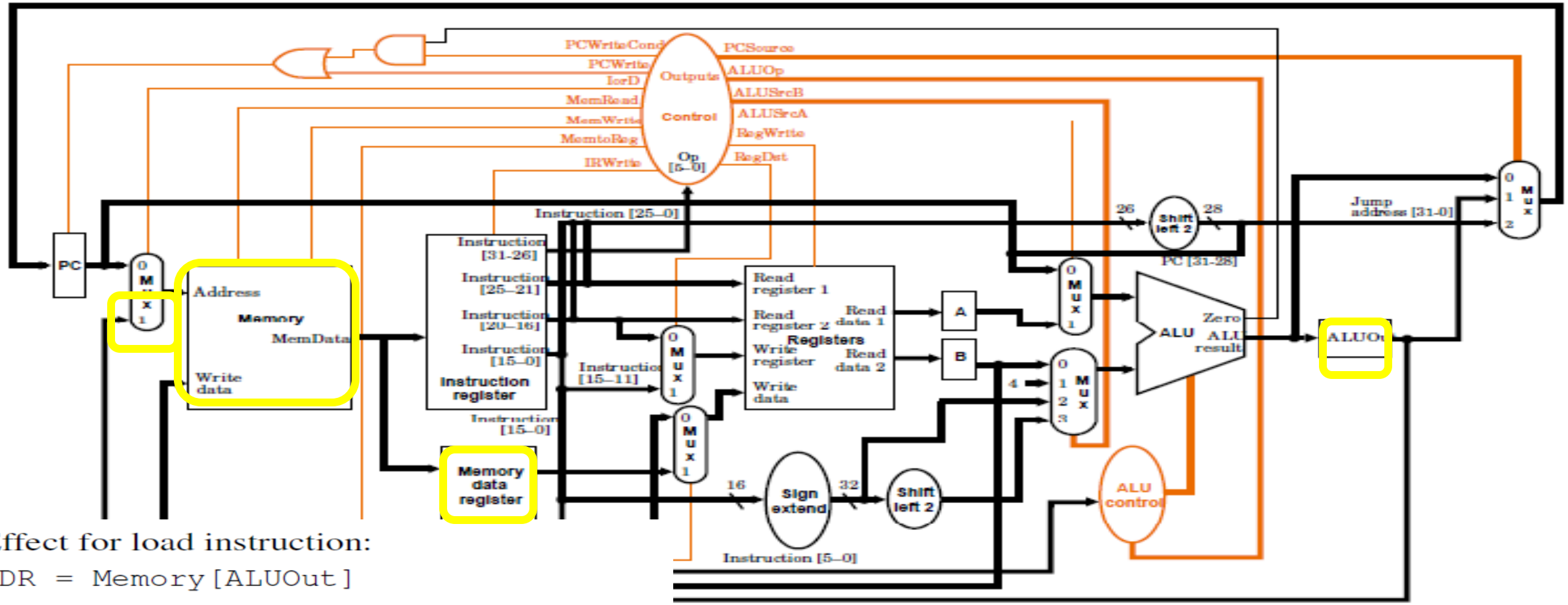
- Effect for load instruction:

  `MDR = Memory[ALUOut]`

- Implementation:
  - MemRead = 1 (memory is read)
  - IorD = 1 (memory address from ALUOut)
- Effect for store instruction:

  `Memory[ALUOut] = B;`

- Implementation:
  - MemWrite = 1 (memory is written)
  - IorD = 1 (memory address from ALUOut)



STEP 04: LW

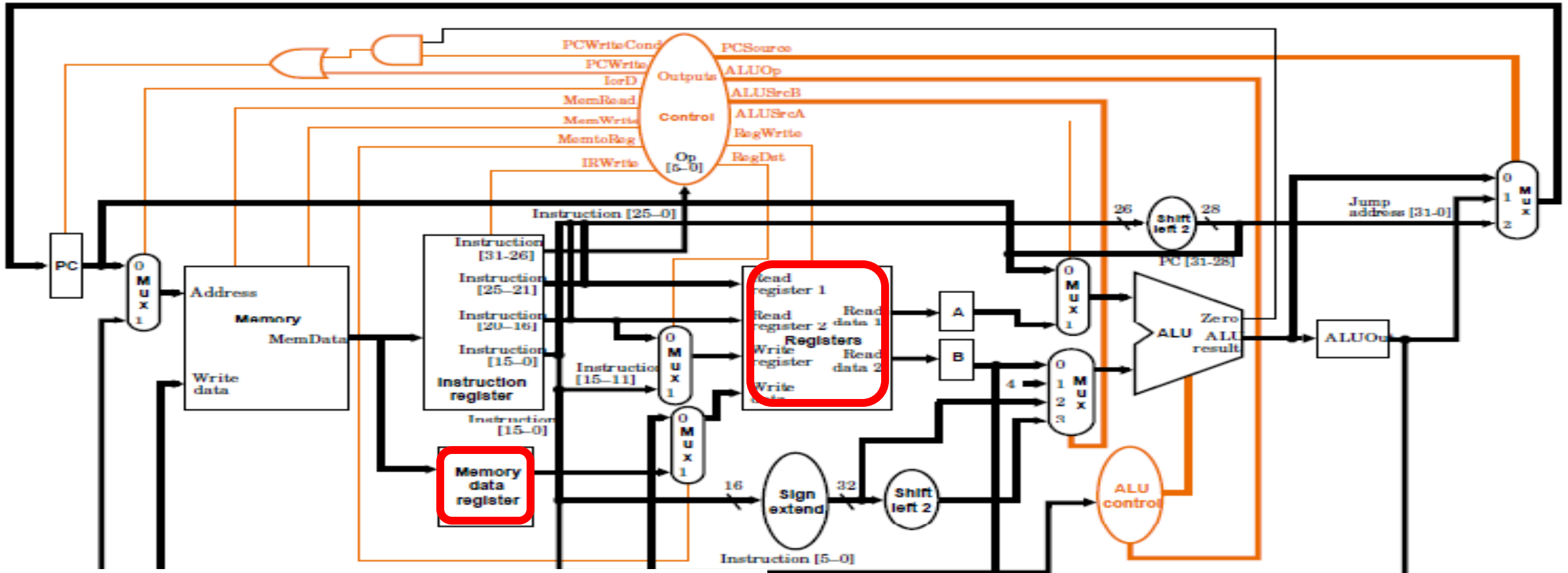
- Effect for load instruction:

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

- Implementation:
  - MemRead = 1 (memory is read)
  - IorD = 1 (memory address from ALUOut)
- Effect for store instruction:

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

- Implementation:
  - MemWrite = 1 (memory is written)
  - IorD = 1 (memory address from ALUOut)

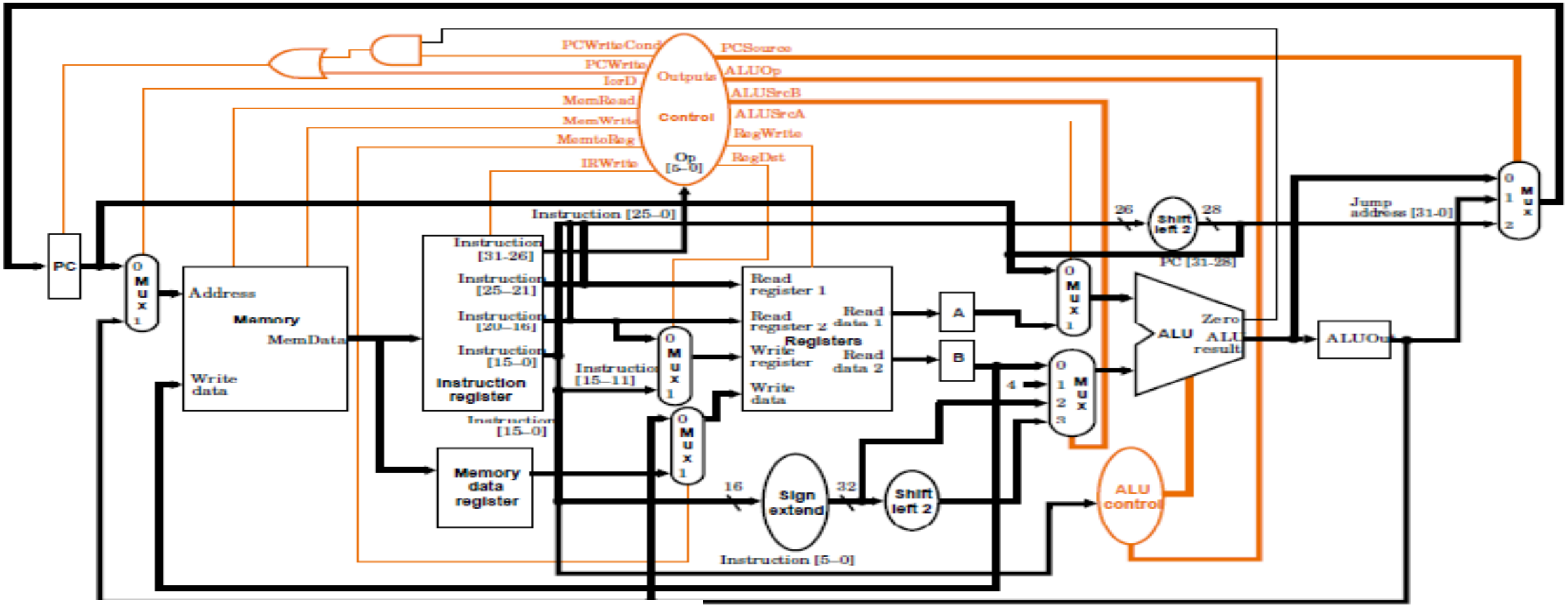


- Effect for load instruction:

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

- Implementation:
  - RegDst = 0 (rt field specifies write register)
  - RegWrite = 1 (register file written)
  - MemToReg = 1 (MDR value used, not ALUOut)

STEP 05: LW Completion



- 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)

Jump

In How many Clock Cycles Does the Jump Instruction Complete? :

A) 2 B) 3 C)1 D) 4

Clock Cycle of Multi-cycle Datapath

- Clock Rate: 1/time of one clock cycle : 200ps : 500MHz
- Instruction Memory / 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: 25ps

- Instruction Memory / 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: 25ps

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