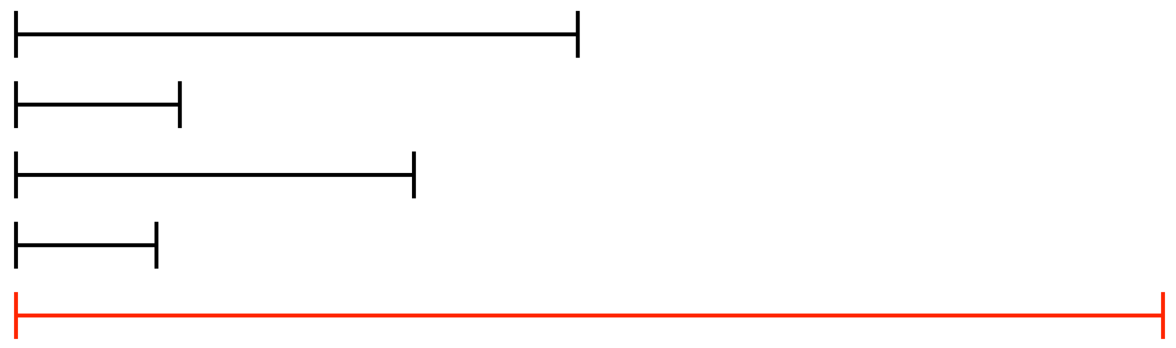
# Multi Cycle CPU

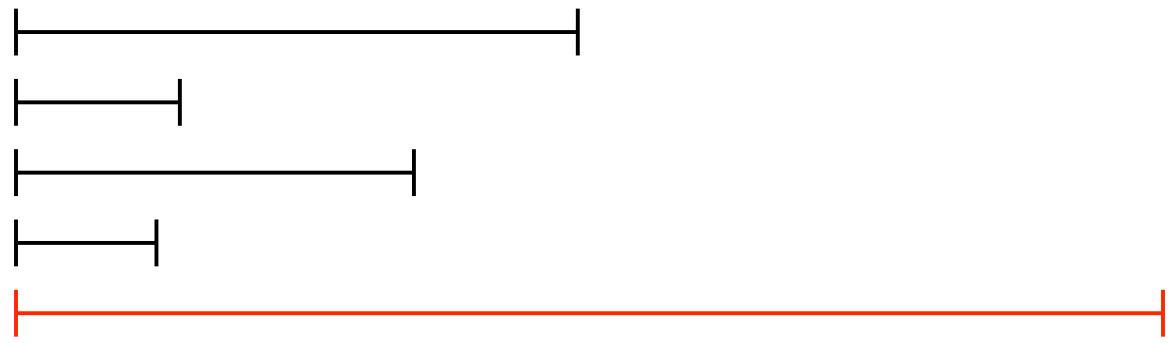
Jason Mars

## Why a Multiple Cycle CPU?



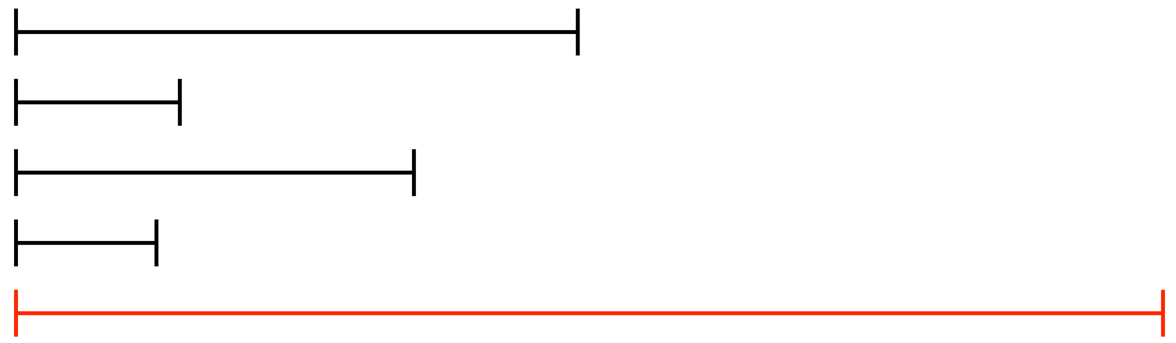
## Why a Multiple Cycle CPU?

- The problem => single-cycle cpu has a cycle time long enough to complete the longest instruction in the machine



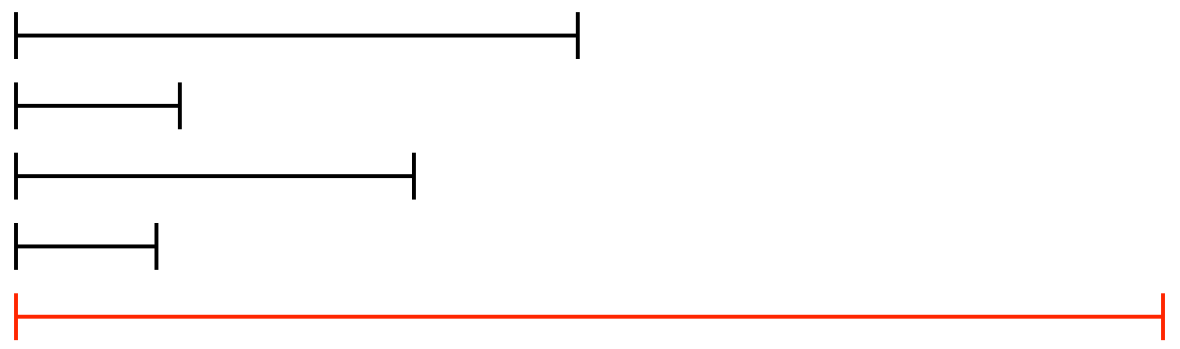
## Why a Multiple Cycle CPU?

- 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



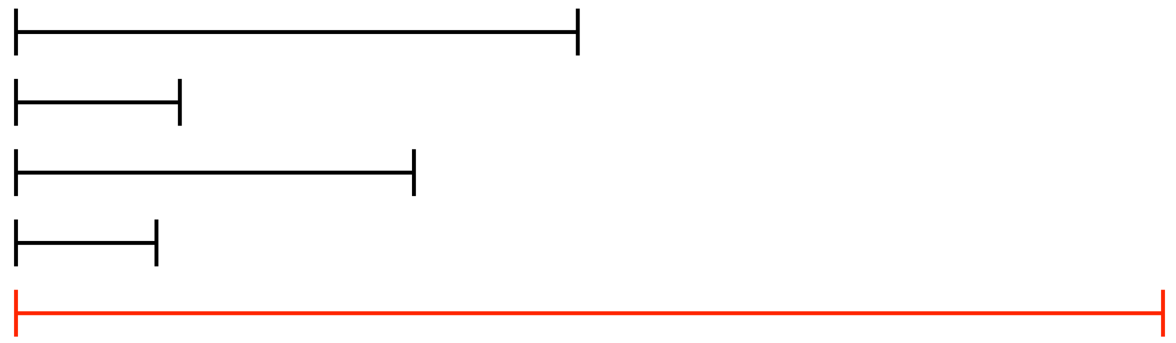
## Why a Multiple Cycle CPU?

- 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)



## Why a Multiple Cycle CPU?

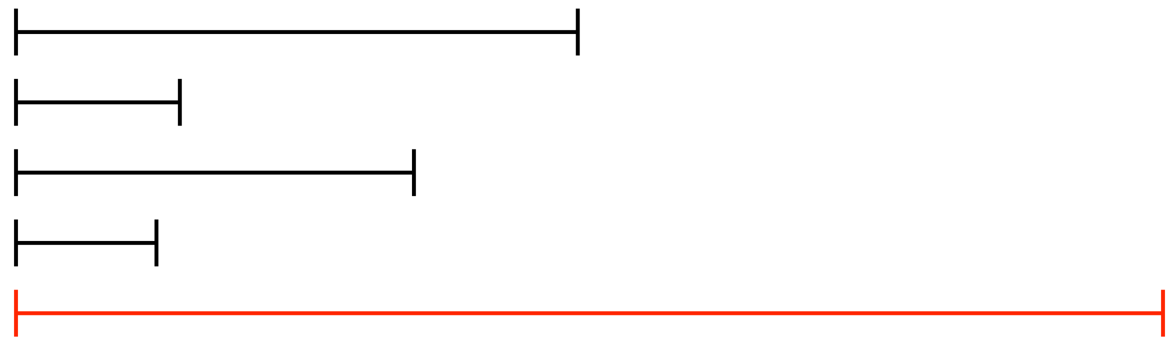
- The problem => single-cycle cpu has a cycle time long enough to complete the longest instruction in the machine
- The solution => break up execution into smaller tasks, each task taking a cycle, different instructions requiring different numbers of cycles or tasks
- Other advantages => reuse of functional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



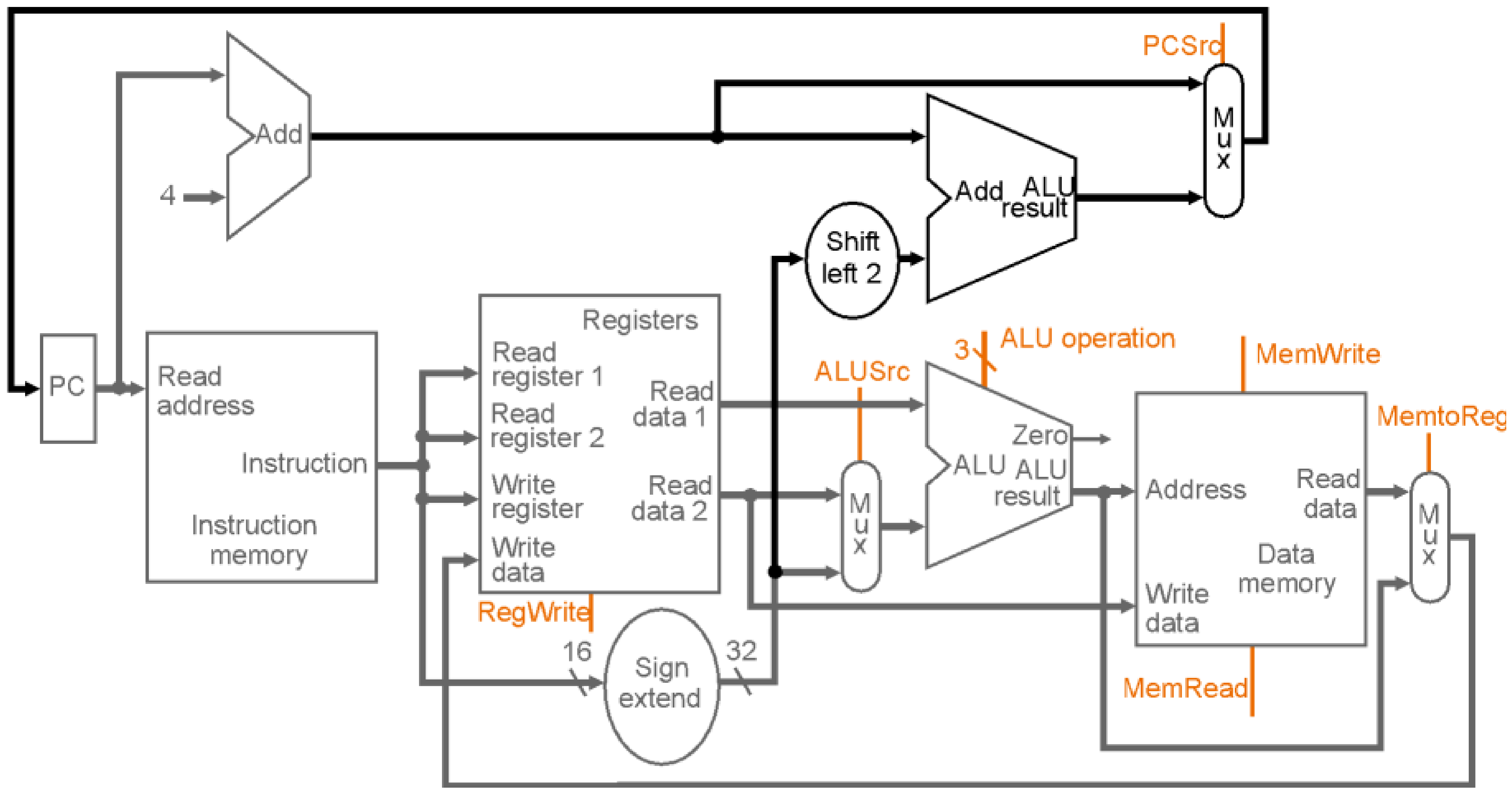
## Why a Multiple Cycle CPU?

- 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



# 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

## 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.

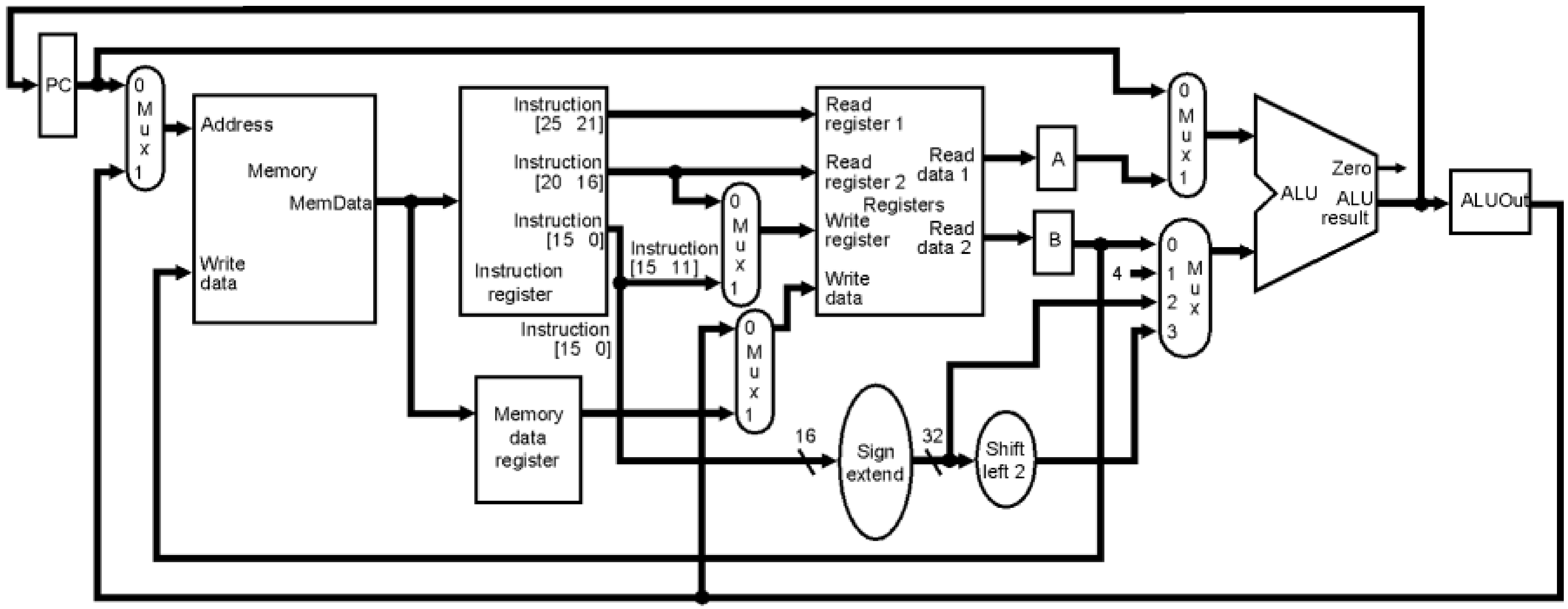
## Breaking Execution into Clock Cycles

- 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?**

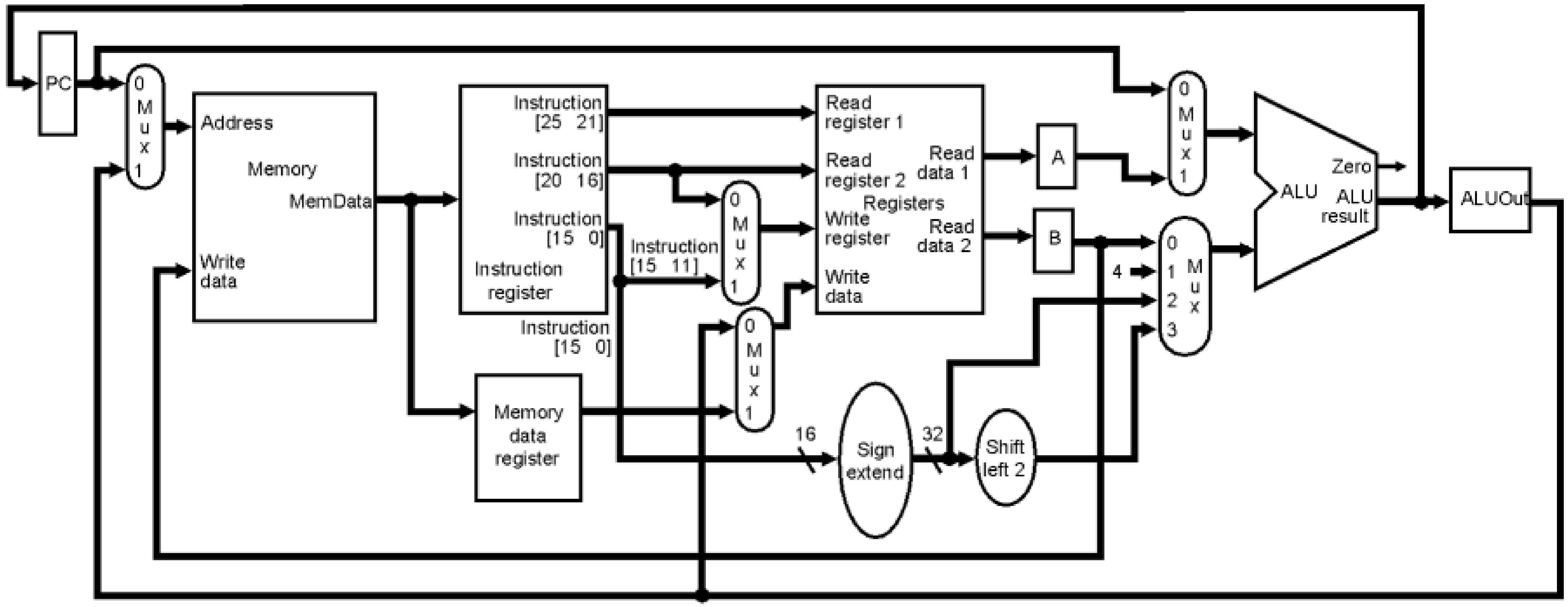
## Breaking Execution into Clock Cycles

- 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

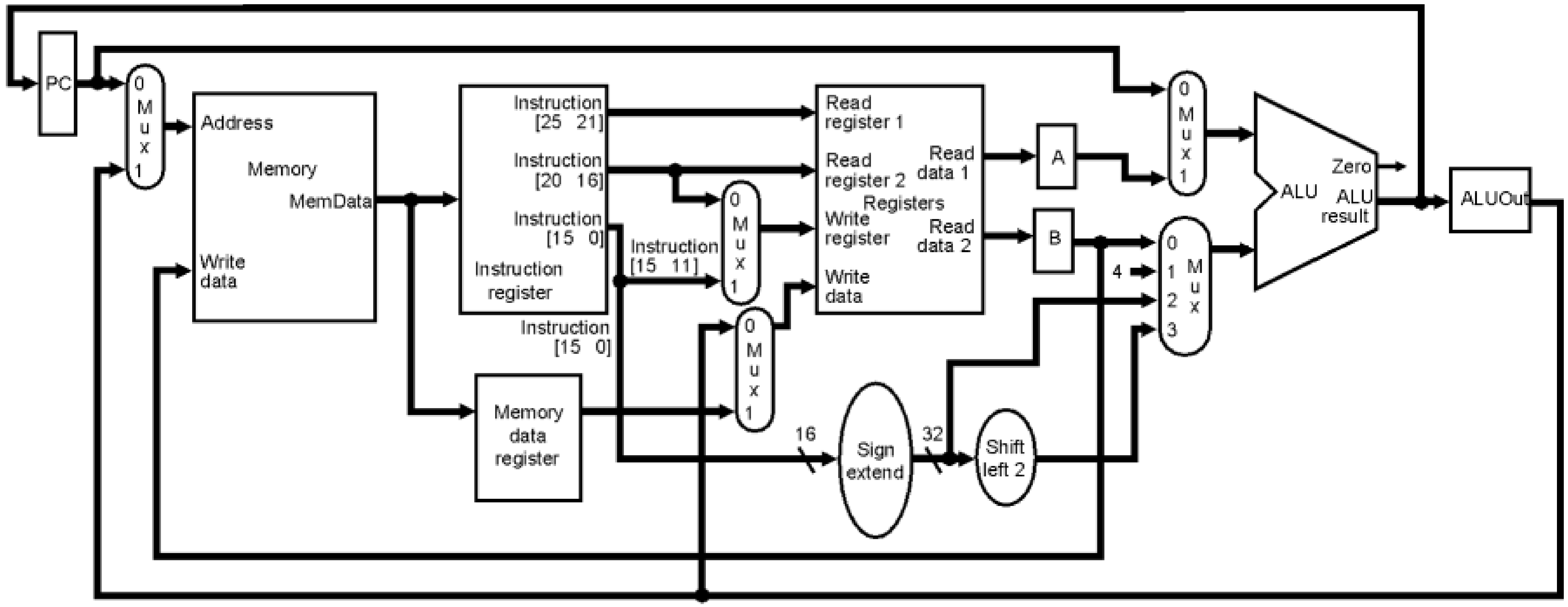


# Multi-Cycle Datapath



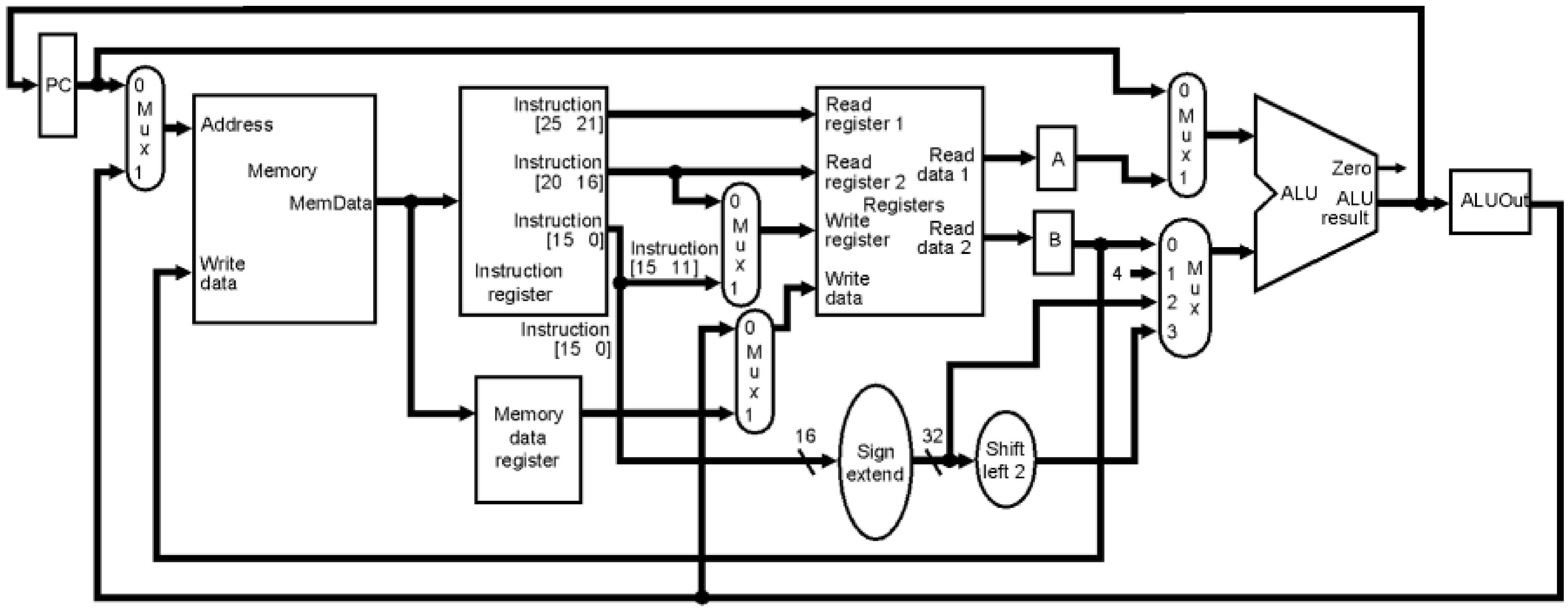
- **More Latches**

# Multi-Cycle Datapath



- **More Latches**
- **One ALU**

# Multi-Cycle Datapath



- **More Latches**
- **One ALU**
- **One Memory Unit**

## 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)

## 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)*

## 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)*

## 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*.

## 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

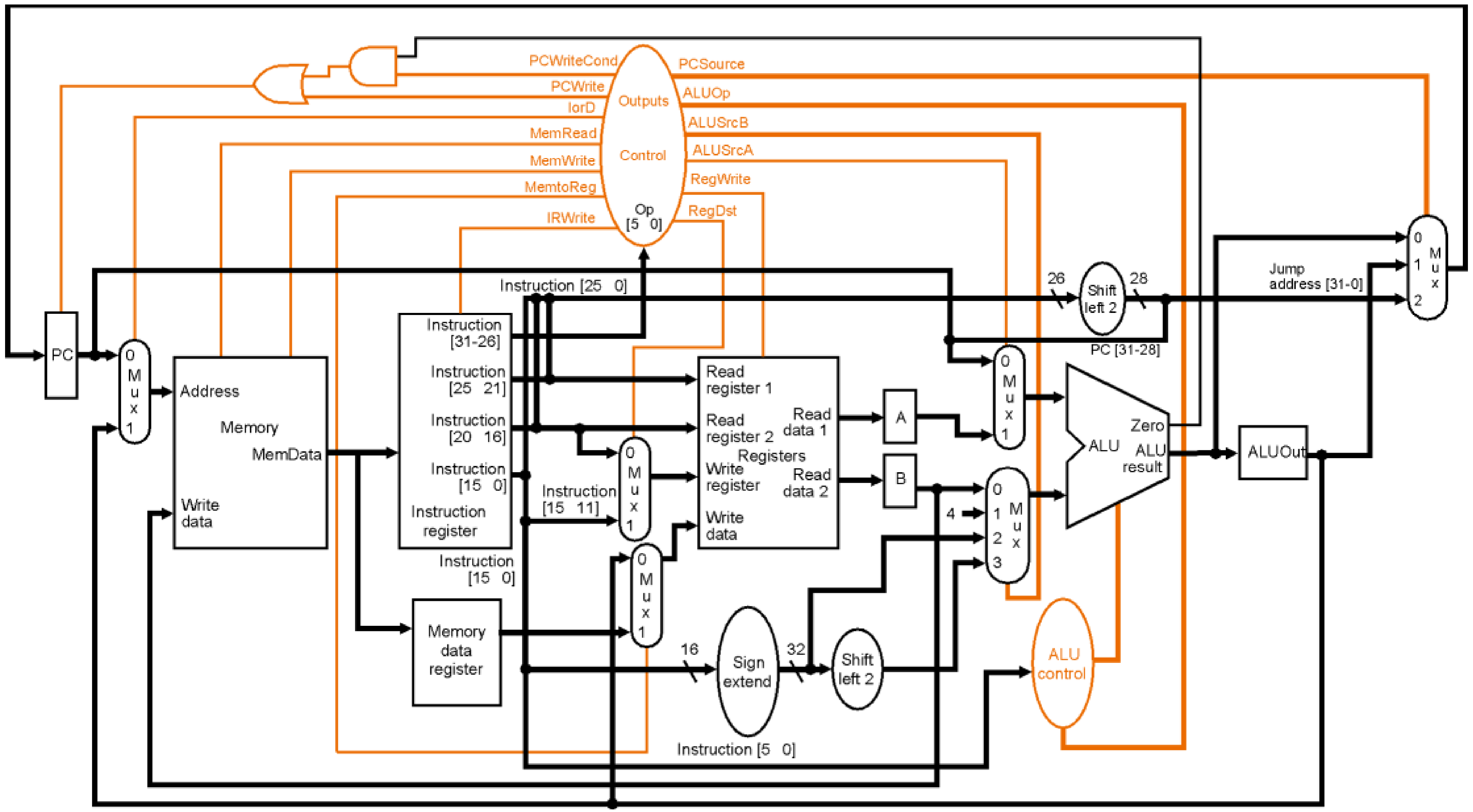
Reg[IR[20-16]] = MDR

*load is complete*

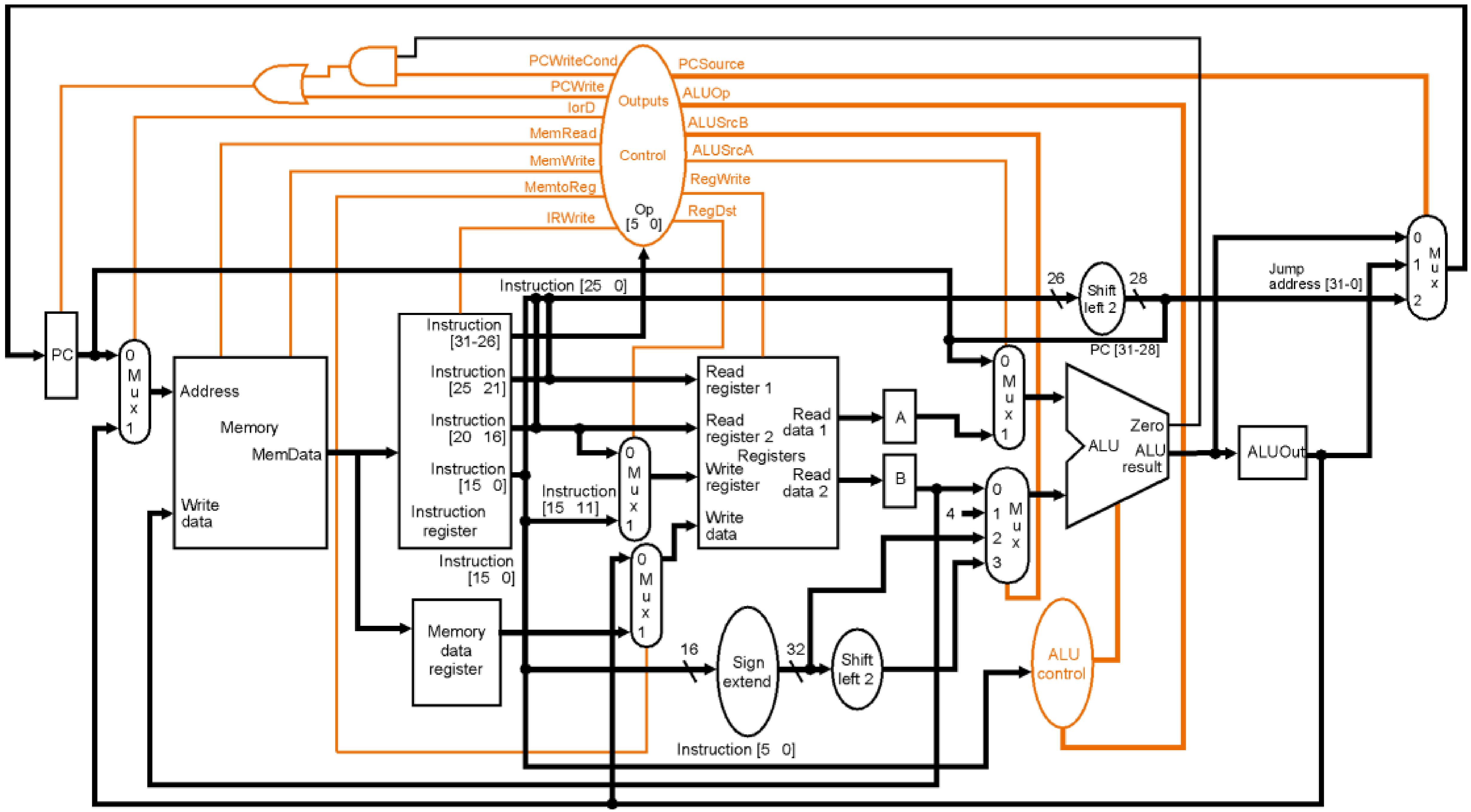
## Summary of Execution Steps

| Step | R-type | Memory | Branch |
|---|---|---|---|
| Instruction Fetch | IR = Mem[PC]<br>PC = PC + 4 | | |
| Instruction Decode/ register fetch | A = Reg[IR[25-21]]<br>B = Reg[IR[20-16]]<br>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]<br>*or*<br>Mem[ALUout]= B | |
| Write-back | | Reg[IR[20-16]] = memory-data | |

# Complete Multi-Cycle Datapath

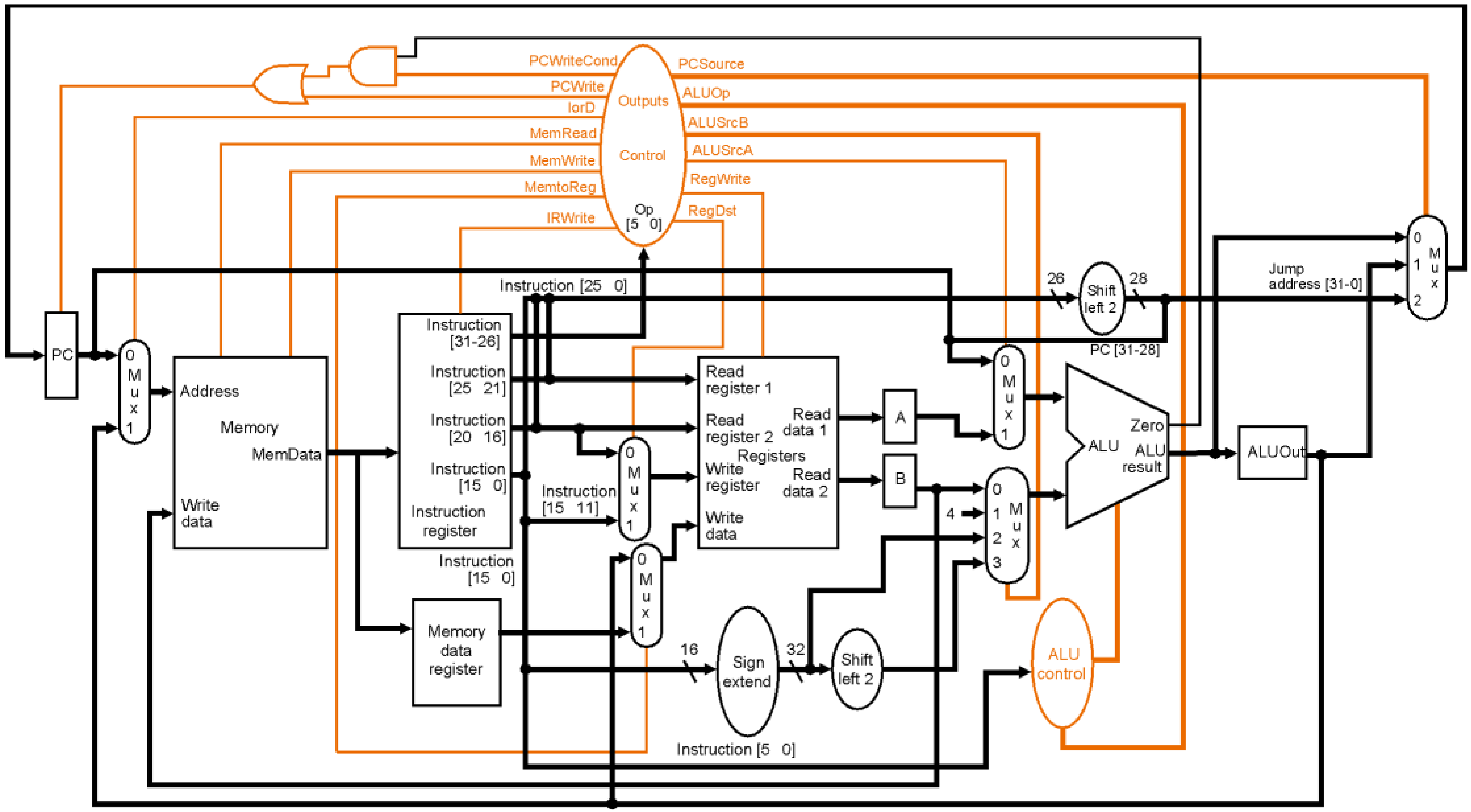


# Complete Multi-Cycle Datapath



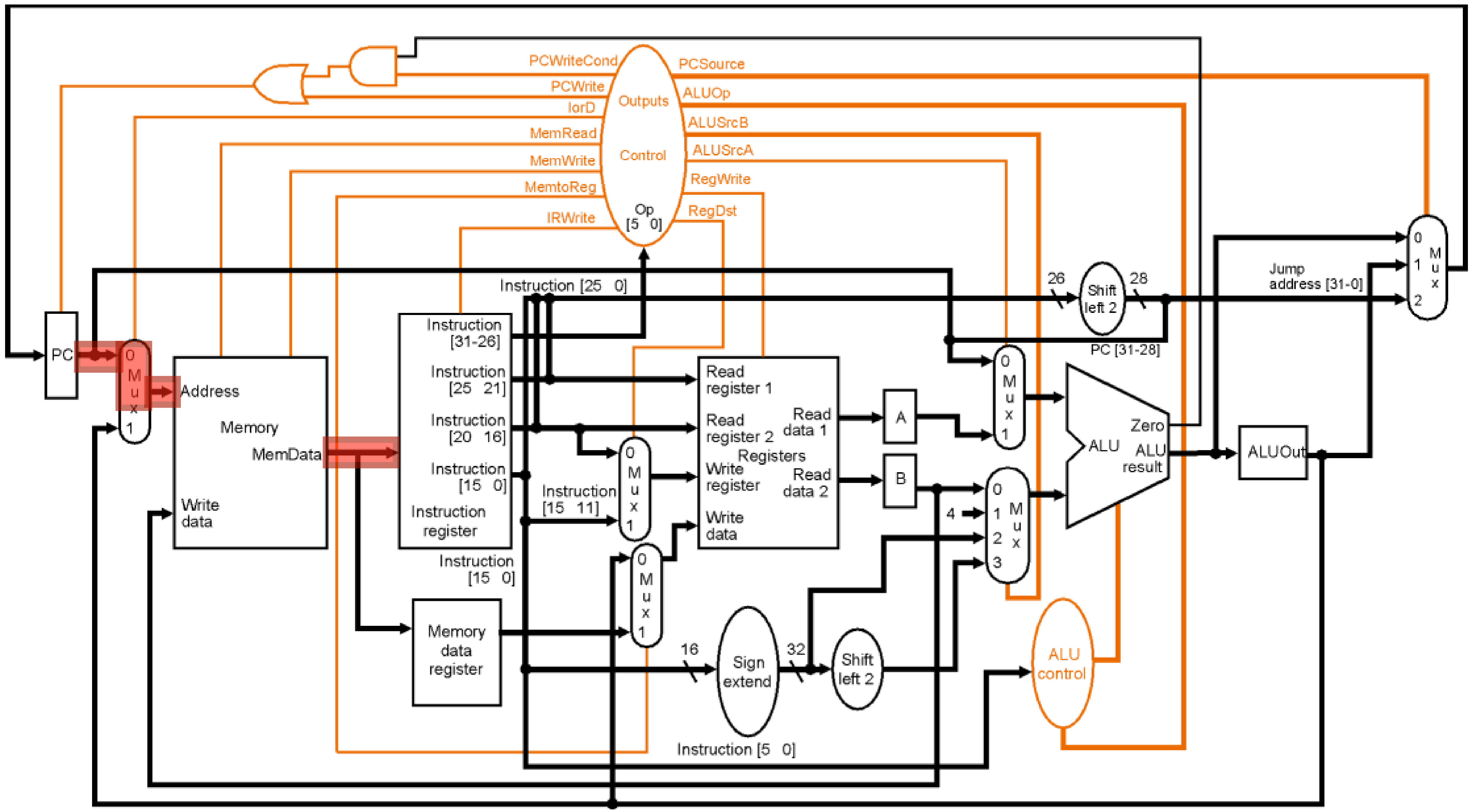
New Instruction Appears Out of Nowhere? Which One?

# 1. Instruction Fetch



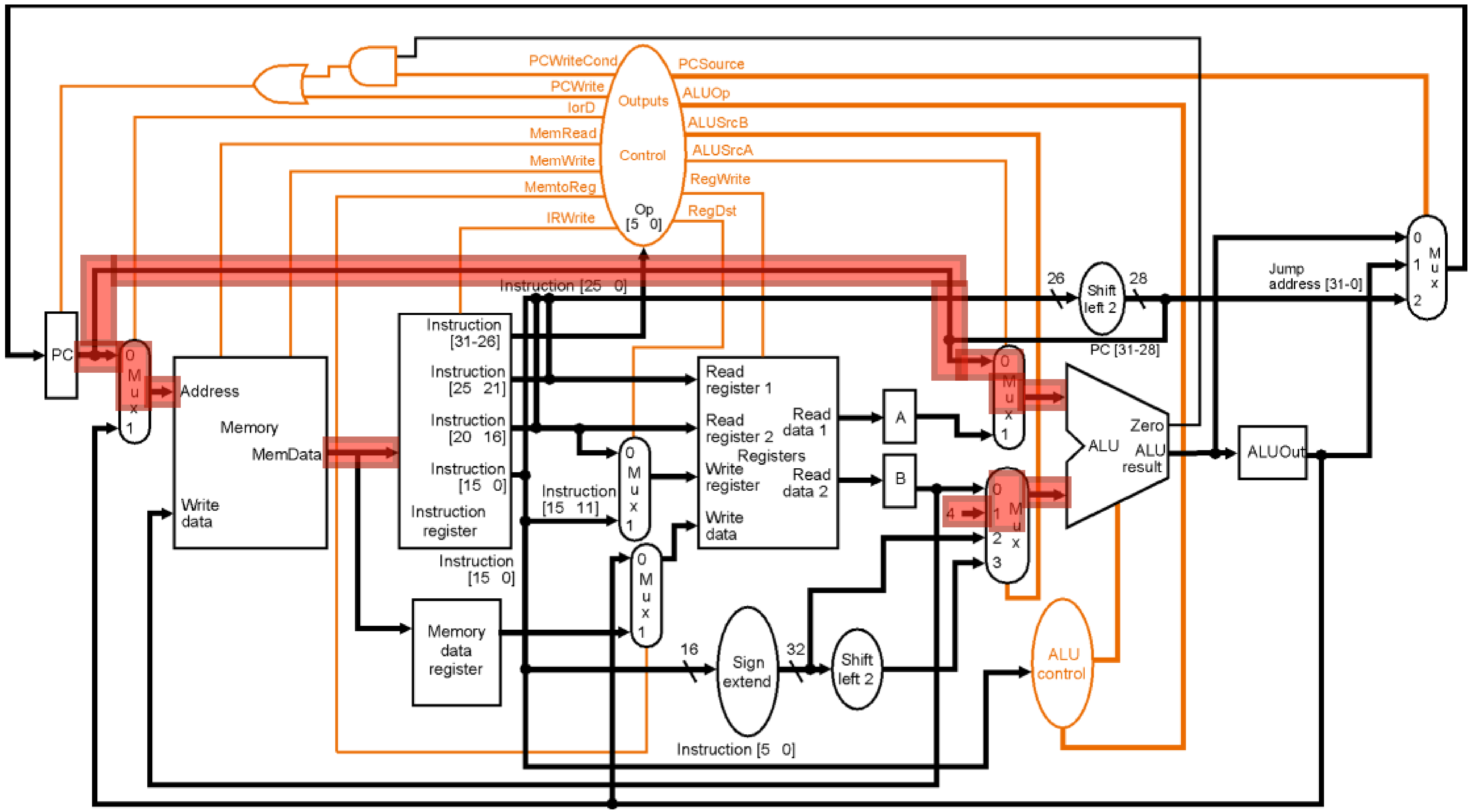
**IR = Memory[PC]**
**PC = PC + 4**

# 1. Instruction Fetch



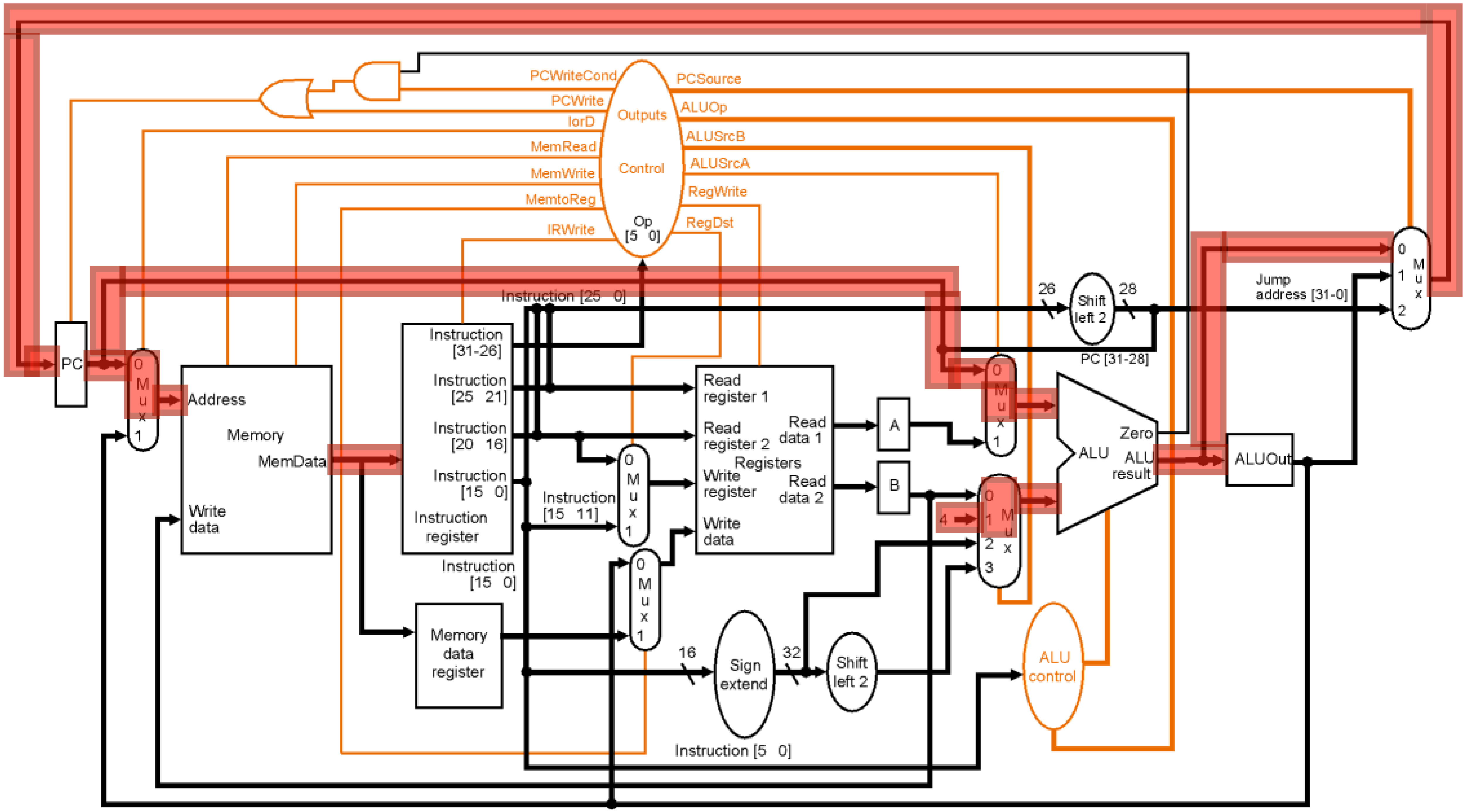
**IR = Memory[PC]**
**PC = PC + 4**

# 1. Instruction Fetch



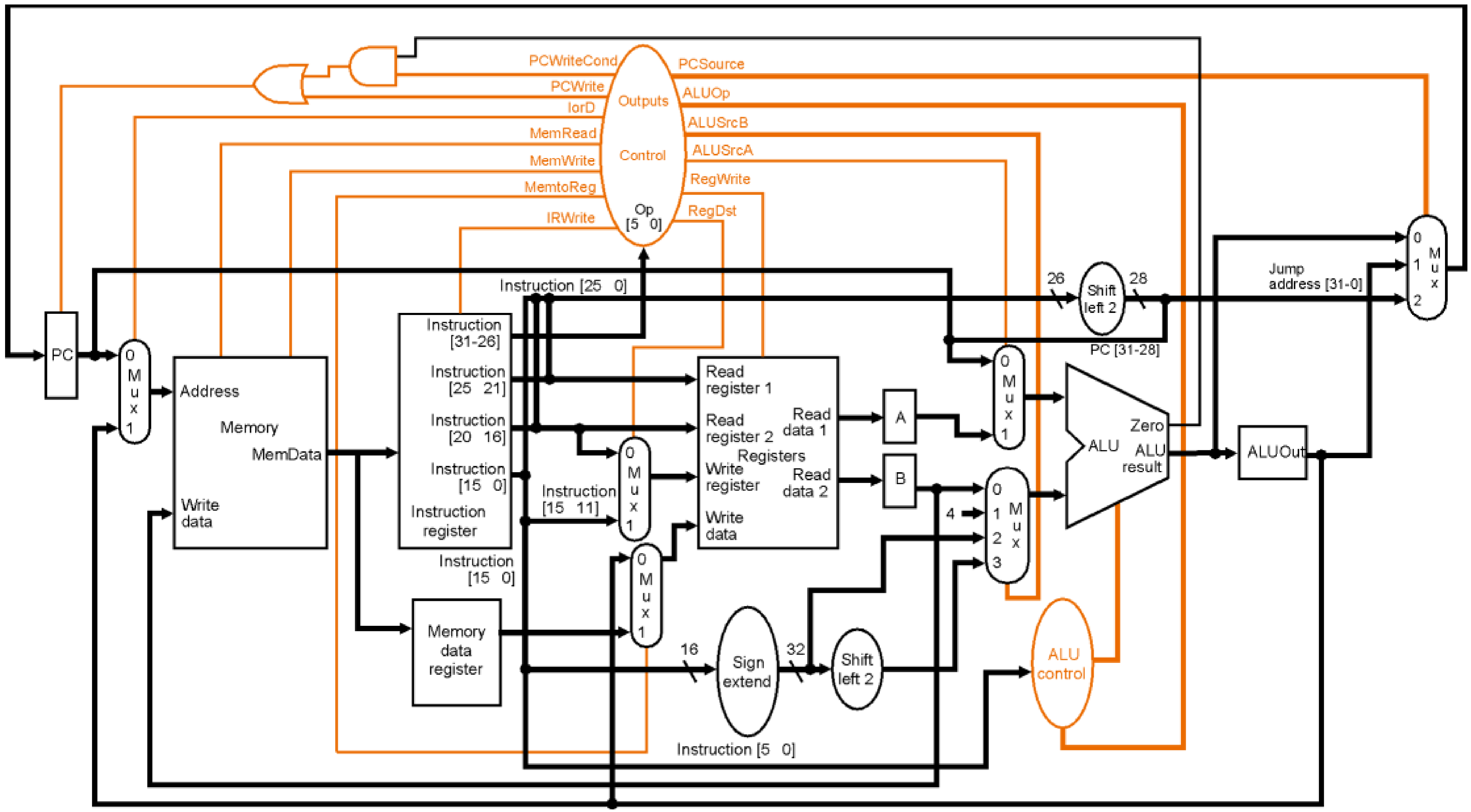
**IR = Memory[PC]**
**PC = PC + 4**

# 1. Instruction Fetch



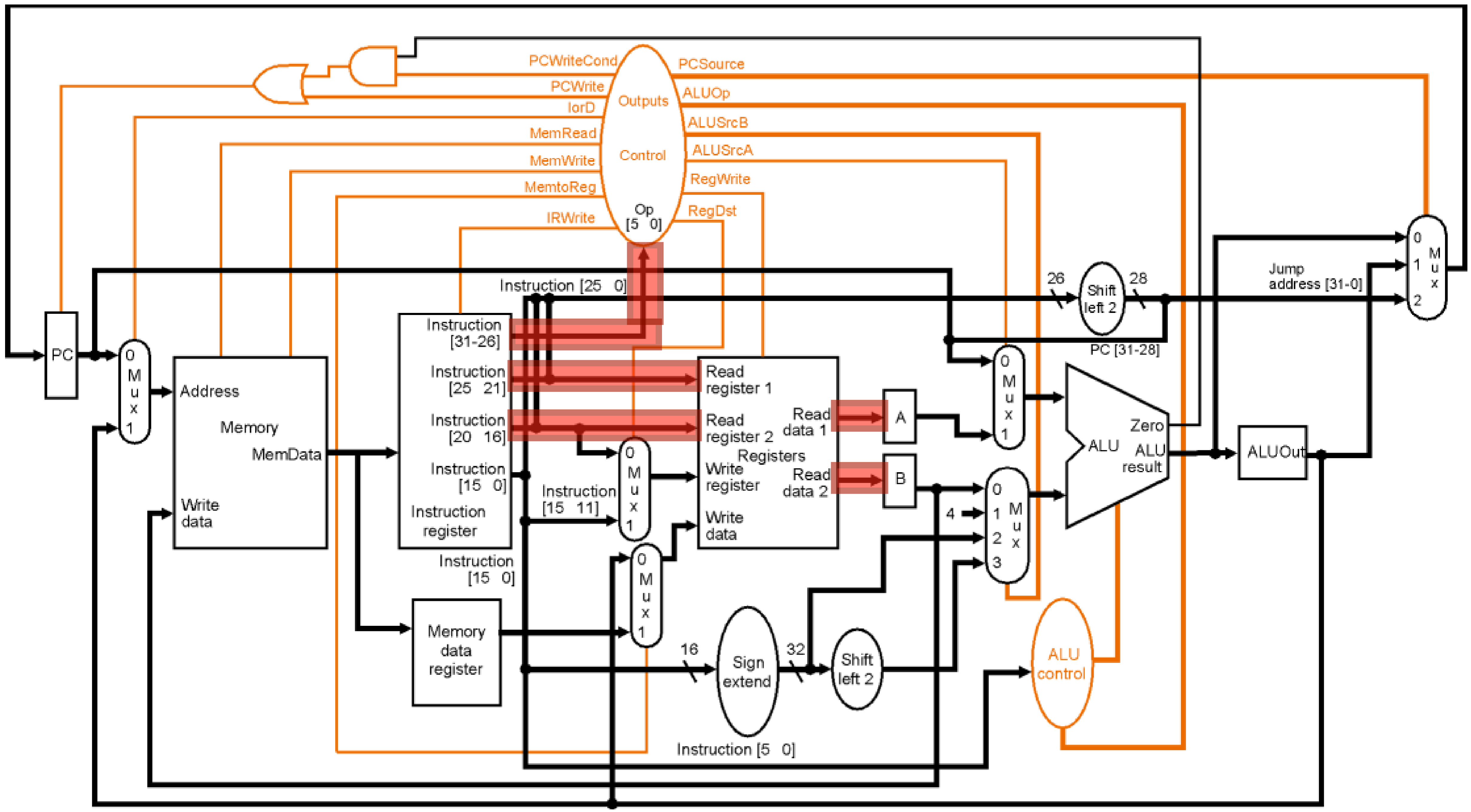
**IR = Memory[PC]**
**PC = PC + 4**

## 2. Instruction Decode and Register Fetch



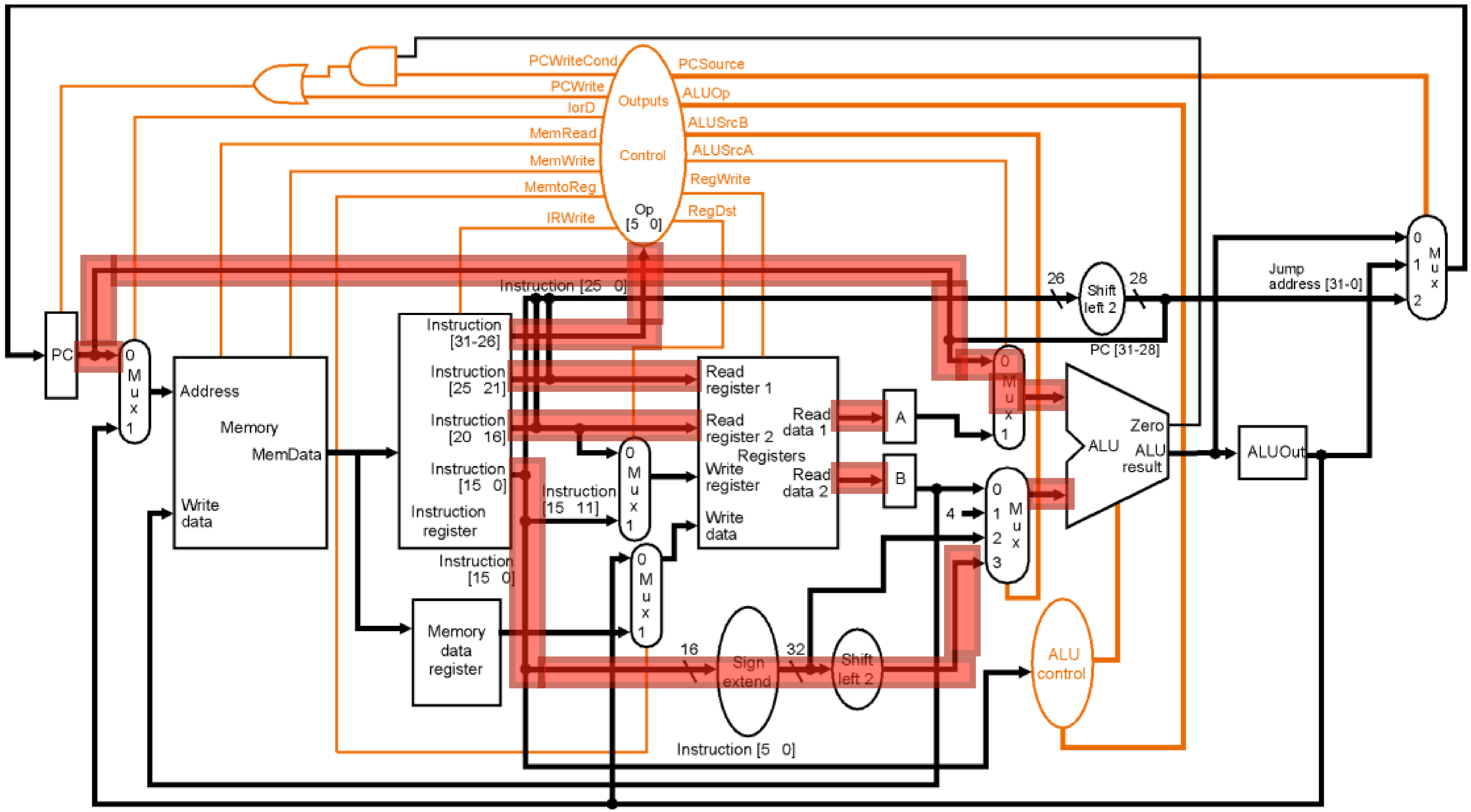
**A = Register[IR[25-21]]**
**B = Register[IR[20-16]]**
**ALUOut = PC + (sign-extend (IR[15-0]) << 2)**

## 2. Instruction Decode and Register Fetch



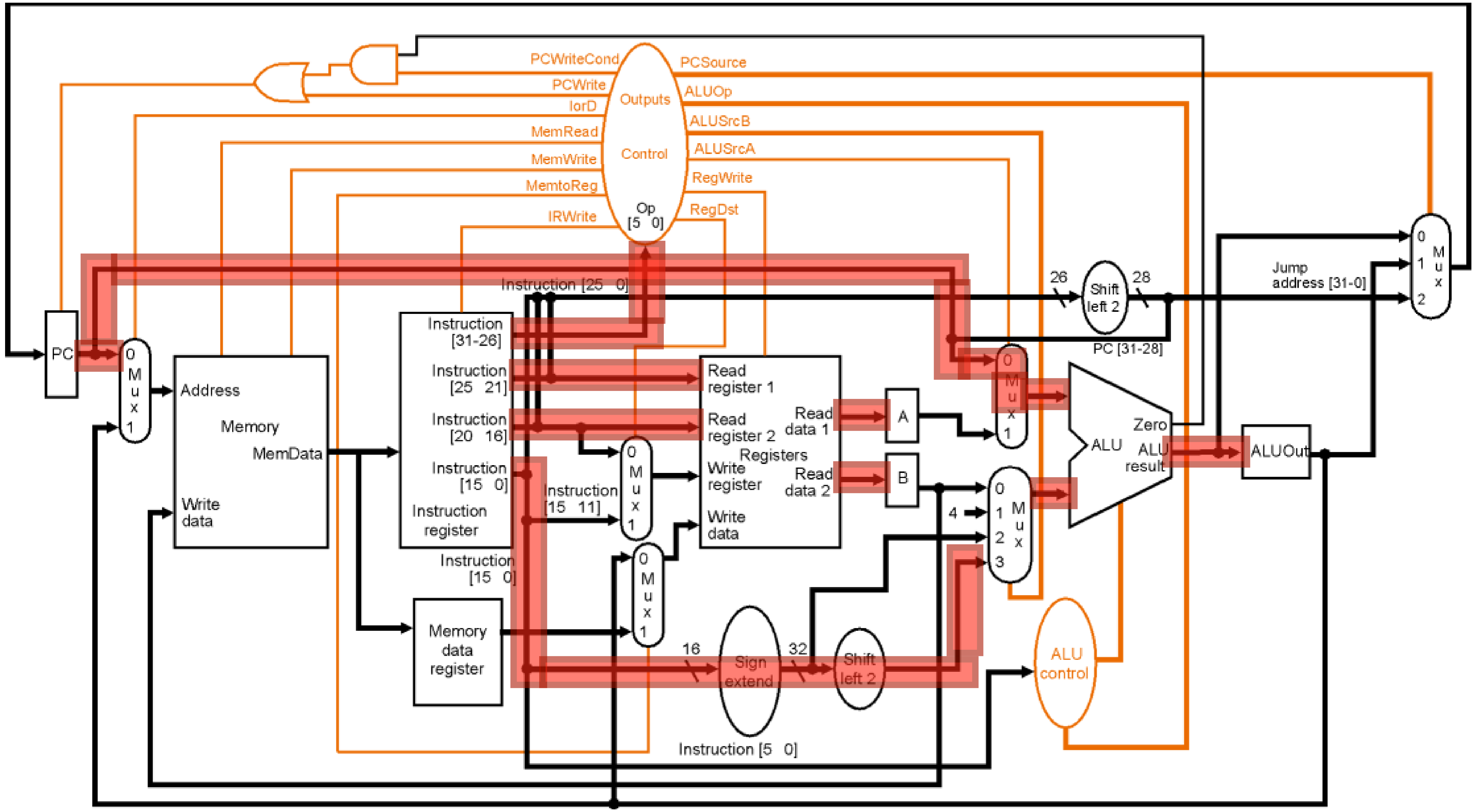
**A = Register[IR[25-21]]**
**B = Register[IR[20-16]]**
**ALUOut = PC + (sign-extend (IR[15-0]) << 2)**

# 2. Instruction Decode and Register Fetch



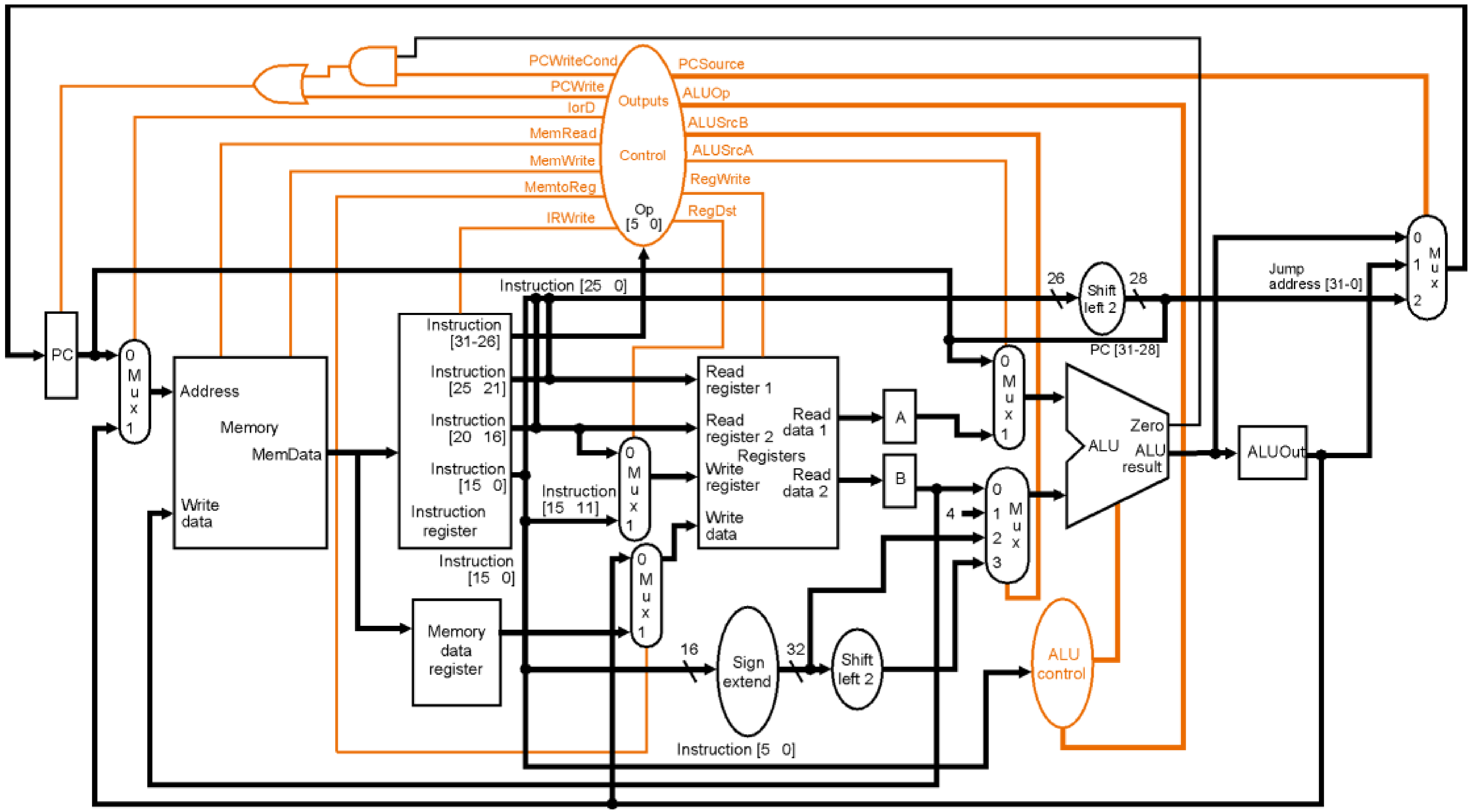
**A = Register[IR[25-21]]**
**B = Register[IR[20-16]]**
**ALUOut = PC + (sign-extend (IR[15-0]) << 2)**

# 2. Instruction Decode and Register Fetch



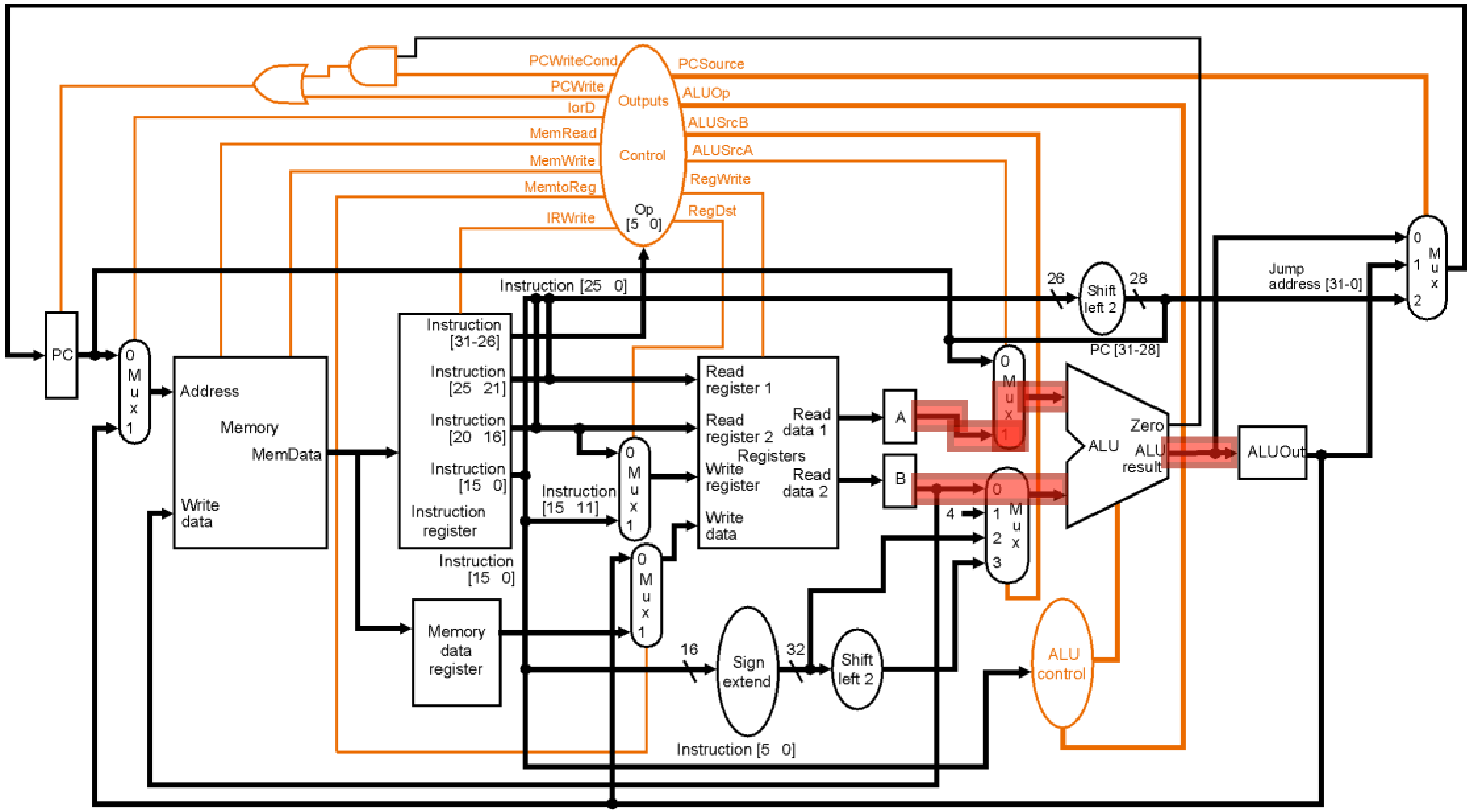
**A = Register[IR[25-21]]**
**B = Register[IR[20-16]]**
**ALUOut = PC + (sign-extend (IR[15-0]) << 2)**

# 3. Execution (R-Type)



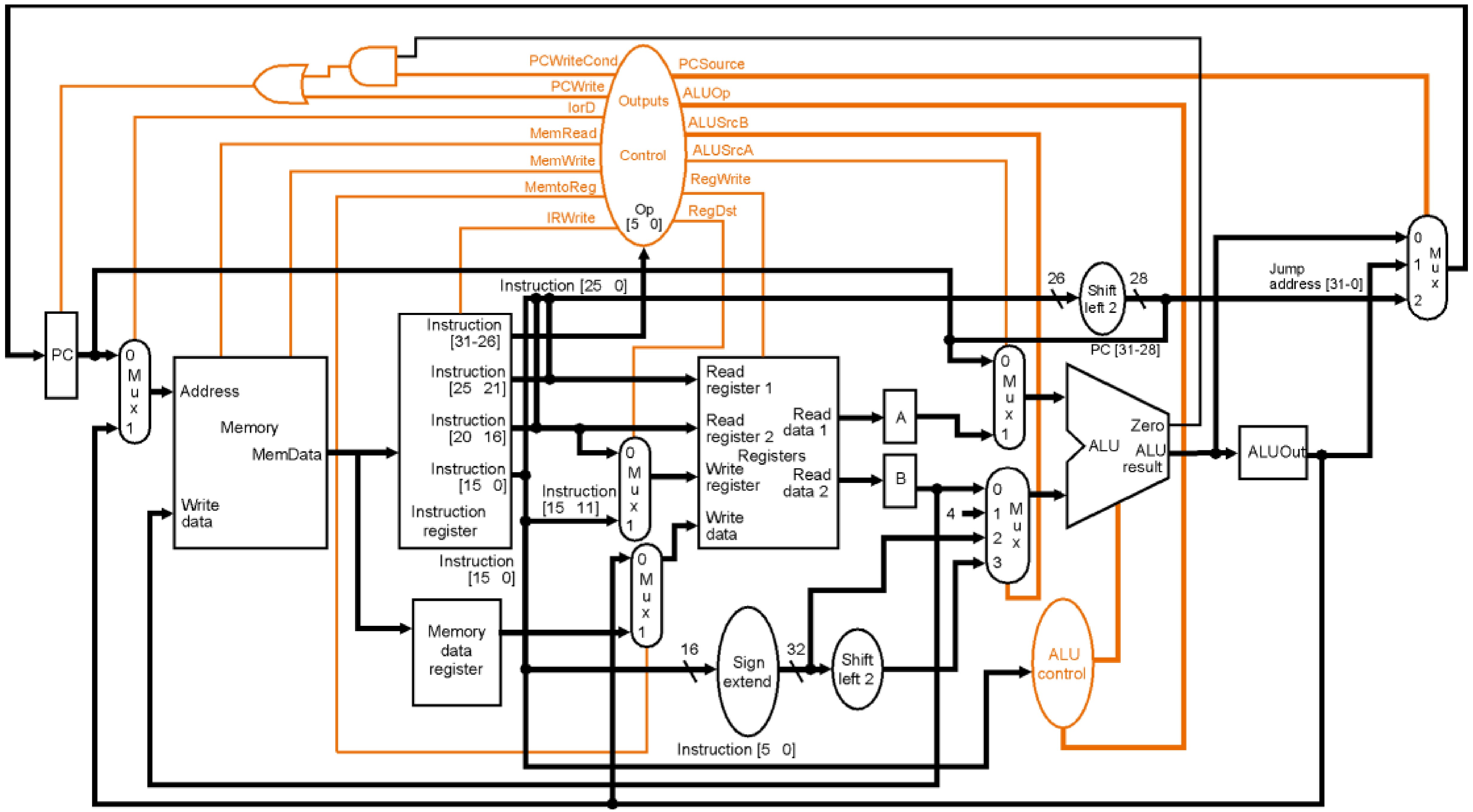
**ALUout = A op B**

# 3. Execution (R-Type)



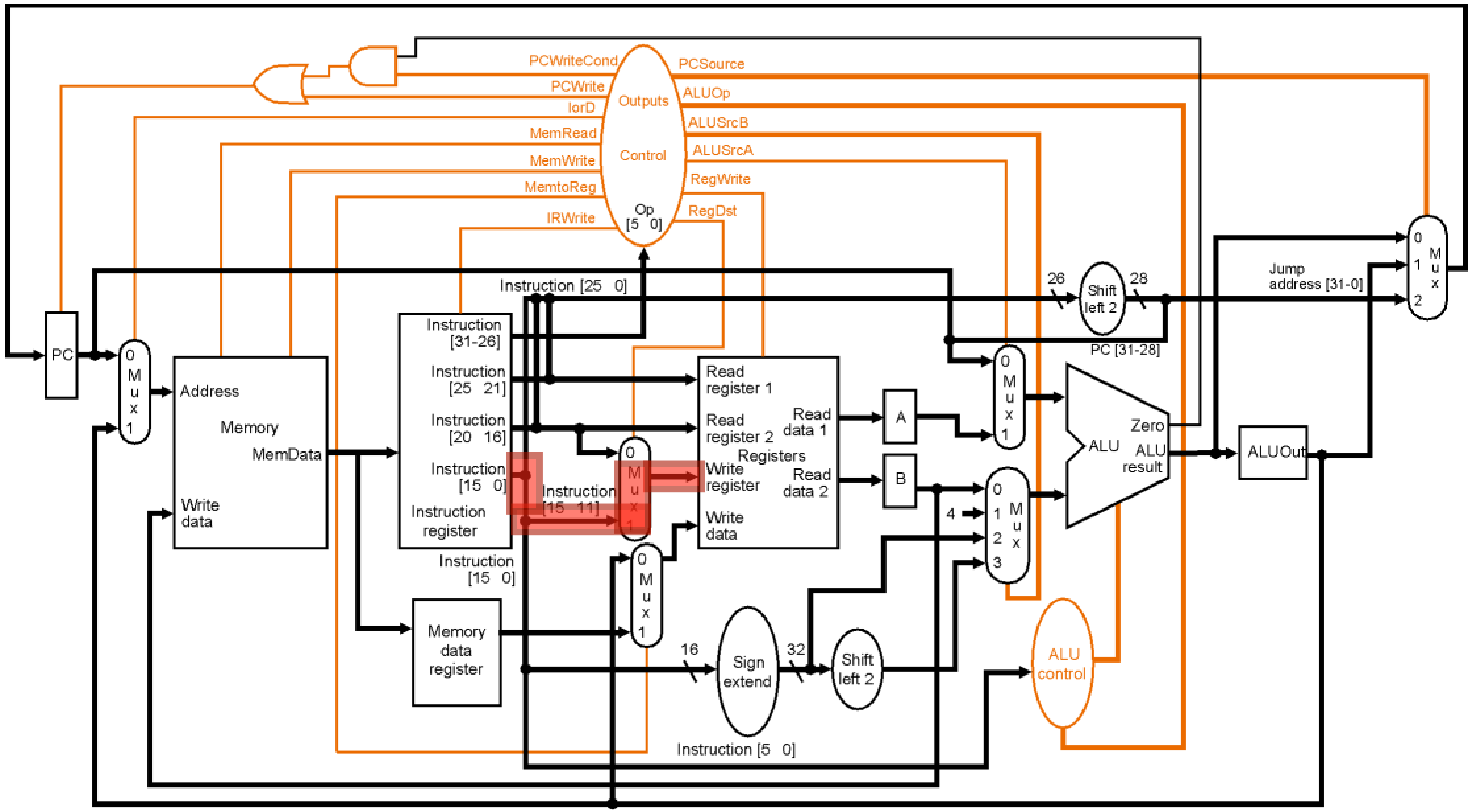
**ALUout = A op B**

# 4. R-Type Completion



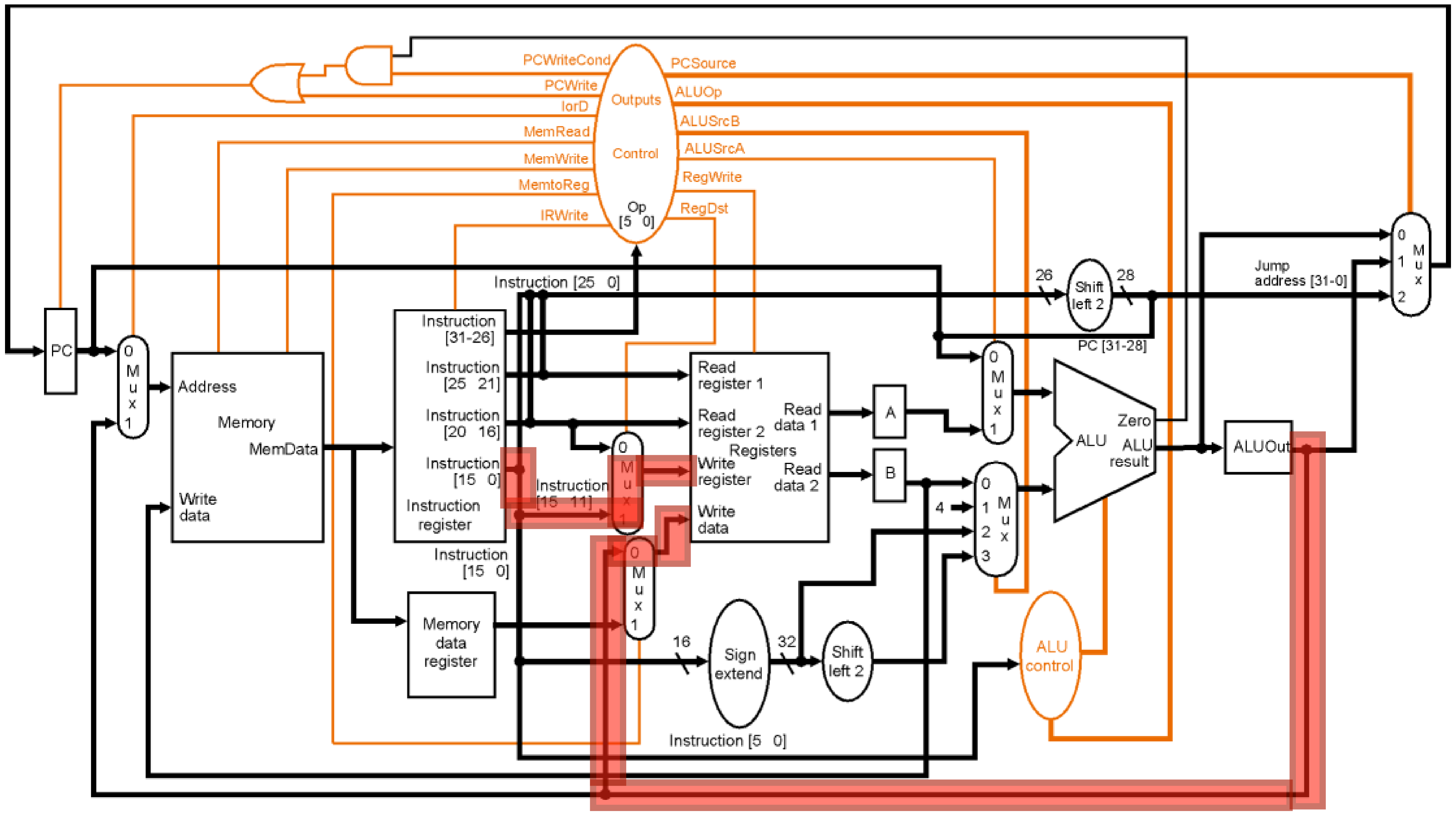
**Reg[IR[15-11]] = ALUout**

# 4. R-Type Completion



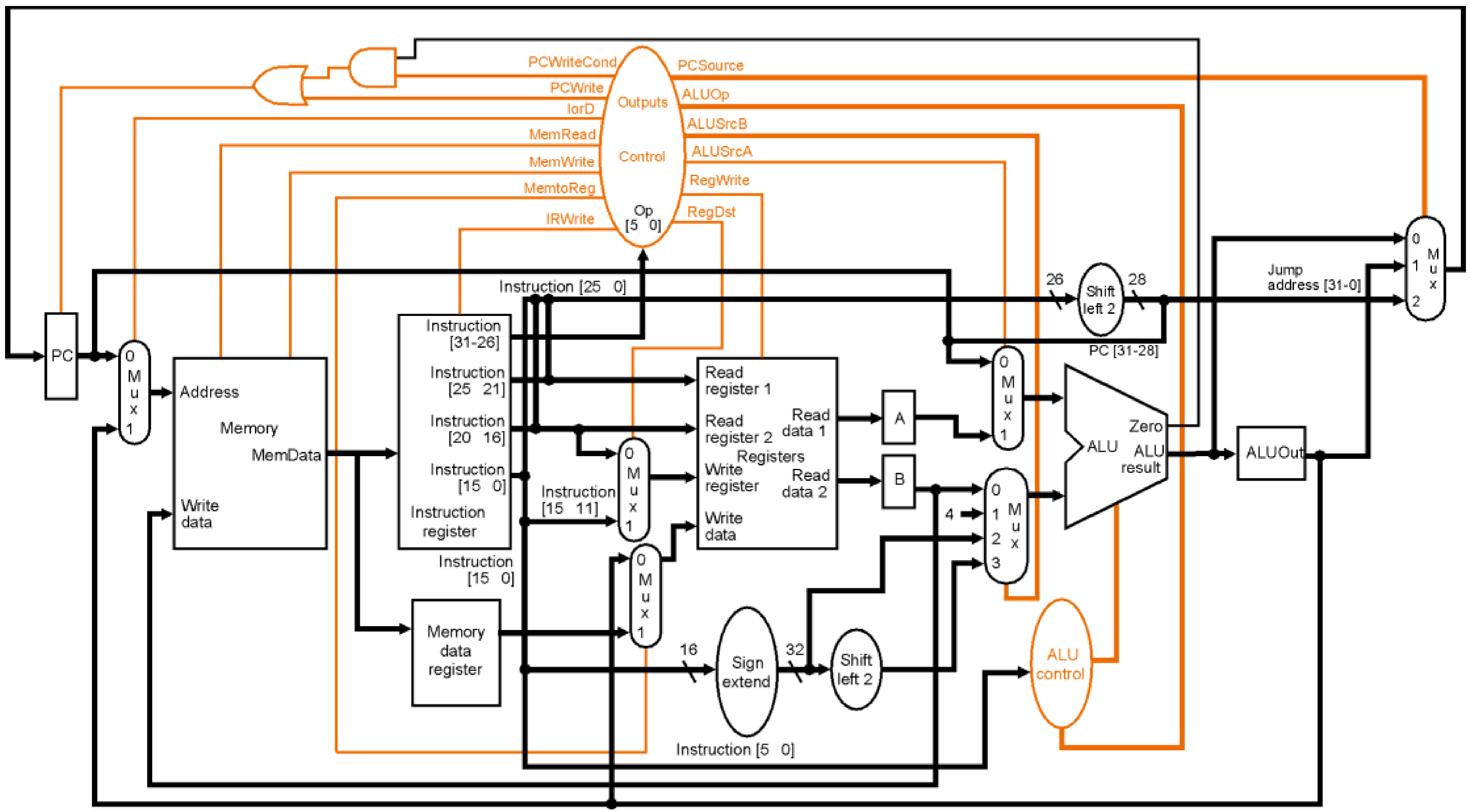
**Reg[IR[15-11]] = ALUout**

# 4. R-Type Completion



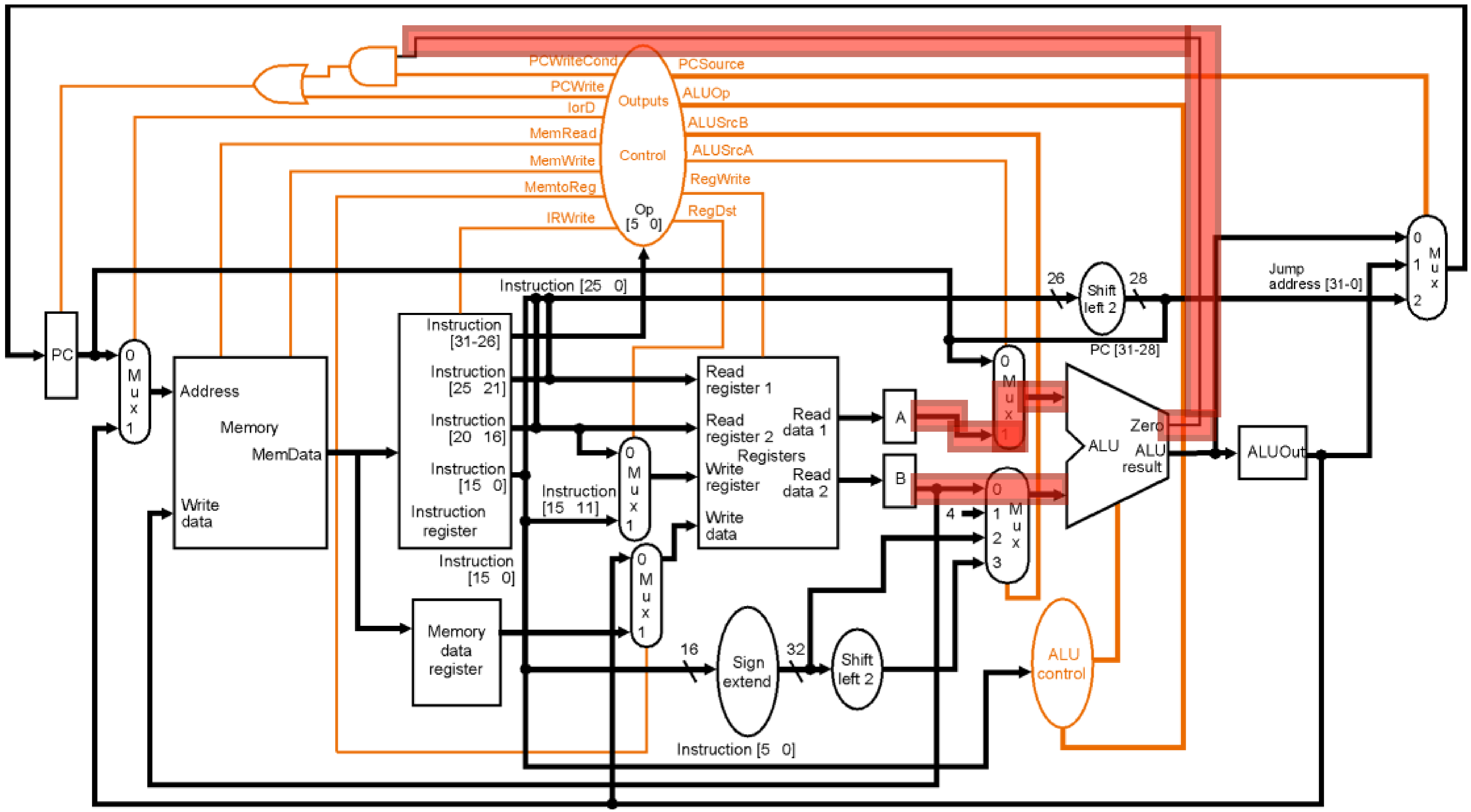
**Reg[IR[15-11]] = ALUout**

# 3. Branch Completion



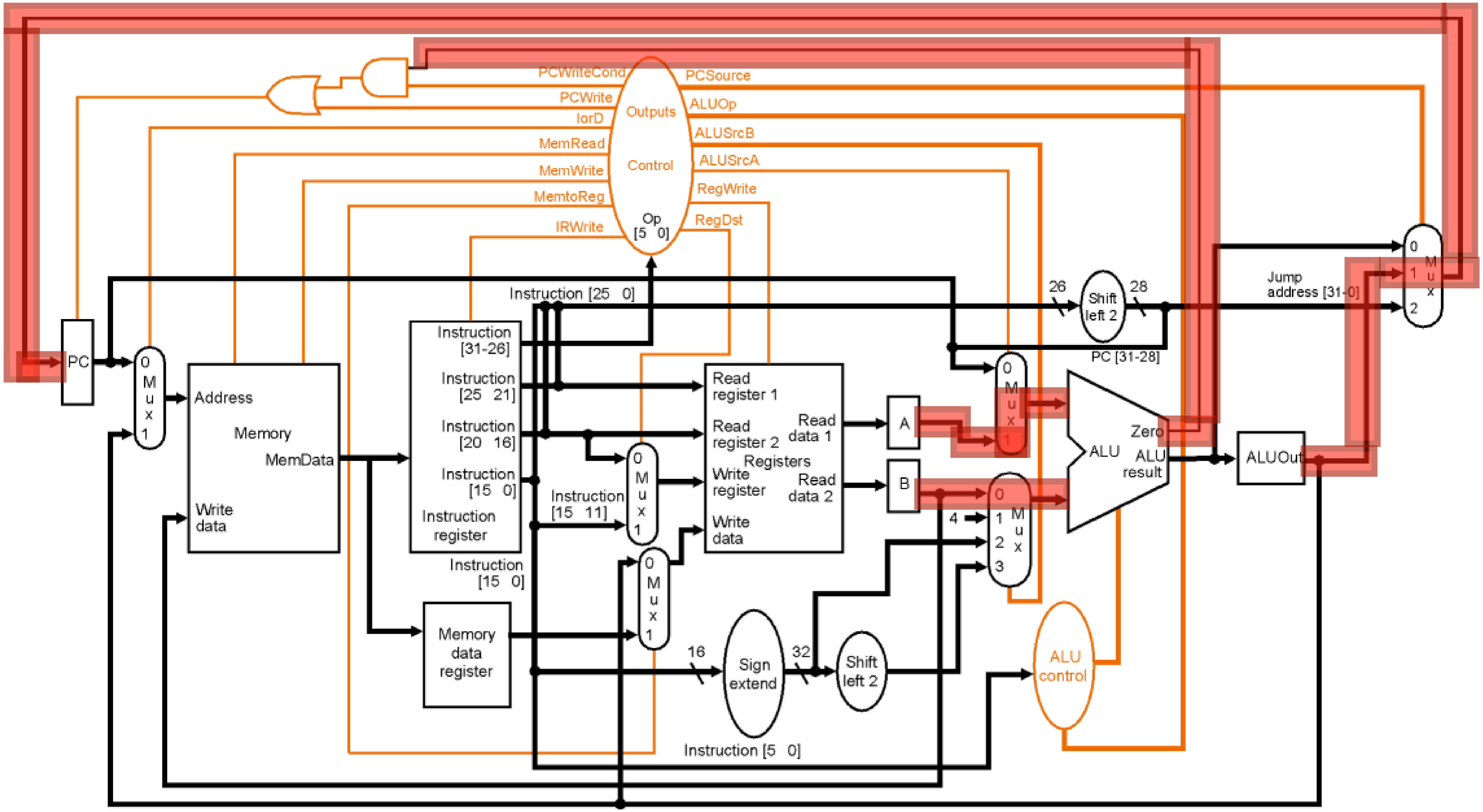
**if (A == B) PC = ALUOut**

# 3. Branch Completion



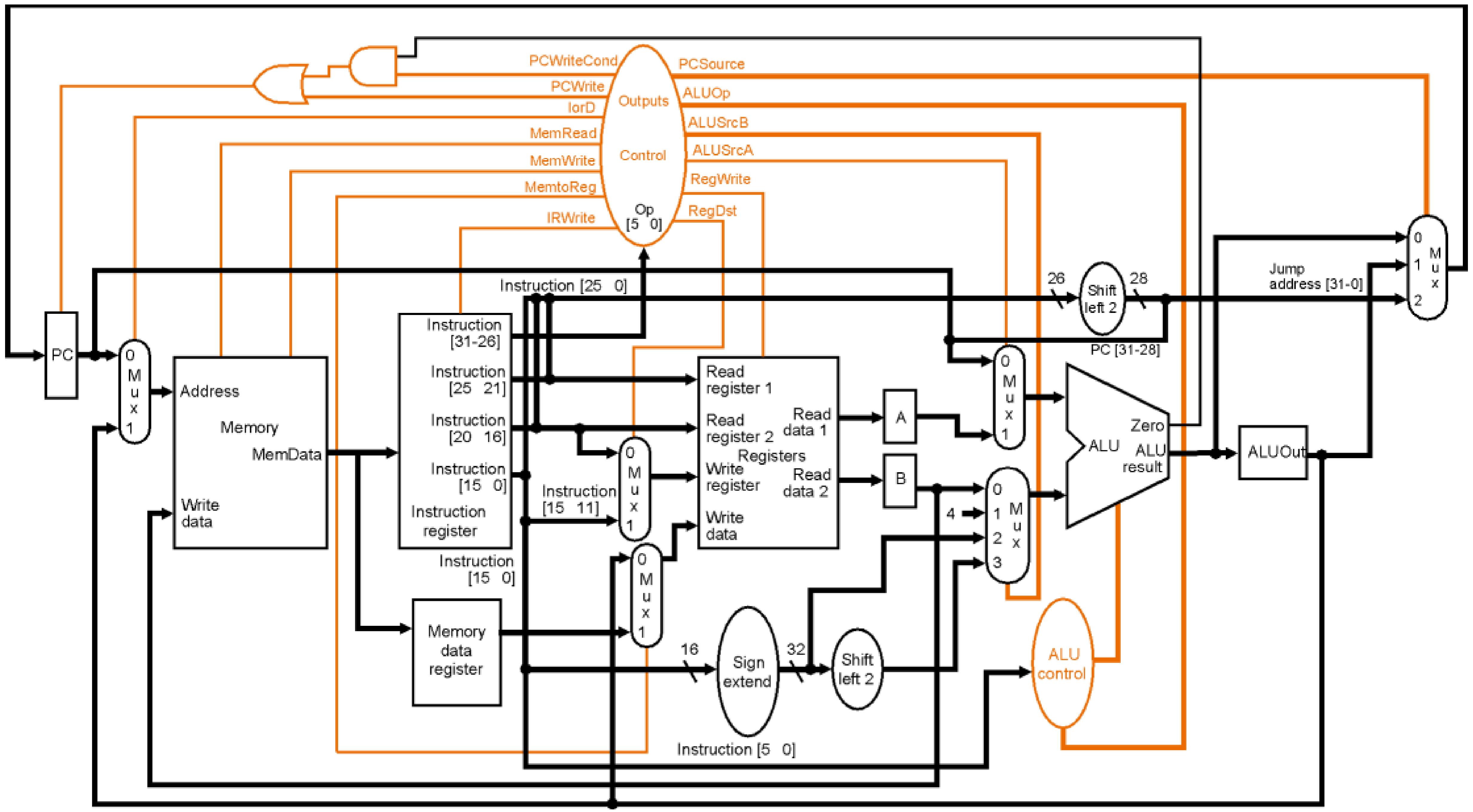
**if (A == B) PC = ALUOut**

# 3. Branch Completion



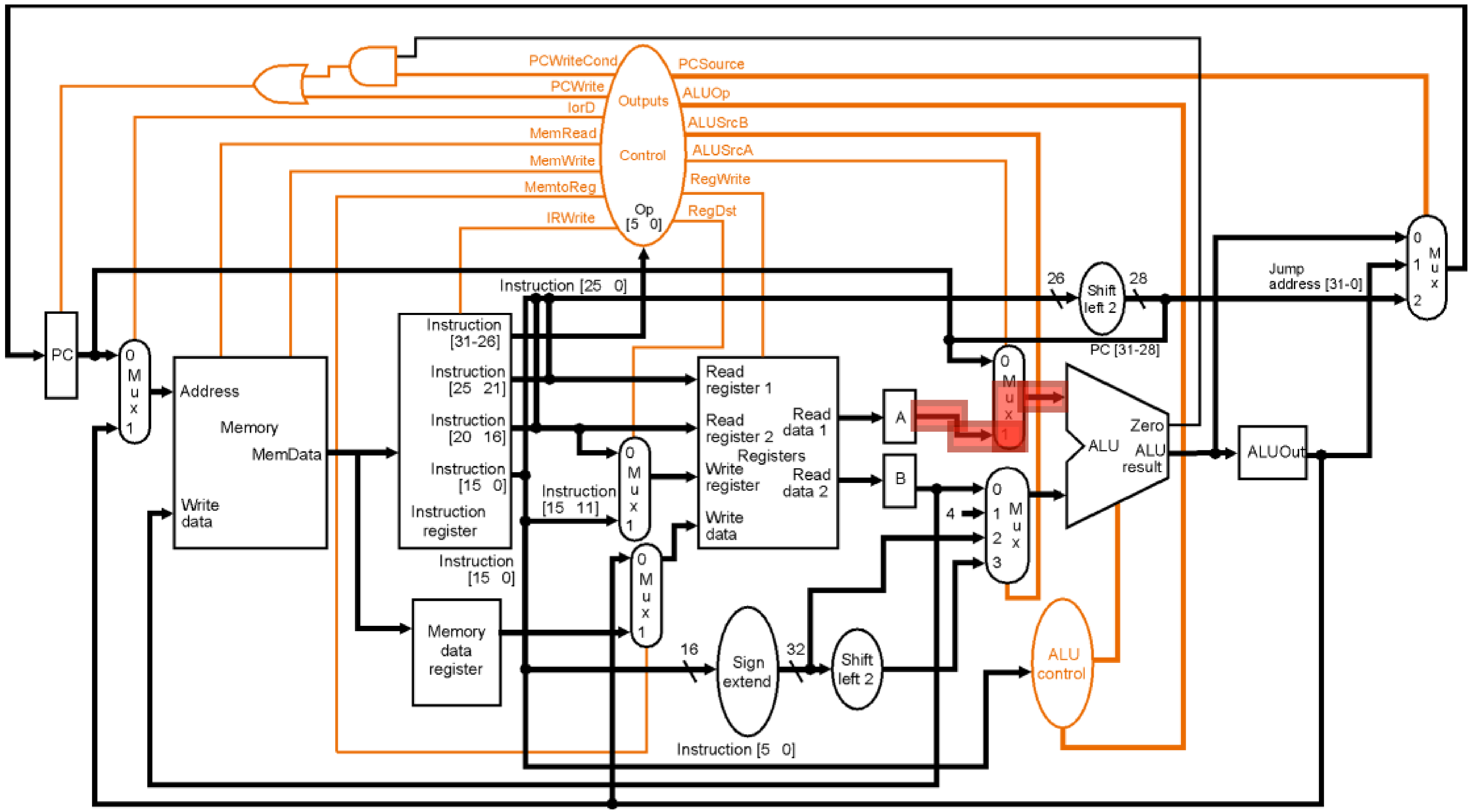
**if (A == B) PC = ALUOut**

# 4. Memory Address Computation



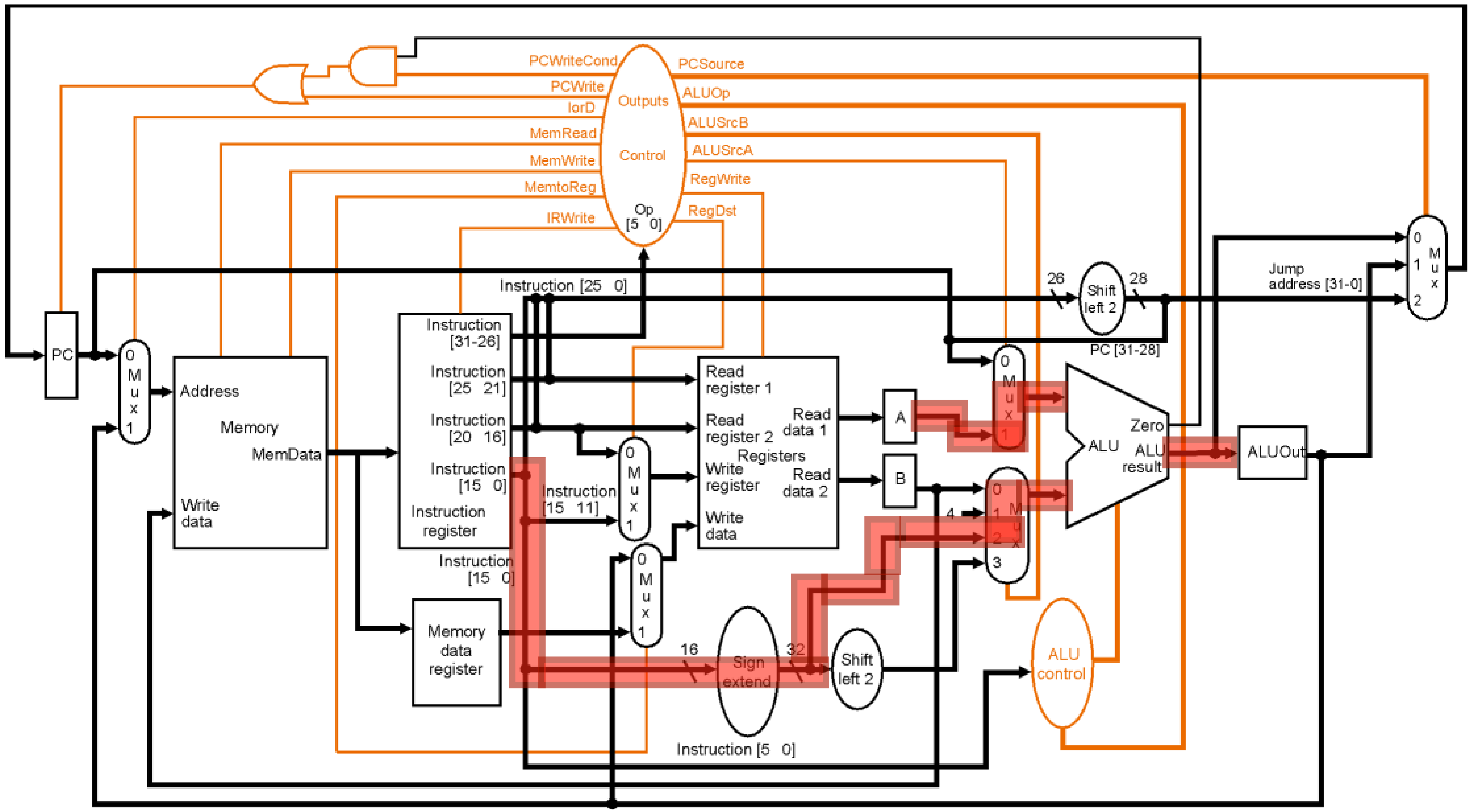
**ALUout = A + sign-extend(IR[15-0])**

# 4. Memory Address Computation



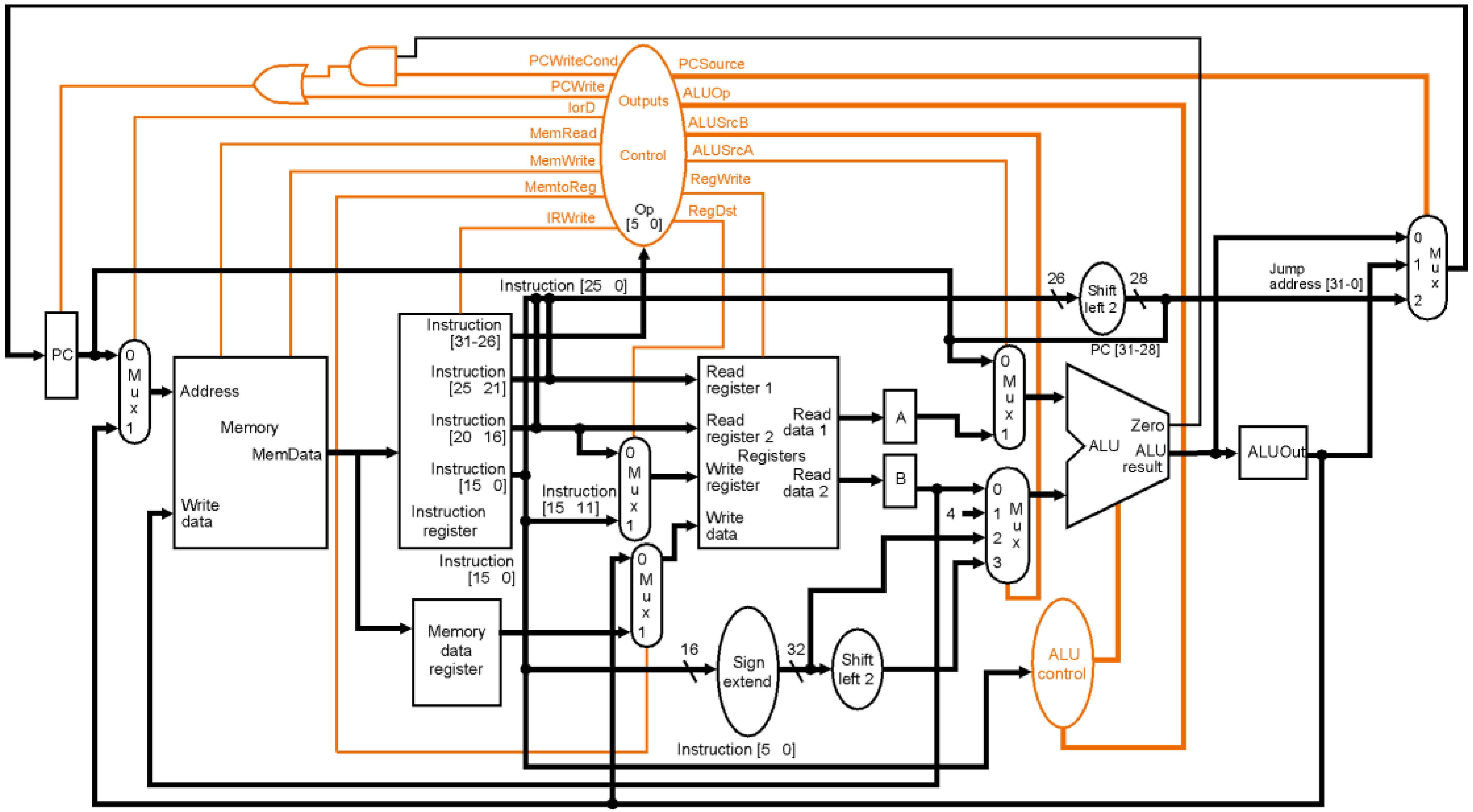
**ALUout = A + sign-extend(IR[15-0])**

# 4. Memory Address Computation



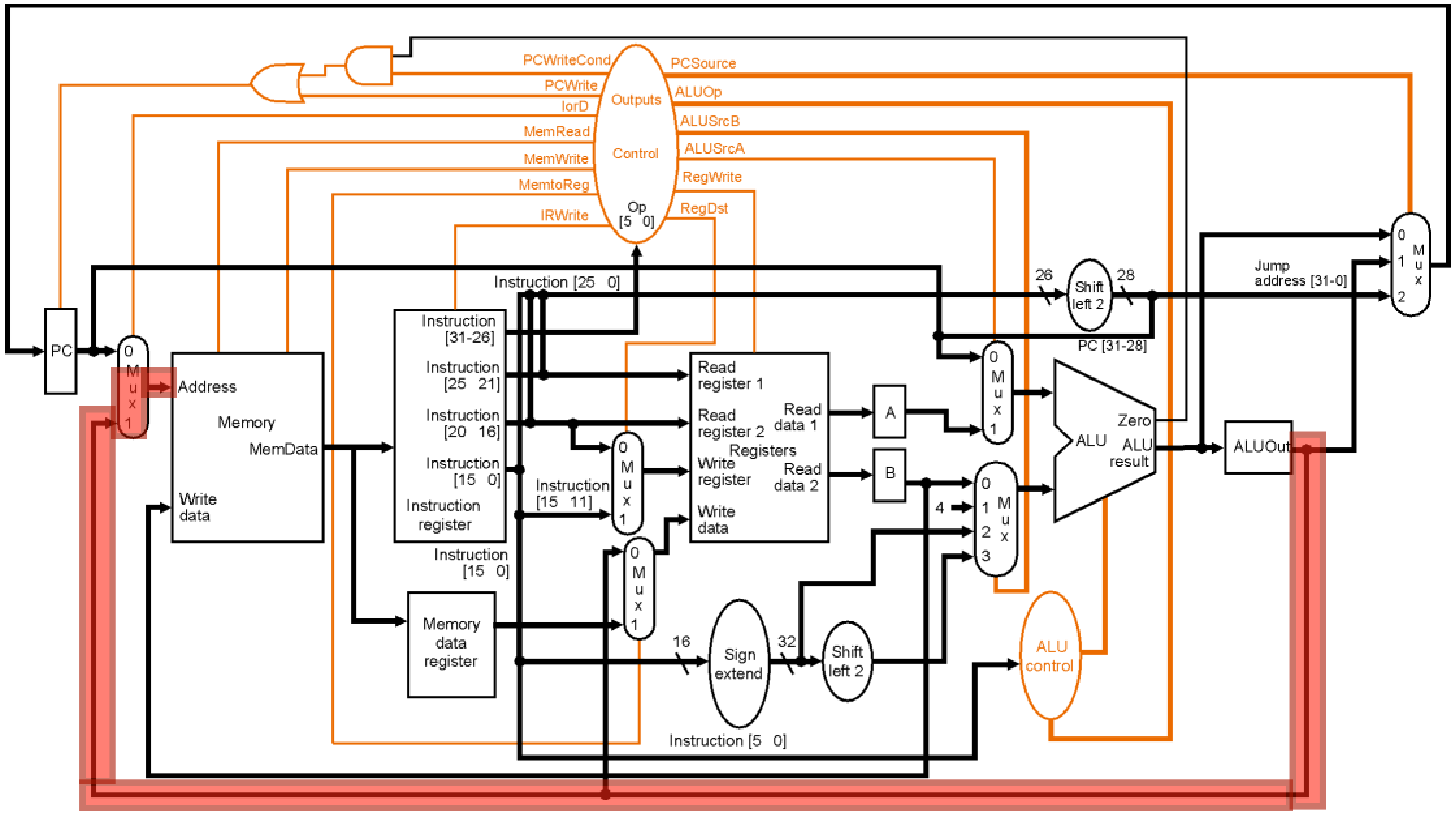
**ALUout = A + sign-extend(IR[15-0])**

# 4. Memory Access Load



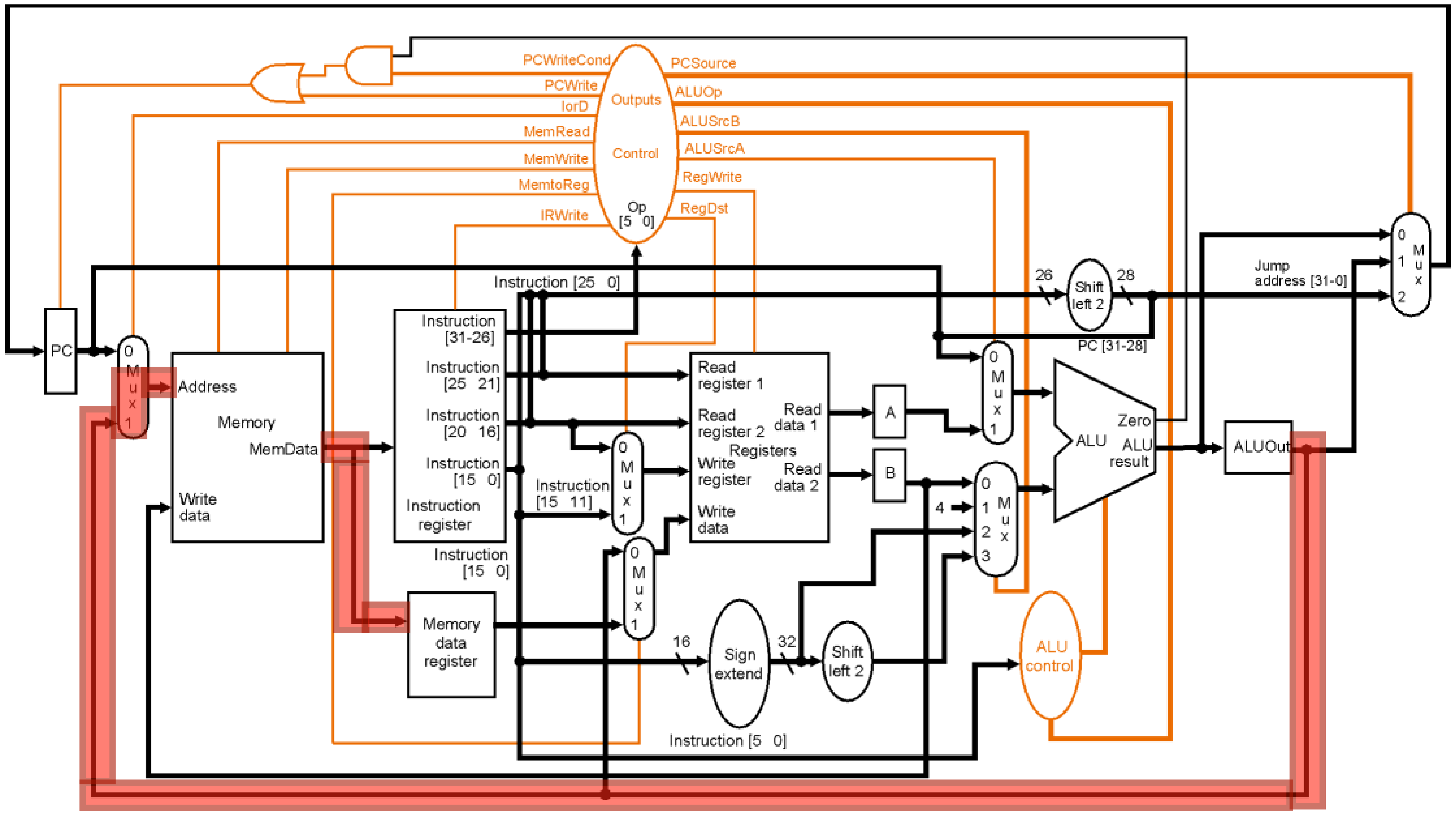
**memory-data = Memory[ALUout]**

# 4. Memory Access Load



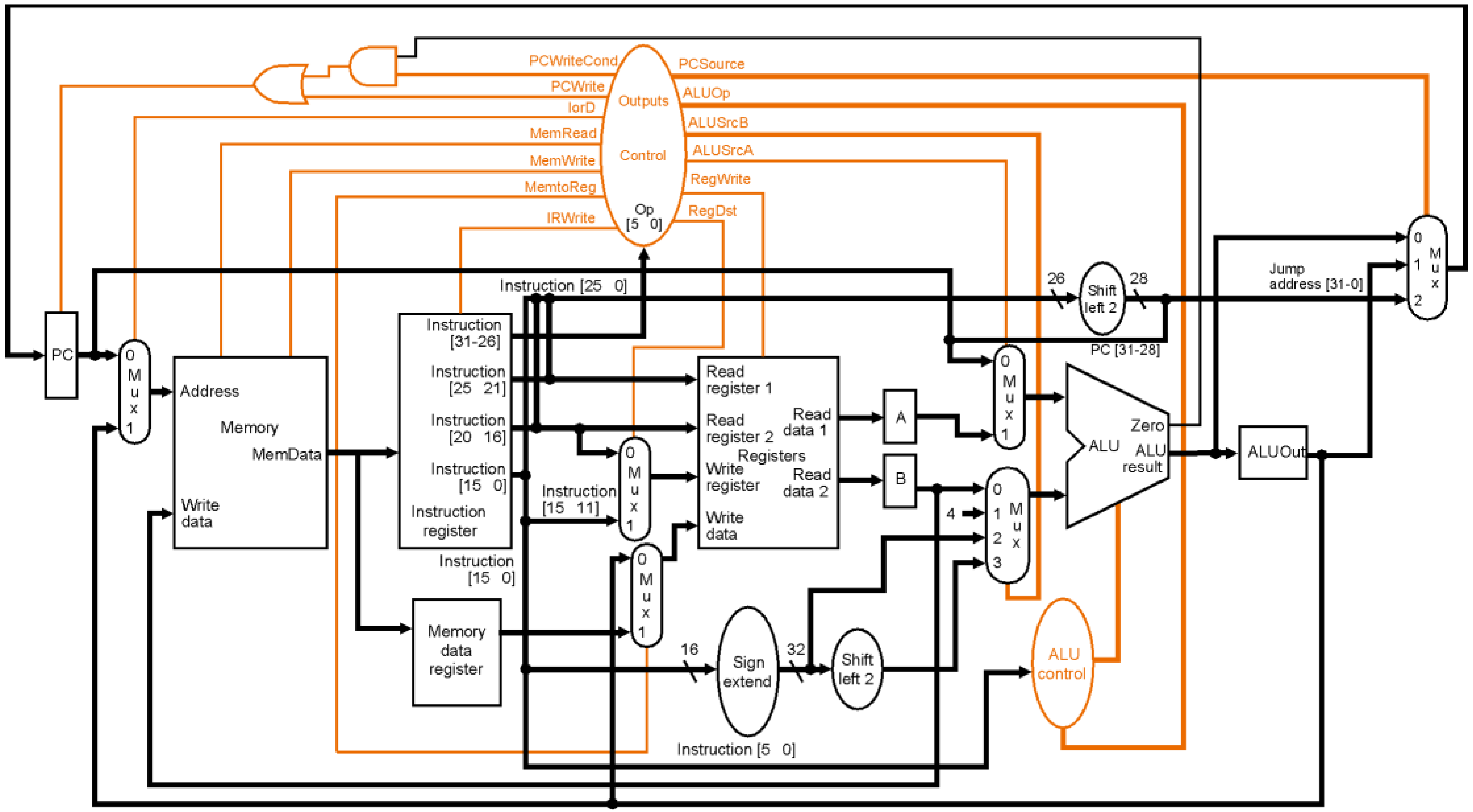
**memory-data = Memory[ALUout]**

# 4. Memory Access Load



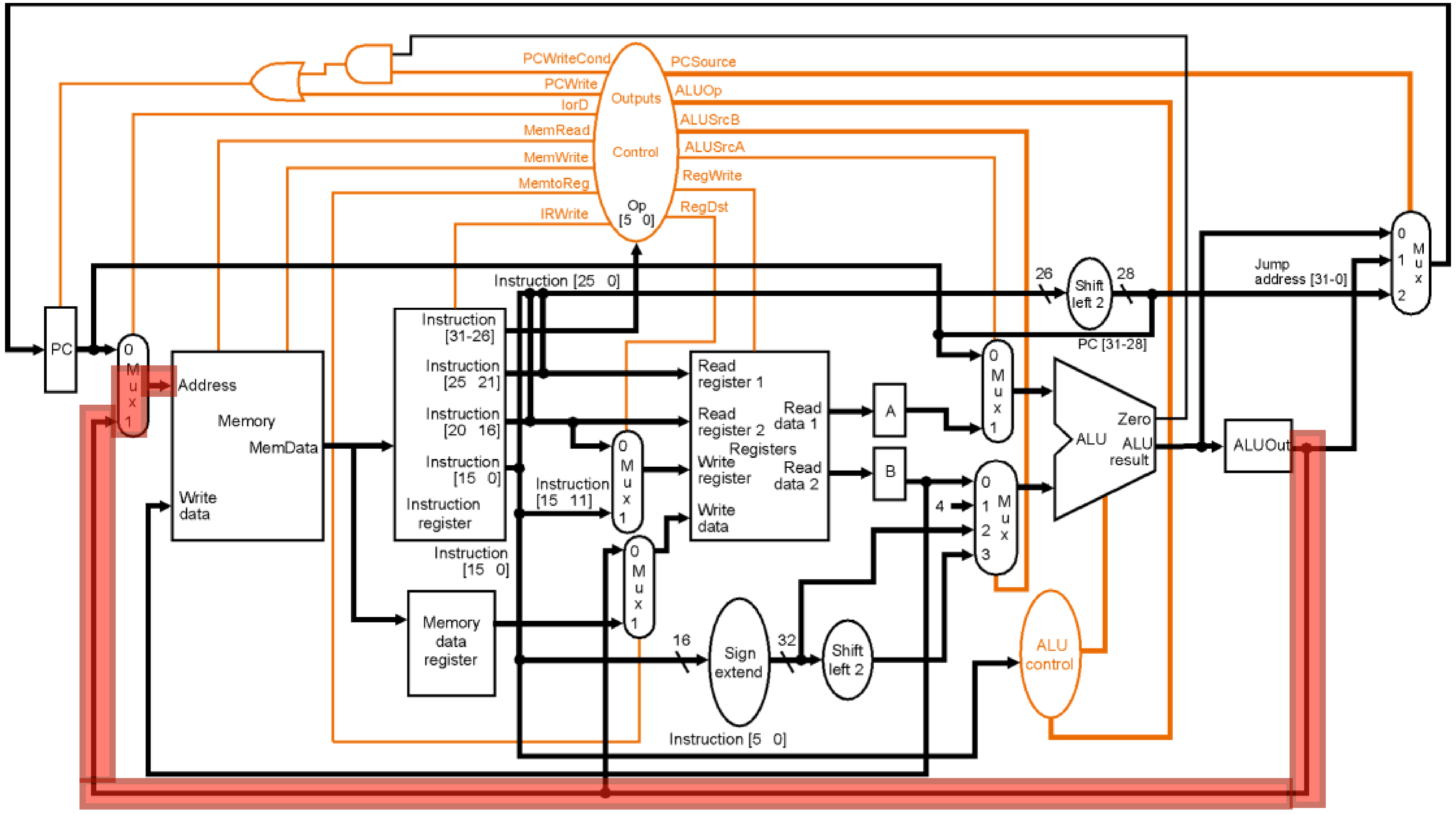
**memory-data = Memory[ALUout]**

# 4. Memory Access Store



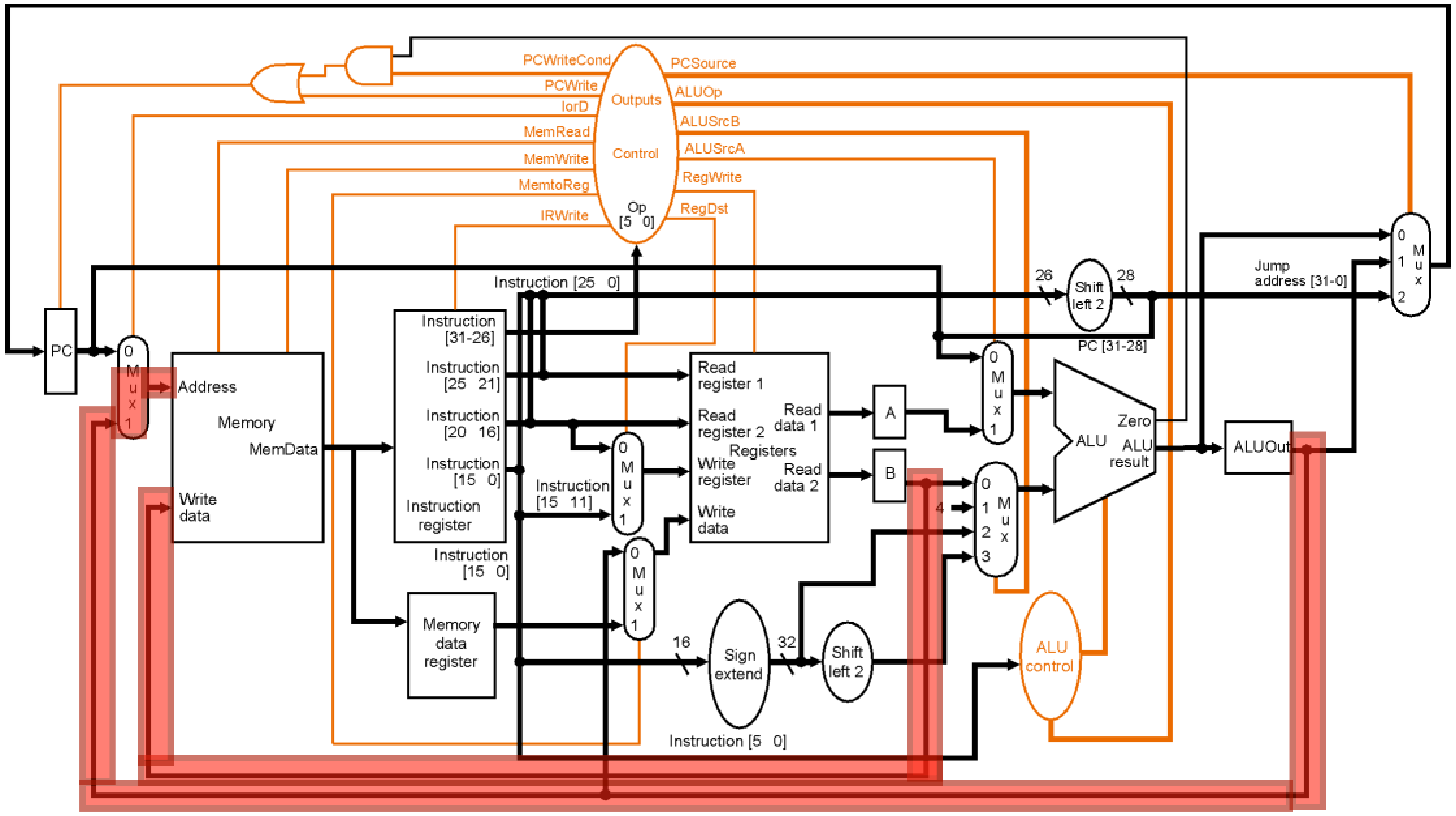
**Memory[ALUout] = B**

# 4. Memory Access Store



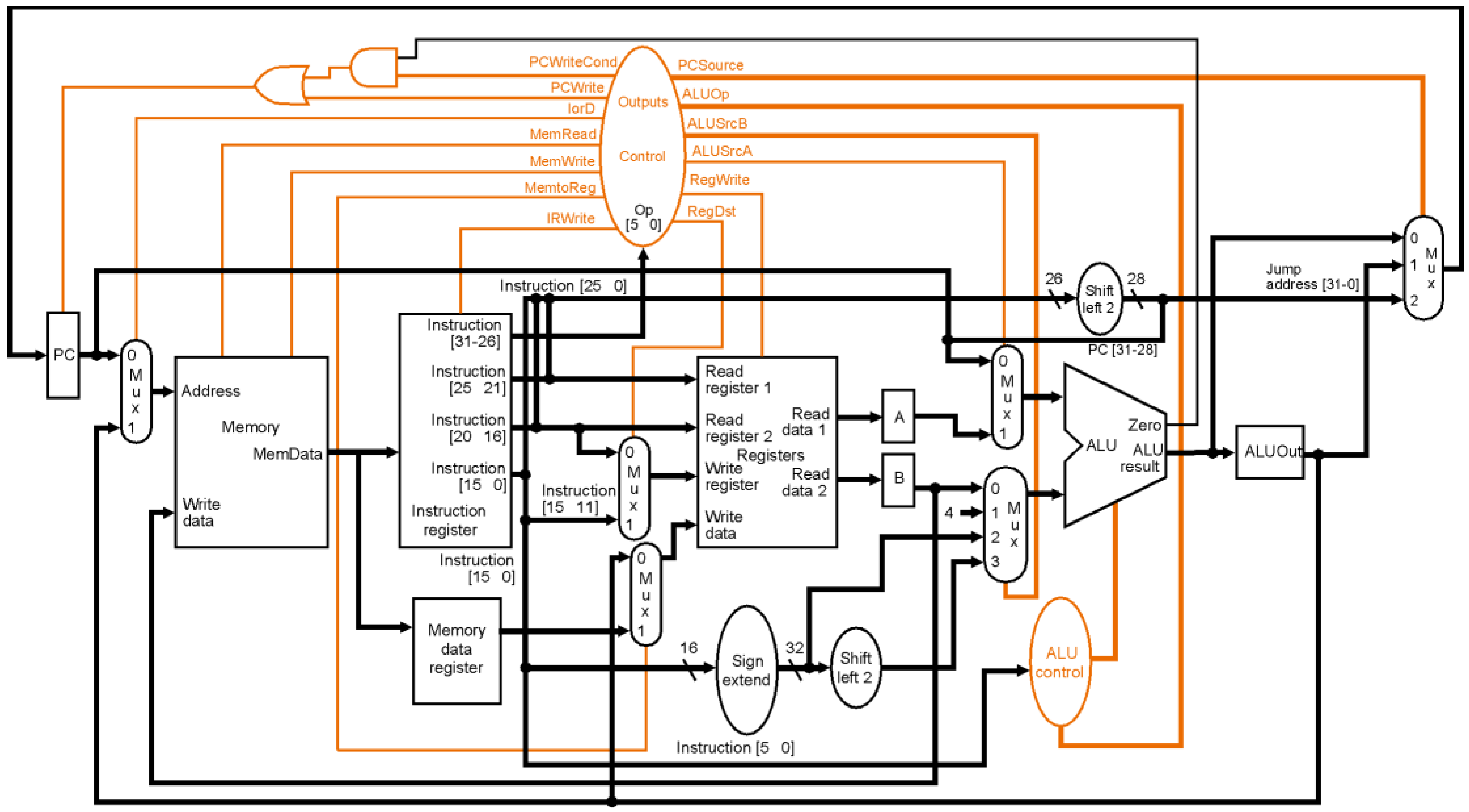
**Memory[ALUout] = B**

## 4. Memory Access Store



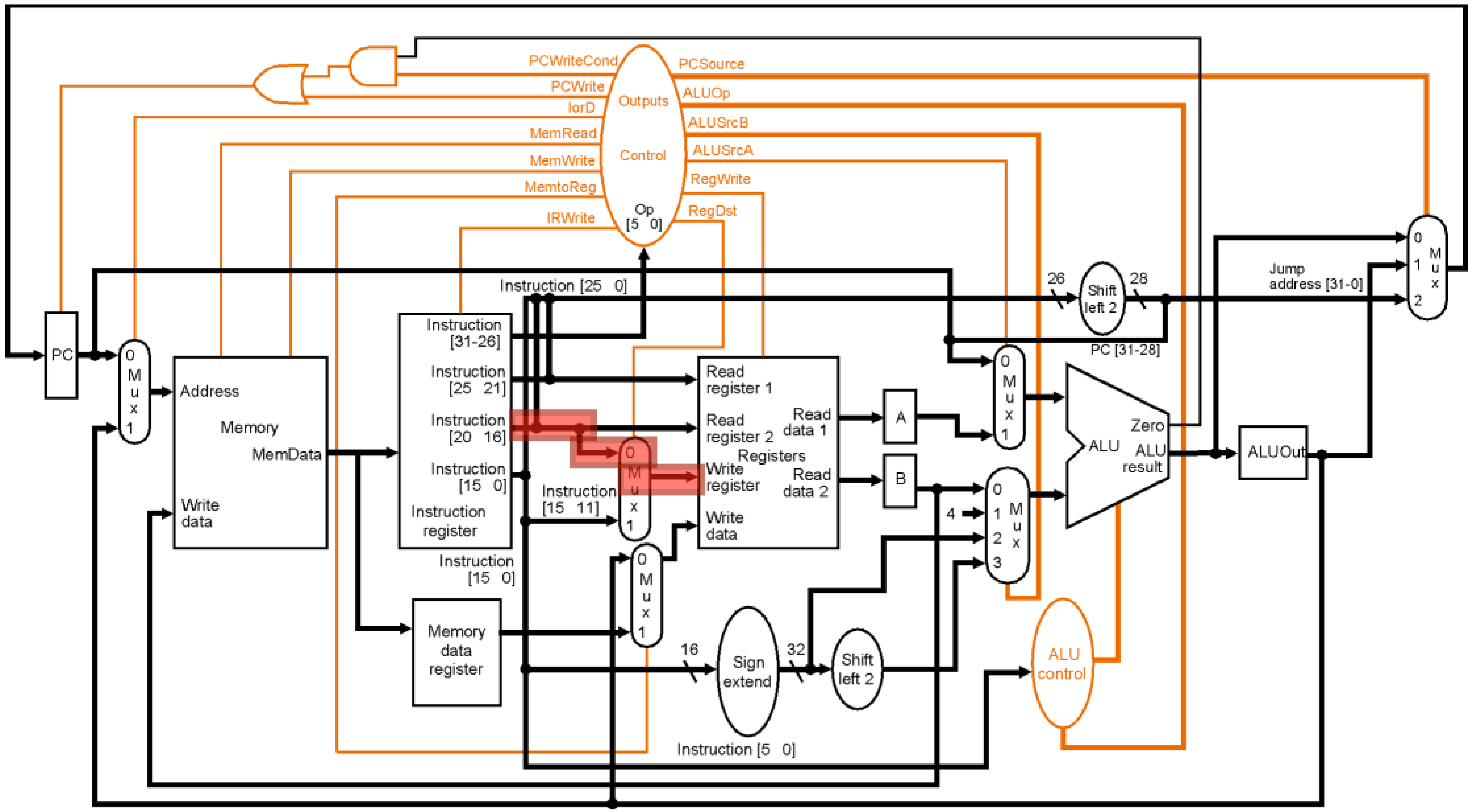
**Memory[ALUout] = B**

# 5. Load Write-Back



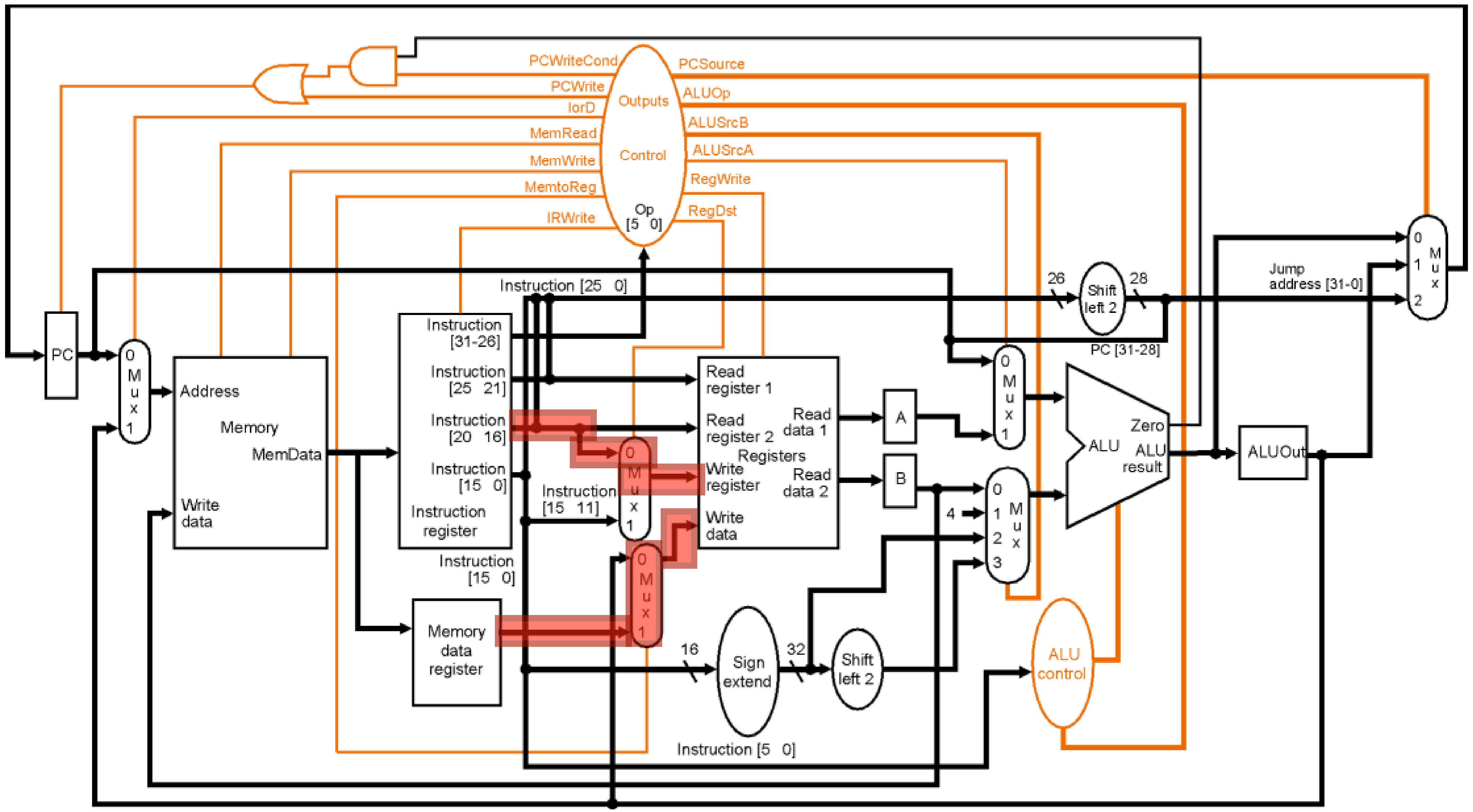
**Reg[IR[20-16]] = memory-data**

# 5. Load Write-Back



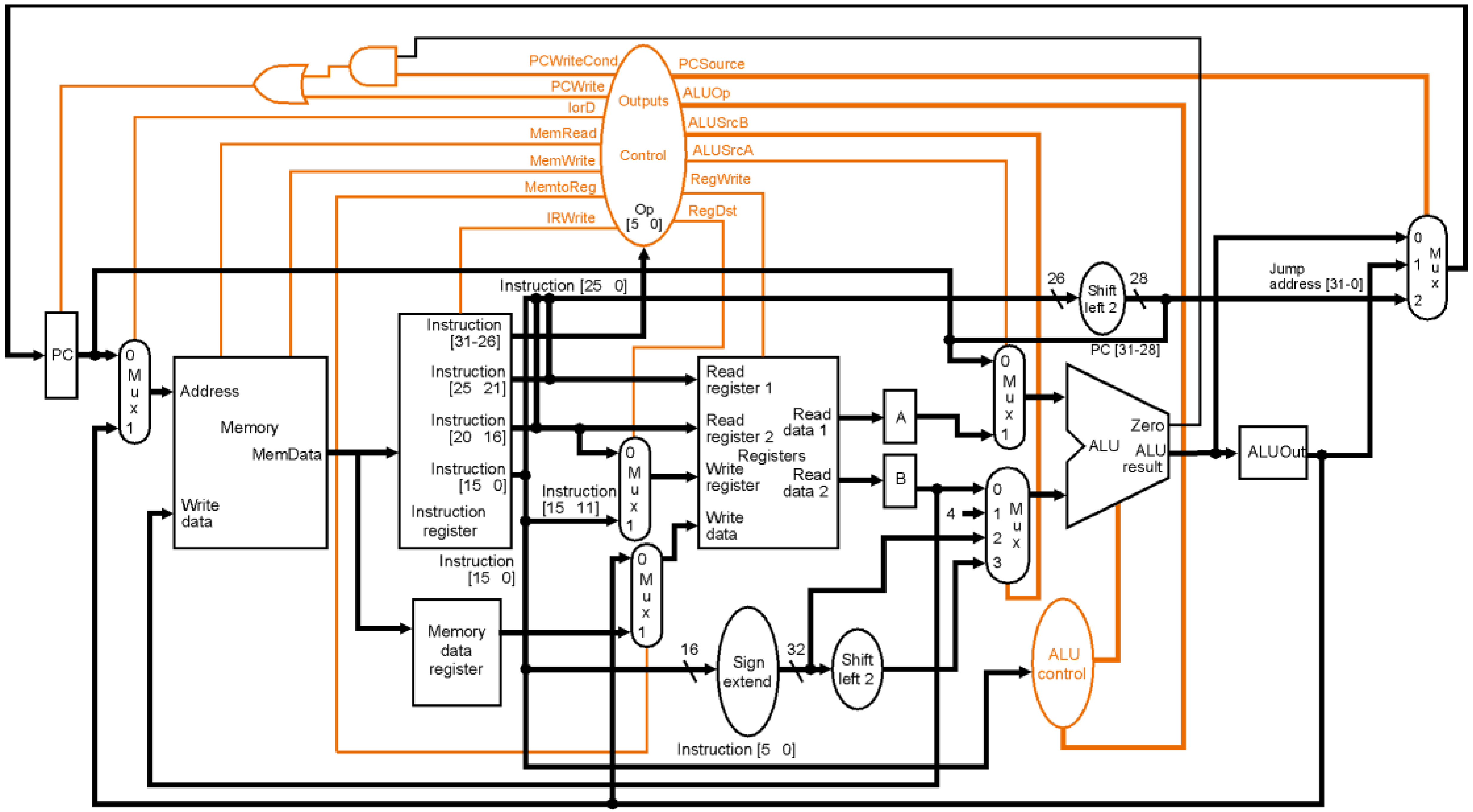
**Reg[IR[20-16]] = memory-data**

# 5. Load Write-Back



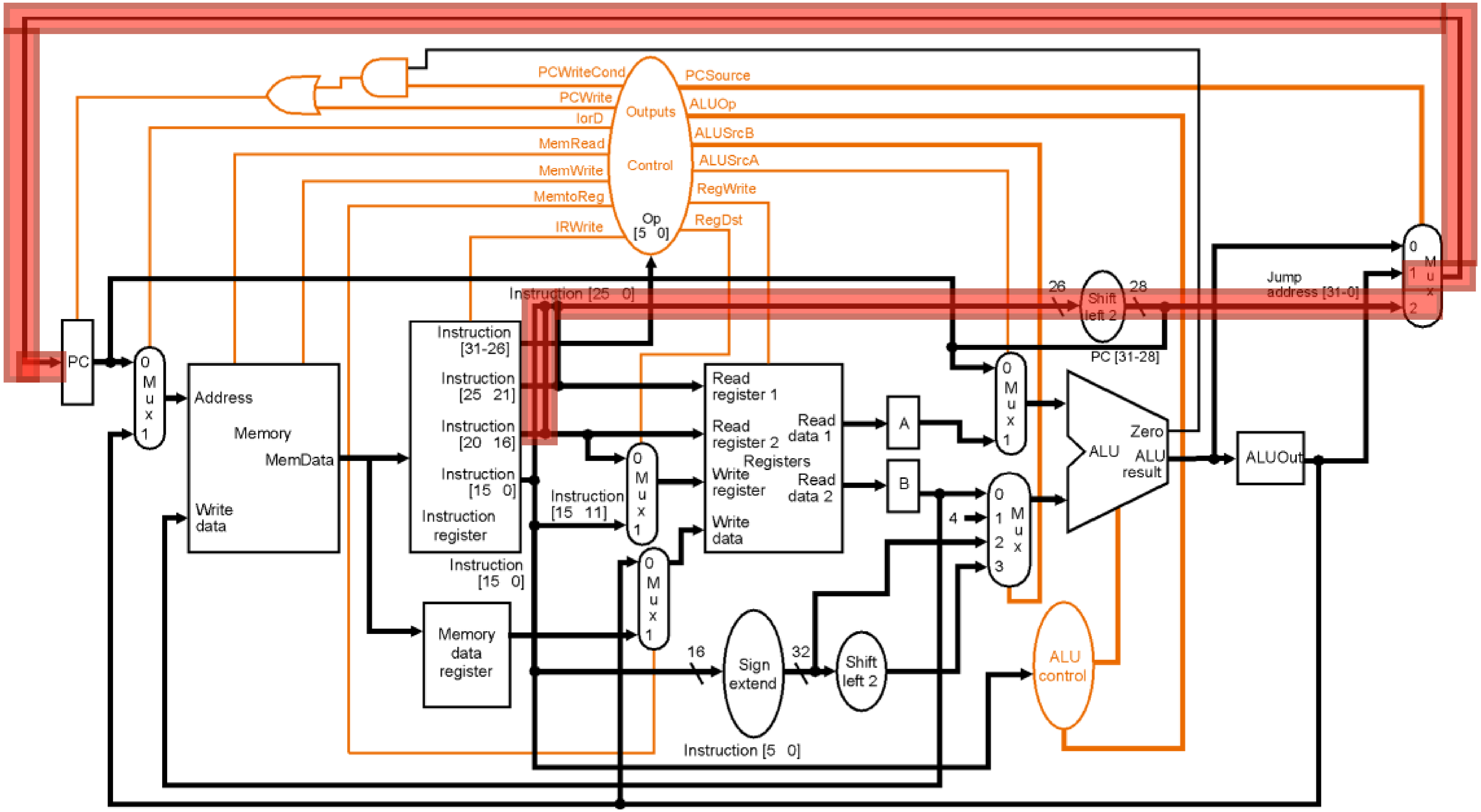
**Reg[IR[20-16]] = memory-data**

# 3. Jump Completion



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

## 3. Jump Completion

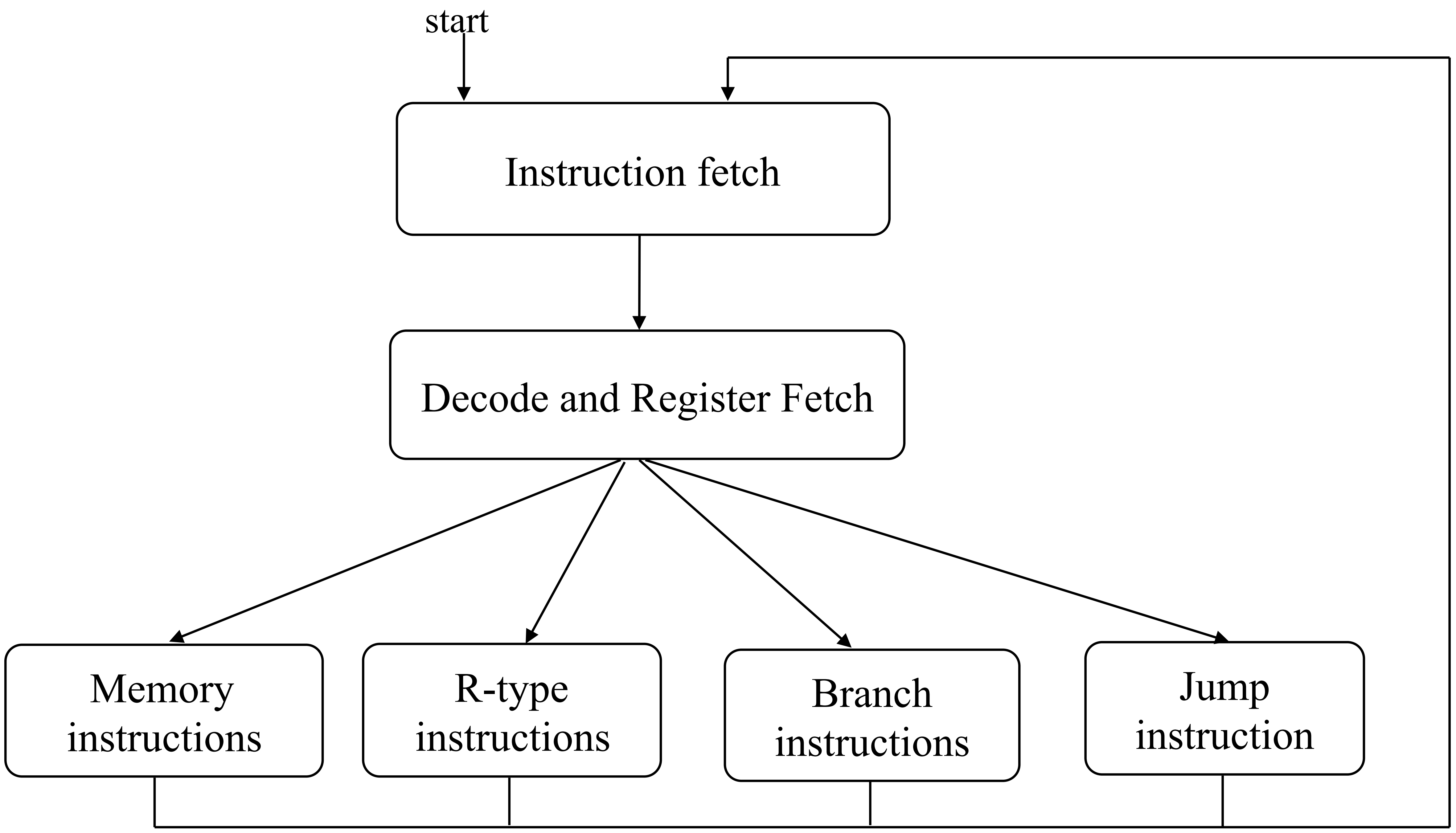


**PC = PC[31-28] | (IR[25-0] <<2)**

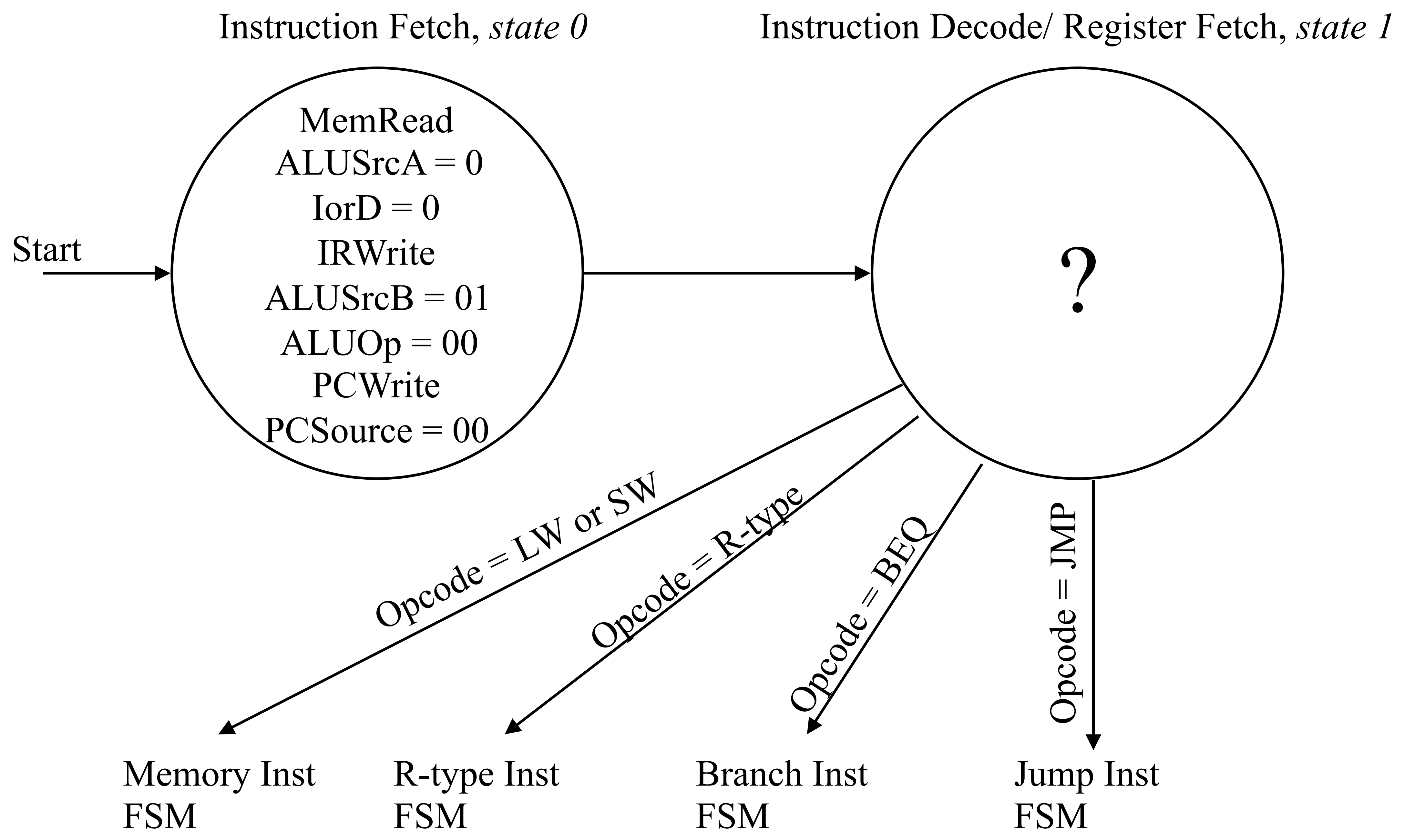
## What About the Control?

- 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

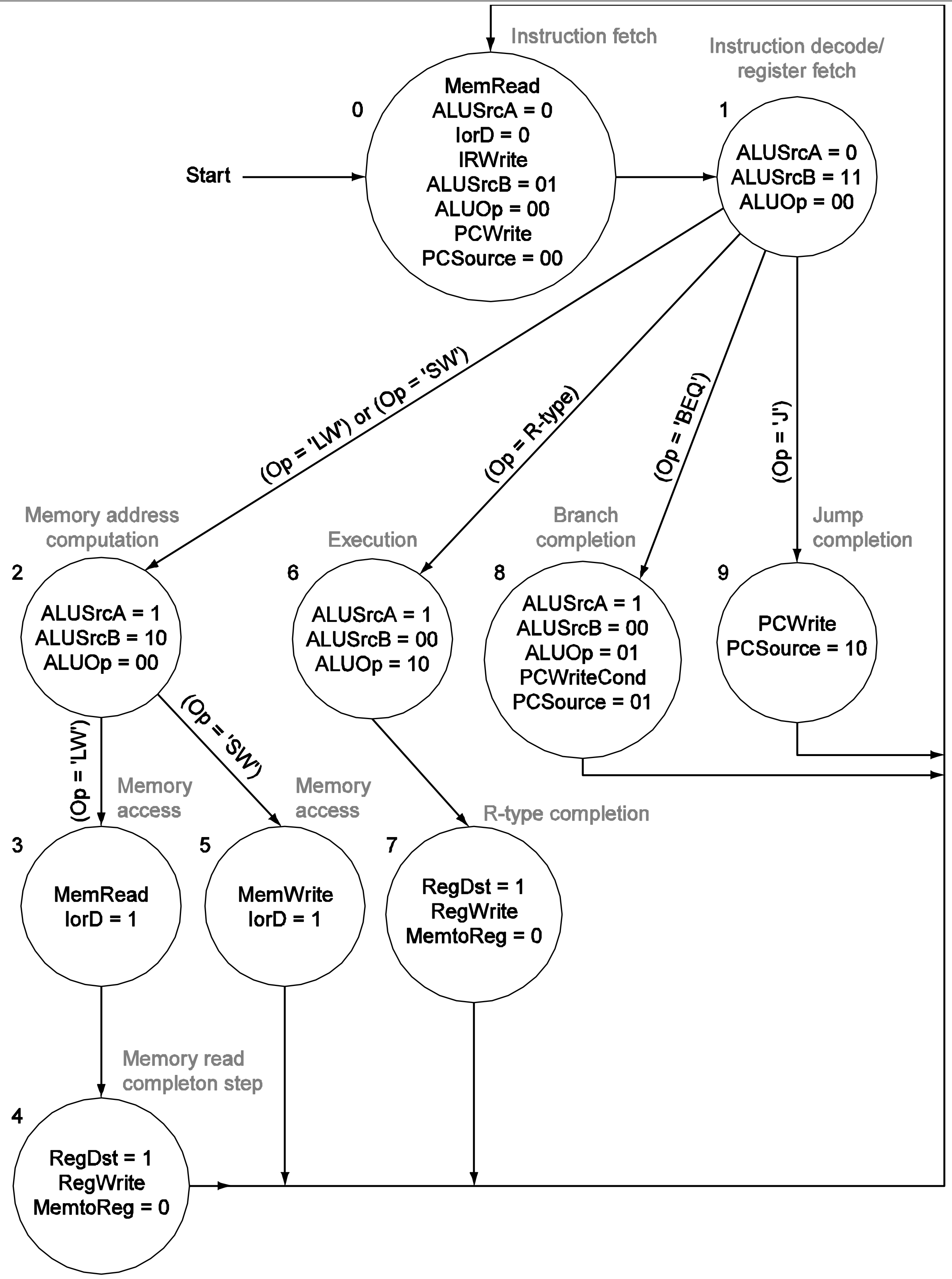
## Multi-Cycle Control



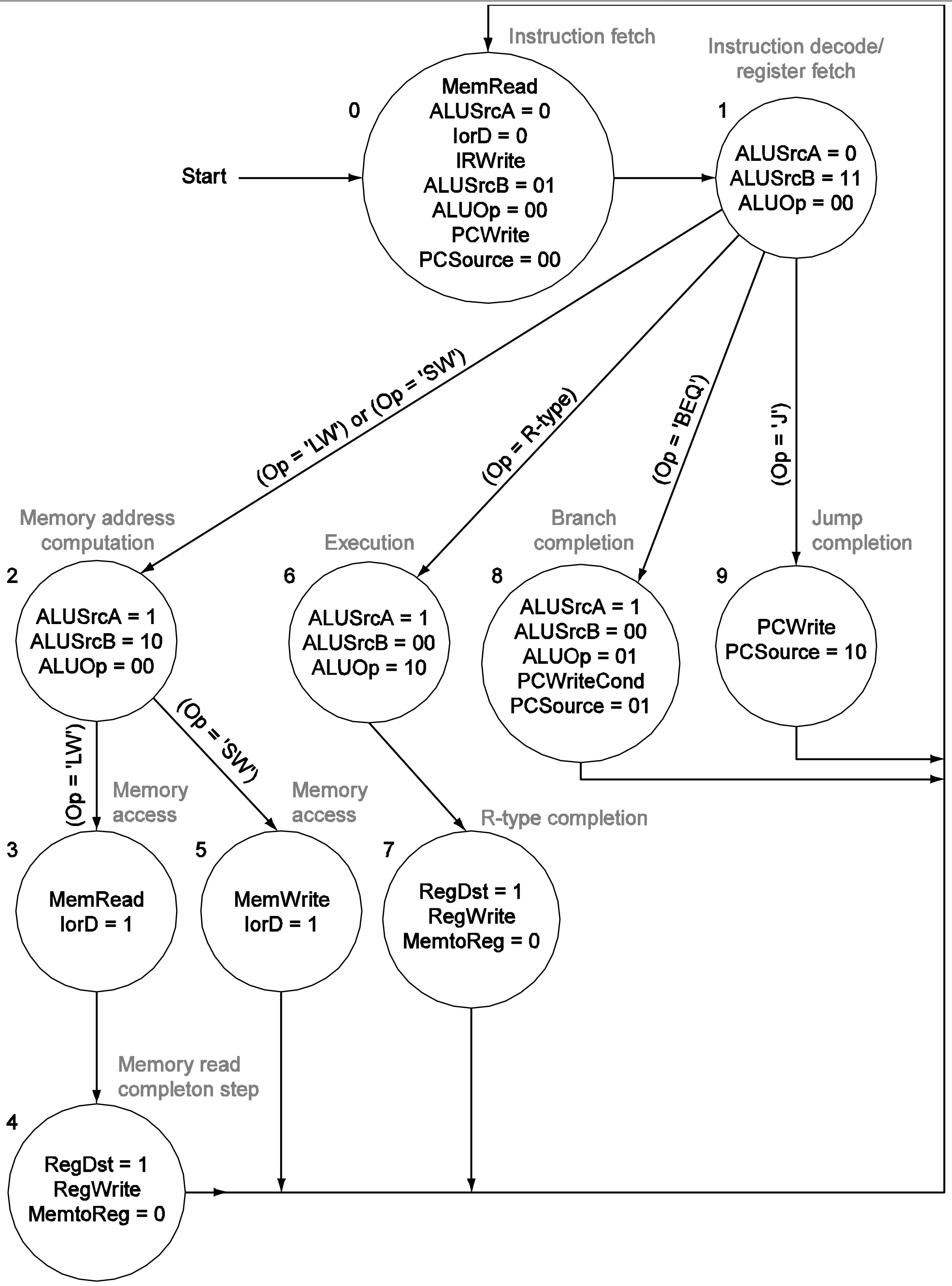
## Multi-Cycle Control



# Multi-Cycle Control - The Full FSM



# Multi-Cycle Control - The Full FSM

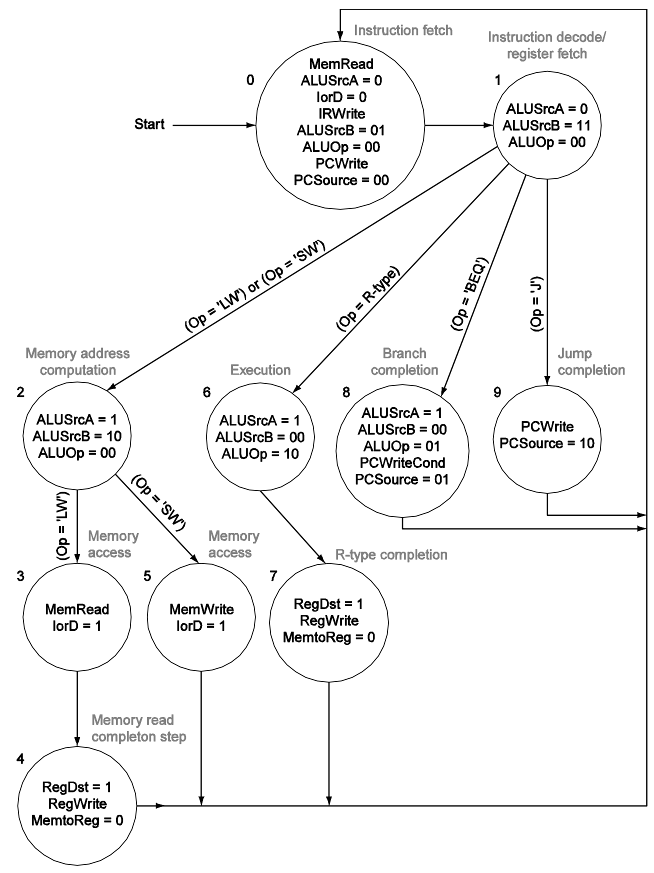


Which type of instruction is the slowest?

# Some Juicy Questions

- How many cycles will it take to execute this code?

```
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label  #assume not taken
add $t5, $t2, $t3
sw $t5, 8($t3)
Label:  ...
```



Instruction fetch
Instruction decode/ register fetch
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
Start
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
Execution
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
Branch completion
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
Jump completion
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Memory read completon step
4
RegDst = 1 RegWrite MemtoReg = 0

- Whats going on during the 8th cycle of execution?

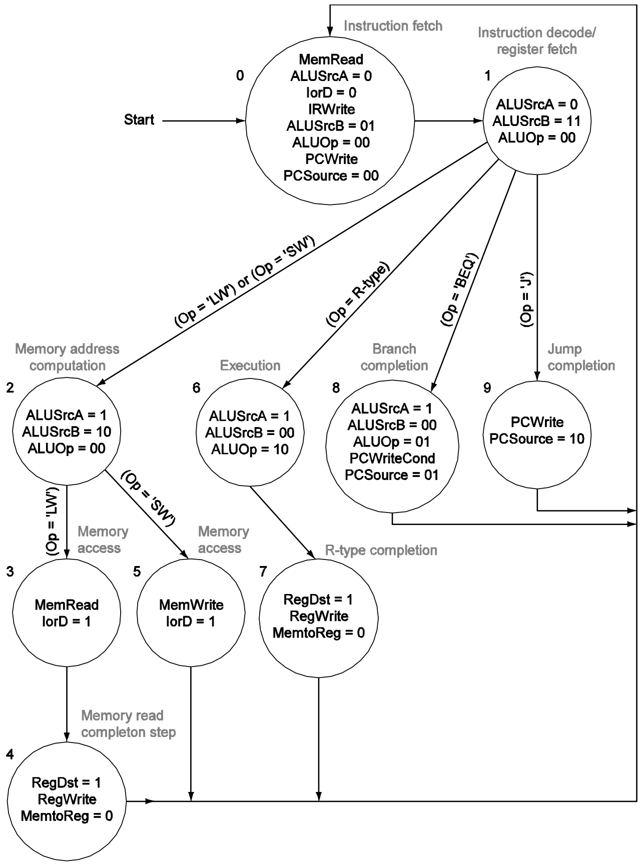
- In what cycle does the actual addition of $t2 and $t3 take place?

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

## Some Juicy Questions

- How many cycles will it take to execute this code?

```
5  lw $t2, 0($t3)
   lw $t3, 4($t3)
   beq $t2, $t3, Label  #assume not taken
   add $t5, $t2, $t3
   sw $t5, 8($t3)
   Label:  ...
```



Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
Execution
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
Branch completion
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
Jump completion
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Memory read completon step
4
RegDst = 1 RegWrite MemtoReg = 0

- Whats going on during the 8th cycle of execution?

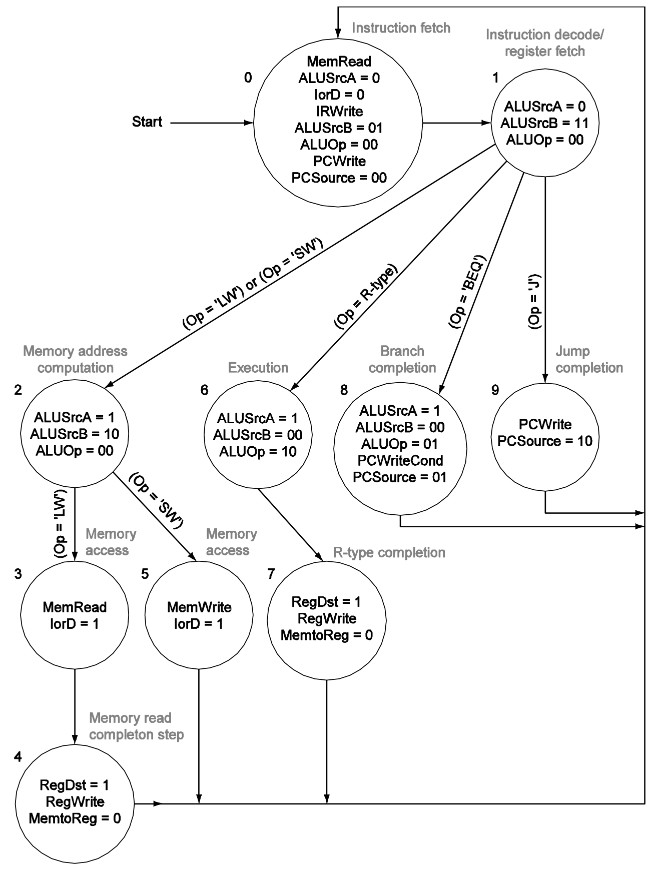
- In what cycle does the actual addition of $t2 and $t3 take place?

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

# Some Juicy Questions

- How many cycles will it take to execute this code?

```
5  lw $t2, 0($t3)
5  lw $t3, 4($t3)
   beq $t2, $t3, Label  #assume not taken
   add $t5, $t2, $t3
   sw $t5, 8($t3)
   Label:  ...
```



Instruction fetch
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
Start
Instruction decode/ register fetch
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
Execution
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
Branch completion
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
Jump completion
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Memory read completon step
4
RegDst = 1 RegWrite MemtoReg = 0

- Whats going on during the 8th cycle of execution?

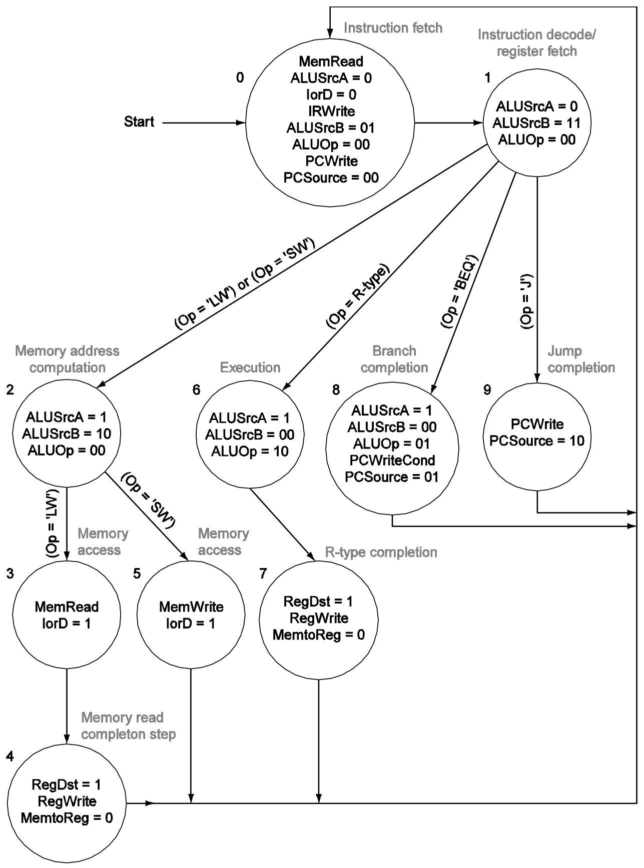
- In what cycle does the actual addition of $t2 and $t3 take place?

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

# Some Juicy Questions

- How many cycles will it take to execute this code?

```
5   lw $t2, 0($t3)
5   lw $t3, 4($t3)
3   beq $t2, $t3, Label   #assume not taken
    add $t5, $t2, $t3
    sw $t5, 8($t3)
    Label:  ...
```



Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
Execution
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
Branch completion
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
Jump completion
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Memory read completon step
4
RegDst = 1 RegWrite MemtoReg = 0

- Whats going on during the 8th cycle of execution?

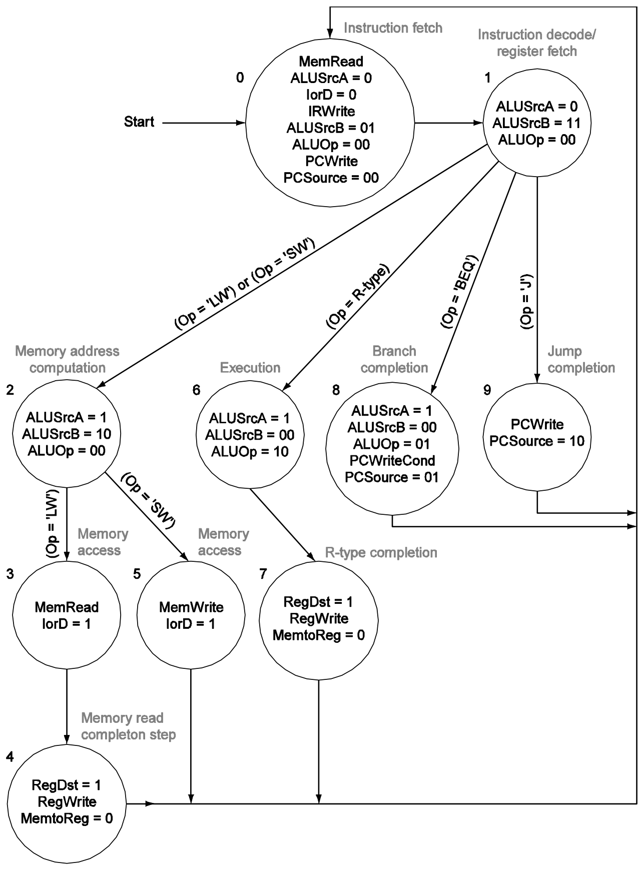
- In what cycle does the actual addition of $t2 and $t3 take place?

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

## Some Juicy Questions

- How many cycles will it take to execute this code?

```
5  lw $t2, 0($t3)
5  lw $t3, 4($t3)
3  beq $t2, $t3, Label   #assume not taken
4  add $t5, $t2, $t3
   sw $t5, 8($t3)
   Label:  ...
```



Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
Execution
Branch completion
Jump completion
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Memory read completon step
4
RegDst = 1 RegWrite MemtoReg = 0

- Whats going on during the 8th cycle of execution?

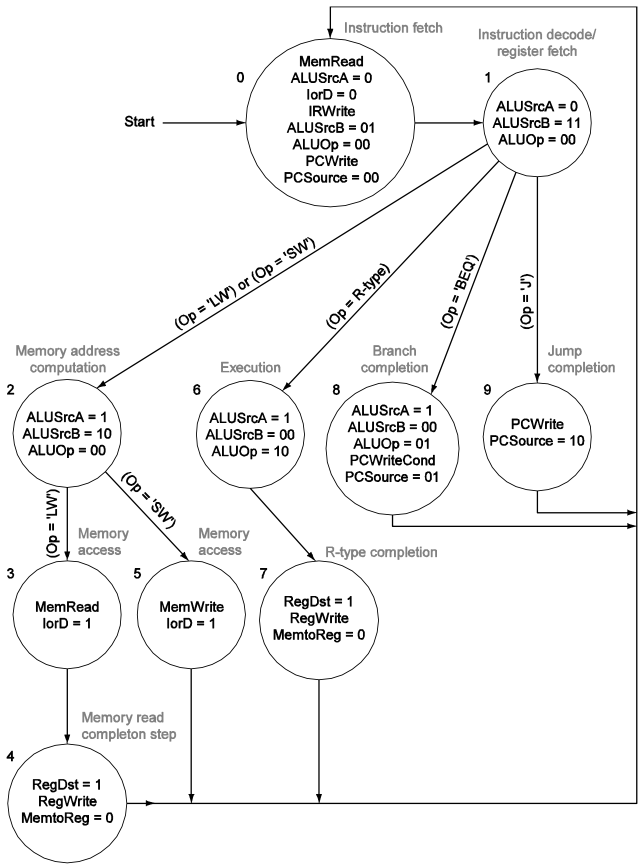
- In what cycle does the actual addition of $t2 and $t3 take place?

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

# Some Juicy Questions

- How many cycles will it take to execute this code?

```
5   lw $t2, 0($t3)
5   lw $t3, 4($t3)
3   beq $t2, $t3, Label   #assume not taken
4   add $t5, $t2, $t3
4   sw $t5, 8($t3)
    Label:  ...
```



- Whats going on during the 8th cycle of execution?

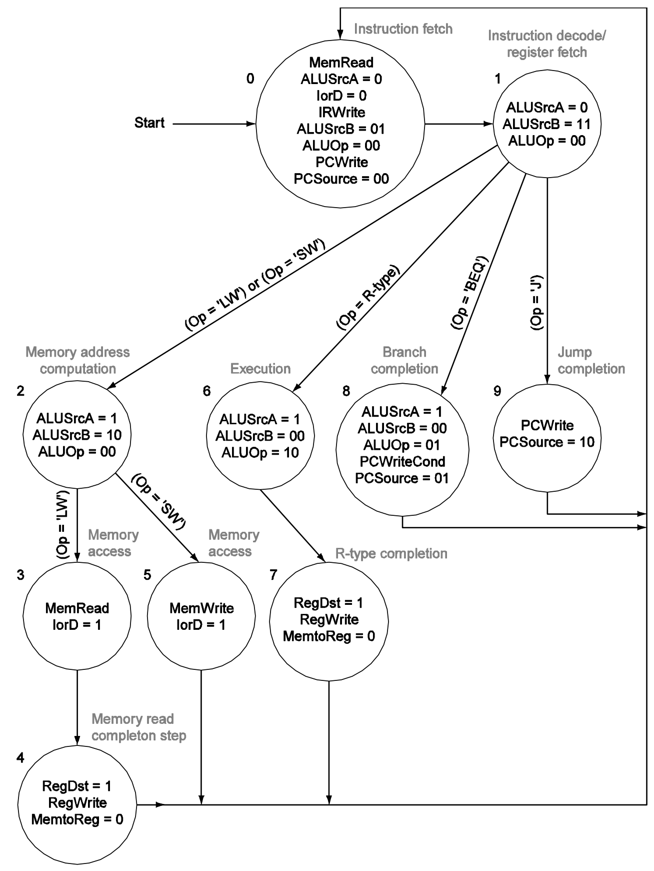
- In what cycle does the actual addition of $t2 and $t3 take place?

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

# Some Juicy Questions

- How many cycles will it take to execute this code?

```
5   lw $t2, 0($t3)
5   lw $t3, 4($t3)
3   beq $t2, $t3, Label   #assume not taken
4   add $t5, $t2, $t3
4   sw $t5, 8($t3)
    Label:  ...
```



Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
Execution
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
Branch completion
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
Jump completion
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Memory read completon step
4
RegDst = 1 RegWrite MemtoReg = 0

- Whats going on during the 8th cycle of execution? **21**

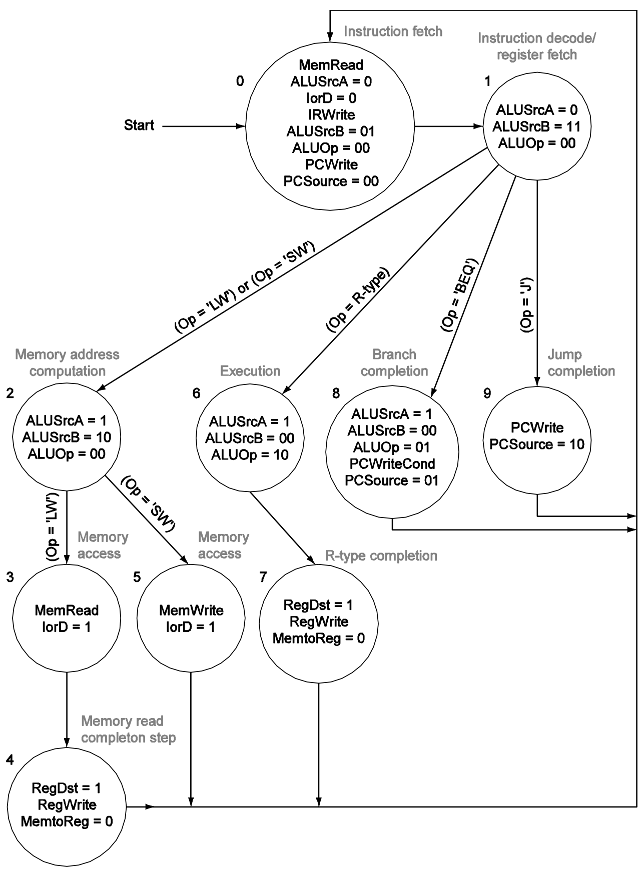
- In what cycle does the actual addition of $t2 and $t3 take place?

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

## Some Juicy Questions

- How many cycles will it take to execute this code?

```
5    lw $t2, 0($t3)
5    lw $t3, 4($t3)
3    beq $t2, $t3, Label   #assume not taken
4    add $t5, $t2, $t3
4    sw $t5, 8($t3)
     Label:  ...
```


Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
Execution
Branch completion
Jump completion
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Memory read completon step
4
RegDst = 1 RegWrite MemtoReg = 0

- Whats going on during the 8th cycle of execution? **21**

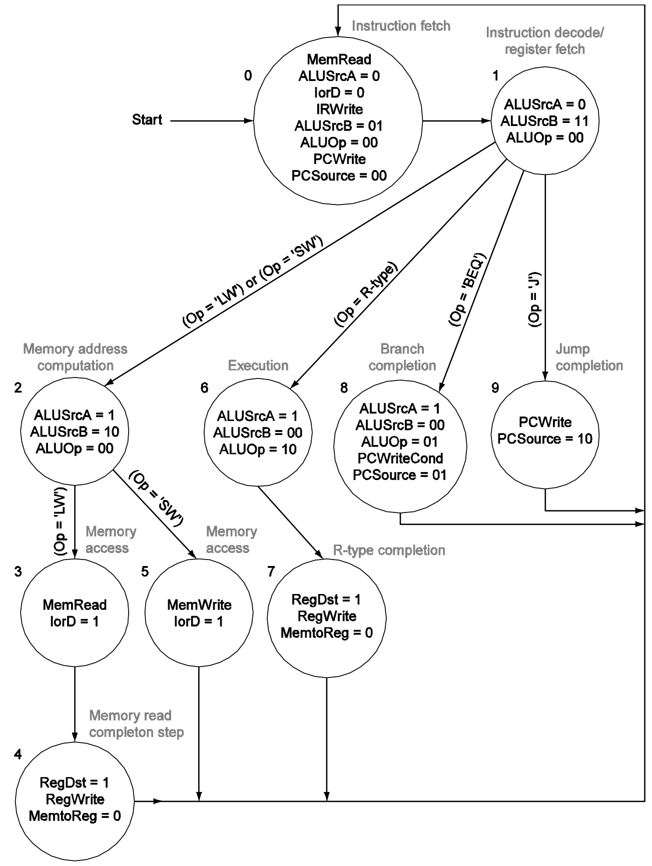
- In what cycle does the actual addition of $t2 and $t3 take place? **16**

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

# Some Juicy Questions

- How many cycles will it take to execute this code?

```
5  lw $t2, 0($t3)
5  lw $t3, 4($t3)
3  beq $t2, $t3, Label   #assume not taken
4  add $t5, $t2, $t3
4  sw $t5, 8($t3)
   Label:  ...
```



- Whats going on during the 8th cycle of execution? **21**

- In what cycle does the actual addition of $t2 and $t3 take place?

**16**

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

**.2*(5) +**

## Some Juicy Questions

- How many cycles will it take to execute this code?

```
5  lw $t2, 0($t3)
5  lw $t3, 4($t3)
3  beq $t2, $t3, Label   #assume not taken
4  add $t5, $t2, $t3
4  sw $t5, 8($t3)
   Label:  ...
```

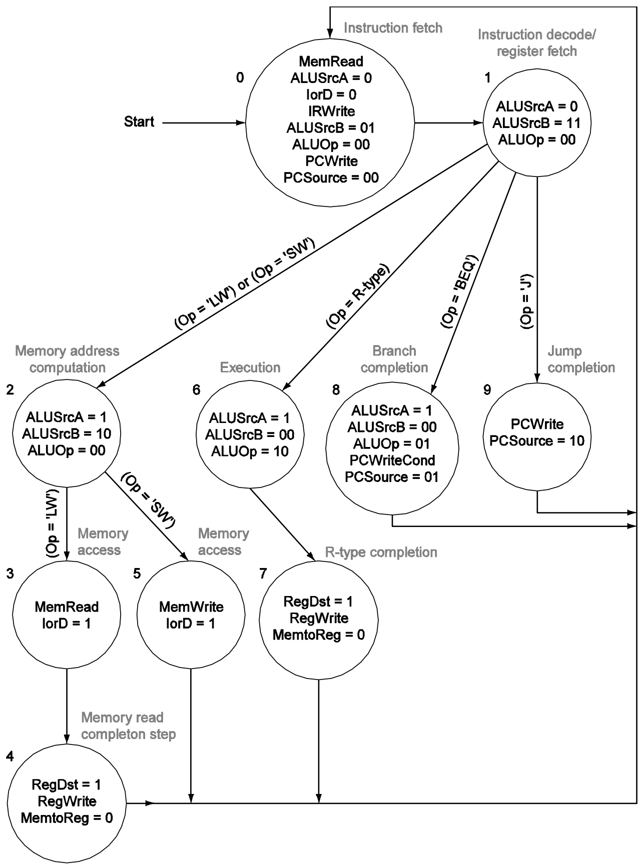
- Whats going on during the 8th cycle of execution? **21**

- In what cycle does the actual addition of $t2 and $t3 take place?

**16**

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

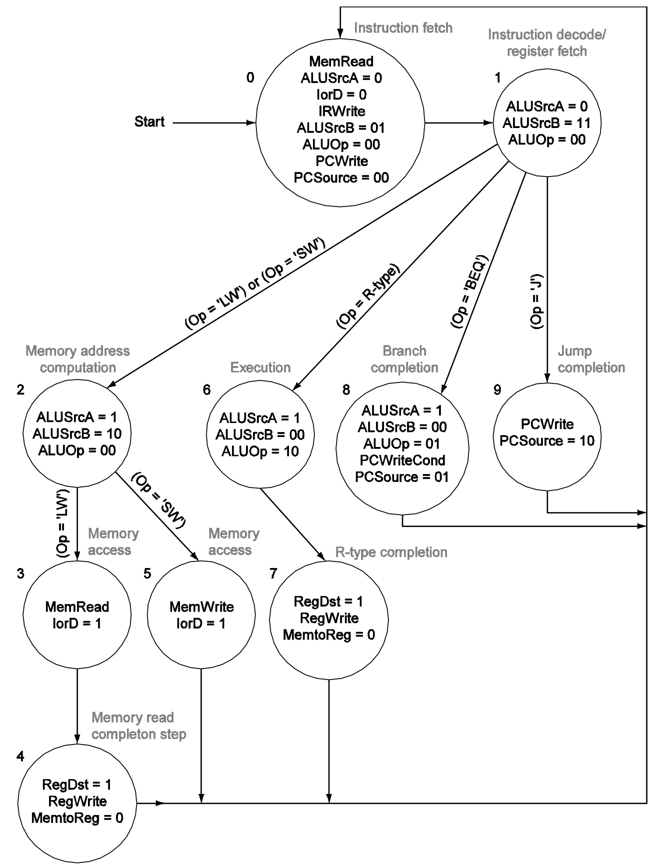
**.2*(5) + .1*(4) +**


Instruction fetch
Instruction decode/ register fetch
0
1
Start
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
Execution
Branch completion
Jump completion
2
6
8
9
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
5
7
MemRead IorD = 1
MemWrite IorD = 1
RegDst = 1 RegWrite MemtoReg = 0
Memory read completon step
4
RegDst = 1 RegWrite MemtoReg = 0

## Some Juicy Questions

- How many cycles will it take to execute this code?

```
5    lw $t2, 0($t3)
5    lw $t3, 4($t3)
3    beq $t2, $t3, Label   #assume not taken
4    add $t5, $t2, $t3
4    sw $t5, 8($t3)
     Label:  ...
```



- Whats going on during the 8th cycle of execution? **21**

- In what cycle does the actual addition of $t2 and $t3 take place?

**16**

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

**.2*(5) + .1*(4) + .5*(4) +**

# Some Juicy Questions

- How many cycles will it take to execute this code?

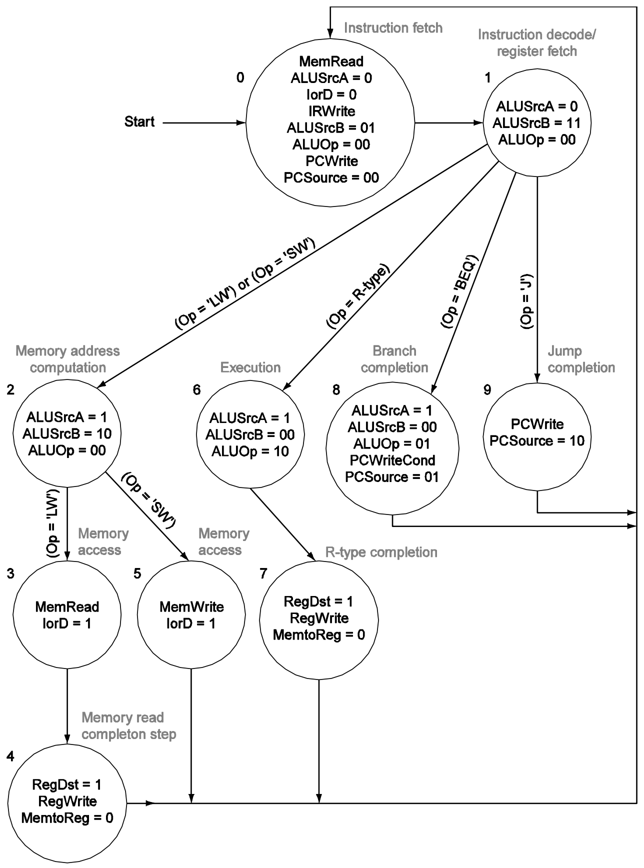
```
5   lw $t2, 0($t3)
5   lw $t3, 4($t3)
3   beq $t2, $t3, Label   #assume not taken
4   add $t5, $t2, $t3
4   sw $t5, 8($t3)
    Label:  ...
```

- Whats going on during the 8th cycle of execution? **21**

- In what cycle does the actual addition of $t2 and $t3 take place? **16**

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

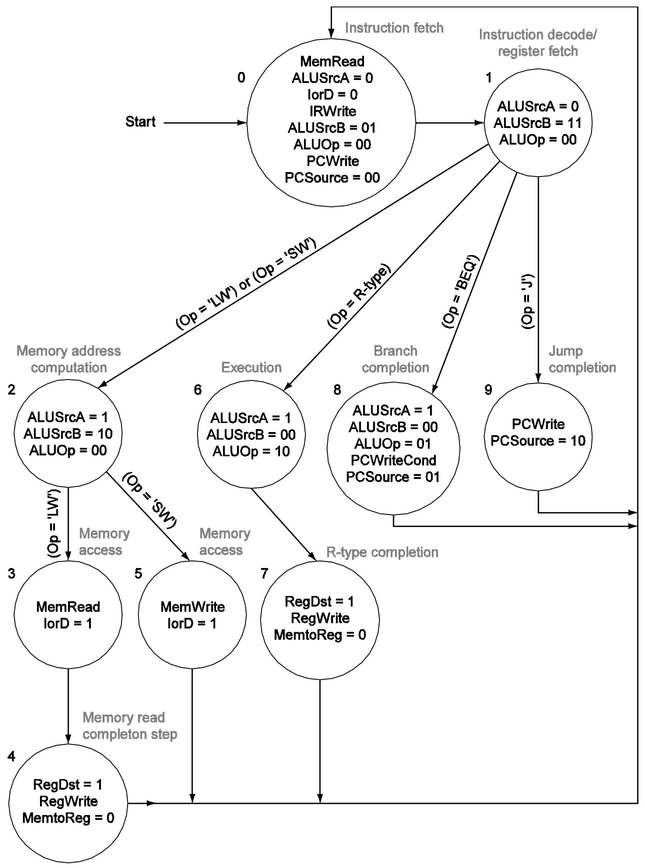
**.2*(5) + .1*(4) + .5*(4) + .2*(3) =**



## Some Juicy Questions

- How many cycles will it take to execute this code?

```
5   lw $t2, 0($t3)
5   lw $t3, 4($t3)
3   beq $t2, $t3, Label   #assume not taken
4   add $t5, $t2, $t3
4   sw $t5, 8($t3)
    Label:  ...
```



Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
Execution
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
Branch completion
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
Jump completion
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Memory read completon step
4
RegDst = 1 RegWrite MemtoReg = 0

- Whats going on during the 8th cycle of execution? **21**

- In what cycle does the actual addition of $t2 and $t3 take place?

**16**

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

**.2*(5) + .1*(4) + .5*(4) + .2*(3) = 4**

## Multi-Cycle Key Points

## Multi-Cycle Key Points

- Performance gain achieved from variable-length instructions

## Multi-Cycle Key Points

- Performance gain achieved from variable-length instructions
- ET = IC * CPI * cycle time

## Multi-Cycle Key Points

- Performance gain achieved from variable-length instructions
- ET = IC * CPI * cycle time
- Required very few new state elements

## 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

# 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

## Exceptions

## Exceptions

- There are two sources of non-sequential control flow in a processor
  - explicit branch and jump instructions
  - exceptions
- *Branches* are synchronous and deterministic
- *Exceptions* are typically asynchronous and non-deterministic
- Guess which is more difficult to handle?

*(control flow refers to the movement of the program counter through memory)*

## Exceptions and Interrupts

## Exceptions and Interrupts

- The terminology is not consistent, but we'll refer to
    - Exceptions as any unexpected change in control flow
    - Interrupts as any externally-caused exception

## Exceptions and Interrupts

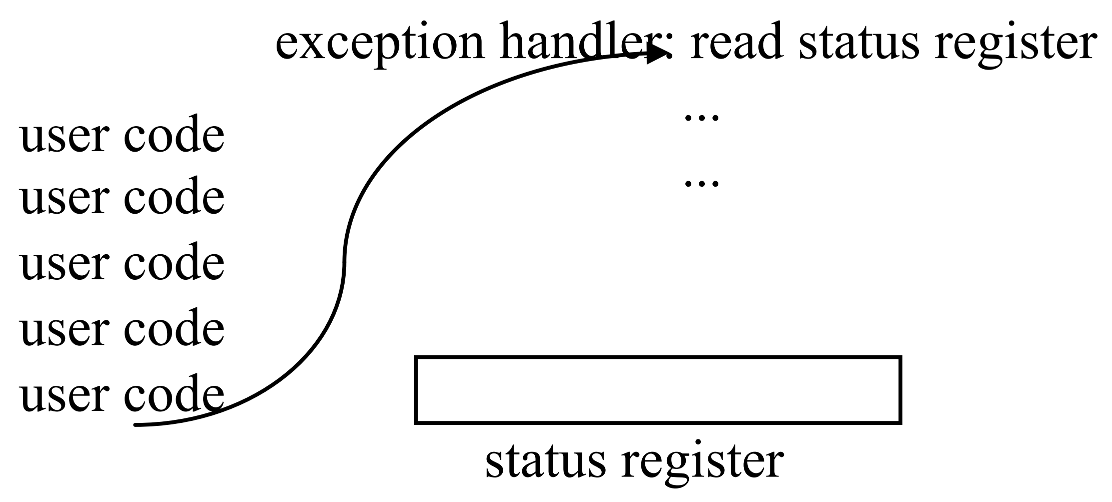
- The terminology is not consistent, but we'll refer to
  - Exceptions as any unexpected change in control flow
  - Interrupts as any externally-caused exception
- So what is...
  - arithmetic overflow
  - divide by zero
  - I/O device signals completion to CPU
  - user program invokes the OS
  - memory parity error
  - illegal instruction
  - timer signal

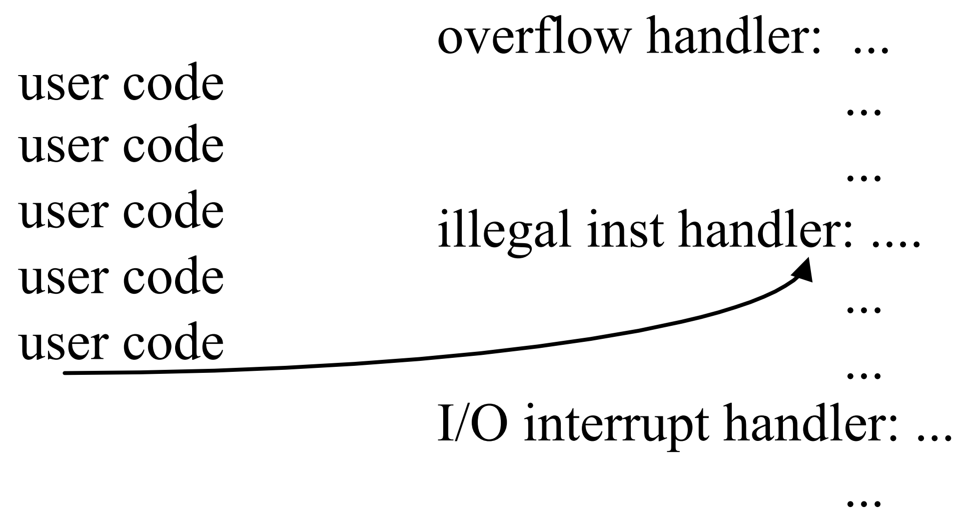
## So Far...

- The machine we’ve been designing in class can generate two types of exceptions.
    - arithmetic overflow
    - illegal instruction
- On an exception, we need to
    - save the PC (invisible to user code)
    - record the nature of the exception/interrupt
    - transfer control to OS

## Handling Exceptions

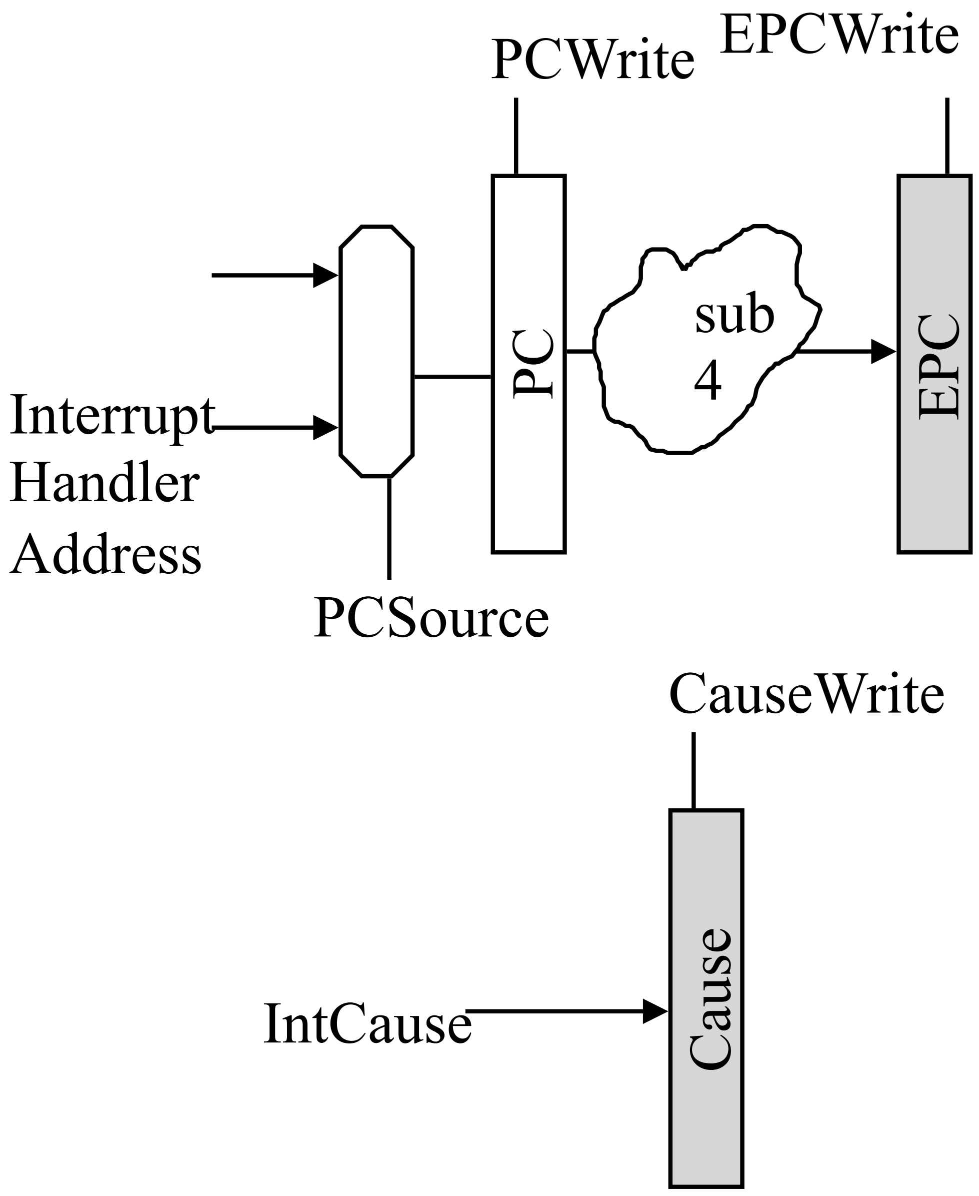
- PC saved in EPC (exception program counter), which the OS may read and store in kernel memory
- A status register, and a single exception handler may be used to record the exception and transfer control, or
- A vectored interrupt transfers control to a different location for each possible type of interrupt/exception



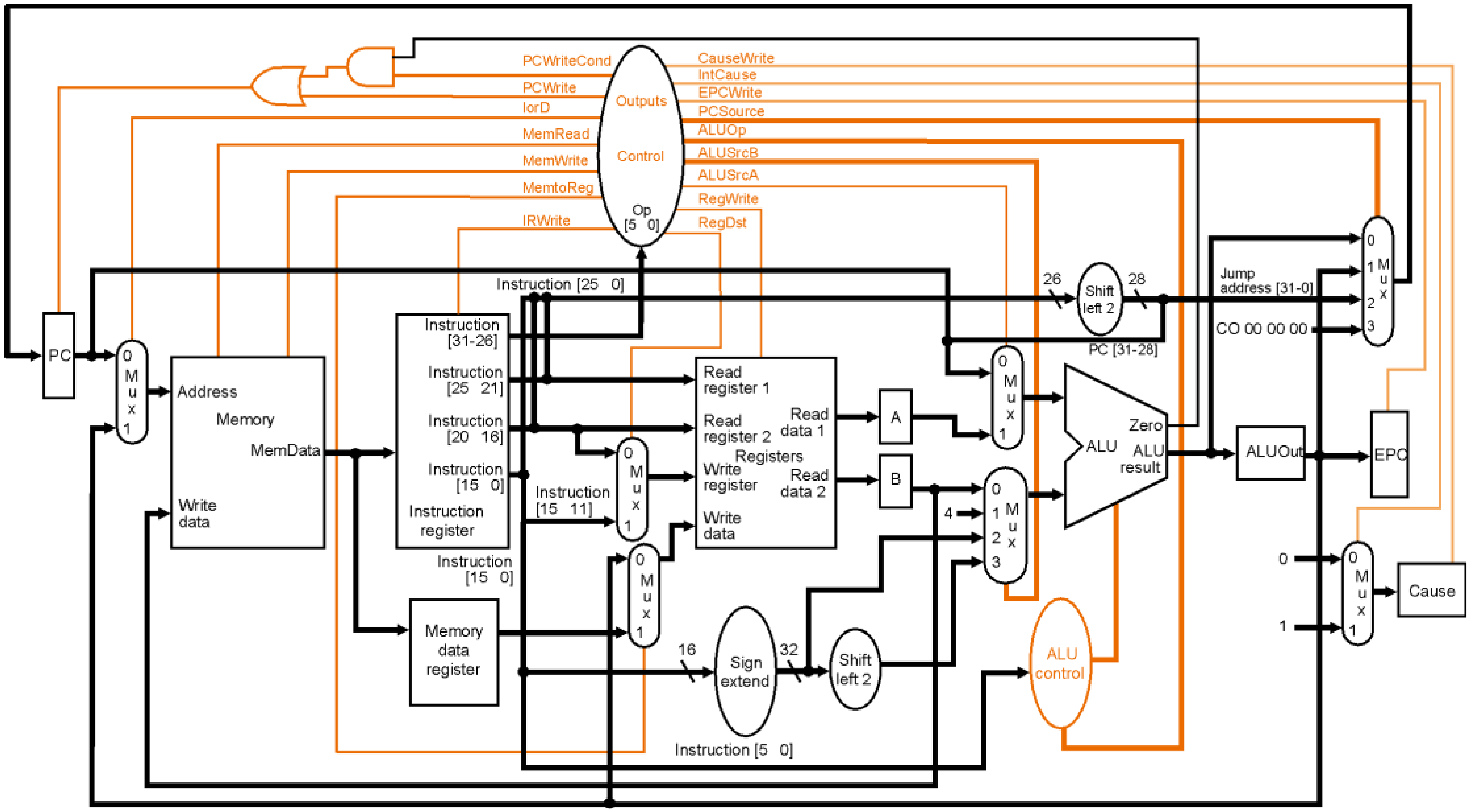


# Supporting Exceptions

- For our MIPS-subset architecture, we will add two registers:
  - EPC: a 32-bit register to hold the user's PC
  - Cause: A register to record the cause of the exception
    - we'll assume undefined inst = 0, overflow = 1
- We will also add three control signals:
  - EPCWrite (will need to be able to subtract 4 from PC)
  - CauseWrite
  - IntCause
- We will extend PCSource multiplexor to be able to latch the interrupt handler address into the PC.



# Supporting Exceptions in our Datapath



## Key Take-away

- Exception-handling is difficult in the CPU
    - because the interactions between the executing instructions and the interrupt are complex and sometimes unpredictable.