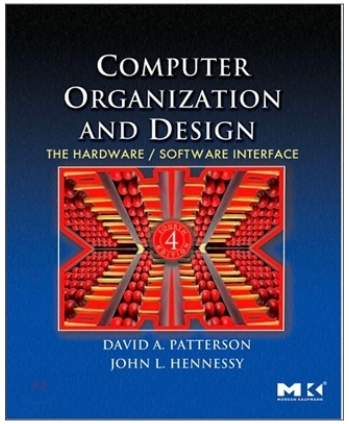
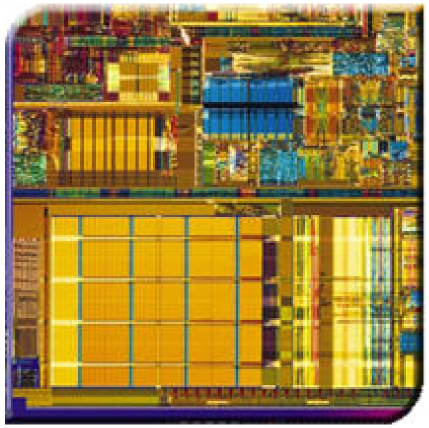
Computer Architecture

Lecture 6 Multi-Cycle Datapath


COMPUTER ORGANIZATION AND DESIGN
THE HARDWARE / SOFTWARE INTERFACE
DAVID A. PATTERSON
JOHN L. HENNESSY

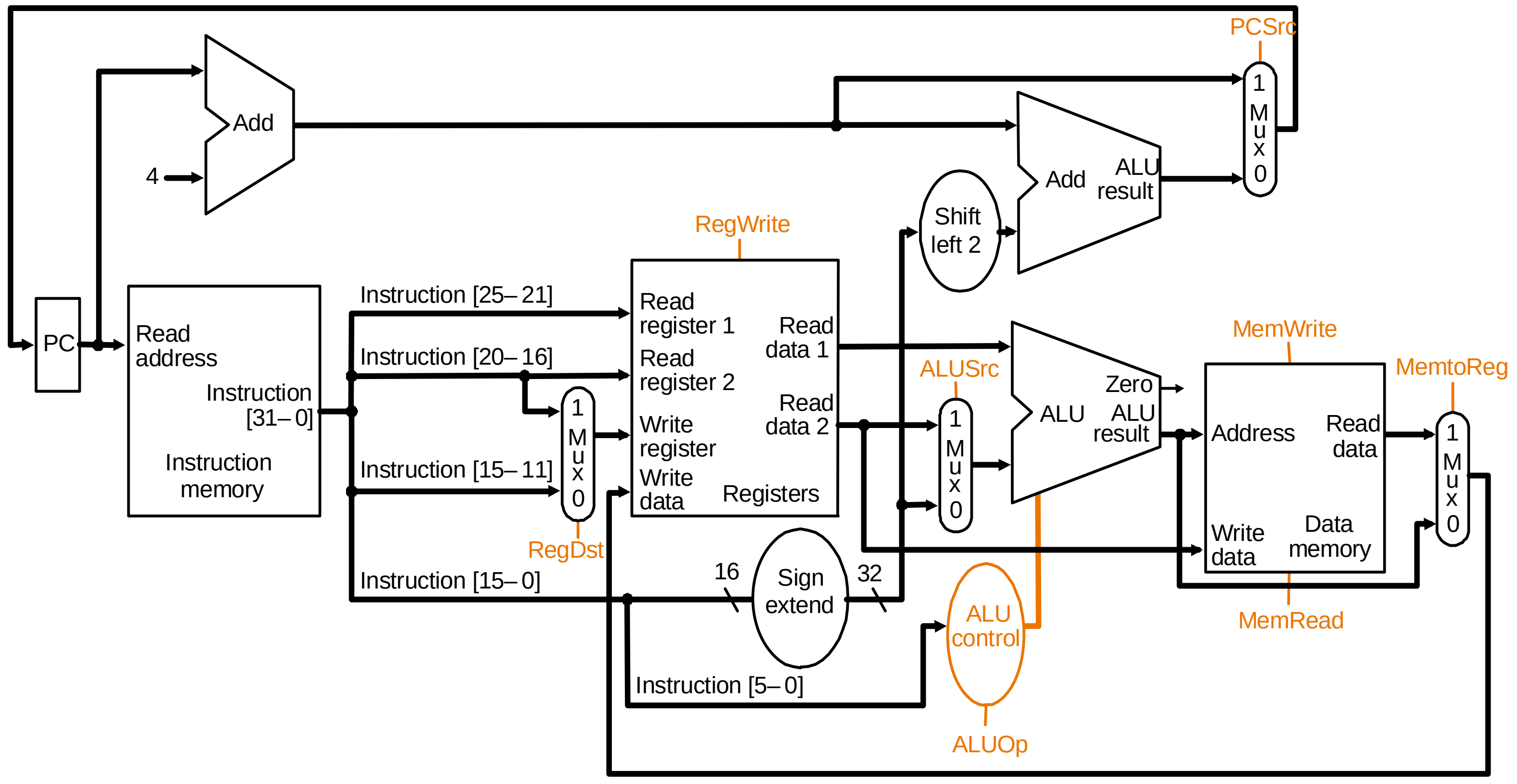


Prof. Jongmyon Kim

울산대학교 UNIVERSITY OF ULSAN

Review: Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)



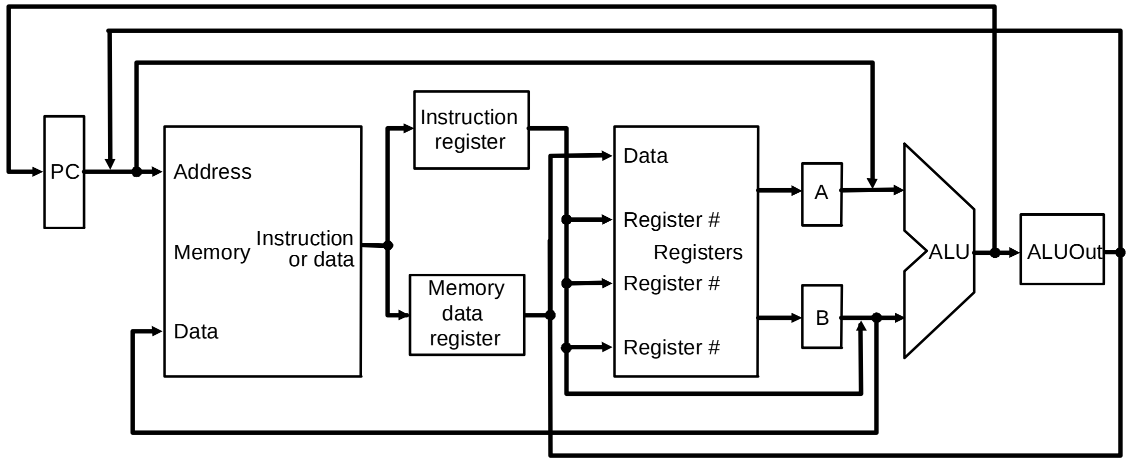
Summary (cont.)

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
Load word	2ns	1ns	2ns	2ns	1ns	8ns
Store word	2ns	1ns	2ns	2ns		7ns
R-format	2ns	1ns	2ns		1ns	6ns
Branch	2ns	1ns	2ns			5ns

- Single cycle datapath
  - Design for the worst case
  - Need to make the cycle time = 8ns per cycle
- Multi-cycle datapth
  - Design for each individual instruction class
  - For the above example: cycle time = 2ns
  - Lw=10ns (5 cycles), sw=8ns (4 cycles), R-format=8ns(4 cycles), beq=6ns (3 cycles)

Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:

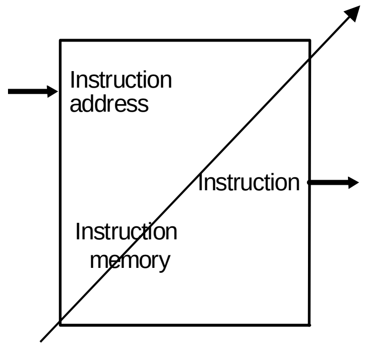


Multi-cycle Approach

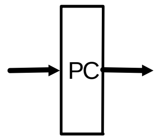
- **Single-cycle design**
  - Clock period is limited by the “worst case instruction timing” (which instructions?)
  - Violate Amdahl’s Law: Make the common case faster
  - Require duplication of some functional units such as adders for PC generation and an ALU for arithmetic
- **Multi-cycle design**
- **We will be reusing functional units and memory**
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- **Our control signals will not be determined solely by instruction**
  - e.g., what should the ALU do for a “subtract” instruction?
- **We’ll use a finite state machine for control**
  - State elements needed for transient data produced by “the same instruction”

Simple Implementation

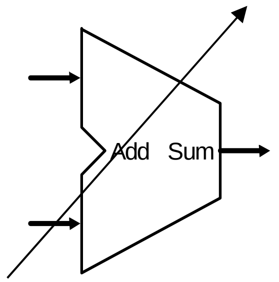
- Single-Cycle Datapath vs. Multi-Cycle Datapath



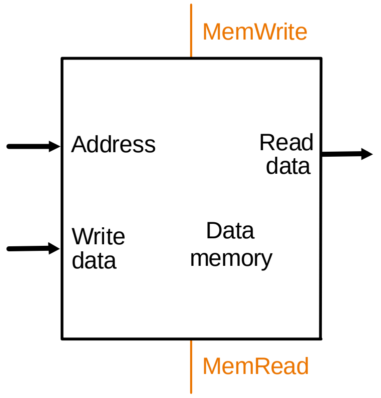
a. Instruction memory



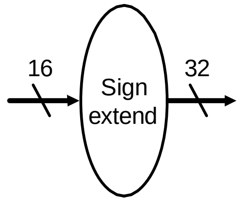
b. Program counter



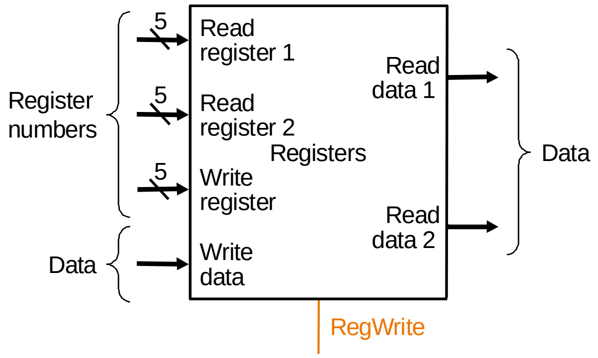
c. Adder



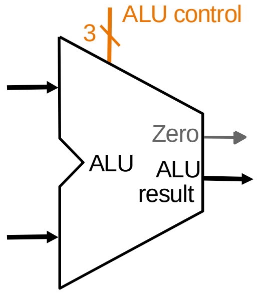
a. Data memory unit



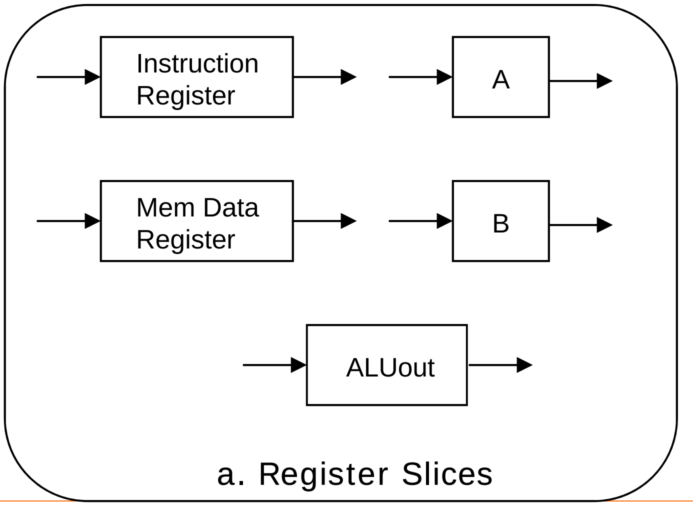
b. Sign-extension unit



a. Registers



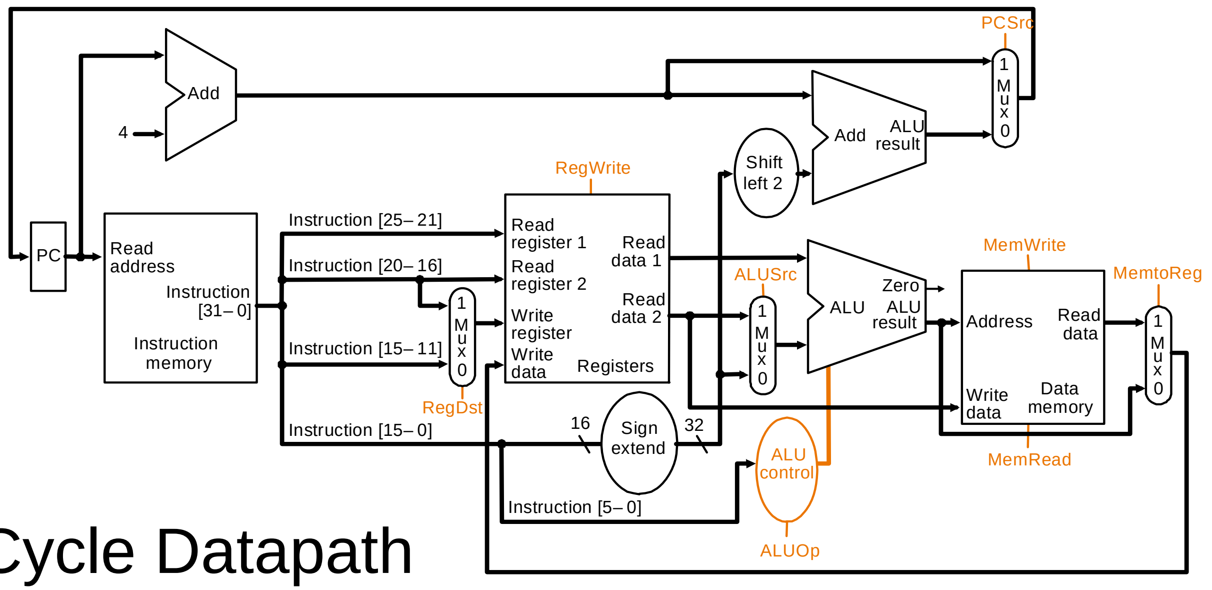
b. ALU



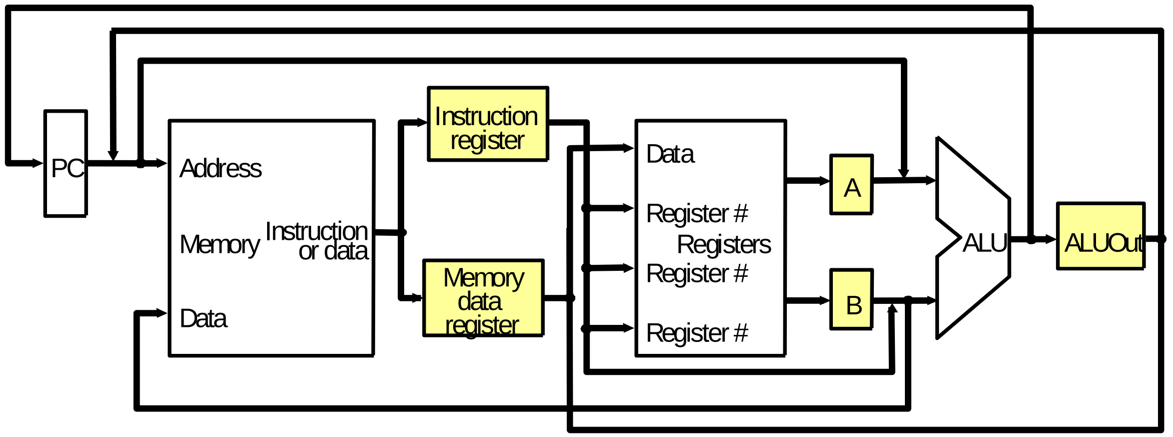
UNIVERSITY OF ULSAN

Single-Cycle vs. Multi-Cycle Datapath

Single-Cycle Datapath

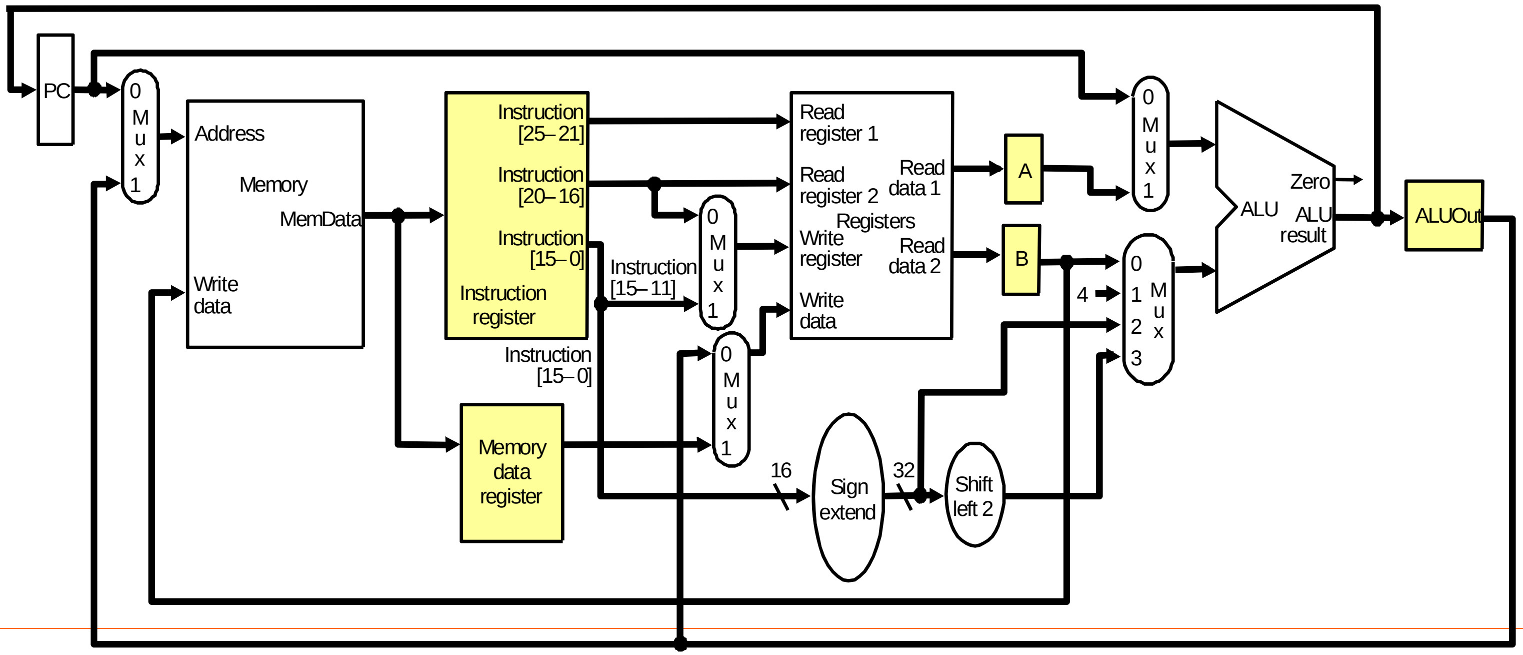


Multi-Cycle Datapath

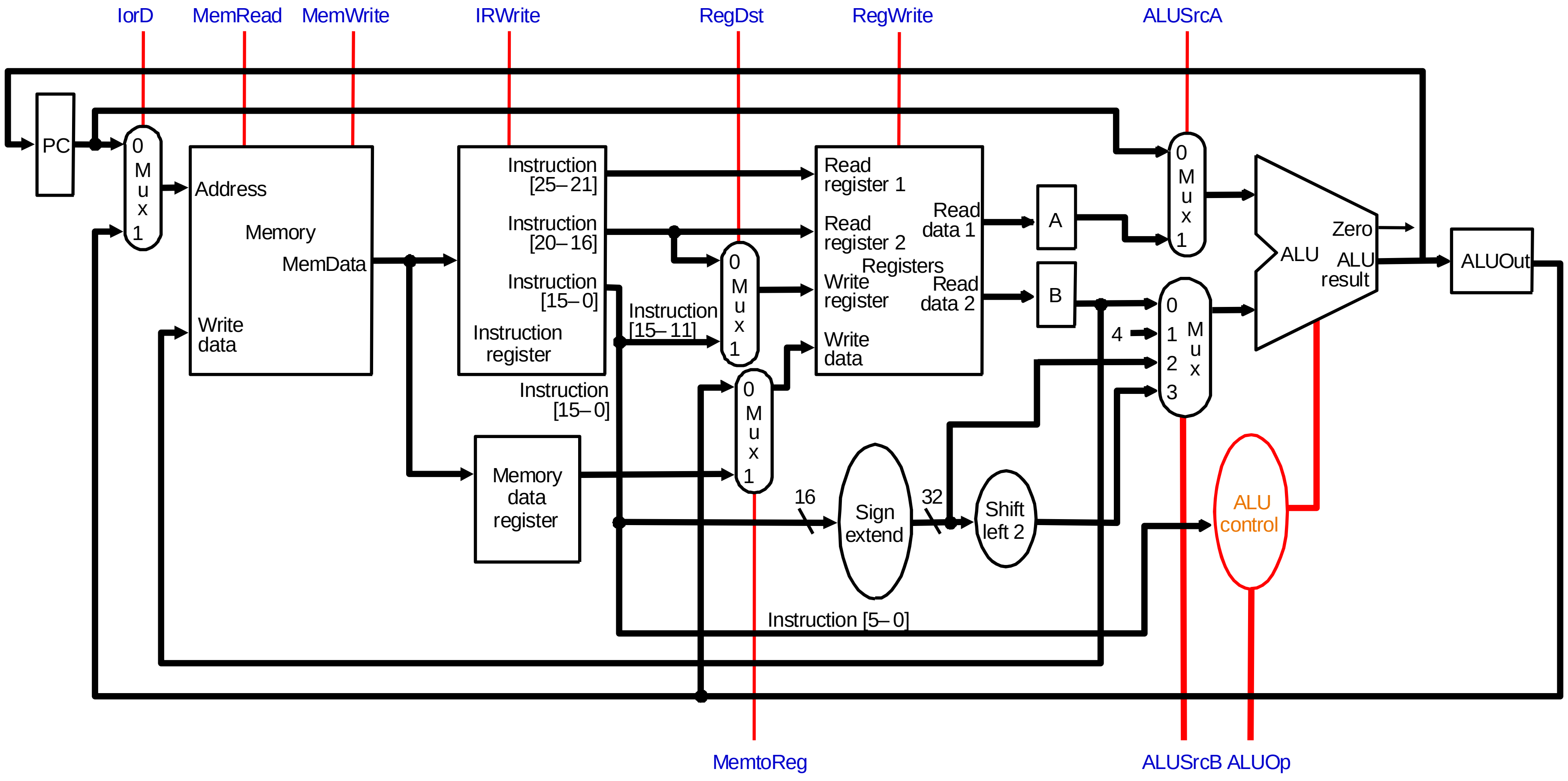


Multi-Cycle Approach

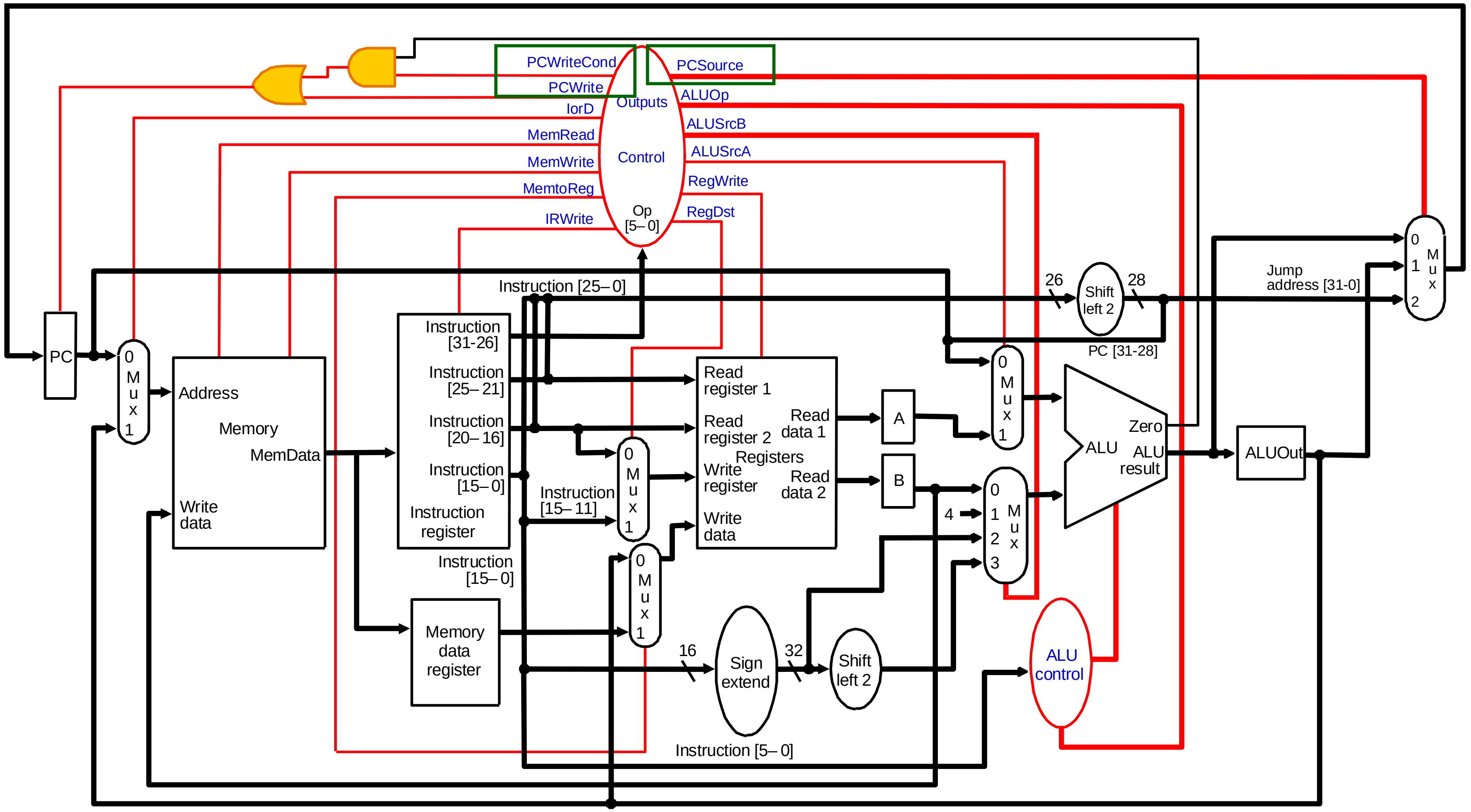
- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers



Control Signals of a Multi-Cycle Datapath



Complete Control Lines



Five Execution Steps

- Instruction Fetch (IF)
- Instruction Decode and Register Fetch (ID)
- Execution, Memory Address Computation, or Branch Completion (EX)
- Memory Access or R-type instruction completion (MEM)
- Write-back step (WB)

Instruction Fetch (IF)

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

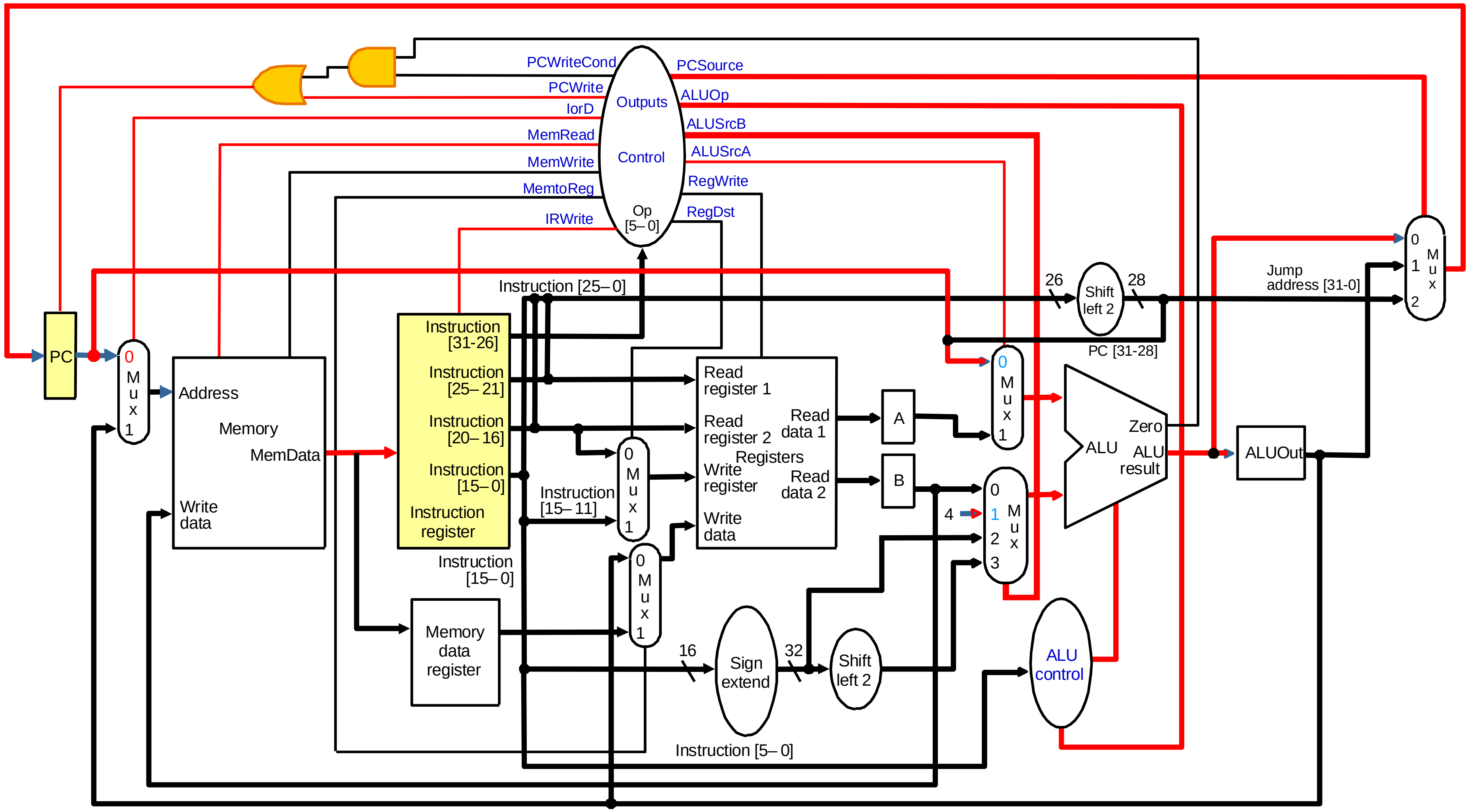
```
    IR = Memory[PC];
    PC = PC + 4;
```

Can we figure out the values of the control signals?

- `IR = Memory[PC];` MemRead=1; IRWrite=1; IorD=0;
- `PC = PC + 4;` ALUSrcA=0; ALUSrcB=01; ALUOp=00 (add); PCSource=00; PCWrite=1

Instruction Fetch Control
IR = Memory[PC]; PC+=4

```
IR = Memory[PC]; MemRead=1; IRWrite=1; IorD=0;
PC = PC + 4;      ALUSrcA=0; ALUSrcB=01; ALUOp=00 (add);
                  PCSource=00; PCWrite=1
```



Instruction Decode and Register Fetch (ID)

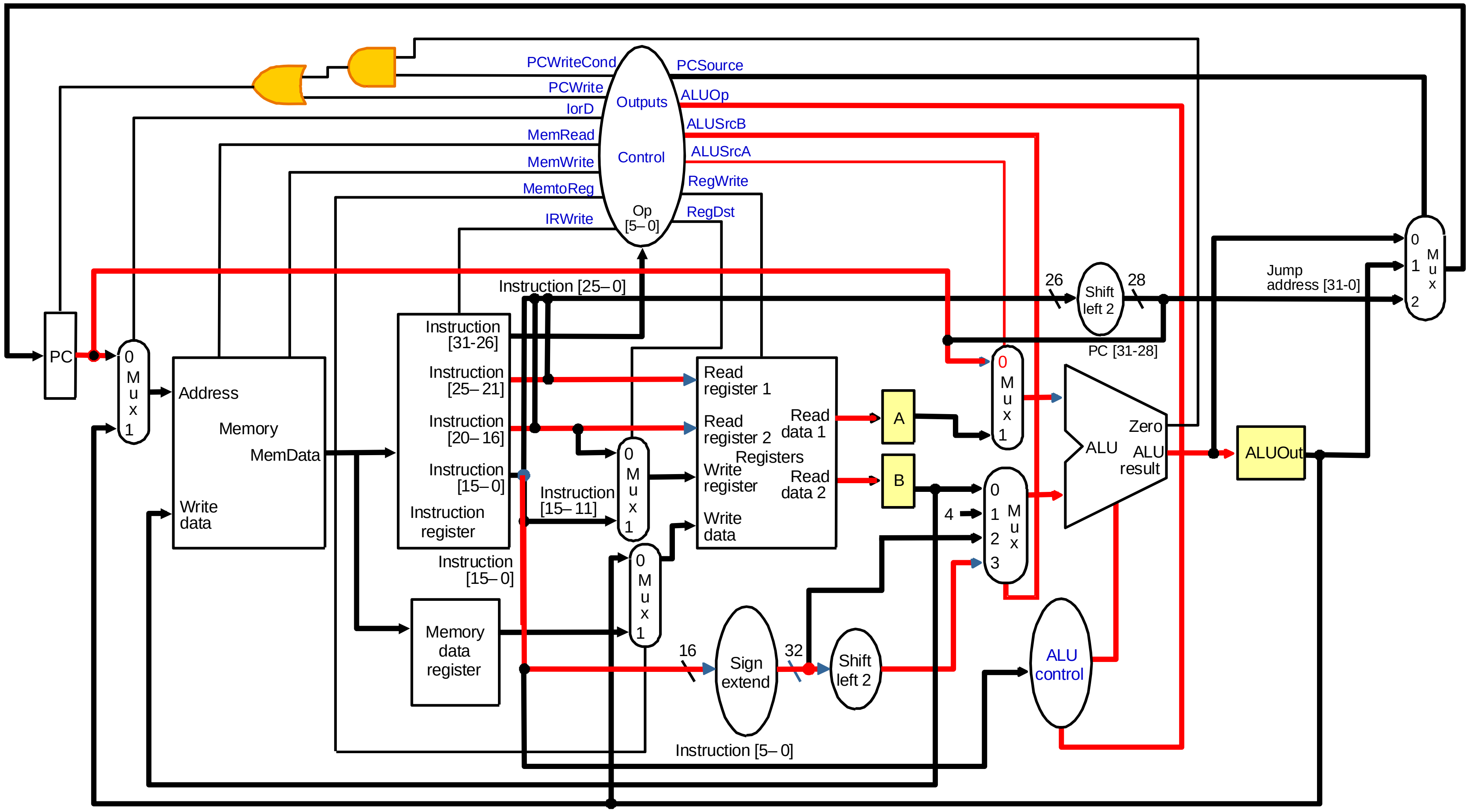
- Still do not have any idea what instruction it is.
- Read registers rs and rt in case we need them
- Compute the branch address (used in next cycle in case the instruction is a branch)
- RTL:

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

ALUSrcA = 0; ALUSrcB = 11; ALUOp = 00 (add); (for branch target)

ID Stage: Assign A and B; Calculate Branch Address

ALUSrcA = 0; ALUSrcB = 11; ALUOp = 00 (add); (for branch target)



Execute, memory or branch (EX: instruction dependent)

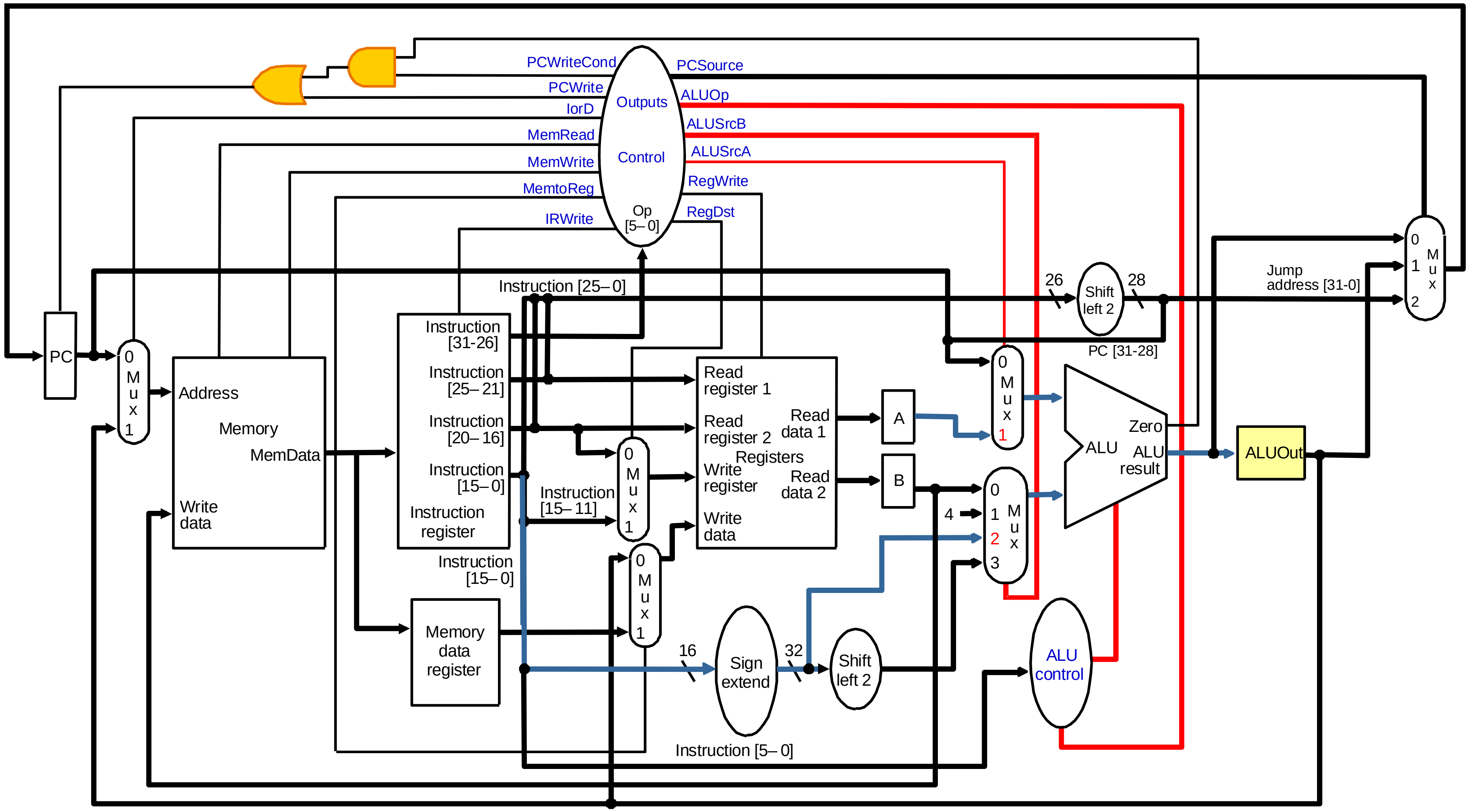
- The first cycle, the operation is determined by the instruction class
- ALU is performing one of the following functions, based on instruction type

- **Memory Reference**:
  ```
  ALUOut = A + sign-extend(IR[15-0]);
  ```
  ALUSrcA=1; ALUSrcB=10; ALUop=00 (add)
- **R-type**:
  ```
  ALUOut = A op B;
  ```
  ALUSrcA=1; ALUSrcB=00; ALUop=10 (funct, inst[5:0], decides op)
- **Branch**:
  ```
  if (A==B) PC = ALUOut;
  ```
  PCSource=01; ALUSrcA=1; ALUSrcB=00; ALUop=01 (sub);
  PCWriteCond=1; PCWrite=0;
- **Jump**:
  ```
  PC = {PC[31:28] || IR[25:0] << 2‘b00};
  ```
  PCSource=10; PCWrite=1;

Execute: Memory Type ALUOut = A + offset (address)

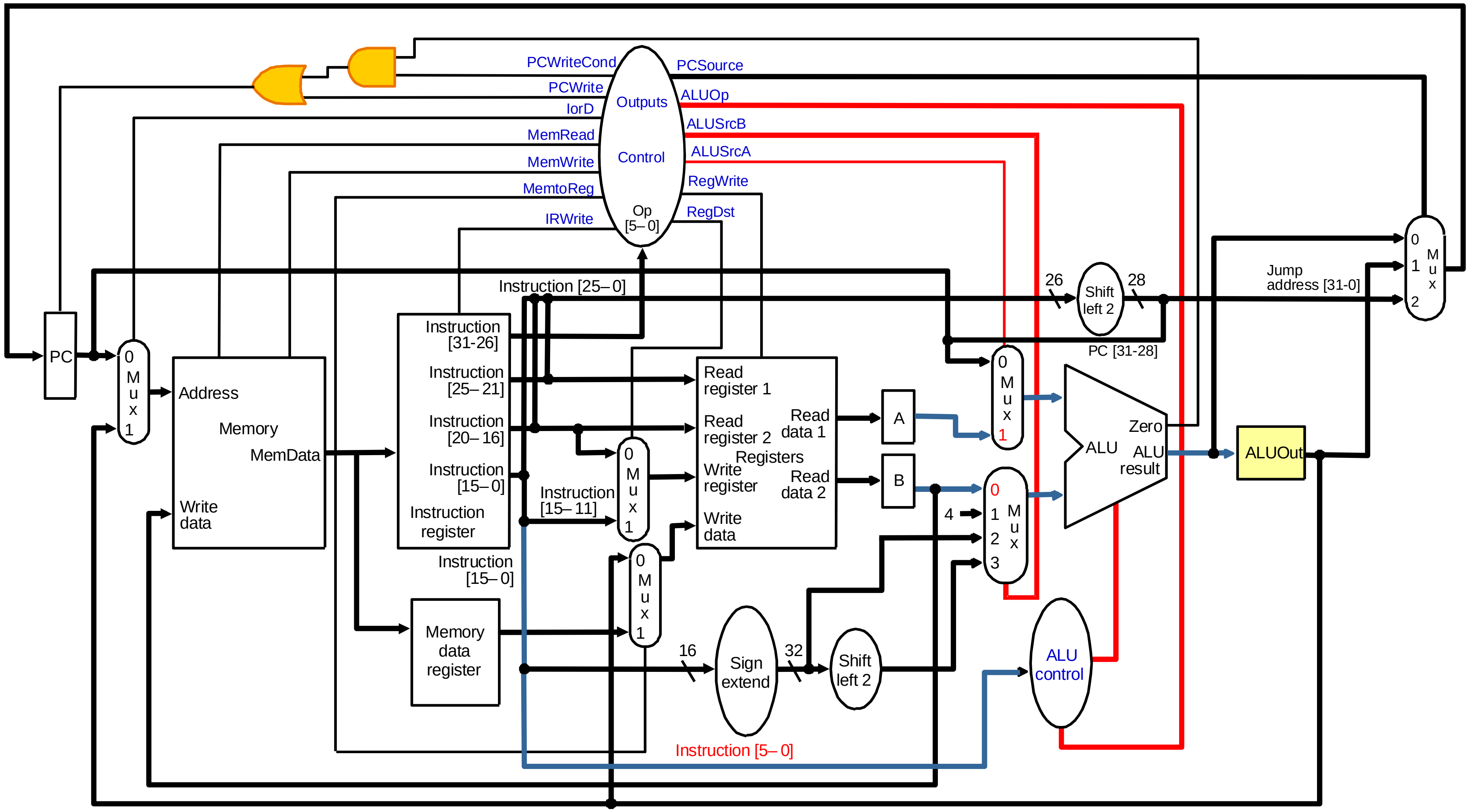
```
ALUOut = A + sign-extend(IR[15-0]);
```

ALUSrcA=1; ALUSrcB=10; ALUop=00 (add)



Execute: R- Type
ALUOut = A op B

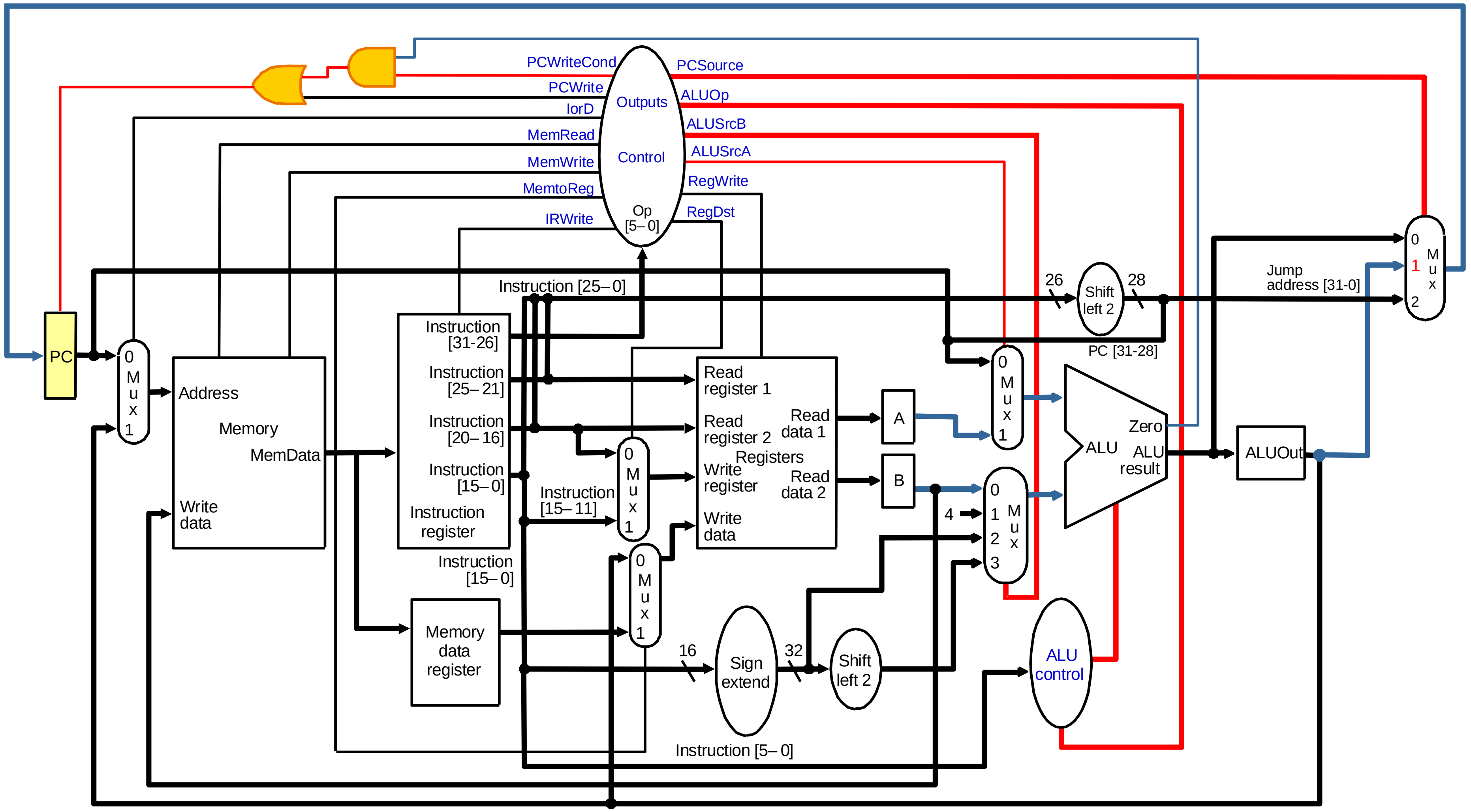
```
ALUOut = A op B;
```
ALUSrcA=1; ALUSrcB=00; ALUop=10 (funct, inst[5:0], decides op)



Execute: Branch Type
if (A==B) PC=ALUOut

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

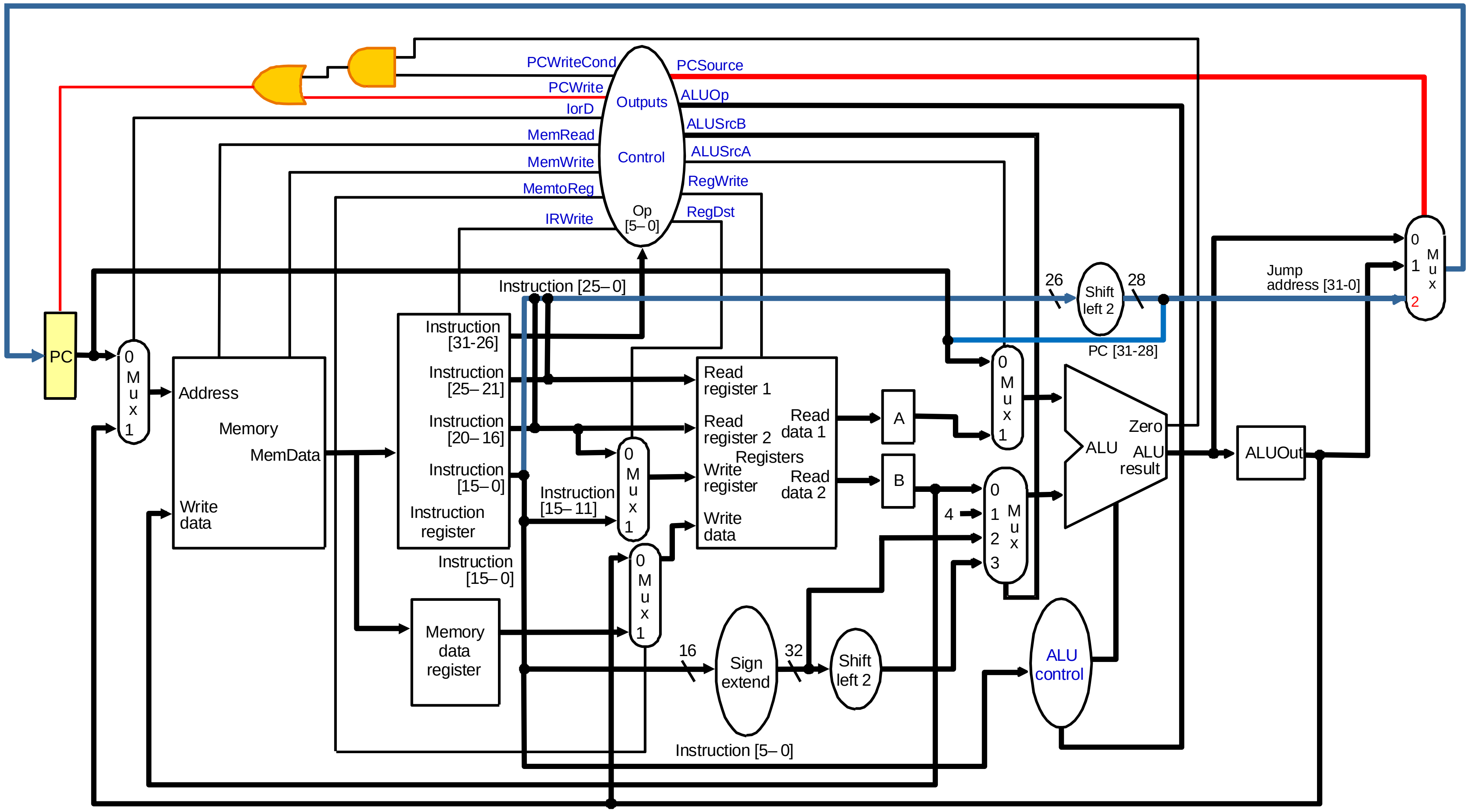
PCSource=01; ALUSrcA=1; ALUSrcB=00; ALUop=01 (sub); PCWriteCond=1; PCWrite=0;



Execute: Jump Type (New PC)

```
PC = {PC[31:28] || IR[25:0] << 2'b00};
```

PCSource=10; PCWrite=1;



Memory Access or R-Type

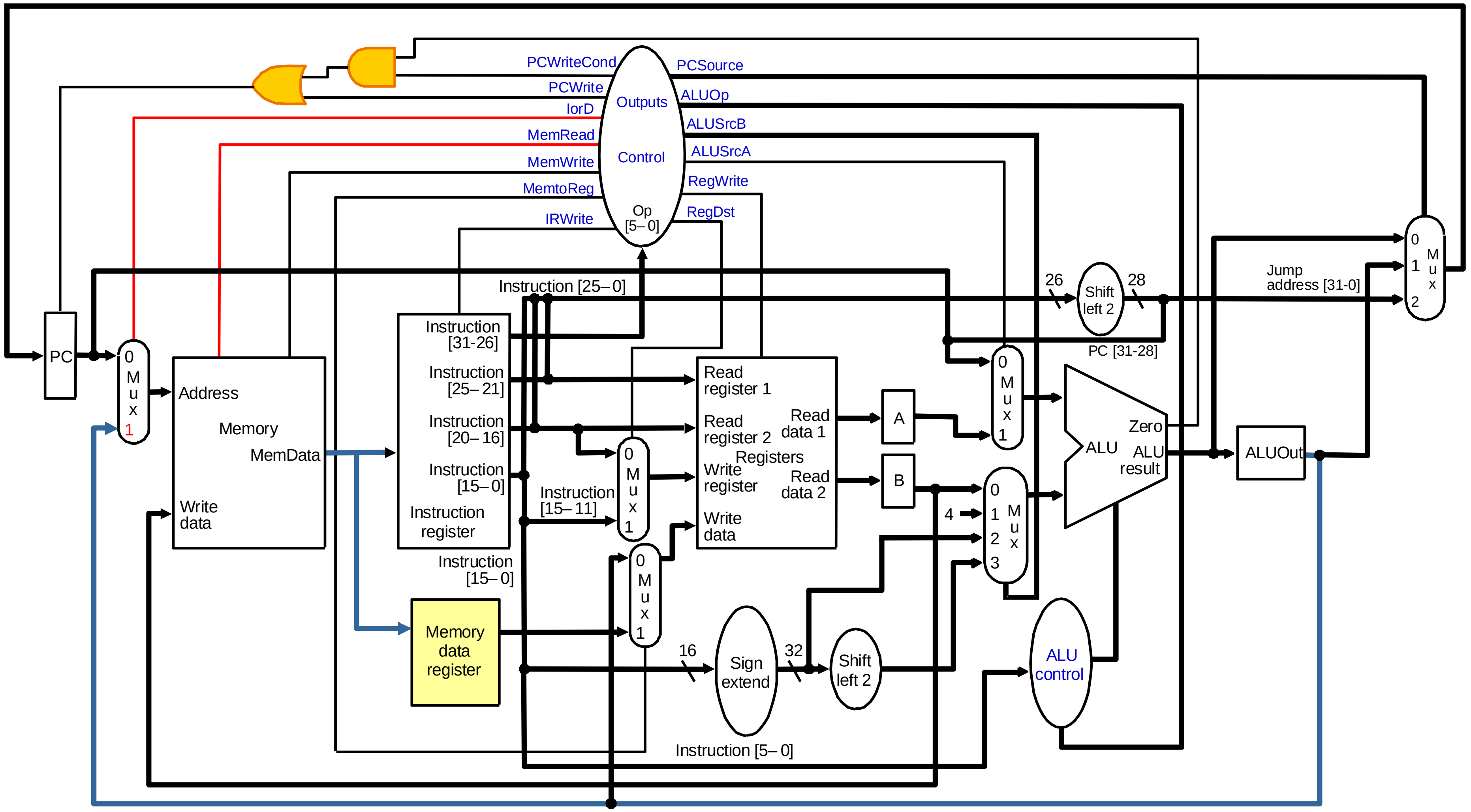
- Loads and stores access memory

```
MDR = Memory[ALUOut];IorD=1; MemRead=1;
or
Memory[ALUOut] = B; IorD=1; MemWrite=1
```

- R-type instructions finish

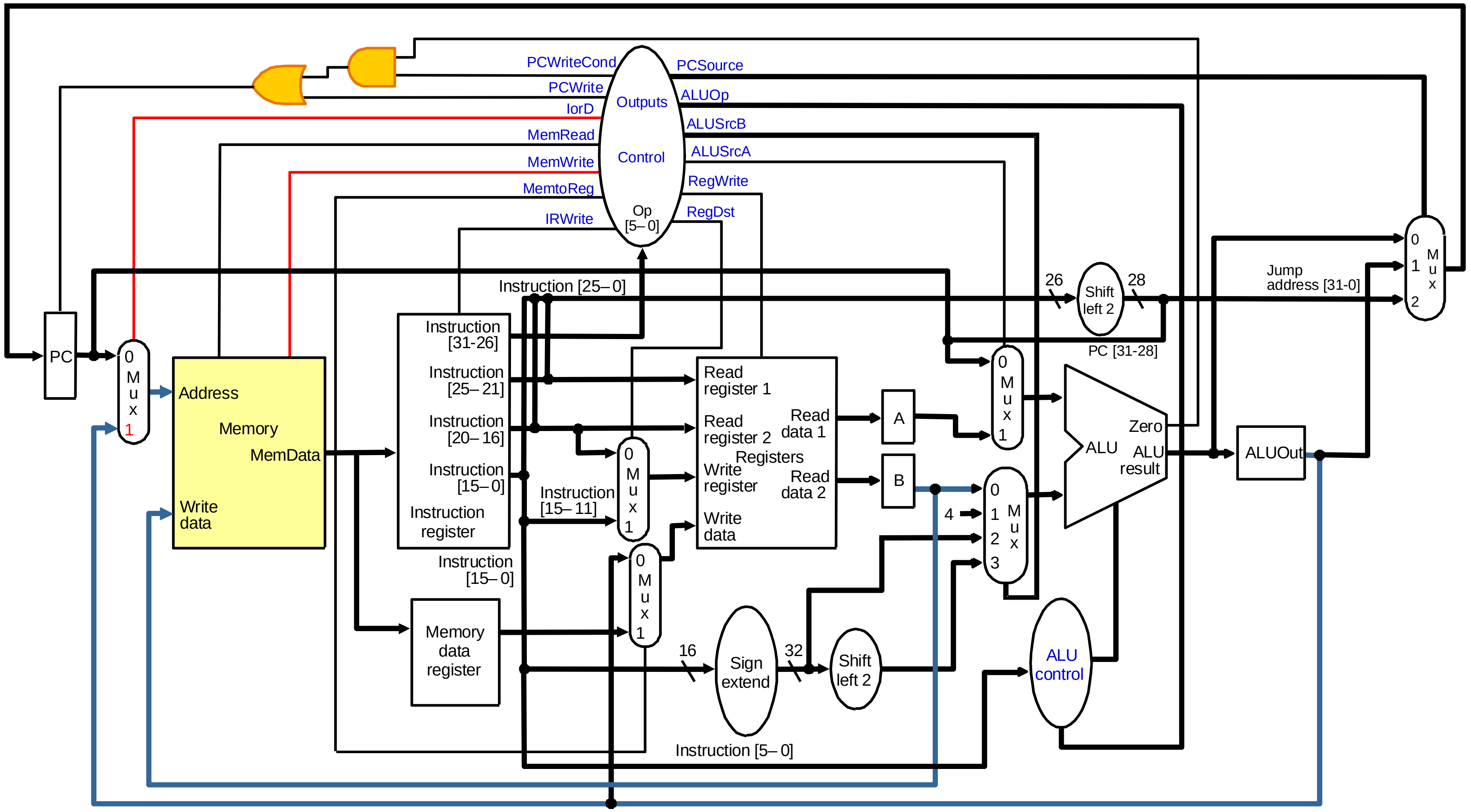
```
Reg[IR[15-11]] = ALUOut; RegDst=1; MemtoReg=0;
                         RegWrite =1
```

MEM: Load

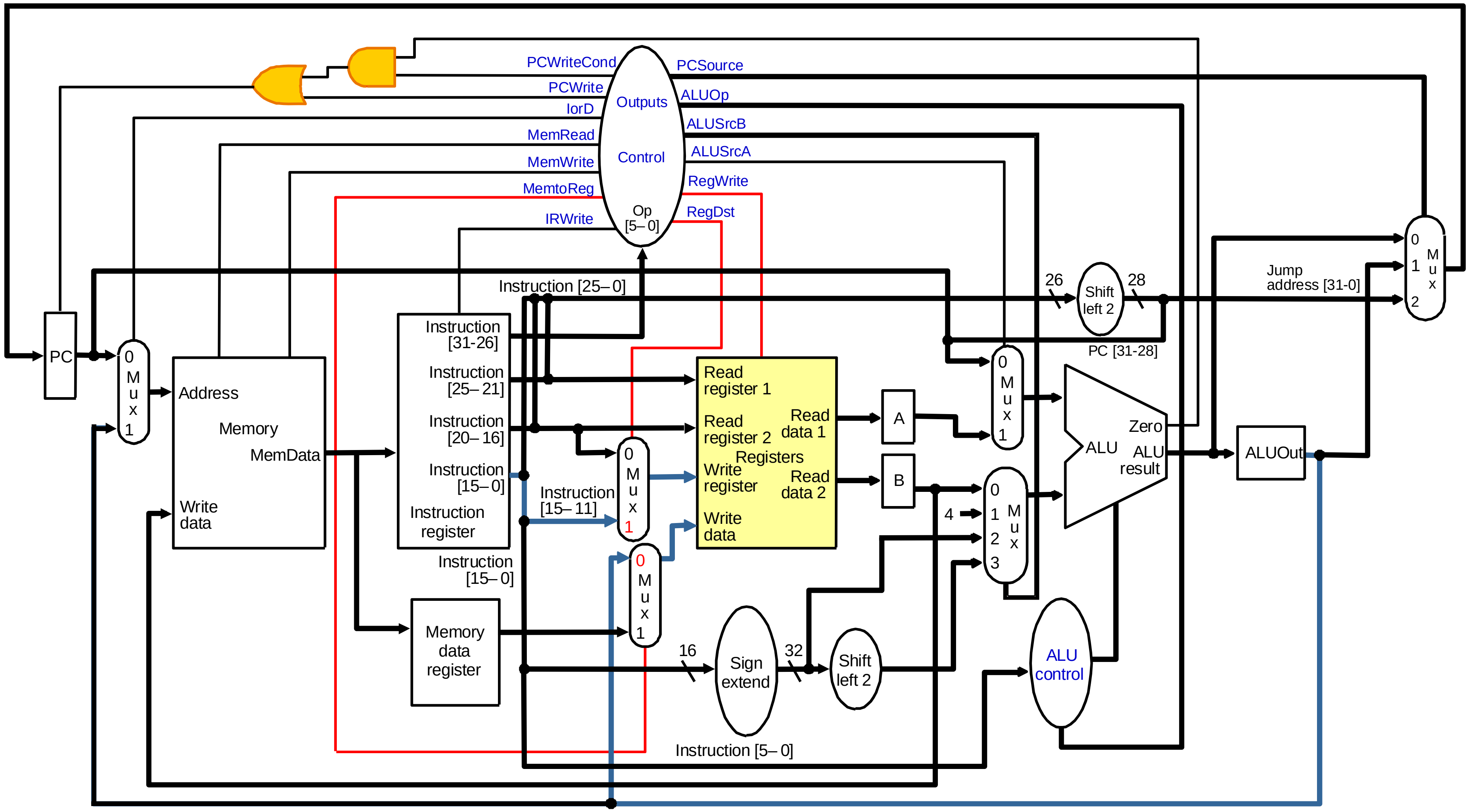


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op [5– 0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0 Mux 1
Address
Memory
MemData
Write data
Instruction [31-26]
Instruction [25– 21]
Instruction [20– 16]
Instruction [15– 0]
Instruction register
Instruction [15– 0]
Memory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

MEM: Store


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

MEM: R-Type Completion

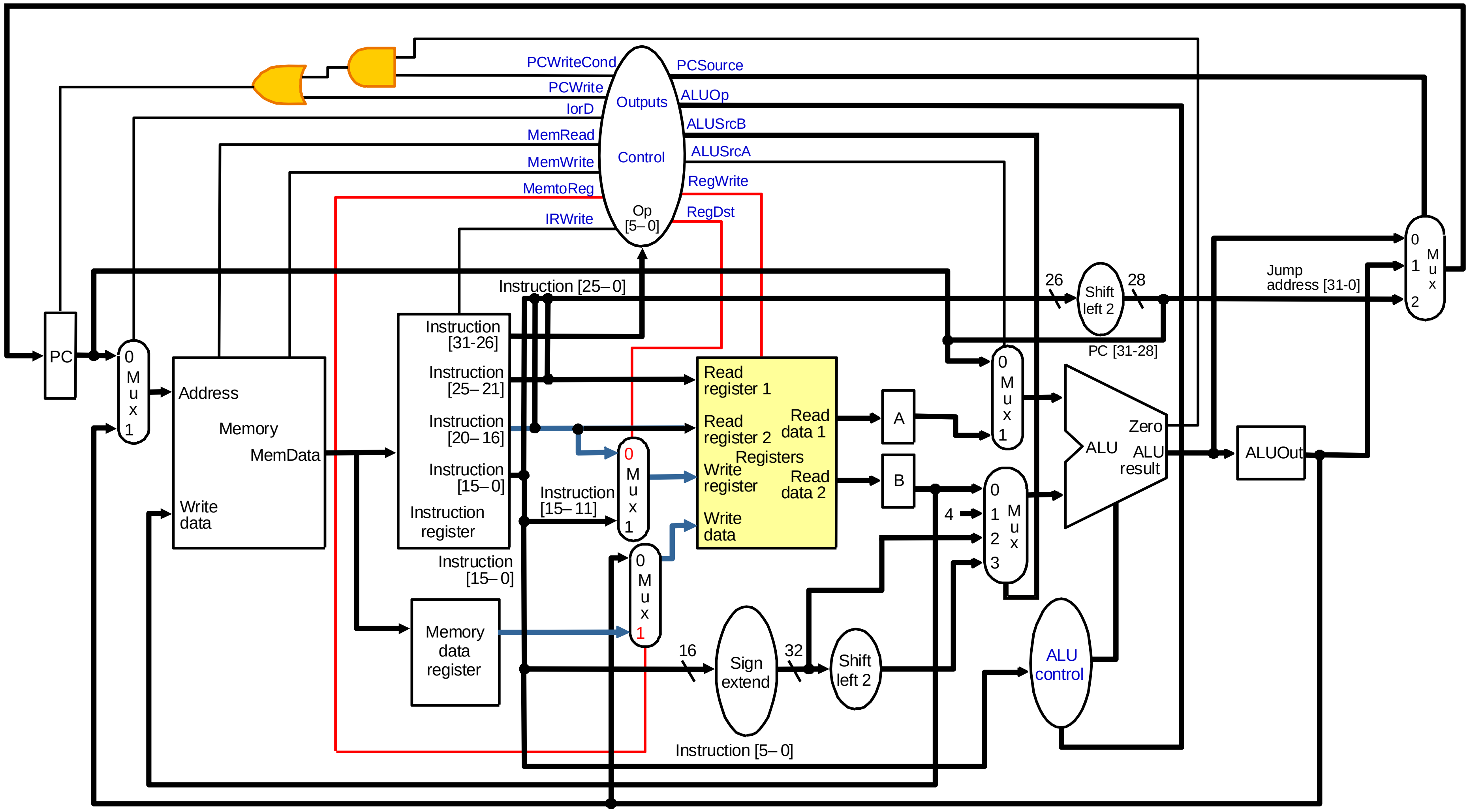


Write-back for load operation

- `Reg[IR[20-16]]= MDR;` MemtoReg=1; RegWrite=1; RegDst=0;

What about all the other instructions?

WB for Load



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

Summary

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

Instruction fetch	IR = Memory[PC]
	PC = PC + 4			

Instruction decode/register fetch	A = Reg [IR[25-21]]
	B = Reg [IR[20-16]]
	ALUOut = PC + (sign-extend (IR[15-0]) << 2)			

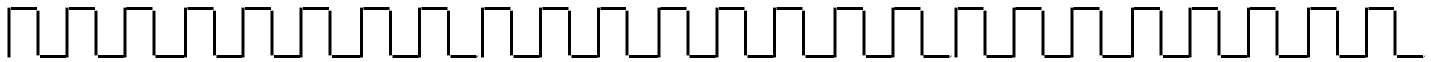
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Simple 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
          add $t5, $t2, $t3
          sw $t5, 8($t3)
Label:    ...
```

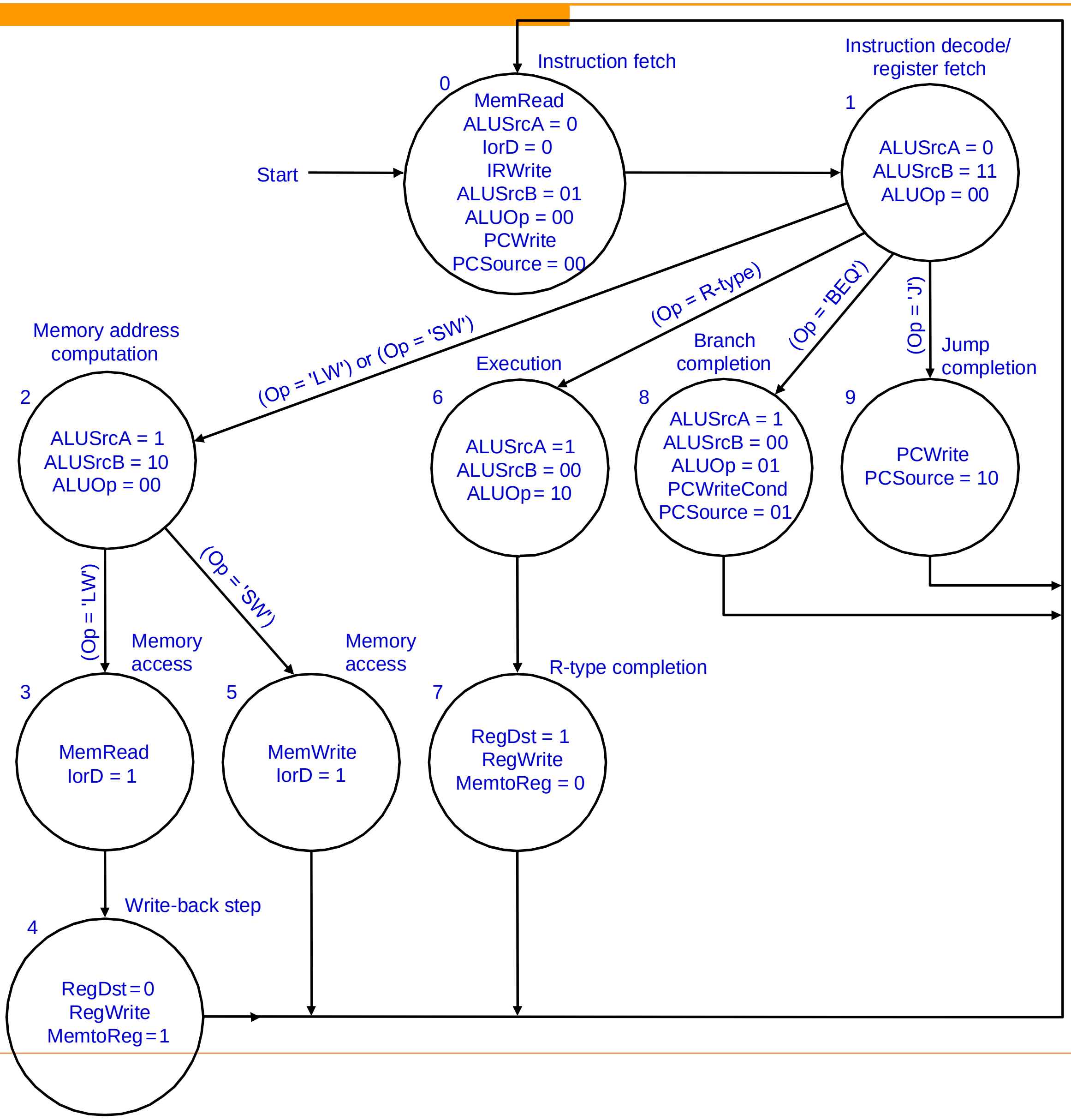
- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` takes place?



Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

Graphical Specification of FSM



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
3
MemRead
IorD = 1
Memory
access
5
MemWrite
IorD = 1
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst=0
RegWrite
MemtoReg=1