Multi-cycle Processor

(Computer Organization: Chapter 4)

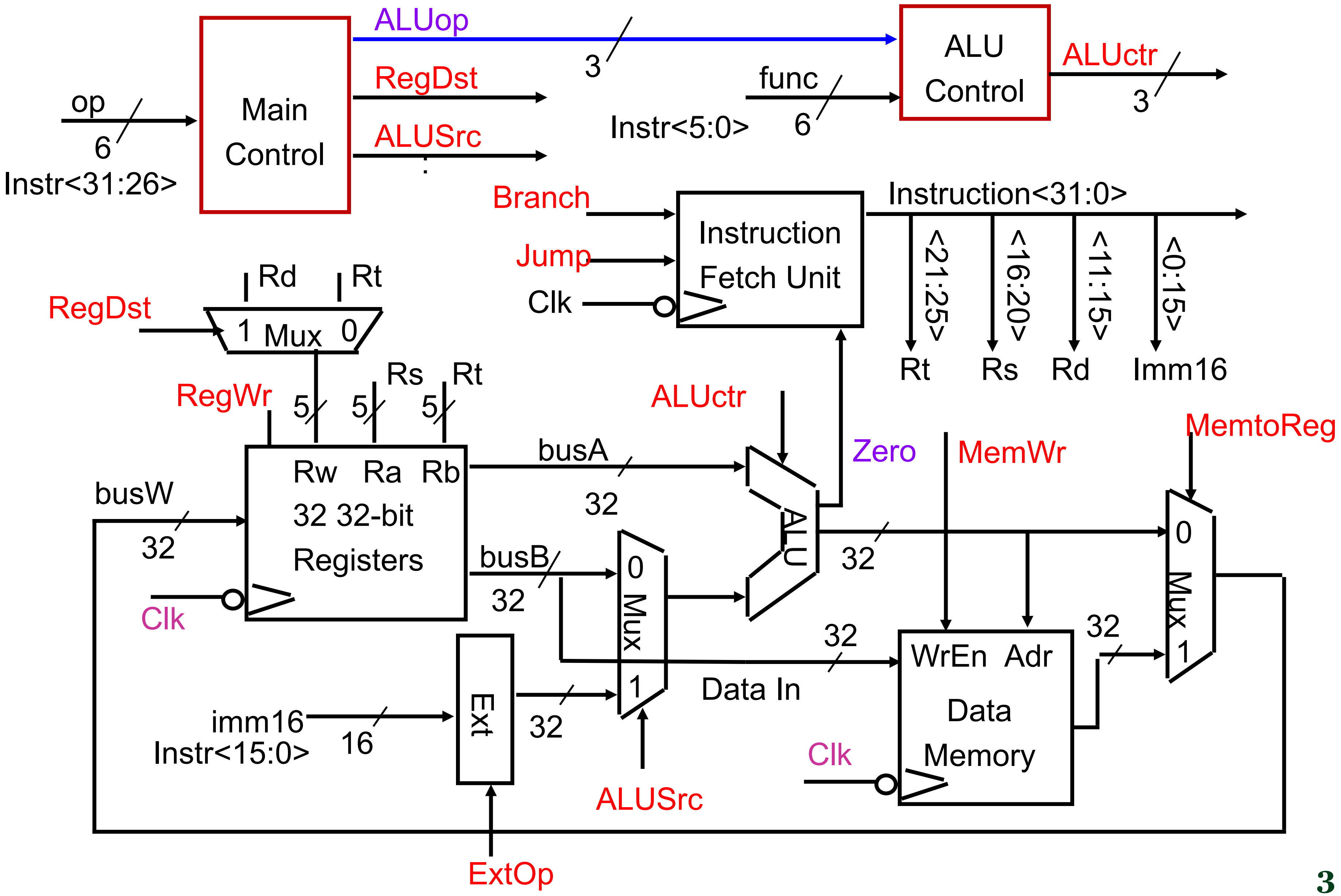
Yanyan Shen

Department of Computer Science and Engineering

Recap: A Summary of Control Signals

inst	Register Transfer	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”	
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”	
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
	ALUsrc = Im, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel = “+4”	
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr, nPC_sel = “+4”	
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, nPC_sel = “+4”	
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4	
	nPC_sel = “Br”, ALUctr = “sub”	

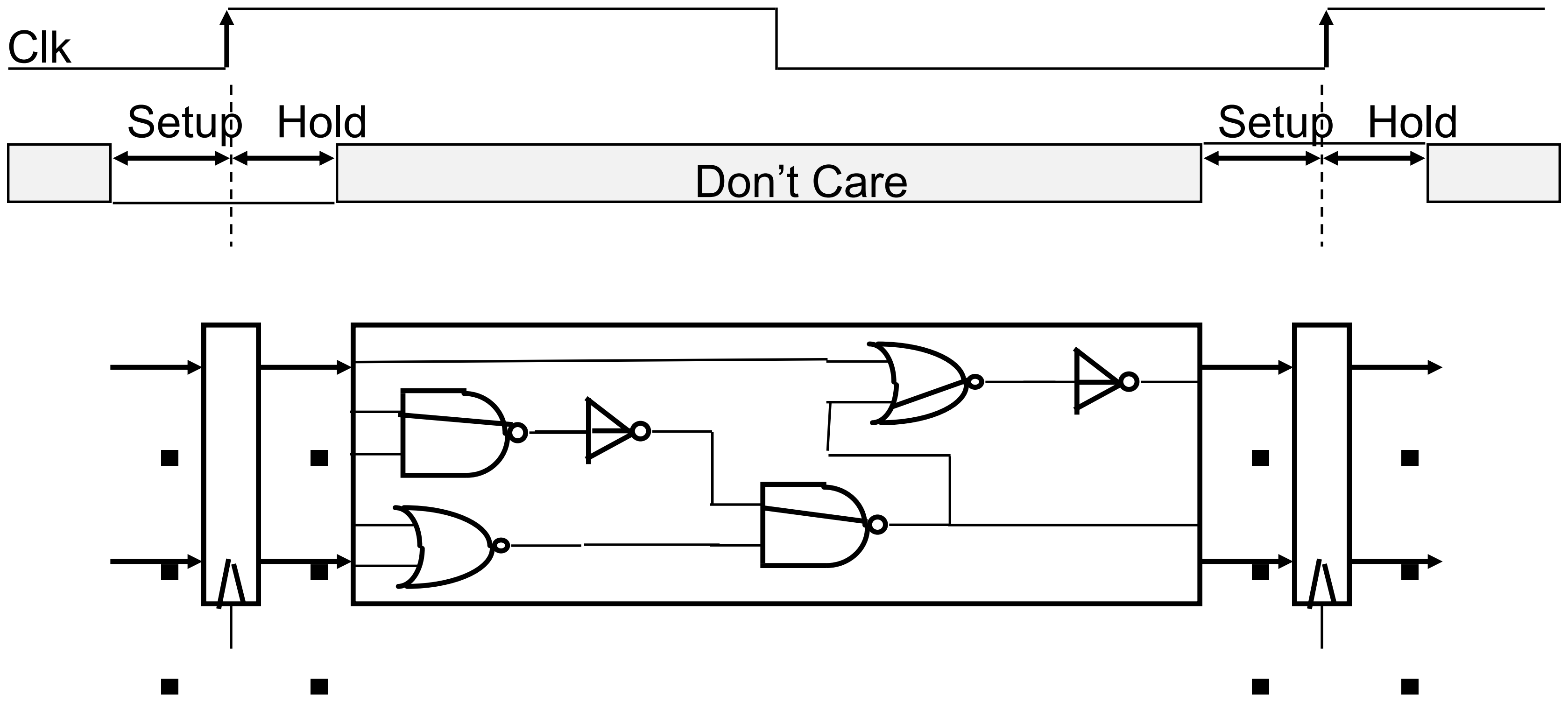
The Complete Single Cycle Data Path with Control



Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path

Clocking Methodology



- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew

Instruction Critical Paths

- Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times) except:

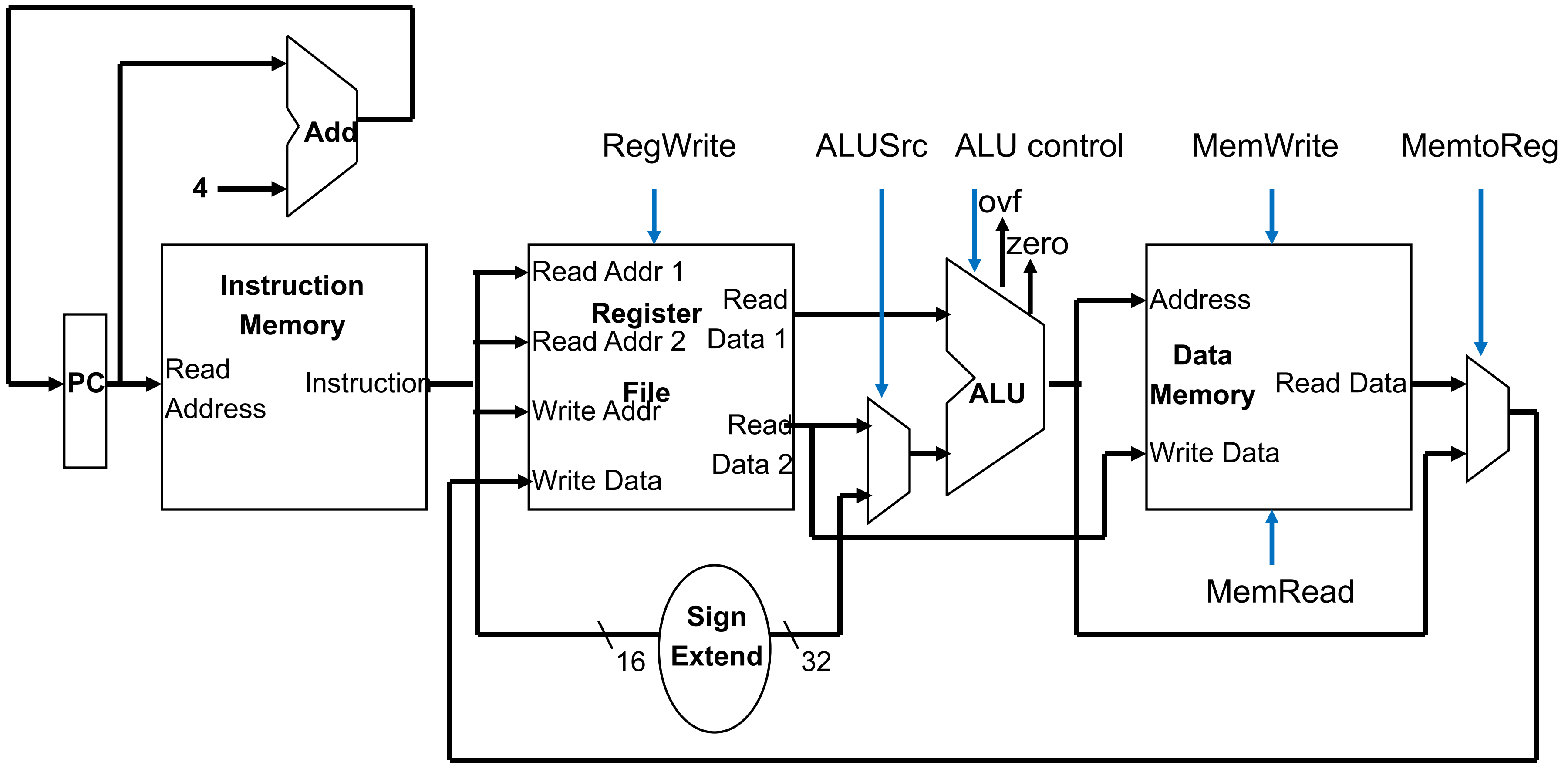
  Instruction and Data Memory (200 ps)

  ALU and adders (100 ps)

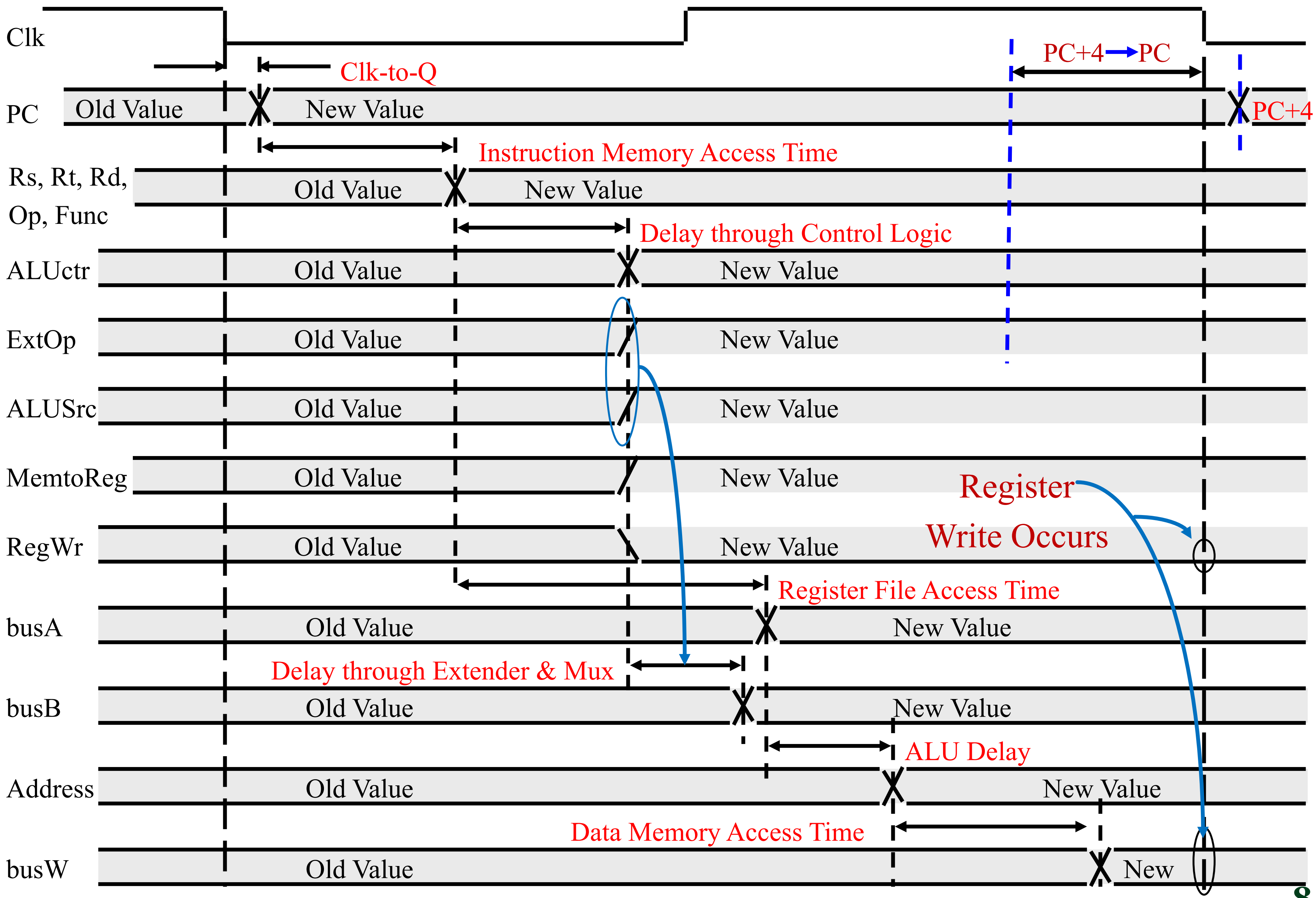
  Register File access (reads or writes) (50 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	50	100	0	50	400
load	200	50	100	200	50	600
store	200	50	100	200		550
beq	200	50	100	0		350
jump	200					200

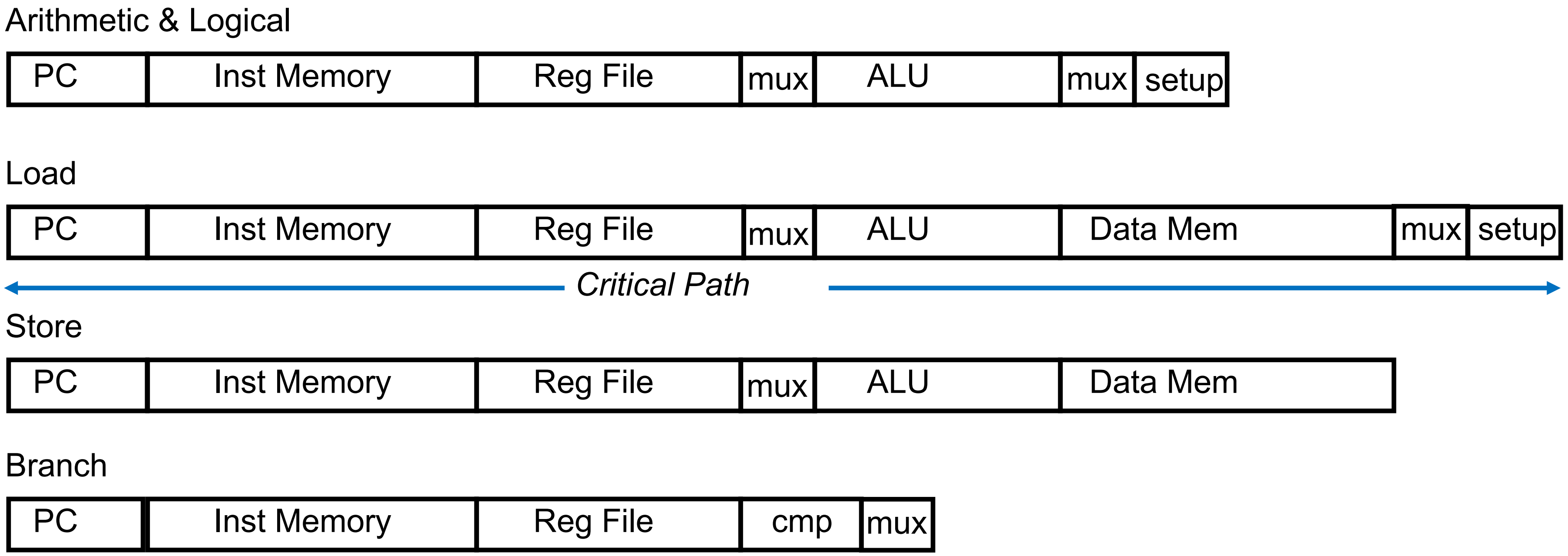
Fetch, R, and Memory Access Portions



Time Delay for LW: Critical Path

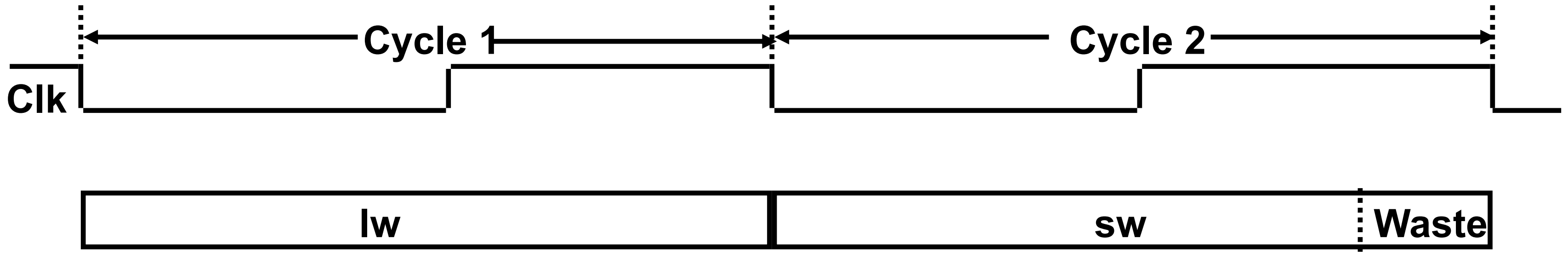


What's wrong with our CPI=1 processor?



Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



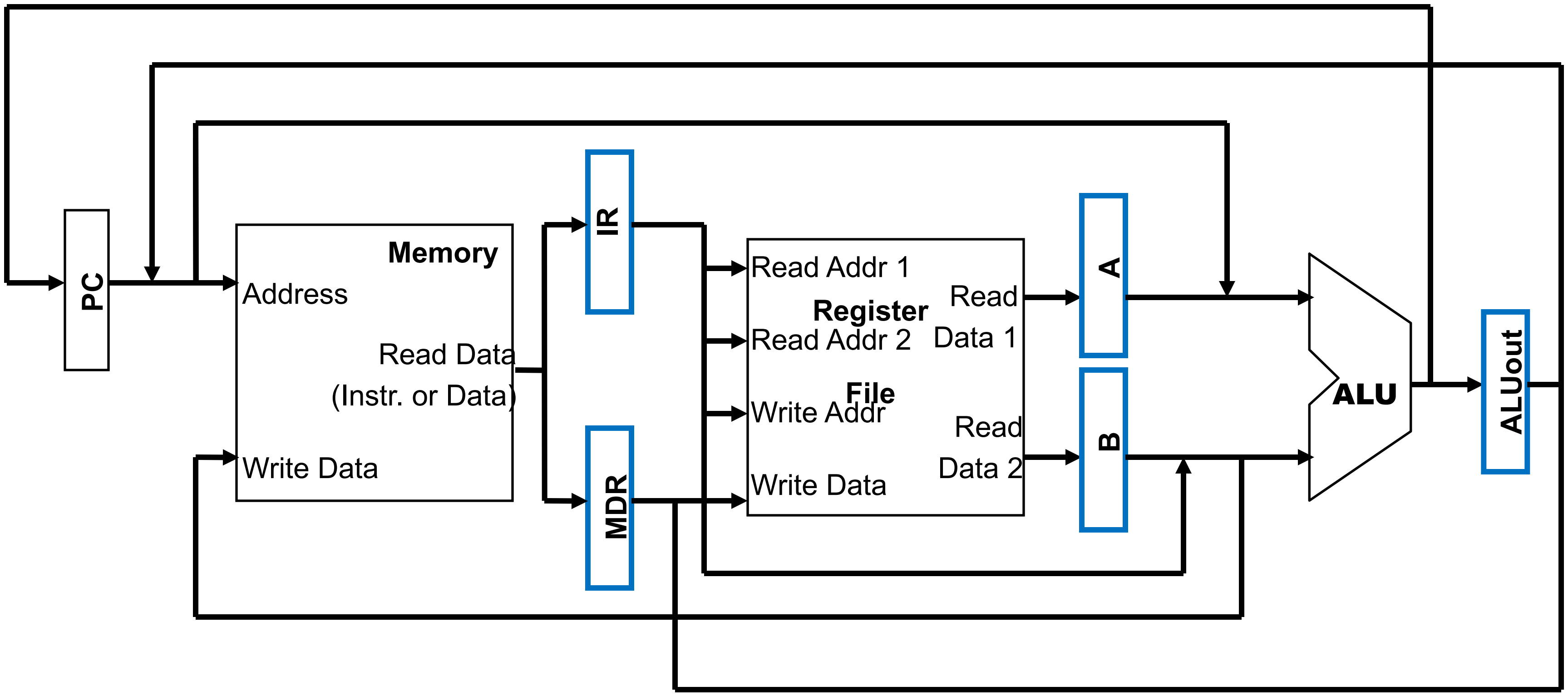
- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle
- But is simple and easy to understand

Multicycle Implementation Overview

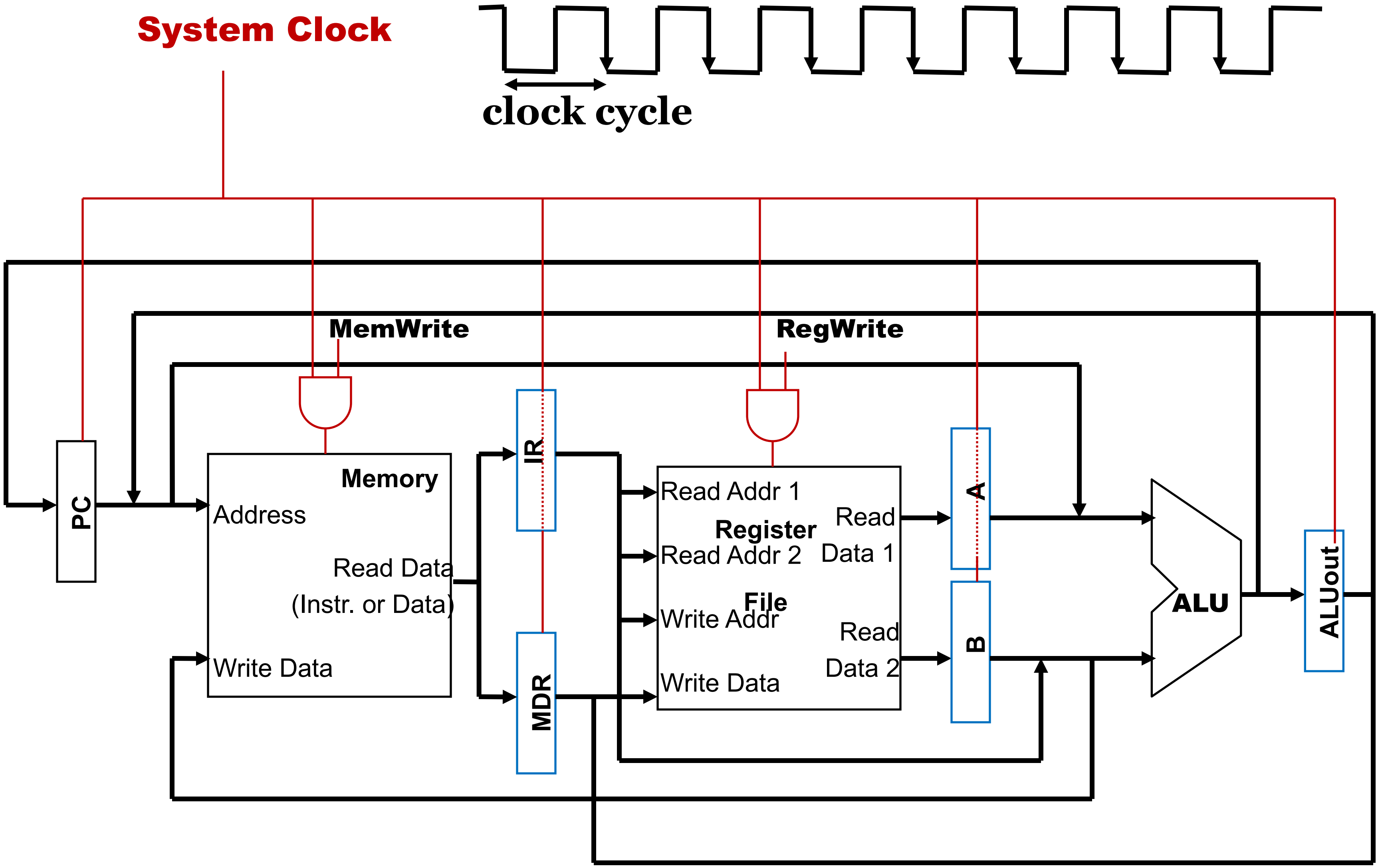
- Each instruction step takes 1 clock cycle
  - Therefore, an instruction takes more than 1 clock cycle to complete
- Not every instruction takes the same number of clock cycles to complete
- Multicycle implementations allow
  - faster clock rates
  - different instructions to take a different number of clock cycles
  - functional units to be used more than once per instruction as long as they are used on different clock cycles, as a result
    - only need one memory
    - only need one ALU/adder

The Multicycle Datapath – A High Level View

- Registers have to be added after every major functional unit to hold the output value until it is used in a subsequent clock cycle



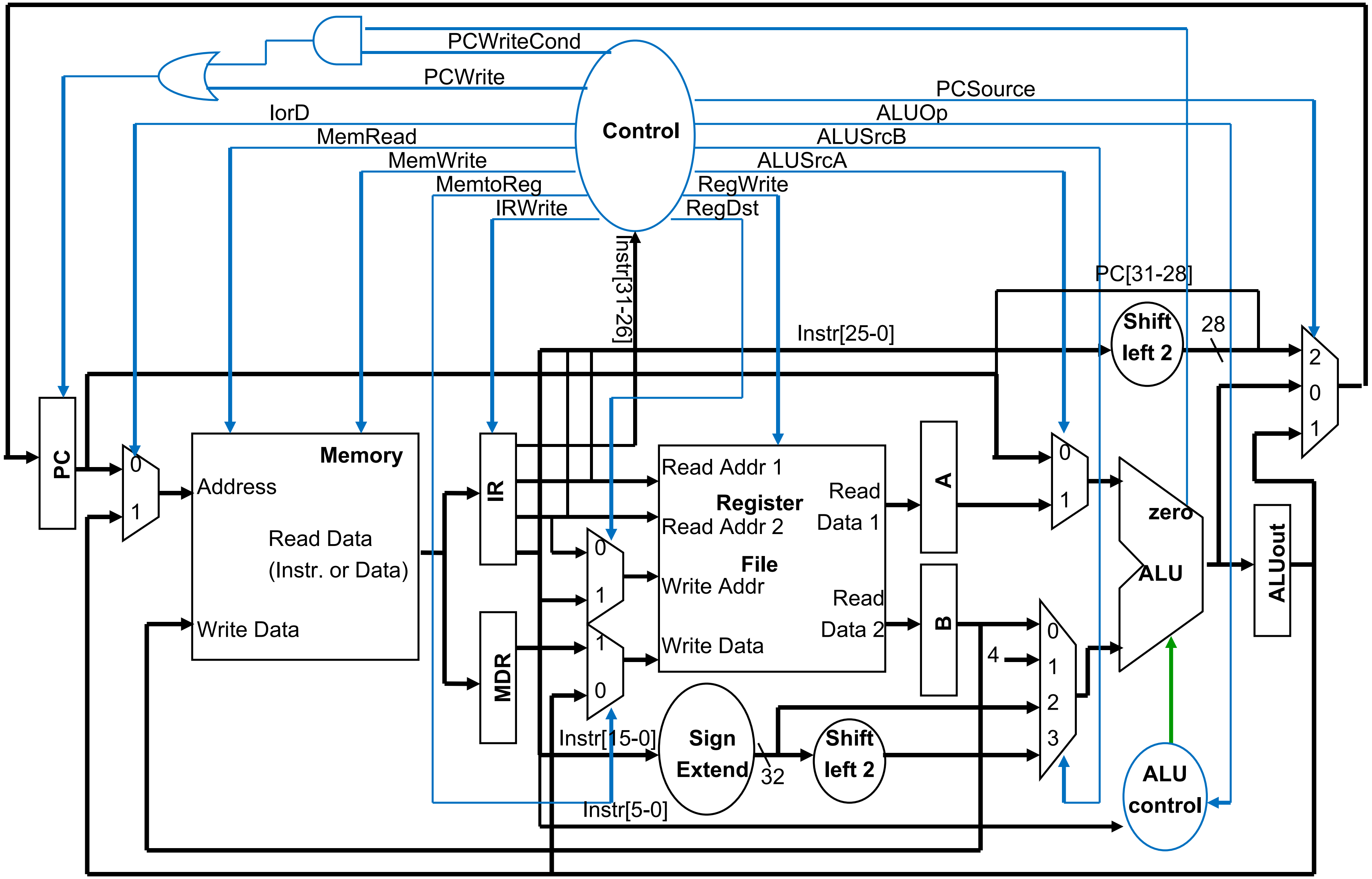
Clocking the Multicycle Datapath



Our Multicycle Approach

- Break up the instructions into steps where each step takes a clock cycle while trying to
  - balance the amount of work to be done in each step
  - use only one major functional unit per clock cycle
- At the end of a clock cycle
  - Store values needed in a later clock cycle by the current instruction in a state element (internal register not visible to the programmer)
    - IR – Instruction Register
    - MDR – Memory Data Register
    - A and B – Register File read data registers
    - ALUout – ALU output register
    - All (except IR) hold data only between a pair of adjacent clock cycles (so they don't need a write control signal)
  - Data used by subsequent instructions are stored in programmer visible state elements (i.e., Register File, PC, or Memory)

The Complete Multicycle Datapath with Control



Our Multicycle Approach, con't

- Reading from or writing to any of the internal registers, Register File, or the PC occurs (quickly) at the beginning (for read) or the end of a clock cycle (for write)
- Reading from the Register File takes ~50% of a clock cycle since it has additional control and access overhead (but reading can be done in parallel with decode)
- Had to add multiplexors in front of several of the functional unit input ports (e.g., Memory, ALU) because they are now shared by different clock cycles and/or do multiple jobs
- All operations occurring in one clock cycle occur in parallel
  - This limits us to one ALU operation, one Memory access, and one Register File access per clock cycle

Five Instruction Steps

1) Instruction Fetch
2) Instruction Decode and Register Fetch
3) R-type Instruction Execution, Memory Read/Write Address Computation, Branch Completion, or Jump Completion
4) Memory Read Access, Memory Write Completion or R-type Instruction Completion
5) Memory Read Completion (Write Back)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

RTL for Instructions

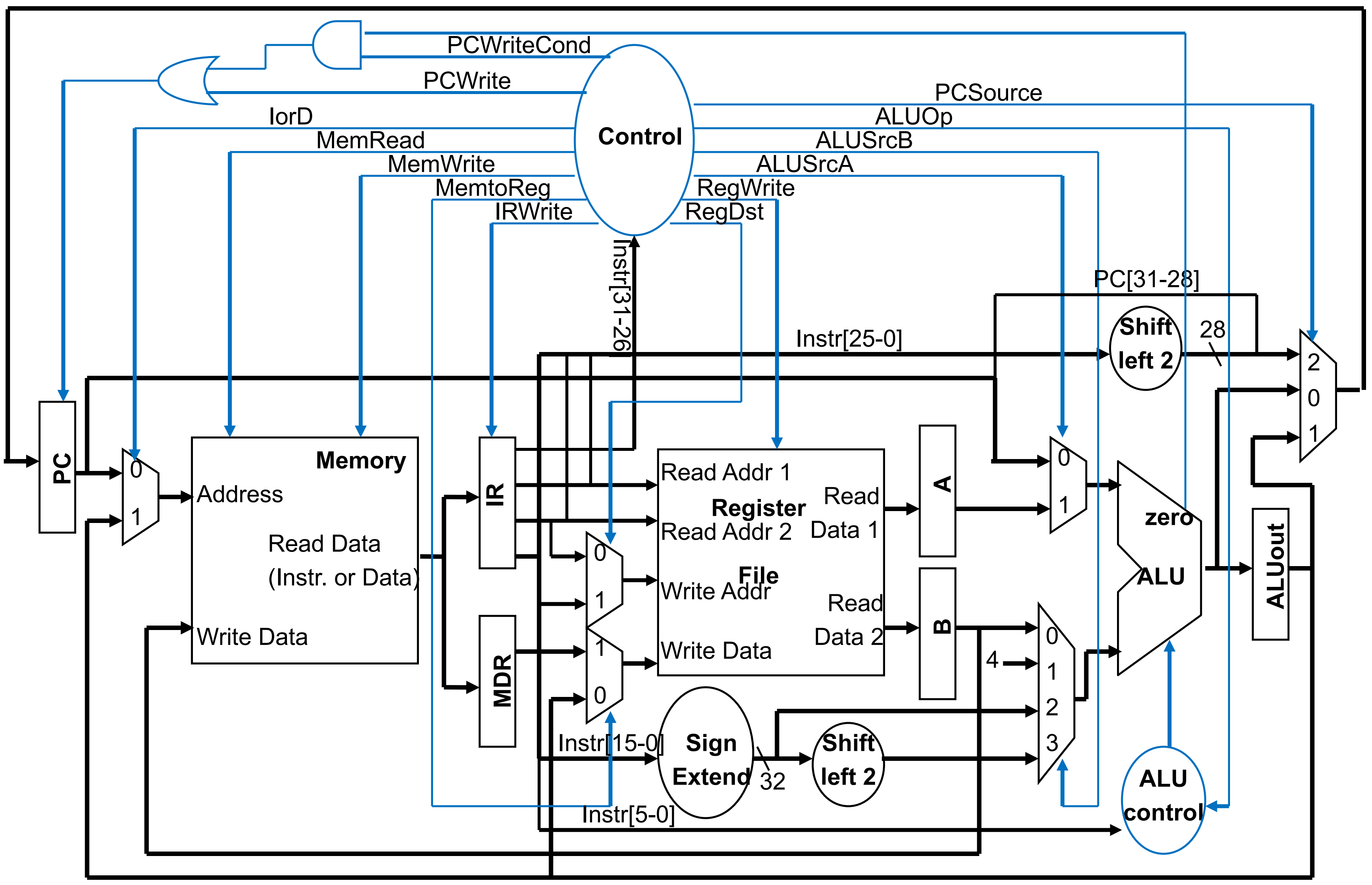
- Common Steps:
  - `Instr fetch   IR = Memory[PC];`
  - `PC Updating   PC = PC + 4;`
- Decode and Register reading

  -
    ```
    A = Reg[IR[25-21]];
    B = Reg[IR[20-16]];
    ```
- Instruction Dependent operation

RTL Summary

Step	R-type	Mem Ref	Branch	Jump
Instr fetch	IR = Memory[PC]; PC = PC + 4;			
Decode	A = Reg[IR[25-21]]; B = Reg[IR[20-16]]; ALUOut = PC +(sign-extend(IR[15-0])<< 2);			
Execute	ALUOut = A op B;	ALUOut = A + sign-extend (IR[15-0]);	if (A==B) PC = ALUOut;	PC = PC[31-28] ||(IR[25-0] << 2);
Memory access	Reg[IR[15-11]] = ALUOut;	MDR = Memory[ALUOut]; or Memory[ALUOut] = B;		
Write-back		Reg[IR[20-16]] = MDR;		

Multi-cycle datapath



Step 1: Instruction Fetch

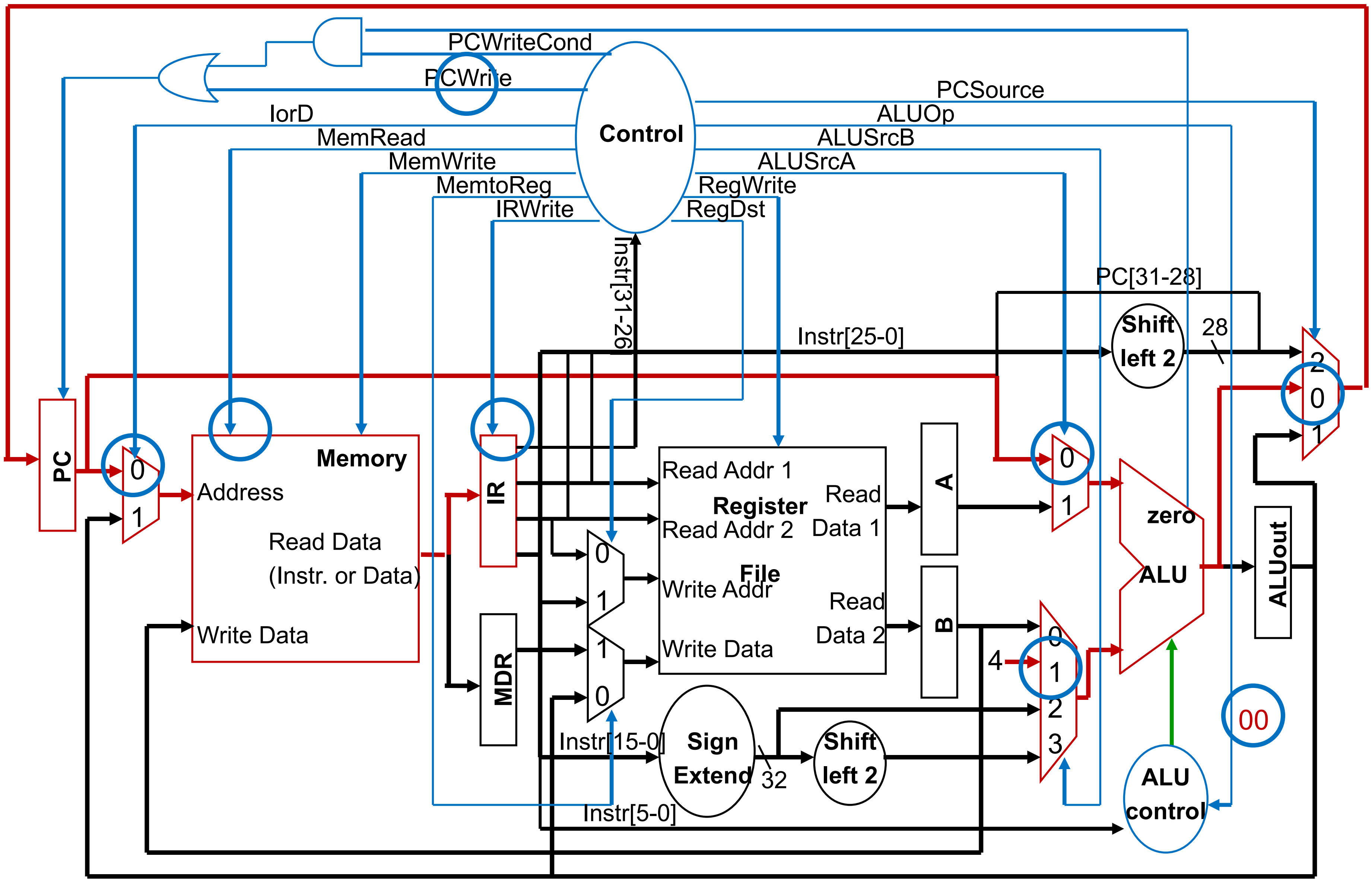
- Use PC to get instruction from the memory and put it in the Instruction Register
- Increment the PC by 4 and put the result back in the PC
- Can be described succinctly using the RTL "Register-Transfer Language“

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

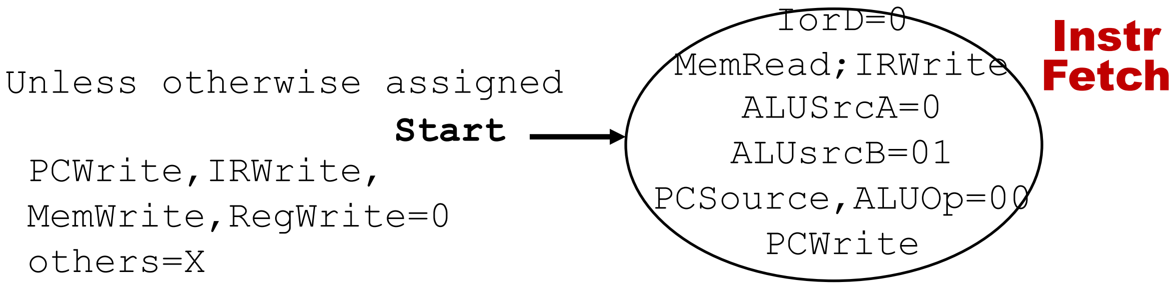
Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Datapath Activity During Instruction Fetch



Fetch Control Signals Settings



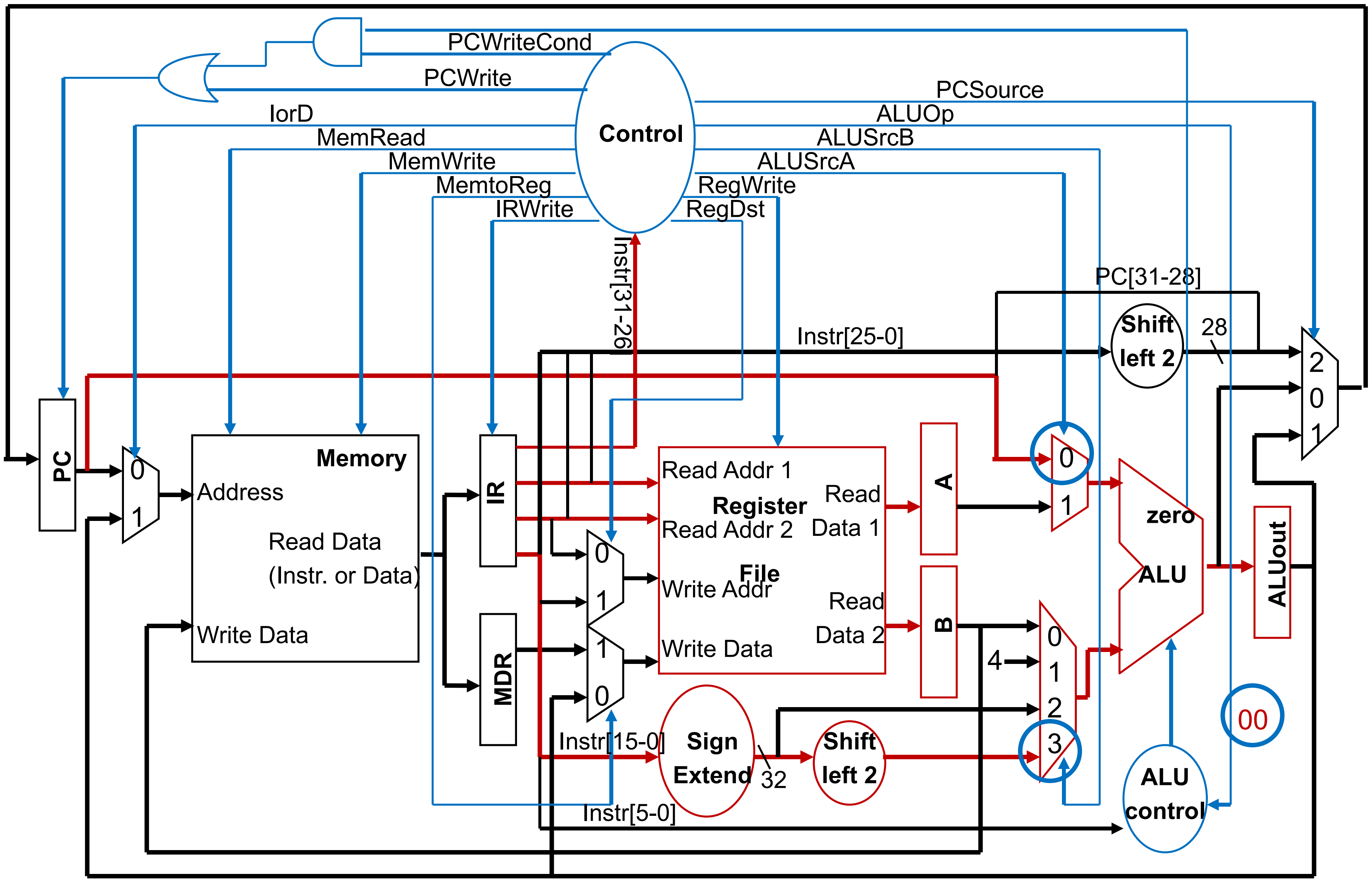
Step 2: Instruction Decode and Register Fetch

- Don’t know what the instruction is yet, so can only
  - Read registers rs and rt in case we need them
  - Compute the branch address in case the instruction is a branch
- The RTL:

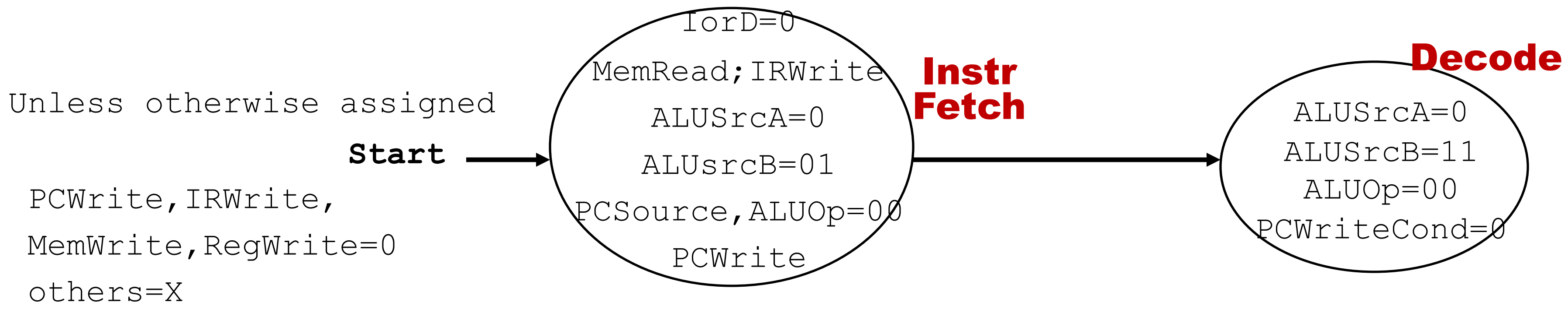
```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC+(sign-extend(IR[15-0])<< 2);
```

- Note we aren't setting any control lines based on the instruction (since we don’t know what it is (the control logic is busy "decoding" the op code bits))

Datapath Activity During Instruction Decode



Decode Control Signals Settings



Step 3 Instruction Dependent Operations

- ALU is performing one of four functions, based on instruction type
- Memory reference (`lw` and `sw`):

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

- R-type:

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

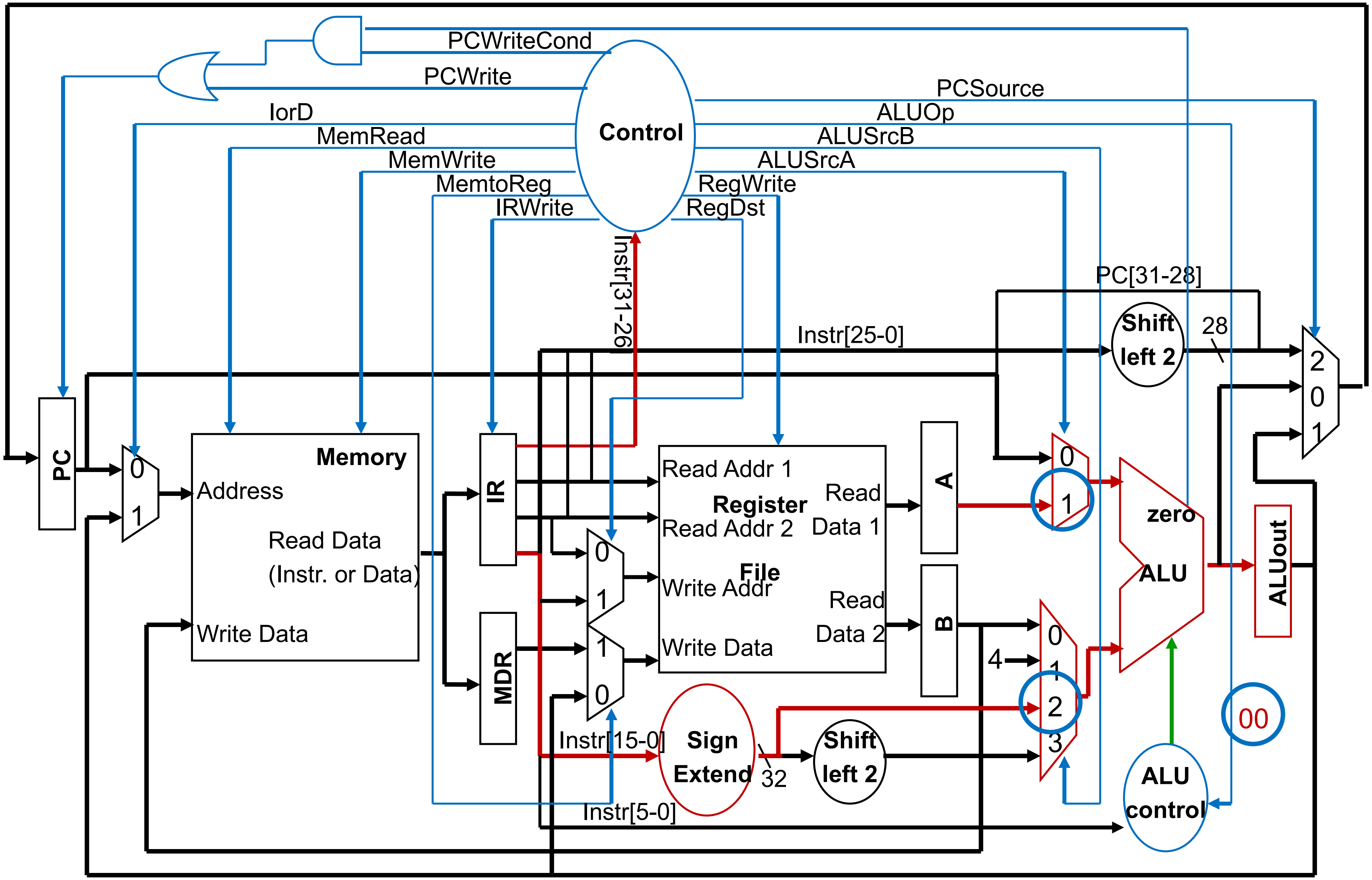
- Branch:

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

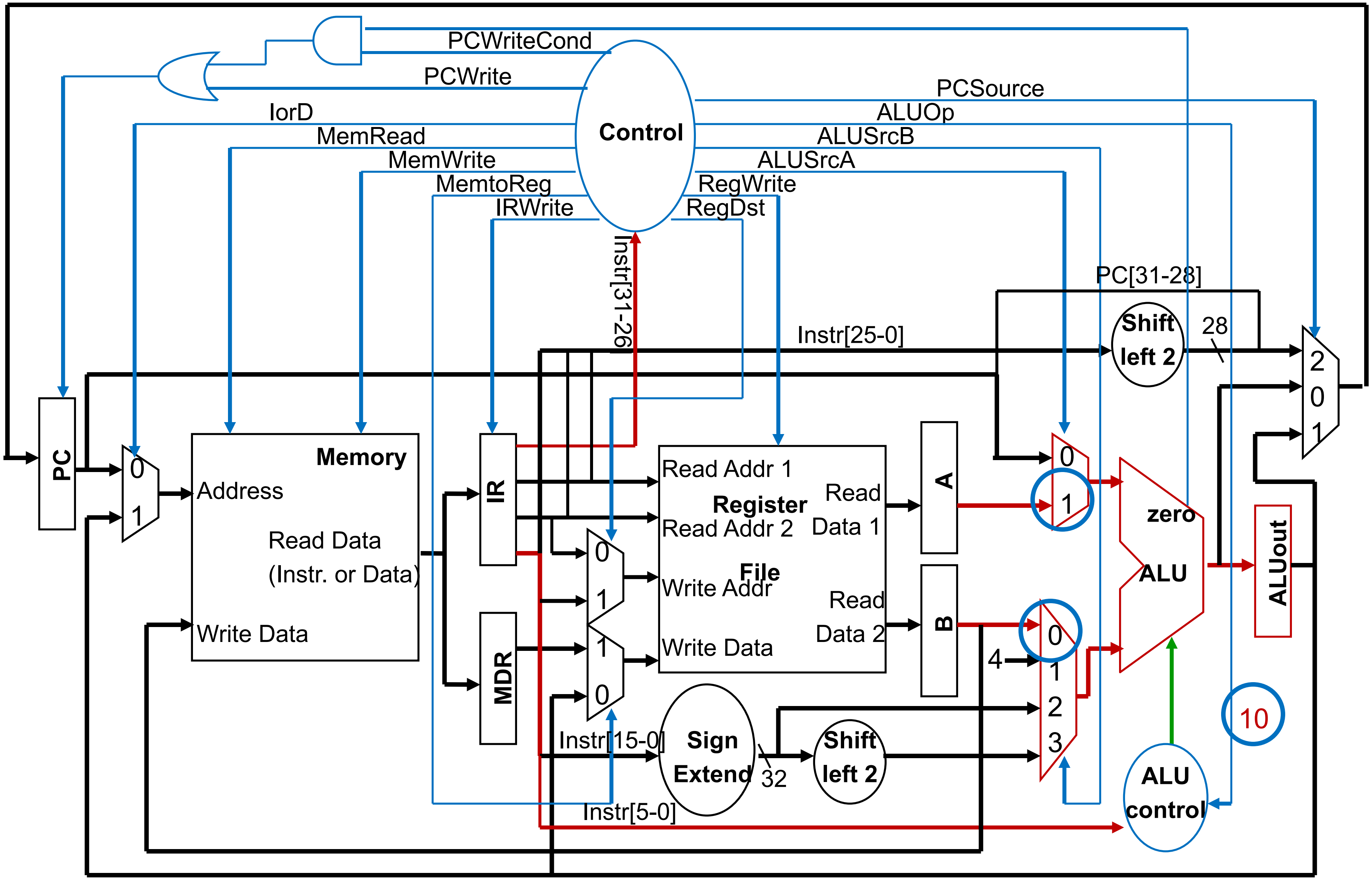
- Jump:

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

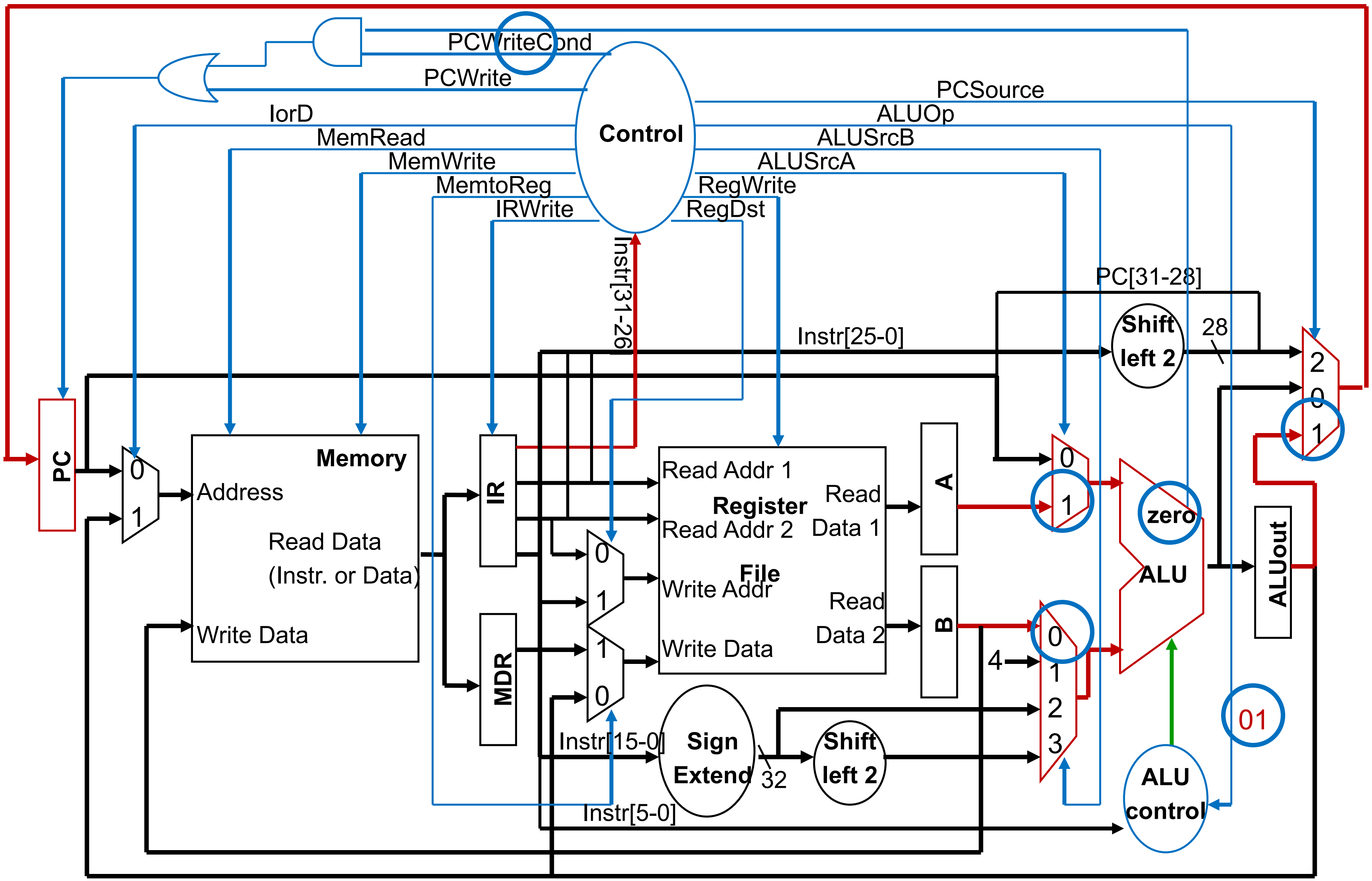
Datapath Activity During `lw` & `sw` Execute



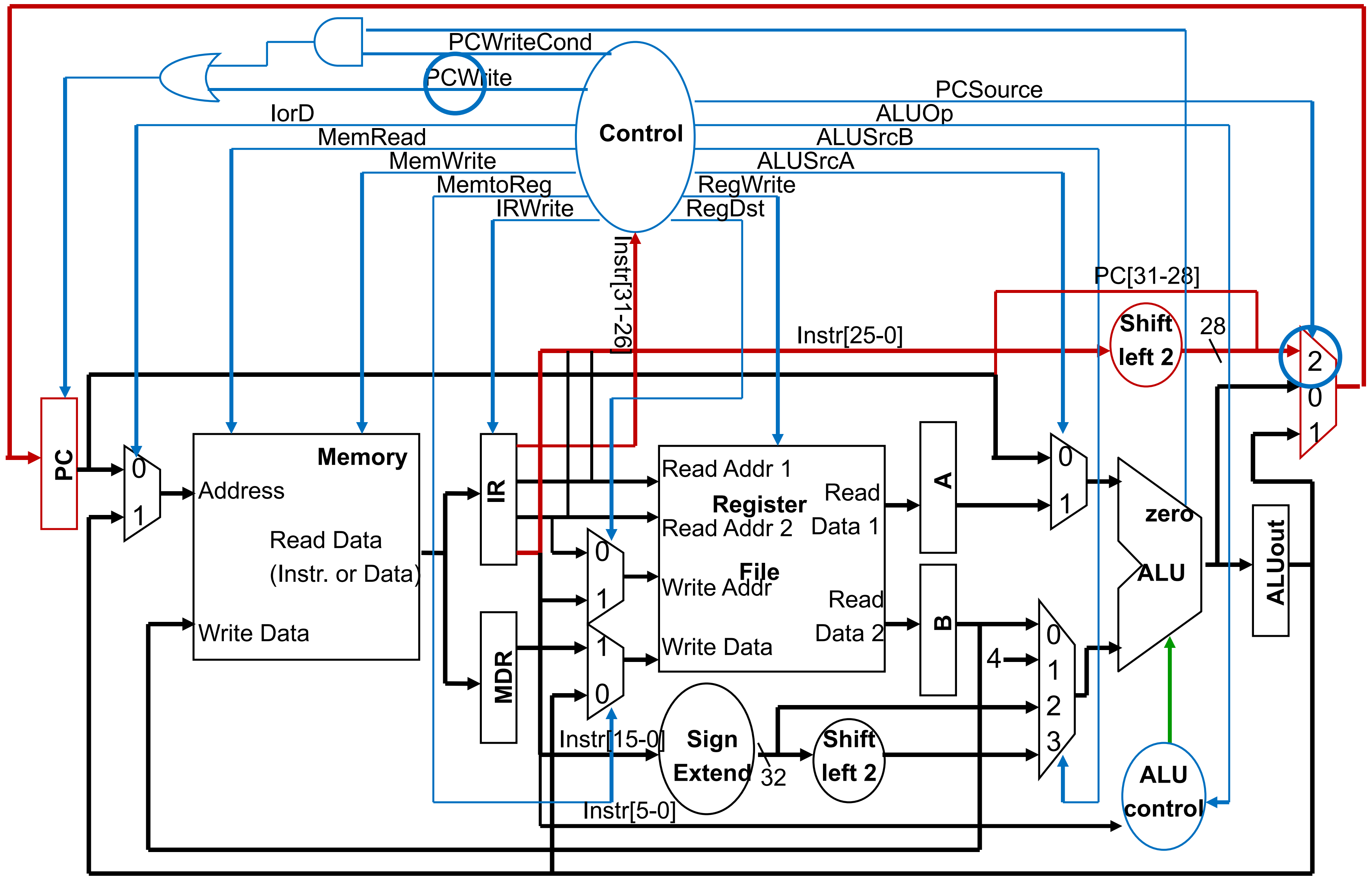
Datapath Activity During R-type Execute



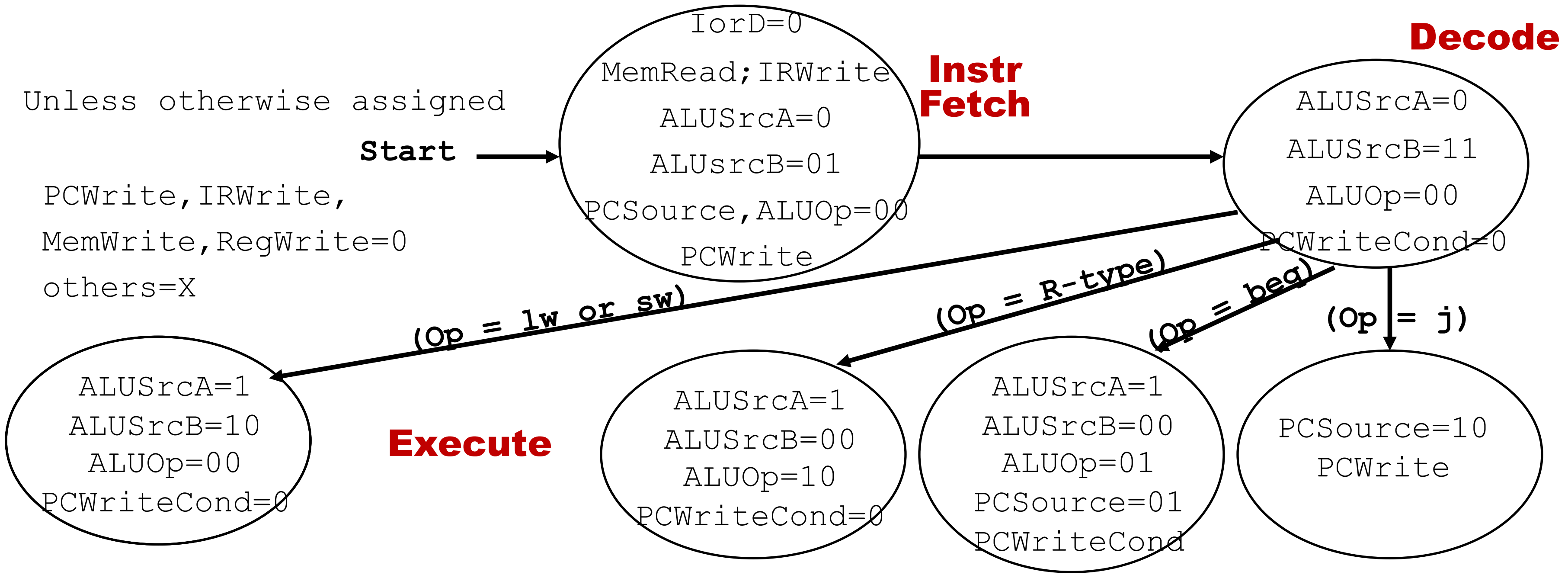
Datapath Activity During beq Execute



Datapath Activity During `j` Execute



Execute Control Signals Settings



Step 4 (also instruction dependent)

- Memory reference:

```
MDR = Memory[ALUOut];      -- lw
```

  or

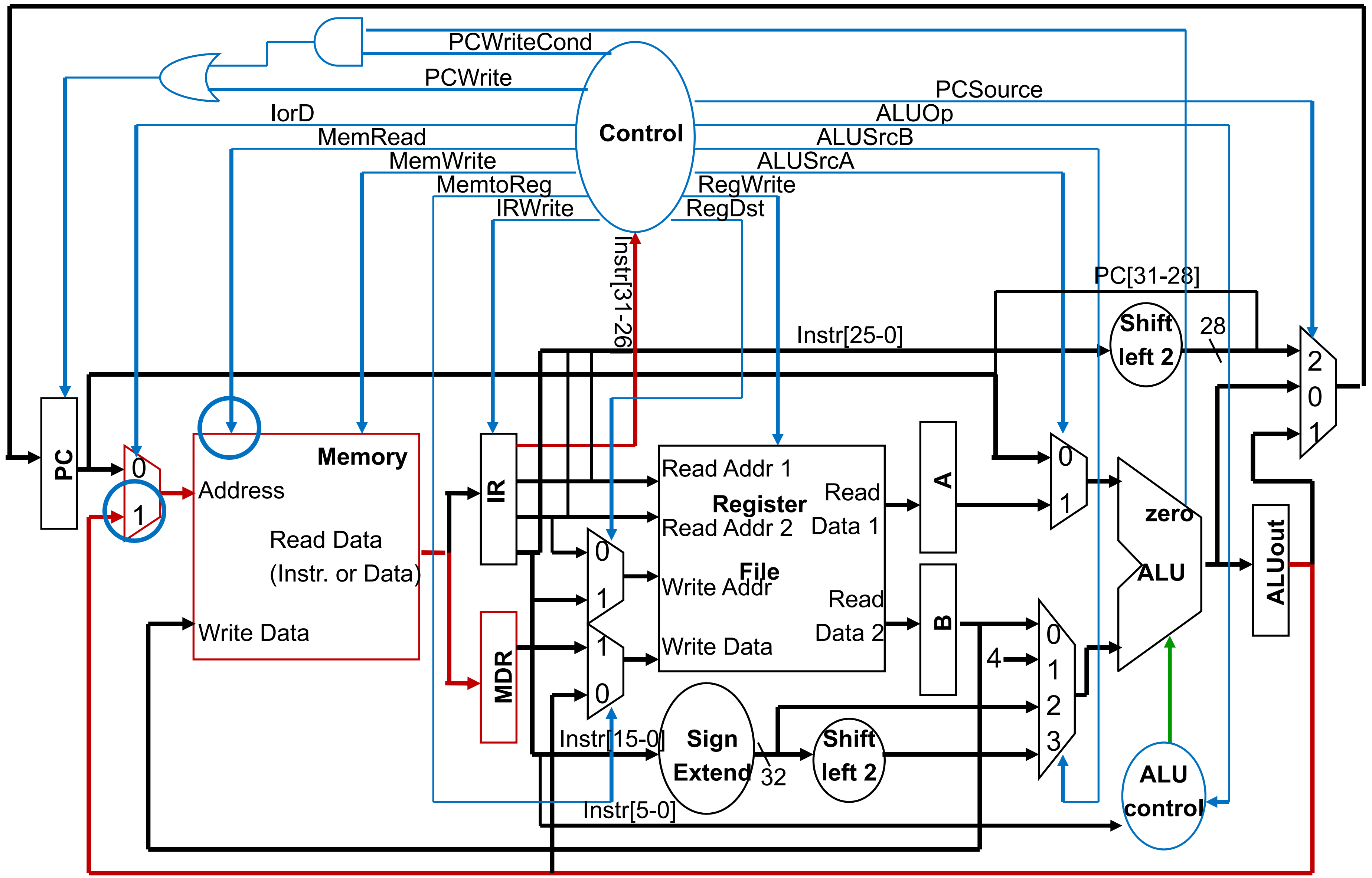
```
Memory[ALUOut] = B;        -- sw
```

- R-type instruction completion

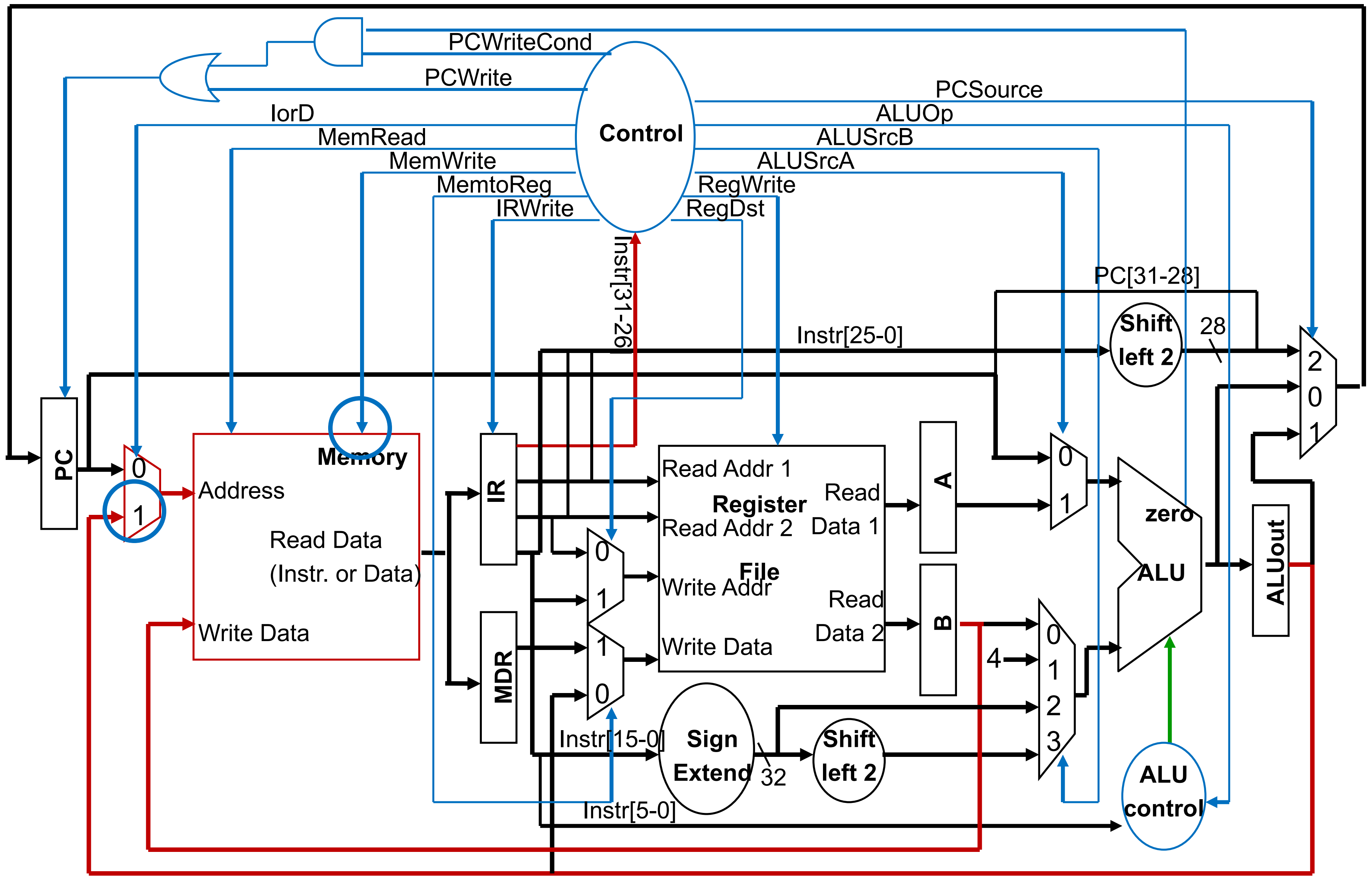
```
Reg[IR[15-11]] = ALUOut;
```

- Remember, the register write actually takes place at the end of the cycle on the clock edge

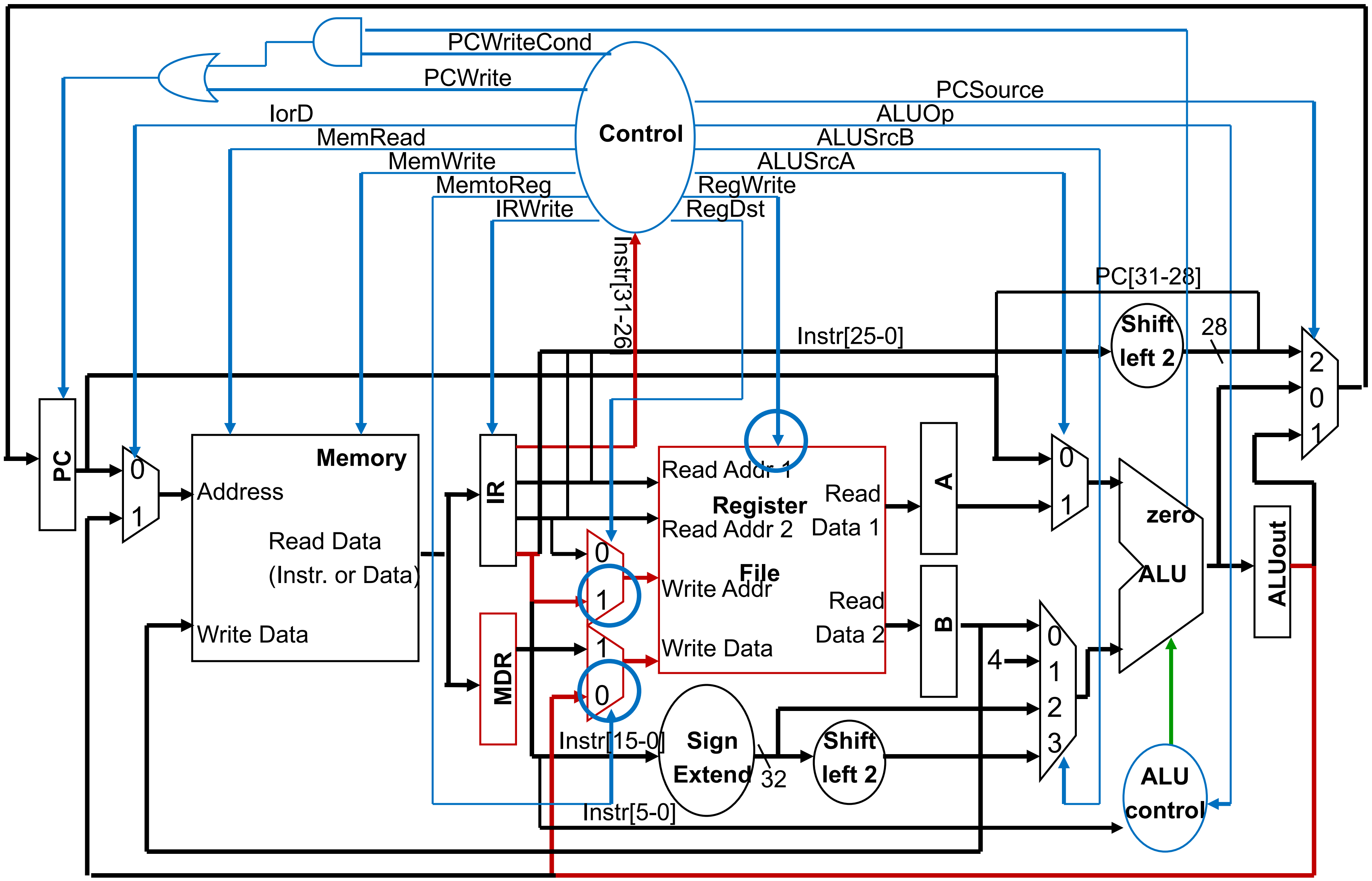
Datapath Activity During `lw` Memory Access



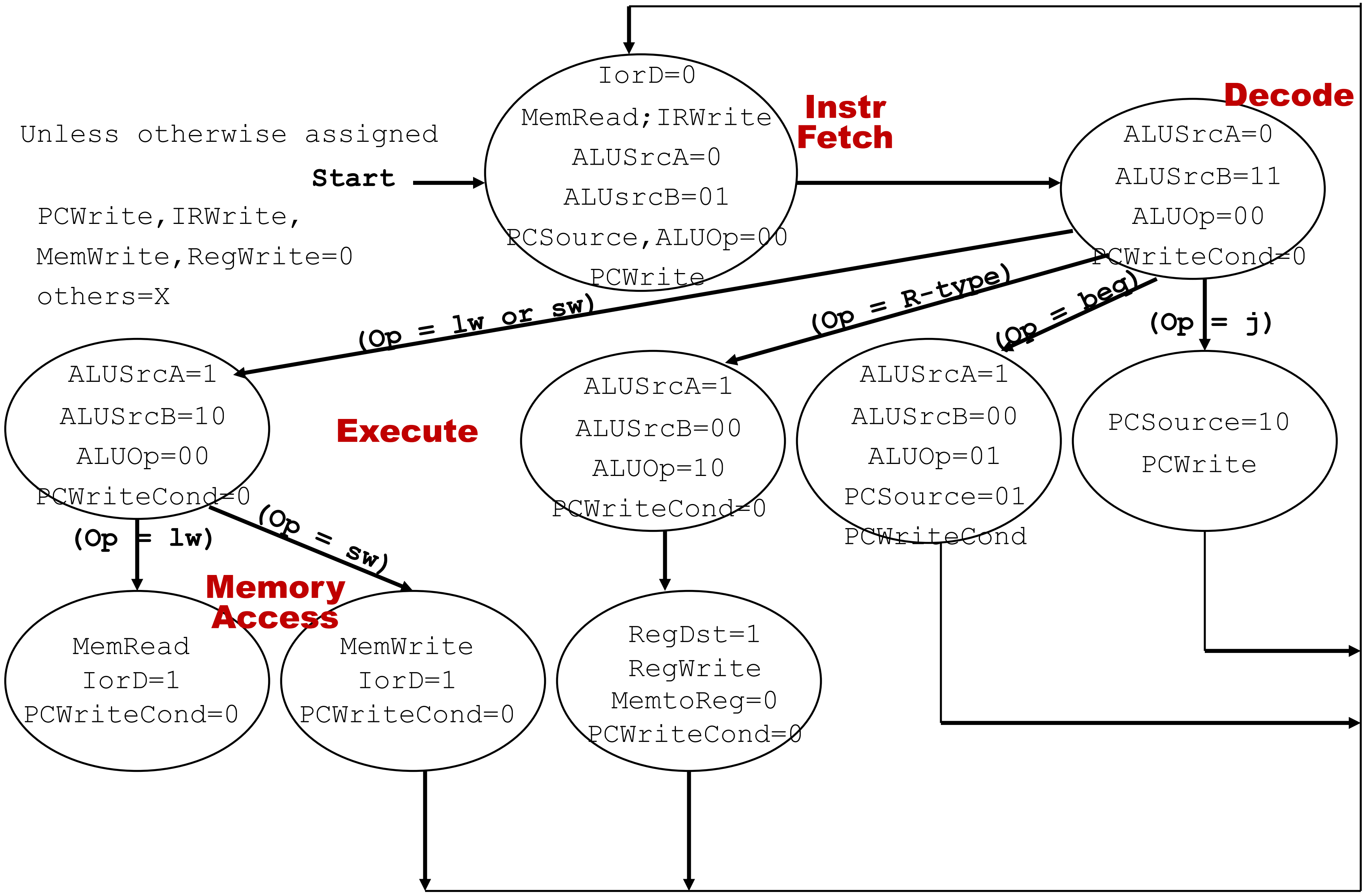
Datapath Activity During sw Memory Access



Datapath Activity During R-type Completion



Memory Access Control Signals Settings



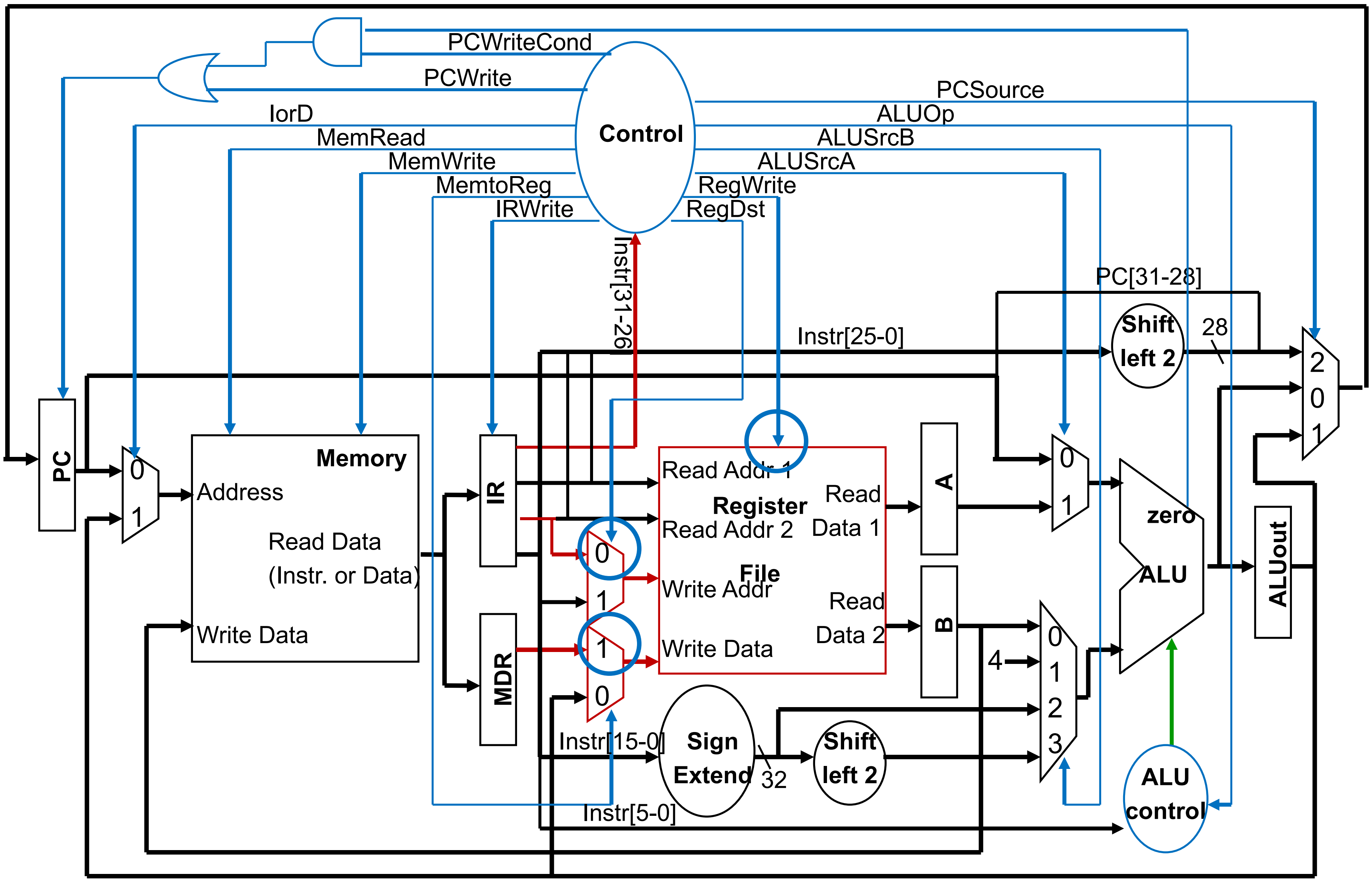
Step 5: Memory Read Completion (Write Back)

- All we have left is the write back into the register file the data just read from memory for lw instruction

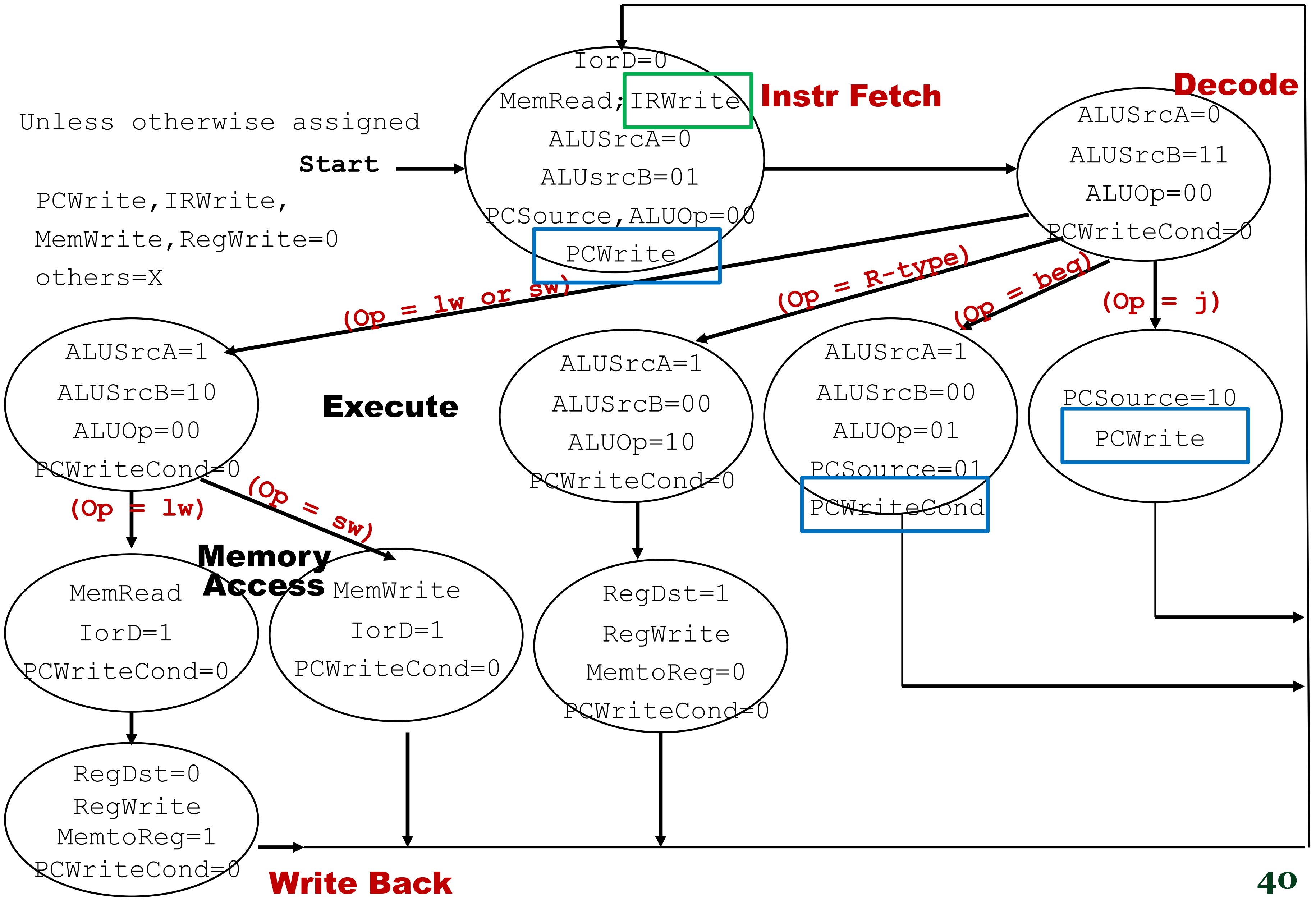
```
        Reg[IR[20-16]]= MDR;
```

What about all the other instructions?

Datapath Activity During `lw` Write Back



Write Back Control Signals Settings



RTL Summary

Step	R-type	Mem Ref	Branch	Jump
Instr fetch	IR = Memory[PC]; PC = PC + 4;			
Decode	A = Reg[IR[25-21]]; B = Reg[IR[20-16]]; ALUOut = PC +(sign-extend(IR[15-0])<< 2);			
Execute	ALUOut = A op B;	ALUOut = A + sign-extend (IR[15-0]);	if (A==B) PC = ALUOut;	PC = PC[31-28] ||(IR[25-0] << 2);
Memory access	Reg[IR[15-11]] = ALUOut;	MDR = Memory[ALUOut]; or Memory[ALUOut] = B;		
Write-back		Reg[IR[20-16]] = MDR;		

Example

Using the following instruction mix, what is the CPI, assuming that each state in the multicycle CPU requires 1 clock cycle?

Load	**25%**
store	10%
branches	11%
jumps	2%
ALU	52%