System I

The Processor: Multi-cycle Design

Haifeng Liu

Zhejiang University

How can I Make the Program Run Faster?

- Reduce the number of instructions
  - Make instructions that *do* more (CISC)
  - Use better compilers
- Use less cycles to perform the instruction
  - Simpler instructions (RISC) – under multi-cycle and related techniques
  - Use multiple units/ALUs/cores in parallel
- Increase the clock frequency
  - Find a *newer* technology to manufacture
  - Redesign time critical components
  - Adopt pipelining

Multi-Cycle CPU Design

- Single-Cycle microarchitecture
  - + Simple
  - – Clock cycle time limited by longest instruction (lw)
  - – Two adders/ALUs and two memories
- Multi-Cycle microarchitecture
  - + Shorter clock cycle period
  - + Simpler instructions run faster
  - + Reuse expensive hardware on multiple cycles
  - – Sequencing overhead paid many times

Key Difference between Single-Cycle and Multi-Cycle

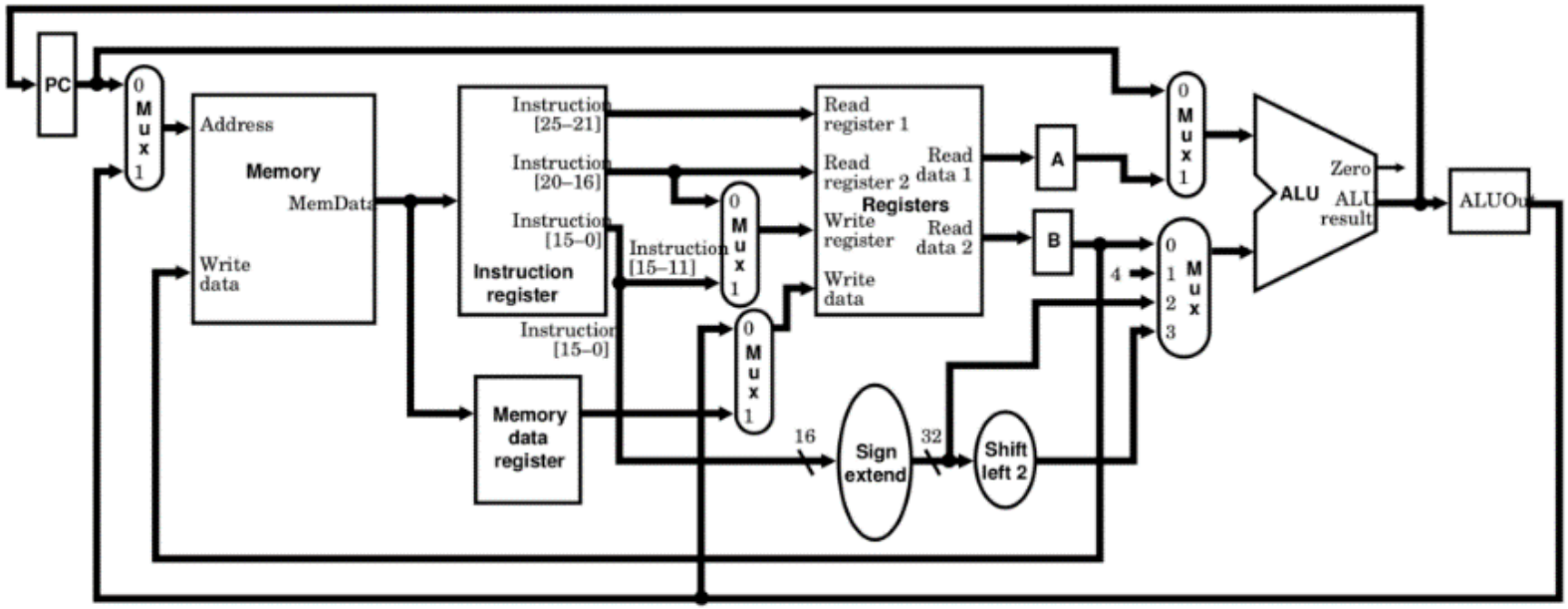
- Single-Cycle microarchitecture
  - One clock cycle for one instruction
- Multi-Cycle microarchitecture
  - One clock cycle for one stage!
- But same design steps: datapath & controller

Basic Principles of Multi-Cycle CPU Design

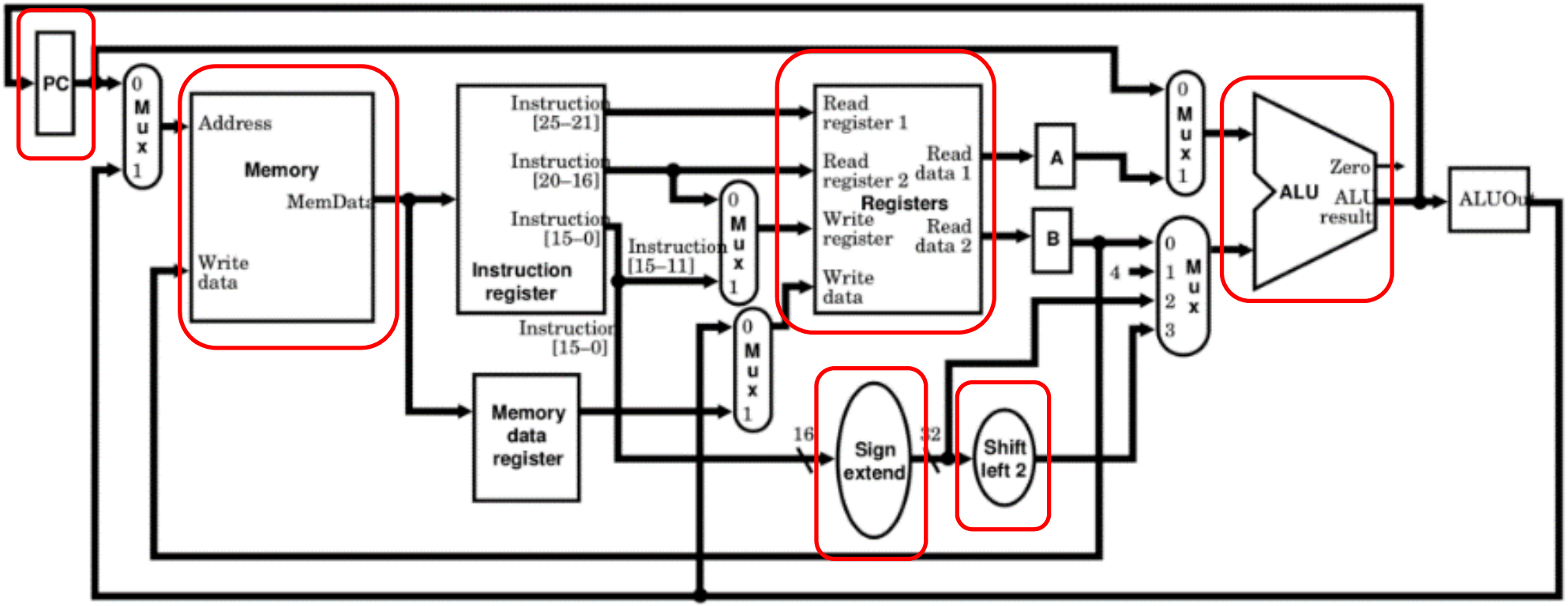
- Separate the execution of instructions into several stages with the same period
  - Hard to achieve the same period, thus aims to be *almost the same*
- Each stage occupies one clock cycle
- Each clock cycle at most completes a memory access, register access, or ALU operation
- Execution result of the previous clock cycle needs to be stored in specific sequential logic components
- Clock cycle period depends on the most complex operation

General Overview of Multi-Cycle Datapath

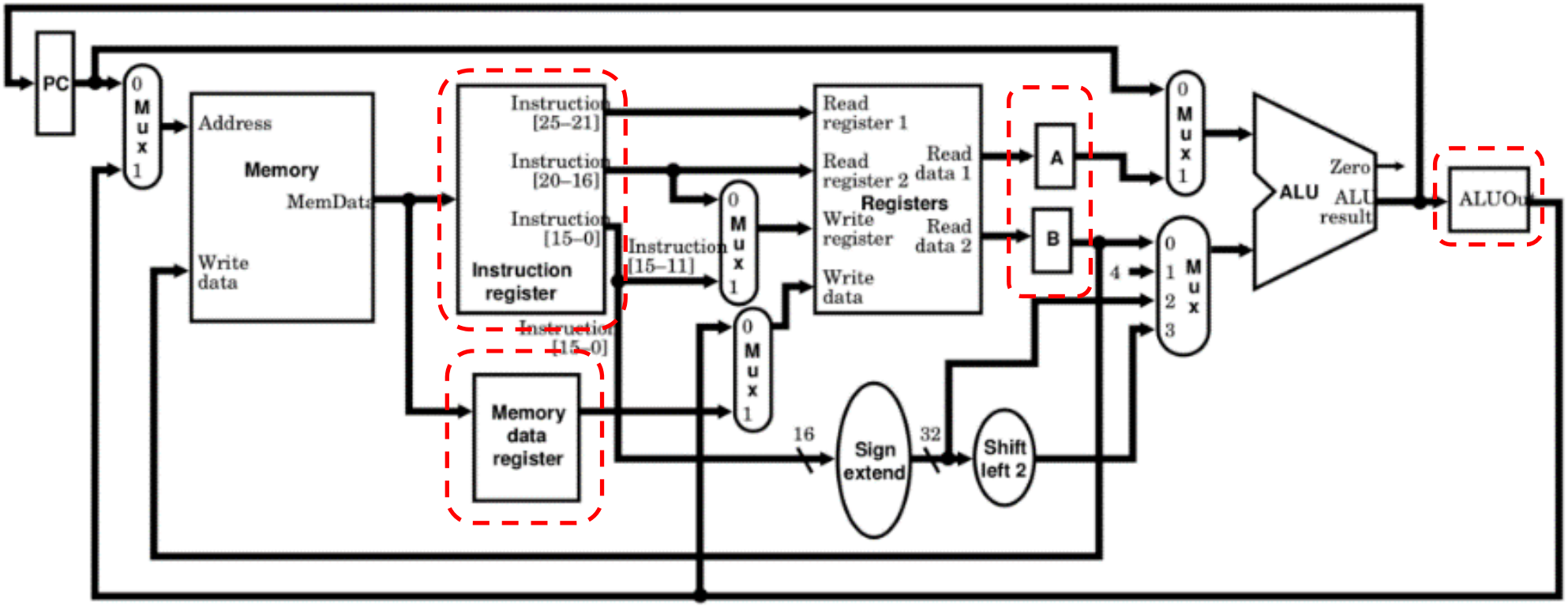
- Only one memory interacts with both instructions and data
  - Now feasible for Von Neumann architecture
- Only one ALU unit without additional adders
- But adding more registers



Same Components in the Datapath



New Components in the Datapath



New Components in the Datapath

New temporary registers

- Instruction register (IR)
  - Holds the instruction after its been pulled from memory
- Memory data register (MDR)
  - Temporarily holds data grabbed from memory until the next cycle
- A
  - Temporarily holds the contents of read register 1 until the next cycle
- B
  - Temporarily holds the contents of read register 2 until the next cycle
- ALUout
  - Temporarily holds the contents of the ALU until the next cycle

Stages for Instruction Execution

- **Fetch** :
  - Take instructions from the instruction memory
  - Modify PC to point the next instruction
- **Instruction decoding & Read Operand**:
  - Will be translated into machine control command
  - Reading Register Operands, whether or not to use
- **Executive Control**:
  - Control the implementation of the corresponding ALU operation
- **Memory access**:
  - Write or Read data from memory
  - Only ld/sd
- **Write results to register**:
  - If it is R-type instructions, ALU results are written to rd
  - If it is I-type instructions, memory data are written to rd
- **Modify PC** for branch instructions

New Components in the Datapath

New temporary registers

- Instruction register (IR)
  - Holds the instruction after its been pulled from memory
- Memory data register (MDR)

All the temporary registers are used to separate the instruction execution into stages physically!

- B
  - Temporarily holds the contents of read register 2 until the next cycle
- ALUout
  - Temporarily holds the contents of the ALU until the next cycle

How does the Datapath Work?

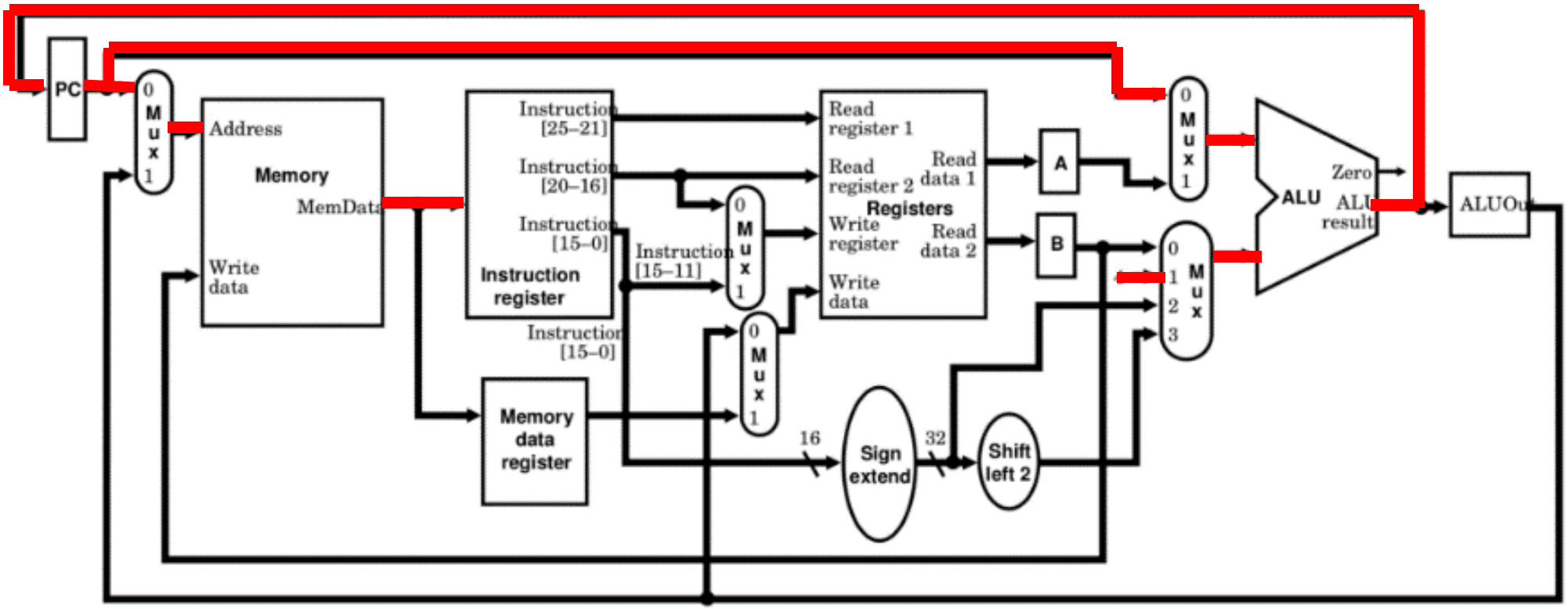
- Each instruction may take 3-5 of the steps to complete
- Let's look into the execution in terms of the steps

- Instruction Fetch Operations

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

  - 1. Send contents of PC to the memory as the address
  - 2. Read instruction from memory
  - 3. Write instruction into IR for use in the next cycle
  - 4. Increase PC by 4

How does the Datapath Work? - IF



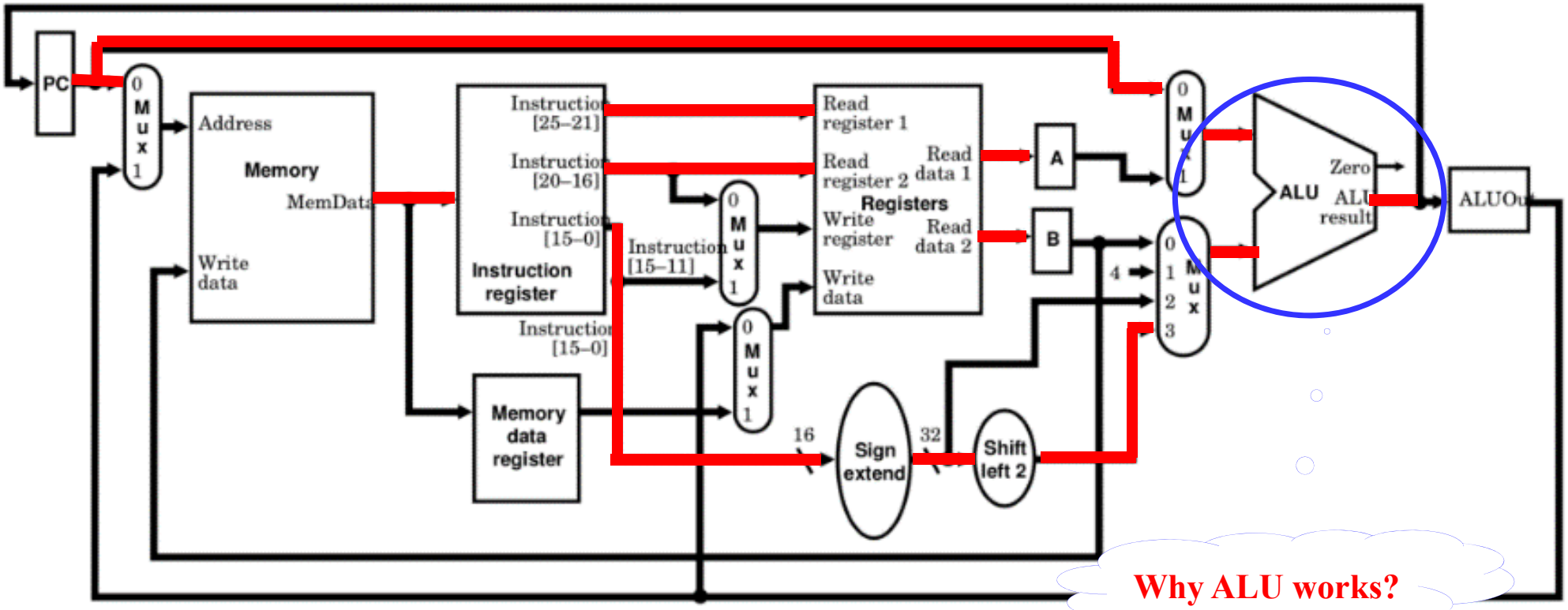
How does the Datapath Work? - ID

- Instruction Decode and Register Fetch Operations

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

  - 1. Decode instruction
  - 2. Optimistically read registers
  - 3. Optimistically compute branch target

How does the Datapath Work? - ID



How does the Datapath Work? - EX

- Execution Operations
  - Instructions diverge
- Memory-related

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

- R-type

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

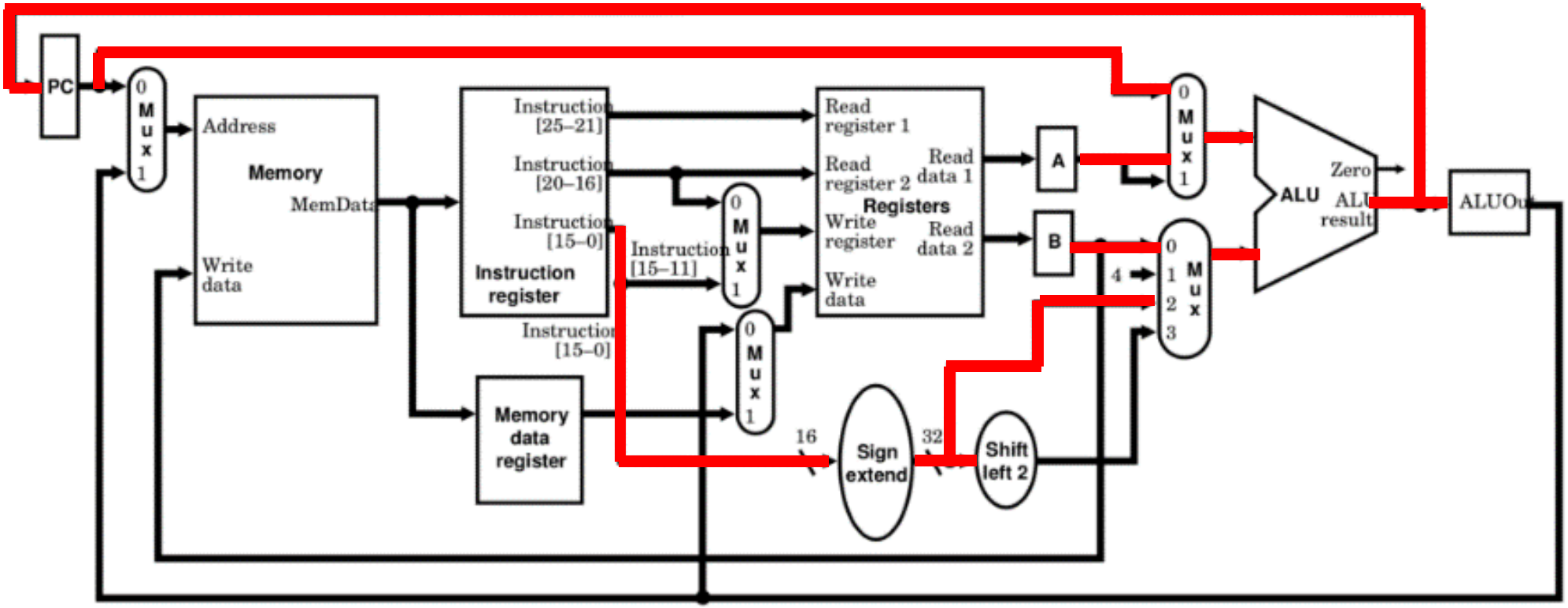
- Branch

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

- Jump

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

How does the Datapath Work? - EX



How does the Datapath Work? - MEM

- Execution Operations
  - Instructions diverge
- Memory-related

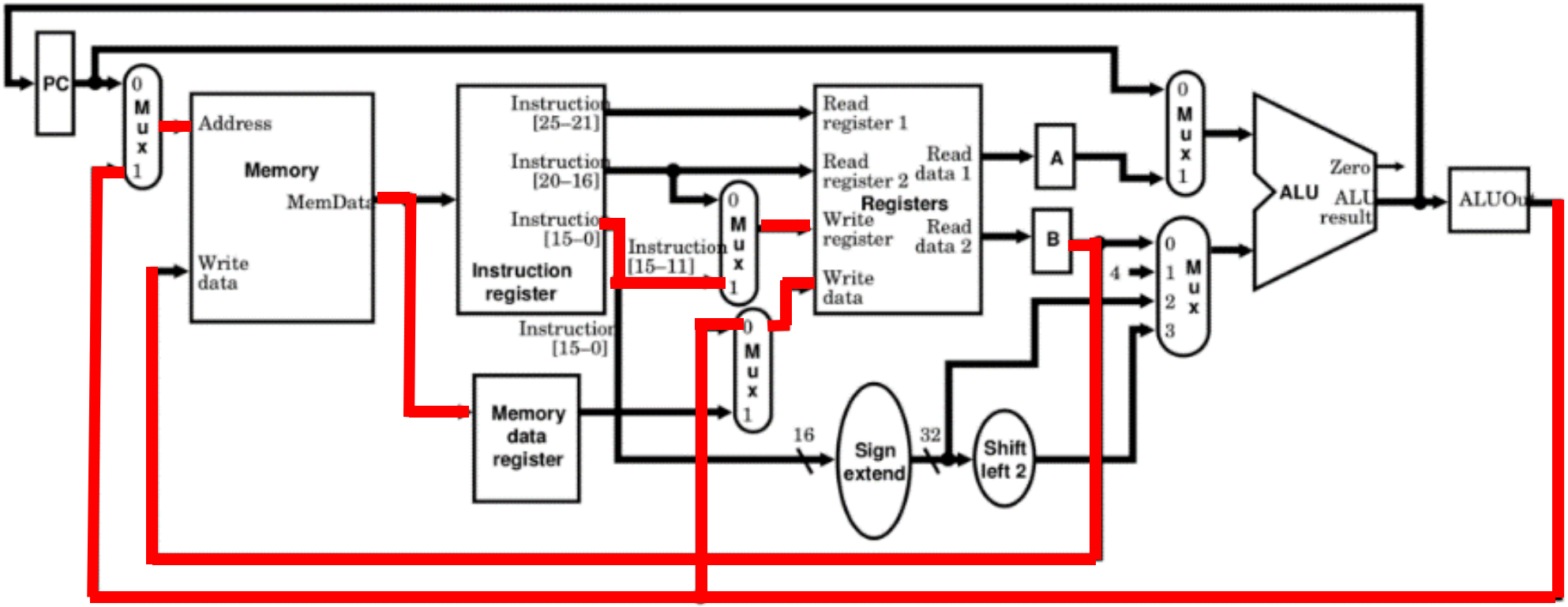
Load: `MDR = Memory[ALUOut];`

Store: `Memory[ALUOut] = B;`

- R-type

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

How does the Datapath Work? - MEM

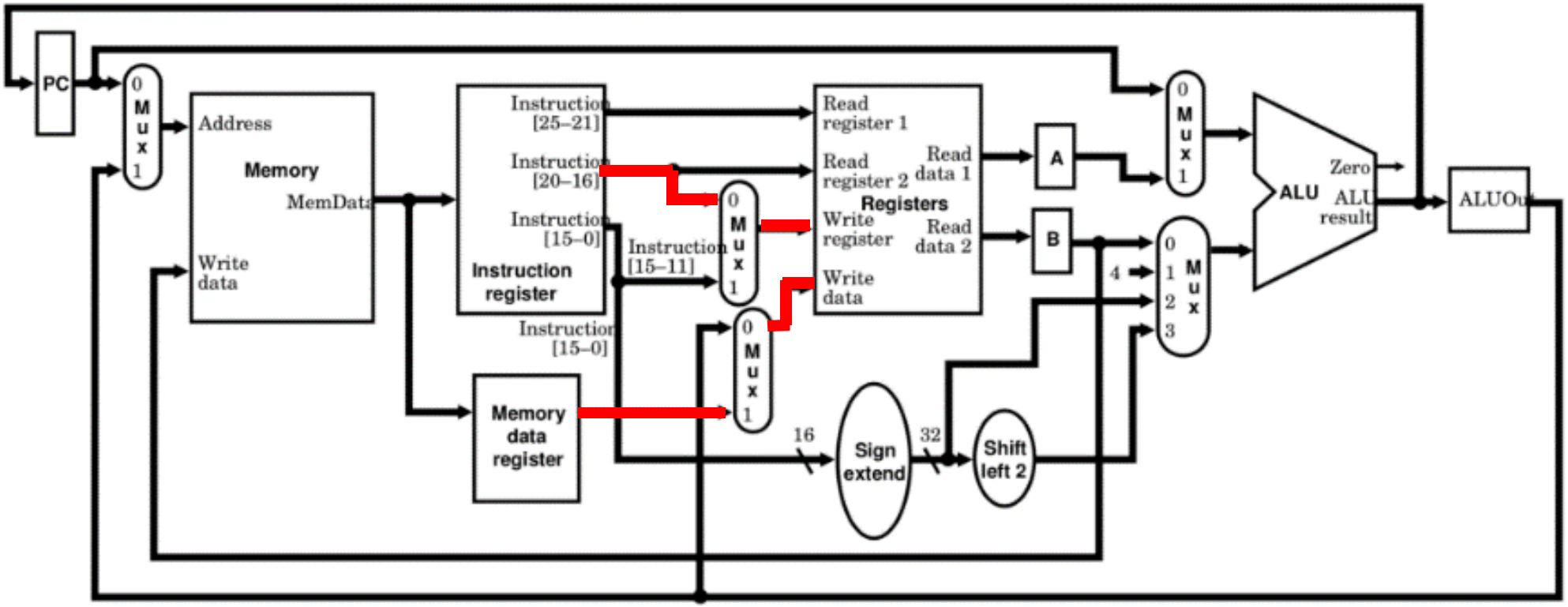


How does the Datapath Work? - WB

- Load operation

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

How does the Datapath Work? – WB



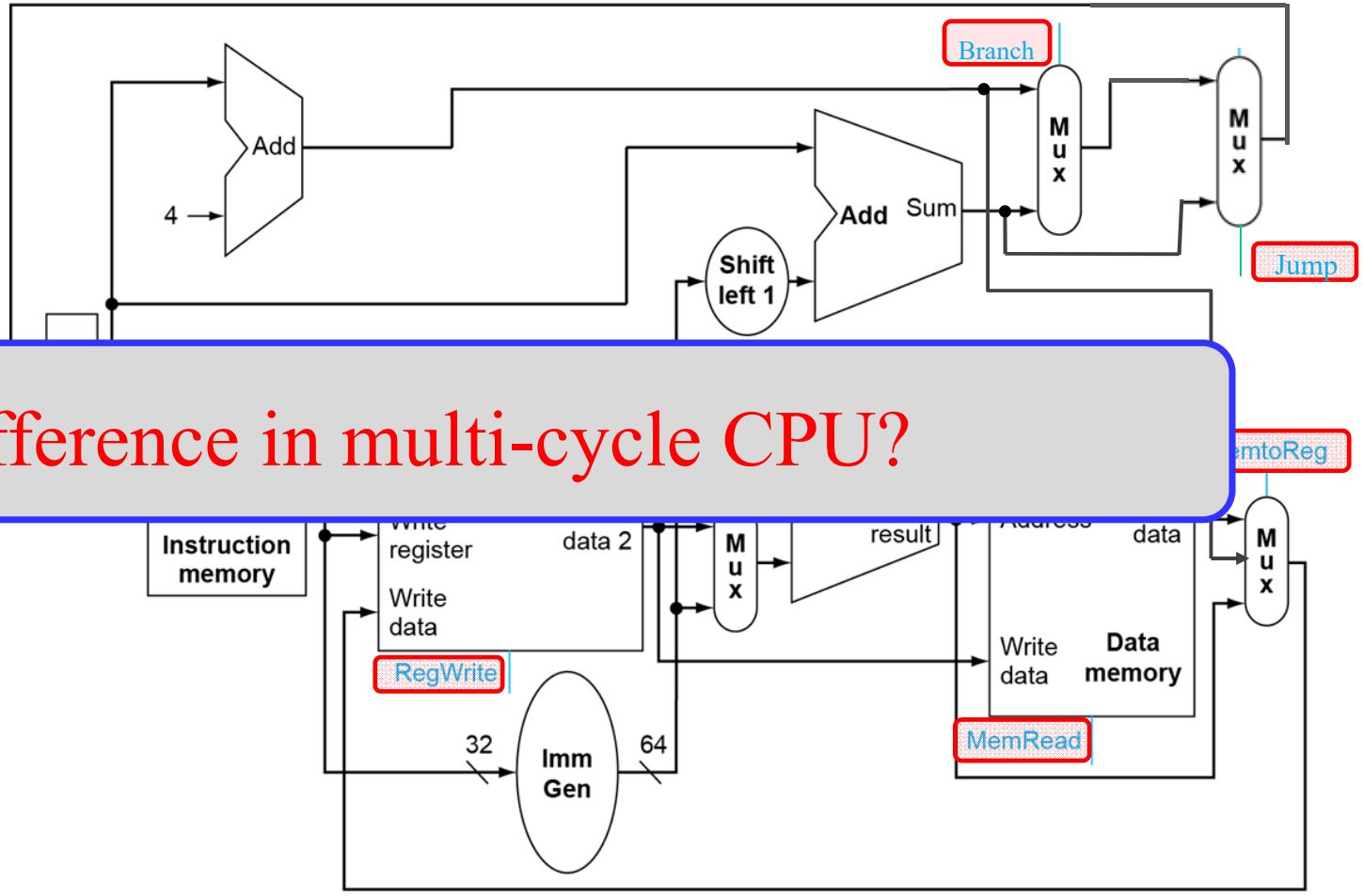
Controller Design

- Now we know how the datapath works after the new components involved
  - Similar datapath with single-cycle CPU
  - But temporary registers are required to store intermediates in each stage

How about the controller?

Revisit Controller Design Route

- Analyses for cause and effect
  - Information comes from the 32 bits of the instruction
  - Selecting the operations to perform by sequential
  - instruction (multiplexor inputs)
  - Controlling the flow of data (multiplexor inputs)
  - ALU's operation based on instruction type and function code



Controller Design

- Of course, yes!
  - In single-cycle CPU, the value of each control signal is fixed for each clock cycle
  - But in multi-cycle CPU, they are different for different clock cycle
  - Example: let's look at the execution of
    - *add x10, x12, x14*

  whose machine code is 0000000 01110 01100 000 01010 0110011

  i.e., the opcode = 0110011

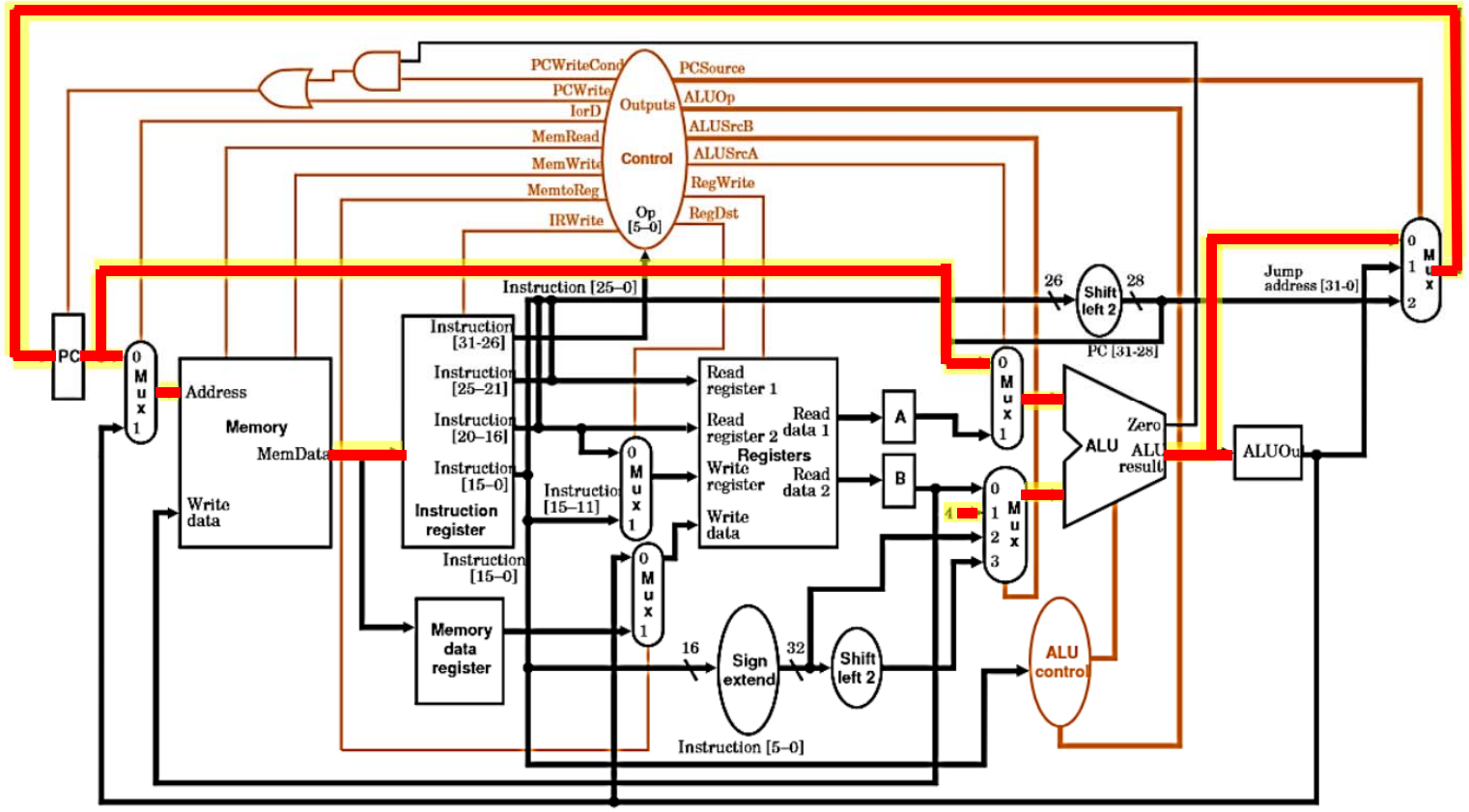
  the funct3 = 000 & funct7 = 0000000

  rd = 01010

  rs1 = 01100 & rs2 = 01110

Controller Design

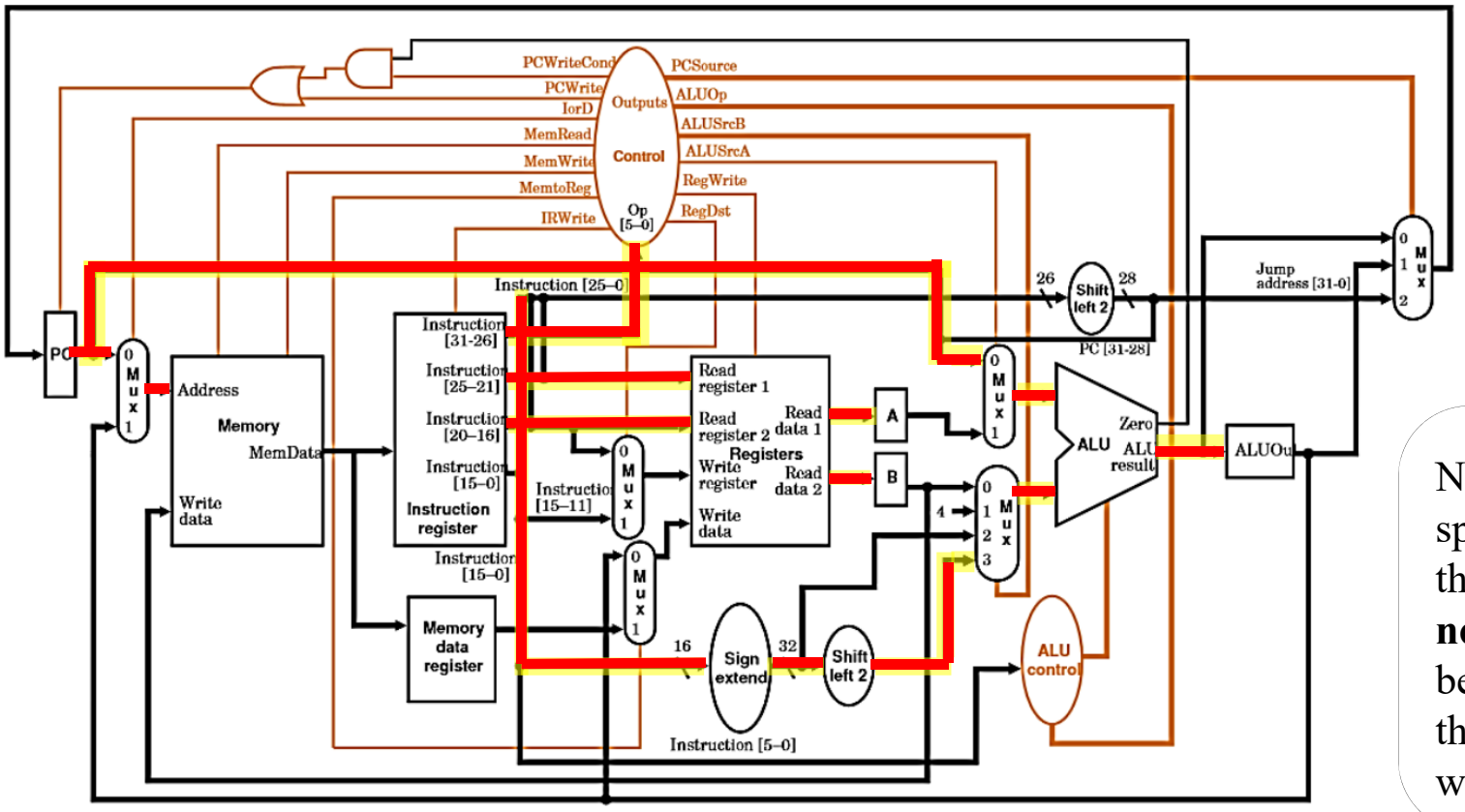
- Cycle 1



Signal	Value
PCWrite	1
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
PCSource	00
ALUOp	00
ALUSrcB	01
ALUSrcA	0
RegWrite	0

Controller Design

- Cycle 2

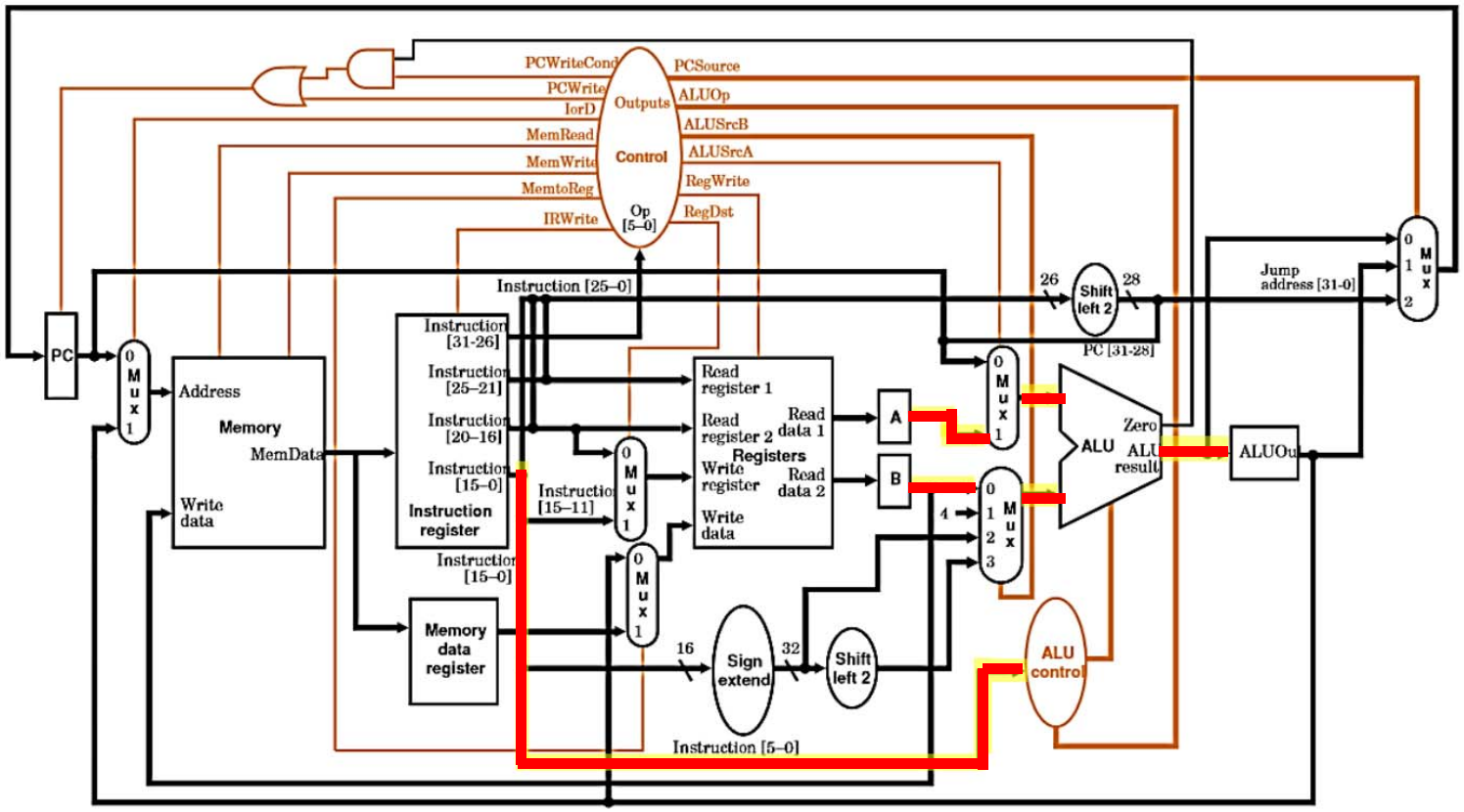


Signal	Value
ALUOp	00
ALUSrcB	11
ALUSrcA	0

Note that we compute the speculative branching target in this step **even though we will not need it**. We have **nothing** better to do while we decode the instruction so we might as well.

Controller Design

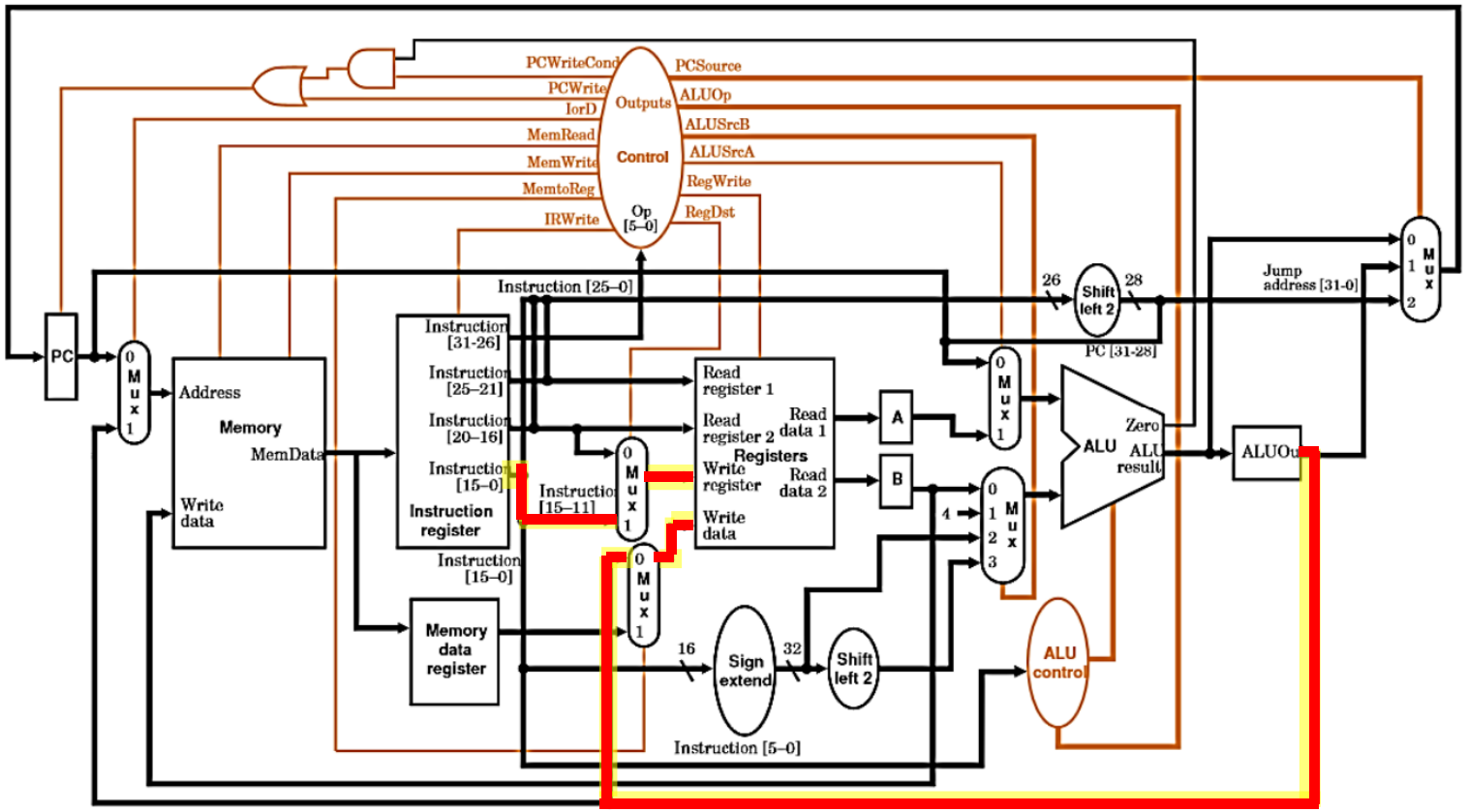
- Cycle 3



Signal	Value
ALUOp	10
ALUSrcB	00
ALUSrcA	1

Controller Design

- Cycle 4



Signal	Value
MemtoReg	0
RegWrite	1
RegDst	1

What we can find?

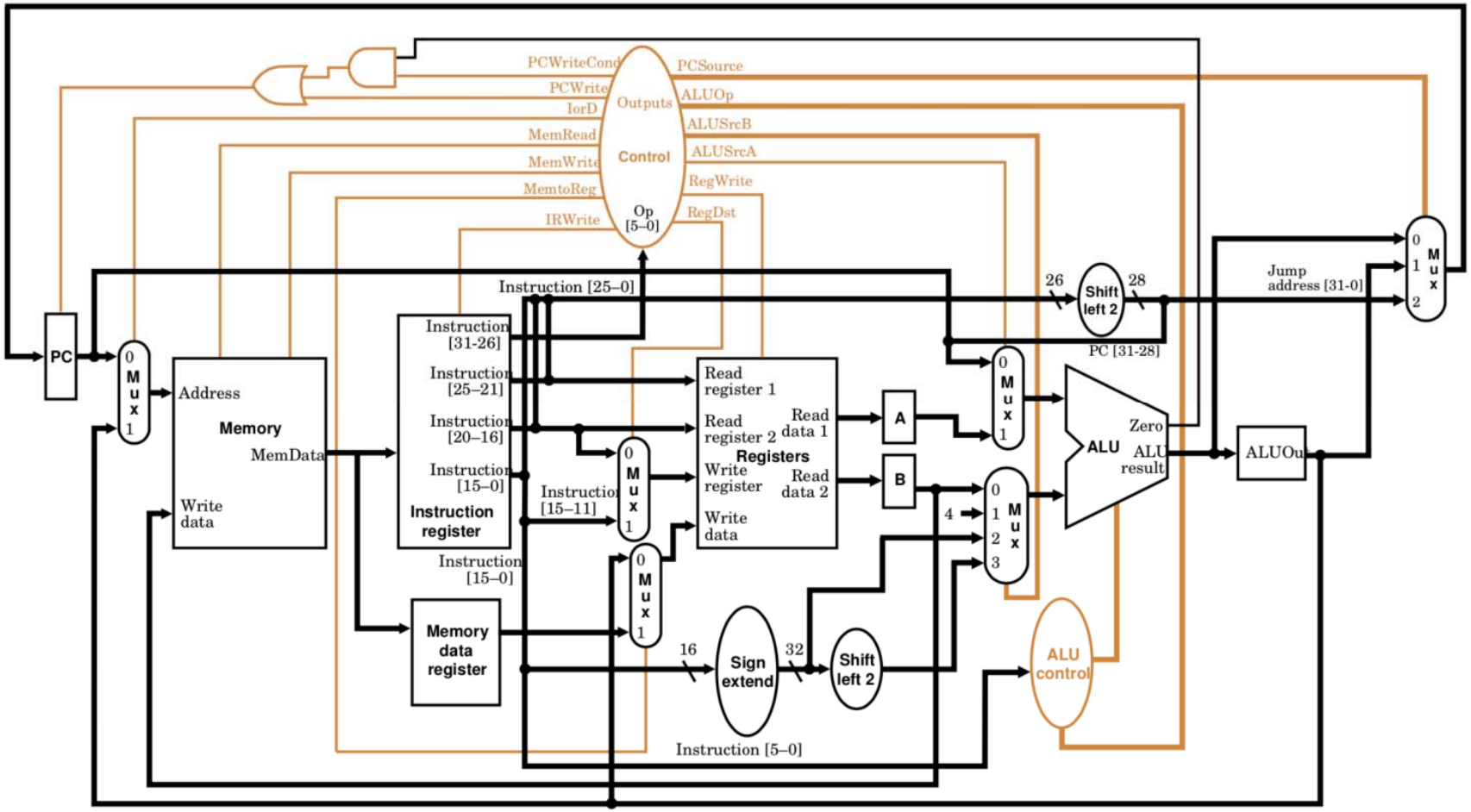
- In Cycle 2
  - Note that we compute the speculative branching target in this step even though we will not need it. We have nothing better to do while we decode the instruction so we might as well
- Cycle 2 vs. 3
  - In different cycles, even the same signal has different values!

Signal	Value
ALUOp	00
ALUSrcB	11
ALUSrcA	0

Signal	Value
ALUOp	10
ALUSrcB	00
ALUSrcA	1

We cannot directly apply controller design method in single-cycle CPU for multi-cycle one!

Controller Design



Control Signals

1-Bit Signal	Effect When Deasserted	Effect When Asserted
RegDst	The register file destination number for the Write register comes from rt	The register file destination number for the Write register comes from rd
RegWrite	None	Write register is written with the value of the Write data input
ALUSrcA	The first ALU operand is PC	The first ALU operand is A register
MemRead	None	Content of memory at the location specified by the Address input is put on the Memory data output
MemWrite	None	Memory contents of the location specified by the Address input is replaced by the value on the Write data input

Control Signals

1-Bit Signal	Effect When Deasserted	Effect When Asserted
MemToReg	The value fed to the register file input is ALUout	The value fed to the register file input comes from Memory data register
IorD	The PC supplies the Address to the Memory element	ALUOut is used to supply the address to the memory unit
IRWrite	None	The output of the memory is written into the Instruction Register (IR)
PCWrite	None	The PC is written; the source is controlled by PC-Source
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active

Control Signals

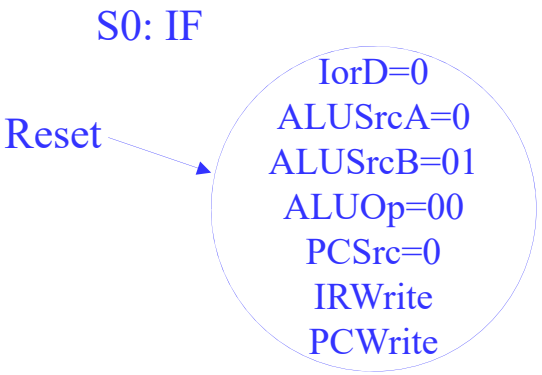
2-Bit Signal	Value	Effect
ALUOp	00	The ALU performs an add operation
	01	The ALU performs a subtract operation
	10	The funct field of the instruction determines the operation
ALUSrcB	00	The second input to ALU comes from the B register
	01	The second input to ALU is 4
	10	The second input to the ALU is the sign-extended, lower 16 bits of the Instruction Register (IR)
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left by 2 bits
PCSrc	00	Output of the ALU (PC+4) is sent to the PC for writing
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing
	10	The jump target address (IR[25-0] shifted left 2 bits and concatenated with PC + 4[31-28]) is sent to the PC for writing

How to Implement the Controller?

- Hardwired Controller
  - Based on Finite State Machine (FSM)
  - Implement by PLA circuit and state register
  - Usually design for RISC
- Micro-program Controller
  - Implement by control storage with micro-instructions
  - Usually design for CISC

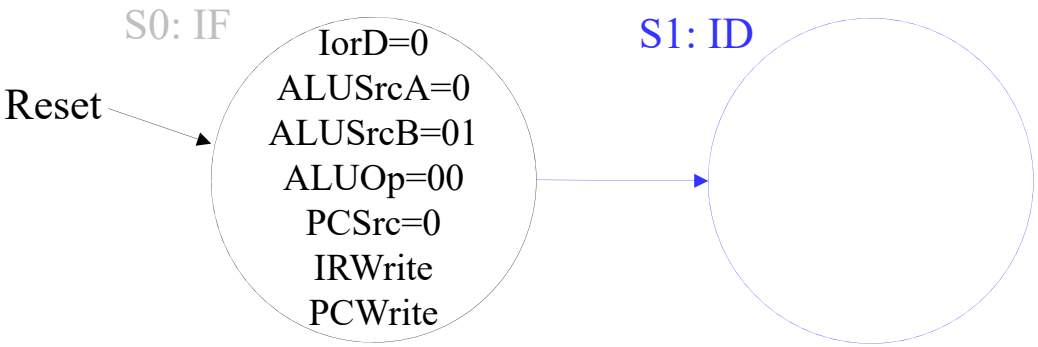
Hardwired Controller

- Based on FSM
  - Let's plot the FSM following the stages of instruction execution

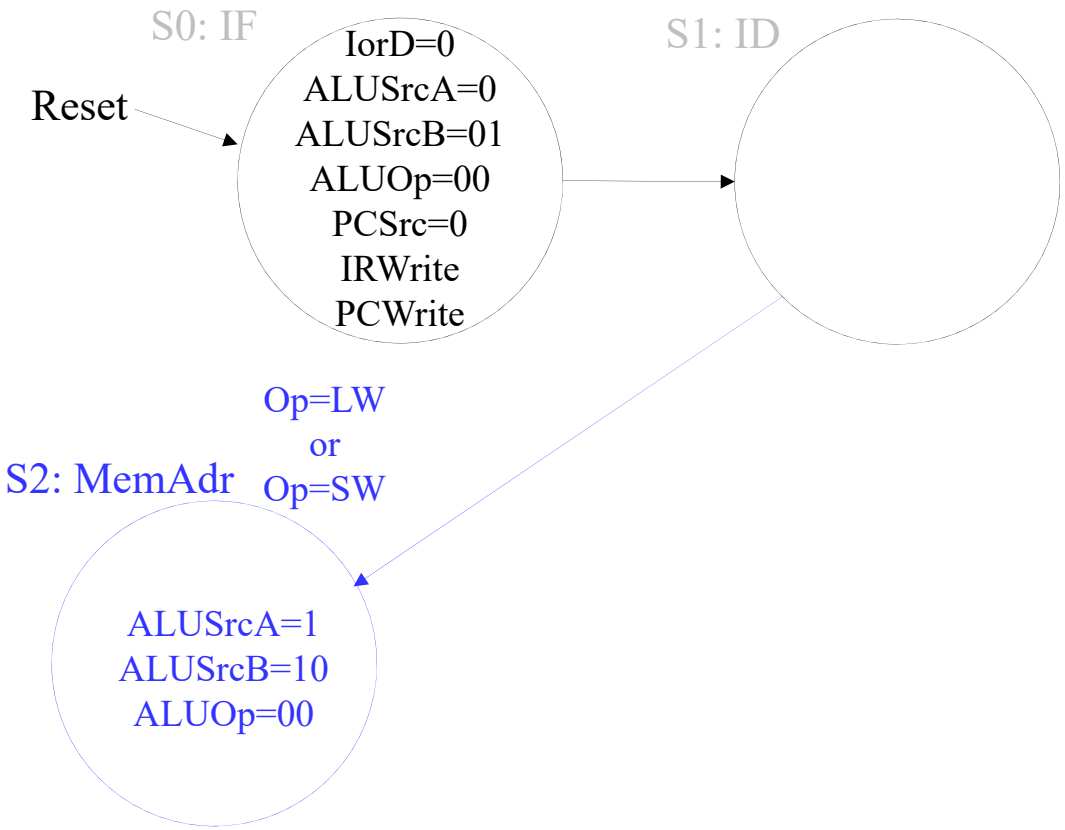


Hardwired Controller

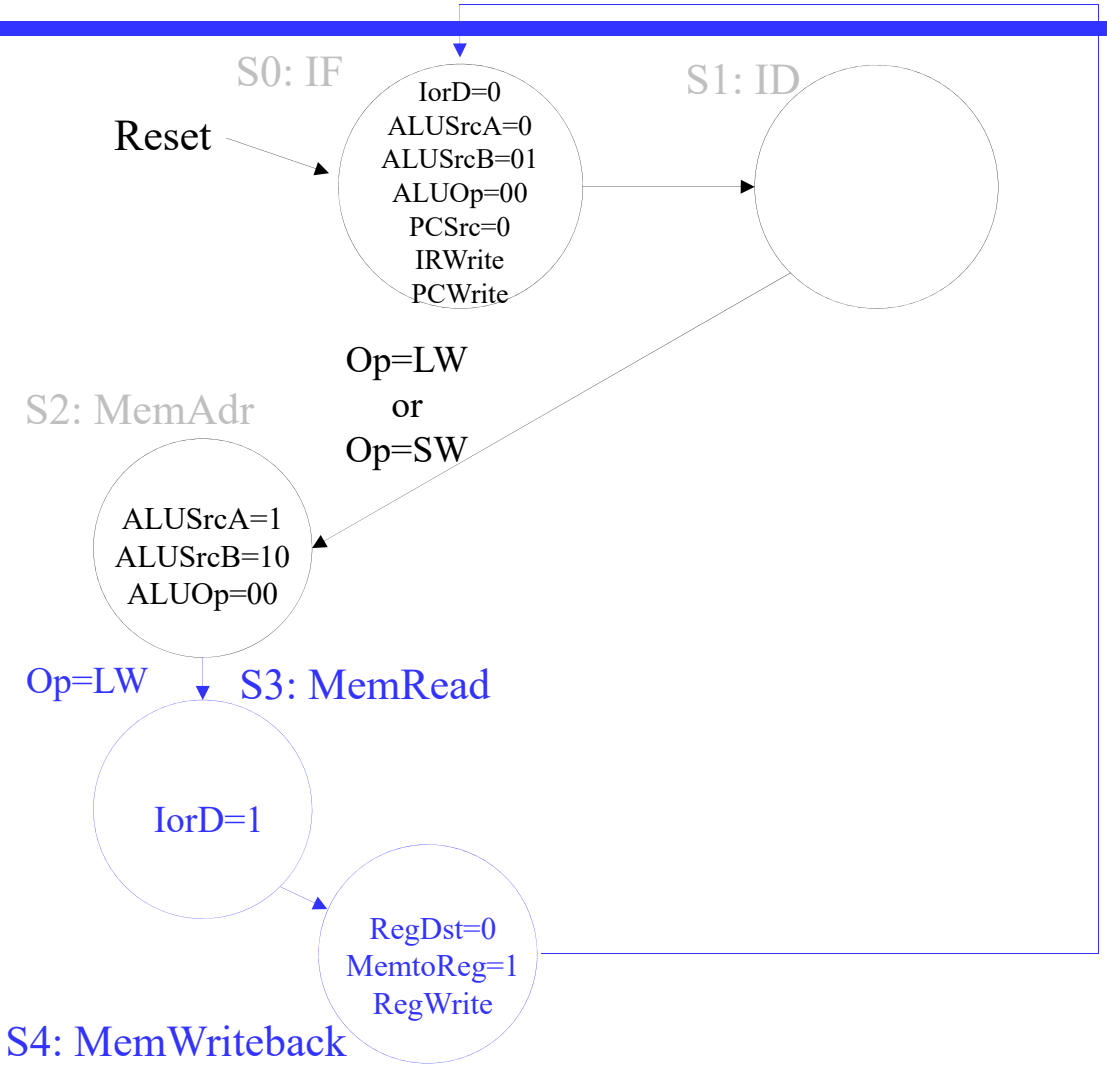
- Based on FSM
  - Let's plot the FSM following the stages of instruction execution



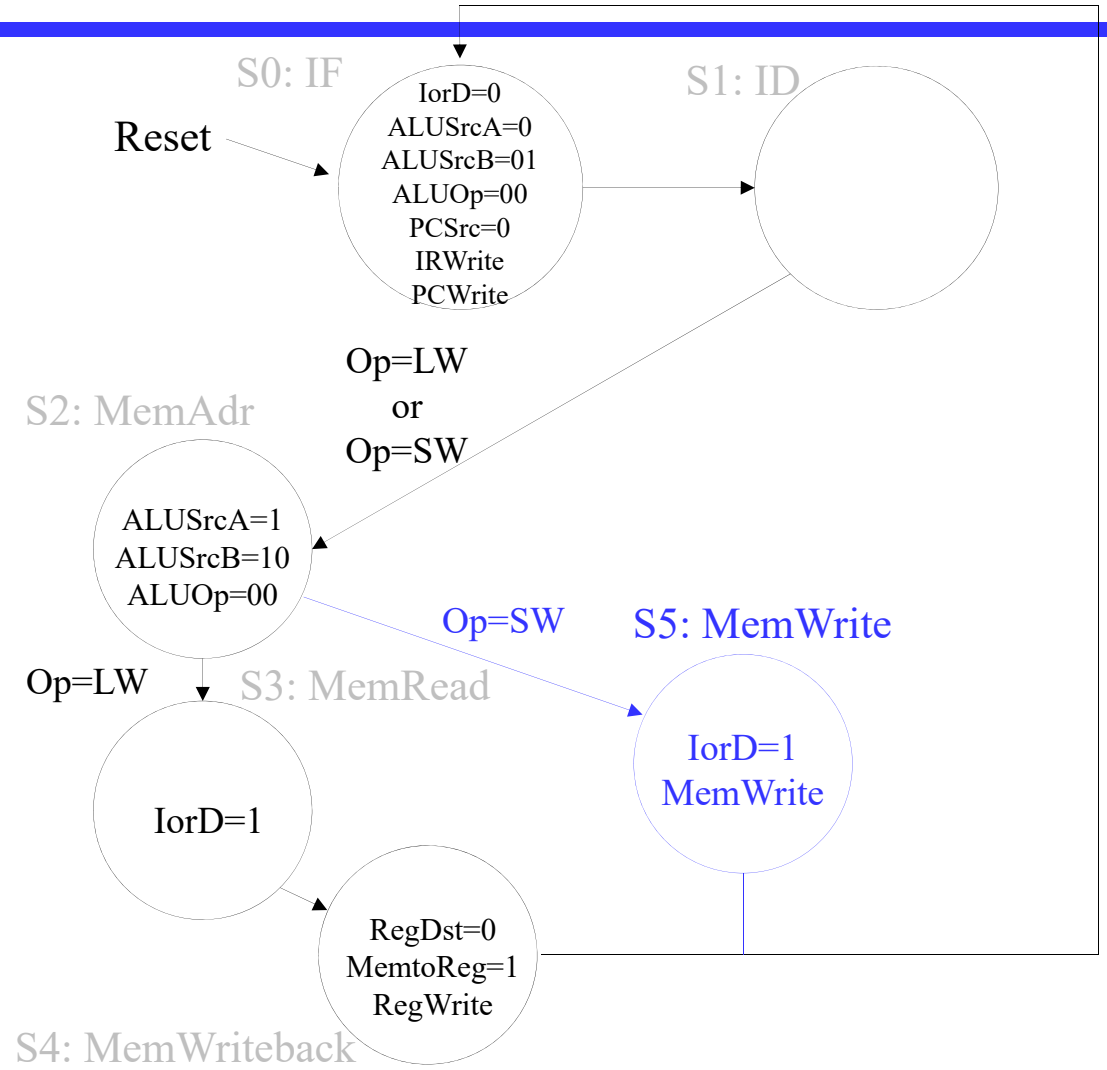
Hardwired Controller



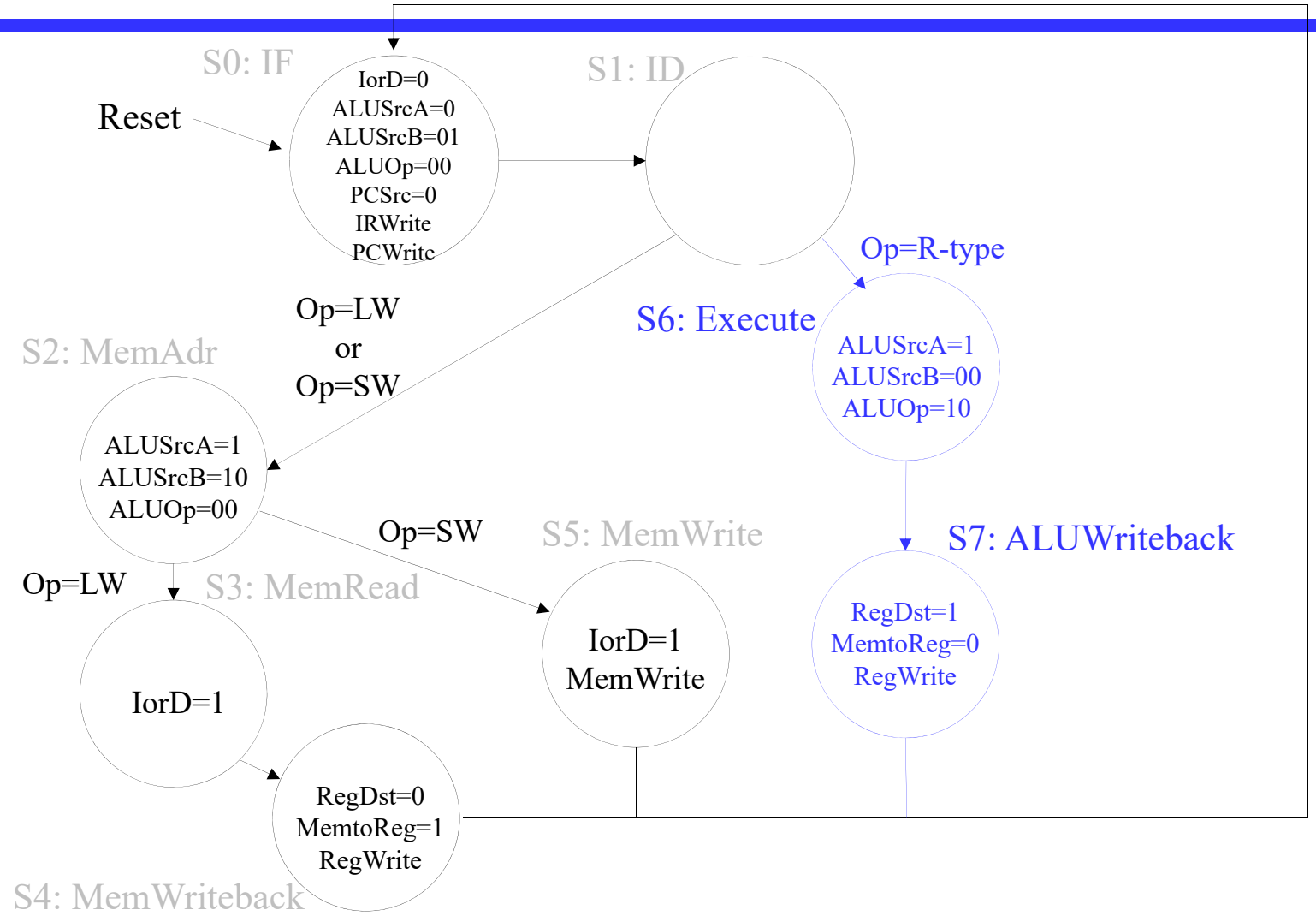
Hardwired Controller - lw



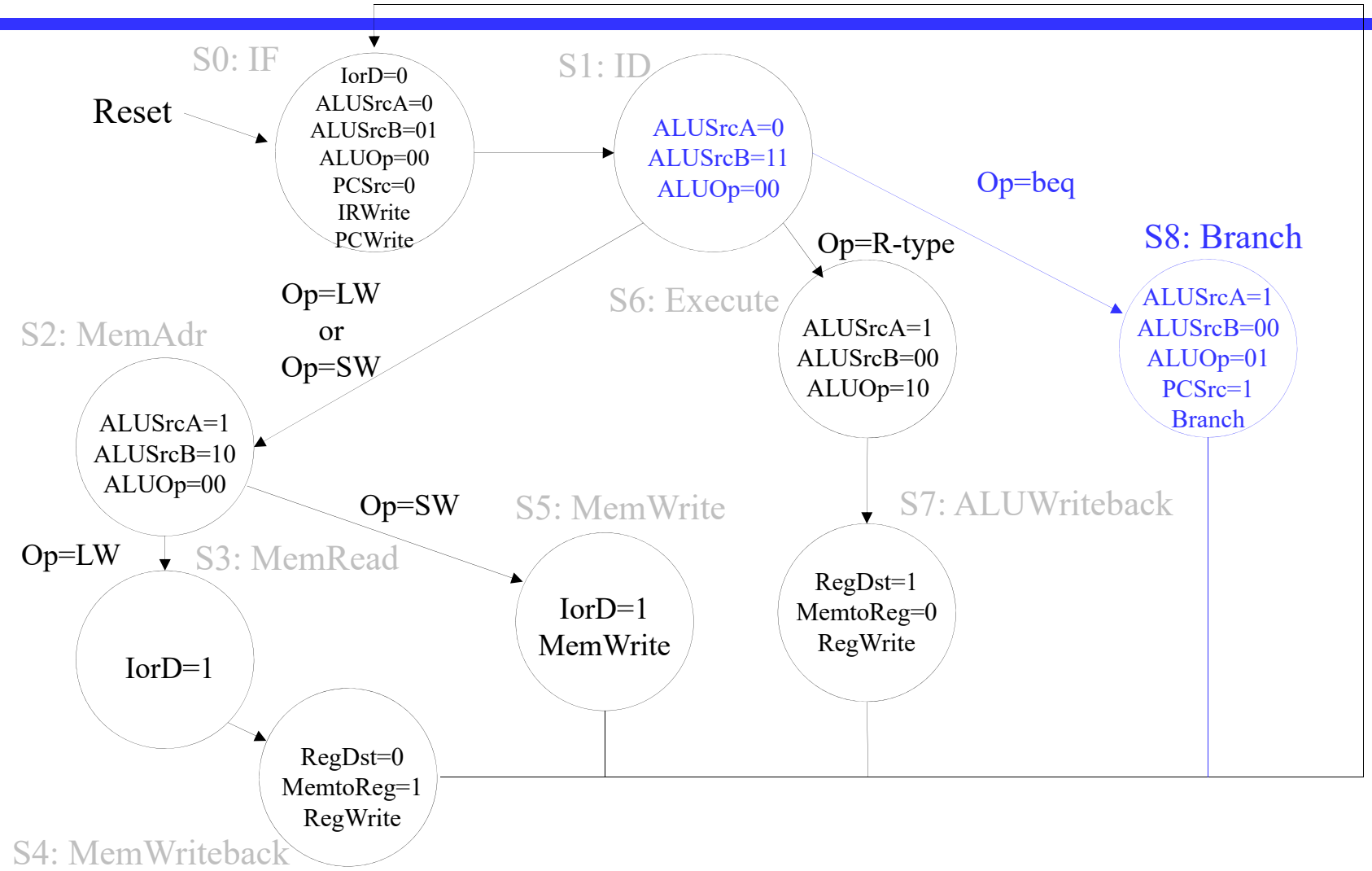
Hardwired Controller - sw



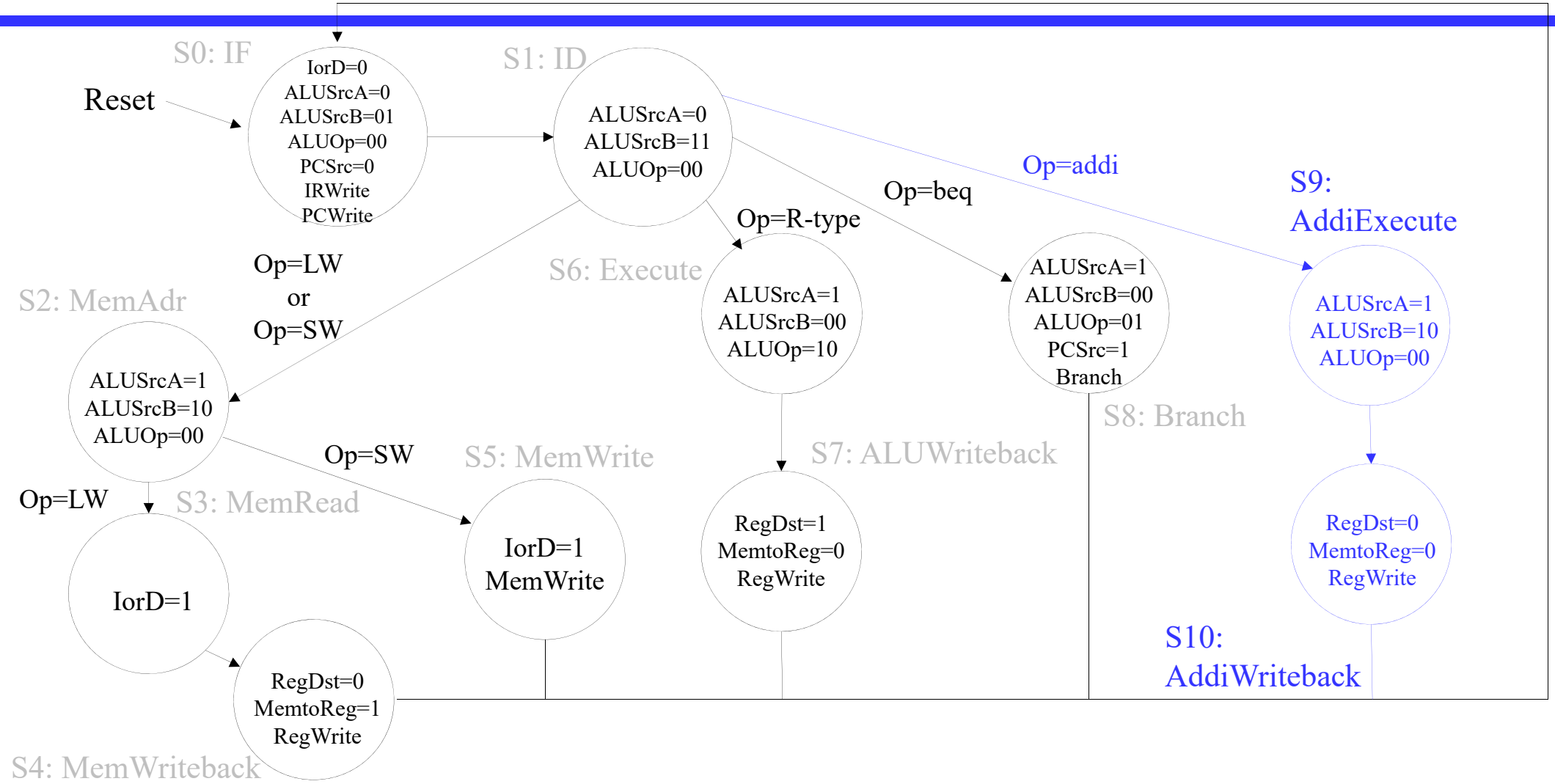
Hardwired Controller – R-type



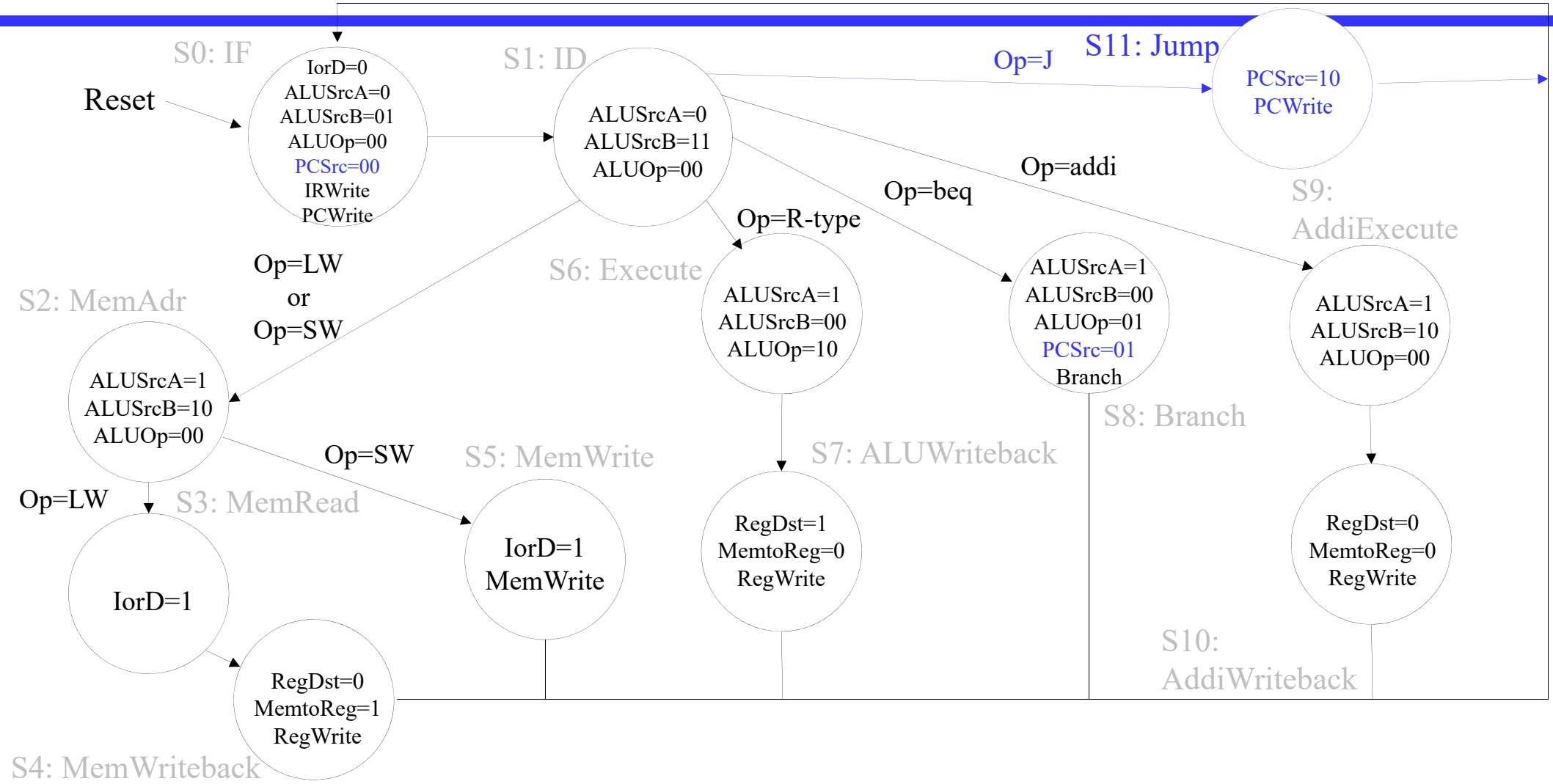
Hardwired Controller – beq



Hardwired Controller – addi

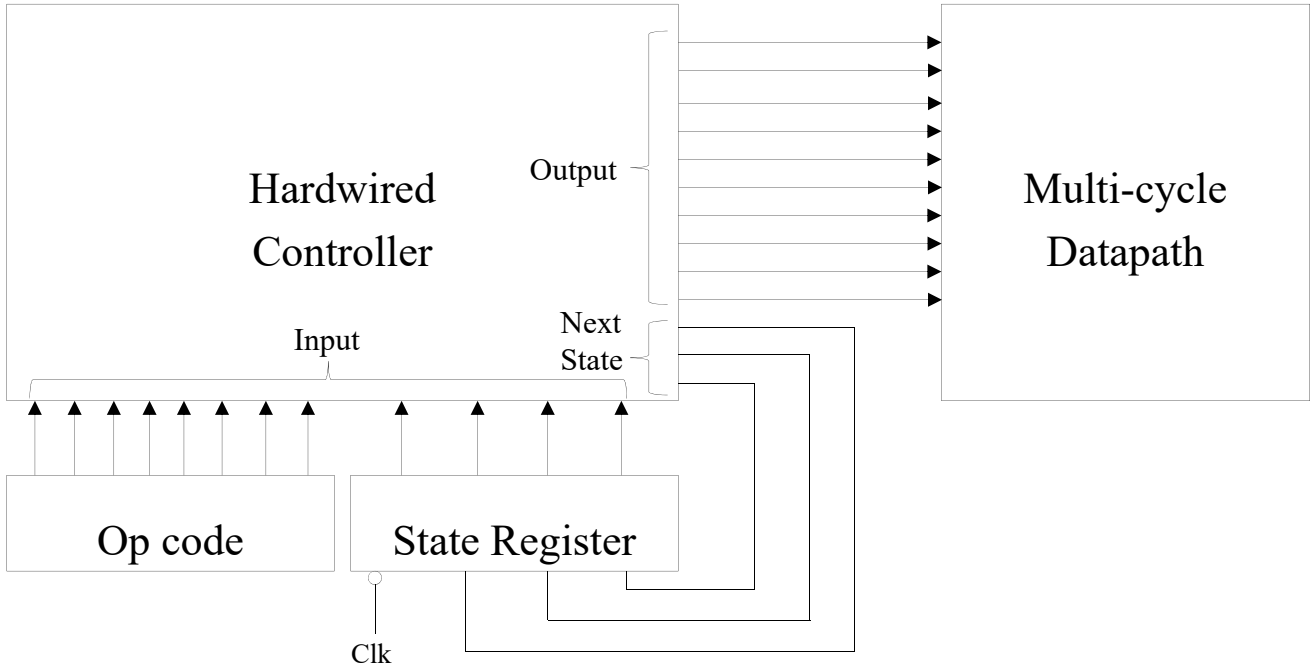


Hardwired Controller – J



Hardwired Controller Implementation

- PLA and state register
  - Based on Finite State Machine (FSM)



Micro-program Controller

- Pros & cons of hardwired controller
  - Pros: generation time of control signals is short
  - Cons: if ISA is complex, it is hard to update and maintain the hardwired controller
- How about CISC?
  - Obviously, hardwired controller is not appropriate
  - Hence comes to micro-program controller

Micro-program Controller

- History
  - Proposed by M. V. Wilkes in 1951
  - First deployed by IBM 360 series in 1964
  - Since 1970s, as the development of VLSI, micro-program design has achieved significant development and application
  - Currently, most CISC computers widely apply micro-program design
- Basic principle
  - Represent the execution of each instruction with a micro program
  - Consist of several micro-instructions, each of which is one state in FSM
  - All micro-programs are stored in a ROM, which is called Control Storage (CS)

Micro-program Controller

- Micro-program
  - A micro-instruction sequence
  - Correspond to each instruction in ISA

- Micro-instruction
  - A 0/1 sequence
  - Consist of several micro-command
  - aka. Control Word (CW)

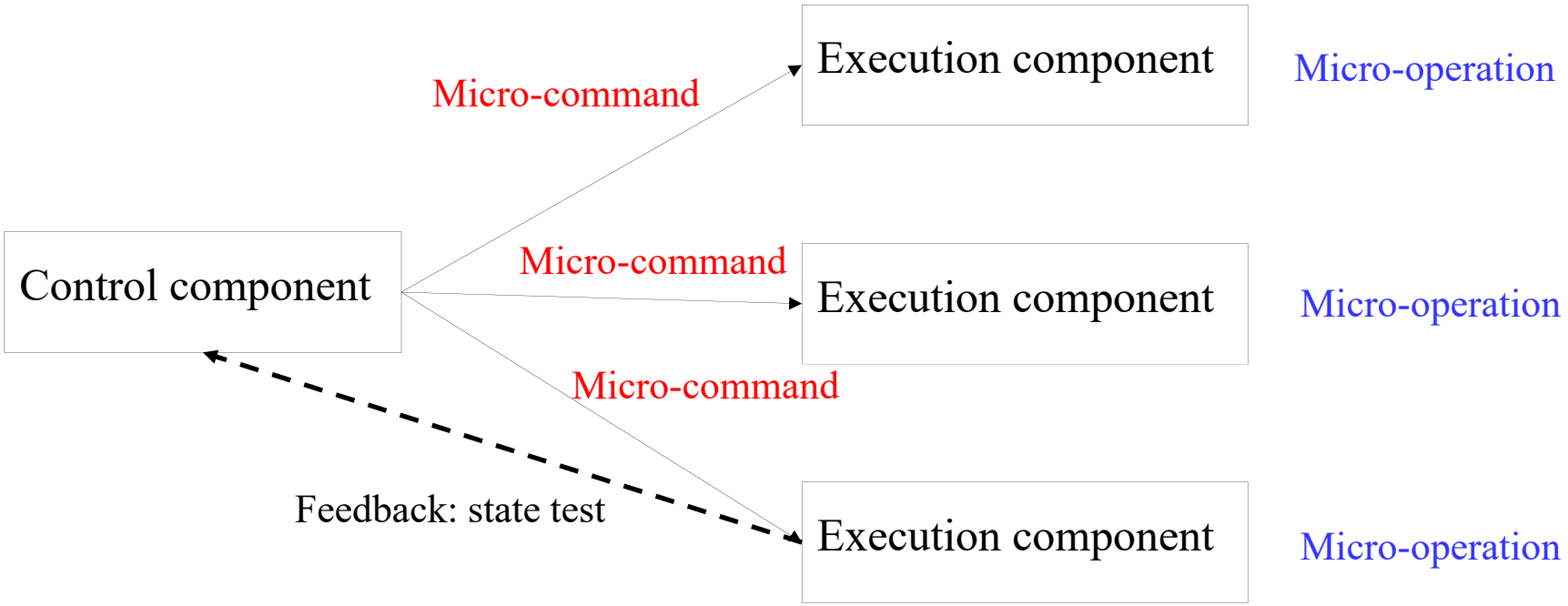
Micro-program Controller

- Micro-command
  - Control command sent by control component to execution component
  - Unit of control sequence
    - E.g., clock signal for register, 0 or 1 signal
- Micro-operation
  - Correspond to micro-command one to one
  - Detailed operations for micro-command

Similar to control signals and operations in datapath

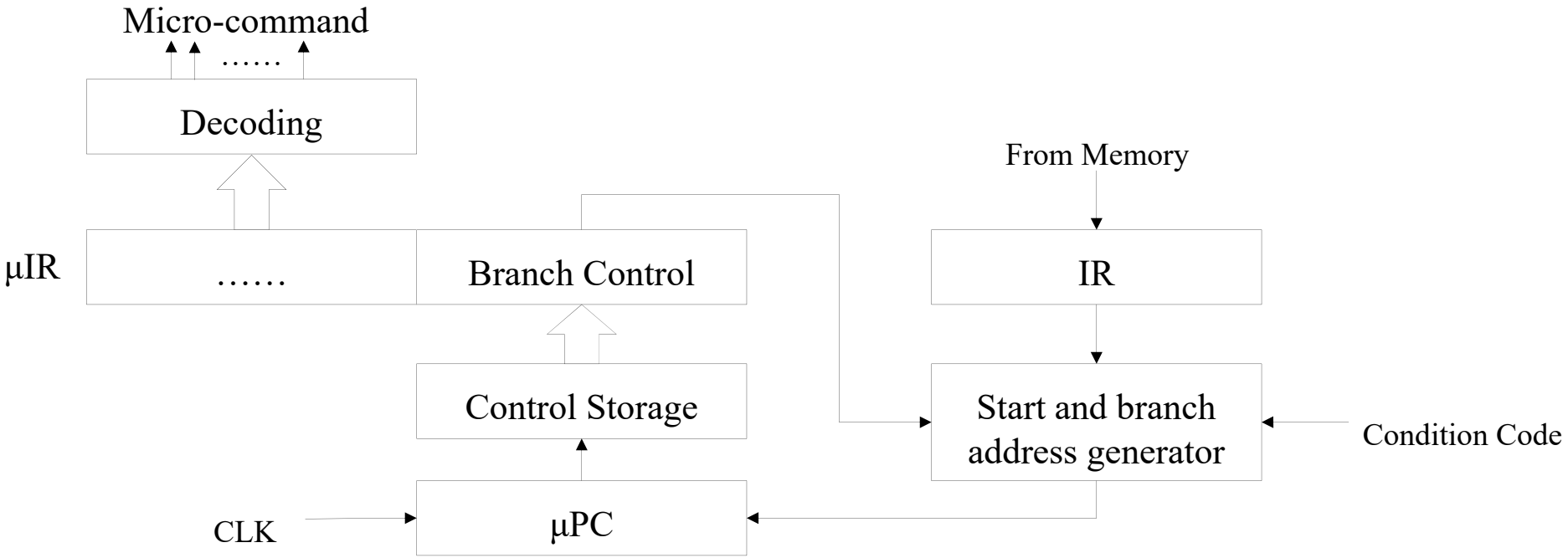
Micro-program Controller

- Micro-command & micro-operation



Micro-program Controller Implementation

- Basic architecture



Micro-program Controller Implementation

- Two problems:
  - Format of micro-instruction and coding of micro-command
  - Determination of next micro-instruction address

- Solutions:
  - Long micro-instruction word and its format refers to instruction format
  - Program counter or explicit address in instruction format

Pros & Cons of Micro-program Controller

- Pros:
  - Significantly lower the difficulty of controller design
  - Improving the flexibility for ISA and CPU design
- Cons:
  - Longer execution time than hardwired controller
  - Lower efficiency

Performance of Multi-cycle CPU

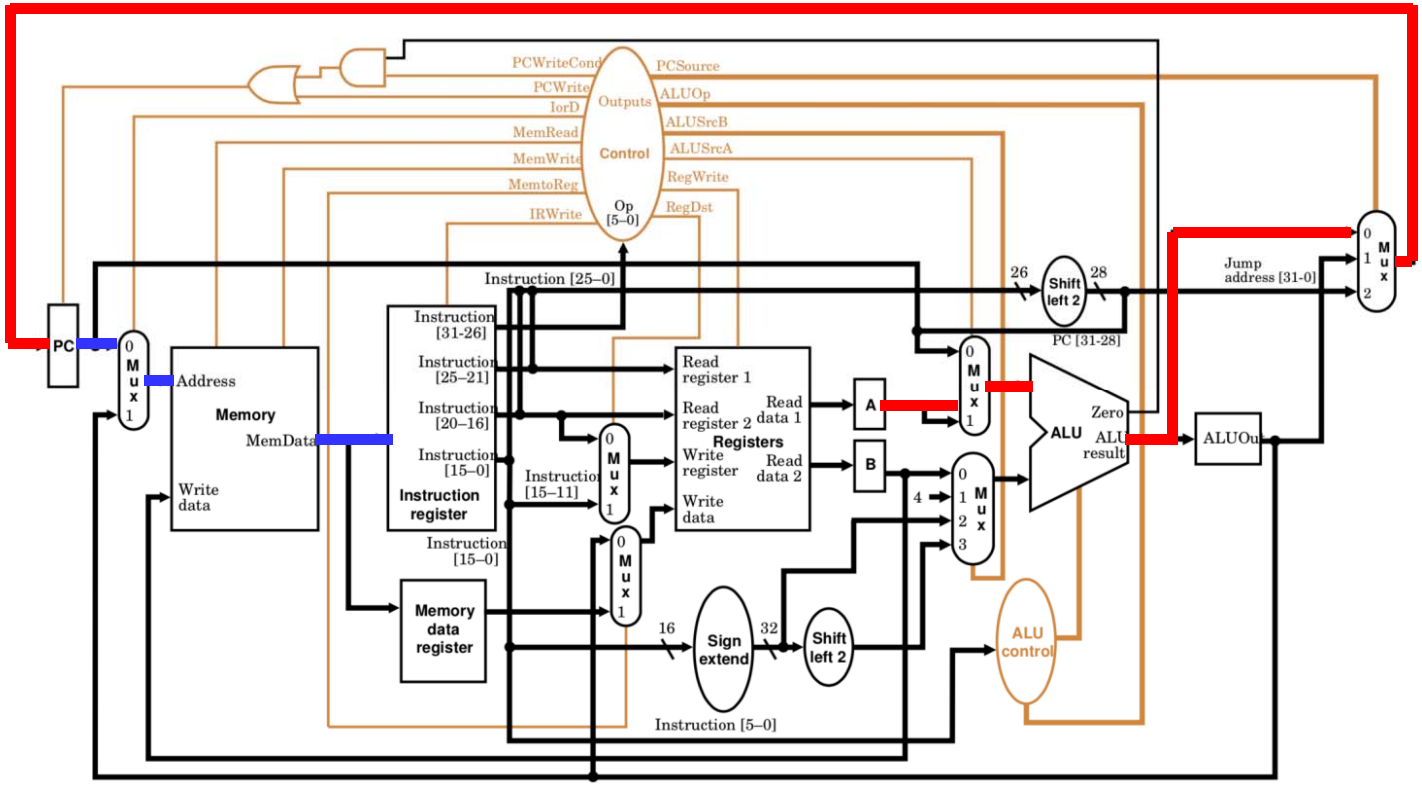
- Instructions take different number of cycles:
  - 3 cycles: beq, j
  - 4 cycles: R-type, sw
  - 5 cycles: lw

- CPI is weighted average:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

- Average CPI = (0.11+0.02)*3+(0.52+0.1)*4+0.25*5 = 4.12

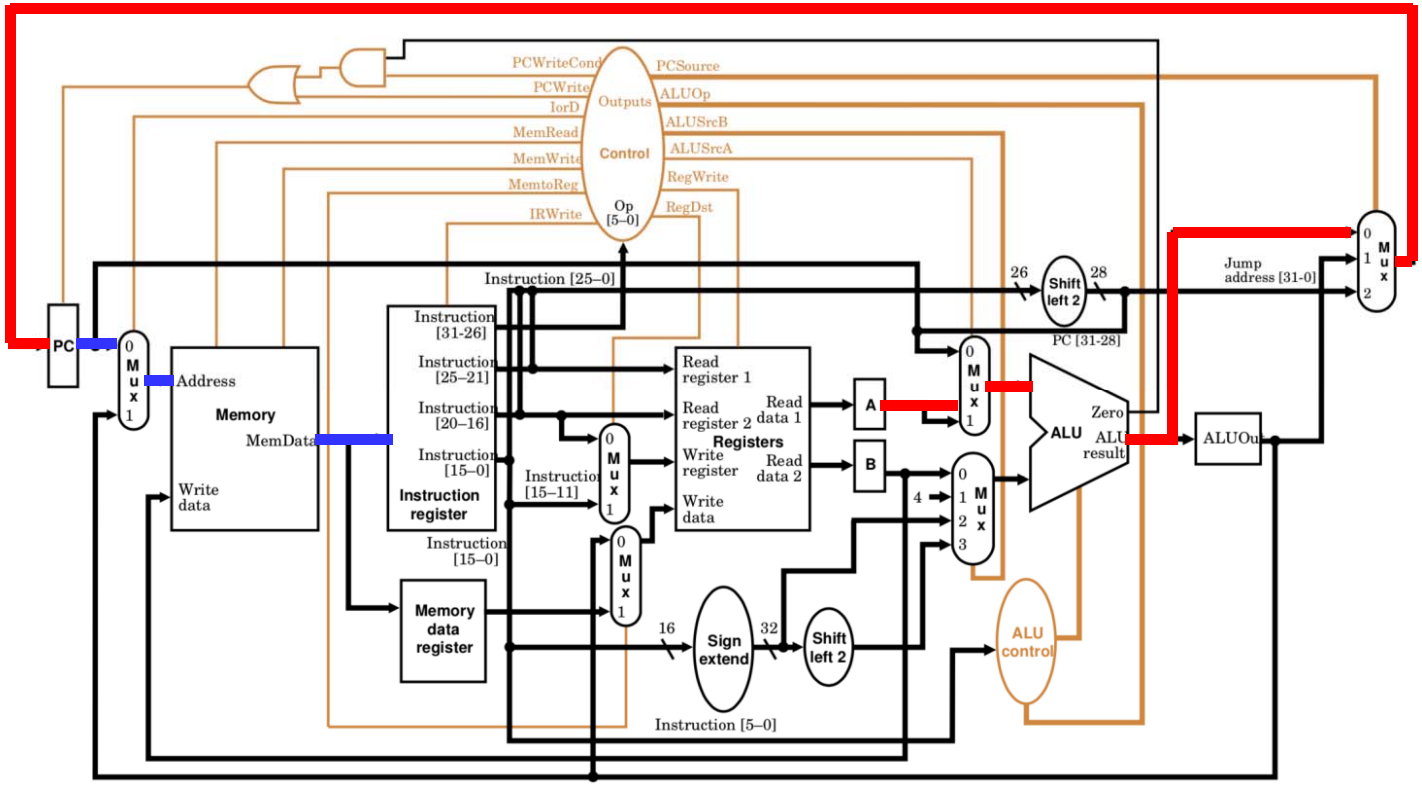
Performance of Multi-cycle CPU

- Multi-cycle critical path:
  - Tc =



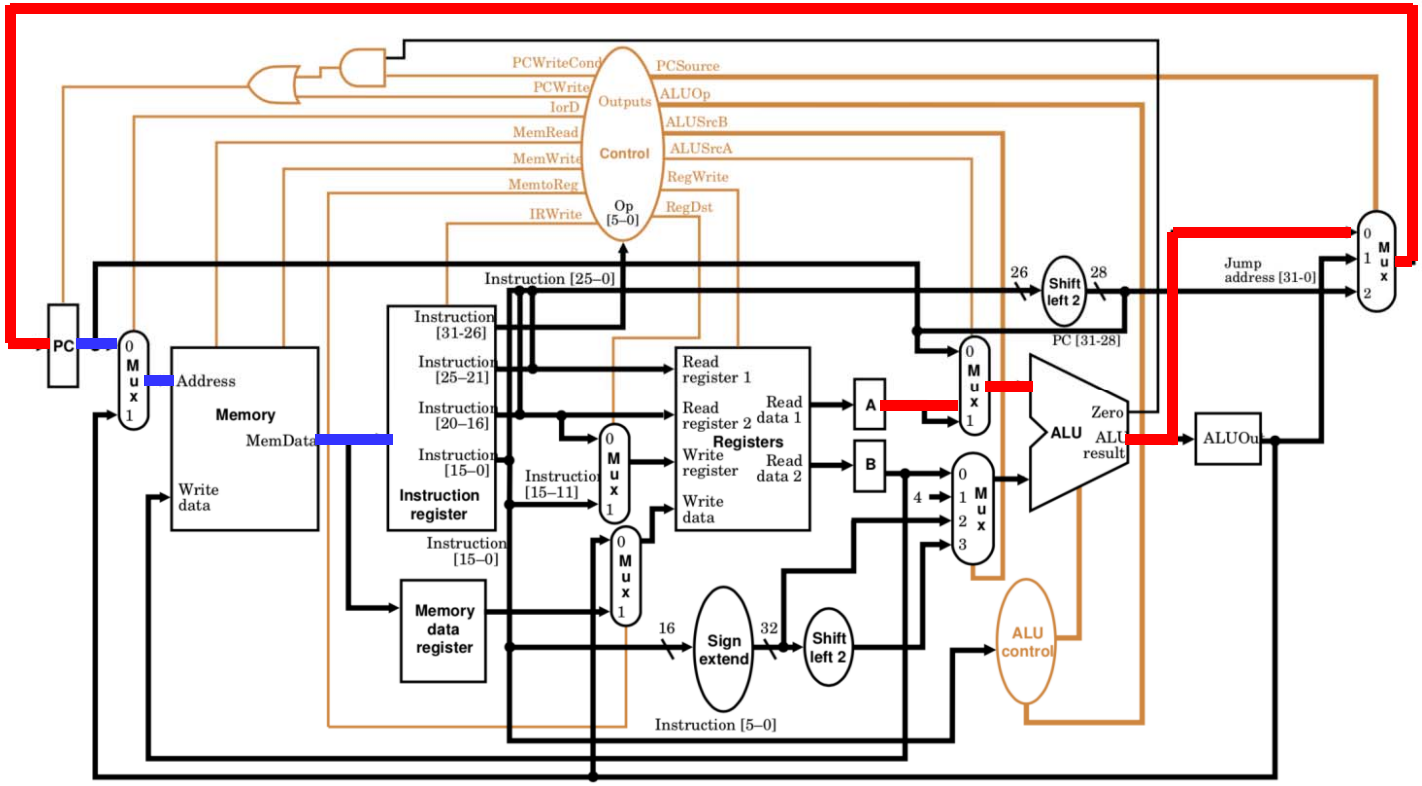
Performance of Multi-cycle CPU

- Multi-cycle critical path:
  - Tc = $t_{pcq} + t_{MUX} + t_{ALU} + t_{MUX} + t_{Setup}$



Performance of Multi-cycle CPU

- Multi-cycle critical path:
  - Tc = $t_{pcq} + t_{MUX} + t_{mem} + t_{Setup}$



Performance of Multi-cycle CPU

- Time for each step:

Step	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq}	30
Register setup	t_{Setup}	20
Multiplexer	t_{MUX}	25
ALU	t_{ALU}	200

Memory read dominates the critical path!

Tc = $t_{pcq} + t_{MUX} + t_{ALU} + t_{MUX} + t_{Setup}$ = 30+25+200+25+20 = 300ps

Tc = $t_{pcq} + t_{MUX} + t_{mem} + t_{Setup}$ = 30+25+250+20 = 325ps

Performance of Multi-cycle CPU

- Example:
  - For a program with 100 billion instructions executing on a multi-cycle RISC-V CPU
    - CPI = 4.12
    - Tc = 325 ps
  - Execution Time = ?
  - Execution Time = $100*10^{9} * 4.12 * 325*10^{-12} = 133.9$ s
  - How about single-cycle CPU?
  - 92.5s < 133.9s

Performance Issues

- Even Longer Time for Instruction Execution
  - Cannot achieve expected performance compared with single-cycle
- But provide a new perspective for CPU design
  - Finer-grained execution in one clock cycle
- We will improve performance by pipelining

System II