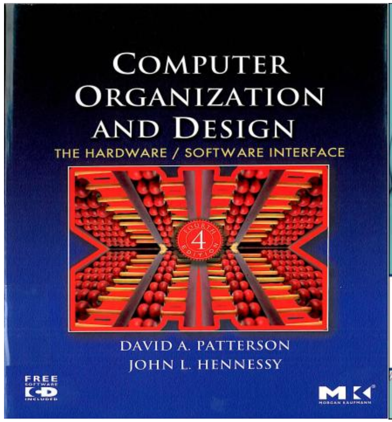
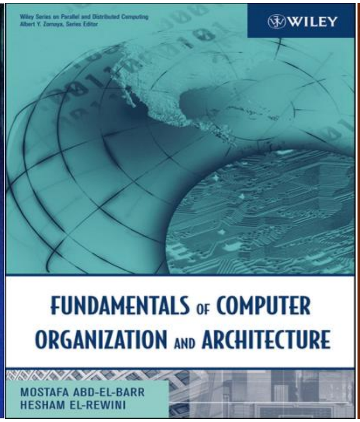
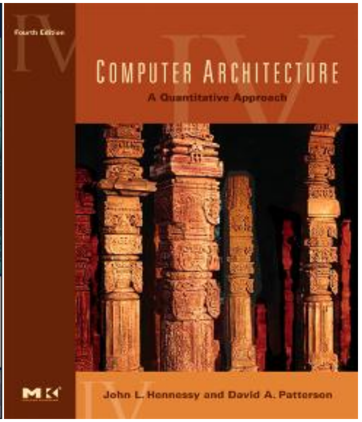
Computer Architecture

Lecture 9&10&11: The processor – DataPath & Control





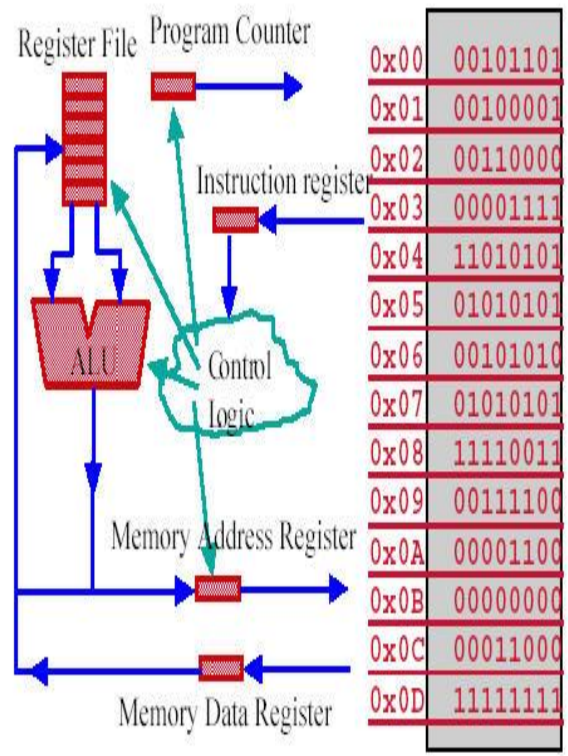


Nguyen Minh Son, Ph.D

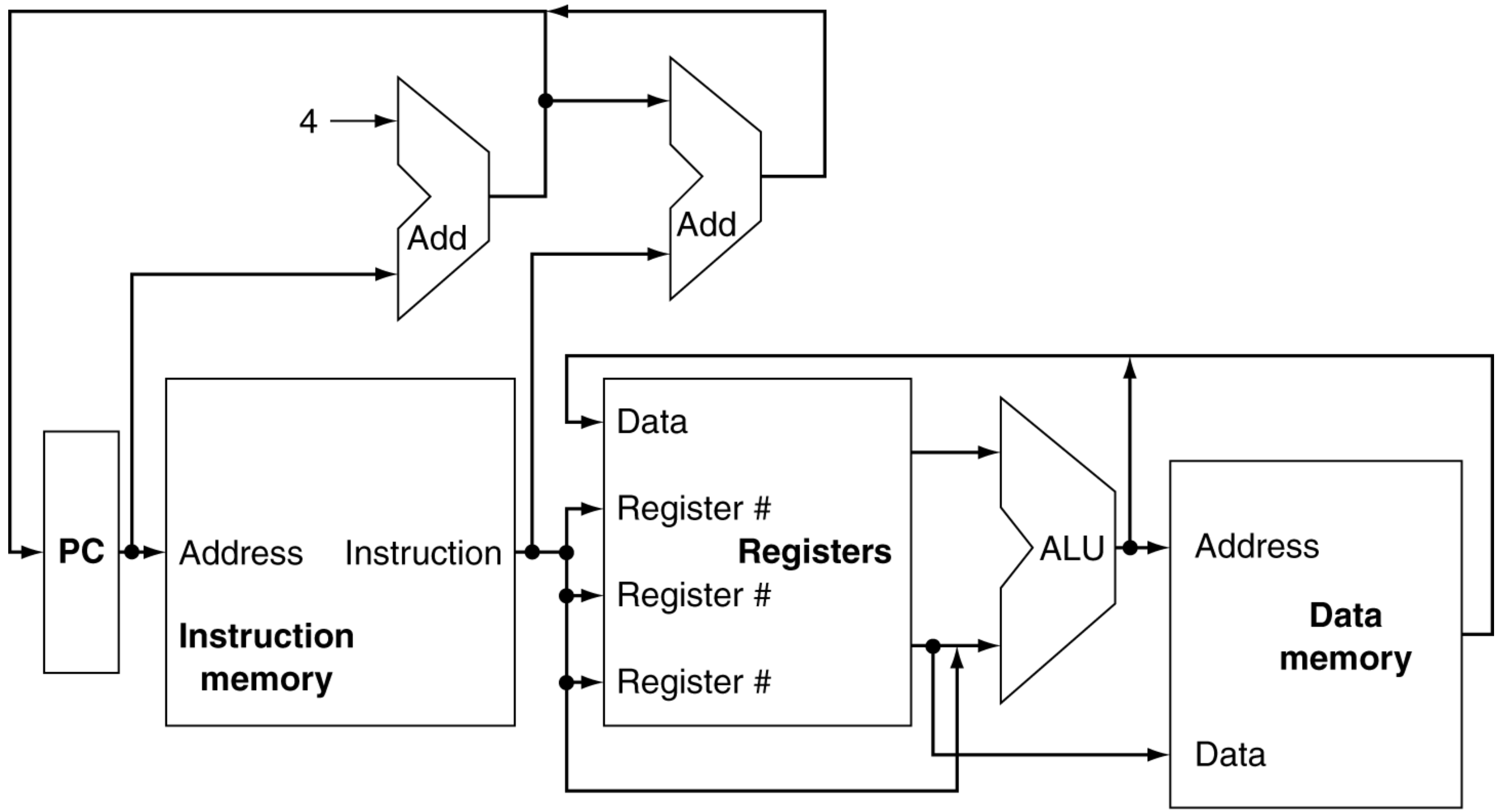
UIT-HCM

Instruction Execution

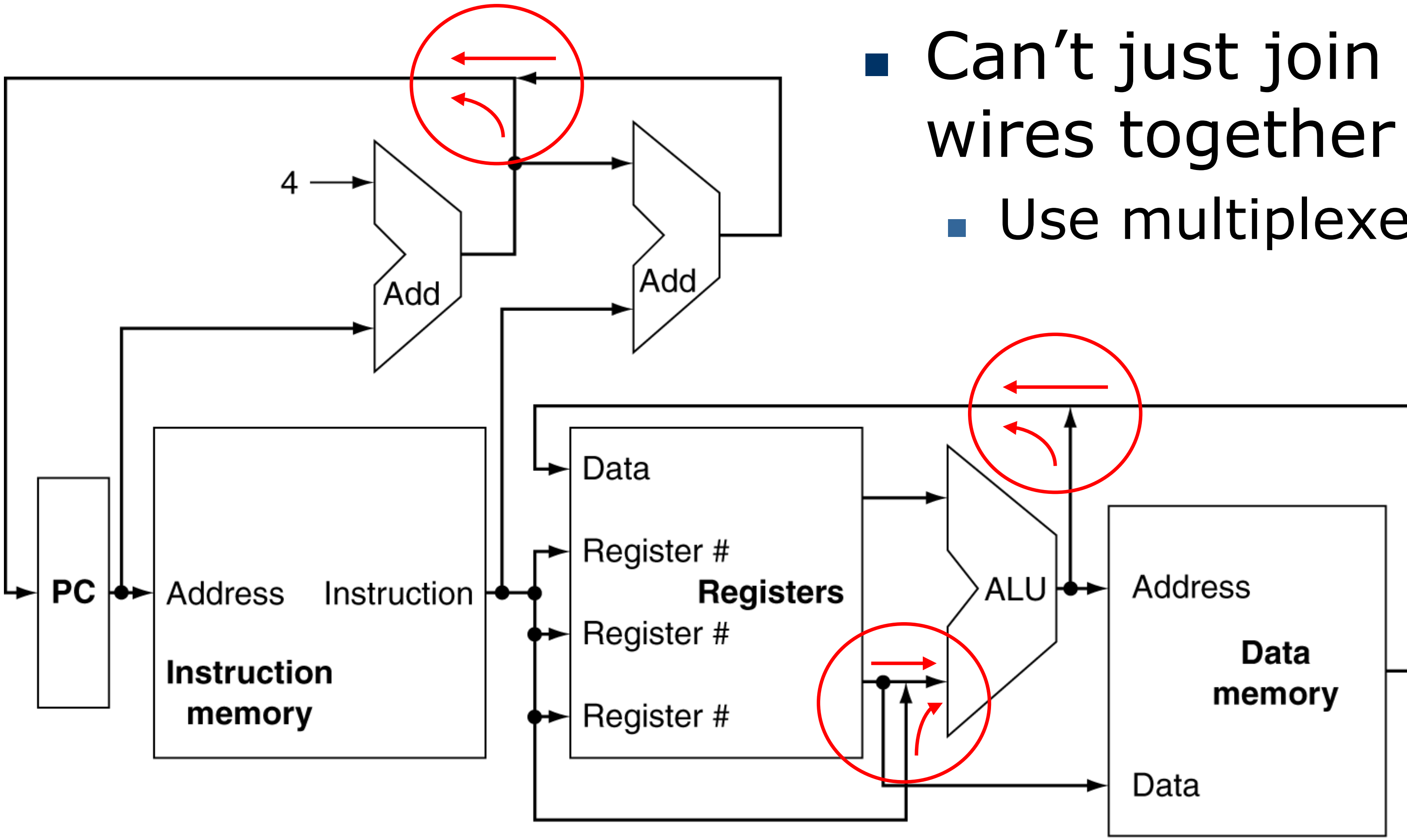
- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address or PC + 4



CPU Overview

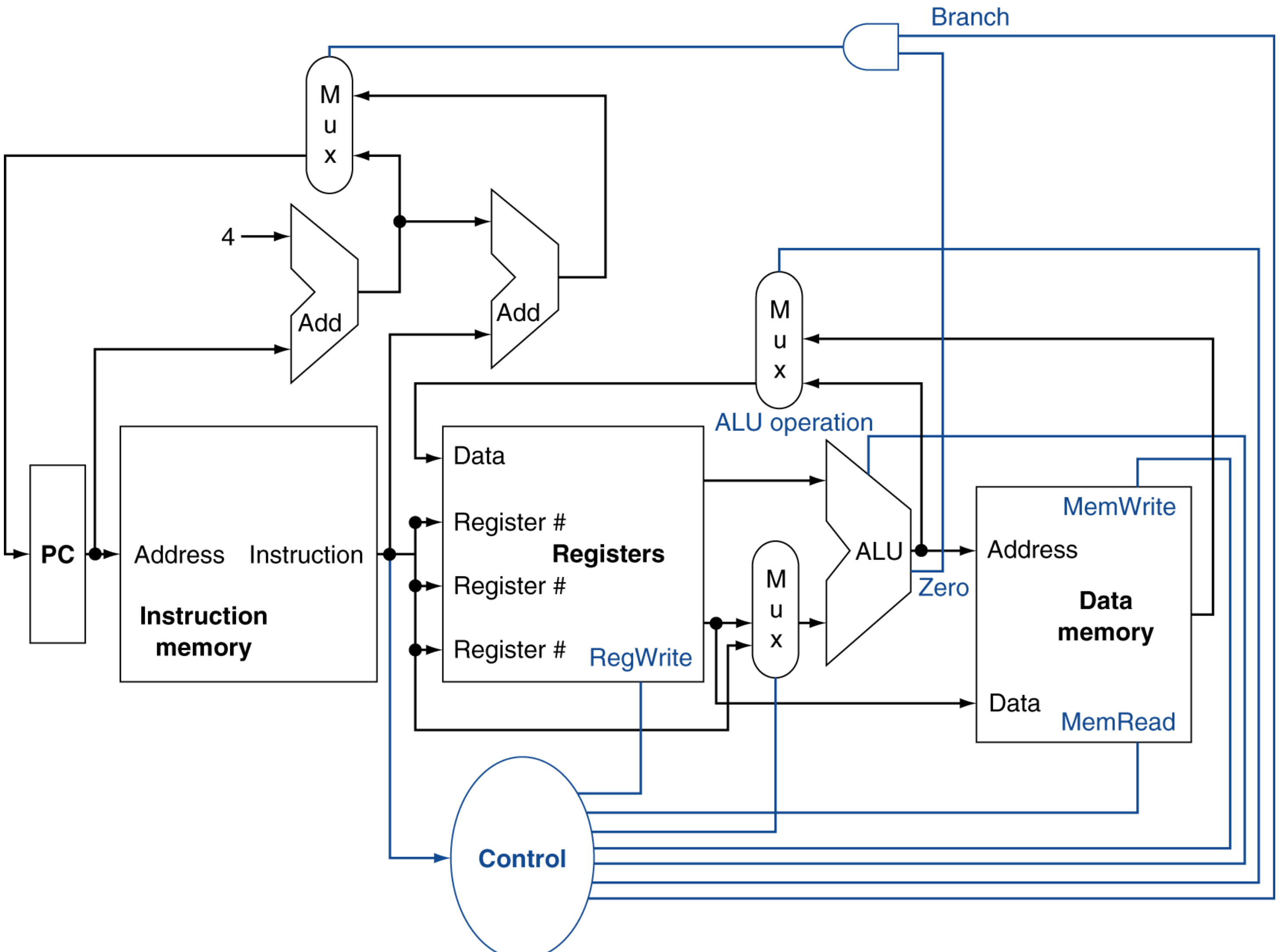


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control



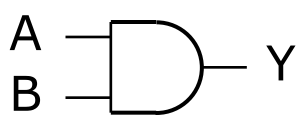
Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Review: Logic Design Basics

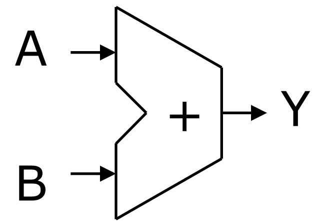
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

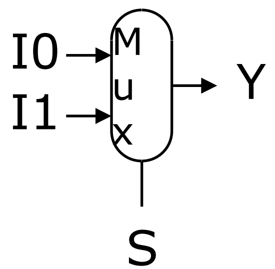
- AND-gate
  - Y = A & B



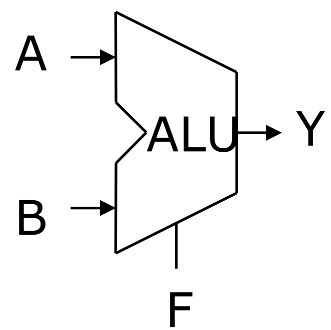
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0



- Arithmetic/Logic Unit
  - Y = F(A, B)

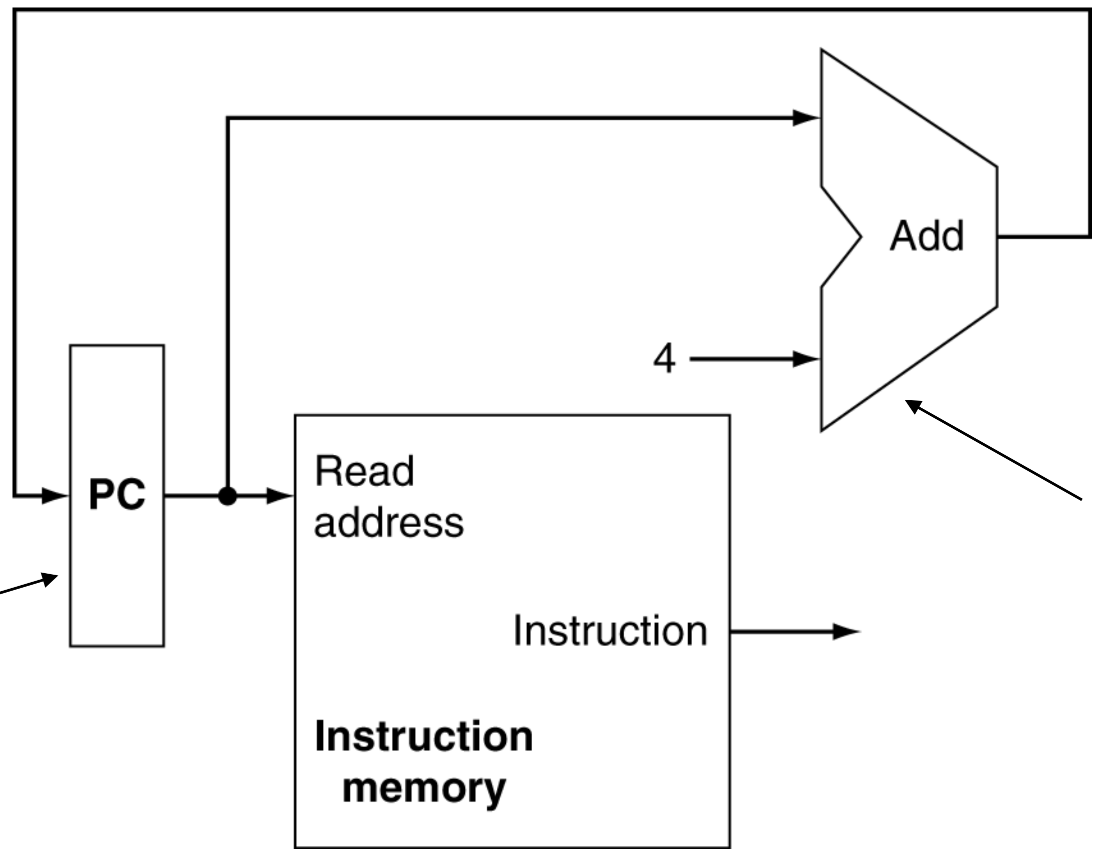


Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Instruction Fetch

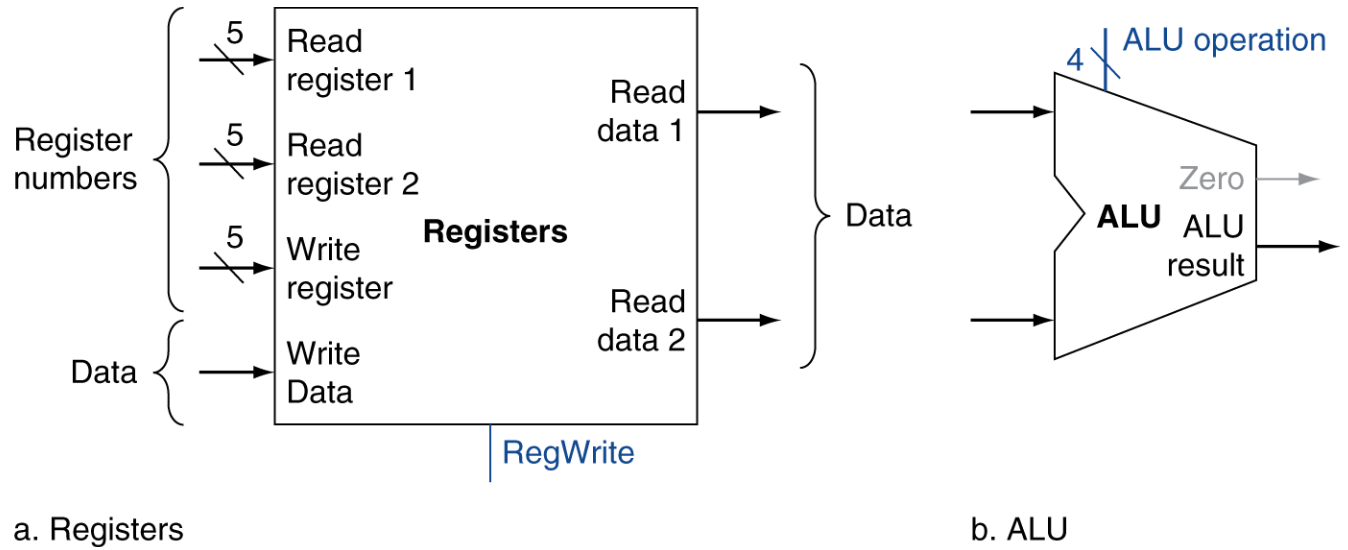
32-bit register



Increment by 4 for next instruction

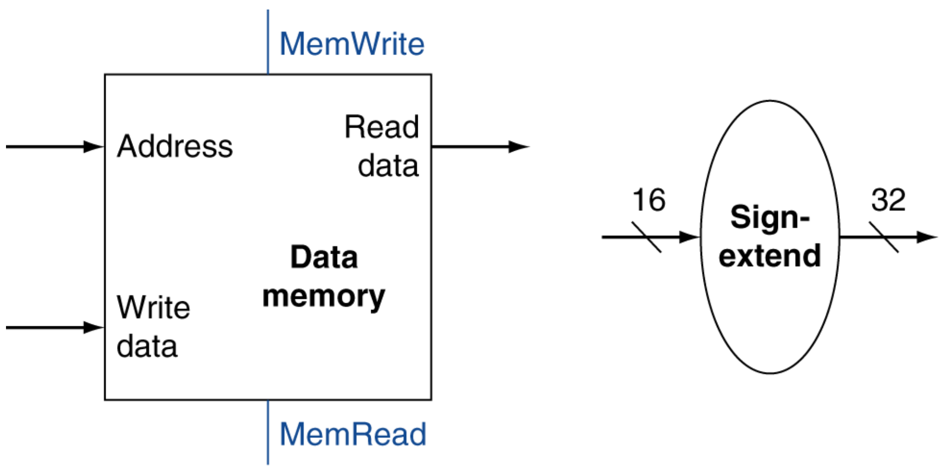
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit

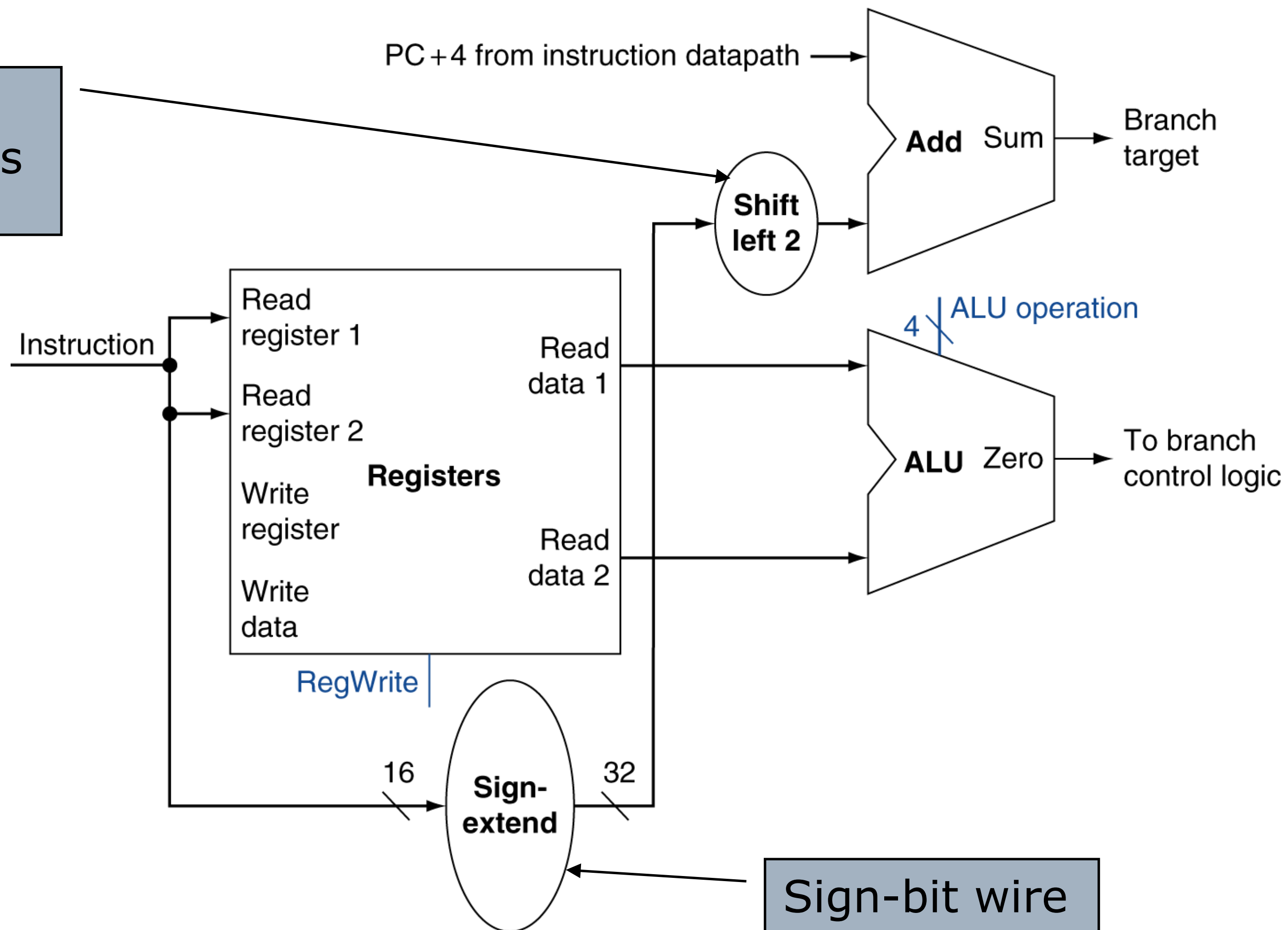
b. Sign extension unit

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Branch Instructions

Just re-routes wires

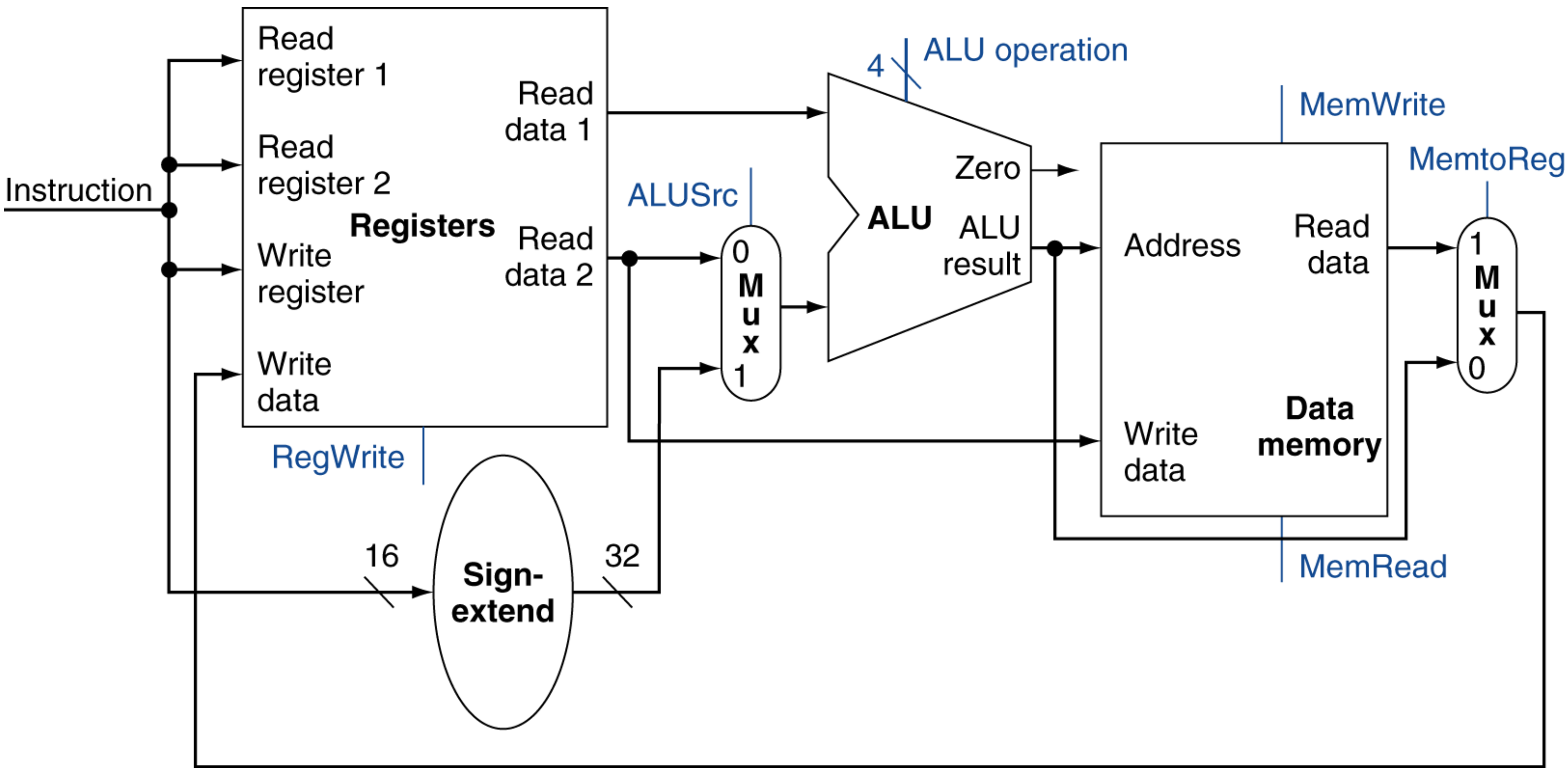


Sign-bit wire replicated

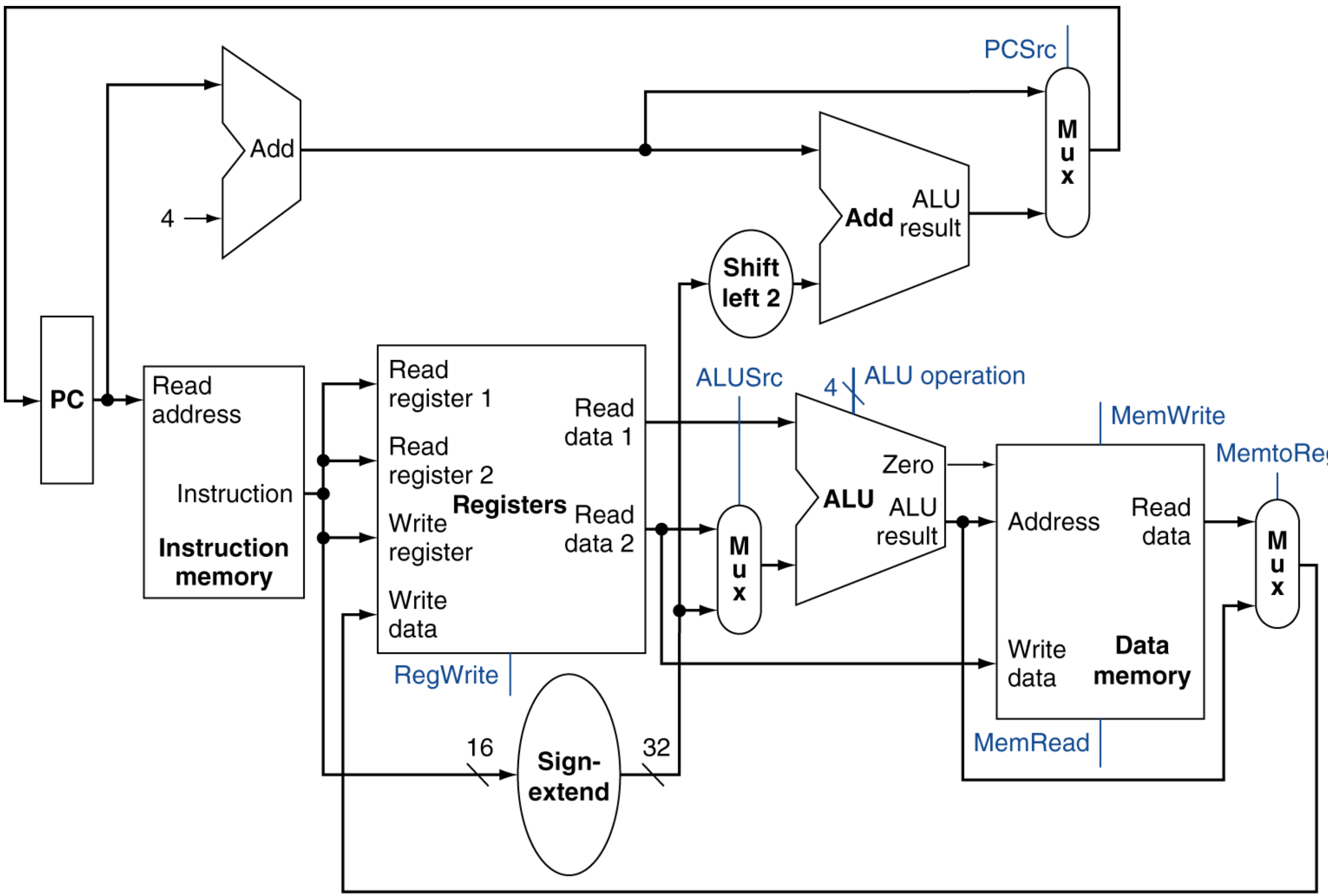
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

- Control signals derived from instruction

R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0

Load/ Store	35 or 43	rs	rt	address
	31:26	25:21	20:16	15:0

Branch	4	rs	rt	address
	31:26	25:21	20:16	15:0

opcode

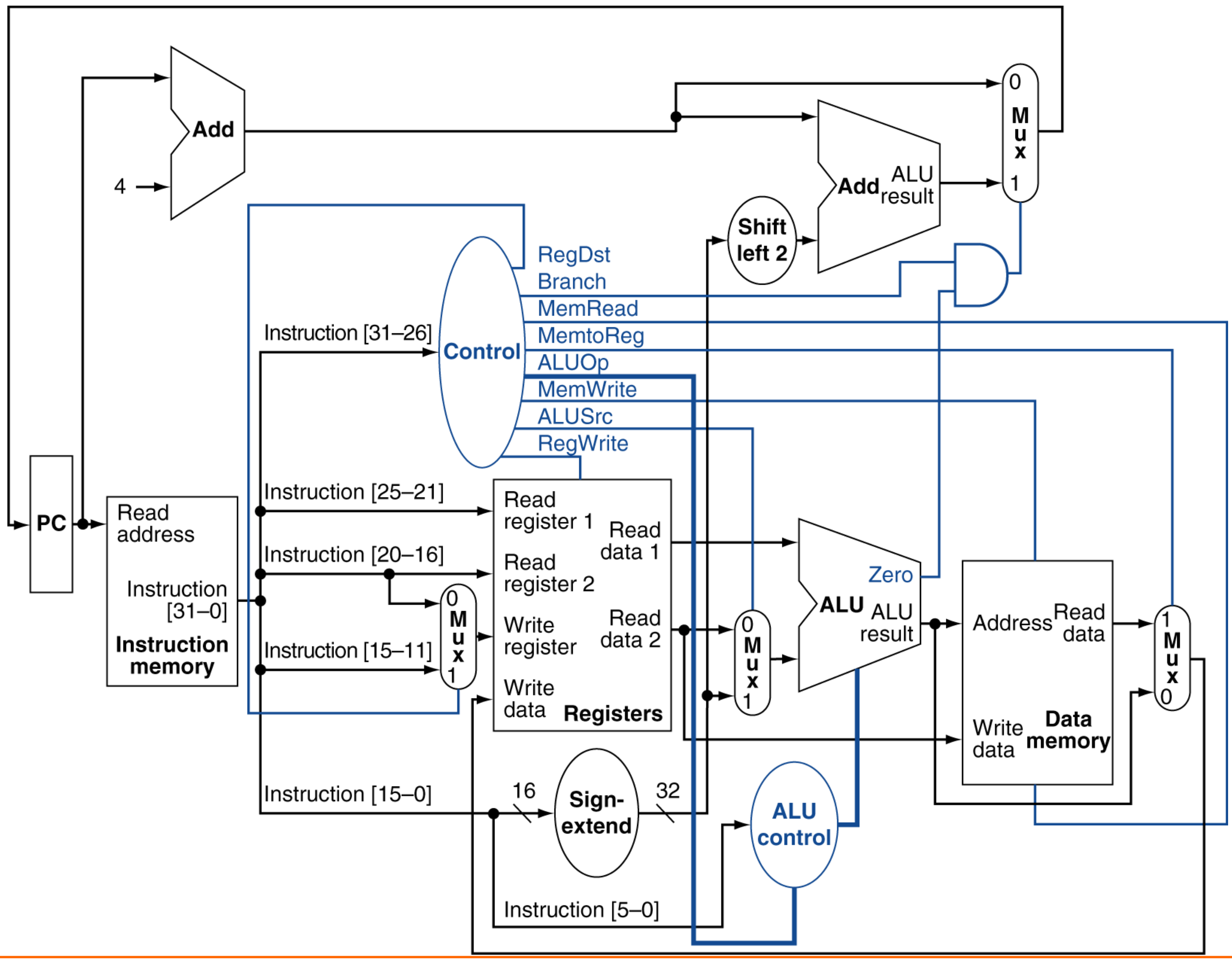
always read

read, except for load

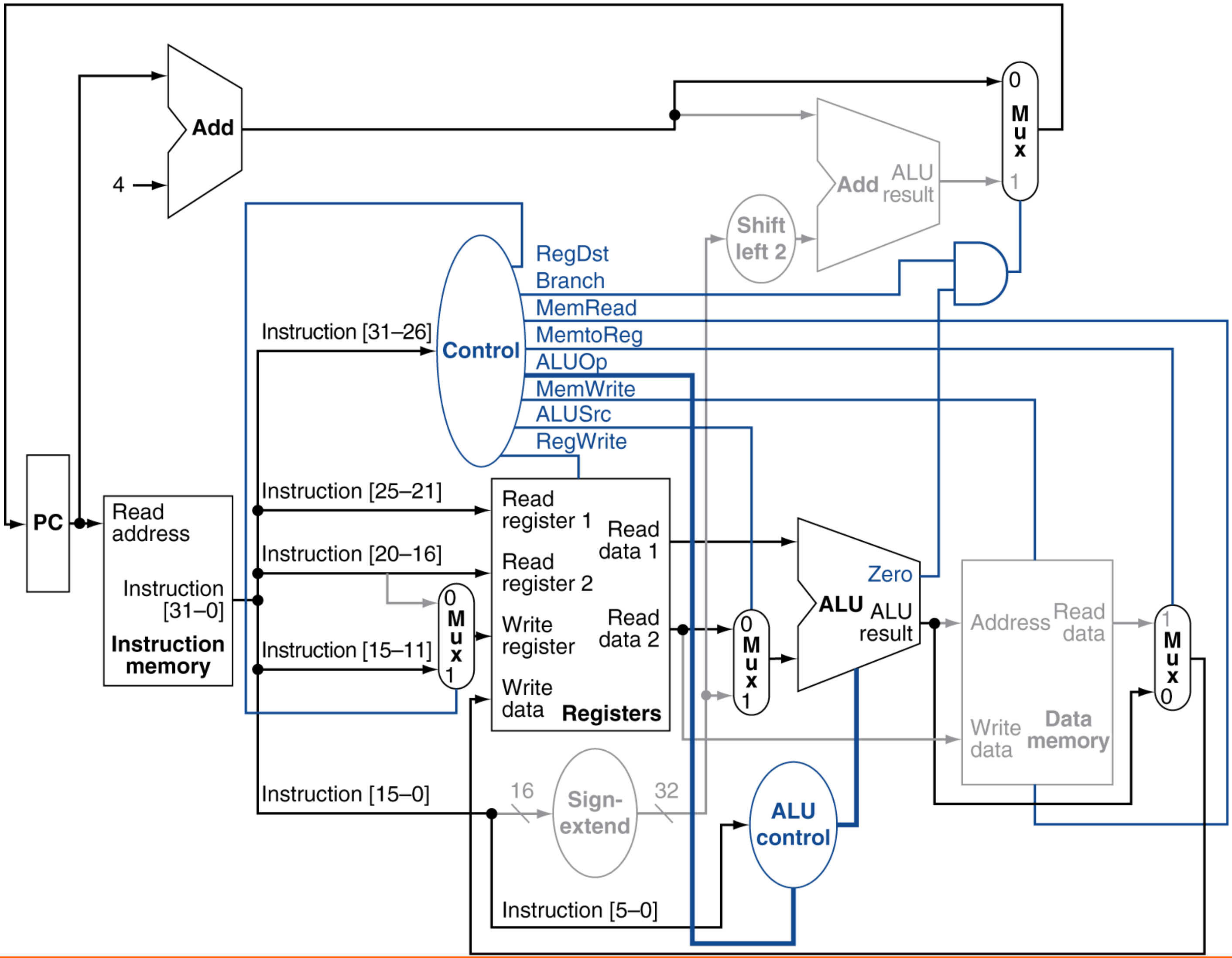
write for R-type and load

sign-extend and add

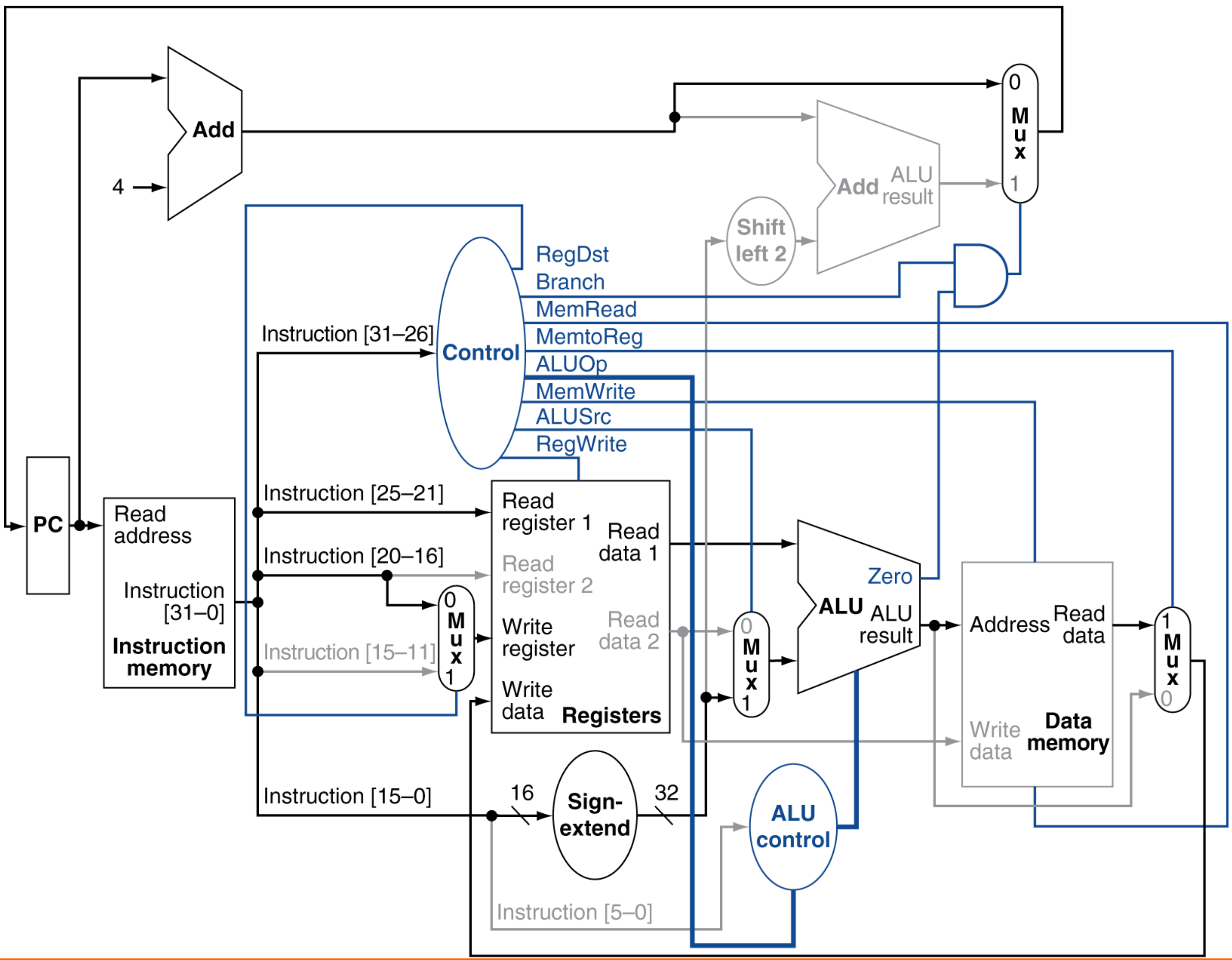
Datapath With Control



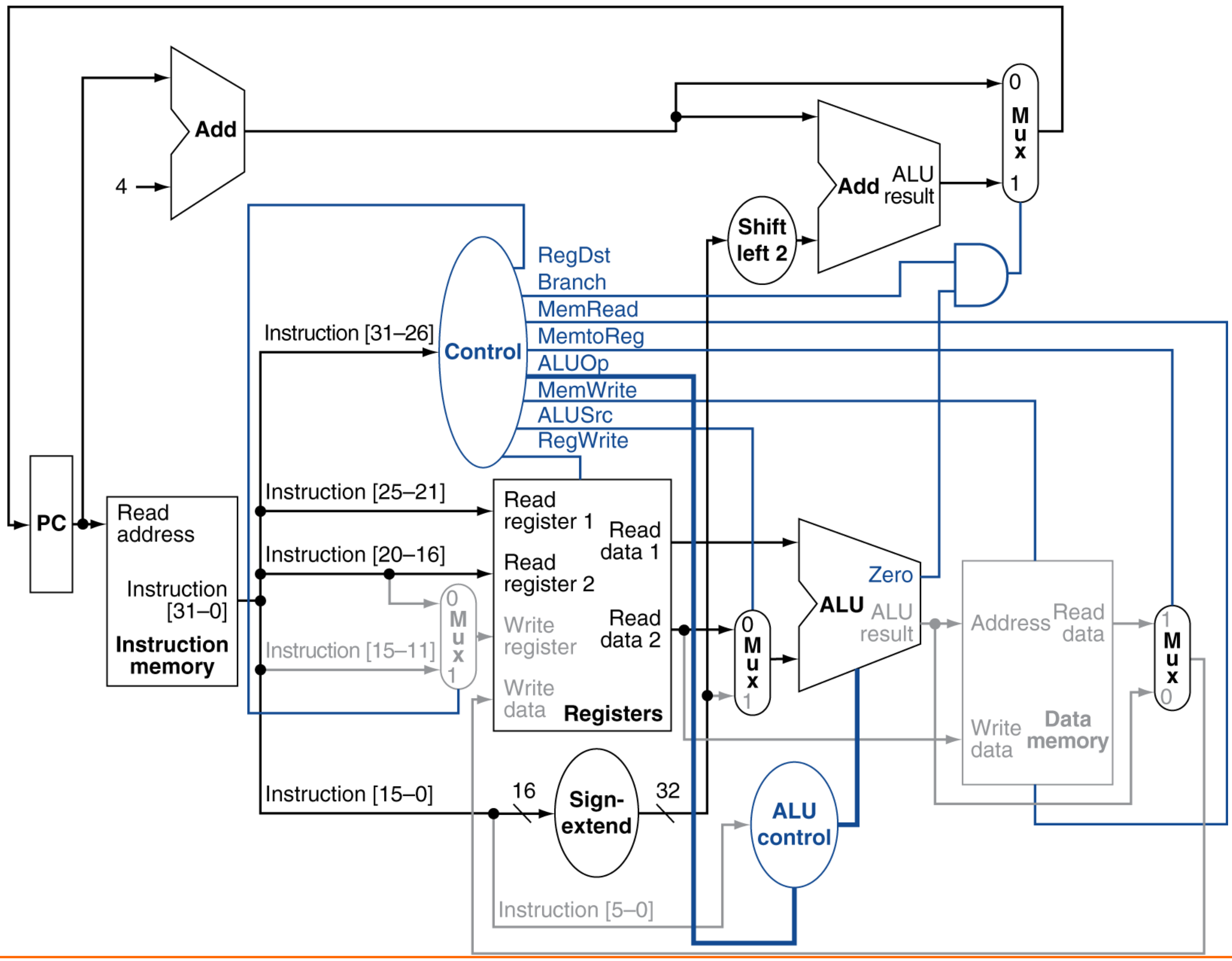
R-Type Instruction



Load Instruction



Branch-on-Equal Instruction

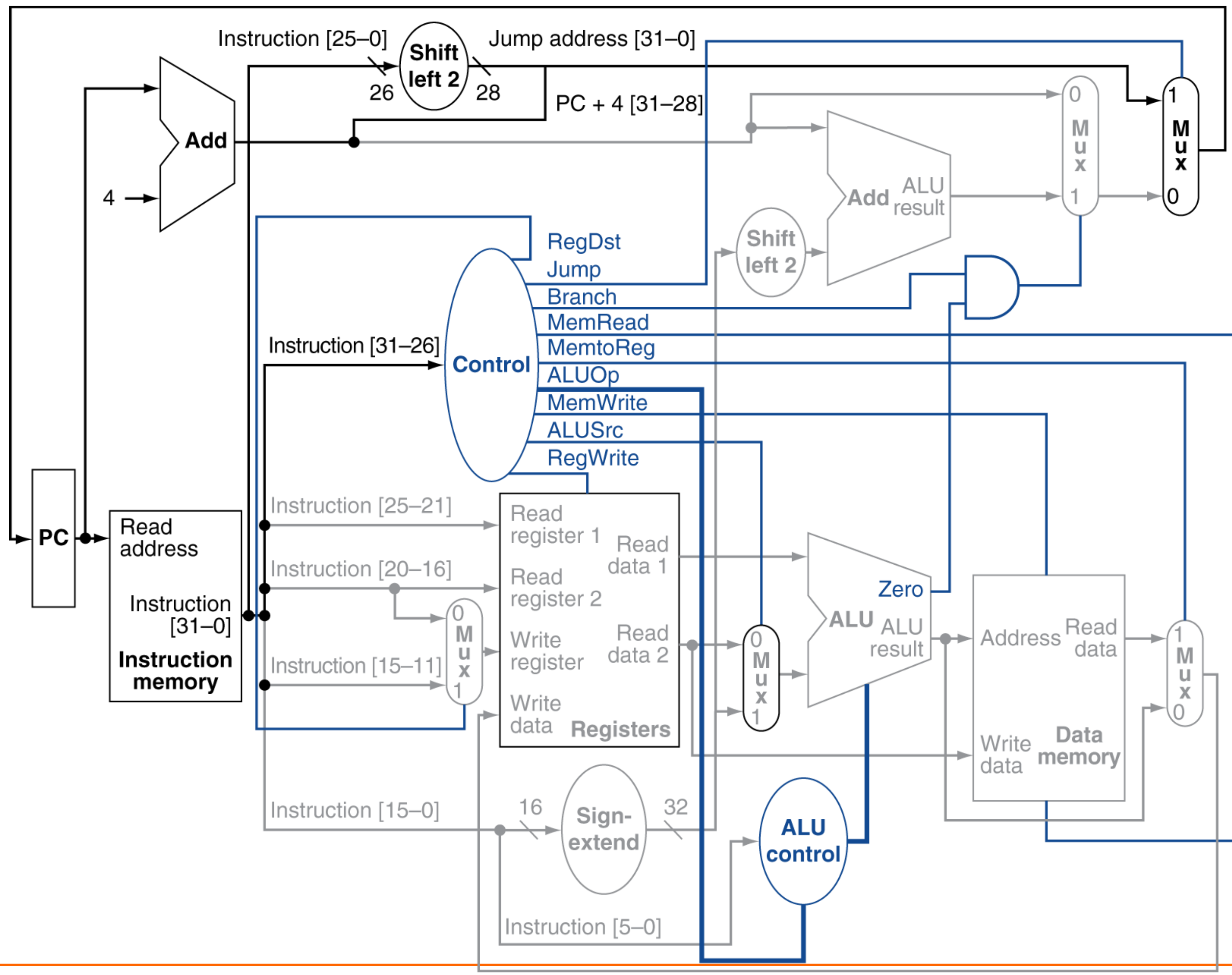


Implementing Jumps

	31:26	25:0
Jump	2	address

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

Datapath With Jumps Added



Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Single-cycle review

- Executing a MIPS instruction can take up to five steps.

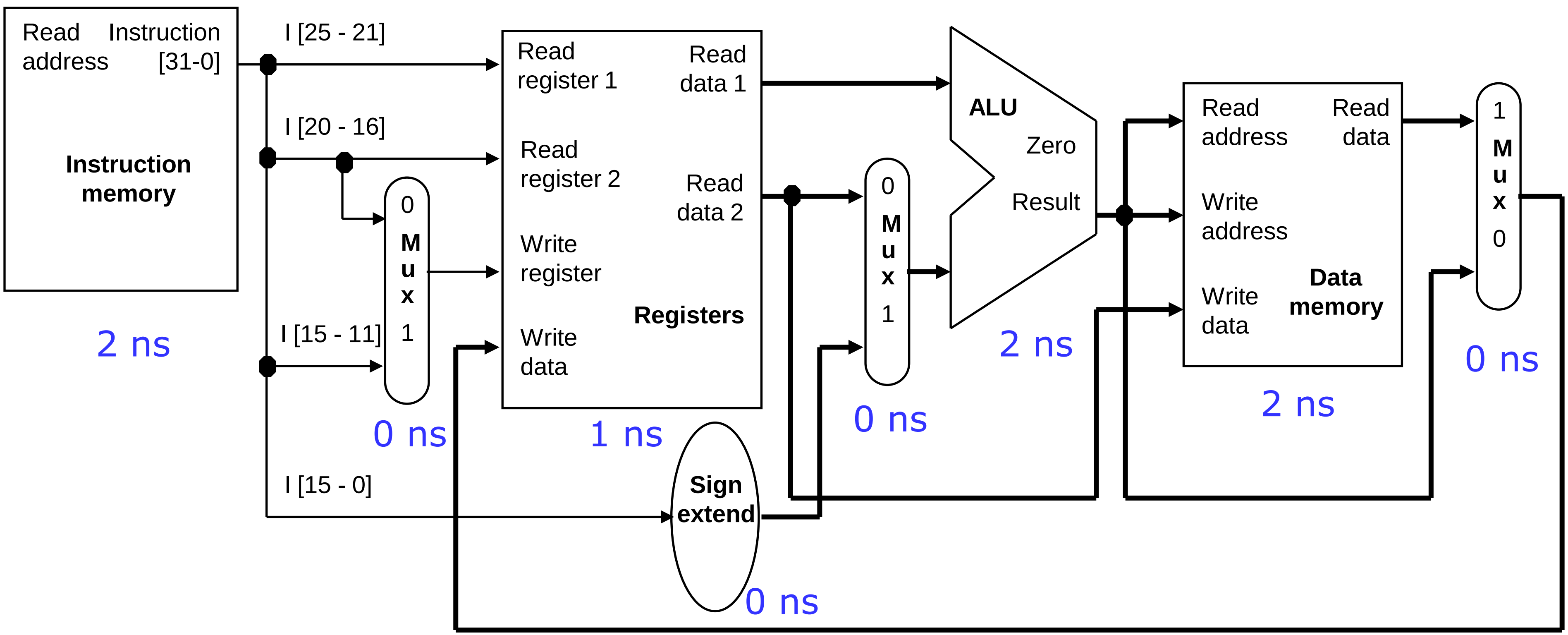
Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch target.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

- However, as we saw, not all instructions need all five steps.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

The slowest instruction ...

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.

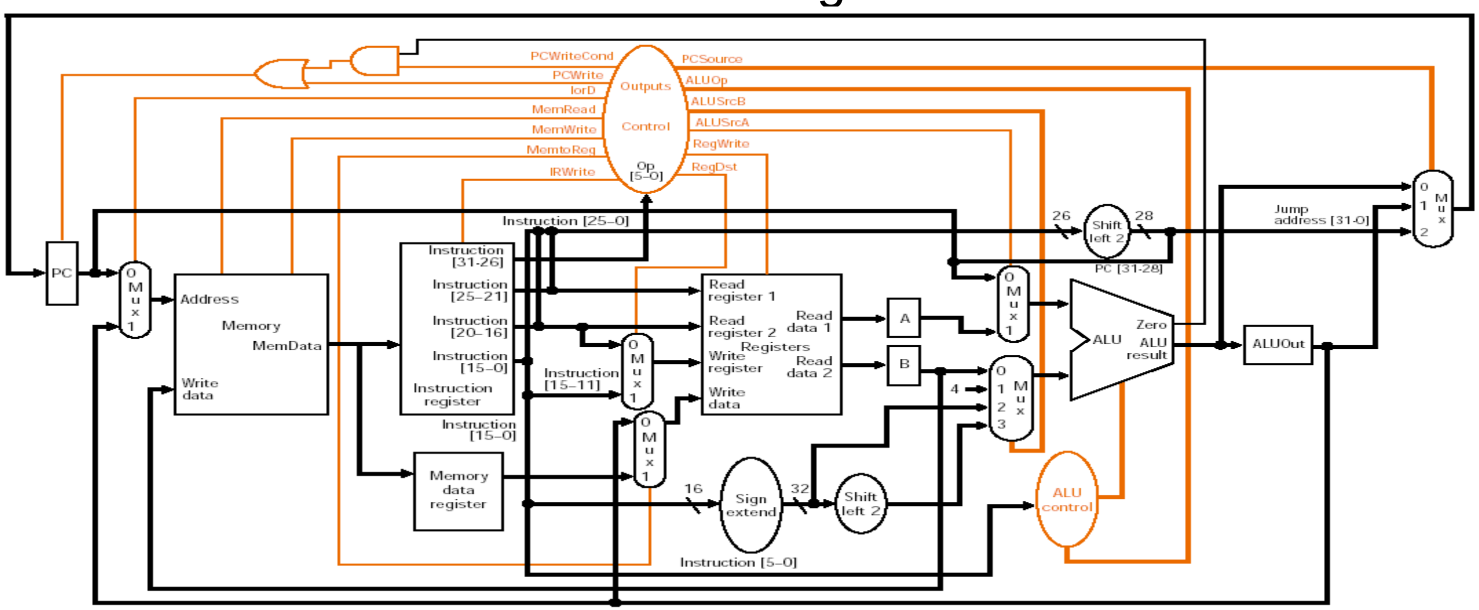


Multicycle Approach

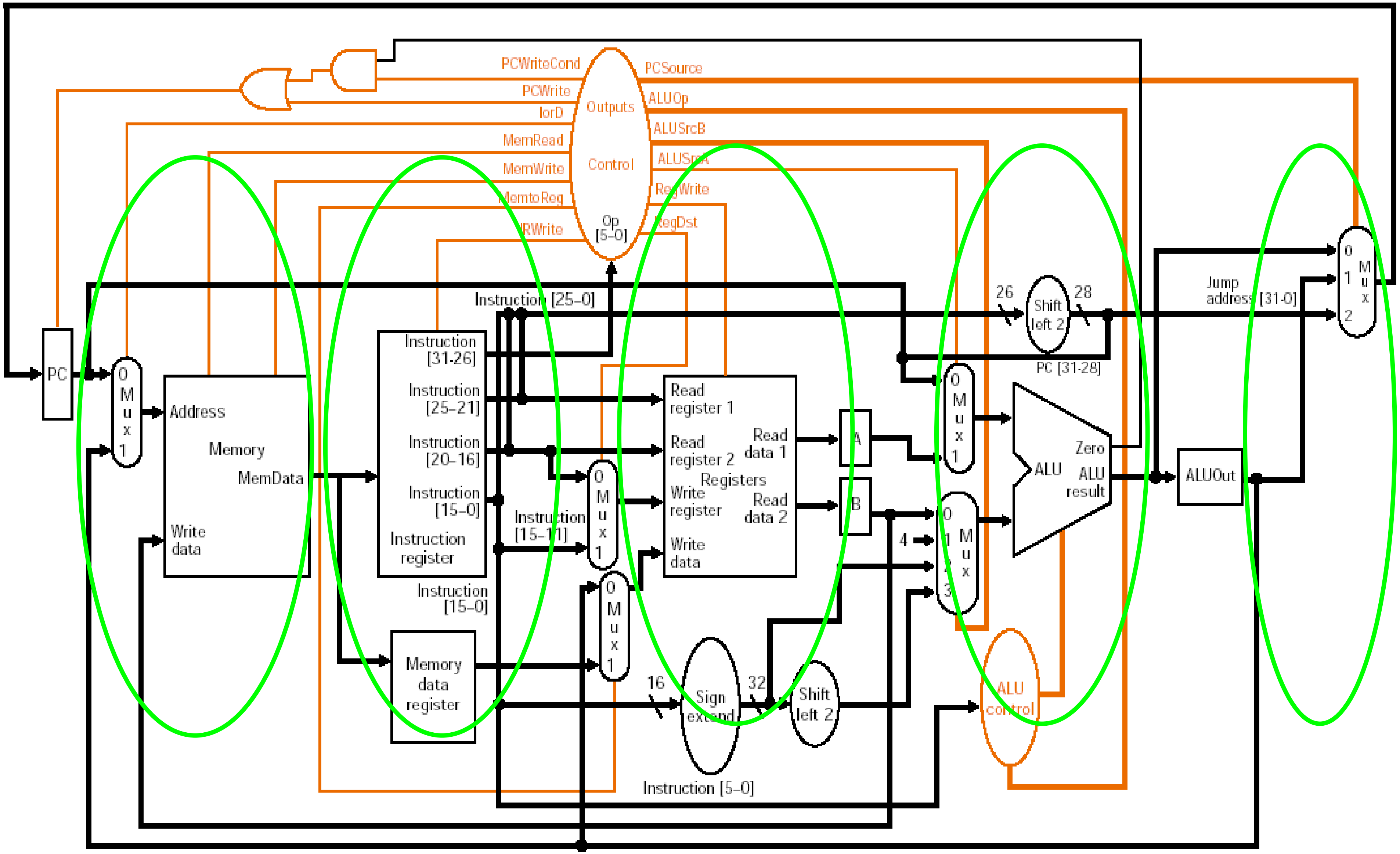
- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a subtract instruction?
- We will use a finite state machine for control

Multicycle 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



The multi cycle datapath



Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- 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;
```
(speculative computing)

- *Can we figure out the values of the control signals?*

  What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Generate the control signals based on the value in the instruction register.
- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

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

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

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

- R-type:

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

- Branch: (complete in this step)

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

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
    MDR = Memory[ALUOut]; (load)
    or
    Memory[ALUOut] = B; (store is
completed)
```

- R-type instructions finish

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

The write actually takes place at the end of the cycle on the edge

Write-back step

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

What about all the other instructions?

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		

Comparing Single-cycle and Multi-cycle impmentations

- Single Cycle: IF + ID/RF + EX/BR/J + MEM + WB
  - Cycle time: 7 ns (average)
  - CPI: 1 for all instructions
- Multi Cycle: IF -> ID/RF -> EX/BR/J -> MEM -> WB
  - Cycle time: 2 ns
  - CPI
    - R-type: 4
    - Memory (load): 5
    - Memory (store): 4
    - Branch/Jump: 3

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

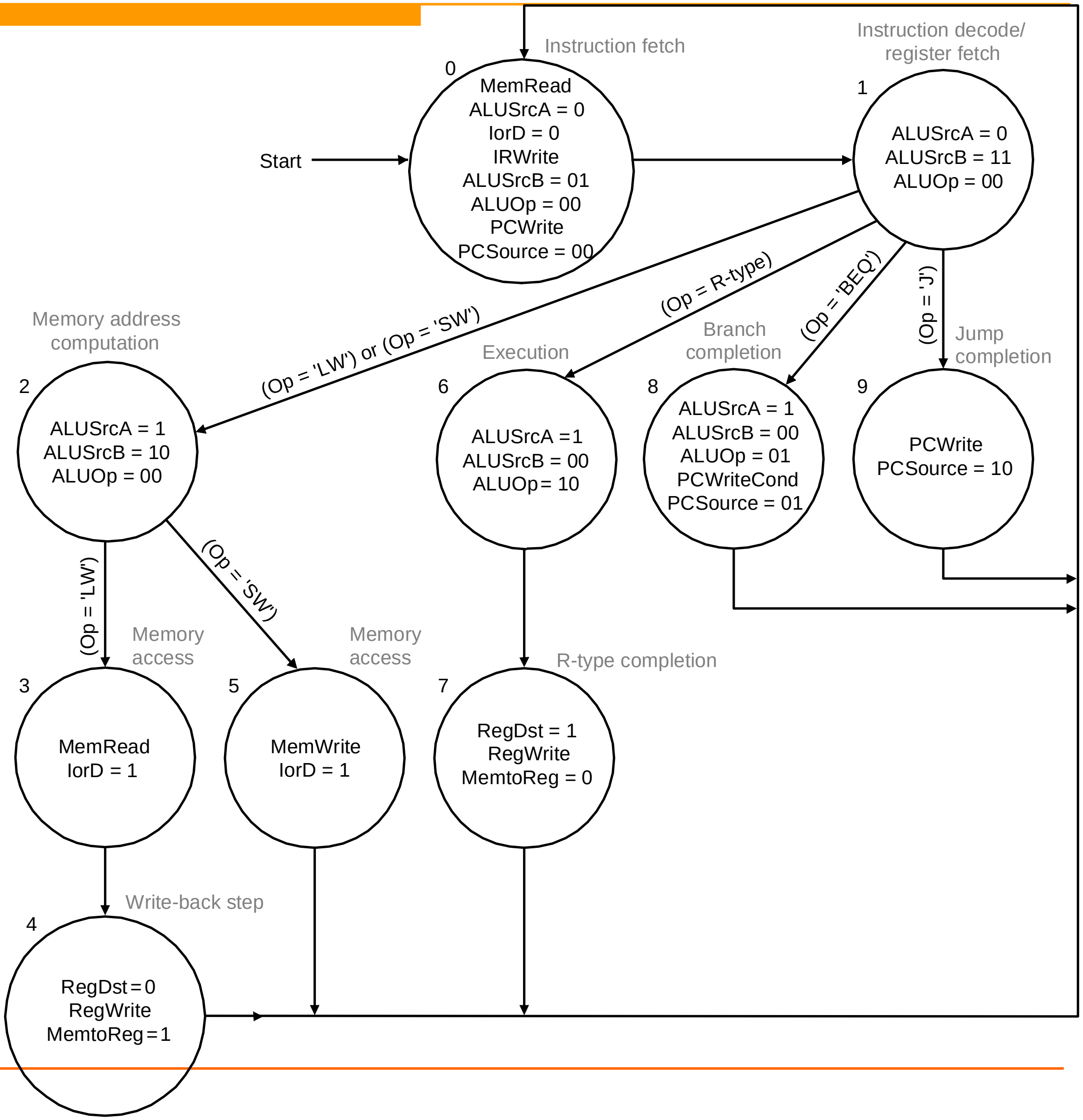
- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we have accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived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

Implementing the Control

- Register update:
  - By the clock signal alone
    - For data with life span of one clock cycle (A, B, ALUout, EPC, Cause)
    - simpler, no control signal, useless updates
  - By both the clock signal and explicit enable signals…need explicit control signals
    - For data with life span of more than one clock cycles (IR, PC, MDR)
    - Need explicit control logics and signal pins; no useless update

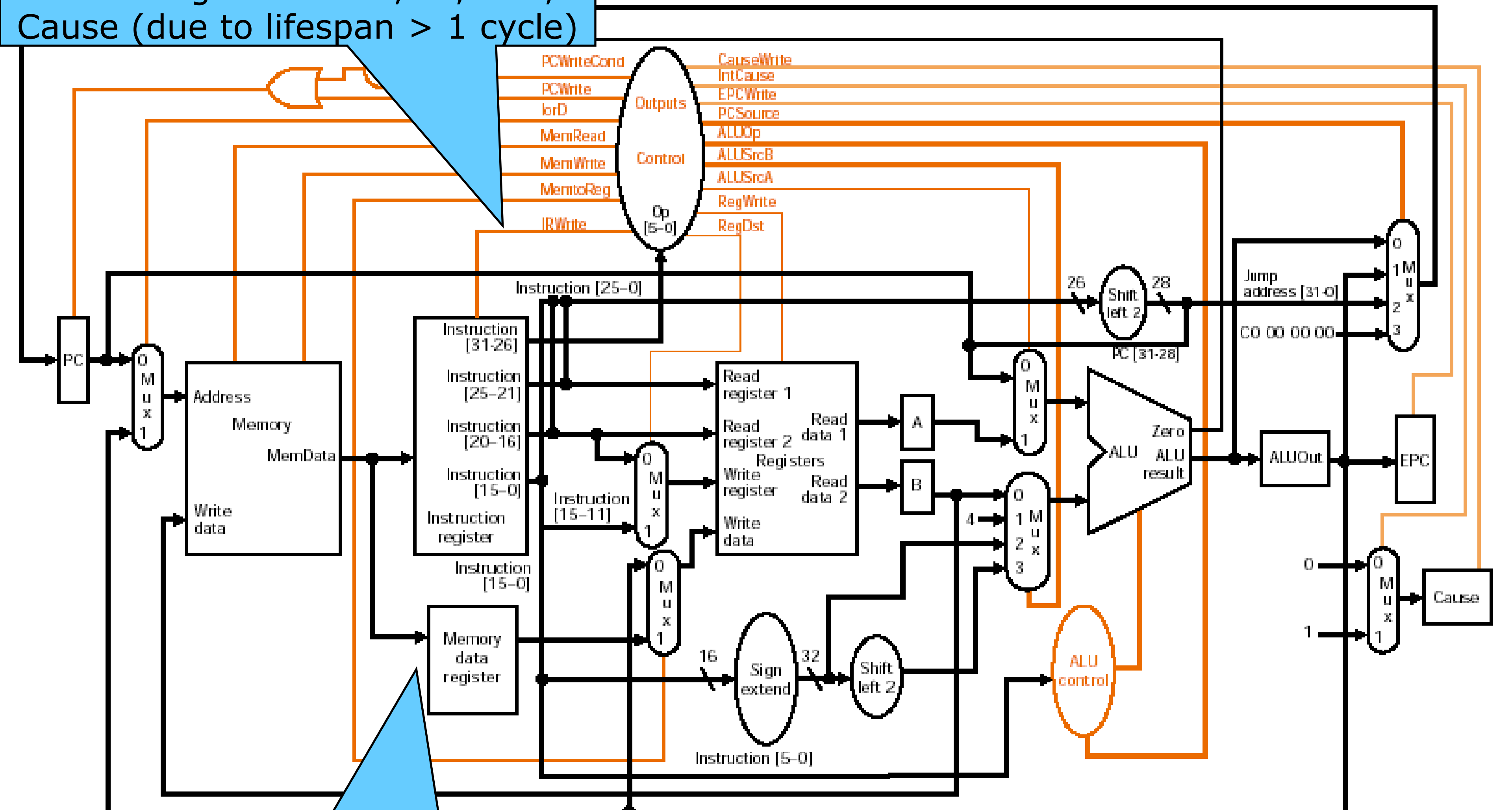
Graphical Specification of FSM

- How many state bits will we need?



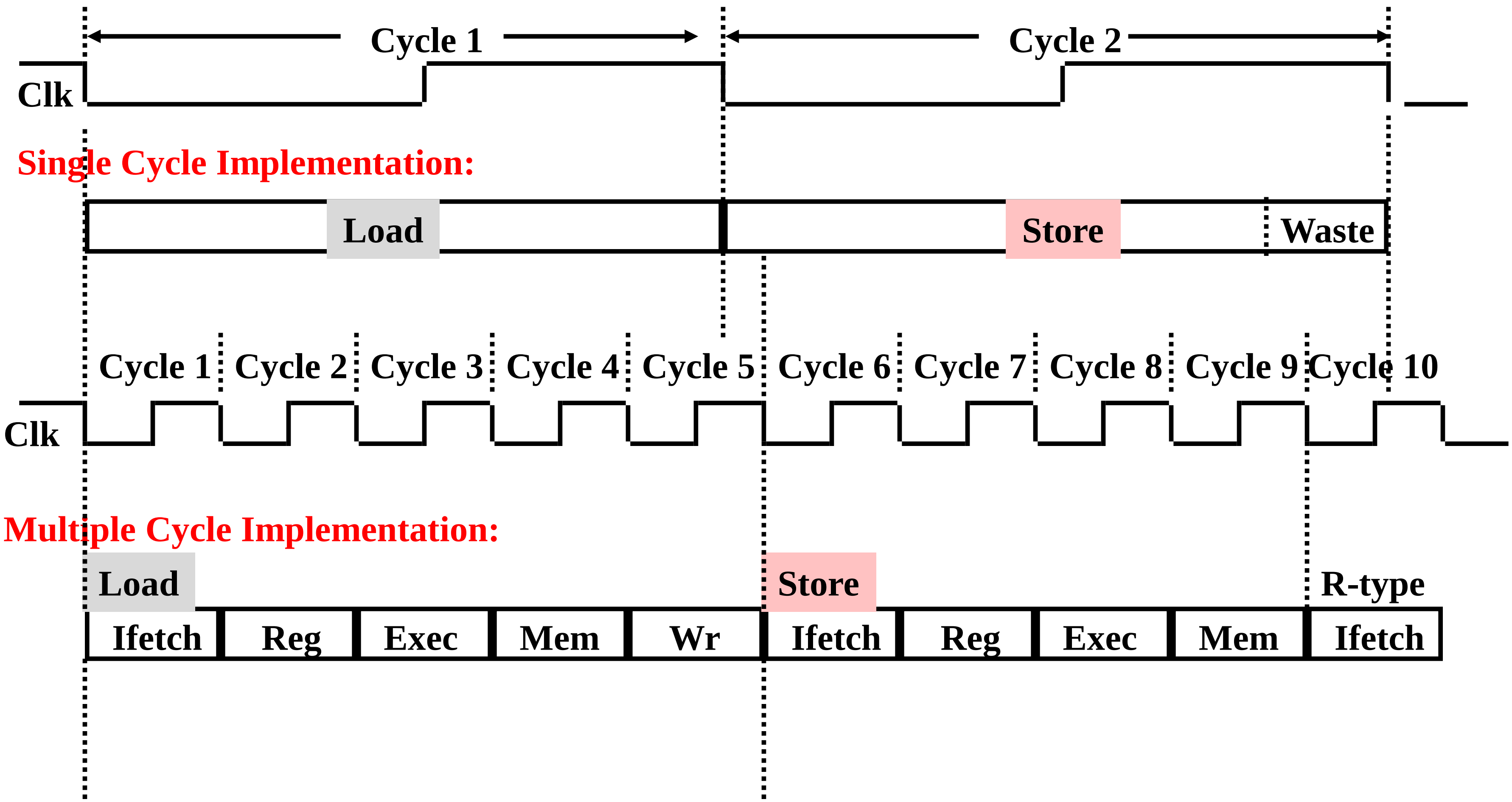
Final data path and control path: multi-cycle

Enable signals for PC, IR, EPC, Cause (due to lifespan > 1 cycle)



No enable signals for MDR, A, B, ALUout

Single Cycle vs. Multiple Cycle



Enjoy !!!

Q&A