SoC Design : Lecture 4

Prof. Jong-Myon Kim

울산대학교
UNIVERSITY OF ULSAN

Contents

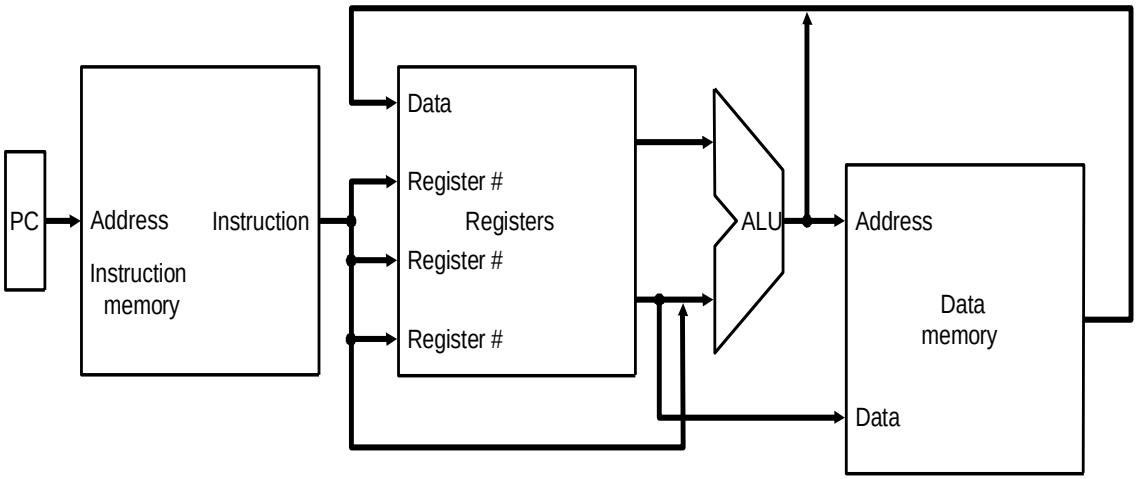
- Single Cycle Datapath Architecture

The Processor: Datapath and Control

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers
  Why? memory-reference? arithmetic? control flow?

More Implementation Details

- Abstract / Simplified View:

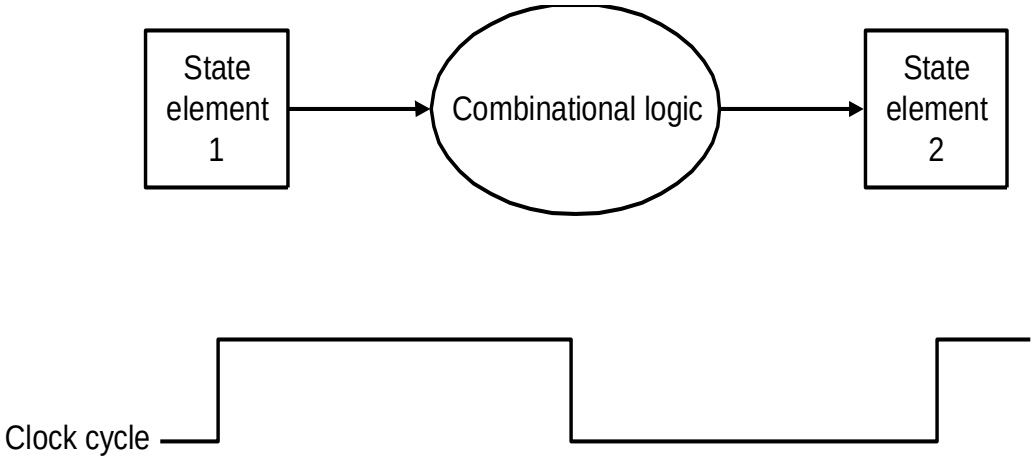


Two types of functional units:

- Elements that operate on data values (combinational)
- Elements that contain state (sequential)

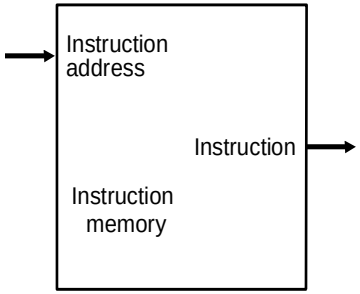
Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements

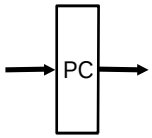


Simple Implementation

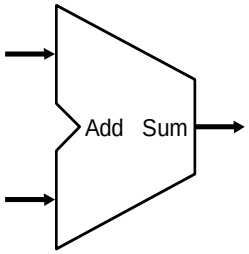
- Include the functional units we need for each instruction



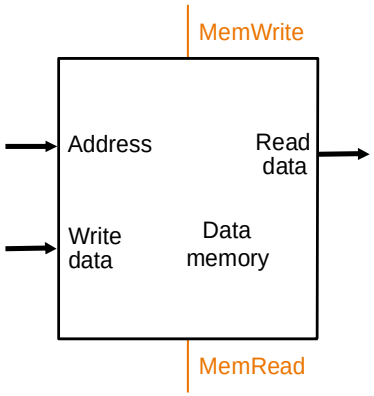
a. Instruction memory



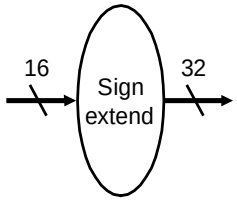
b. Program counter



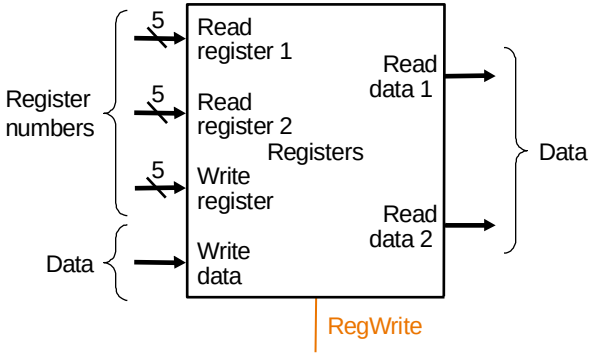
c. Adder



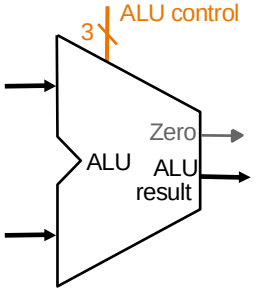
a. Data memory unit



b. Sign-extension unit



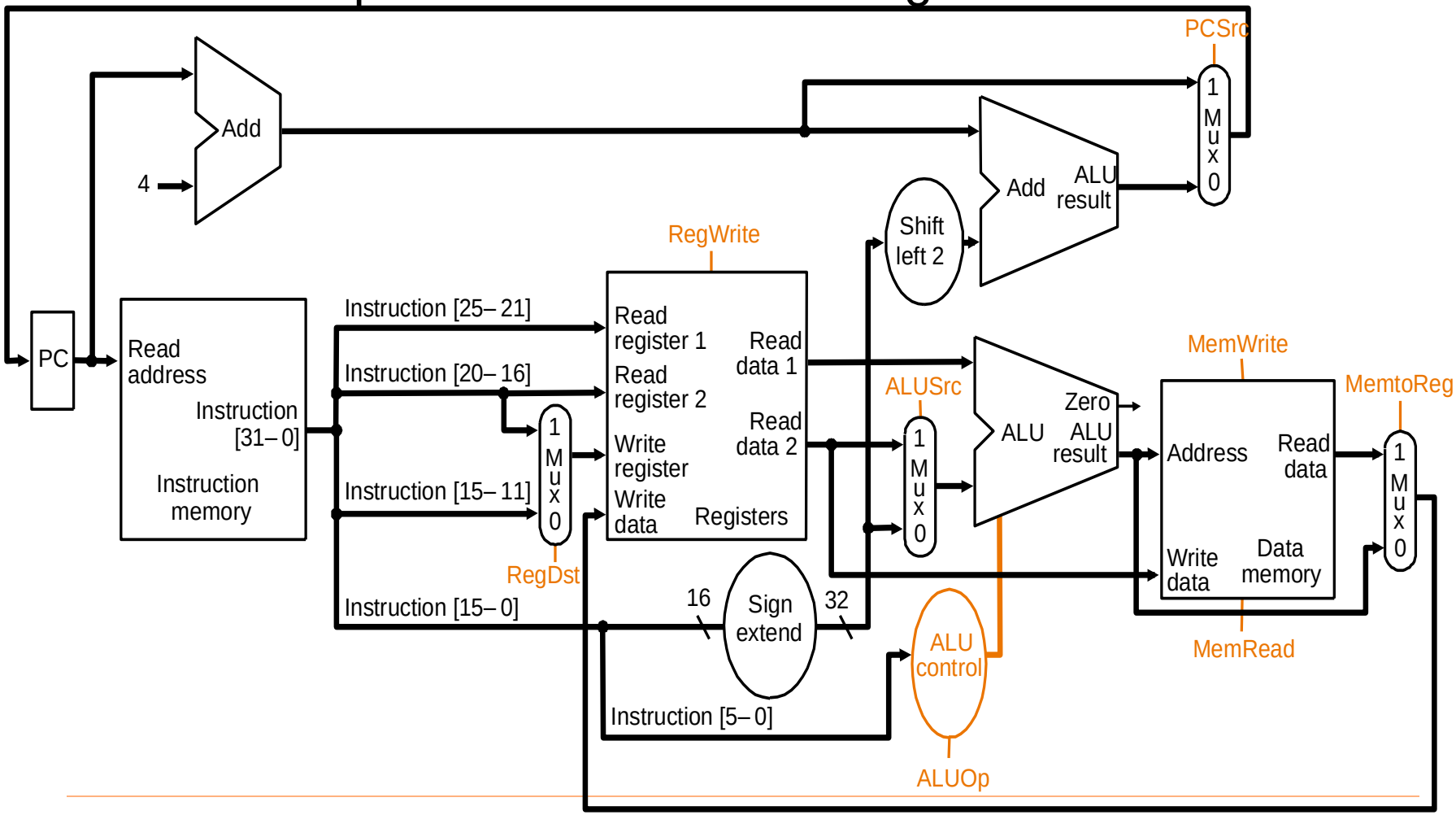
a. Registers



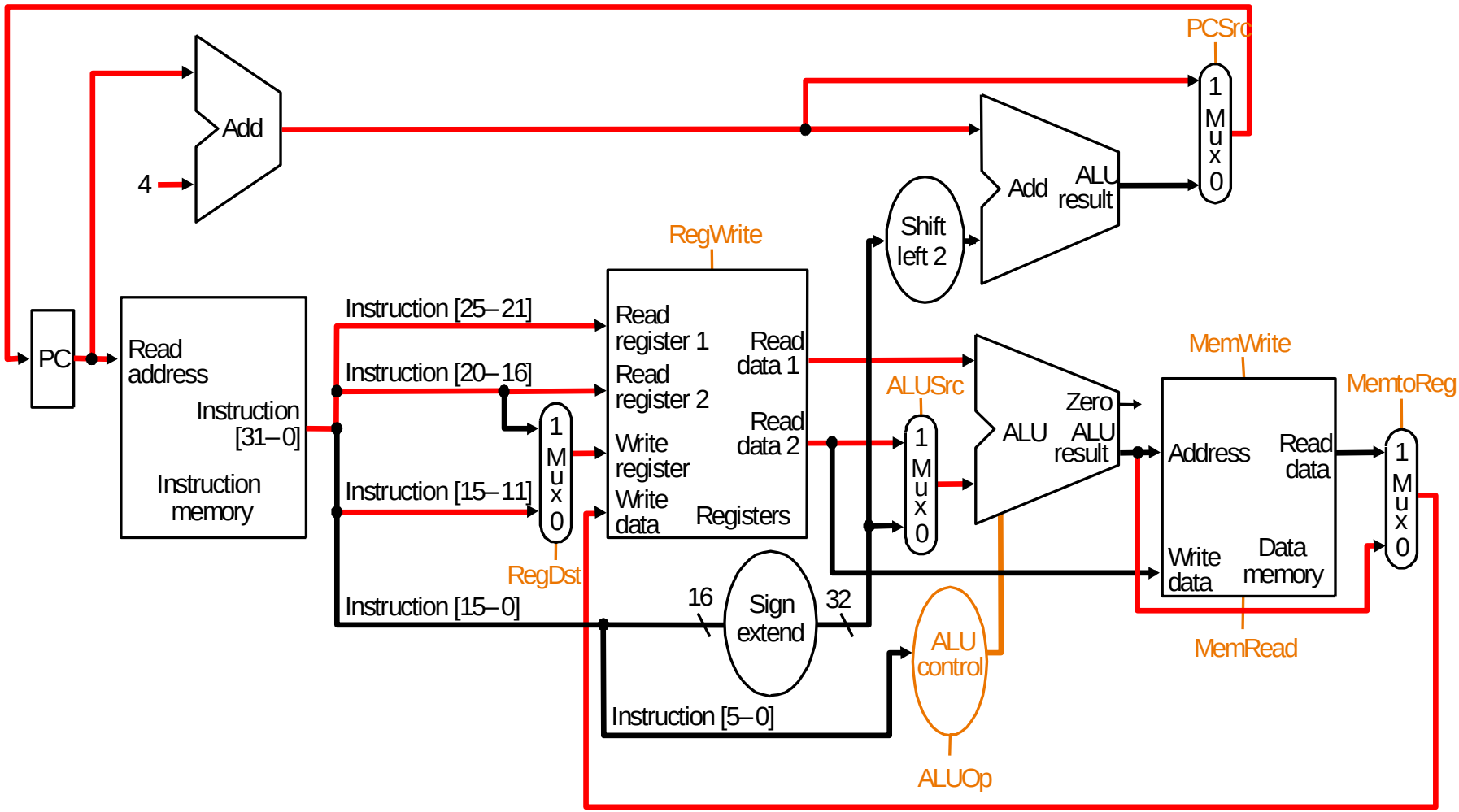
b. ALU

Building the Datapath

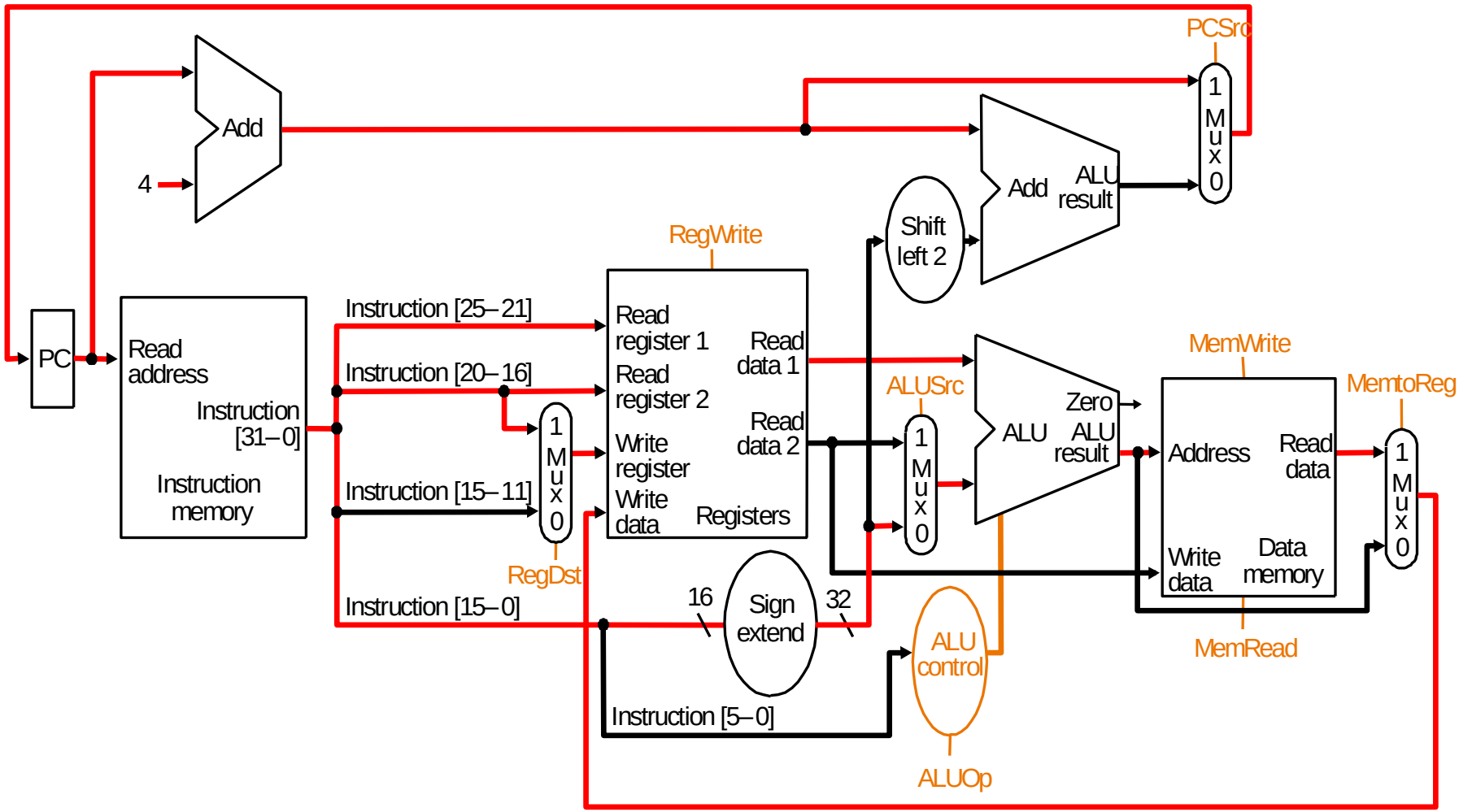
- Use multiplexers to stitch them together



R-Type Instructions
(e.g. add $2, $3, $4; Not JR/JALR)

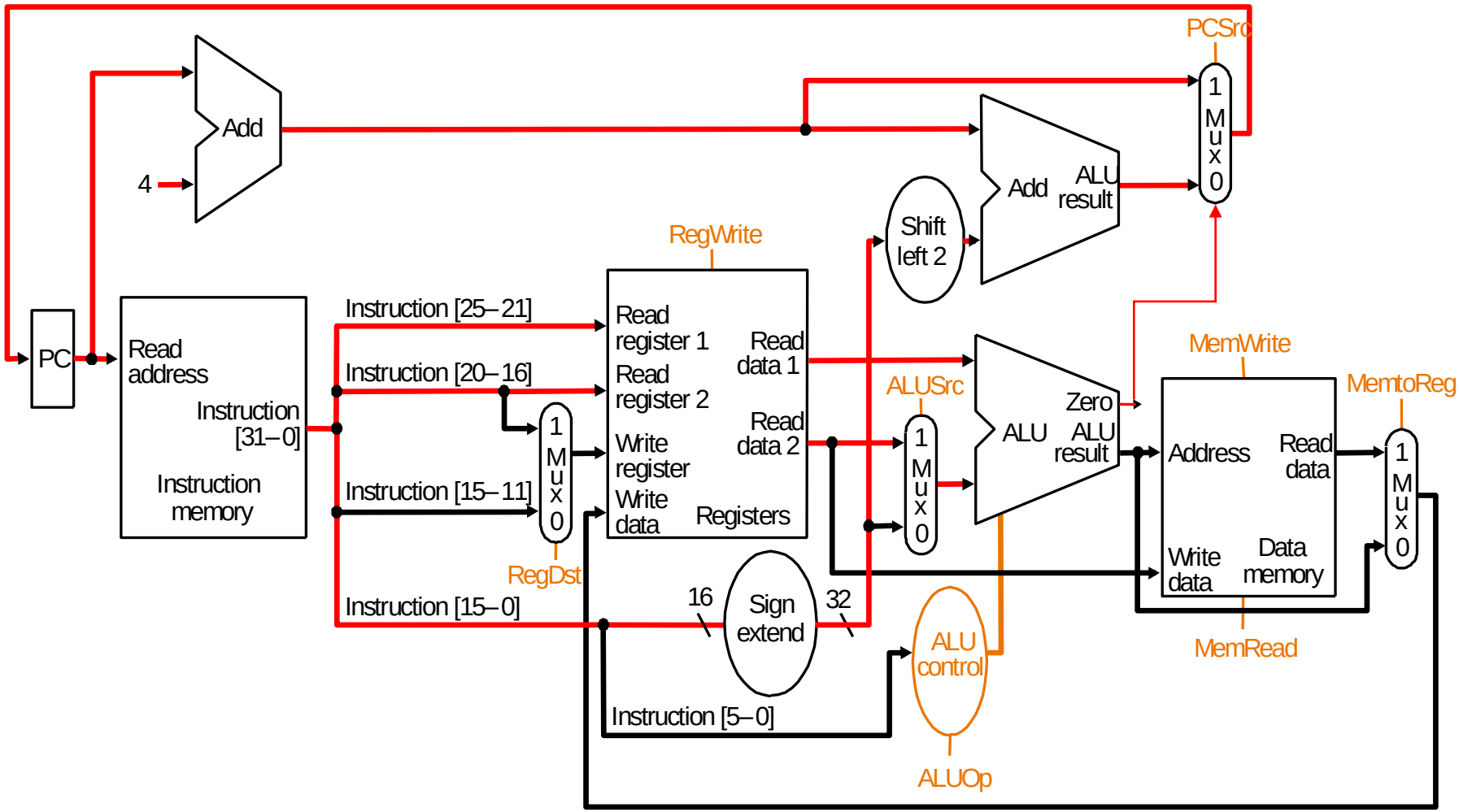


I-Type Instructions
(e.g. lw $4, 1000($15))



I-type Instruction for Branches

(e.g. beq $4, $5, Label7)



Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexer inputs)
- Information comes from the 32 bits of the instruction
- Example:

add $8, $17, $18 **Instruction Format:**

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- ALU's operation based on instruction type and function code

Control

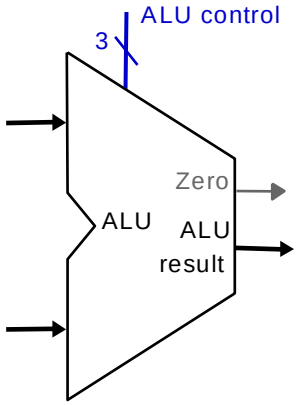
- e.g., what should the ALU do with this instruction
- Example: lw $1, 100($2)

35	2	1	100

op	rs	rt	16 bit offset

- ALU control input

```
000     AND
001     OR
010     add
110     subtract
111     set-on-less-than
```



- Why is the code for subtract 110 and not 011? What do you need for slt instruction?

Control the ALU

- Must describe hardware to compute 3-bit ALU control input
  - given instruction type
    00 = lw, sw
    01 = beq,
    11 = arithmetic (incl. slt)
  - function code for arithmetic

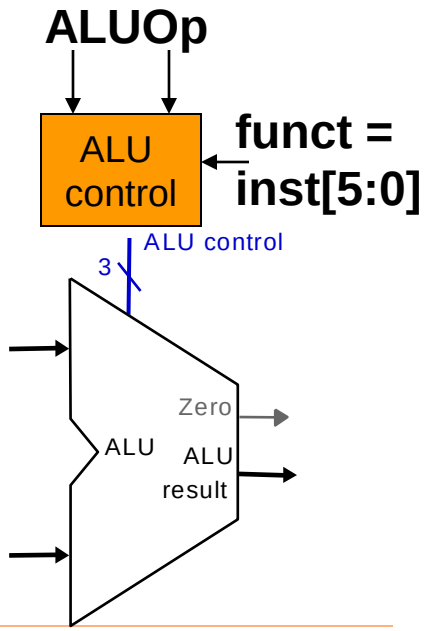
ALUOp
computed from instruction type

- Describe it using a truth table (can turn into gates):

	ALUOp		Funct field						ALU Control	
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
lw/sw	0	0	X	X	X	X	X	X	010	add
beq	X	1	X	X	X	X	X	X	110	sub
arith	1	X	X	X	0	0	0	0	010	add
	1	X	X	X	0	0	1	0	110	sub
	1	X	X	X	0	1	0	0	000	and
	1	X	X	X	0	1	0	1	001	or
	1	X	X	X	1	0	1	0	111	slt

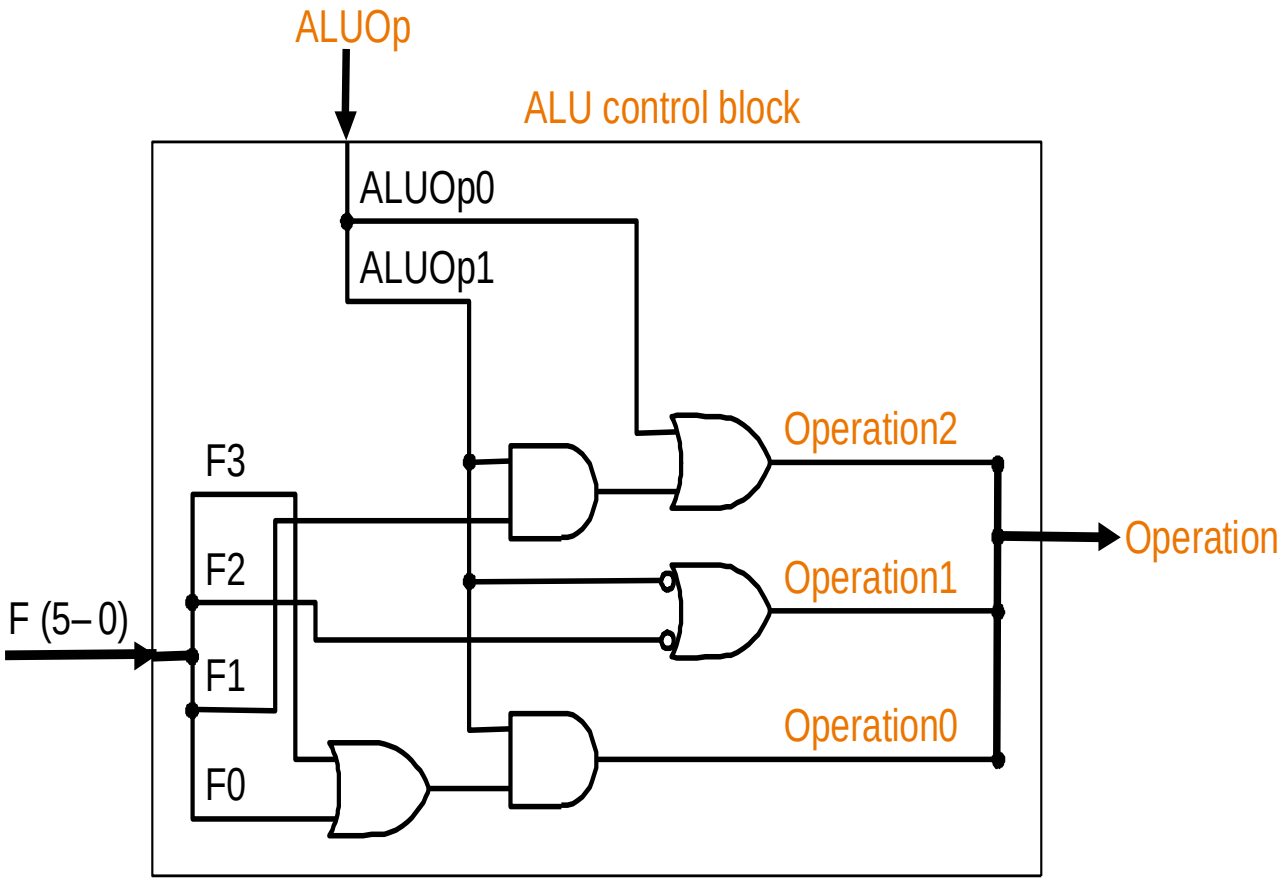
Generated from Decoding inst[31:26]

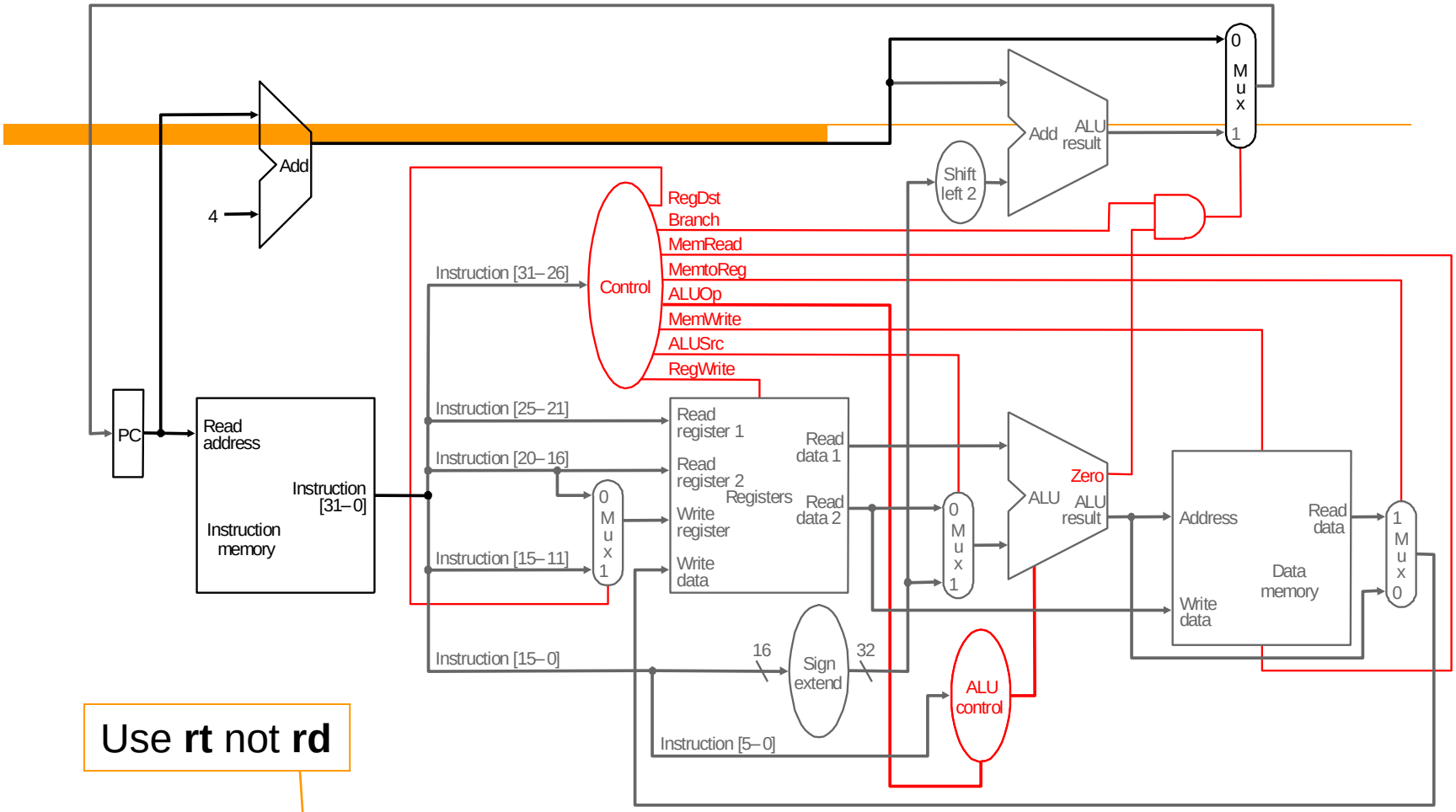
inst[5:0]



ALU Control

- Simple combinational logic (truth tables)



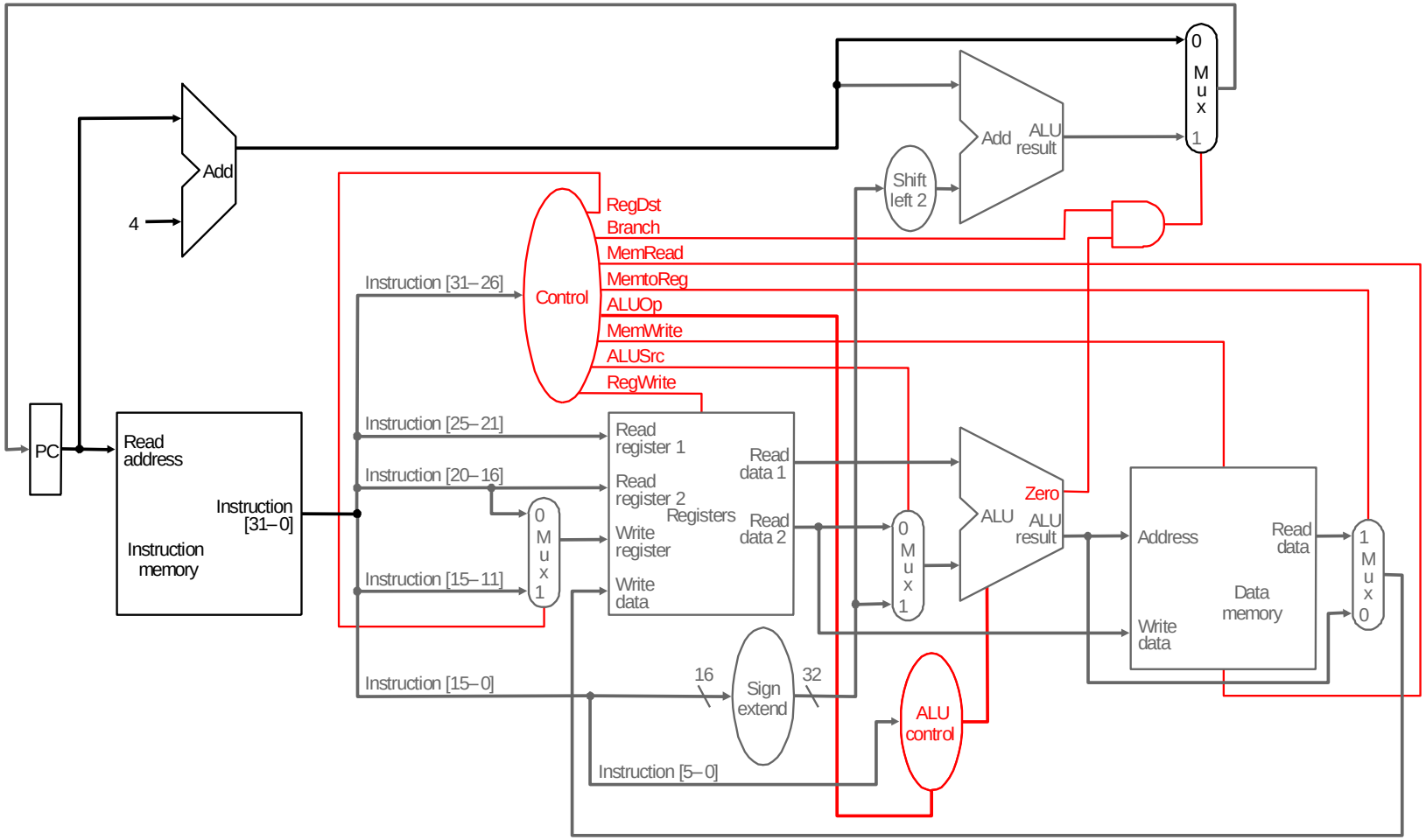


Use **rt** not **rd**

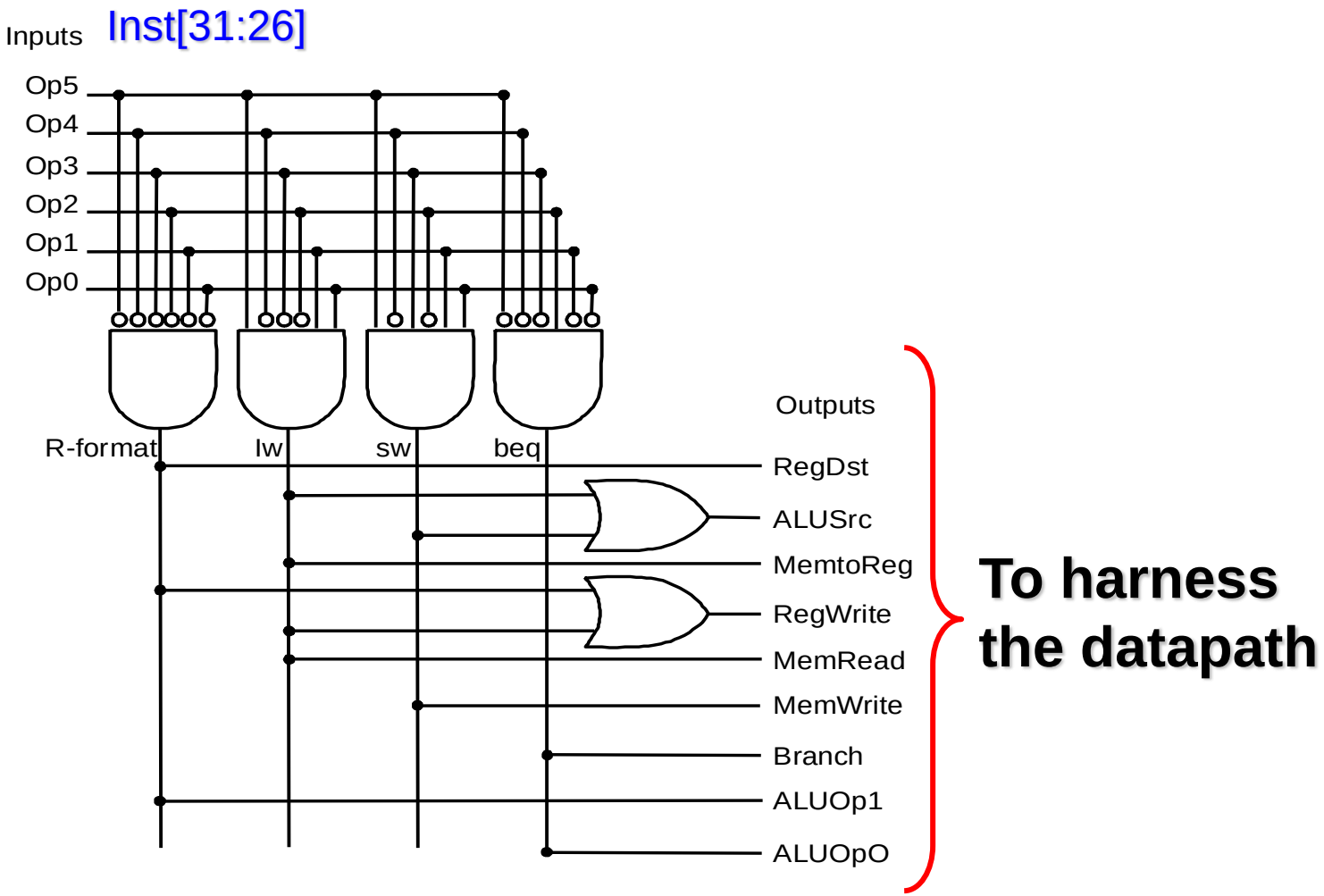
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Use **rt** not **rd**



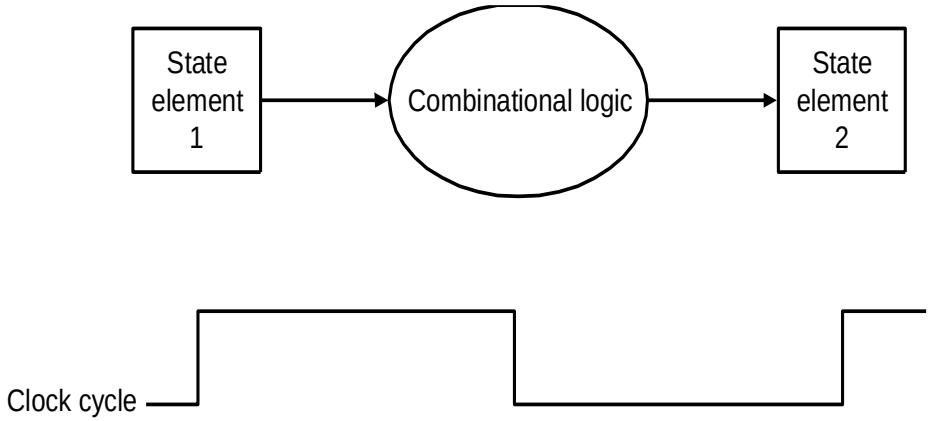
Control Unit Signals



To harness the datapath

Our Simple Control Structure

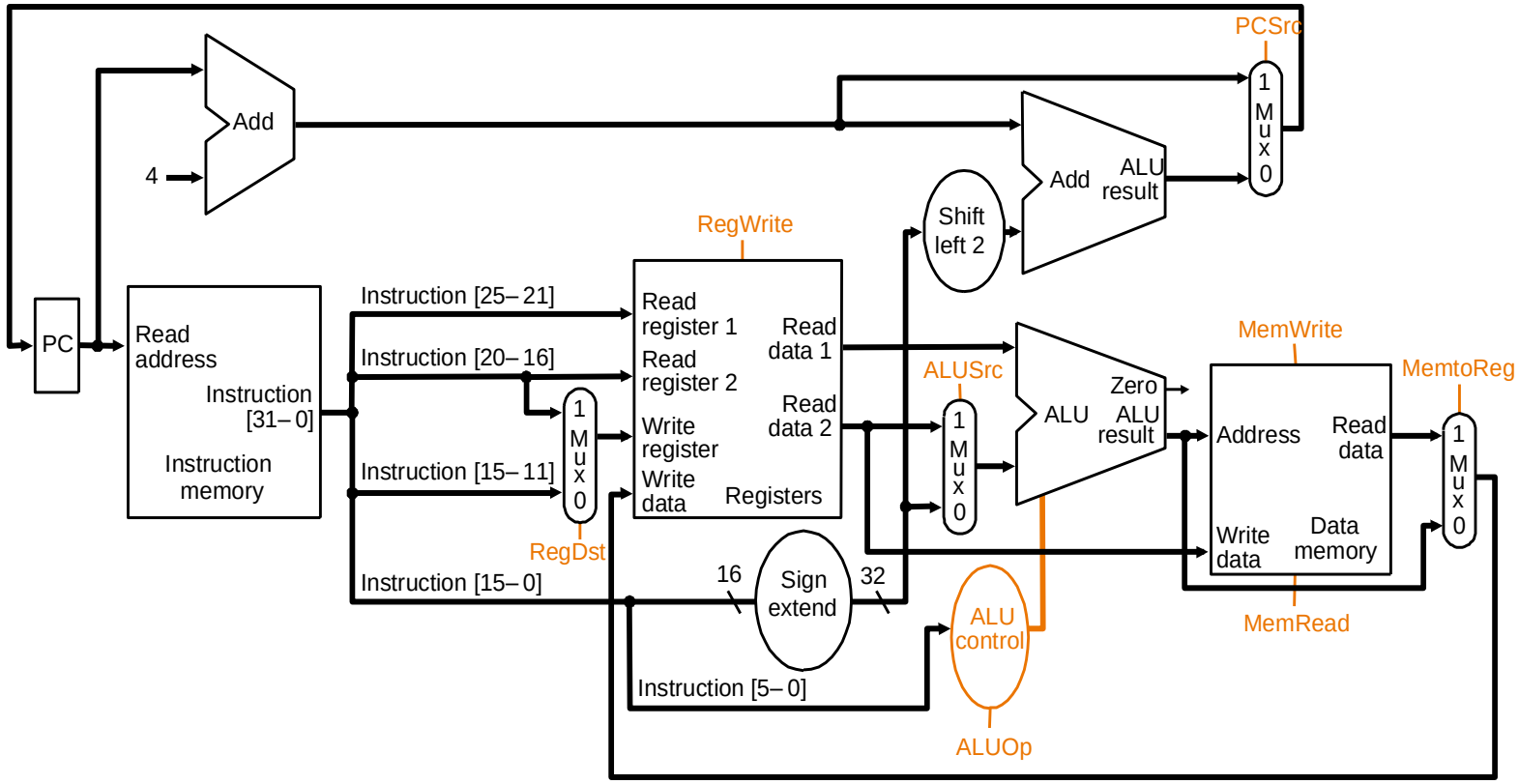
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

Single Cycle Implementation

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



Single Cycle Implementation (Sol)

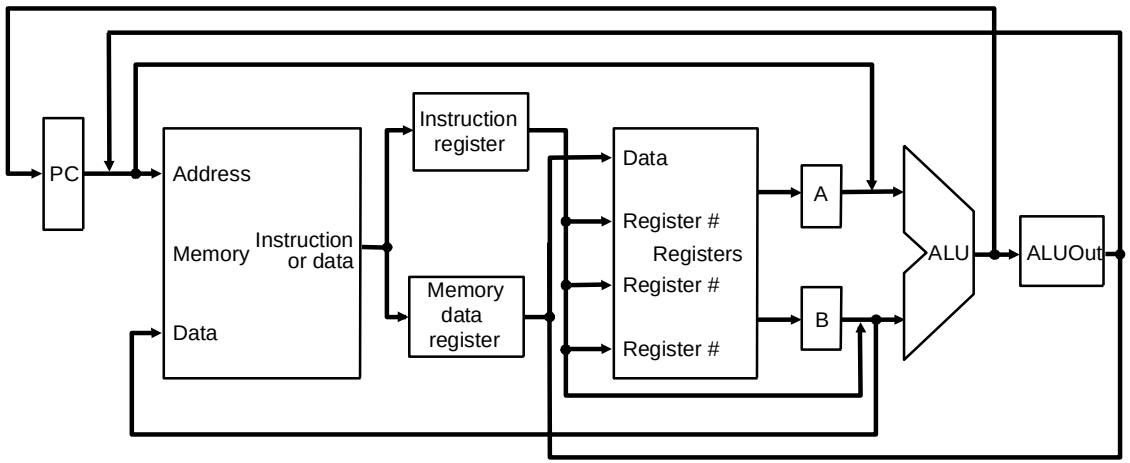
- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write
R-type	2	1	2	0	1
Load word	2	1	2	2	1
Store word	2	1	2	2	
Branch	2	1	2		
Jump	2				

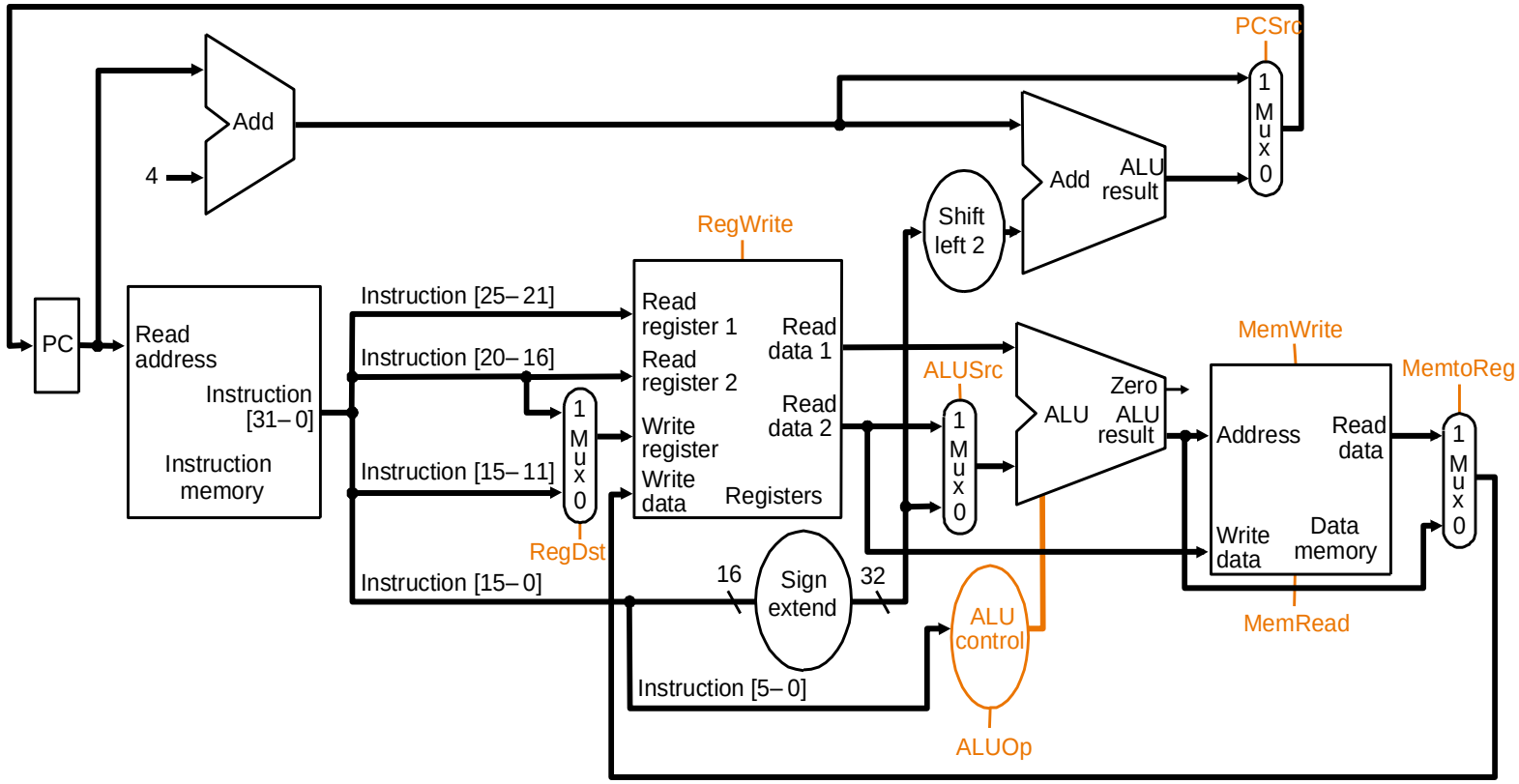
Where we are headed

- 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:



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)