



# **Chapter 4**

## **The Processor**

# In this chapter,

- We will examine two MIPS implementations in logic gate level.
  - first, a simplified version
  - second, a more realistic pipelined version
- Our implementation will execute a simple subset of MIPS instructions.
  - Memory reference instructions: lw, sw
  - Arithmetic/logical instructions: add, sub, and, or, slt
  - Branch instructions: beq, j

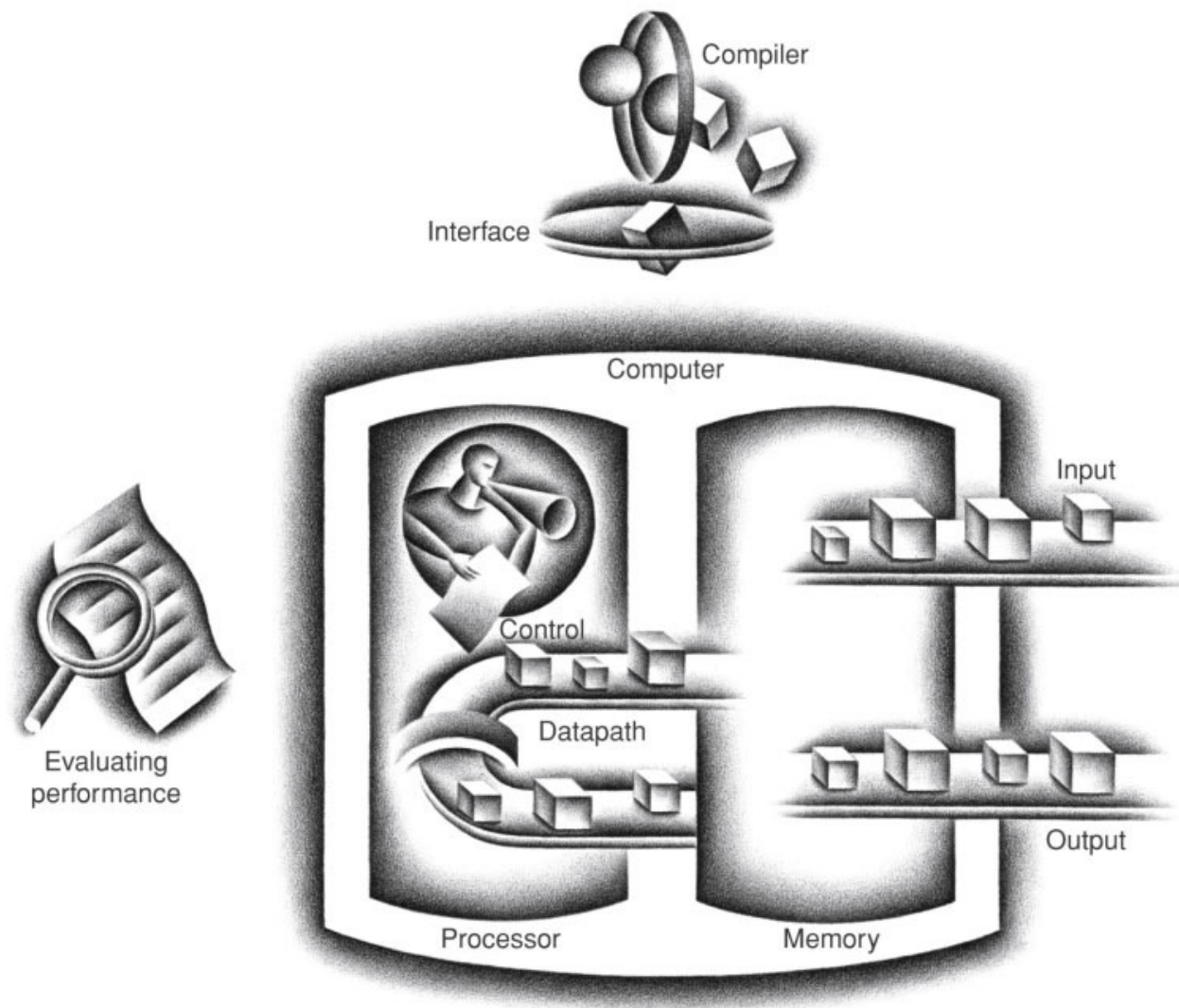
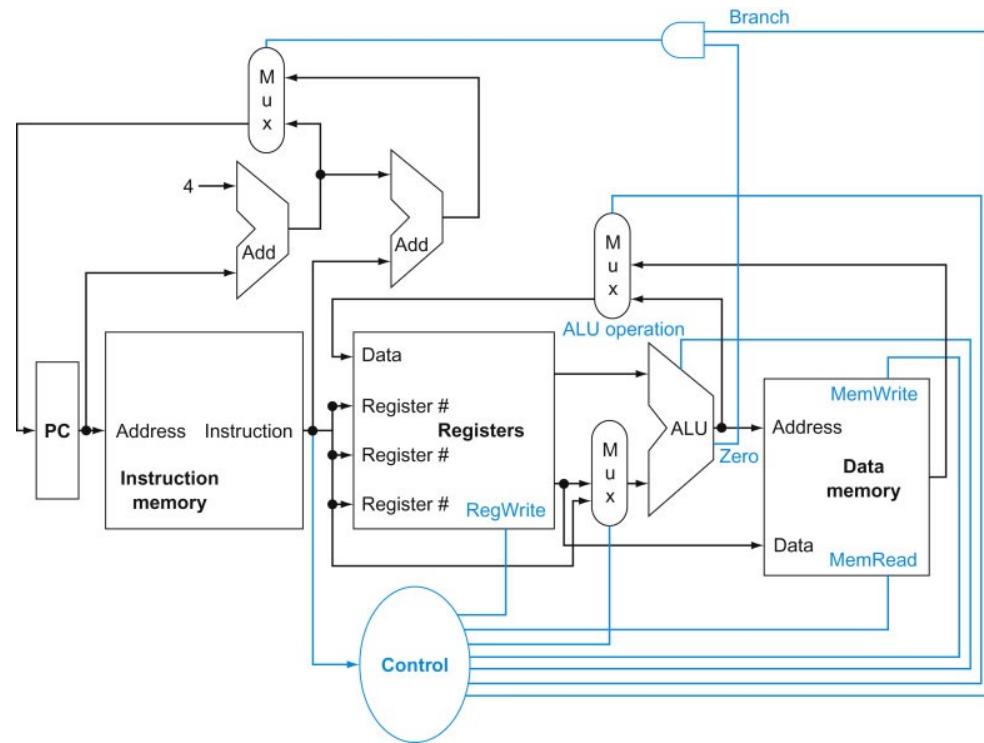
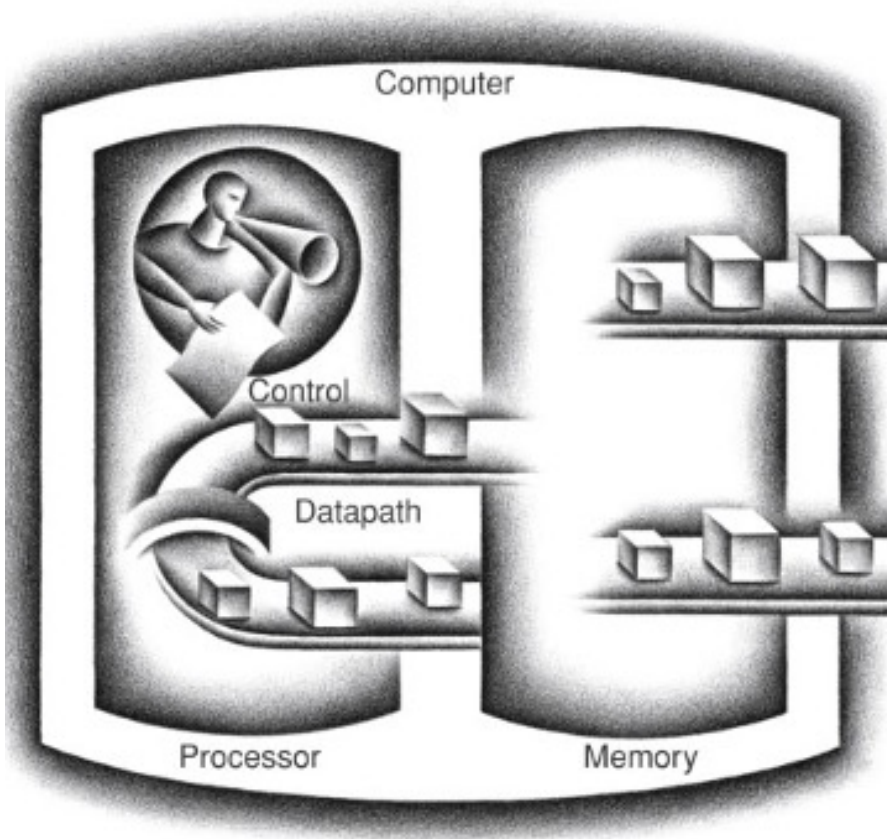


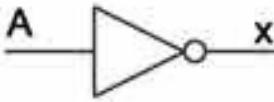



FIGURE 1.5 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.



# Logic Gates

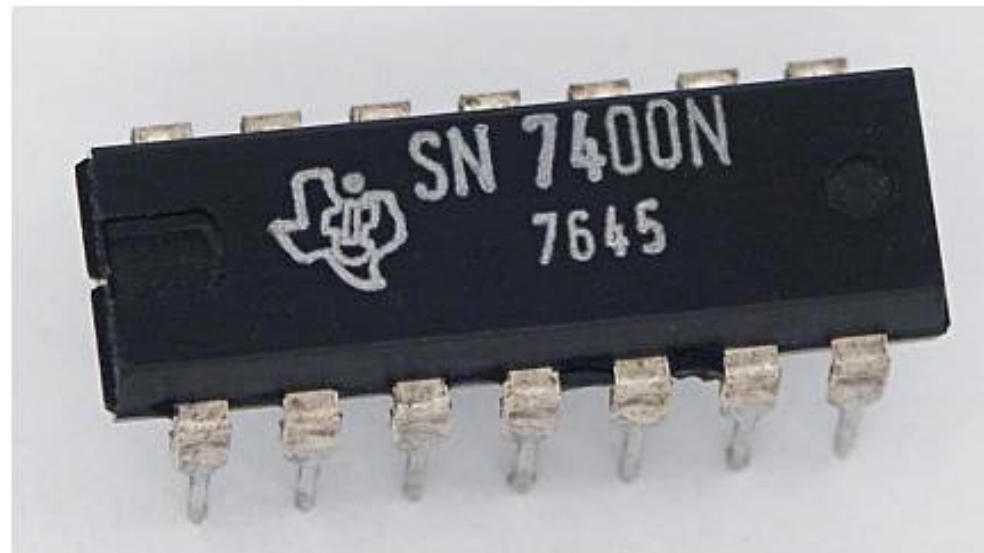
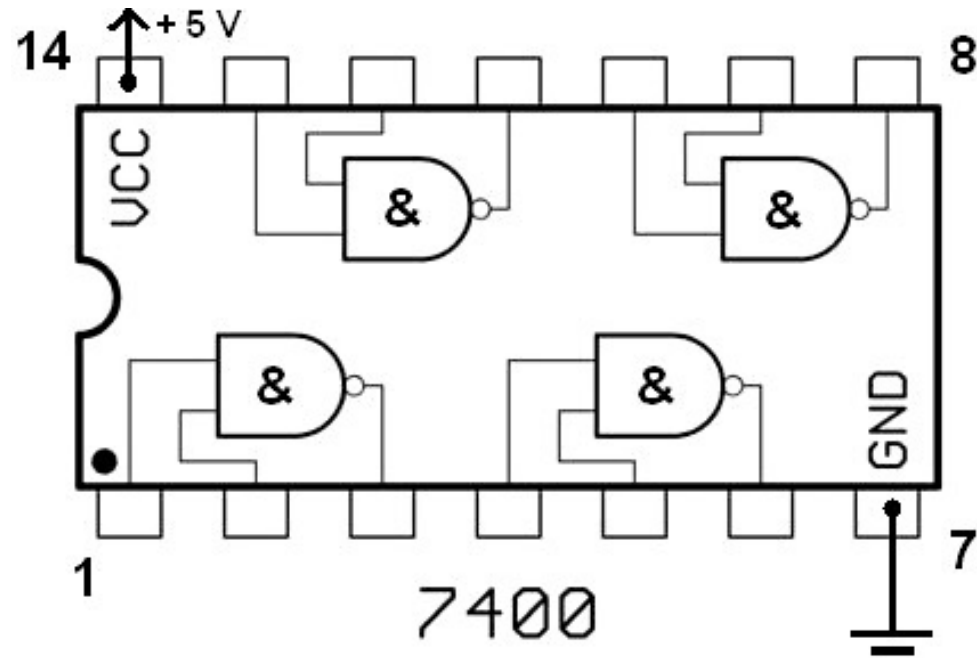
논리곱  $\wedge$

논리합  $\vee$

Name	NOT	AND	NAND	OR																																																			
Alg. Expr.	$\overline{A} = A'$	$AB$	$\overline{AB}$	$A + B$																																																			
Symbol																																																							
Truth Table	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	1
A	X																																																						
0	1																																																						
1	0																																																						
B	A	X																																																					
0	0	0																																																					
0	1	0																																																					
1	0	0																																																					
1	1	1																																																					
B	A	X																																																					
0	0	1																																																					
0	1	1																																																					
1	0	1																																																					
1	1	0																																																					
B	A	X																																																					
0	0	0																																																					
0	1	1																																																					
1	0	1																																																					
1	1	1																																																					

# NAND gate

---



# Logic Design Basics (Appendix B)

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element (조합회로)
  - 입력 데이터를 연산해서 새로운 값을 만드는데 사용된다.
  - $\text{Output} = f(\text{current input})$
- State (sequential) elements (순차회로)
  - 연산된 데이터를 저장해 두었다가 나중에 읽어내기 위해 사용된다.
  - $\text{Output} = f(\text{current input, previous input})$

# 다음의 진리표를 논리회로로 구현하라.

simplest example : 2 bit inputs A and B, 2 bit outputs S and C

→ 각 output 을 논리식 (sum of products) 으로 표현

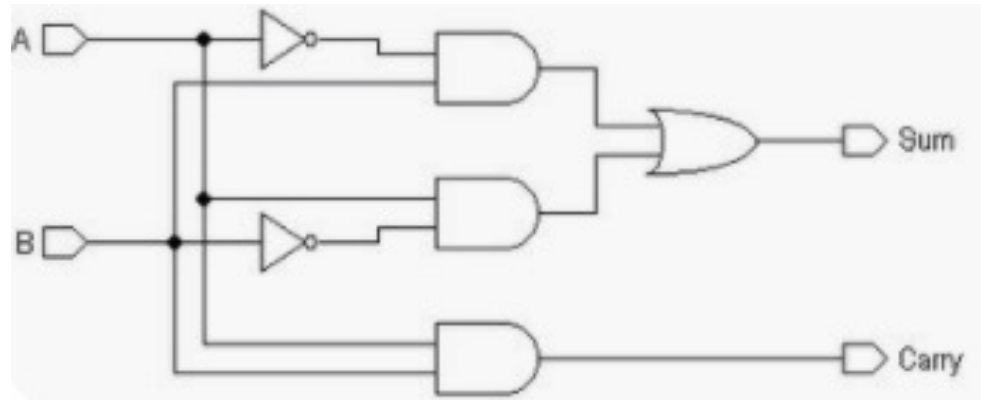
	input		output	
	A	B	S	C
A'B'	0	0	0	0
A'B	0	1	1	0
AB'	1	0	1	0
AB	1	1	0	1

$$S = A'B + AB'$$

$$C = AB$$

or 연산 (sum)

→ 논리식을 논리 게이트로 표현



not 연산  
and 연산

A=0 이고 B=0 일 때만  
1이 되는 논리곱(product)



# 1 bits half adder ( CarryIn 이 없음)

A	0	0	1	1
B	+0	+1	+0	+1
	0	1	1	0 (carry 1)

→ 각 output 을 논리식 (sum of products) 으로 표현

input		output	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$A'B'$

$A'B$

$AB'$

$AB$

not 연산  
and 연산

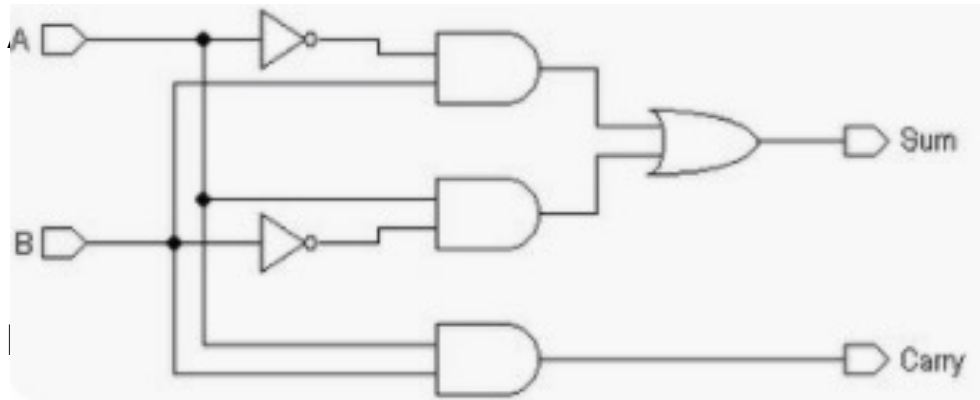
$$S = A'B + AB'$$

$$C = AB$$

or 연산 (sum)

→ 논리식을 논리 게이트로 표현

A=0 이고 B=0 일 때만  
1이 되는 논리곱(product)



# 모든 진리표는 Combinational circuit 으로 구현될 수 있다.

- 1-bit full-adder

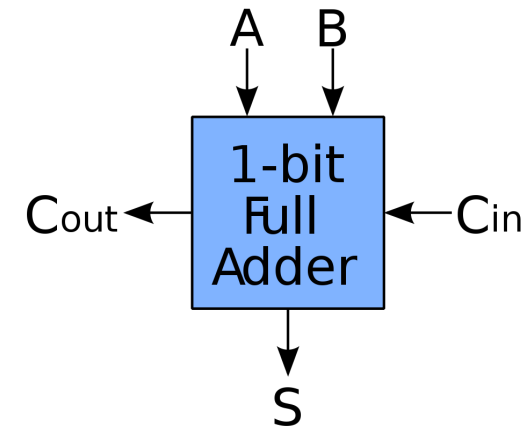
진리표

→ 각 output 의 논리식 → 논리식을 논리 게이트로 표현

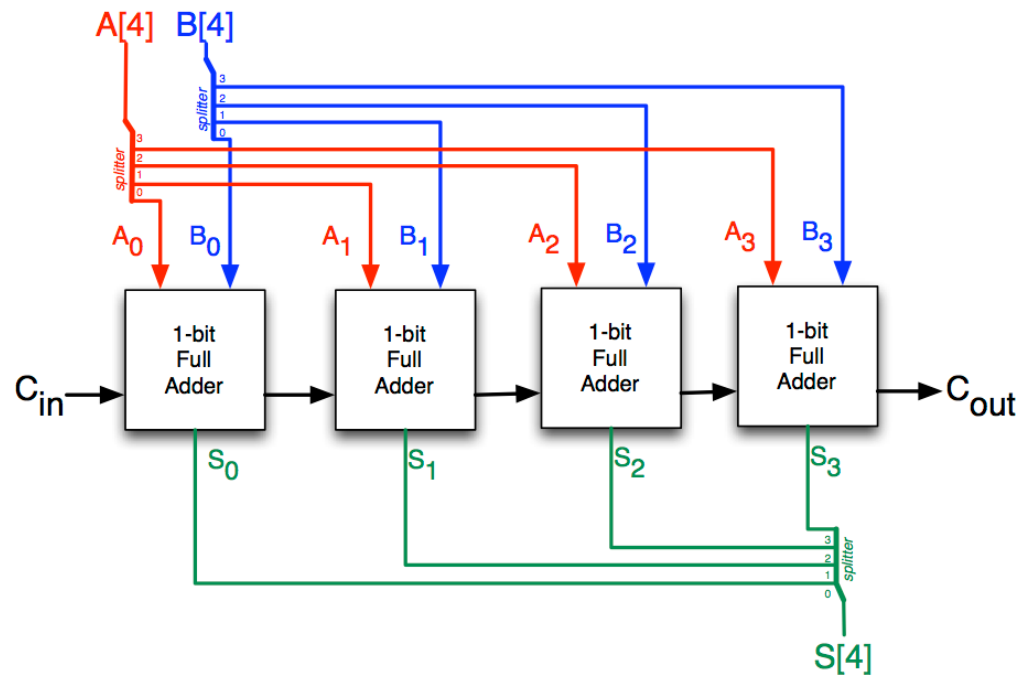
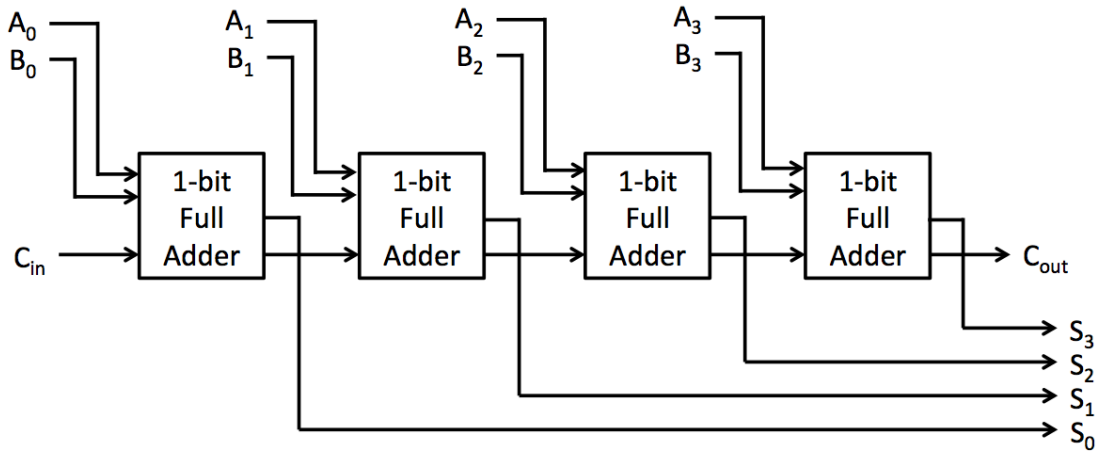
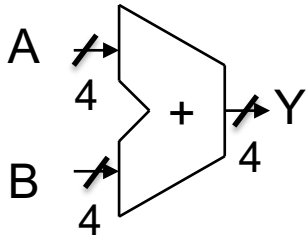
	Inputs			Outputs	
	A	B	C <sub>in</sub>	C <sub>out</sub>	S
A'B'C <sub>in</sub> '	0	0	0	0	0
A'B'C <sub>in</sub>	0	0	1	0	1
A'BC <sub>in</sub> '	0	1	0	0	1
A'BC <sub>in</sub>	0	1	1	1	0
AB'C <sub>in</sub> '	1	0	0	0	1
AB'C <sub>in</sub>	1	0	1	1	0
ABC <sub>in</sub> '	1	1	0	1	0
ABC <sub>in</sub>	1	1	1	1	1

$$C_{out} = A'BC_{in} + AB'C_{in} + ABC_{in}' + ABC_{in}$$

$$S = A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$$



# 4-bit adder

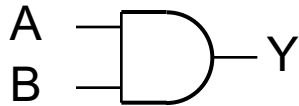


# Combinational Elements

Output =  $f$ (current input)

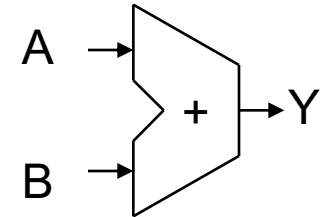
- AND-gate

- $Y = A \wedge B$



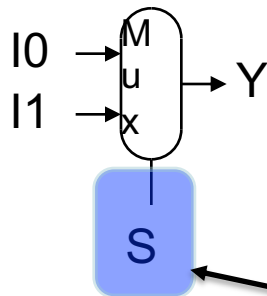
- Adder

- $Y = A + B$



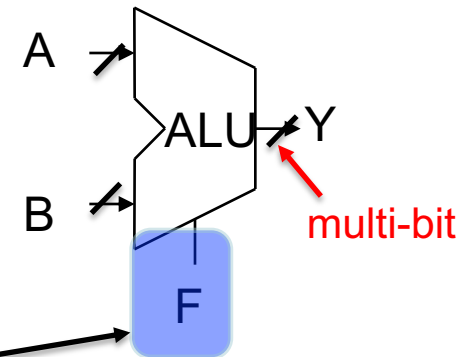
- Multiplexer

- $Y = S ? I1 : I0$



- Arithmetic/Logic Unit

- $Y = F(A, B)$



control input

# Combinational circuit 은 진리표만 주어지면 만들 수 있다.

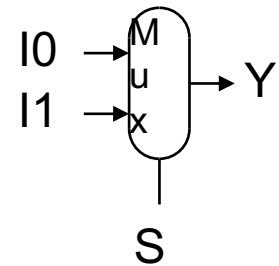
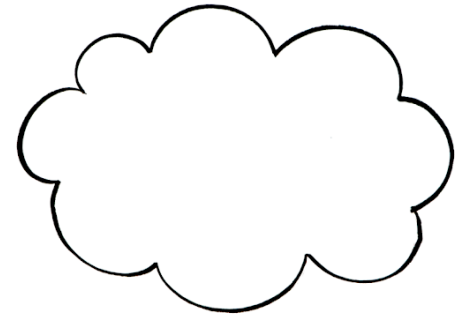
- example : 2-to-1 1-bit multiplexer

진리표

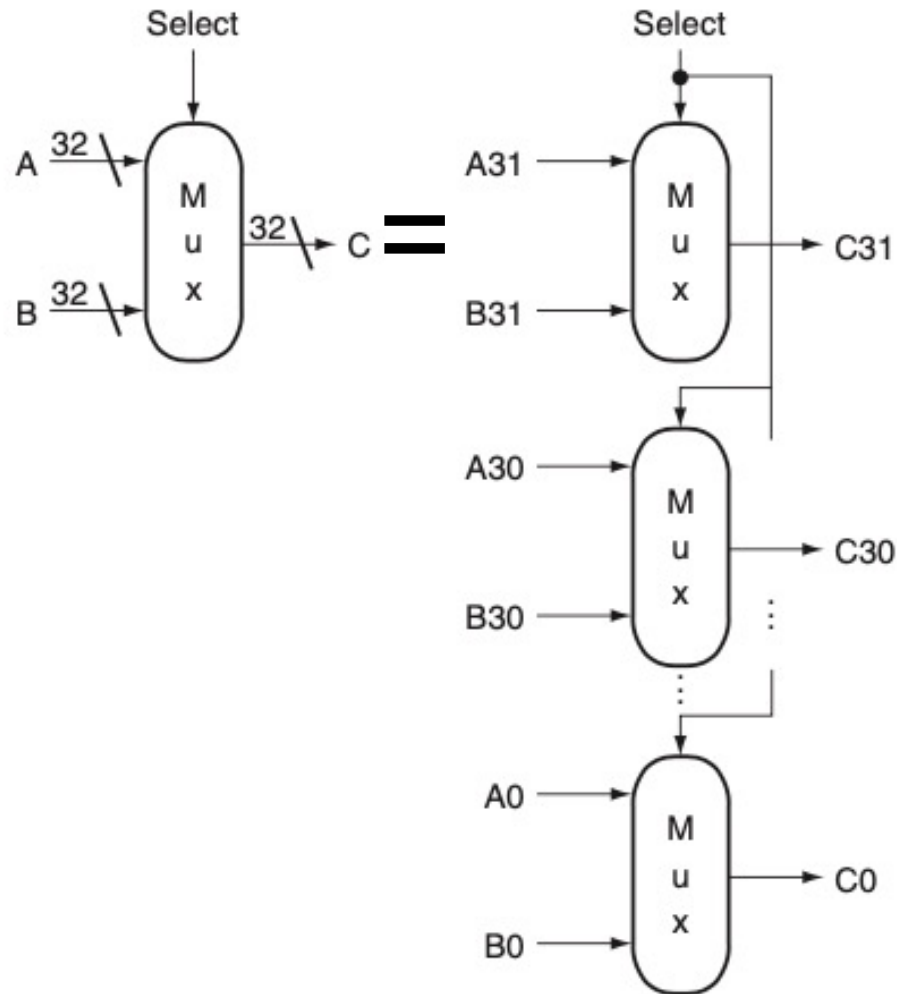
input			out
S	I <sub>0</sub>	I <sub>1</sub>	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

→ 각 output 의 논리식 → 논리식을 논리 게이트로 표현

$$Y = S'I_0I_1' + S'I_0I_1 + SI_0'I_1 + SI_0I_1$$



# 32-bit 2-to-1 MUX



a. A 32-bit wide 2-to-1 multiplexer

b. The 32-bit wide multiplexer is actually an array of 32 1-bit multiplexers

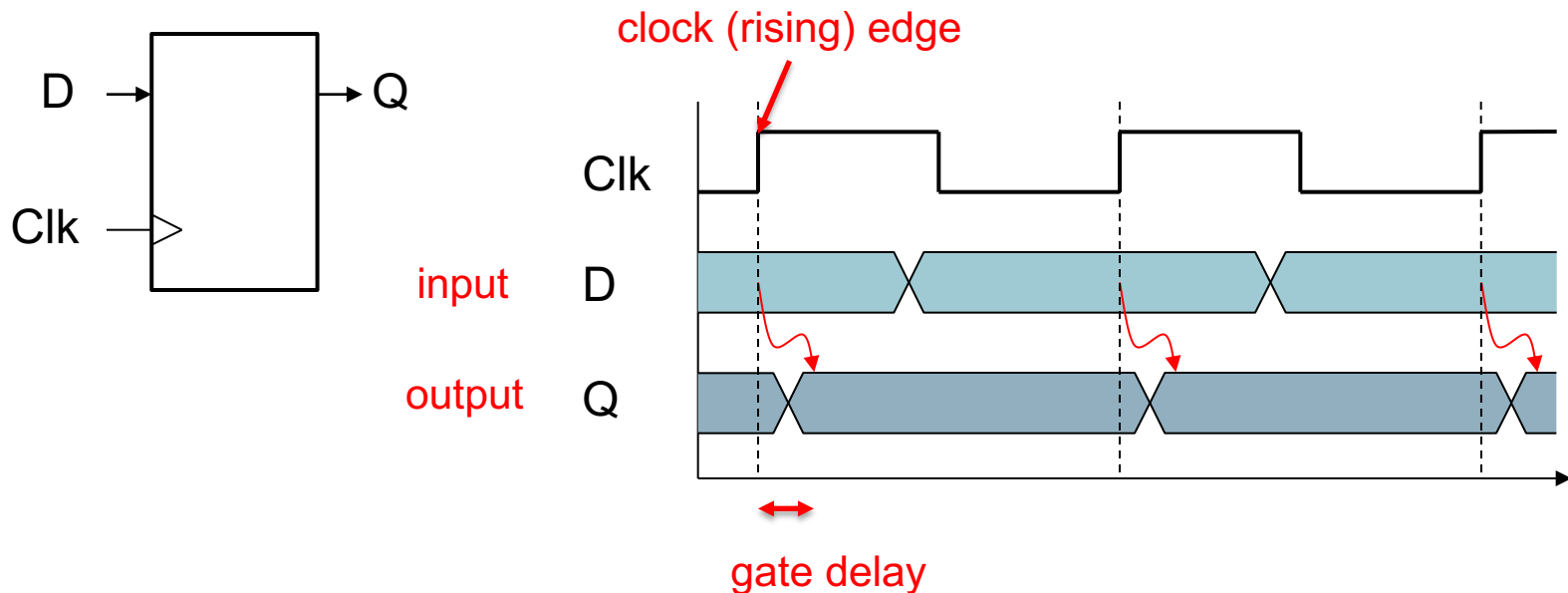
**FIGURE B.3.6 A multiplexer is arrayed 32 times to perform a selection between two 32-bit inputs.** Note that there is still only one data selection signal used for all 32 1-bit multiplexers.

# State (sequential) elements (순차회로)

- 연산된 데이터를 저장해 두었다가 나중에 읽어내기 위해 사용된다.
- $\text{Output} = f(\text{current input}, \text{previous input})$

# Sequential Elements

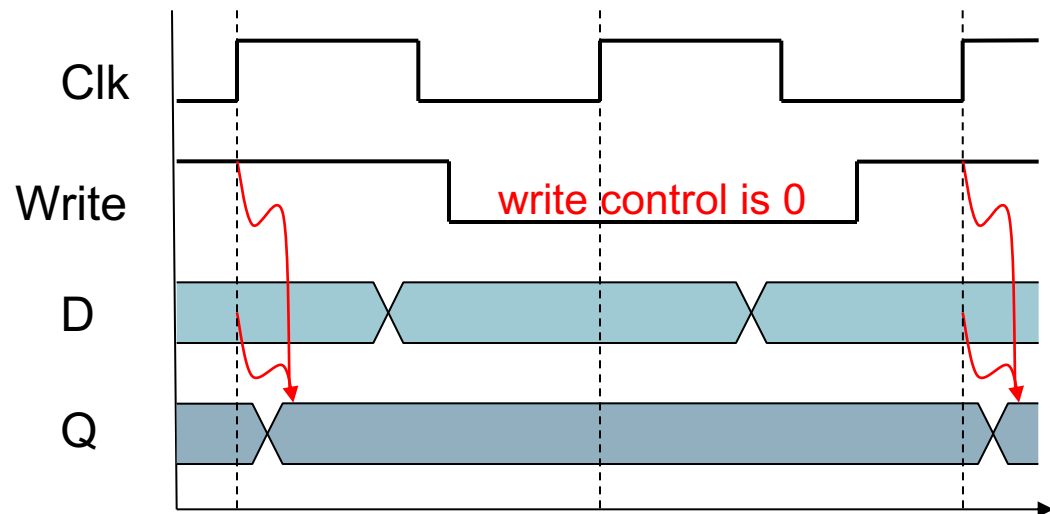
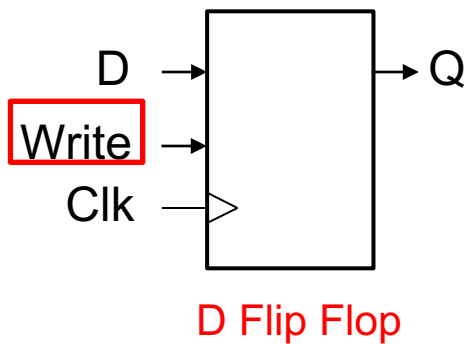
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1





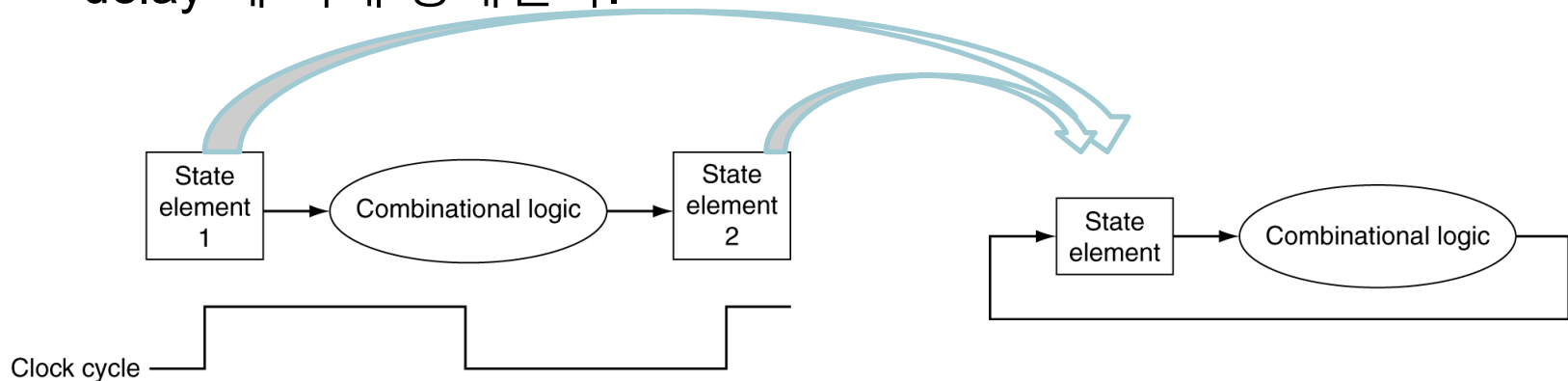
# Sequential Elements

- **Register with write control (no read control)**
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



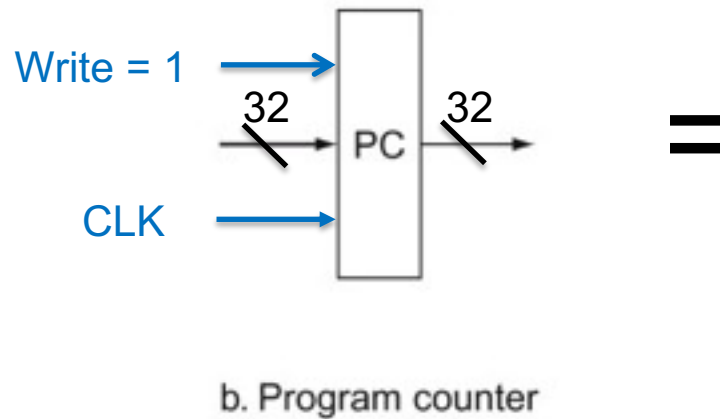
# Synchronous Digital 회로의 동작

- 클럭 사이클 동안(Between clock edges) 에 Combinational 회로에서 입력 신호에 대한 출력 신호를 만든다
- combinational 회로의 input은 state elements의 output이다.
- combinational 회로의 output은 state elements의 input이다.
- clock period (clock edge 간의 간격) 은 combinational 회로의 longest delay 에 의해 정해진다.



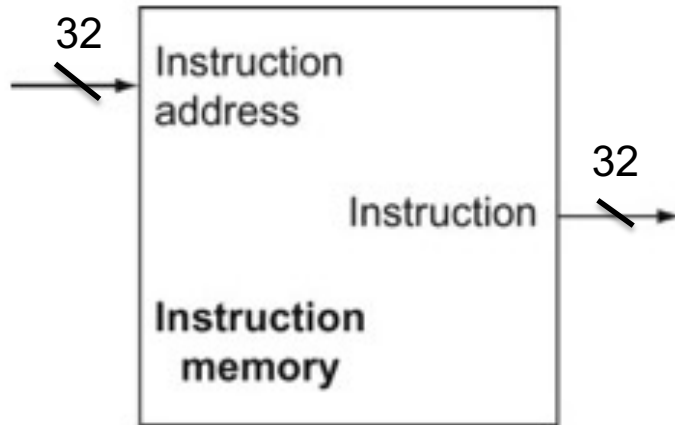
# Sequential Circuit : Register (clock signal 은 생략)

- a 32-bit register = 1-bit D flip flop x 32
- write signal 이 없으면 write signal 은 항상 1이라는 의미
- clock signal 도 있지만 (그림에서) 생략

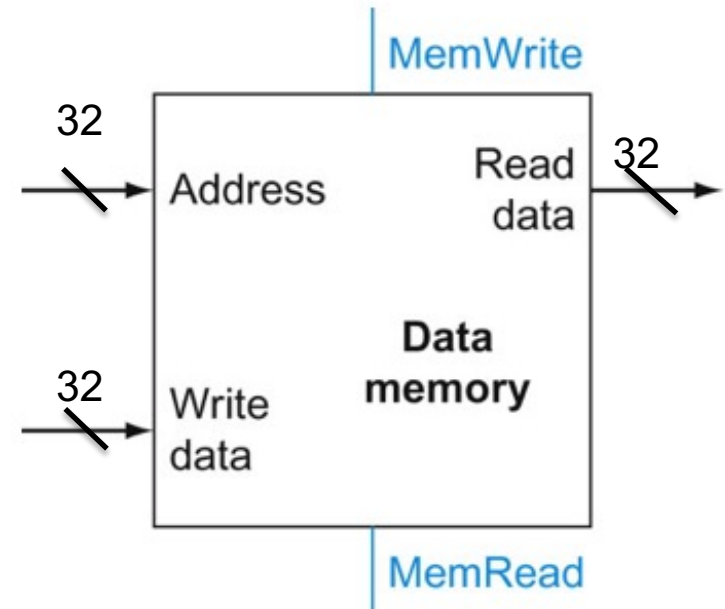


# Sequential Circuit : Memory (clock signal 은 생략)

Instruction memory is read-only in this implementation



a. Instruction memory  
program is pre-loaded

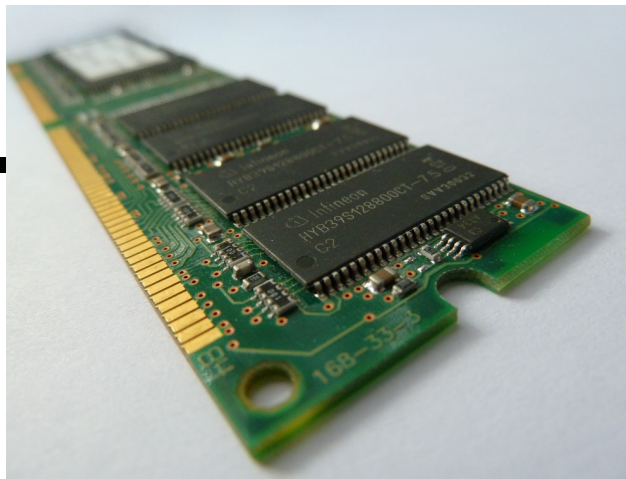


a. Data memory unit



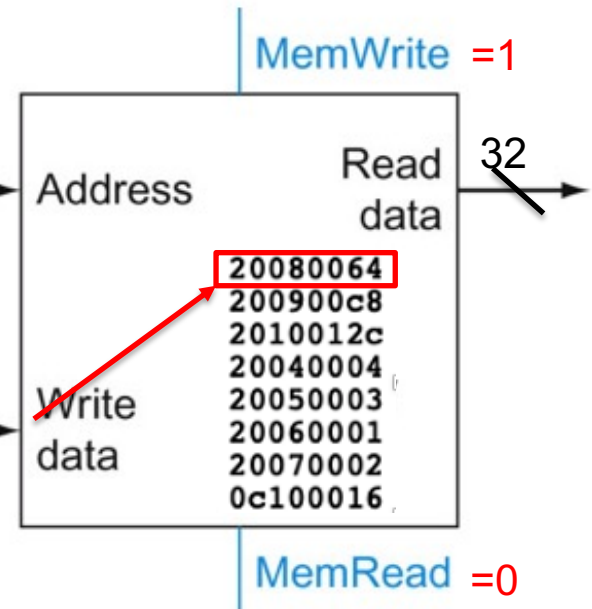
# Memory Write

at clock cycle 100



[00400024]	20080064
[00400028]	200900c8
[0040002c]	2010012c
[00400030]	20040004
[00400034]	20050003
[00400038]	20060001
[0040003c]	20070002
[00400040]	0c100016

0x20080064



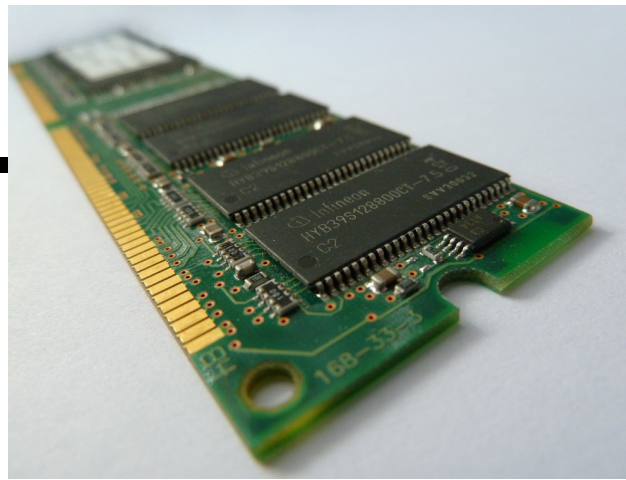
address 는 메모리에 저장되지 않는다는 말의 의미

a. Data memory unit

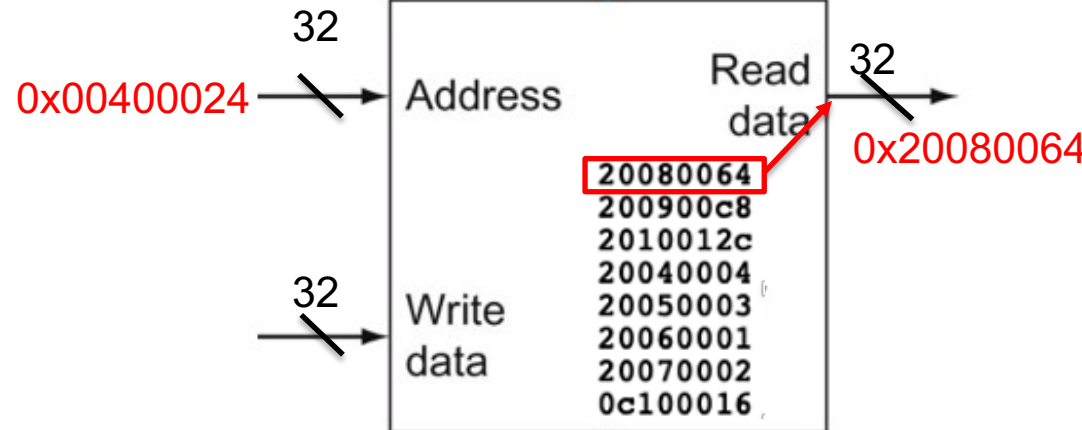


# Memory Read

at clock cycle 200



[00400024]	20080064
[00400028]	200900c8
[0040002c]	2010012c
[00400030]	20040004
[00400034]	20050003
[00400038]	20060001
[0040003c]	20070002
[00400040]	0c100016



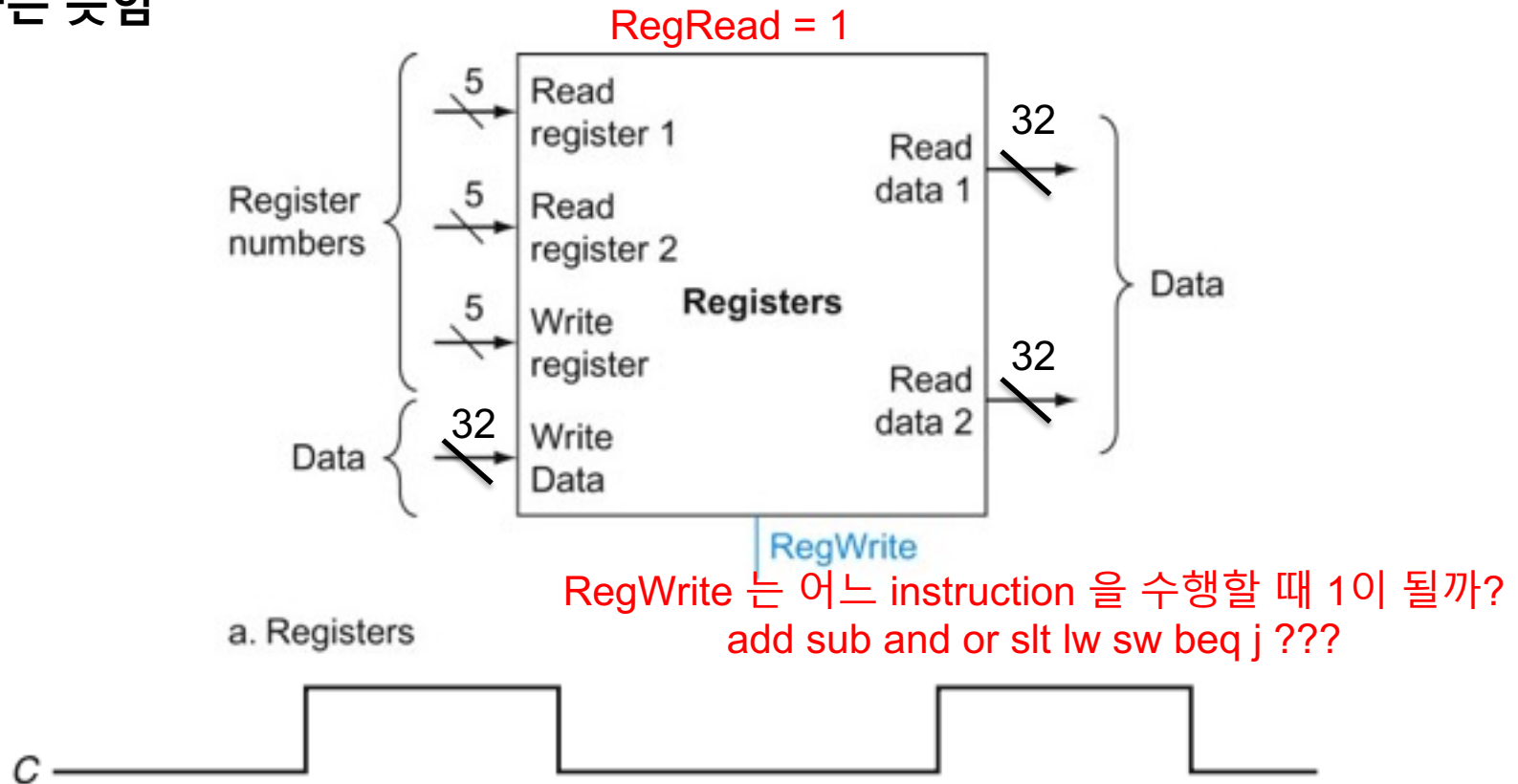
address 는 메모리에 저장되지 않는다는 말의 의미

a. Data memory unit



# Sequential Circuit : Register File (clock signal 은 생략)

- 32 x 32-bit registers
- RegWrite =1 일 때만 레지스터의 값이 바뀌고
- RegRead signal 이 없는 것은 항상 (매 클럭마다) 레지스터가 읽혀져 출력값이 된다는 뜻임



# 1) Build a datapath first and 2) add a control

add, sub, and, or, slt  
lw, sw  
beq, j

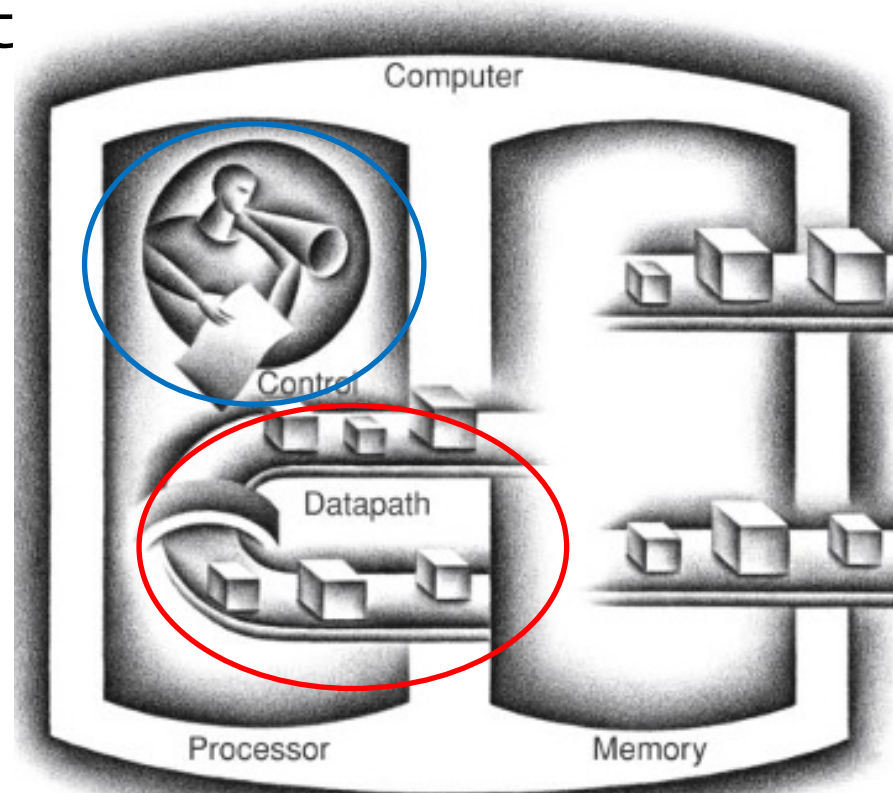


FIGURE 1.5 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.



# Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, ...
  - 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

# How instructions are executed in a processor

1) Fetch instruction : instruction memory 중 PC가 가리키는 주소에서 수행할 instruction 을 프로세서로 읽어온다.

(PC 는 4만큼 자동 증가)

2) Execute instruction : 그 명령어를 수행한다.

2-1)명령어를 해석하는 동시에 Register file 에서 레지스터 값을 읽는다.

2-2) 명령어 종류에 따라 다음을 실행한다.

Arithmetic/Logic 연산 -> 결과를 레지스터에 쓴다. **add, sub, and, or, slt**

Memory 주소 계산 -> data memory 에 쓴다. **sw**

**lw** -> data memory 를 읽는다 -> 결과를 레지스터에 쓴다.

**beq, j** Branch target address 계산 -> BTA 를 PC 에 쓰거나 쓰지 않는다.

```
add $8, $3, $4
lw $9, 4($7)
sw $10, 8($11)
beq $5, $6, L
j FN
```

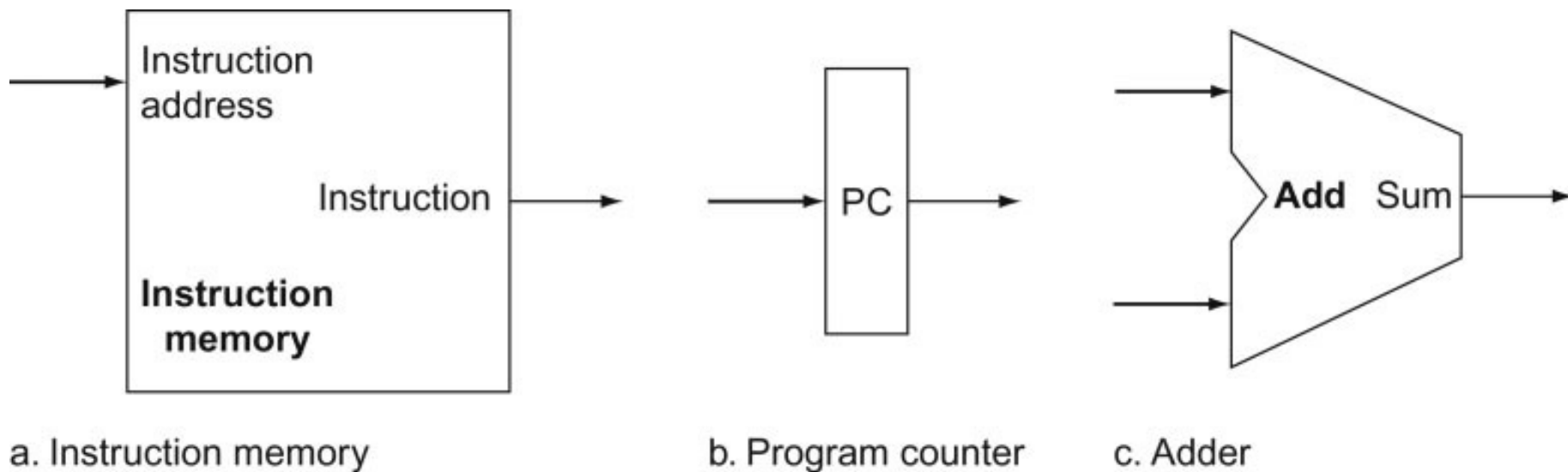


FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

# 1) Instruction Fetch

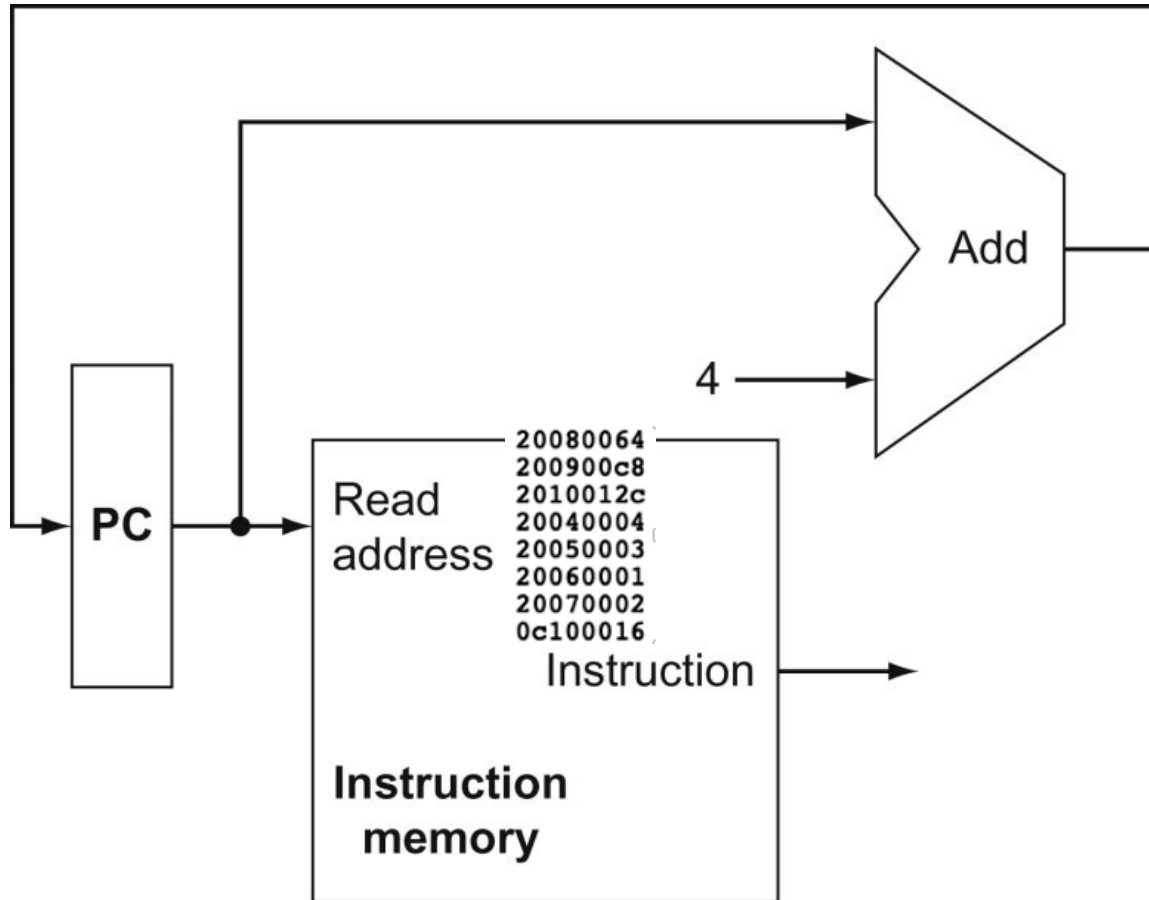
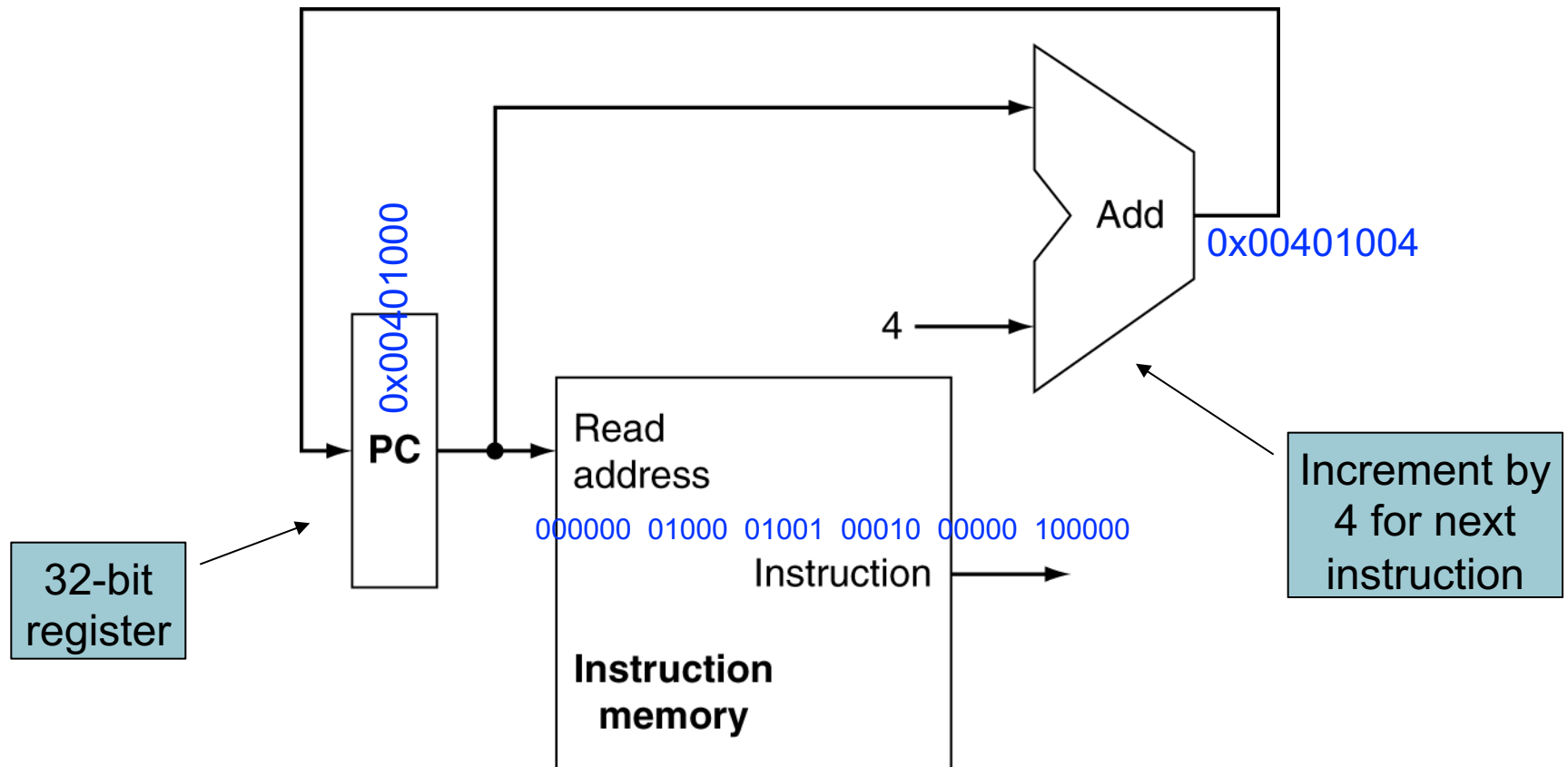


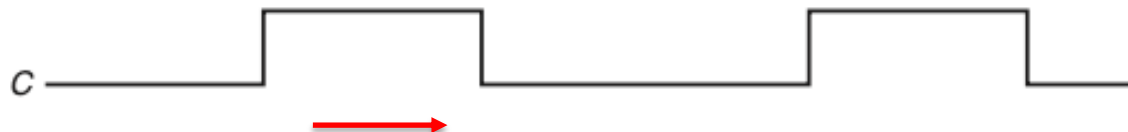
FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

# Instruction Fetch(datapath part 1)

- A portion of datapath used for fetching instruction and incrementing PC



[0x00401000] add \$2, \$8,\$9 000000 01000 01001 00010 00000 100000

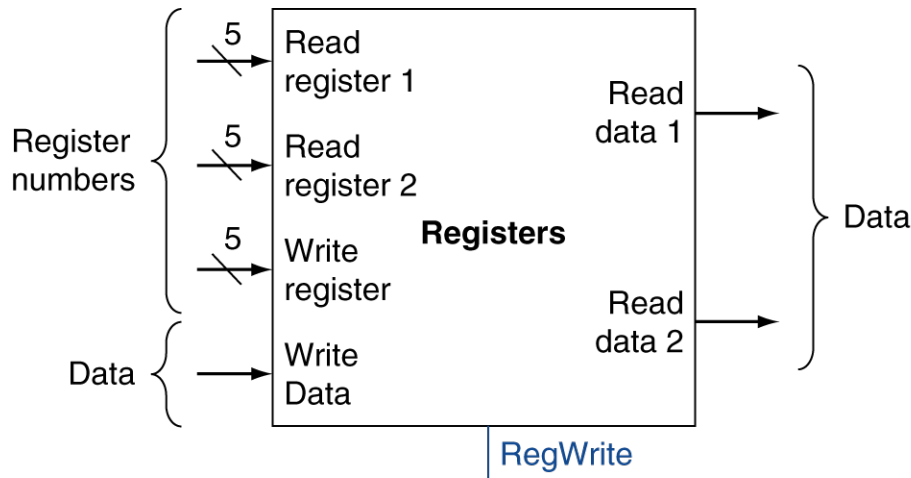


## 2) Execution of Arithmetic/Logic Instructions

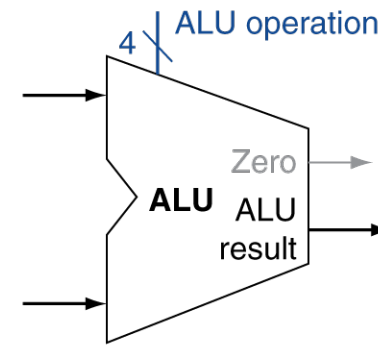
add \$2, \$8, \$9

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

ALU operation	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR



a. Registers

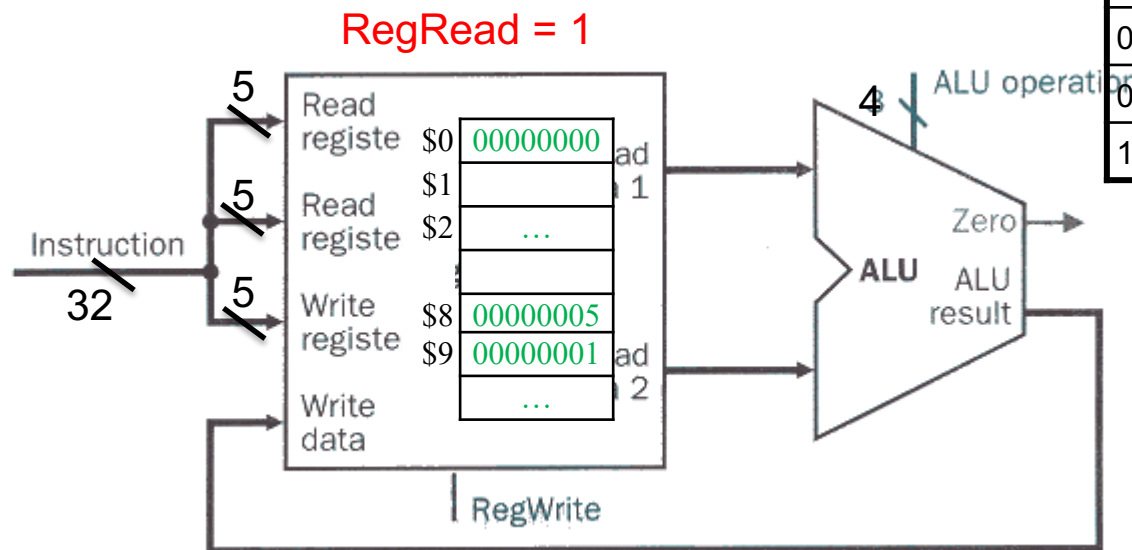


b. ALU

# Datapath part 2: Execution of A/L Instructions

add \$2, \$8,\$9 000000 01000 01001 00010 00000 100000

000000 01000 01001 00010 00000 100000

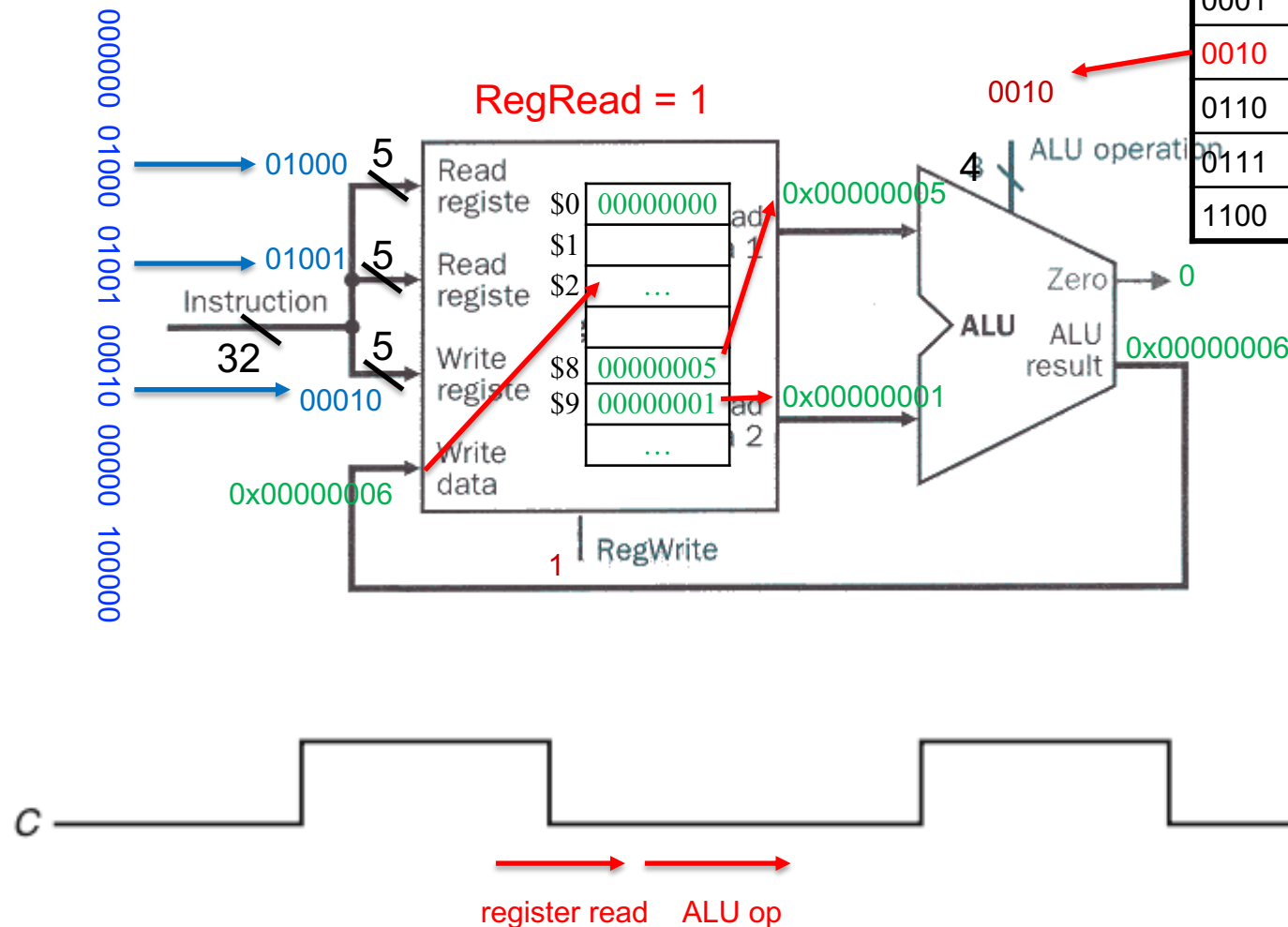


ALU operation	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR



# Datapath part 2: Execution of A/L Instructions

add \$2, \$8,\$9 000000 01000 01001 00010 00000 100000





# ALU Control

add, sub, and, or, slt

lw, sw  
beq, j

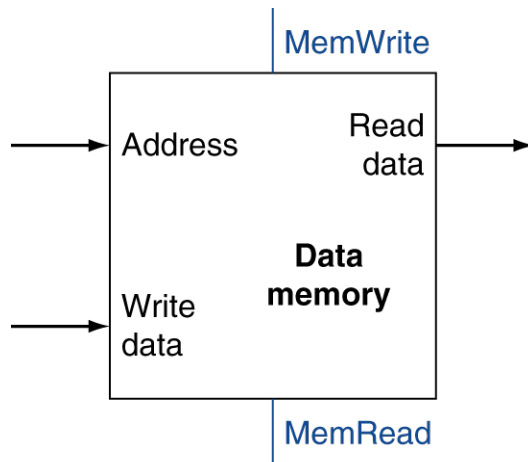
- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

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

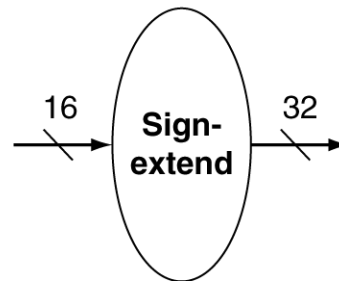
### 3) Execution of Load/Store Instructions

`lw $2, 4($8)`

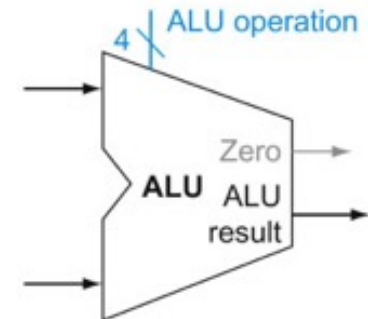
- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit

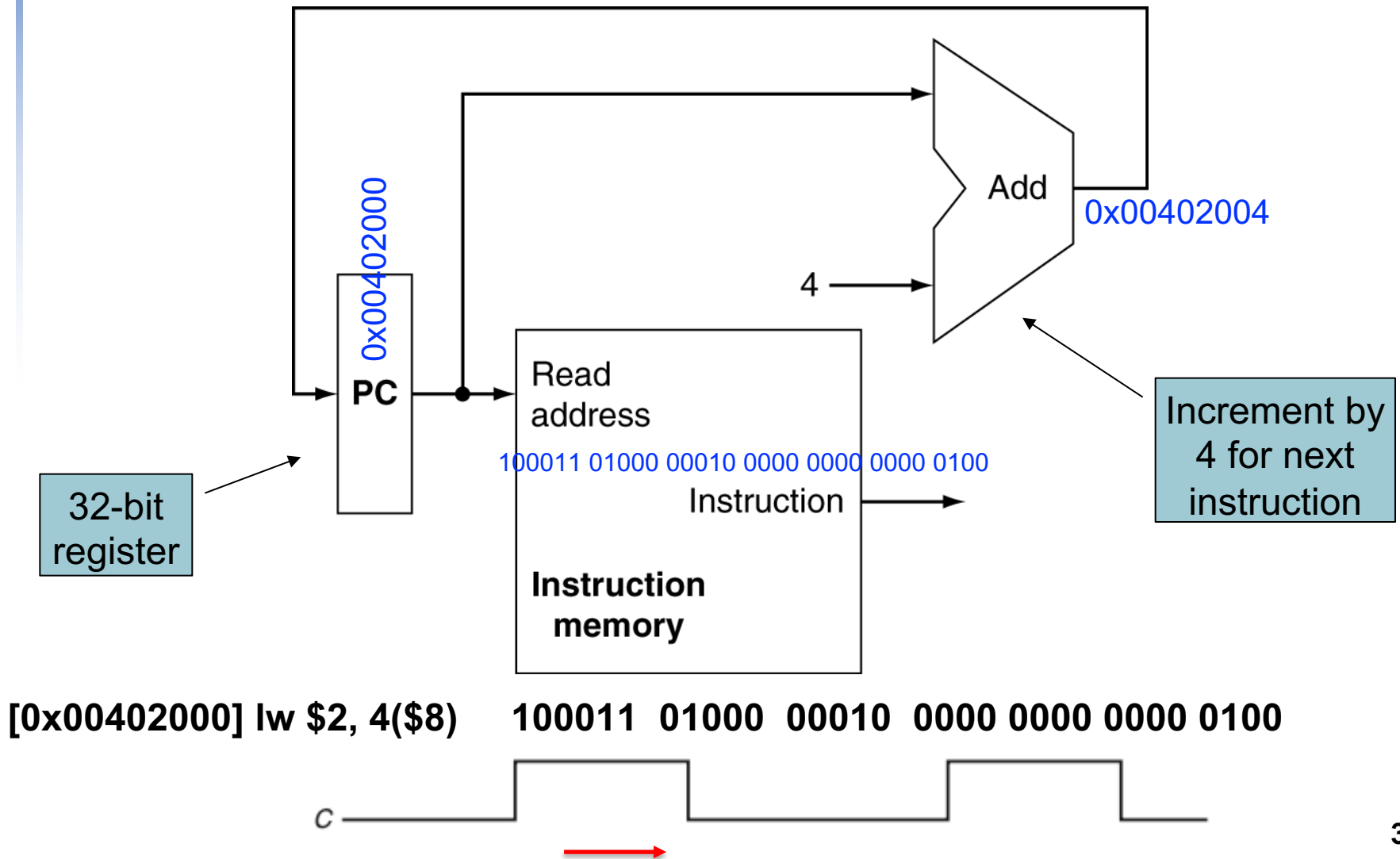


b. Sign extension unit



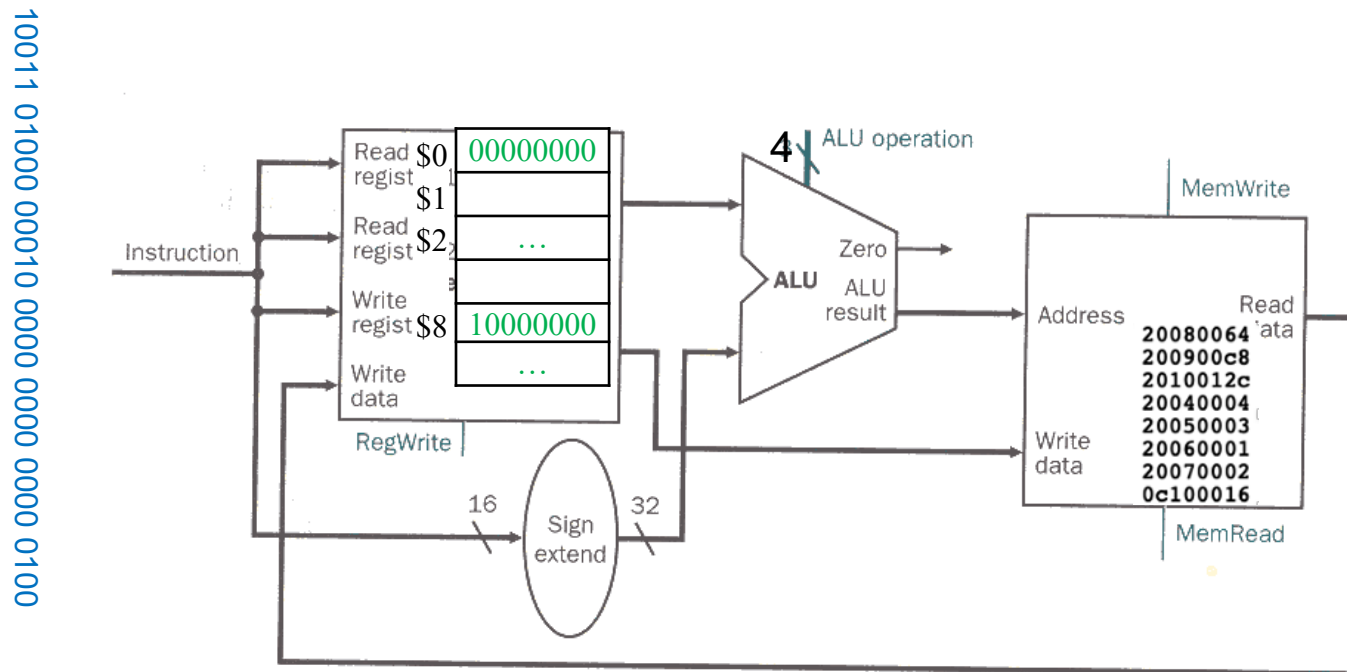
b. ALU

# Part 1: Instruction Fetch



# Datapath part 3: Execution of Data Transfer Instructions

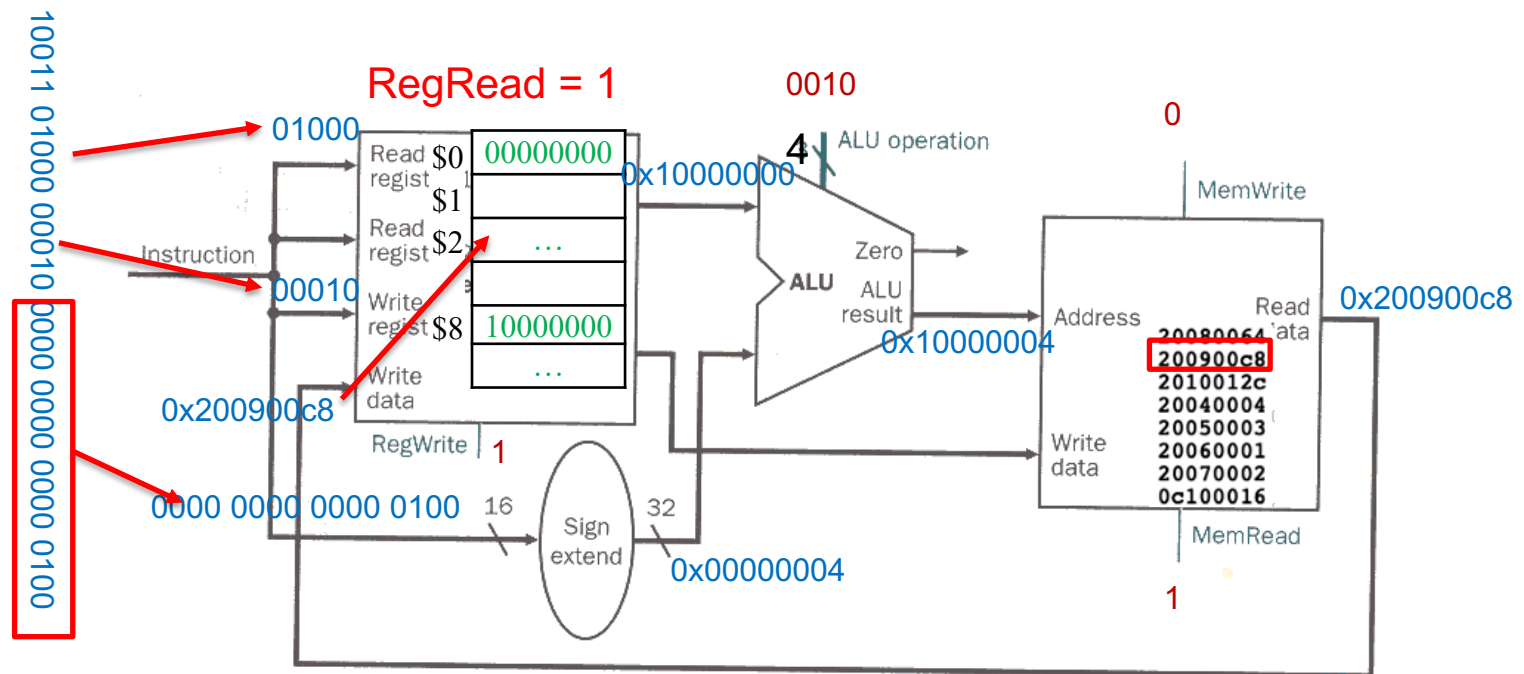
- A portion of datapath used for register access, followed by a memory address calculation, then a read or write from memory, and a write into the register file if the instruction is a load.



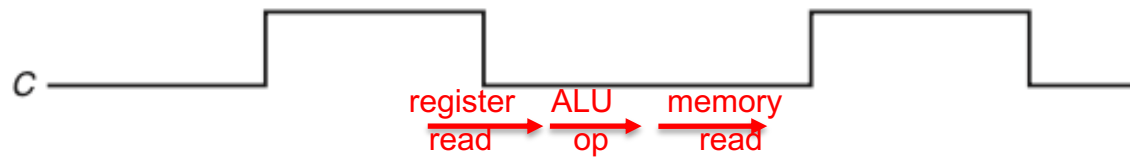
**lw \$2, 4(\$8)    100011 01000 00010 0000 0000 0000 0100**



# Datapath part 3: Execution of Data Transfer Instructions

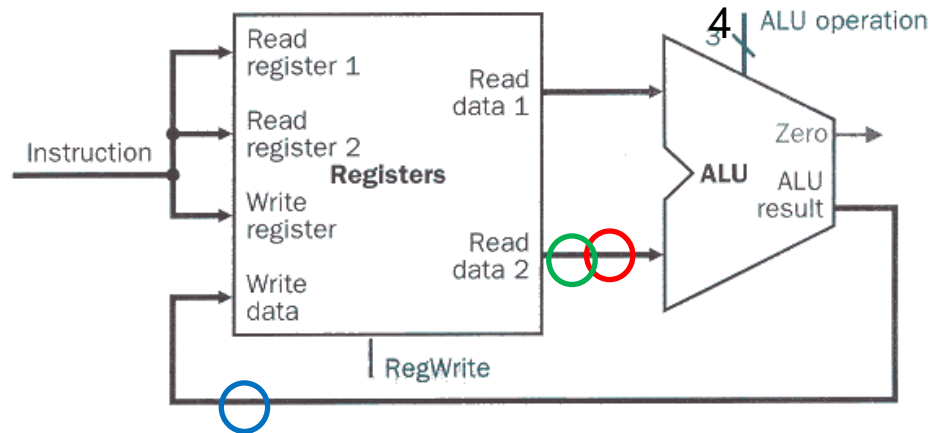


`lw $2, 4($8)`    `10011 01000 00010 0000 0000 0000 0100`



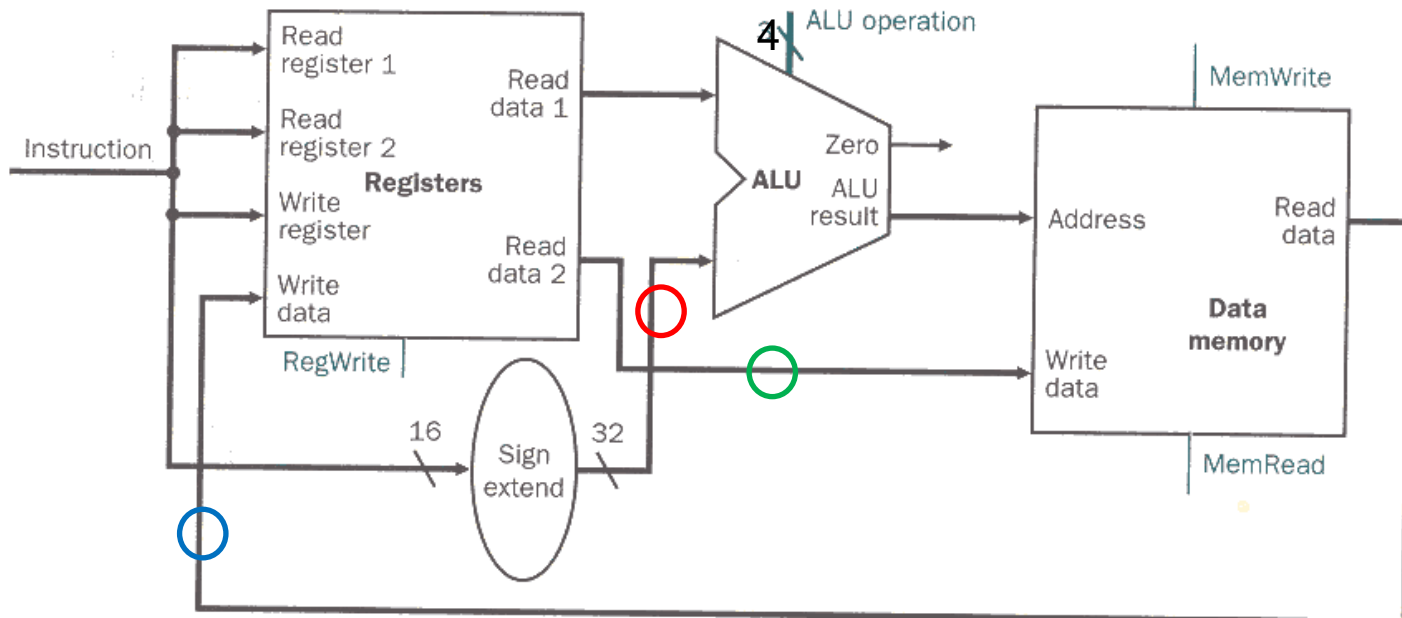
## part 2 for ALU instructions (add/sub/and/or/slt)

---

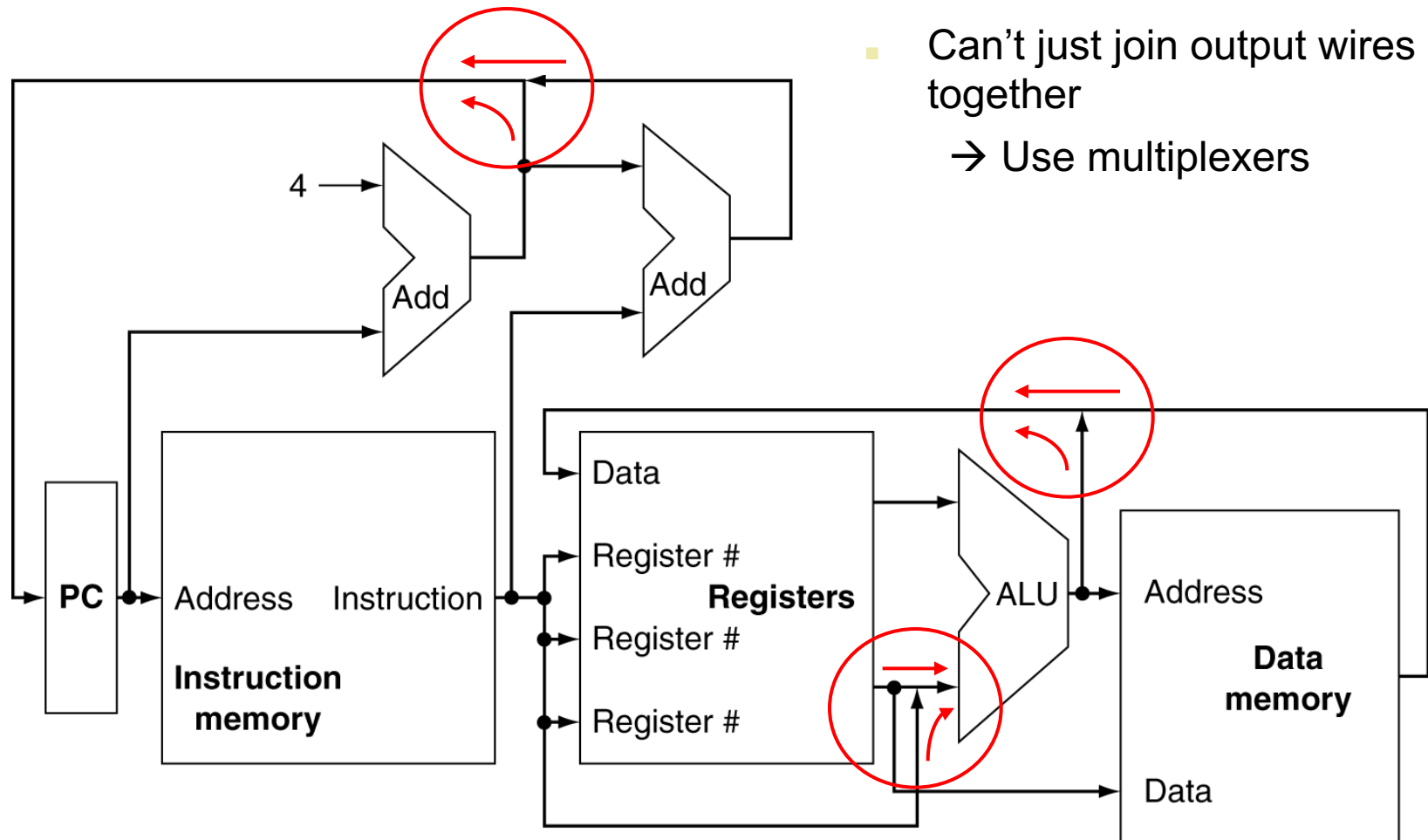


+

## part 3 for lw/sw instructions

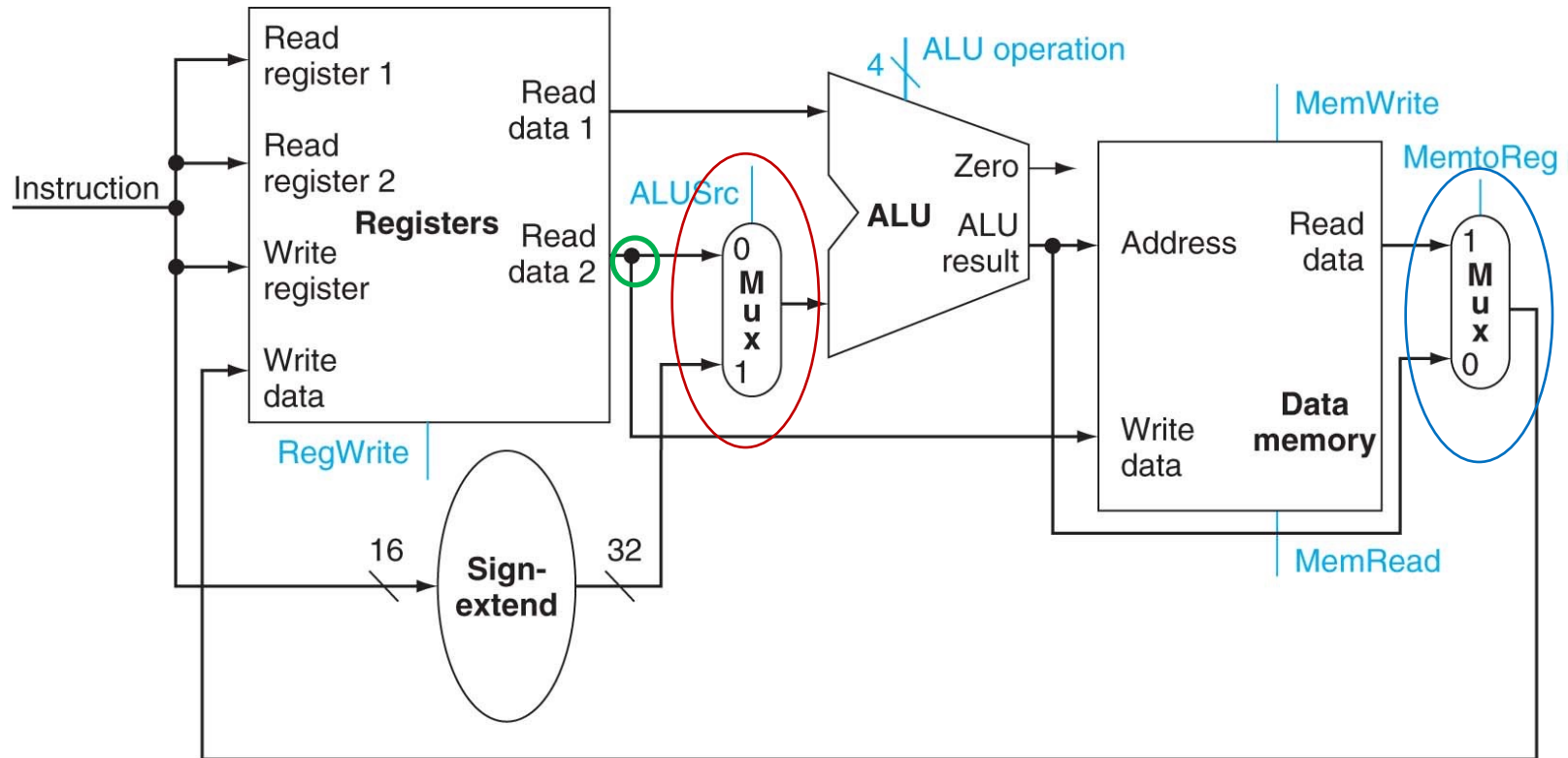


# Role of Multiplexers



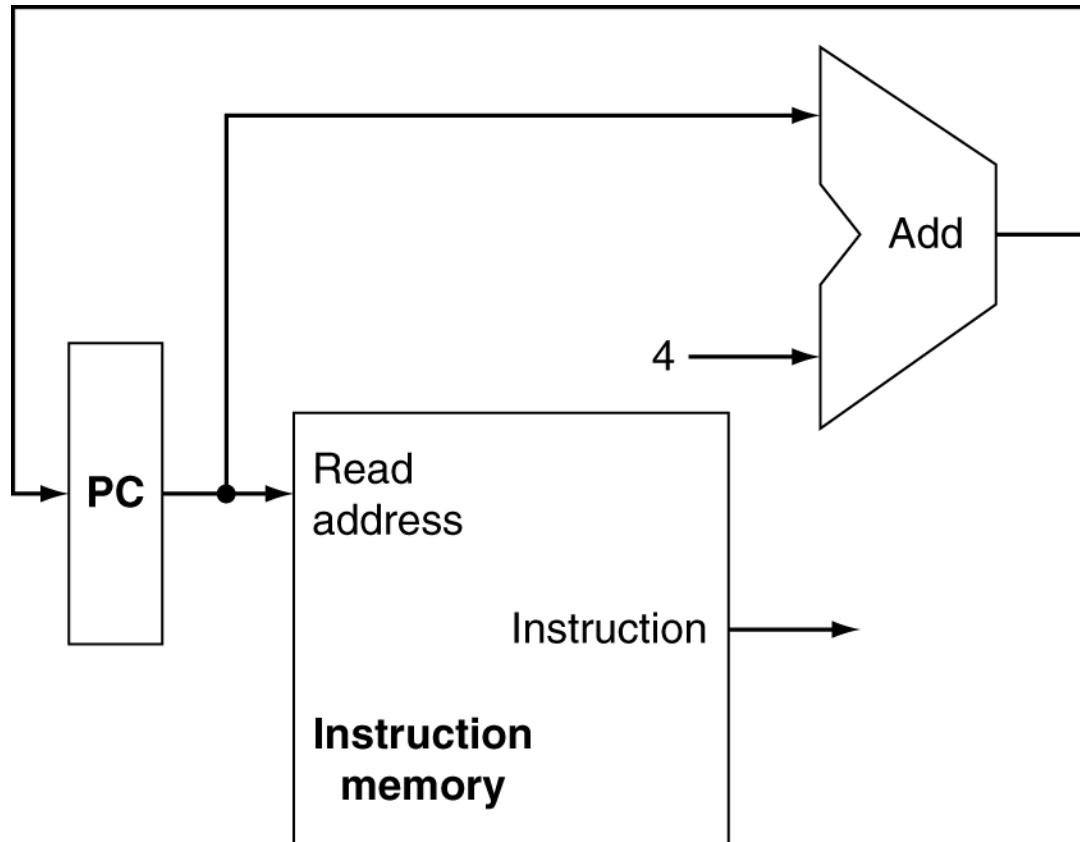
- Can't just join output wires together  
→ Use multiplexers

# Combining part 2 and 3

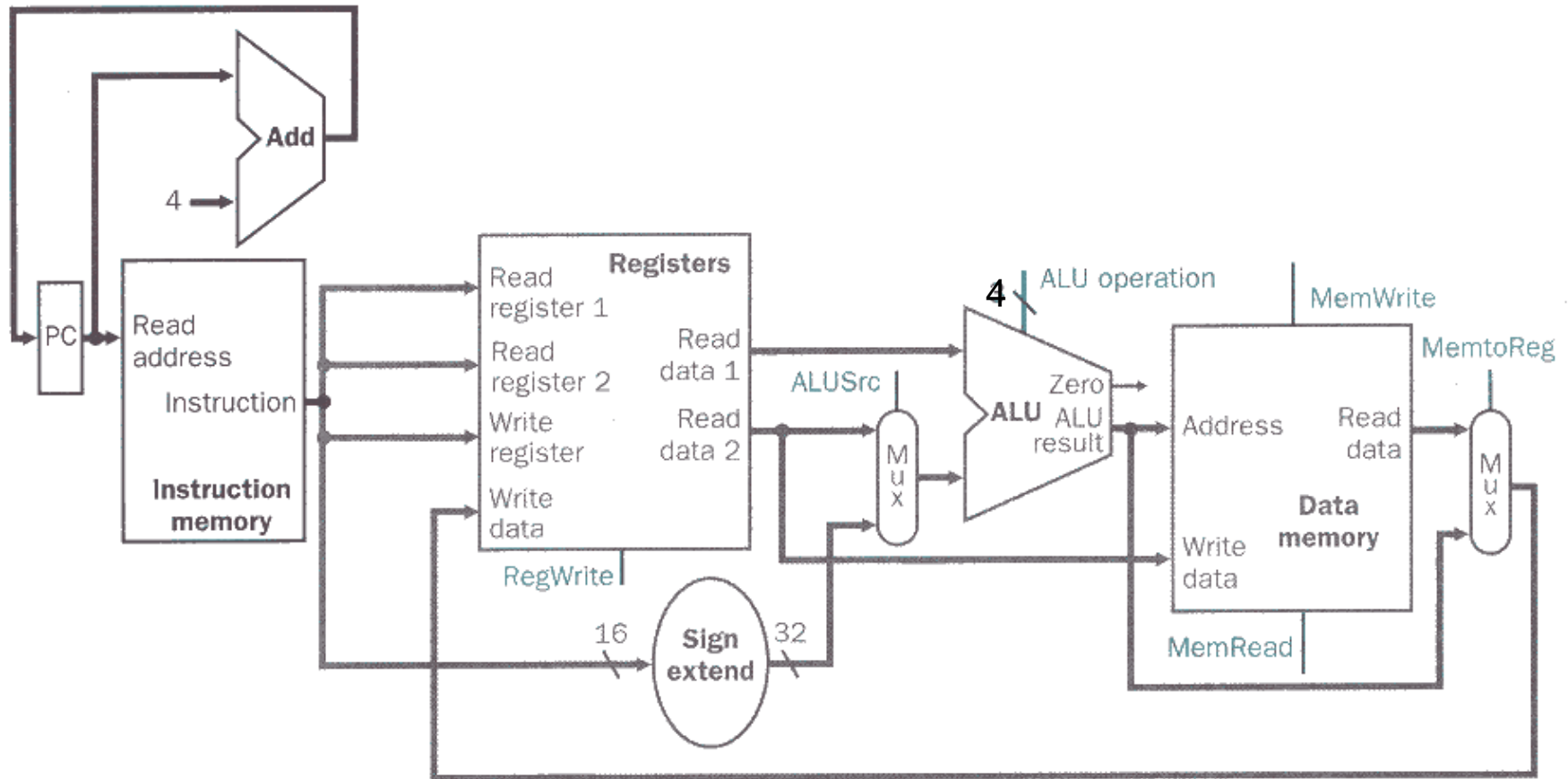




# Part 1: Instruction Fetch



# Adding part 1



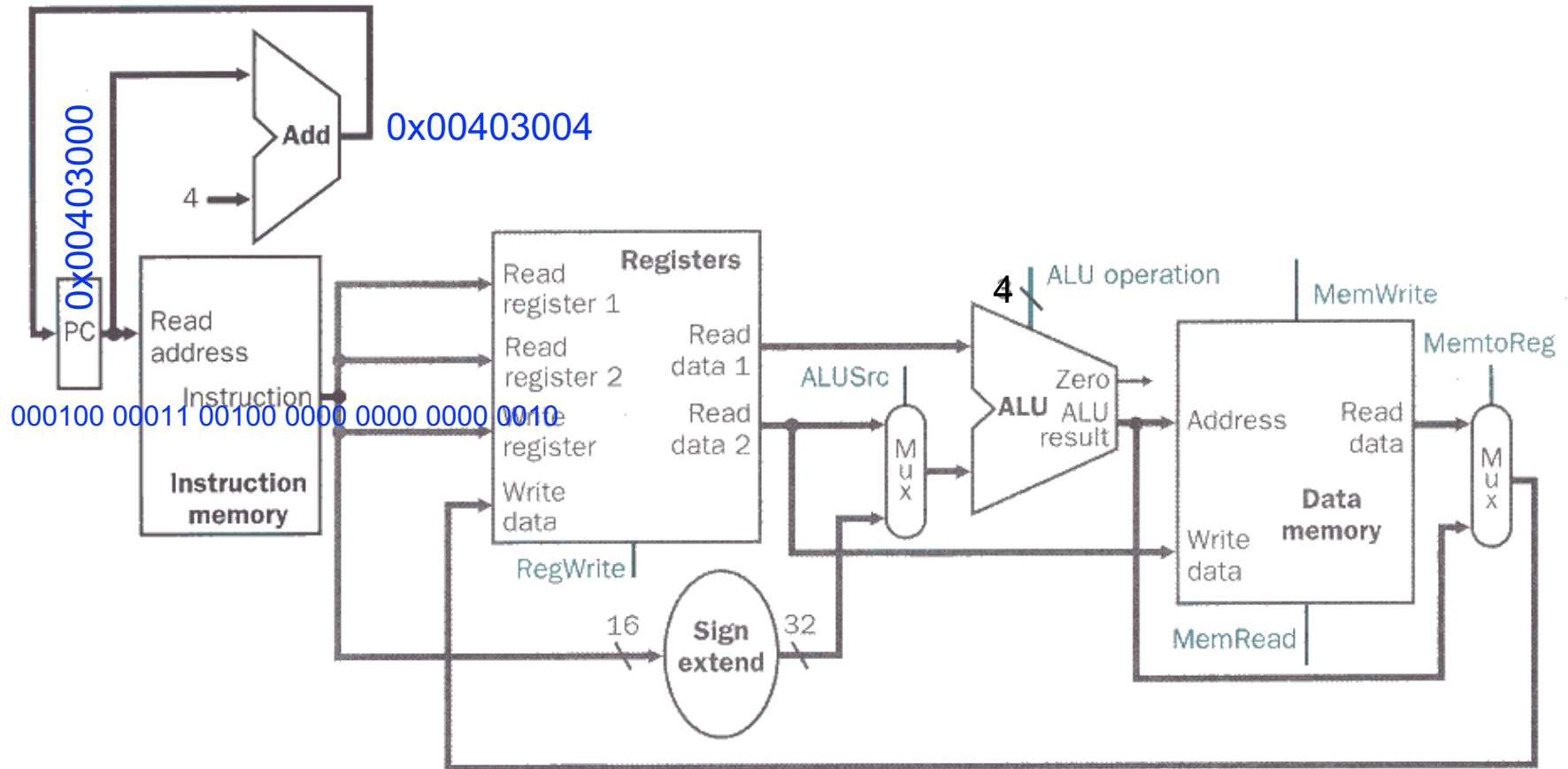
# 4) Execution of Branch Instructions

---

beq \$3, \$4, L

- 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)

# Fetching **beq** instruction

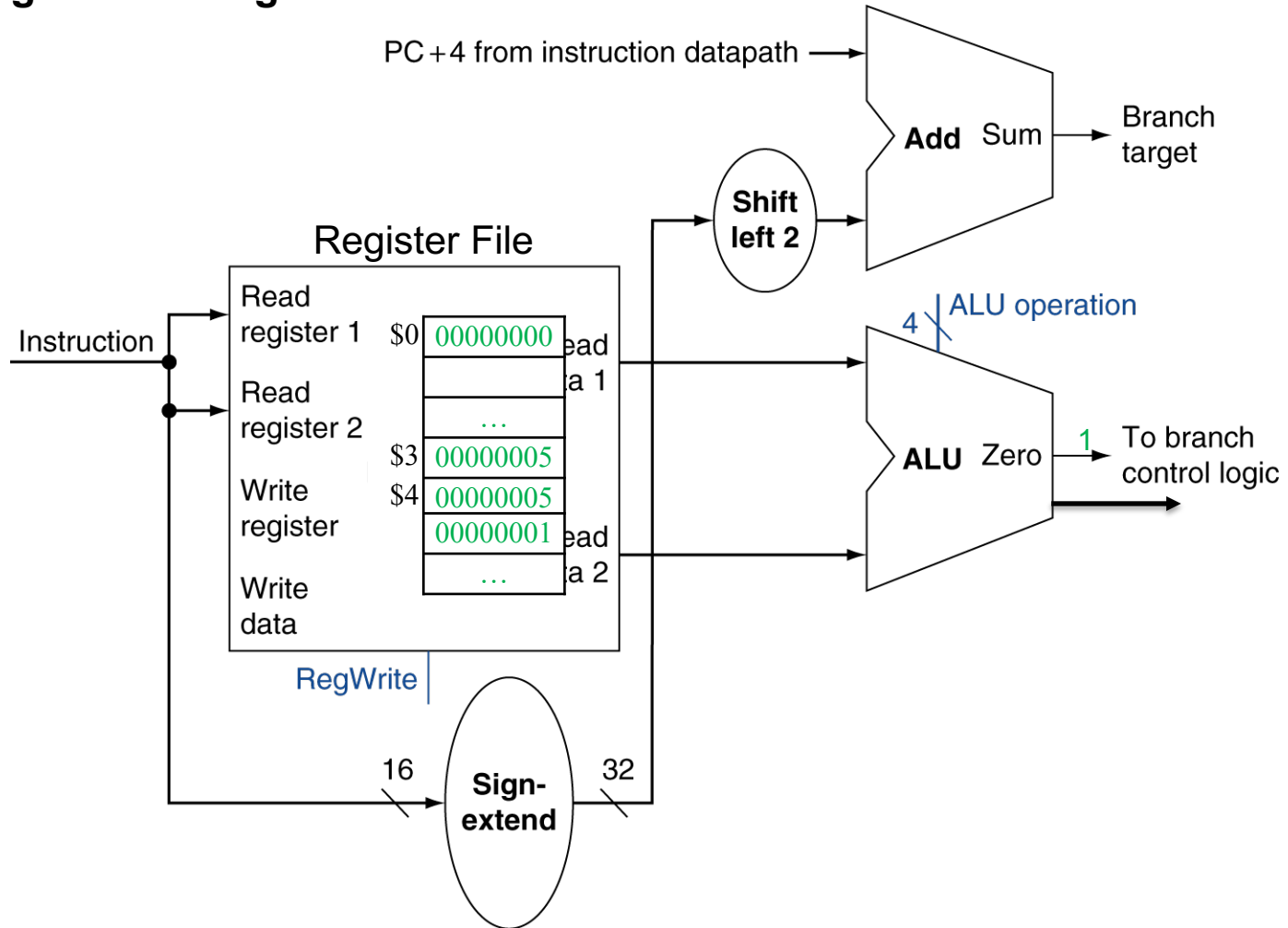


[0x00403000] beq \$3, \$4, L 000100 00011 00100 0000 0000 0000 0010

# part 4: Branch Instructions

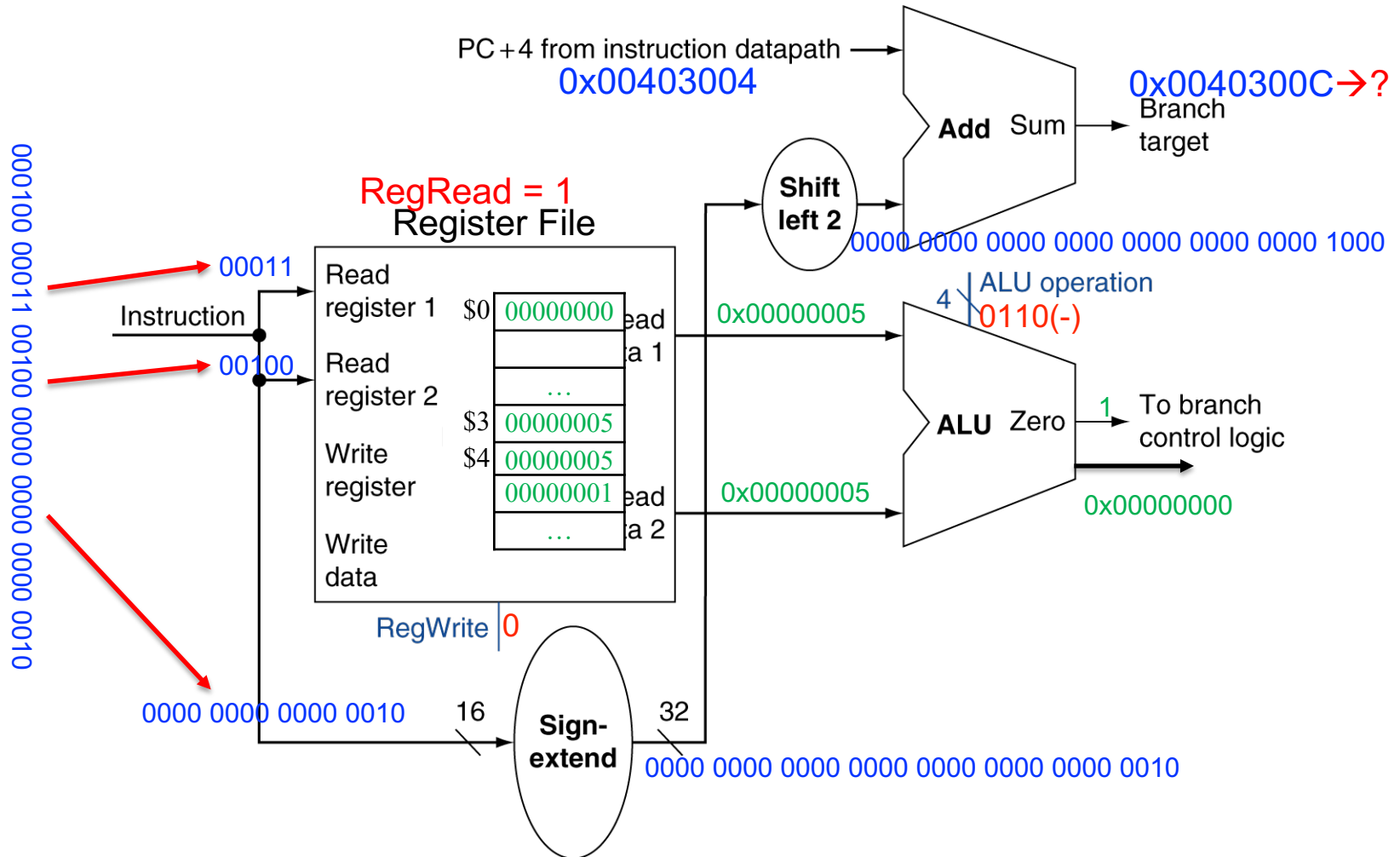
A portion of datapath used for evaluating a branch condition and computing branch target address

000100 00011 00100 0000 0000 0000 0010



beq \$3, \$4, L 000100 00011 00100 0000 0000 0000 0010

# part 4: Branch Instructions



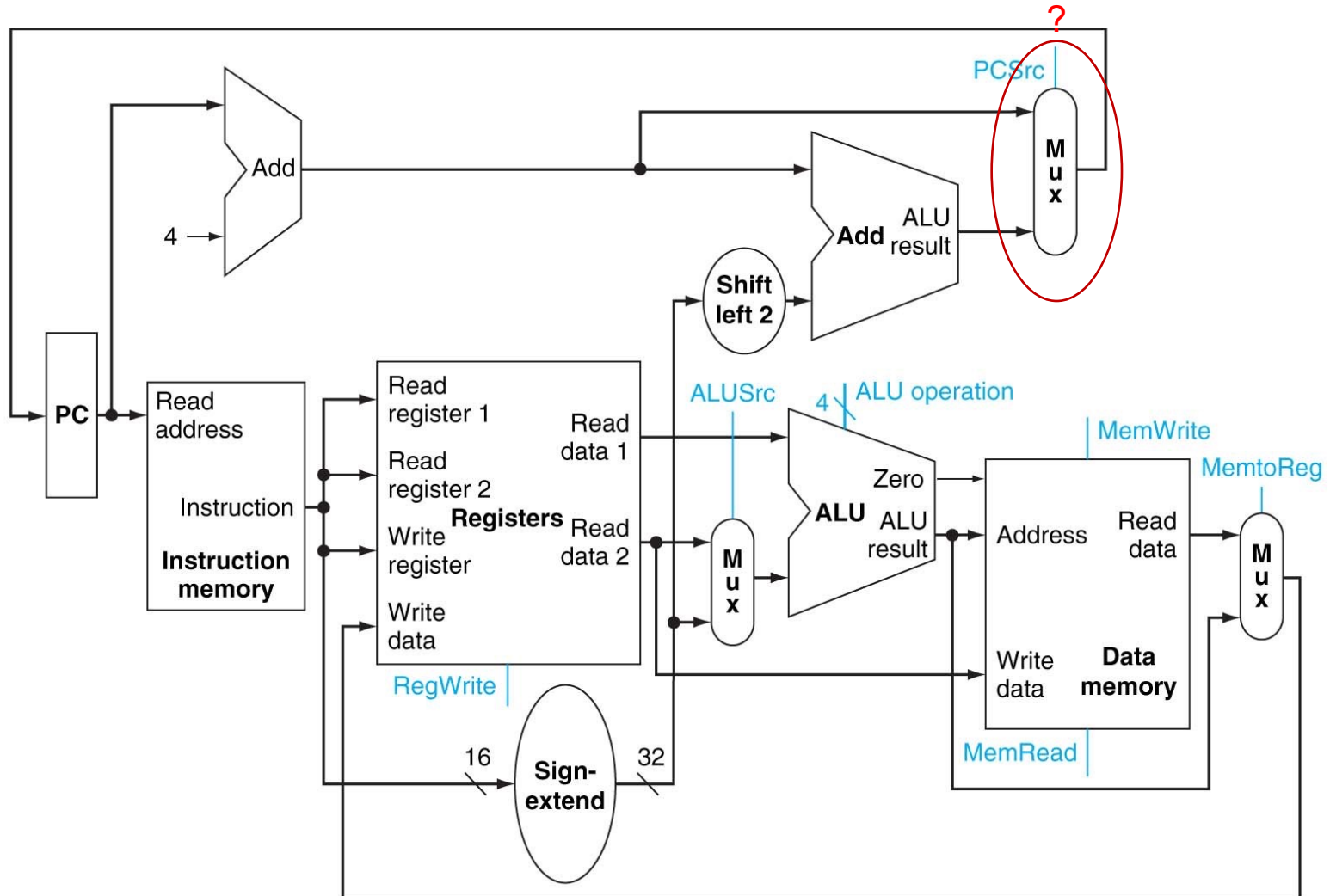
`beq $3, $4, L`    `000100 00011 00100 0000 0000 0000 0010`

# ALU Control

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

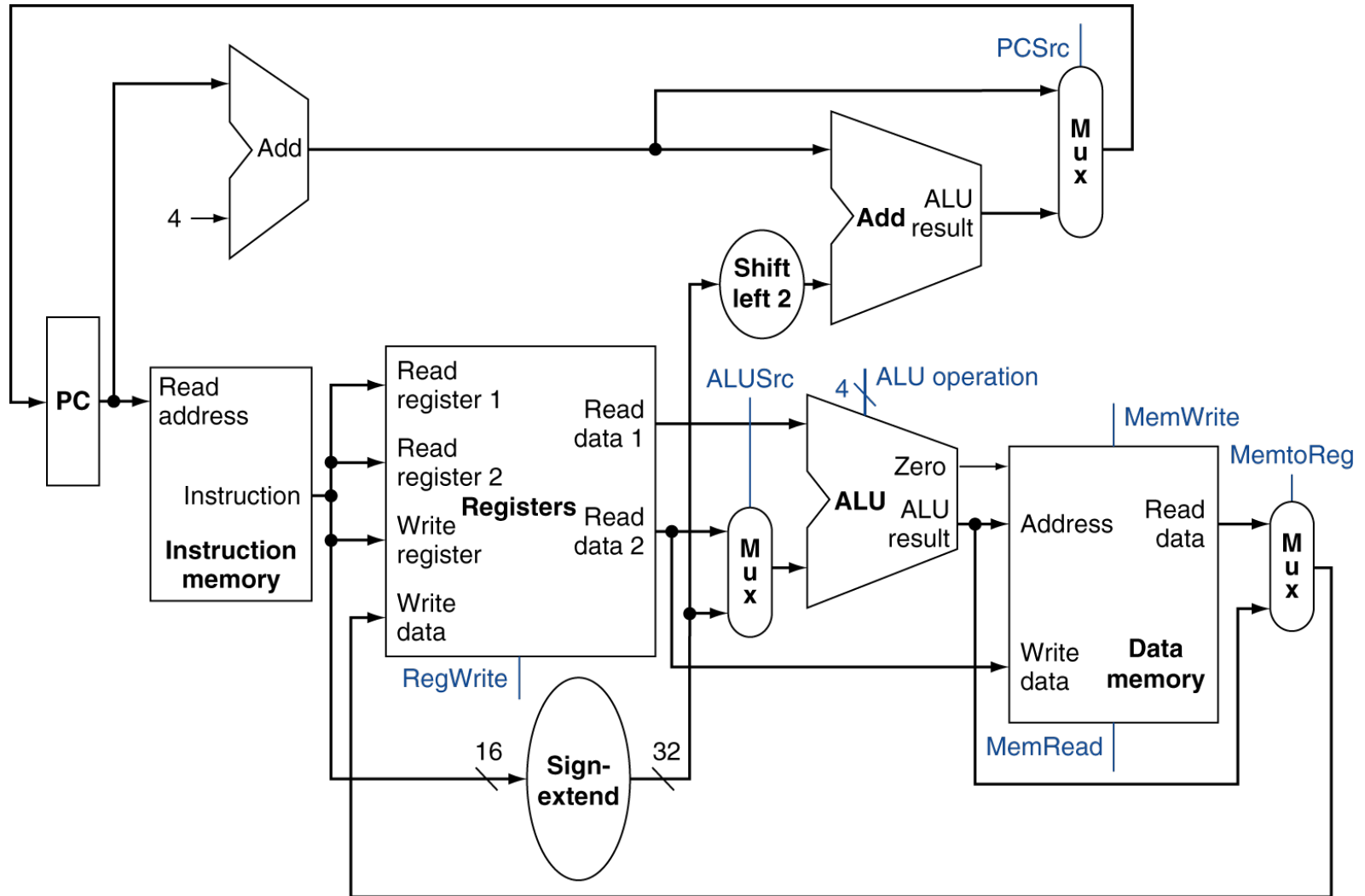
ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

# Adding part 4

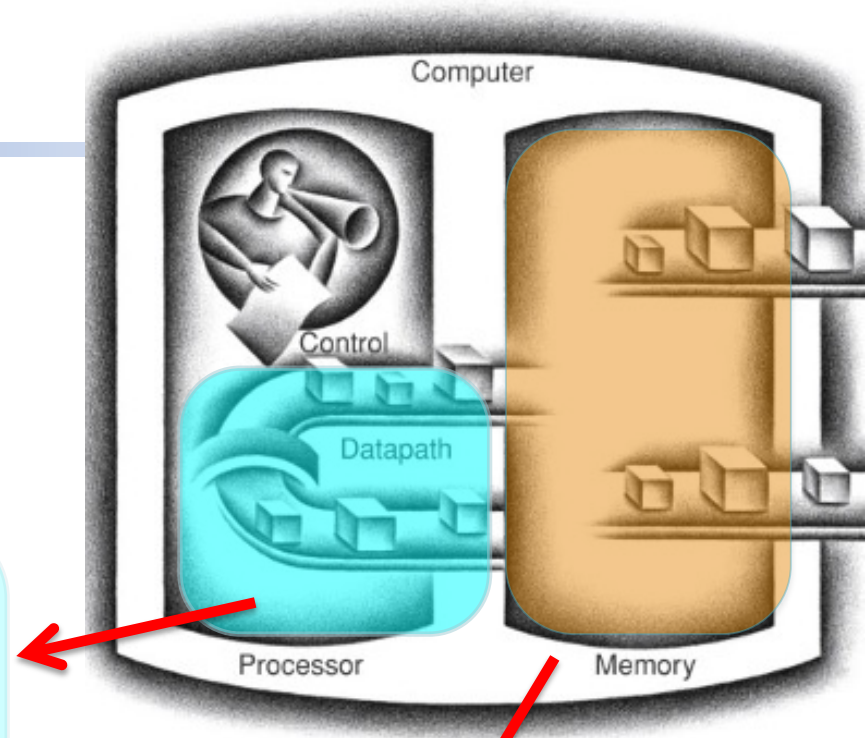
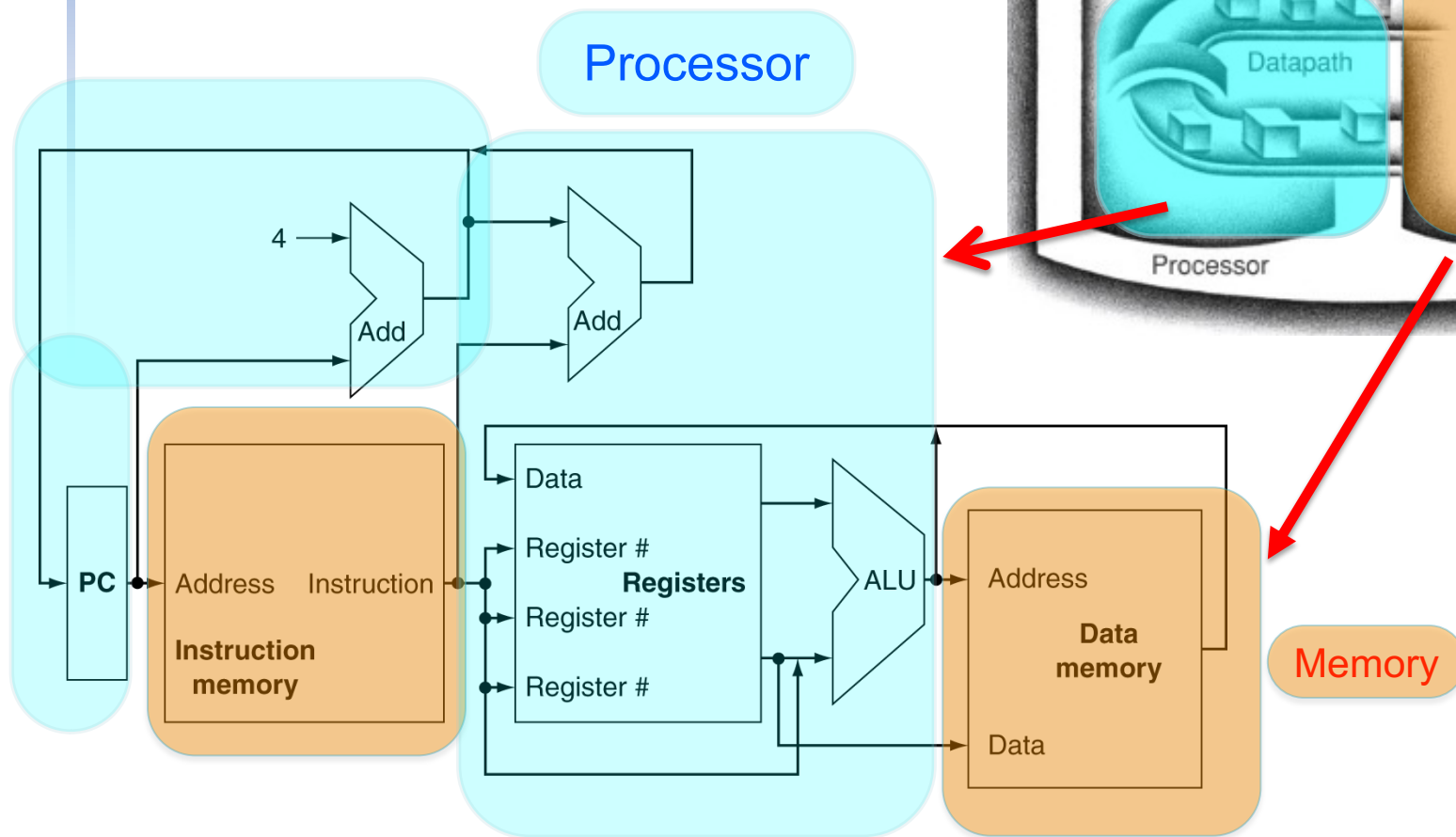




# Full Datapath

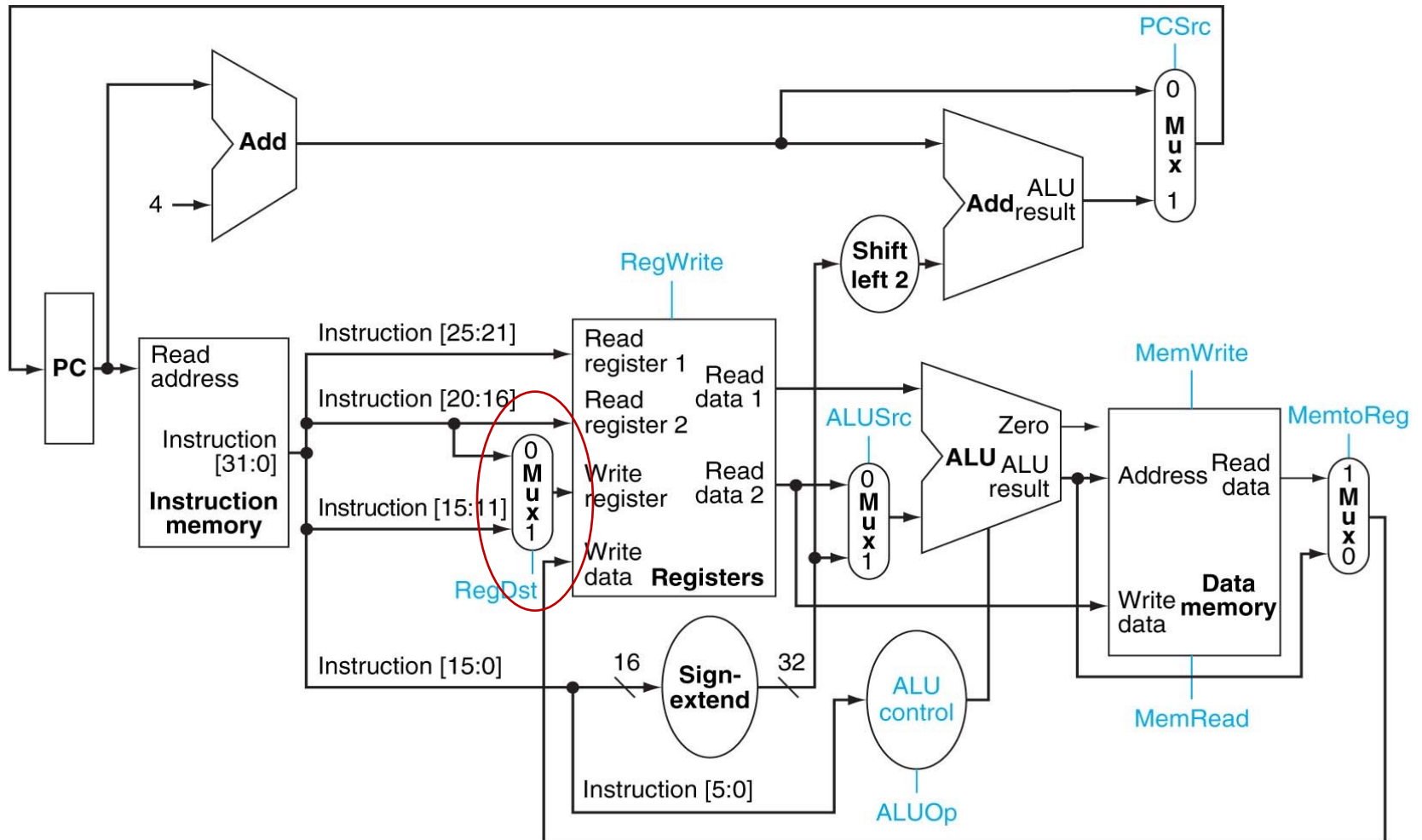


# CPU Overview



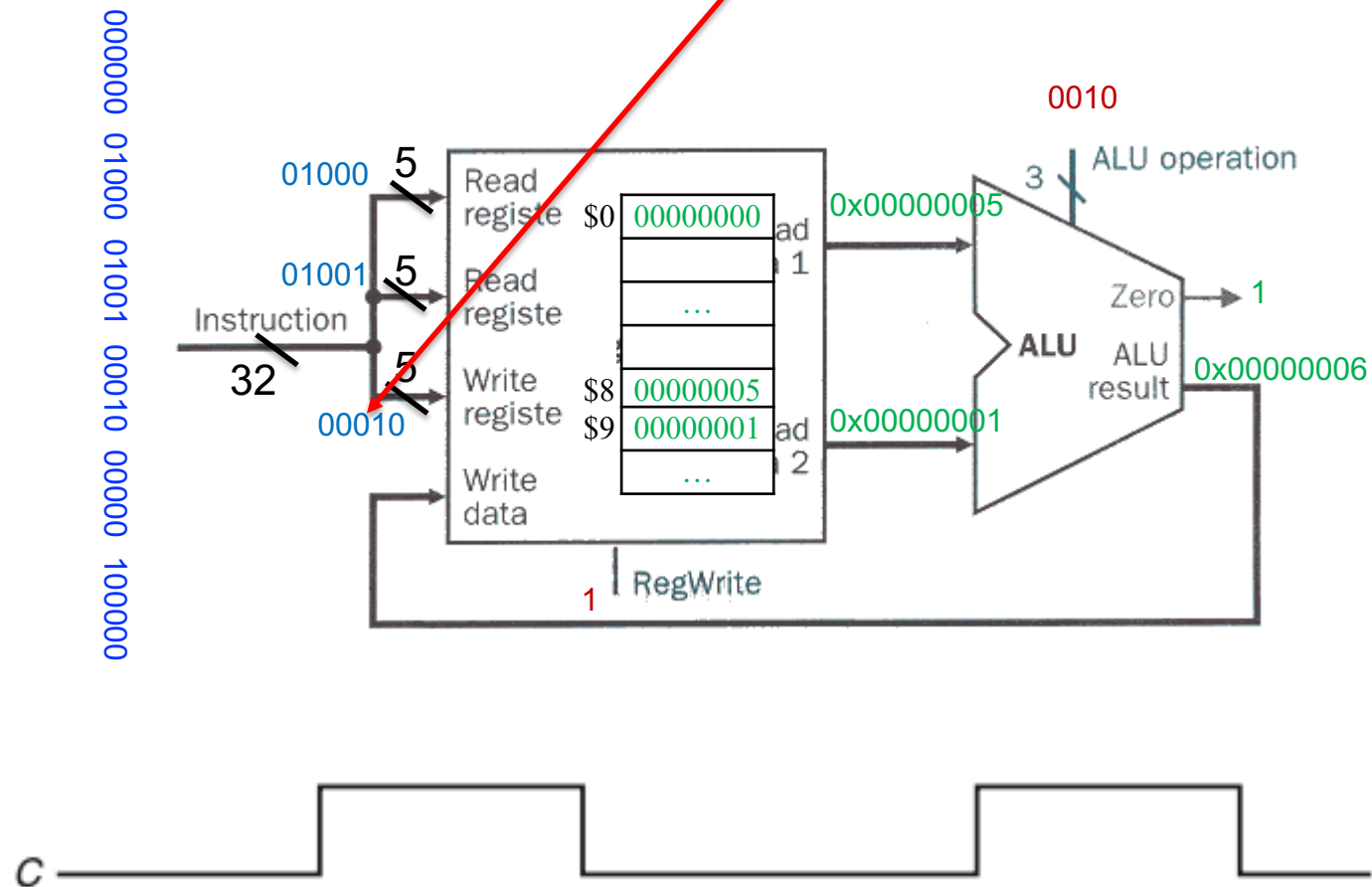
# Datapath

- Use multiplexors to stitch them together



# Datapath part 2: Execution of A/L Instructions

add \$2, \$8,\$9 000000 01000 01001 00010 00000 100000



1. *Journal of Management Studies*, 1990, 27, 1, 1-14.

**lw \$2, 4(\$8)    100011 01000 00010 0000 0000 0000 0100**

