Big Picture

- Issue 1: Fundamental concepts and principles
  - What is computer, CSE, computer architecture?
- Issue 2: ISA (HW-SW interface) design
  - Ch. 1: computer performance
  - Ch. 2: language of computer; ISA
  - Ch. 3: data representation and ALU
- Issue 3: implementation of ISA (internal design)
  - Ch. 4: processor (data path, control, pipelining)
  - Ch. 5: memory system (cache memory)
- Short introduction to parallel processors

Big Picture

- Part 3: implementation of ISA

  "Given ISA, what is a good implementation?"

  - Ch. 4: processor
    - Simplified version (fetch-decode-execution)
    - Pipelined version
  - Ch 5: memory system design
    - Cache memory (part of processor)

❖ 기능이 아니라 성능 (efficient implementation)

Class Topics (클래스 홈페이지 참조)

- Part 1: Fundamental concepts and principles
- Part 2: 빠른 컴퓨터를 위한 ISA design
  - Topic 1 Computer performance and ISA design (Ch. 1)
  - Topic 2 RISC (MIPS) instruction set (Chapter 2)
  - Topic 3 Computer arithmetic and ALU (Chapter 3)
- Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)
  - Topic 4 Processor design (Chapter 4)
    - Single cycle implementation
    - Pipelined implementation
  - Topic 5 Memory system design (Chapter 5)

Chapter 4

The Processor (Implementing ISA)

Part 1: Single Cycle Design

Some of authors' slides are modified

Introduction

- High-level organization (and low-level circuits design)
  - Determine CPI (and clock cycle time)
    - ISA determines IC (and affect CPI and clock cycle)
- We will examine two MIPS implementations
  - Single cycle implementation
  - Multi-cycle implementation (concept only)
  - Pipelined implementation
- Datapath and control (VLSI chip design)

Introduction

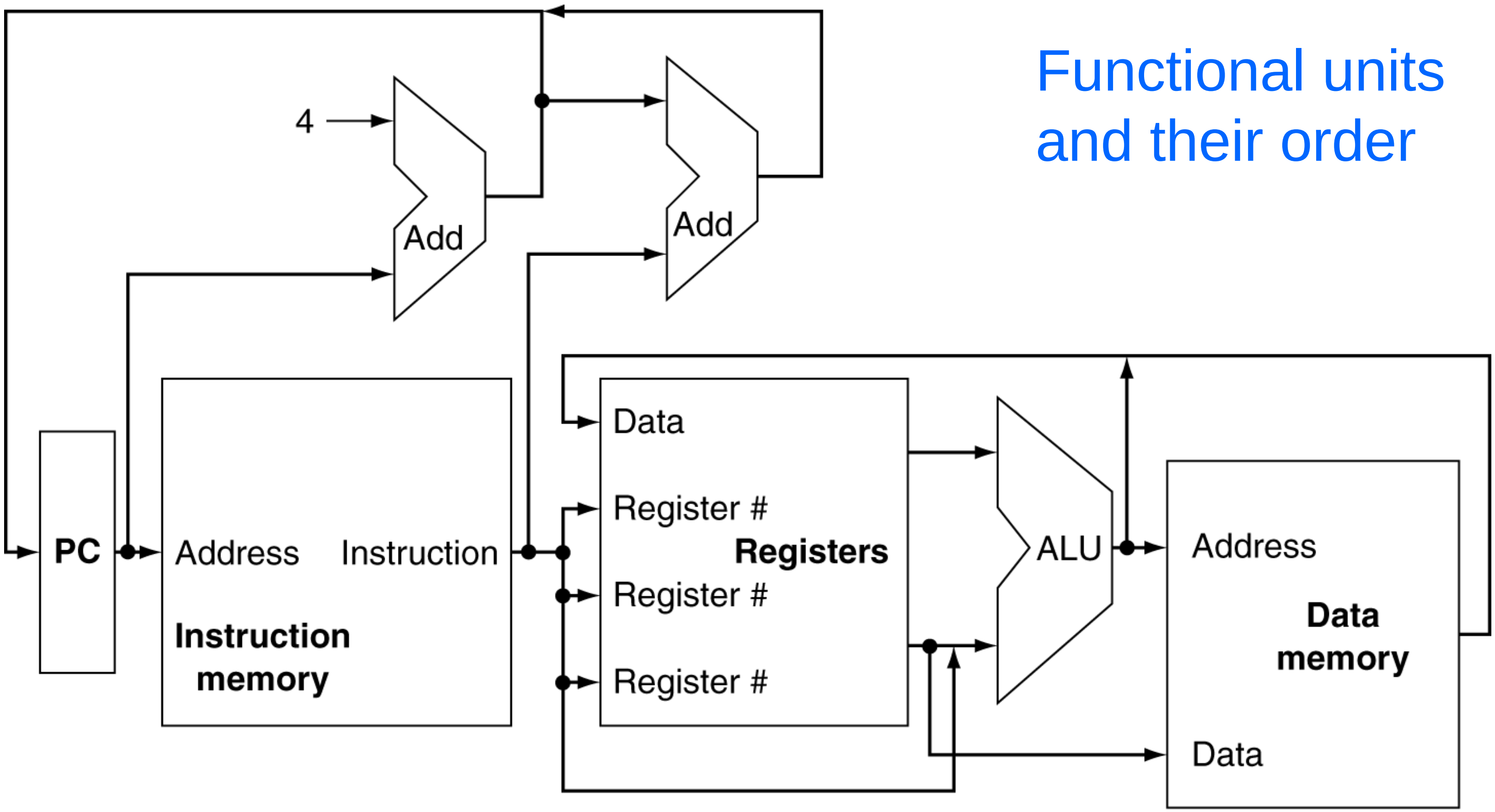
- Simple subset, shows most aspects
  - Memory-reference: **lw, sw**
  - Arithmetic-logical: **add, sub, and, or, slt**
  - Control transfer: **beq, j**
- Generic Implementation: repeat the following
  - Instruction fetch (IF) // 공통
  - Instruction decode (ID) // 공통
    - Use the instruction to decide exactly what to do
  - Instruction execute (EX)

Instruction Execution (미리보기)

- Fetch instruction: PC → instruction memory, PC←PC+4
- Decode instruction (opcode)
  - Register numbers → register file, read registers (?)
- Execute instruction: depend 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

MIPS CPU Overview

CPU Overview (Schematic)

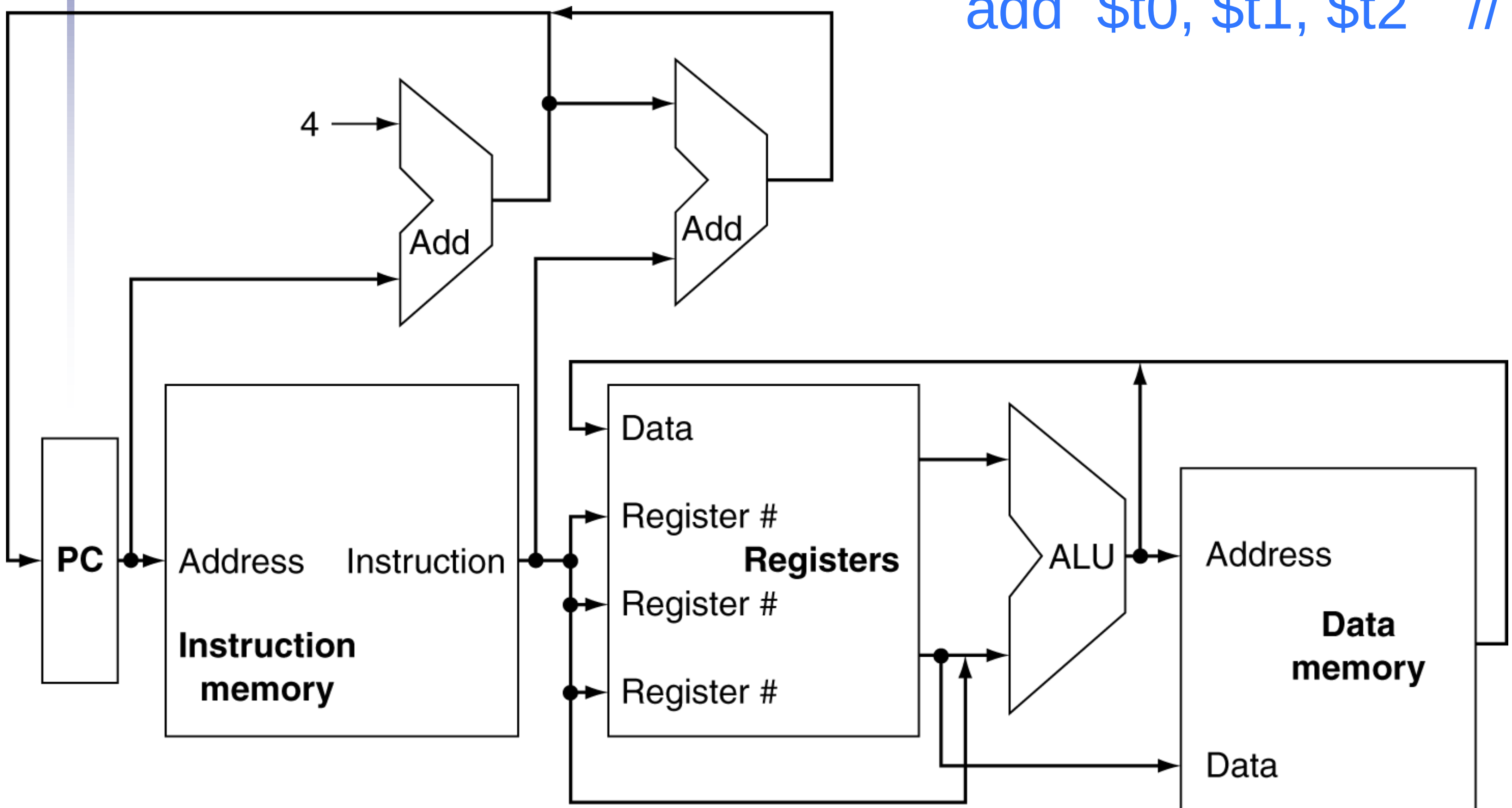


Functional units and their order

CPU Overview

op	rs	rt	rd	shamt	funct

add $t0, $t1, $t2   // R-type



CPU Overview

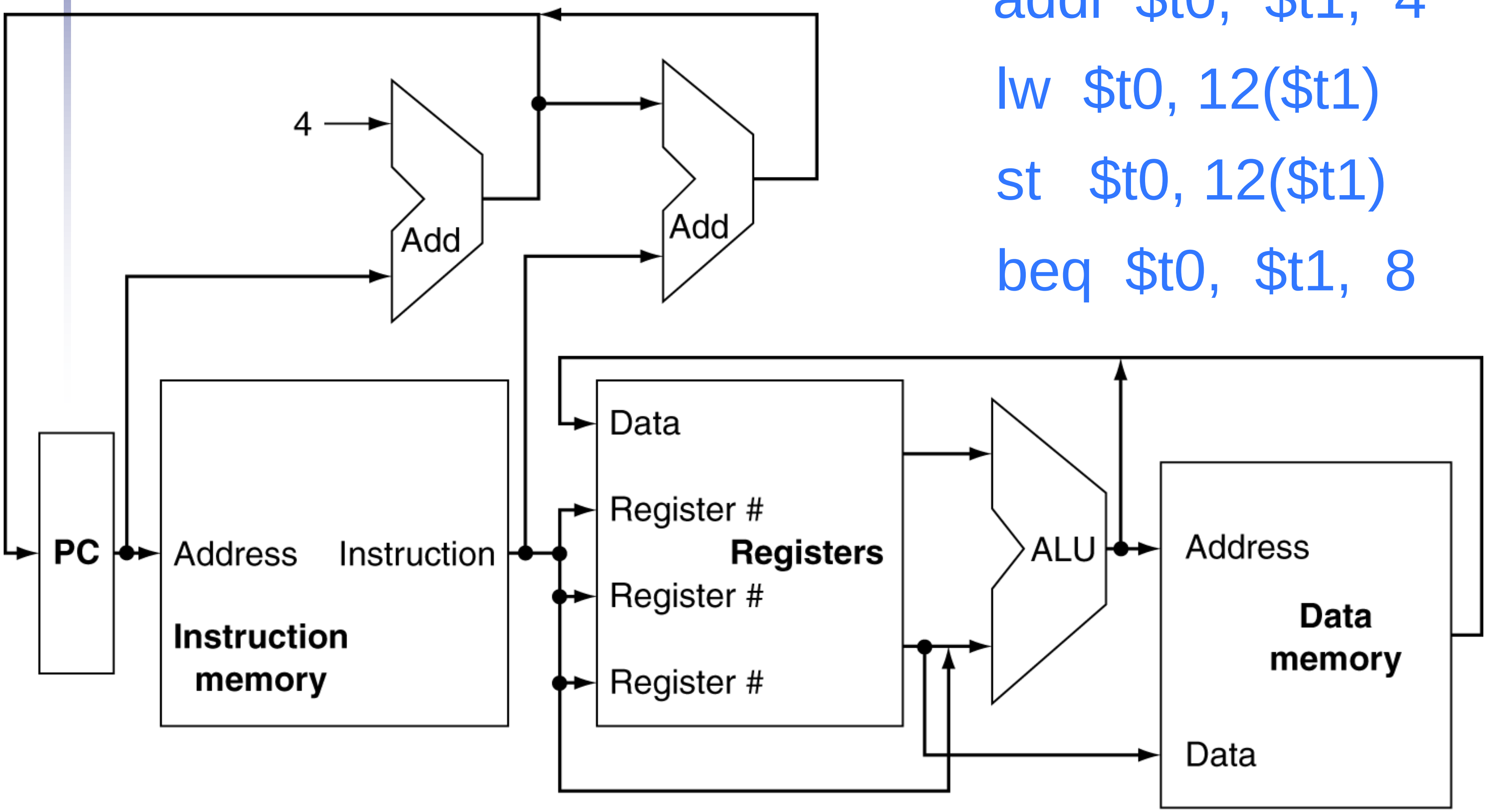
op	rs	rt	16 bit address

addi $t0, $t1, 4   // I-type

lw $t0, 12($t1)

st $t0, 12($t1)

beq $t0, $t1, 8

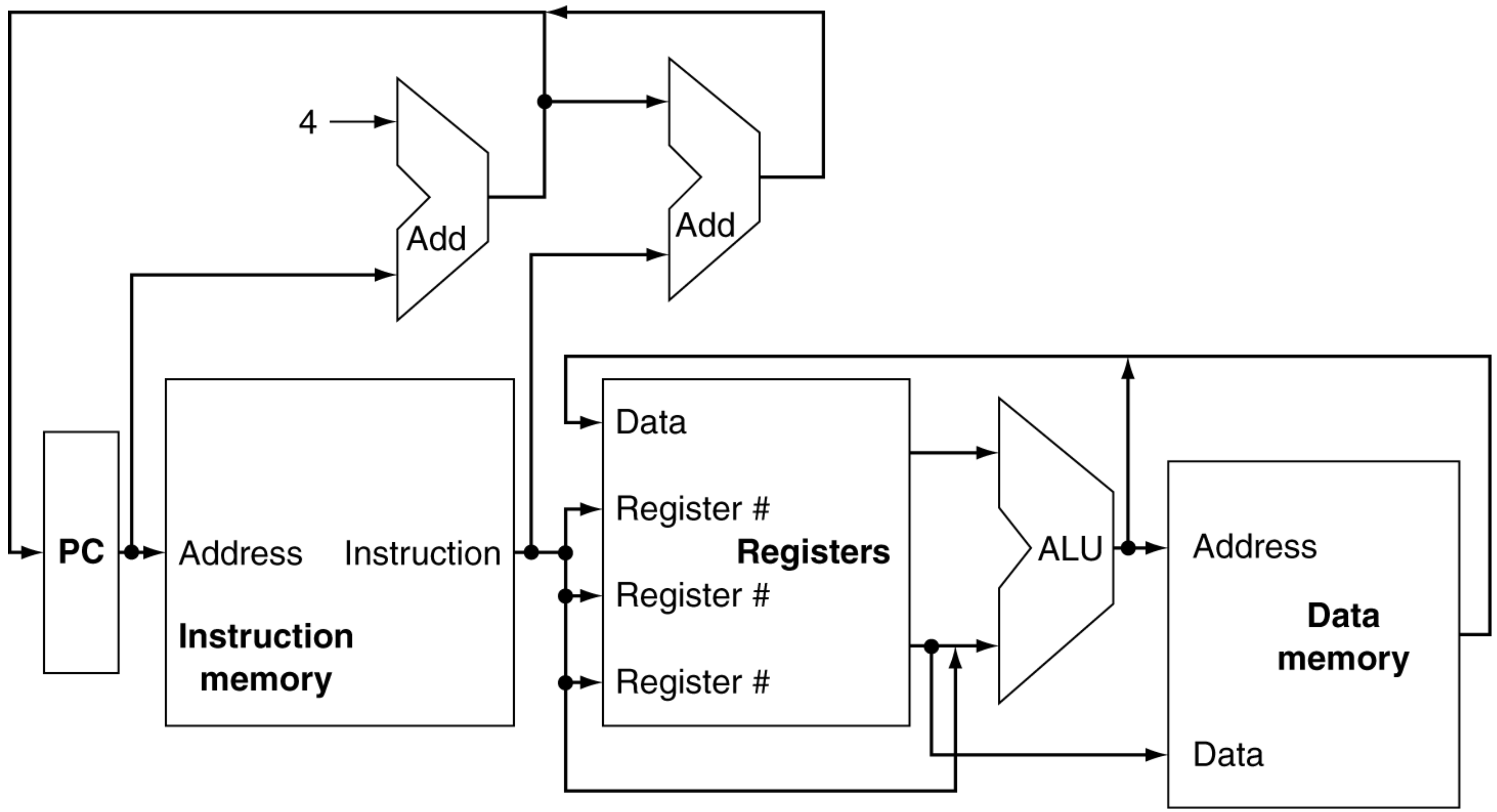


Executing MIPS Instructions (부연)

- ALU
  - add $t0, $t1, $t2 // R-type
  - addi $t0, $t1, 4 // I-type
- Data transfer
  - lw $t0, 12($t1) // I-type
  - st $t0, 12($t1)
- Branch
  - beq $t0, $t1, 8 // I-type

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit offset		
J	op	26 bit address				

High-Level Implementation (Schematic)

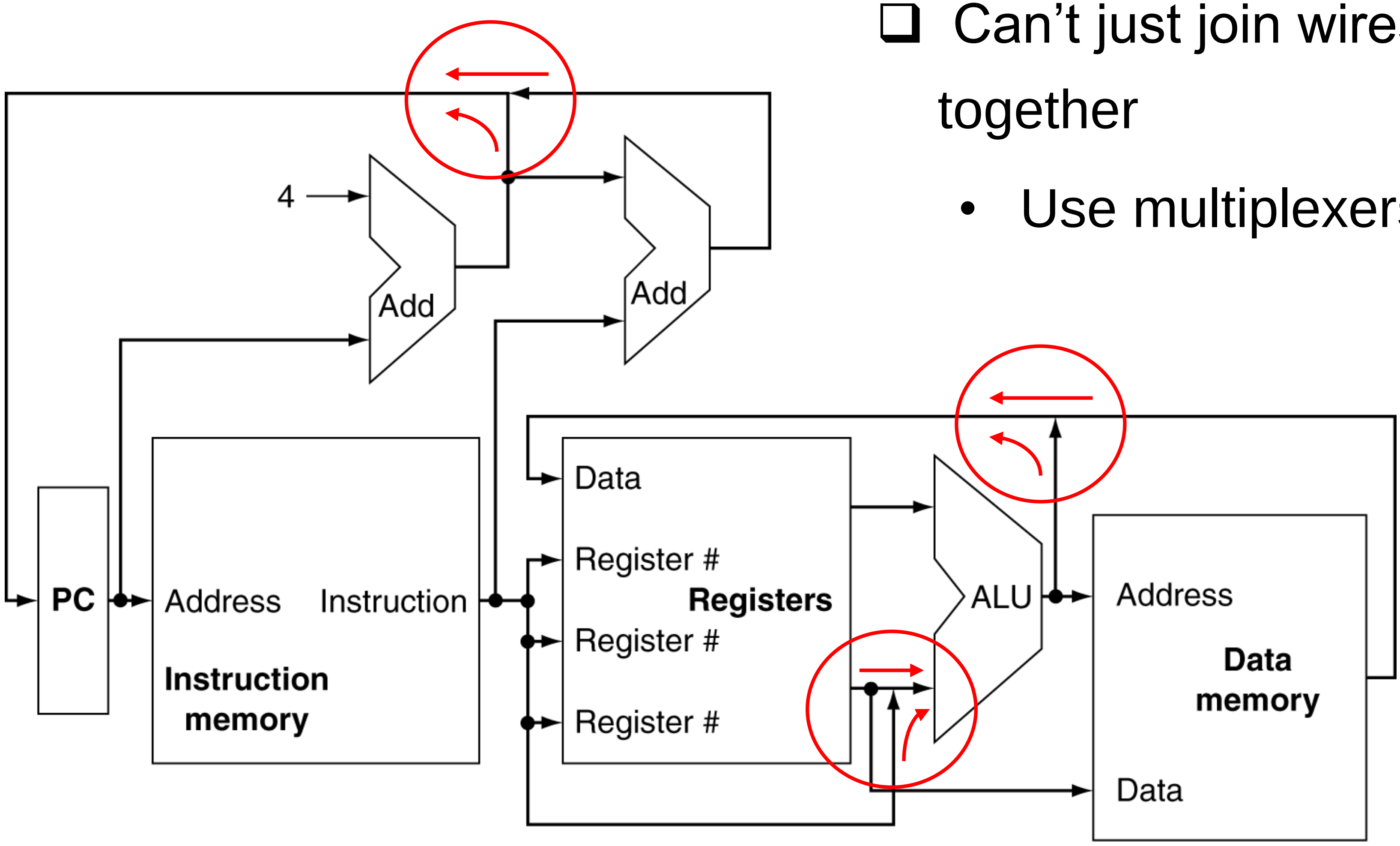


- Does this look familiar? (It's just ISA)
- ISA determines functional units and their order

Instruction Decode

- Where is it in the datapath?
  - Not shown (will come back to it)

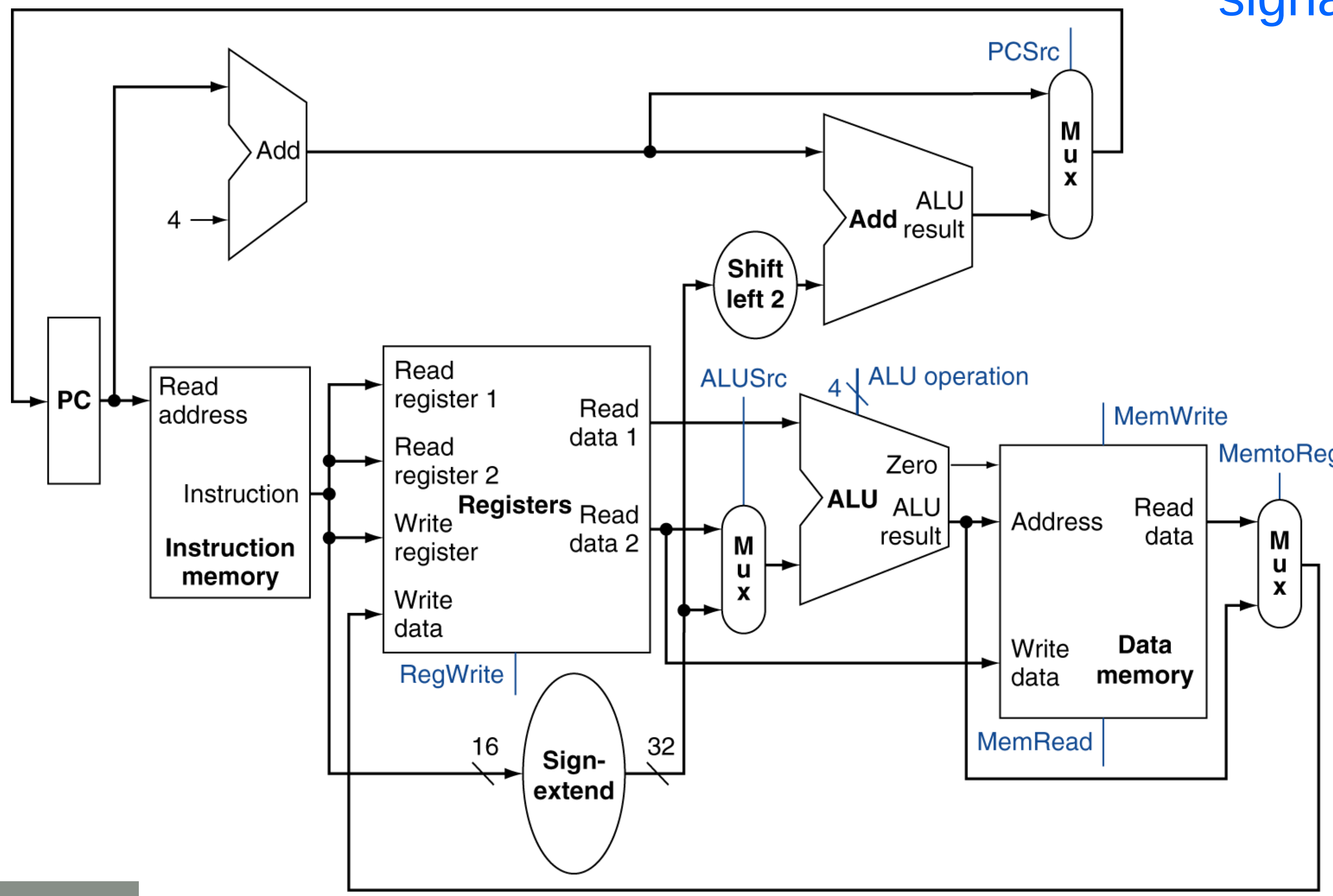
Multiplexers



- Can't just join wires together
  - Use multiplexers

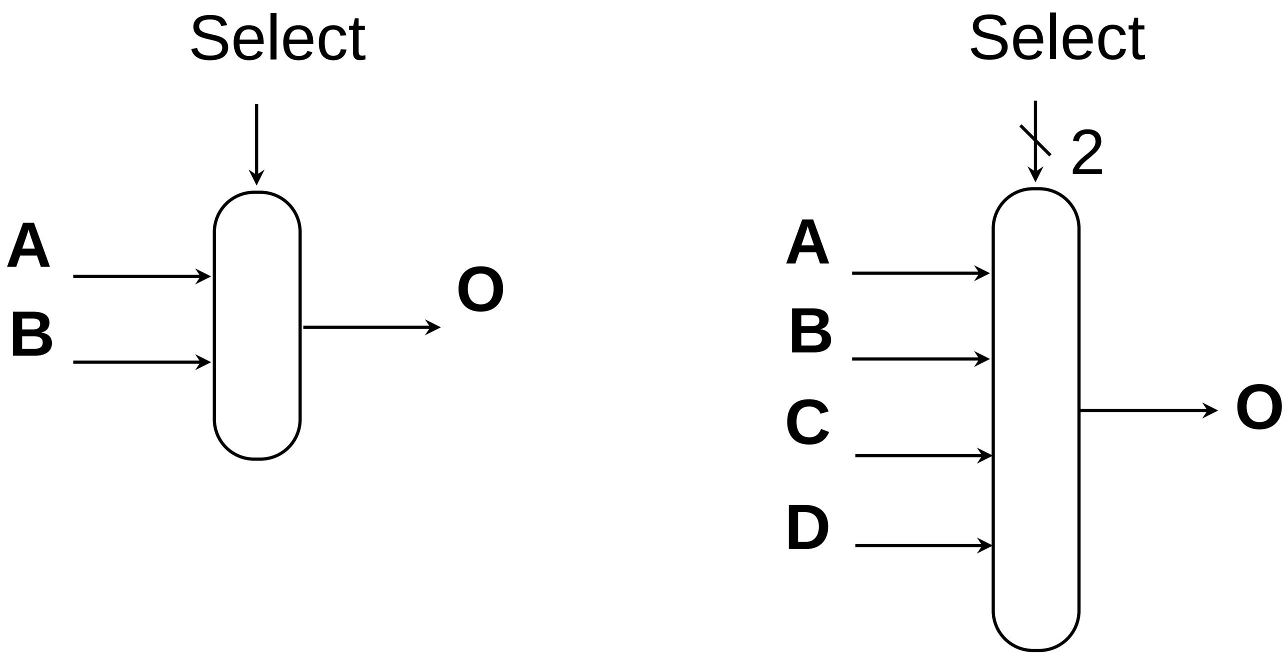
Full Datapath (미리보기)

colored select signals



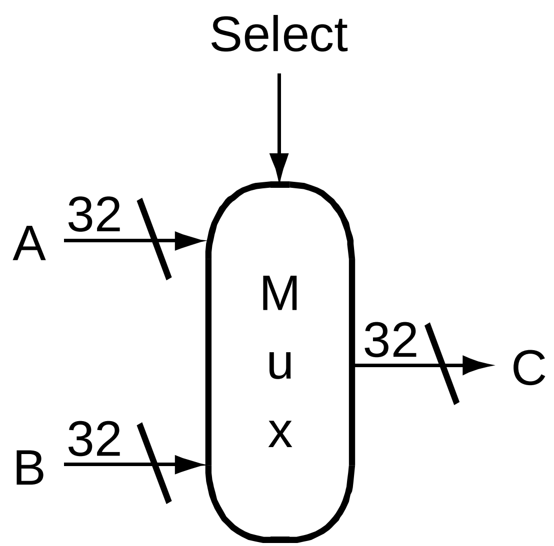
Multiplexers (복습)

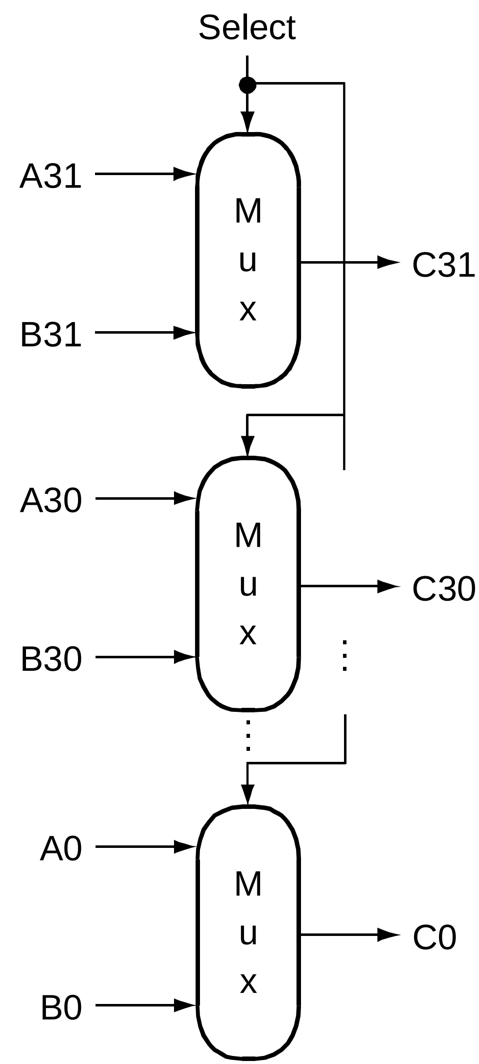
- 2-to-1 MUX, 4-to-1 MUX



Multiplexers (복습)

- 32 of 2-to-1 mux



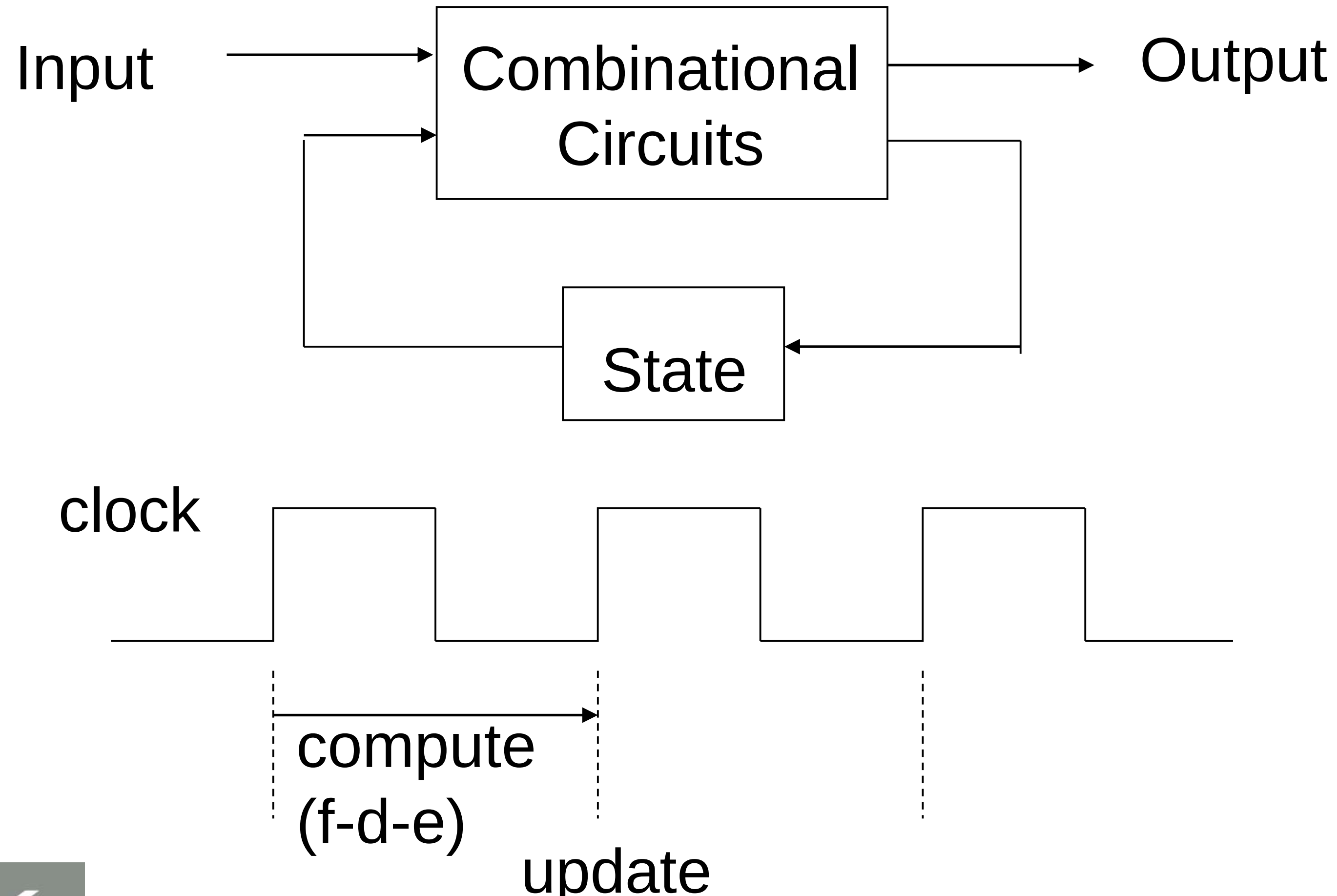


Digital Logic Design (복습)

- Given: AND, OR, NOT
- Design
  - Combinational logic circuits
    - Decoders, mux, …, ALUs
  - Sequential logic circuits
    - Latches, flip-flops, registers, counters, …, CPUs
- Notion of abstraction
- VLSI 개발 환경 (비교: software 개발 환경)

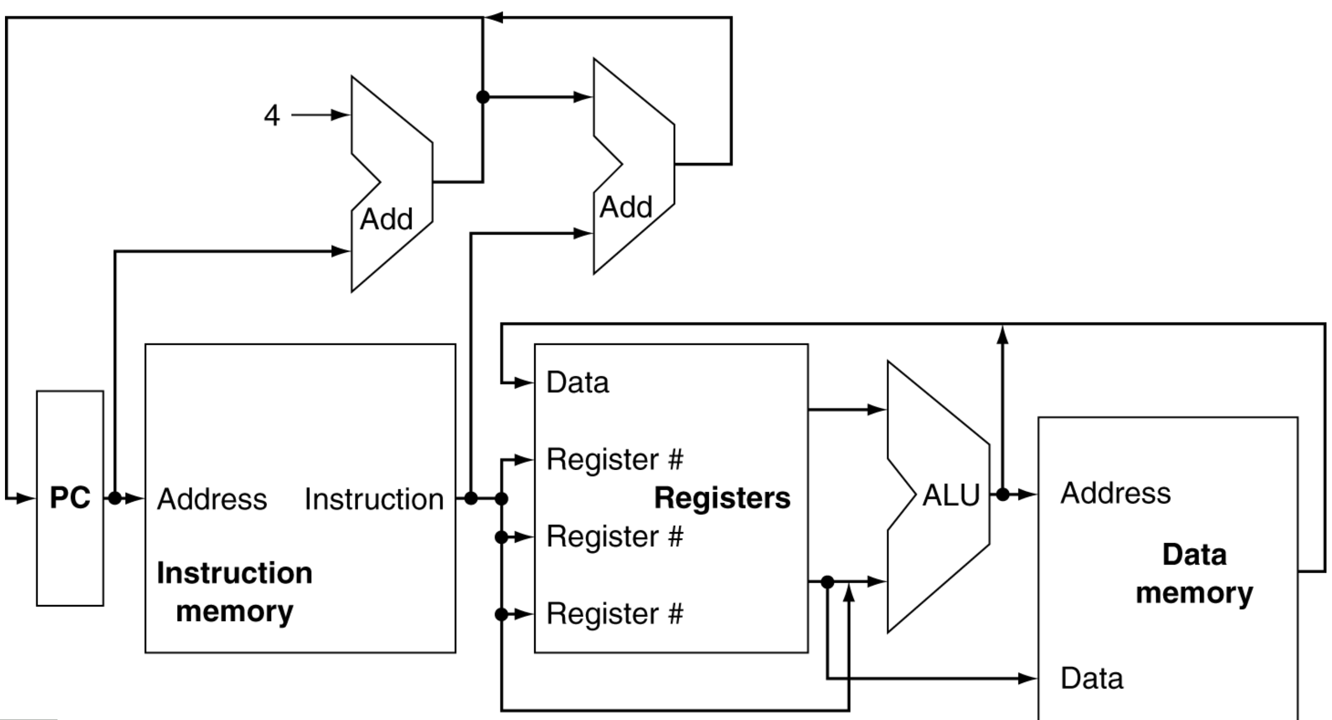
Synchronous Sequential Logic (복습)

- Meaning of single cycle implementation



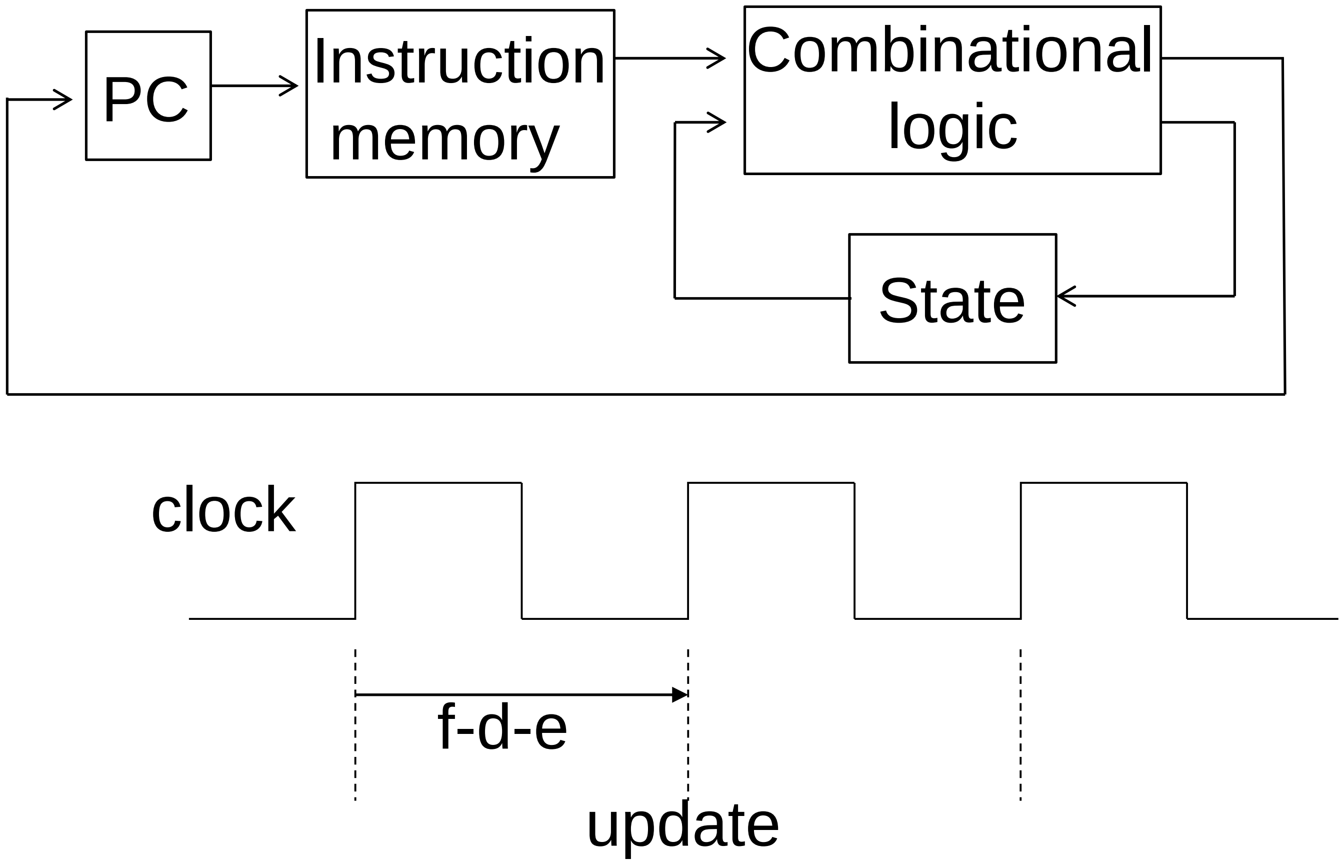
CPU: Synchronous Sequential Logic

- What are the states?
  - Result of “fetch-decode-execute” updated at the end of each clock cycle



CPU: Providing Input at High Speed

- 1GHz (light speed); use PC to get new input by itself



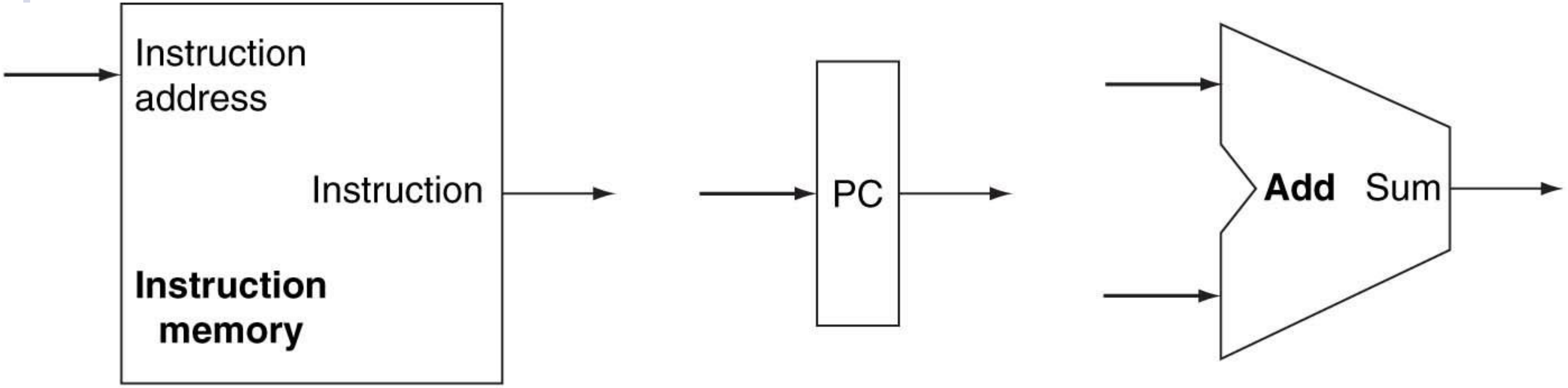
Let’s Build a Datapath

which can execute MIPS instructions,

using smaller building blocks

Building Blocks for IF (1)

- Functional units (abstractions) we need for each instruction

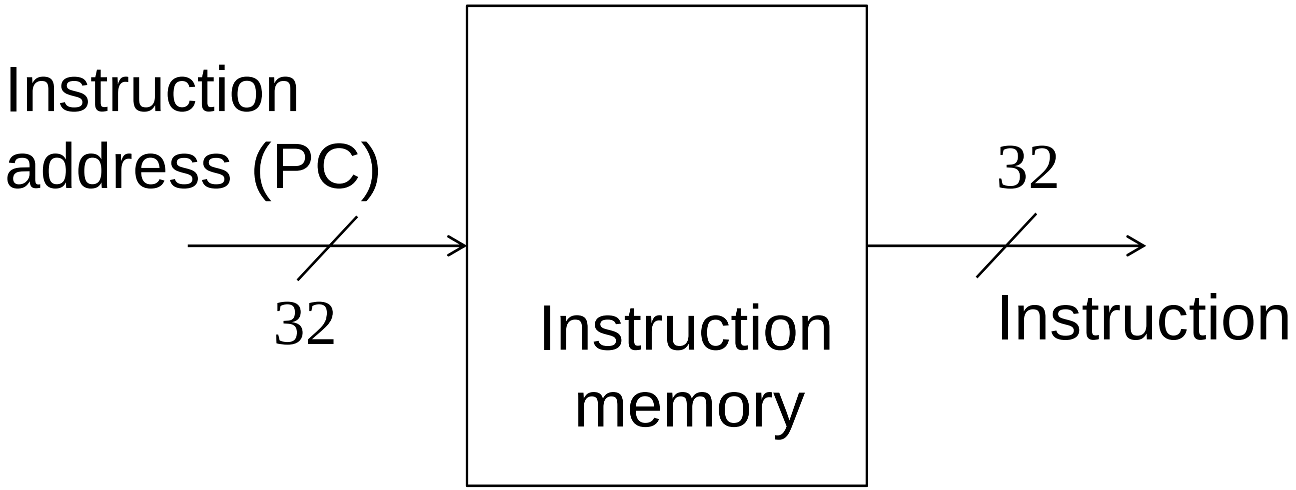


a. Instruction memory

b. Program counter

c. Adder

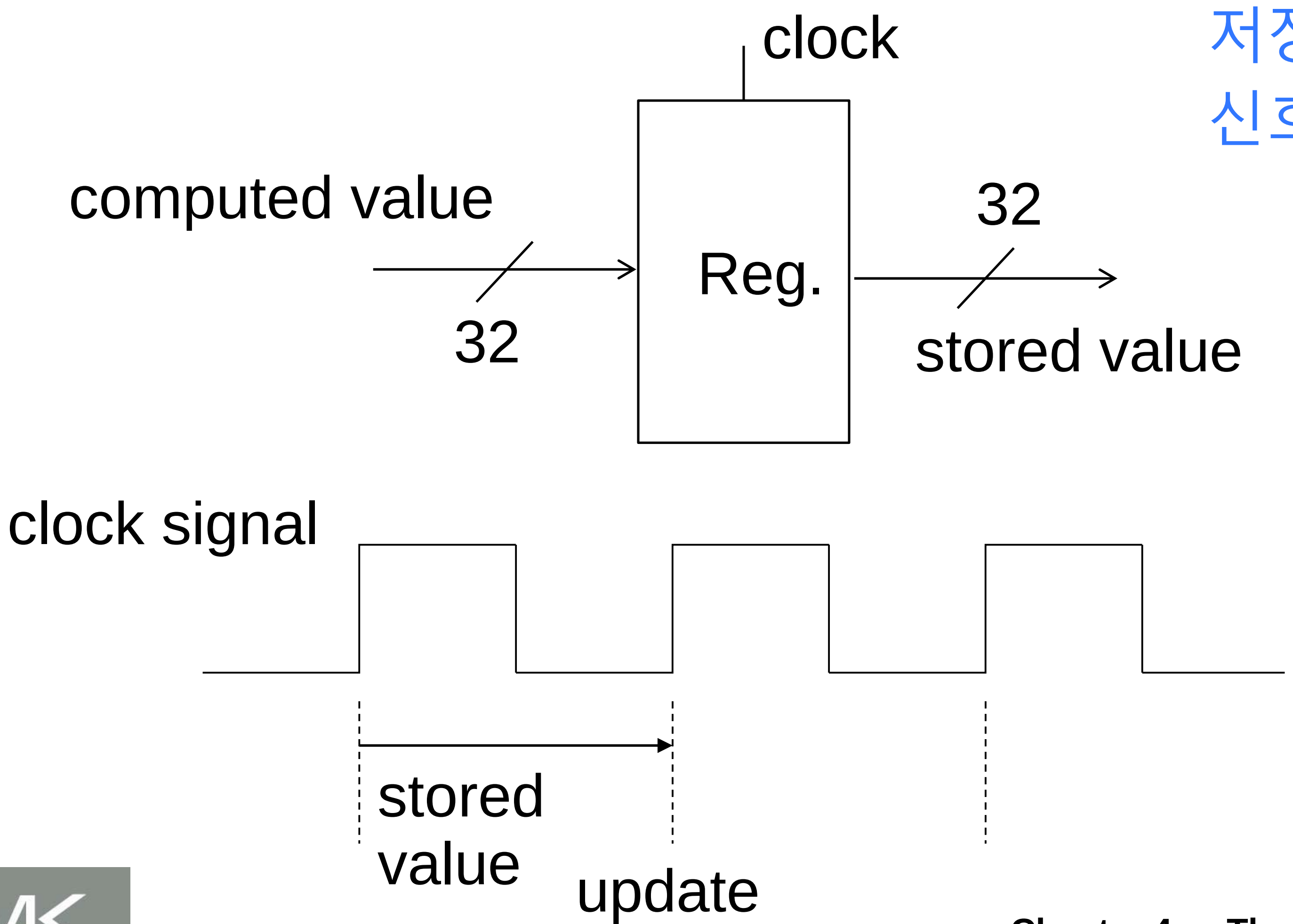
Instruction Memory



- 여기서는 프로그램이 고정되었다고 생각 (read only)
  - 변경 원하면, write 기능 추가

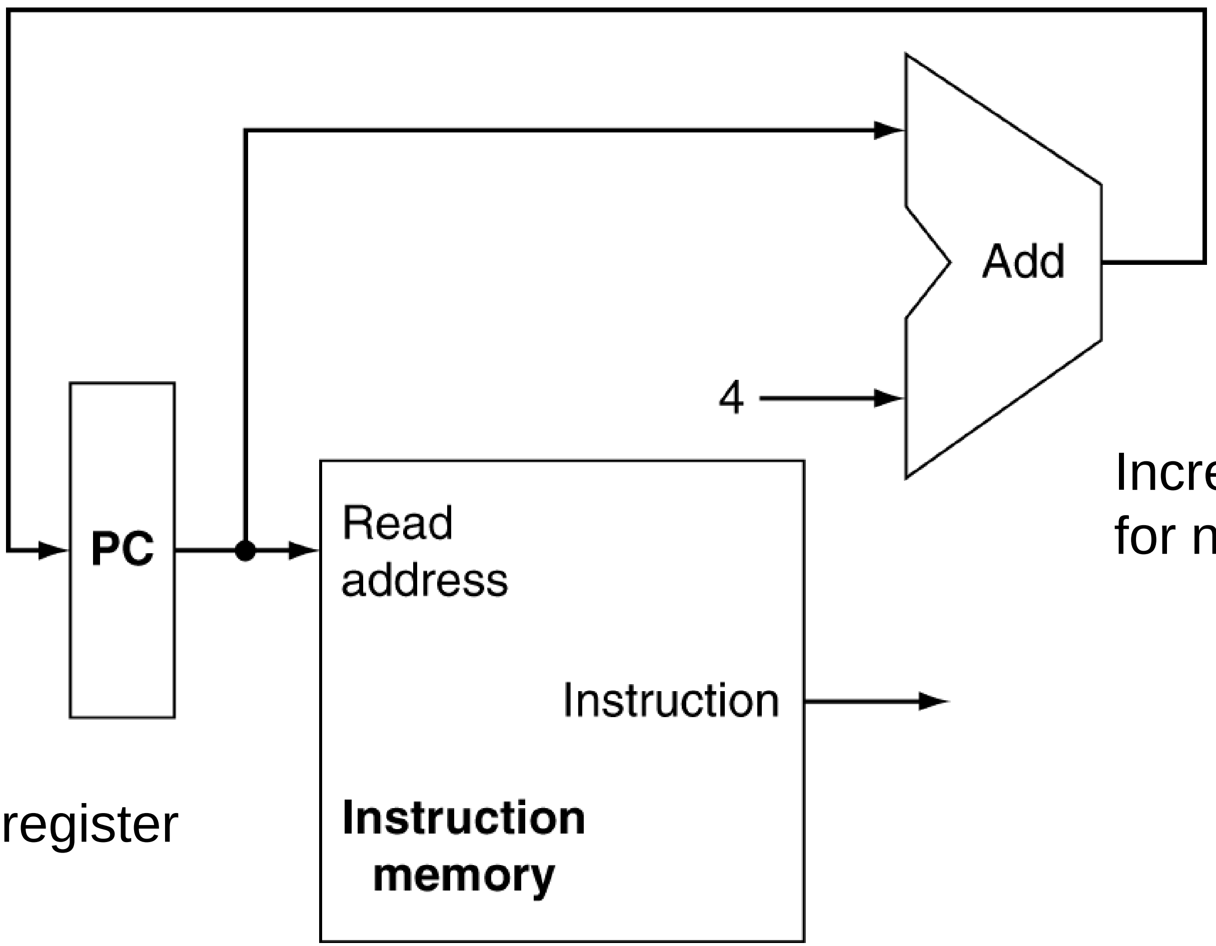
Registers (including PC)

- Sequential blocks, has states, in sync with clock



저장 및
신호 insulation

Instruction Fetch



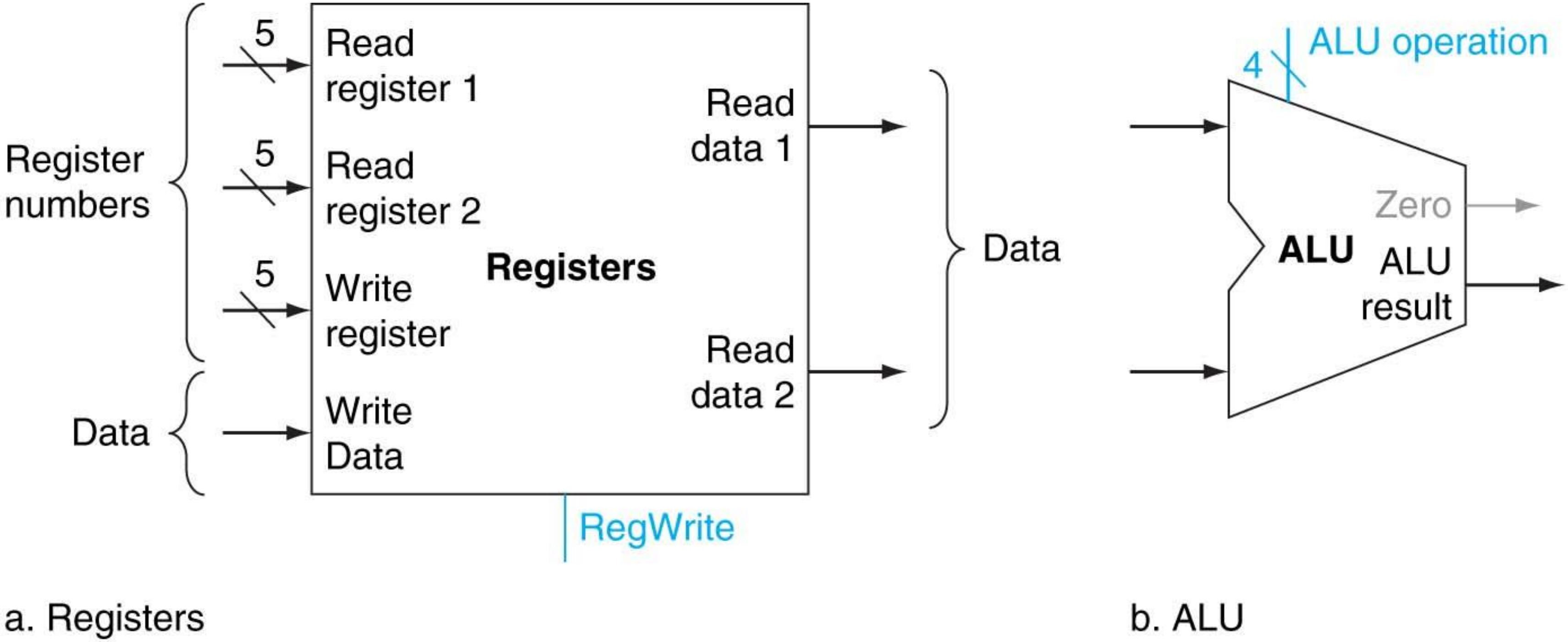
Increment by 4
for next instruction

32-bit register

- 모든 instruction 에 공통

Building Blocks for R-Type (2)

- Functional units (abstractions) we need for each instruction



add \$3, \$1, \$2

- (colored) select/enable signals – how to determine?

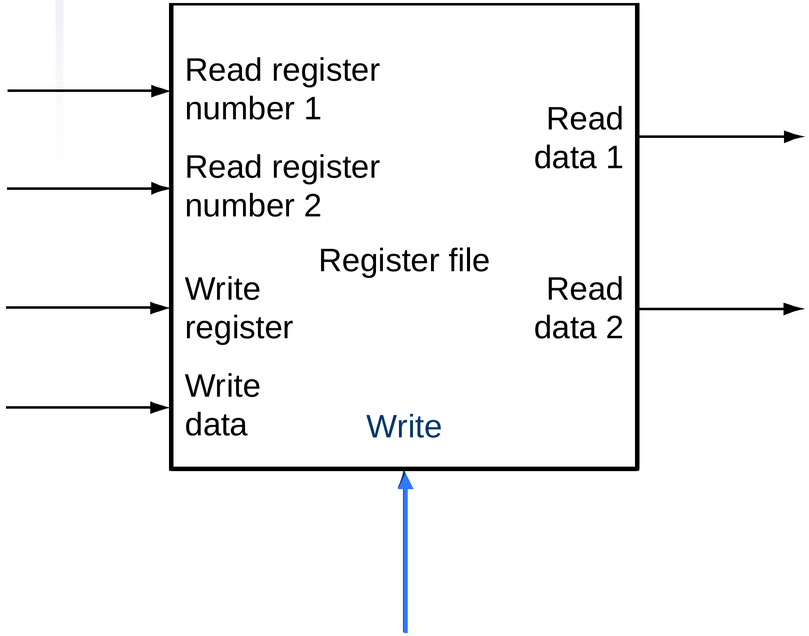
Register File

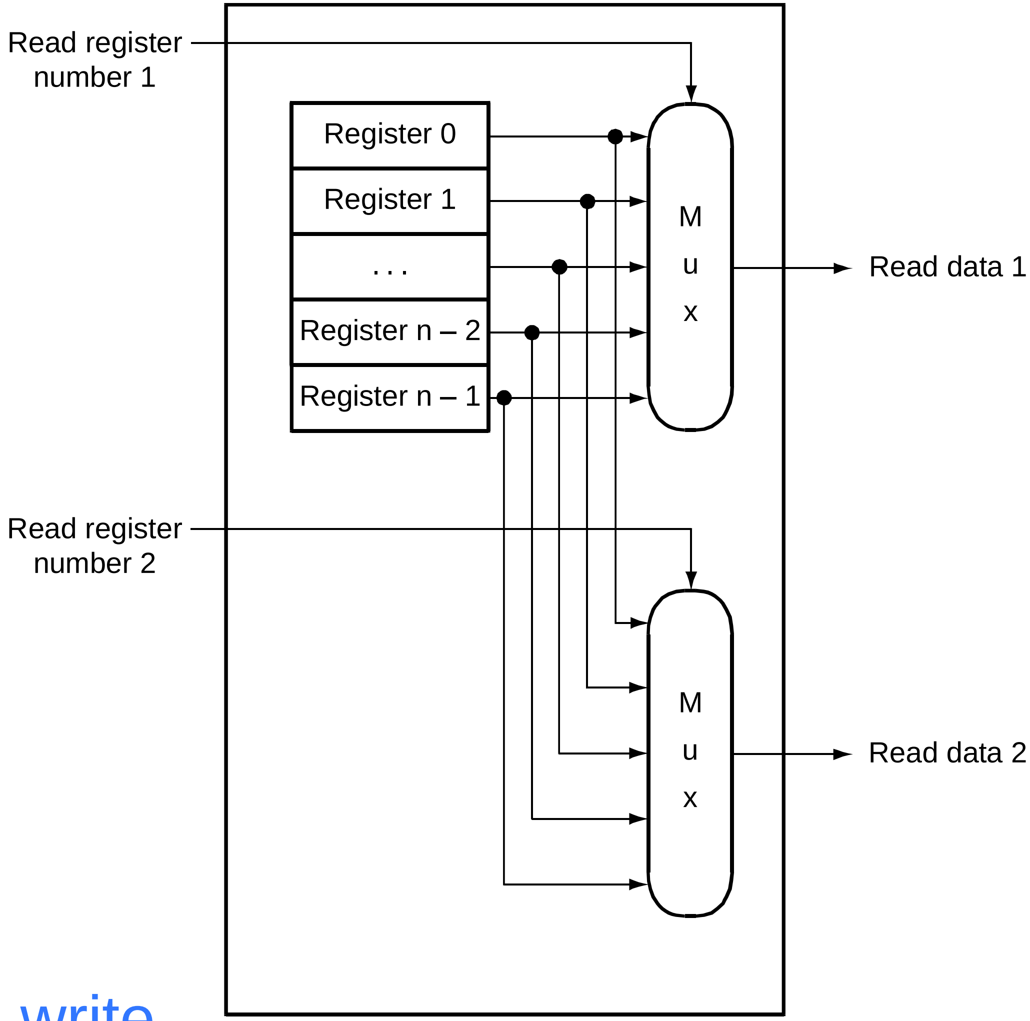
- Register read

add $3, $1, $2

beq $1, $2, 8

32-to-1 mux
(5-bit register no.)





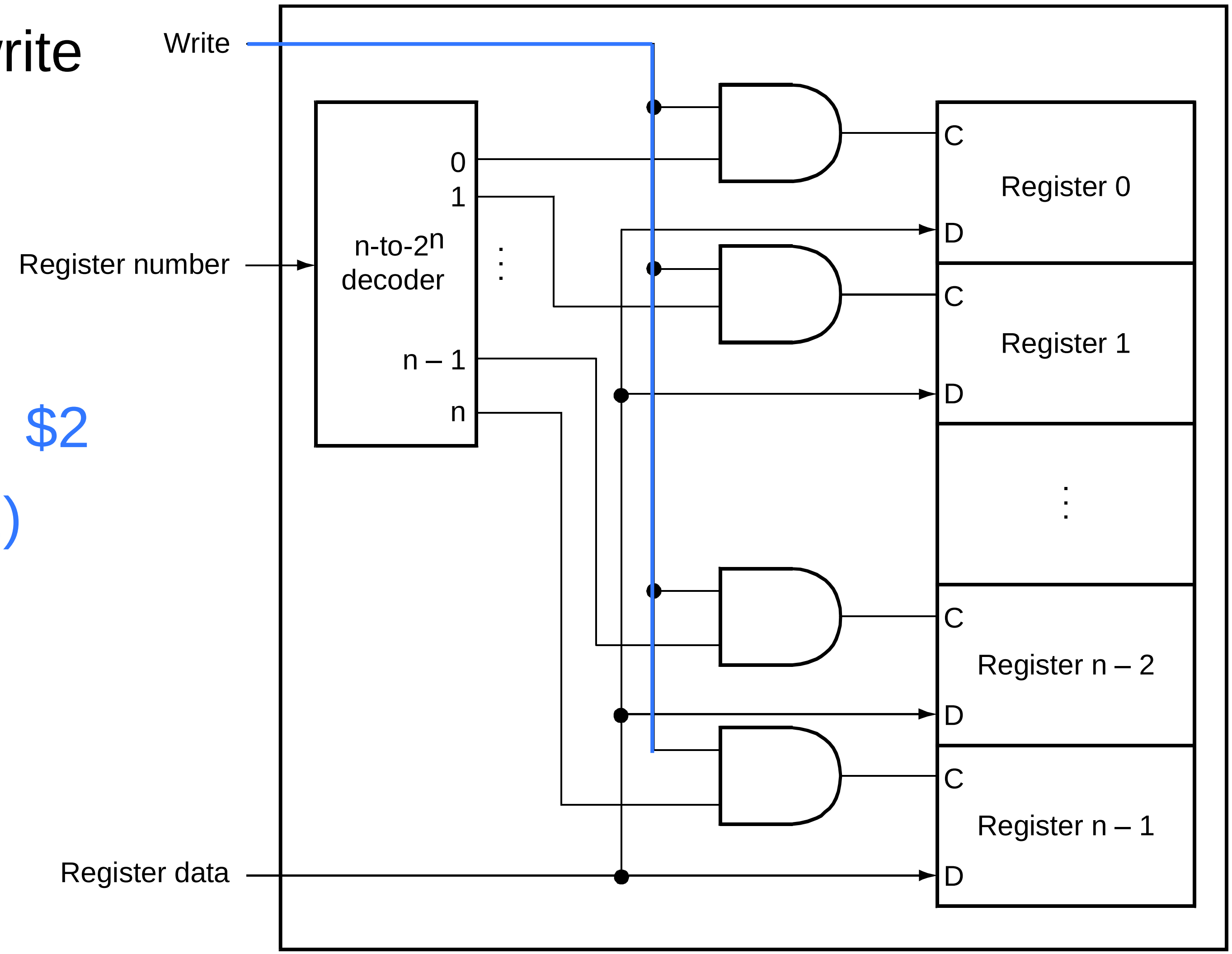
2 reads and 1 write

Register File

5-to-32 decoder
(5-bit register no.),
only 1 of 32 outputs: “1”

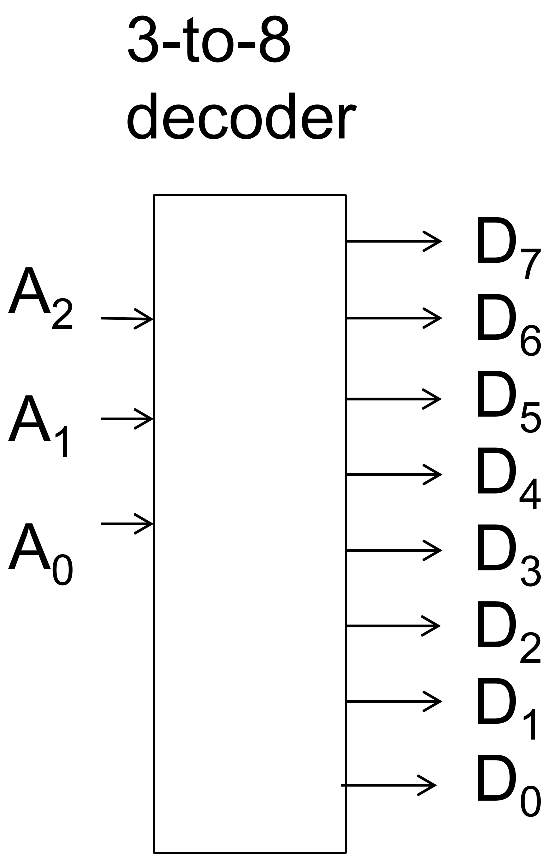
- Register write

add $3, $1, $2
lw $3, 8($2)



3-to-8 Decoder (복습)

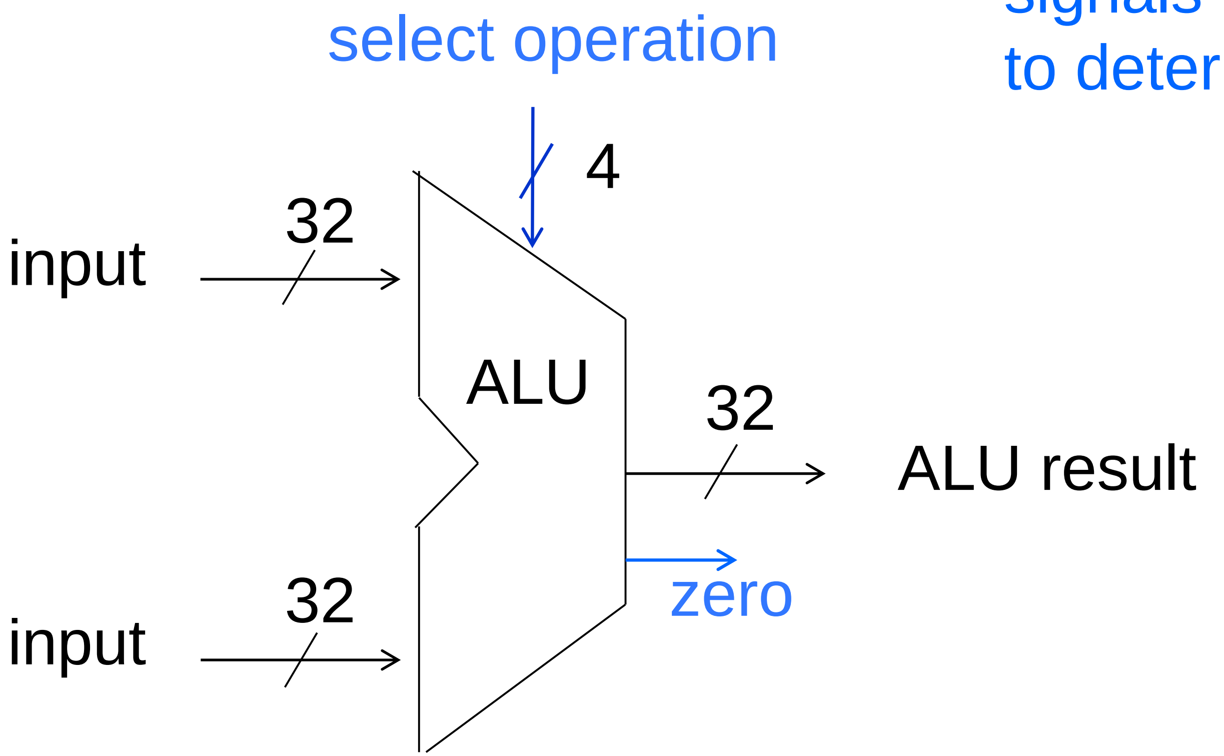
A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



- 5-to-32 decoder, 2-to-4 decoder, …

ALU

- Combinational blocks, no states

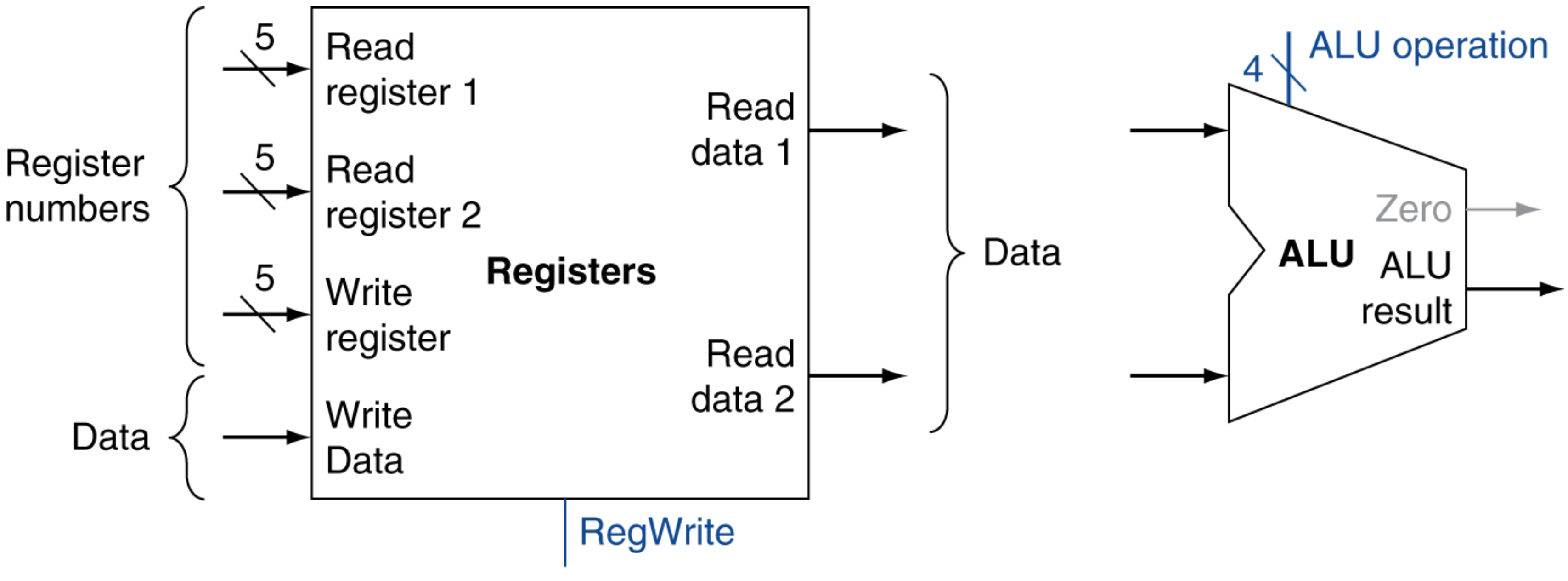


colored select signals (how to determine?)

R-Format Instructions

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

add $3, $1, $2

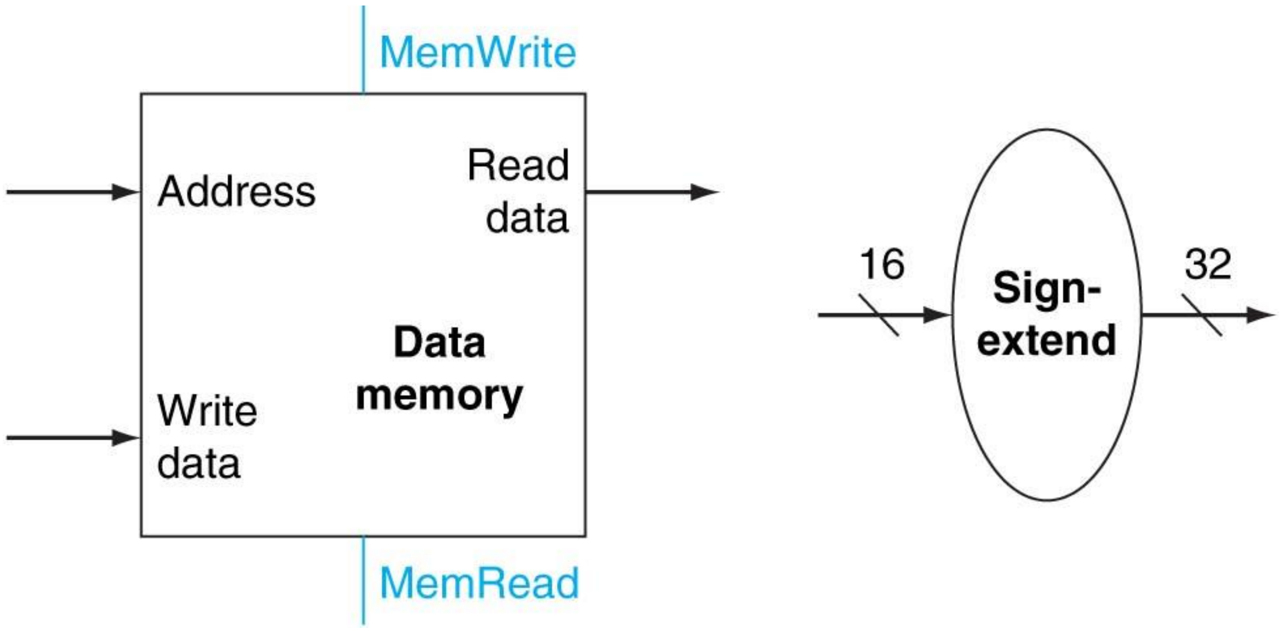


a. Registers

b. ALU

Building Blocks for "lw/sw" (3)

- Functional units (abstractions) we need for each instruction

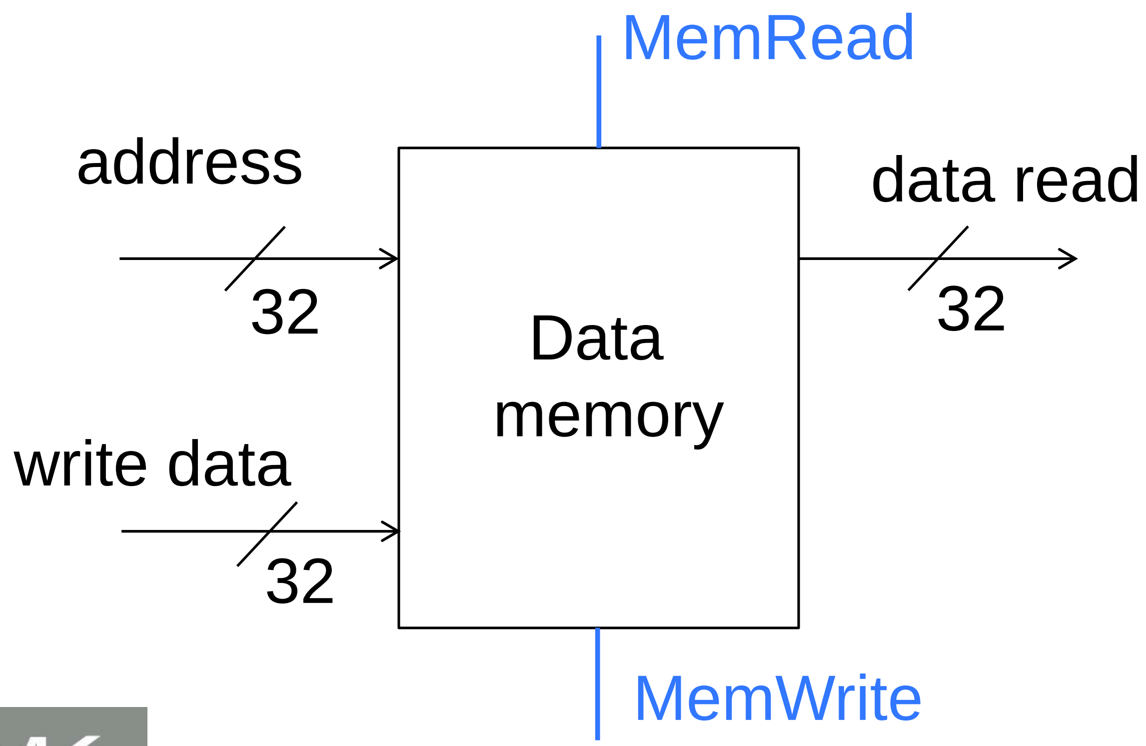


a. Data memory unit

b. Sign extension unit

Data Memory

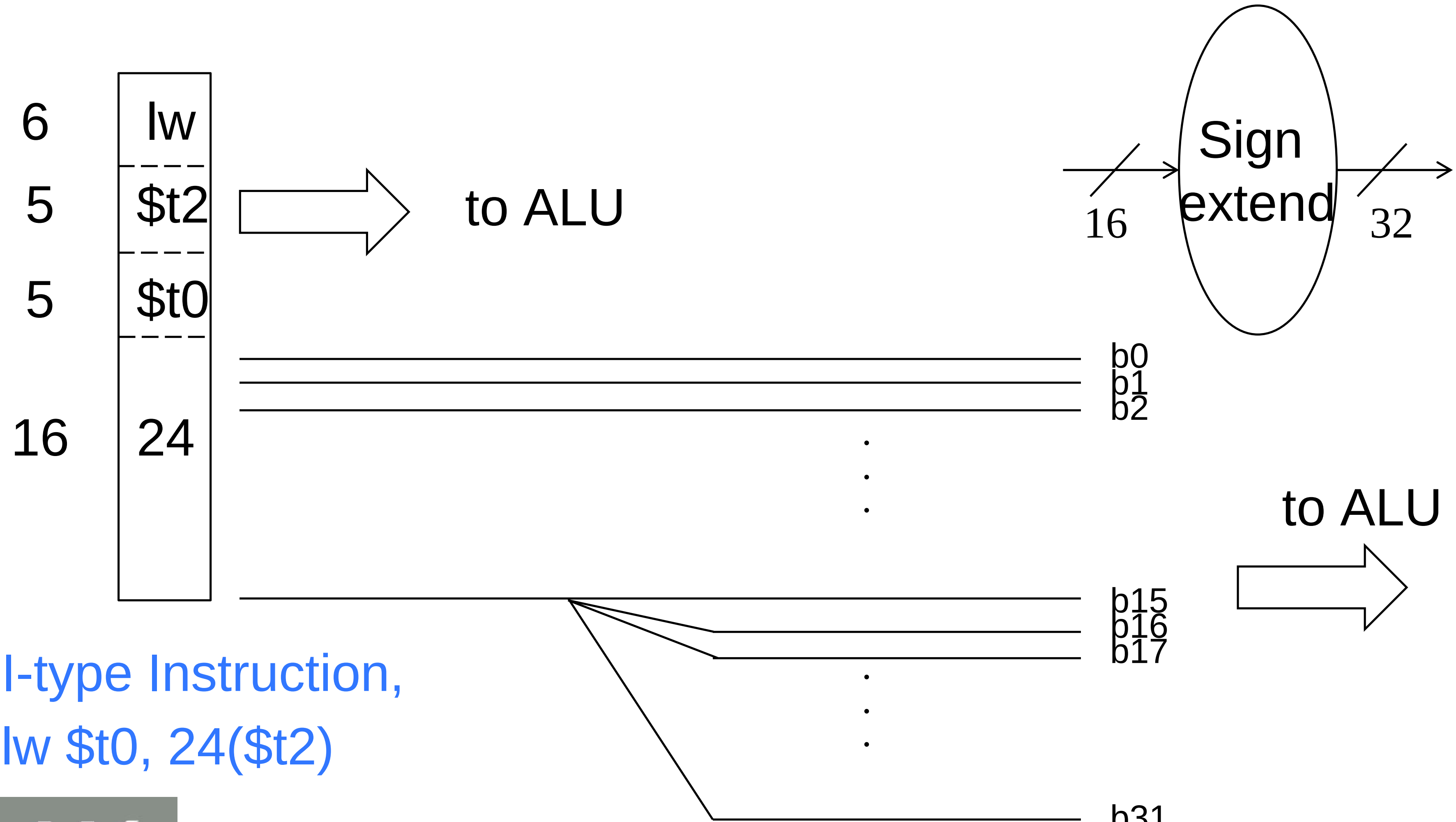
- Memory read (MemRead = 1) // load
- Memory write (MemWrite = 1) // store



Colored enable signals (how to determine?)

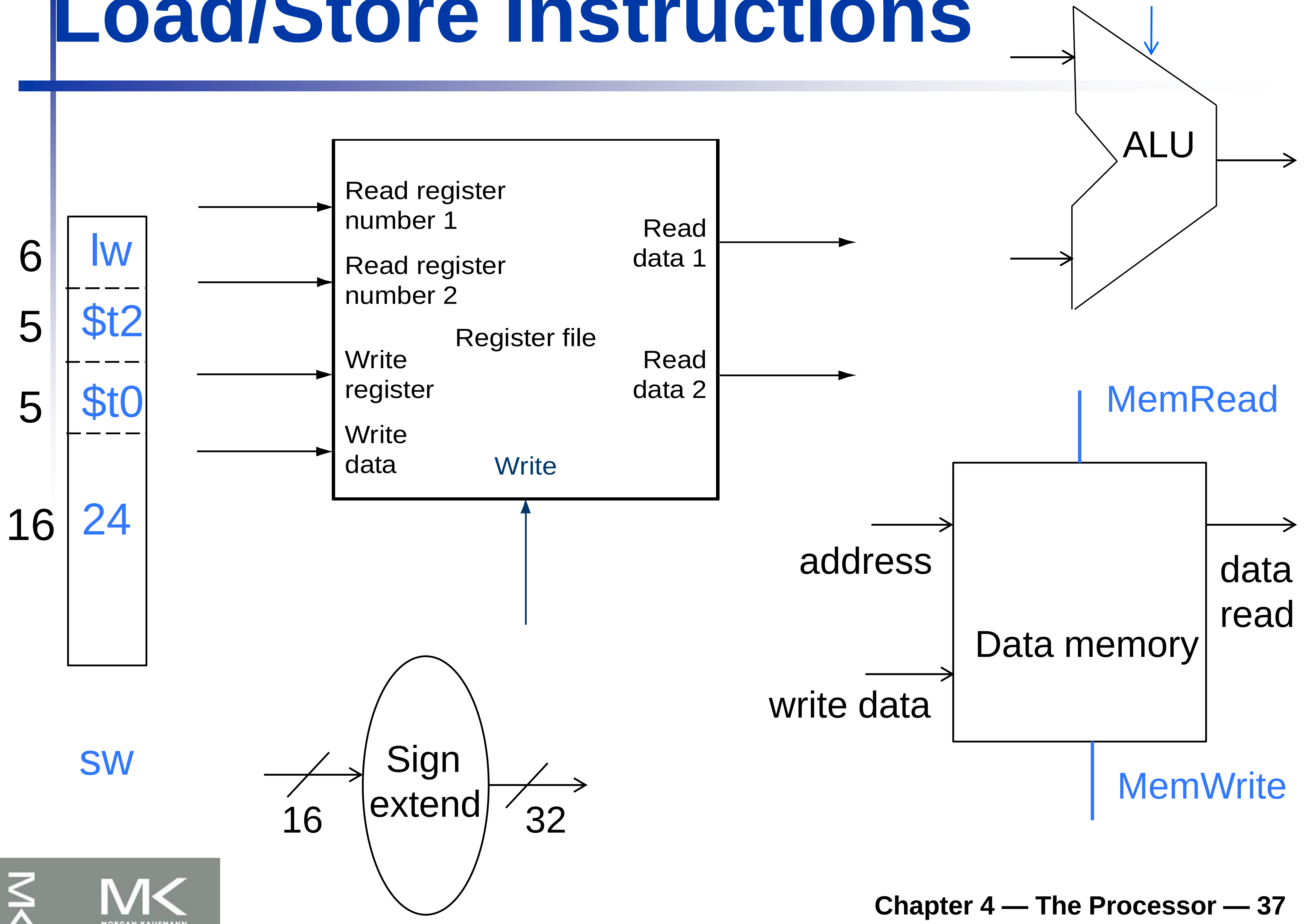
Sign Extension

- 16-bit immediate extended to 32-bit before ALU operation



I-type Instruction,
lw $t0, 24($t2)

Load/Store Instructions

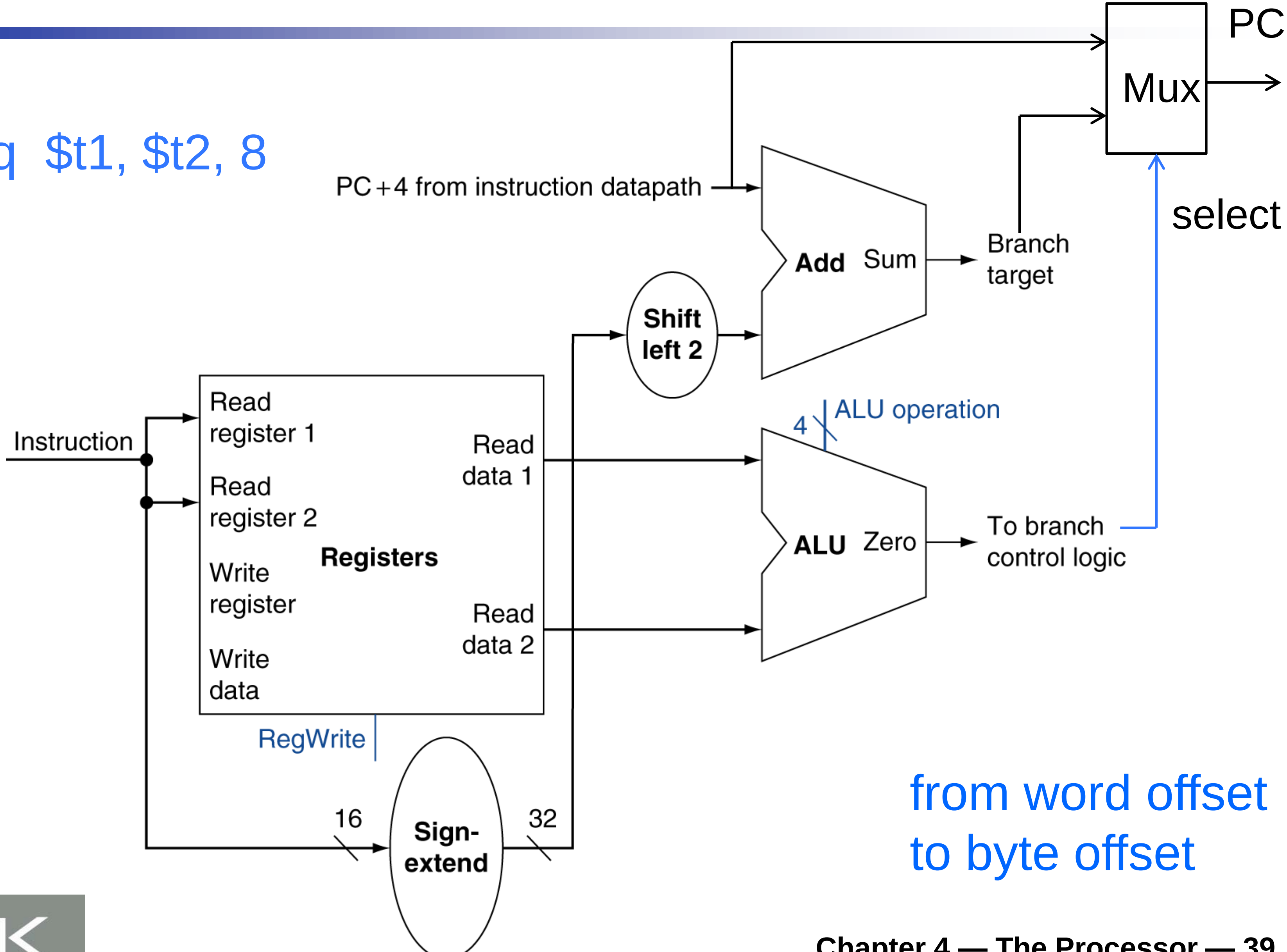


Load/Store Instructions (부연)

- 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

Branch Instructions

beq $t1, $t2, 8



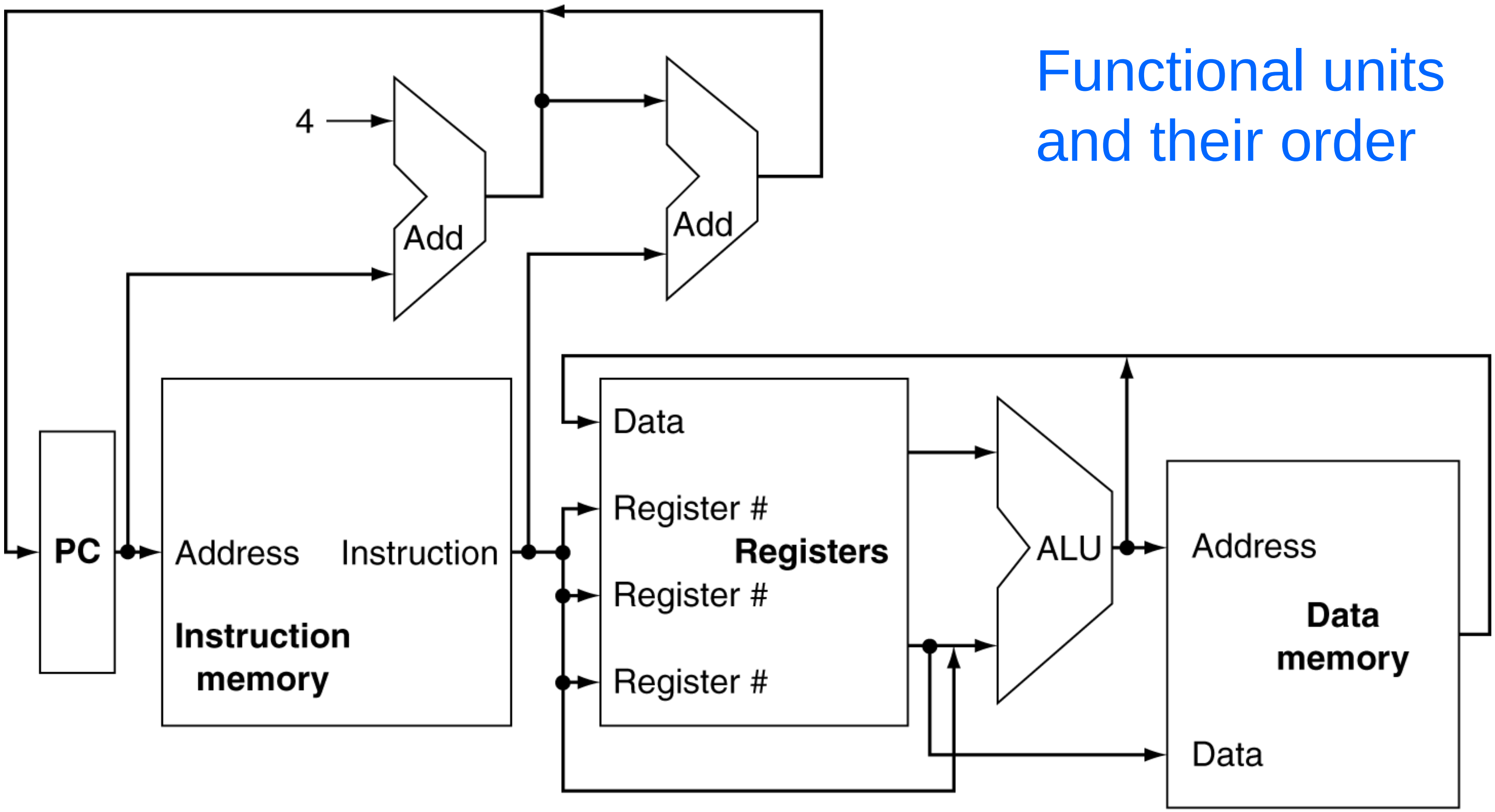
from word offset
to byte offset

Branch Instructions (부연)

- 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

Full Datapath

CPU Overview (반복)

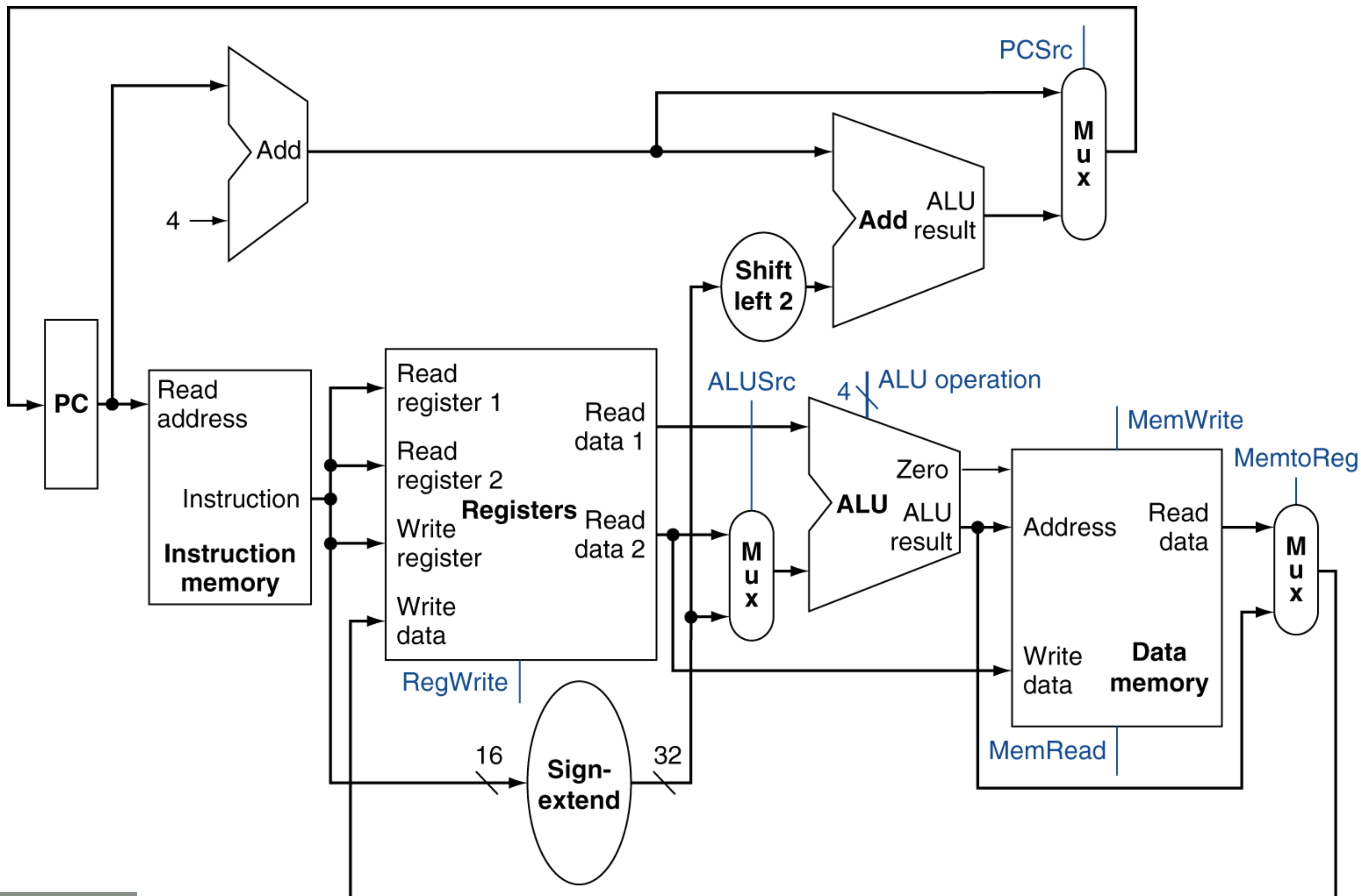


Functional units and their order

Full Datapath

add $t0, $t1, $t2

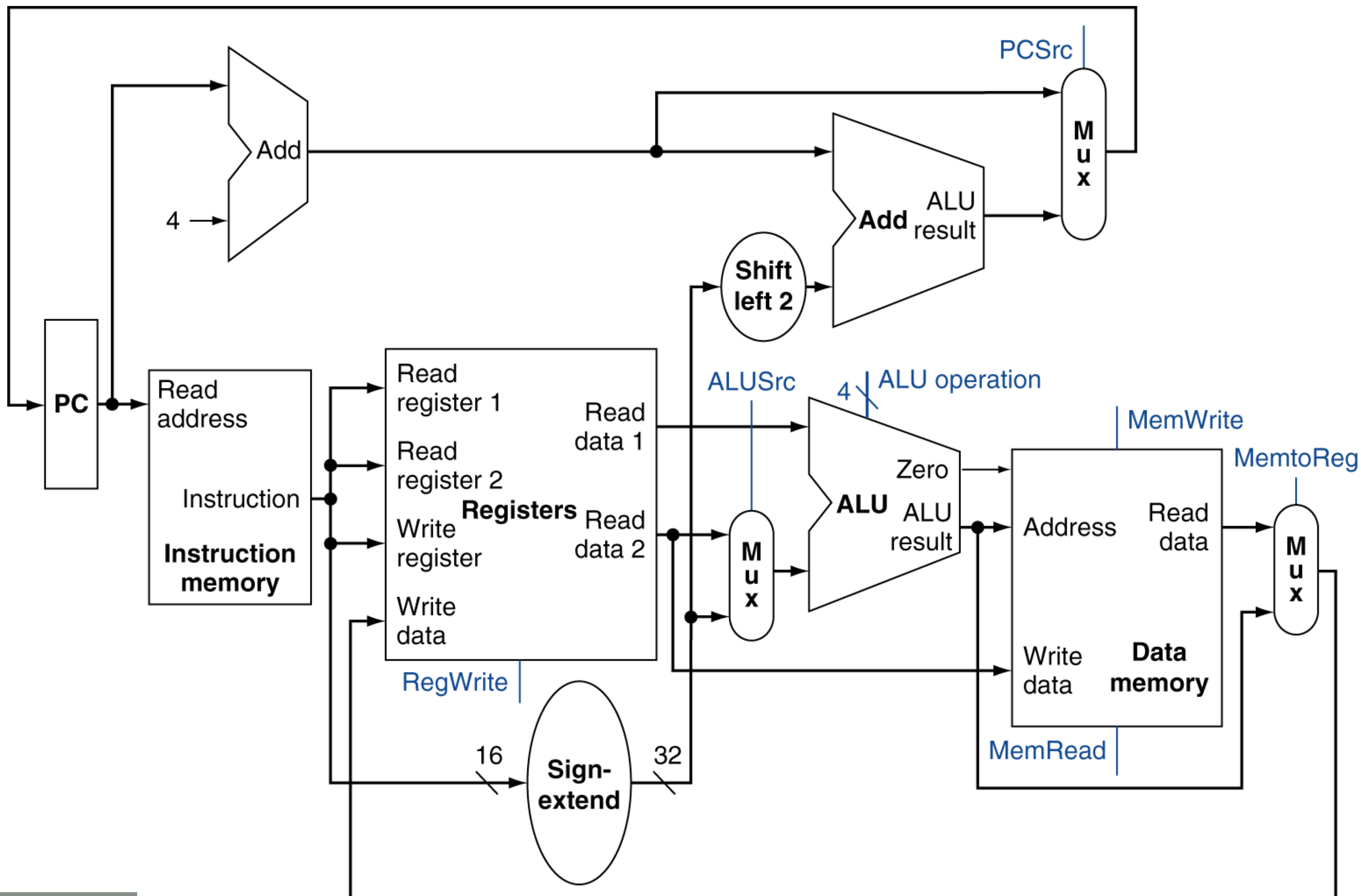
addi $t0, $t1, 4



Full Datapath

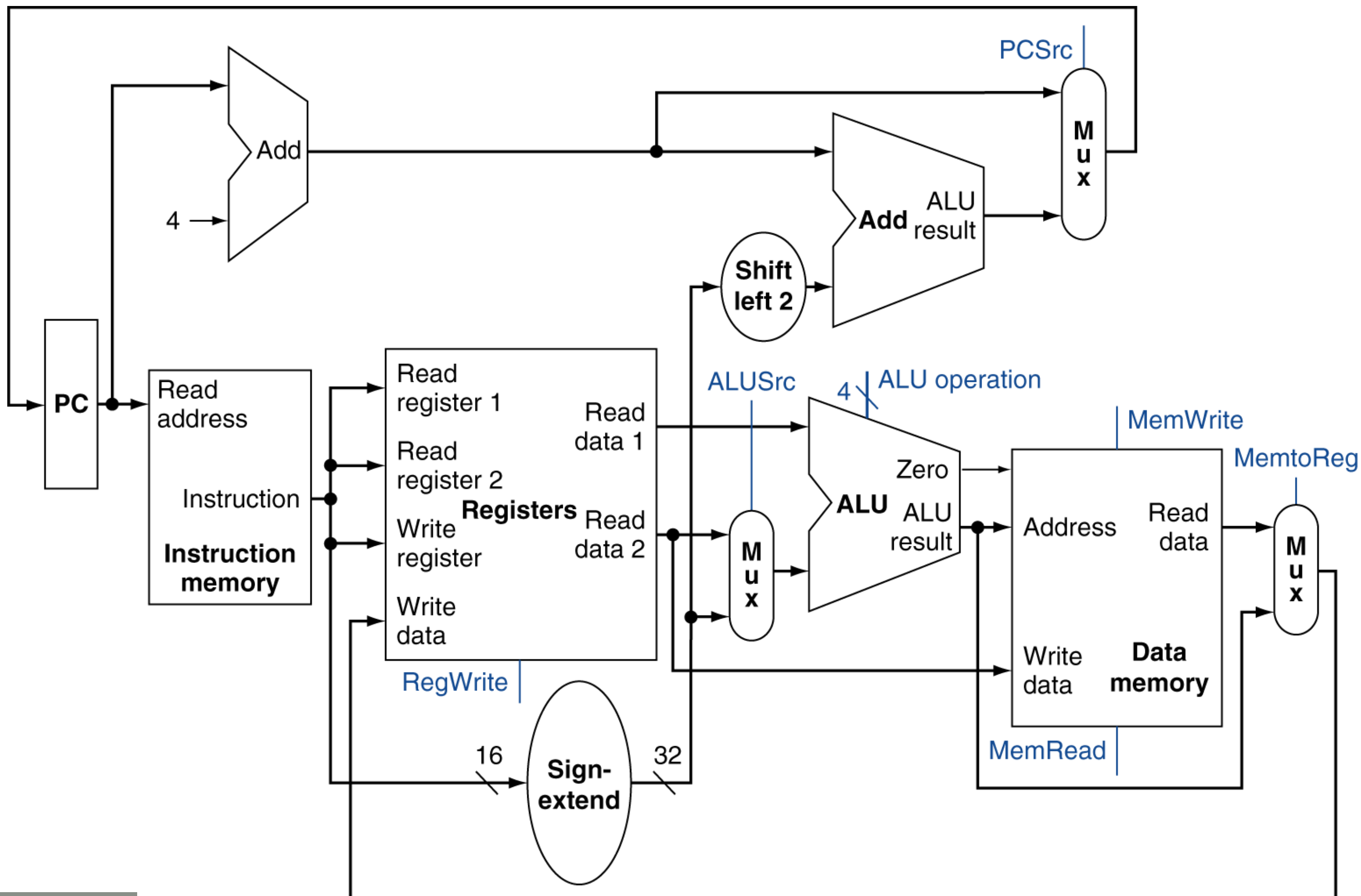
lw $t0, 8($t1)

st $t0, 8($t1)



Full Datapath

beq $t0, $t1, 8



Building the Datapath

- What is datapath?
  - Major functional units, flow of data between them
  - Must execute all MIPS instructions
- Does it look familiar?
  - Because you understand high-level behavior (ISA)
- Datapath design is high-level organization
  - Determine CPI and affect clock cycle time
    - Meaning of single-cycle implementation

Control Design

(Datapath and control)

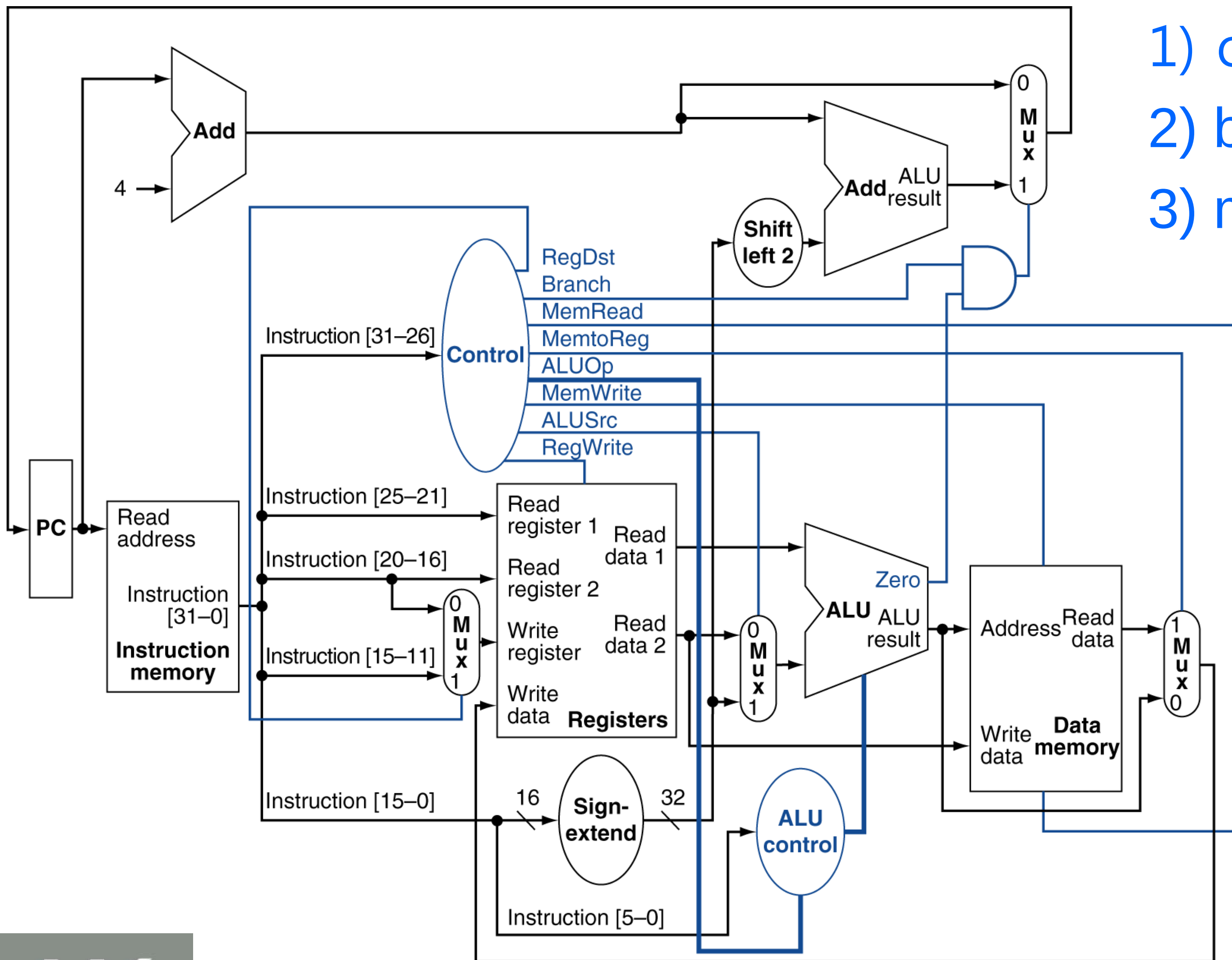
Building the Datapath

- Colored control signals (result of datapath design)
  - Use multiplexers where alternate data sources are used for different instructions
  - Enable or disable functional units
  - Select ALU operation
- How to determine the values of control signals?
  - Opcode (and function code for R-type)
- What is “instruction decode”?

Control Signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Datapath With Control



1) opcode/funct

2) bit 표시,

3) mux 추가,

Building the Datapath

- Datapath and control are 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

Instruction Decode

- Given an instruction, determine values of control signals
  - That's "instruction decode"
- Not shown in the datapath? (because it's control design)
  - Instead, what have "read two operands (registers)"
    - But we don't know the instruction yet
      - † It does no harm
      - † "**lw, addi**" 에서는 하나는 쓰지 않고 버림
- RISC
  - Parallel instruction decoding and register read

Control Design (or Decode)

- Decoding: determine the values of colored control signals
  - Opcode (and function code for R-type)

lw $1, 100($2)

35	2	1	100

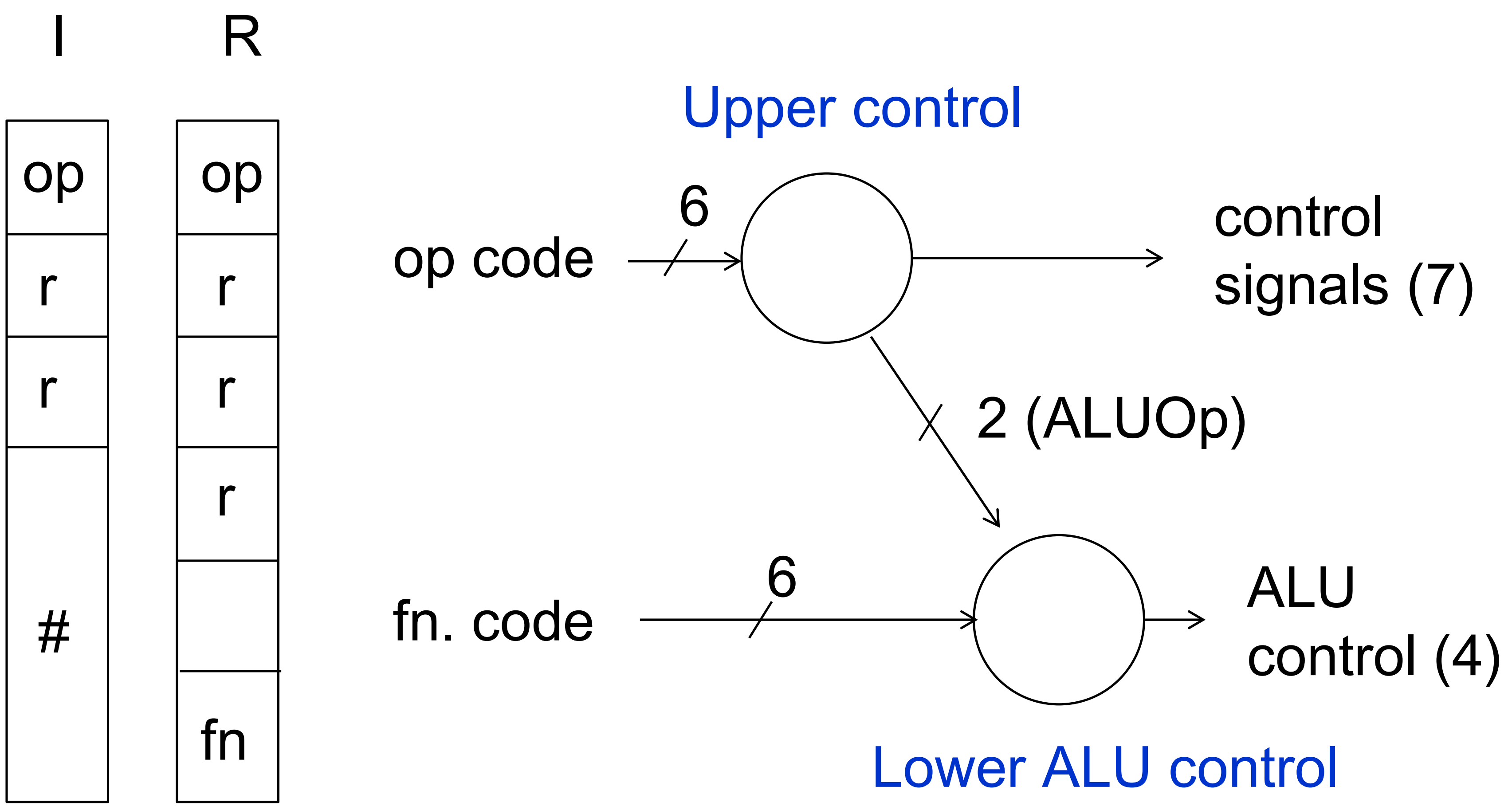
op	rs	rt	16-bit offset

add $8, $17, $18

0	17	18	8	0	32

op	rs	rt	rd	shamt	funct

Control Design (or Decode)



Lower ALU Control Unit

Lower ALU Control Unit

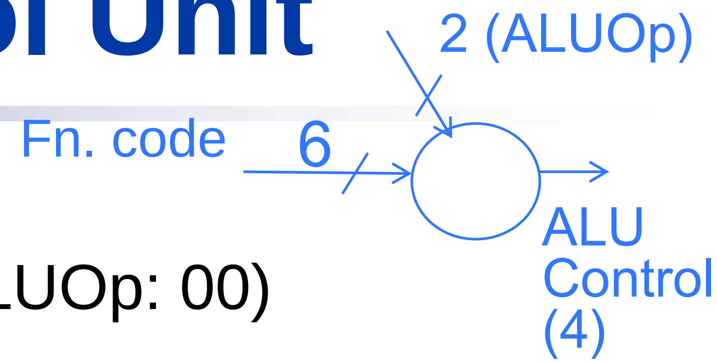
- Non-R-type instructions: see opcode
  - What should the ALU do with this instruction?
    - Example: lw $1, 100($2)

35	2	1	100

op	rs	rt	16-bit offset

  - What should the ALU do for “beq”?
- What about R-type instructions?
  - See function code

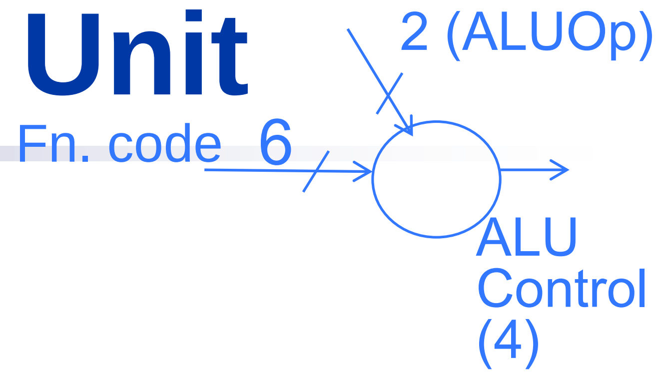
Lower ALU Control Unit



- ALU used for
  - Load/Store: F = add (ALUOp: 00)
  - Branch: F = subtract (ALUOp: 01)
  - R-type: F depends on funct field (ALUOp: 10)
- ALU designed such that:

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

Lower ALU Control Unit



- 2-bit ALUOp derived from opcode

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

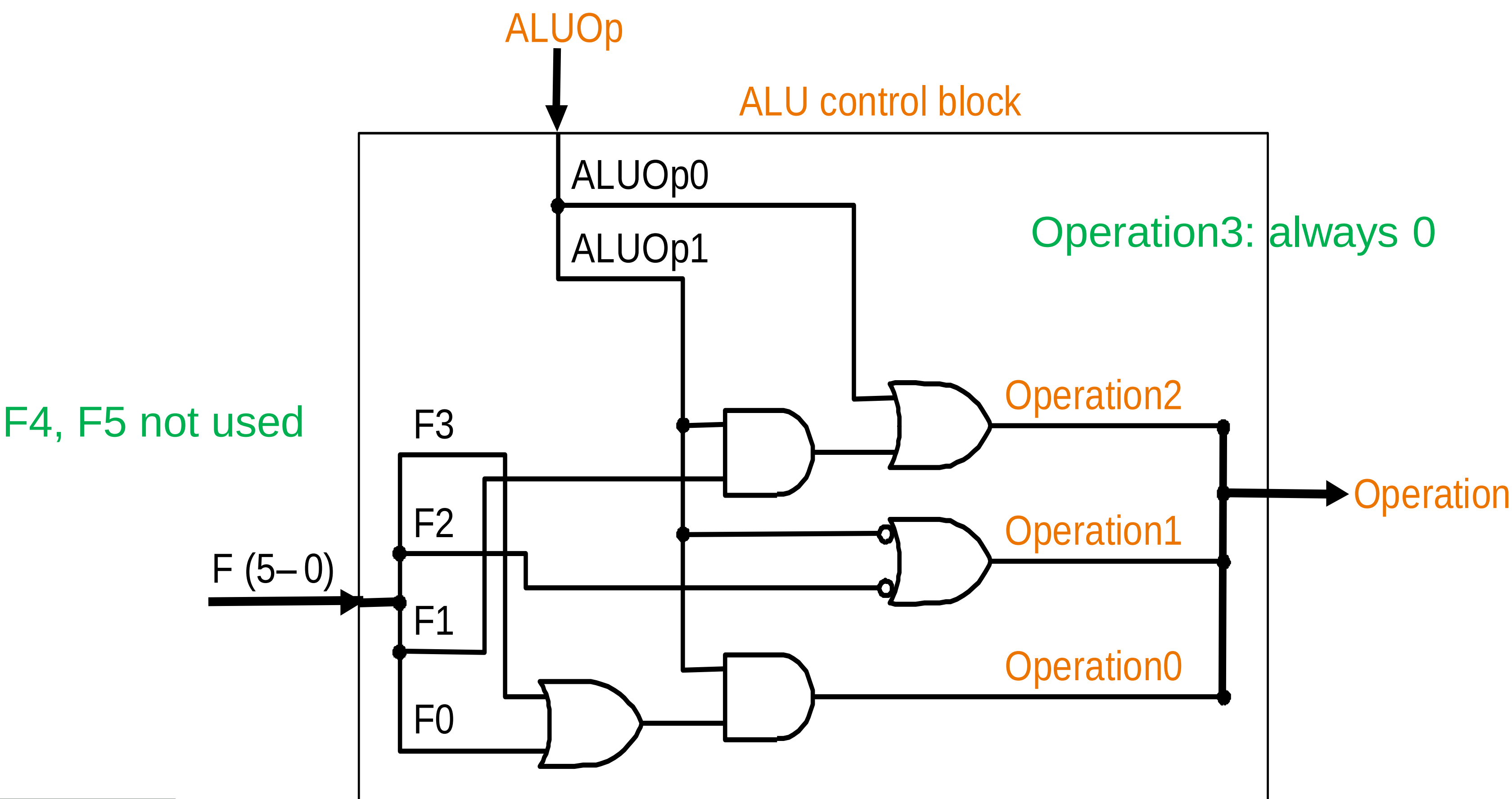
(input: 8 bits, output: 4 bits)

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

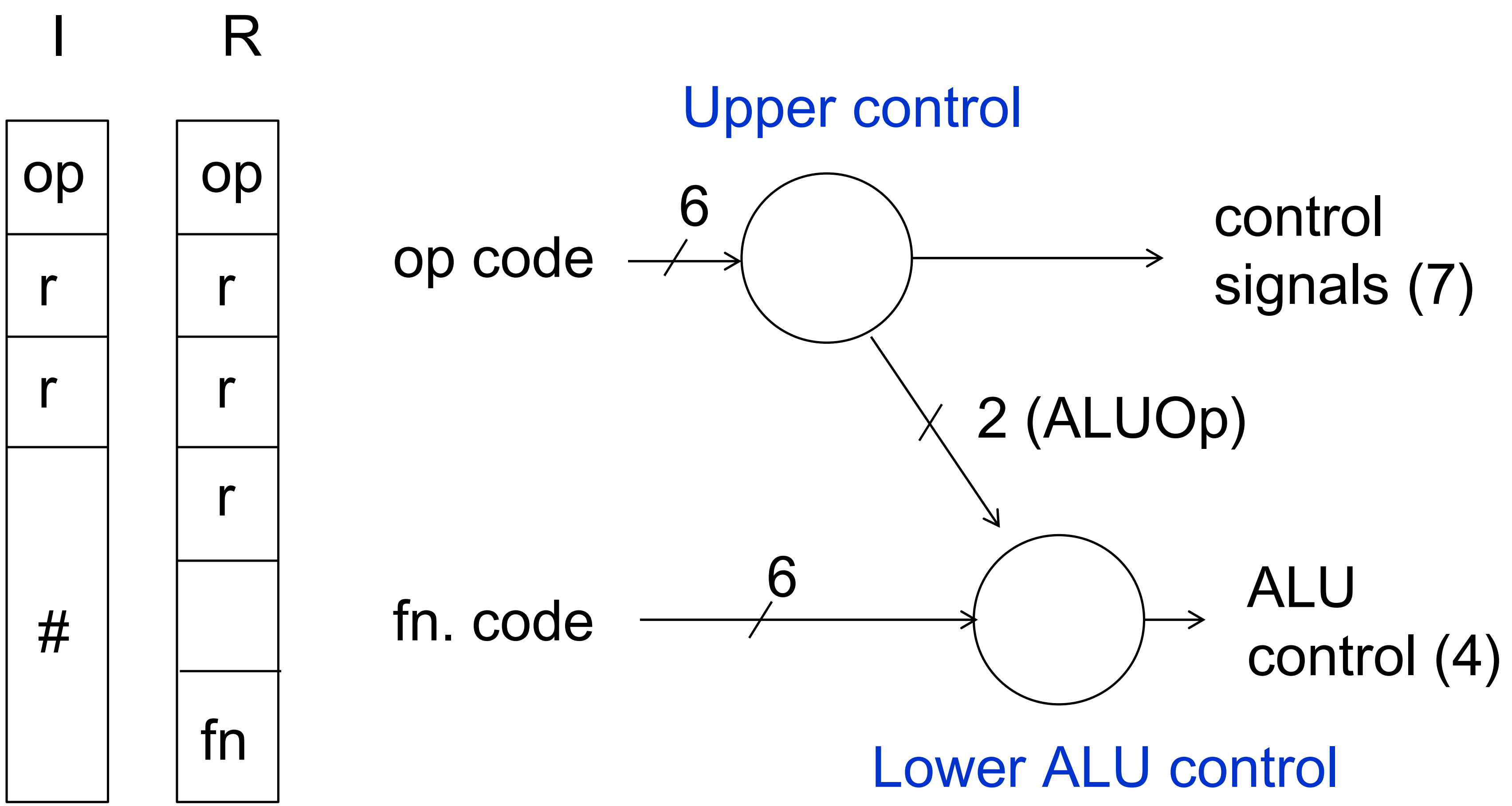
ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Lower ALU Control Unit

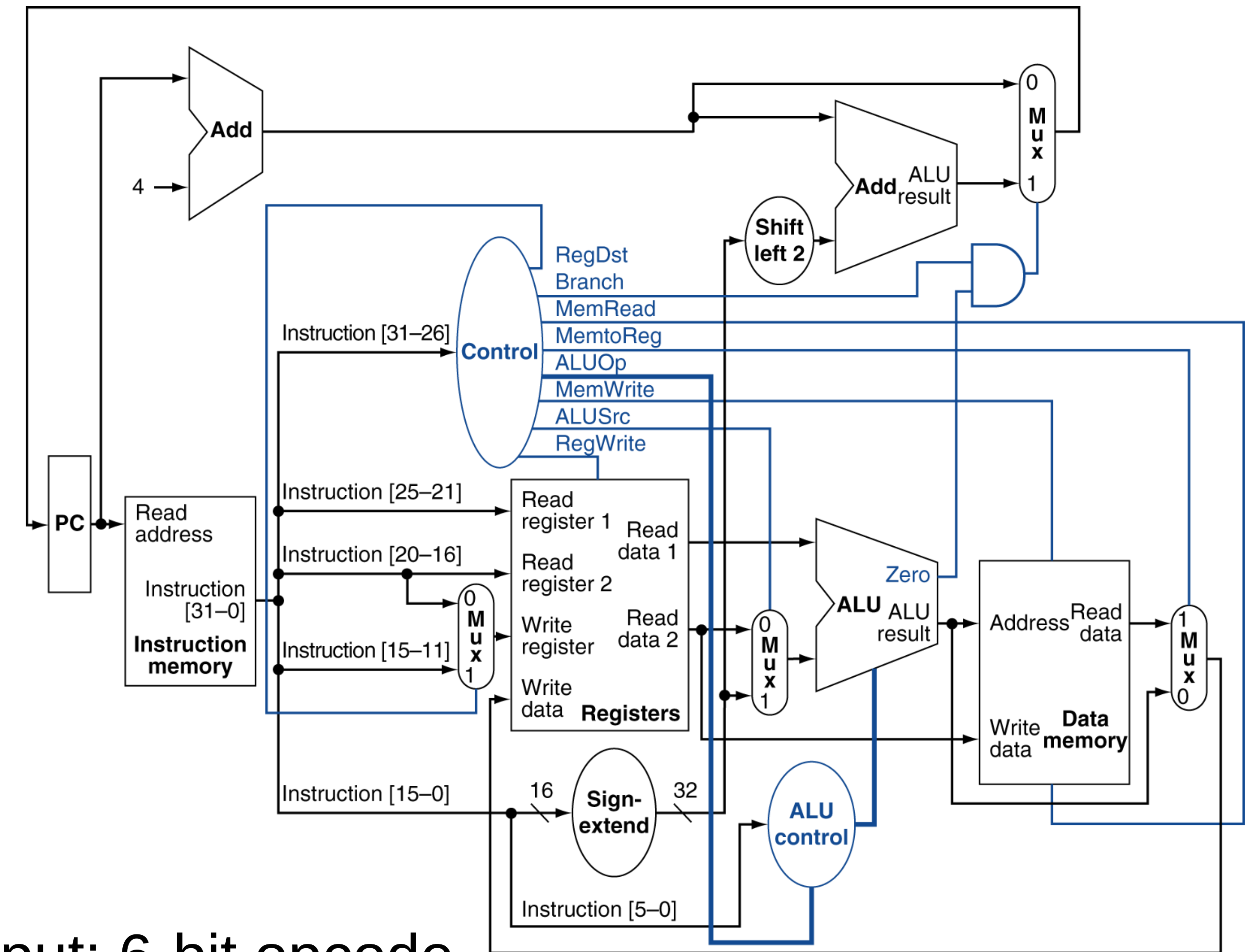
- Simple combinational logic (truth tables) – run CAD tools



Control Design (or Decode) (반복)



Upper Control Unit



Design of upper control unit

Imagine what happens in your PC, notebook or smartphone

input: 6-bit opcode

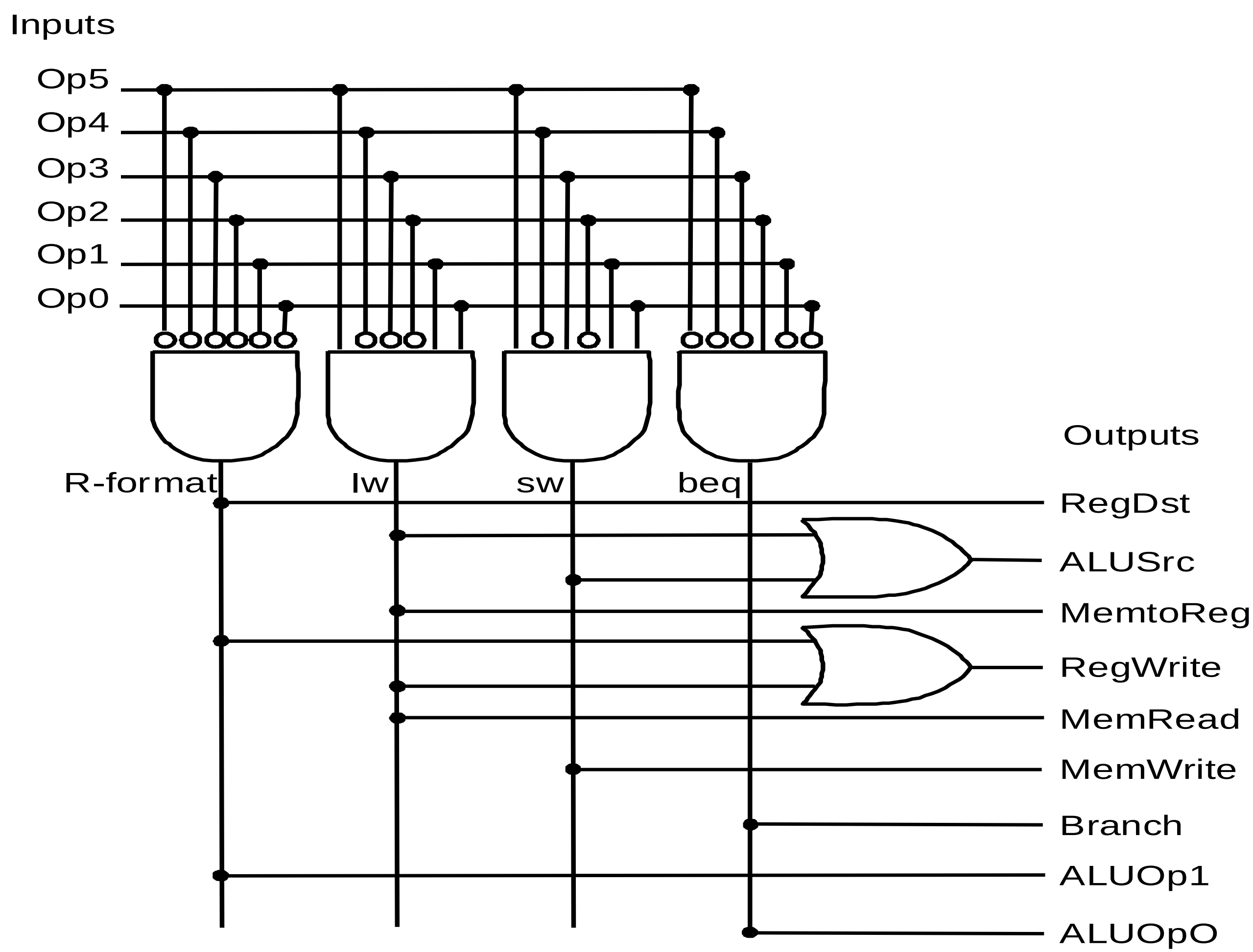
Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Upper Control Unit

- Simple combinational logic (truth tables): hand design

AND plane

OR plane



Control Design

- Low-level circuit design
  - Determine clock cycle time (not affect IC or CPI)
- Do you understand the terms datapath and control?
  - ISA designer (architect) consider them in ISA design
    - Higher-level design requires deep understanding of lower-level designs

MIPS Implementation Done

- ❑ How do we fabricate our processor?
  - What if we visit Intel or Samsung?
    - Same situation when industry develop prototypes
- ❑ Field programmable logic (FPGA) by Altera or Xilinx
  - Software tool (and FPGA chips)
    - VHDL/Verilog description of our design
    - Compile and test
    - Dump to FPGA chips

Implementing More Instructions

Implementation 8 Instructions

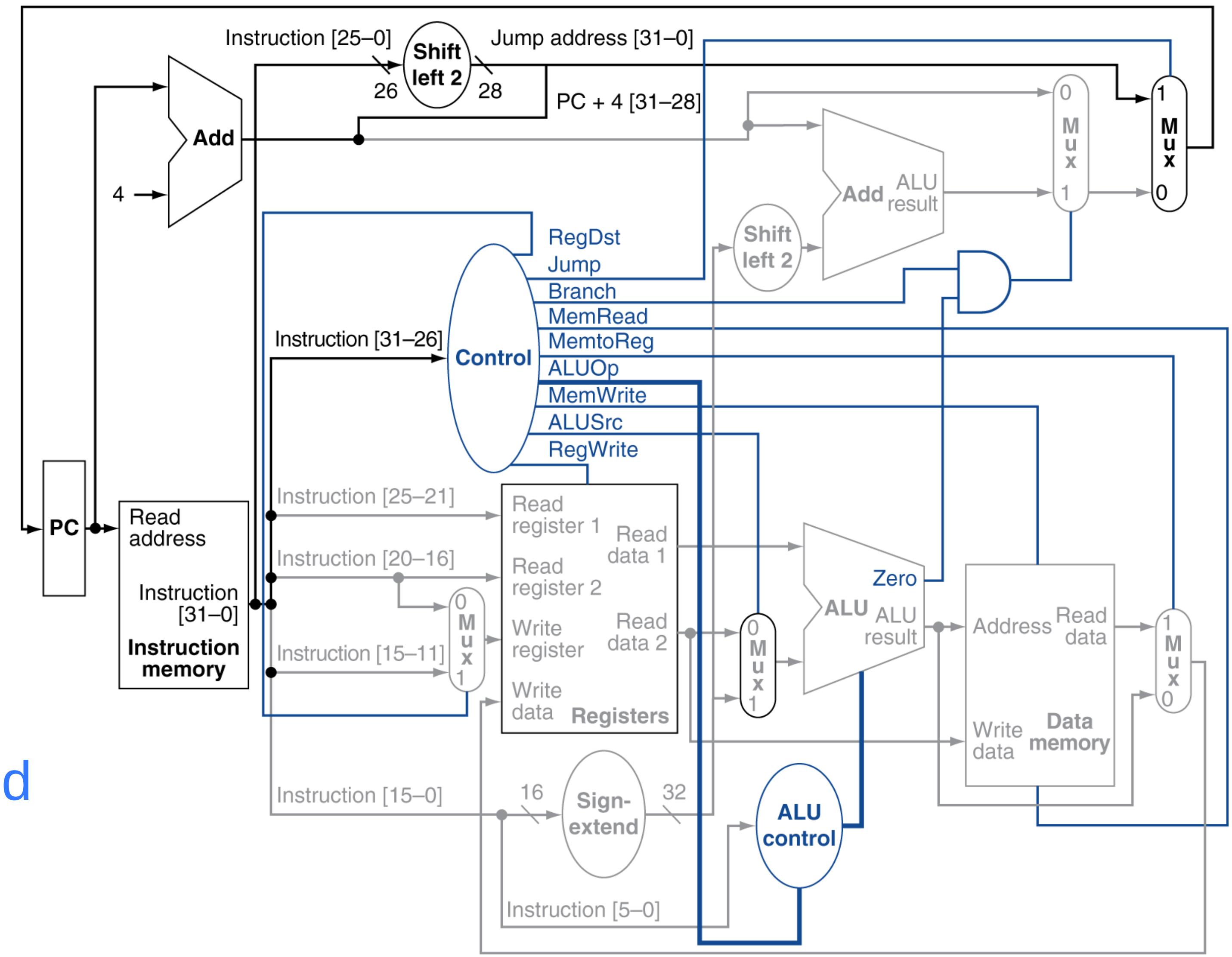
- Simple subset, shows most aspects
  - Memory-reference: **lw, sw**
  - Arithmetic-logical: **add, sub, and, or, slt**
  - Control transfer: **beq, (j)**
- What if we want to add "jump" instruction?
  - Same principles
  - Only more datapath and control

Implementing Jumps

Jump	2	address
	31:26	25:0

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

Datapath With Jumps Added



More datapath and control

Control for “jump” Instruction

- One more output signal for control unit
  - Let’s call it “jump”
  - One more column in truth table
- New jump instruction to implement
  - One more row in truth table

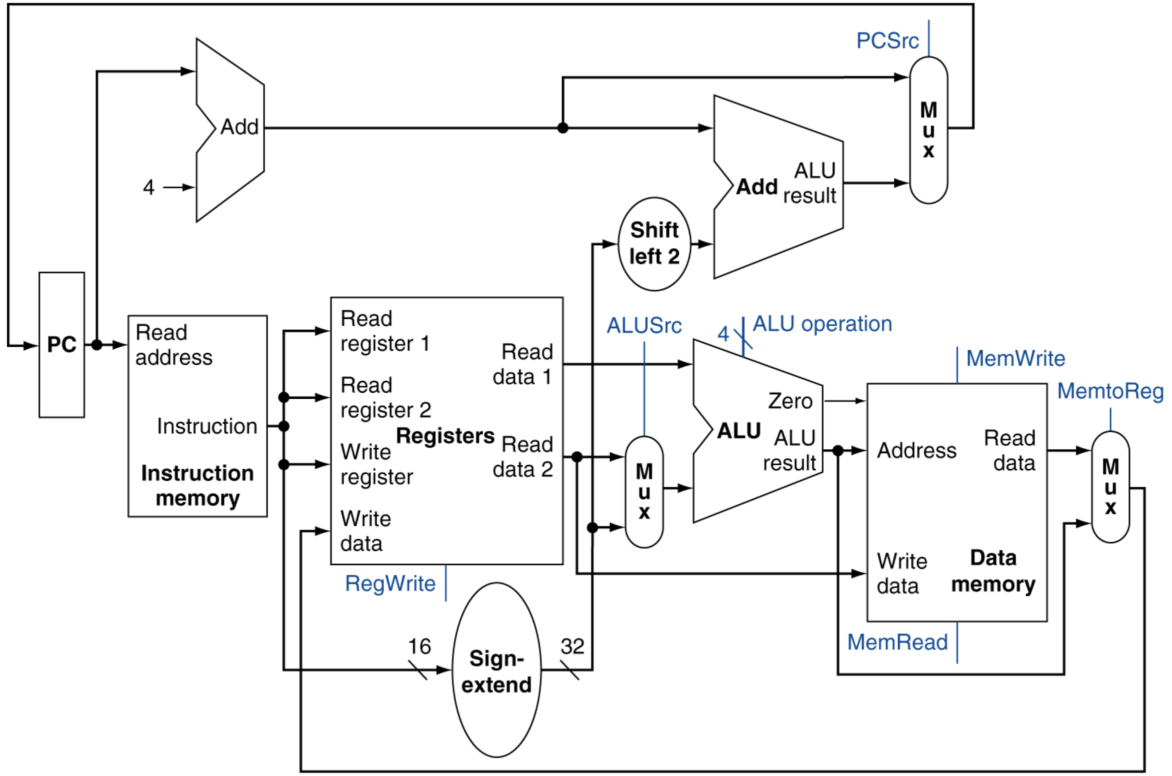
Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0	Jump
R-format	1	0	0	1	0	0	0	1	0	0
lw	0	1	1	1	1	0	0	0	0	0
sw	X	1	X	0	0	1	0	0	0	0
beq	X	0	X	0	0	0	1	0	1	0
J	X	X	X	0	0	0	0	X	X	1

Performance Issues

(How good is our implementation?)

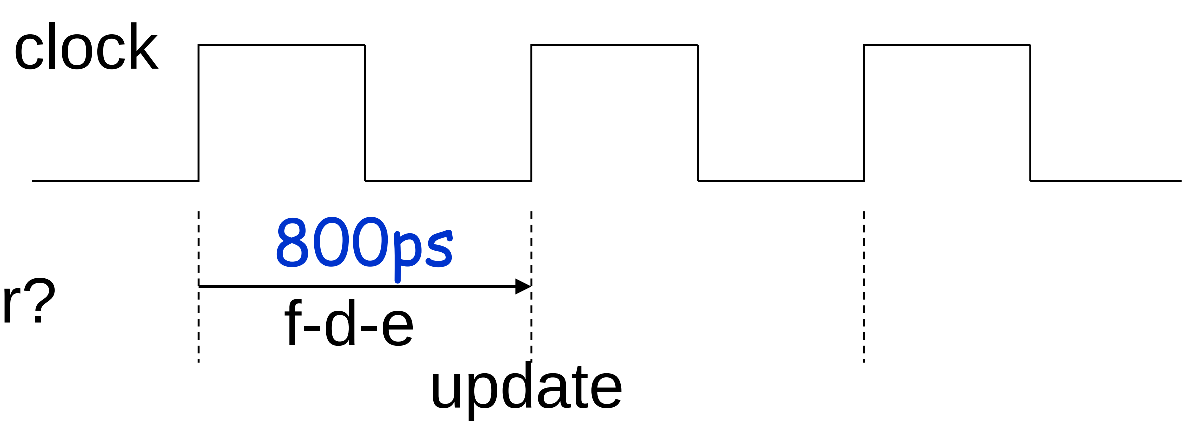
Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - Memory access (200ps), ALU and adders (200ps), register file access (100ps)



Single Cycle Implementation

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps



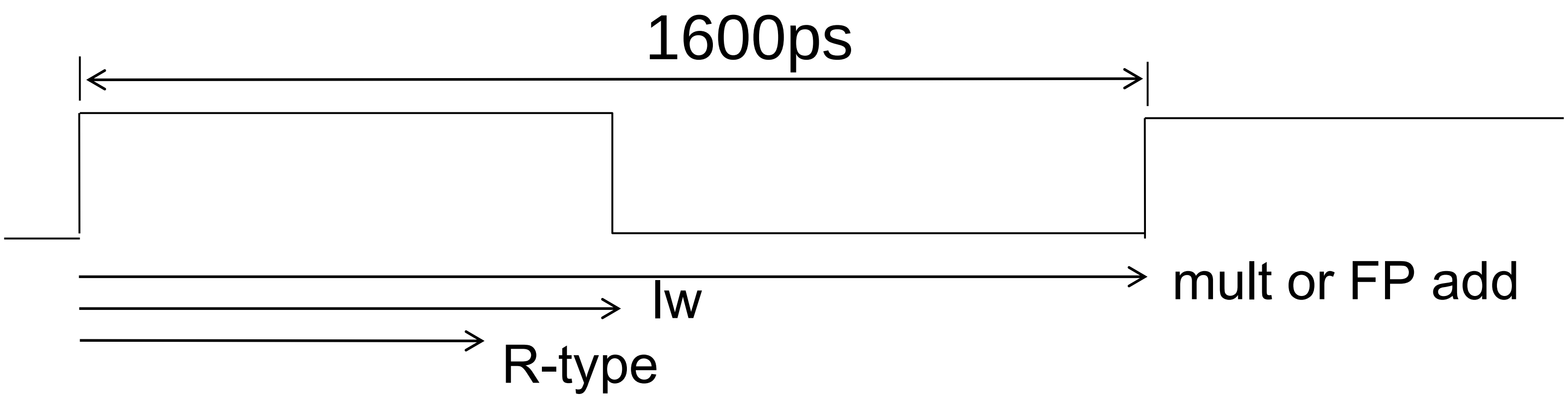
- IM/DM access fast?
- Register access faster?

Performance

- Performance model
  - Execution time = IC $\times$ CPI $\times$ clock cycle time
  - CPI = 1, cct = 800 ps
- Longest delay determines clock period
  - Critical path: load instruction
- Is "load" really the critical path?
  - What about "mult" or FP operations?

High-Level Org.: CPI & Clock Cycle

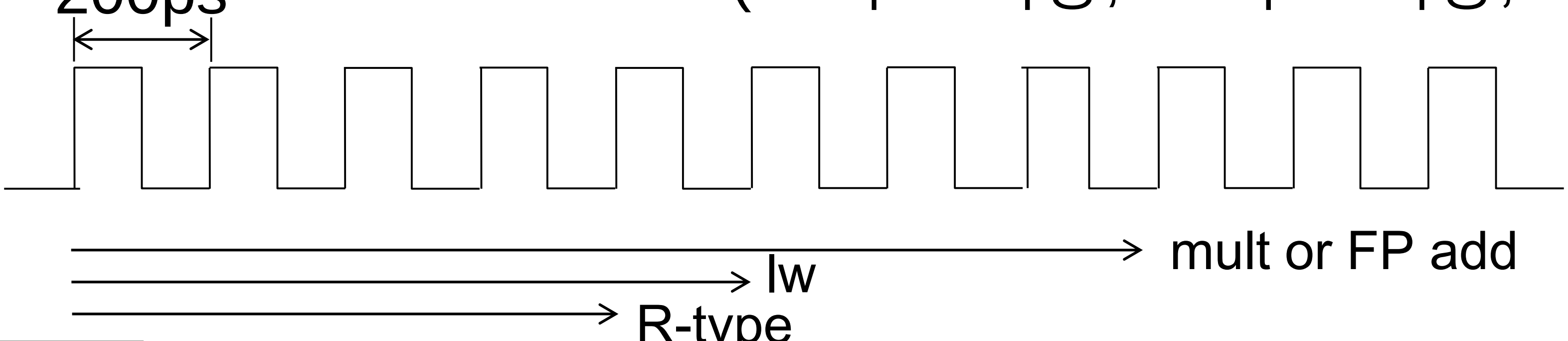
- Single-cycle: CPI = 1, clock cycle



- Multi-cycle: CPI ↑, clock cycle ↓, overall performance ↑

(IF 하고 저장, ID 하고 저장, …)

200ps



MIPS Multicycle (Core instruction)

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Multicycle vs. Pipeline

(clock cycle time = 200ps)

Multicycle (CPI = 5)

Time	IF	ID	EX	DM	WB
CC1	I_0				
CC2		I_0			
CC3			I_0		
CC4				I_0	
CC5					I_0
CC6	I_1				

Pipeline, 5-stage (CPI = 1)

Time	IF	ID	EX	DM	WB
CC1	I_0				
CC2	I_1	I_0			
CC3	I_2	I_1	I_0		
CC4	I_3	I_2	I_1	I_0	
CC5	I_4	I_3	I_2	I_1	I_0
CC6	I_5	I_4	I_3	I_2	I_1

Where We Are Heading (부연)

- Single Cycle Problems:
  - What if we had a more complicated instruction like floating point?
- Multicycle implementation
  - Use a “smaller” cycle time
  - Different instructions take different numbers of cycles
- Pipelining (pipelined datapath and control)
  - Overlapped instruction execution (CPI = 1)
  - Instruction-level parallelism

Homework #11 (see Class Homepage)

1) Write a report summarizing the materials discussed in Topic 4-1 (이번 주 수업 내용)
2) Write a report summarizing the materials discussed in Topic 4-2 (이번 주 수업 내용)

** 문장으로 써도 좋고 파워포인트 형태의 개조식 정리도 좋음

3) Solve Chapter 4 exercises 4.2, 4.3, 4.5, 4.6 (crosstalk faults, stuck-at-0 faults), 4.7

- Due: see Blackboard
  - Submit electronically to Blackboard

Class Topics (클래스 홈페이지 참조)

- Part 1: Fundamental concepts and principles
- Part 2: 빠른 컴퓨터를 위한 ISA design
  - Topic 1 Computer performance and ISA design (Ch. 1)
  - Topic 2 RISC (MIPS) instruction set (Chapter 2)
  - Topic 3 Computer arithmetic and ALU (Chapter 3)
- Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)
  - Topic 4 Processor design (Chapter 4)
    - Single cycle implementation
    - Pipelined implementation
  - Topic 5 Memory system design (Chapter 5)