Chapter 4

Implementing ISA
(Fetch, Decode, Execute)

Part 1: Single Cycle Design

Big Picture

- Part 1: what is computer, CSE, computer architecture?
  - Fundamental concepts and principles
- Part 2: ISA (externals)
  - Ch. 1: performance
    - Exe. time, benchmark, model, RISC, power, multicore
  - Ch. 2: language of computer; ISA
    - What is a good ISA? Today's RISC-style ISA (MIPS)
  - Ch. 3: computer arithmetic
    - Data representation and ALU, ISA – data perspective
- Part 3: implementation of ISA (internals)
  - Ch. 4: processor
  - Ch. 5: memory system

Big Picture

- Part 3: implementation of ISA (internals)
  - High-level organization, not circuits design
  - Ch. 4: processor
    - Given ISA, what is a good implementation?
    - Instruction sequencing (fetch-decode-execution)
      - Datapath and control, pipelining
  - Ch 5: memory system design
    - Cache memory (a part of processor)

Another Engineering Paradigm

- Requirement engineering 속성
  - Requirements in processor design?
- External design interface
  - Go through detailed simulation of internal design
- Internal design implementation
  - Finalize internal design
- Verification & validation
- Maintenance bug fix

iterative

† Iterative and hierarchical

Implementation of ISA

- Datapath and control (internals, chip design)
  - High-level organization (and low-level circuits design)
    - Affect CPI and clock cycle time pipeline으로 변화큼
      - † ISA affect IC (instruction count), CPI, clock
- Three implementations
  - Single cycle implementation
  - Multi-cycle implementation (optional)
  - Pipelined implementation

The Processor: Datapath & 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: repeat the following
  - Instruction fetch (IF)
    - Get the instruction from memory, PC $\leftarrow$ PC + 4
  - Instruction decode (ID)
    - Use the instruction to decide exactly what to do
  - Instruction execute (EX)
    - Do it

Instruction Execution

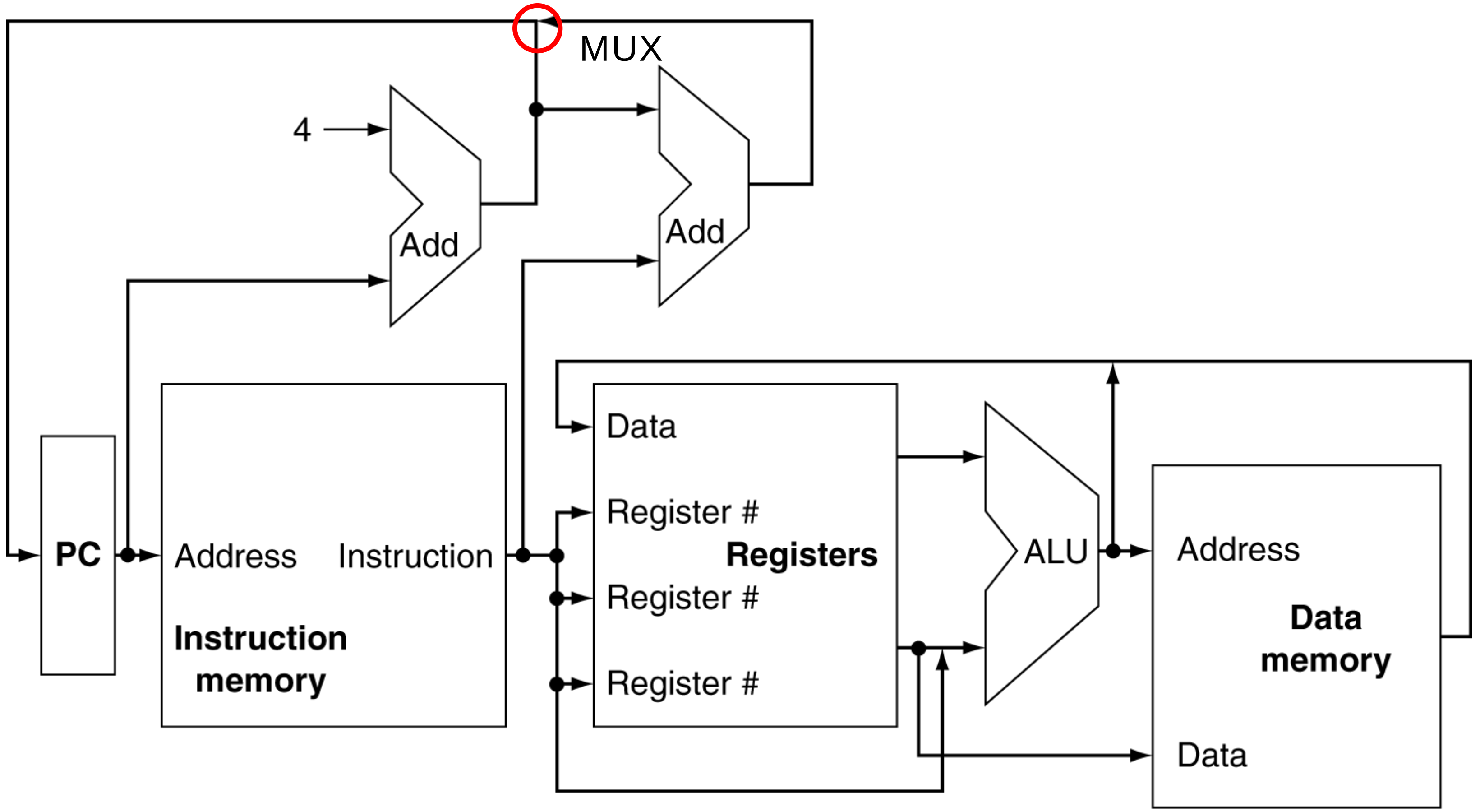
- PC $\rightarrow$ instruction memory, fetch instruction, PC $\leftarrow$ PC+4
- Decode instruction (opcode)
  - Register numbers $\rightarrow$ register file, read registers
- Execute instruction: depending on instruction class
  - Use ALU to calculate
    - Arithmetic result

      (then store result in register)
    - Memory address for load/store

      (load: read data memory, store data in register)

      (store: write to data memory)
    - Branch target address (then PC $\leftarrow$ target address)
  - Functional units and their order

CPU Overview: Schematic Diagram



- Why does this look familiar?
- ISA determine functional units and their order

How do we execute MIPS instructions?

- ALU instructions

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

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

- Data transfer instructions

  lw $t0, 12($t1) // I-type

  st $t0, 12($t1)

- Branch instructions

  beq $t0, $t1, 8 // I-type

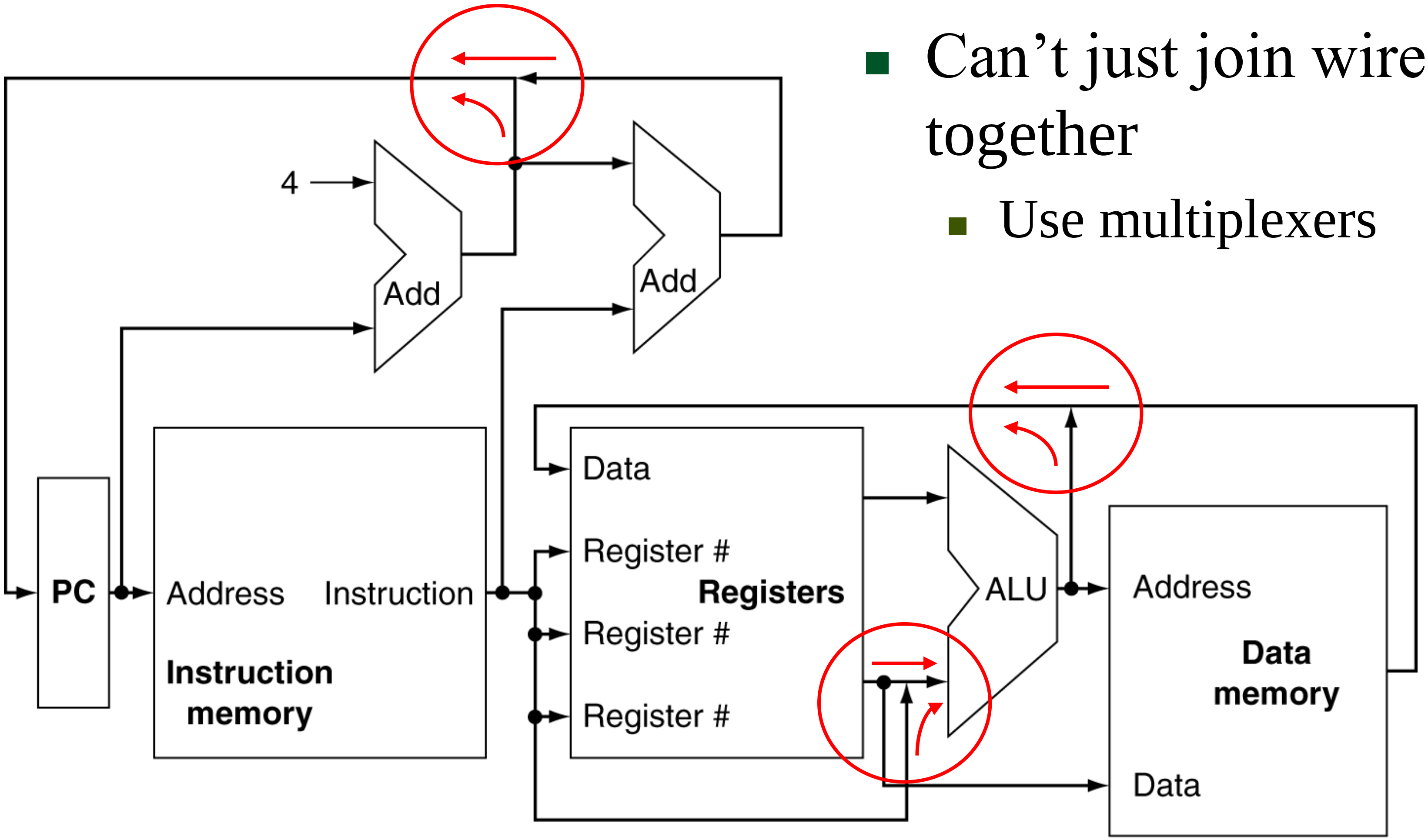
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Instruction Decode

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

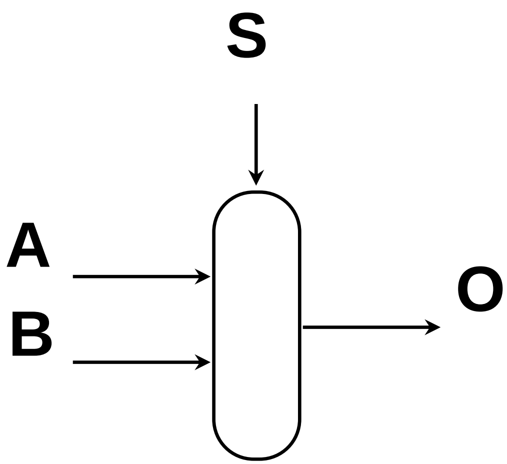
Need to Use Multiplexers

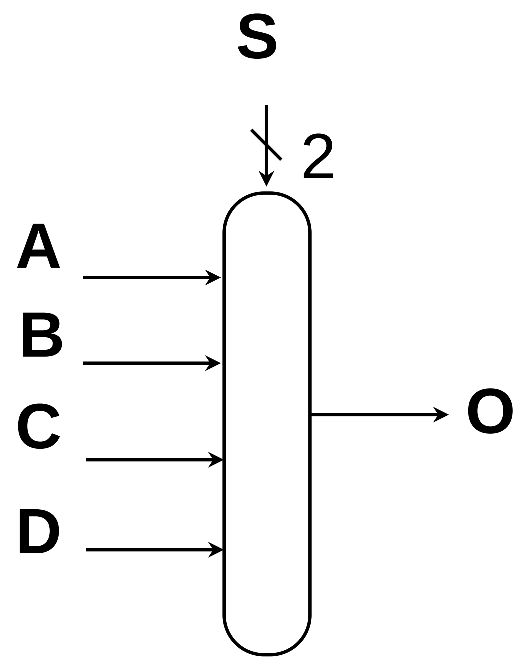
- Can't just join wires together
  - Use multiplexers



Multiplexers (반복)

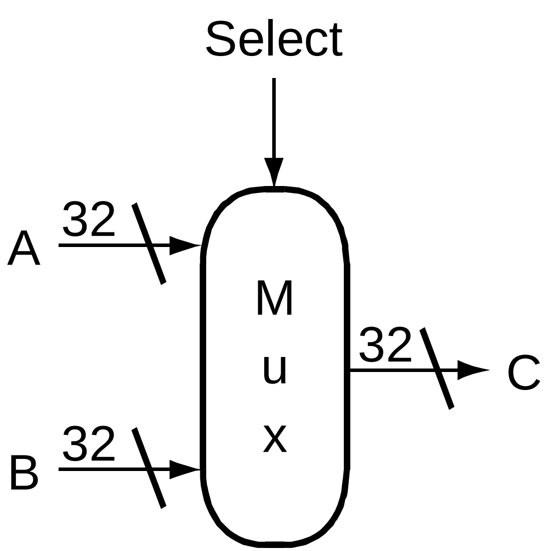
- 2-to-1 MUX, 4-to-1 MUX (c.f., Demultiplexer)

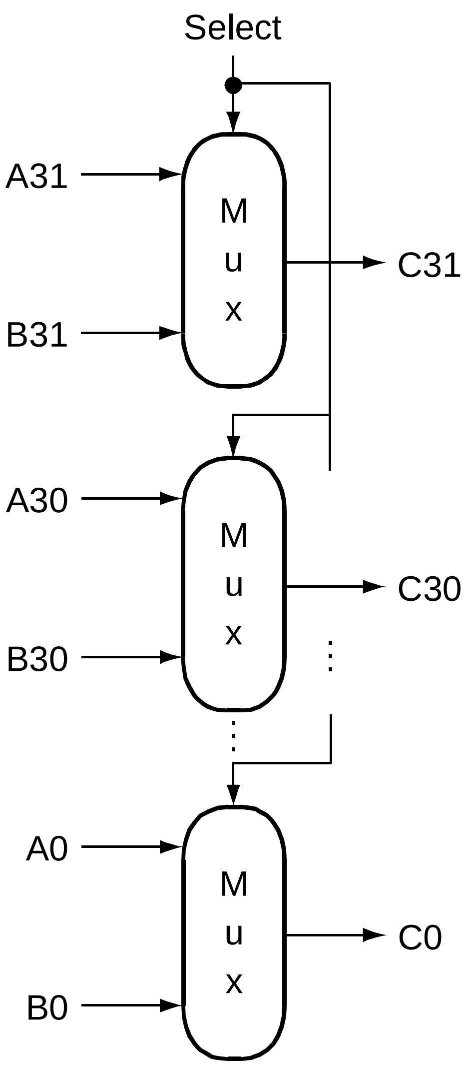




Multiplexer Abstraction (반복)

- Make sure you understand the abstractions!
- 32 of 2-to-1 mux



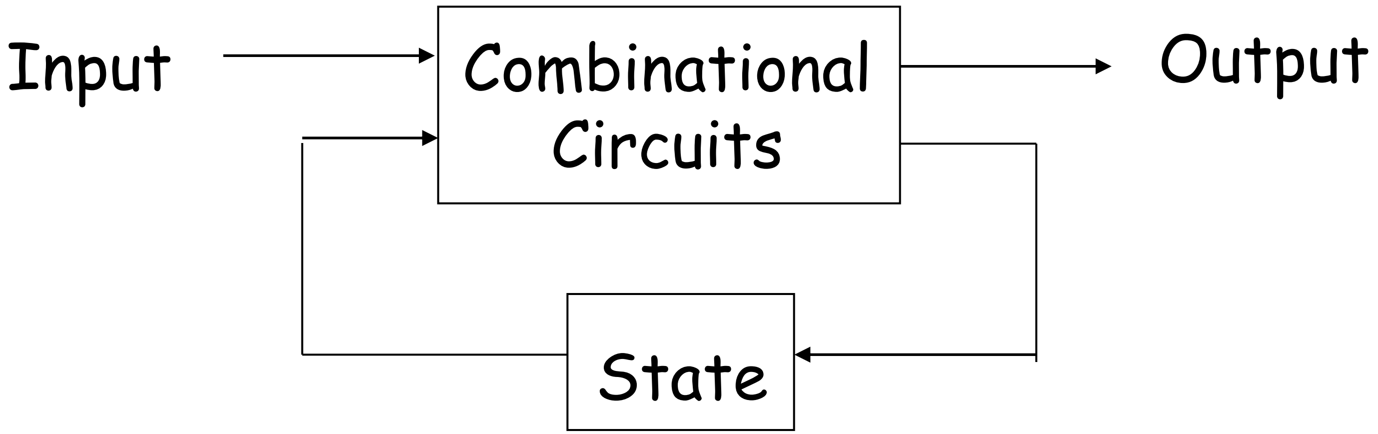


Digital Logic Design (반복)

- Given: AND, OR, NOT
- Design
  - Combinational logic circuits
    - Decoder, mux, …, ALU
  - Sequential logic circuits
    - Latch, flip-flop, register, counter, …, CPU
- Notion of abstraction
- VHDL/Verilog simulation (software flavor)

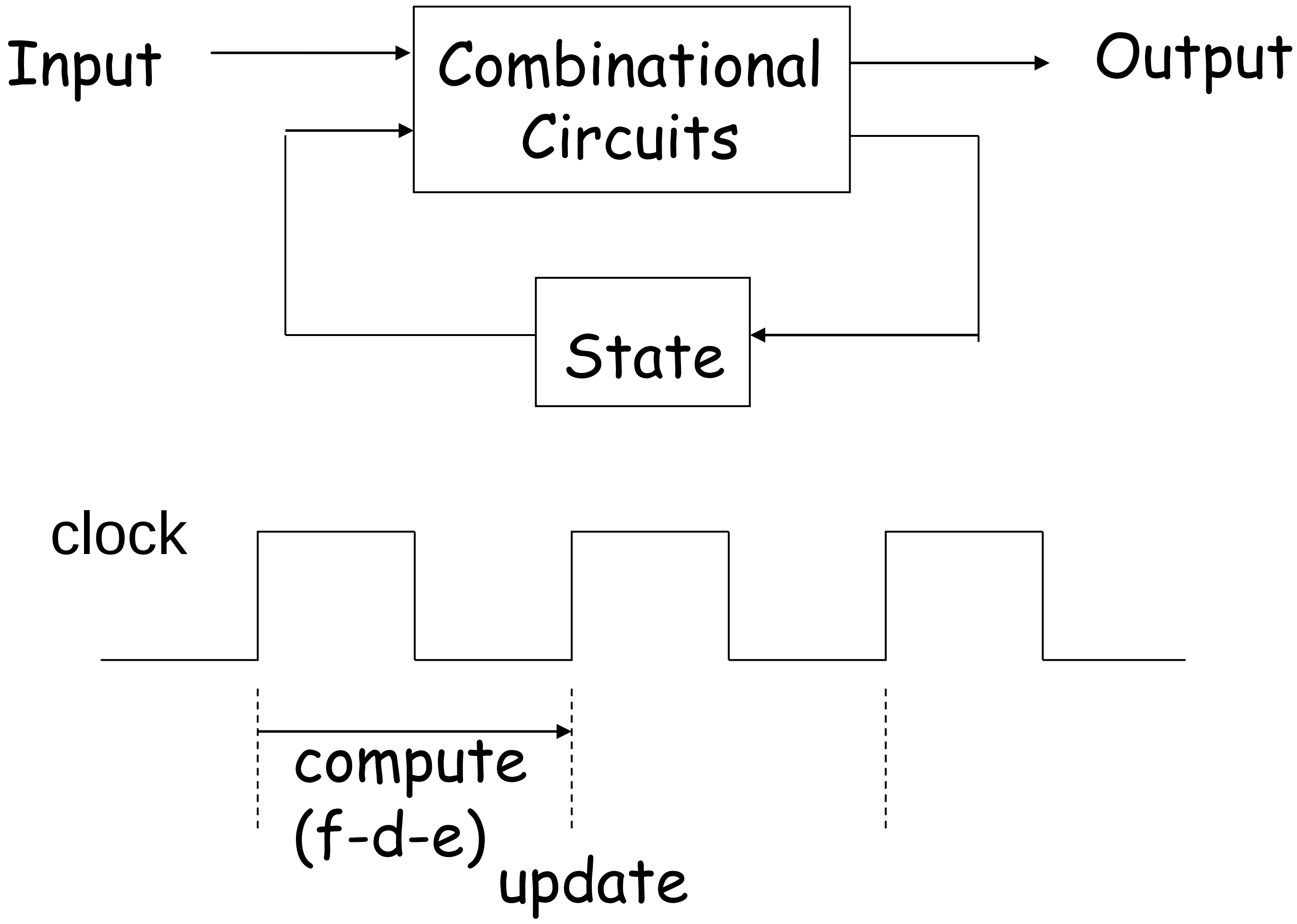
† How do you define combinational or sequential logic?

Sequential circuits (state diagram) (반복)



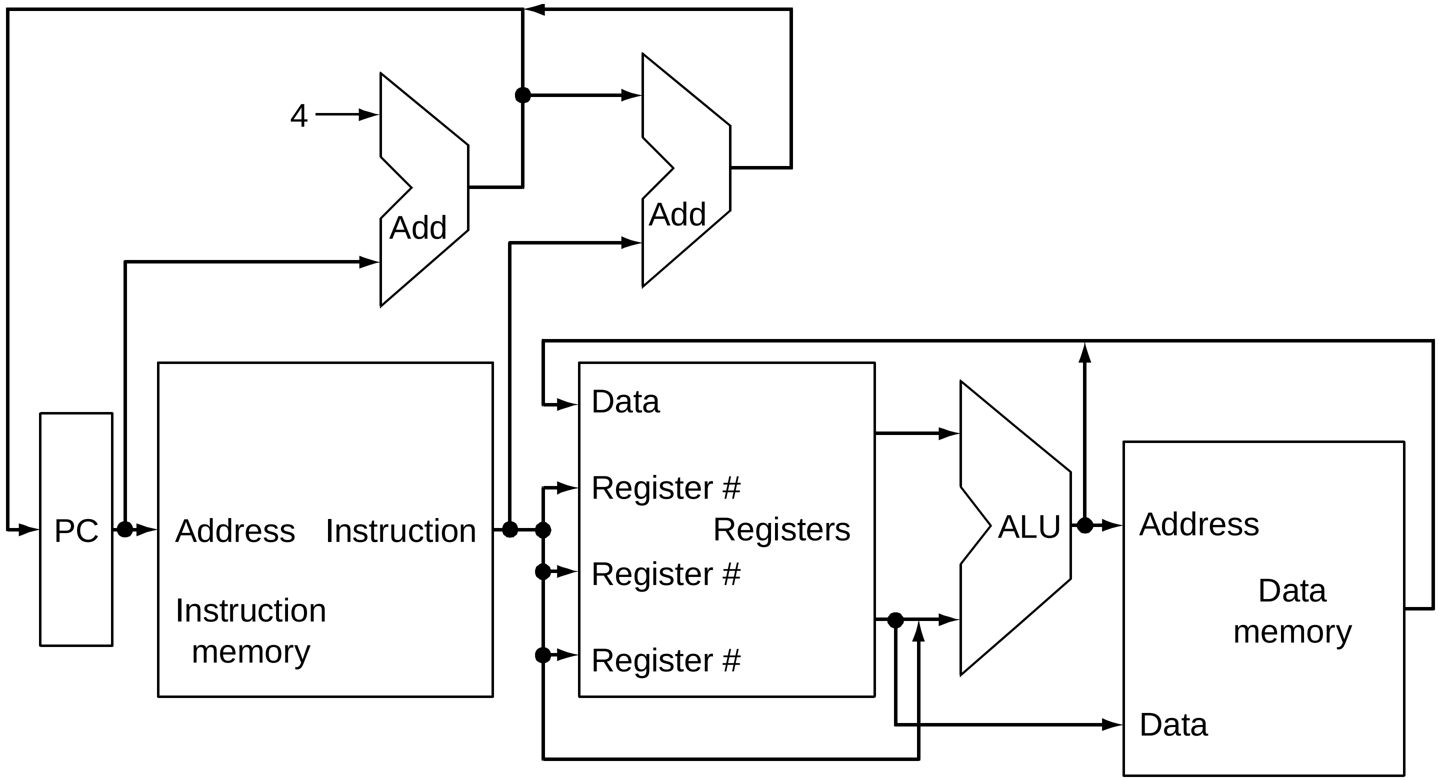
- Two types of functional units:
  - elements that operate on data values (combinational – truth table)
  - elements that contain state (sequential – state diagram)

Sequential circuits (반복)



† Synchronous logic circuits

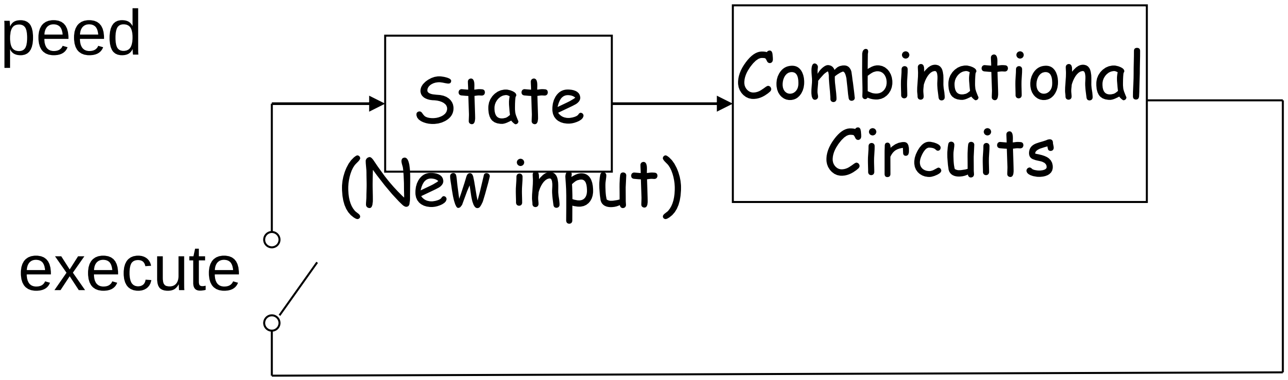
CPU: synchronous sequential logic circuits

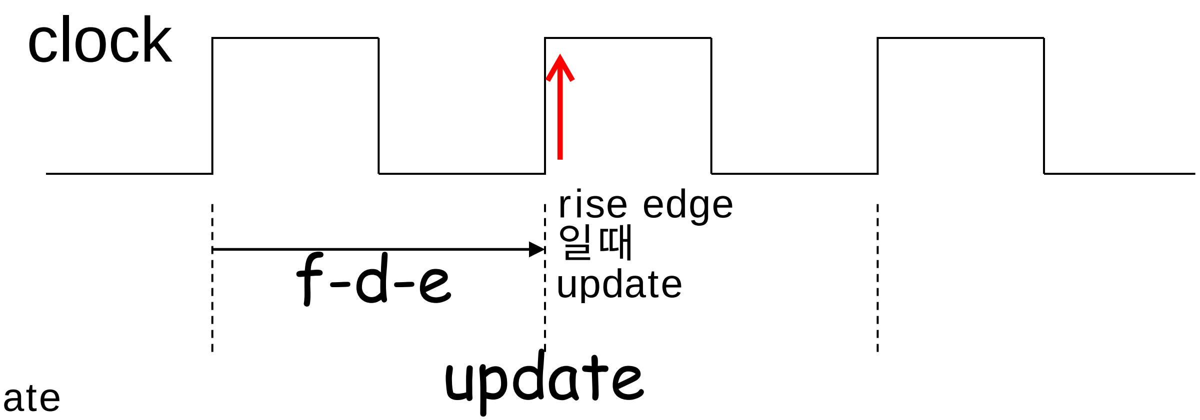


- What are the states?
  - Result of "fetch-decode-execute" updated at the end of each clock cycle

How do we provide inputs at high speed?

- Pentium at 1GHz
  - Light speed





state
1. pc
2. register
3. data memory
한 clock 끝나면 state update

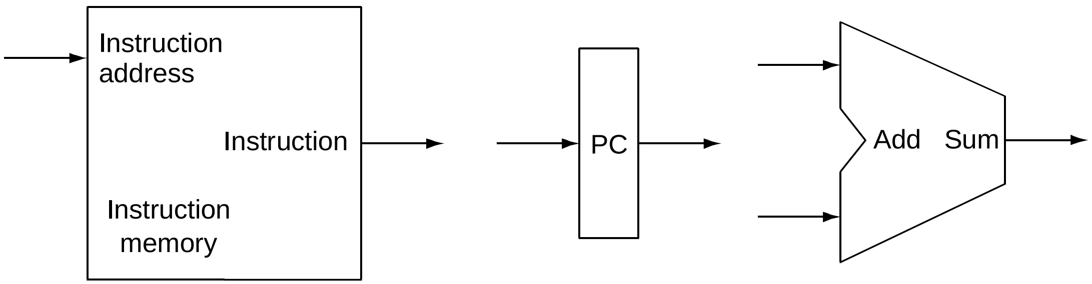
- Use PC to get new input by itself

Let’s build a datapath

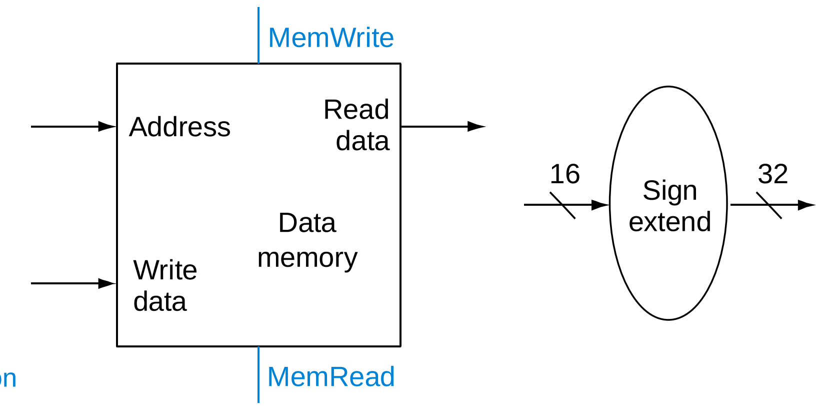
- Which can execute MIPS instructions
- Using smaller building blocks

Building Blocks (Functional Units; Abstractions)

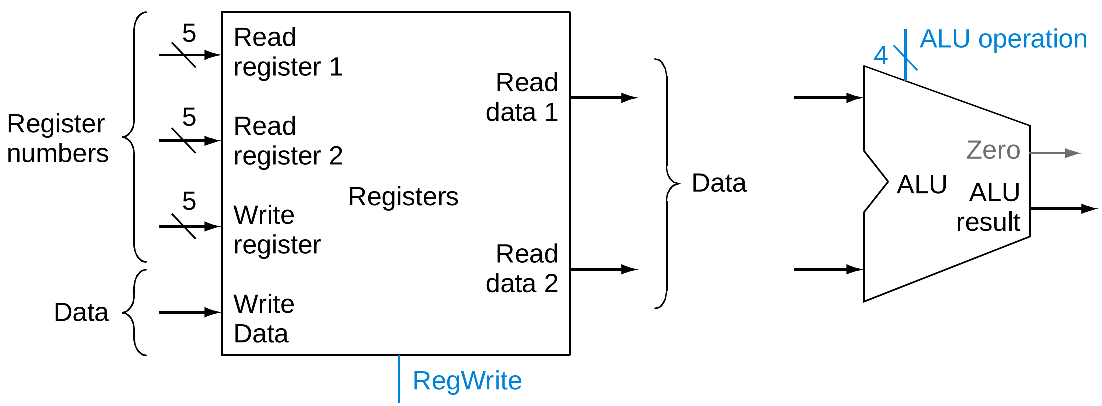
- Include the functional units we need for each instruction



a. Instruction memory b. Program counter c. Adder



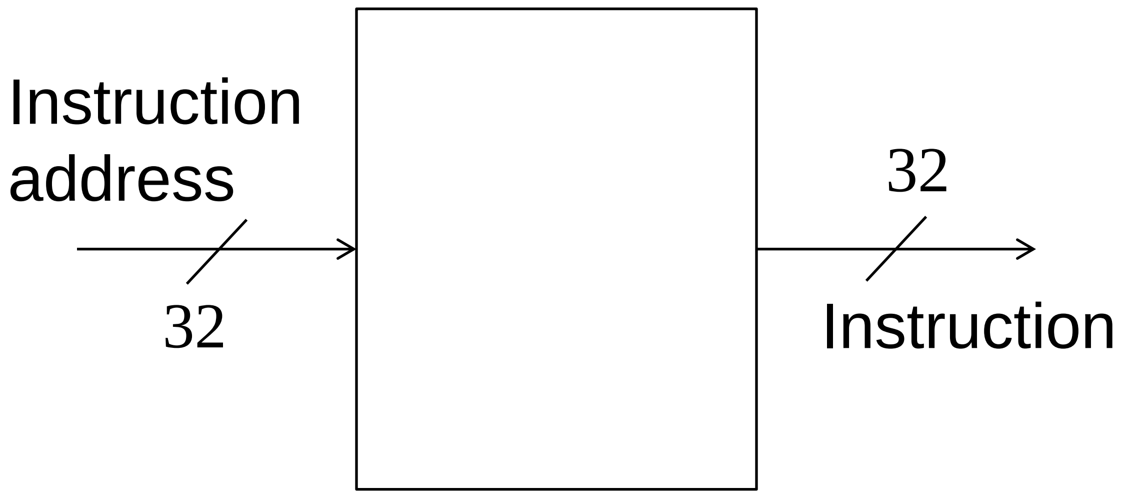
a. Data memory unit b. Sign-extension unit



a. Registers b. ALU

Instruction Memory, Adder, ALU

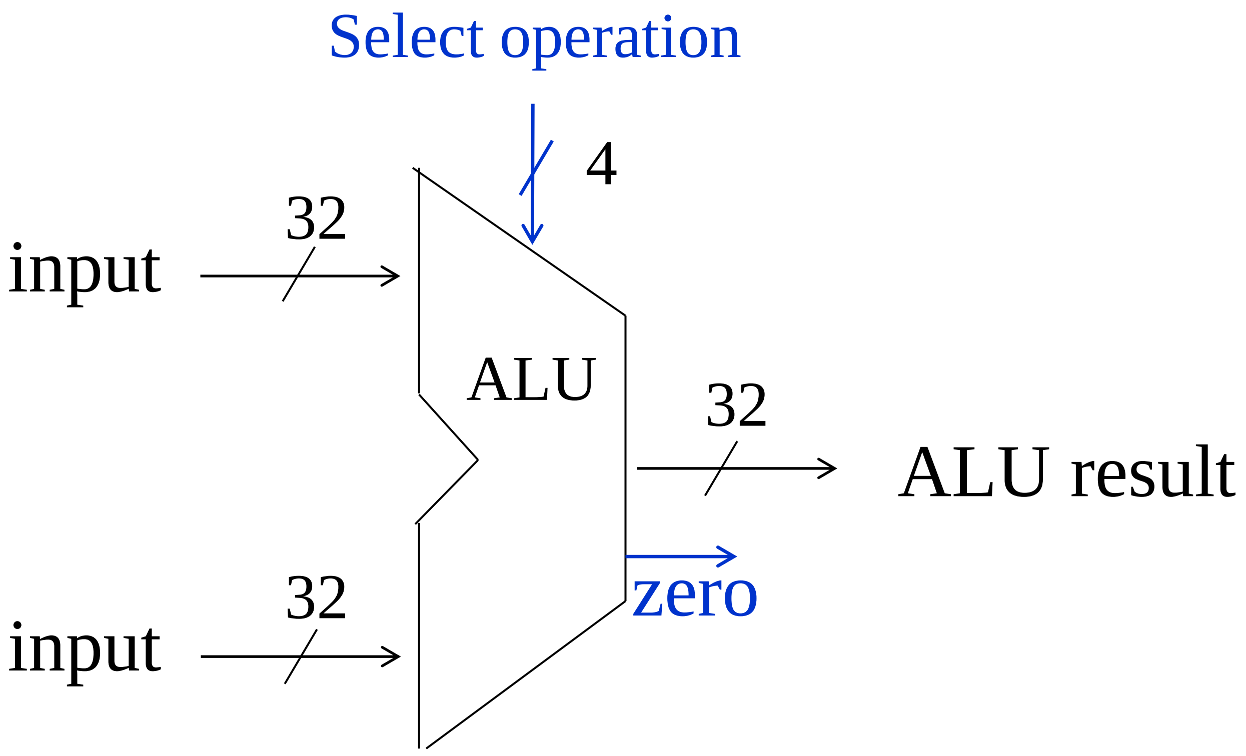
- Combinational blocks: no states, nothing to do with clock



Instruction memory

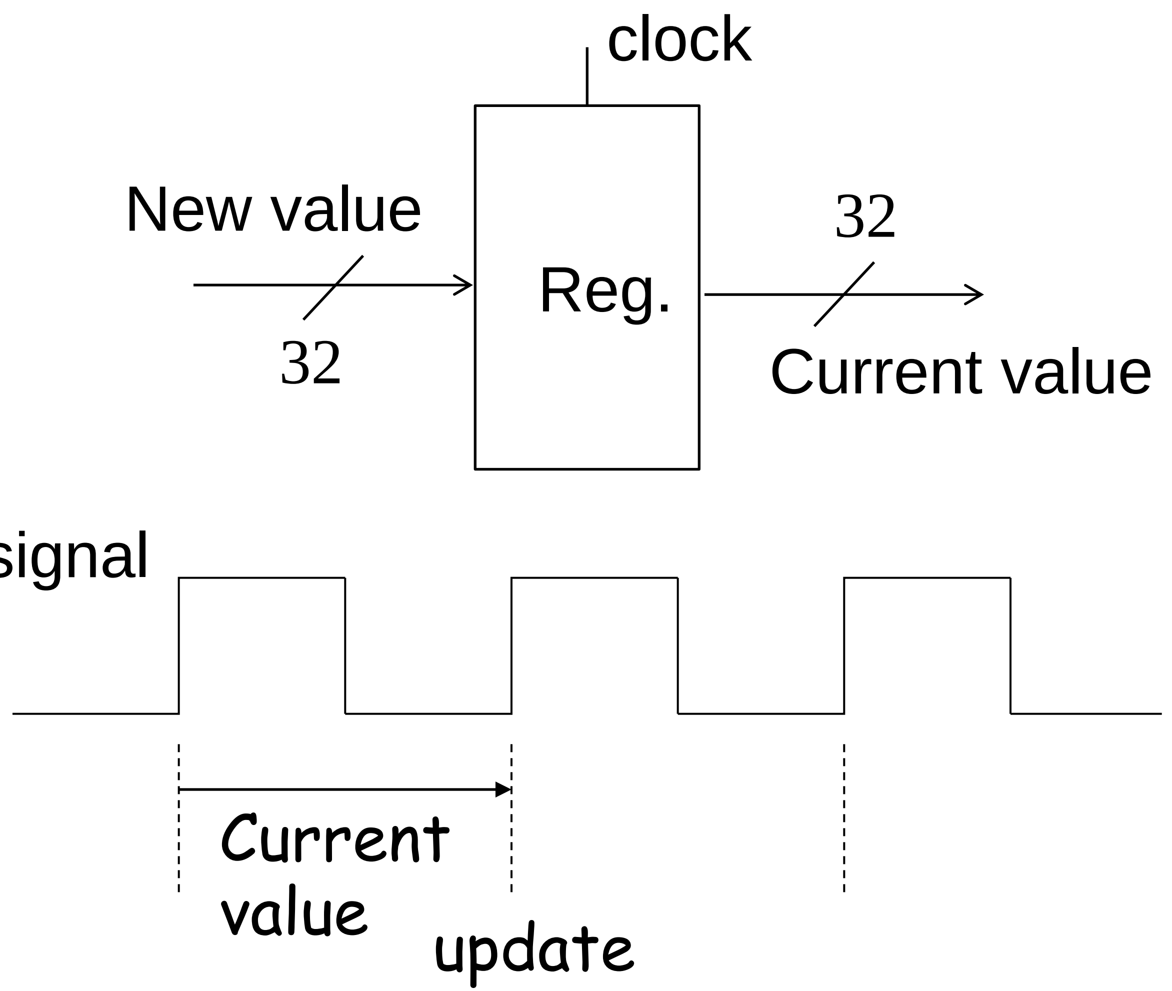
Instruction Memory, Adder, ALU

- Combinational blocks: no states, nothing to do with clock



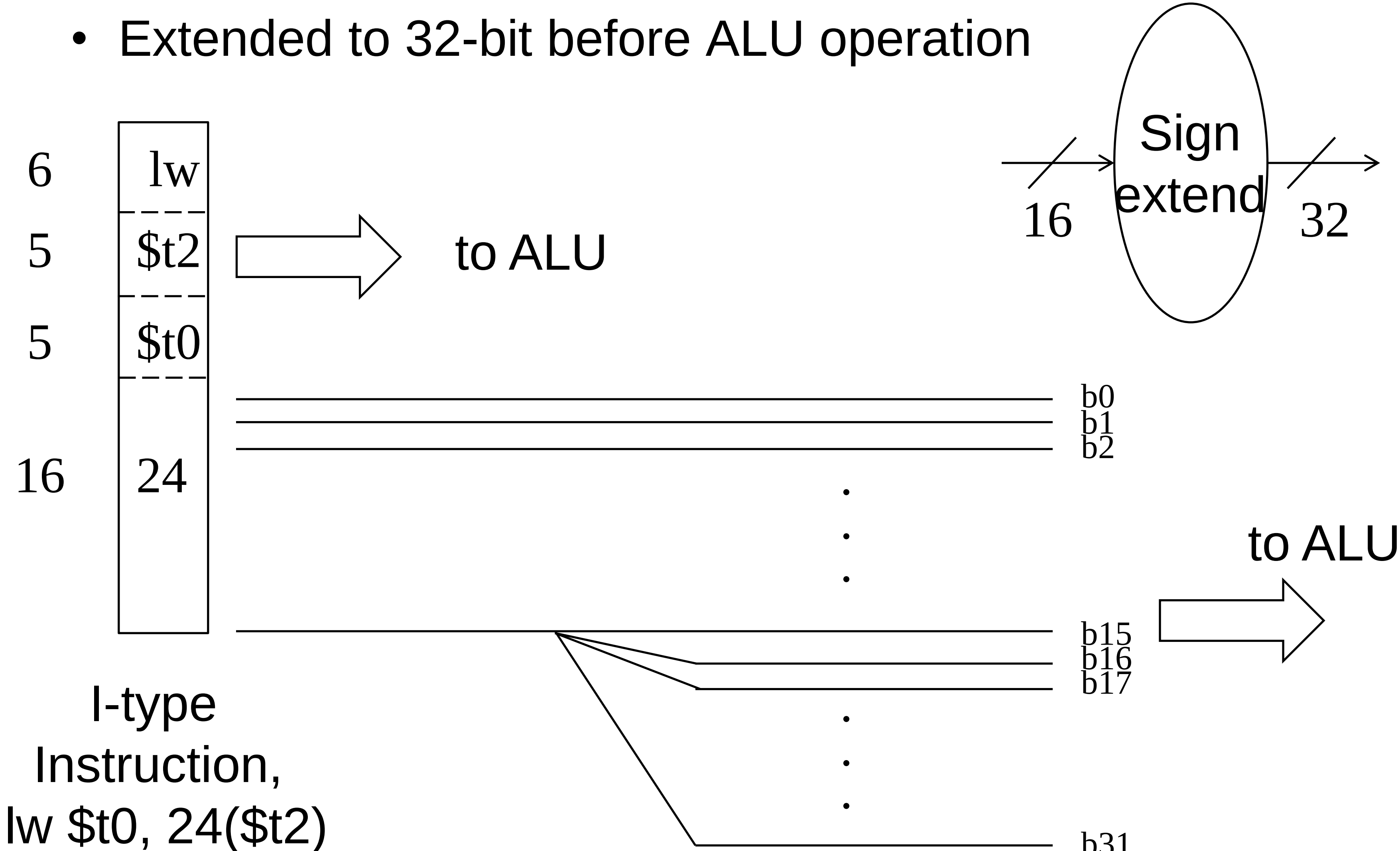
Registers (including PC)

- Sequential blocks: has states, in sync with clock



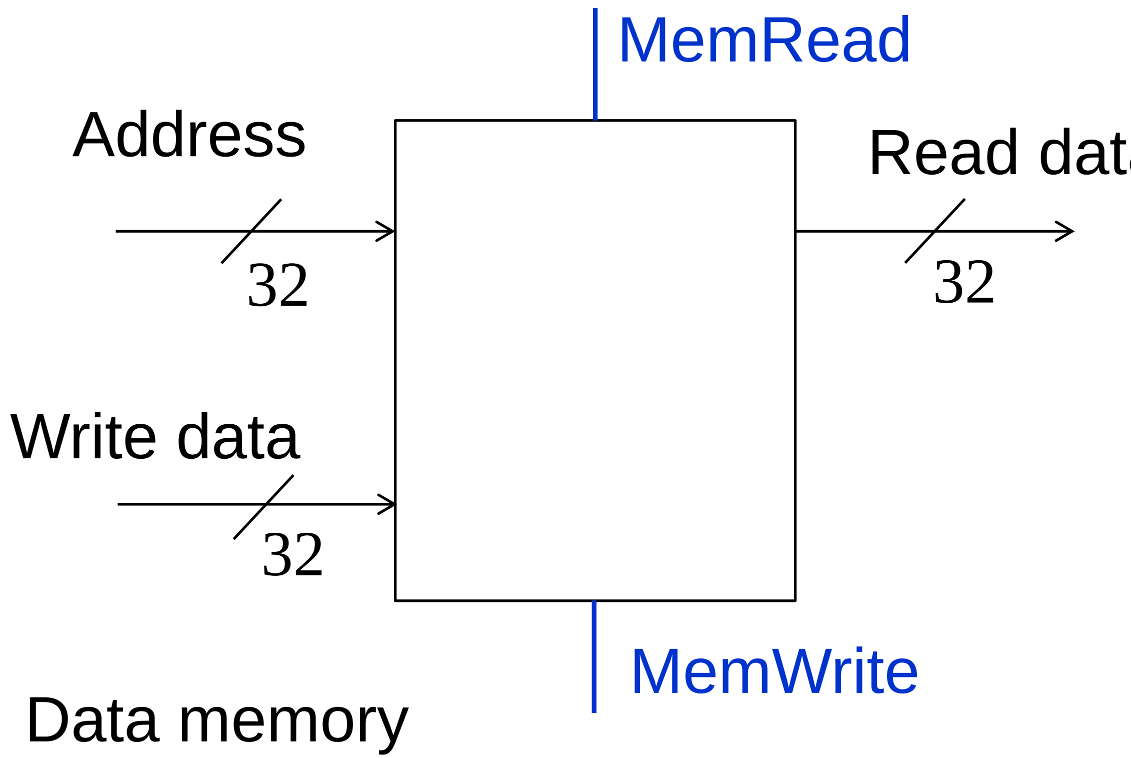
Sign Extension

- 16-bit immediate in instruction (2's complement)
  - Extended to 32-bit before ALU operation

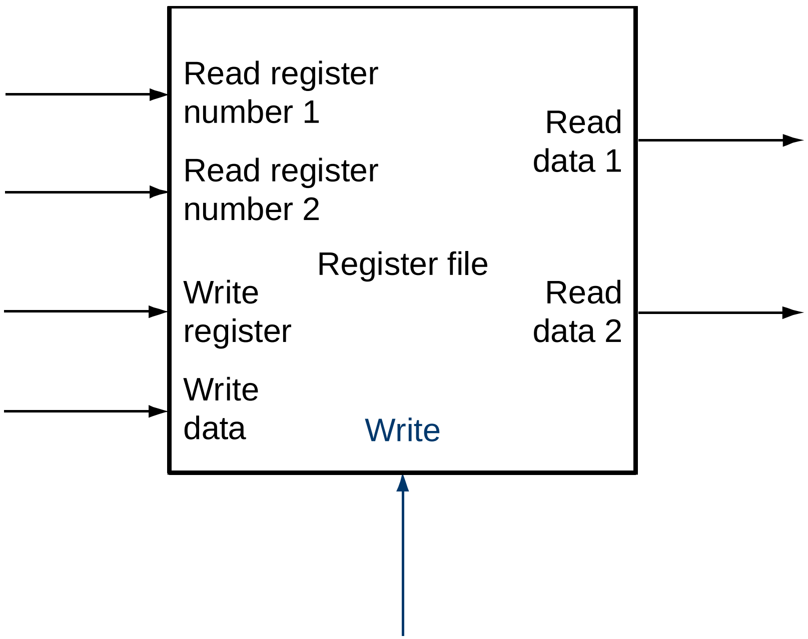


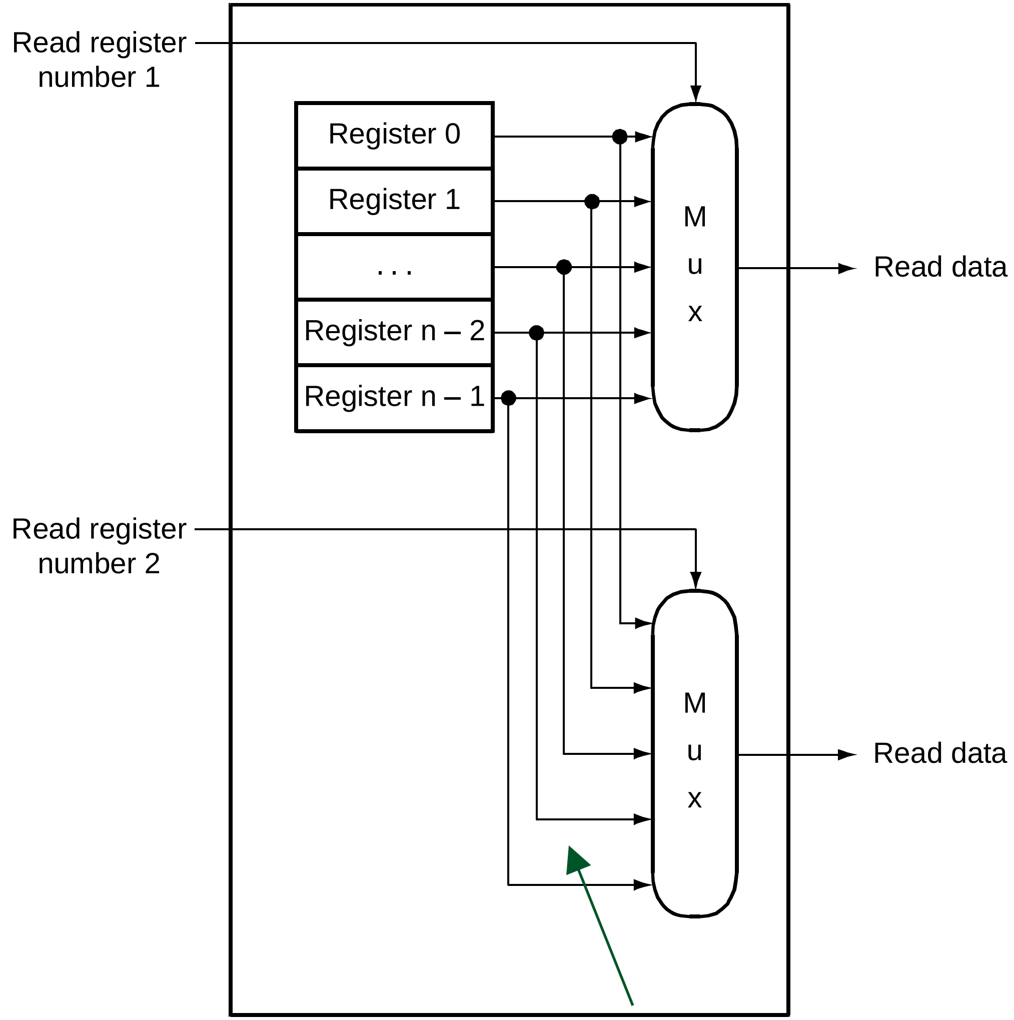
Data Memory

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



Register File

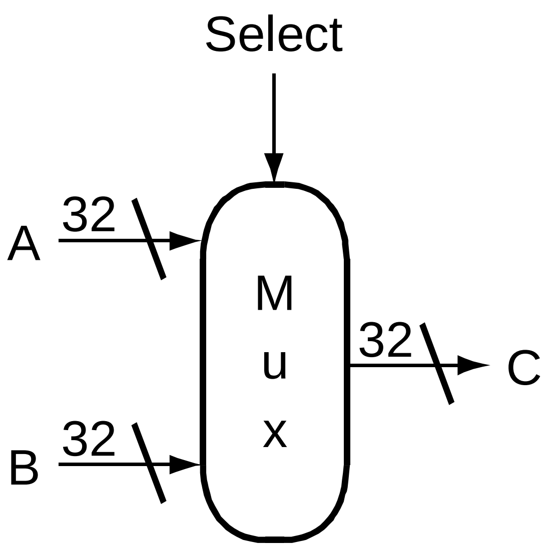


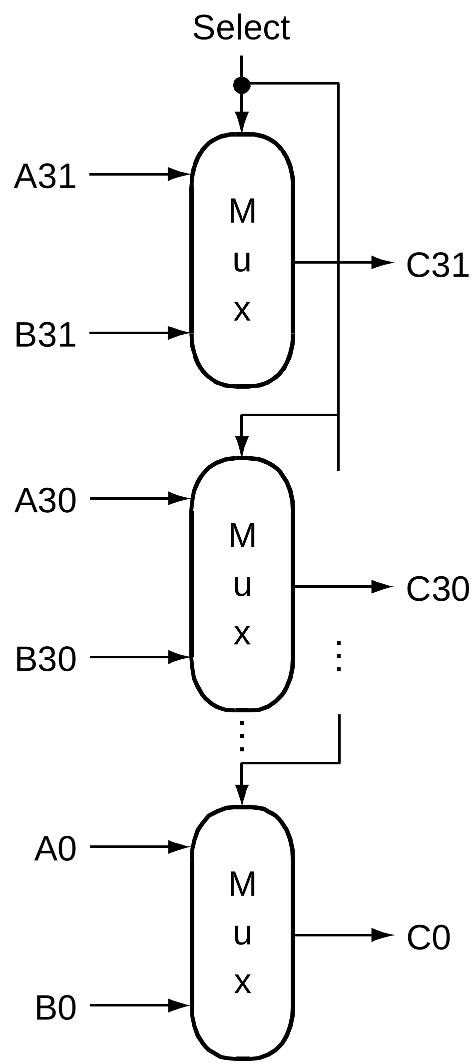


- Register read
  - Register: built with 32 flip-flops

Multiplexer Abstraction (반복)

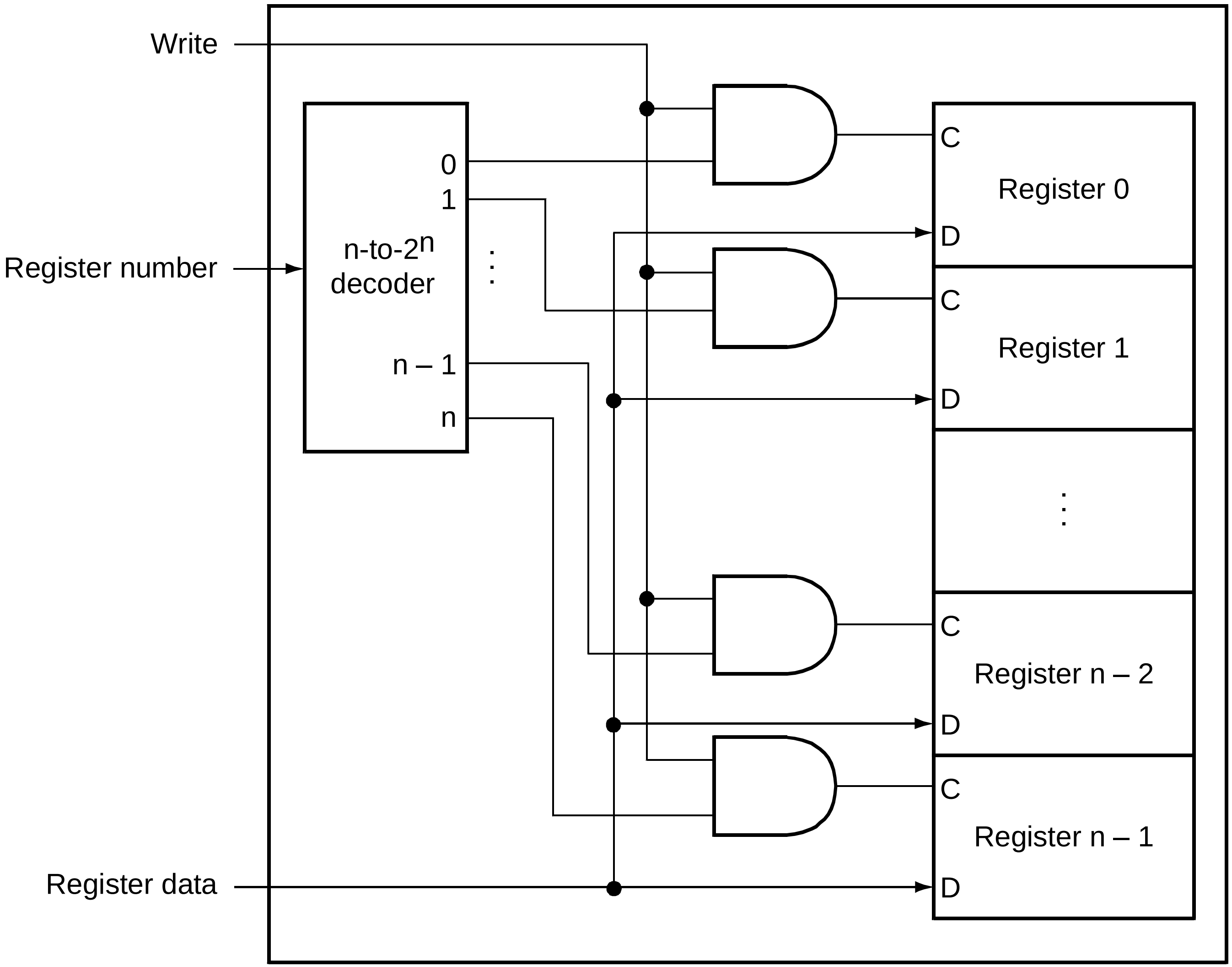
- Make sure you understand the abstractions!
- 32 of 2-to-1 mux





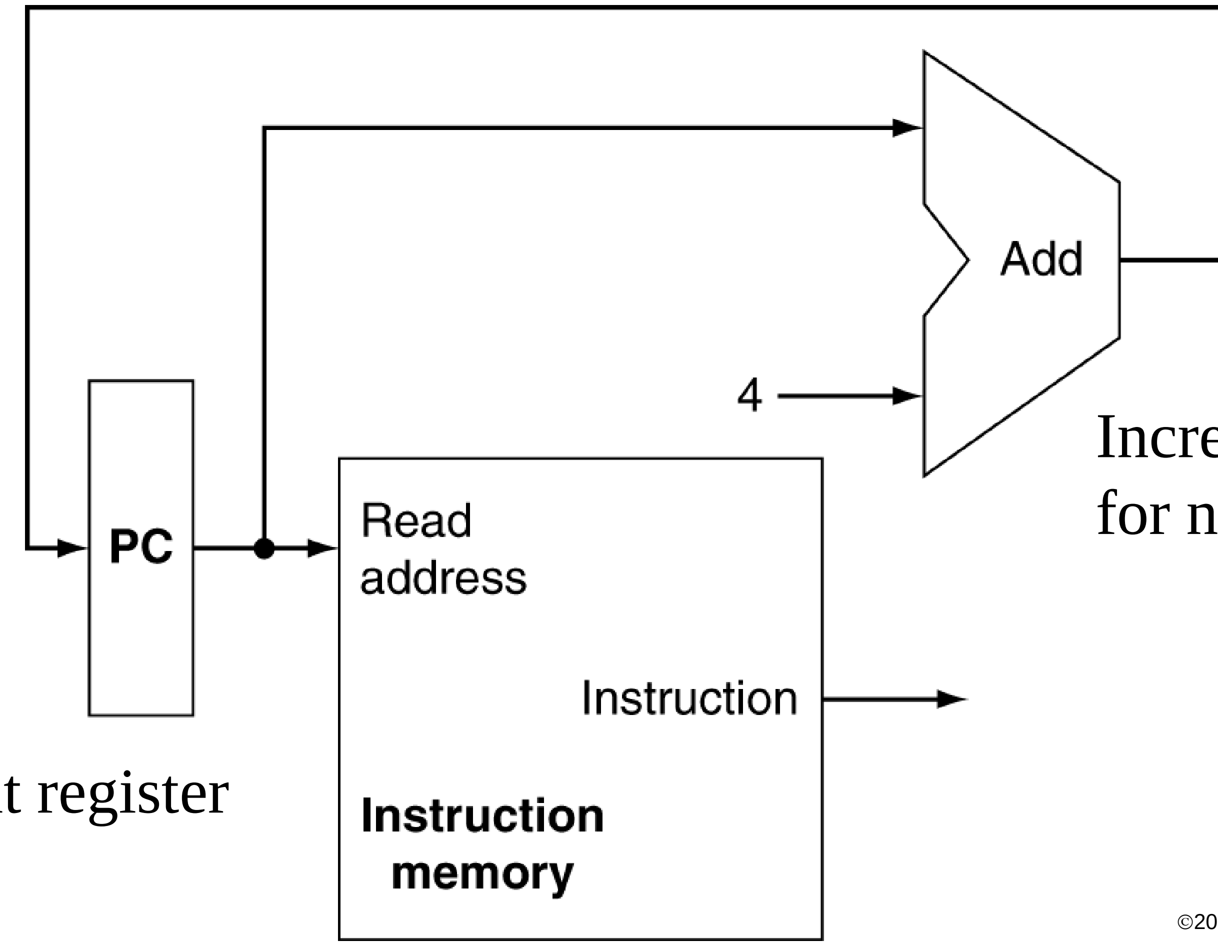
Register File

- Write



Instruction Fetch

- What happens?
- Combinational or sequential?

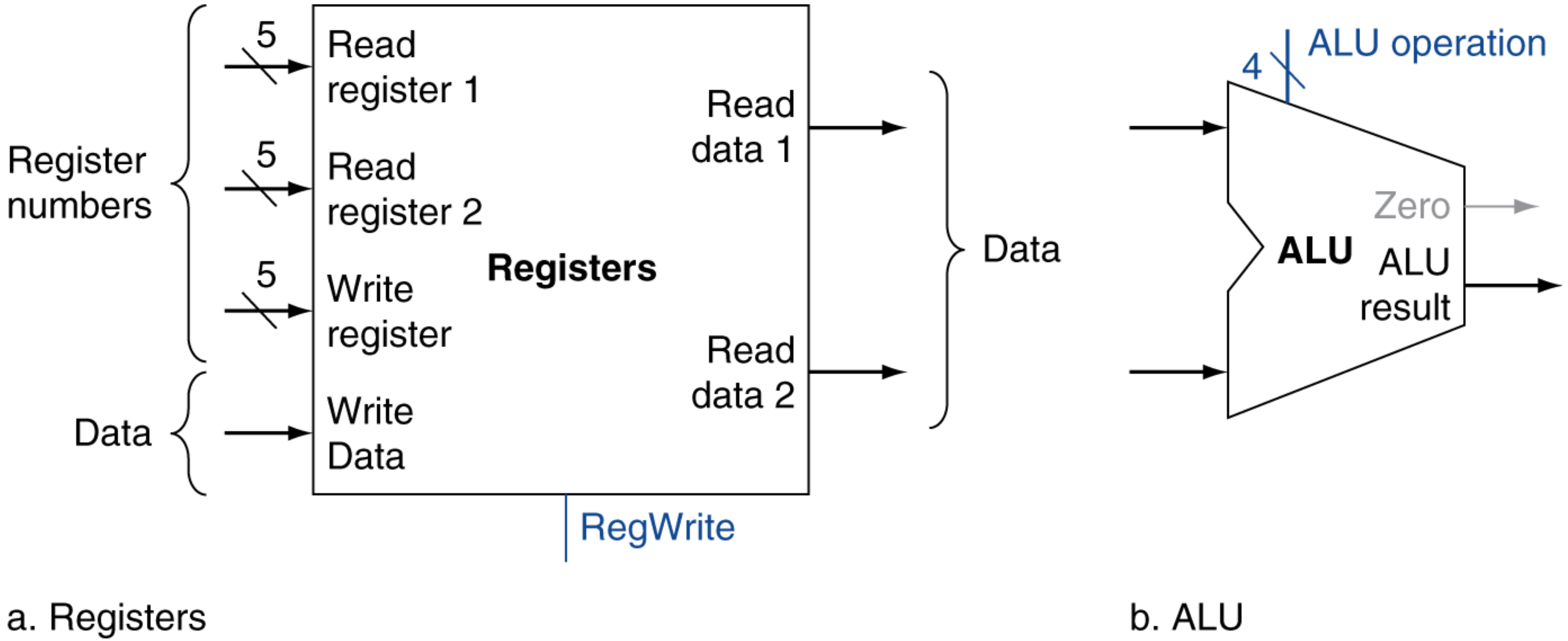


Increment by 4
for next instruction

32-bit register

R-Format Instructions

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



How do we execute MIPS instructions?

- ALU instructions

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

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

- Data transfer instructions

  lw $t0, 12($t1) // I-type

  st $t0, 12($t1)

- Branch instructions

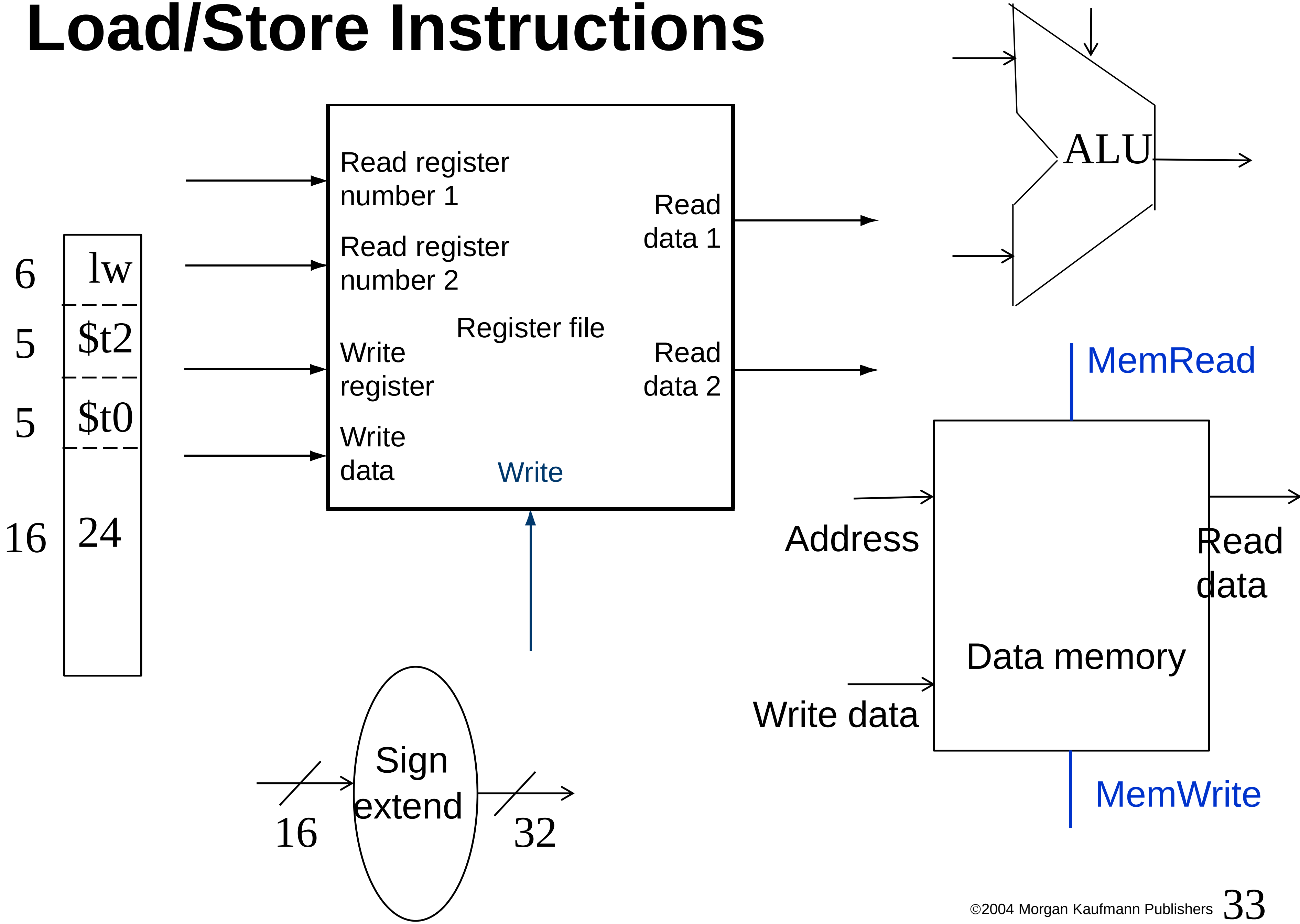
  beq $t0, $t1, 8 // I-type

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

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

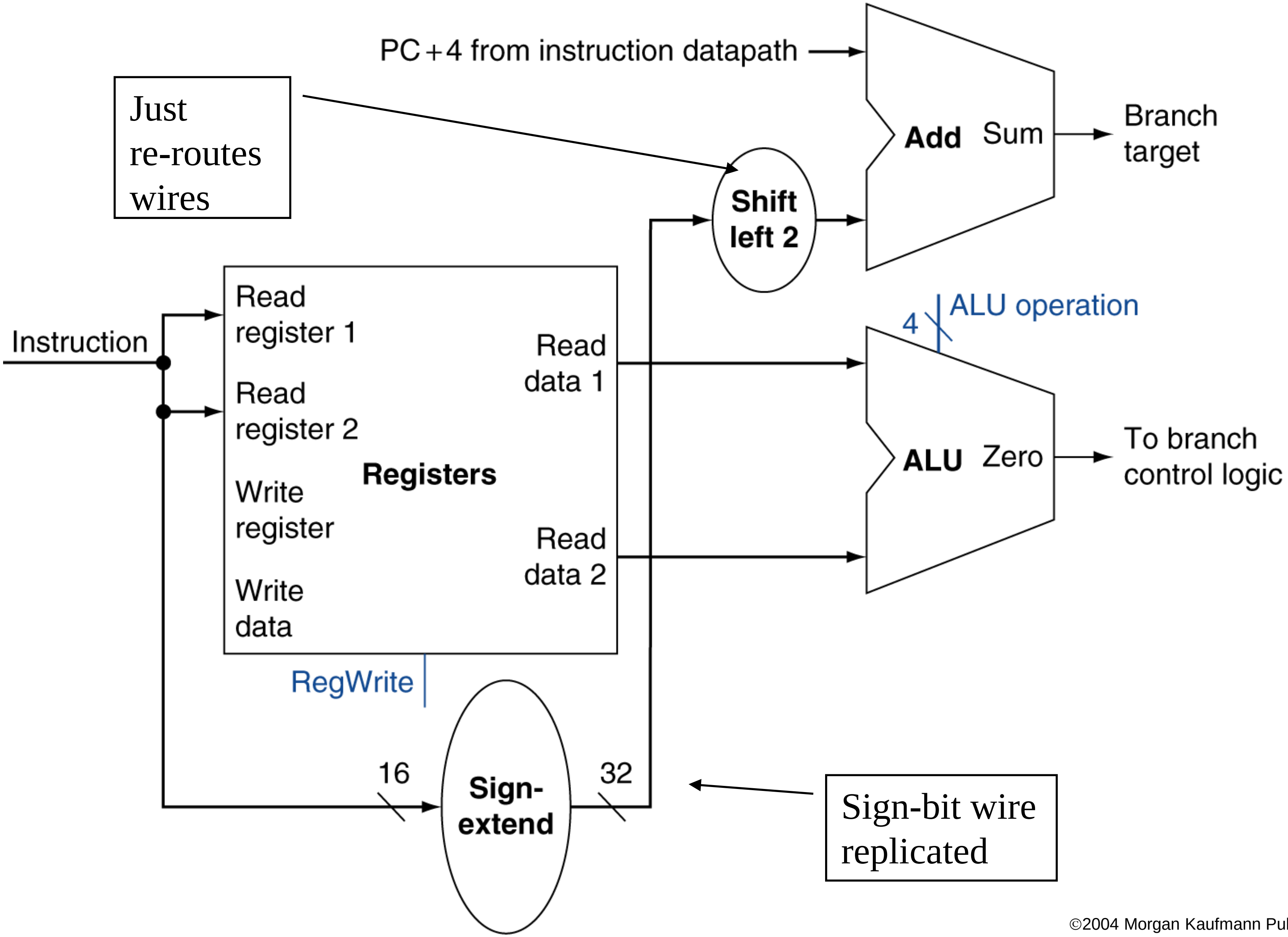
Load/Store Instructions



Branch Instructions

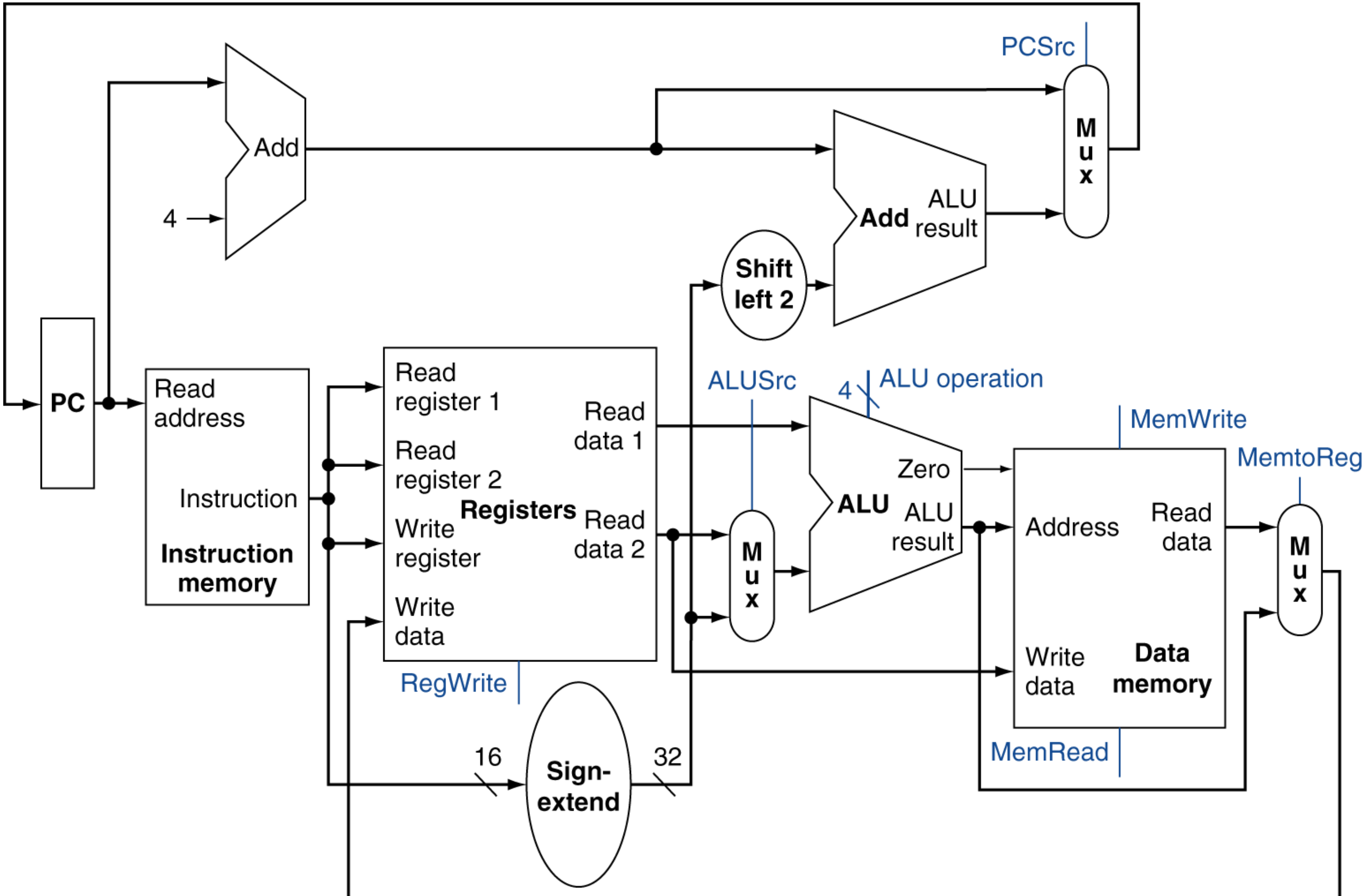
- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Branch Instructions



Building the Datapath

- Use multiplexors to stitch them together

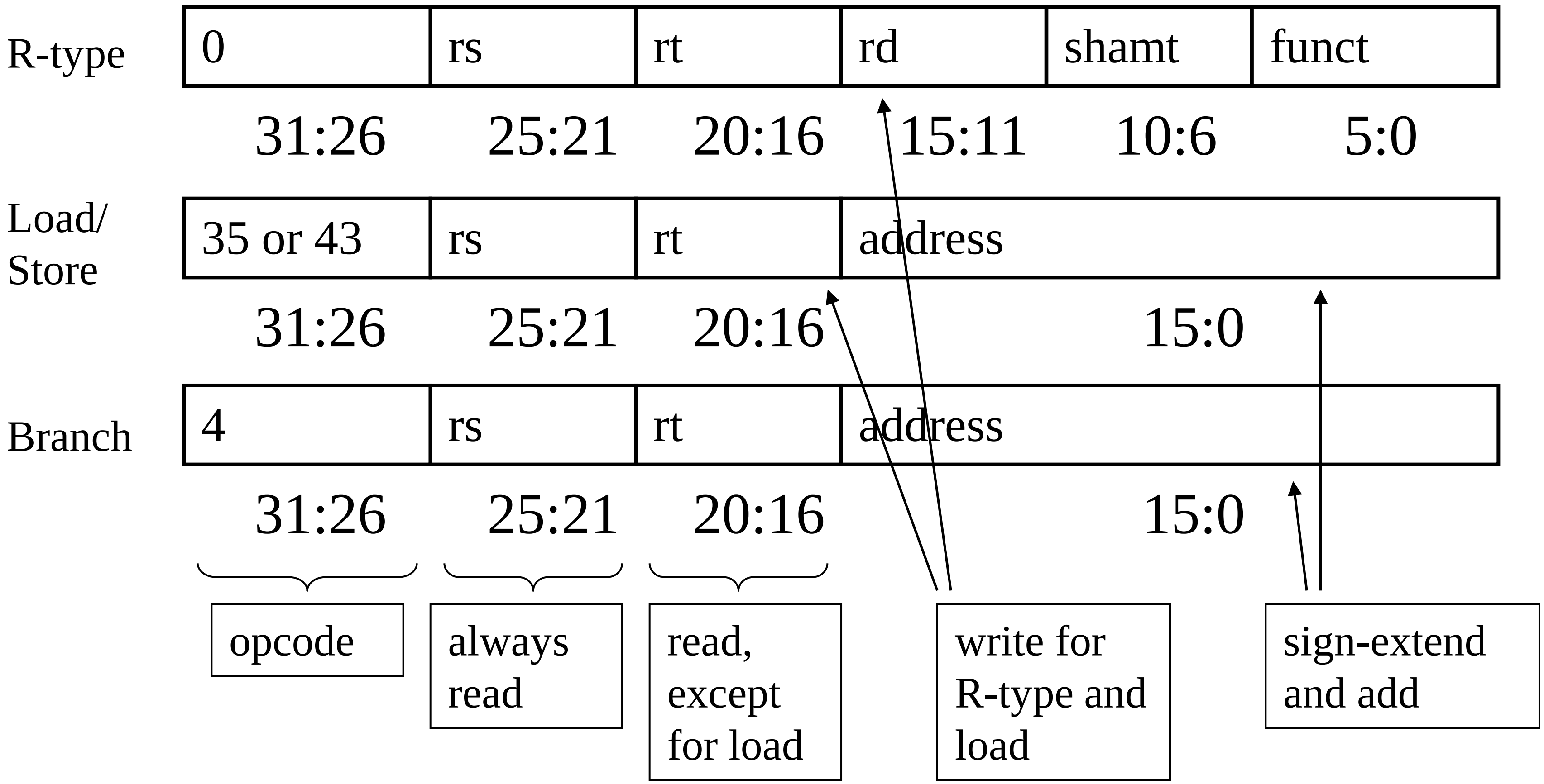


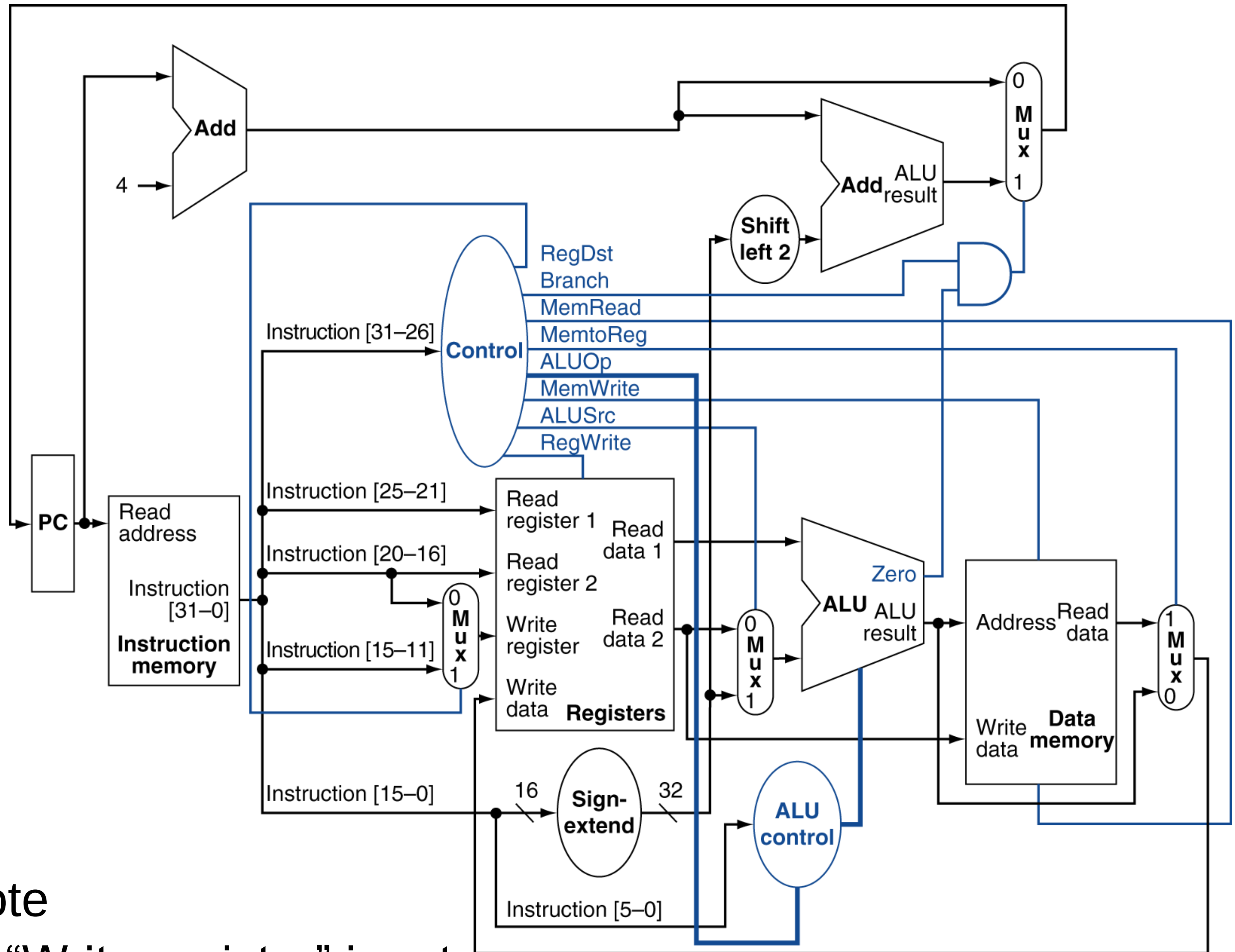
Building the Datapath

- What is datapath?
  - Major functional units, flow of data between them
  - † Must execute all MIPS instructions
- Use multiplexers where alternate data sources are used for different instructions
  - Colored control signals
- Does it look familiar?
  - Because you understand high-level behavior (ISA)
- Datapath design is high-level organization
  - Affect CPI and clock cycle time
    - Will come back to this in pipelined design

Building the Datapath

- Datapath derived from instruction

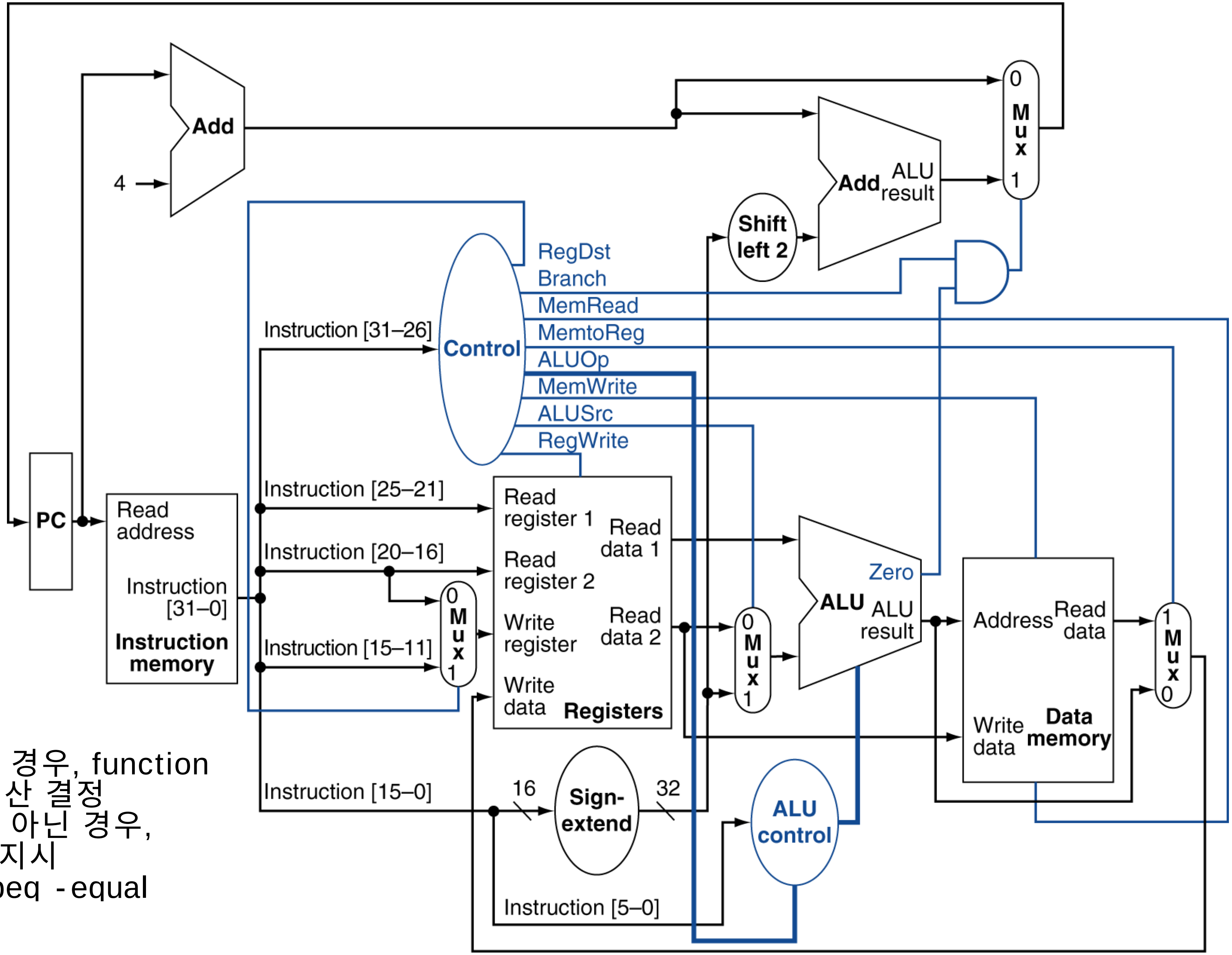




- Note
  - “Write register” input
  - Colored control signals – what is “decode”?

Control Design

- Control
  - Given instruction, determine values of control signals
  - That's decoding
- Note: simultaneous decode and register read
- Do you understand the terms: datapath and control?
  - Think about any chip design (e.g., MPEG2, DES)
    - † Performance - quantitative approach
- ISA designer (architect) consider them during ISA design
  - Higher-level design require deep understanding of lower-level designs



ALU control
1. R-format인 경우, function field 참고후 연산 결정
2. R-format이 아닌 경우, alu operation 지시
ex) lw - add,beq -equal

❑ Note

- Simultaneous decode and register read

Control Design

- Decoding: determine the values of colored control signals
  - Selecting operations to perform (ALU, read/write, etc.)
  - Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
  - opcode and function code

add $8, $17, $18

`000000`	`10001`	`10010`	`01000`	`00000`	`100000`

`op`	**`rs`**	**`rt`**	**`rd`**	**`shamt`**	**`funct`**

Control Design

- Two separate control units for simplicity
  - Lower ALU control unit: select ALU operation
  - Upper control unit: the rest of control
- Let’s design lower ALU control unit first

Lower ALU Control Unit

- What should the ALU do with this instruction? add
  - Example: lw $1, 100($2)

35	2	1	100

op	rs	rt	16-bit offset

- What ALU should do for “beq”? minus
- What ALU should do for R-type instructions? function field 참고

Lower ALU Control Unit

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

- 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

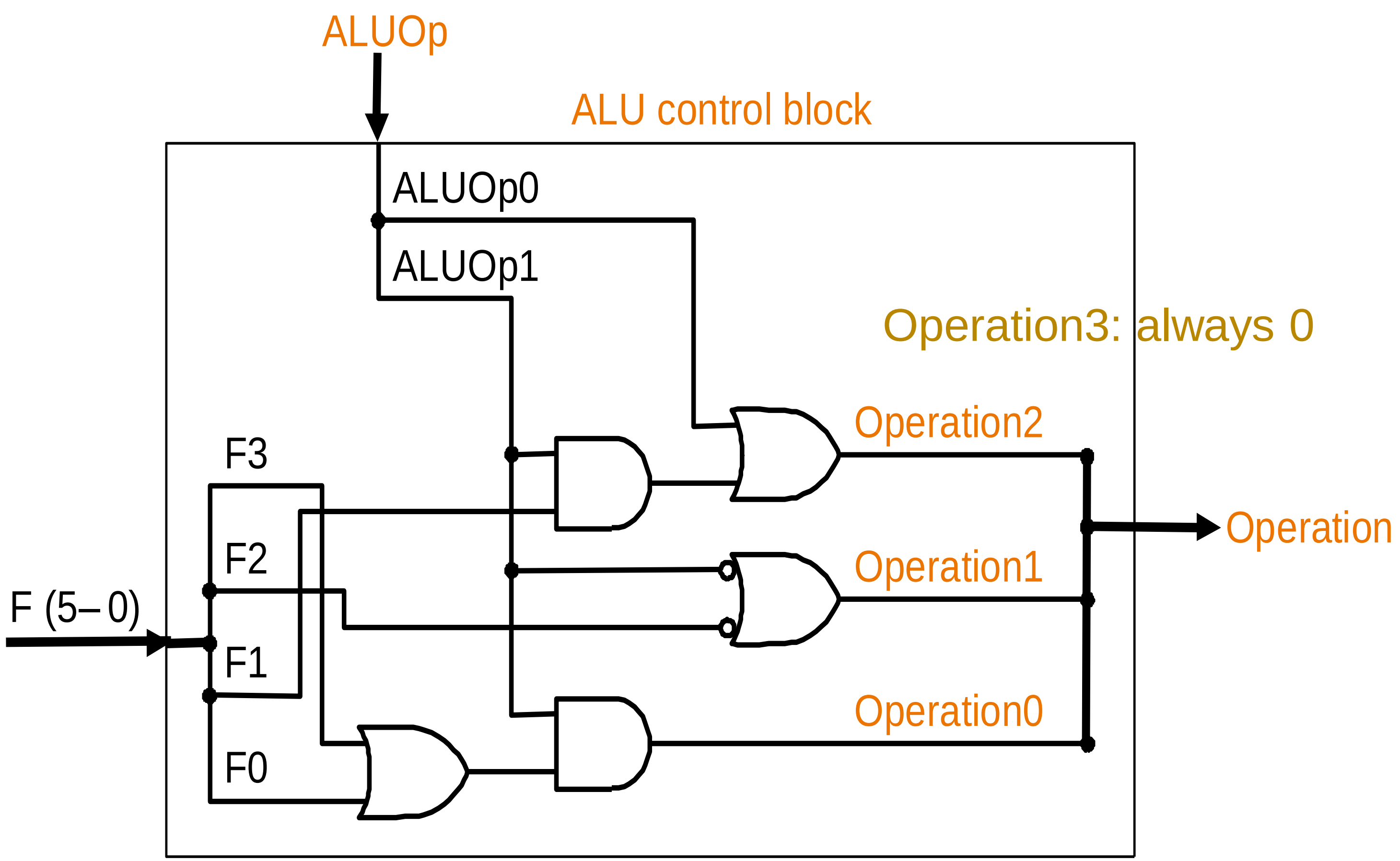
Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	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						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

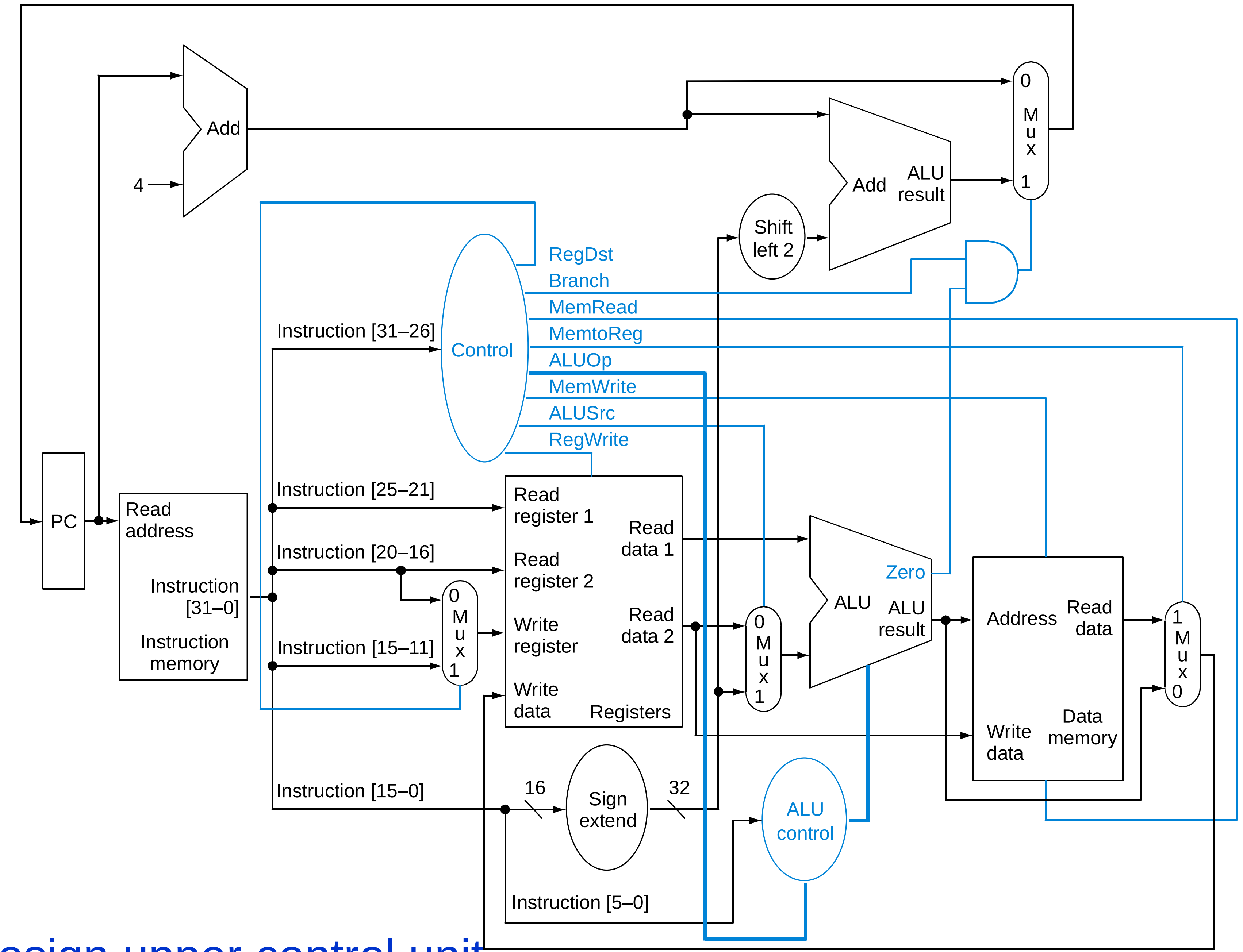
don't care
- 2bit 사용인데 3가지 종류만 사용하기 때문에 (don't care 사용시 logic 디자인 유리)

Lower ALU Control Unit

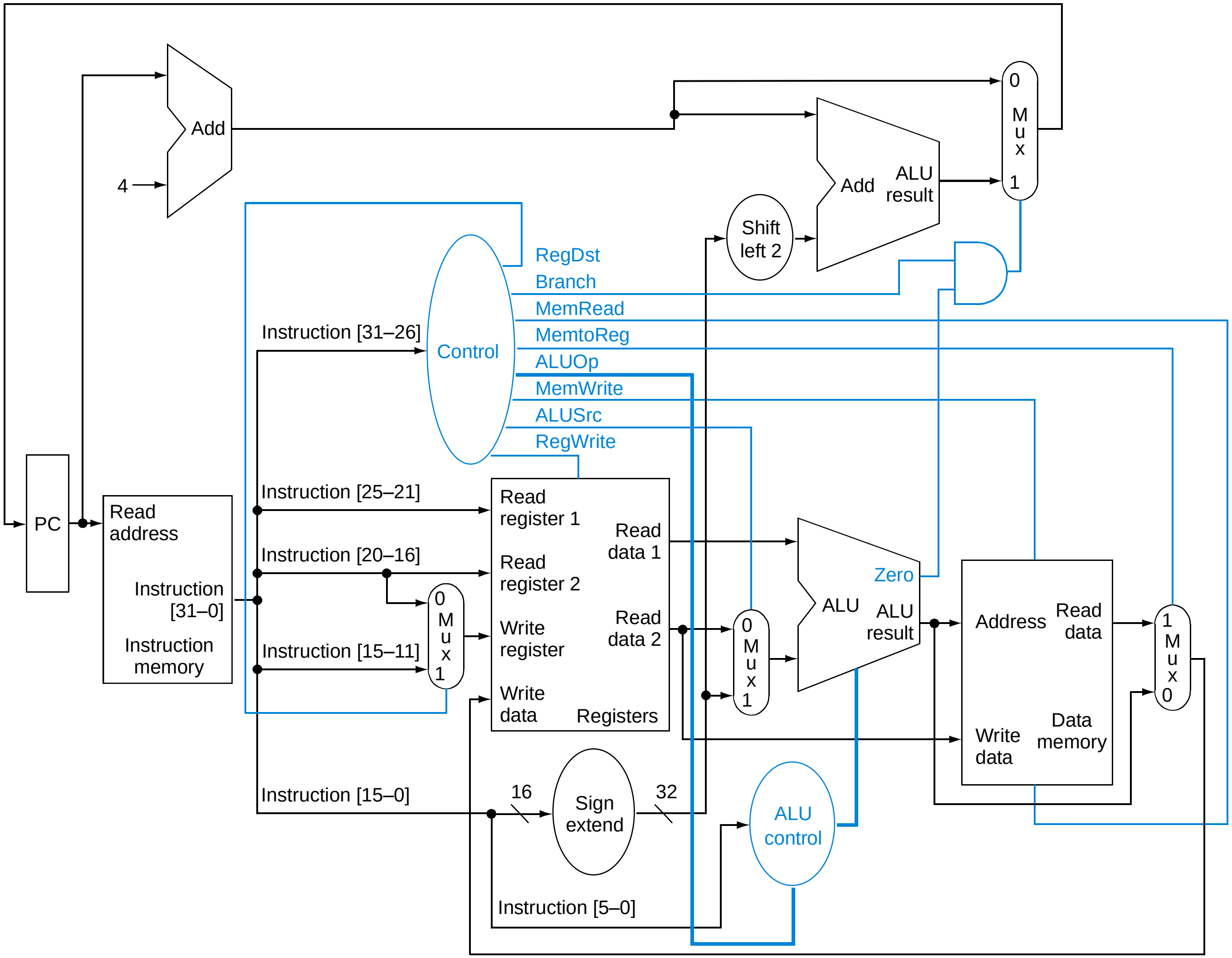
- Simple combinational logic (truth tables) – run CAD tools



카르노맵 - 쉬운 algorithm 실용적x
VLSI cad tools - truth table을 optimal combinational logic 만들어줌



Design upper control unit

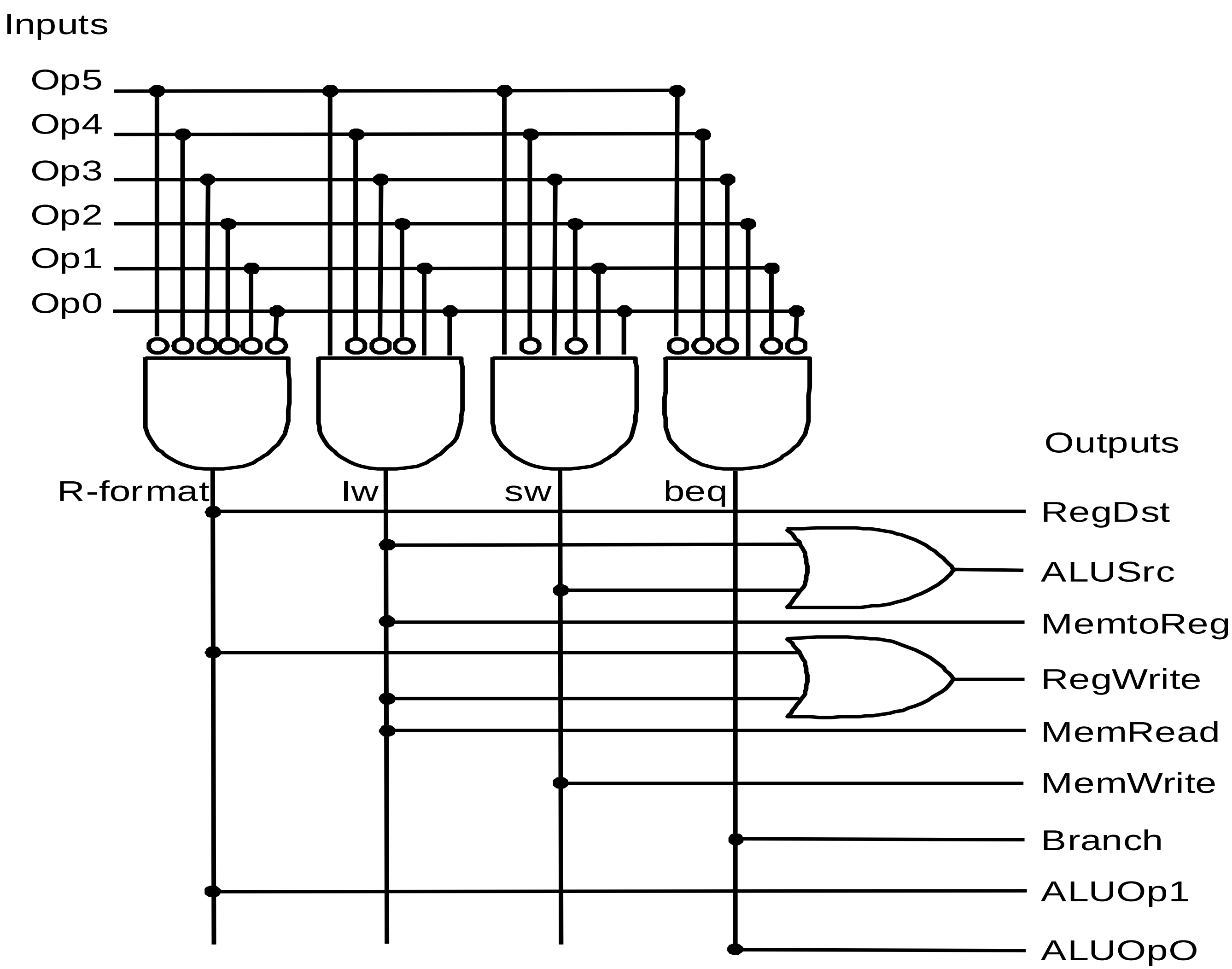


register에 write
하지 않아서
don't care 처리 가능

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

Upper Control Unit

- Simple combinational logic (truth tables)

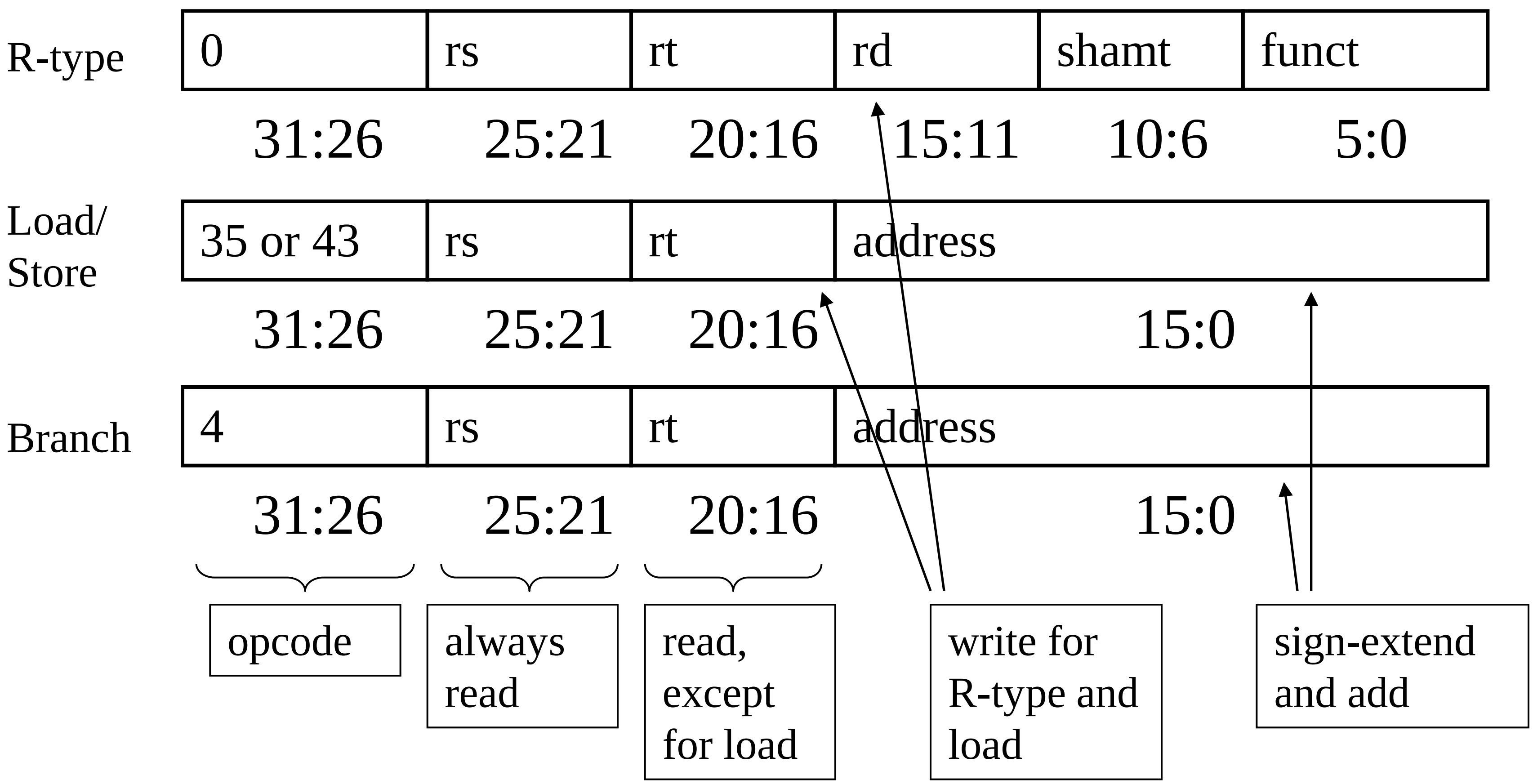


Instruction Decode (반복)

- Where is it in the datapath?
  - Not shown (will come back to it)
  - Instead, what is shown is “read two operands (registers)”
    - But we do not know the instruction
      - † It does no harm
- RISC
  - Parallel instruction decoding and operand access

Control Design (반복)

- Control signals derived from instruction



Implemented 8 instructions

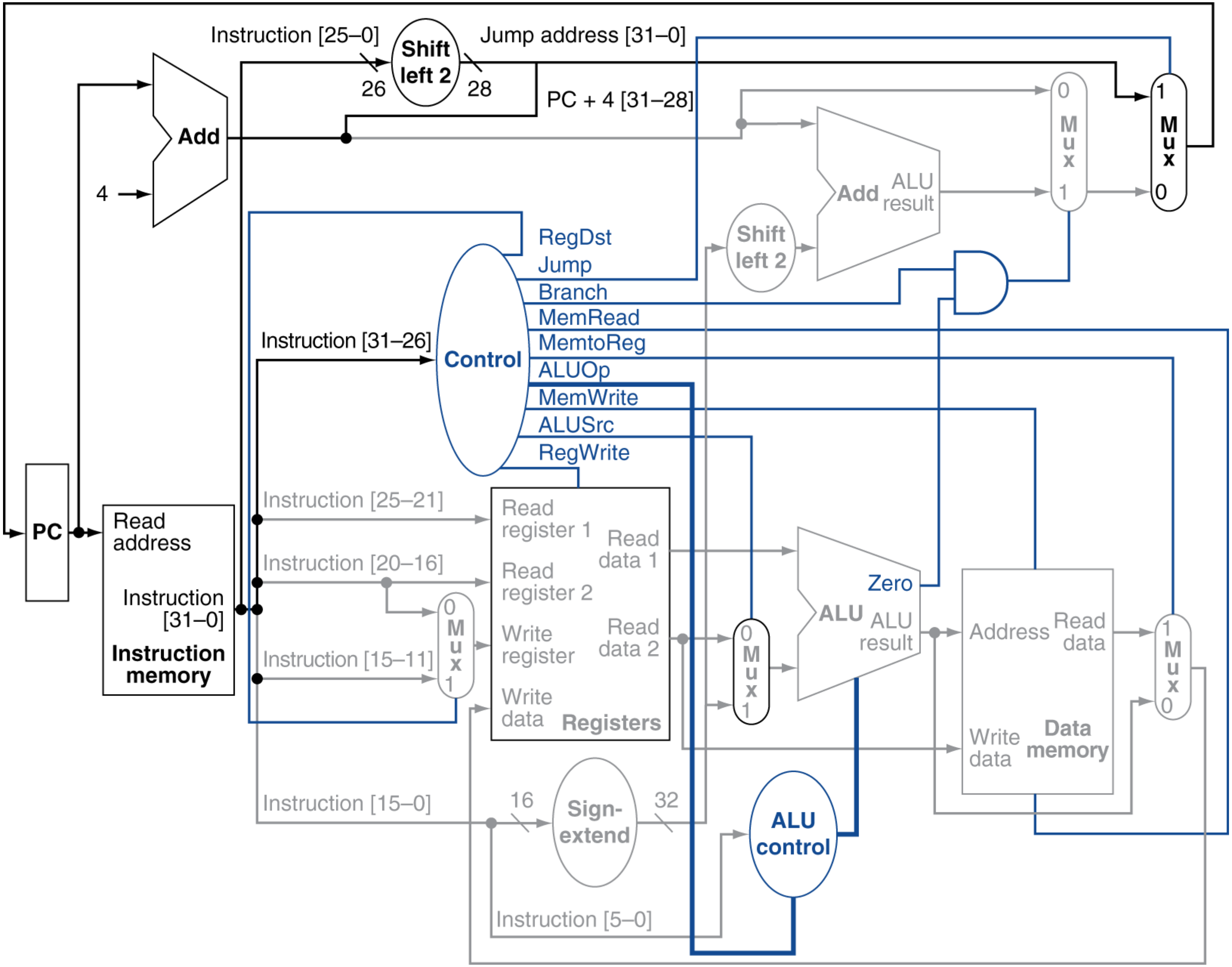
- Processors have many more instructions
- What if we want to add “jump” instruction?
  - Same principles
  - Only more datapath and control

Want to implement “jump” instruction?

- More datapath and control

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
    - † 28-bit byte address
- Need an extra control signal decoded from opcode



Instruction [25–0]
Shift left 2
Jump address [31–0]
26
28
PC + 4 [31–28]
Add
4
0
M u x
1
1
M u x
0
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
PC
Read address
Instruction [31–0]
Instruction memory
0
M u x
1
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
1
M u x
0
Instruction [15–0]
16
Sign-extend
32
ALU control
Instruction [5–0]

Control for “jump” instructions

- 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	ALUp0	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
Jump	X	X	X	0	X	0	X	X	X	1

read 같은 instruction은 data을 변경하지 않지만,
write의 경우 data를 변경하기 때문에 꼭 설정해줘야 한다.

Processor Design Done

- How do we fabricate our processor?
  - What if we visit Intel or Samsung?
    - Same situation when developing industry 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

† You will do the first two steps as class project

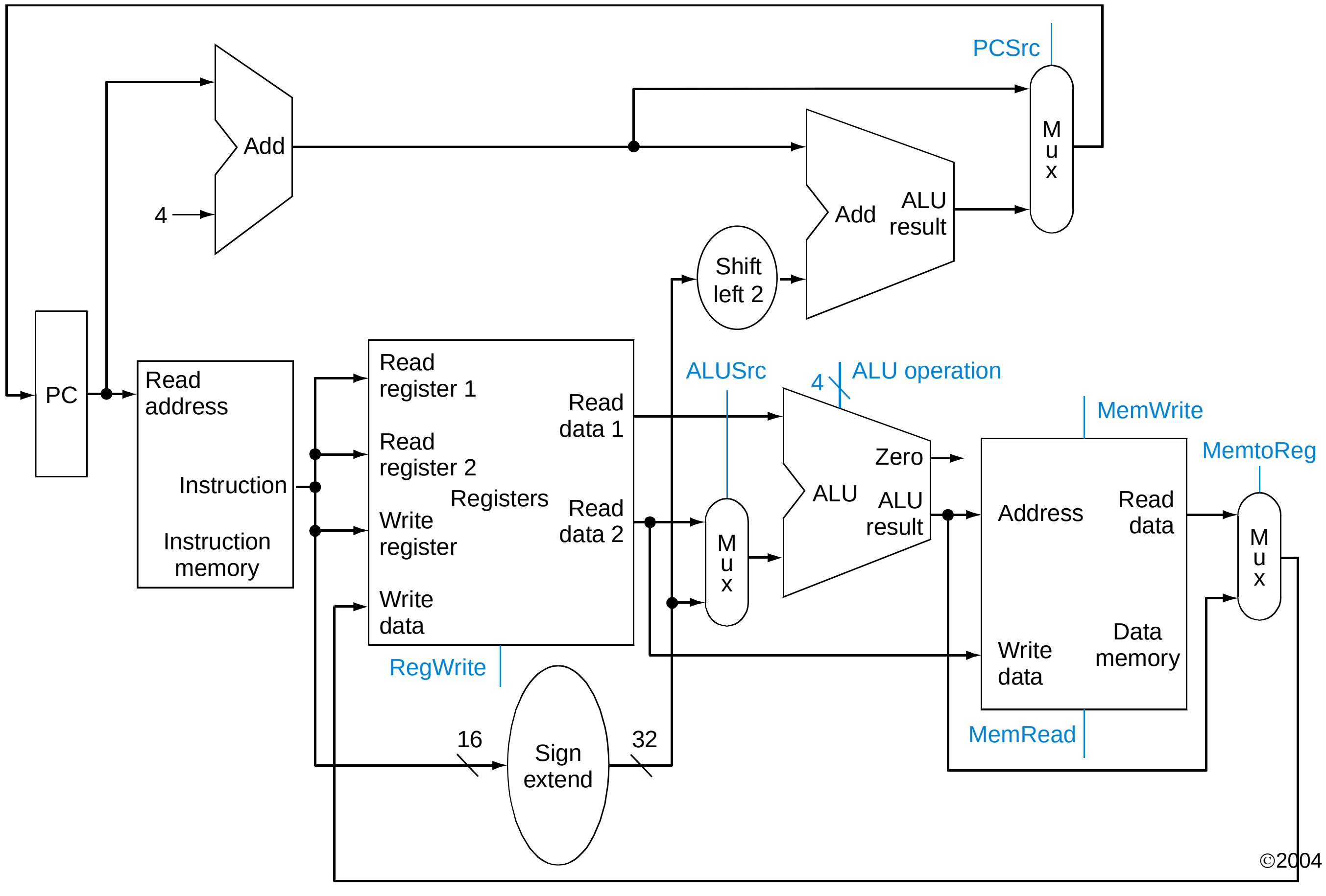
- See related materials in class homepage

RISC Processor Design Project

- Descriptions in class homepage
- TA hour?

Single Cycle Implementation

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

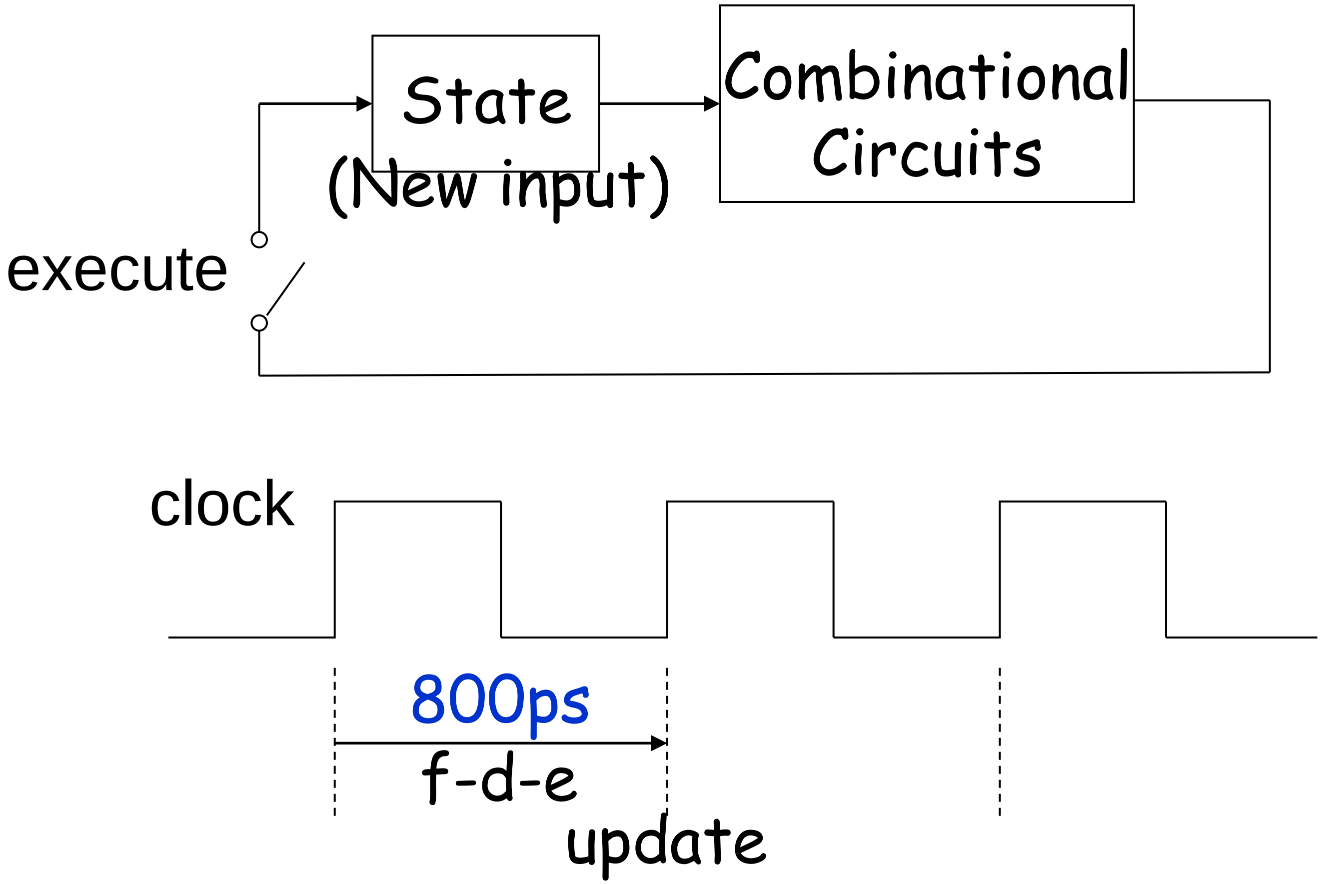


Single cycle implementation

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

- To think about
  - Longest delay determines clock period
    - Critical path: load instruction (also think about “mult”)
  - Violates design principle
    - Making the common case fast
  - We will improve performance by pipelining

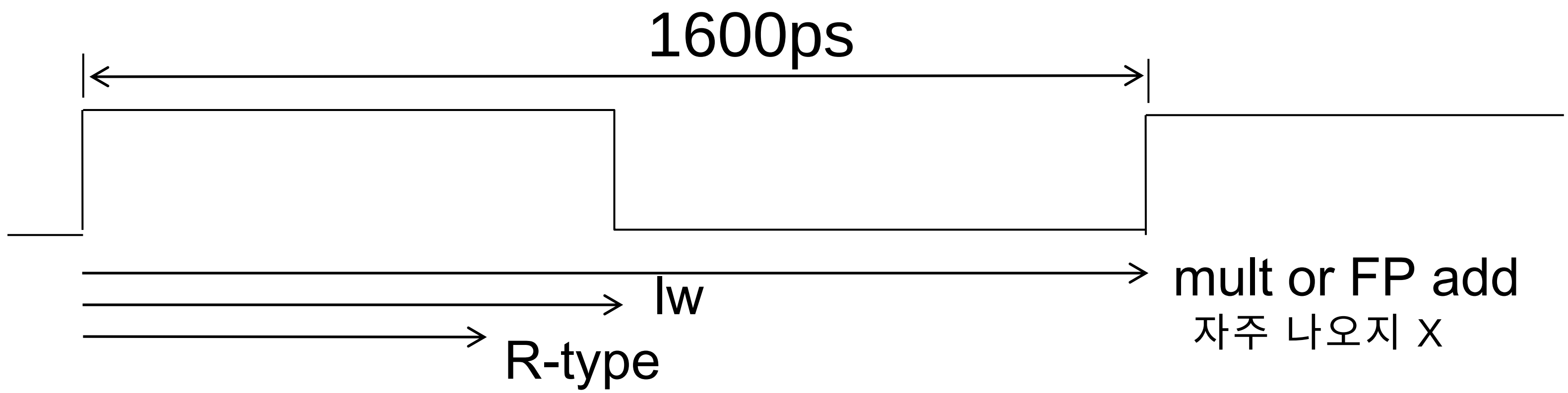
Our Single Cycle Implementation



- Cycle time determined by length of the longest path

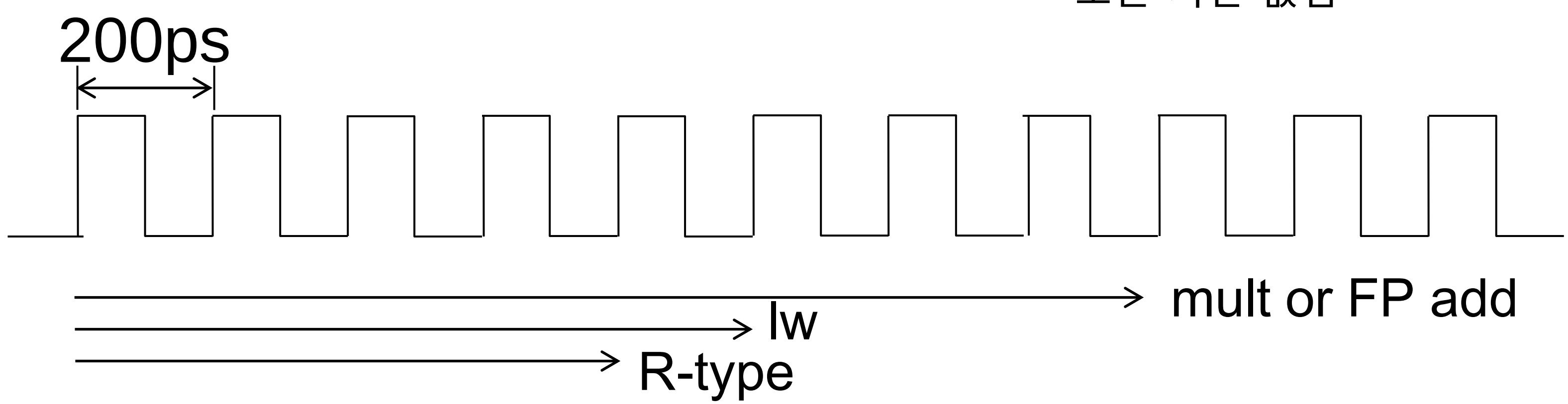
High-Level Org.: CPI and Clock Cycle

- Single-cycle: CPI = 1, clock cycle



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

노는 시간 없앰



- Pipelined: CPI = 1, clock cycle ↓

MIPS Pipeline

- 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

- Each stage run different instruction

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

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

Where we are headed

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

