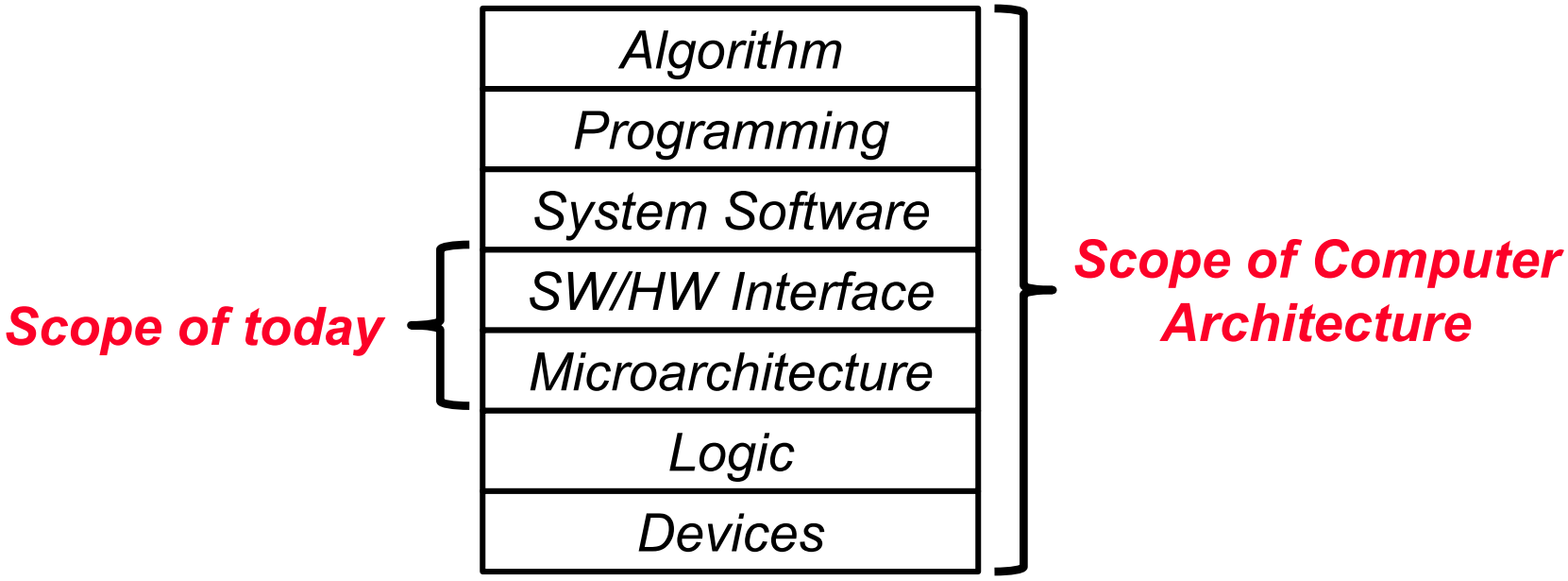
Lecture 4:
CPU – Single Cycle

Hunjun Lee
hunjunlee@hanyang.ac.kr

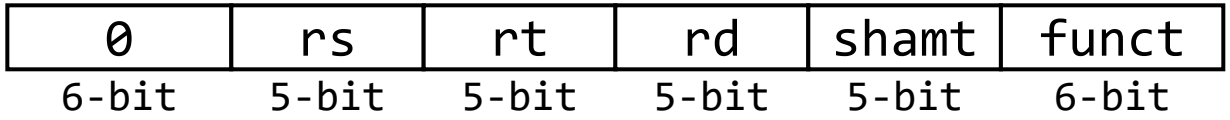
Computing Problem Hierarchy

- Let's point out how your program run in computer!



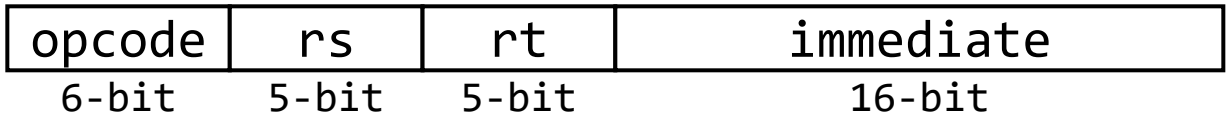
Recall: MIPS instruction formats

- Three simple formats
  - R-type, 3 register operands



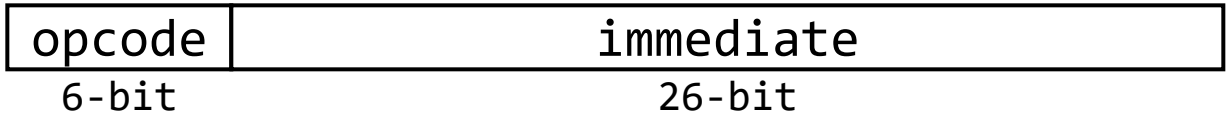
R-type

  - I-type, 2 register operands and 16-bit immediate operand



I-type

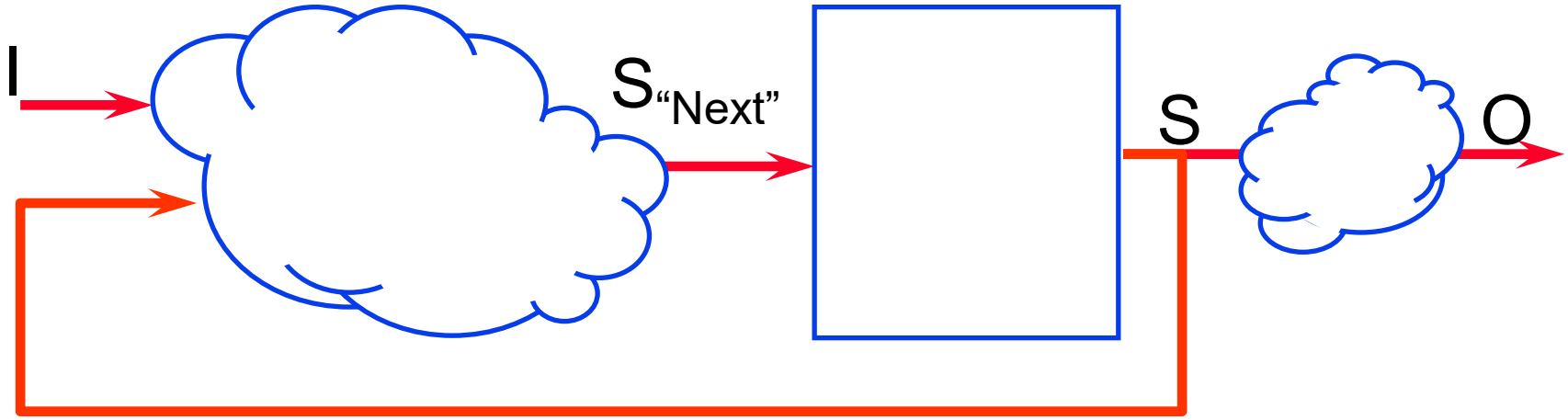
  - J-type, 26-bit immediate operand



J-type

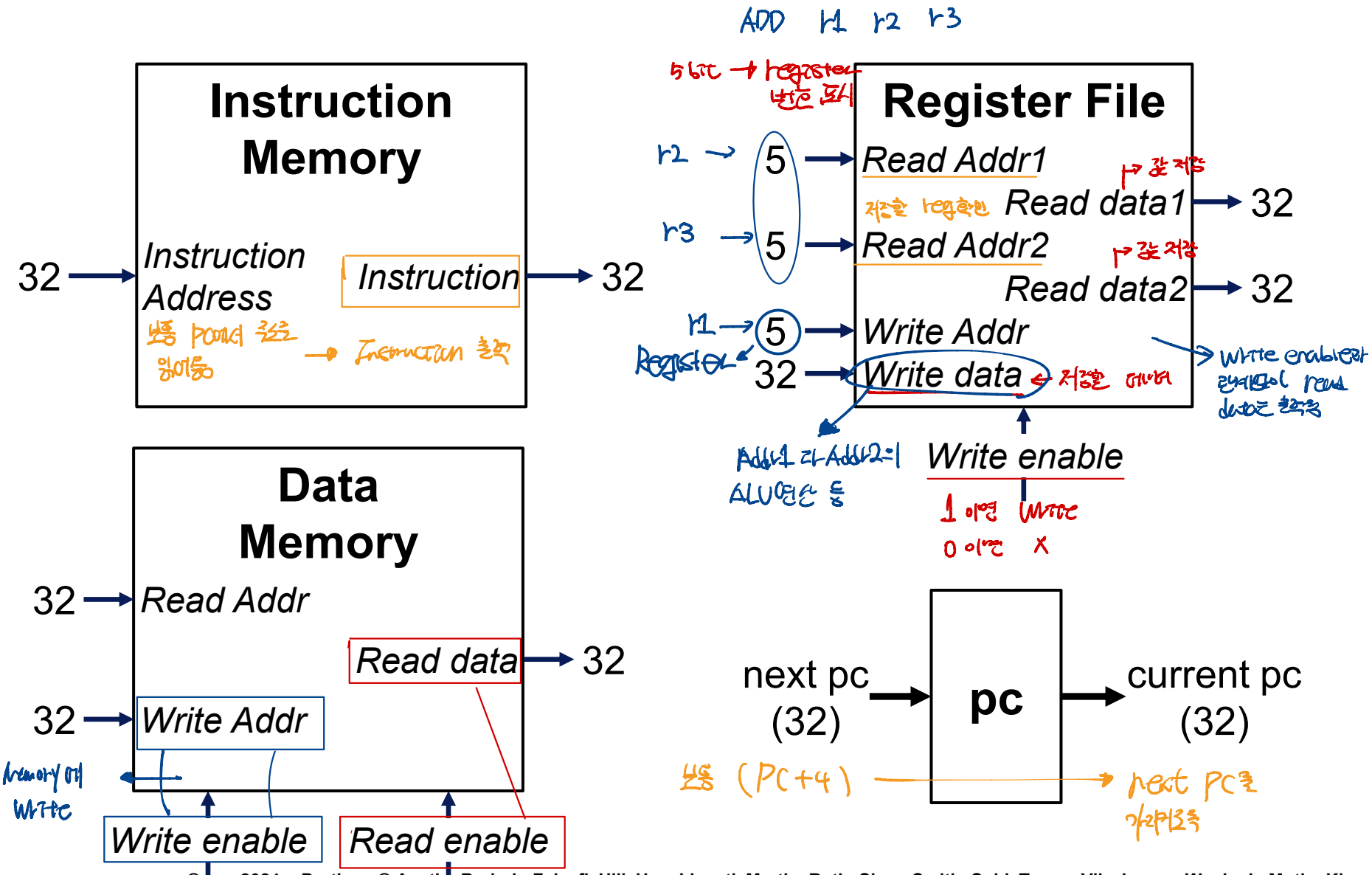
- Simple Decoding
  - 4 bytes per instruction, regardless of format (fixed size)
  - Must be 4-byte aligned (2 LSB of PC must be 2b'00)
  - Format and fields readily extractable

Recall: Instruction processing FSM



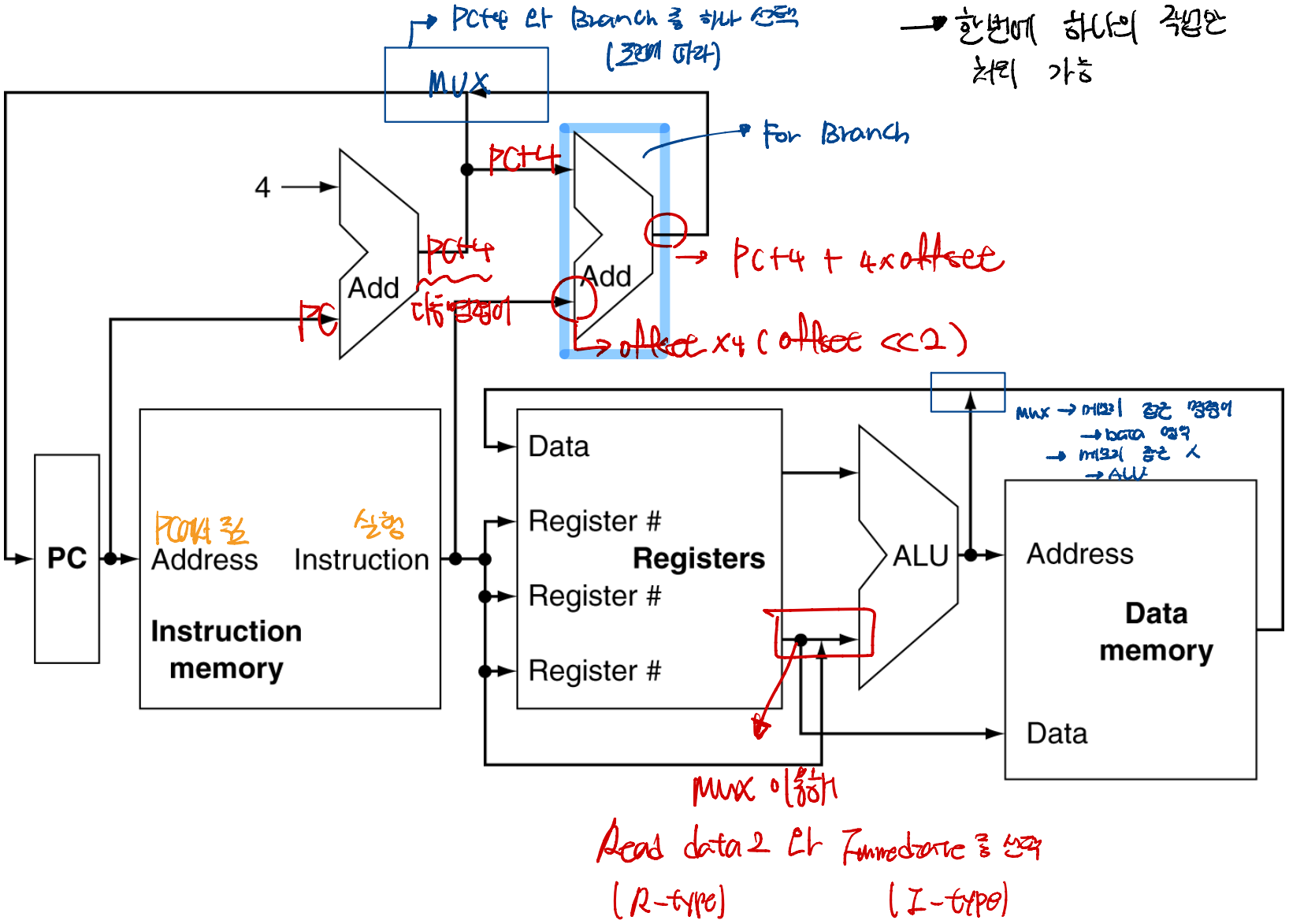
- An ISA describes an abstract finite-state machine (FSM)
  - State = program visible state
  - Next-state logic = instruction execution
- Nice ISAs have atomic instruction semantics
  - One state transition per instruction in abstract FSM
- **Implementation of FSMs can vary**

Recall: MIPS architectural state

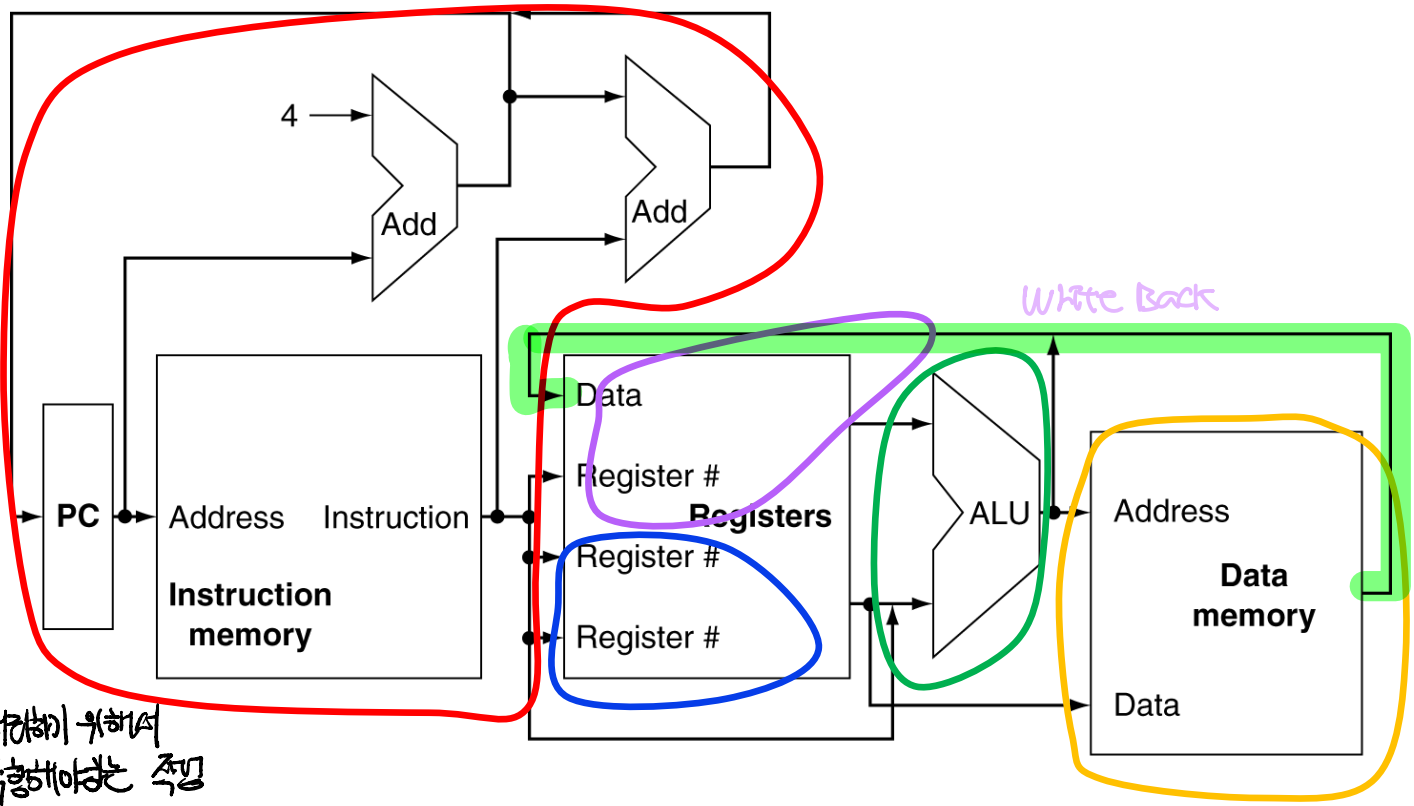


Let’s dive into the implementation details

CPU Overview – Single Core



Five generic steps

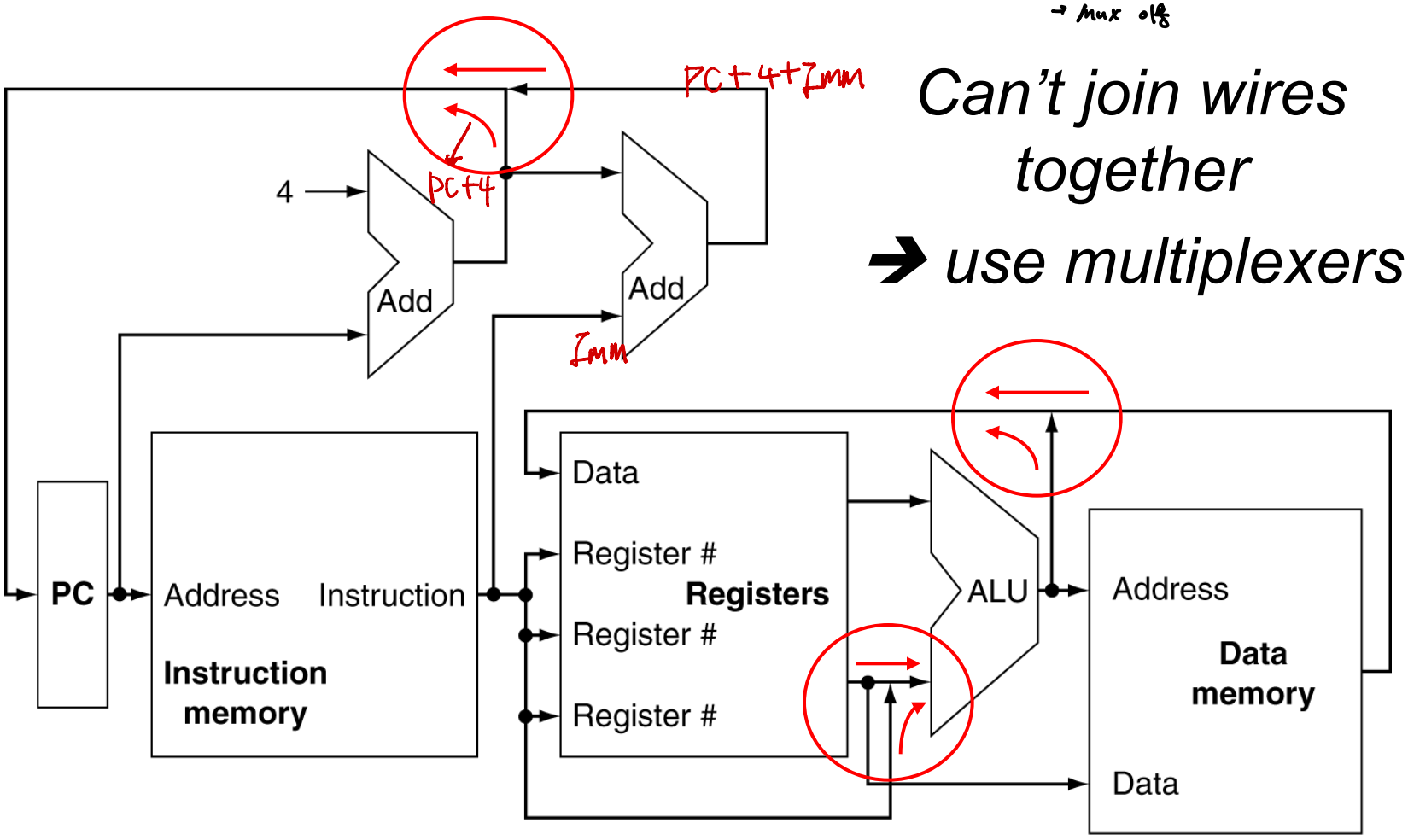


하나의 명령을 처리하기 위해서
하나의 CPU가 수행해야할 작업

- **IF:** Instruction fetch (calculate instruction address & load target instruction)
- **ID:** Instruction decode and fetch operand
- **EX:** ALU/execute
- **MEM:** Memory access (only for load & store data)
- **WB:** Write-back

Using multiplexers

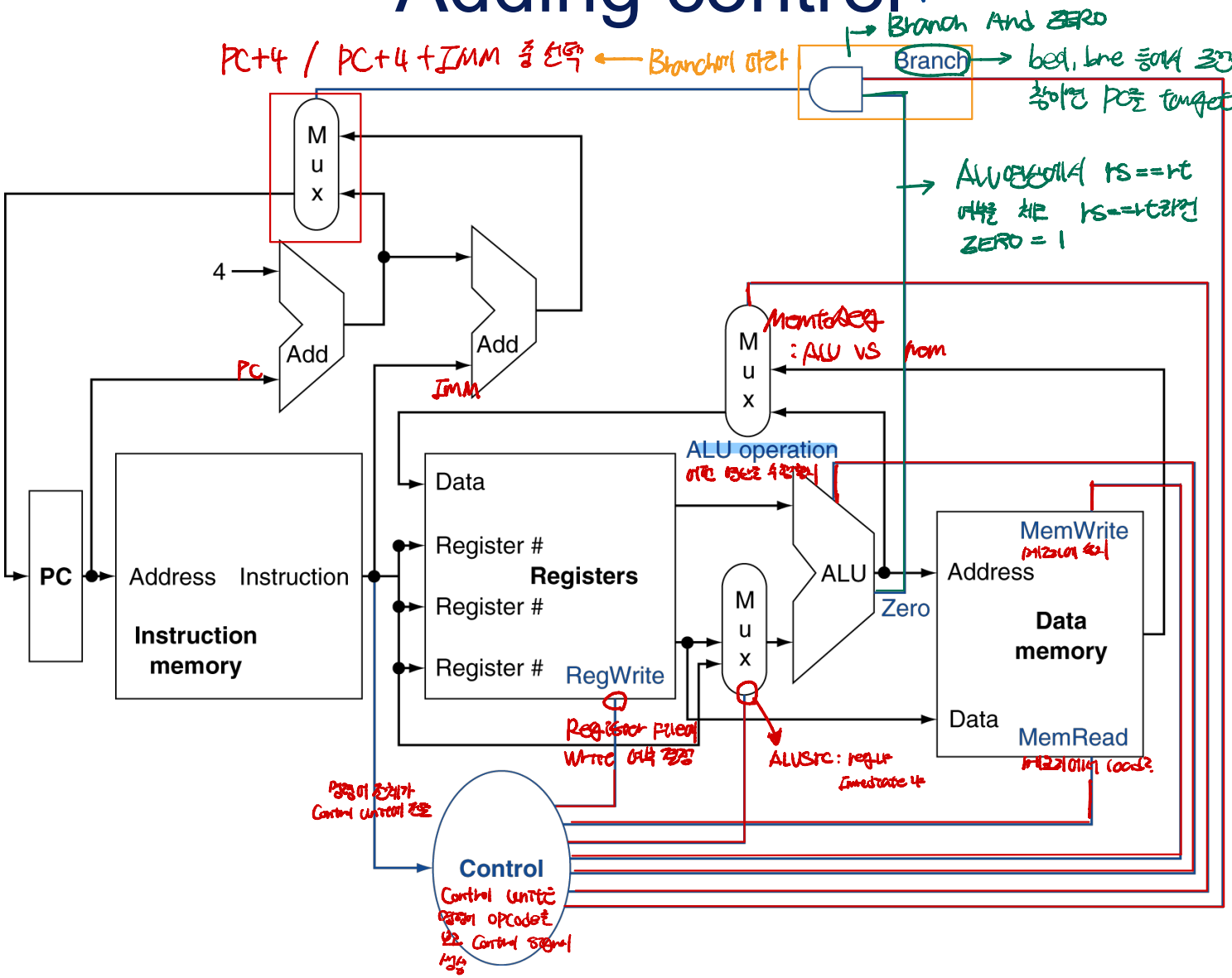
Wire : 값을 저장하지 않고 전달만
→ 두 개 이상의 source가 wire에 값 assign 하려고 하면 안된다.
→ mux 이용



Can't join wires together

➔ *use multiplexers*

Adding control



Recall: General instruction classes

- **Arithmetic and logical operations** (e.g., add, sub, and, or)
  1) Load operands from specified locations
  2) Compute a result as a function of the operands
  3) Store result to a specified location
  4) Update PC to the next sequential instruction
- **Data movement operations** (e.g., load, store)
  1) Fetch operands from specified locations
  2) Store operand values to specified locations
  3) Update PC to the next sequential instruction
- **Control flow operations** (e.g., branch)
  1) Fetch operands from specified locations
  2) Compute a branch condition and a target address
  3) If "branch condition is true" then PC ← target address
     else PC ← next seq. instruction

Recall: General instruction classes

- **Arithmetic and logical operations** (e.g., add, sub, and, or)
  1) Load operands from specified locations
  2) Compute a result as a function of the operands
  3) Store result to a specified location
  4) Update PC to the next sequential instruction

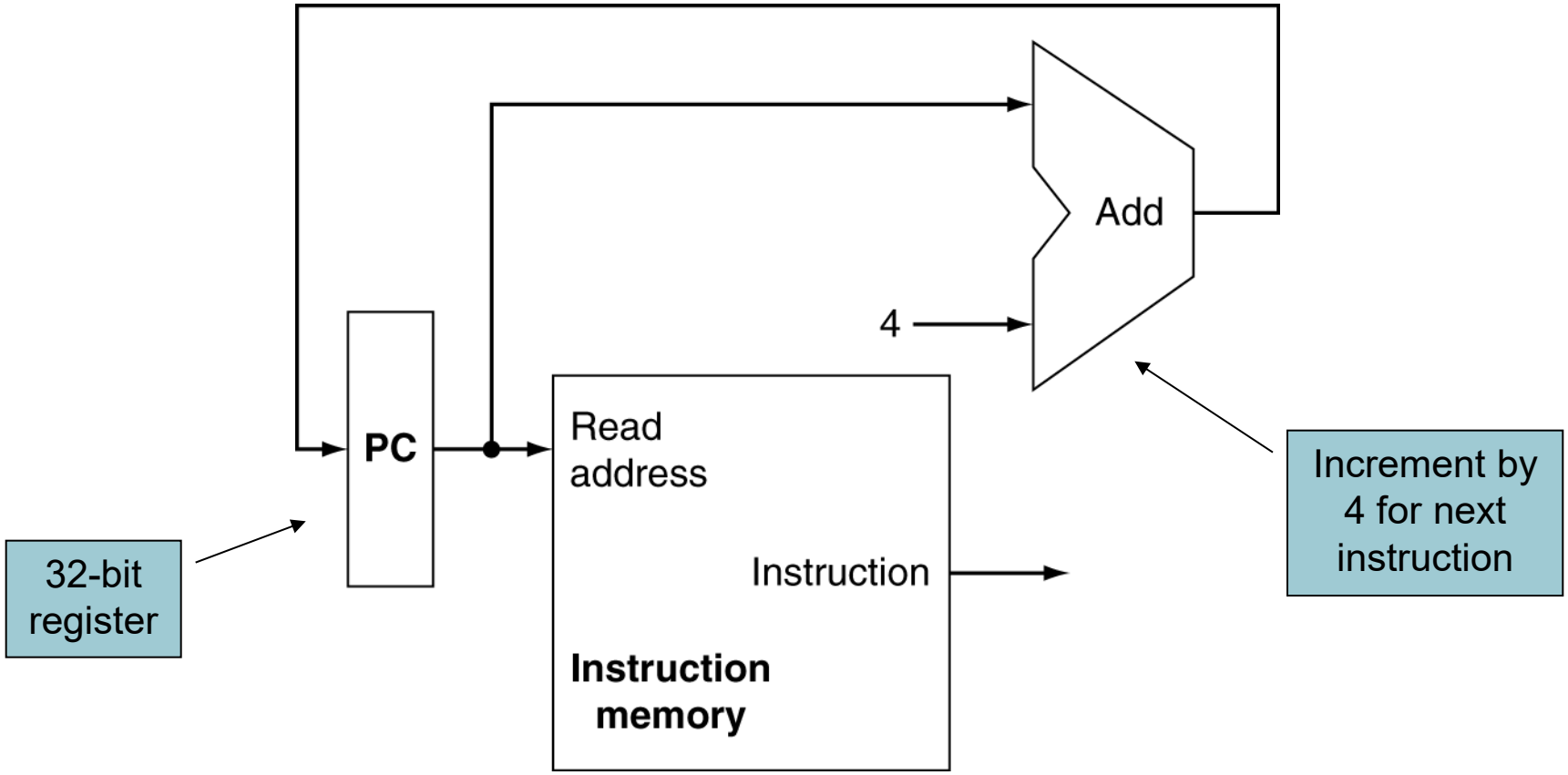
PC값은 언제 Update??
가변 ←

- **Data movement operations** (e.g., load, store)
  1) Fetch operands from specified locations
  2) Store operand values to specified locations
  3) Update PC to the next sequential instruction

- **Control flow operations** (e.g., branch)
  1) Fetch operands from specified locations
  2) Compute a branch condition and a target address
  3) If "branch condition is true" then PC ← target address
     else PC ← next seq. instruction

Instruction fetch unit

- We should fetch the next instruction in the `PC + 4`



Recall: General instruction classes

- **Arithmetic and logical operations** (e.g., add, sub, and, or)
  1) Load operands from specified locations
  2) Compute a result as a function of the operands
  3) Store result to a specified location
  ~~4) Update PC to the next sequential instruction~~
- **Data movement operations** (e.g., load, store)
  1) Fetch operands from specified locations
  2) Store operand values to specified locations
  ~~3) Update PC to the next sequential instruction~~
- **Control flow operations** (e.g., branch)
  1) Fetch operands from specified locations
  2) Compute a branch condition and a target address
  3) If "branch condition is true" then PC ← target address
     else PC ← next seq. instruction

Recall: ALU instructions

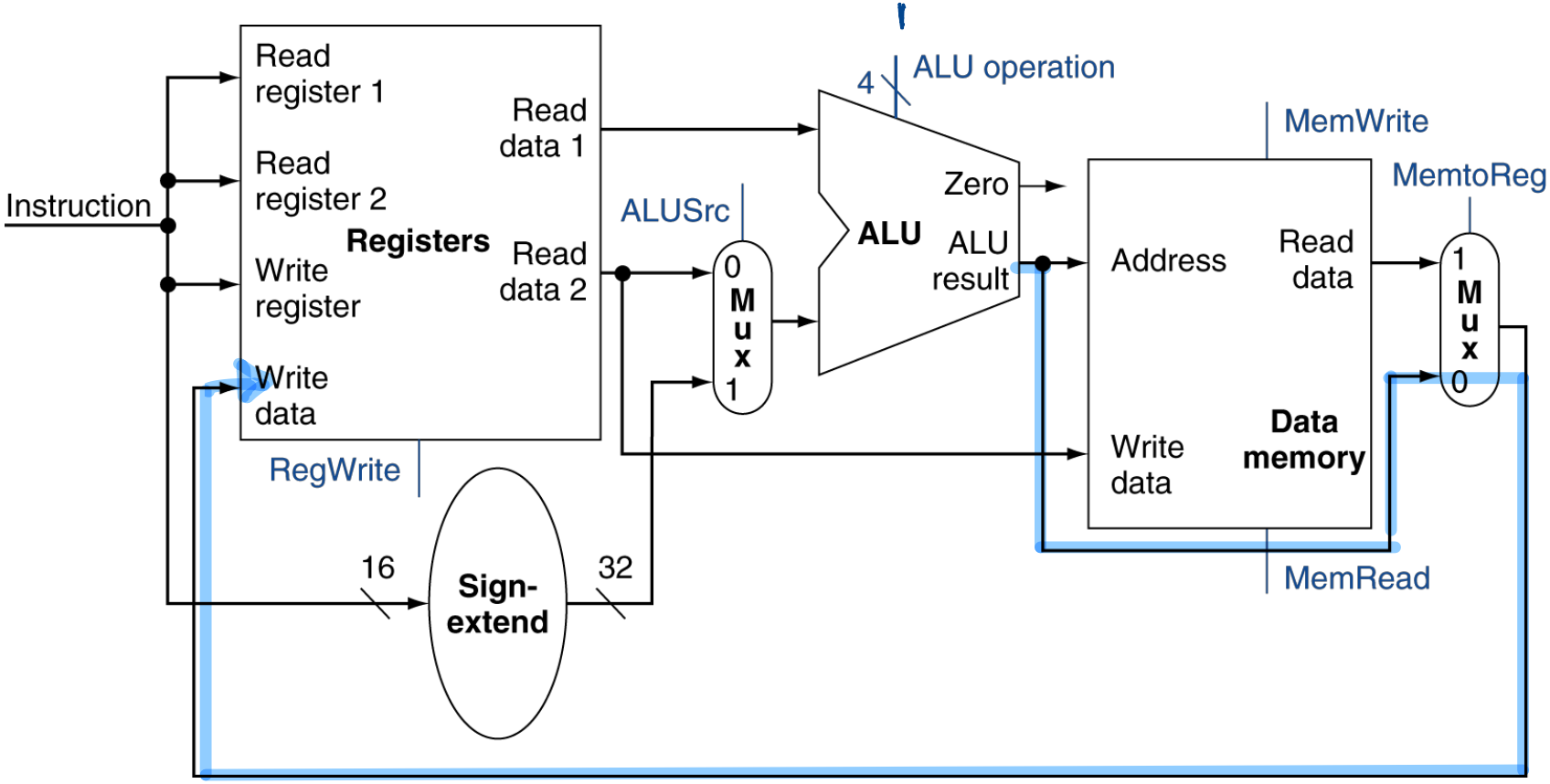
- R-type ALU instructions
  - `add/sub/… rd rs rt`
- I-type ALU instructions
  - `addi/subi/… rt rs 100`
- Semantics
  - `Given an instruction → add rd rs rt`
    - `GPR[rd] = GPR[rs] + GPR[rt]`
    - `PC = PC + 4`
  - `Given an instruction → addi rt rs 100`
    - `GPR[rt] = GPR[rs] + sign-extend(100)`
    - `PC = PC + 4`

Recall: Memory instructions

- I-type ALU instructions
  - `load rt rs offset`
  - `store rt rs offset`

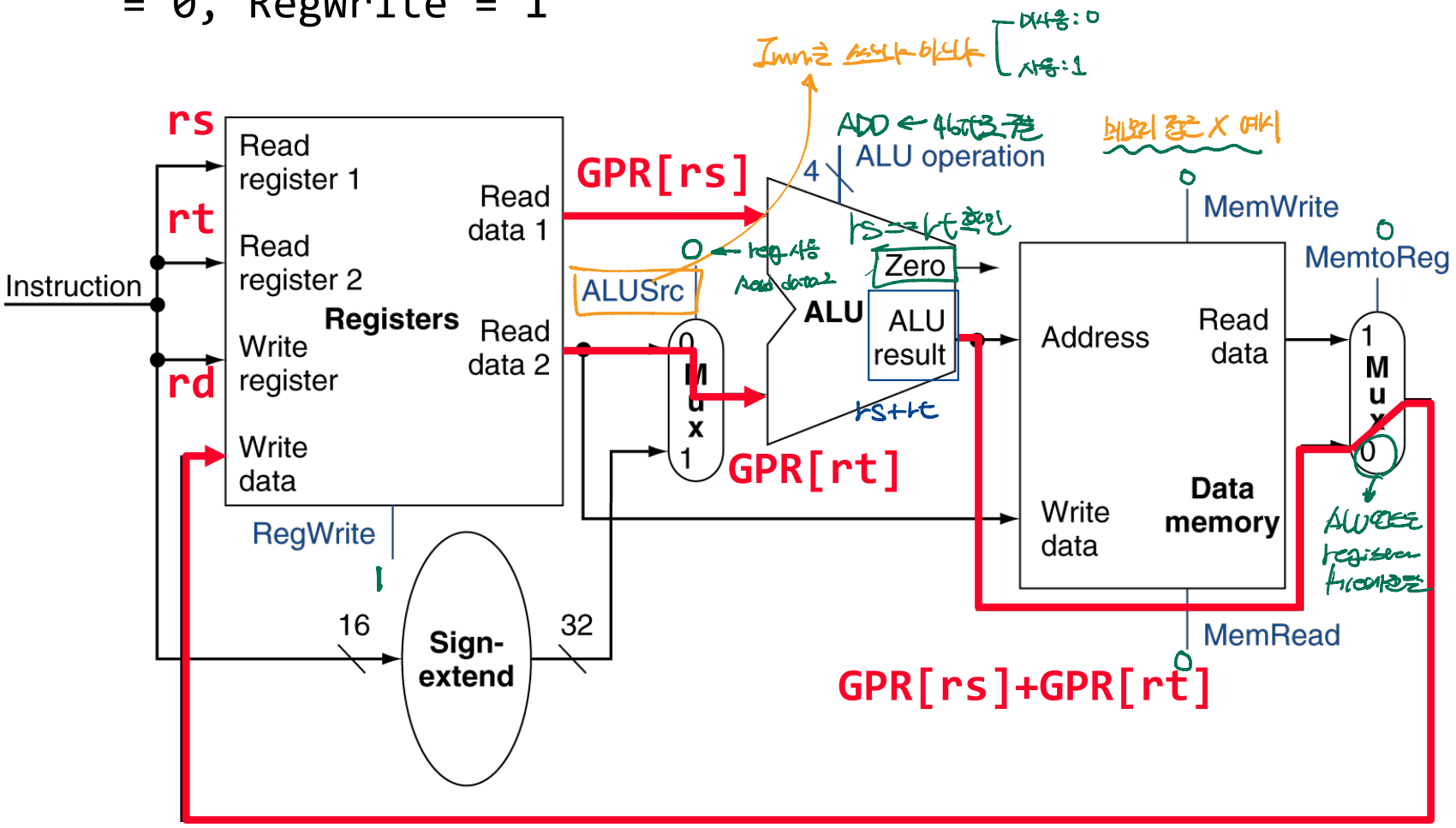
- Semantics (using `rs, rt, and offset`)
  - `Given an instruction → load/store rt rs offset`
    - `GPR[rt] = MEM[GPR[rs] + sign-extend(offset)] // load`
    - `MEM[GPR[rs] + sign-extend(offset)]= GPR[rt] // store`
    - `PC = PC + 4 (use the next instruction)`

Datapath for ALU and memory operation



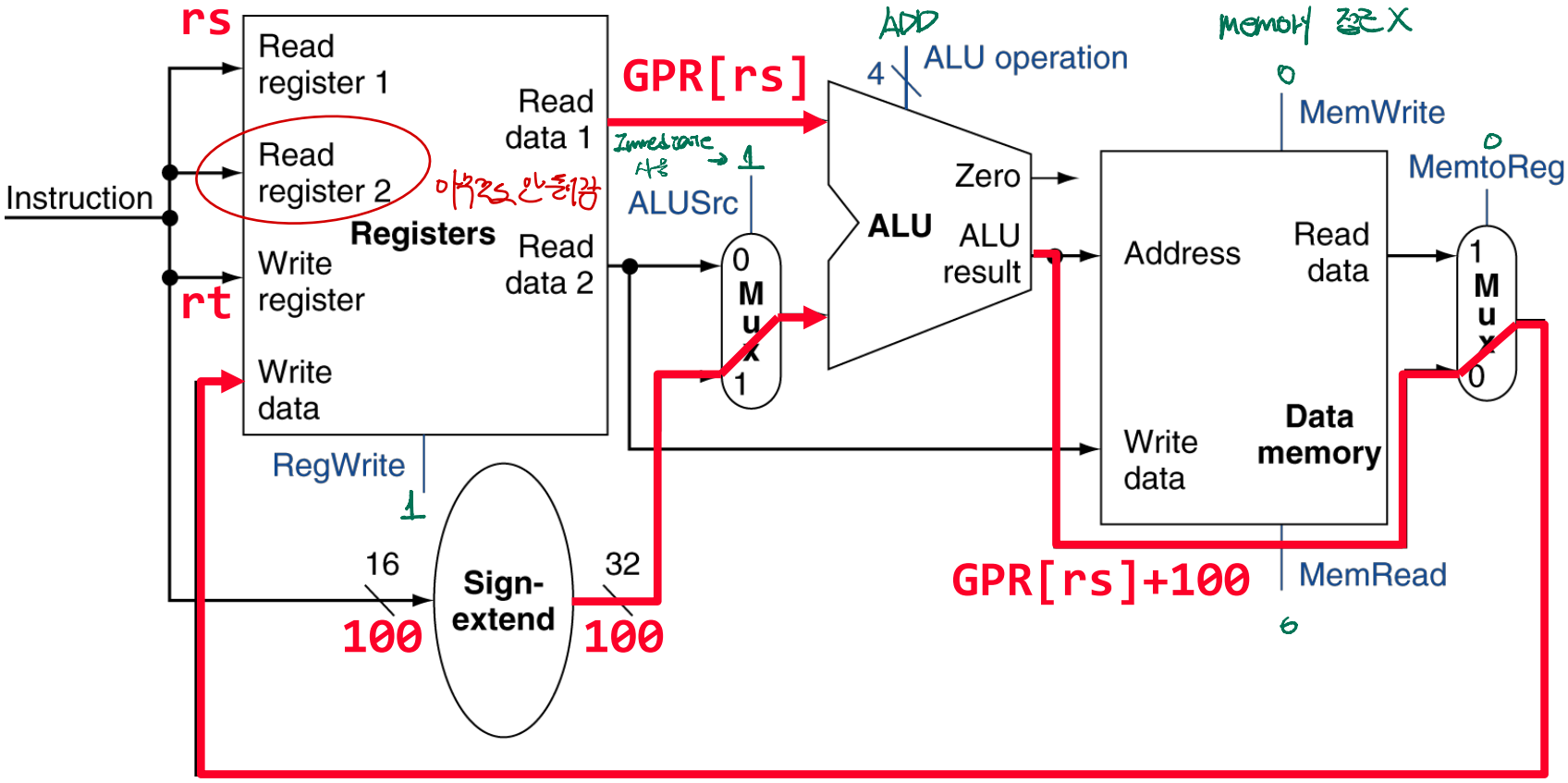
ALU / Memory operation datapath

- Datapath for an R-type ALU operation (`add rd rs rt`)
  - `ALUSrc = 0, ALU op = ADD, MemRead/Write = 0, MemtoReg = 0, RegWrite = 1`



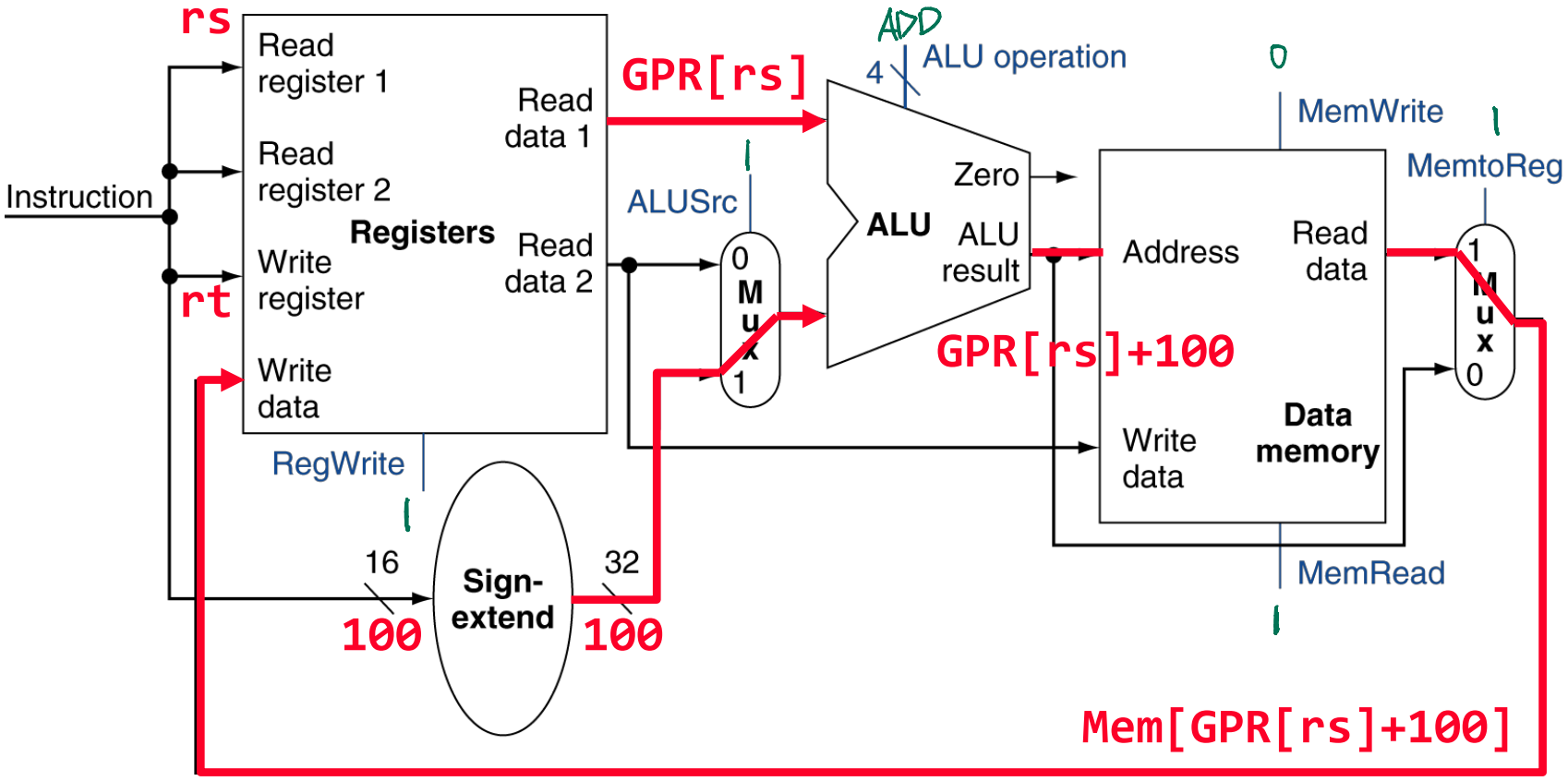
ALU / Memory operation datapath

- Datapath for an I-type ALU operation (`addi rt rs 100`)
  - `ALUSrc = 1, ALU op = ADD, MemRead/Write = 0, MemtoReg = 0, RegWrite = 1`



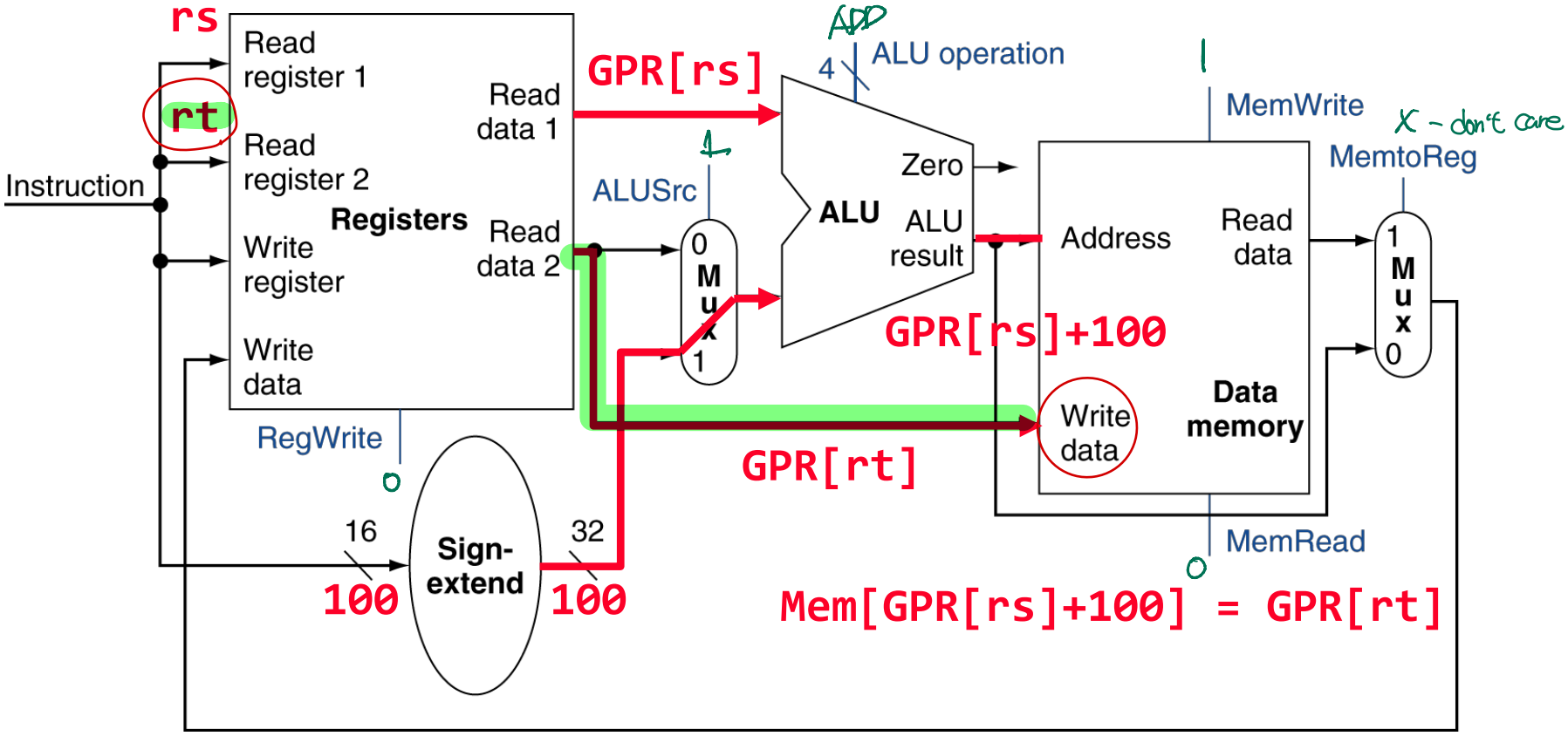
ALU / Memory operation datapath

- Datapath for an I-type Mem operation (`load rt rs 100`)
  - `ALUSrc = 1, ALU op = ADD, MemRead = 1, MemWrite = 0, MemtoReg = 0, RegWrite = 1`



ALU / Memory operation datapath

- Datapath for an I-type Mem operation (`store rt rs 100`)
  - `ALUSrc = 1, ALU op = ADD, MemRead = 0, MemWrite = 1, MemtoReg = X, RegWrite = 0`



Recall: General instruction classes

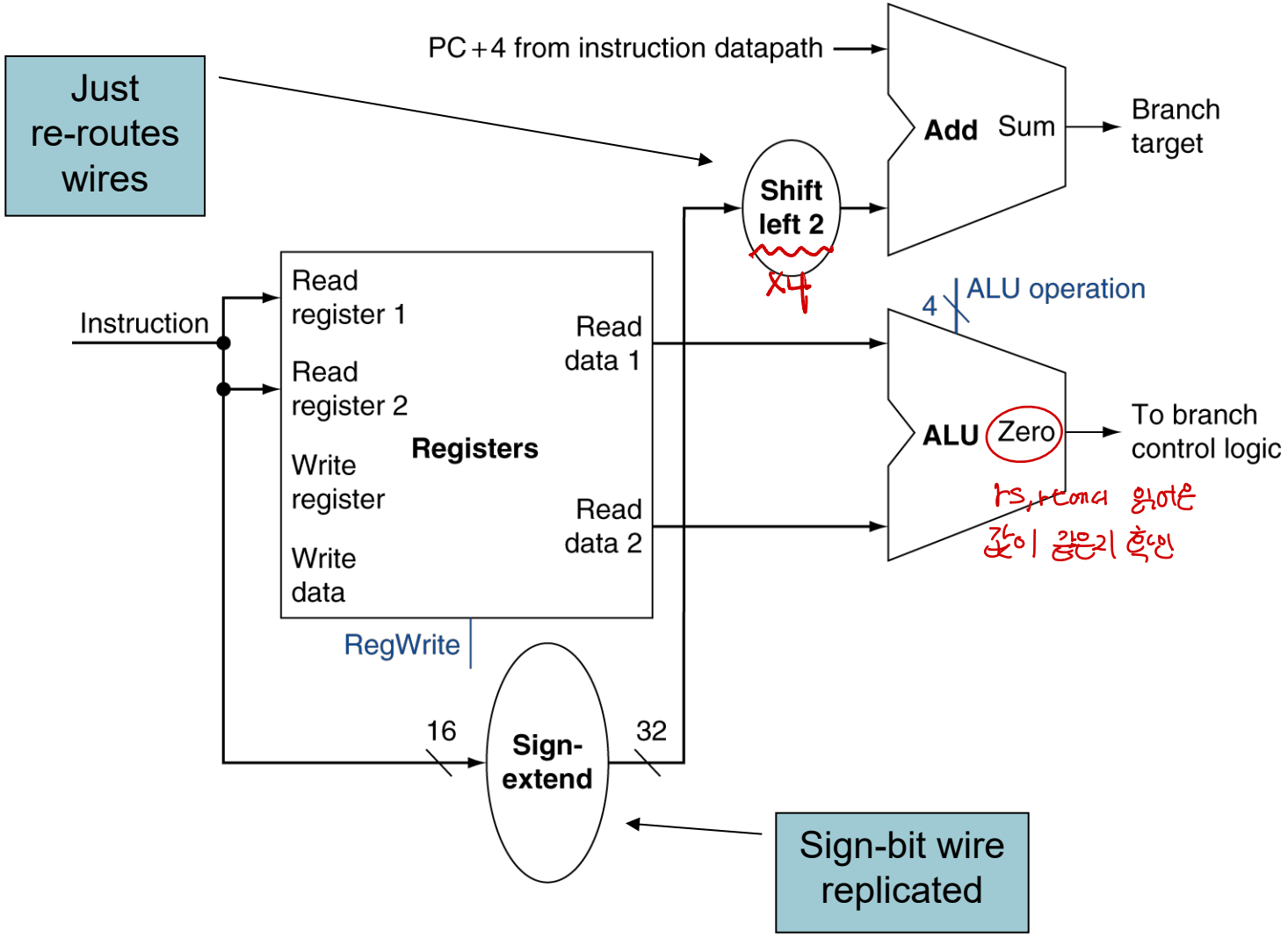
- **Arithmetic and logical operations** (e.g., add, sub, and, or)
  - ~~1) Load operands from specified locations~~
  - ~~2) Compute a result as a function of the operands~~
  - ~~3) Store result to a specified location~~
  - ~~4) Update PC to the next sequential instruction~~
- **Data movement operations** (e.g., load, store)
  - ~~1) Fetch operands from specified locations~~
  - ~~2) Store operand values to specified locations~~
  - ~~3) Update PC to the next sequential instruction~~
- **Control flow operations** (e.g., branch)
  - 1) Fetch operands from specified locations
  - 2) Compute a branch condition and a target address
  - 3) If "branch condition is true" then PC ← target address
    else PC ← next seq. instruction

Recall: Conditional branch instructions

- I-type control instructions
  - `beq/bne rs rt label  // Conditional jump`

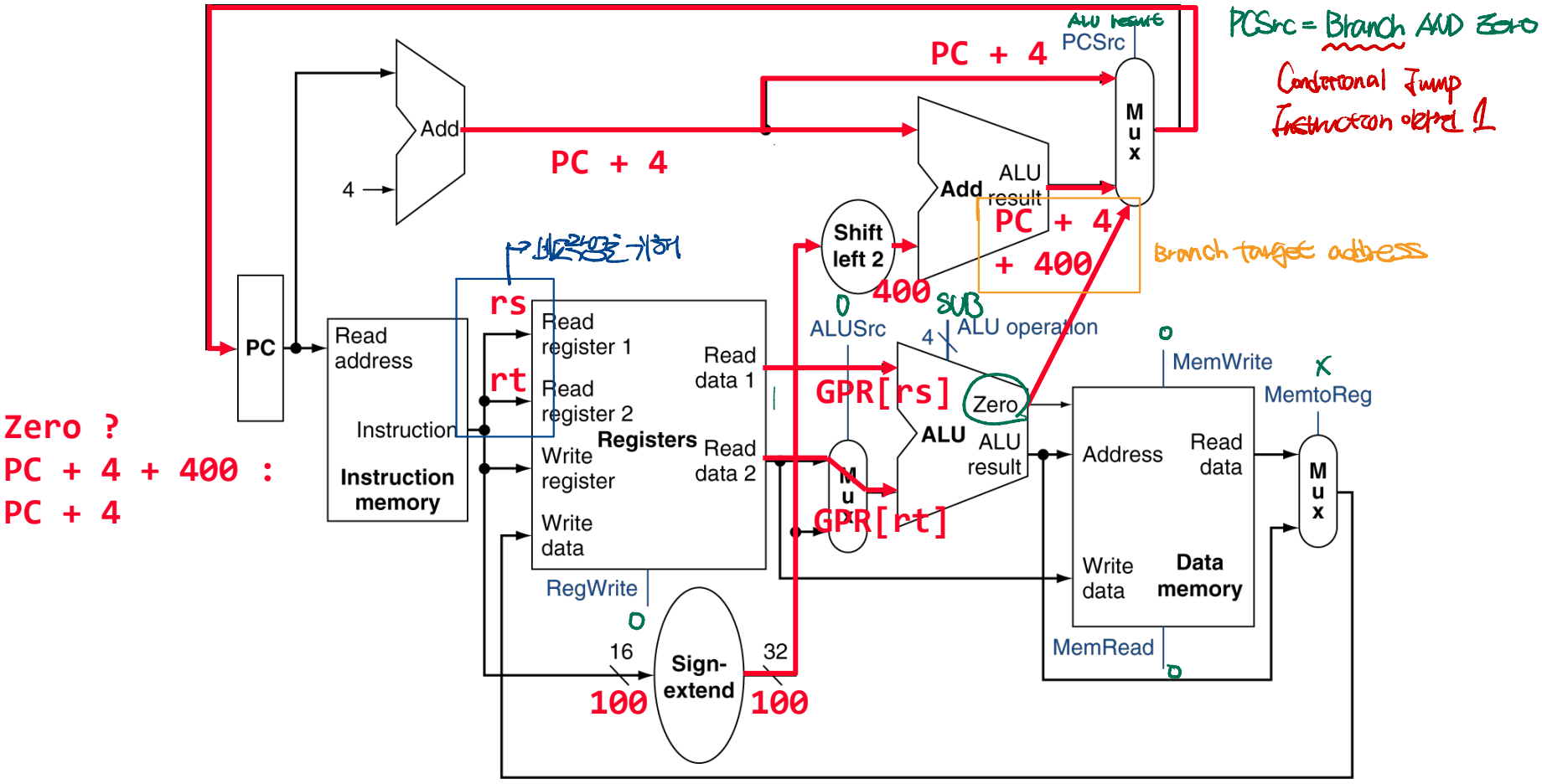
- Semantics
  - `beq rs rt label`
    - `target = (PC + 4) + sign-extend(label) x 4`
    - `if (GPR[rs] == GPR[rt]) PC = target else PC = PC + 4`

Instruction fetch for branch instructions



Add support for control instructions

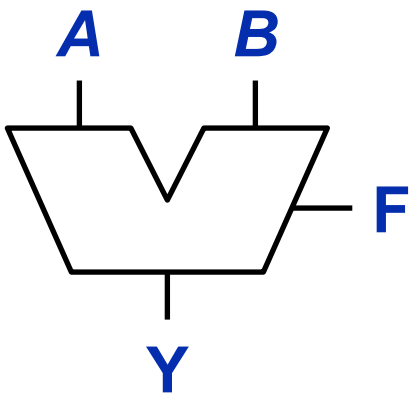
- Datapath for an I-type Mem operation (`beq rs rt 100`)
  - `ALUSrc = 0, ALU op = SUB, MemRead/Write = 0, MemtoReg = X, RegWrite = 0, PCSrc = ALU Result`



Let’s discuss the controller
(Skip the J-type instructions for now …)

ALU control in MIPS

- ALU is used for
  - `lw/sw → ALU Operation = ADD`
  - `beq → EQ`
  - `R-type → ALU Operation depends on the funct field`



F[4:0]	Y
0000	A AND B
0001	A OR B
0010	A + B
0110	A – B
0111	If (A < B) 1 else 0 (A SLT B)
1100	A NOR B

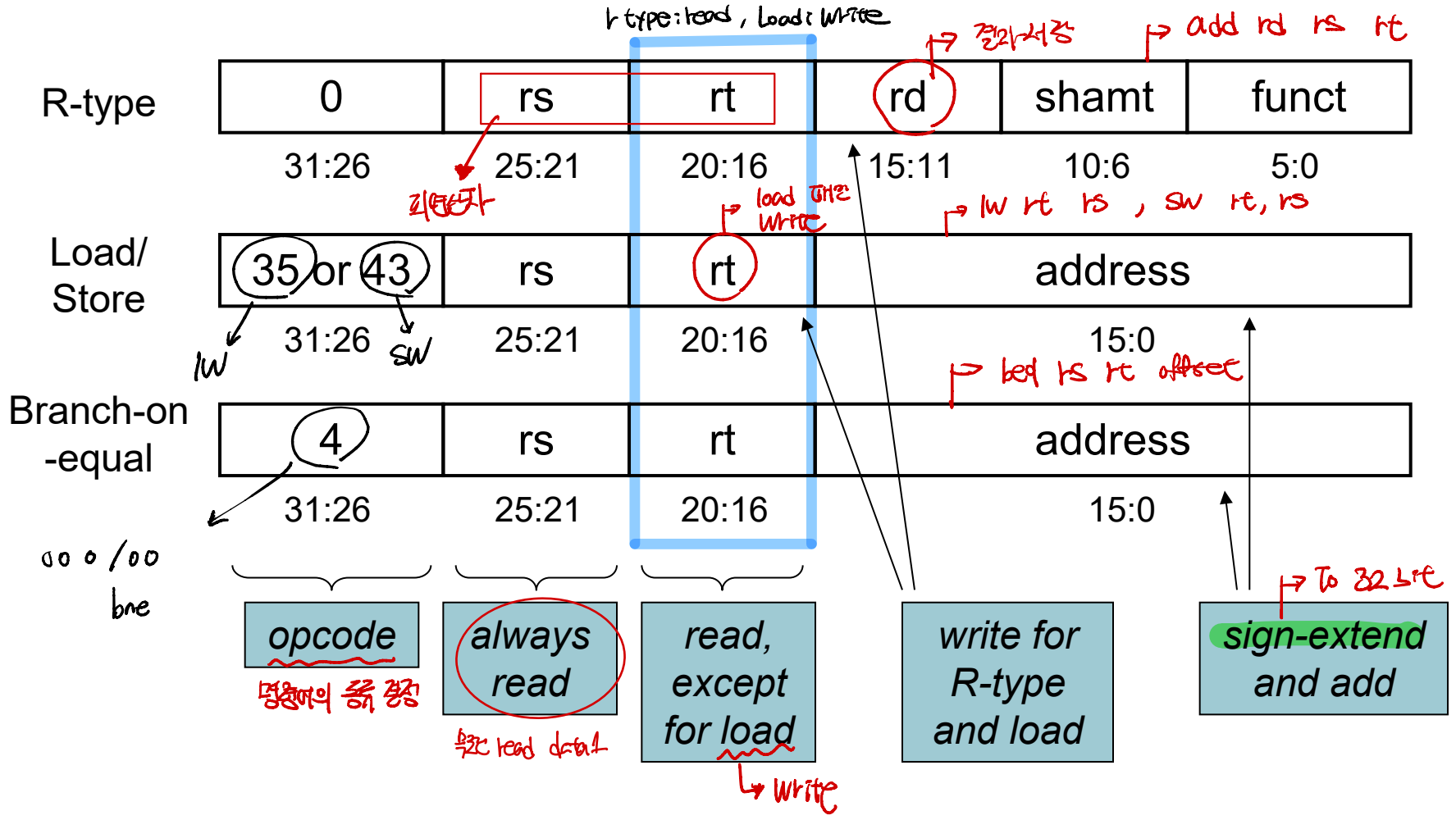
ALU control and instructions

opcode	Operation	funct	ALU function	ALU control
lw	load word	XXXXXX	ADD	0010
sw	store word	XXXXXX	ADD	0010
beq	branch equal	XXXXXX	EQ	0100
R-type	ADD	100000	ADD	0010
	SUB	100010	SUB	0110
	AND	100100	AND	0000
	OR	100101	OR	0001
	SLT	101010	SLT	0111

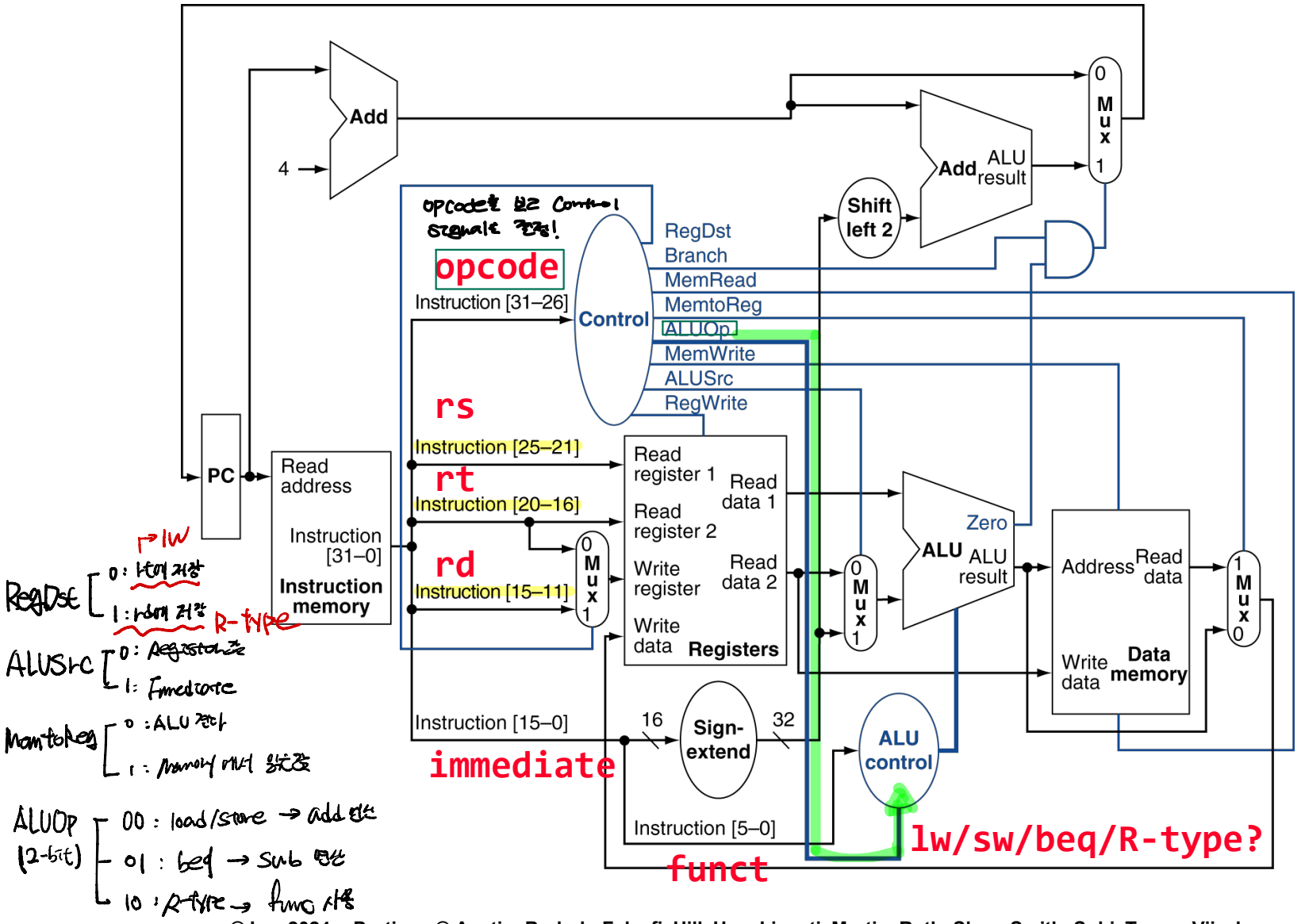
How should we support bne?

Main control unit

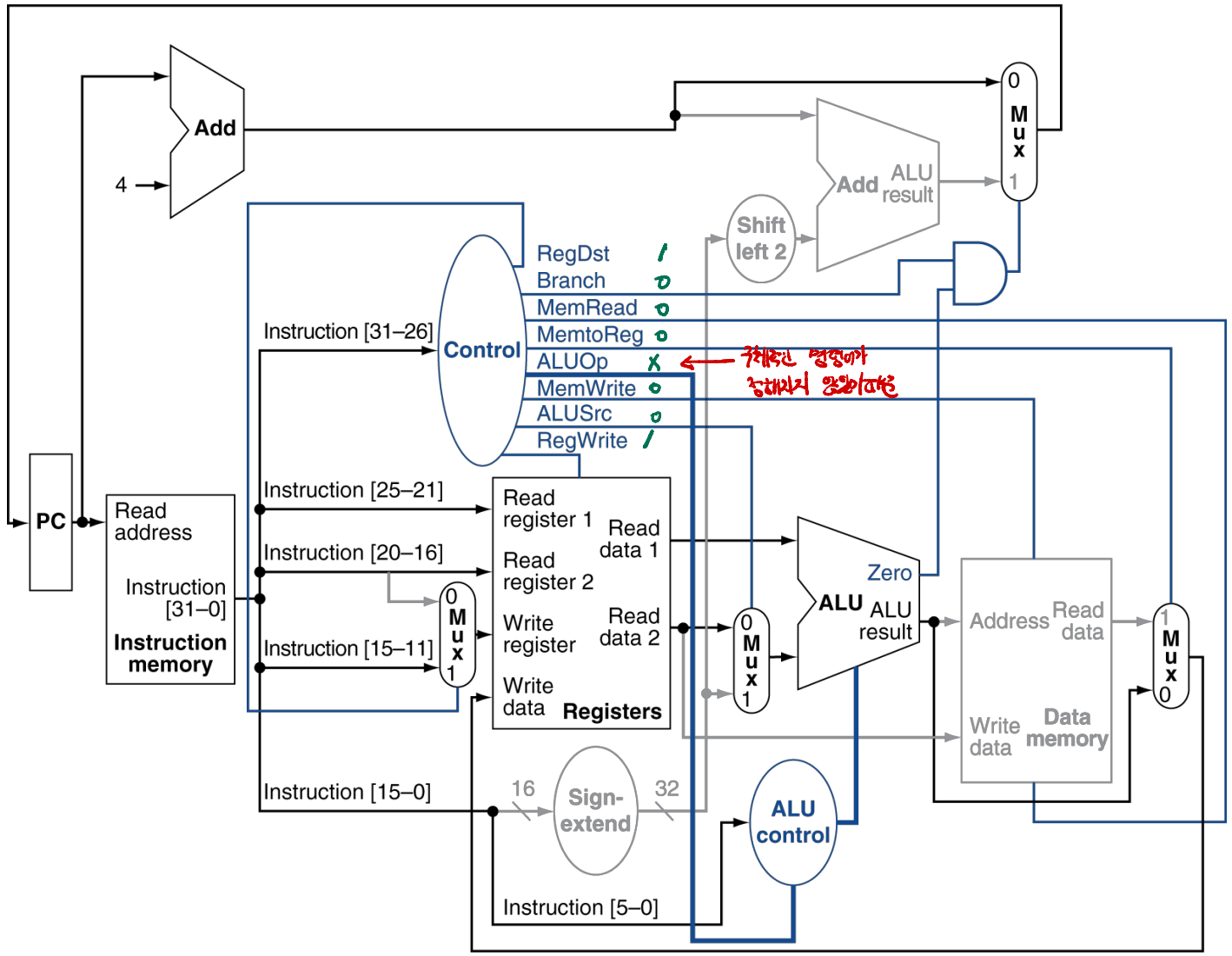
- Control signals derived from instruction



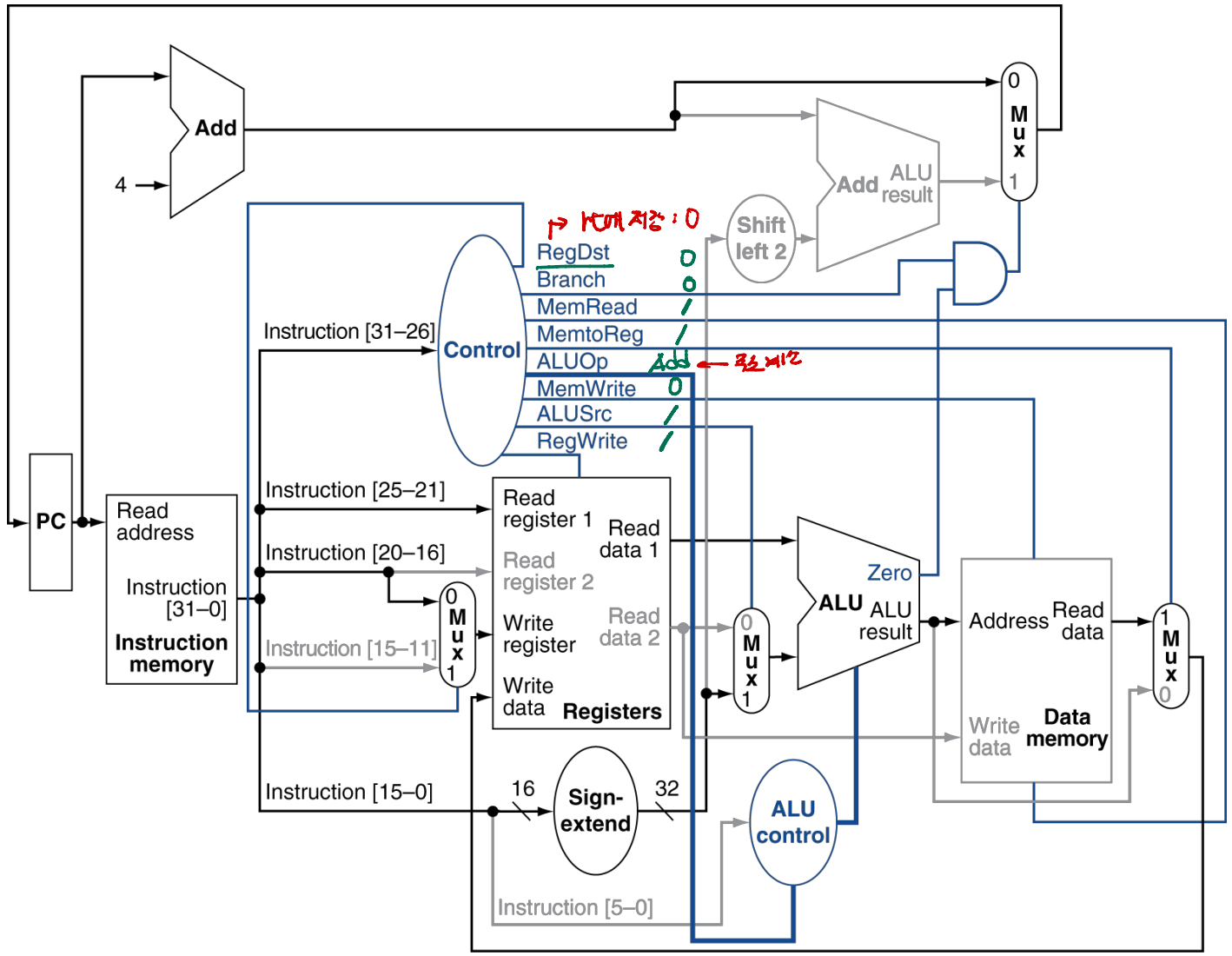
Datapath with control (Decode)



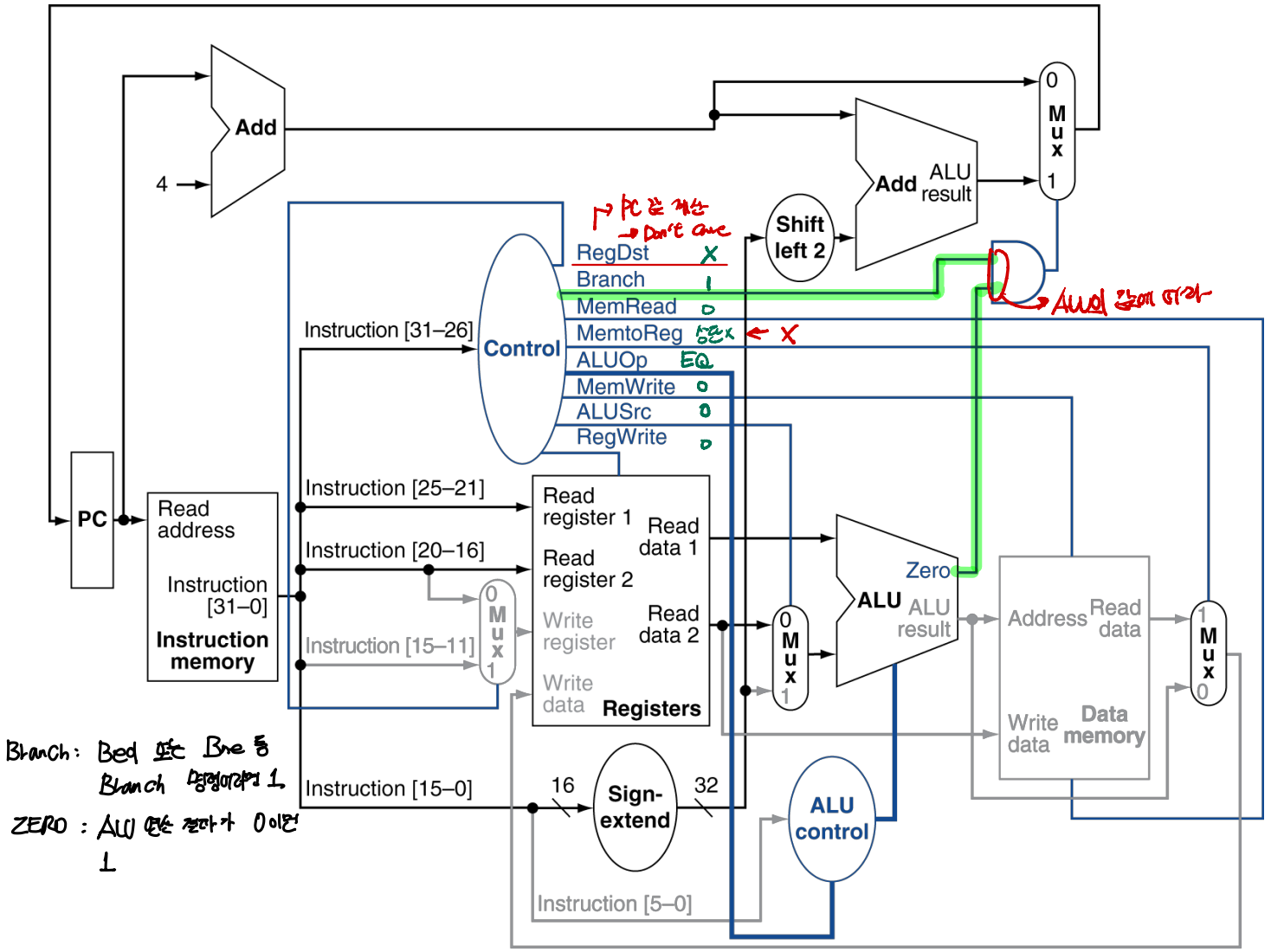
R-Type Instruction



Load Instruction

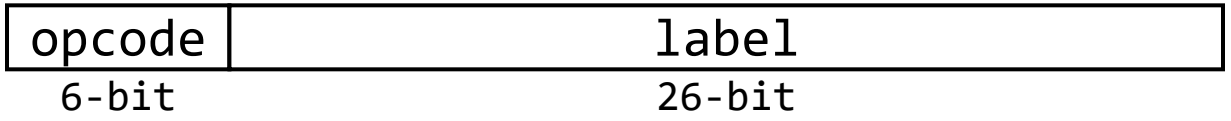


Branch-on-Equal Instruction



Back to the J-type instructions

Recall: J-type Instructions



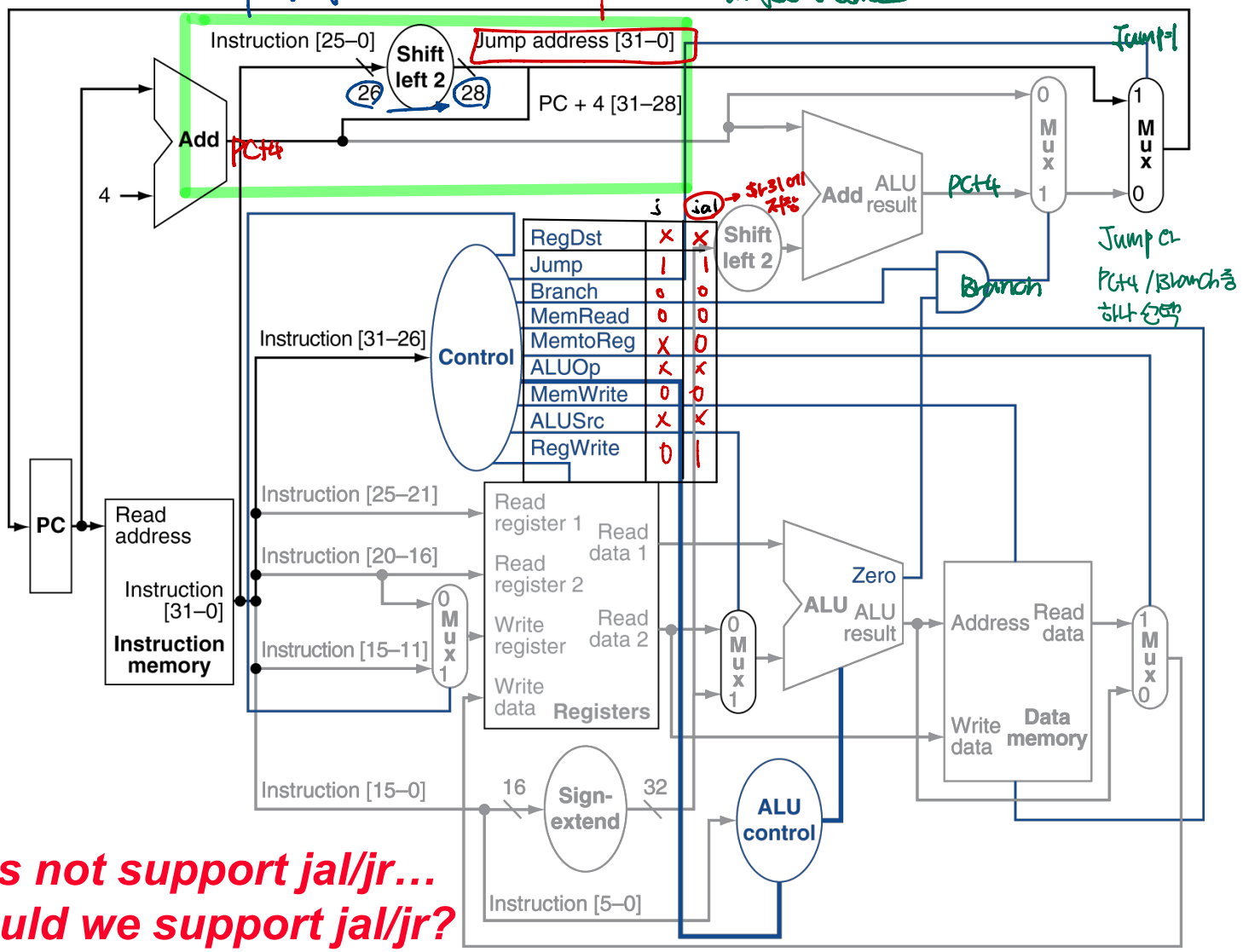
J-type

- opcode: there are eight immediate memory instructions
  - j, jal

현재 PC +4를 $ra에 저장 후 target address로 넘어감.

- Semantics (using label)
  - target = (PC+4)[31:28]x2^{28} $|_{\text{bitwise-or}}$ zero-extend(label) x 4
  - // use the first four bits of PC and append label x 4
  - Given an instruction → j label
    - PC = target
  - Given an instruction → jal label
    - GPR[ra] = PC + 4
      - save the next PC to ra (a dedicated register)
    - PC = target

Jump instructions



This does not support jal/jr…
How should we support jal/jr?

Designing control logic - 1

	When De-asserted	When asserted	Equation
RegDest	GPR write select according to rt, i.e., inst[20:16]	GPR write select according to rd, i.e., inst[15:11]	opcode==0 R-type → RegDest = 1
ALUSrc	2nd ALU input from 2nd GPR read port	2nd ALU input from sign-extended 16-bit immediate	(opcode!=0) && (opcode!=BEQ) && (opcode!=BNE)
MemtoReg	Steer ALU result to GPR write port	steer memory load to GPR wr. port	opcode==LW
RegWrite	GPR write disabled	GPR write enabled	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

Designing control logic - 2

	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port return load value	opcode==LW
MemWrite	Memory write disabled	Memory write enabled	opcode==SW
Jump	According to Branch	next PC is based on 26-bit immediate jump target	(opcode==J) || (opcode==JAL)
Branch	next PC = PC + 4	next PC is based on 16-bit immediate branch target	(opcode==Bxx) && "bcond is satisfied"

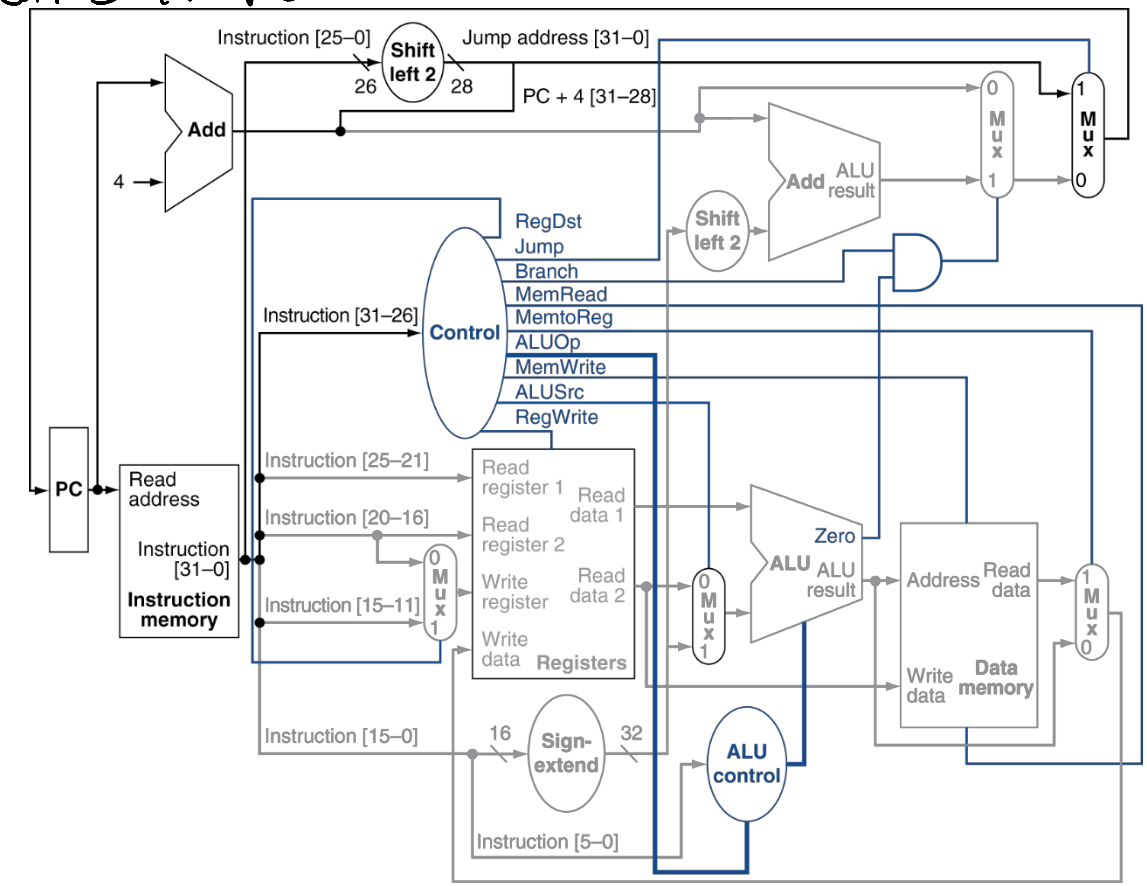
So… this is how you design your computer for a given ISA …

But is this the best design?

Hint for the next lecture

- This is a single-cycle CPU ➔ the clock frequency is set according to the worst-case latency!

→ 모든 각 명령어가 한 cycle에 끝나야해서 가장 느린 명령어에 맞춰 Clock frequency를 세팅해야한다.



Question?

Announcements:

Reading: *Finish reading P&H Ch.4*

Handouts: *None*