Computer Architecture and Organization

INSTRUCTOR: YAN-TSUNG PENG

DEPT. OF COMPUTER SCIENCE, NCCU

NATIONAL CHENGCHI UNIVERSITY

The Structure of a CPU

From a Single-cycle Processor

$$\text{CPU Time} = \frac{\text{Seconds}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

IC

CPI

Clock cycle time

CPU performance factors

- Instruction count
  - Determined by ISA and compiler
- CPI and Cycle time
  - Determined by CPU hardware

Introduction

- We will be examining an implementation that includes a subset of the core MIPS instructions:
  - Memory-reference instructions: lw, sw
  - Arithmetic-logical instructions: add, sub, AND, OR, slt
  - Branch instructions: beq, j
- These will illustrate the key principles used in creating a datapath and designing the control.

Instruction Execution

- For every instruction, the first two steps are identical

  1. PC → instruction memory, fetch instruction

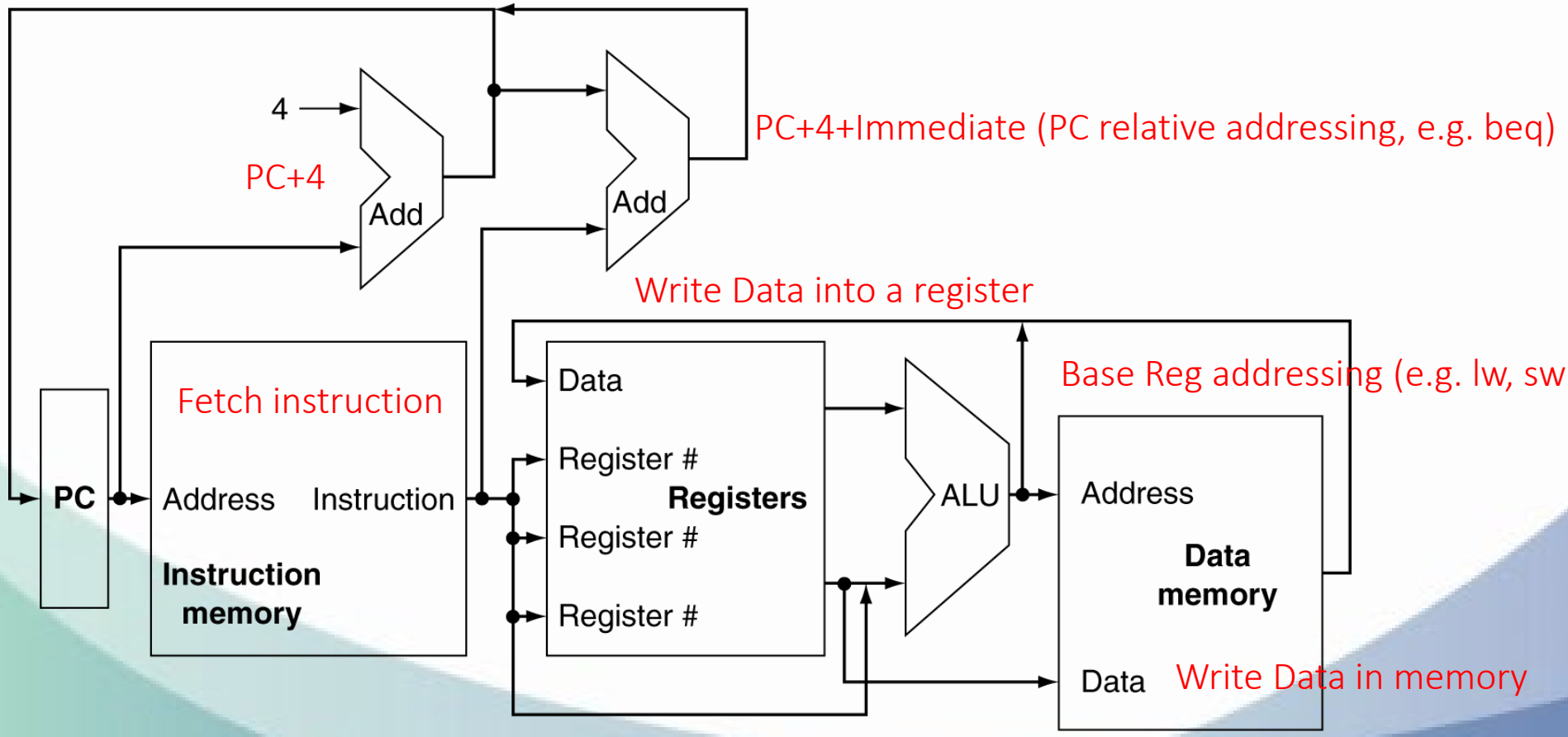
  2. From the instruction: Register numbers → register file, read registers

- Depending 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

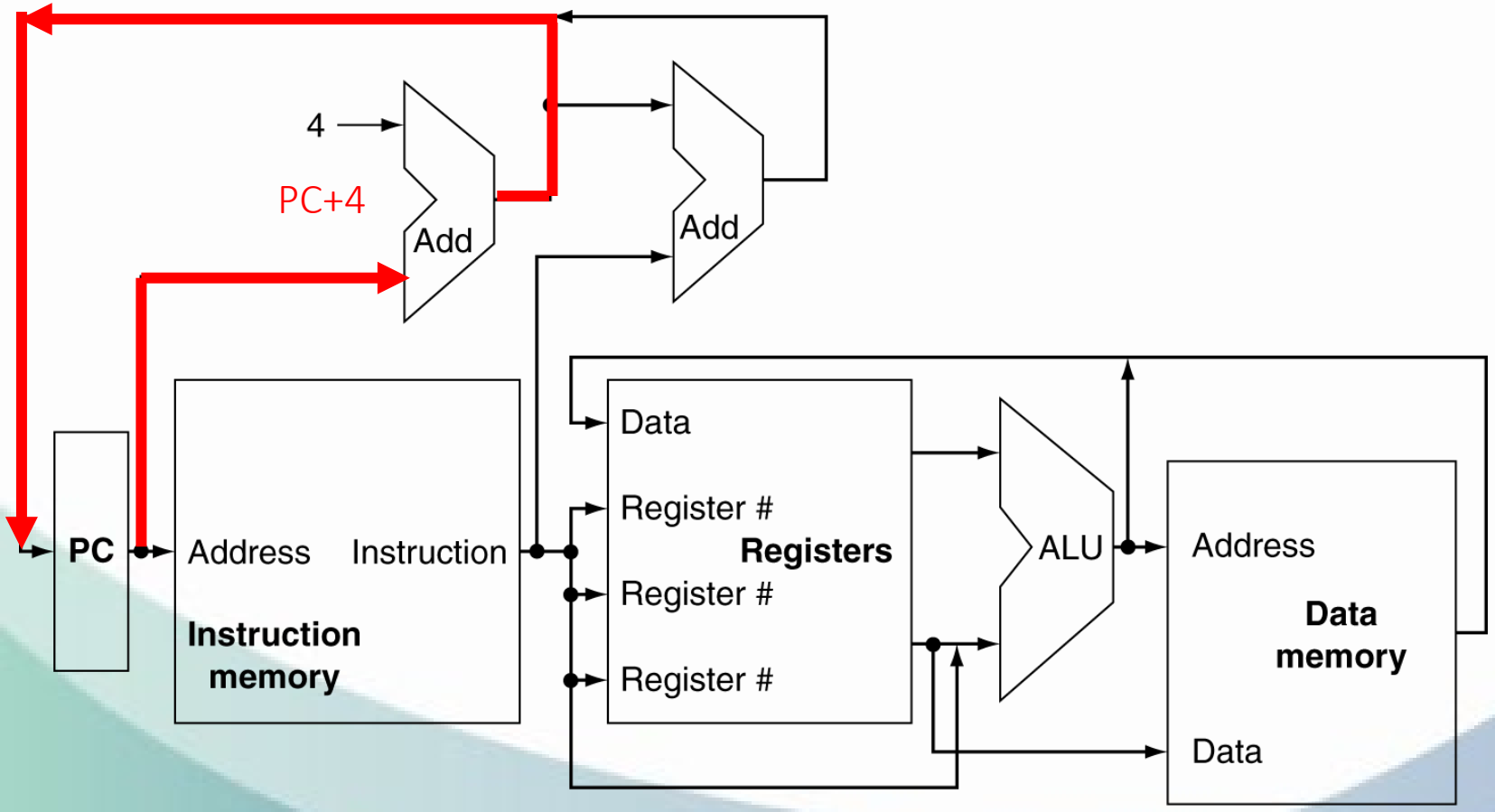
Instruction Execution

- General steps (not all the steps needed for every instruction):
  1. Start with fetching the instruction from memory
  2. Read registers (Register numbers)
  3. ALU
  4. Access data memory
  5. Write registers (PC←PC+4)

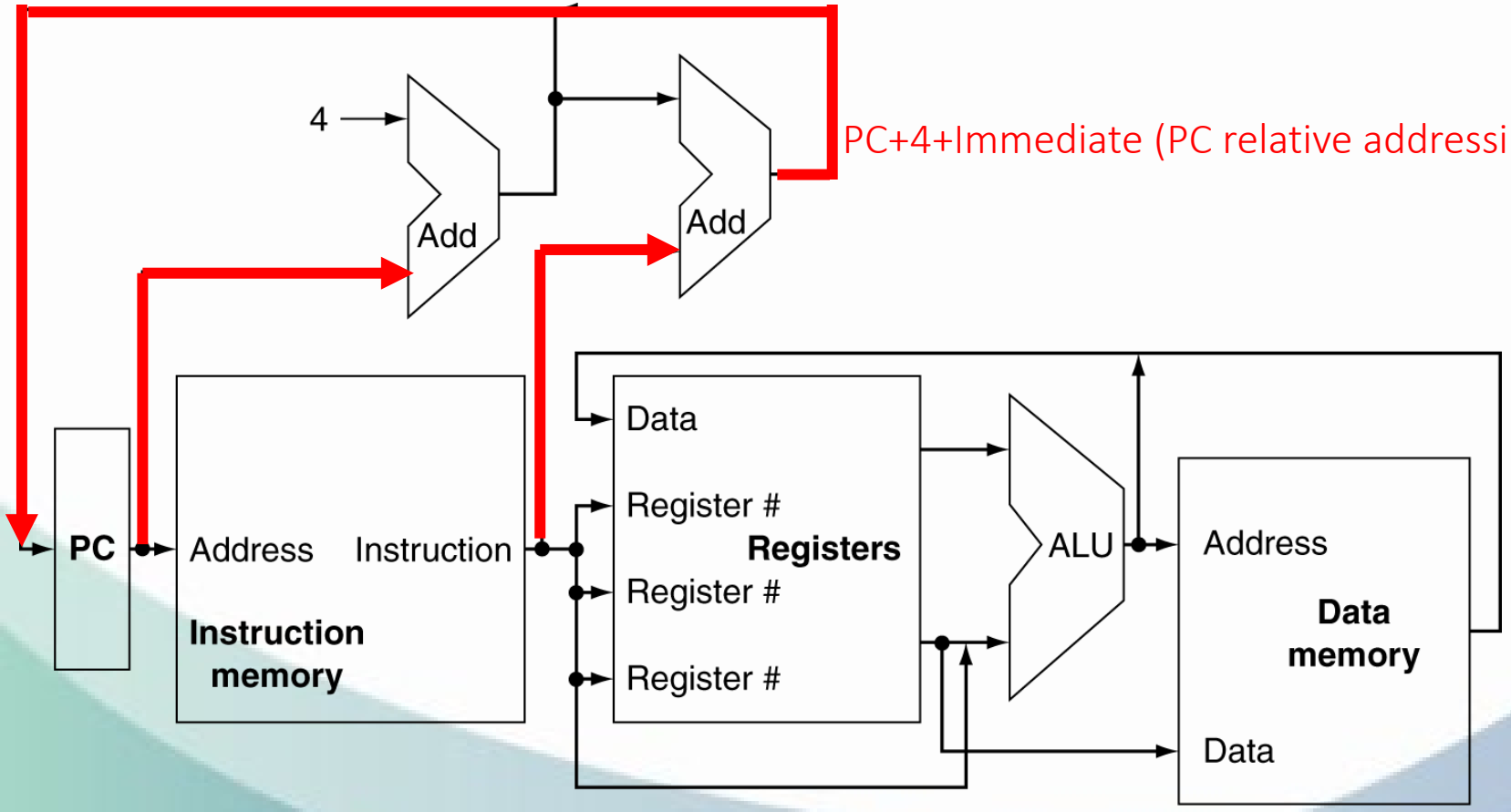
Processor Overview



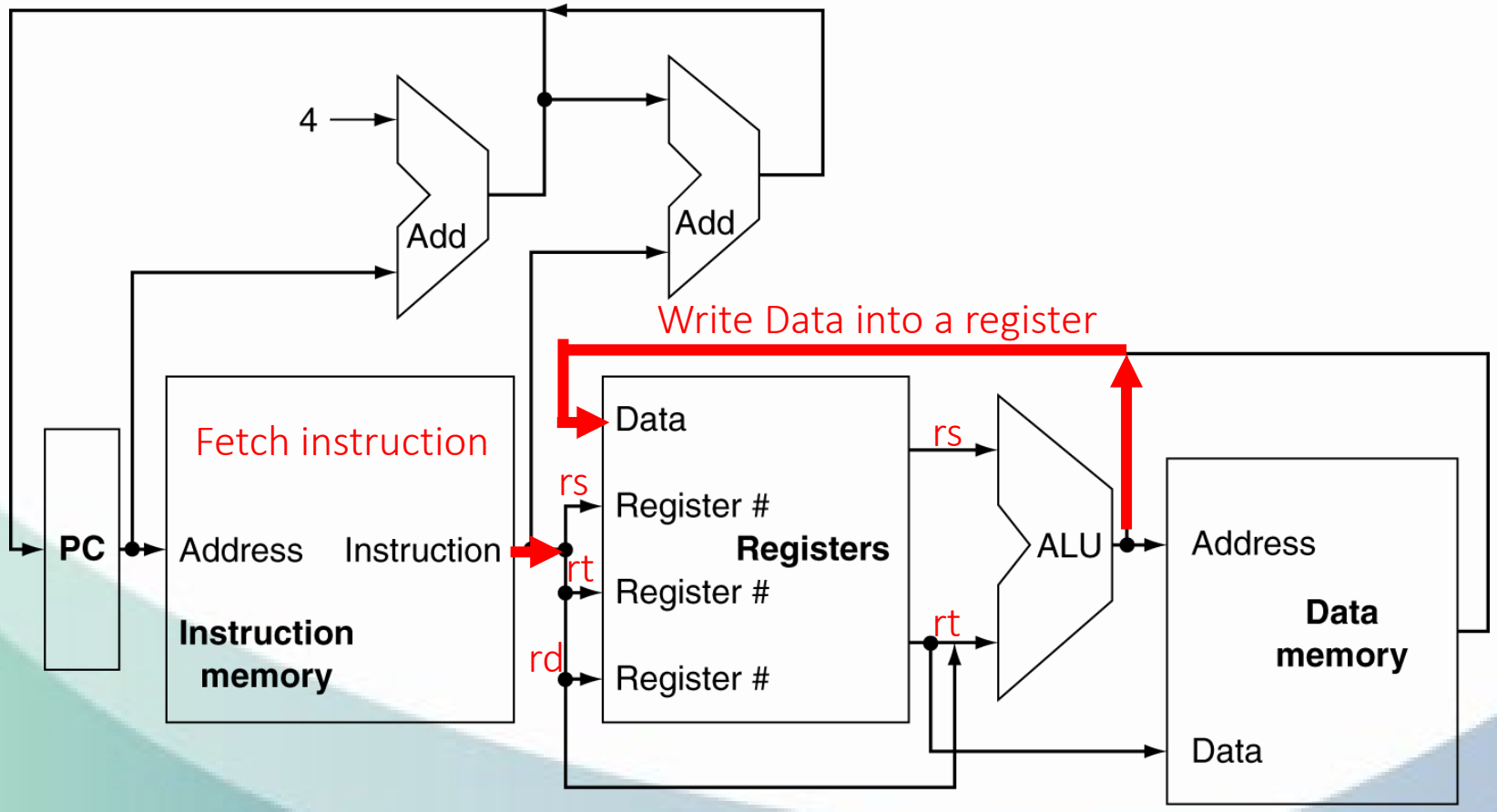
Processor Overview



Processor Overview



Processor Overview

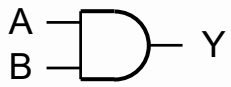


Logic Design Basics

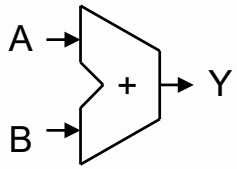
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Logic

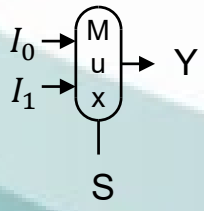
- AND-gate
  - Y = A & B



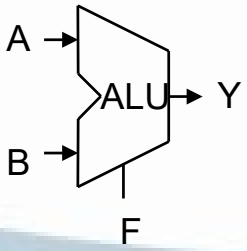
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0

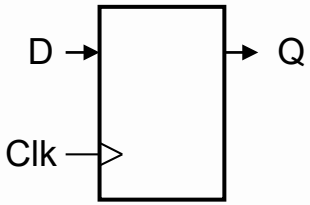


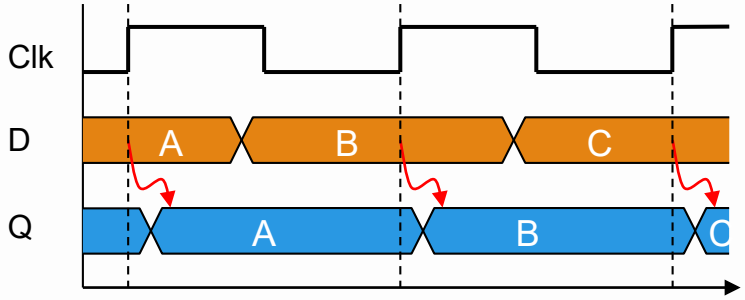
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Logic

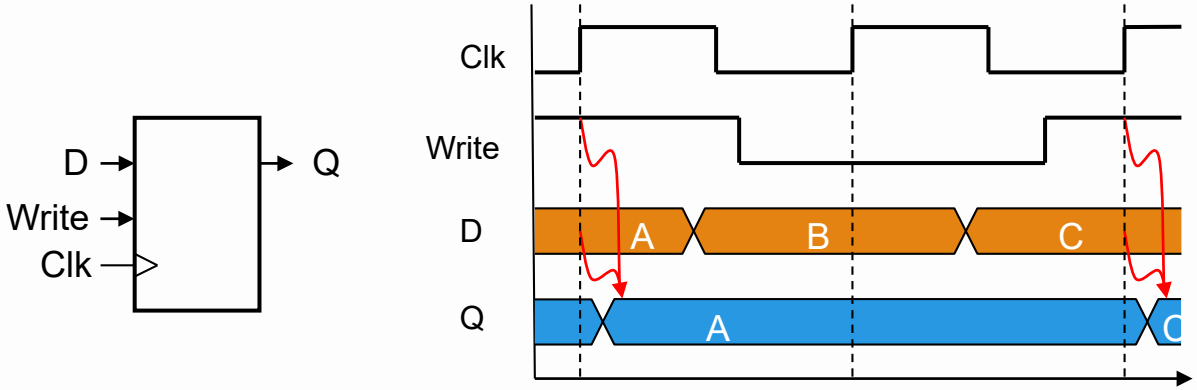
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1





Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

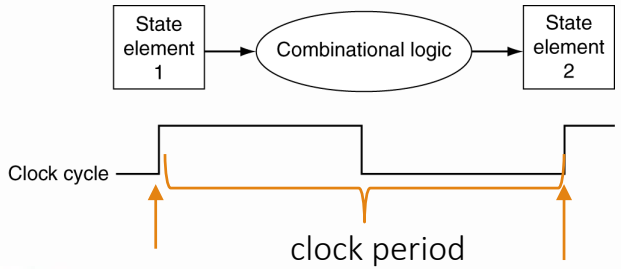


  - Setup time: the time span when the input signal needs to keep stable before the rising/falling) clock edge
  - Hold time: the time span when the input signal needs to keep stable after the rising/falling clock edge to prevent the signal from being overwritten by another signal

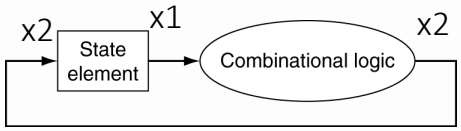
Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

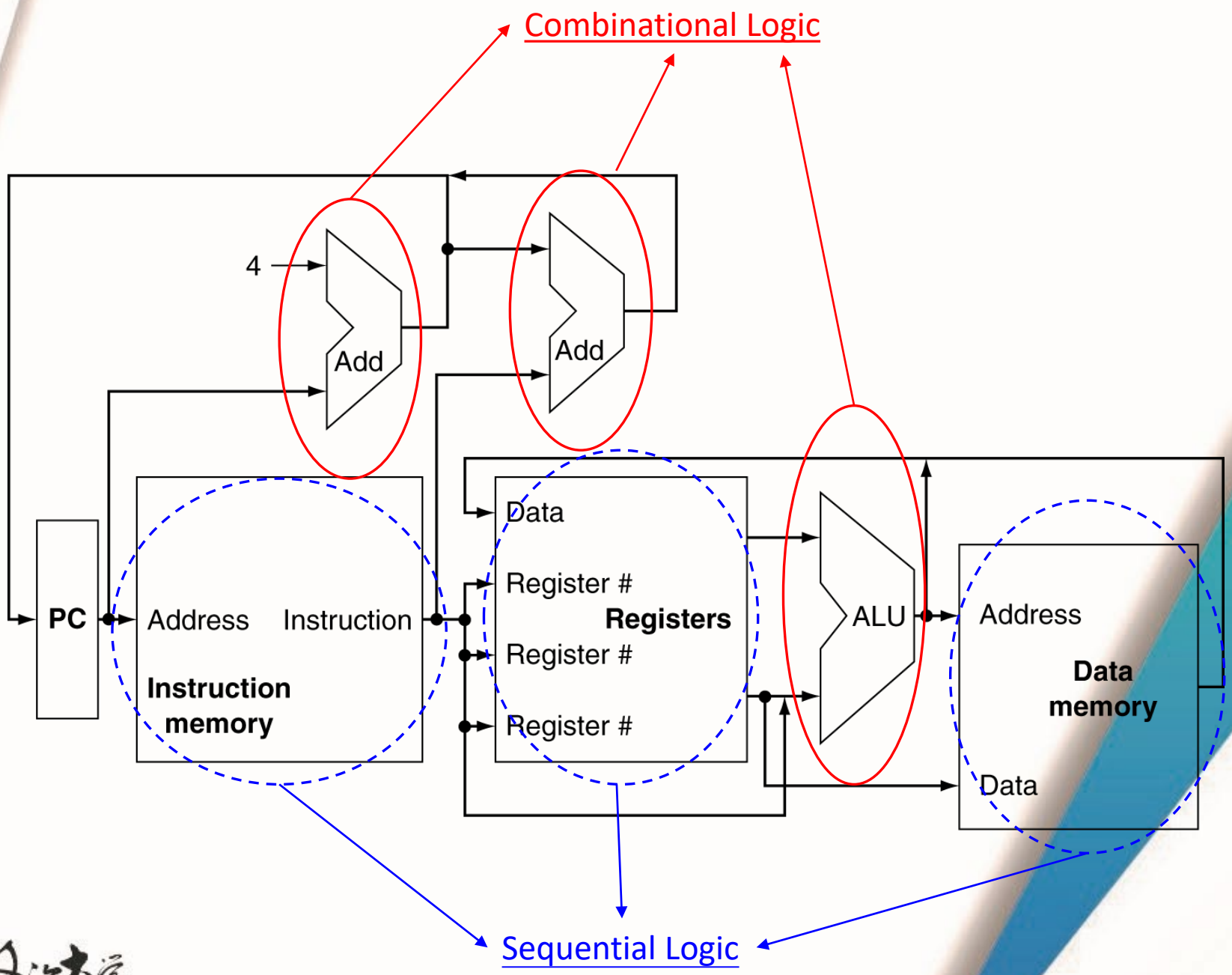
A signal has to be transformed from S1 to S2 through C



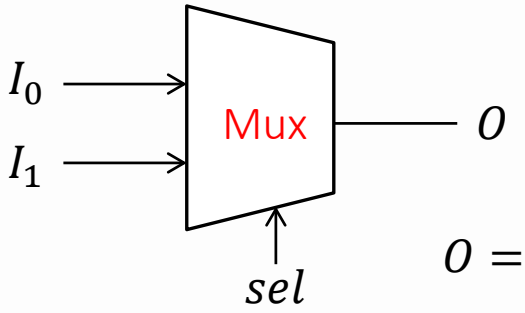
rising edge-triggered



State will be x2 upon the next rising edge.
Before that, it will still be x1.

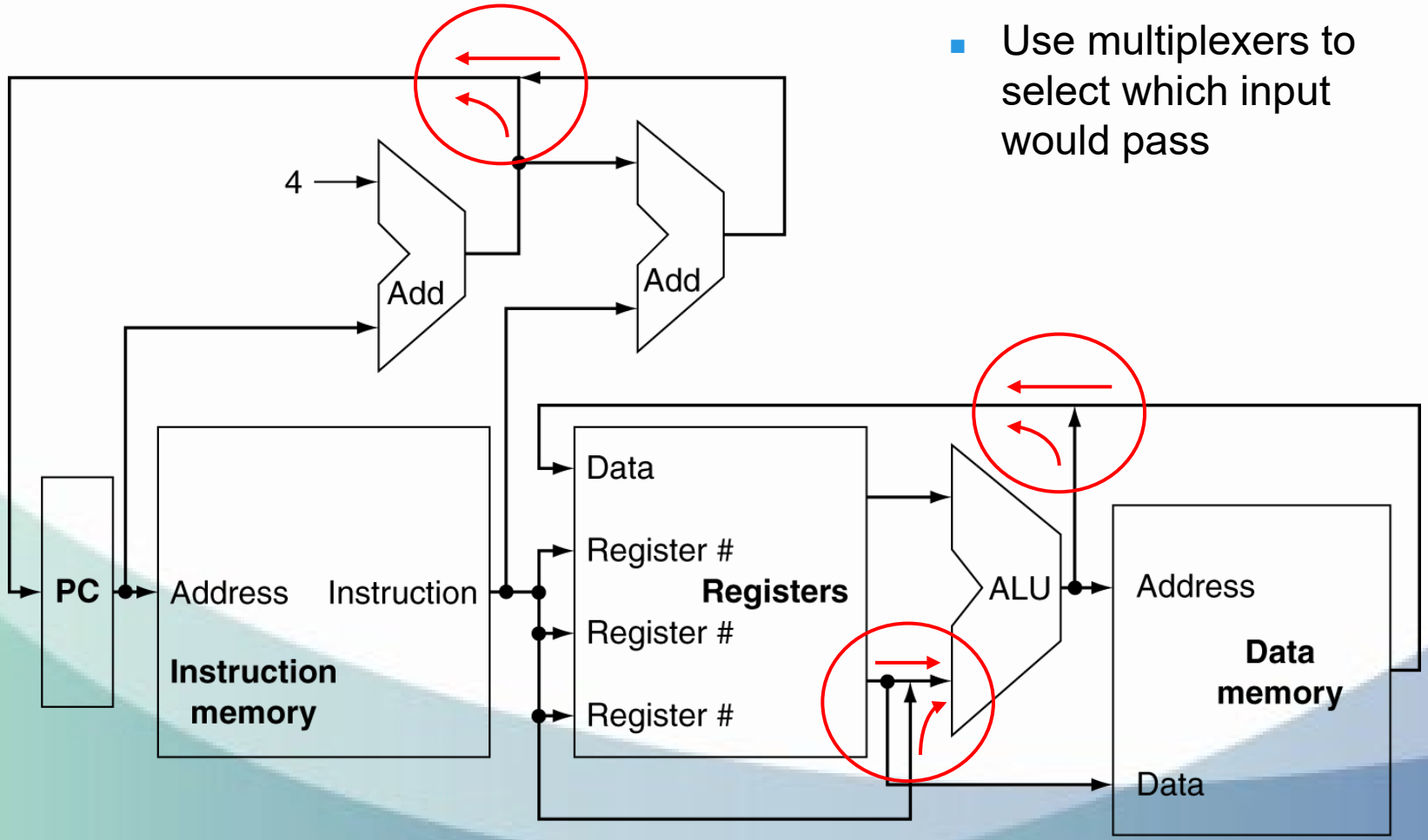

Combinational Logic
4
Add
Add
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
ALU
Address
Data memory
Data
Sequential Logic

Multiplexers

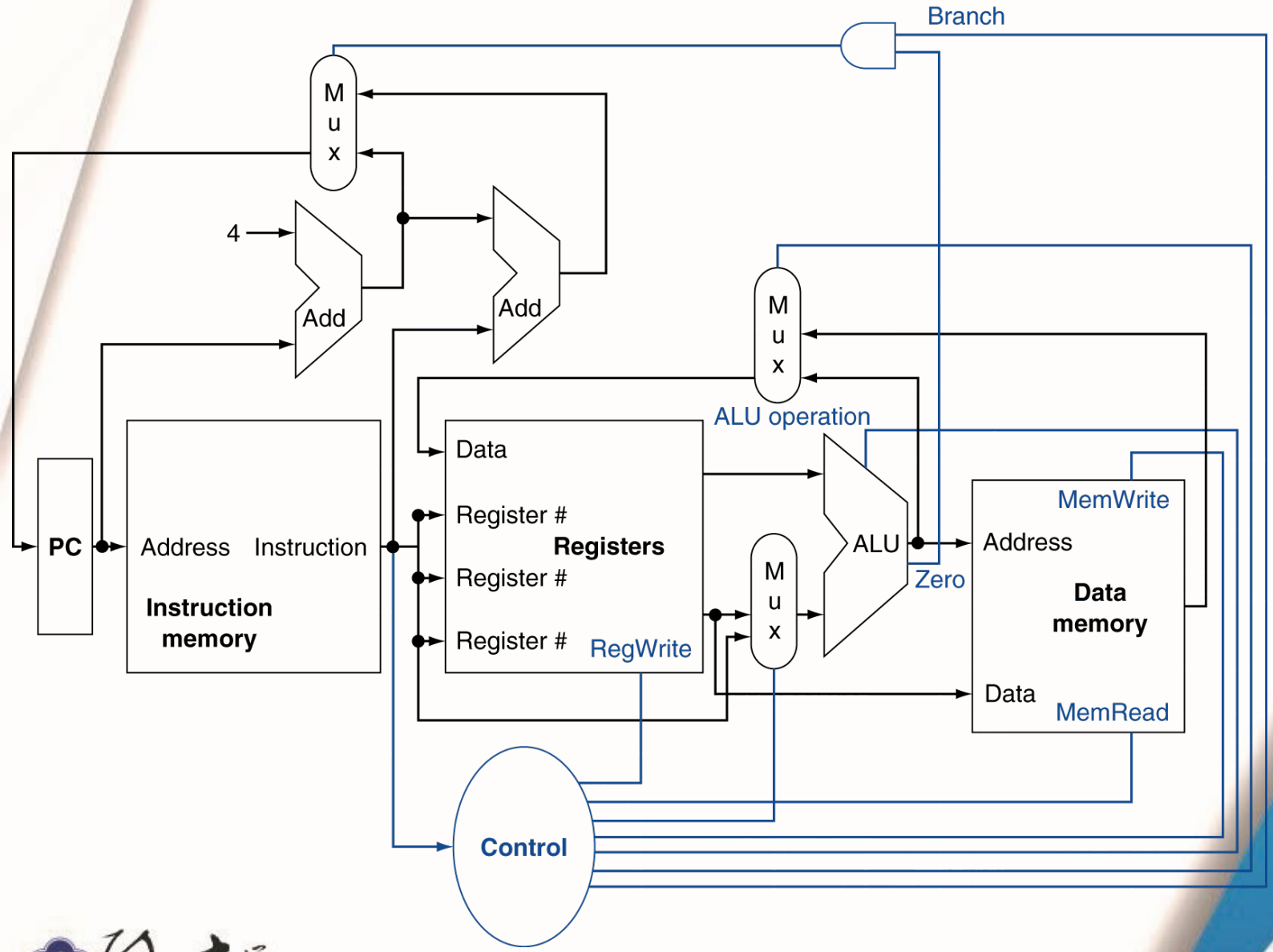


$$O = \begin{cases} I_0 & \text{if } sel = 0 \\ I_1 & \text{if } sel = 1 \end{cases}$$

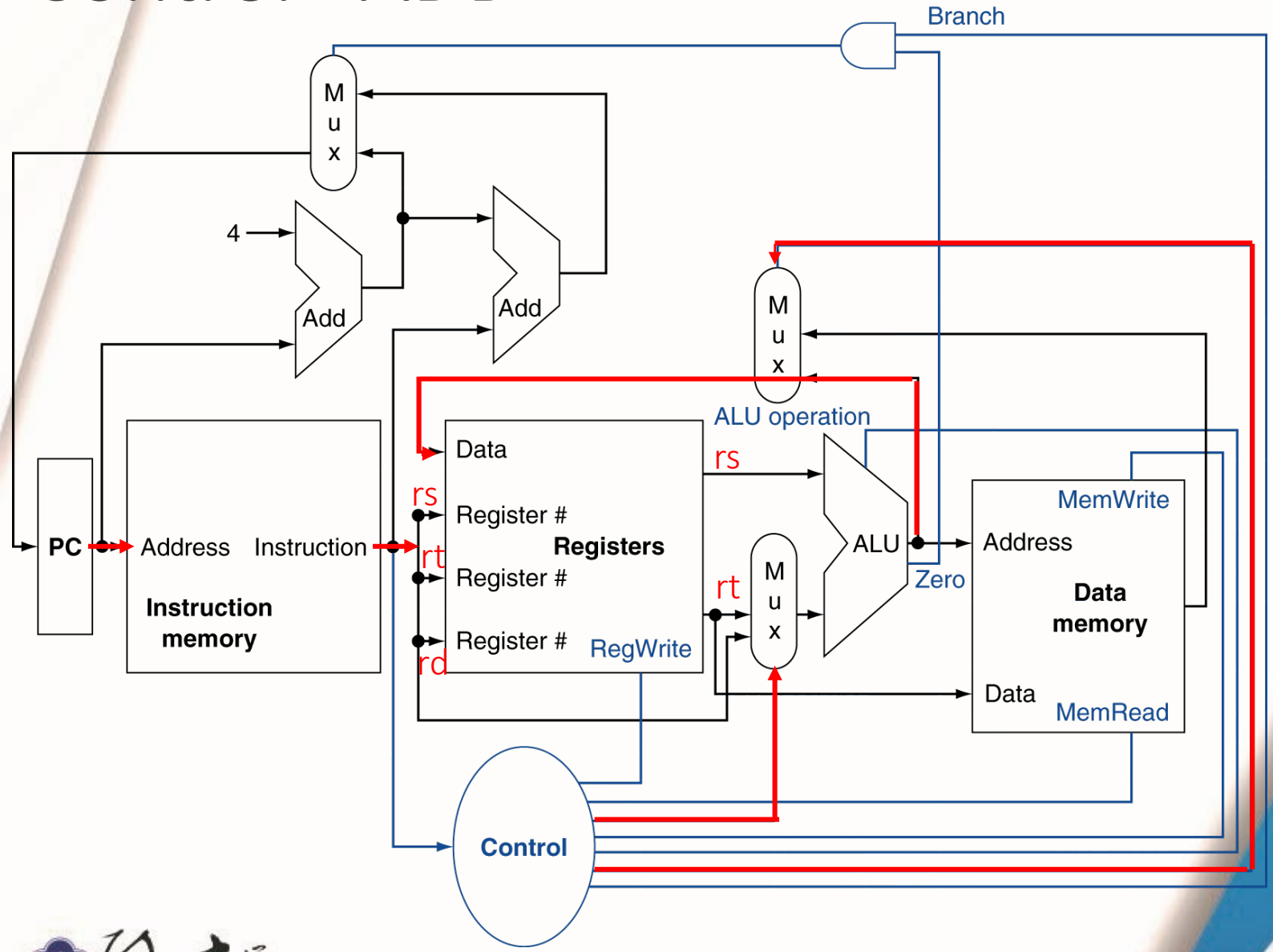
- Use multiplexers to select which input would pass



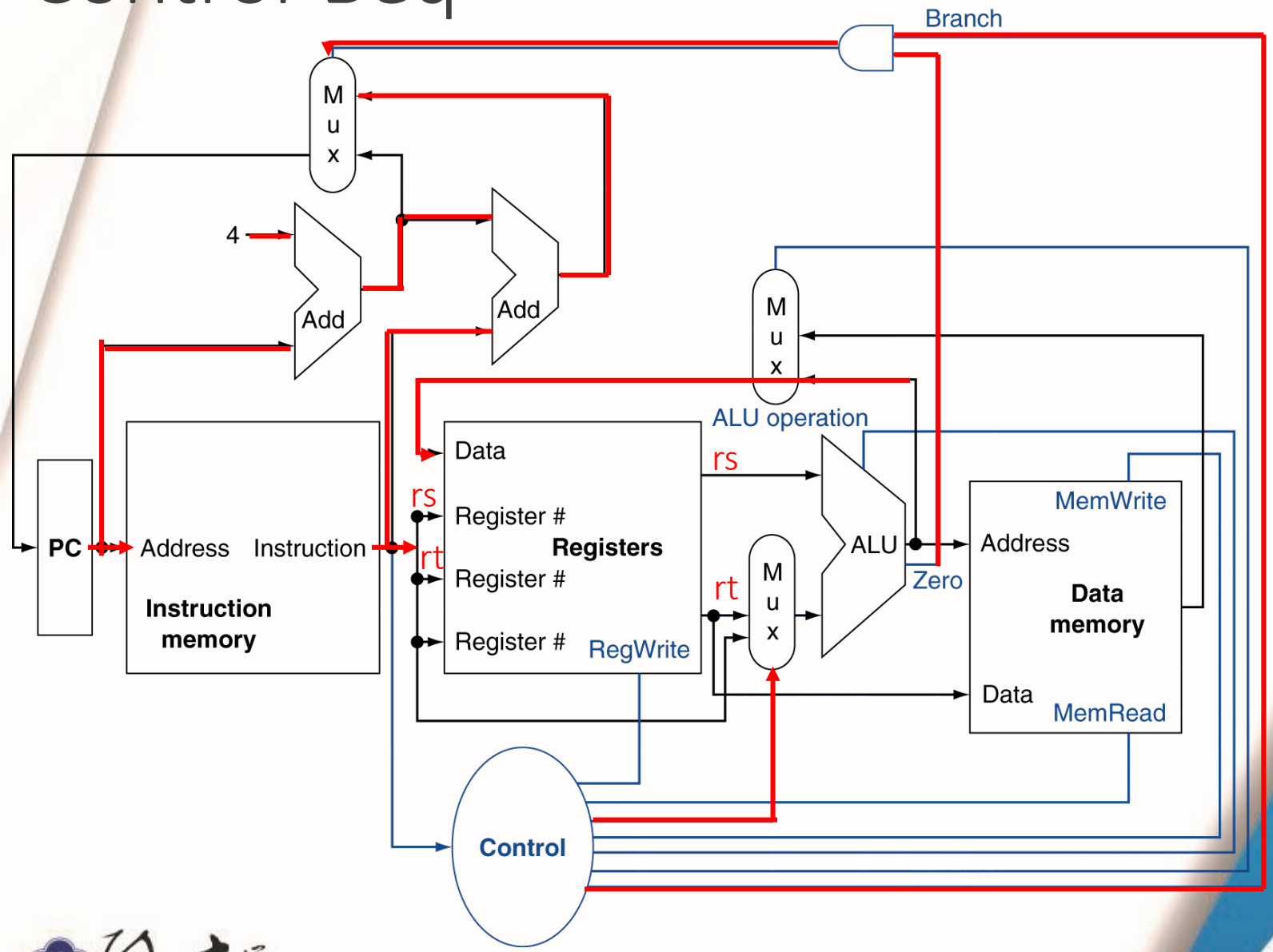
Control Unit



Control - ADD


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
rs
rt
rd
rs
rt
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

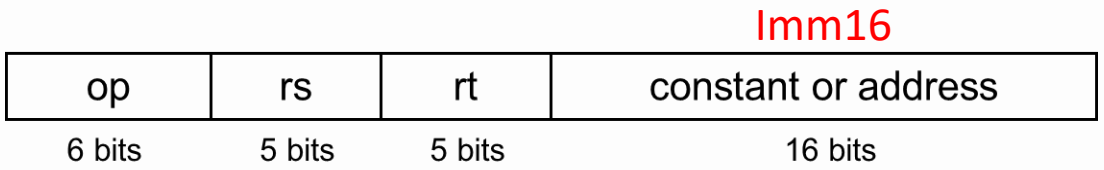
Control-Beq


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
rs
rt
rs
rt
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

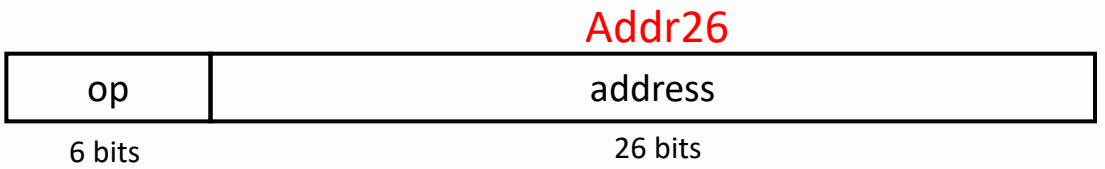
Instruction Review

- R-type
  - add, sub, or, slt

- I-type
  - lw, sw
  - addi
  - beq



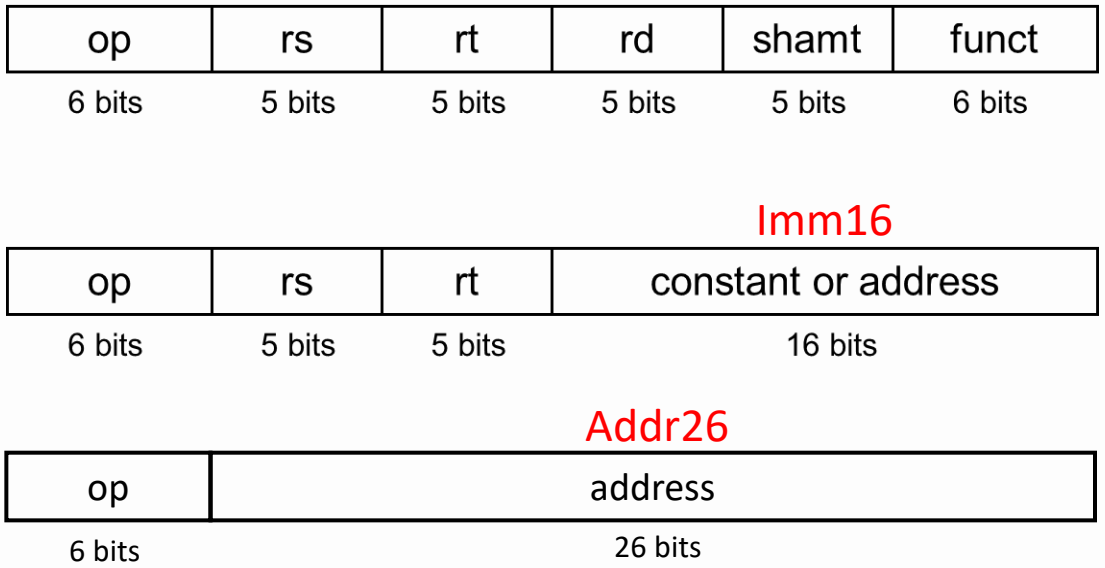
- J-type:
  - j



- Analyze instructions, including fields, addressing modes, for a CPU design

Instruction Review

- R-type
  - add rd, rs, rt
  - sub rd, rs, rt
  - and rd, rs, rt
  - or rd, rs, rt
  - slt rd, rs, rt
- I-type
  - lw rt,rs,imm16
  - sw rt,rs,imm16
  - addi rt,rs,imm16
  - beq rs,rt,imm16
- J-type:
  - j tar_address



Data Transfer in Memory and Register

IR←Mem[PC]

rs←IR[25:21] rt←IR[20:16] rd←IR[15:11] Imm16←IR[15:0] Addr26←IR[25:0]

add/sub: R[rd] ← R[rs]±R[rt]

lw: R[rt] ← Mem[R[rs]+sign_ext[Imm16]

sw: Mem[R[rs]+sign_ext[Imm16] ← R[rt]

addi: R[rt] ← R[rs]+sign_ext[Imm16]

(PC←PC+4)

beq: $PC \leftarrow \begin{cases} PC+4+(sign\_ext[Imm16] \ll 2), & \text{if } R[rs] == R[rt] \\ PC+4, & otherwise. \end{cases}$

j: PC ← PC[31:28] || Addr26 || 00

Datapath for Instruction Execution

I←Mem[PC] PC←PC+4

add/sub: R[rd] ← R[rs]±R[rt]

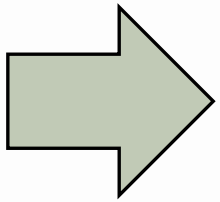
lw: R[rt] ← Mem[R[rs]+sign_ext[Imm16]

sw: Mem[R[rs]+sign_ext[Imm16] ← R[rt]

addi: R[rt] ← R[rs]+sign_ext[Imm16]

beq: PC ← $\begin{cases} \text{PC+4+(sign\_ext[Imm16]} \ll 2), & \text{if R[rs]} == \text{R[rt]} \\ \text{PC+4}, & otherwise. \end{cases}$

j: PC ← PC[31:28] || Addr26 || 00

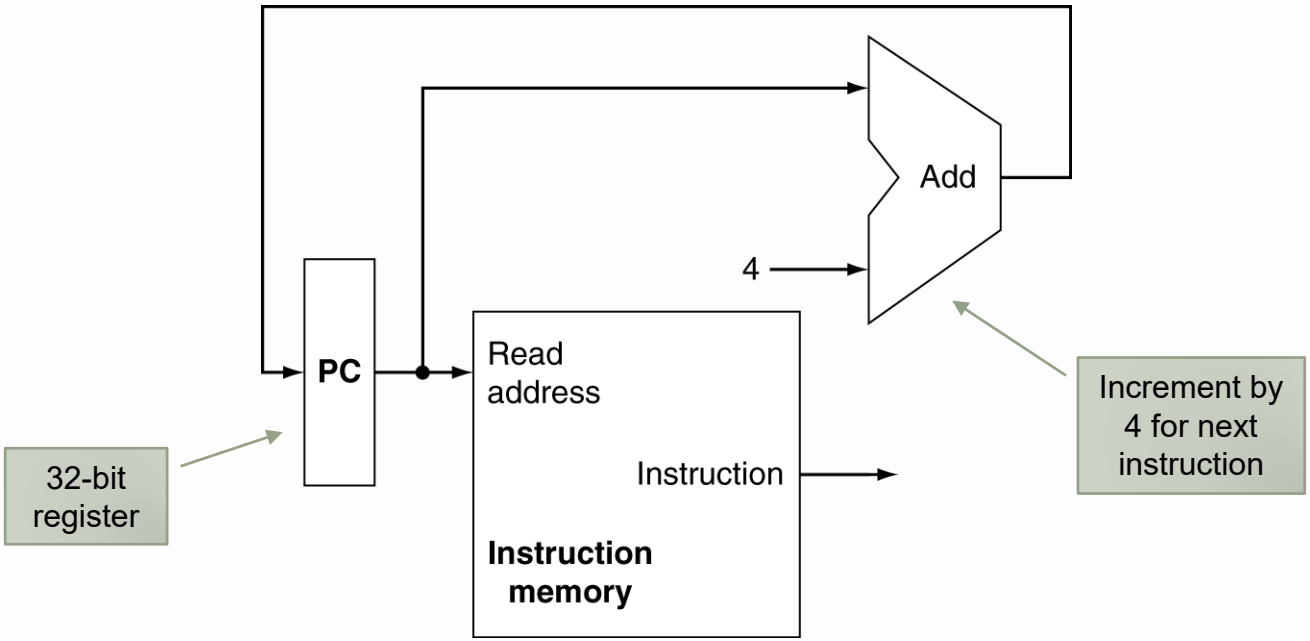


- A const added to PC
- Reg to Reg transfer
- Mem to Reg
- Reg to Mem
- Reg added to Reg
- An imm added to Reg
- Sign extension
- Concat with 00
- Concat with PC

Building a Processor

1. Analyze datapath & check which datapath elements each instruction needs
   - Datapath must have storage elements (register or memory)
2. Select datapath elements, which include a unit used to operate on or hold data in the CPU, and establish clocking methodology
   - Register file, ALUs, adder, memories
3. Need Program counter (PC)
   - The register having the address of the instruction in the program being executed
4. Analyze each instruction to design the control unit

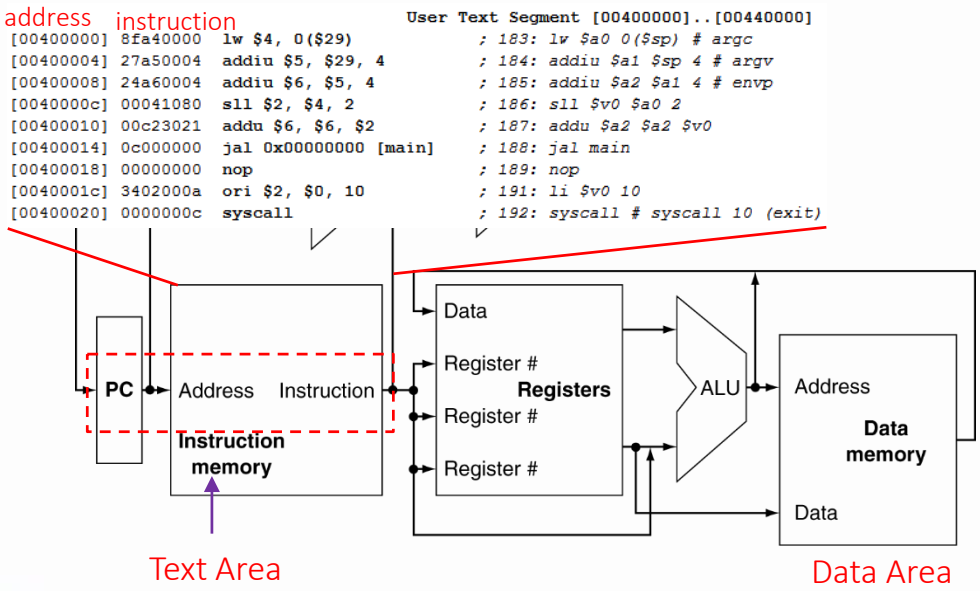
Instruction Fetch



- When Branch and Jump: PC would be assigned to a specified address (Mux)

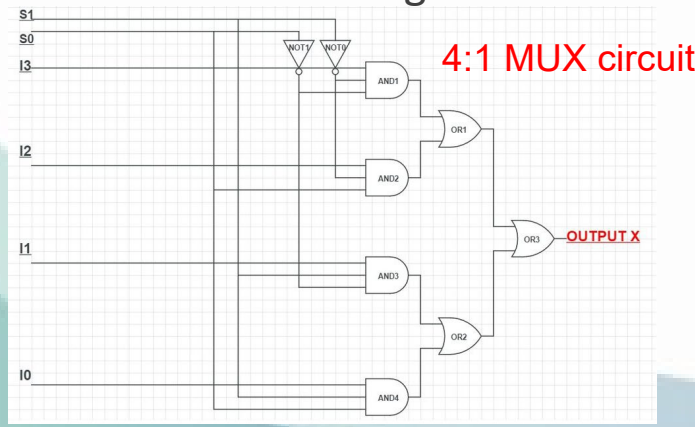
Instruction Fetch (IF)

- Send PC to the instruction memory to fetch the instruction



Register File

- Stacking multiple registers to construct a register file
- Register can be made by D Flip Flops with N-bit input and output
- Write Enable:
  - 0: Read, and Data Out does not change



4:1 MUX circuit

When "Read"

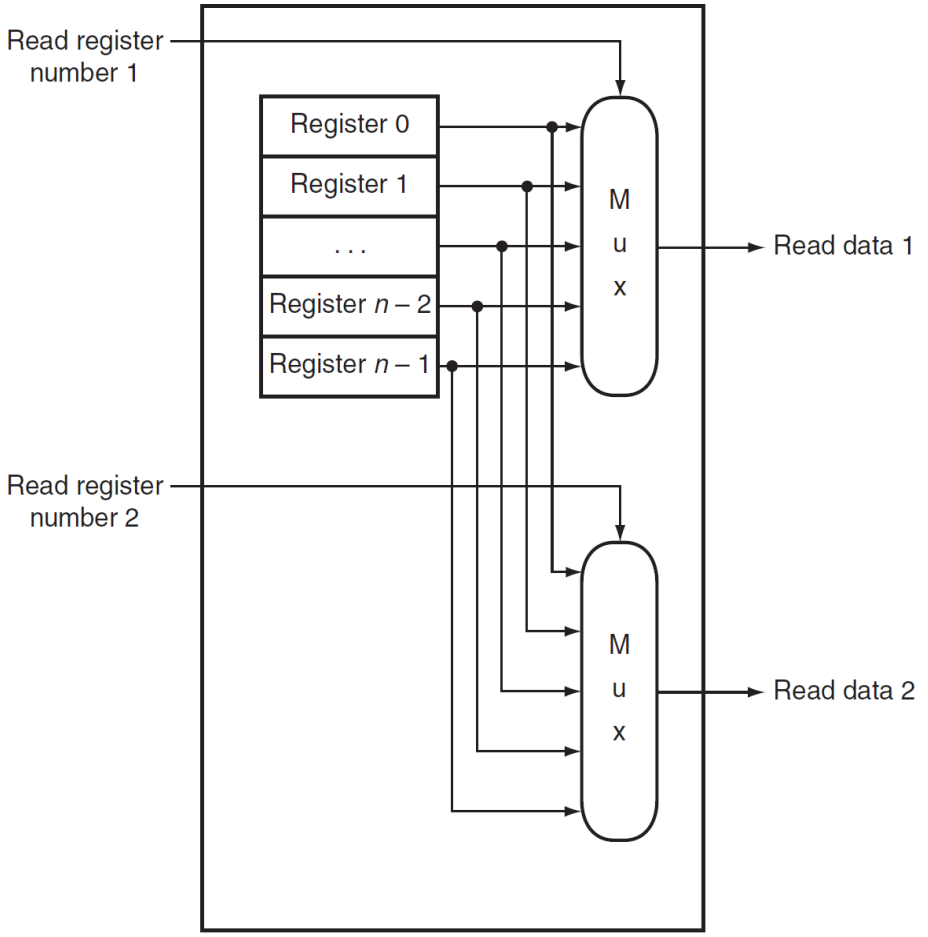


FIGURE B.8.8 from "Computer Organization and Design MIPS Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)" 5th Edition by David A. Patterson (Author), John L. Hennessy

https://en.wikipedia.org/wiki/Multiplexer

Register File

- Write Enable:
  - 0: Read, and Data Out does not change
  - 1: Write, and Data Out becomes Data In

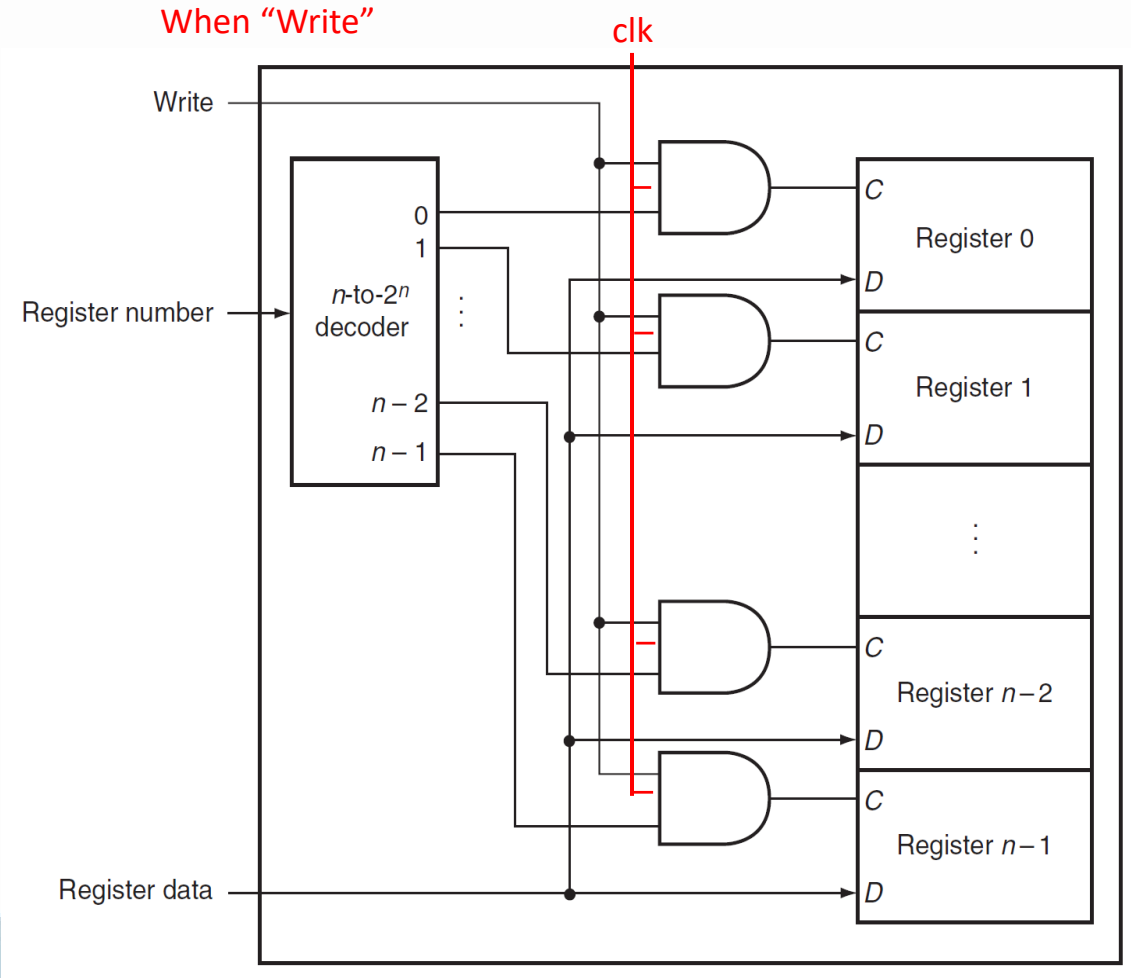
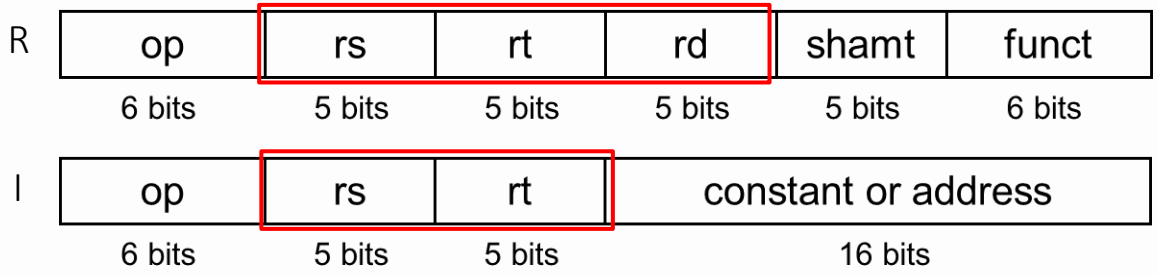


FIGURE B.8.9 from "Computer Organization and Design MIPS Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)" 5th Edition by David A. Patterson (Author), John L. Hennessy

Instruction Decode (ID)

- Register numbers → read registers



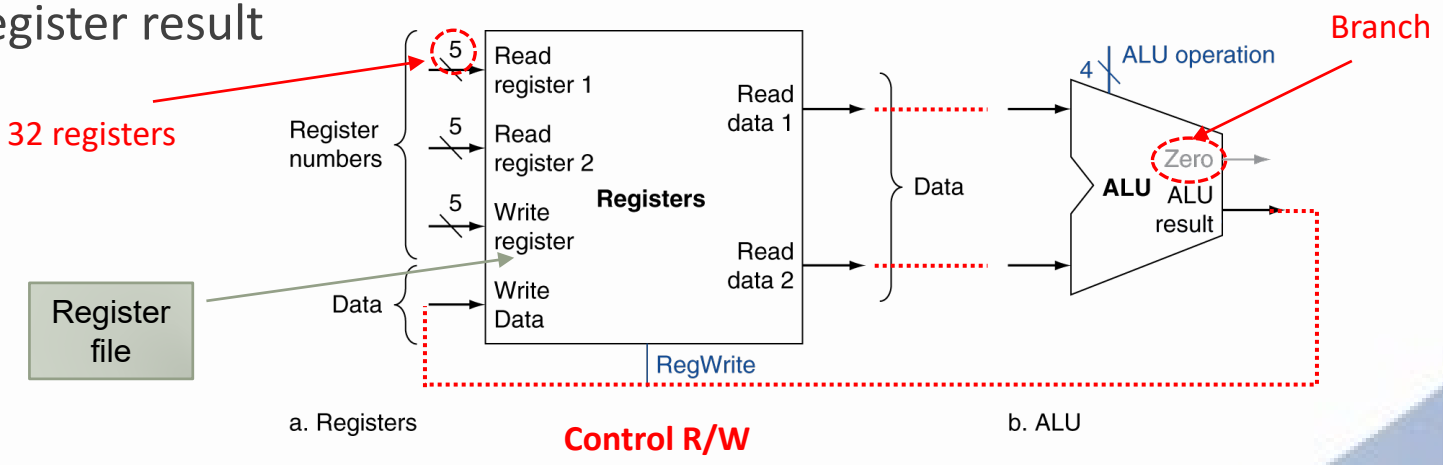
For R-type instructions, it decodes rs and rt
Take **lw,** an I-type instruction as an example, it decodes rs (base reg)

Not yet touched rd of R-type instructions or rt of **lw** because it is in the "write back" stage (WB)

R-Format Instructions

- After fetching the instruction, the CPU does

1. Read two register operands
2. Perform arithmetic/logical operation (add, sub, and, or, and slt)
3. Write register result

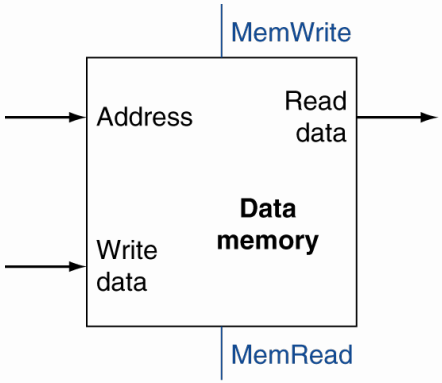


Load/Store Instructions

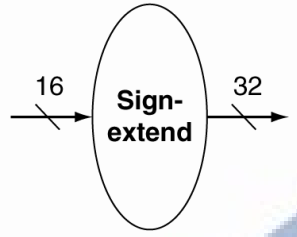
lw: R[rt] ← Mem[R[rs]+sign_ext[Imm16]

sw: Mem[R[rs]+sign_ext[Imm16] ← R[rt]

- Read register operands
  - a source/destination register and the base register
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
  - sign_extend(offset_value) + R[rs]
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit



b. Sign extension unit

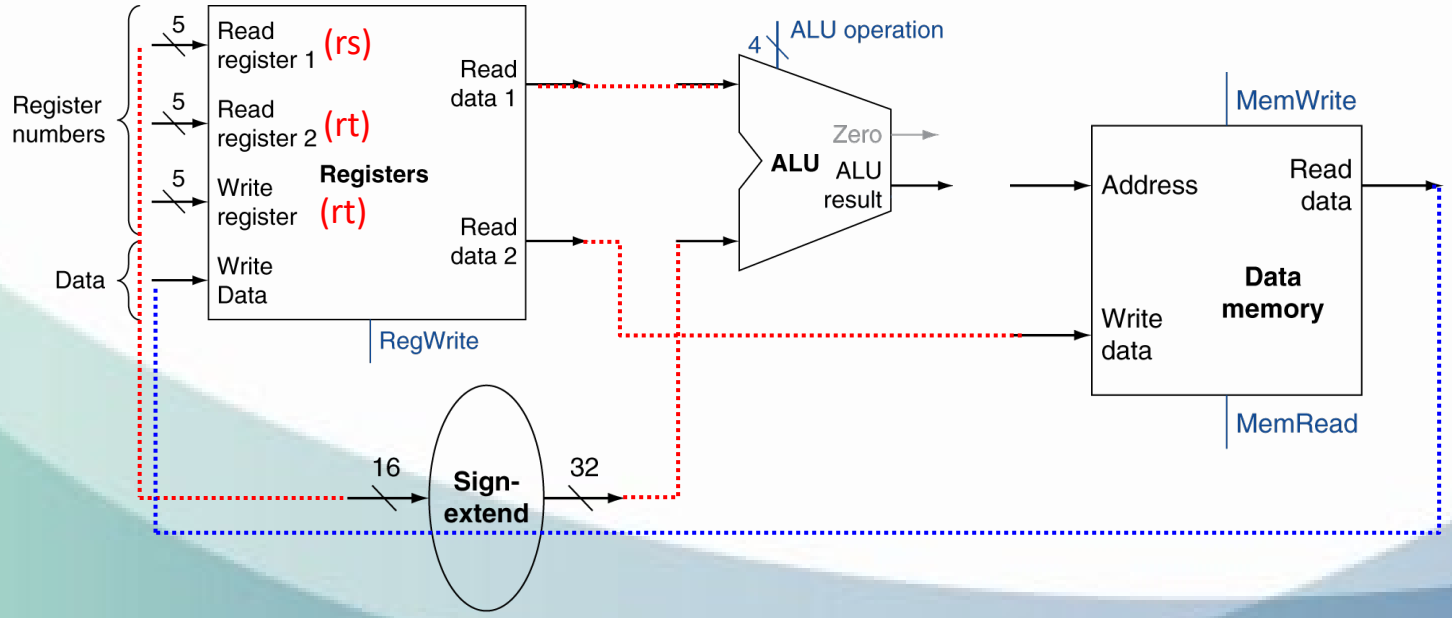
Load/Store Instructions

Imm16

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

lw: R[rt] ← Mem[R[rs]+sign_ext[Imm16]

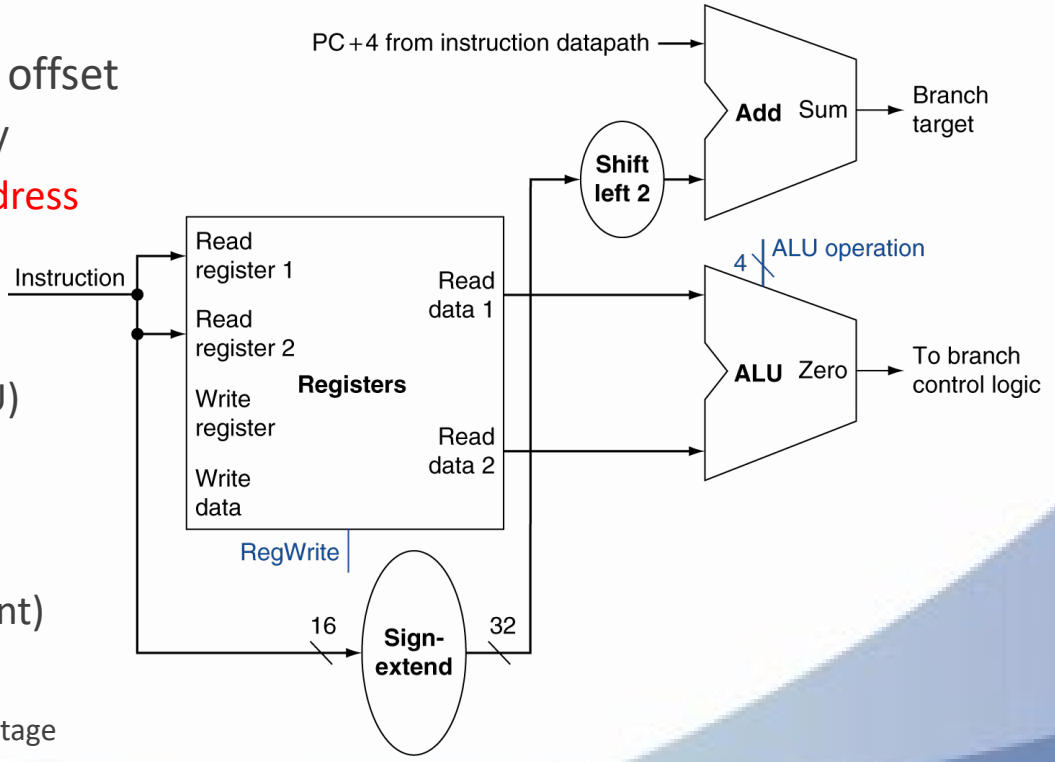
sw: Mem[R[rs]+sign_ext[Imm16] ← R[rt]



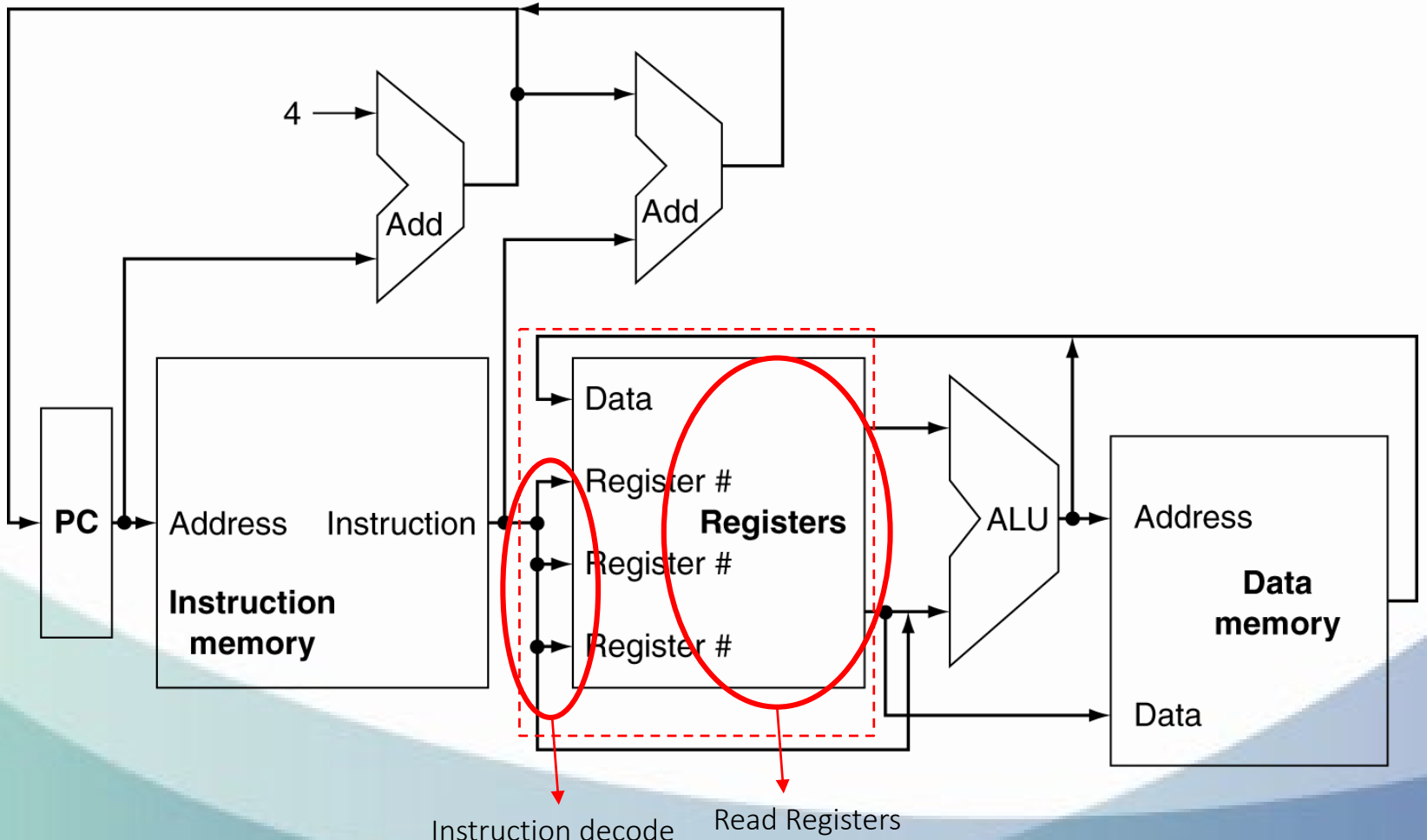
Branch Instructions

beq: $PC \leftarrow \begin{cases} PC+4+(sign\_ext[Imm16] \ll 2), & \text{if } R[rs] == R[rt] \\ PC+4, \ otherwise. \end{cases}$

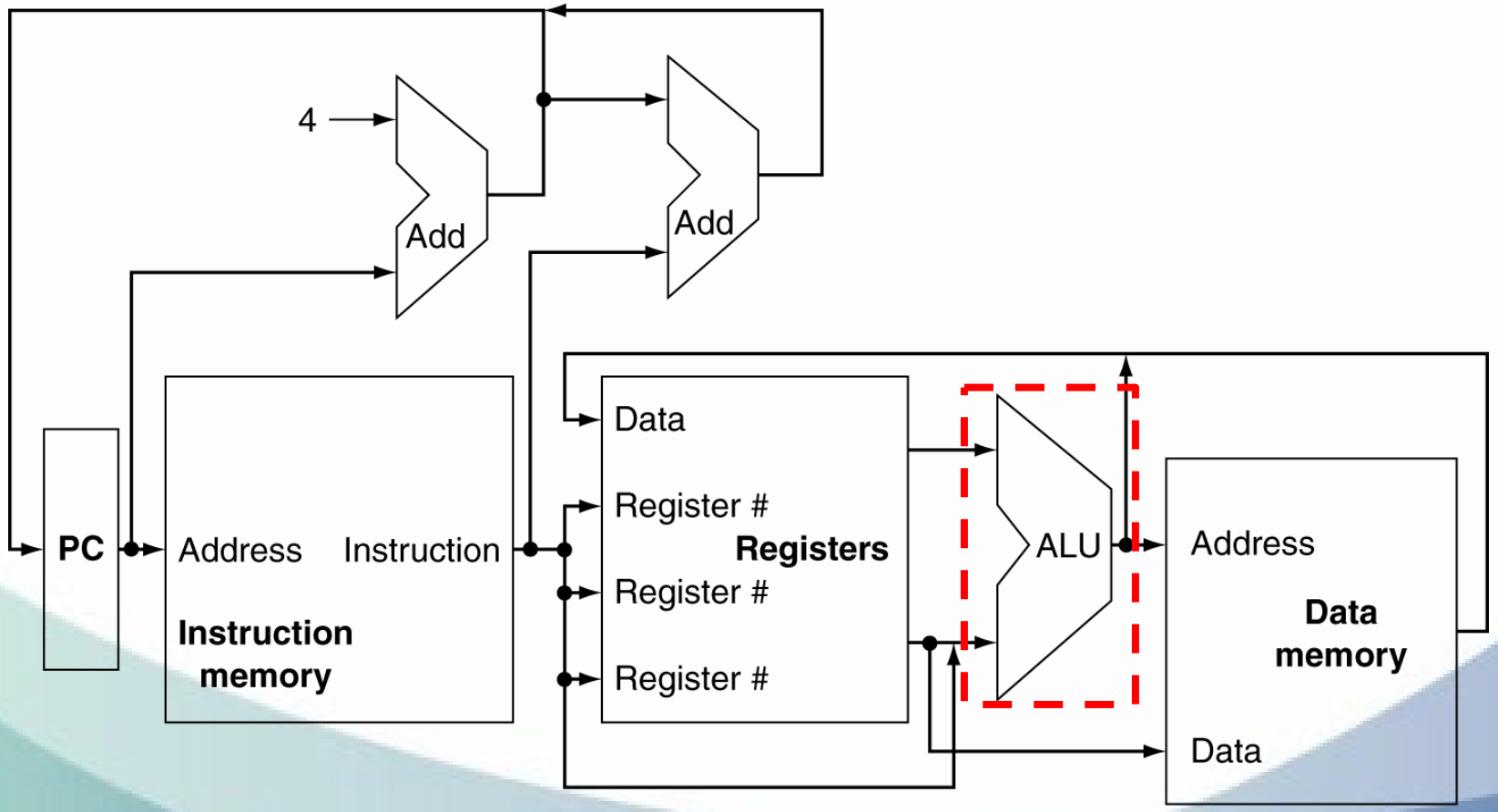
- Read two register operands and one offset
  - two registers compared for equality
  - a 16-bit offset for branch target address
    - Left shifted sign-extended offset + (PC+4)
- Compare two register operands
  - Subtract and check Zero signal (ALU)
- Calculate target address
  - Sign-extend the 16-bit offset
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated in the "Instruction Fetch" stage



Instruction Decode

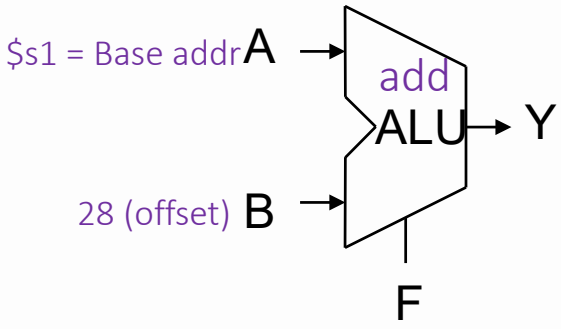


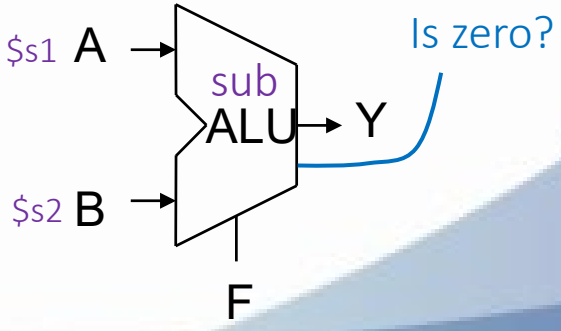
Execution (ALU)



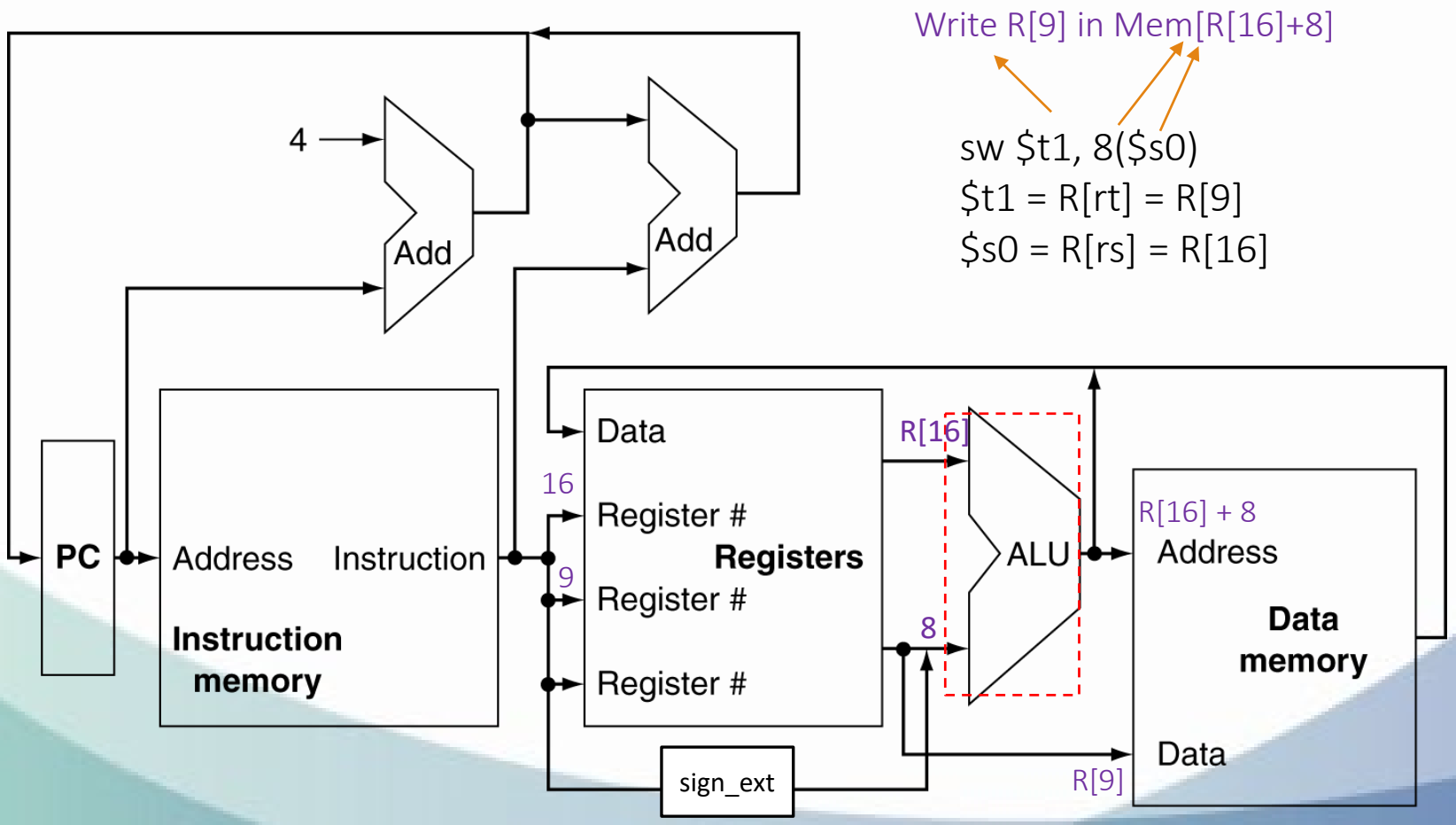
Execution (ALU)

- Arithmetic operations (R-type)
- Memory address calculation for load/store
  - lw $s2, 28($s1)
    - A=$s1, B=28
- Branch condition
  - beq $s1, $s2, Exit
    - $s1-$s2 (rs-rt==0?)





Store Memory



Write Back to Register

- Reg to Reg (R-type) or Mem to Reg (lw)
  - Write ALU result back to R[rd] (for R-type)
  - Write memory data to R[rt] (for lw)