

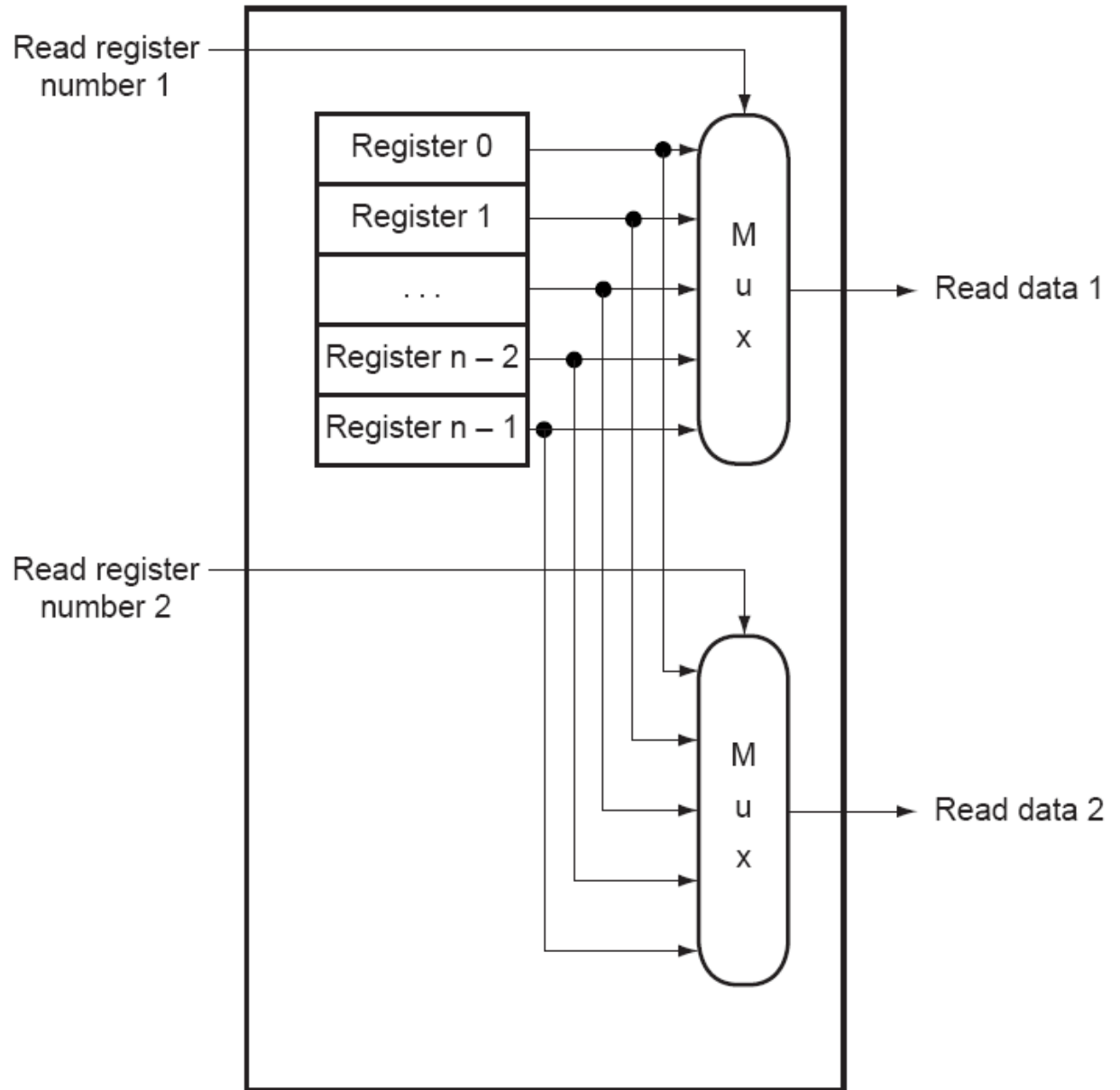
# MIPS Processor

# Registers in MIPS

- In MIPS, there are 32 Registers.
- We need read up to two registers, and write to up to one register.
- Think registers as D flip-flops. Each register has 32 Dffs.
- The control signals are:
  - readReg1, readReg2: 5 bits. Used to specify which reg to read.
  - writeReg: 5-bits. Used to specify which reg to write.
  - Data: if write, what data should be written into the reg.
  - RegWrite: whether to write or not.



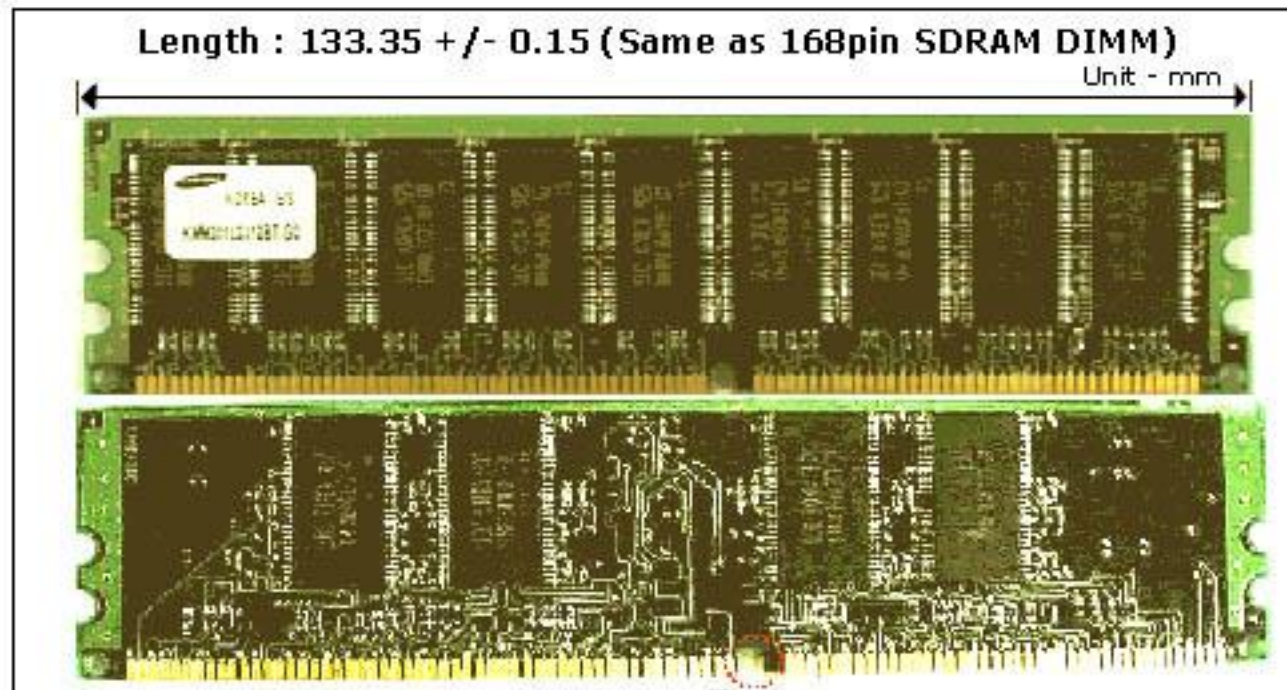
- This is for read.



# To write to a register

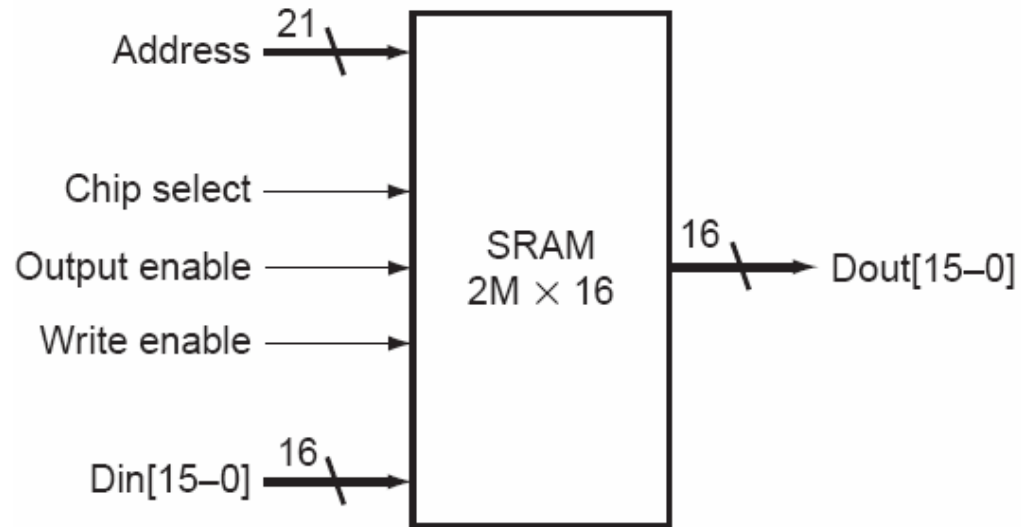
- The data is connected to every register.
- Use RegWrite, generate a ``LOAD'' signal for the register you want to write to.
  - Every register has a LOAD signal. If that signal is `1', new data will be set.
  - Only the target register's LOAD signal is `1'.

# RAM



# A RAM Example

- RAM. Control signals:
  - address: If write, which location to write to. If read, which location to read from.
  - Chip select: whether to use this chip or not.
  - Output enable: whether to enable output (output some voltage or in high-impedance state)
  - Write enable: whether to read or write.
  - Din: if write, what data should be written into the location specified by address.

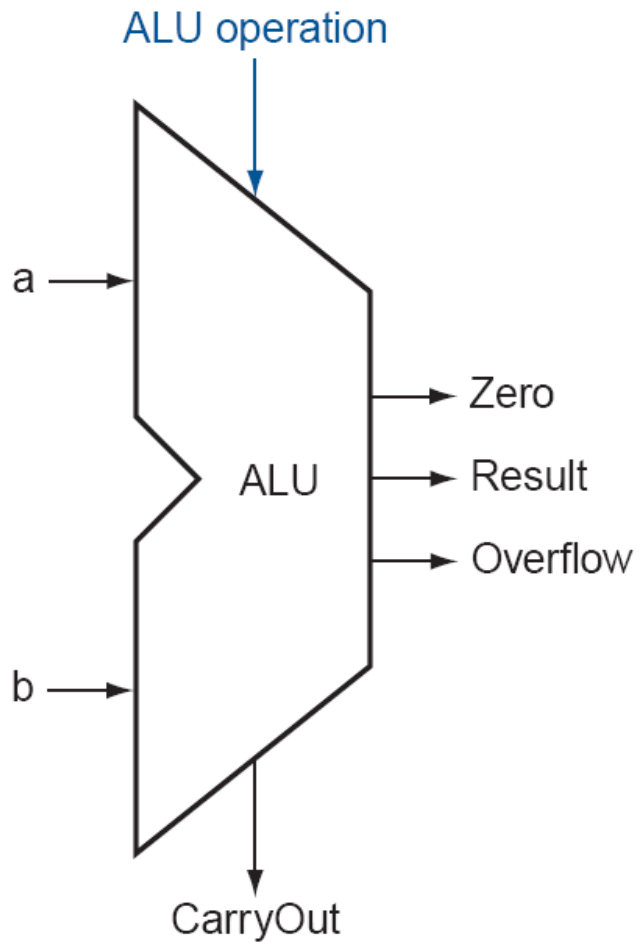


- Assume that there is a RAM with only 2 address lines and two bit data lines. How many bits can it hold?

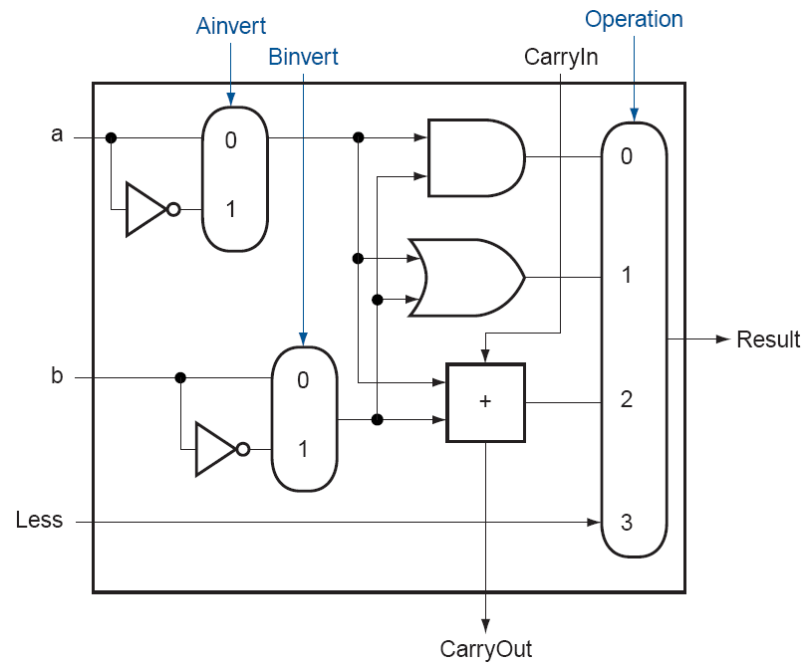
# The processor

- We now know all the parts in the processor.
  - ALU
  - PC
  - Register file
- Also
  - RAM
- How to put them together? How to make them execute an instruction as we need?

# ALU



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





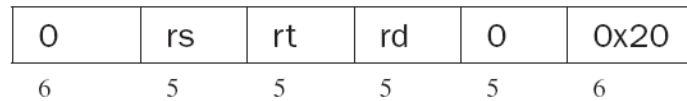
# The execution of an instruction

- First we need to fetch the instruction at the address given by the current PC from instruction memory
- Then we need to decode the instruction
- Based on the instruction, we need to do accordingly
- For sequential instructions, we then go the next instruction by increasing the PC. For jump and branch instructions, PC will be changed

# Basic MIPS Implementation

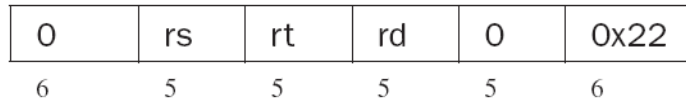
- We will focus on design of a basic MIPS processor that includes a subset of the core MIPS instruction set
  - The arithmetic-logic instructions add, sub, and, or, and slt
  - The memory-reference instructions load word and store word
  - The instructions branch equal and jump

add rd, rs, rt

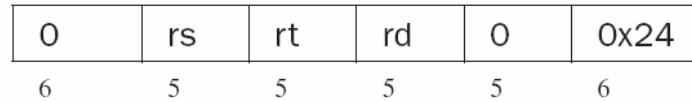


Put the sum of registers *rs* and *rt* into register *rd*.

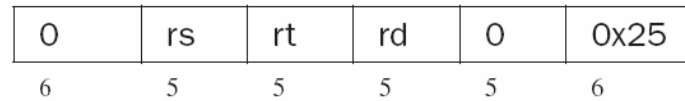
sub rd, rs, rt



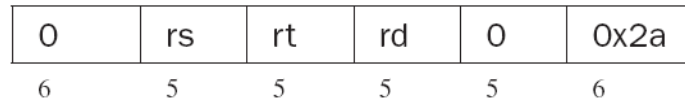
and rd, rs, rt



or rd, rs, rt

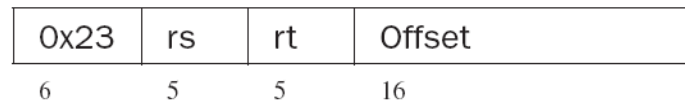


slt rd, rs, rt



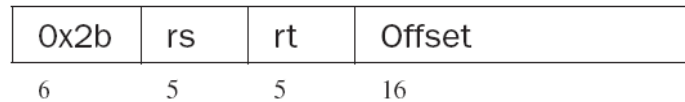
Set register *rd* to 1 if register *rs* is less than *rt*, and to 0 otherwise.

lw rt, address

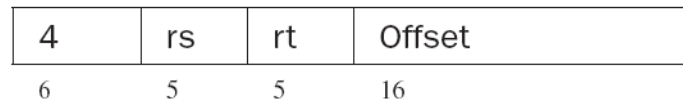


Load the 32-bit quantity (word) at *address* into register *rt*.

sw rt, address

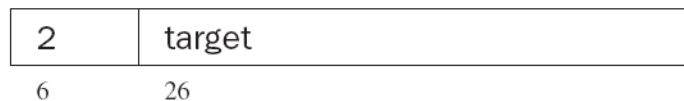


beq rs, rt, label



Conditionally branch the number of instructions specified by the offset if register *rs* equals *rt*.

j target



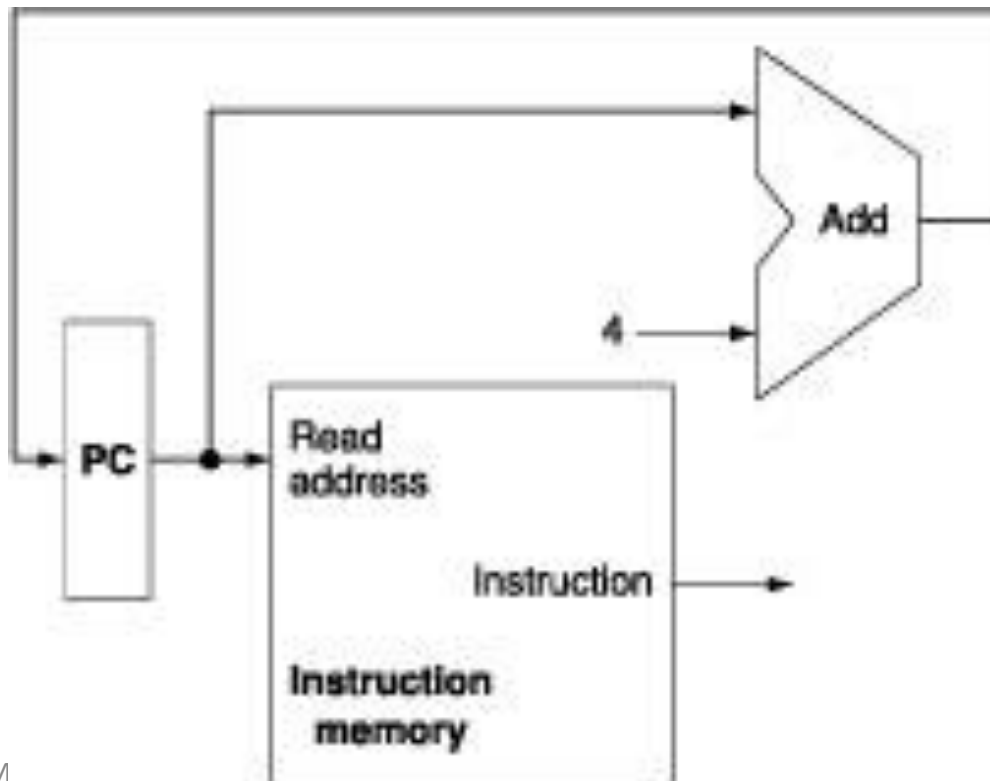
Unconditionally jump to the instruction at *target*.

# MIPS Implementation Overview

- For every instruction, the first two steps are identical
  - Fetch the instruction from the memory according to the value of the program counter
  - Read one or two registers (using fields of instructions to select the registers)
    - For load word, we need to read only one register
    - Most other instructions (except jump) require we read two registers
  - After the two steps, the actions required depend on the instructions
    - However, the actions are similar

# Instruction Fetch and PC Increment

- Since for every instruction, the first step is to fetch the instruction from memory
  - In addition, for most instructions, the next instruction will be at  $PC + 4$

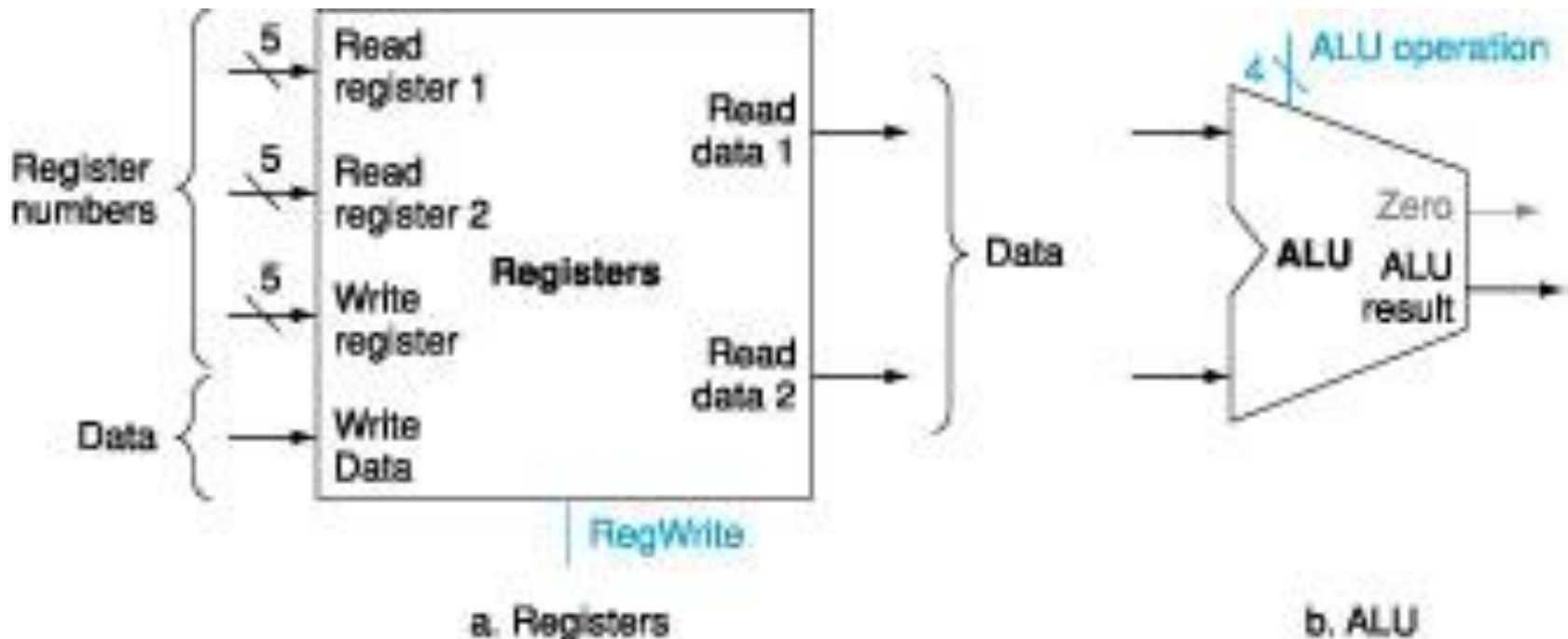


# R-type Instructions

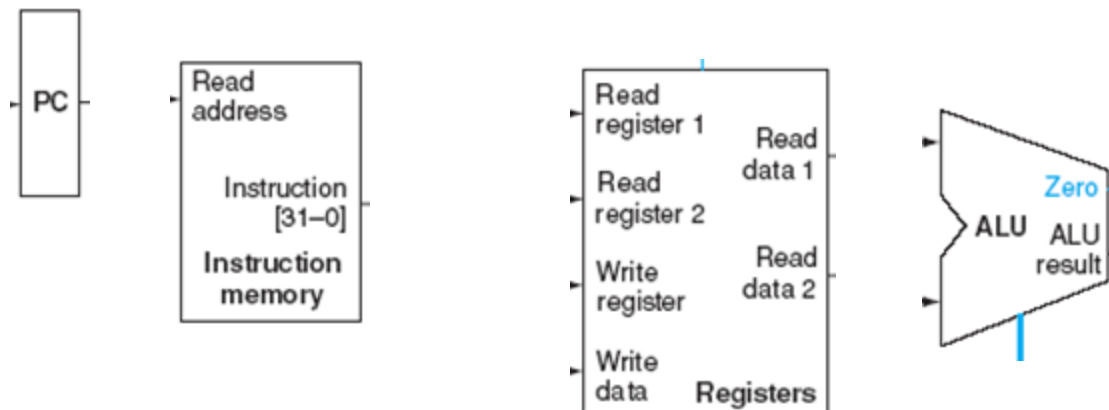
- Also called arithmetic-logical instructions
  - Including add, sub, and, or, and slt
  - Each one reads from two registers, performs an arithmetic or logical operation on the registers, and then write the result to a register

# R-type Instructions

- Suppose the instruction is `add $t0, $t1, $t2`, what are the read reg1, read reg2, and write reg? What is the value of RegWrite? How to control the ALU to do add?

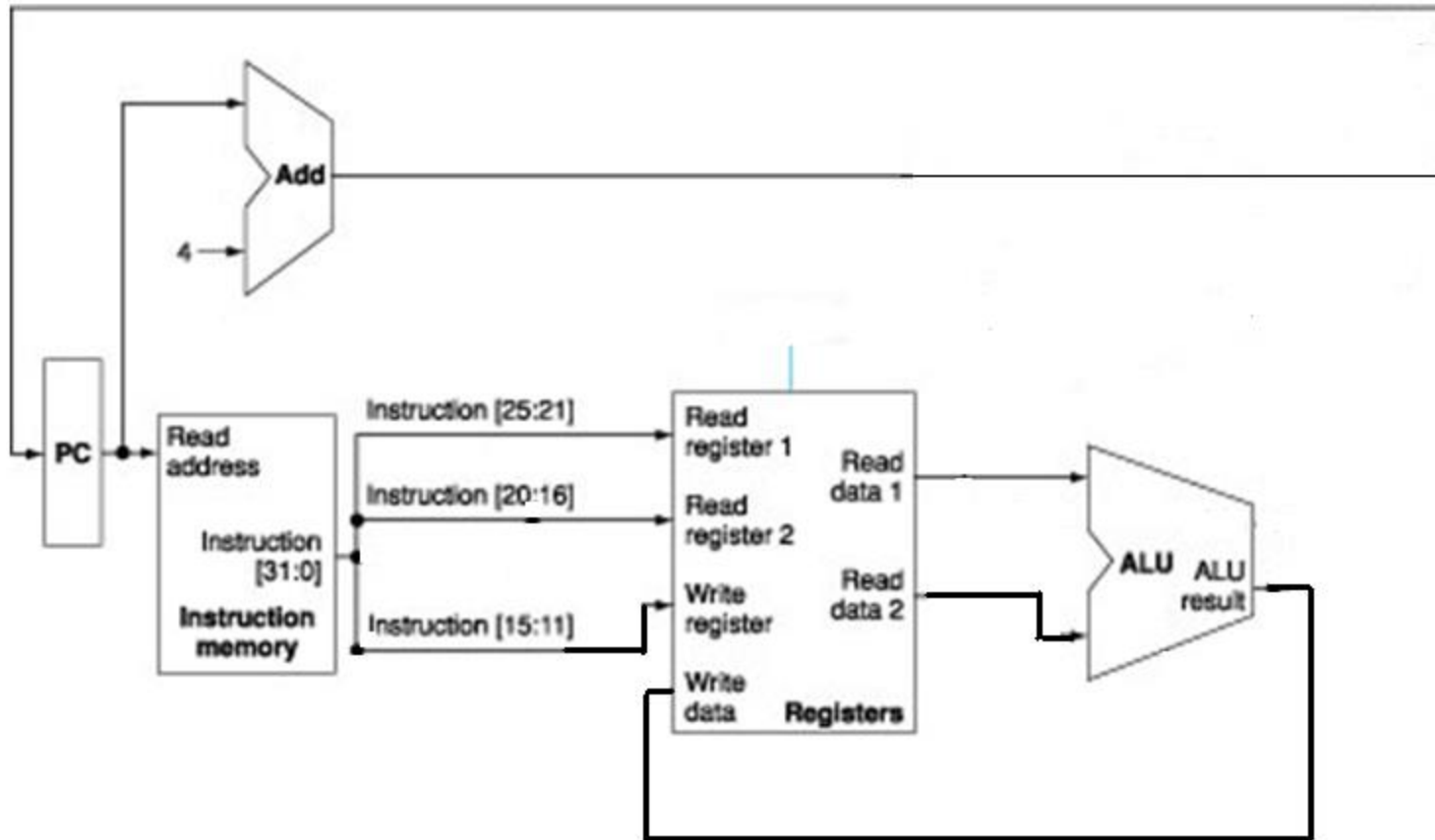


# Datapath only for R-type instructions





# Datapath only for R-type instructions (Answer)

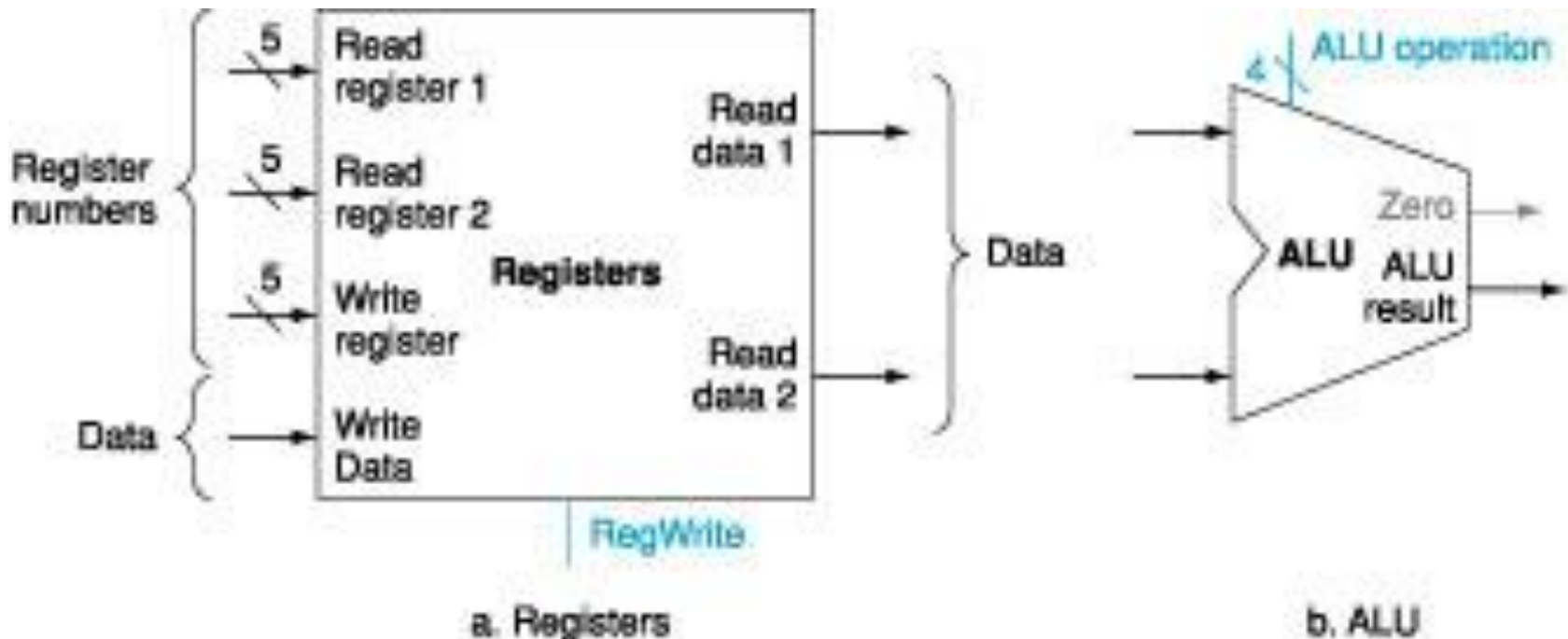


# Data Transfer Instructions

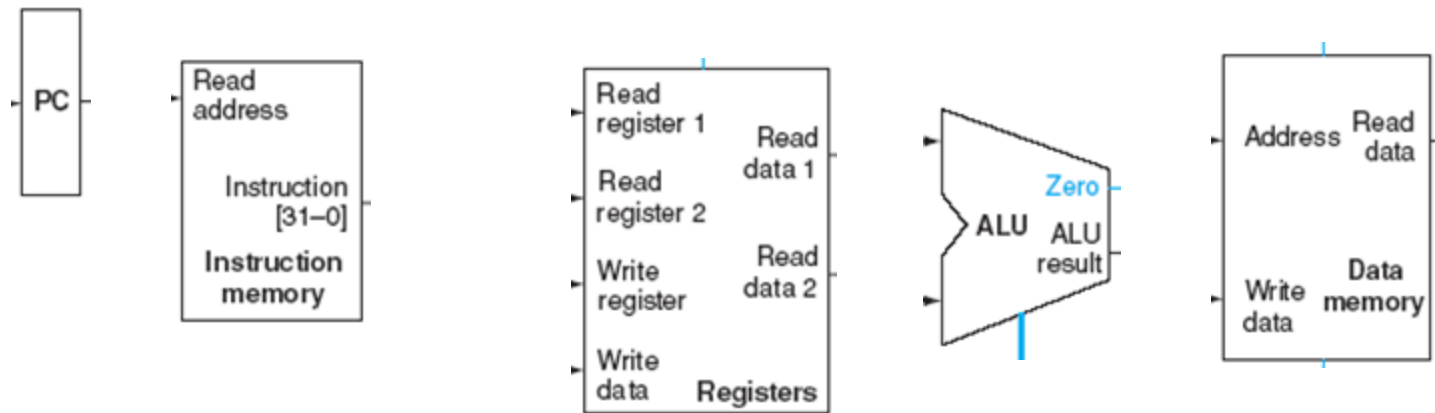
- Load word and store word instructions have the following general form
  - `lw $rt, offset_value($rs)`
    - opcode (6 bits) rs (5 bits) rt (5 bits) offset (16 bits)
  - `sw $rt, offset_value($rs)`
    - opcode (6 bits) rs (5 bits) rt (5 bits) offset (16 bits)
  - They need to compute a memory address by adding the base register to the sign-extended 16-bit offset

# Load

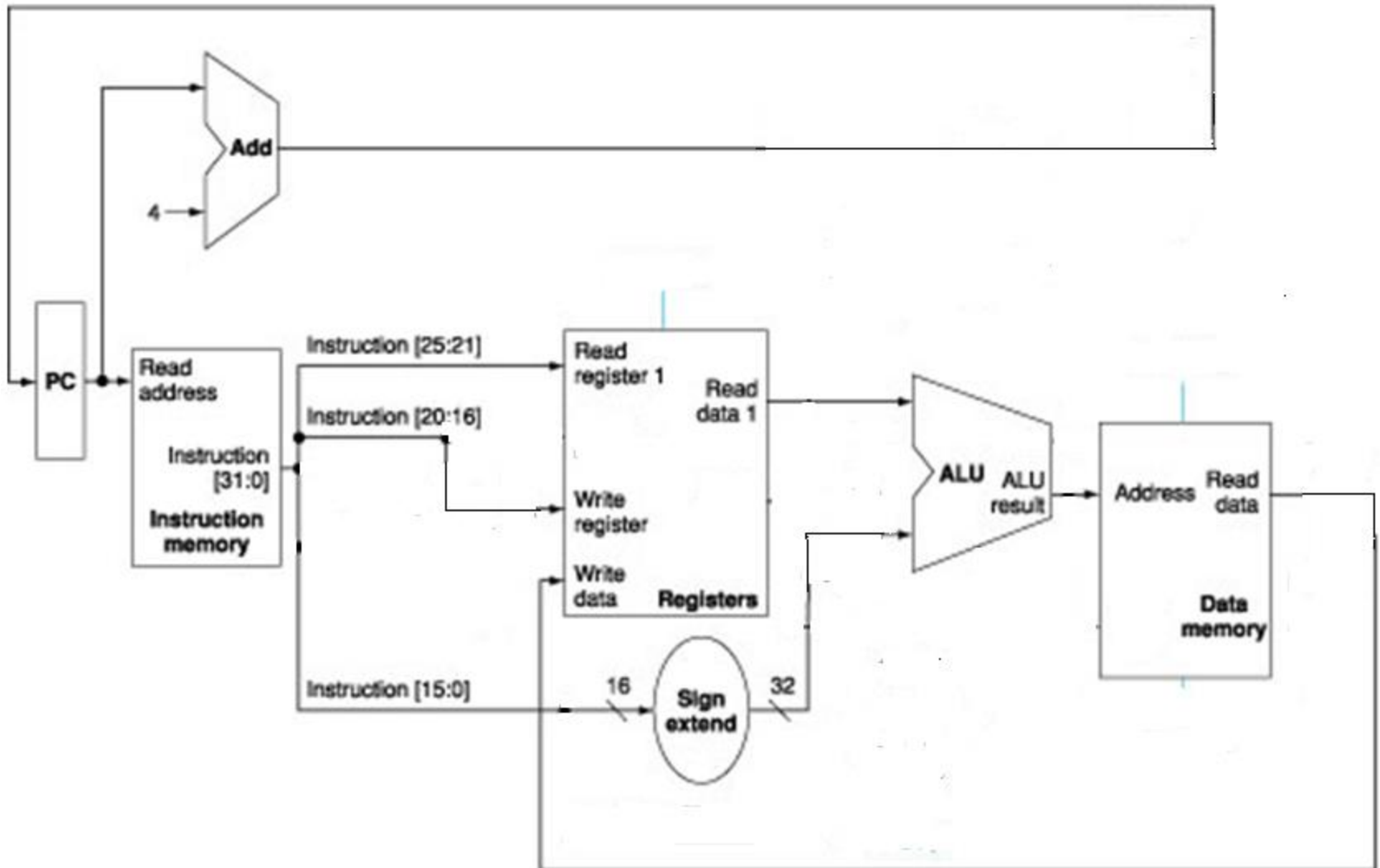
- For instruction `load $t0, 16($t1)`, what should the control signal be?



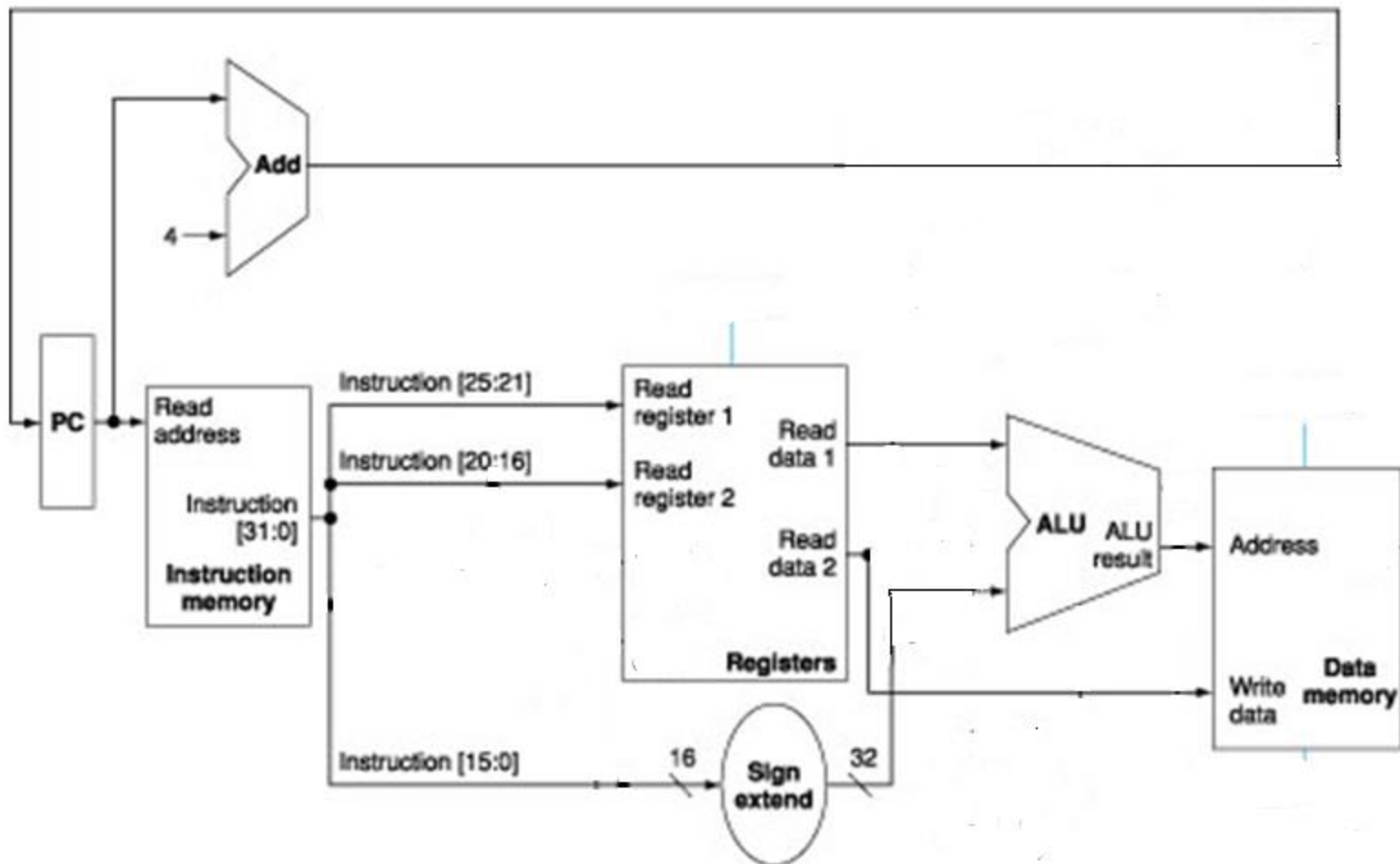
# Data path



# Data path only for lw (answer)



# Data path only for sw (answer)

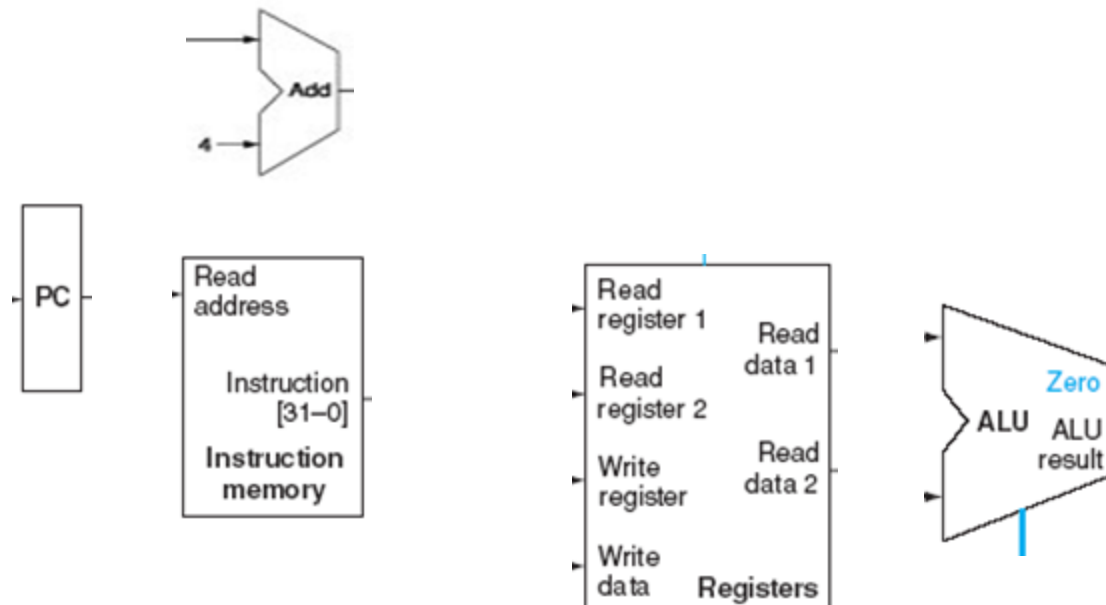


# Branch Instruction

- Beq has three operands, two registers that are compared for equality and a 16-bit offset used to compute the branch-target address
  - `beq $rs, $rt, offset`
    - opcode (6 bits) rs (5 bits) rt (5 bits) offset (16 bits)
  - Note that the 16-bit offset is given in terms of instructions, not bytes and is relative to PC + 4

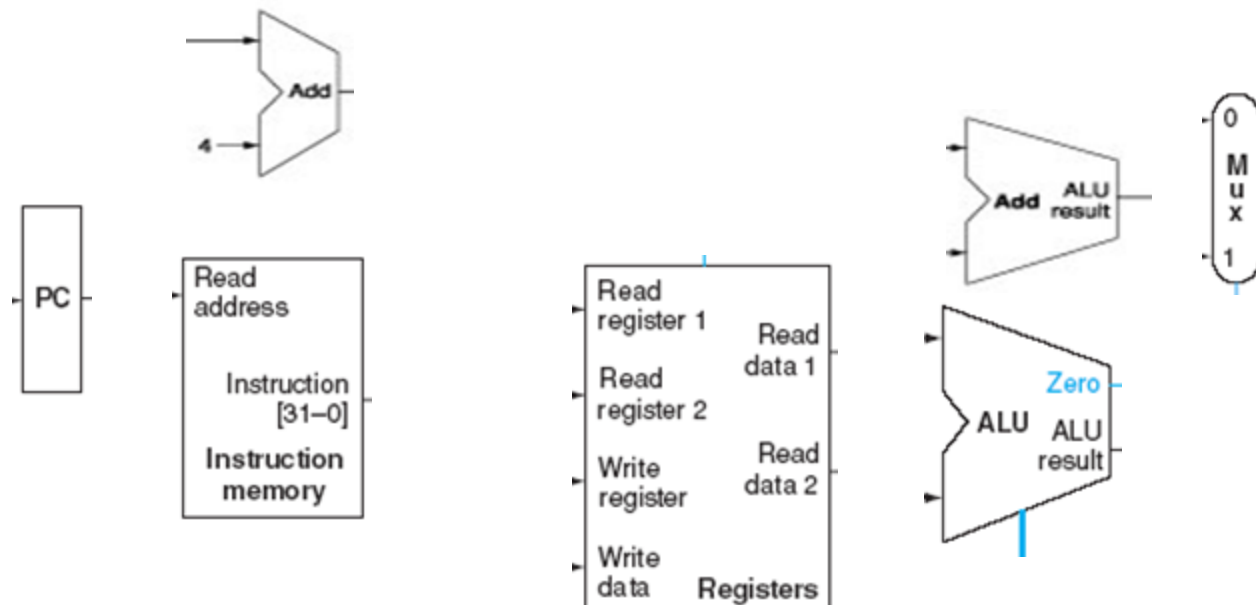
# Designing a processor only for beq

Are these enough? How many adders do we need? How to do the selection?

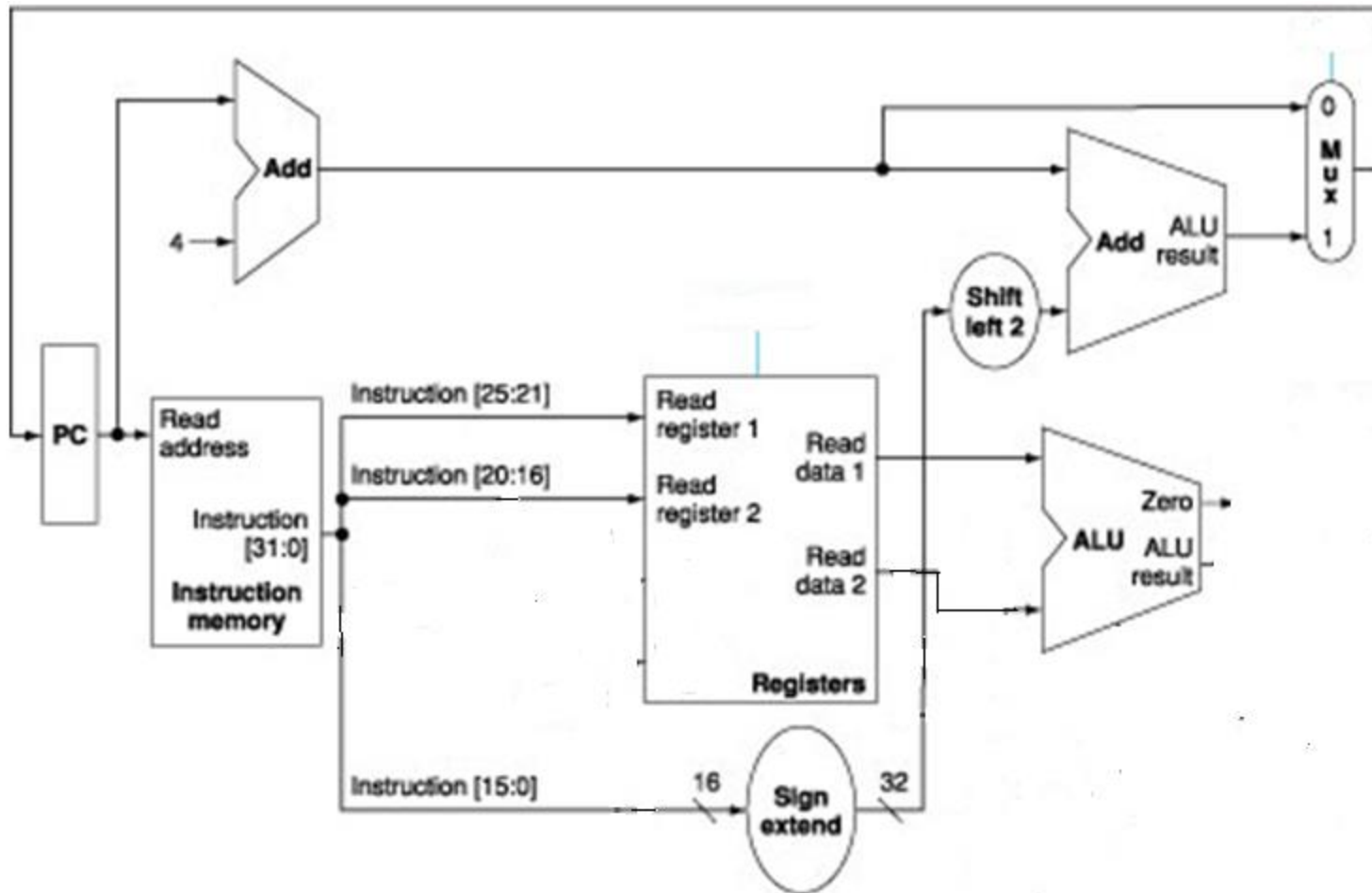




# Designing a processor only for beq



# Designing a processor only for beq (answer)



# In class exercise

- Design a simplified MIPS processor that supports **only** `addi`. Assume the control signals have been generated and only the data path needs to be designed.

`addi $rt, $rs, imm`

opcode (6 bits) rs (5 bits) rt (5 bits) imm (16 bits)

