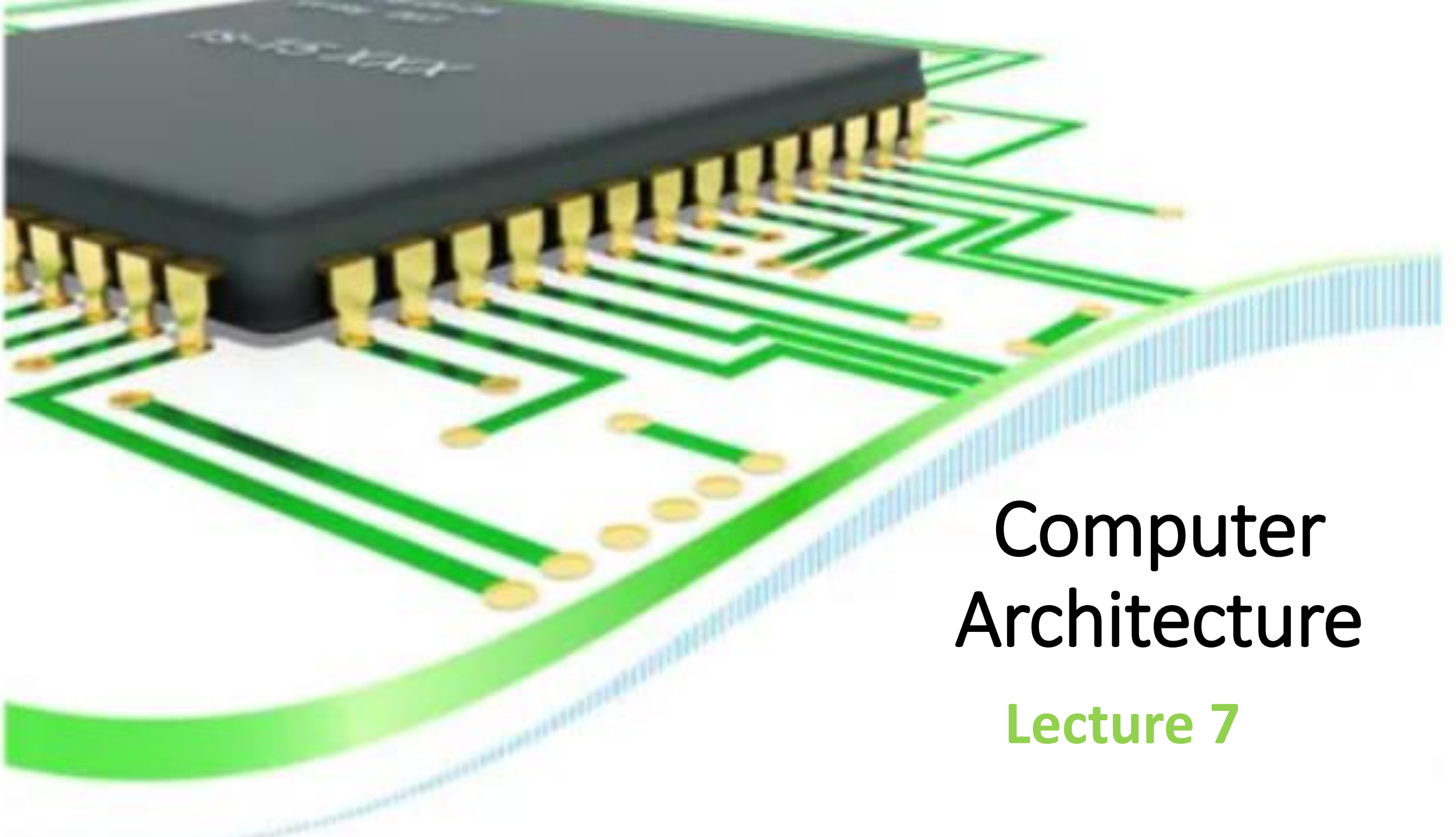


# Computer Architecture

## Lecture 7

# R-format instructions

| opcode | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

32 bits

$s0 → R16
$s1 → R17
$s2 → R18

**add $s2, $s0, $s1**

| | | | | | |
|---|---|---|---|---|---|
| *Representation in decimal →* | **0** | **16** | **17** | **18** | **0** | **32** |
| *Representation in binary →* | **000000** | **10000** | **10001** | **10010** | **00000** | **100000** |

# I-format instructions

| opcode | rs | rt | immediate |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

32 bits

$s0 → R16
$s1 → R17

**addi $s1, $s0, 20**

| | | | |
|---|---|---|---|
| *Representation in decimal →* | **8** | **16** | **17** | **20** |
| *Representation in binary →* | **001000** | **10000** | **10001** | **010100** |

# I-format instructions

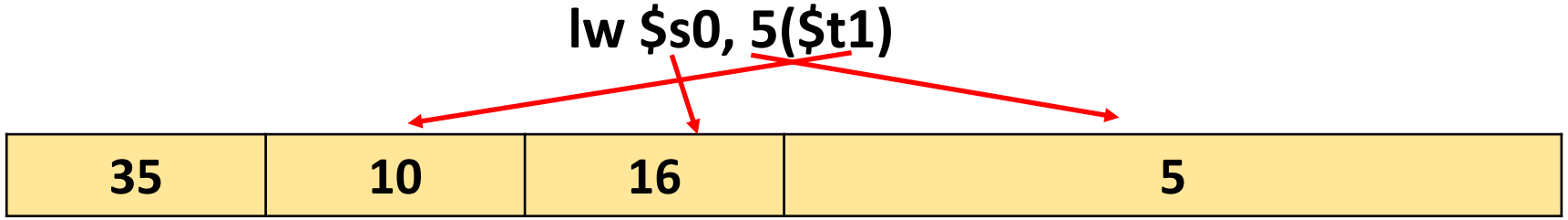
| opcode | rs | rt | immediate |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

32 bits

*N.B.: Branch, some load and store instructions are **I-format** instructions that have exceptions when translating them to binary → we will discuss that later*

# Load and store instructions

**I-format**

# Load and store instructions

**I-format**

| opcode | rs | rt | offset |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

32 bits

**sw $s1, 8($t1)**

| 43 | 10 | 17 | 8 |
|---|---|---|---|

# Branch instruction

3 instructions to the place of branch

```
li $s1, 5
li $s2, 3
beq $s1, $s2, label
add $t0, $t1, $t2
mul $t3, $t1, $t2
div $t3, $t1, $t2

label: sub $t0, $t1, $t2
```

**beq $s1, $s2, label**

| 5 | 16 | 17 | 3 |
|---|---|---|---|

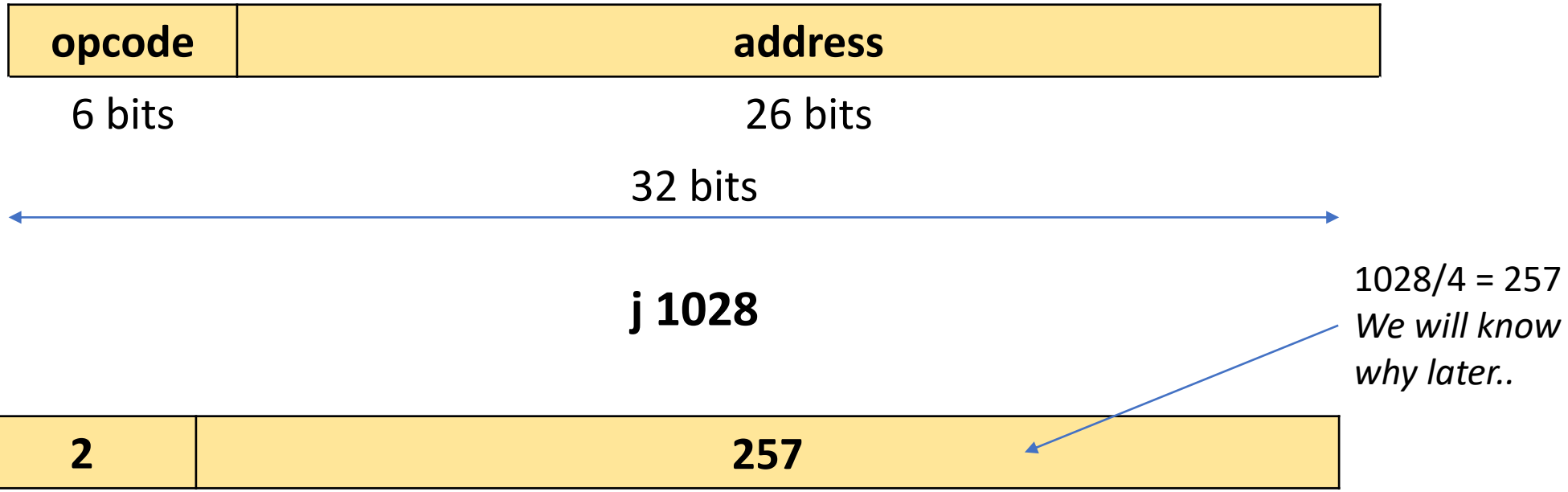
How to get the real address:
(PC+4) + (offset*4)

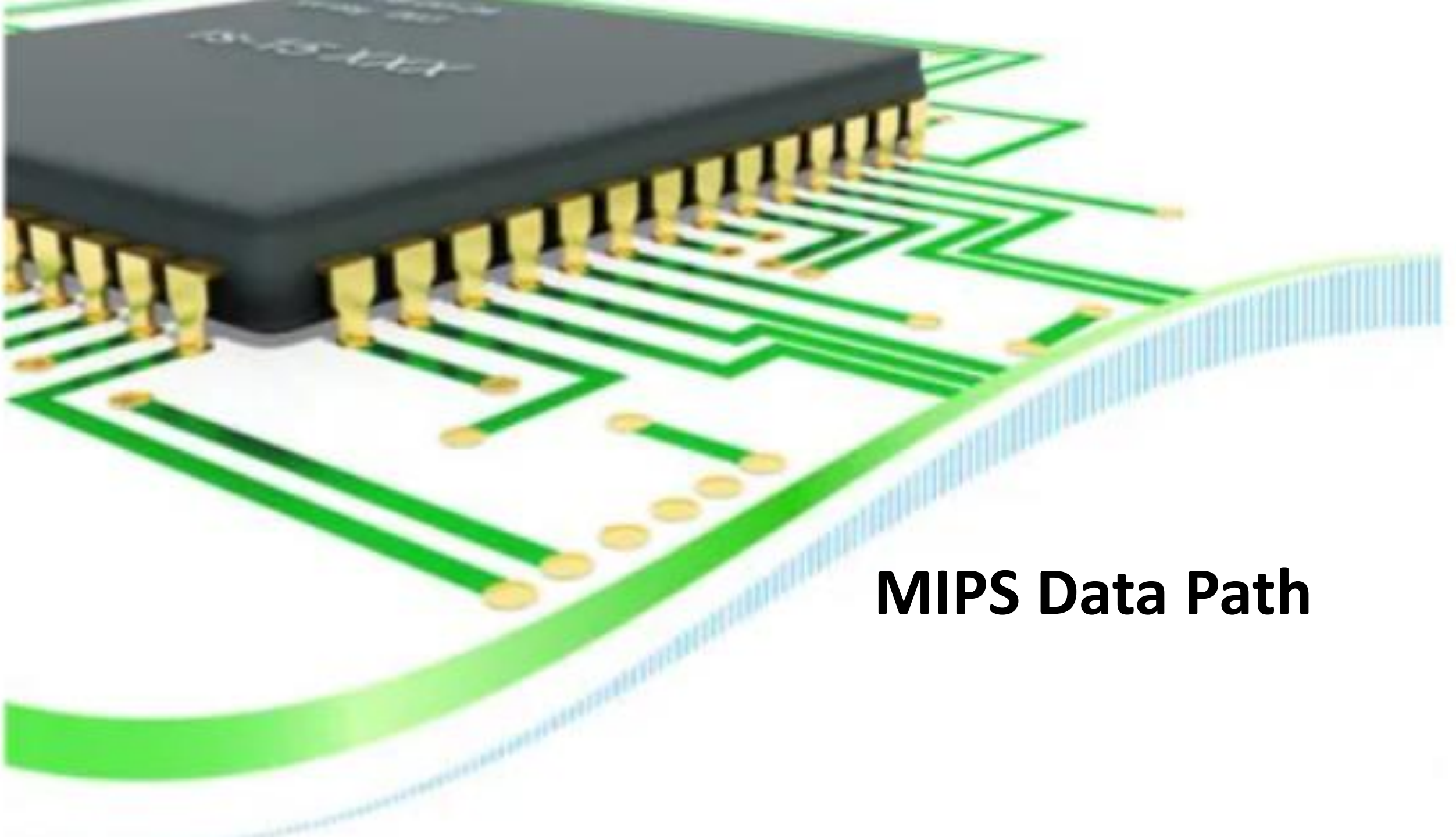
**I-format**

| opcode | rs | rt | offset |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

32 bits

# J-format instructions





# MIPS Data Path

# Datapath

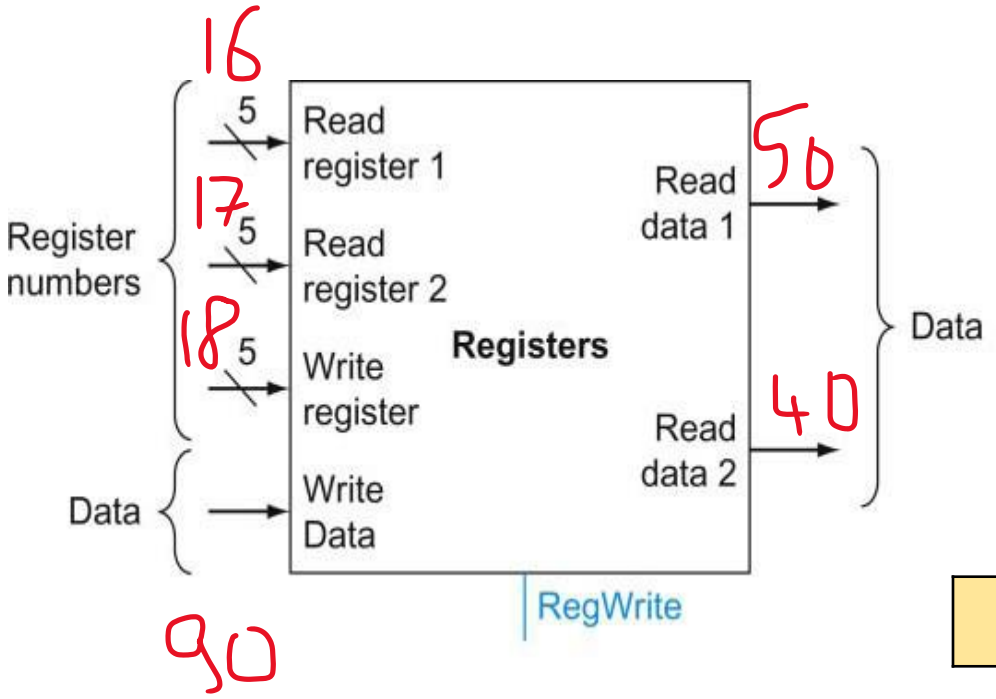
- A data path is the hardware composed of a collection of functional units *(elements)* that perform data processing operations, for example, ALU, registers, multiplexers, and internal buses.

# Datapath elements

- The datapath elements are the functional blocks within a microprocessor that actually interact to perform computational operations.

- These tasks include reading/writing to memory, arithmetic, logic operations, and numerical shift operations.

# Registers File




| opcode | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

32 bits

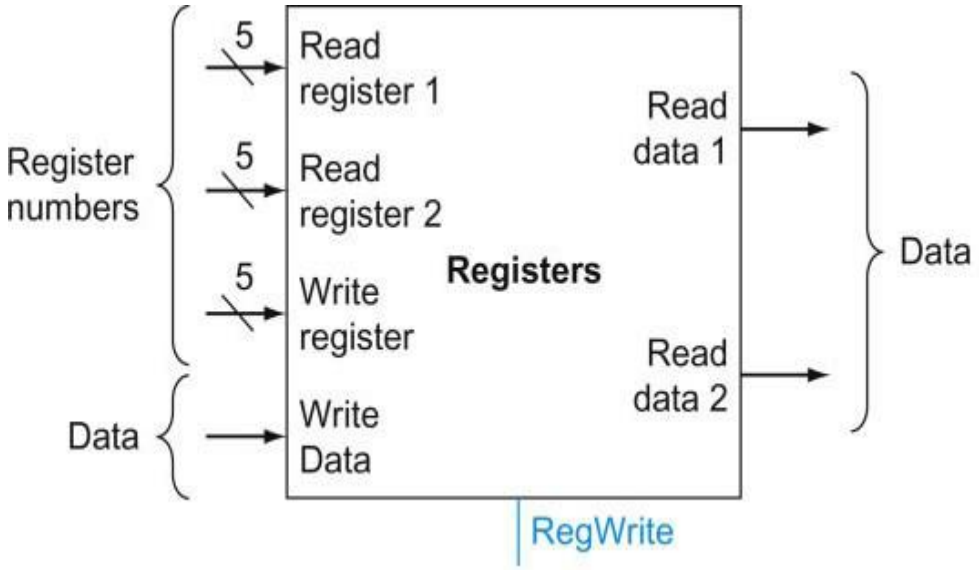
**add $s2, $s0, $s1**

| 0 | 16 | 17 | 18 | 0 | 32 |
|---|---|---|---|---|---|

| $s0 | $s1 | $s2 |
|---|---|---|
| 50 | 40 | 90 |

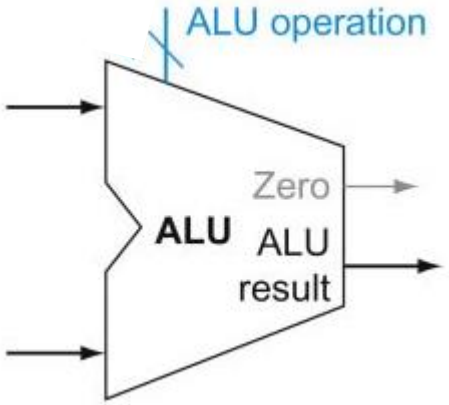

All inputs and outputs are in binary

# Registers File



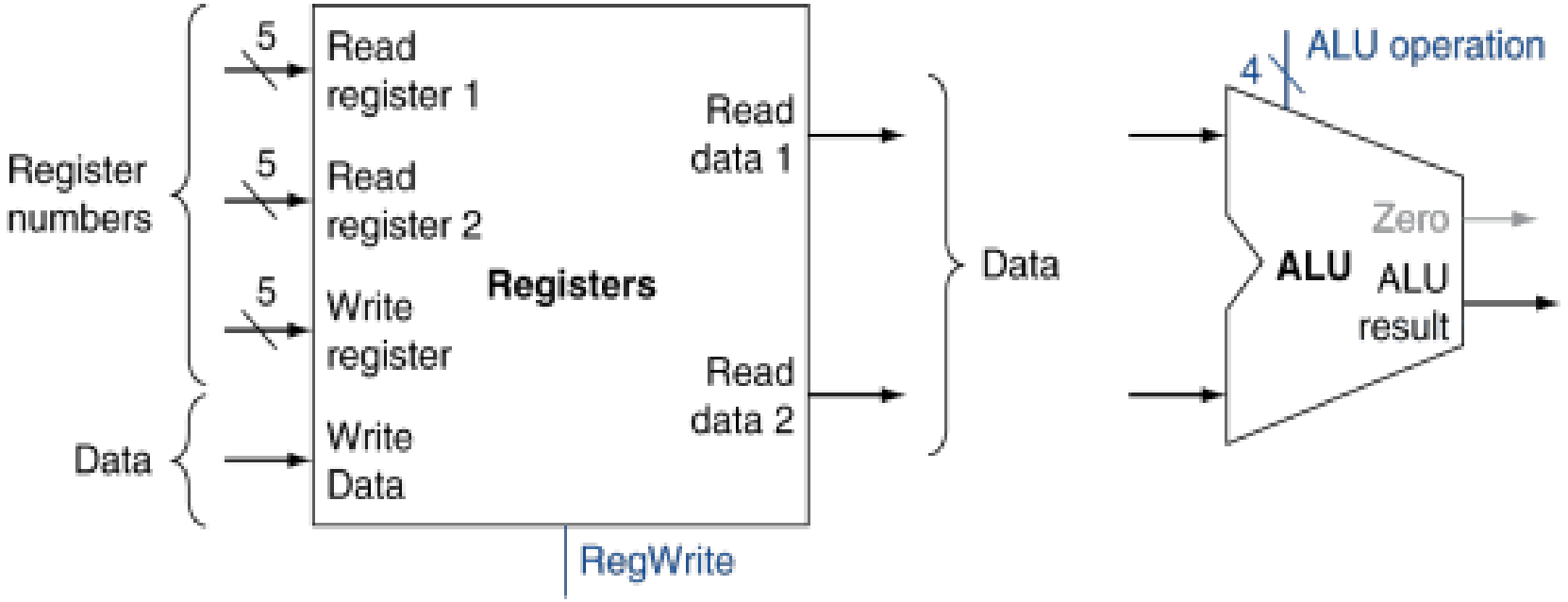

- The register file is a unit that contains the 32 general purpose registers.
- Any register can be read or written by specifying the register number.
- The register file has two 5-bit inputs that specify the numbers of the registers to be read.
- The register file can read and output the two registers at the same time.
- The register file also has a 5-bit input that specifies the number of the register to write.
- The register file can read and write during the same clock cycle.

# ALU

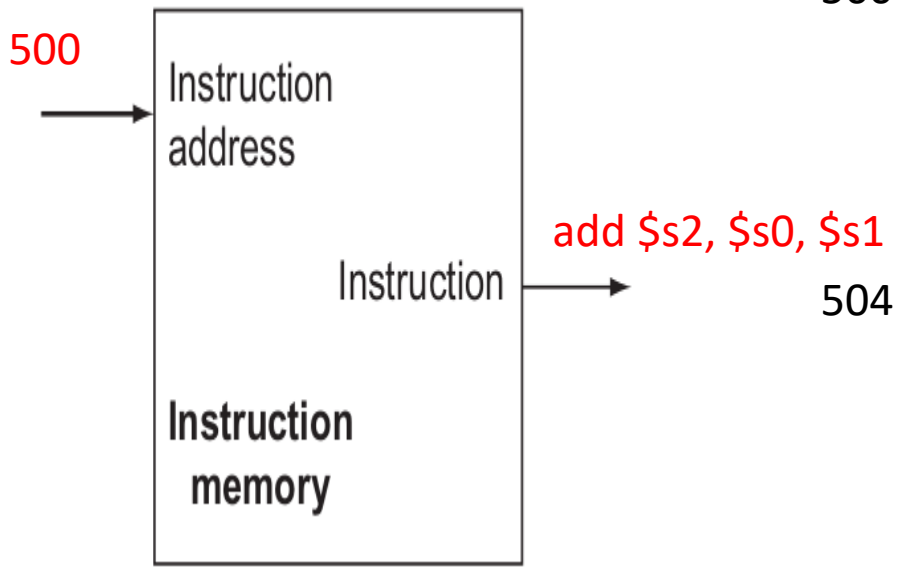


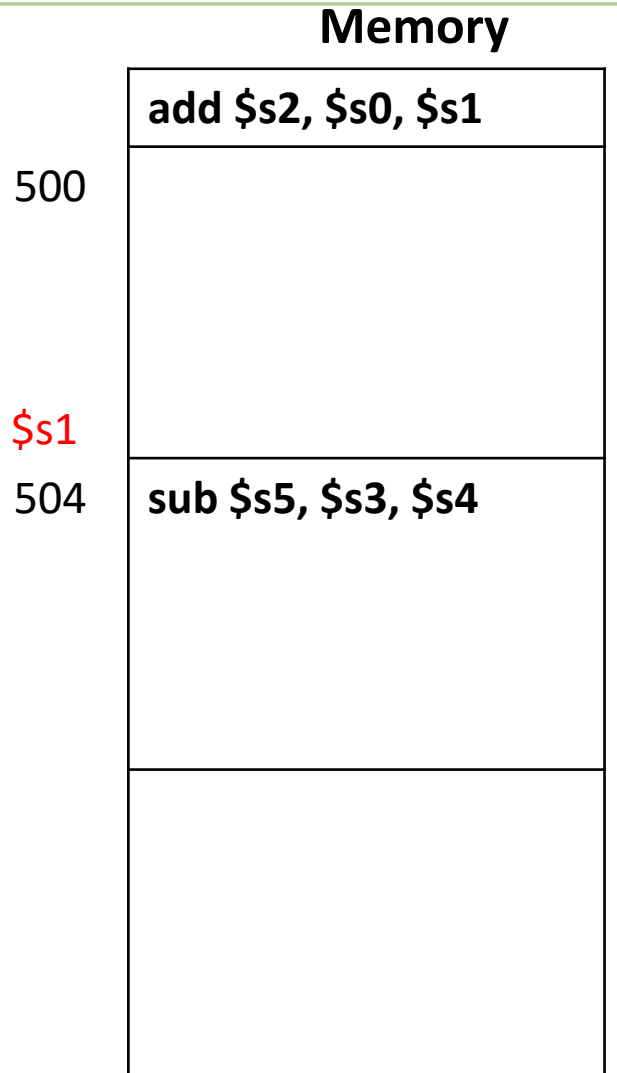

- ALU is an element that performs arithmetic and logical operations.
- It has two inputs: the data from the two registers and an output ALU result, which is the result of the operation.
- Another output is the ‘Zero’ which will be set to ‘1’ if the ALU result is zero and cleared to ‘0’ if the ALU result is any other value.
- ALU is also used to compute a memory address or perform a comparison for a branch.

# Registers file and ALU



# Instruction Memory



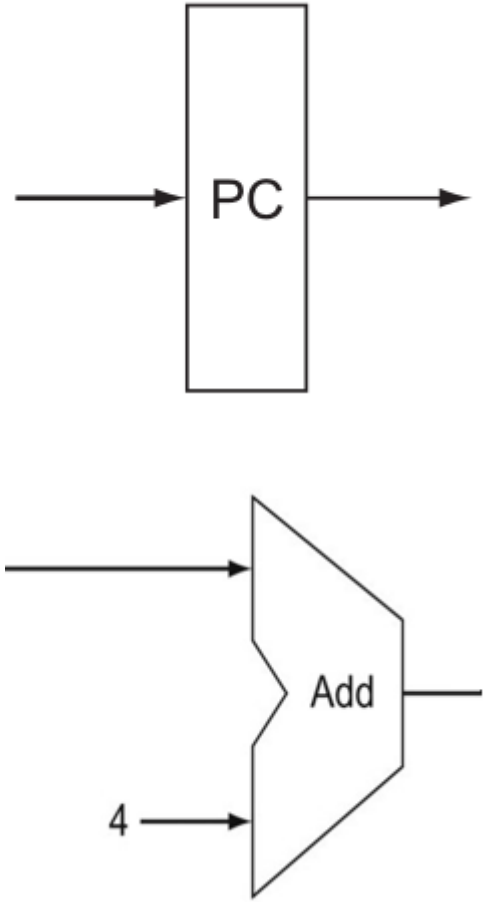




- Instruction memory is an element that provides read access to the instructions of a program.
- Given an address as input, it supplies the corresponding instruction at that address.

**Note: MIPS has separate instruction and data memory.**


All inputs and outputs are in binary

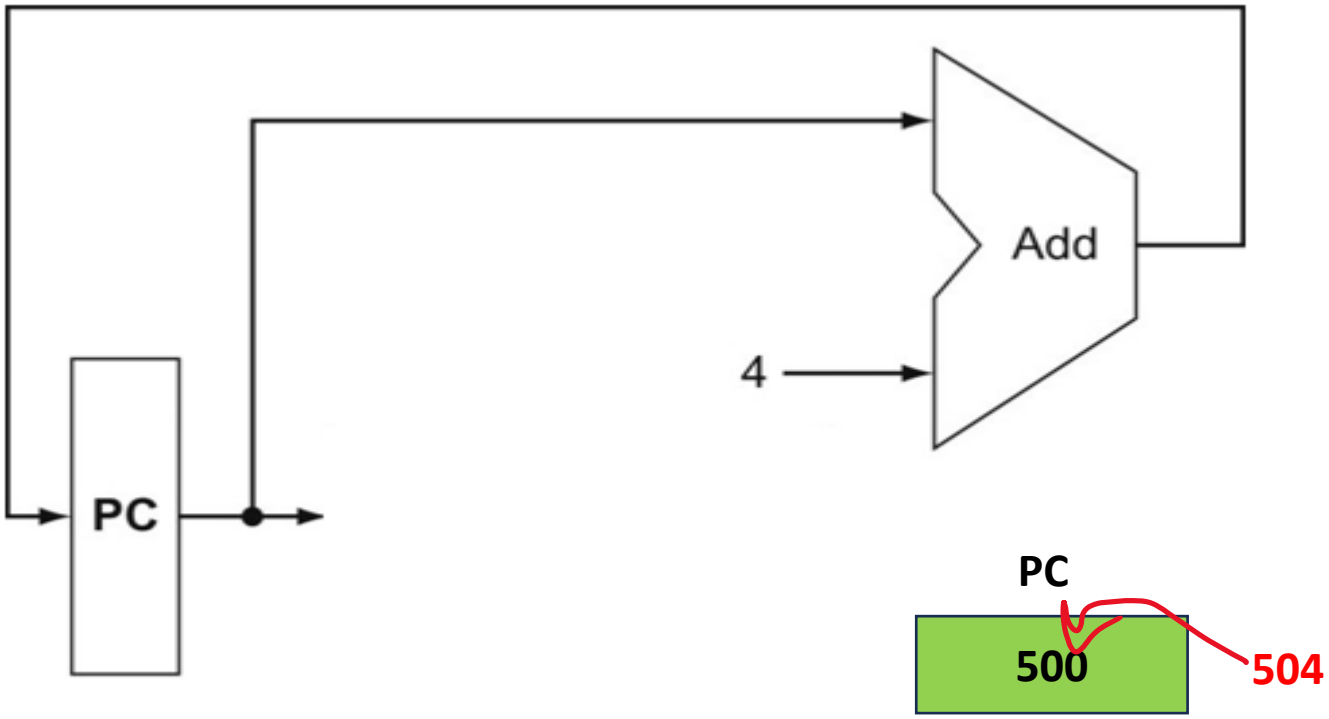
# Program Counter and an Adder




- The PC is an element that holds the address of the current instruction.
- It is just a 32-bit register which holds the instruction address.
- After reading the instruction from the memory, PC counter is incremented by 4 to point to the next instruction.
- The increment by 4 as the next instruction is after 4 cells in the byte-addressable memory.

- An adder is an element that adds value four to an input.

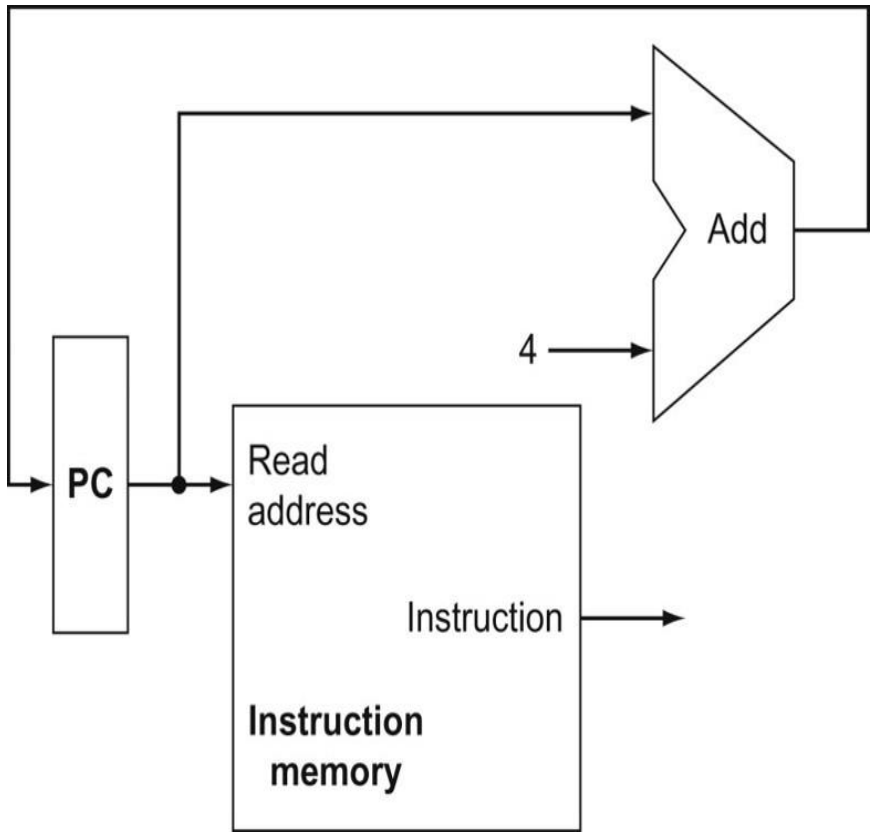
# Program Counter and an Adder




**Memory**

| | |
|---|---|
| 500 | **add $s2, $s0, $s1** |
| | |
| 504 | **sub $s5, $s3, $s4** |
| | |

# Program Counter, an Adder, and Instruction Memory

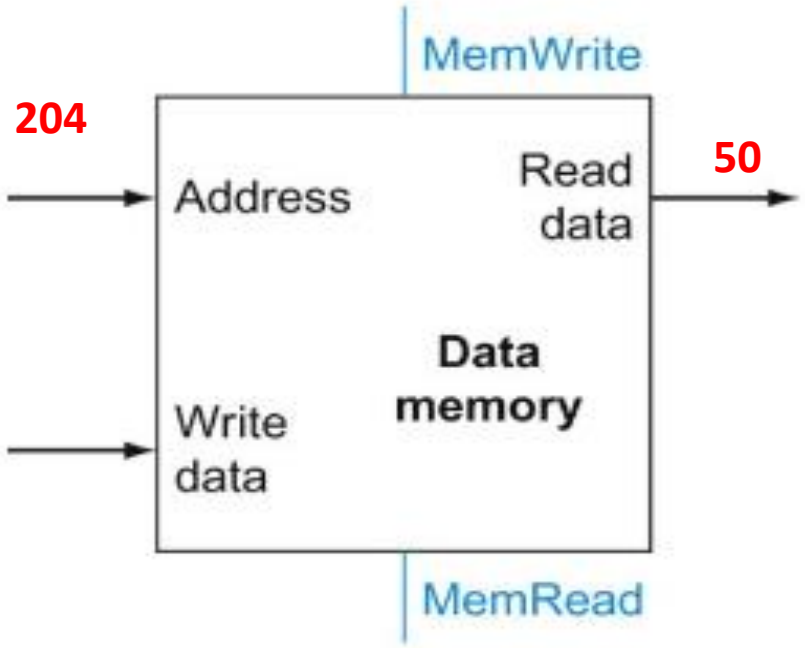



•The address from the PC goes to the instruction memory, which returns the instruction.

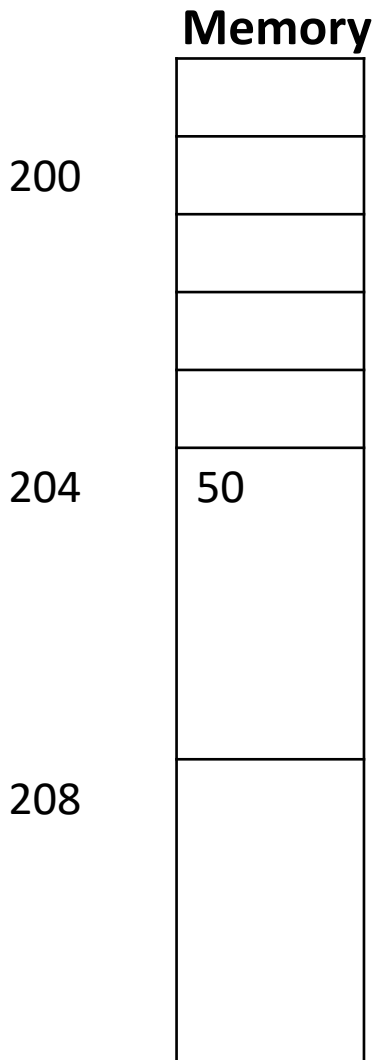
•The PC goes to an adder which adds 4 to get the next value of the PC.

•Later, we will that there is a separate adder dedicated to updating the PC.

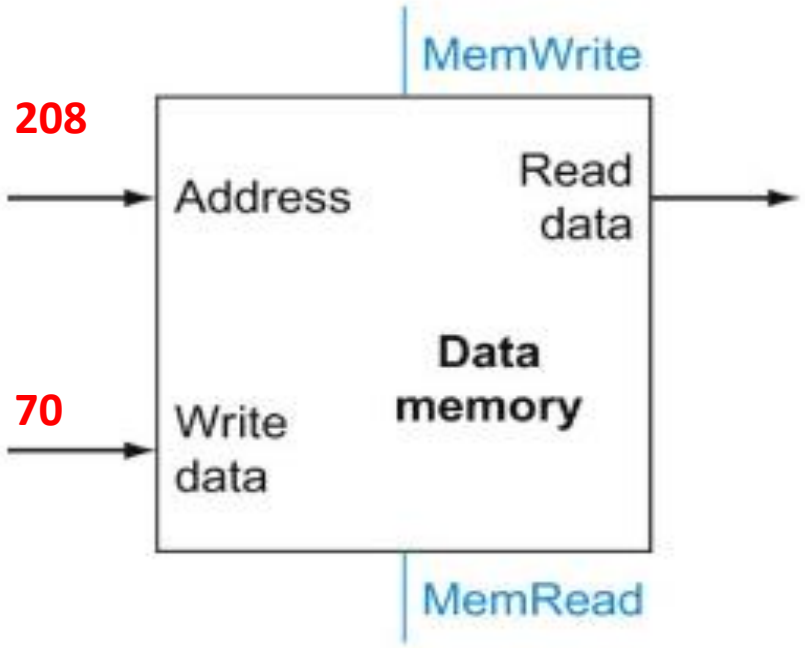
# Data Memory Unit



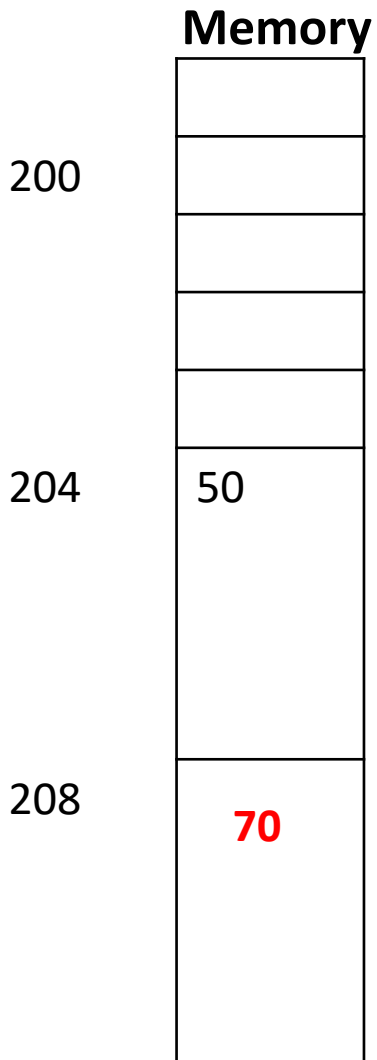

**Reading from address 204**



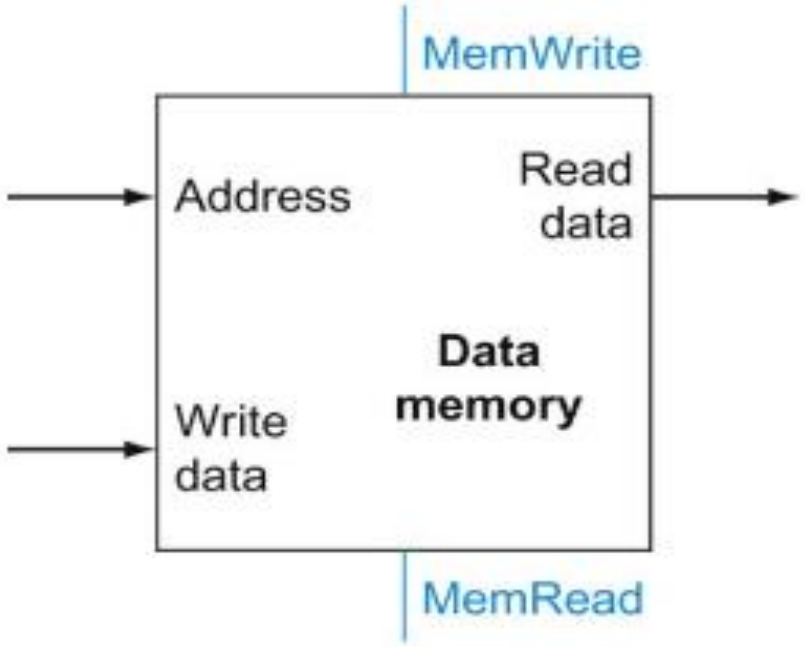
# Data Memory Unit




**Writing 70 to address 208**



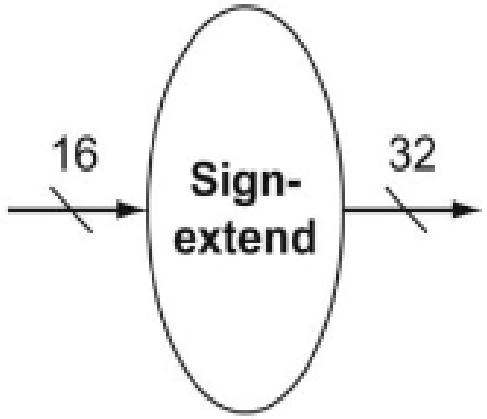
# Data Memory Unit




- The data memory stores ALU results.
- The data memory accepts a address and either accepts data through WriteData port if the instruction is writing in memory or outputs data if the instruction is reading from memory through WriteData port at the indicated address.

**Note: Separate instruction and data memory.**

# Sign Extender

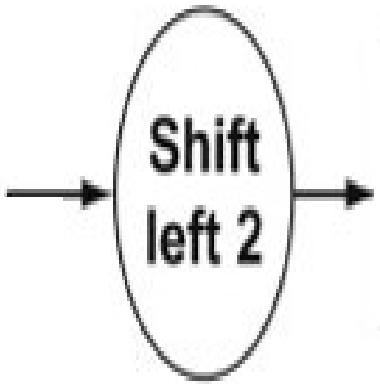



- The sign extender adds 16 leading digits to a 16-bit word with most significant bit *b*, to output the same value in a 32-bit word.
- The additional 16 digits have the same value as *b*.

0000000000000000 0000000000011010

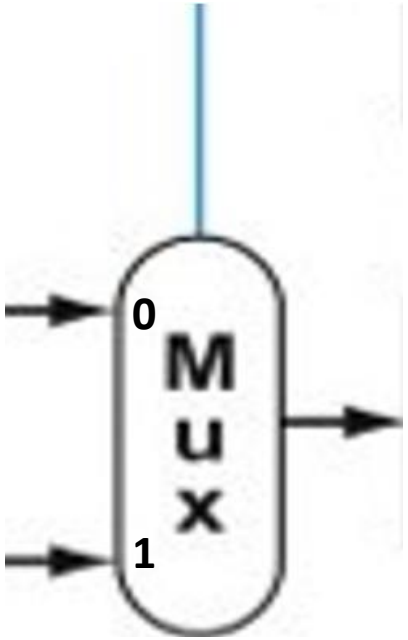
1111111111111111 1111111111101100

# Shift left 2




- The “Shift left 2” element, performs a multiplication by 4 to the input value.

# 2x1 Multiplexers

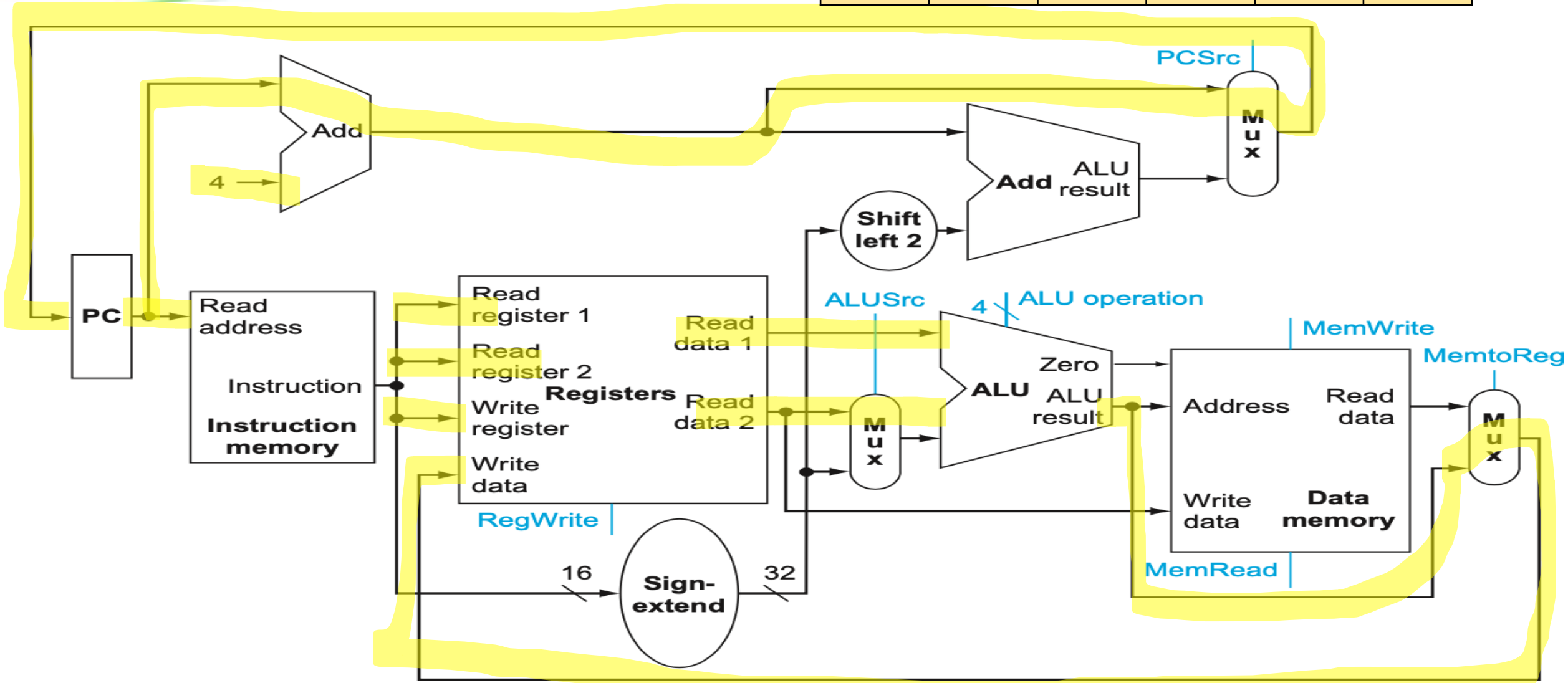



- In the Datapath, we will need 2x1 multiplexers
- Each 2x1 multiplexer has one selection line.

# Scenario for R-format instructions

**add $s2, $s0, $s1**

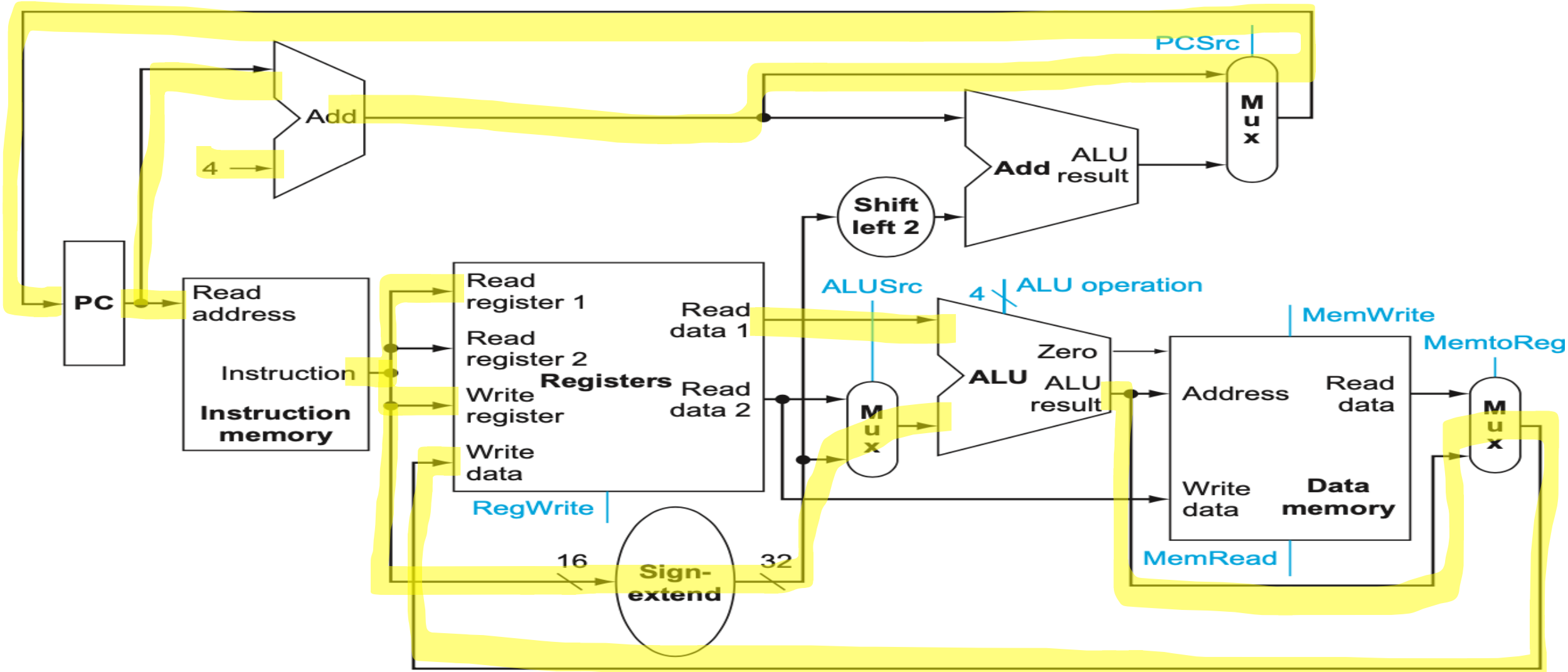
| 0 | 16 | 17 | 18 | 0 | 32 |
|---|---|---|---|---|---|



# Scenario for I-format instructions

**addi $s1, $s0, 20**

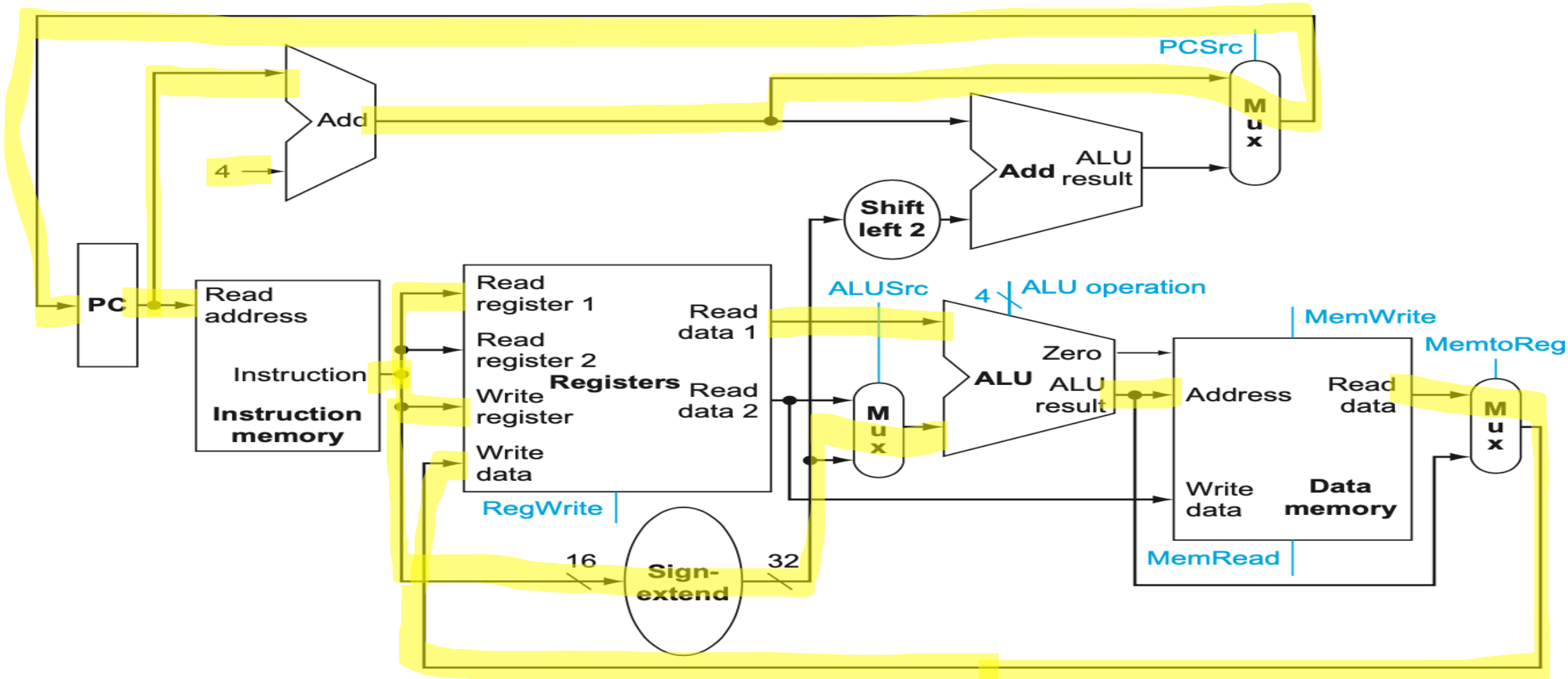
| 8 | 16 | 17 | 20 |
|---|---|---|---|



# Scenario for I-format instructions

**lw $s0, 5($t1)**

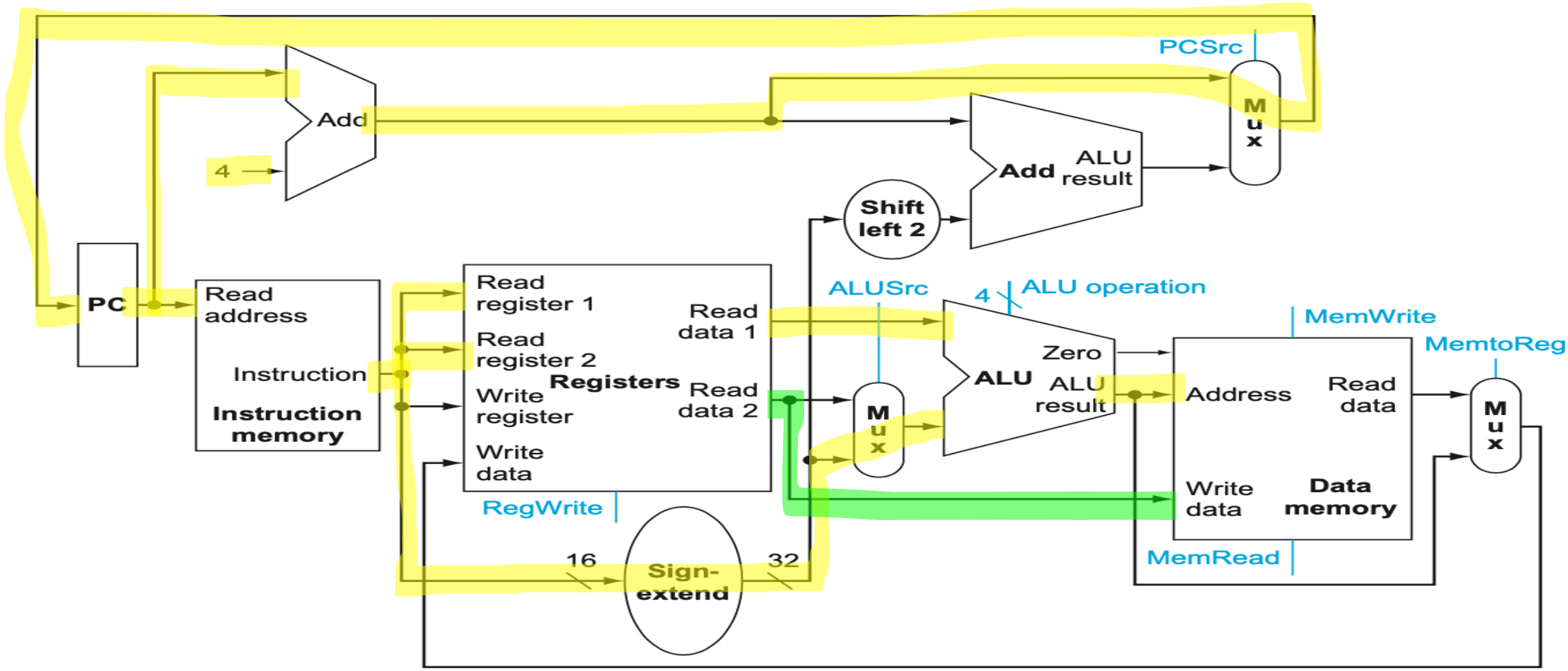
| 35 | 10 | 16 | 5 |
|---|---|---|---|



# Scenario for I-format instructions

**sw $s1, 8($t1)**

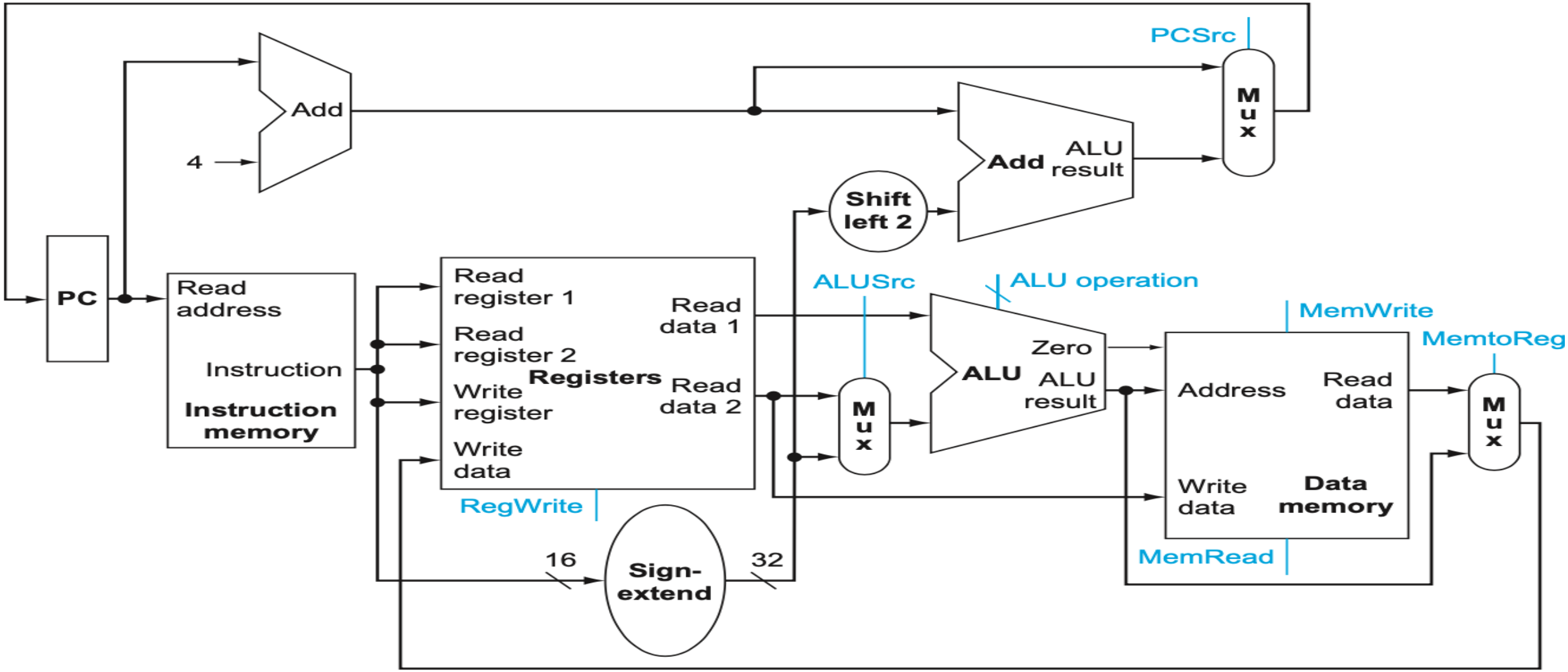
| 43 | 10 | 17 | 8 |
|---|---|---|---|



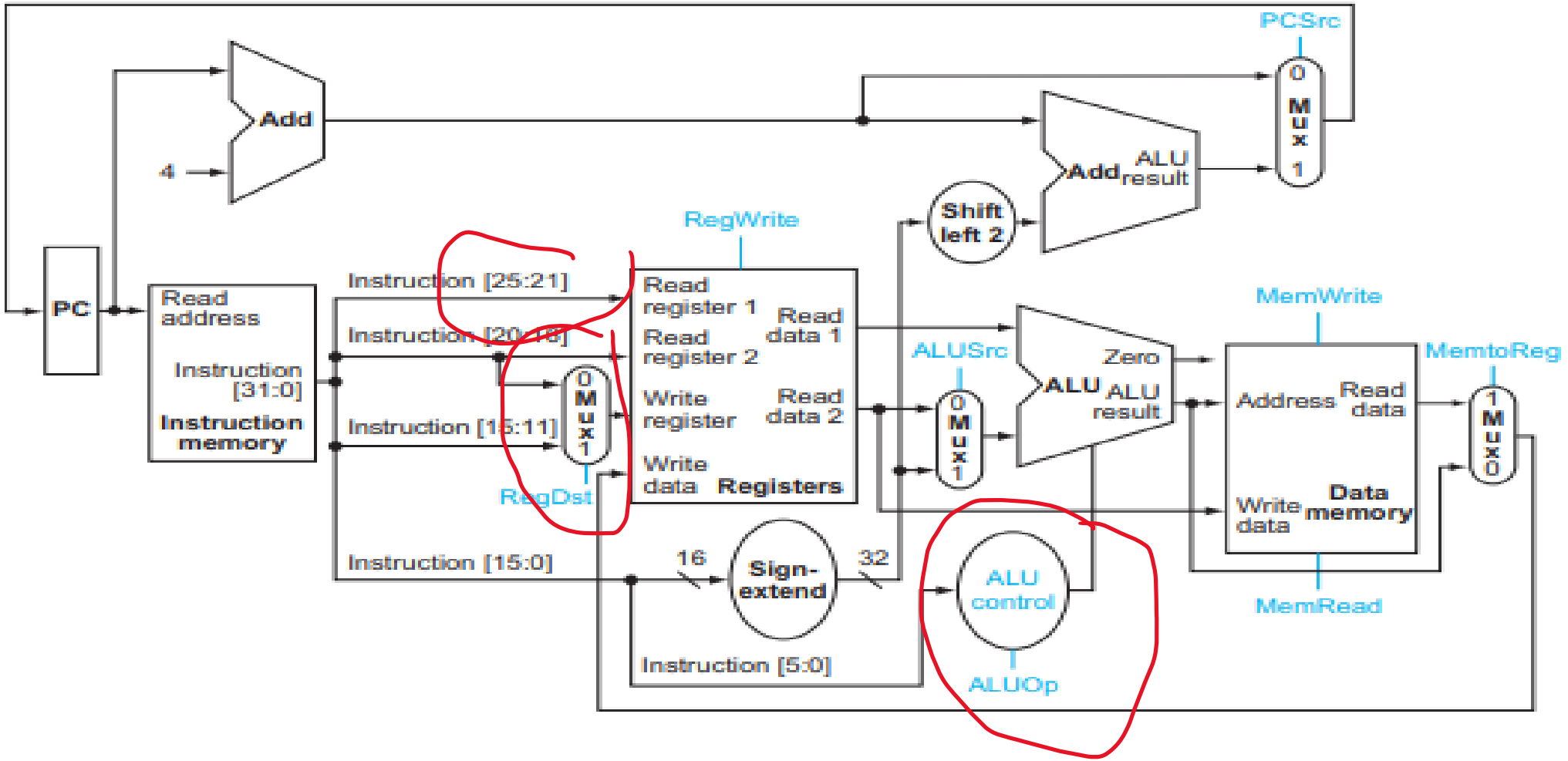
So why do we need MUXs?

The datapath is dealing with different formats and various options

# Datapath with some control lines



# Datapath with some control lines

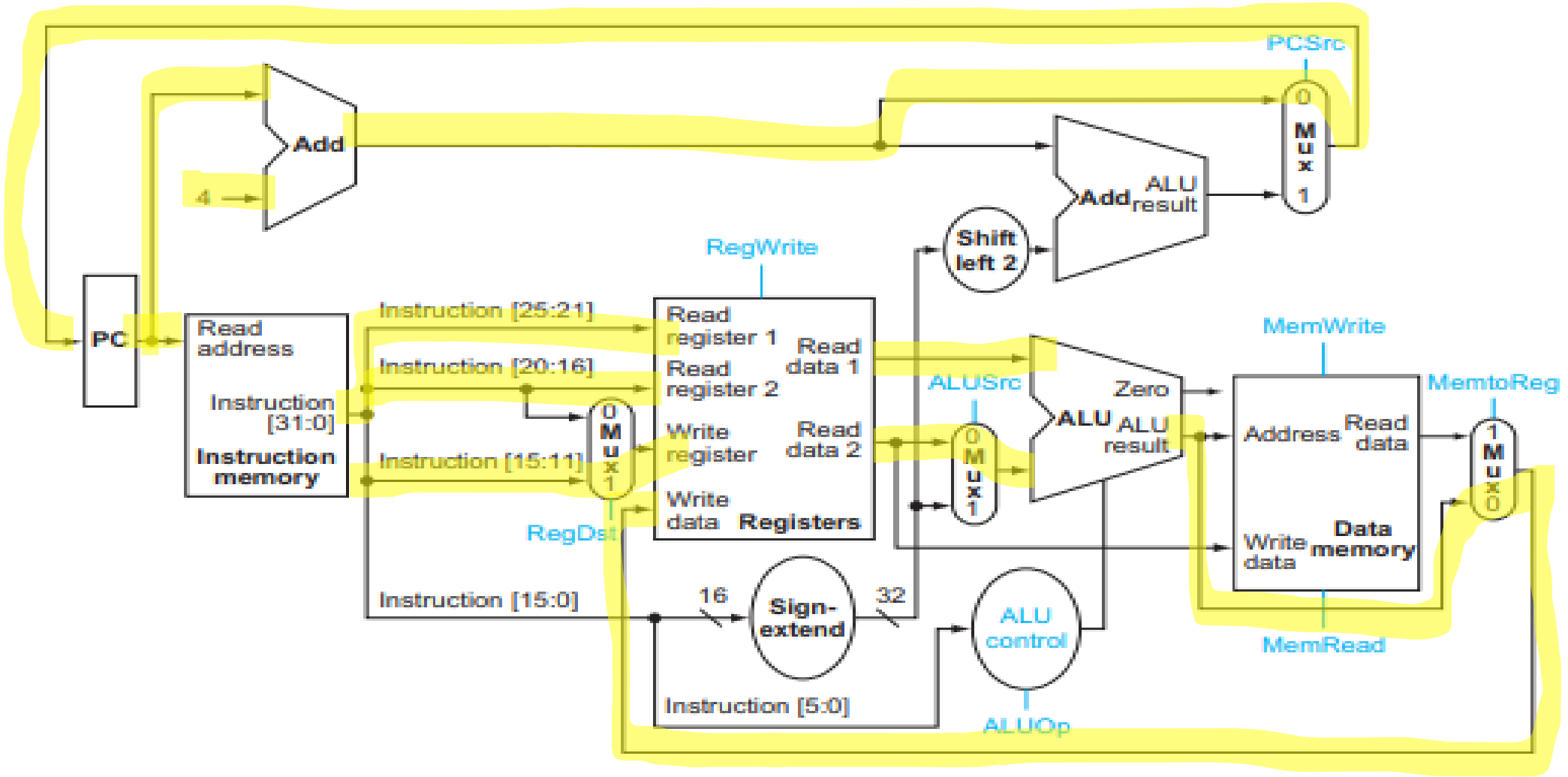


# R-format instructions

**add $s2, $s0, $s1**

| 0 | 16 | 17 | 18 | 0 | 32 |
|---|---|---|---|---|---|
| bits 31 –26 | bits 25--21 | bits 20-16 | bits 15--11 | bits 10--6 | bits 5—0 |
| opcode | Reading Register number | Reading register number | Writing register number | shamt | funct |

# Scenario for R-format instructions



# Control Signal Table

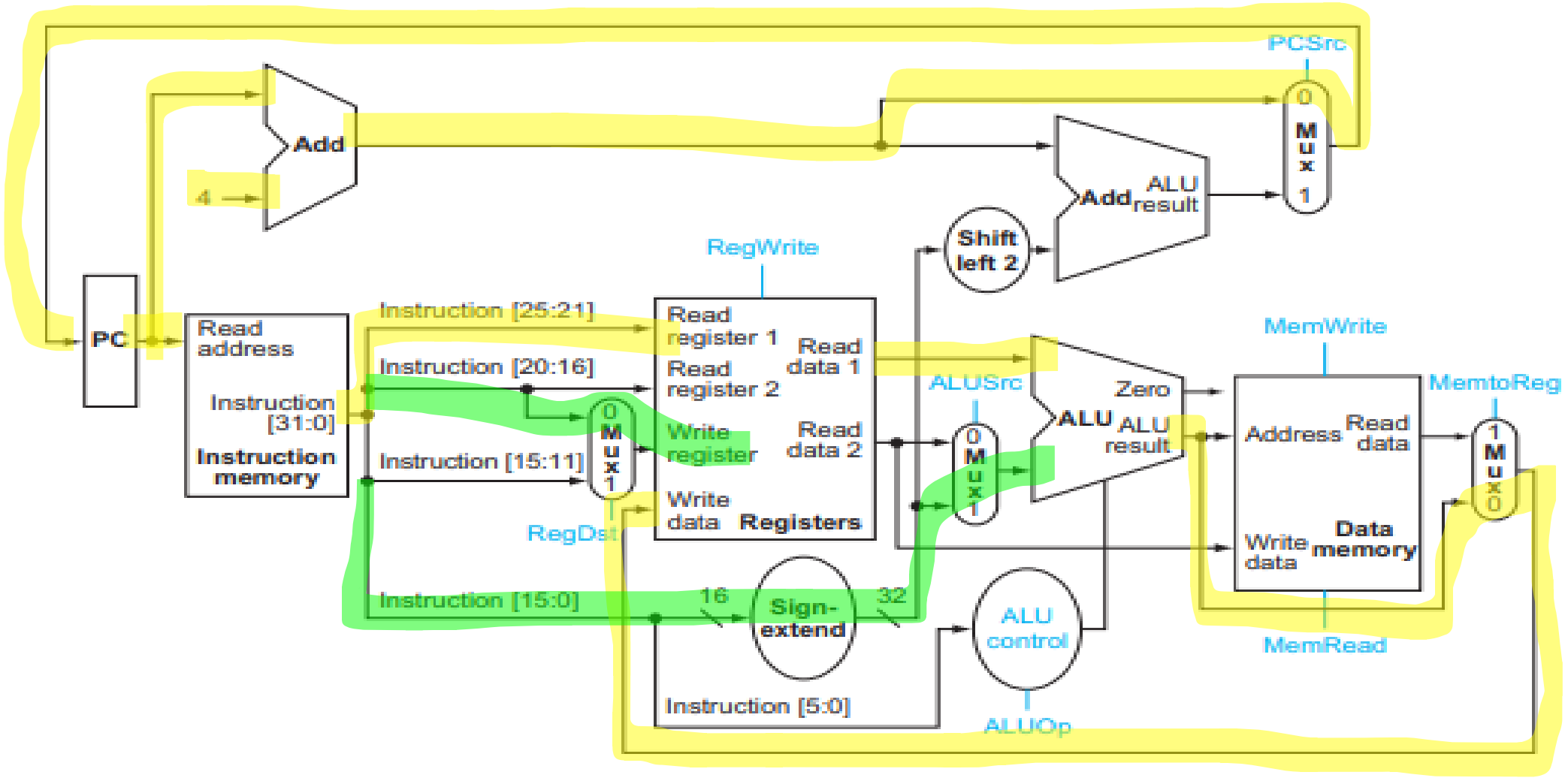
| instruction | RegDst | RegWrite | ALUSrc | MemWrite | MemRead | MemToReg | PCSrc |
|---|---|---|---|---|---|---|---|
| add | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| sub | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| and | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| or | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

# I-format instructions

**addi $s1, $s0, 20**

| 0 | 16 | 17 | 20 |
|---|---|---|---|
| bits 31 –26 | bits 25--21 | bits 20-16 | bits 15--0 |
| opcode | Reading Register number | Writing register number | immediate |

# Scenario for addi instruction



# Control Signal Table

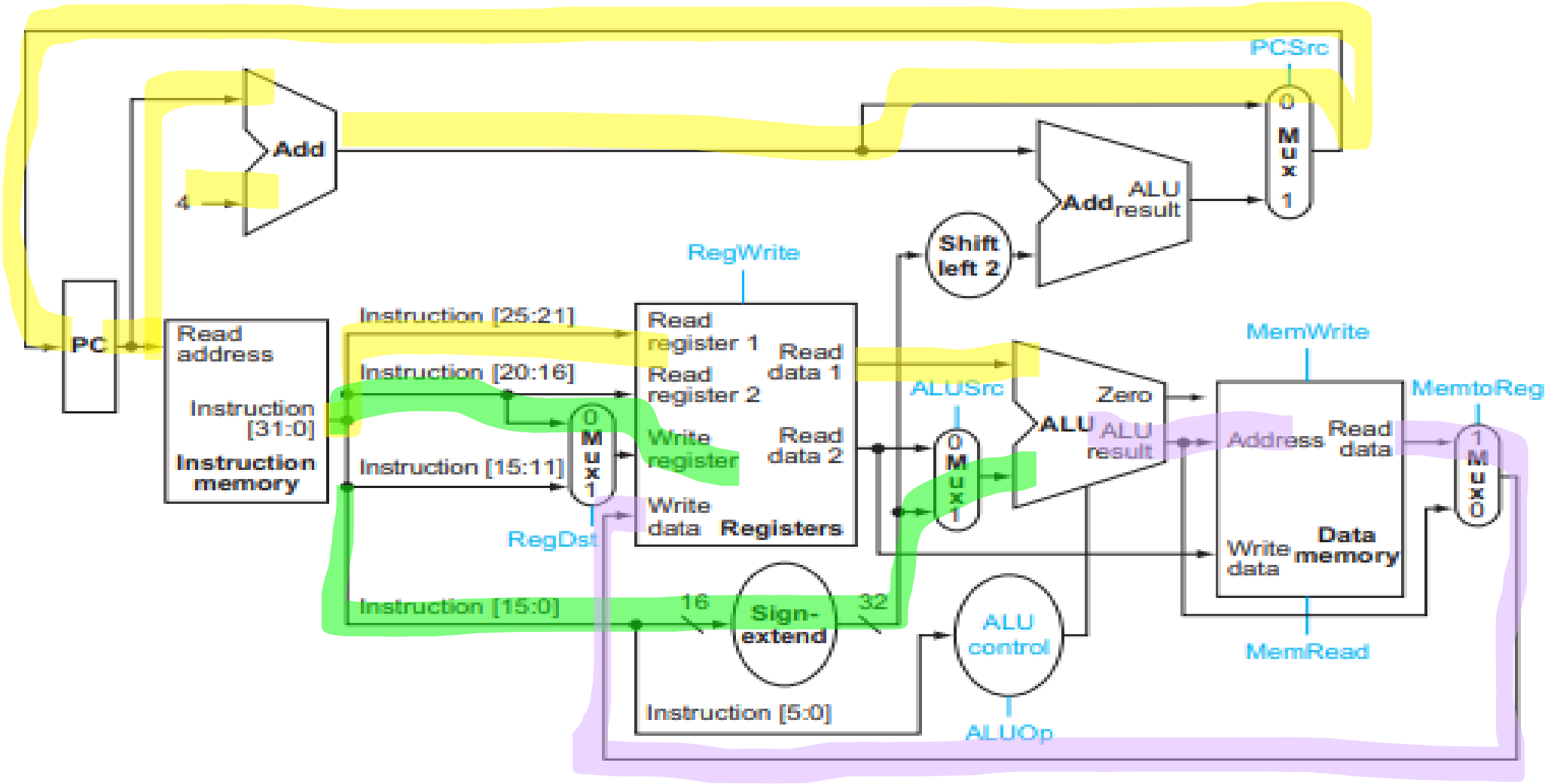
| instruction | RegDst | RegWrite | ALUSrc | MemWrite | MemRead | MemToReg | PCSrc |
|---|---|---|---|---|---|---|---|
| add | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| sub | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| and | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| or | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| addi | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

# I-format instructions

lw $s0, 5($t1)

| 35 | 10 | 16 | 5 |
|---|---|---|---|
| bits 31 –26 | bits 25--21 | bits 20-16 | bits 15--0 |
| opcode | Reading Register number | Writing register number | offset |

# Scenario for lw instruction



# Control Signal Table

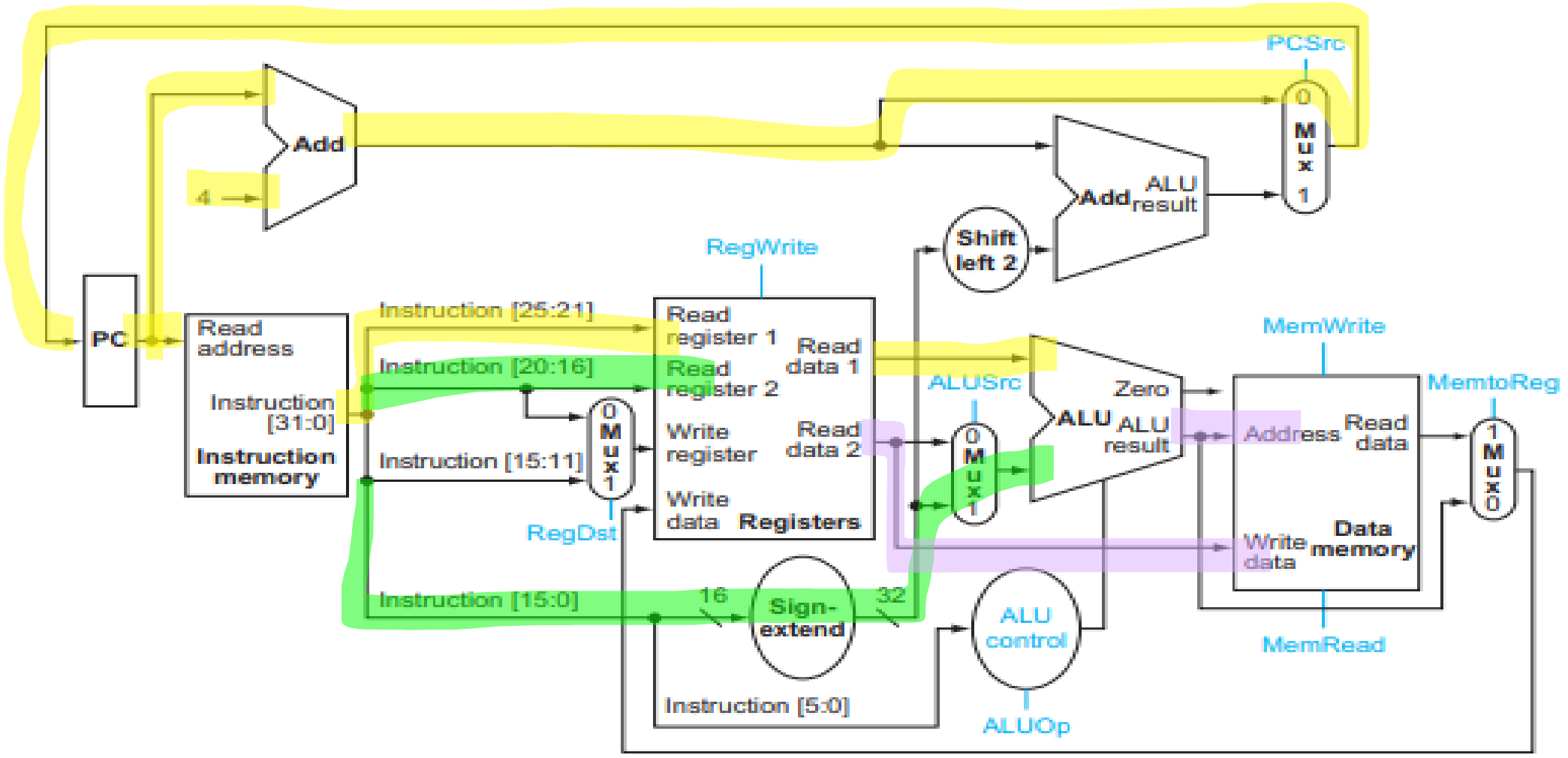
| instruction | RegDst | RegWrite | ALUSrc | MemWrite | MemRead | MemToReg | PCSrc |
|---|---|---|---|---|---|---|---|
| add | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| sub | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| and | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| or | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| addi | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| lw | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

# I-format instructions

sw $s1, 8($t1)

| 43 | 10 | 17 | 8 |
|---|---|---|---|
| bits 31 –26 | bits 25--21 | bits 20-16 | bits 15--0 |
| opcode | Reading Register number | Reading register number | offset |

# Scenario for sw instruction



# Control Signal Table

| instruction | RegDst | RegWrite | ALUSrc | MemWrite | MemRead | MemToReg | PCSrc |
|---|---|---|---|---|---|---|---|
| add | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| sub | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| and | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| or | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| addi | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| lw | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| sw | x | 0 | 1 | 1 | 0 | x | 0 |

# BEQ instruction

3 instructions to the place of branch

```
li $s1, 5
li $s2, 3
beq $s1, $s2, label
add $t0, $t1, $t2
mul $t3, $t1, $t2
div $t3, $t1, $t2

label: sub $t0, $t1, $t2
```

**beq $s1, $s2, label**

| 5 | 17 | 18 | 3 |
|---|---|---|---|
| | | | |

How to get the real address:
(PC+4) + (offset*4)

**I-format**

| opcode | rs | rt | offset |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

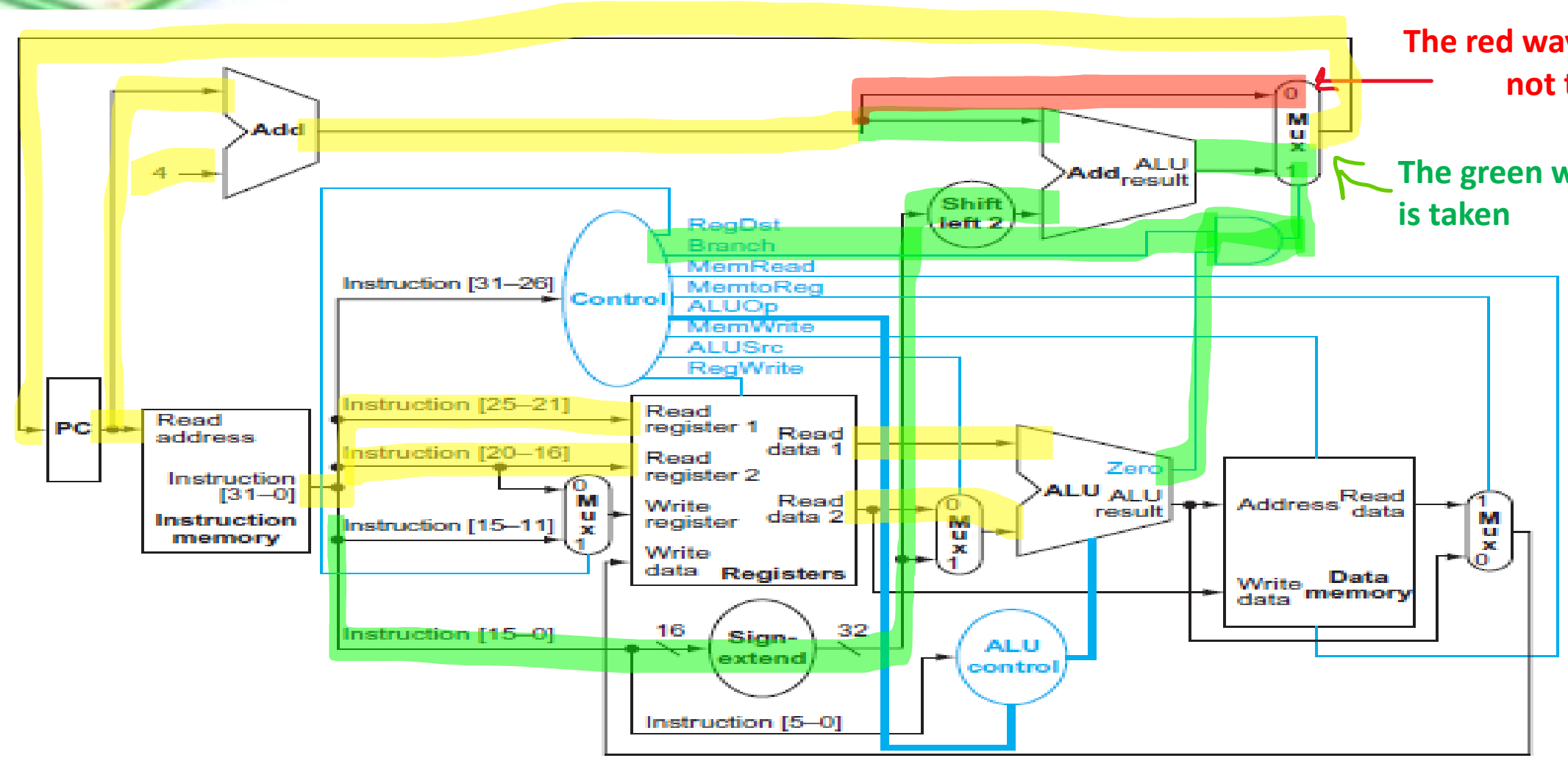
32 bits

# I-format instructions

**beq $s1, $s2, 3**

| 5 | 17 | 18 | 3 |
|---|---|---|---|
| **bits 31 –26** | **bits 25--21** | **bits 20-16** | **bits 15--0** |
| **opcode** | **Reading Register number** | **Reading register number** | **offset**<br>How to get the real address:<br>(PC+4) + (offset*4) |

# Scenario for beq instruction



# Control Signal Table

| instruction | RegDst | RegWrite | ALUSrc | MemWrite | MemRead | MemToReg | PCSrc |
|---|---|---|---|---|---|---|---|
| add | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| sub | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| and | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| or | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| addi | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| lw | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| sw | x | 0 | 1 | 1 | 0 | x | 0 |
| beq | x | 0 | 0 | 0 | 0 | x | 1 |

RegDst = 0 is also correct

PCSrc = 1 if Branch =1
PCSr =0 if Branch =0

# Problem 1

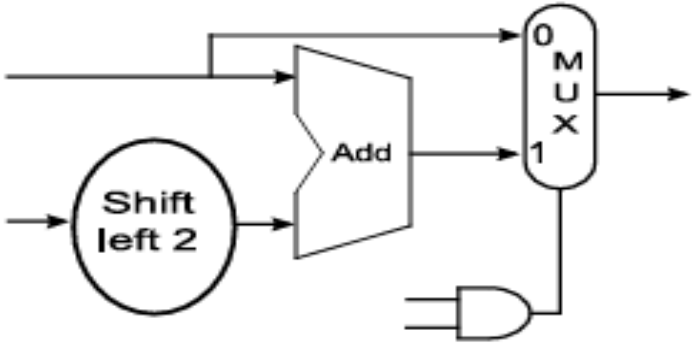
- In order to correctly execute the beq instruction, the control signals Zero and Branch were added to the design of the datapath.
- Explain why neither of those control signals would be sufficient by itself.

**Solution:**

- The Zero signal indicates whether the result computed by the ALU equals zero; that has nothing to do with affecting the Program Counter.
- The Branch signal indicates whether the current instruction is beq, but nothing about whether the two registers are equal.
- The branch to the target instruction if the current is beq AND the two registers involved are equal. So, we need both signals.

# Problem 2

- The hardware shown was inserted when the datapath was extended for a specific instruction. Identify the instruction and explain what is the need for this extension.




**Solution:**

This hardware is used to compute the branch target address and determine which address to pass back to the PC.

- ❑ If the branch signal is 1, the PC will have the address of the target instruction by the beq instruction.
- ❑ If the branch signal is 0, the Pc will have the value PC+4 referring to the next instruction and indicating that the branch will not be taken.

# Problem 3

- What is gained by applying sign extension to bits from addi instruction?

**Solution:**

- The add immediate adds a value from a register of 32 bits to the an immediate value, the immediate value is 16-bits from the instruction.

Thus, the immediate value should be extended to 32 bits so the ALU could operate on two 32-bits operands

# Problem 4

- What is gained by applying Shift left 2 to bits from beq instruction?

**Solution:**

- Because beq instruction is in I-format and the least significant 16 bits represents the number of instructions between the beq and the target instruction.
- As each instruction takes 4 cells in memory, the number of instructions is first multiplied by 4 using shift left 2 before adding it to PC+4 to get the address of the branch target instruction.

# Thank You