

Computer Architecture

Lecture 6



Agenda

- Load and Store instructions
- MIPS Instruction formats
- Building MIPS-32 Datapath



MIPS-32 Instruction Set Architecture

- MIPS-32 uses 32 general purpose registers, each 32 bits wide
- The MIPS architecture can support up to 32 address lines.
- MIPS is a byte-addressable architecture
- The word size is 32-bits
- Each instruction 32-bits

To store an address or the content of a register, or an instruction you need **4** cells.





MIPS Instruction Formats

add \$s2, \$s1, \$s0
sub \$s2, \$s1, \$s0

**instructions with three
registers**

R-format

addi \$s2, \$s1, 10
beq \$s2, \$s1, 300

**instructions with two
registers and a value**

I-format

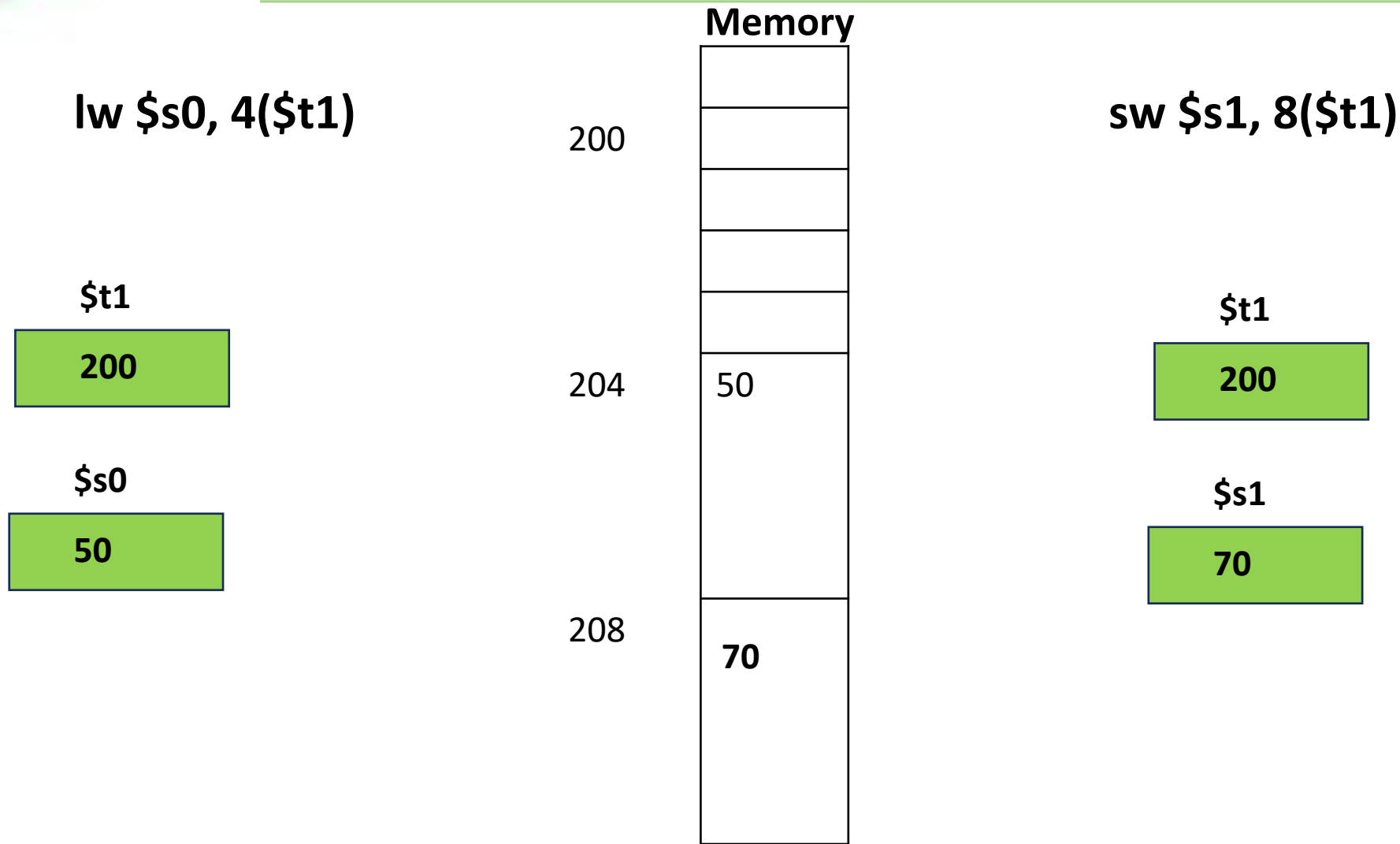
j 400

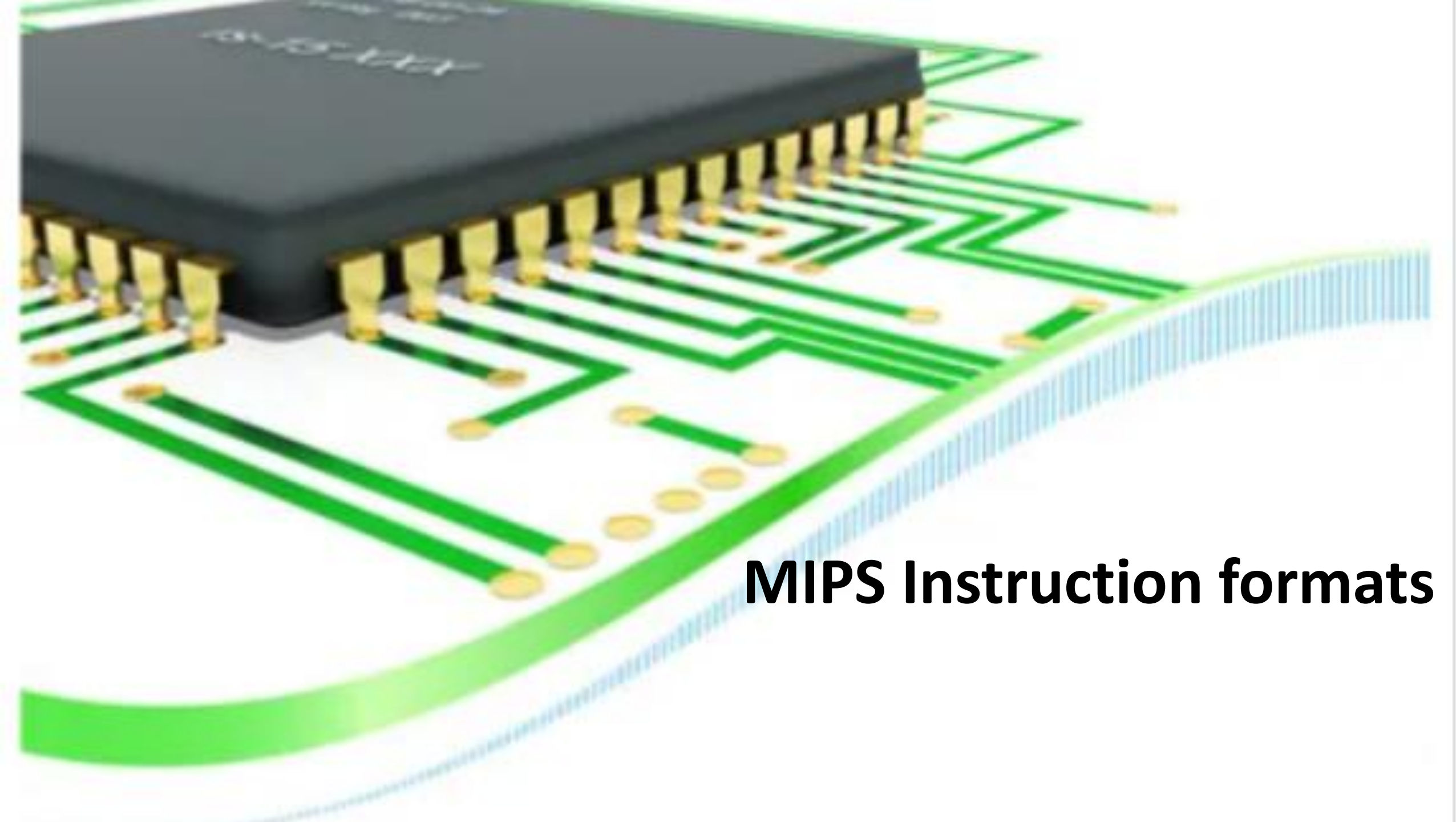
**instructions with one
value**

J-format



Load and store instructions

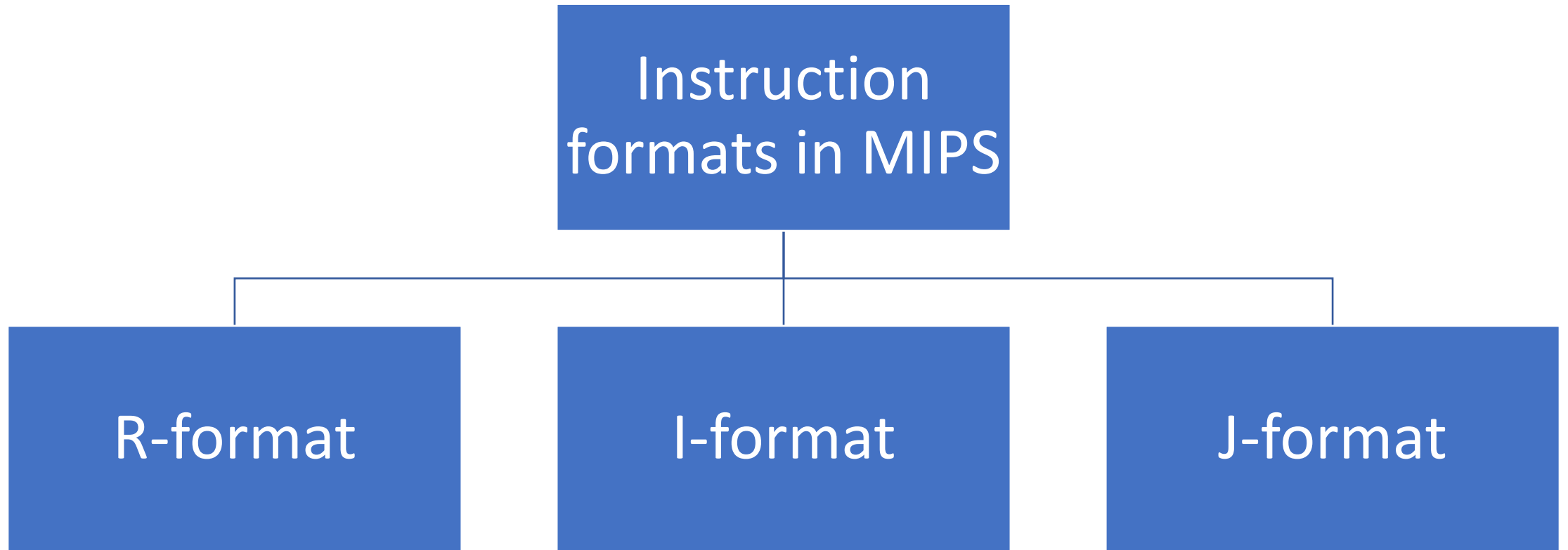




MIPS Instruction formats



MIPS Instruction Formats





R-format instructions

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

\$s0 → R16

\$s1 → R17

\$s2 → R18

32 bits

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



\$s0 → R16
\$s1 → R17

32 bits

addi \$s1, \$s0, 20

*Representation
in decimal →*

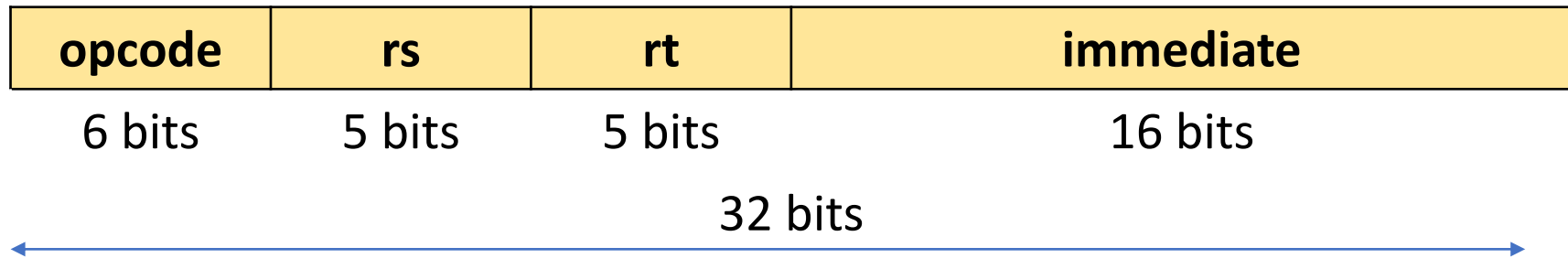
8	16	17	20
---	----	----	----

*Representation
in binary →*

001000	10000	10001	010100
--------	-------	-------	--------



I-format instructions

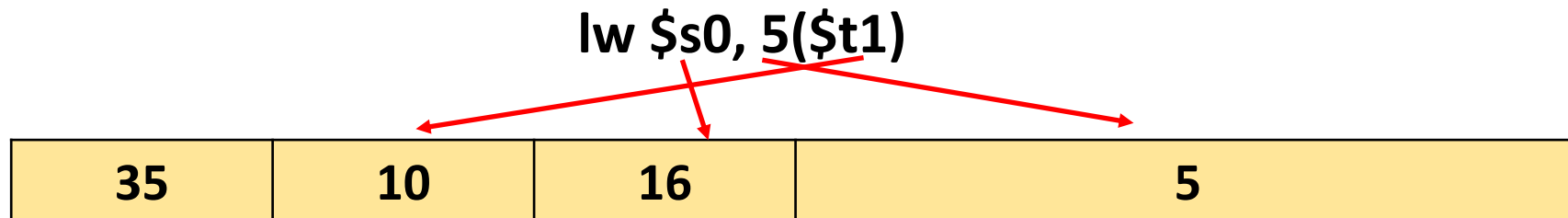
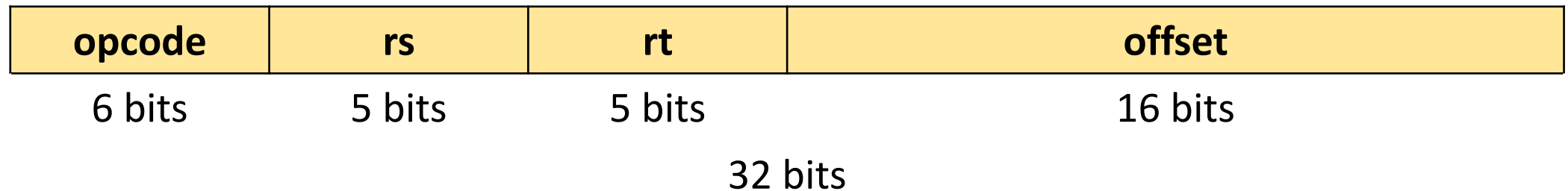


*N.B.: Branch, some load and store instructions are **I-format** instructions that have exceptions when translating them to binary → we will discuss that later*



Load and store instructions

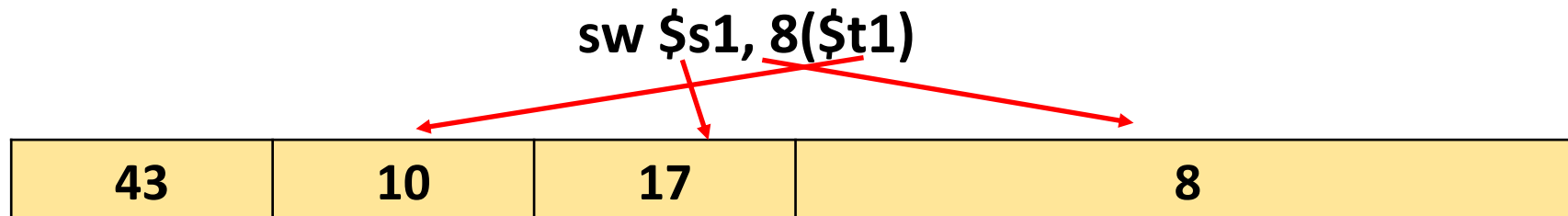
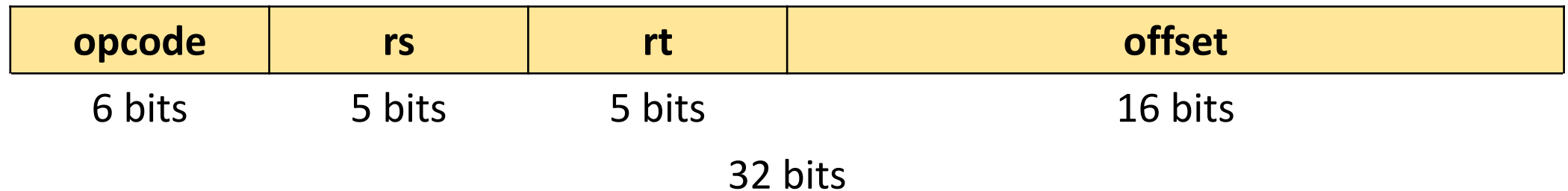
I-format





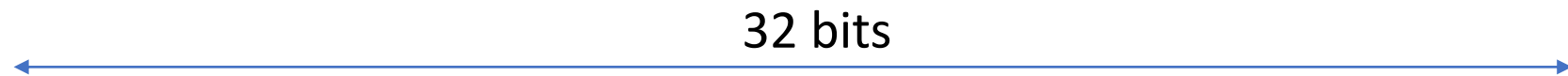
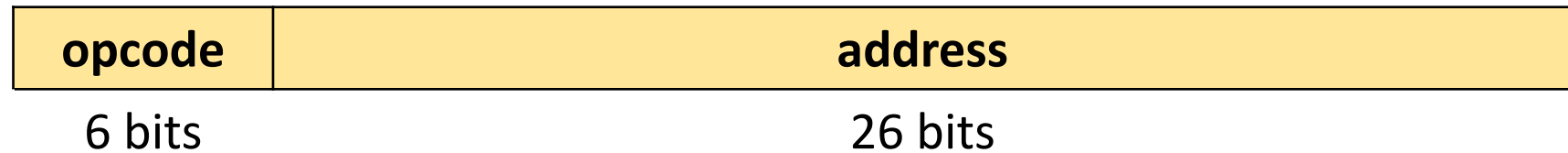
Load and store instructions

I-format





J-format instructions



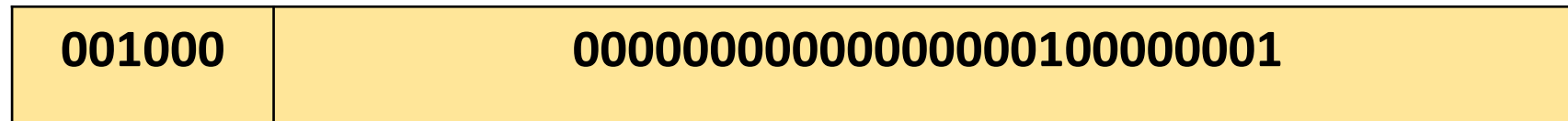
j 1028

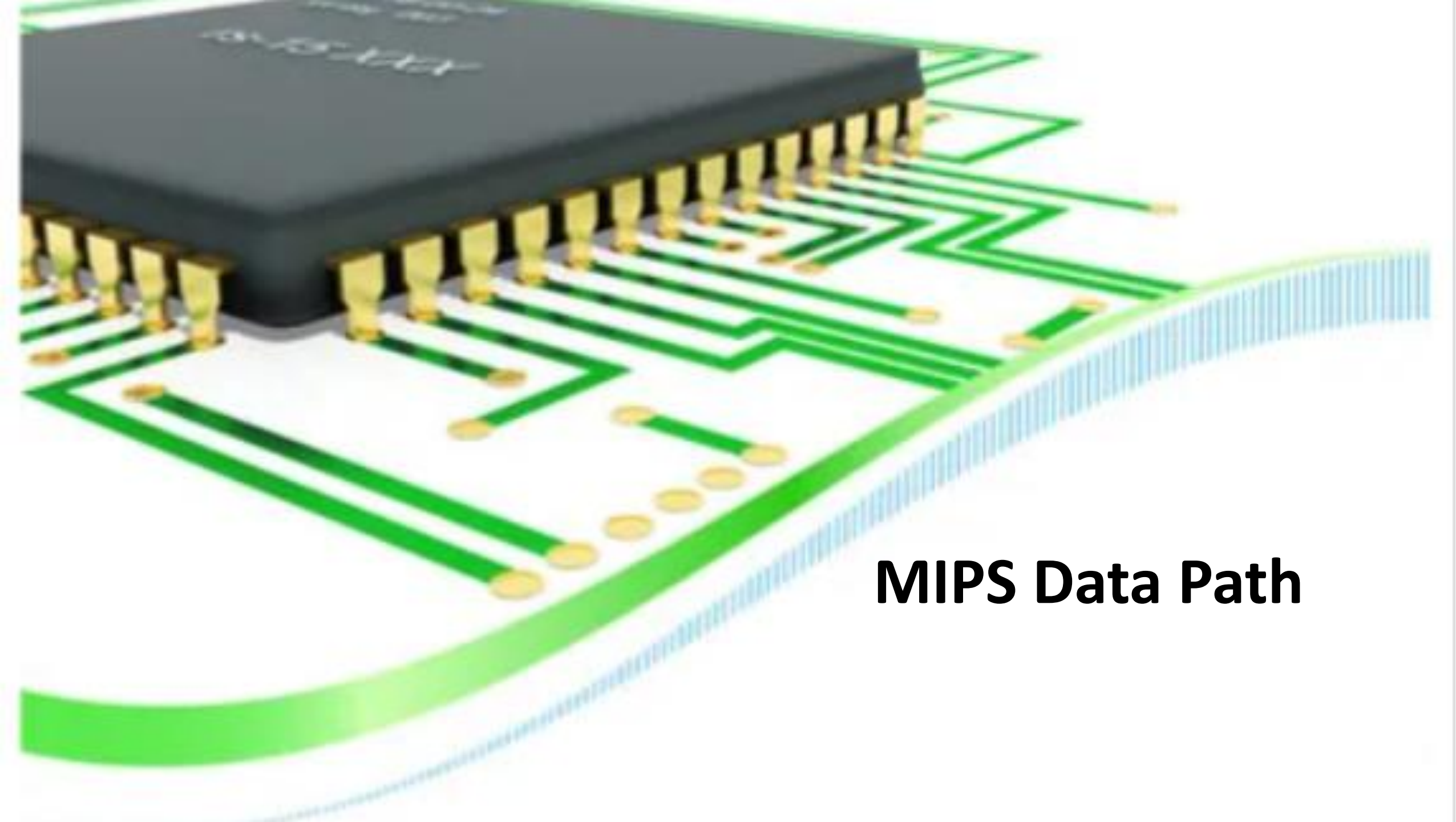
$1028/4 = 257$
We will know why later..

Representation
in decimal →



Representation
in binary →





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.

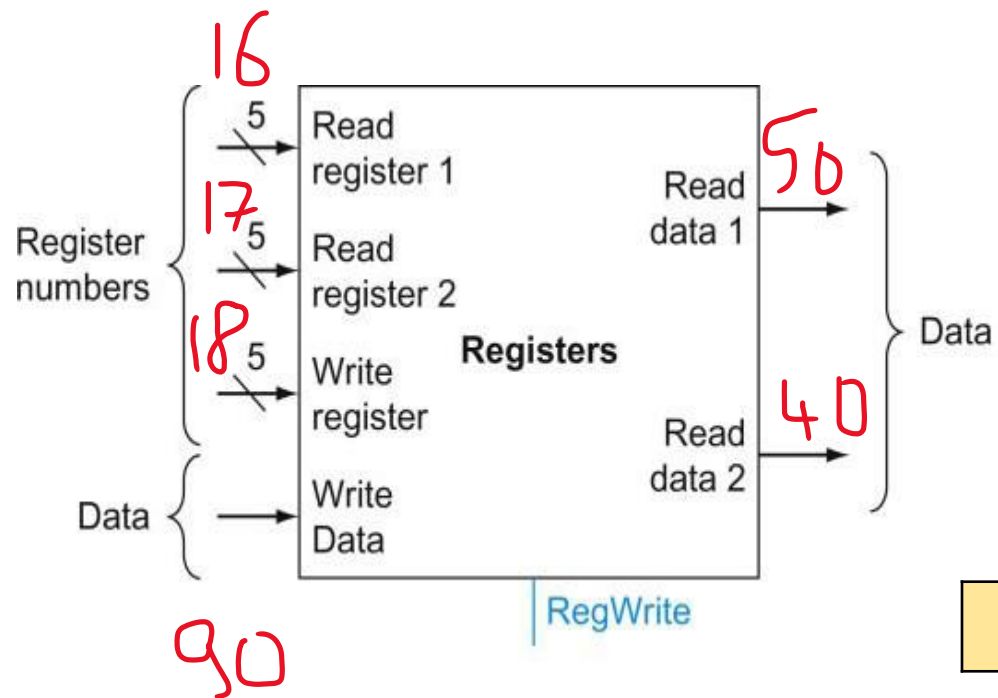


Datapath elements

- 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

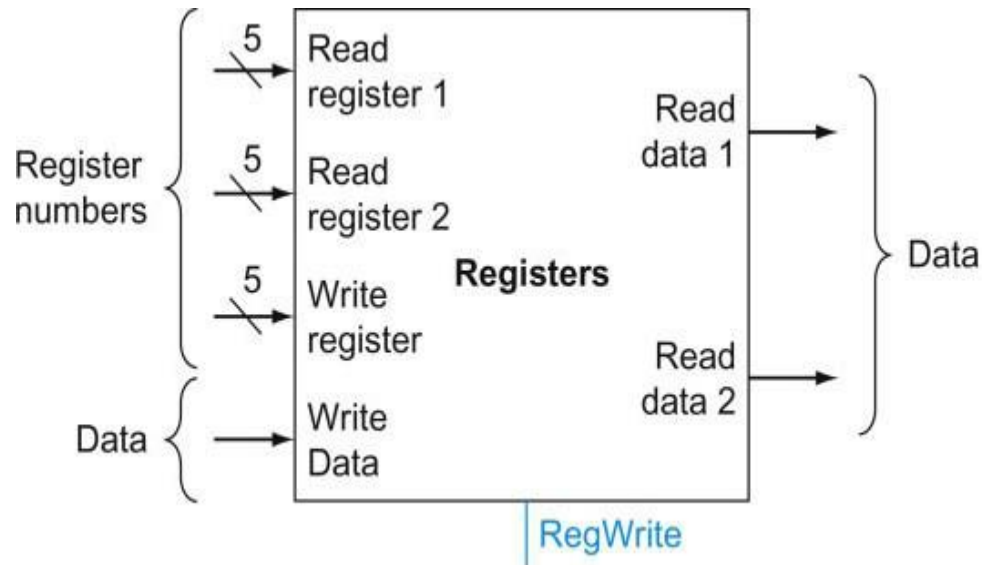
90



All inputs and outputs
are in binary



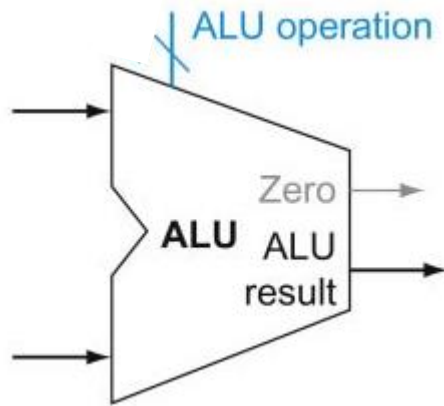
Registers File



- 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.



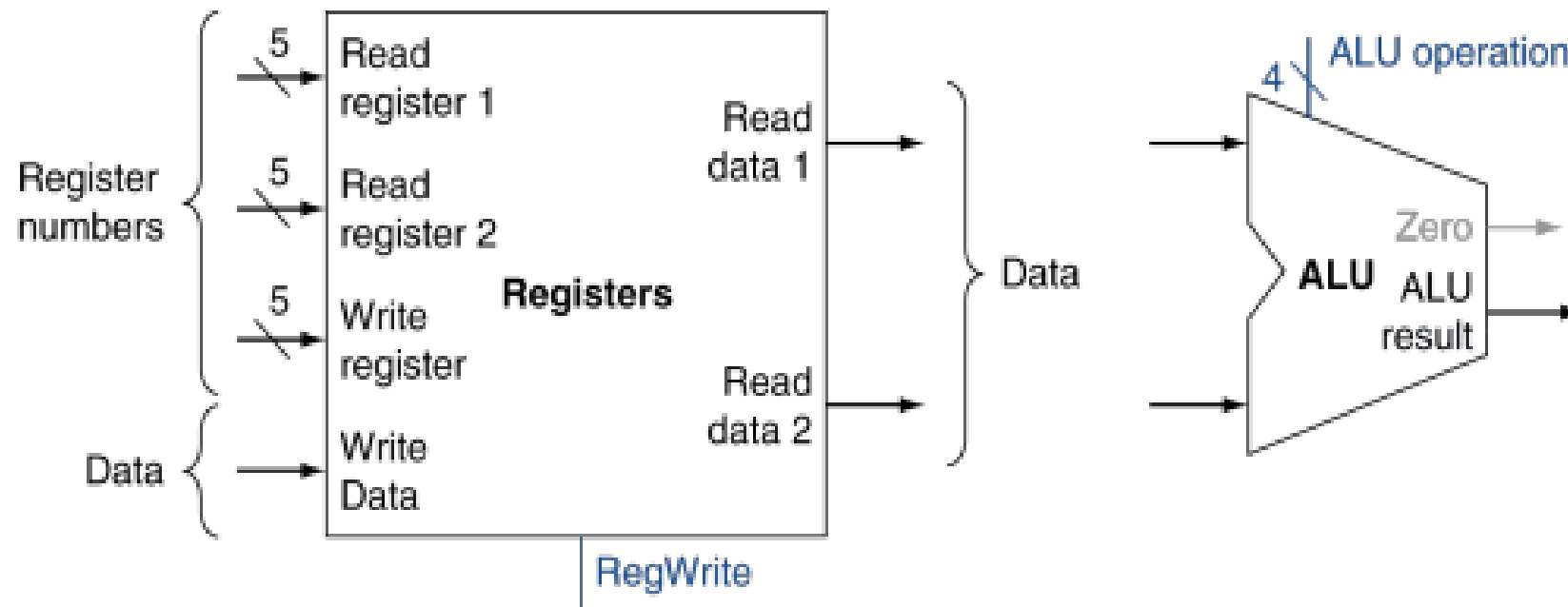
ALU



- 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.

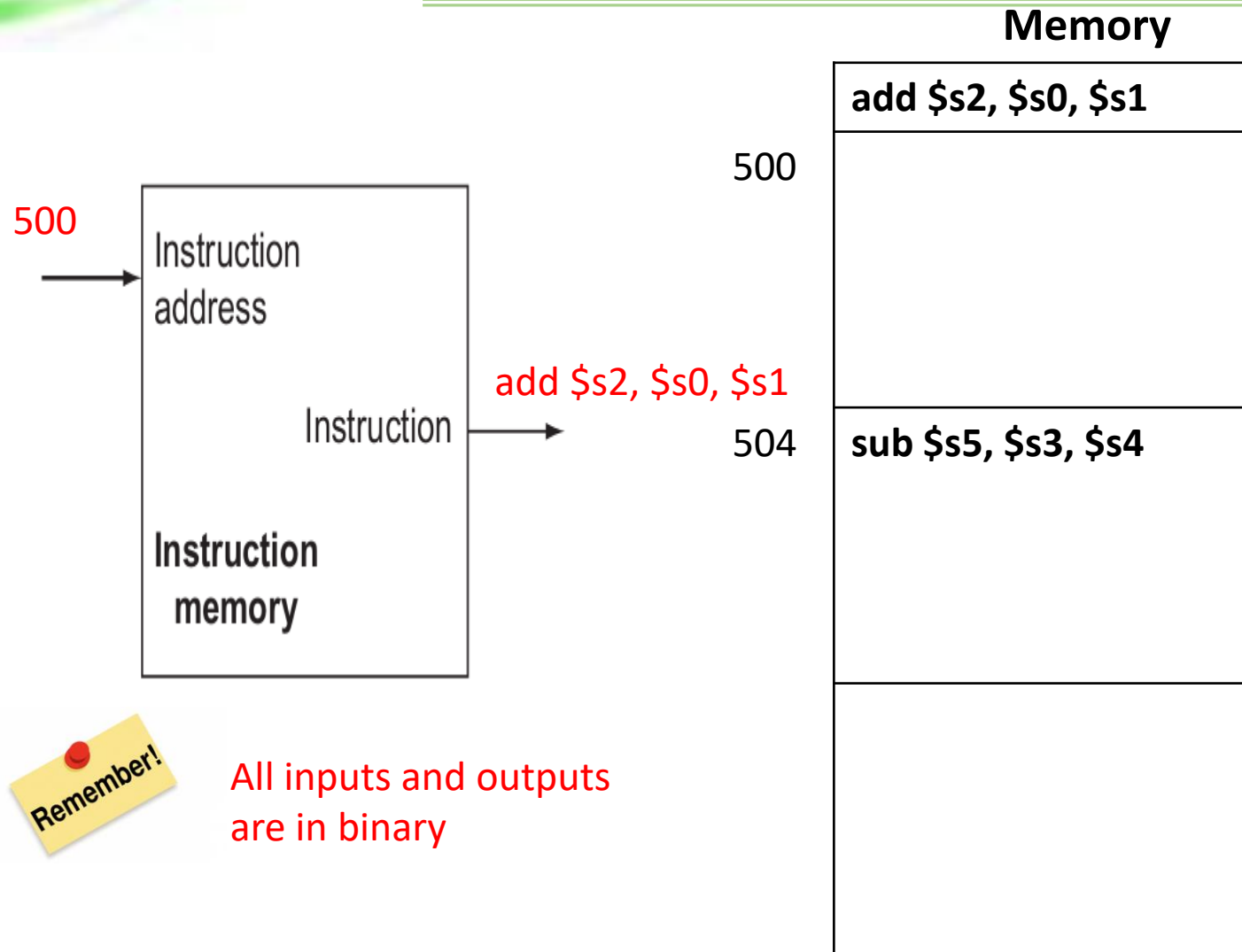


Registers file and ALU





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.

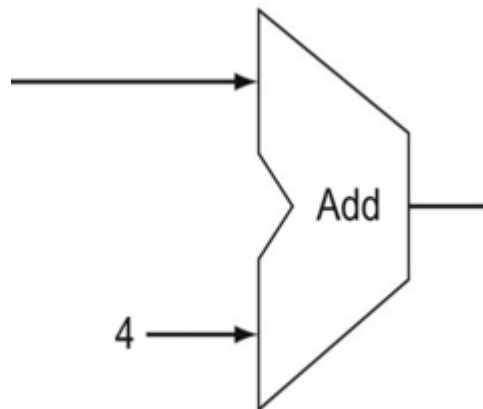
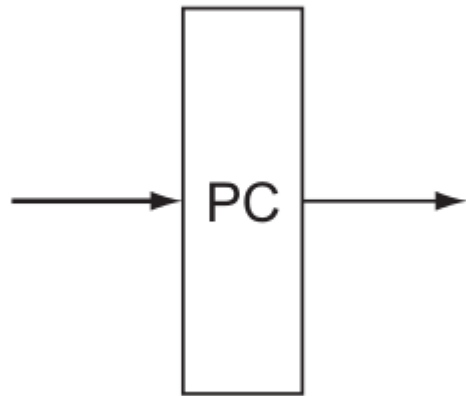
Note: MIPS has separate instruction and data memory.



All inputs and outputs are in binary



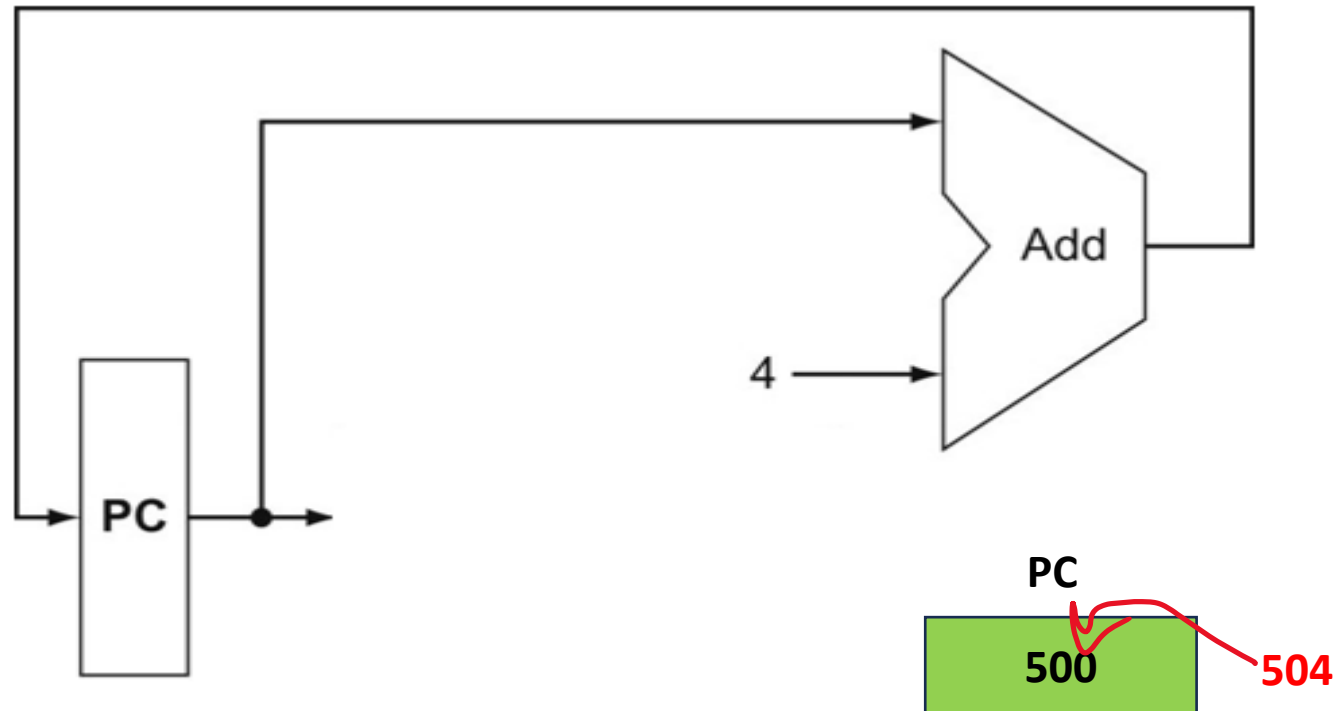
Program Counter and an Adder



- 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.



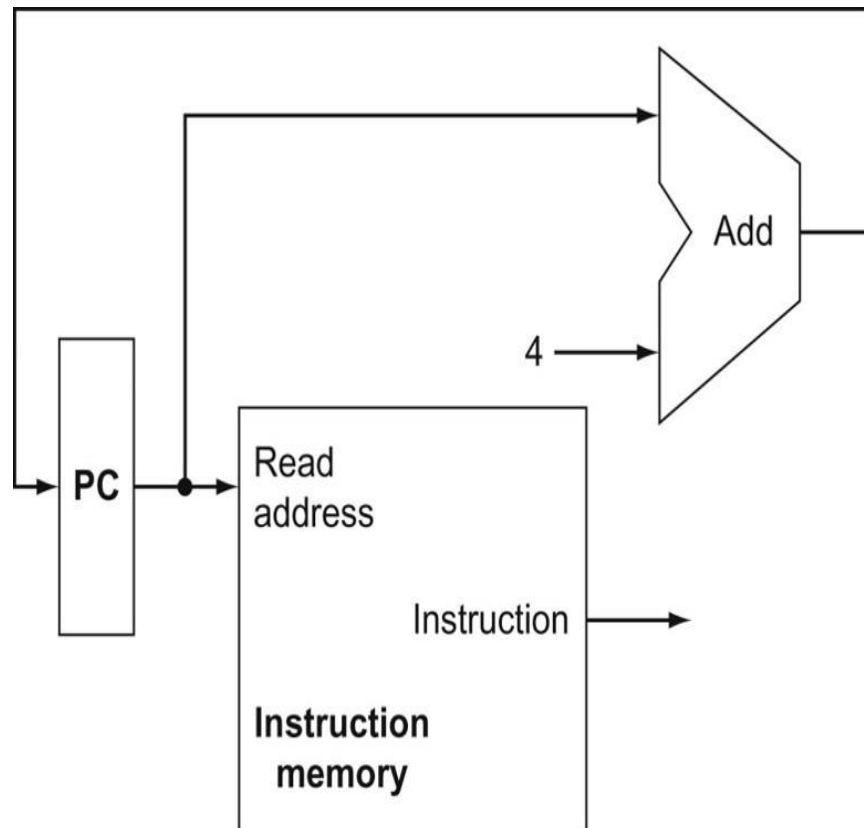
Program Counter and an Adder



Memory	
500	add \$s2, \$s0, \$s1
504	sub \$s5, \$s3, \$s4



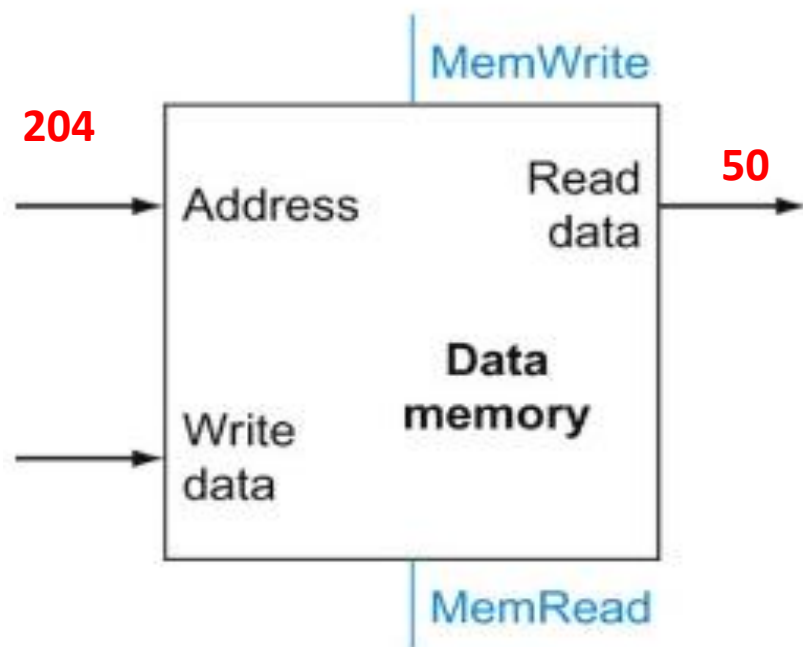
Program Counter, an Adder, and Instruction Memory



- 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 see that there is a separate adder dedicated to updating the PC.



Data Memory Unit

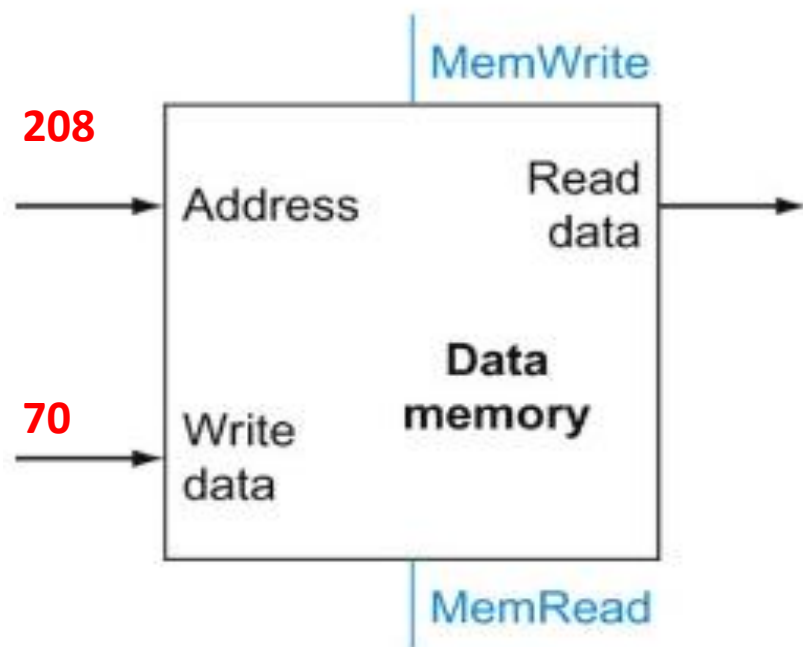


Reading from address
204

Memory	
200	
204	50
208	



Data Memory Unit

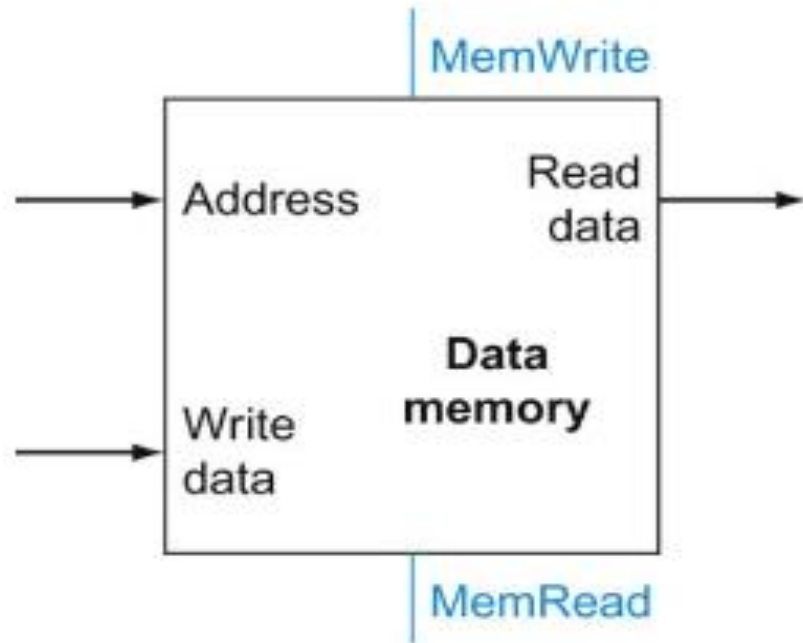


Writing 70 to address
208

Memory	
200	
204	50
208	70



Data Memory Unit

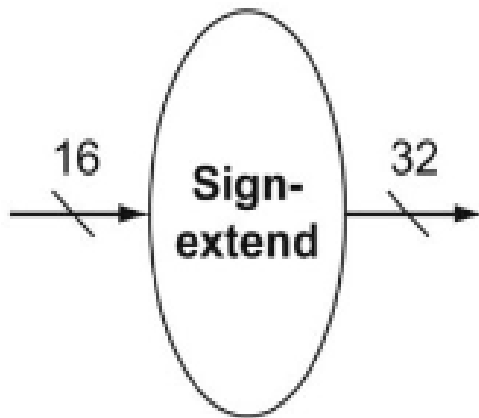


- The data memory stores ALU results.
- The data memory accepts a address and either accepts data through WriteData port if the instruction is writing in memory or outputs data if the instruction is reading from memory through WriteData port at the indicated address.

Note: Separate instruction and data memory.



Sign Extender



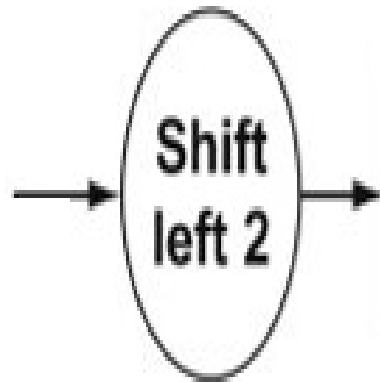
- The sign extender adds 16 leading digits to a 16-bit word with most significant bit b , to output the same value in a 32-bit word.
- The additional 16 digits have the same value as b .

0000000000000000 000000000011010

1111111111111111 111111111101100



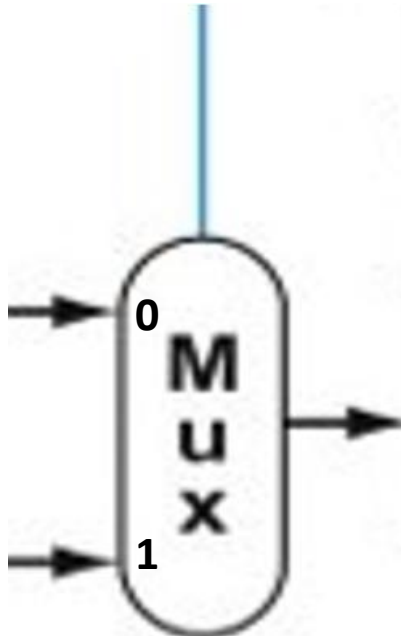
Shift left 2



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



2x1 Multiplexers



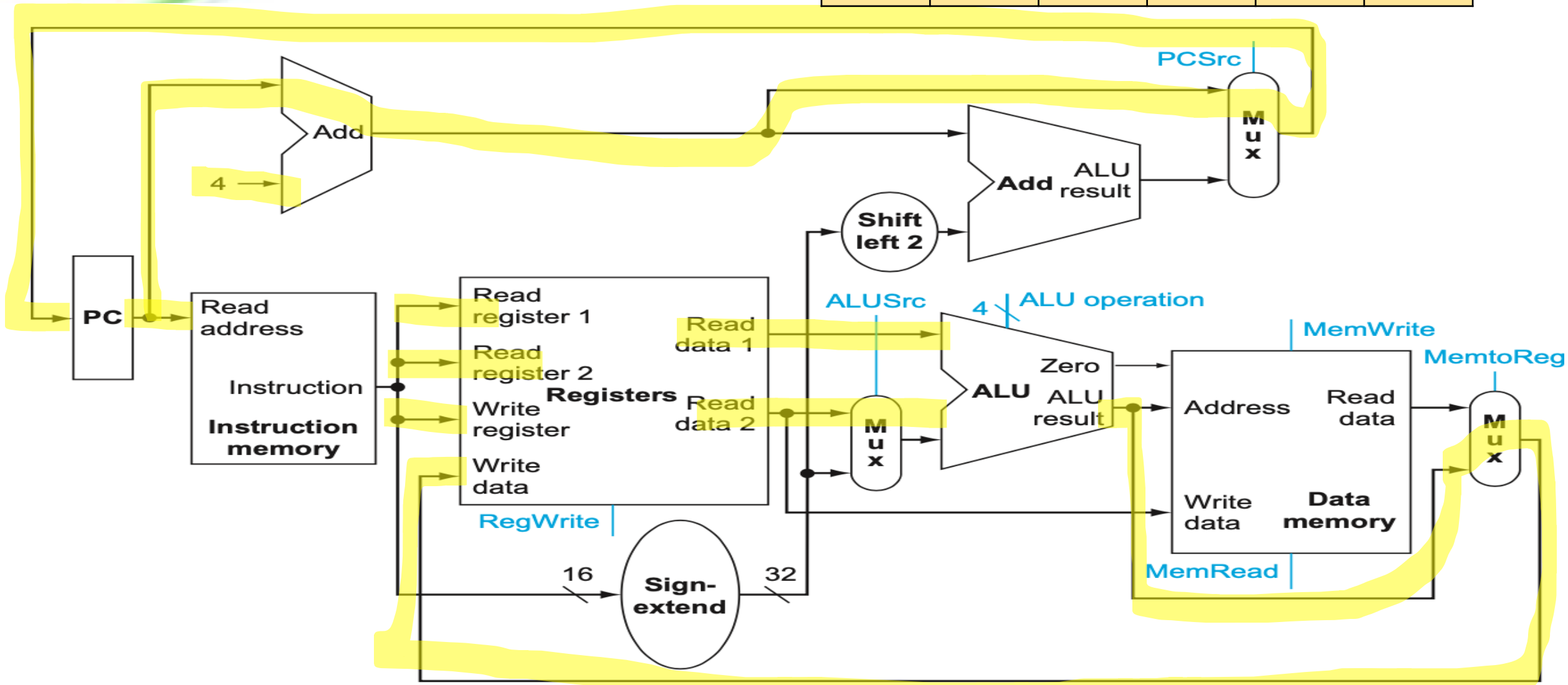
- In the Datapath, we will need 2x1 multiplexers
- Each 2x1 multiplexer has one selection line.



Scenario for R-format instructions

add \$s2, \$s0, \$s1

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

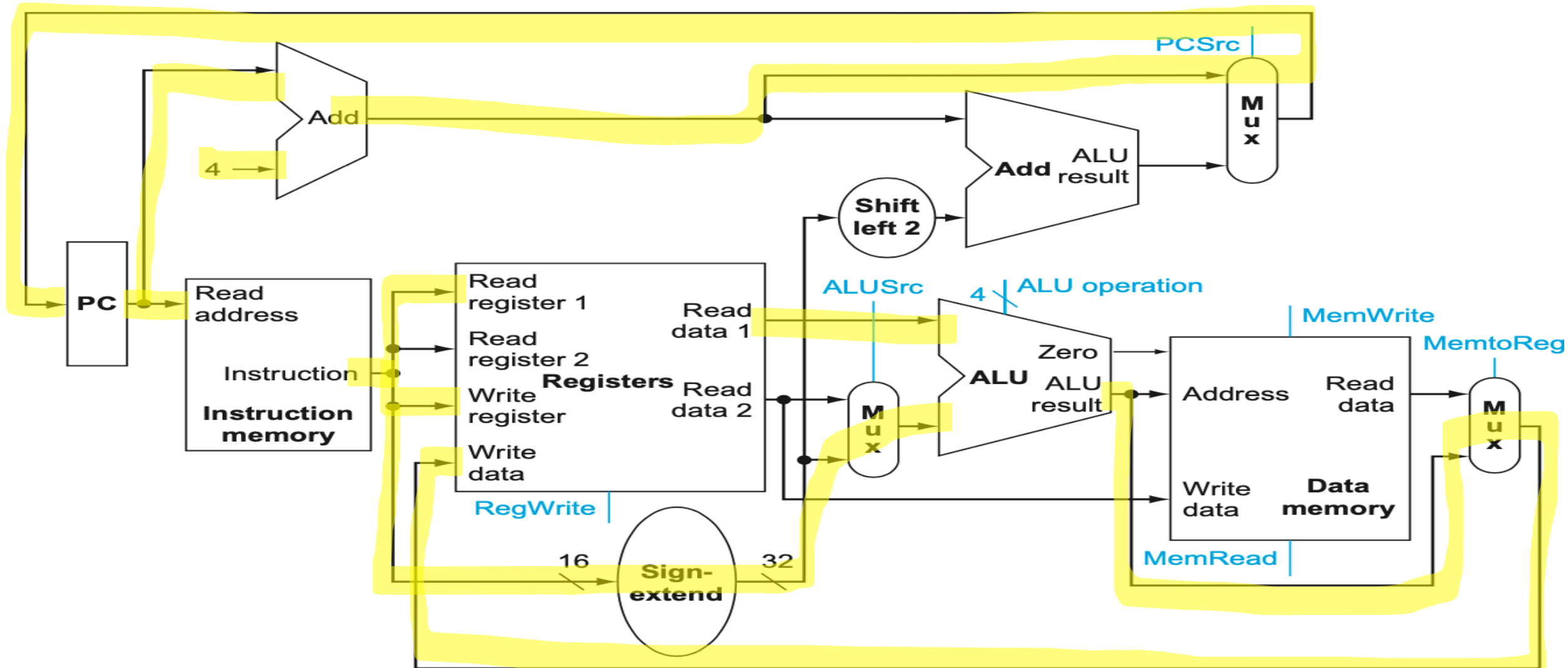




Scenario for I-format instructions

addi \$s1, \$s0, 20

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

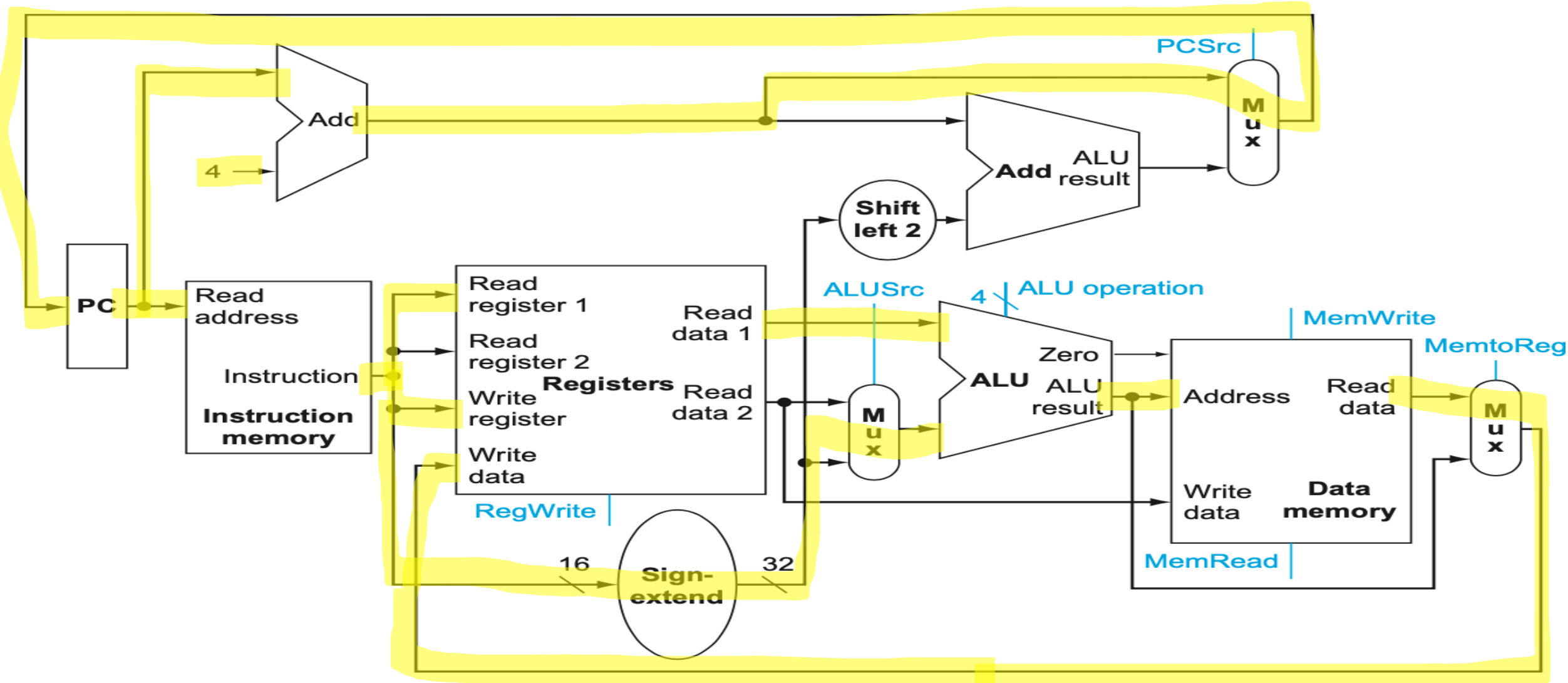




Scenario for I-format instructions

lw \$s0, 5(\$t1)

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

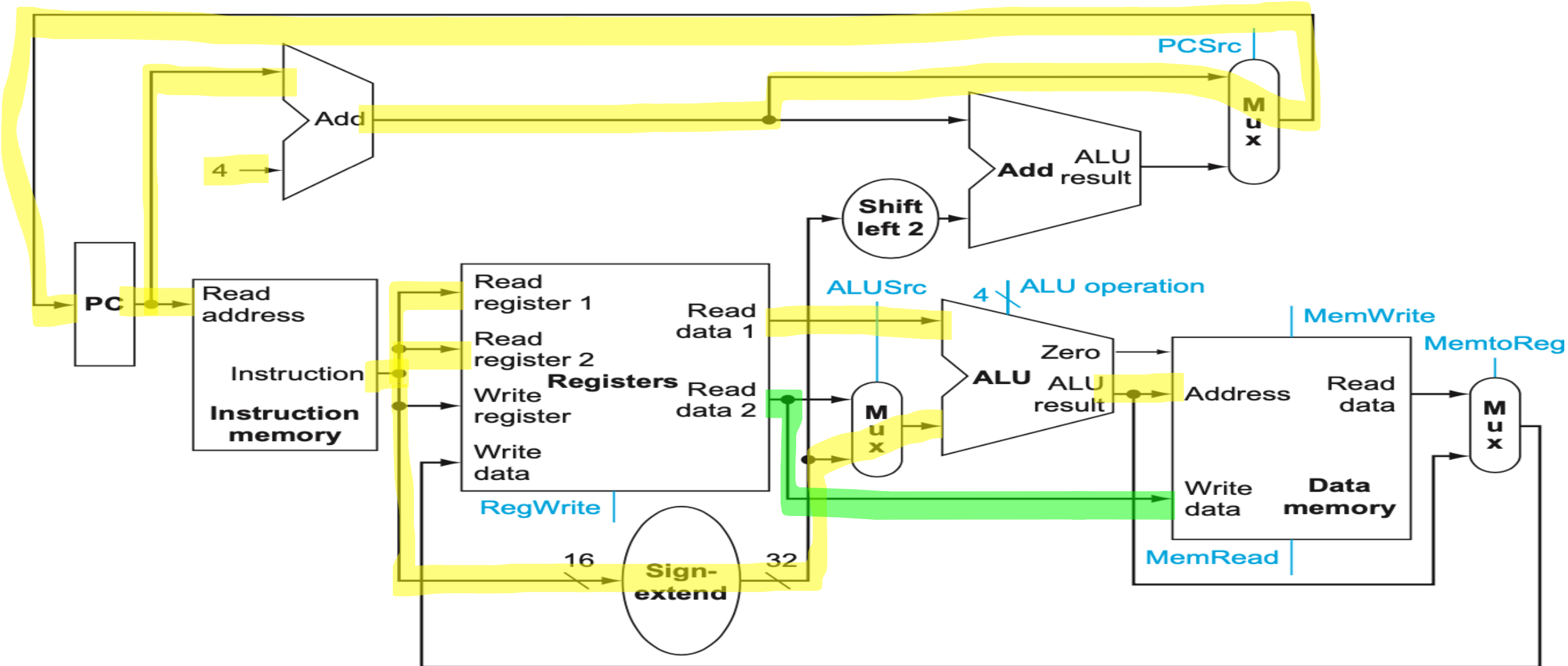




Scenario for I-format instructions

sw \$s1, 8(\$t1)

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



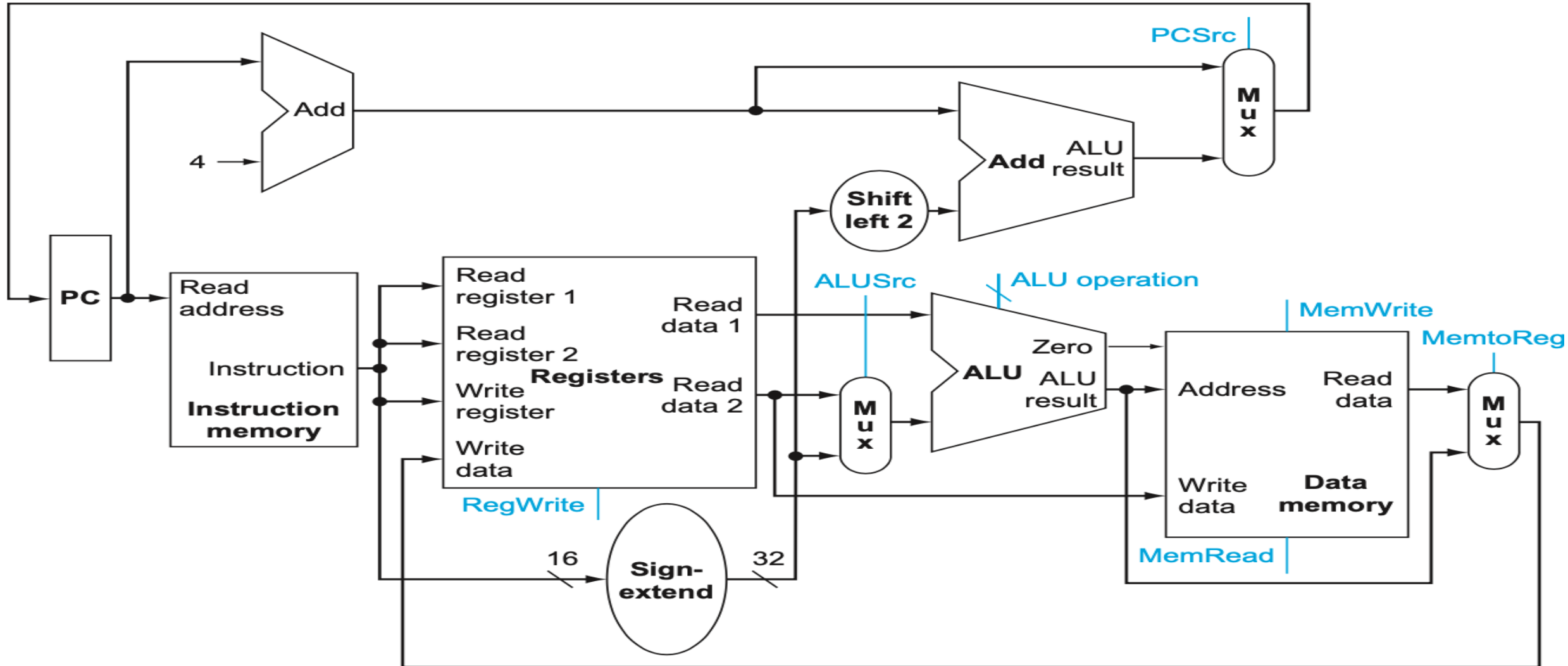


So why do we need MUXs?

The datapath is dealing with different formats and various options



Datapath with some control lines





Next lecture

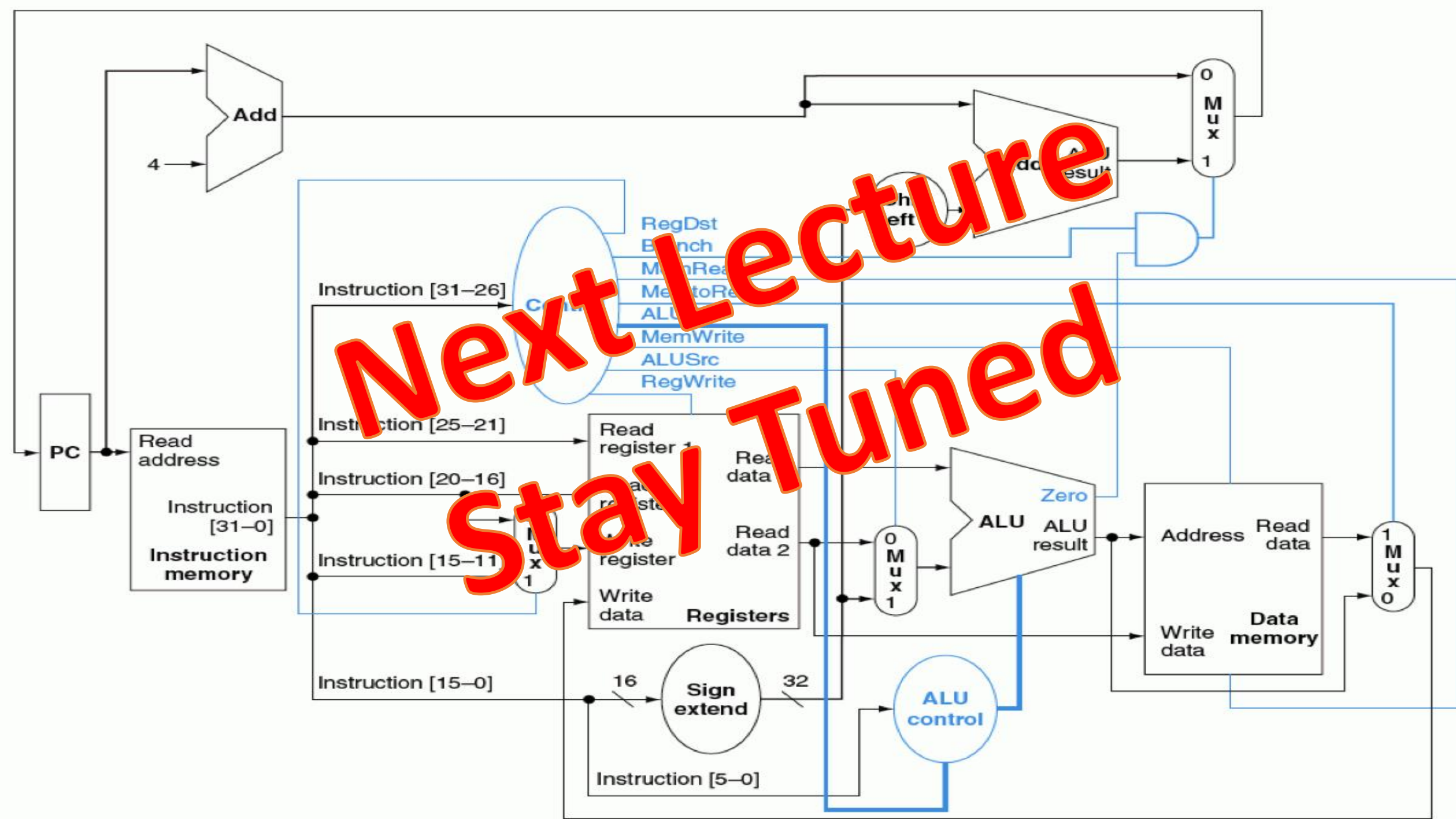


Control Signal Table

Just for Info



Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg	PCSrc
add	1	1	0	010	0	0	0	0
sub	1	1	0	110	0	0	0	0
and	1	1	0	000	0	0	0	0
or	1	1	0	001	0	0	0	0
lw	0	1	1	010	0	1	1	0
sw	X	0	1	010	1	0	X	0





Thank You

