## 4.2. BUILDING DATA PATH:

- **Single-cycle Datapath:**

  Each instruction executes in a single cycle
- **Multi-cycle Datapath:**

  Each instruction is broken up into a series of shorter steps
- **Pipelined Datapath:**

  Each instruction is broken up into a series of steps; Multiple instructions execute at once

**Differences between single cycle and multi cycle datapath**

❖ ***Single cycle Data Path*:**

- Each instruction is processed in one (long) clock cycle
- Two separate memory units for instructions and data.

❖ ***Multi-cycle Data Path*:**

- Divide the processing of each instruction into 5 stages and allocate one clock cycle per stage
- Single memory unit for both instructions and data
- Single ALU for all arithmetic operations
- Extra registers needed to hold values between each steps
  - Instruction Register (IR) holds the instruction
  - Memory Data Register (MDR) holds the data coming from memory
  - A, B hold operand data coming from the registers
  - ALUOut holds output coming out of the ALU

## Creating a single cycle datapath

❖ This simplest datapath will attempt to execute all instructions in **one clock cycle**. This means that no datapath resource can be used more than once per

instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated, many of theelements can be shared by different instruction flows.

- To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instructions. Let's start at the top by looking at which **datapath elements** each instruction needs, and then work our way down through the levels of **abstraction**. When we show the datapath elements, we will alsoshow their control signals. We use abstraction in this explanation, starting from the bottom up.

**Datapath Element**

A unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

**Program Counter (PC)**

- Figure 3.3a shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Figure 3.3 b also shows the **program counter (PC)**, the register containing the address of the instruction in theprogram being executed.

- Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU simply by wiring the control lines so that the control always specifies an add operation.

- We will draw such an ALU with the label *Add*, as in Figure 3.3c, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 byteslater.

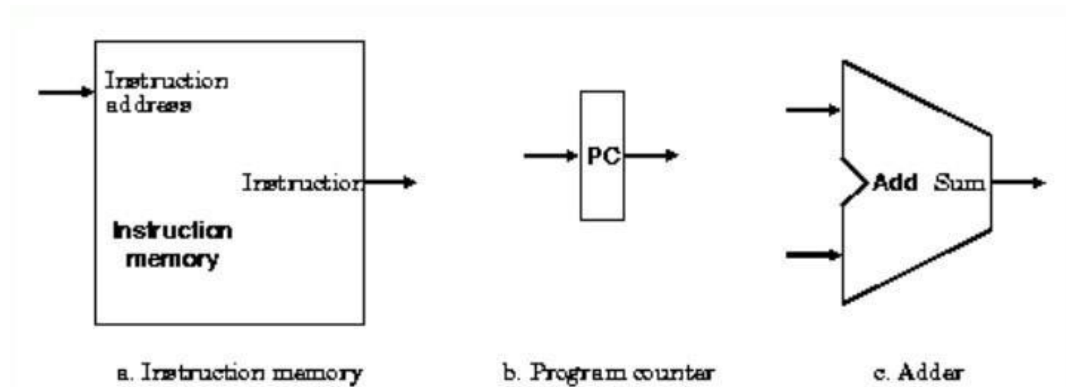



**Fig 3.3: Two state elements are needed to store and access instructions, and an adder is neededto compute the next instruction address.**

Figure 3.4 shows how to combine the three elements to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

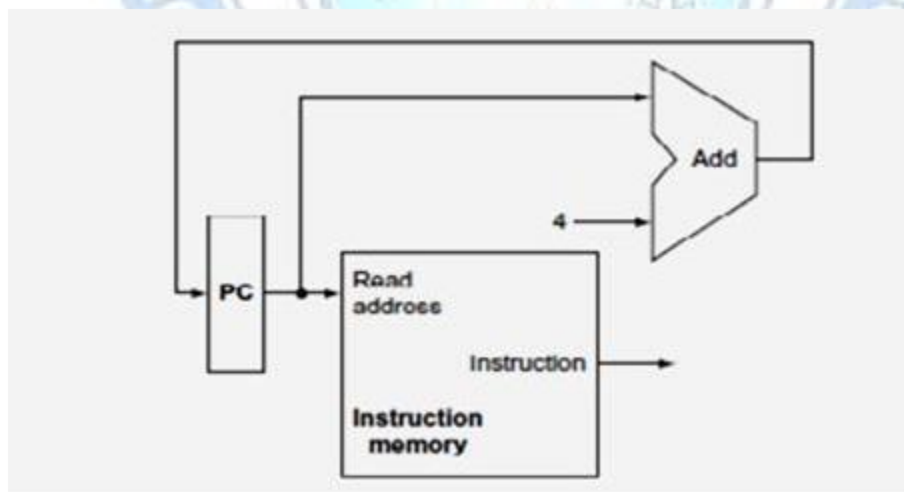



**Fig 3.4: A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath**

## R-FORMAT INSTRUCTIONS

- To perform any operation we required two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes add, sub, AND, OR, and slt,

- The processor's 32 general-purpose registers are stored in a structure called a **registerfile**. A register file is a collection of registers in which any register can be read orwritten by specifying the number of the register in the file.

- R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction.

- For each data word to be read from the registers, we need an input to the register file that specifies the *register number* to be read and an output from the register file that will carry the value that has been read from the registers.

- To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the *data* to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge.

- Figure 3.5a shows the result; we need a total of four inputs (three for register numbers and one for data) and two outputs (both for data). The register number inputs are 5

  bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide.

- Figure 3.5b shows the ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0.

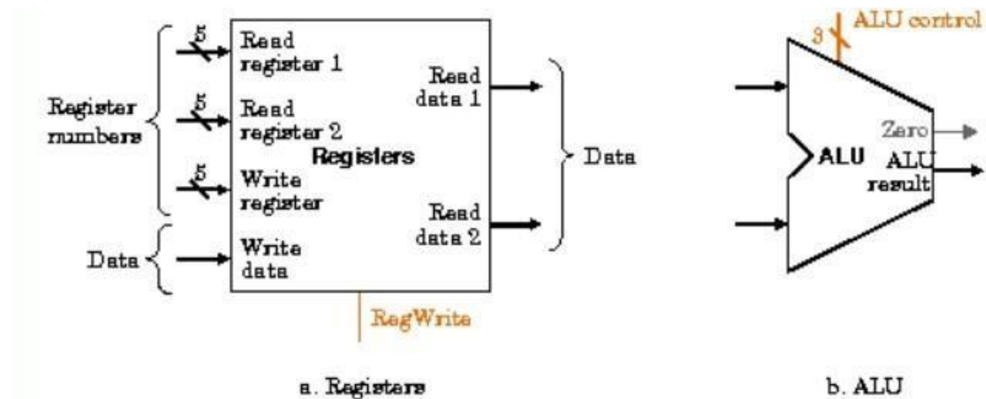



**Fig 3.5: The two elements needed to implement R-format ALU operations are the register file and the ALU.**

## DATAPATH SEGMENT FOR *Load Word* and *Store Word* INSTRUCTION

- Now, consider the MIPS load word and store word instructions, which have the general form lw $t1,offset_value($t2) or sw $t1,offset_value ($t2).

- In these instructions $t1 is a data register and $t2 is a base register. The memory address is computed by adding the base register($t2), to the 16-bit signed off set, values specified in the instruction.

- If the instruction is a store, the value to be stored must also be read from the register file where it resides in $t1. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is $t1. Thus, we will need both the register file and the ALU from Figure 3.5.

- In addition, we will need a unit to sign-extend the 16-bit off set field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; Figure 3.6 shows these two elements.

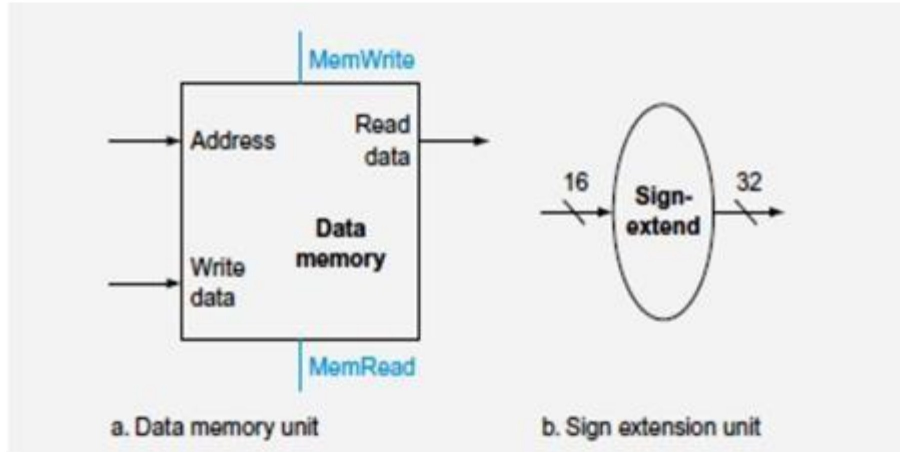


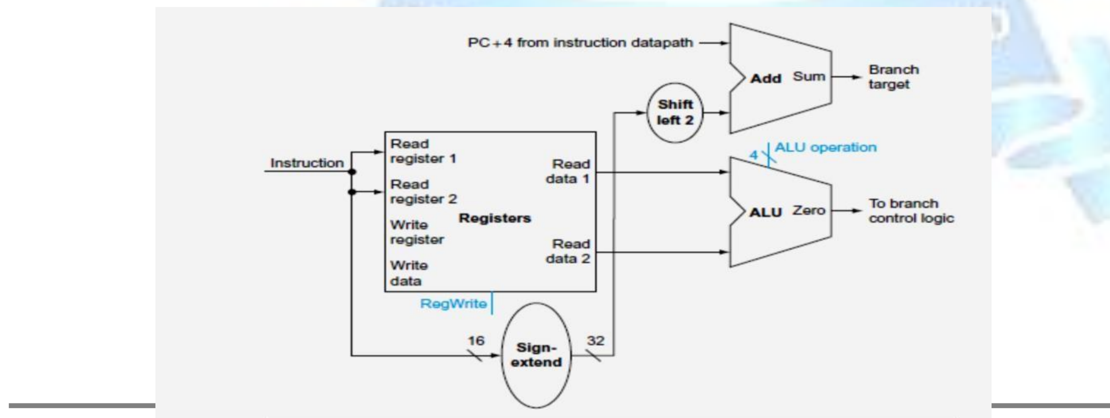

**Fig 3.6: The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 3.5**

## DATA PATH SEGMENT FOR *Branch* INSTRUCTION

For computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address.

When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, and we say that the **branch is taken**. If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch is not taken**.

Figure 3.7 shows the structure of the datapath segment that handles branches. To compute the branch target address, the branch datapath includes a sign extension unit, and an adder.



Since that ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract. If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equal test of branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

The jump instruction operates by replacing the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits.