UNIT – I
BASIC STRUCTURE OF COMPUTER SYSTEM

Functional Units - Basic Operational Concepts - Performance - Instructions: Language ofthe Computer - Operations, Operands - Instruction representation - Logical operations - decision making - MIPS Addressing.

1. FUNCTIONAL UNITS

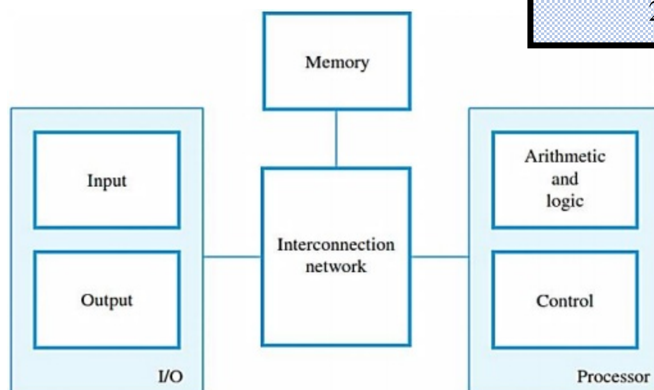
A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units.

Components Of Computer System

1. Input Unit
2. Memory Unit
   - 2.1. Primary Memory
     - 2.1.1. Word
     - 2.1.2. Word Length
     - 2.1.3. Basic Operation
     - 2.1.4. Address
   - 2.2. Secondary Memory
     - 2.2.1. Access Time
   - 2.3. Arithmetic And Logic Unit
     - 2.3.1. Registers
     - 2.3.2. One register – One word
   - 2.4. Output Unit
   - 2.5. Control unit
     - 2.5.1. Timing and synchronization

1.1. INPUT UNIT

- Computers accept coded information through input units.
- Most common input device is the keyboard.
- When a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.



Other kinds of input devices are

- ✓ **Touchpad**,
- ✓ **Mouse**,
- ✓ **Joystick**,
- ✓ **Trackball**.
- ✓ **Microphones** – used to capture audio input which is then sampled and converted into digital codes for storage and processing.
- ✓ **Cameras** – used to capture video input and it is sampled similar to microphone

1.2. MEMORY UNIT

- The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

1.2.1. PRIMARY MEMORY

- It is also called main memory. It is a fast memory that operates at electronic speeds.

- Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each cell can store one bit of information.
- These cells are rarely read or written individually. In general, they are handled in groups of fixed size called **WORDS**. The memory is organized so that **one word** can be stored or retrieved in **one basic operation**.
- The number of bits in each word is referred to as the **WORD LENGTH** of the computer, typically 16, 32, or 64 bits.
- To provide easy access to any word in the memory, a unique **ADDRESS** is associated with each word location.
- Addresses are consecutive numbers, starting from 0, that identify locations. A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.

1.2.2. Secondary Storage

- Primary memory is expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondary storage is used when large amounts of data and many programs have to be stored, for information that is accessed infrequently.
- **Access times** for secondary storage are **longer** than for primary memory.
- Some secondary storage devices are magnetic disks, optical disks (DVD and CD), and flash memory devices.

1.3. Arithmetic and Logic Unit

- Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor.
- For example: If two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may be stored in the memory or kept in the processor for immediate use.
- When operands are brought into the processor, they are stored in high-speed storage elements called **REGISTERS**.
- **One register** can store **one word** of data.
- Access times to registers are even shorter than access times to the cache unit on the processor chip.

1.4. Output Unit

- The function of output device is to send processed results to the outside world.

- Most common example of output device is a printer. Printers are mechanical devices, and are slow compared to the electronic speed of a processor.
- Some units, such as graphic displays, provide both output function, and input function, through touchscreen capability. The dual role of such units is the reason for using the single name input/output (I/O) unit in many cases.

1.5. CONTROL UNIT

- The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations.
- Coordinating the operation of these units is the responsibility of the control unit.
- Control circuits are responsible for ***generating the timing signals*** that control the transfers and determine when a given action is to take place.
- Data transfers between the processor and the memory are also managed by the control unit through timing signals.
- Most of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for **TIMING AND SYNCHRONIZATION** of events in all units.

2. BASIC OPERATIONAL CONCEPTS

To perform a given task an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be stored are also stored in the memory.

Examples: - Add LOCA, R_0

This instruction adds the operand at memory location LOCA, to operand in register R_0 & places the sum into register. This instruction requires the performance of several steps,

1. First the instruction is fetched from the memory into the processor.
2. The operand at LOCA is fetched and added to the contents of R_0
3. Finally the resulting sum is stored in the register R_0

The preceding add instruction combines a memory access operation with an ALU Operations. In some other type of computers, these two types of operations are performed by separate instructions for performance reasons.

```
Load   LOCA, R1
Add   R1, R0
```

Transfers between the memory and the processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data are then transferred to or from the memory.

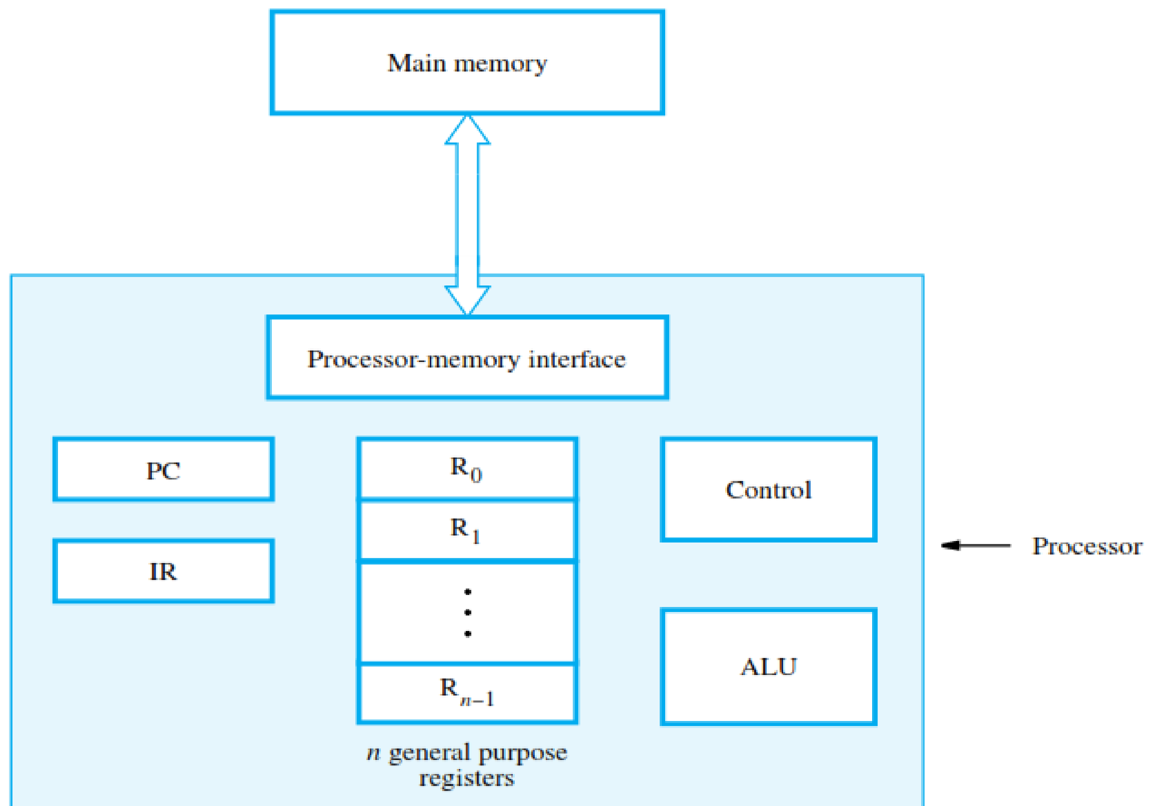


Fig 1.1 : Connections between the processor and the memory

The fig shows how memory & the processor can be connected. In addition to the ALU & the control circuitry, the processor contains a number of registers used for several different purposes.

The instruction register (IR):- Holds the instructions that is currently being executed. Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

The program counter PC:-

This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed.

Besides IR and PC, there are n-general purpose registers R0 through R_{n-1}

The other two registers which facilitate communication with memory are: -

1. **MAR – (Memory Address Register):-** It holds the address of the location to be accessed.
2. **MDR – (Memory Data Register):-** It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the I/P unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
7. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.
8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
12. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Normal execution of a program may be preempted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal.

An interrupt is a request signal from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine.

The Diversion may change the internal stage of the processor its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue.

3. PERFORMANCE OF A COMPUTER

Determining the performance of computers can be tough. Main reasons for this difficulty are,

- **Size and Complexity** of modern software systems,
- Wide range of **performance improvement techniques** used by hardware designers.

Performance of A Computer

- Size and Complexity
- performance improvement techniques

1. Defining Performance

- Response time
- Throughput

$$Performance_x = \frac{1}{Execution\ Time_x}$$

2. Relative Performance

$$\frac{Performance_A}{Performance_B} = \frac{Execution\ Time_B}{Execution\ Time_A}$$

3. Measuring Performance

- clock time

4. CPU Execution Time

4.1. Types of CPU time

- User CPU time
- System CPU time
- System clock
- Clock cycles

3.1. DEFINING PERFORMANCE

If a program is run on two different desktop computers, in general the faster one is the desktop computer that finishes the job done first.

Consider a datacenter that had several servers running jobs submitted by many users, in general the faster computer is the one that completed the most jobs during a day.

INDIVIDUAL computer users are interested in **reducing response time**.

DATACENTER MANAGERS are often interested in **increasing throughput**

5. Clock period

6. CPU Performance and its Factors

$$CPU\ Execution\ Time\ for\ a\ program = \frac{CPU\ Clock\ Cycle\ for\ a\ program}{Clock\ Rate}$$

7. Instruction Performance

$$CPU\ clock\ cycles = Instructions\ for\ a\ program \times Average\ clock\ cycles\ per\ instruction$$

8. Clock Cycles per Instruction (CPI)

9. CPU Performance Equation

$$CPU\ Time = \frac{Instruction\ count\ \times CPI}{Clock\ Rate}$$

- Measuring CPU Execution Time
- Measuring Clock Cycle Time
- Measuring Instruction Count

3.1.1. RESPONSE TIME

It is also called as **EXECUTION TIME**. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, etc.

3.1.2. THROUGHPUT

It is also called **BANDWIDTH**. It is the number of tasks completed per unit time.

To maximize performance, we should minimize response time or execution time for some task. We can relate performance and execution time for a computer X as, as,

$$\boldsymbol{Performance_x = \frac{1}{Execution\ Time_x}}$$

Consider two computers X and Y, if the performance of X is greater than the performance of Y, then

$$Performance_x > Performance_y$$

$$\frac{1}{Execution\ Time_x} > \frac{1}{Execution\ Time_y}$$

$$Execution\ Time_y > Execution\ Time_x$$

The execution time on Y is more than that of X, if X is faster than Y.

3.1.3. Relative Performance

In order to calculate the relative performance consider the following example:

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B.

$$\frac{Performance_A}{Performance_B} = \frac{Execution\ Time_B}{Execution\ Time_A}$$

Thus the performance ratio is

$$15/10 = 1.5$$

So A is faster than B in 1.5 times

$$\frac{Performance_A}{Performance_B} = 1.5$$

PERFORMANCE AND EXECUTION TIME ARE RECIPROCALS, increasing performance decreases execution time.

3.2. Measuring Performance

Time is used to measure performance of a computer, the computer that performs the same amount of work in the least time is considered as faster.

Program execution time is measure in **SECONDS PER PROGRAM**.

The most common definition of time is wall **clock time, response time or elapsed time**.

Processor may work on several programs simultaneously in order to optimize throughput instead of minimizing the elapsed time for one program.

We need to differentiate the elapsed time and amount of time which the processor is working on the program. CPU execution time is used to differentiate this.

3.2.1. CPU Execution Time

CPU execution time also called CPU time. It is defined as the actual time the CPU spends computing for a specific task.

3.2.2. Types of CPU time

CPU time can be classified into two types

1. **User CPU time** - CPU time spent in the program
2. **System CPU time** - CPU time spent in the operating system performing tasks on behalf of the program.

Differentiating between system and user CPU time is difficult to do accurately, to increase the system performance the computer designers must know **how fast the hardware can perform basic functions**.

3.2.3. System Clock

All computers are constructed using a **CLOCK** that determines when events take place in the hardware.

3.2.4. Clock cycles

Clock cycles also called tick, clock tick, clock period, clock or cycle. **Clock cycle** is the time for one clock period usually of the processor clock, which runs at constant rate.

3.2.5. Clock period

The length of each clock cycle is known as **CLOCK PERIOD**.

3.3. CPU Performance and its Factors

Users and designers have different metrics to measure the performance.

In general CPU performance is a one of metrics for measuring the performance. To know the CPU performance we must find the CPU execution time.

$$\begin{aligned} &\textit{CPU Execution Time for a program} \\ &\quad = \textit{CPU Clock Cycle for a program} \times \textit{Clock Cycle Time} \end{aligned}$$

$$\textit{Clock Cycle Time} = \frac{1}{\textit{Clock Rate}}$$

$$\textit{CPU Execution Time for a program} = \frac{\textit{CPU Clock Cycle for a program}}{\textit{Clock Rate}}$$

It is clear that the hardware designer can **IMPROVE PERFORMANCE** by **reducing the number of clock cycles required for a program or the length of the clock cycle**.

3.4. Instruction Performance

Performance equations mentioned above does not include any reference to the number of instructions needed for the program. But the execution time depends on the number of instructions in a program, so the equation is modified as follows.

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \text{Average clock cycles per instruction}$$

3.4.1. Clock Cycles per Instruction (CPI)

- Average number of clock cycles per instruction for a program or program fragment
- Different instructions may take different amounts of time depending on what they do.
- CPI is an average of all the instructions executed in the program.
- CPI provides one way of comparing two different implementations of the same instruction set architecture.

3.5. CPU Performance Equation

CPU performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time is as follows

$$\text{CPU Time} = \text{Instruction count} \times \text{CPI} \times \text{Clock Cycle Time}$$

The clock rate is the inverse of clock cycle time

$$\text{CPU Time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock Rate}}$$

These formulas are used to **compare two different implementations** or to evaluate a design alternative.

3.5.1. Measuring CPU Execution Time

- We can measure the CPU execution time by running the program.

3.5.2. Measuring Clock Cycle Time

- The clock cycle time is usually provided by the computer manufacturer.

3.5.3. Measuring Instruction Count

- We can measure the instruction count by using software tools or by using a simulator of the architecture or using hardware counters,
- Most of the processors have hardware counters, to record the number of instructions executed, the average CPI, and other sources of performance loss.
- The instruction count depends on the architecture, but not on the exact implementation, so instruction count can be measured without knowing all the details of the implementation.
- The CPI depends on a wide variety of design details in the computer, including both the memory system and the processor structure.

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

Consider the computer with three instruction classes and CPI measurement as given below and instruction count for each instruction class for the same program from two different compilers are given. Assume that the computer`s clock rate is 4GHz. Which code sequence will execute faster according to execution time

Code from	CPI for this Instruction Class		
	A	**B**	**C**
CPI	**1**	**2**	**3**
Code from	**Instruction Count for each class**		
	A	**B**	**C**
Compiler 1	**2**	**1**	**2**
Compiler 2	**4**	**1**	**1**

[Nov/Dec 2014 – 6M]

$$CPU\ Time = \frac{Instruction\ count\ \times CPI}{Clock\ Rate}$$

COMPILER 1:

Total CPU time of Compiler 1 is the sum of CPU time of all Instruction Classes

$$Total\ CPU\ Time_{C1} = CPU\ Time_{C1A} + CPU\ Time_{C1B} + CPU\ Time_{C1C}$$

$$CPU\ Time_{C1A} = \frac{2 \times 1}{4} = \frac{1}{2}$$

$$CPU\ Time_{C1B} = \frac{1 \times 2}{4} = \frac{1}{2}$$

$$CPU\ Time_{C1C} = \frac{2 \times 3}{4} = \frac{3}{2}$$

$$Total\ CPU\ Time_{C1} = \frac{1}{2} + \frac{1}{2} + \frac{3}{2} = \frac{5}{2} = 2.5$$

COMPILER 2:

Total CPU time of Compiler 2 is the sum of CPU time of all Instruction Classes

$$Total\ CPU\ Time_{C2} = CPU\ Time_{C2A} + CPU\ Time_{C2B} + CPU\ Time_{C2C}$$

$$CPU\ Time_{C2A} = \frac{4 \times 1}{4} = 1$$

$$CPU\ Time_{C2B} = \frac{1 \times 2}{4} = \frac{1}{2}$$

$$CPU\ Time_{C2C} = \frac{2 \times 1}{4} = \frac{1}{2}$$

$$Total\ CPU\ Time_{C2} = 1 + \frac{1}{2} + \frac{1}{2} = 2$$

The CPU time required for the execution various class of instruction by compiler 2 is less than that of compiler 1, ***therefore compiler 1 is faster than compiler 2***.

4. INSTRUCTIONS: LANGUAGE OF THE COMPUTER

Instructions

- Instruction Set

1. Operations of the Computer Hardware

- Only one operation
- Exactly Three Variables

Design Principle 1: Simplicity Favours Regularity

2. Compiling C statements

An **INSTRUCTION** is a command given to a computer hardware. The set of all instructions understood by a specific computer architecture is called as **INSTRUCTION SET OF THE ARCHITECTURE.**

Each architecture has its own instruction set, but the similarity between the instructions are very high because of the reason that all the hardware work on similar principals and the basic instructions processed similarly.

Common goal of all the computer architects are as follows

- ✓ Find an easy language for building hardware
- ✓ Maximize Performance
- ✓ Minimize Cost and Energy
- ✓ Maintain simplicity in design

4.1. OPERATIONS OF THE COMPUTER HARDWARE

Every computer must be able to perform arithmetic operations.

Consider the following MIPS assembly language notation

```
add   a,  b, c
```

It instructs a computer to add the two variables **b** and **c** and to put their sum in **a**.

- Each MIPS arithmetic instruction performs **ONLY ONE OPERATION**
- It must always have **EXACTLY THREE VARIABLES**.

For example, If we want to calculate the sum of four variables b, c, d, e and to place the result in the variable **a**.

The following sequence of instructions adds the four variables:

```
add   a, b, c          #  The sum of b and c is placed in a
add   a, a, d          #  The sum of b, c, and d is now in a
add   a, a, e          #  The sum of b, c, d, and e is now in a
```

Thus, it takes three instructions to sum the four variables.

In MIPS assembly language **comments** always terminate at the end of a line.

Every instruction has exactly three operands, this rule is followed to keep the hardware simple

Design Principle 1: Simplicity favours regularity.

4.1.1. Compiling C statements

Consider this segment of a C program contains the five variables a, b, c, d, and e.

```
a = b + c;
d = a - e;
```

The above statement is compiled into MIPS assembly language as

```
add   a, b, c
sub   d, a, e
```

Consider a complex C program statement contains the five variables **f, g, h, i**, and **j** in the same statement,

```
f =  (g +  h) –  (i +  j);
```

The compiler must break this statement into several assembly instructions, since only one operation is performed per MIPS instruction.

The first MIPS instruction calculates the sum of **g** and **h**. We must place the result in a temporary variable, called **t0**:

```
add   t0, g, h          #  temporary variable  t0 contains g  +  h
```

Next we need to calculate the sum of **i** and **j**.

The second instruction places the sum of **i** and **j** in another temporary variable called **t1**:

```
add   t1, i, j          #  temporary variable  t1 contains i +  j
```

Next we need to subtract the second sum from the first sum and places the difference in the variable **f**,

```
sub   f, t0, t1         #  f gets t0 –  t1, which is (g +  h) –  (i +  j)
```

5. Operands of the Computer Hardware

The operands that can be used for arithmetic instructions are restricted in MIPS assembly language.

5.1. The Registers

Registers are used in hardware design, which the programmer can also use, these registers are used as operands in MIPS. The size of a register in the MIPS architecture is 32 bits. The group of 32 bits is called as **Word**.

A major difference between the variables of a programming language and registers is the limited number of registers.

The three operands of MIPS arithmetic instructions must each be chosen from one of the available 32 registers each 32-bits.

The reason to limit the number of registers is

Design Principle 2: Smaller is faster.

A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

Operands of the Computer Hardware

1. **The Registers**
   - ⇨ Word

   Design Principle 2: Smaller is faster.
2. **MIPS Conventions**
3. **Memory Operands**
   - ⇨ data transfer instructions
   - ⇨ memory address

   3.1. Load Instruction
   - ⇨ load word
   - ⇨ alignment restriction
   - ⇨ big-endian addressing

   3.2. Store Instruction
   - ⇨ store word
4. **Constant or Immediate Operands**
   - ⇨ Immediate instruction
   - ⇨ Common case fast

5.1.1. MIPS Conventions

The MIPS convention is to use two-character names following a dollar sign to represent a register.

$\$s0, \$s1, \ldots$ for **registers** that correspond to variables in C and Java programs

$\$t0, \$t1, \ldots$ for **temporary registers** needed to compile the program into MIPS instructions.

Compiling C Statements

```
f =  (g +  h) -  (i +  j);
```

The above C statement is compiled into MIPS assembly language using registers as follows

```
add   $t0, $s1, $s2     #  register  $t0 contains g  +  h
add   $t1, $s3, $s4     #  register  $t1 contains i +  j
sub   $s0, $t0, $t1     #  f gets $t0 -  $t1, which is (g +  h)-(i +  j)
```

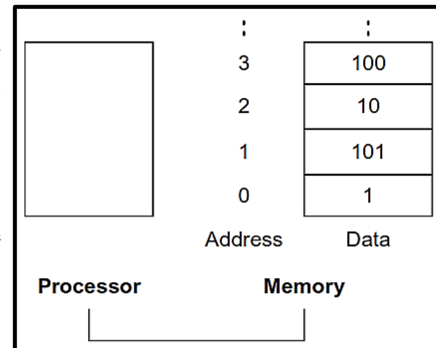
5.2. Memory Operands

Programming languages have complex data structures, such as arrays and structures. These complex data structures can contain more data elements than the total number of available registers in a computer.

The processor can keep only a small amount of data in registers data structures (arrays and structures) are kept in memory.

Arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called **DATA TRANSFER INSTRUCTIONS.**

To access a word in memory, the instruction must know the **MEMORY ADDRESS**. An address is a value used to define the location of a specific data element within a memory array.



Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. The address of the third data element is 2, and the value of Memory [2] is 10.

5.2.1. LOAD INSTRUCTION

The data transfer instruction that copies data from memory to a register is generally called **LOAD**. The actual MIPS name for this instruction is **lw**, standing for **LOAD WORD**.

The **format of the load instruction** is the name of the operation followed by the register to be loaded, then a constant and register used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address.

Consider C assignment statement

```
g = h + A[8];
```

There is a single operation in this assignment statement, but one of the operands is in memory, so we must first transfer **A[8]** to a register.

The address of this array element is the sum of the base of the array **A**, plus the number of the element. The data should be placed in a temporary register for use in the next instruction.

Consider the base of the array **A** is in **$S3**

```
lw    $t0, 32($s3)          #Temporary reg  $t0  gets  A[8]
```

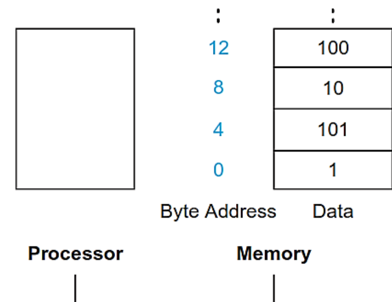
Continue the compilation as usual

```
add   $s1, $s2, $t0         #  g  =  h  +  A[8]
```

The constant in a **data transfer instruction** (32) is called the **offset**, and the register used to store the base address ($s3) is called the **base register**.

The actual MIPS addresses for the words is the byte address of the third word is 32.In MIPS, words must start at addresses that are multiples of 4. This requirement is called an **ALIGNMENT RESTRICTION.**

Some computers use the address of the leftmost or "big end" byte as the word address these computers are called **big-endian** and other computers use the rightmost or "little end" byte and these computers are called as **little-endian**. ***MIPS follows big-endian addressing***.



5.2.2. STORE INSTRUCTION

The instruction used to copy data from a register to memory is called **STORE**.

The actual MIPS name is **sw**, standing for **STORE WORD**.

The format of a store is similar to that of load: the name of the operation, followed by the register to be stored, then offset to select the array element, and finally the base register.

Consider C assignment statement

```
A[12] =  h  +  A[8];
```

Two of the operands are in memory.

The first two instructions are the same as in the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select **A[8]**, and the add instruction places the sum in **$t0**:

```
lw    $t0, 32($s3)            #  Temporary reg  $t0  gets  A[8]
add   $t0, $s2, $t0           #  Temporary reg  $t0  gets  h  +  A[8]
sw    $t0, 48($s3)            #  Stores h  +  A[8] back into  A[12]
```

5.2.3. CONSTANT OR IMMEDIATE OPERANDS

In many statements a programmer will use a constant in an operation, for example, incrementing index of an array.

In order to use constants in an operation, the constants should be placed in the memory when the program is loaded and we have to load it from memory to execute the statement.

For example, to add the constant 4 to register $s3, we could use the code

```
lw    $t0, AddrConstant4($s1)      #  $t0 =  constant  4
add   $s3, $s3, $t0                #  $s3 =  $s3 +  $t0 ($t0 ==  4)
```

An alternative to avoid the load instruction is to use the "**IMMEDIATE**" instruction. The quick add instruction with one constant operand is called **ADD IMMEDIATE** or **addi**.

To add **4** to register **$s3**, the code is

```
addi  $s3, $s3, 4                  #  $s3 =  $s3 +  4
```

Constant operands occur frequently, and by including constants inside arithmetic instructions, operations are **much faster and use less energy** than if constants were loaded from memory. This is an example of making the **COMMON CASE FAST**.

6. REPRESENTING INSTRUCTIONS

- Instructions are kept in the computer as a series of high and low electronic signals and can be represented as numbers.
- Each instruction can be represented as a field of binary number, this representation is called the **instruction format**. Once the MIPS instructions are converted as binary numbers then the instruction is called as **Machine Instruction**.
- All of the MIPS instructions are of **32 BITS LONG** and can be classified into any one of the three formats or types. They are,
  - R-type or R-format
  - I-type or I-format
  - J-type or J-format

Representing Instructions

- ➡ R-type or R-format
- ➡ I-type or I-format
- ➡ J-type or J-format

1. R-type or R-format

- ➡ 3 operands as registers

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

2. I-type or I-format

- ➡ 16 bit constant or address field

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

3. J-type Instructions

- ➡ Jump & goto

op	Jump Address
6 bits	28 bits

6.1. R-TYPE INSTRUCTIONS

The R-type instructions are those have all the **3 OPERANDS AS REGISTERS**. These instructions follow the format shown below,

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

- `op` represents Operation, it is called as ***opcode***.
- `rs` is the first register source operand
- `rt` is the second source operand
- `rd` is the destination register to store the result
- `shamt` is the shift amount
- `funct` is the function field, it specifies the variant of the operation.

Example: Translate the MIPS assembly instruction into Machine Instruction

```
add $t0, $S1, $S2
```

Solution:

The instruction is a R-type instruction, the format of the R-type instruction is

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

op – It is add, the numeric code for **add** is **0**
rs – It is the first source operand **$S1**, the register number of **$S1** is **17**
rt – It is the second source operand **$S2**, the register number of **$S2** is **18**
rd – It is the **$t0**, the register number of **$t0** is **8**
shamt – Shift amount is not used, set as **0**
funct – function code for add is **32**

0	17	18	8	0	32

The above decimal representation should be converted as binary in order to get the machine instruction.

000000	10001	10010	01000	00000	100000

6.2. I-TYPE INSTRUCTION

I-type instructions are those that have **TWO REGISTERS AND ONE CONSTANT** as their operands. These instructions follow the following format,

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

The similarity between the R-type and I-type instructions are that the length of both the type of instructions are 32 bits. The description of the fields are as follows.

The first 3 fields are same as the R-type instruction, since the I-type instruction does not have a second source operand register rt is used as the destination register and the last three fields of the R-type instruction is combined together as a **16 bit constant or address field**.

Example: Translate the MIPS assembly instruction into Machine Instruction

addi $t0, $S1, 20

Solution:
The instruction is a R-type instruction, the format of the **R-TYPE** instruction is

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

op – It is add immediate, the numeric code for **addi** is **8**
rs – It is the source operand **$S1**, the register number of **$S1** is **17**
rt – It is the destination operand **$t0**, the register number of **$t0** is **8**
The constant in the above instruction is **20**

8	17	8	20

The above decimal representation should be converted as binary in order to get the machine instruction.

001000	10001	01000	0000000000010100

Some of the most commonly used instructions and their values are

Instruction	Format	op	rs	rt	rd	shamt	funct	address
Add	R	0	reg	reg	reg	0	32ten	n.a.
sub(subtract)	R	0	reg	reg	reg	0	34ten	n.a.
add immediate	I	8ten	reg	reg	n.a.	n.a.	n.a.	constant
lw(load word)	I	35ten	reg	reg	n.a.	n.a.	n.a.	address
sw(store word)	I	43ten	reg	reg	n.a.	n.a.	n.a.	address

6.3. J-TYPE INSTRUCTIONS

J-type instructions are instructions that are used by MIPS to transfer the control to a specific part of the program. The name of the instruction used for this purpose is called **`jump`**, it is the equivalent of C language`s **`goto`** statement.

J-type instruction is also similar to R-type and I-type instructions, it is of the same size 32 bits. Even though the designers of MIPS could not use the same instruction format for all instructions, they have used the ***same number of bits for all instruction*** following the design principle.

Good Design Demands Good Compromises

The J-type instruction follows the format

op	Jump Address
6 bits	28 bits

Example: Translate the MIPS assembly instruction into Machine Instruction

`j Exit`

`Exit` is the label to which the control has to jump, consider it is at the location **`1200`**

Solution:

The instruction is a J-type instruction, the format of the J-type instruction is

op	Jump Address
6 bits	28 bits

`op` – It is Jump, the numeric code for **`j`** is **`2`**
The jump address in the above instruction is **`1200`**

2	1200

The above decimal representation should be converted as binary in order to get the machine instruction.

000010	0000000000000000010010110000

7. LOGICAL OPERATIONS

In the beginning the operations performed where limited to an entire word, but later operations had to be performed on individual bits. The following table shows the MIPS supported logical operations along with the C and Java equivalent of those operations.

Logical Operation	C Operation	Java Operation	MIPS Instruction
Shift left	<<	<<	sll

Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

7.1. Shift Operations

The first two instructions shift left and shift right are classified as **SHIFT** operations. They are used to move the bits to right or left and to fill the empty bits with **0**.

Example: Consider two registers, $t2 and $S0. The register $S0 contains a value 9. What would be the content of $t2 after the following instructions are executed?

a) sll $t2, $S0,4
b) srl $t2, $S0, 2

Solution:
sll $t2, $S0, 4 is the equivalent of the $t2 = $S0 << 4
$S0 contains the value 9
It is represented in binary as

0000	0000	0000	0000	0000	0000	0000	1001

The last 4 bits had to be moved left,

0000	0000	0000	0000	0000	0000	1001	0000

The above binary value is equal to 144.

Therefore the value stored in $t2 is 144

srl $t2, $S0,2 is the equivalent of $t2 = $S0 >> 2
$S0 contains the value 9
It is represented in binary as

0000	0000	0000	0000	0000	0000	0000	1001

The last 2 bits had to be moved right,

0000	0000	0000	0000	0000	0000	0000	0010

The above binary value is equal to 2

Therefore the value stored in $t2 is 2

The actual name of the two MIPS shift instructions are called shift left logical (**sll**) and shift right logical (**srl**).

sll uses the shift amount field in the R-types instruction.

```
sll $t2,$s0,4
```

The machine language of the above instruction is

op	Rs	rt	rd	shamt	funct
0	0	16	10	4	0

`shamt` field used in the ***shift instructions***, stands for shift amount.

The opcode of `sll` is **0** in both the ***op*** and ***funct*** fields are set as **0**, ***rd*** contains 10 (register $t2), ***rt*** contains 16 (register $s0), and `shamt` contains 4. The ***rs*** field is unused and thus is set as **0**.

Shift left logical provides a bonus benefit. Shifting left by ***i*** bits gives the same result as multiplying by *2i*.

7.2. Logical AND Operation

AND is a logical bit-by-bit operation with two operands that calculates a 1 only if there is a 1 in both operands.

For example, if register ***$t2*** contains

```
0000 0000 0000 0000 0000 1101 1100 0000
```

And register $t1 contains

```
0000 0000 0000 0000 0011 1100 0000 0000
```

Then, after executing the MIPS instruction

```
AND   $t0, $t1, $t2           # reg $t0 = reg $t1 & reg $t2
```

The value of register $t0 would be

```
0000 0000 0000 0000 0000 1100 0000 0000
```

AND can be applied to set a bit pattern, to force 0s where there is a 0 in the bit pattern. Such a bit pattern made using AND is called a ***mask***, the mask is used to “conceal” some bits.

7.3. Logical OR Operation

It is a bit-by-bit operation that places a 1 in the result if either operand bit is a 1.

For example, if register ***$t2*** contains

```
0000 0000 0000 0000 0000 1101 1100 0000
```

And register ***$t1*** contains

```
0000 0000 0000 0000 0011 1100 0000 0000
```

The result of the MIPS instruction

```
OR    $t0, $t1, $t2           # reg $t0 = reg $t1 | reg $t2
```

The value in the register ***$t0***:

```
0000 0000 0000 0000 0011 1101 1100 0000
```

7.4. Logical NOT Operation

NOT takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. Using our prior notation, it calculates $\bar{x}$

The designers of MIPS decided to include the instruction NOR (NOT OR) instead of NOT. If one operand is zero, then it is equivalent to

```
NOT: A NOR 0 = NOT (A OR 0) = NOT (A)
```

For example, register ***$t1*** contains

```
0000 0000 0000 0000 0011 1100 0000 0000
```

Register ***$t3*** has the value ***0***, the result of the MIPS instruction

```
NOR   $t0, $t1, $t3            # reg $t0 = ~ (reg $t1 | reg $t3)
```

The value in register ***$t0***:

```
1111 1111 1111 1111 1100 0011 1111 1111
```

Constants are rare for NOR, since its main use is to invert the bits of a single operand; thus, the MIPS instruction set architecture has no immediate version of NOR.

8. DECISION MAKING

The difference between a computer and a simple calculator is its ability to make decisions. Decision making is commonly represented in programming languages using the "**if**" statement, sometimes combined with "**goto**" statements and labels.

Decision Making

1. Conditional Branch

1.1. Branch-if-Equal Statement

```
beq register1, register2, L1
```

1.2. Branch-if-not-Equal Statement

```
bne register1, register2, L1
```

2. Loop Instruction

```
Loop: sll $t1,$s3,2
```

3. Case/Switch Statement

- ⇨ Jump address table or jump table
- ⇨ Jump register (JR)

8.1. Conditional Branch

An instruction that compares two values and transfers control to a new address in the program based on the outcome of the comparison.

MIPS assembly language includes two decision-making instructions, similar to an "**if**" statement with a "**goto**".

8.1.1. Branch-if-Equal Statement

```
beq register1, register2, L1
```

This instruction means go to the statement labelled L1 if the value in **register1** equals the value in **register2**. The mnemonic **beq** stands for ***branch if equal***.

8.1.2. Branch-if-not-Equal Statement

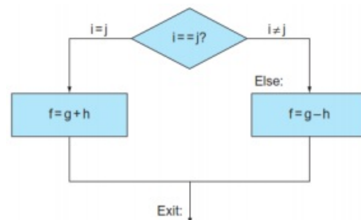
```
bne register1, register2, L1
```

It means go to the statement labelled L1 if the value in **register1** does not equal the value in **register2**. The mnemonic **bne** stands for ***branch if not equal***.

Example: In the following code segment, f, g, h, i, and j are variables. If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C if statement?

```
if (i == j)
  f = g + h;
else
  f = g – h;
```

Solution:



The first expression compares for equality, so it looks like it is better to use branch-if-equal instruction (**beq**).

In general, the code will be more efficient if we test for the opposite condition so we use the branch if not equal instruction (**bne**):

```
bne $s3,$s4,Else   # go to Else if i ≠ j
```

The next assignment statement performs a single operation,

```
add $s0,$s1,$s2    # f = g + h (skipped if i ≠ j)
```

Now we need to go to the end of the if statement, Unconditional branch statement (Jump instruction) can be used,

```
J Exit   #goto Exit
```

The assignment statement in the else portion of the if statement can be compiled into a single instruction. We have to append the label **Else** to this instruction.

```
Else: sub $s0,$s1,$s2  # f = g – h (skipped if i = j)
```

We need to end the if-then-else statement using the **Exit** label

```
Exit:
```

8.2. Loop Instruction

Loops are used to iterate a particular computational statement in a programming language. Iterations in MIPS assembly language can be carried out as follows,

Consider the below mentioned iteration

```
while (save[i] == k)
      i += 1;
```

Assume that i and k correspond to registers *$s3* and *$s5* and the base of the array save is in *$s6*.

The first step is to load `save[i]` into a temporary register. Before we can add `i` to the base of array save to form the address, we must multiply the index `i` by 4 due to the byte addressing problem.

Instead of multiplying index `i` by 4, we can use shift left logical, shifting left by 2 bits is equal to multiplying by 2^2 or 4.

```
Loop: sll $t1,$s3,2      # Temp reg $t1 = i * 4
```

To get the address of `save[i]`, we need to add `$t1` and the base of save in `$s6`:

```
add $t1,$t1,$s6          # $t1 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
lw $t0,0($t1)            # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if `save[i] ≠ k`:

```
bne $t0,$s5,Exit         # go to Exit if save[i] ≠ k
```

The next instruction adds 1 to `i`:

```
addi $s3,$s3,1           # i = i + 1
```

The end of the loop branches back to the while test at the top of the loop. We add the Exit label after it,

```
j Loop                   # go to Loop
Exit:
```

The test for equality or inequality is probably the most popular test, but sometimes it is useful to see if a variable is less than another variable.

A ***for*** loop tests if the index variable is less than **`0`**. The MIPS instruction used to test is called ***set on less than***, or ***`slt`***. If the test is true then a third register is set to ***`1`*** else it is set to ***`0`***.

```
slt $t0, $s3, $s4        # $t0 = 1 if $s3 < $s4 else $t0 = 0.
```

Register ***`$t0`*** is set to ***`1`*** if the value in register ***`$s3`*** is less than the value in register ***`$s4`*** else, register ***`$t0`*** is set to ***`0`***.

Constant operands are used in most of the comparisons, so there is an immediate version of the ***`set on less than`*** instruction. To test if register **`$s2`** is less than the constant **`10`**, we can just write

```
slti $t0,$s2,10        # $t0 = 1 if $s2 < 10
```

MIPS compilers use the **slt, slti, beq, bne,** and **0** to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal.

A binary number with a 1 in the most significant bit can represents a negative number, it is less than any positive number. Positive numbers have a 0 in the most significant bit. In unsigned integers a 1 in the most significant bit represents a number that is larger than any number that begins with a 0.

To handle unsigned numbers MIPS provides the following alternatives,

- **Set on less than (slt)** and **set on less than immediate (slti)** work with signed integers
- **Set on less than unsigned (sltu)** and **set on less than immediate unsigned (sltiu)** works with unsigned integers

Suppose register $s0 has the binary number
1111 1111 1111 1111 1111 1111 1111 1111

and that register $s1 has the binary number
0000 0000 0000 0000 0000 0000 0000 0001

What are the values of registers $t0 and $t1 after these two instructions?

```
slt $t0, $s0, $s1       # signed comparison
sltu $t1, $s0, $s1      # unsigned comparison
```

Solution:
The content of the register $S0 has a 1 in the most significant bit, in case of a signed number comparison it would be treated as a negative number, so the content of $S0 is less than the content of the register $S1.

While using a unsigned number comparison the content of $S0 is larger than the content of $S1.

The values of the registers $t0 and $t1 would be 0 and 1 respectively

8.3. CASE/SWITCH STATEMENT

Switch Case statement allows the programmer to select one of many choices based on a single value. The simplest way to implement switch is by using a chain of ***if-then-else*** statements.

A more efficient alternative is to encode as a table of addresses, called a ***jump address table*** or ***jump table***, the program indexes into the table and then jump to the correct instruction to be executed. The jump table is an array of words containing addresses that correspond to labels in the code. The program loads the correct address and label from the jump table into a register. It then needs to jump using the address in the register.

MIPS include a **JUMP REGISTER (JR)** instruction, It is an unconditional jump to the address specified in the register.

9. ADDRESSING MODES OF MIPS

Addressing Modes of MIPS

1. 32-Bit Immediate Operands
   - ➩ Load Upper Immediate
   - ➩ Assembler Temporary
2. **MIPS Addressing Mode**
   - **2.1. Immediate addressing**
   - **2.2. Register addressing**
   - **2.3. Base or displacement addressing**
   - **2.4. PC-relative addressing**
   - **2.5. Pseudodirect addressing**

All MIPS instructions are 32 bits long in order to keep the hardware as simple as possible, sometimes it is needed to have a 32-bit constant or 32-bit address in the instruction.

9.1. 32-BIT IMMEDIATE OPERANDS

The MIPS instruction set includes the instruction **load upper immediate (lui)**. It is used to set the upper 16 bits of a constant in a register, the lower 16 bits of the constant is set using **ori** instruction.

The machine language version of

```
lui $t0, 255 #$t0 is register 8:
```

001111	00000	01000	0000 0000 1111 1111

Contents of register $t0 after executing

```
lui $t0, 255:
```

0000 0000 1111 1111	0000 0000 0000 0000

Example:
What is the MIPS assembly code to load this 32-bit constant into register $s0?
`0000 0000 0011 1101 0000 1001 0000 0000`

First, we would load the upper 16 bits, which is 61 in decimal, using lui:
```
lui $s0, 61         # 61 decimal = 0000 0000 0011 1101 binary
```

The value of register $s0 afterward is
`0000 0000 0011 1101 0000 0000 0000 0000`

The next step is to insert the lower 16 bits, whose decimal value is 2304:
```
ori $s0, $s0, 2304  # 2304 decimal = 0000 1001 0000 0000
```

The final value in register $s0 is the desired value:
`0000 0000 0011 1101 0000 1001 0000 0000`

The compiler or the assembler must break large constants into pieces and then reassemble them into a register. If this is done by the assembler, then the assembler must have a temporary register available in which to create the long values. This a use of the register **`$at (assembler temporary)`**, which is reserved for the assembler.

9.2. MIPS ADDRESSING MODE

Multiple forms of addressing are generically called addressing modes. The MIPS addressing modes are the following,

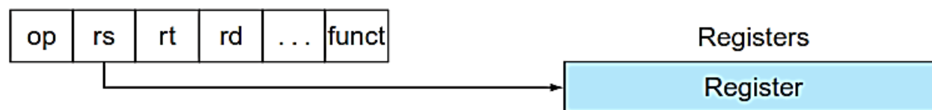
9.2.1. IMMEDIATE ADDRESSING

The operand is a constant within the instruction itself

```
add $S3, $t0, #10
```

9.2.2. REGISTER ADDRESSING

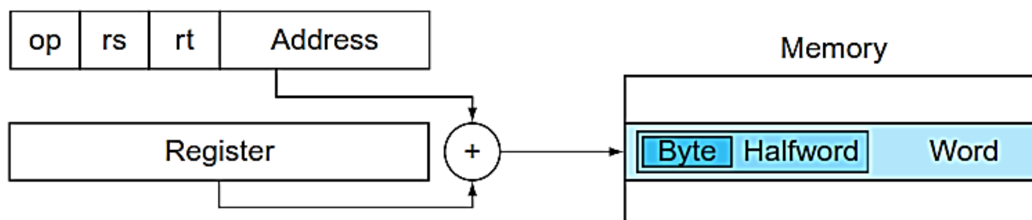
The operand is a register




```
add $S3, $S4, $t0
```

9.2.3. BASE OR DISPLACEMENT ADDRESSING

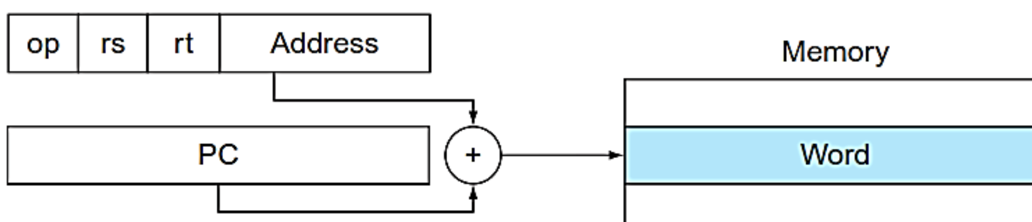
The operand is at the memory location whose address is the sum of a register and a constant in the instruction




```
lw $S5,16($S2)
```

9.2.4. PC-RELATIVE ADDRESSING

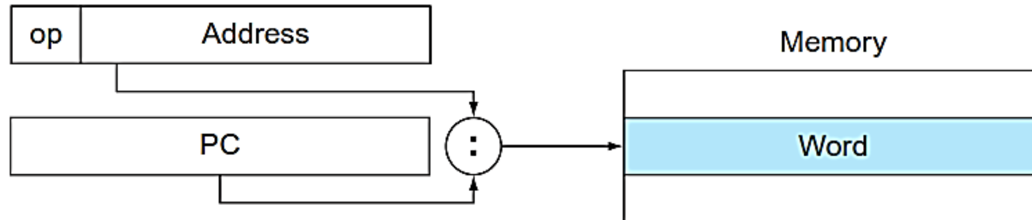
The branch address is the sum of the PC and a constant in the instruction



```
beq $t0,$t3,Label
```

9.2.5. PSEUDODIRECT ADDRESSING

The jump address is the 26 bits of the instruction concatenated with the upper bits of the PC




```
j Exit
```

Assume a two address format specified as source, destination. Examine the following sequence of instructions and explain the addressing modes used and the operation done in every instruction.

i. Move (R5)+, R0
ii. Add (R5)+, R0
iii. Move R0, (R5)
iv. Move 16(R5), R3
v. Add #40, R5

Move (R5)+, R0

Register-Indirect with increment Addressing mode, Increments the contents of R5 by 1 and loads it to R0.

Add (R5)+ R0

Register-Indirect with increment Addressing mode, Increments the contents of R5 by 1, adds the result to R0 and stores the result in R0

R0 = ((R5)+1)+R0.

Move R0, (R5)

Register Differed addressing mode, content of R0 is moved to the memory location of R5.

Move 16(R5), R3

Displacement Addressing mode, M(16+R5) is moved to the R3 register

Add #40, R5

Immediate Addressing Mode, R5= R5+40

PART – A

COMPONENTS OF A COMPUTER SYSTEM

1. What are the five classic components of a computer?[April/May-2017][Nov/Dec-2017]

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor.

PROCESSOR TECHNOLOGY

2. What is meant by Stored Program Concept? [May/June 2007]

Today's computers are built on two key principles:

1. Instructions are represented as numbers.
2. Programs are stored in memory to be read or written, just like data.

These principles lead to the stored-program concept. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems.

3. What are the features of Von-Newmann Model?

The main features of Von-Newmann model are,

1. It uses stored programming concept
2. Memory is addressed by location – irrespective of type of data it contains
3. Instructions are executed sequentially.

4. What do you mean by Von-Newmann Bottleneck?

By using stored programming concept the performance of the processor is only as good as the performance of the memory, this limits the maximum utilization of the available processing capability. This condition is called as Von-Newmann Bottleneck

5. Distinguish pipelining from parallelism. [April/May 2015]

Pipelining is the process of making the functional units of the CPU independent and this helps in increasing the throughput of the processor.

Parallelism is the process of executing more than one instruction in parallel, this requires redundant hardware for functional units, whereas pipelining does not require redundancy, Parallelism decreases execution time.

PERFORMANCE OF A COMPUTER

6. What is defined as performance of a computer?

Performance of a computer is the level of useful work accomplished by a computer compared to the time and resources used.

- Short response time for a given piece of work
- High throughput.

7. Define Response time and Throughput

Response time is the actual execution time of a single program or an individual task.

Throughput is also called as Bandwidth, it is the number of tasks completed per unit time.

8. What is Execution Time?

It is also called as response time. It is the total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, etc.

9. What is CPU Execution Time?

It is also called as CPU time, it is the actual amount of time the CPU spends computing for a particular task.

10. Define CPI

CPI is Clock Cycles per Instruction. It is the average number of clock cycles each instruction takes to execute. CPI is useful in comparing two different implementations of the same instruction set architecture.

INSTRUCTIONS – OPERATIONS AND OPERANDS

11. Define – ISA

The instruction set architecture, or simply architecture of a computer is the interface between the hardware and the lowest-level software. It includes anything programmers need to know to make a binary machine language program work correctly, including instructions, I/O devices, and so on.

12. Define – ABI

Typically, the operating system will encapsulate the details of doing I/O, allocating memory, and other low-level system functions so that application programmers do not need to worry about such details. The combination of the basic instruction set and the operating system interface provided for application programmers is called the application binary interface (ABI).

13. What are the fields in an MIPS instruction?

MIPS fields are

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

Where,

op: Basic operation of the instruction, traditionally called the opcode.
rs: The first register source operand.
rt: The second register source operand.
rd: The register destination operand. It gets the result of the operation.
shamt: Shift amount.
funct: Function.

14. Write an example for immediate operand.

The quick add instruction with one constant operand is called add immediate or addi. To add 4 to register $s3, we just write

```
addi $s3,$s3,4        #$s3 = $s3+4
```

REPRESENTING INSTRUCTIONS

15. Define Opcode.

Opcode is abbreviated from operation code, it is the portion of a machine language instruction that specifies the operation to be performed.

16. What does the design principle "Good Design Demands Good Compromises" mean?

Even though the designers of MIPS could not use the same instruction format for all instructions, they have used the same number of bits for all instruction in order to reduce the complexity of the design, based on the design principle "Good Design Demands Good Compromises".

ADDRESSING AND ADDRESSING MODES

17. What are the various addressing modes supported by MIPS?

The MIPS addressing modes supported by MIPS are:

1. Immediate addressing
2. Register addressing
3. Base or displacement addressing
4. PC-relative addressing
5. Pseudo direct addressing

18. Why immediate addressing mode is considered as an important addressing mode?

In Immediate Addressing mode, the operand is a constant within the instruction itself, this eliminates the need for loading the constant operand into a memory location and using the memory address to perform the operation. By doing so the ***total number of individual instructions required is reduced***, for this reason Immediate Addressing is considered to be important and more commonly used.

19. Brief about Relative Addressing mode with an example. [Nov/Dec 2014]

In relative addressing mode, the branch address is the sum of the program counter and a constant in the instruction.

```
beq $S0,$S3,Label
```

20. State the need for indirect addressing.[April/May 2017]

PART – B

COMPONENTS OF A COMPUTER SYSTEM

1. **Explain the various components of computer System with neat diagram [Nov/Dec 2014-16M][April/May-2018-8M**].*[Refer Pg No:1.1]*

PERFORMANCE OF A COMPUTER

2. **Discuss in detail the various measures of performance of a computer [Nov/Dec 2014 – 8M]** *[Refer Pg No:1.6]*
3. **Consider the computer with three instruction classes and CPI measurement as given below and instruction count for each instruction class for the same program from two different compilers are given. Assume that the computer`s clock rate is 4GHz. Which code sequence will execute faster according to execution time**

Code from	CPI for this Instruction Class		
	A	B	C
CPI	1	2	3
Code from	Instruction Count for each class		
	A	B	C
Compiler 1	2	1	2
Compiler 2	4	1	1

[Nov/Dec 2014 – 6M]

INSTRUCTIONS – OPERATIONS AND OPERANDS

4. **Explain in detail various operations of a computer Hardware by providing relevant MIPS instructions.[Nov/Dec-2017-8M]** *[Refer Pg No:1.11]*

[OR]

Explain in detail how a high-level programming statement are compiled into operands understandable by MIPS.

5. **Explain various operands of computer Hardware and how they can be accessed from the memory.[Nov/Dec-2017-8M]** *[Refer Pg No:1.12]*

REPRESENTING INSTRUCTIONS

6. **Explain how the high-level language statements are represented as assembly language instructions in a computer.** *[Refer Pg No:1.16]*

[OR]

Explain the difference between how the humans provide instructions to a computer and the computer views it. *[Refer Pg No:1.16]*

[OR]

7. **Discuss about various techniques to represent instructions in a computer system [April / May 2015 – 16M] [Nov/Dec-2017-13M]** *[Refer Pg No:1.16]*

LOGICAL & CONTROL OPERATIONS

8. **Discuss various logical operations that are executable by a MIPS processor.** *[Refer Pg No:1.18]*
9. **What are the various control operations included in the MIPS ISA? Explain in detail.** *[Refer Pg No:1.25]*

[OR]

Explain in detail various decision making instructions that are executable by a MIPS processor. *[Refer Pg No:1.25]*

ADDRESSING AND ADDRESSING MODES

10. **Define Addressing mode and explain the basic MIPS addressing modes with an example for each.** *[Refer Pg No:1.21]*

[OR]

What is the need for addressing in a computer system? Explain different addressing modes with example [April/May 2015 – 16M] *[Refer Pg No:1.21]*

11. **Assume a two address format specified as source, destination. Examine the following sequence of instructions and explain the addressing modes used and the operation done in every instruction.**
    1. **Move (R5)+, R0**
    2. **Add (R5)+, R0**
    3. **Move R0, (R5)**
    4. **Move 16(R5), R3**
    5. **Add #40, R5**

[Nov/Dec 2014 – 10M]