Computer Organization (CO)

UNIT-I

Syllabus: Basic structure of Computers:

- Functional Units
- Basic Operational concepts
- Bus structures
- System Software
- Performance
- The history of computer development

TEXT-BOOK:

1. Computer Organization, Carl Hamacher, Zvonks Vranesic, Safea Zaky, 5th Edition, McGraw Hill

(1) Computer types:

→ Computer is a fast electronic calculating machine that accepts digitized input information, processes it according to a list of internally stored instructions and produces the resulting output information.

→ The list of instructions is called a Computer program, and the internal storage is called computer memory.

→ The different computer types are,

(i) Desktop computers
- personal Computer

(ii) notebook Computers

(iii) WorkStations

(iv) Enterprise Systems

(v) Servers

(vi) Super Computers.

→ Desktop Computers have processing and storage units, visual display and audio output units, and a keyboard that can all be located easily on a

home (or) office desk.

→ The ~~most~~ storage media includes,
- hard disks
- CD-ROMs etc.

→ The most common form of desktop computers is personal computer, which has found wide use in homes, schools, and business offices.

→ Portable notebook computers are a compact version of the personal computer with all of these components packaged into a single unit the size of a thin briefcase.

→ Workstations have more computational power than personal computers, used in engineering applications, especially for interactive design work.

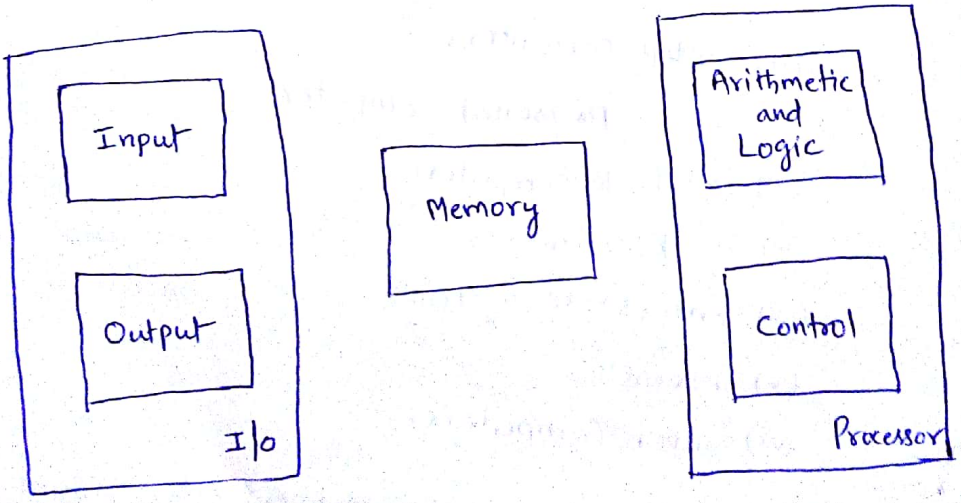
→ Enterprise Systems (or) Mainframes, are used for business data processing in medium to large corporations that require much more computing power and storage capacity than workstations can provide.

→ Servers contain sizable database storage units and are capable of handling large volumes of requests to access the data.

→ Super computers are used for the large-scale numerical calculations required in applications such as weather forecasting and aircraft design and simulation.

(2) Functional Units:

→ Basic Functional Units of a Computer is depicted as,



→ A computer consists of five functionally independent main parts
- input
- memory
- Arithmetic and Logic
- output and
- control unit.

→ Instructions (or) Machine Instructions, are explicit commands that
- govern the transfer of information within a computer as well as between the computer and its I/O devices.
- Specify the arithmetic and logic operations to be performed.

(i) Input Unit:

→ Computers accept coded information through input units, which read the data.
→ The most well-known input device is the Keyboard.
→ The other Input devices are,
- Mouse
- Joystick
- Scanner
- Touch Screen
- Light Pen, .. etc.

(ii) Memory Unit:

→ The function of the memory unit is to store programs and data.
→ There are two classes of storage
- Primary
- Secondary

→ Primary Storage is a fast memory that operates at electronic speeds.
→ Programs must be stored in the memory while they are being executed.
→ The Primary memory of a Computer is RAM (Random Access Memory)
→ The Secondary Storage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently
→ The different Secondary Storage devices are,
- HDD (Hard disk drive)
- FDD (Floppy disk drive)
- Magnetic disks and taps
- Optical disks (CD-ROM) - compact disk Read Only Memory

(iii) Arithmetic and Logic Unit :

→ Most computer operations are executed in the arithmetic and Logic Unit (ALU) of the Processor.

→ Consider an example, Suppose two numbers located in the memory are to be added.

→ They are brought into the processor, and the actual addition is carried out by the ALU.

→ The sum may then be stored in the memory (or) retained in the processor for immediate use.

(iv) Output Unit:

→ The output unit is the counter part of the input unit.

→ Its function is to send processed results to the outside world.

→ The different output devices are,

- Monitor
- Printer
- Plotter

(v) Control Unit:

→ All activities inside the machine are directed and controlled by the control unit.

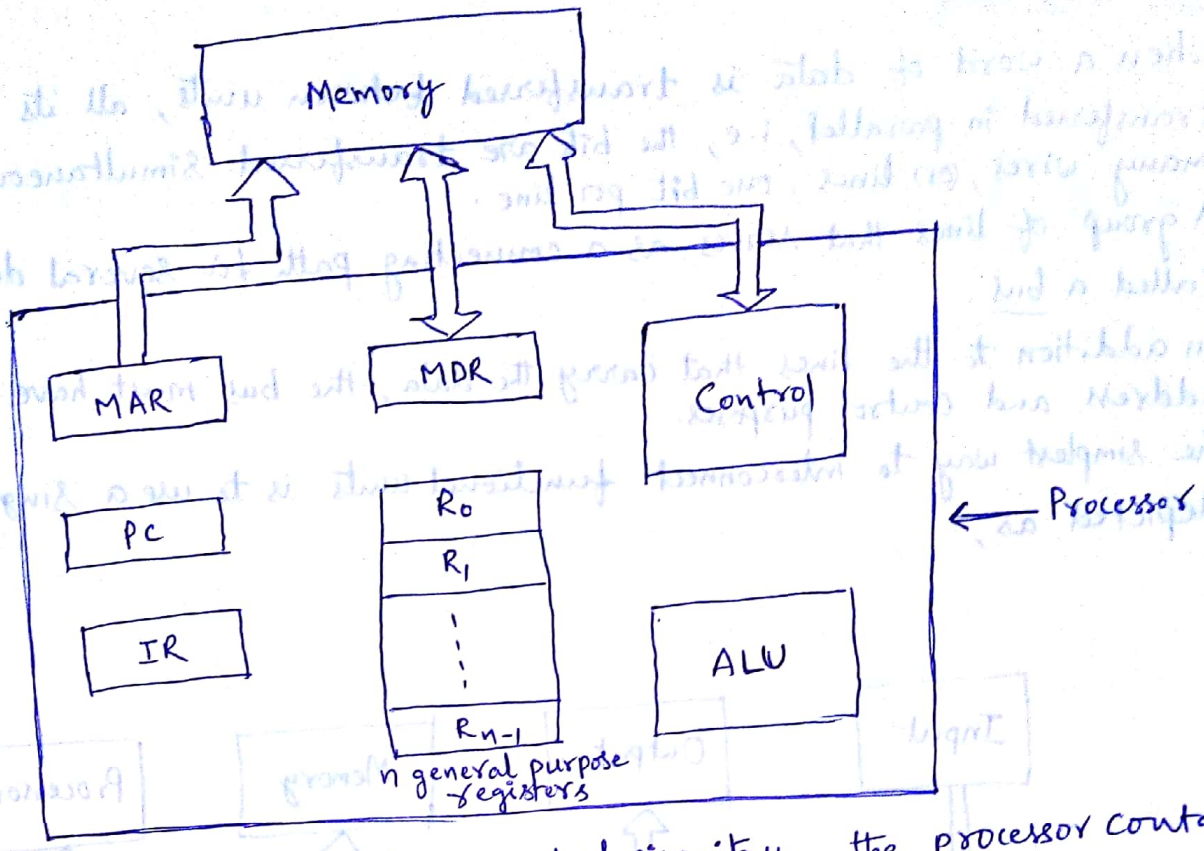
→ The control unit is effectively the nerve center that sends control signals to other units and senses their states.

(3) Basic Operational Concepts

→ 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.

→ The connections between the processor and the memory is depicted as,



→ In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes.

→ The Instruction Register (IR) holds the instruction that is currently being executed.

→ Its output is available to the control circuits, which generates the timing signals that control the various processing elements involved in executing the instruction.

→ The Program Counter (PC) keeps track of the execution of a program and it contains the memory address of the next instruction to be fetched and executed.

→ During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed.

→ The PC points to the next instruction that is to be fetched from memory.

→ It has also n-general purpose registers R_0 through R_{n-1}

→ Finally, two registers facilitate communication with the memory

→ These are the Memory Address Register (MAR) and Memory Data Register (MDR)

→ The MAR holds the address of the location to be accessed.

→ The MDR contains the data to be written into (or) read out of the addressed location.

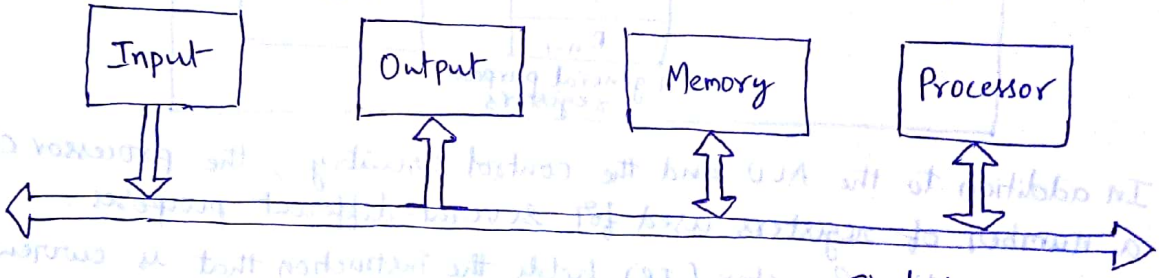
(4) Bus Structures :

→ When a word of data is transferred between units, all its bits are transferred in parallel, i.e, the bits are transferred simultaneously over many wires, (or) lines, one bit per line.

→ A group of lines that serves as a connecting path for several devices is called a bus.

→ In addition to the lines that carry the data, the bus must have lines for address and control purposes.

→ The simplest way to interconnect functional units is to use a single bus is depicted as,



Single bus structure

→ The bus can be used for only one transfer at a time, only two units can actively use the bus at any given time.

→ Bus control lines are used to arbitrate multiple requests for use of the bus.

→ The main virtue of the single-bus structure is its low cost and its flexibility for attaching peripheral devices.

→ Systems that contain multiple buses achieve more concurrency in operations by allowing two (or) more transfers to be carried out at the same time

→ This leads to better performance but at an increased cost.

(5) Software : (System Software)

→ In order for a user to enter and run an application program, the computer must already contain some system software in its memory.

→ System software is a collection of programs that are executed as needed to perform functions such as,

- Receiving and interpreting user commands.
- Entering and Editing application programs and storing them as files in secondary storage devices.
- Managing the storage and retrieval of files in secondary storage devices.

- Running standard application programs such as processors, spreadsheets, (or) games, with data supplied by the user.
- Controlling I/o units to receive input information and produce output results.
- Translating programs from source form prepared by the user into object form consisting of machine instructions.
- Linking and running user-written application programs with existing standard library routines, such as numerical computation packages.

→ System Software is thus responsible for the coordination of all activities in a Computing System.

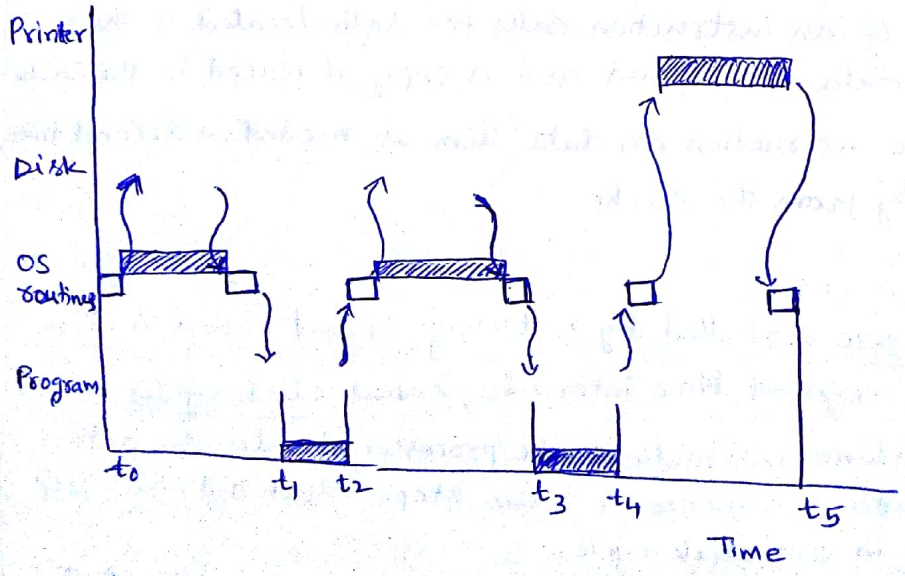
→ Application Programs are usually written in a high-level programming language, such as C, C++, Java, in which the programmer specifies mathematical (or) text-processing operations.

→ Operating System is a large program, (or) actually a collection of routines, that is used to control the sharing of and interaction among various computer units as they execute application programs.

→ The OS routines perform the tasks required to assign computer resources to individual application Programs.

→ A System program that all programmers use is a text editor. It is used for entering and editing application programs.

→ User Program and OS routine sharing of the processor is depicted as,



→ The Operating System manages the concurrent execution of several application programs called as multiprogramming (or) multitasking

⑥ Performance:

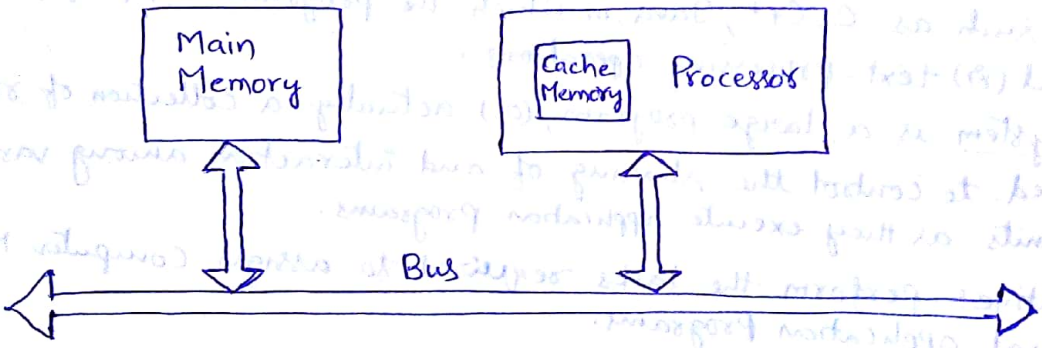
→ The most important measure of the performance of a computer is how quickly it can execute programs.

→ The speed with which a computer executes programs is affected by the design of its hardware and its machine language instructions.

→ To represent the performance of the processor, we should consider only the periods during which the processor is active.

→ These are the periods labeled Program and OS routines. The sum of these periods as the processor time needed to execute the program.

→ The processor cache is depicted as



→ At the start of execution, all program instructions and the required data are stored in the main memory

→ As execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache.

→ When the execution of an instruction calls for data located in the main memory, the data are fetched and a copy is placed in the cache

→ Later, if the same instruction (or) data item is needed a second time, it is read directly from the cache.

(i) Processor clock:

→ Processor circuits are controlled by a timing signal called a clock.

→ The clock defines regular time intervals, called clock cycles.

→ To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps, such that each step can be completed in one clock cycle.

→ The length P of one clock cycle is an important parameter that affects processor performance.

→ Its inverse is the clock Rate, $R = 1/P$, which is measured in cycles per second.

→ In standard electrical engineering terminology, the term cycles per second is called Hertz.

(ii) Basic Performance Equation:

→ Let T be the processor time required to execute a program

→ Assume that complete execution of the program requires the execution of N machine language instructions.

→ The number N is the actual number of instruction executions, and is not necessarily equal to the number of machine instructions in the object program.

→ Suppose that the average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle.

→ If the clock rate is R cycles per second, the program execution time is given by

$$T = \frac{N \times S}{R}$$

This is referred as Basic Performance Equation

(iii) Pipelining and SuperScalar Operation:

→ A substantial improvement in performance can be achieved by overlapping the execution of successive instructions, using a technique called pipelining.

→ Consider the instruction

Add R_1, R_2, R_3

- which adds the contents of registers R1 and R2, and places the sum into R3.
- The contents of R1 and R2 are first transferred to the inputs of the ALU.
- After the add operation is performed, the sum is transferred to R3.
- The processor can read the next instruction from the memory while the addition operation is being performed.
- Then if that instruction also uses the ALU, its operands can be transferred to the ALU inputs at the same time that the result of the Add instruction is being transferred to R3.

→ A higher degree of concurrency can be achieved if multiple instruction pipelines are implemented in the processor.
→ This means that multiple functional units are used, creating parallel paths through which different instructions can be executed in parallel.
→ With such an arrangement, it becomes possible to start the execution of several instructions in every clock cycle. This mode of operation is called Superscalar execution.

(iv) Clock Rate:

→ There are two possibilities for increasing the clock rate, R.
- First, improving the Integrated-Circuit (IC) technology makes logic circuits faster, which reduces the time needed to complete a basic step
  This allows the clock period, P, to be reduced and the clock rate R, to be increased.
- Second, reducing the amount of processing done in one basic step also makes it possible to reduce the clock period P.

(v) CISC and RISC:

→ Simple instructions require a small number of basic steps to execute
→ Complex instructions involve a large number of steps.
→ A key consideration in comparing the two choices is the use of pipelining.
→ CISC and RISC are used for complex instructions.
→ CISC – Complex Instruction Set Computers
  RISC – Reduced Instruction Set Computers.

(vi) Compilers:

→ A compiler translates a high-level language program into a sequence of machine instructions.
→ To reduce N, we need to have a suitable machine instruction set and a compiler that makes good use of it.
→ An optimizing compiler takes advantage of various features of the target processor to reduce the product NXS, which is the total number of clock cycles needed to execute a program.

(vii) Performance Measurement:

→ Computer designers use performance estimates to evaluate the effectiveness of new features.
→ Manufacturers use performance indicators in the marketing process.
→ Buyers use such data to choose among many available computer models.
→ The computer community adopted the idea of measuring computer performance using benchmark programs.
→ To make comparisons possible, standardized programs must be used.
→ The performance measure is the time it takes a computer to execute a given benchmark.
→ The accepted practice today is to use an agreed-upon selection of real application programs to evaluate performance.
→ A nonprofit organization called SPEC (System Performance Evaluation Corporation) selects and publishes representative application programs for different application domains, together with test results for many commercially available computers.
→ The SPEC rating is computed as,

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}.$$

→ The overall SPEC rating for the computer is given by

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} \text{SPEC}_i\right)^{1/n}$$

where n is the number of programs in the suite.

(7) The History of Computer Development:

→ Computers as we know them today have been developed over the past 60 years
→ A long slow evolution of mechanical calculating devices proceded the development of computers.
→ Development of the technologies used to fabricate the processors, memories, and I/O units of computers has been divided into four generation.

(i) First Generation (1945-55)
(ii) Second Generation (1955-65)
(iii) Third generation (1965-75)
(iv) Fourth Generation (1975 - present)

(i) The First Generation:

→ The key concept of a stored program was introduced by John von Neumann
→ Programs and their data were located in the same memory, as they are today.
→ Assembly language was used to prepare programs and was translated into machine language for execution.
→ Basic arithmetic operations were performed in a few milliseconds using vacuum tube technology to implement logic functions.
→ Magnetic core memories and magnetic tape storage devices were also developed.

(ii) The Second Generation:

→ The transistor was invented at AT&T Bell Laboratories in the late 1940s and quickly replaced the vacuum tube.
→ This basic technology shift marked the start of the second generation.
→ Magnetic core memories and magnetic drum storage devices were more widely used in the second generation.
→ High-level languages such as FORTRAN were developed.
→ Compilers were developed to translate these high-level language programs into a corresponding assembly language program.
→ Seperate I/O processors were developed.
→ IBM became a major computer manufacturer during this time.

(iii) The Third Generation:

→ The ability to fabricate many transistors on a single silicon chip, called Integrated Circuit (IC) technology developed.
→ Integrated circuit memories began to replace magnetic core memories.
→ Other developments included the introduction of micsoprogramming, parallelism, and Pipelining.
→ Operating System software allowed efficient sharing of a Computer system by several user programs.
→ Cache and virtual memories were developed.
→ Cache memory makes the main memory appear faster than it really is, and virtual memory makes it appear larger.
→ System 360 mainframe computers from IBM and the line of PDP mini-computers from Digital Equipment Corporation were dominant commercial products of the third generation.

(iv) The Fourth Generation:

→ Tens of thousands of transistors could be placed on a single chip, and the name Very Large Scale Integration (VLSI) was coined to describe this technology.

→ VLSI technology allowed a complete processor to be fabricated on a single chip called Microprocessor.

→ Companies such as Intel, National Semiconductor, Motorola, Texas Instruments, and Advanced Micro devices, were the driving forces of this technology.

→ Organizational concepts such as concurrency, pipelining, caches, and virtual memories evolved to produce the high-performance computing systems of today.

→ Portable Notebook computers, desktop PCs, and workstations, interconnected by local area networks (LAN), WAN (Wide area networks), and the Internet.

→ Centralized computing on mainframes is now used primarily for business applications in large companies.

Computer Organization

Syllabus : Machine Instruction and Programs

- Instructions and Instruction Sequencing
  - Register Transfer Notation
  - Assembly language Notation
  - Basic Instruction types
- Addressing Modes
- Basic Input Output Operations
- The Role of Stacks and Queues in Computer Programming Equation
- Component of Instructions
  - Logic Instructions
  - Shift and Rotate Instructions.

Text Book :

1. Computer Organization, Carl Hamacher, Zvonks Vranesic, Safea Zaky, 5th Edition, McGraw Hill.

(1) Instructions and Instruction Sequencing:

→ A computer must have instructions capable of performing four types of operations.

- Data transfers between the memory and the processor registers.
- Arithmetic and Logic operations on data
- Program Sequencing and Control.
- I/O Transfers.

(i) Register Transfer Notation:

→ We need to describe the transfer of information from one location in the computer to another.

→ Possible locations that may be involved in such transfers are memory locations, processor registers, (or) registers in the I/O subsystem.

→ Names for the addresses of memory locations may be LOC, PLACE, A, VAR2 processor register names may be R_0, R_5 and I/O register names may be DATAIN, OUTSTATUS, and so on.

→ The contents of a location are denoted by placing square brackets around the name of the location.

Example: $R1 \leftarrow [LOC]$

- means that the contents of memory location LOC are transferred into processor register R1.

Example: $R3 \leftarrow [R1] + [R2]$

- means the operation that adds the contents of registers R1 and R2 and then places their sum into register R3.

→ This type of notation is known as Register Transfer Notation (RTN)

→ The right-hand side of RTN is always denotes a value and the left-hand side is the name of a location where the value is to be placed.

(ii) Assembly Language Notation:

→ Assembly language format is a notation to represent machine instructions and programs.

Example: MOVE LOC, R1

- Here data transferred from memory location LOC to processor register R1
- The contents of LOC are ~~not~~ unchanged by the execution of this instruction, but the old contents of Register R1 are overwritten.

Example: Add R1, R2, R3

- Here, adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly language statement

~~Add~~ ~~R1,~~

(iii) Basic Instruction Types:

→ There are 4 types of Instructions available

a) Three-address Instructions
b) Two-address Instructions
c) One-address Instructions
d) Zero-address Instructions

a) Three-address Instructions:

→ An instruction having three operands

Syntax: Opcode Source1, Source2, Destination

Example: Adding two numbers

ADD A,B,C (∵ C ← [A] + [B])

- where A,B are called Source Operands, C is called destination operand.
- Opcode means Operation Code.

b) Two-address Instructions:

→ An instruction having two operands

Syntax: opcode Source, Destination

Example: MOVE B,C (∵ C ← [B])
ADD A,C (∵ C ← [A] + [C])

c) One-address Instructions:

→ An instruction having only one operand
→ When a second operand is needed, as in the case of an ADD instruction, a processor register, called Accumulator is used.

Example: LOAD A (∵ AC ← [A])
ADD B (∵ AC ← [AC] + [B])
STORE C (∵ C ← [AC])

d) Zero-address Instructions:

→ An instruction having zero operands.
→ It uses stack operations PUSH and POP to perform operations.

Example:

```
PUSH A    (∵ TOS ← [A])
PUSH B    (∵ TOS ← [B])
ADD       (∵ TOS ← [A] + [B])
POP C     (∵ C ← [TOS])
```

Another Example: Evaluate X = (A+B) * (C+D)

Three Address:

```
ADD A, B, R1    (∵ R1 ← [A] + [B])
ADD C, D, R2    (∵ R2 ← [C] + [D])
MUL R1, R2, X   (∵ X ← [R1] * [R2])
```

Two Address:

```
MOV A, R1    (∵ R1 ← [A])
ADD B, R1    (∵ R1 ← [R1] + [B])
MOV C, R2    (∵ R2 ← [C])
ADD D, R2    (∵ R2 ← [R2] + D)
MUL R1, R2   (∵ R2 ← [R1] * [R2])
MOV R2, X    (∵ X ← [R2])
```

One-Address:

```
LOAD A     (∵ AC ← [A])
ADD B      (∵ AC ← [AC] + [B])
STORE T1   (∵ T1 ← [AC])
LOAD C     (∵ AC ← [C])
ADD D      (∵ AC ← [AC] + [D])
MUL T1     (∵ AC ← [AC] * [T1])
STORE X    (∵ X ← [AC])
```

Zero-Address:

```
PUSH A    (∵ TOS ← [A])
PUSH B    (∵ TOS ← [B])
ADD       (∵ TOS ← [A] + [B])
PUSH C    (∵ TOS ← [C])
PUSH D    (∵ TOS ← [D])
ADD       (∵ TOS ← [C] + [D])
MUL       (∵ TOS ← ([A] + [B]) * ([C] + [D]))
POP X     (∵ X ← [TOS])
```

② Addressing Modes:

→ The different ways in which the location of an operand is specified in an instruction are referred to as Addressing Modes.

→ The different Generic Addressing Modes is given as

S.NO	Name	Assembler Syntax	Addressing function
1	Immediate	#Value	Operand = Value
2	Register	R_i	EA = R_i
3	Absolute (Direct)	LOC	EA = LOC
4	Indirect	(R_i) (LOC)	EA = [R_i] EA = [LOC]
5	Index	X(R_i)	EA = [R_i] + X
6	Base with Index	(R_i, R_j)	EA = [R_i] + [R_j]
7	Base with index and Offset	X (R_i, R_j)	EA = [R_i] + [R_j] + X
8	Relative	X(PC)	EA = [PC] + X
9	Auto Increment	(R_i) +	EA = [R_i]; Increment R_i
10	Auto Decrement	−(R_i)	Decrement R_i EA = [R_i]

(i) Implementation of Variables and Constants:

→ In Assembly language, a variable is represented by allocating a register (or) a memory location to hold its value.

→ There are two addressing modes to access variables

a) Register mode

b) Absolute mode.

*) Register Mode:

→ The operand is the contents of a processor Register

Example: MOVE R1, R2

- Here R_1, R_2 are registers.

*) Absolute (Direct) Mode:

→ The operand is in a memory location

Example: ADD A,B

→ Address and Data constants are represented in Assembly language using the Immediate mode.

*) Immediate Mode:

→ The operand is given explicitely in the instruction.

Example: MOVE #200, R0

→ places the value 200 in Register R0

→ # sign indicates that this value is to be used as an immediate operand.

Example:

```
MOVE  B, R1
ADD   #7, R1
MOVE  R1, A
```

(ii) Indirection and pointers:

→ In Addressing modes, the instruction does not give the operand (or) its address explicitly.

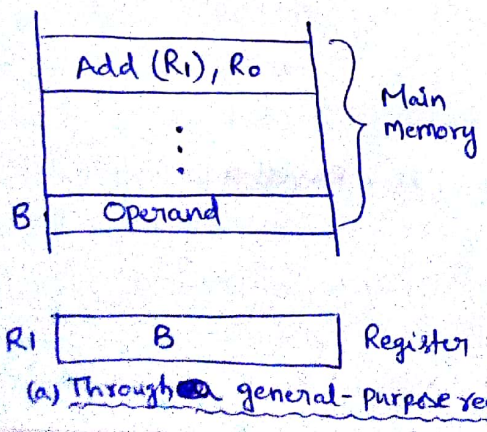
→ Instead, it provides information from which the memory address of the operand can be determined. This address is called as Effective Address(EA)

*) Indirect Mode:

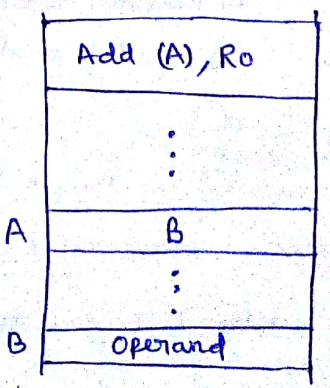
→ The effective address of the operand is the contents of a register (or) memory location whose address appears in the instruction

→ We denote indirection by placing the name of the register (or) the memory address given in the instruction in parantheses.

→ Indirect Addressing is depicted as



(a) Through a general-purpose register



(b) Through a memory location

(a) - To execute the Add instruction, the processor uses the value of B, which is register R1, as the effective address of the operand.
- It requests a read operation from the memory to read the contents of location B.
- The value read is the desired operand, which the processor adds to the contents of register R0.

(b) - In Indirect Addressing through a memory location, the processor first reads the contents of memory location A, then requests a second read operation using the value B as an address to obtain the operand.
- The register (or) memory location that contains the address of an operand is called a pointer.

→ Consider the C-language statement

A = *B;

- where B is a pointer variable
- This statement may be compiled into

MOVE B, R1
MOVE (R1), A

- Using indirect addressing through memory,

MOVE (B), A

(iii) Indexing and Arrays:

→ It is useful in dealing with lists and Arrays.

*) Index Mode:

→ The effective address of the operand is generated by adding a constant value to the contents of a register
→ The register used may be either a special register (or) general-purpose register called Index register.
→ Index mode symbolically represented as,

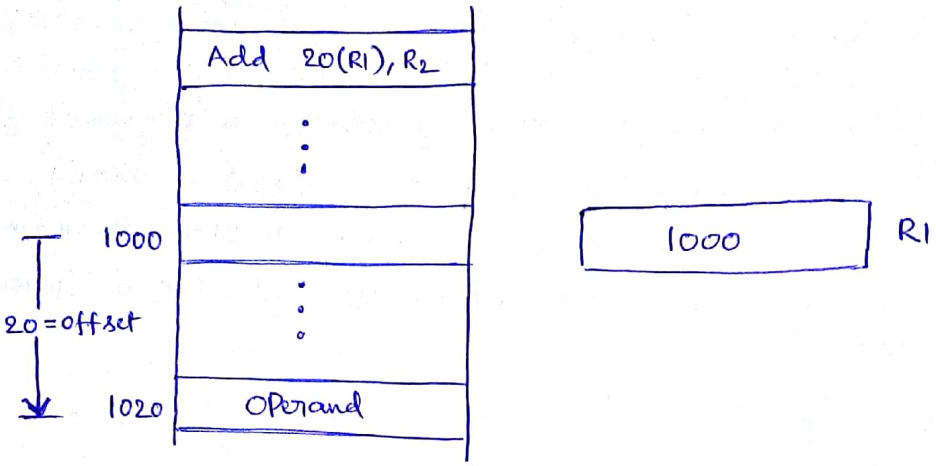
X(Ri)

where X denotes the constant value contained in the instruction
Ri denotes the name of the register involved.
→ The effective address of the operand is given by

EA = X + [Ri]

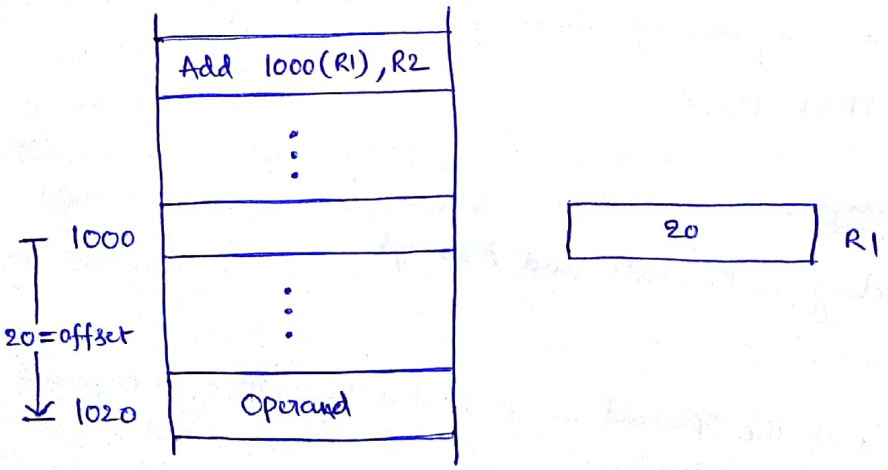
→ The contents of the index register are not changed in the process of generating the effective address.

→ Two ways of using Index mode (Index Addressing) is depicted as,



(a) Offset is given as a Constant

→ The index register R1 contains the address of a memory location, and the value X defines an offset (or) displacement) from this address to the location where the operand is found.



(b) offset is in the index register

→ Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand.

→ In either case, the effective address is the sum of two values
- One is given explicitly in the instruction, and
- the other is stored in a register.

(iv) Relative Addressing:

→ Index mode is used for general-purpose processor registers

→ In Relative Addressing, the program counter (PC) is used instead of a general purpose register.

*) Relative Mode:

→ The effective address is determined by the Index mode using the program counter in place of general purpose register R_i

→ Relative mode Symbolically represented as,

$$X(PC)$$

→ The effective address of the operand is given by

$$EA = X + [PC]$$

→ This mode can be used to access data operands

(v) Additional Modes:

(a) Autoincrement Mode

(b) Autodecrement Mode

*) Autoincrement Mode:

→ The effective address of the operand is the contents of a register specified in the instruction

→ After Accessing the operand, the contents of this register are automatically incremented to point to the next item in the list.

→ Symbolically represented as

$$(R_i)+$$

- It normally increments one, but in byte-sized operands (or) byte-addressable memory

  1 for 8-bit operands

  2 for 16-bit operands

  4 for 32-bit operands.

→ The effective Address of the operand is

$$EA = [R_i];$$
Increment R_i

(*) Autodecrement Mode:

→ The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand

→ Symbolically represented as,

$$-(R_i)$$

→ The effective address of the operand is

Decrement R_i;
$EA = [R_i]$

(3) Basic Input/Output Operations:

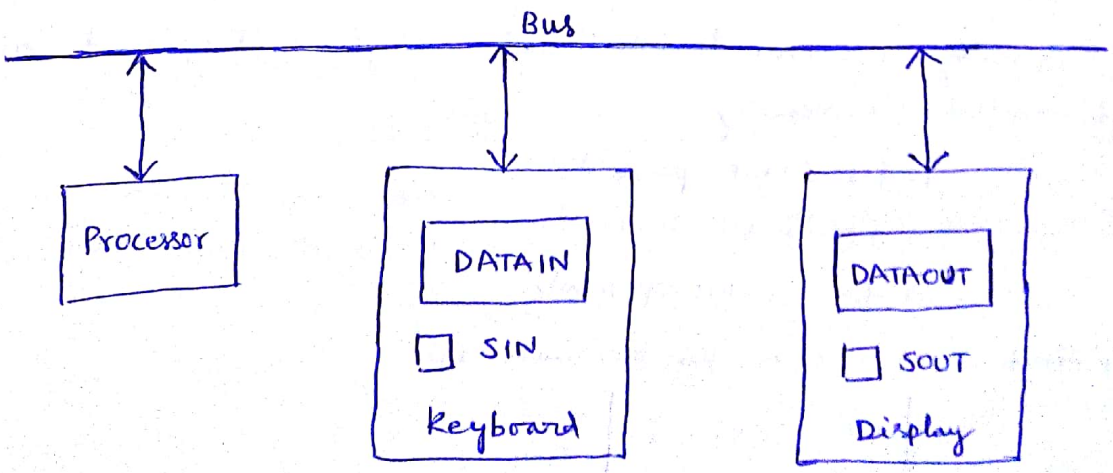
→ I/O operations are essential, and the way they are performed can have a significant effect on the performance of the computer.

→ Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as program-controlled I/O.

→ The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second.

→ The rate of output transfers from the computer is determined by the rate at which characters can be transmitted over the link between the computer and the display device, typically several thousand characters per second.

→ Bus connection for processor, keyboard, and display are depicted as,



→ The Keyboard and the display are separate devices.
→ Consider the problem of moving a character code from the keyboard to the processor.
→ Striking a key stores the corresponding character code in an 8-bit buffer register associated with the keyboard.
→ Let us call this register DATAIN
→ To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1.
→ A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN.
→ When the character is transferred to the processor, SIN is automatically cleared to 0.
→ If the second character is entered at the keyboard, SIN is again set to 1 and the process repeats.
→ When characters are transferred from the processor to the display, a buffer register, DATAOUT, and the status control flag, SOUT, are used for this transfer.
→ The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a device Interface.

Example:

→ The processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to Register R1 by the following sequence of operations

```
READWAIT   Branch to READWAIT If SIN = 0
           Input from DATAIN to R1
```

→ Sequence of operations used for transferring output to the display is given as,

```
WRITEWAIT  Branch to WRITEWAIT If SOUT = 0
           Output from R1 to DATAOUT
```

→ The contents of the keyboard character buffer DATAIN can be transferred to Register R1 in the processor by the instruction

```
MOVEBYTE   DATAIN, R1
```

→ Similarly, the contents of Register R1 can be transferred to DATAOUT by the instruction

```
MOVEBYTE   R1, DATAOUT
```

④ The Role of STACKS and QUEUES in Computer Programming Equations

→ Inorder to organize the control and information linkage between the main program and the subroutine, a datastructure called a stack is used.

→ A stack is a list of data elements, words (or) bytes, with the accessing restriction that elements can be added (or) removed at one end of the list only.

→ It uses a pointer variable called Top of the stack.

Example :- pile of Trays in a Cafeteria

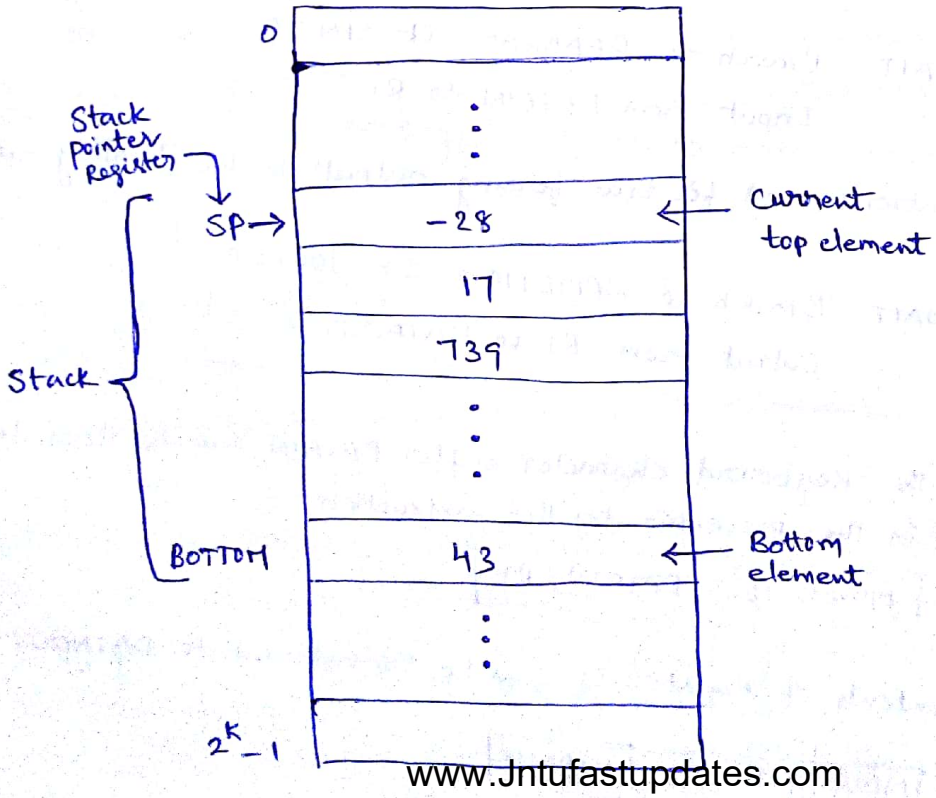
- Customer pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile.

→ Stack uses LIFO (Last In First Out) mechanism, which describes the last data item placed on the stack is the first one removed when retrieval begins.

→ PUSH and POP are the operations of stack, which describes placing the new item on the stack and removing the top item from the stack, respectively.

→ Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations.

→ A stack of words in the memory is depicted as



→ It contains numerical values, with 43 at the bottom and -28 at the top.
→ A processor register is used to keep track of the address of the element of the stack that is at the top at any given time, called Stack pointer (SP)
→ In a byte-addressable memory with a 32-bit word length, the PUSH operation can be implemented as,

```
SUB  #4, SP
MOVE NEWITEM, (SP)
```

→ The POP operation can be implemented as

```
MOVE (SP), ITEM
ADD  #4, SP
```

→ If the processor has the Autoincrement and Autodecrement addressing modes, then the PUSH operation can be performed by the single instruction

```
MOVE NEWITEM, -(SP)
```

→ The pop operation can be performed by the single instruction

```
MOVE (SP)+, NEWITEM
```

→ Another useful datastructure that is similar to the stack is called Queue
→ Data are stored in and retrieved from a queue on a First In First Out (FIFO) basis.
→ Here, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.
→ It uses two pointers FRONT and REAR.
→ INSERT and DELETE are the operations of queue.

⑤ Components of Instructions :

(i) Logic Instructions
(ii) Shift and Rotate Instructions

(i) Logic Instructions :

→ Logical operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits
→ It is also useful to be able to perform logic operations in software

Example : 1's complement

```
NOT DST
```

– Complements all bits contained in the destination operand, changing 0's to 1's and 1's to 0's

Example : 2's Complement

```
NOT  R0
ADD  #1, R0
```

→ Many computers have a single instruction for 2's complement

```
NEGATE R0
```

→ Logical operators AND, OR, NOT represented as bits in a table as,

AND

Operand1	Operand2	AND
0	0	0
0	1	0
1	0	0
1	1	1

OR

Operand1	Operand2	OR
0	0	0
0	1	1
1	0	1
1	1	1

NOT

Operand	NOT
0	1
1	0

(ii) Shift and Rotate Instructions:

(a) Shift Instructions:

→ There are many applications that require the bits of an operand to be shifted right (or) left some specified number of bit positions.

→ There are 2 types of Shift Instructions

(*) Logical Shift Instructions

(*) Arithmetic Shift Instructions

(*) Logical Shift Instructions:

→ There are 2 Logical Shift Instructions

- Logical Shift Left (LShiftL)
- Logical Shift Right (LShiftR)

→ These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.

• Logical Shift Left (LShiftL):

→ The general form of Logical Shift Left Instruction is

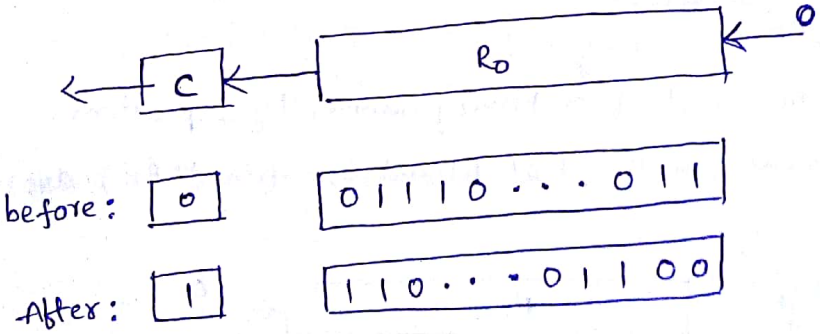
```
LShiftL  count, DST
```

→ The count operand may be an immediate operand (or) it may be contained in a processor register.

Example:

```
LShiftL  #2, R0
```

→ This is depicted as,



Logical Shift Left

→ It shifts the contents of Register R0 left by two positions

→ Vacated positions are filled with 0's.

- Logical Shift Right:

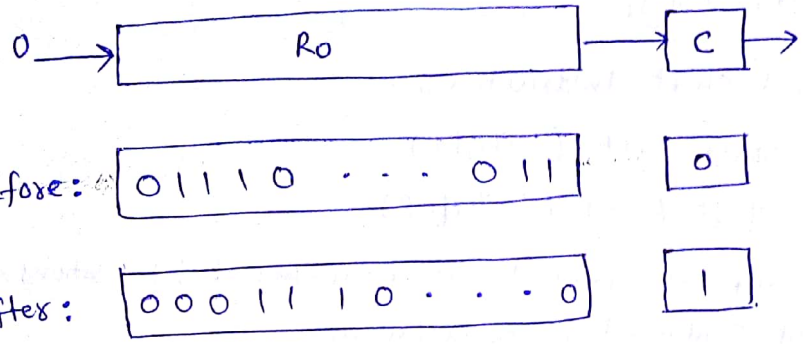
→ The general form is,

LShiftR Count, DST

Example :

LShiftR #2, R0

→ This is depicted as,



Logical Shift Right

→ It shifts the contents of register R0 right by two bit positions.

→ Vacated positions are filled with zeros.

(*) Arithmetic Shift Instructions:

→ There are two types in Arithmetic shift Instructions

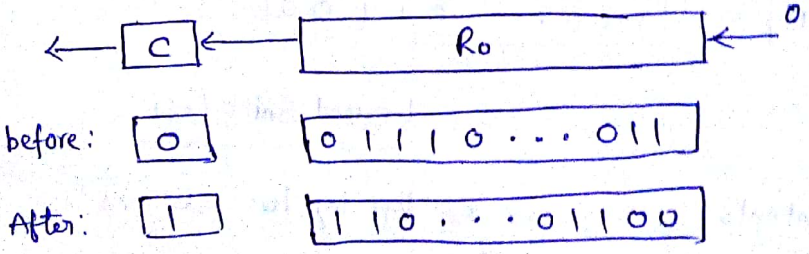
- Arithmetic Shift Left (AShiftL)
- Arithmetic Shift Right (AShiftR)

- Arithmetic ShiftLeft:

AShiftL #2, R0

→ A Left Arithmetic shift of a binary number by 2 positions

→ The empty positions in the LSB (Least Significant Bit) are filled with zeros.



Arithmetic Shift Left

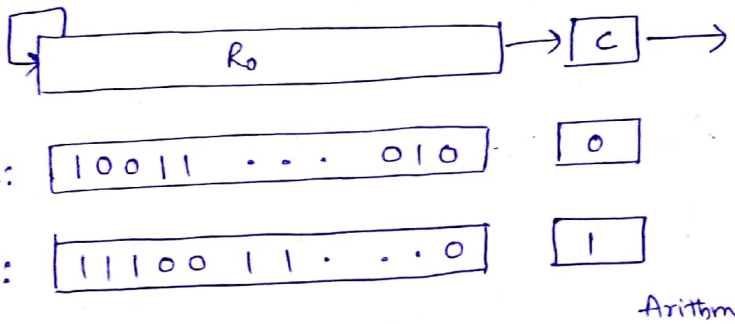
→ It works same like Logical Shift Left operation

• Arithmetic Shift Right:

AShiftR #2, R0

→ A right arithmetic shift of a binary number by 2 positions.

→ The empty positions in the MSB (Most Significant Bit) are filled with copies of the original MSB bit.



Arithmetic shift Right

(b) Rotate Instructions:

→ In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the carry flag C.

→ To preserve all bits, a set of rotate instructions can be used.

→ They move the bits that are shifted out of one end of the operand back into the other end

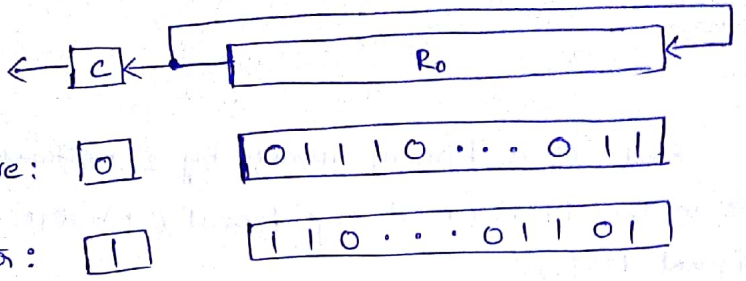
→ The different Rotate Instructions are,

* Rotate Left without Carry (RotateL)
* Rotate Left with Carry (RotateLC)
* Rotate Right without Carry (RotateR)
* Rotate Right with Carry (RotateRC

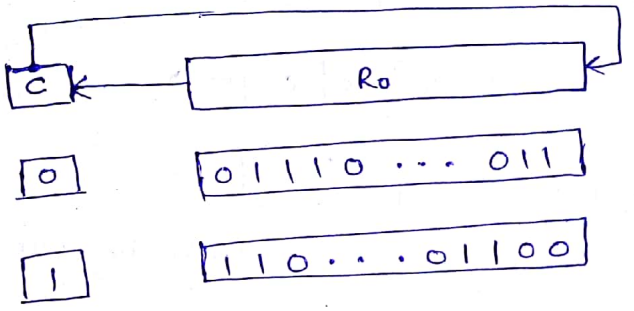
→ The different Rotate Operations are depicted as,

ASR-CO-2.18

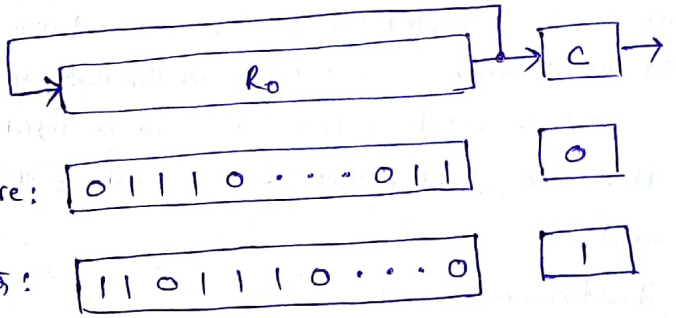
(*) Rotate Left without Carry : RotateL #2, R0



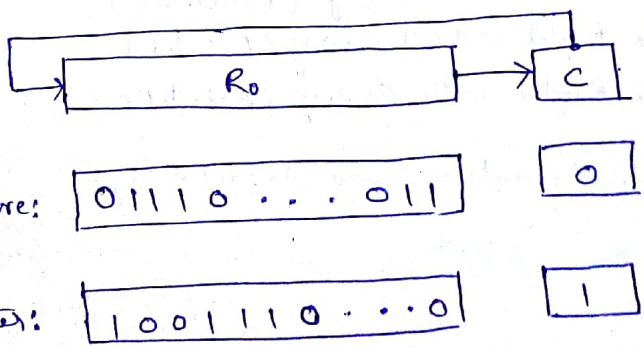
(*) Rotate Left with carry : RotateLC #2, R0



(*) Rotate Right without Carry : RotateR #2, R0



(*) Rotate Right with Carry : RotateRC #2, R0



Types of Instructions

Syllabus:

- Arithmetic and Logic Instructions
- Branch Instructions
- Input/Output Operations
- Addressing Modes

Text Book:

1. Computer Organization, Carl Hamacher, Zvonks Vranesic, Safea Zaky, 5th Edition, McGrawHill.

① Arithmetic and Logic Instructions:

(i) Arithmetic Instructions

(ii) Logic Instructions

(i) Arithmetic Instructions:

* ADD
* SUB
* MUL
* MLA

(*) ADD:

→ The basic Assembly Language expression for arithmetic instruction is

OPCode R_d, R_n, R_m

- where the operation specified by the OPCode is performed using the operands in general-purpose registers R_n and R_m.
- the result is placed in Register R_d

Example: ADD R_0, R_2, R_4 ($\because R_0 \leftarrow [R_2] + [R_4]$)

→ Instead of being contained in Register R_m, the second operand can be given in the immediate mode

Example: ADD $R_0, R_3, \#17$ ($\because R_0 \leftarrow [R_3] + 17$)

→ The second operand can be shifted (or) rotated before being used in the operation.

→ When a shift (or) rotation is required, the instruction is given as

ADD R0, R1, R5, LSL #4

- Here, the second operand, which is contained in Register R5, is shifted left 4 bit positions, and it is then added to the contents of Register R1
- the sum is placed in Register R0

(*) SUB:

SUB R0, R6, R5 (∵ R0 ← [R6] − [R5])

(*) MUL:

→ Two versions of a Multiply instruction are provided.

(I) The first version multiplies the contents of two registers and places the low-order 32-bits of the product in a third register

The high-order bits of the product, if there are any, are discarded.

Example:

MUL R0, R1, R2 (∵ R0 ← [R1] × [R2])

(II) The Second version specifies a fourth register whose contents are added to the product before storing the result in the destination register.

Example: MLA R0, R1, R2, R3 (∵ R0 ← [R1] × [R2] + [R3])

- This is called Multiply-Accumulate Operation
- It is used in numerical algorithms for digital signal processing.

(ii) LOGIC Instructions:

→ The different Logic Instructions are,

* AND
* OR
* XOR (exclusive OR)
* BIC (Bit-Clear)

→ Logic Instructions have the same format as the Arithmetic Instructions

AND R_d, R_n, R_m ($\because R_d \leftarrow [R_n] \wedge [R_m]$)

- which is a bitwise logical AND between the operands in Registers R_n and R_m

Example:

AND R_0, R_0, R_1 ($\because R_0 \leftarrow [R_0] \wedge [R_1]$)

- If register R_0 contains the hexadecimal pattern 02FA62CA and R_1 contains the pattern 0000FFFF, then the result 000062CA being placed in register R_0.

Examples:

(i) OR R_0, R_1, R_2 ($\because R_0 \leftarrow [R_1] \vee [R_2]$)

(ii) XOR R_0, R_1, R_2 ($\because R_0 \leftarrow [R_1] \oplus [R_2]$)

→ The BIC (Bit-Clear) instruction, is closely related to the AND instruction
→ It complements each bit in operand R_m before ANDing them with the bits in register R_n

BIC R_0, R_0, R_1 ($\because R_0 \leftarrow [R_0] \wedge (\text{NOT}[R_1])$)

- If R_0 is 02FA62CA and R_1 is 0000FFFF, then the result is 02FA0000 being placed in R_0.

→ The MOVE NEGATION instruction, with the OP-code mnemonic MVN, Complements the bits of the source operand and places the result in R_d

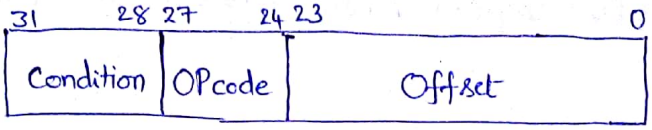
Example: MVN R_0, R_3

→ If the contents of R3 are the hexadecimal pattern 0F0F0F0F, then it places the result F0F0F0F0 in register R0

(2) Branch Instructions.

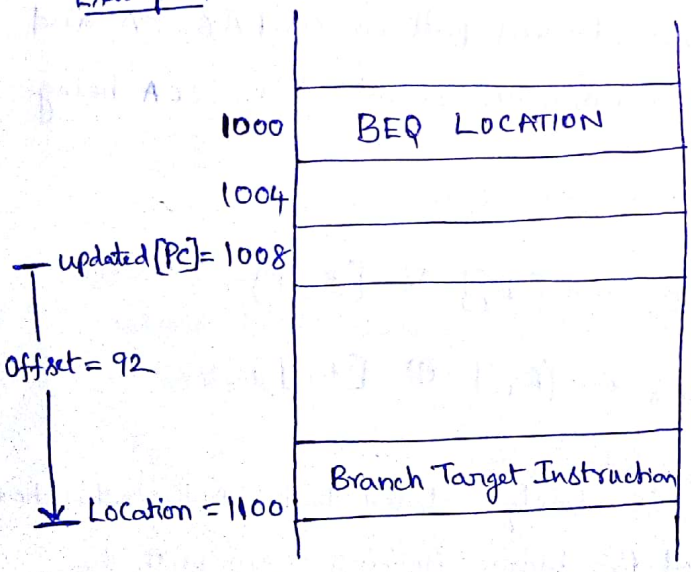
→ Conditional branch instructions contain a signed, 2's complement, 24-bit offset that is added to the updated contents of the program counter (PC) to generate the branch target address

→ The format for the branch instructions is depicted as,



Instruction Format

Example:



Determination of a Branch Target Instruction

→ The BEQ (Branch if Equal to 0) causes a branch if the Z flag is set to 1

→ The condition to be tested to determine whether (or) not branching should take place is specified in the high-order 4-bits, b_{31-28}, of the instruction word.

→ A Branch instruction is executed in the same way as any other ARM instruction, that is, it is executed only if the current state of the condition code flags corresponds to the condition specified in the condition field of the instruction

→ In Example, If the Branch instruction is at address location 1000, and the branch target address is 1100, then the offset has to be 92 because

the contents of the updated PC will be 1000+8 = 1008 when the address 1100 is computed.

(i) Setting Condition Codes:

→ Some instructions, such as COMPARE, given by

CMP Rn, Rm

- which performs the operation,

[Rn] - [Rm]

- have the sole purpose of setting the condition code flags based on the result of the subtract operation.

→ On the other hand, the arithmetic and Logic instructions affect the condition code flags only if explicitely specified to do so by a bit in the OP-code field.

→ This is indicated by appending the suffix S to the assembly language OP-code mnemonic

Example:

→ The instruction

ADDS R0, R1, R2

sets the condition code flags, but

ADD R0, R1, R2

does not.

③ I/O Operations:

→ The ARM architecture uses memory-mapped I/O

→ Reading a character from a keyboard (or) sending a character to a display can be done using program-controlled I/O

→ Suppose that bit 3 in each of the device status registers INSTATUS (Keyboard) and OUTSTATUS (display) contains the respective control flags SIN and SOUT

→ Also assume that the Keyboard DATAIN and display DATAOUT registers are located at addresses INSTATUS +4 and OUTSTATUS+4, immediately following the status register locations.

→ The read and write wait loops can then be implemented as,

Example:

- Assume that address INSTATUS has been located into Register R_1
- The instruction sequence given as,

```
READWAIT   LDR   R3, [R1]
           TST   R3, #8
           BEQ   READWAIT
           LDRB  R3, [R1, #4]
```

- It reads a character into register R3 when a key has been pressed on the keyboard
- The test (TST) instruction performs the bitwise logical AND operation on its two operands and sets the condition code flags based on the result.
- The immediate operand 8 has a single one in the bit 3 position.
- Therefore the result of the TST operation will be zero if bit 3 of INSTATUS is zero and will be nonzero if bit 3 is one, signifying that a character is available in DATAIN.
- The BEQ instruction branches back to READWAIT if the result is zero, looping until a key is pressed, which sets bit 3 of INSTATUS to One.

- Assuming that address OUTSTATUS has been loaded into register R2.
The instruction sequence is given as,

```
WRITEWAIT  LDR  R4, [R2]
           TST  R4, #8
           BEQ  WRITEWAIT
           STRB R3, [R2,#4]
```

- It sends the character in Register R3 to the DATAOUT register when the display is ready to receive it.

④ Addressing Modes:

→ The 68000 processor addressing modes is depicted in a table as,

Name	Assembler Syntax	Addressing Modes
Immediate	#Value	Operand = Value
Absolute Short	Value	EA = Sign Extended WValue
Absolute Long	Value	EA = Value
Register	Rn	EA = Rn (∴Operand = [Rn])
Register Indirect	(An)	EA = [An]
Indexed basic	WValue (An)	EA = WValue + [An]
Indexed Full	BValue (An, RK.S)	EA = BValue + [An] + [RK]
Relative basic	WValue (PC) (or) Label	EA = WValue + [PC]
Relative Full	BValue (PC, RK.S) (or) Label (RK)	EA = BValue + [PC] + [RK]
Auto increment	(An) +	EA = [An] Increment An ;
Autodecrement	−(An)	Decrement An EA = [An] ;

→ The 68000 processor have several addressing modes.

(1) Immediate Mode:

- The operand is contained in the instruction
- Four sizes of operands can be specified, Byte, Word, Long-word, OP-code word.
- The fourth size consists of very small numbers that can be

included directly in the OP-code word of some instructions.

(ii) Absolute Mode: (Direct Mode)

- The absolute address of an operand is given in the instruction, following the OP code.
- There are two versions of this mode - long and short.
- In the long mode, a 24-bit address is specified explicitly.
- In the short mode, a 16-bit value is given in the instruction to be used as the low-order 16-bits of an address.
- The sign bit of this value is extended to provide the high-order eight bits of the address.
- Since the sign bit is either 0 (or) 1, it follows that in the short mode only two pages of the addressable space can be accessed.
- These are the 0 page and the FF8 page, each containing 32K bytes.

(iii) Register Mode:

- The operand is in a processor register specified in the instruction.

(iv) Register Indirect mode:

- The effective address of the operand is in an address register specified in the instruction.

(v) Basic Index Mode:

- A 16-bit signed offset and an address register, An, are specified in the instruction
- The sum of this offset and the contents of An is the effective address of the operand.

(vi) Full Index Mode:

- An 8-bit signed offset, an address register An, and an index register R_K (either an address (or) a data register) are given in the instruction.
- The effective address of the operand is the sum of the offset and the contents of registers An and R_K
- Either all 32-bits (or) the sign-extended low-order 16 bits of R_K are used in the derivation of the address.

(vii) Basic Relative Mode:

→ This is same as the basic index mode except that the program Counter is used instead of an address register, An.

(viii) Full Relative Mode:

→ This is same as the full index mode except that the program Counter is used instead of an address register, An.

(ix) Autoincrement Mode:

- The effective address of the operand is in an address register, An, specified in the instruction
- After the operand is accessed, the contents of An are incremented by 1, 2, (or) 4, depending on whether a byte, a word, (or) a long-word operand, respectively, is involved.

(x) Autodecrement Mode:

- The contents of an address register, An, Specified in the instruction are decremented by 1, 2, (or) 4, depending on whether a byte, a word, (or) a long-word operand, respectively, is involved.
- The effective address of the operand is the decremented contents of An.