

1.2 FUNCTIONAL UNITS

- A computer consists of **five** functionally independent main parts: **input, memory, arithmetic and logic, output, and control units**, as shown in **Figure 1.1**.
- The **input unit accepts coded information** from human operators, from electromechanical devices such as keyboards, or from other computers over digital communication lines.
- The information received is either **stored in the computer's memory** for later reference or immediately **used by the arithmetic and logic circuitry** to perform the desired operations.
- The **processing steps** are determined by a **program stored in the memory**. Finally, the **results are sent back** to the outside world through the output unit.
- All these actions are **coordinated by the control unit**.
- We refer to the arithmetic and logic circuits, in conjunction with the main control circuits, as the **processor**, and input and output equipment is often collectively referred to as the **input-output (I/O) unit**.

Information is either **instructions or data**. 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.

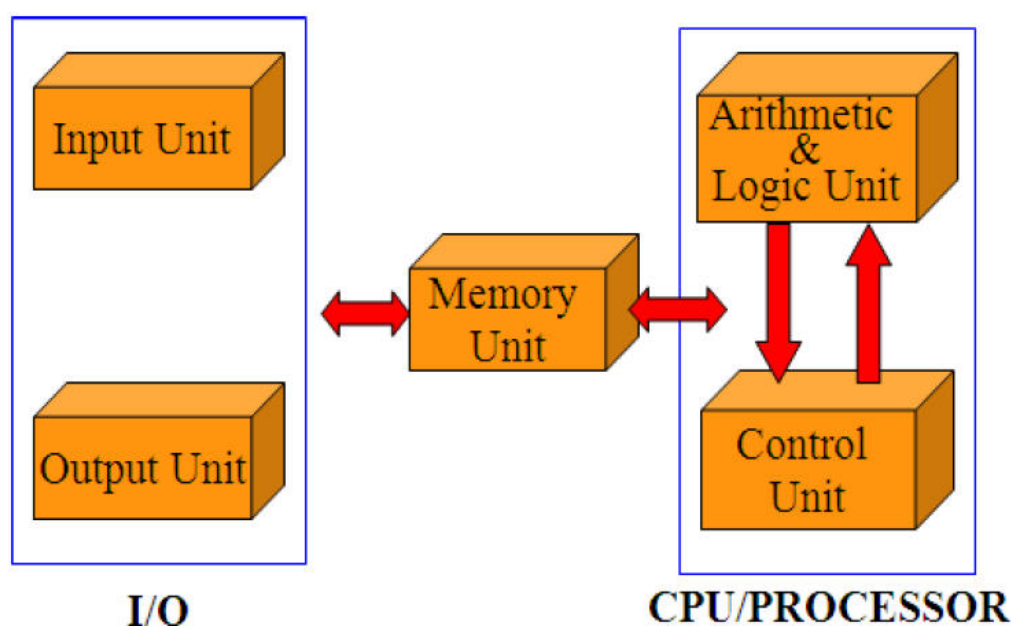


Figure:Basic Functional Units of a Computer

A list of **instructions** that performs a task is called a **program**. **Data** are numbers and encoded characters that are used as **operands**, by the instructions.

The task of **compiling** a high-level language **source program** into a list of machine instructions constituting a machine language program, called the **object program**.

1.2.1 INPUT UNIT

Computers accept coded information through input units, which read the data. The most well-known input device is the **keyboard**. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding **binary code** and transmitted over a cable to either the **memory or the processor**.

Many other kinds of input devices are available, including joysticks, trackballs, and mouses, Microphones.

1.2.2 MEMORY UNIT

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

Primary Memory

- Primary storage is a **fast memory** that operates at electronic speeds. Programs must be stored in the memory while they are being executed.
- The memory contains many semiconductor **storage cells**, each capable of storing one **bit** of information. These cells are rarely **read or written** as individual cells but instead are processed in groups of fixed size called **words**.
- The memory is organized so that the contents of one word, containing n bits, can be stored or retrieved in one **basic operation**.
- To provide easy access to any word in the memory, a **distinct address** is associated with each word location. A given word is accessed by specifying its address and issuing a **control command** that starts the storage or retrieval process.
- The number of bits in each word is often referred to as the **word length** of the computer. Typical word lengths range from **16 to 64 bits**.
- The **capacity** of the memory is one factor that characterizes the **size** of a computer. Small machines typically have only a few tens of millions of words, whereas medium and large machines normally have many tens or hundreds of millions of words.
- Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of the processor.

- Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called **random-access memory (RAM)**.
- The time required to access one word is called the **memory access time**. This time is fixed, independent of the location of the word being accessed. It typically ranges from a **few nano seconds (ns) to about 100 ns** for modern RAM units.
- The small, fast, RAM units are called **caches**. They are tightly coupled with the processor and are often contained on the same integrated circuit chip to achieve high performance.
- The largest and slowest unit is referred to as the **main memory**.

Secondary Memory

Although primary storage is essential, it tends to be **expensive**. Thus additional, cheaper, **secondary storage** is used when **large amounts of data** and many programs have to be stored, particularly for, information that is accessed infrequently. A wide selection of secondary storage devices is available, including magnetic disks and tapes and optical disks (CD-ROMs).

1.2.3 Arithmetic and Logic Unit (ALU)

- Most **computer operations** are executed in the arithmetic and logic unit (ALU) of the processor.
- Consider a typical **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.
- When operands are brought into the processor, they are stored in high-speed storage elements called **registers**. Each register can store one word of data.
- **Access times** to registers are somewhat faster than access times to the fastest cache unit in the memory hierarchy.

1.2.4 OUTPUT UNIT

- Its function is to send processed **results** to the outside world. The most familiar example of such a device is a printer.
- Some units, such as **graphic displays**, provide both an output function and an input function. The dual role of such units is the reason for using the single name I/O unit in many cases.

1.2.5 CONTROL UNIT

- The memory, arithmetic and logic, and input and output units store and process information and perform input and output operations. The operation of these units must be **coordinated** in some way. This is the task of the **control unit**.
- The control unit sends **control signals** to other units and senses their states.
- **I/O transfers**, consisting of input and output operations, are **controlled by the instructions** of I/O programs that identify the devices involved and the information to be transferred.
- **Timing signals** are signals that determine when a given action is to take place. **Data transfers** between the processor and the memory are also controlled by the control unit through timing signals.
- It is reasonable to think of a control unit as a well-defined, **physically separate unit** that interacts with other parts of the machine.

The operation of a computer can be summarized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched, under program control, into an arithmetic and logic unit, where it is processed.
- Processed information sends to the computer through an output unit.
- All activities inside the machine are directed by the control unit.

MODULE 1**BASIC OPERATIONAL CONCEPTS**

prof. Neelakantappa.T.T
Asst-prof. CS+E Dept
S.J.M.I.T

- The processor contains ALU, control circuitry and many registers.
- The instruction register (IR) holds the instruction that is currently being executed.
- The instruction is then passed to the control-unit, which generates the Timing signals that determine when a given action is to take place.
- The PC (Program Counter) contains the memory-address of the next instruction to be fetched & executed.
- During the execution of an instruction, the contents of PC are updated to point to next instruction.
- The processor also contains n general-purpose registers R0 through Rn-1.
- The MAR (Memory Address Register) holds the address of the memory-location to be accessed.
- The MDR (Memory Data Register) contains the data to be written into or read out of the addressed location.

Following are the steps that take place to execute an instruction

- The address of first instruction (to be executed) gets loaded into PC.
- The contents of PC (i.e. address) are transferred to the MAR & control-unit issues Read signal to memory.
- After certain amount of elapsed time, the first instruction is read out of memory and placed into MDR.
- Next, the contents of MDR are transferred to IR. At this point, the instruction can be decoded & executed.
- To fetch an operand, its address is placed into MAR & control-unit issues Read signal. As a result, the operand is transferred from memory into MDR, and then it is transferred from MDR to ALU.
- Likewise required number of operands is fetched into processor.
- Finally, ALU performs the desired operation.
- If the result of this operation is to be stored in the memory, then the result is sent to the MDR.
- The address of the location where the result is to be stored is sent to the MAR and a Write cycle is initiated.
- At some point during execution, contents of PC are incremented to point to next

instruction in the program. [The instruction is a combination of opcode and operand].

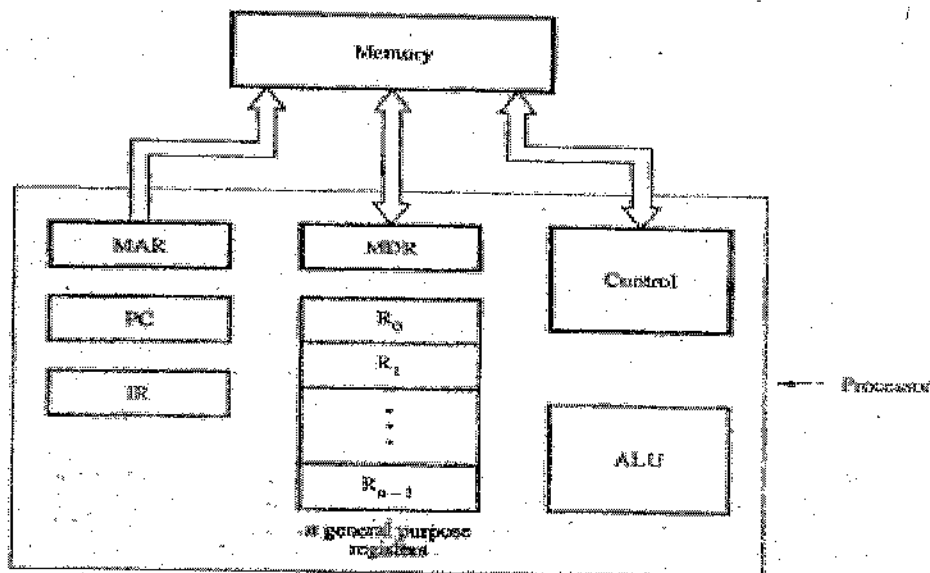


Figure 1.2 Connections between the processor and the memory.

EXAMPLE:1 Add LOCA, R₀

This instruction adds the contents of memory location LOCA with the contents of Register R₀ and the result is stored in R₀ Register.

The symbolic representation of this instruction is

$$[LOCA] + [R_0] \rightarrow R_0$$

The contents of memory location LOCA and Register R₀ before and after the execution of this instruction is as follows

Before instruction execution

LOCA = 23H

R₀ = 22H

After instruction execution

LOCA = 23H

R₀ = 45H

The steps for instruction execution are as follows

1. Fetch the instruction from memory into the IR.
2. Decode the instruction
3. Access the Memory Operand
4. Access the Register Operand
5. Perform the operation according to the Operation Code.
6. Store the result into the Destination Memory location or Destination Register.

BUS STRUCTURE

- A bus is a group of lines that serves as a connecting path for several devices.
- Bus must have lines for data transfer, address & control purposes.
- Because the bus can be used for only one transfer at a time, only 2 units can actively use

the bus at any given time.

- Bus control lines are used to arbitrate multiple requests for use of the bus.
- **Main advantage of single bus:** Low cost and flexibility for attaching peripheral devices.

Disadvantage: Slower.

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

Advantage: better performance.

Disadvantage: increased cost.

- The devices connected to a bus vary widely in their speed of operation. To synchronize their operational speed, the approach is to include buffer registers with the devices to hold the information during transfers.

Buffer registers prevent a high-speed processor from being locked to a slow I/O device during a sequence of data transfers.

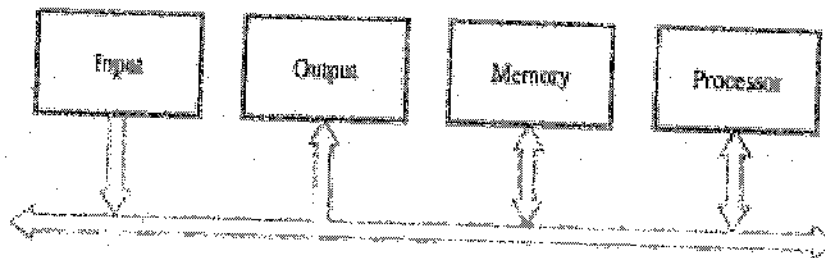


Figure 1.3 Single-bus structure.

There are 3 of buses, namely

1. Addressbus
2. Data busand
3. Control busl.

1. Address bus :

- It is unidirectional.
- The CPU sends the address of an I/O device or Memory device by means of this bus.

2. Data bus

- It is a bidirectionalbus.
- The CPU sends data from Memory to CPU and vice versa as well as from I/O to CPU and vice versa by means of this bus.

3. Controlbus:

- This bus carries control signals for Memory and I/O devices. It generates control signals for Memory namely MEMRD and MEMWR and control signals for I/O devices

namely IORD and IOWR.

- It also generates special control signal to differentiate between Memory and I/O device. This control signal is called as M/I/O.
- $M/I/O = 1$ for Memory operations and $M/I/O = 0$ for I/O operations.

PROCESSOR CLOCK

- Processor circuits are controlled by a timing signal called a clock.
- The clock defines regular time intervals called clockcycles.
- 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 clockcycle.
- Let P =length of one clock cycle R =clock rate. Relation between P and R is given by $R=1/P$ which is measured in cycles per second.
- Cycles per second is also called hertz (Hz).

To execute one instruction there are 3 basic steps namely

- ✓ 1. Fetch
- ✓ 2. Decode
- ✓ 3. Execute.

The processor uses one clock cycle to perform one operation as shown in

the figure ClockCycle → T1 T2 T3
Instruction → Fetch Decode Execute

BASIC PERFORMANCE EQUATION

- Let T =processor time required to execute a program
 N =actual number of instruction executions
 S =average number of basic steps needed to execute one machine instruction
 R =clock rate in cycles per second.
- The program execution time is given by

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

- Equation is referred to as the basic performance equation.
 - To achieve high performance, the computer designer must reduce the value of T , which means reducing N and S , and increasing R .
- The value of N is reduced if source program is compiled into fewer machine instructions.
 → The value of S is reduced if instructions have a smaller number of basic steps to perform.

→ The value of R can be increased by using a higher frequency clock.

- Care has to be taken while modifying the values since changes in one parameter may affect the other.

PIPELINING & SUPERSCALAR OPERATION

- Normally, the computer executes the instruction in sequence one by one. An improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called *pipelining*.

- Consider the instruction

Instruction 1: Add R1, R2, R3

Instruction 2: Move R4, R5

Let us assume that both operations take 3 clock cycles each for completion.

	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2				Fetch	Decode	Execute

As shown in above figure, 6 clock cycles are required to complete two operations.

	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2		Fetch	Decode	Execute		

- As shown in above figure, if we use pipelining & prefetching, only 4 cycles are required to complete same two operations.
- While executing the Add instruction, the processor can read the Move instruction from memory.
- In the ideal case, if all instructions are overlapped to the maximum degree possible, execution proceeds at the rate of one instruction completed in each clock cycle.
- A higher degree of concurrency can be achieved if multiple instruction pipelines are implemented in the processor i.e. multiple functional units can be used to execute different instructions parallelly. This mode of operation is known as superscalar execution.
- With Superscalar arrangement, it is possible to complete the execution of more than one instruction in every clock cycle.

PERFORMANCE MEASUREMENT

- * Basic performance equation depends on many parameters & hence is not reliable indicator of expected performance in many cases.
- * To differentiate & compare different processors, we can use standardized programs called "benchmark programs".
- * An organization called System performance Evaluation Corporation (SPEC) selects and publishes prog. for different application domains.
- * The SPEC was developed in the year 1995 & later it was modified in the year 2000 called as SPEC95 & SPEC 2000.
- * programs are selected from various fields like games, system software, database, numerical calculations etc. The prog. is compiled for the computer under test, & the running time on real comp is measured. The same prog. is compiled and run on reference comp.
- * For SPEC 95 reference comp is SUN SPARC station 10/40
for SPEC 2000 reference comp is Ultra SPARC10 workstation

$$\text{SPEC Rating} = \frac{\text{Running time on reference computer}}{\text{Running time on computer under test}}$$

Ex:- SPEC Rating = 20 indicates comp. under test 20 times faster than reference comp for that particular benchmark.

- * The test is repeated for all progs in the SPEC suite, & geometric mean of the result is computed.

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

where n is the no. of progs in the suite.

MACHINE INSTRUCTIONS AND PROGRAMS

Memory Addressing [Memory Locations and Addresses]

- 1) Information may be either an instruction or a number or an encoded character. And this information is stored in main memory. Memory is organized such that a group of n -bits can be stored or retrieved in a single operation.
- 2) So ' n ' is called as word length and group of n -bits is referred to as a word of information. The word length can vary from 8 to 64 bits.
- 3) Accessing the memory for reading and writing an item requires distinct addresses for each location. If K is the number of address lines, then 2^K addresses constituted the address space of the computer. If $K=20$, we have an address space $2^{20} = 1M$ (1 mega) and if $K=30$, the address space is $2^{30} = 1G$ (1 giga) locations.
- 4) If the word length is 32 bits, then a memory word can store a 32 bit signed integer or 4 ASCII characters as shown in fig 2 below and also a schematic representation of main memory is as shown in fig 1 below:

Memory Structure

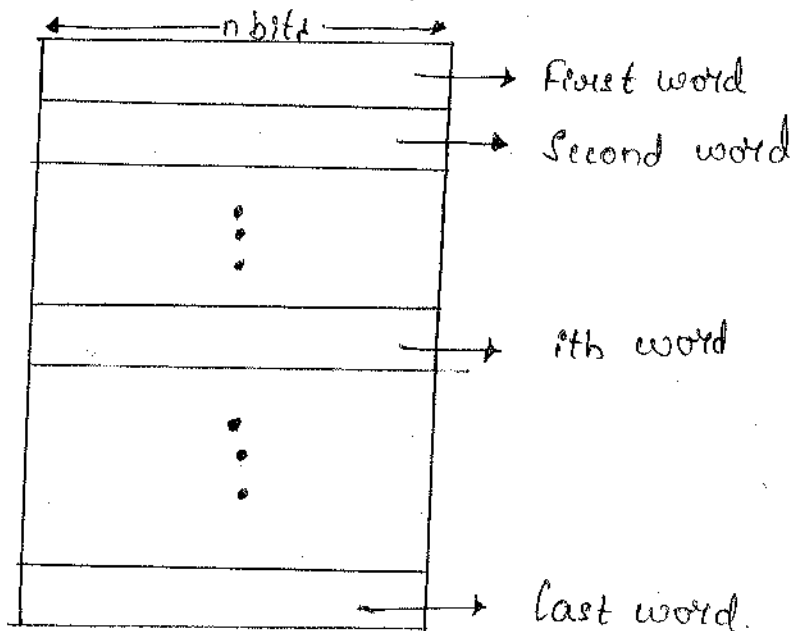


Fig 1 shows the memory structure. In this, the size of the word is 32 bits, and for each word location, we have addresses as shown. To execute our program, we have to move our program from secondary storage to this main memory. Along with this we need to store the required data also.

#) Encoded information in a 32-bit word.

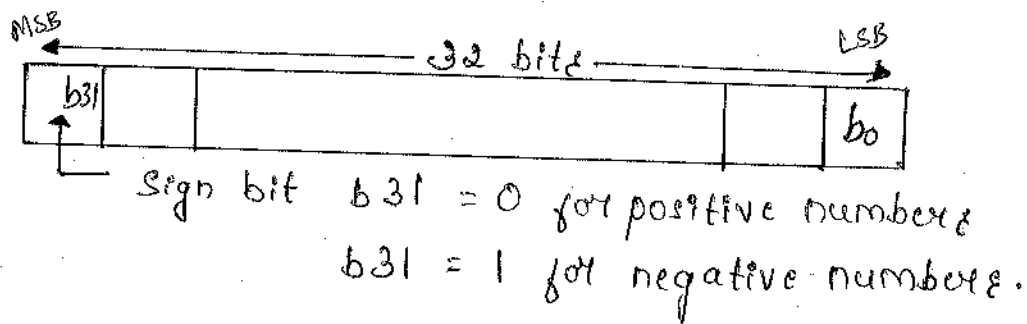


Fig 2(a) : A signed integer.

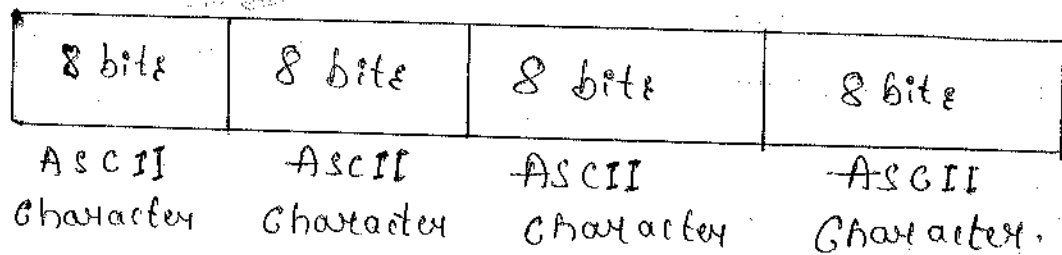


Fig 2(b) : Examples Of encoded information in 32-bit word.

Fig 2(a) shows the representation or storage of a signed number. Here we can store a 32 bit signed integer. The last bit (MSB bit) is used as sign bit. If this bit is 1, then the number is negative. If this bit is 0, then the number is positive.

Fig 2(b) shows the storage or representation of four ASCII character.

Byte Addressability :

(9)

The General arrangement of memory is to have "successive addresses refer to successive byte locations in the memory." That is, each location contains one byte of information. This is called byte addressability.

If the word length is 32 bits, then they are divided into 4 bytes and store in 4 consecutive address locations. Successive words are located at addresses 0, 4, 8, ... with each word having 4 bytes.

Hence, we have two types of byte arrangements as follows :

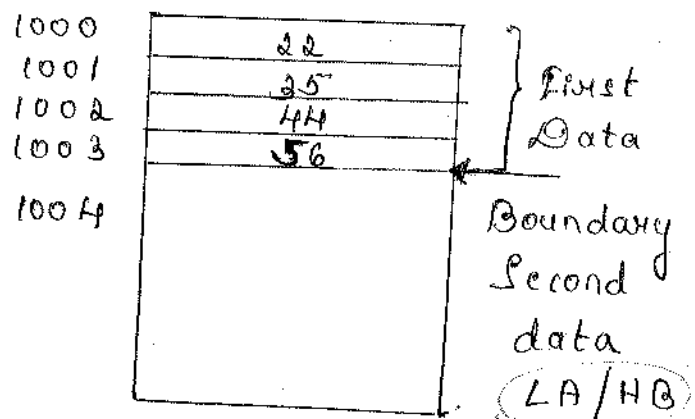
- 1) Big-Endian assignment.
- 2) Little-Endian assignment.

(*1) Big-Endian Assignment

Storage of Given data in big endian assignment :

Given data = 22354456

Word Address	Byte Address			
0	0	1	2	3
4	4	5	6	7
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$



In Big-Endian assignment, a word address is same as that of the most significant (leftmost) byte of the word.

This system is used in Motorola and Power PC Processor.

Hence, higher byte of data is stored at lower address and lower byte of data is stored at higher address.

(*) Little - Endian - Assignment.

Word Address	Byte Address			
0	3	2	1	0
4	7	6	5	4
$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$	$2^k - 4$

Storage of Given data in

little-endian assignment:

Given Data = 22354456

1000	56	} First Data
1001	44	
1002	35	
1003	22	
		} Boundary Second Data.

LA/LB

In Little - Endian assignment, a word address is same as that of the least significant (rightmost) byte of the word. This system is used in Intel processor.

Here, lowest addresses contain low-order bytes of data.

Accessing Numbers, Characters, and Character Strings

- In case of a 32 bit, boundaries occur at addresses 0, 4, 8 --- as shown in fig above. So word locations have aligned addresses. To access a number in memory, its word address is specified. Similarly, characters can be accessed by specifying their byte address.
- To handle a character string of variable length, the beginning of the string is indicated by giving the address of the first character. The length of the string in bytes can be saved in a memory location or processor register.
- Another way to know the length of the string is to be use a special control character as the last character in the string with the meaning "end of string".

Memory Operations :

To execute our program, program instructions and the required operands must be stored in the memory. To execute an instruction, the processor has to obtain data from the memory using control signals. The result of the operation may also be copied to the memory. We have 2 types of memory operations as explained below.

1) Load (Read or fetch) :

This operation transfers a copy of the contents of the specified memory locations to the processor register. Contents of memory are not changed, but the previous content of the register is lost. The steps are :

- ⇒ Processor sends the address of the desired memory location.
- ⇒ Processor issues a read control signal to memory to request the data.
- ⇒ Memory sends the data to the processor.
- ⇒ Data gets stored in processor register.

2) Store (write) :

This operation transfers the information from the processor register to the memory location specified in the instruction. This will destroy the original contents of that memory location. The steps are :

- ⇒ Processor sends the address of the memory location where it wants to store data.
- ⇒ Processor issues a write control signal to memory to store the data.
- ⇒ The content of the processor register is written into the specified memory location. (data remains in the register also).

Instructions And Instruction Sequencing :

-) Register Transfer Notation.
-) Assembly Language Notation
-) Types of Instruction.
-) Straight line sequencing and instruction execution
-) Branching and condition codes.

A computer performs its task according to the program stored in memory. A program is a finite collection of the instructions, which tells the processor to perform basic operations like addition, reading from keyboard etc. A computer should have instructions to perform at least four type of operations listed below:

-) Data transfer b/w memory and processor registers.
-) Arithmetic and logical operation on data.
-) Program sequencing and control.
-) I/O Transfer.

A complete instruction set is offered to as the Instruction Set Architecture (ISA) of the processor.

In addition to specifying instruction, an ISA also specifies the addressing methods used for accessing data operands and processor registers.

Register Transfer Notation :

This is also called as Register Transfer Language (RTL). In this, we can identify the memory location by a symbolic name in uppercase alphabets. The name indicates its address.

For Example, LOCN, NUM, TABLE etc.

indicates memory locations (instead of the actual addresses like 1000, 2000 etc). Similarly, processor registers are indicated by an uppercase alphabet followed by a numeral such as R1, R4 etc. I/O registers are expressed as DATIN, DATAOUT buffer etc.

The content of a location is denoted by the placing square brackets around the name. The right-hand side of an RTN (Register Transfer Notation) expression is always denotes a value and the left-hand side is the symbolic name where the value is to be stored. All

RTN statements will have an 'arrow' symbol, whose left side is the destination and the right side is the source. Source contents are not modified. Destination contents are changed.

Example: Consider a RTN Statement (i) $R1 \leftarrow [LOCN]$

This operation transfers the contents of memory location LOCN into processor register R1. The contents of LOCN is not changed, but the previous content of R1 is lost, by overwriting with a new value.

Consider another example (ii) $R3 \leftarrow [R1] + [R2]$

This will be add the contents of two registers R1 and R2 and place the sum into register R3.

Assembly Language Notation :

- 1) The RPN notation is easy to understand, but cannot be represent machine instructions. Since the processor understands only binary format, we need to use a notation which could be converted into machine language format effectively.
- 2) For this, a new format called Assembly Language is used. Here, we use English like words (called mnemonics) to represent an operation uniquely. For example, statement (i) above can be written in assembly as: Move LOCN, R1. Similarly, the addition operation in (ii) can be expressed as Add R1, R2, R3.

Types Of Instruction.

we have 4 types of instruction as follows :
To study these instructions consider the operation
 $C = A + B$ operation.

- 1) Three address instruction
- 2) Two address instruction
- 3) One Or Single address instruction
- 4) Zero address instruction.

To study the above instruction types: we have 2 conventions

- ✓ ① While writing the instruction, source operands are written first, and then destination operands. (Motorola 68000)
- ② While writing the instruction, destination operands are written first, and then source operands. (Intel and Power PC). In this note you have to follow first convention.

(a) Three Address Instruction :

The general format or syntax of this three-address instruction is as follows :

Operation Source 1, Source 2, Destination.

To perform $C = A + B$ operation we can write a 3 address instruction as follows : Add A, B, C.

Here A and B are source operands and C is destination operand and Add is the operation to be performed. This type has the disadvantages that, instruction code will be too large to fit in one word location in memory.

(b) Two address Instruction :

The general format or syntax of this 2-address instruction is not sufficient for performing this operation. We need two.]^x

Operation Source, Destination

To perform $C = A + B$ operation we can write a 2 address instructions as follows : Here single instruction is not sufficient for performing this operation. We need two instructions as follows :

Move B, C	or	Move A, C
Add A, C		Add B, C

(c) One Single Address Instruction

The general format or syntax of this one address instruction is as follows.

Operation Source / Destination.

Here, an instruction specifies only memory operand which may be source or destination. The other operand is implied to be a processor register called accumulator.

All operations are to be performed on the contents of accumulator.

To perform $C = A + B$ operation we can write a one address instructions as follows. Here ~~single~~

single instruction is not sufficient for performing this operation.

We need three instructions as follows

Load A	or	Load B
Add B		Add A
Store C		Store C

(d) Zero Address Instruction

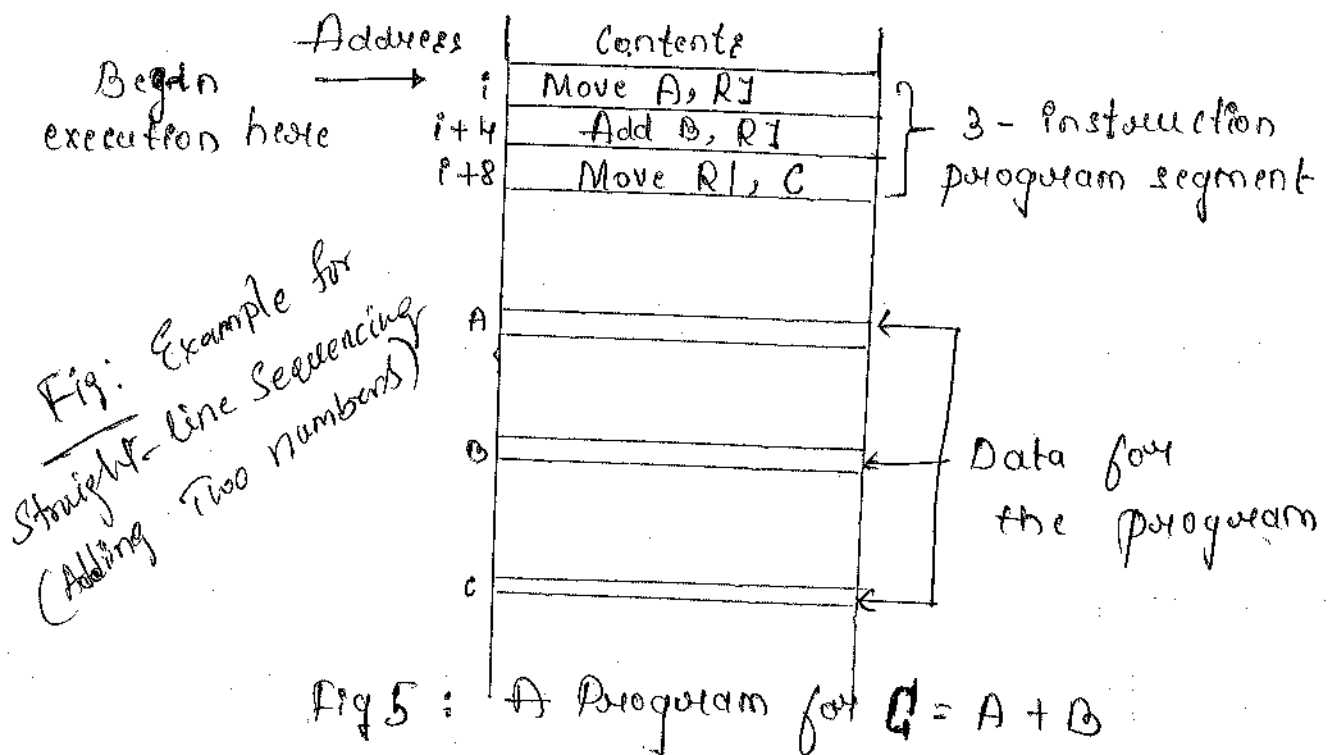
The General format or syntax of this zero address instruction is as follows

Operation	no operand (s)
-----------	----------------

-) In this case, the locations of all operands are defined implicitly. The operands are stored in a structure called pushdown stack. Instruction example : push and pop instructions on pushdown stack.
-) In the above examples, it can be seen that, when operands are in memory, the data has to be moved b/w processor and memory frequently. So this will increase the execution time.
-) Therefore, the data transfers that involve memory are much slower than the transfer within the processor. Hence substantial speed improvement can be achieved when all operations are performed on data present in processor registers.
-) But, this is always not possible since number of processor registers is less. Therefore, data is generally stored in memory.

★ Straight - Line Sequencing And Instruction Execution.

Consider the same operation $C = A + B$: To perform this operation, three instructions are written and stored in the main memory as shown in figure below :



This figure shows the main memory structure or layout. In this layout or structure we can see a 3-instruction program for performing $C = A + B$ operation. Execution begins at i^{th} memory location.

Using the first instruction, contents of memory location A is fetched and moved to processor register R1.

Using the second instruction, contents of memory location B is fetched and it is added to processor register R1, sum is stored in R1.

Using the 3rd instruction, contents of processor register R1 is moved to memory location C.

Here in the program execution, instructions are executed in sequence, so this type of instruction execution is called as straight line sequencing & execution.

After the execution of all 3 instructions, sum is stored into memory location C. This value can be fetched later for displaying or for printing.

Branching And Condition Codes

A straight-line program for adding 'n' numbers.

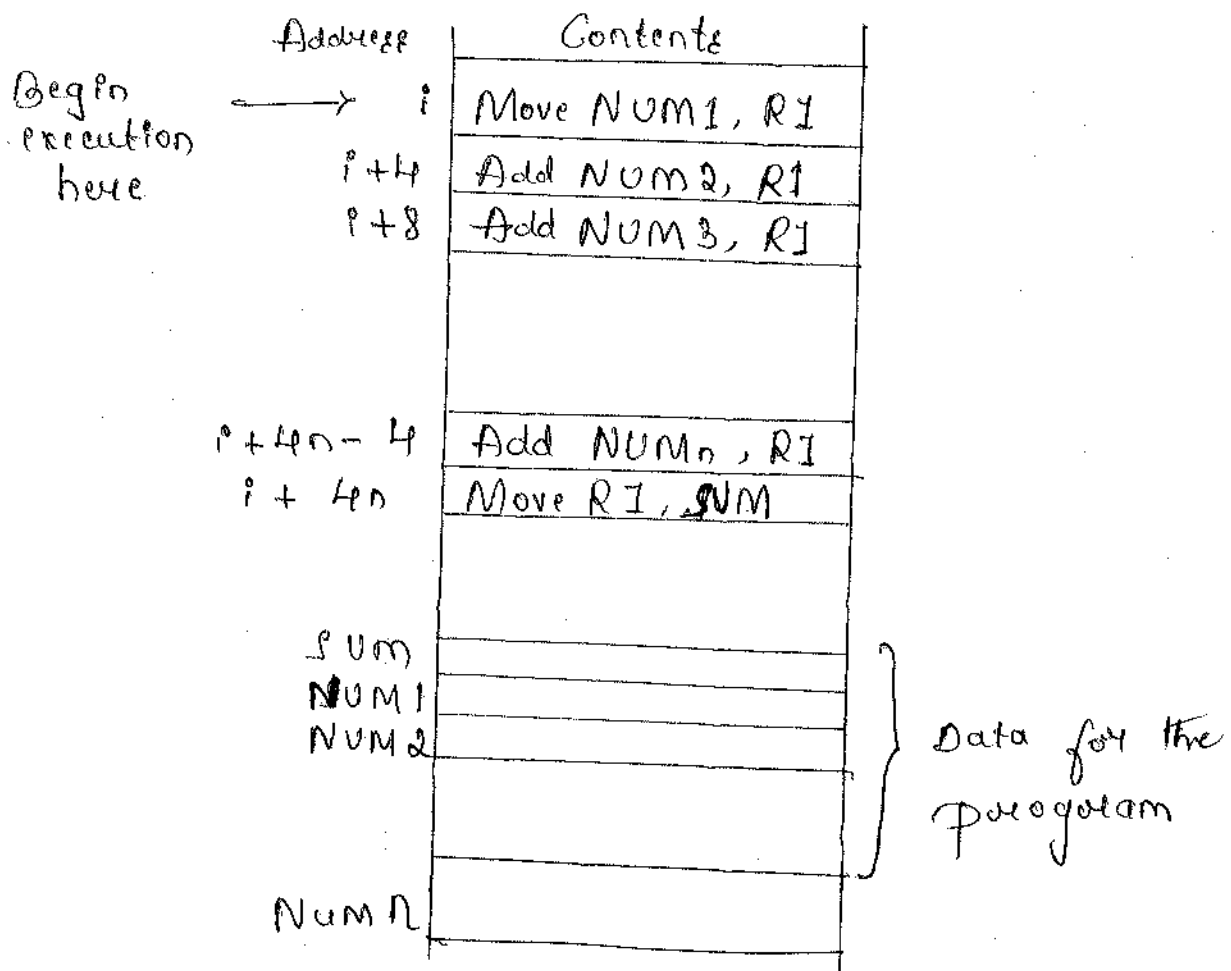


Fig 6: A straight-line program for adding 'n' numbers.

Fig 6 shows the main memory structure or layout. In this layout or structure we can see program for adding 'n' numbers using straight-line sequencing technique. Execution begins at this memory location.

Using the 1st instruction, contents of memory location NUM1 is fetched and moved to processor register R1.

Using the 2nd instruction, contents of memory location NUM2 is fetched and it is added to processor register R1. Sum is stored in R1.

Using the 3rd instruction, contents of memory location NUM3 is fetched and it is added to processor register R1. Sum is stored in R1.

Here in this program execution, 'n' number of add instructions are required to add a number (if we use straight line sequence technique). So nth add instruction will add nth value to Register R1. Sum is stored in R1.

Move R1, SUM instructions move the contents of processor register R1 to memory location SUM.

Finally after the execution of this program sum is stored into memory location SUM. This value can be fetched later for displaying or for printing.

★ Drawback of this technique is that more numbers of add instructions are required for adding 'n' numbers. This will take more memory space & also it will take more time to execute. To overcome this drawback we can use branching instructions to add 'n' numbers are explained below.

Program Structure Using a loop to add 'n' numbers.

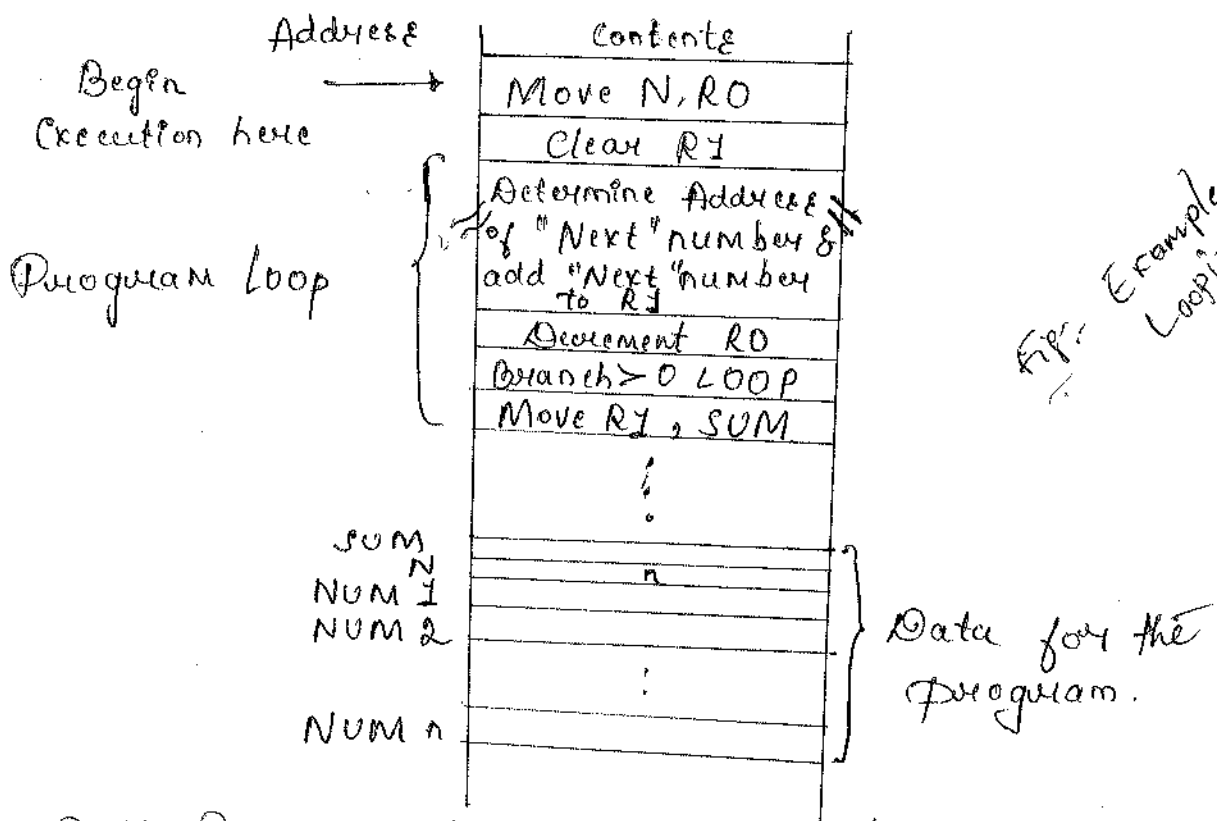


Fig 1: Program structure using a loop to add 'n' numbers.

Fig 1 shows the main memory structure & layout. In this layout & structure, we can use program for adding 'n' numbers using branching technique. Execution begins from Move N, R0 instructions.

In this layout, both program and required data are stored as shown in figure.

Using Move N, R0 instruction, content of memory location N is fetched & moved to processor register R0.

Using Clear R1 instruction, contents of processor register R1 is cleared (R1 set to 0).

Next using some instructions,

we can determine address of "Next" number and add "Next" number to R1. Sum is stored in R1.

Using Decrement RO instruction, contents of register RO is decremented by 1.

Using Branch > 0 LOOP instruction, content of register RO is checked whether it is 0 or not. If it is greater than 0, control must be transferred back to LOOP label. Otherwise the program should be terminated. Sum value is available in R1 register. It is stored in SUM memory location.

Condition Codes

(*) Processor has to keep information about the result of arithmetic and/or logical operation for subsequent use in conditional branching instructions.

(*) Information is recorded in individual bits are called condition code flags. The flag is nothing but a "flip flop" which can be set to 1 or cleared to 0 depending on the results of the Arithmetic and/or Logical operation performed (i.e., content of Accumulator).

(*) These flags are grouped in a special register called Condition Code Register. This information is used in case of branching instructions. Condition Code Register is also called as status Register.

In Condition Code Register we have flags, each flag contains one bit of information as explained below :

Negative flag (N)	Zero flag (Z)	Overflow flag (V)	Carry flag (C)
----------------------	------------------	----------------------	-------------------

Condition Code Register Or Status Register.

N (Negative) — Set to 1 if the result is negative ;
else cleared to 0

Z (Zero) — Set to 1 if the result is zero ;
else cleared to 0

V (Overflow) — Set to 1 if arithmetic overflow
occurs ; else cleared to 0

C (Carry) — Set to 1 if carry-out results
from the operation ; else cleared to 0

Based on the value of the flags, a conditional branch instruction can be used in the program, to alter the program sequence.

ADDRESSING MODES

The different ways or methods used for accessing an operand are referred to as *addressing modes*.

OR

The various formats of representing operand of an instruction or location of an operand is called as "Addressing Mode".

The different types of Addressing Modes are

- a) Register Addressing
- b) Direct Addressing
- c) Immediate Addressing
- d) Indirect Addressing
- e) Index Addressing
- f) Relative Addressing
- g) Auto Increment Addressing
- h) Auto Decrement Addressing

} Additional Addressing modes.

Variables & constants are the simplest data-types and are found in almost every computer program.

In assembly language, a variable is represented by allocating a register (or memory-location) to hold its value. Thus, the value can be changed as needed using appropriate instructions.

Effective Address:

It is the address of memory location where actually the operand is stored.

a) **Register Mode**

- The operand is the contents of a register.
- The name (or address) of the register is given in the instruction.
- Registers are used as temporary storage locations where the data in a register are accessed.

$$EA = R_i$$

Example: *Move R1, R2*

Copy content of register R1 into register R2

b) **Absolute (Direct) Mode** or Memory Addressing mode

- The operand is in a memory-location.
- The address of memory-location is given explicitly in the instruction.
- For example, the instruction,

Example: Move LOC, R2

$$EA = LOC$$

Copy content of memory-location LOC into register R2

c) **Immediate Mode**

- The operand is given explicitly in the instruction.

- For example, the instruction,

Example: Move #200, R0

$$EA = \text{operand} = \text{value}$$

Place the value 200 in register R0

- Clearly, the immediate mode is only used to specify the value of a source-operand.

d) **Indirect Addressing Mode**

In this case, 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.

We refer to this address as the *effective address (EA)* of the operand.

- The EA of the operand is the contents of a **register or memory-location** whose address appears in the instruction.
- Indirect addressing through a register that contains the address of an operand is called a *pointer*. {The indirection is denoted by () sign around the register or memory-location}.

Example: Add (R1), R0

$$EA = (R1)$$

The operand is in memory. Register R1 gives the effective-address (B) of the operand. The data is read from location B and added to contents of register R0.

* To execute the Add instruction in fig (a), the processor uses the value which is in register R1, as the EA 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.

* Indirect addressing through a memory location is also possible as shown in fig (b). In this case, the processor first reads the contents of memory location A, and then requests a second read operation using the value B as an address to obtain the operand.

$$EA = [LOC]$$

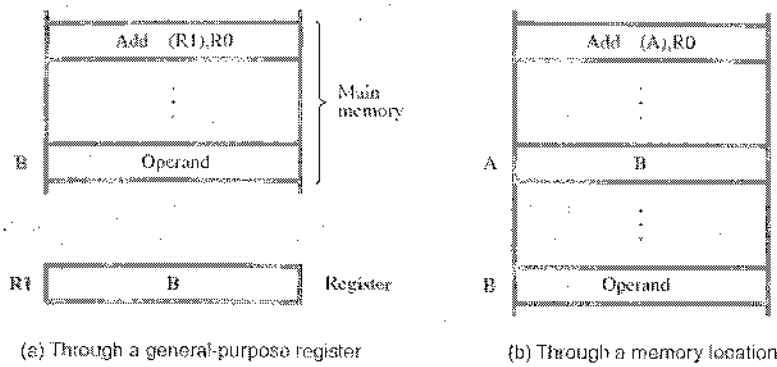
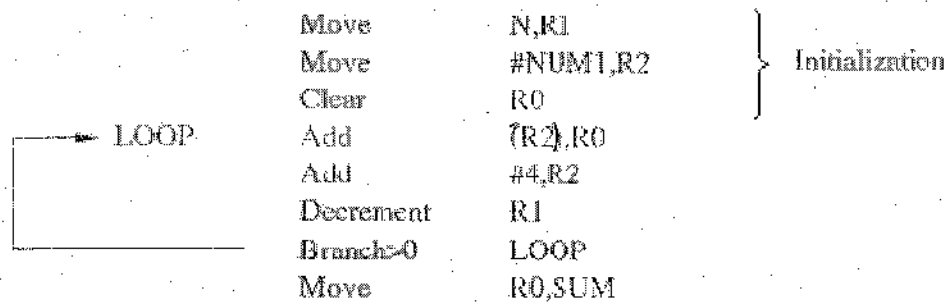


Figure 2.11 Indirect addressing.

Example:

* In above program, Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2.

* The initialization-section of the program loads the counter-value n from memory-location N into $R1$ and uses the immediate addressing-mode to place the address value $NUM1$, which is the address of the first number in the list, into $R2$. Then it clears $R0$ to 0.

* The first two instructions in the loop implement the unspecified instruction block starting at $LOOP$.

* The first time through the loop, the instruction $Add (R2), R0$ fetches the operand at location $NUM1$ and adds it to $R0$.

* The second Add instruction adds 4 to the contents of the pointer $R2$, so that it will contain the address value $NUM2$ when the above instruction is executed in the second pass through the loop.

e) Index Addressing Mode

A different kind of flexibility for accessing operands is useful in dealing with lists and arrays.

Index mode

- The operation is indicated as $X(R_i)$

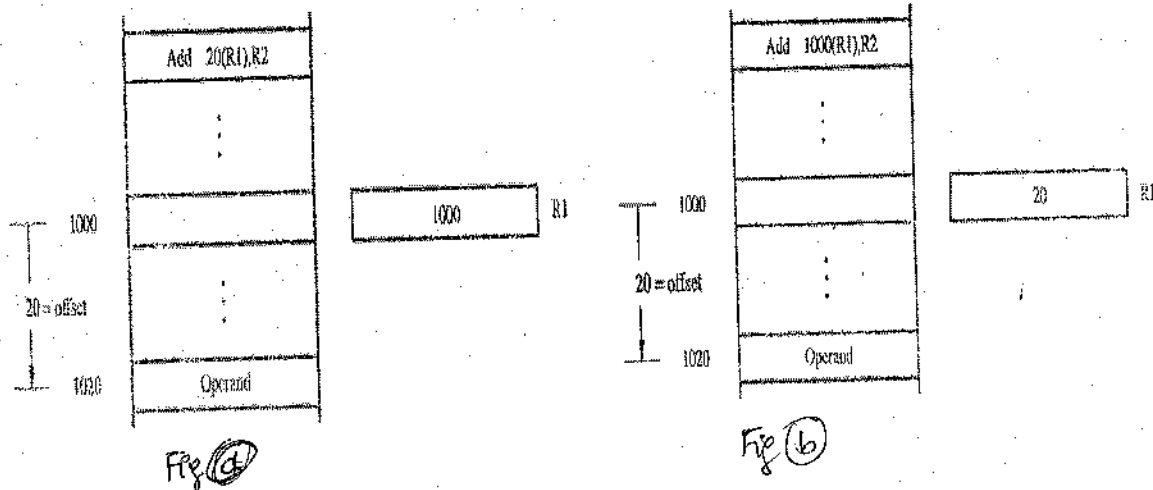
$$EA = [R_i] + X$$

where X = the constant value contained in the instruction, R_i = the name of the index register.

- The effective-address of the operand is given by $EA = X + [R_i]$
- The contents of the index-register are not changed in the process of generating the effective-address.
- In an assembly language program, the constant X may be given either
 - as an explicit number or
 - as a symbolic-name representing a numerical value.

* Fig(a) illustrates two ways of using the Index mode. In fig(a), the index register, R_1 , contains the address of a memory location, and the value X defines an offset (also called a displacement) from this address to the location where the operand is found.

* An alternative use is illustrated in fig(b). 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.



Base with Index Mode

- Another version of the Index mode uses 2 registers which can be denoted as (R_i, R_j)
- Here, a second register may be used to contain the offset X .
- The effective-address of the operand is given by $EA = [R_i] + [R_j]$
- This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Base with Index & Offset Mode

- Another version of the Index mode uses 2 registers plus a constant, which can be

denoted as $X(R_i, R_j)$

- The effective-address of the operand is given by $EA = X + [R_i] + [R_j]$
- This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (R_i, R_j) part of the addressing- mode. In other words, this mode implements a 3-dimensional array.

f) Relative Mode

- This is similar to index-mode with an exception: The effective address is determined using the PC in place of the general purpose register R_i .
- The operation is indicated as $X(PC)$. $EA = [PC] + X$
- $X(PC)$ denotes an effective-address of the operand which is X locations above or below the current contents of PC.
- Since the addressed-location is identified "relative" to the PC, the name Relative mode is associated with this type of addressing.
- This mode is used commonly in conditional branch instructions.
- An instruction such as

Example: Branch > 0 LOOP Causes program execution to go to the branch target location identified by name LOOP if branch condition is satisfied.

ADDITIONAL ADDRESSING MODES

The following 2 modes are useful for accessing data items in successive locations in the memory.

g) Auto-increment Mode

- The effective-address of operand is the contents of a register specified in the instruction (Fig:2.16).
- After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
- Implicitly, the increment amount is 1.
- This mode is denoted as $(R_i) +$ where R_i = pointer register

Example:

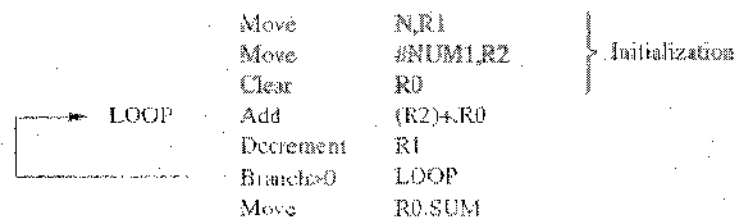


Fig: 2.16

h) Auto-decrement 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.
- This mode is denoted as $EA = \text{Decrement } R_i$
 $-(R_i)$ where R_i = pointer register
- These 2 modes can be used together to implement an important data structure called a stack.
- This mode used to accessing array elements in reverse order.

Example

```

MOVE N1, R1
MOVE #Nmemn+4, R2
clear R0
Loop  Add-(R2), R0
      Decrement R1
      Branch >0 Loop
      Move R0, SUM
  
```

Assembly Language

- Computer understands only machine language, But writing machine language prog is time consuming & less intelligible method.
- In assembly language programs are written by using simple English like statements known as "mnemonics".
- programs are specific to processor family.
- A complete set of mnemonics for processor constitute "Assembly Language".

- The set of rules governing the use of mnemonics in the program is called the 'syntax of the language'.
- Assembly language is m/c dependent.