

Module 3

BASIC STRUCTURE OF COMPUTERS

TYPES OF COMPUTERS

1. Desktop Computers

- These are most commonly used computers in home, schools and offices.
- This has
 - processing- & storage-units
 - video & audio output-units
 - Keyboard & mouse input-units.

2. Notebook Computers (Laptops)

- This is a compact version of a personal-computer (PC) made as a portable-unit.

3. Workstations

- These have more computational-power than PC.

4. Enterprise Systems (Mainframes)

- These are used for business data-processing.
- These have large computational-power and larger storage-capacity than workstations.
- These are referred to as
 - server at low-end and
 - Super-computers at high end.

5. Servers

- These have large database storage-units and can also execute requests from other computers.
- These are used in banks & educational institutions.

6. Super Computers

- These are used for very complex numerical-calculations.
- These are used in weather forecasting, aircraft design and military applications.

FUNCTIONAL UNITS

- A computer consists of 5 functionally independent main parts:
1) input, 2) memory, 3) arithmetic & logic, 4) output and 5) control units.

Input Unit

- The computer accepts the information in the form of program & data through an input-device.
Eg: keyboard
- Whenever a key is pressed, the corresponding letter/digit is automatically translated into its corresponding binary-code and transmitted over a cable to either the memory or the processor.

Memory Unit

- This unit is used to store programs & data.
- There are 2 classes of storage:
 - 1) **Primary-storage** is a fast-memory that operates at electronic-speed. Programs must be stored in the memory while they are being executed.

2) **Secondary-storage** is used when large amounts of data & many programs have to be stored. Eg: magnetic disks and optical disks(CD-ROMs).

- The memory contains a large number of semiconductor storage cells (i.e. flip-flops), each capable of storing one bit of information. The memory is organized so that the contents of one word can be stored or retrieved in one basic operation. Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called RAM (Random Access Memory).

ALU (Arithmetic & Logic Unit)

- This unit is used for performing arithmetic & logical operations. Any arithmetic operation is initiated by bringing the required operand into the processor (i.e. registers), where the operation is performed by the ALU.

Output Unit

- This unit is used to send processed-results to the outside world. Eg: printer, graphic displays etc.

Control Unit

- This unit is used for controlling the activities of the other units (such as memory, I/O device). This unit sends control-signals (read/write) to other units and senses their states. Data transfers between processor and memory are also controlled by the control-unit through timing-signals. Timing-signals are signals that determine when a given action is to take place.

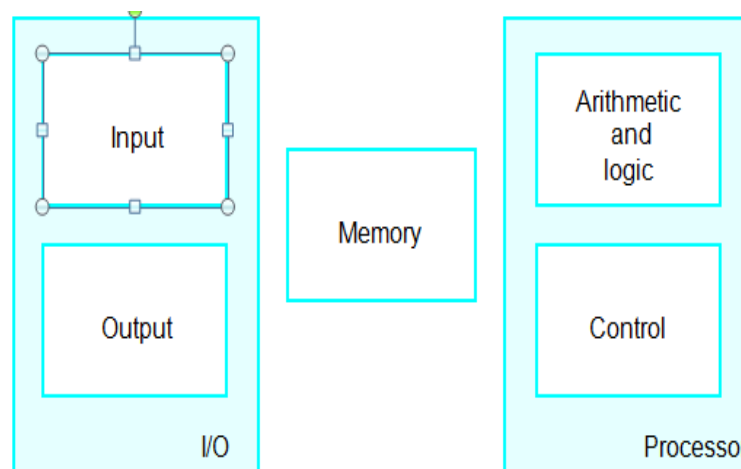


Fig: Basic functional units of computer

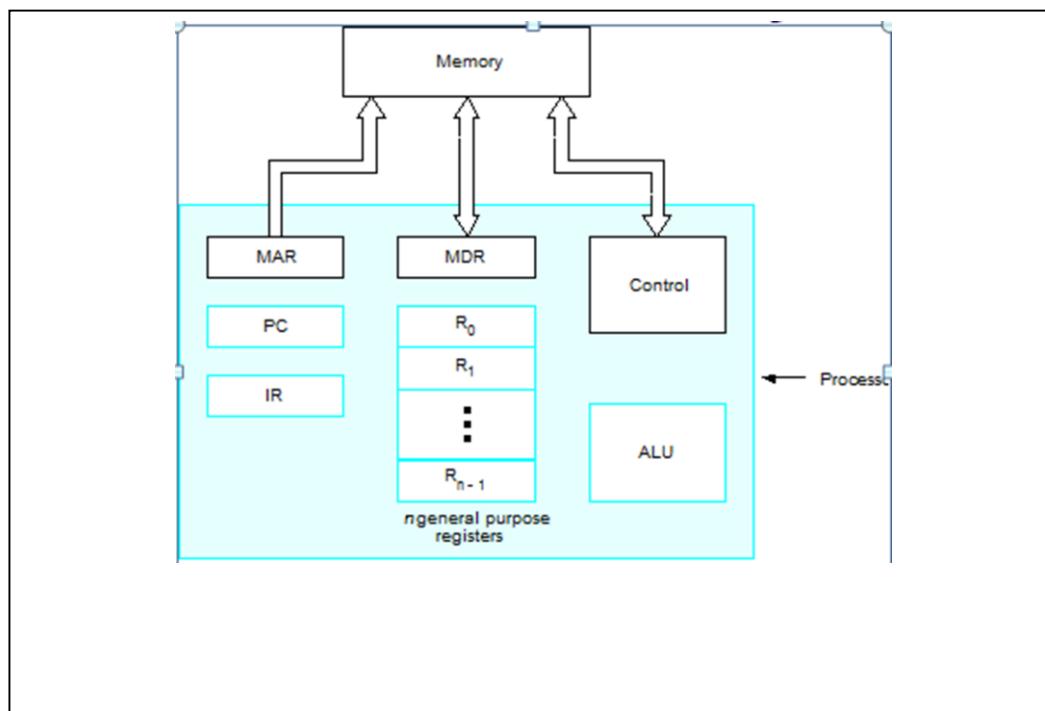
BASIC OPERATIONAL CONCEPTS

- 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, it's 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].



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.
- Systems that contain multiple buses achieve more concurrency in operations by allowing 2 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.

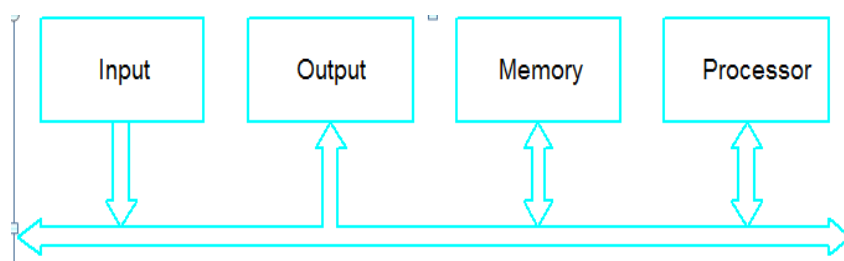


Fig: Single bus structure

PERFORMANCE

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.
- 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)

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

$$T = \frac{N \times S}{R} \quad \text{-----(1)}$$

- This 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.

PERFORMANCE MEASUREMENT

- SPEC (System Performance Evaluation Corporation) selects & publishes the standard programs along with their test results for different application domains. The SPEC rating is computed as follows

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

- If SPEC rating=50 means that the computer under test is 50 times as fast as reference computer.
- The test is repeated for all the programs in the SPEC suite, and the geometric mean of the results is computed.

Let SPEC_i be the rating for program i in the suite. The overall SPEC rating for the computer is given by

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

where n=number of programs in the suite

MACHINE INSTRUCTIONS & PROGRAMS

MEMORY LOCATIONS & ADDRESSES

- The memory consists of many millions of storage cells (flip-flops), each of which can store a bit of information having the value 0 or 1 (Figure 2.5).
- Each group of n bits is referred to as a word of information, and n is called the word length.
- The word length can vary from 8 to 64 bits.
- A unit of 8 bits is called a byte.
- Accessing the memory to store or retrieve a single item of information (either a word or a byte) requires distinct addresses for each item location. (It is customary to use numbers from 0 through $2^k - 1$ as the addresses of successive locations in the memory).
- If $2^k =$ number of addressable locations, then 2^k addresses constitute the address space of the computer. For example, a 24-bit address generates an address space of 2^{24} locations (16MB).

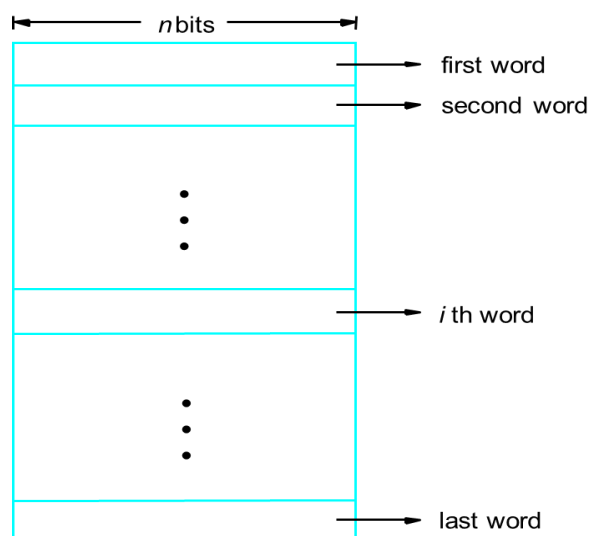
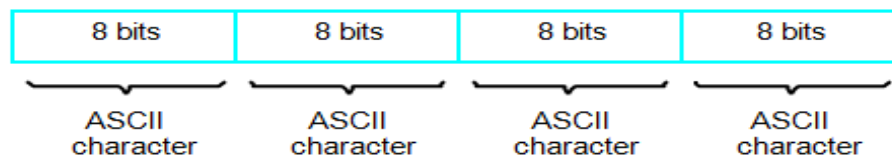
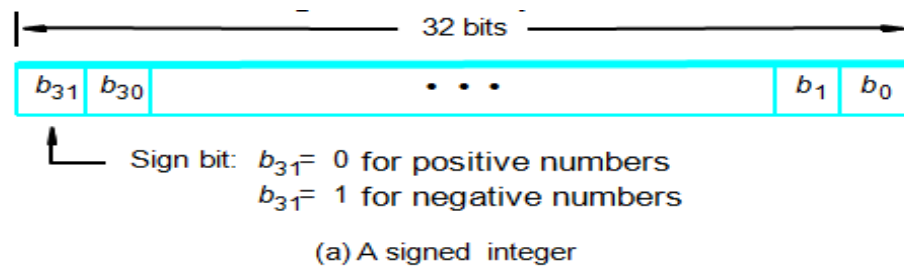


Figure 2.5. Memory words.



Characters can be letters of the alphabet, decimal digits, punctuation marks and so on.

- Characters are represented by codes that are usually 8 bits long. E.g. ASCII code
- The three basic information quantities are: bit, byte and word.
- A byte is always 8 bits, but the word length typically ranges from 1 to 64 bits.
- It is impractical to assign distinct addresses to individual bit locations in the memory.

BYTE ADDRESSABILITY

- In byte addressable memory, successive addresses refer to successive byte locations in the memory.
- Byte locations have addresses 0, 1, 2,
- If the word length is 32 bits, successive words are located at addresses 0, 4, 8, . . with each word having 4 bytes.

BIG-ENDIAN AND LITTLE-ENDIAN ASSIGNMENTS

- There are two ways in which byte addresses are arranged.
 - 1) **Big-endian assignment:** lower byte addresses are used for the more significant bytes of the word (Figure 2.7).
 - 2) **Little-endian assignment:** lower byte addresses are used for the less significant bytes of the word
- In both cases, byte addresses 0, 4, 8, are taken as the addresses of successive words in the memory.

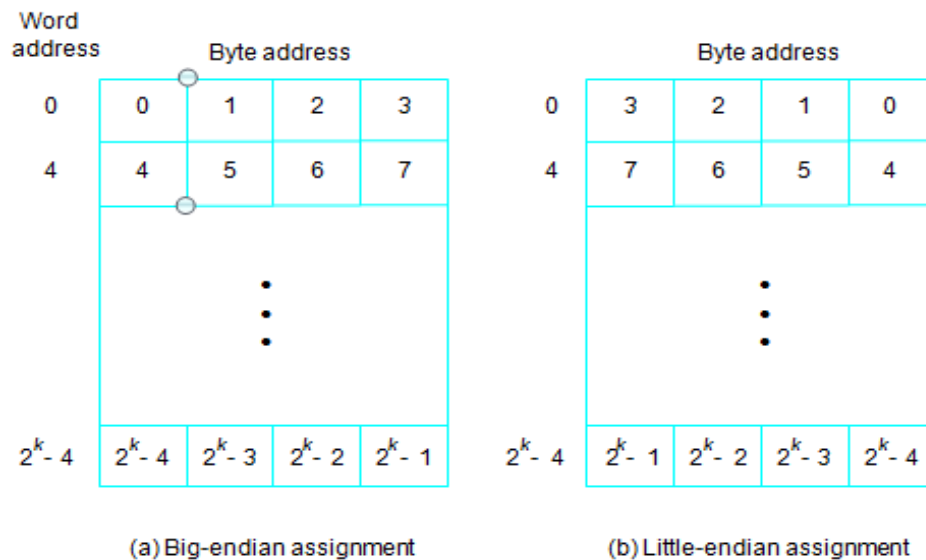


Fig: Byte and word Addressing

WORD ALIGNMENT

- Words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.
- For example, if the word length is 16(2 bytes), aligned words begin at byte addresses 0, 2, 4 . . . And for a word length of 64, aligned words begin at byte addresses 0, 8, 16.
- Words are said to have unaligned addresses, if they begin at an arbitrary byte address.

ACCESSING NUMBERS, CHARACTERS & CHARACTERS STRINGS

- A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.
- There are two ways to indicate the length of the string.
 - a special control character with the meaning "end of string" can be used as the last character in the string, or
 - a separate memory word location or processor register can contain a number indicating the length of the string in bytes.

MEMORY OPERATIONS

- Two basic operations involving the memory are:
 - Load(Read/Fetch) and
 - Store(Write).
- The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged.
- The steps for Load operation:
 - 1) Processor sends the address of the desired location to the memory
 - 2) Processor issues „read“ signal to memory to fetch the data
 - 3) Memory reads the data stored at that address

- 4) Memory sends the read data to the processor
- The Store operation transfers the information from the processor register to the specified memory location. This will destroy the original contents of that memory location.
- The steps for Store operation are:
 - 1) Processor sends the address of the memory location where it wants to store data
 - 2) Processor issues „write“ signal to memory to store the data
 - 3) Content of register(MDR) is written into the specified memory location.

INSTRUCTIONS & INSTRUCTION SEQUENCING

- A computer must have instructions capable of performing 4 types of operations:
 - 1) Data transfers between the memory and the processor registers (MOV, PUSH, POP, XCHG),
 - 2) Arithmetic and logic operations on data (ADD, SUB, MUL, DIV, AND, OR, NOT),
 - 3) Program sequencing and control (CALL.RET, LOOP, INT),
 - 4) I/O transfers (IN, OUT)

REGISTER TRANSFER NOTATION (RTN)

- We identify a memory location by a symbolic name (in uppercase alphabets).
For example, LOC, PLACE, NUM etc indicate memory locations R0, R5 etc indicate processor register. DATAIN, OUTSTATUS etc indicate I/O registers.
- For example,
 $R \leftarrow [LOC]$ means that the contents of memory location LOC are transferred into processor register R1 (The contents of a location are denoted by placing square brackets around the name of the location). $R3 \leftarrow [R1] + [R2]$ indicates 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 RTN (Register Transfer Notation).

ASSEMBLY LANGUAGE NOTATION

- To represent machine instructions and programs, assembly language format can be used.
 - For example,
Move LOC, R1; This instruction transfers data from memory-location LOC to processor-register R1. The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.
- Add R1, R2, R3;* This instruction adds 2 numbers contained in processor-registers R1 and R2, and places their sum in R3.

A computer performs its task according to the program stored in memory. A program is a collection of instructions which tell the processor to perform a basic operation like addition, reading from keyboard etc.

- Possible locations that may be involved in data transfers are memory locations, processor registers or registers in the I/O subsystem.

BASIC INSTRUCTION TYPES

- $C=A+B$; This statement is a command to the computer to add the current values of the two variables A and B, and to assign the sum to a third variable C.
- When the program is compiled, each variable is assigned a distinct address in memory.
- The contents of these locations represent the values of the three variables
- The statement $C\leftarrow[A]+[B]$ indicates that the contents of memory locations A and B are fetched from memory, transferred to the processor, sum is computed and then result is stored in memory location C.

Three-Address Instruction

- The instruction has general format

Operation Source1, Source2, Destination

- For example, *Add A, B, C*; operands A and B are called the source operands, C is called the destination operand, and Add is the operation to be performed.

Two-Address Instruction

- The instruction has general format

Operation Source, Destination

- For example, *Add A, B*; performs the operation $B\leftarrow[A]+[B]$.
- When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that operand B is both a source and a destination.
- The operation $C\leftarrow[A]+[B]$ can be performed by the two-instruction sequence

Move B, C

Add A, C

One-Address Instruction

- The instruction has general format

Operation Source/Destination

- For example,
 - *Add A* ; Add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator.
 - *Load A*; This instruction copies the contents of memory location A into the accumulator and
 - *Store A*; This instruction copies the contents of the accumulator into memory location A.

The operation $C\leftarrow[A]+[B]$ can be performed by executing the sequence of instructions

Load A

Add B

Store C

- The operand may be a source or a destination depending on the instruction. In the Load instruction, address A specifies the source operand, and the destination location, the accumulator, is implied. On the other hand, C denotes the destination location in the Store instruction, whereas the source, the accumulator, is implied.

Zero-Address Instruction

- The locations of all operands are defined implicitly. The operands are stored in a structure called pushdown stack. In this case, the instructions are called zero-address instructions.

- Access to data in the registers is much faster than to data stored in memory locations because the registers are inside the processor.

- Let R_i represent a general-purpose register. The instructions

Load A, R_i

Store R_i, A

Add A, R_i

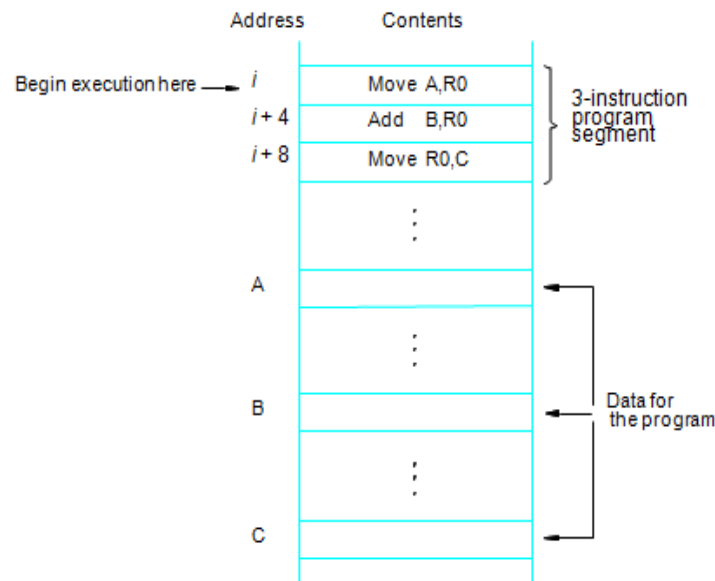
are generalizations of the Load, Store and Add Instructions for the single-accumulator case, in which register R_i performs the function of the accumulator.

- In processors where arithmetic operations are allowed only on operands that are in processor registers, the $C=A+B$ task can be performed by the instruction sequence

INSTRUCTION EXECUTION & STRAIGHT-LINE SEQUENCING

The program is executed as follows:

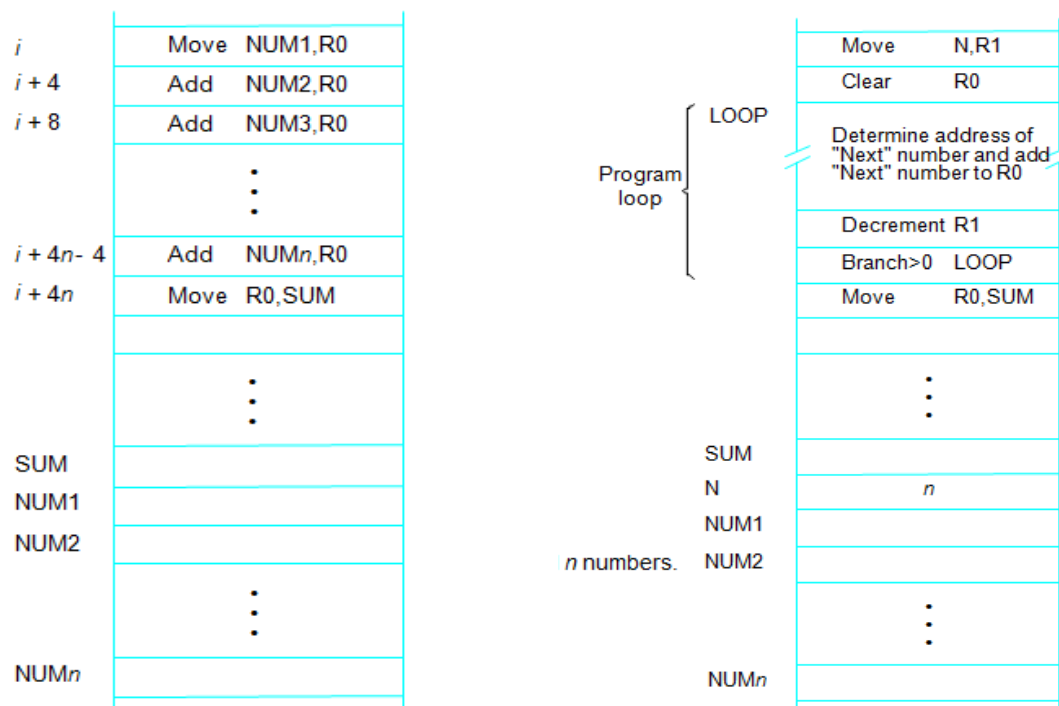
- 1) Initially, the address of the first instruction is loaded into PC (Program counter is a register which holds the address of the next instruction to be executed)
- 2) Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing* (Figure 2.8)
- 3) During the execution of each instruction, the PC is incremented by 4 to point to the next instruction.
- 4) Executing given instruction is a **two-phase procedure**.
 - i) In fetch phase, the instruction is fetched from the memory location (whose address is in the PC) and placed in the IR of the processor.
 - ii) In execute phase, the contents of IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor.

Figure 2.8. A program for $C \leftarrow [A] + [B]$.

BRANCHING

Consider the task of adding a list of n numbers (Figure 2.10).

- The loop is a straight line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction *Branch*>0.
- During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to R0.
- Register R1 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R1 at the beginning of the program.
- Within the body of the loop, the instruction *Decrement R1* reduces the contents of R1 by 1 each time through the loop.
- Then *Branch* instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address called the branch target.
- A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.



CONDITION CODES

- The processor keeps track of information about the results of various operations. This is accomplished by recording the required information in individual bits, called *condition code flags*.
- These flags are grouped together in a special processor-register called the *condition code register* (or status register).
- Four commonly used flags are
 - N (negative) set to 1 if the result is negative, otherwise cleared to 0
 - Z (zero) set to 1 if the result is 0; otherwise, cleared to 0
 - V (overflow) set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
 - C (carry) set to 1 if a carry-out results from the operation; otherwise cleared to 0.

ADDRESSING MODES

- The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes* (Table 2.1).

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R _i	EA = R _i
Absolute (Direct)	LOC	EA = LOC
Indirect	(R _i) (LOC)	EA = [R _i] EA = [LOC]
Index	X(R _i)	EA = [R _i] + X
Base with index	(R _i , R _j)	EA = [R _i] + [R _j]
Base with index and offset	X(R _i , R _j)	EA = [R _i] + [R _j] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R _i) ⁺	EA = [R _i]; Increment R _i
Autodecrement	-(R _i)	Decrement R _i ; EA = [R _i]

EA = effective address
Value = a signed number

IMPLEMENTATION OF VARIABLE AND CONSTANTS

- 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.

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.
- For example, the instruction,

Move R1, R2 ; Copy content of register R1 into register R2

Absolute (Direct) Mode

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

Move LOC, R2 ; Copy content of memory-location LOC into register R2

Immediate Mode

- The operand is given explicitly in the instruction.
- For example, the instruction,

Move #200, R0 ; Place the value 200 in register R0

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

INDIRECTION AND POINTERS

- 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.

Indirect Mode

- The EA of the operand is the contents of a register (or memory-location) whose address appears in the instruction.

- The register (or memory-location) that contains the address of an operand is called a *pointer*. {The indirection is denoted by () sign around the register or memory-location}.

E.g: *Add (R1), R0*; 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, then requests a second read operation using the value B as an address to obtain the operand.

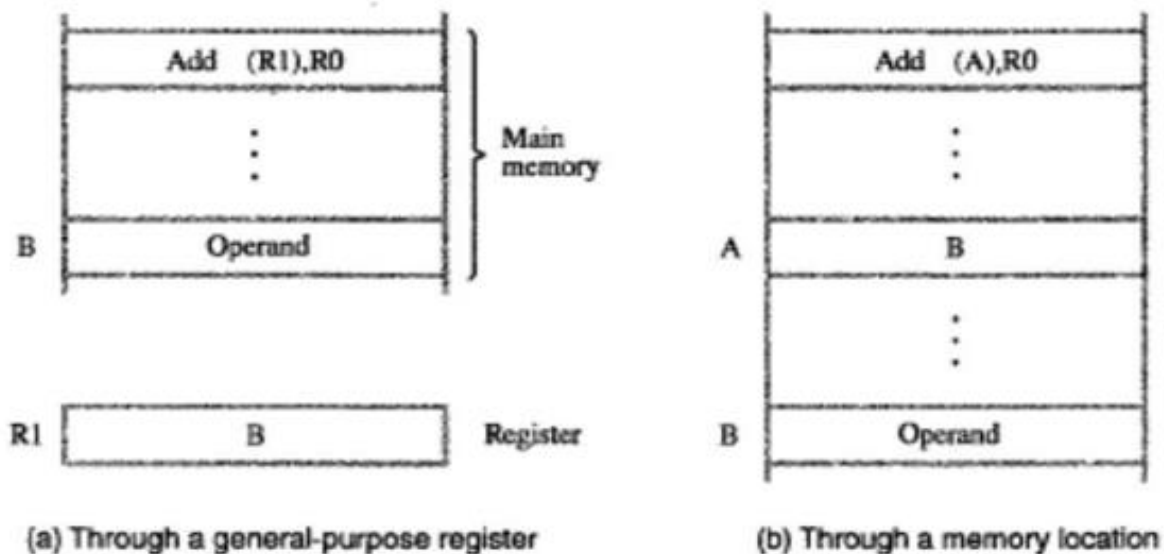


Figure 2.11 Indirect addressing.

Address	Contents	
	Move	N,R1
	Move	#NUM1,R2
	Clear	R0
	Add	(R2),R0
	Add	#4,R2
	Decrement	R1
	Branch>0	LOOP
	Move	R0,SUM

Diagram showing a loop structure: An arrow labeled 'LOOP' points to the 'Branch>0' instruction, which then loops back to the 'Add (R2),R0' instruction.

Initialization section (indicated by a bracket):

- Move N,R1
- Move #NUM1,R2
- Clear R0

Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

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.

INDEXING AND ARRAYS

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

Index mode

- The operation is indicated as $X(R_i)$
 - 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, R1, 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.

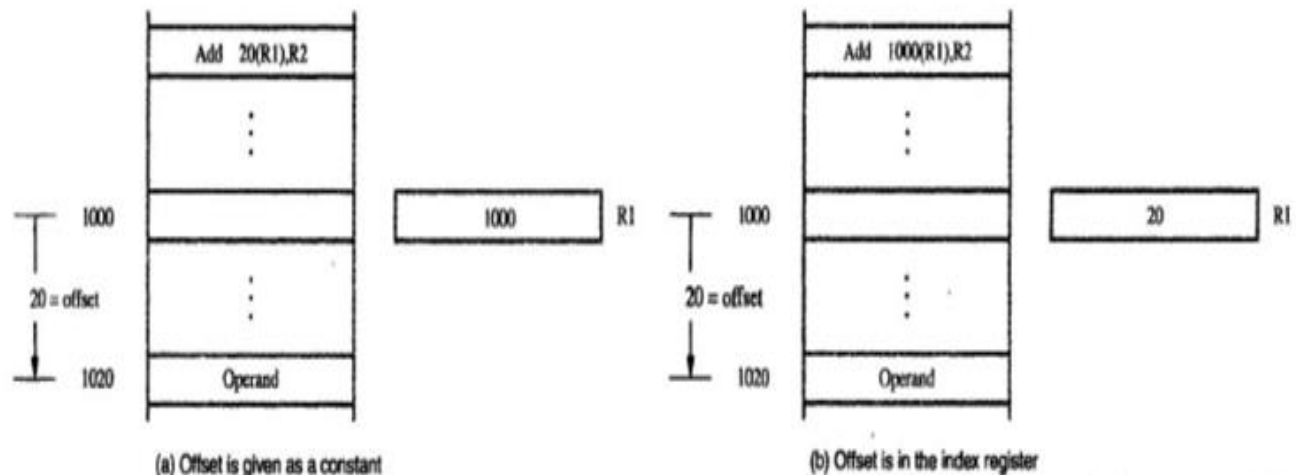


Figure 2.13 Indexed addressing.

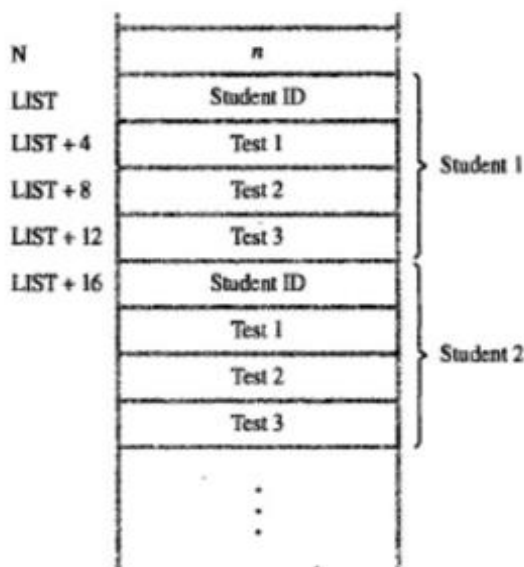


Figure 2.14 A list of students' marks.

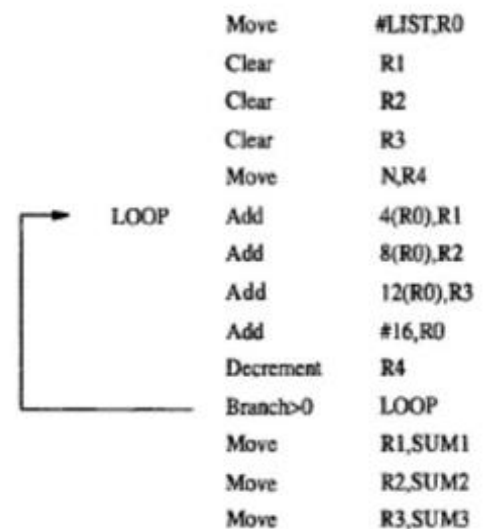


Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

Base with Index Mode

- Another version of the Index mode uses 2 registers which can be denoted as (Ri, Rj)

- Here, a second register may be used to contain the offset X.
- The second register is usually called the *base register*.
- The effective-address of the operand is given by $EA=[Ri]+[Rj]$
- 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 $X(Ri, Rj)$
- The effective-address of the operand is given by $EA=X+[Ri]+[Rj]$
- 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 (Ri, Rj) part of the addressing- mode. In other words, this mode implements a 3-dimensional array.

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 Ri .
- The operation is indicated as $X(PC)$.
- $X(PC)$ 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
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.

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

$(Ri)+$; where Ri =pointer register

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 $-(R_i)$;where R_i =pointer register
- These 2 modes can be used together to implement an important data structure called a stack.

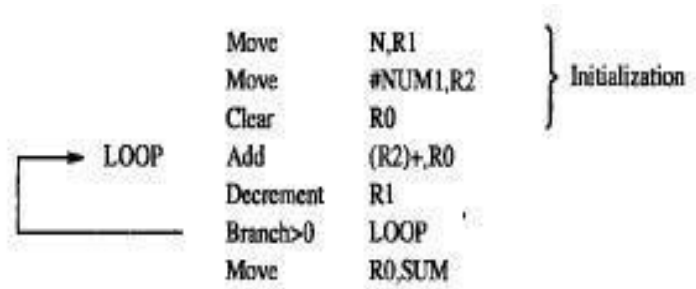


Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.