COMPUTER ORGANIZATION (18EC35)

MODULE-1

BASIC STRUCTURE OF COMPUTERS

1.1 Computer types

A computer can be defined as a fast electronic calculating machine that accepts the (data) digitized input information process it as per the list of internally stored instructions and produces the resulting information.

List of instructions are called programs & internal storage is called computer memory.

The different types of computers are

1. **Personal computers: -** This is the most common type found in homes, schools, Business offices etc., It is the most common type of desk top computers with processing and storage units along with various input and output devices.
2. **Note book computers: -** These are compact and portable versions of PC
3. **Work stations: -** These have high resolution input/output (I/O) graphics capability, but with same dimensions as that of desktop computer. These are used in engineering applications of interactive design work.
4. **Enterprise systems: -** These are used for business data processing in medium to large corporations that require much more computing power and storage capacity than work stations. Internet associated with servers have become a dominant worldwide source of all types of information.
5. **Super computers: -** These are used for large scale numerical calculations required in the applications like weather forecasting etc.,

1.2 Functional unit

A computer consists of five functionally independent main parts input, memory, arithmetic logic unit (ALU), output and control unit.

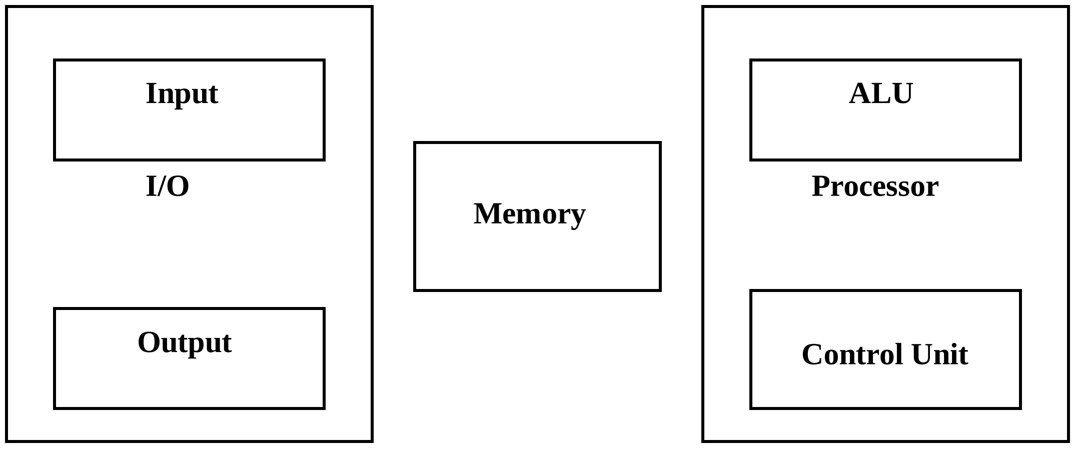


Fig a : Functional units of computer

Input device accepts the coded information as source program i.e. high level language. This is either stored in the memory or immediately used by the processor to perform the desired operations. The program stored in the memory determines the processing steps. Basically the computer converts one source program to an object program. i.e. into machine language.

Finally the results are sent to the outside world through output device. All of these actions are coordinated by the control unit.

Input unit: -

The source program/high level language program/coded information/simply data is fed to a computer through input devices keyboard is a most common type. Whenever a key is pressed, one corresponding word or number is translated into its equivalent binary code over a cable & fed either to memory or processor.

Joysticks, trackballs, mouse, scanners etc are other input devices.

Memory unit: -

Its function into store programs and data. It is basically to two types

1. **Primary memory**
2. **Secondary memory**

1. Primary memory: - Is the one exclusively associated with the processor and operates at the electronics speeds programs must be stored in this memory while they are being executed. The memory contains a large number of semiconductors storage cells. Each capable of storing one bit of information. These are processed in a group of fixed site called word.

To provide easy access to a word in memory, a distinct address is associated with each word location. **Addresses are** numbers that identify memory location.

Number of bits in each word is called word length of the computer. Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of 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 in called memory access time. Memory which is only readable by the user and contents of which can't be altered is called read only memory (ROM) it contains operating system.

Caches are the small fast RAM units, which are coupled with the processor and are aften contained on the same IC chip to achieve high performance. Although primary storage is essential it tends to be expensive.
2 Secondary memory: - Is used where large amounts of data & programs have to be stored, particularly information that is accessed infrequently.

Examples: - Magnetic disks & tapes, optical disks (ie CD-ROM's), floppies etc.,

Arithmetic logic unit (ALU):-

Most of the computer operators are executed in ALU of the processor like addition, subtraction, division, multiplication, etc. the operands are brought into the ALU from memory and stored in high speed storage elements called register. Then according to the instructions the operation is performed in the required sequence.

The control and the ALU are may times faster than other devices connected to a computer system. This enables a single processor to control a number of external devices such as key boards, displays, magnetic and optical disks, sensors and other mechanical controllers.

Output unit:-

These actually are the counterparts of input unit. Its basic function is to send the processed results to the outside world.

Examples:- Printer, speakers, monitor etc.

Control unit:-

It effectively is the nerve center that sends signals to other units and senses their states. The actual timing signals that govern the transfer of data between input unit, processor, memory and output unit are generated by the control unit.

MODULE 2

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).

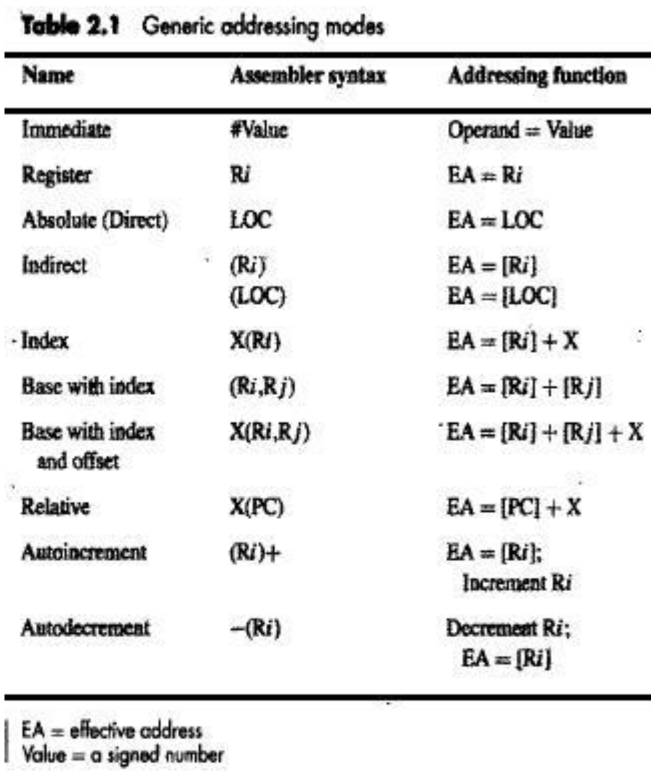


Table 2.1 Generic addressing modes

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

• **Variable** is represented by allocating a memory-location to hold its value.
• Thus, the value can be changed as needed using appropriate instructions.
• There are 2 accessing modes to access the variables:
  1) Register Mode
  2) Absolute 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.
• The absolute mode can represent global variables in the program.
• For example, the instruction

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

Immediate Mode

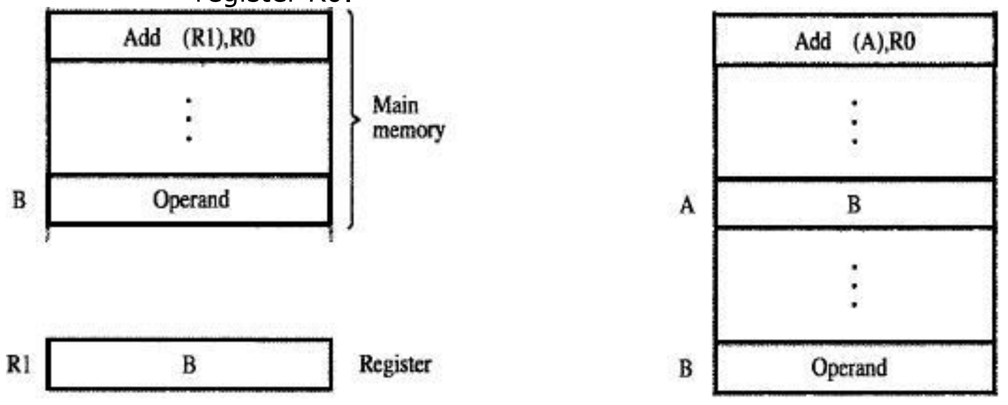
• The operand is given explicitly in the instruction.
• For example, the instruction

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

INDIRECTION AND POINTERS

• Instruction does not give the operand or its address explicitly.
• Instead, the instruction provides information from which the new address of the operand can be determined.
• This address is called **Effective Address (EA)** of the operand.

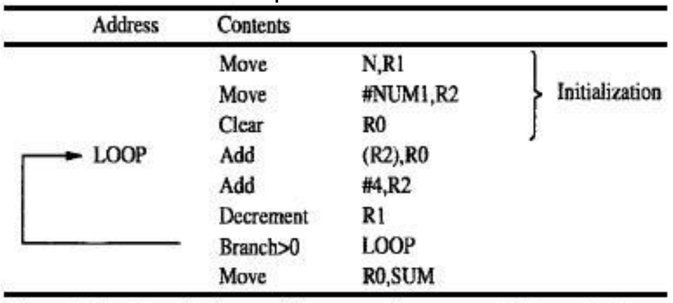
• The EA of the operand is the contents of a register(or memory-location).
• The register (or memory-location) that contains the address of an operand is called a **Pointer**.
• We denote the indirection by
→ name of the register or
→ new address given in the instruction.
E.g: *Add (R1),R0* ;The operand 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.



(a) Through a general-purpose register (b) Through a memory location

Figure 2.11 Indirect addressing.

• To execute the Add instruction in fig 2.11 (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 2.11(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.



```
Address     Contents

            Move        N,R1        }
            Move        #NUM1,R2    } Initialization
            Clear       R0          }
LOOP        Add         (R2),R0
            Add         #4,R2
            Decrement   R1
            Branch>0    LOOP
            Move        R0,SUM
```

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

Program Explanation

• 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(Ri)

where X=the constant value which defines an offset(also called a displacement).

Ri=the name of the index register which contains address of a new location.

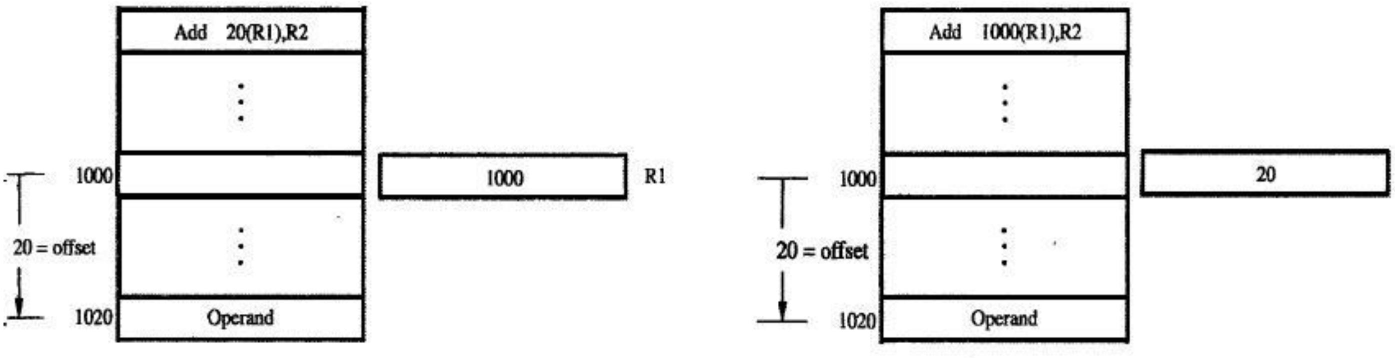
• The effective-address of the operand is given by EA=X+[Ri]

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

• The constant X may be given either

→ as an explicit number or

→ as a symbolic-name representing a numerical value.



(a) Offset is given as a constant (b) Offset is in the index register

Figure 2.13 Indexed addressing.

• 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.

• To find EA of operand:

• 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.

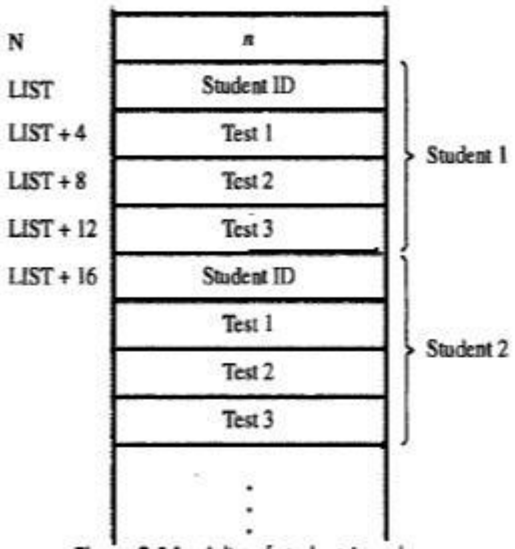


Figure 2.14 A list of students' marks.

```
        Move        #LIST,R0
        Clear       R1
        Clear       R2
        Clear       R3
        Move        N,R4
LOOP    Add         4(R0),R1
        Add         8(R0),R2
        Add         12(R0),R3
        Add         #16,R0
        Decrement   R4
        Branch>0    LOOP
        Move        R1,SUM1
        Move        R2,SUM2
        Move        R3,SUM3
```

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.

• 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 one difference:
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.

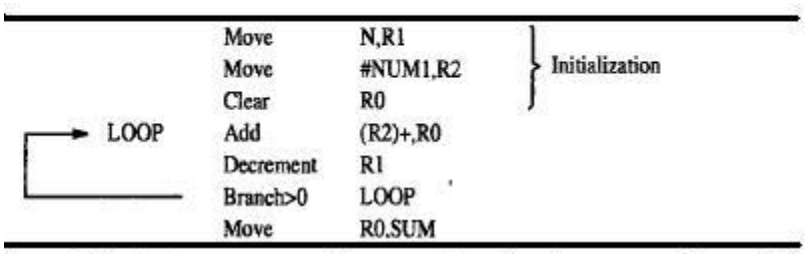
ADDITIONAL ADDRESSING MODES

1) Auto Increment Mode

➢ Effective-address of operand is 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

➢ 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

➢ These 2 modes can be used together to implement an important data structure called a stack.



```
            Move        N,R1       }
            Move        #NUM1,R2   } Initialization
            Clear       R0         }
LOOP        Add         (R2)+,R0
            Decrement   R1
            Branch>0    LOOP
            Move        R0,SUM
```

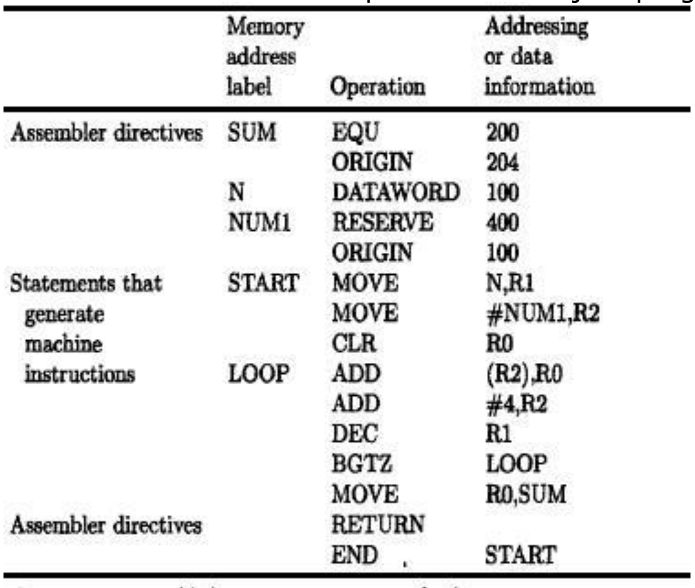
Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

ASSEMBLY LANGUAGE

• We generally use symbolic-names to write a program.
• A complete set of symbolic-names and rules for their use constitute an **Assembly Language**.
• The set of rules for using the mnemonics in the specification of complete instructions and programs is called the **Syntax** of the language.
• Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an **Assembler.**
• The user program in its original alphanumeric text formal is called a **Source Program**, and the assembled machine language program is called an **Object Program**.
For example:
MOVE R0,SUM ;The term MOVE represents OP code for operation performed by instruction.
ADD #5,R3 ;Adds number 5 to contents of register R3 & puts the result back into registerR3.

ASSEMBLER DIRECTIVES

• **Directives** are the assembler commands to the assembler concerning the program being assembled.
• These commands are not translated into machine opcode in the object-program.



	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

Figure 2.18 Assembly language representation for the program in Figure 2.17.

• **EQU** informs the assembler about the value of an identifier (Figure: 2.18).
Ex*: SUM EQU 200* ;Informs assembler that the name SUM should be replaced by the value 200.
• **ORIGIN** tells the assembler about the starting-address of memory-area to place the data block.
Ex*:* ORIGIN 204 ;Instructs assembler to initiate data-block at memory-locations starting from 204.
• **DATAWORD** directive tells the assembler to load a value into the location.
Ex: *N DATAWORD 100 ;*Informs the assembler to load data 100 into the memory-location N(204).
• **RESERVE** directive is used to reserve a block of memory.
Ex: *NUM1 RESERVE 400 ;*declares a memory-block of 400 bytes is to be reserved for data.
• **END** directive tells the assembler that this is the end of the source-program text.
• **RETURN** directive identifies the point at which execution of the program should be terminated.
• Any statement that makes instructions or data being placed in a memory-location may be given a **label**. he label(say N or NUM1) is assigned a value equal to the address of that location.

GENERAL FORMAT OF A STATEMENT

• Most assembly languages require statements in a source program to be written in the form:

Label	*Operation*	*Operands*	*Comment*

1) Label is an optional name associated with the memory-address where the machine language instruction produced from the statement will be loaded.
2) Operation Field contains the OP-code mnemonic of the desired instruction or assembler.
3) Operand Field contains addressing information for accessing one or more operands, depending on the type of instruction.
4) Comment Field is used for documentation purposes to make program easier to understand.

ASSEMBLY AND EXECUTION OF PROGRAMS

• Programs written in an assembly language are automatically translated into a sequence of machine instructions by the **Assembler**.

• **Assembler Program**

→ replaces all symbols denoting operations & addressing-modes with binary-codes used in machine instructions.
→ replaces all names and labels with their actual values.
→ assigns addresses to instructions & data blocks, starting at address given in ORIGIN directive
→ inserts constants that may be given in DATAWORD directives.
→ reserves memory-space as requested by RESERVE directives.

• **Two Pass Assembler** has 2 passes:

1) First Pass: Work out all the addresses of labels.
➢ As the assembler scans through a source-program, it keeps track of all names of numerical-values that correspond to them in a symbol-table.

2) Second Pass: Generate machine code, substituting values for the labels.
➢ When a name appears a second time in the source-program, it is replaced with its value from the table.

• The assembler stores the object-program on a magnetic-disk. The object-program must be loaded into the memory of the computer before it is executed. For this, a **Loader Program** is used.

• **Debugger Program** is used to help the user find the programming errors.

• Debugger program enables the user

→ to stop execution of the object-program at some points of interest &
→ to examine the contents of various processor-registers and memory-location.

BASIC INPUT/OUTPUT OPERATIONS

• Consider the problem of moving a character-code from the keyboard to the processor (Figure: 2.19). For this transfer, buffer-register DATAIN & a status control flags(SIN) are used.

• When a key is pressed, the corresponding ASCII code is stored in a **DATAIN** register associated with the keyboard.

- ➢ **SIN=1** → When a character is typed in the keyboard. This informs the processor that a valid character is in DATAIN.
- ➢ **SIN=0** → When the character is transferred to the processor.

• An analogous process takes place when characters are transferred from the processor to the display. For this transfer, buffer-register DATAOUT & a status control flag SOUT are used.

- ➢ **SOUT=1** → When the display is ready to receive a character.
- ➢ **SOUT=0** → When the character is being transferred to DATAOUT.

• The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a **device interface**.

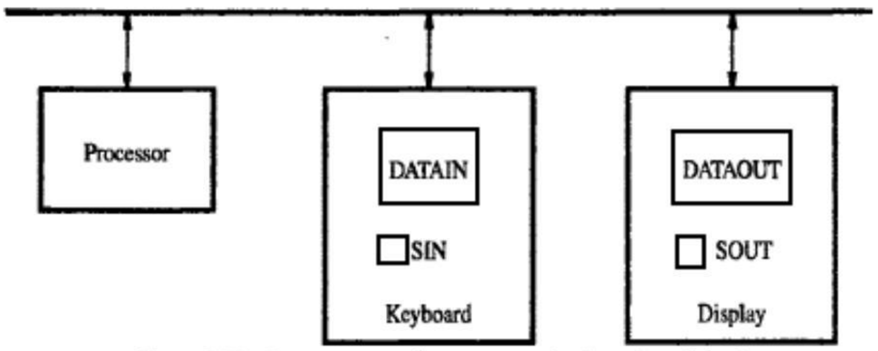


Figure 2.19 Bus connection for processor, keyboard, and display.

Program to read a line of characters and display it

```
        Move       #LOC,R0          Initialize pointer register R0 to point to the
                                      address of the first location in memory
                                      where the characters are to be stored.
READ    TestBit    #3,INSTATUS      Wait for a character to be entered
        Branch=0   READ               in the keyboard buffer DATAIN.
        MoveByte   DATAIN,(R0)      Transfer the character from DATAIN into
                                      the memory (this clears SIN to 0).
ECHO    TestBit    #3,OUTSTATUS     Wait for the display to become ready.
        Branch=0   ECHO
        MoveByte   (R0),DATAOUT     Move the character just read to the display
                                      buffer register (this clears SOUT to 0).
        Compare    #CR,(R0)+        Check if the character just read is CR
                                      (carriage return). If it is not CR, then
        Branch≠0   READ               branch back and read another character.
                                      Also, increment the pointer to store the
                                      next character.
```

Figure 2.20 A program that reads a line of characters and displays it.

MEMORY-MAPPED I/O

• Some address values are used to refer to peripheral device buffer-registers such as DATAIN & DATAOUT.No special instructions are needed to access the contents of the registers; data can be transferred between these registers and the processor using instructions such as Move, Load or Store.

• For example, contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction

MoveByte DATAIN,R1

• The MoveByte operation code signifies that the operand size is a byte.

• The **Testbit** instruction tests the state of one bit in the destination, where the bit position to be tested is indicated by the first operand.

STACKS

• A **stack** is a special type of data structure where elements are inserted from one end and elements are deleted from the same end. This end is called the **top** of the stack (Figure: 2.14).

• The various operations performed on stack:

1) Insert: An element is inserted from top end. Insertion operation is called **push** operation.
2) Delete: An element is deleted from top end. Deletion operation is called **pop** operation.

• 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. This register is called the **Stack Pointer (SP)**.

• If we assume a byte-addressable memory with a 32-bit word length,

1) The push operation can be implemented as

2) The pop operation can be implemented as

Move (SP), ITEM
Add #4, SP

• Routine for a safe pop and push operation as follows:

```
SAFEPOP   Compare    #2000,SP      Check to see if the stack pointer contains
          Branch>0   EMPTYERROR      an address value greater than 2000. If it
                                     does, the stack is empty. Branch to the
                                     routine EMPTYERROR for appropriate
                                     action.
          Move       (SP)+,ITEM    Otherwise, pop the top of the stack into
                                     memory location ITEM.
```

(a) Routine for a safe pop operation

```
SAFEPUSH  Compare    #1500,SP      Check to see if the stack pointer
          Branch≤0   FULLERROR       contains an address value equal
                                     to or less than 1500. If it does, the
                                     stack is full. Branch to the routine
                                     FULLERROR for appropriate action.
          Move       NEWITEM,-(SP) Otherwise, push the element in memory
                                     location NEWITEM onto the stack.
```

(b) Routine for a safe push operation

Figure 2.23 Checking for empty and full errors in pop and push operations.

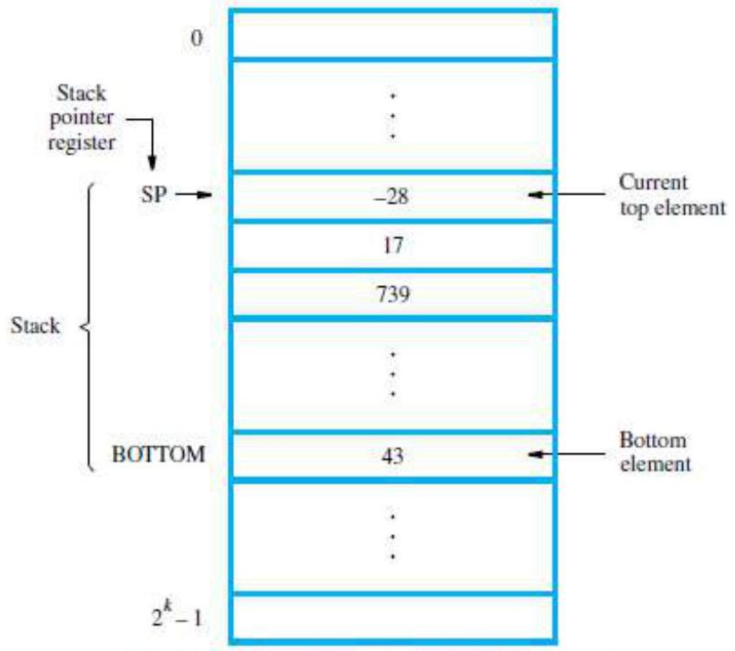


Figure 2.14 A stack of words in the memory.

QUEUE

• Data are stored in and retrieved from a queue on a FIFO basis.

• Difference between stack and queue?

1) One end of the stack is fixed while the other end rises and falls as data are pushed and popped.

2) In stack, a single pointer is needed to keep track of top of the stack at any given time.
In queue, two pointers are needed to keep track of both the front and end for removal and insertion respectively.

3) Without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a circular buffer.

SUBROUTINES

• A subtask consisting of a set of instructions which is executed many times is called a **Subroutine**.

• A Call instruction causes a branch to the subroutine (Figure: 2.16).

• At the end of the subroutine, a return instruction is executed

• Program resumes execution at the instruction immediately following the subroutine call

• The way in which a computer makes it possible to call and return from subroutines is referred to as its **Subroutine Linkage** method.

• The simplest subroutine linkage method is to save the return-address in a specific location, which may be a register dedicated to this function. Such a register is called the **Link Register**.

• When the subroutine completes its task, the Return instruction returns to the calling-program by branching indirectly through the link-register.

• The **Call Instruction** is a special branch instruction that performs the following operations:

→ Store the contents of PC into link-register.

→ Branch to the target-address specified by the instruction.

• The **Return Instruction** is a special branch instruction that performs the operation:

→ Branch to the address contained in the link-register.

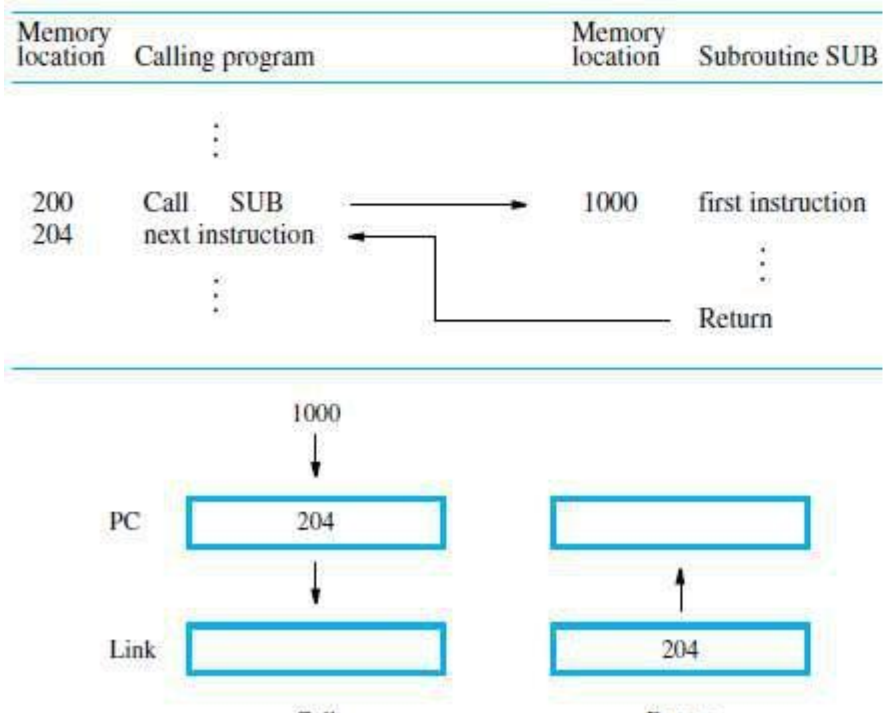


Figure 2.16 Subroutine linkage using a link register.

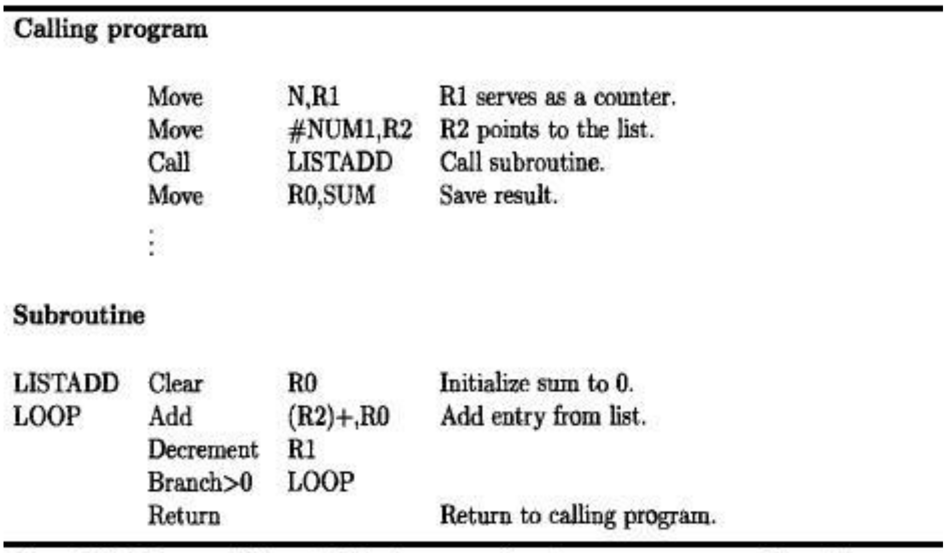
SUBROUTINE NESTING AND THE PROCESSOR STACK

- **Subroutine Nesting** means one subroutine calls another subroutine.
- In this case, the return-address of the second call is also stored in the link-register, destroying its previous contents.
- Hence, it is essential to save the contents of the link-register in some other location before calling another subroutine. Otherwise, the return-address of the first subroutine will be lost.
- Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it.
- The return-address needed for this first return is the last one generated in the nested call sequence. That is, return-addresses are generated and used in a LIFO order.
- This suggests that the return-addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the SP(Stack Pointer) to be used in this operation.
- SP is used to point to the processor-stack.
- Call instruction pushes the contents of the PC onto the processor-stack.

  Return instruction pops the return-address from the processor-stack into the PC.

PARAMETER PASSING

- The exchange of information between a calling-program and a subroutine is referred to as **Parameter Passing** (Figure: 2.25).
- The parameters may be placed in registers or in memory-location, where they can be accessed by the subroutine.
- Alternatively, parameters may be placed on the processor-stack used for saving the return-address.
- Following is a program for adding a list of numbers using subroutine with the parameters passed through registers.



```
Calling program

          Move       N,R1        R1 serves as a counter.
          Move       #NUM1,R2    R2 points to the list.
          Call       LISTADD     Call subroutine.
          Move       R0,SUM      Save result.
          :

Subroutine

LISTADD   Clear      R0          Initialize sum to 0.
LOOP      Add        (R2)+,R0    Add entry from list.
          Decrement  R1
          Branch>0   LOOP
          Return                 Return to calling program.
```

Figure 2.25 Program of Figure 2.16 written as a subroutine; parameters passed through registers.