SYLLABUS

Machine Instruction and Programs:

Instruction and Instruction Sequencing: Register Transfer Notation, Assembly Language Notation, Basic Instruction Types, Addressing Modes, Basic Input/output Operations, The role of Stacks and Queues in computer programming equation. Component of Instructions: Logic Instructions, shift and Rotate Instructions

QUESTIONS

1. What are four types of operations performed by computer instructions?
2. What is register transfer notation? Write and explain these notations to three- address, two-address, single address and zero-address instruction types.
3. Discuss briefly about Assembly language notations.
4. Define and discuss about straight-line sequencing.
5. Define and discuss about instruction execution.
6. Differentiate the instruction execution for adding ‘n’ numbers using Straight line sequencing and branching.
7. In how many ways the location of an operand is specified in an instruction? Explain each mode with suitable examples.
8. Explain the following addressing modes i) Register mode ii) Immediate mode iii) Indirect mode iv) Absolute mode
9. Explain the following addressing modes.
   i) Index mode ii) Auto increment mode iii) Auto decrement mode.
10. With an example write about relative addressing.
11. Write short notes on additional addressing modes.
12. Illustrate the concept of assembly directives with an assembly language program
13. List basic input and output operations.
14. Discuss briefly about basic input/output operations.
15. Explain I/O operations of computer architecture.
16. Write about various means by which data are transferred between memory of a computer and outside world.
17. Explain the role of stack and queues in computer programming equation.
18. Discuss about Condition Register (CR) and Integer Exception Register (XER).
19. Write the subroutines for parameter passing through registers.
20. Give example for left and right shift operations.
21. Write a note on shift instruction
22. Write a short note on rotate instructions.
23. Write short notes on shift and rotate instructions.
24. Write a note on logic instructions.

UNIT-4

Instructions and Instruction Sequencing

Q1. What are four types of operations performed by computer instructions?

A computer must have instructions capable of performing four types of operations:

1. Data transfers between the memory and the processor registers
2. Arithmetic and logic operations on data
3. Program sequencing and control
4. I/O transfers

Q2. What is register transfer notation?

Information transfer from one register to another register is called **register transfer.**

Register Transfer Language (RTL) Notation is a symbolic notation used to represent register transfer using replacement operator (←).The possible locations that are involved in register transfer are:
1) Memory locations
2) Processor registers and
3) Registers in I/O device.

Location	**Symbolic names for Hardware Binary Address**	**Example**	**Description**
Memory locations	LOC, PLACE, NUM, A	R1 ← [LOC]	Contents of memory location LOC are transferred into register R1.
Processor registers	R0, R1 ,R2	R3 ← [R1]+[R2]	Add the contents of register R1 &R2 and places their sum into R3.
I/O Registers	DATAIN, DATAOUT, OUTSTATUS, INSTATUS	R1 ← DATAIN	Contents of I/O register DATAIN are transferred into register R1.

- Every location is identified by a symbolic name given for its hardware binary address.
- The contents of location are denoted by placing [] around name of the location.
- Registers are denoted by capital letters and are sometimes followed by numerals.

	Notation	**Description**
Register Transfer	R2 ← [R1]	RHS of RTL expression denotes a value, and LHS denotes the name of a location where the value is to be placed, overwriting the old contents of that location. control condition terminates with a colon.
Simultaneous Transfer	R2 ←R1, R1 ←R2	
Conditional Transfer (Control Function)	P: R2←R1 (or) if(P=1)then R2←R1	
Conditional, Simultaneous Transfer	T: R2 ←R1, R1 ←R2	

Q3. Write and explain RTL notations to three- address, two-address, single address and zero-address instruction types.

A program is a set of instructions, that specify the operations, operands and the sequence by which processing has to occur.

Format of an instruction:
An instruction consists of 2 fields.

Opcode	Memory Address

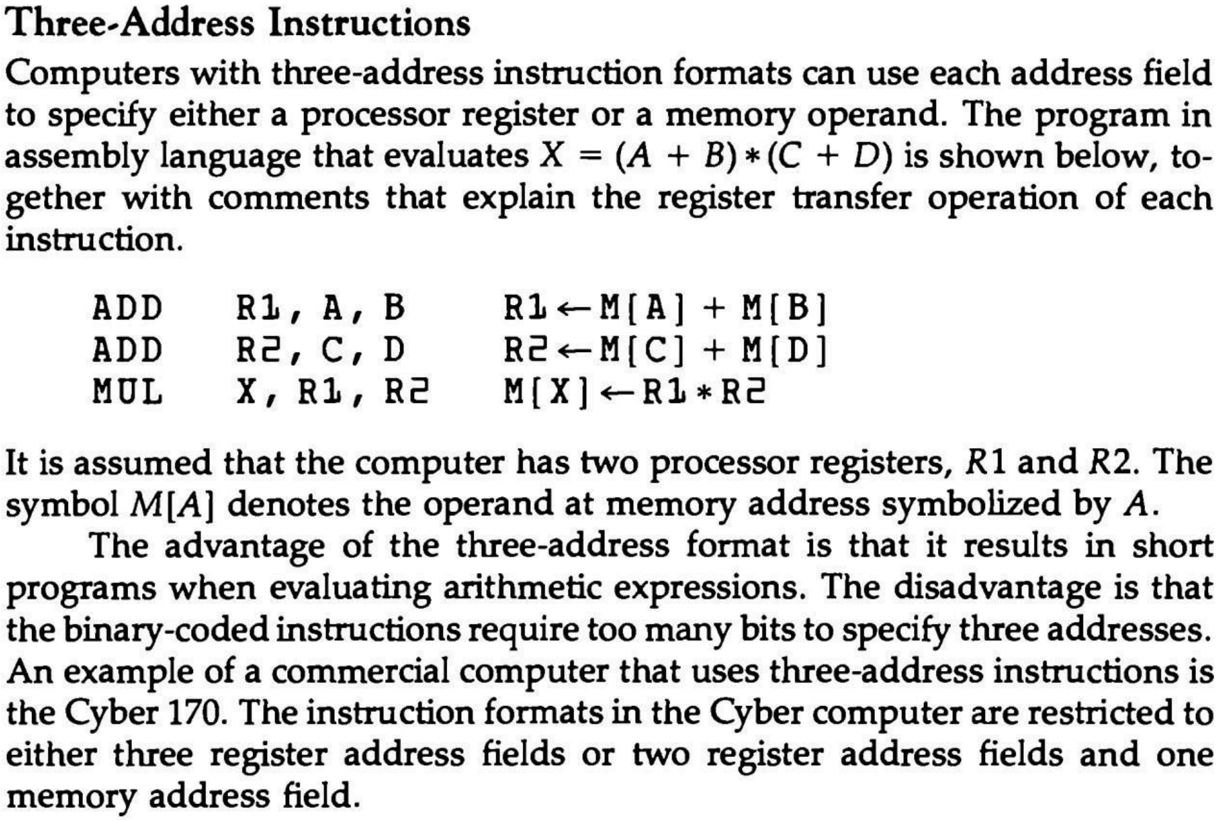
Opcode field specifies the operation to be performed.

Address field specifies the location of an operand i.e., register or memory location.

Basic Instruction Types:

There are four basic types of instructions based upon number of address fields it contain. They are

1. Three address instructions
2. Two address instructions
3. One address instructions
4. Zero address instructions



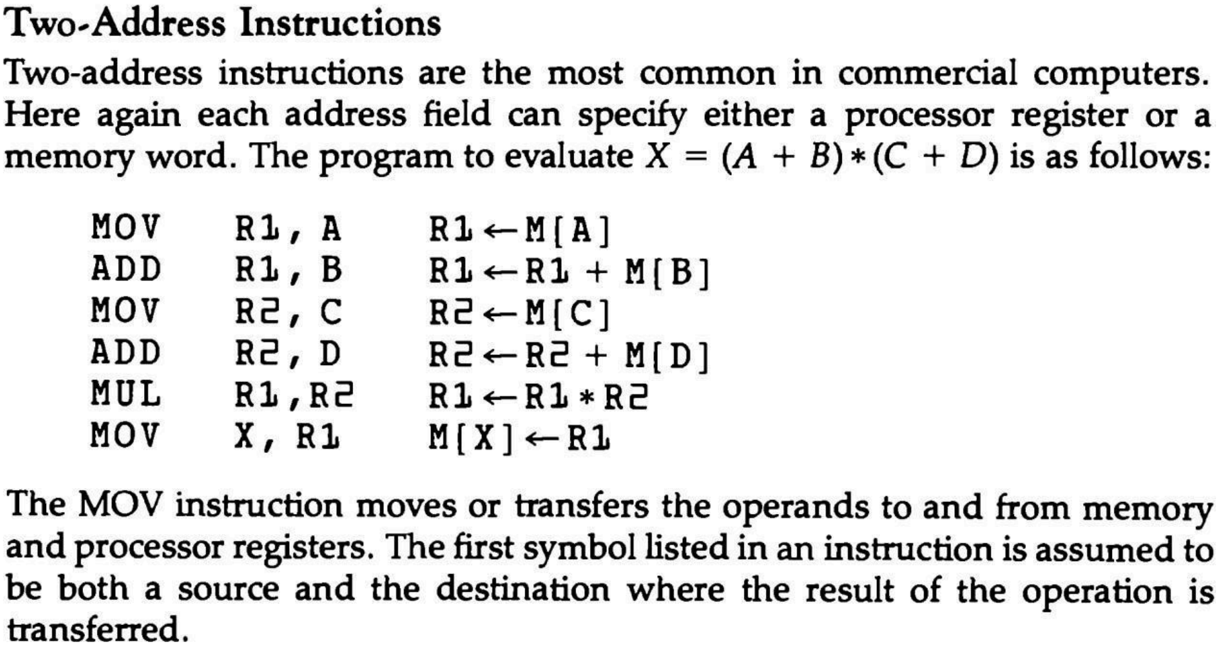
Three-Address Instructions

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates $X = (A + B) * (C + D)$ is shown below, together with comments that explain the register transfer operation of each instruction.

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

It is assumed that the computer has two processor registers, $R1$ and $R2$. The symbol $M[A]$ denotes the operand at memory address symbolized by A.

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses. An example of a commercial computer that uses three-address instructions is the Cyber 170. The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field.

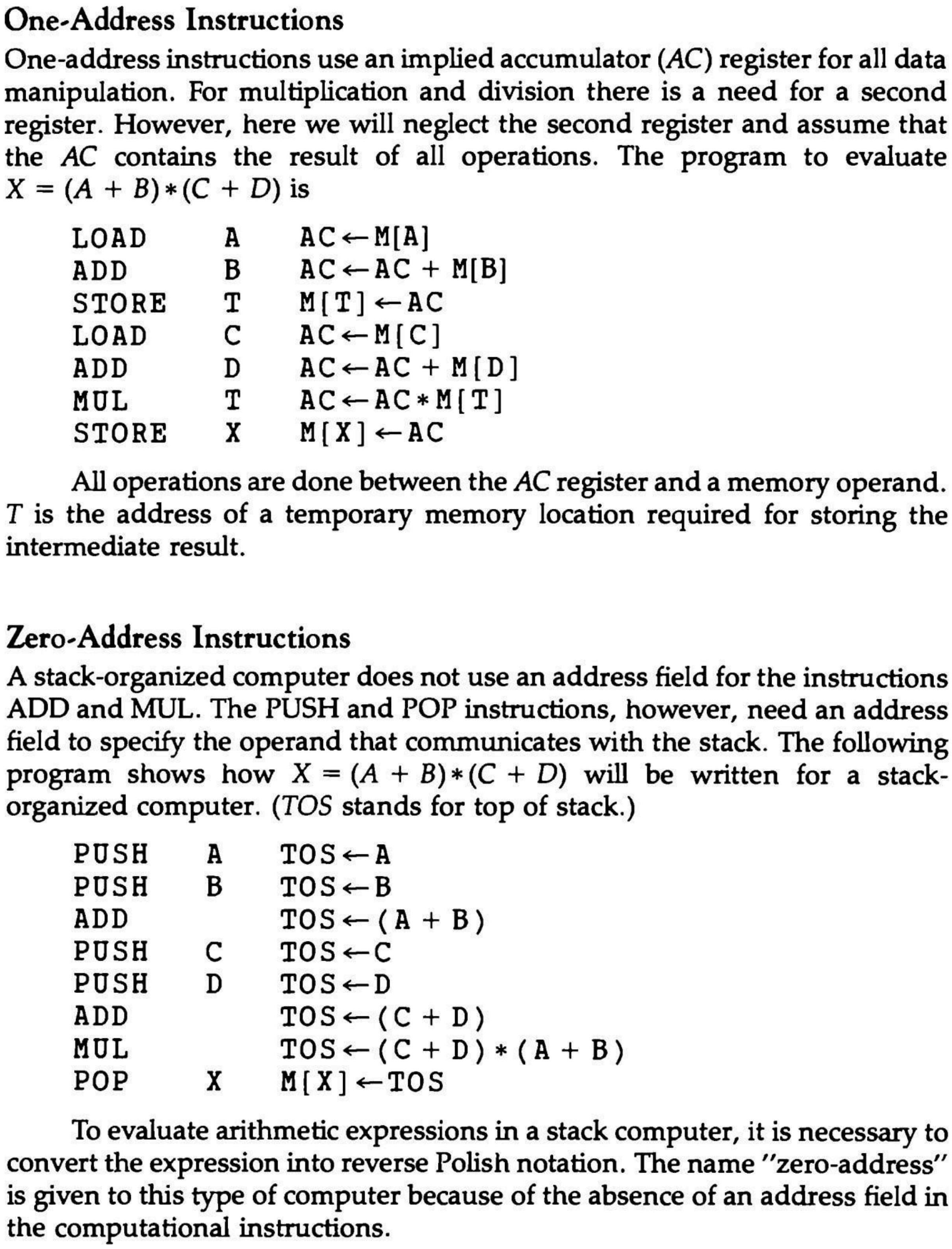


Two-Address Instructions

Two-address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate $X = (A + B) * (C + D)$ is as follows:

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

The MOV instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.



One-Address Instructions

One-address instructions use an implied accumulator (*AC*) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the *AC* contains the result of all operations. The program to evaluate $X = (A + B) * (C + D)$ is

```
LOAD     A     AC←M[A]
ADD      B     AC←AC + M[B]
STORE    T     M[T]←AC
LOAD     C     AC←M[C]
ADD      D     AC←AC + M[D]
MUL      T     AC←AC*M[T]
STORE    X     M[X]←AC
```

All operations are done between the *AC* register and a memory operand. *T* is the address of a temporary memory location required for storing the intermediate result.

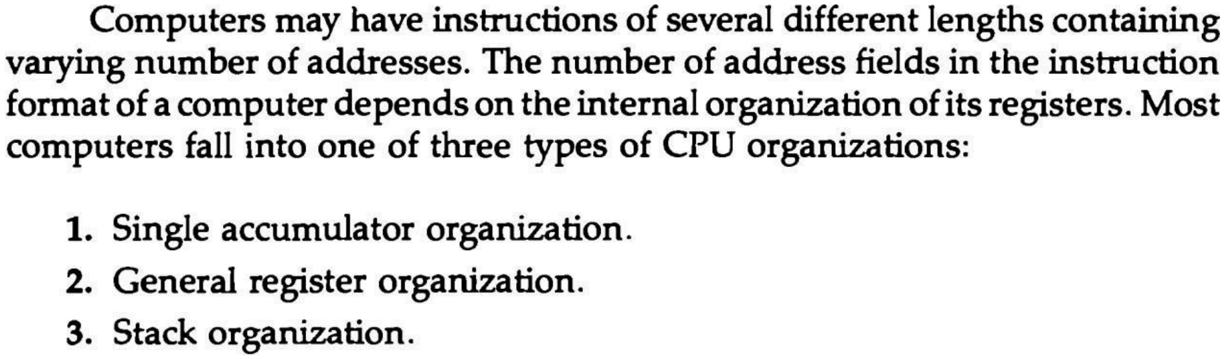
Zero-Address Instructions

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how $X = (A + B) * (C + D)$ will be written for a stack-organized computer. (*TOS* stands for top of stack.)

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

To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation. The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

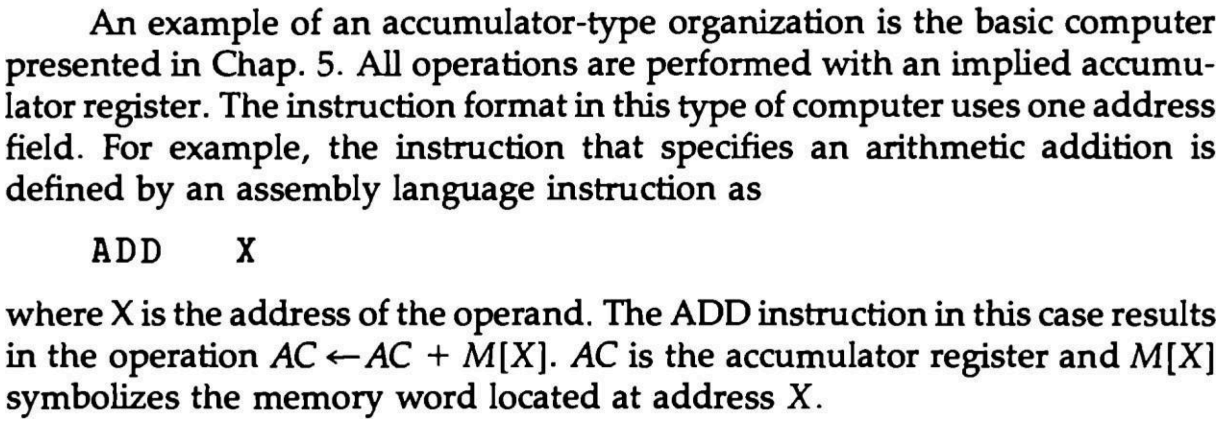
CPU Organization Types:



Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

1. ***Single Accumulator organization*** - In Single Accumulator organization, operation is done involving a special register called accumulator. (one –address instructions)

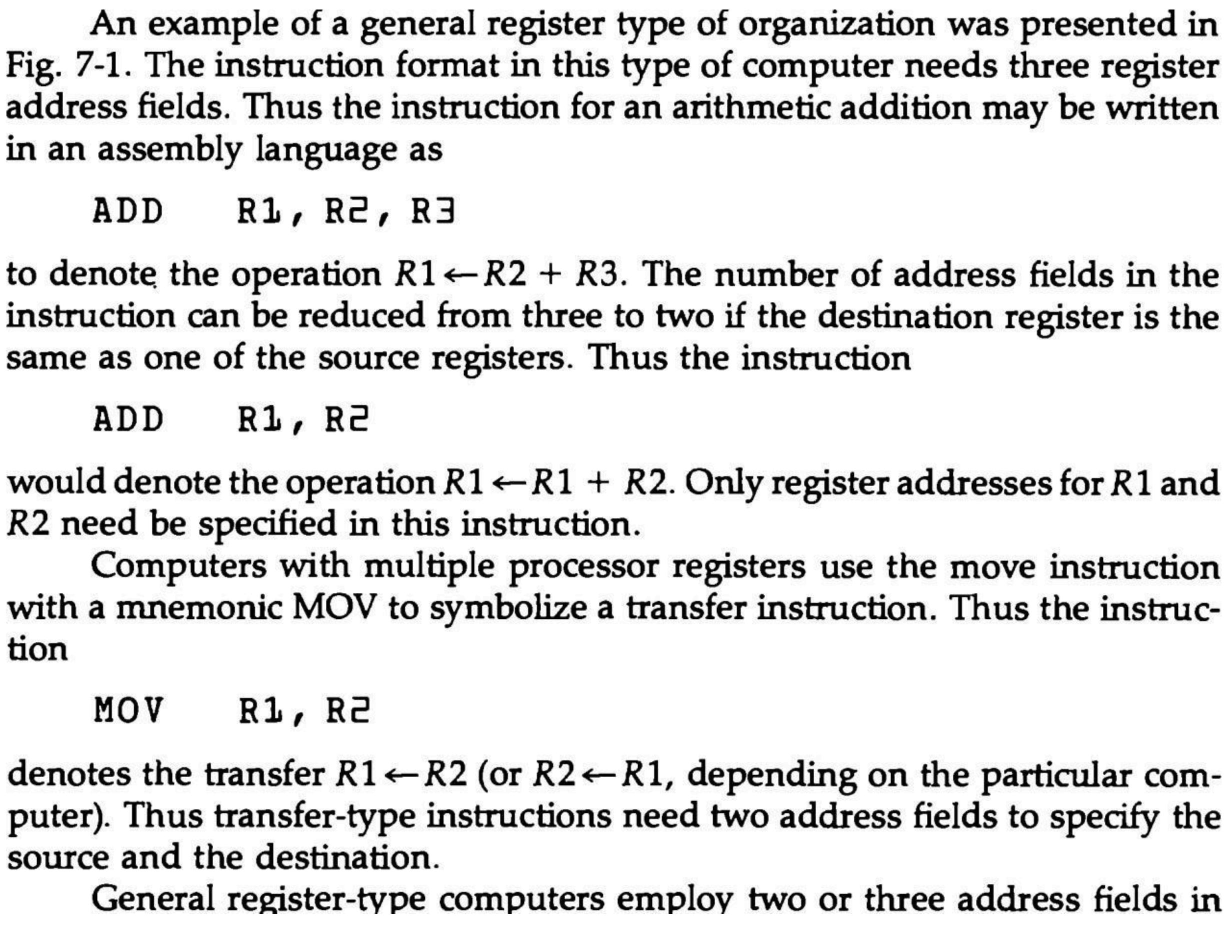


An example of an accumulator-type organization is the basic computer presented in Chap. 5. All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as

```
ADD    X
```

where X is the address of the operand. The ADD instruction in this case results in the operation $AC \leftarrow AC + M[X]$. AC is the accumulator register and $M[X]$ symbolizes the memory word located at address X.

2. ***General register organization*** - In General register organization, multiple registers are used for the computation purpose. . (two –address instructions, three –address instructions)



An example of a general register type of organization was presented in Fig. 7-1. The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as

```
ADD    R1, R2, R3
```

to denote the operation $R1 \leftarrow R2 + R3$. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

```
ADD    R1, R2
```

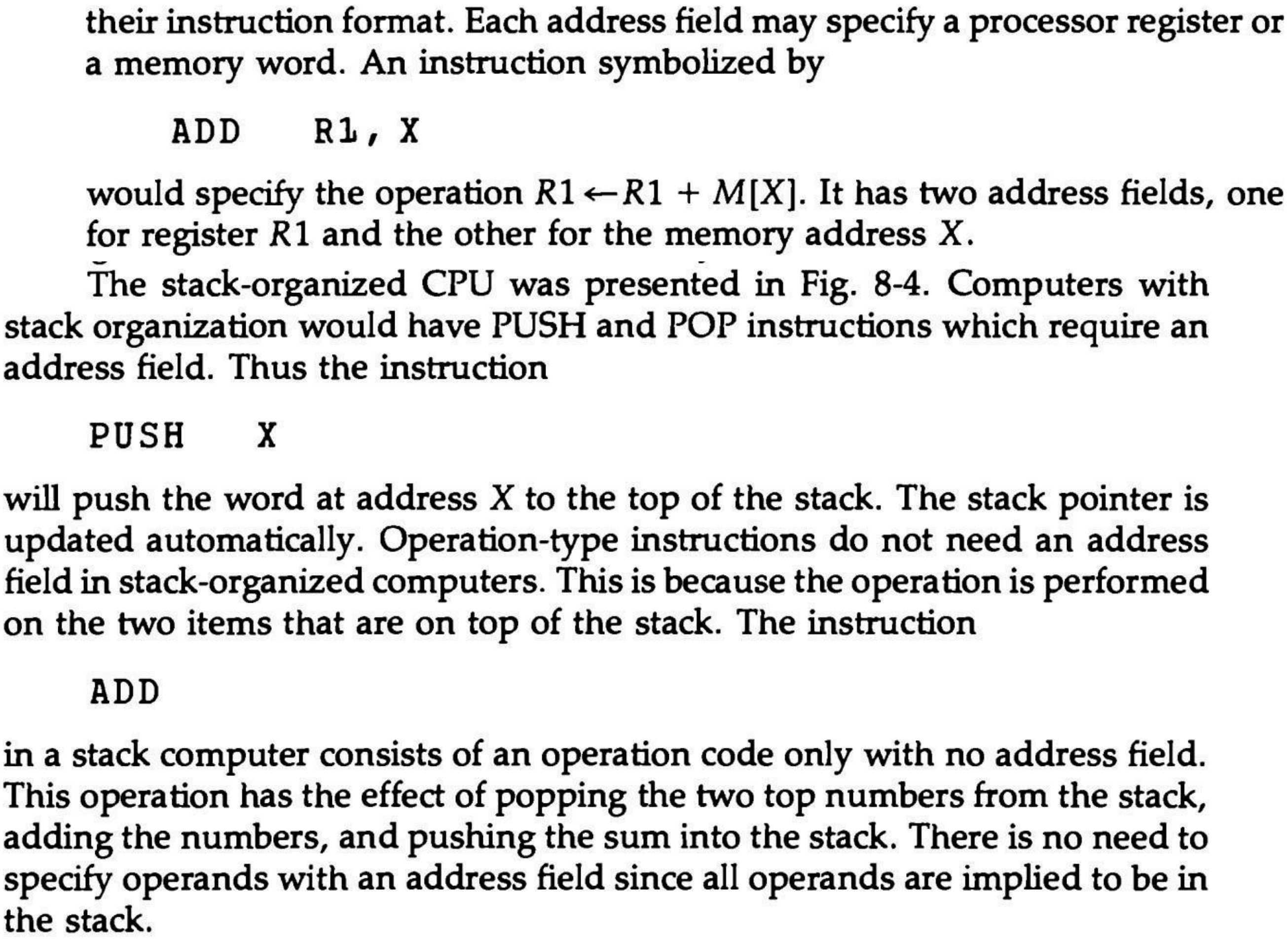
would denote the operation $R1 \leftarrow R1 + R2$. Only register addresses for $R1$ and $R2$ need be specified in this instruction.

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction

```
MOV    R1, R2
```

denotes the transfer $R1 \leftarrow R2$ (or $R2 \leftarrow R1$, depending on the particular computer). Thus transfer-type instructions need two address fields to specify the source and the destination.

General register-type computers employ two or three address fields in



their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by

```
ADD    R1, X
```

would specify the operation $R1 \leftarrow R1 + M[X]$. It has two address fields, one for register $R1$ and the other for the memory address X.

The stack-organized CPU was presented in Fig. 8-4. Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction

```
PUSH    X
```

will push the word at address X to the top of the stack. The stack pointer is updated automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction

```
ADD
```

in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack.

4.

In single accumulator organization, generally Load, Store and Add instructions are specified as

Load A,R_i
Add B,R_i
Store R_i,C

Where R_i performs the function of Accumulator.

When only one memory address is directly specified, the instruction may not fit into one word. But, if only registers are specified, normally the instruction will fit into one word.

When data are transferred among registers, Load and Store instructions are replaced with Move instruction. Thus

Load A,R_i can be replaced with Move A,R_i
Store R_i,A can be replaced with Move R_i, A

Example:

Case 1: The program in assembly language that evaluates C=A+B using only *operands in registers* is

Move A,R_i	$R_i \leftarrow [A]$
Move B,R_j	$R_j \leftarrow [B]$
ADD R_i, R_j	$R_j \leftarrow [R_i] + [R_j]$
Move R_i, C	$C \leftarrow [R_j]$

Case 2: The program in assembly language that evaluates C=A+B where *one operand must be in register* and *other operand may be in memory* is

Move A,R_i	$R_i \leftarrow [A]$
ADD B, R_i	$R_j \leftarrow [B] + [R_i]$
Move R_i, C	$C \leftarrow [R_i]$

Access to registers is much faster than to data stored in memory locations. Thus, A substantial increase in speed is achieved if operations are performed on data in processor registers without the need to copy data to/ from memory.

Q4. Give a short sequence of machine instructions for the task "Add the contents of memory-location A to those of location B, and place the answer in location C".
Instructions: Load LOC, Ri and
Store Ri, LOC
are the only instructions available to transfer data between the memory and the general purpose register Ri. Do not change the contents of either location A or B.

```
Load A,R0
Load B,R1
Add
R0,R1
Store R1,C
```

(b) Suppose that Move and Add instructions are available with the formats Move Location1, Location2 and
Add Location1, Location2

These instructions move or add a copy of the operand at the second location to the first location, overwriting the original operand at the first location. Either or both of the operands can be in the memory or the general-purpose registers. Is it possible to use fewer instructions of these types to accomplish the task in part (a)? If yes, give the sequence.

```
Move B,C
Add A,C
```

Q5. Write a program that can evaluate the expression A*B+C*D in a single-accumulator processor. Assume that the processor has Load, Store, Multiply, and Add instructions and that all values fit in the accumulator

The program in assembly language that evaluates **A*B+C*D**

```
Load A
Multiply B
Store RESULT
Load C
Multiply D
Add RESULT
Store RESULT
```

Q6. Define and discuss about straight-line sequencing.

- To execute a program, initially, the address of the first instruction is loaded into PC (program counter).
- 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.**
- During the execution of each instruction, PC is incremented to point to next instruction.

- **Example:** The possible program segment for performing C=A+B as it appears in the memory of a computer is shown below.

- The program in assembly language that evaluates C=A+B where *one operand must be in register* and *other operand may be in memory* is

```
Move   A,R0          R0 ← [A]
ADD   B, R0          R0 ← [B] + [R0]
Move   R0, C         C ← [R0]
```

Assume that the word length is 32 bits and the memory is byte-addressable. The three instructions of the program are in successive word locations, starting at location i. Since each instruction is 4 bytes long, the second, and third instructions are at addresses i+4, i+8. After executing the instruction at address i+8, PC is incremented by 4 and contains the value i+12 to point to first instruction of next program segment.

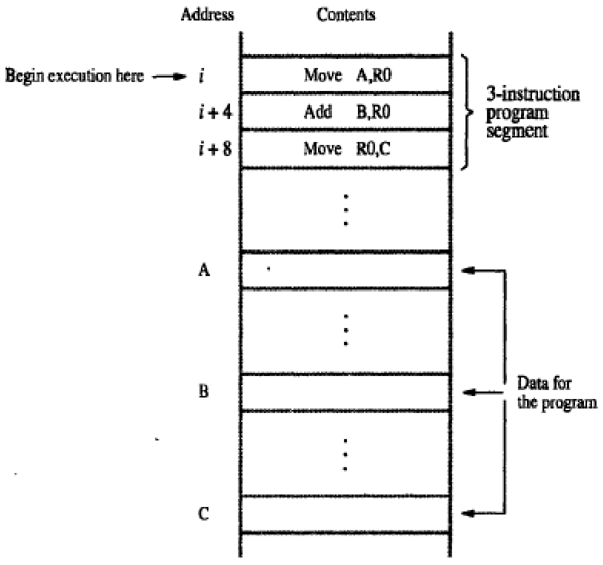


Figure 2.8 A program for C ← [A] + [B].

Q7. Define and discuss about Instruction Execution.

There are 2 phases for Instruction Execution:

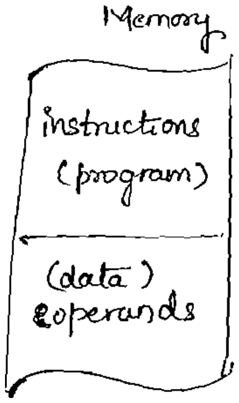
1. ***Fetch Phase:*** The instruction is fetched from the memory location, whose address is in PC and placed in the IR (Instruction Register) in the processor.

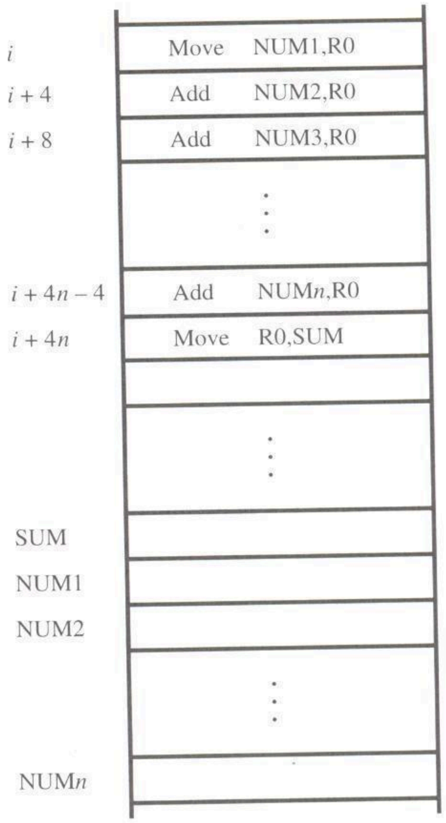
2. ***Execute Phase:***The contents of IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This involves fetching operands from memory or processor register and performing ALU operation and storing the result in destination location.

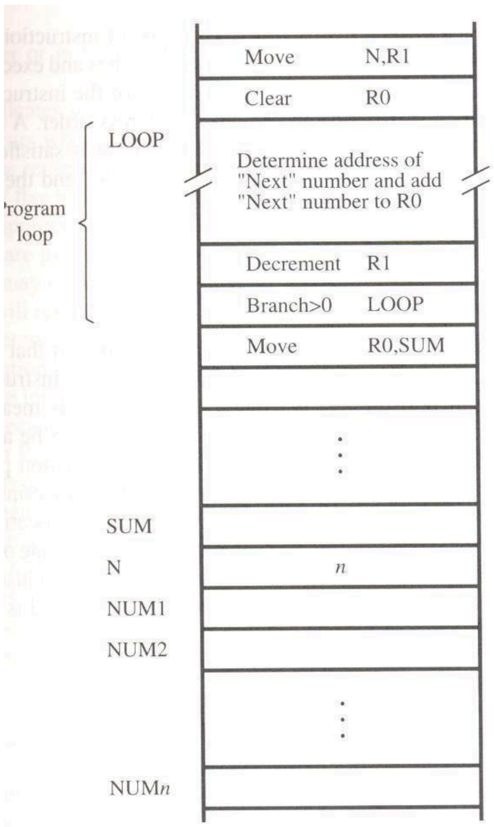
Once the execution phase is completed, PC contains the address of next instruction and a new fetch phase begins. Hence it is called as Fetch- Execute cycle.

Q8. Differentiate the instruction execution for adding ‘n’ numbers using Straight line sequencing and branching.

Memory contains both program and data in the stored program organization.







<u>Consider the program for adding a list of n numbers using Straight line sequencing.</u>

- Assume that the word length is 32 bits and the memory is byte-addressable. The instructions of the program are in successive word locations, starting at location i. Since each instruction is 4 bytes long, the sequence of instructions are at addresses i+4, i+8… ,i+4n.
- The Address of the memory locations containing the n numbers are symbolically given as NUM1, NUM2…..NUMn.
- Separate Add instruction is used to add each number to the contents of register R0.
- After all the numbers have been added, the result is placed in memory location SUM.

<u>Consider the program for adding a list of n numbers using Branching.</u>

Instead of using a long list of Add instructions as in straight line sequencing, it is possible to place single Add instruction in a program loop. The Loop is a straight line sequence of instructions executed as many times as needed.

- Assume that memory location N contains number of entries of list (let it be ‘n’).
- Register R1 is used as a counter to determine the number of times the loop is executed.
- At the beginning of the program, contents of location N are loaded into register R1.

  Move N, R1
- The loop starts at location LOOP and ends at the instruction Branch>0.
- During each pass,

  → address of the next number in list is determined and

  → that number is fetched and added to R0.
- The instruction **Decrement R1** reduces the contents of R1 by 1 each time through the loop.
- The Loop is repeated until R1 value >0.
- 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 Address***. (LOOP)
- A **Conditional Branch Instruction** (Branch >0 LOOP) 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 after BRANCH in sequential address order is fetched and executed.

  Move R0, SUM

Addressing modes

Q9. Explain about various addressing modes.

Addressing mode: The way in which the location of an operand can be specified in an instruction. It generates the effective address (the actual address that contains the operand).

Types of Addressing modes:

The number of addressing modes that a processor supports changes according to the instruction set it is based on, however there are a few generic ones that are present in almost all processors and are thus of utmost importance.They are:

1. Immediate mode
2. Absolute mode
3. Register mode
4. Indirect mode
5. Register Indirect mode
6. Index mode
7. Base with index
8. Base with index and offset
9. Relative mode
10. Auto increment mode
11. Auto decrement mode   Additional Addressing modes

Immediate mode:

In this mode, the operand is specified in the instruction itself. It is the actual data to be used. Data constants can be represented using this mode.

Syntax: opcode #VALUE

Effective address: no address, therefore operand=value

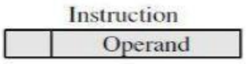


Figure 1: Immediate mode

E.g. LDAC #5

LDAC loads data from memory to accumulator.

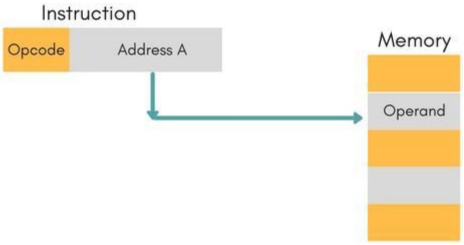
Therefore, AC=5.

Direct mode (or) Absolute mode:

In direct mode, the instruction includes memory address as an operand. i.e., the instruction gets data from that memory location. Global variables are represented using this addressing mode.

Syntax: opcode LOC   where LOC is memory address

Effective address: EA= Address of memory location= LOC



E.g : LDAC 5

EA = 5

∴ instruction reads data from memory location 5 .

0 : LDAC 5

5 : 10 → stores value in AC .

· AC = 10 ·

Register mode:

In register mode, the instruction includes register address as an operand. i.e., the instruction gets data from that register. Processor registers are often used for intermediate storage during arithmetic operations. This addressing mode is used at that time to access the registers.

Syntax: opcode Ri where Ri is Regiser Address

Effective address: EA= Address of register= Ri

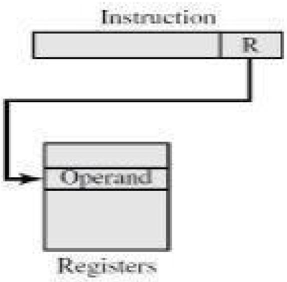


Figure 2: Register mode

E.g : LDAC R

instruction reads data from register R

LDAC R

R : 5 → store value in Ac

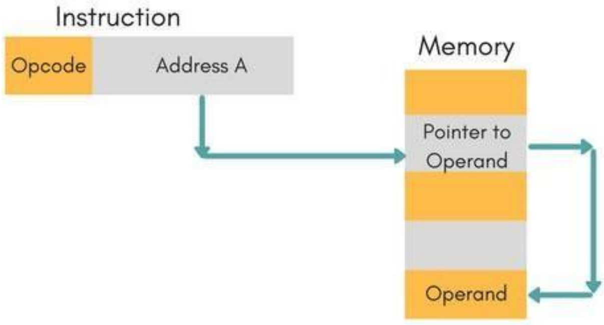
∴ Ac = 5

Indirect mode:

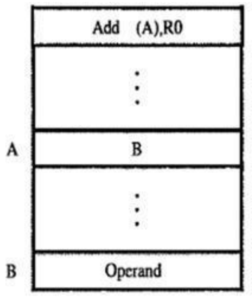
In Indirect mode, the instruction includes memory address as an operand. But, the address specified in the instruction is not the address of the operand. It is the address of a memory location that contains the address of the operand.

Syntax: opcode @LOC (or) opcode (LOC)

Effective address: EA= contents of LOC = [LOC]



Example: ADD(A),R0



(b) Through a memory location

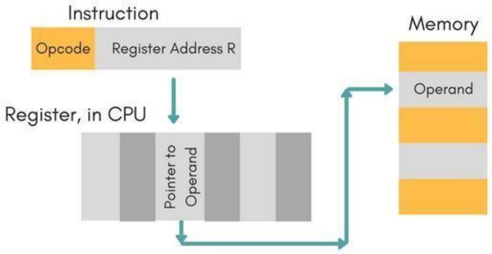
| Indirect addressing.

This instruction fetches the operand from the memory location B (the contents of the memory location A) and add them to R0.

Register Indirect mode: In Indirect mode, the instruction includes register address as an operand. But, the address specified in the instruction is not the address of the operand. It is the address of a register that contains the address of the operand.

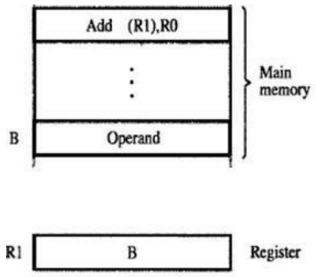
Syntax: opcode @Ri (or) opcode (Ri)

Effective address: EA= Contents of Ri = [Ri]



E.g. Add (R1), R0

This instruction fetches the operand from the memory location B (the contents of the register R1) and adds them to R0.



(a) Through a general-purpose register

Indirect addressing.

The register (or memory-location) that contains the address of an operand is called a Pointer.

Index mode (or) Displacement mode:

In Index mode, the instruction includes two values in address field, an address A and index register as an operand. But, the address specified in the instruction is not the actual address of the operand. The effective address of the operand is calculated by adding a constant value to the contents of a register. This mode provides more flexibility in dealing with lists and arrays.

There are two ways to use an index mode.

1. An address A holds base value (or constant value or offset or displacement) and index register Ri that holds the memory address.
2. An address A holds the memory address and index register Ri that holds constant value.

Index mode can be defined using general purpose register or special purpose register.

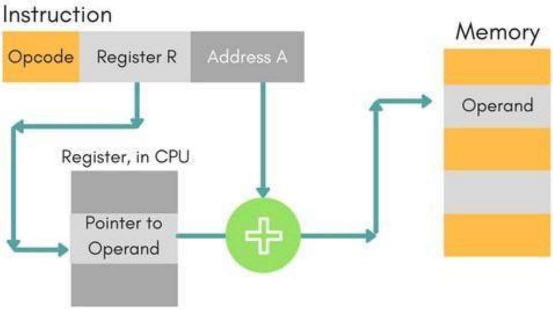
Variants of index mode:

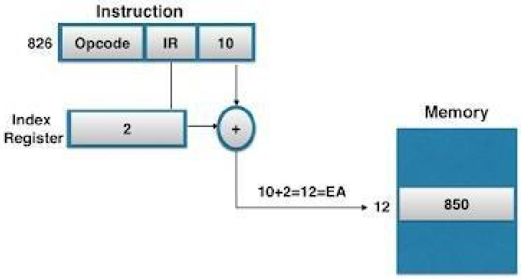
1. Index mode
2. Base with index   Index mode using general purpose register
3. Base with index and offset
4. Relative mode   →   Index mode using special purpose register.

Syntax: opcode X(Ri)

Effective address: EA= X + [Ri]

= X + contents of index register Ri

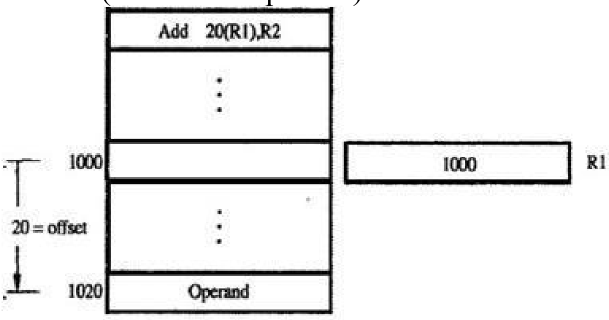




E.g. ADD 20(R1),R2

Here X=20 (offset) and R1=1000 (memory address)

EA=20+1000=1020 (address of operand)



(a) Offset is given as a constant

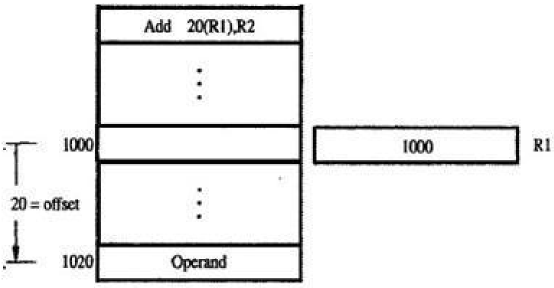
Figure 2.13 Indexed addressing.

First go to Reg R1 (using address)-read the content from R1-1000
Add the content 1000 with offset 20 get the result.
1000+20=1020

(a) ADD 1000(R1),R2

Here X=1000 (memory address) and R1=20 (offset)

EA=1000 +20=1020 (address of operand)



(a) Offset is given as a constant

Figure 2.13 Indexed addressing.

Base with index mode: The effective address is the sum of contents of two registers. The first register is called the index and the second register is called the base register. This mode provides more flexibility in accessing operands since both the components of effective address are registers and can be changed.

Syntax: opcode (Ri, Rj)

Effective address: EA= [Ri] + [Rj]

E.g. Move (R0, R1), R2

Here, Contents at address R0+R1 are moved to R2.

Base with index and offset mode: The effective address is the sum of contents of two registers and a constant. The constant value in this case is often called the offset or the displacement. This mode provides more flexibility 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.

Syntax: opcode X(Ri, Rj)

Effective address: EA= X + [Ri] + [Rj]

E.g. Move X (R0, R1), R2

Contents at address X+R0+R1 are moved to R2.

Relative mode:

In relative mode, the implicitly referenced register is the program counter (PC).

i.e., The effective address is the sum of offset value and contents of PC (the next instruction address). Typically, the address field is treated as a twos complement number for this operation. Thus, the effective address is a displacement relative to program counter.

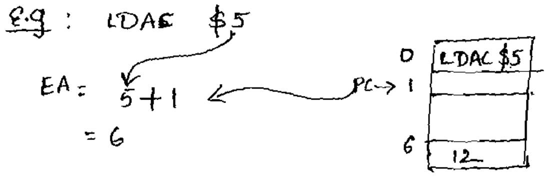
Relative addressing exploits the concept of locality. This mode provides more flexibility in accessing data operands and use to specify the target address in branch instructions.

Eg. Branch>0 Loop

It causes the program execution to go to the branch target location identified by the name loop if the branch condition is satisfied.

Syntax: opcode X(PC) (or) opcode $X

Effective address: EA= X + [PC]



∴ AC=12 .

Auto increment mode: The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to the next value. This increment is 1 for byte sized operands, 2 for 16 bit operands, 4 for 32 bit operands and so on.

Syntax: opcode (Ri)+

Effective address: EA= [Ri], Ri = Ri + 1

E.g : LDAC (R)

instruction reads address from register R .

R : 5

instruction reads data from memory location .

5 : 10 → store in AC

AC = 10

R : 5+1 = 6 .

Auto decrement mode: The effective address of the operand is the contents of a register specified in the instruction. Before accessing the operand, the contents of this register are automatically decremented and then the value is accessed.

Syntax: opcode -(Ri)

Effective address: Ri = Ri – 1, EA= [Ri]

E.g : LDAC (R)

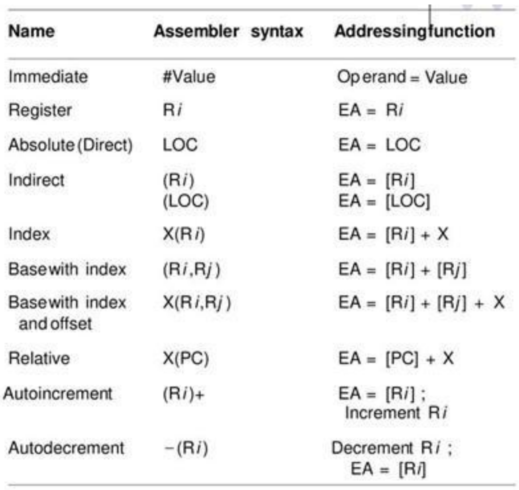
R : 6 instruction reads address from register R

R : 6-1=5 decrement value in register R .

5:10 instruction reads data from memory location .

∴ AC = 10 .

The auto increment addressing mode and the auto decrement addressing mode are widely used for the implementation of data structures like Stack. There may be other addressing modes that are unique to some processors. However the addressing modes mentioned above are common to many of the popular processors out there.



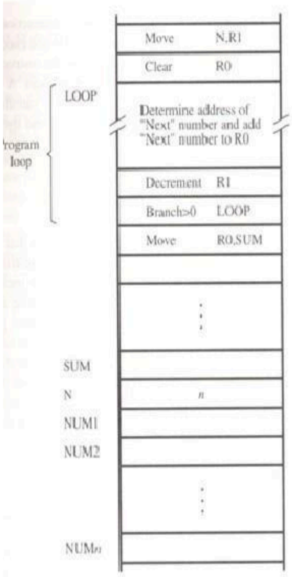
Name	Assembler syntax	Addressingfunction
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

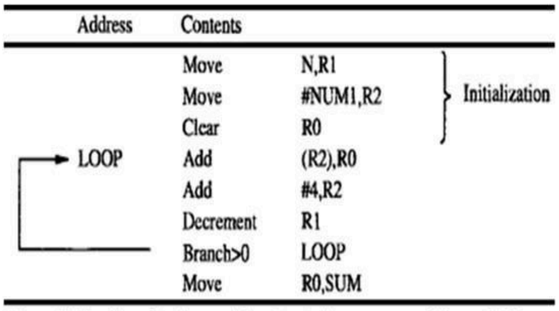
Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		EA = [LOC]

Index	X(R*i*)	EA = [R*i*] + X
Basewith index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Basewith 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*]

Q10. Write Assembly language program for the following statements.

1. Adding 'n' numbers using register indirect addressing mode





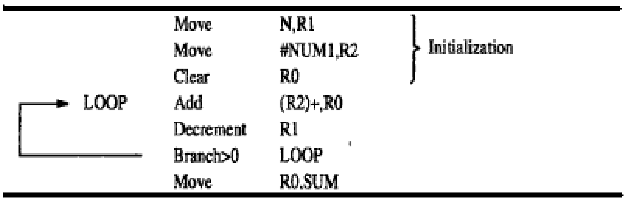
```
Address   Contents
          Move        N,R1      }                 ; N = Numbers to add
          Move        #NUM1,R2  } Initialization  ; R2= Address of 1st no.
          Clear       R0        }                 ; R0 = 00
LOOP      Add         (R2),R0                     ; R0 = [NUM1] + [R0]
          Add         #4,R2                       ; R2= To point to the next number
          Decrement   R1                          ; R1 = [R1] -1
          Branch>0    LOOP                        ; Check if R1>0 or not if yes go to Loop
          Move        R0,SUM                      ; SUM= Sum of all no.
```

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

Example: Adding 5 numbers

R0	R2	(R2)	R1	
00	1000H	10	5	>0 LOOP
10+00=10	1004H	20	4	>0 LOOP
20+10=30	1008H	30	3	>0 LOOP
30+30=60	1012H	40	2	>0 LOOP
40+60=100	1016H	50	1	>0 LOOP
50+100=150	1020H		0	SUM=150

2. Adding ‘n’ numbers using auto increment addressing mode



```
              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.

3. A=B+6

M
o
v
e
B
,
R
1
A
d
d
#
6
,
R
1
Move R1,A

4. A=*B

Move (B),A (or) Move B,R1
Move (R1),A

Q11. Consider a task involving a list of test scores for students taking a given course. Assume that the list of scores, beginning at location LIST, is structured as shown in Figure 2.14. A four-word memory block comprises a record that stores the relevant information for each student. Each record consists of the student’s identification number (ID), followed by the scores the student earned on three tests. There are n students in the class, and the value n is stored in location N immediately in front of the list. The addresses given in the figure for the student IDs and test scores assume that the memory is byte addressable and that the word length is 32 bits. Note that the list in Figure 2.14 represents a two-

dimensional array having n rows and four columns. Each row contains the entries for one student, and the columns give the IDs and test scores. Suppose that we wish to compute the sum of all scores obtained on each of the tests and store these three sums in memory locations SUM1, SUM2, and SUM3. Write an assembly language program to perform this task using index addressing mode.

Memory word location N contains the number of students, n. The list of student marks begins at memory word location LIST in the format shown in Figure 2.14. The parameter Stride = 16 is the distance in bytes between scores on a particular test for adjacent students in the list.

The Index addressing mode 4(R0), 8(R0) and 12(R0) are used to access the scores on a test1, test2 and test3 respect incremented by 4 in the loop to access scores on the test1. Register R0 points to the test score for student 1, and is incremented by 4 in the loop to access scores on the test1 by successive students in the list, incremented by 8 in the loop to access scores on the test2, incremented by 12 in the loop to access scores on the test3.

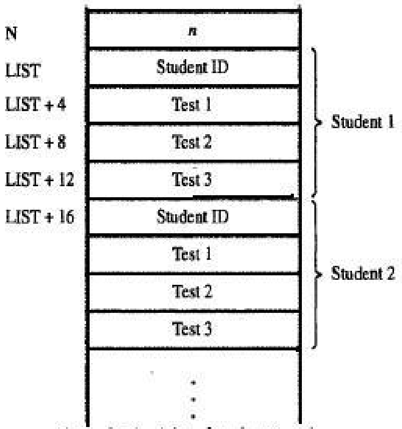
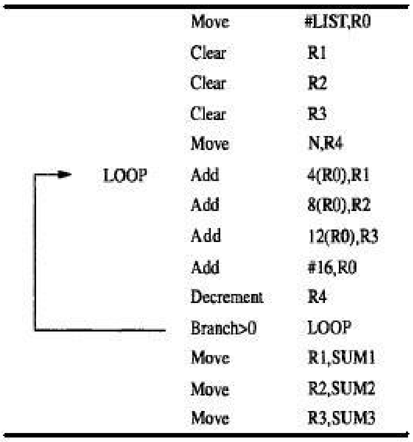


Figure 2.14 A list of students' marks.



```
                 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.

Q12. The list of student marks shown in Figure 2.14 is changed to contain j test scores for each student. Assume that there are n students. Write an assembly language program for computing the sums of the scores on each test and store these sums in the memory word locations at addresses SUM, SUM + 4, SUM + 8, ... The number of tests, j, is larger than the number of registers in the processor, so the type of program shown in Figure 2.15 for the 3-test case cannot be used. Use two nested loops. The inner loop should accumulate the sum for a particular test, and the outer loop should run over the number of tests, j. Assume that the memory area used to store the sums has been cleared to zero initially.
Memory word location J contains the number of tests, j, and memory word location N contains the number of students, n. The list of student marks begins at memory word location LIST in the format shown in Figure 2.14. The parameter Stride = 4(j + 1) is the distance in bytes between scores on a particular test for adjacent students in the list.

The Base with index addressing mode (R0,R2) is used to access the scores on a particular test. Register R0 points to the test score for student 1, and R2 is incremented by Stride in the inner loop to access scores on the same test by successive students in the list.

```
Move J,R4
Increment R4      Compute and place Stride (offset) = 4(j + 1) into register
R4 Multiply #4,R4

Move #LIST,R0       Initialize base register R0 to the ID of student 1.
Add #4,R0           location of the test 1 score for student 1 (LIST + 4)
Move #SUM,R1        Initialize register R1 to the location of the sum for test 1.
```

```
        Move             Initialize outer loop counter R10
OUT     J,R10            to j. Initialize inner loop counter
ER      Move             R11 to n. Clear index register R2
        N,R11            to zero.
        Clear R2         Clear sum register R3 to zero.
INNE    Clear R3         Accumulate the sum of test scores in
R       Add              R3.
        (R0,R2),R3       Increment index register R2 by Stride
        Add R4,R2        value.
        Decrement R11    check if all student scores on current test have
        been Branch>0 INNER   accumulated.

        Move             Store sum of current test
        R3,(R1)          scores increment sum
        Add #4,R1        location pointer.
        Add #4,R0        Increment base register to next test score for
                         student 1.
        Decrement R10    Check if the sums for all tests have been
        computed. Branch>0 OUTER
```

Q13. Registers R1 and R2 of a computer contain the decimal values 1200 and 4600. What is the effective address of the memory operand in each of the following instructions?

(i) Load 20(R1),R5
(ii) Move #3000,R5
(iii) Store R5,30(R1,R2)
(iv) Add –(R2),R5
(v) Subtract (R1)+,R5

(i)	Load 20(R1),R5	EA=20+[R1]= 20+1200=1220
(ii)	Move #3000,R5	EA: operand=value=3000
(iii)	Store R5,30(R1,R2)	EA=30+[R1]+[R2]= 30+1200+4600=5830
(iv)	Add –(R2),R5	EA=4600-1=4599
(v)	Subtract (R1)+,R5	EA=1220

Q14. Illustrate the concept of assembly directives with an assembly language program.

Assembly Language:

- Assembly Language uses symbolic-names called as ***mnemonics*** to write a program.
- 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 (binary pattern) by a program called an ***Assembler***.
- The user program in its original alphanumeric text form is called a ***Source Program***, and the assembled machine language program is called an ***Object Program***.

Example:

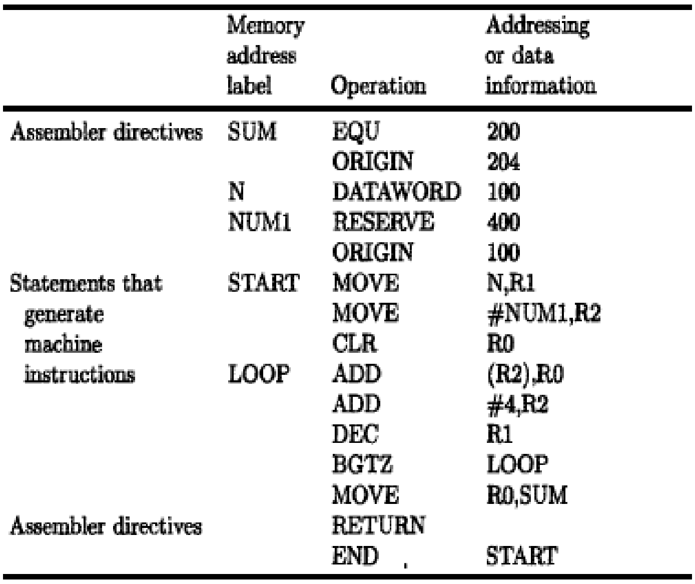
MOVE R0,SUM	The mnemonic MOVE represents OPcode for operation performed by instruction. It transfers contents of R0 to SUM
ADD #5,R3	# sign denotes immediate addressing mode. Instruction adds number 5 to contents of register R3 & puts the result back into registerR3.
ADDI 5,R3	suffix I in the mnemonic ADDI states that the given operand is in the Immediate addressing mode.

Assembler directives:

- Assembler Directives are the commands given to the assembler while it translates source program into object program.
- Assembler directives are the directions to the assembler which indicate how an operand or section of the program is to be processed. These are also called pseudo operations which are not executable by the microprocessor.
- The various directives are explained below.

EQU (Equate)	informs the assembler about the value of an identifier. It is used to give a name to some value or symbol**.** Every time the assembler finds the given name in the program, it will replace the name with the value or symbol, we have equated with that name
ORIGIN	tells the assembler about the starting-address of memory-area to place the data block or program instructions.
DATAWORD	tells the assembler to load a value into the location.
RESERVE	It is used to reserve a block of memory.
END	tells the assembler that this is the end of the source-program text.
RETURN	Identifies the point at which execution of the program should be terminated.

Example:



	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.

SUM EQU 200	Informs assembler that the name SUM should be replaced by the value 200.
ORIGIN 204	Instructs assembler to initiate data-block at memory-locations starting from 204.
N DATAWORD 100	Informs the assembler to load data 100 into the memory-location N(204).
NUM1 RESERVE 400	declares a memory-block of 400 bytes is to be reserved for data.

- Any statement that makes instructions or data being placed in a memory-location may be given a label. The label (SUM,N,NUM1,START,LOOP) is assigned a value equal to the address of that location.

Q15. Discuss briefly about Assembly language notations.

Assembly Language Notation is a symbolic notation used to represent machine instructions and programs.

General format of Assembly language statement:

Assembly language statements in a source program are written in the form:

Label	Operation	Operands	Comment

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

Example:

RTL Notation	Assembly Language Format	Description
R1 ← [LOC]	Load LOC, R1	Transfer data from memory location LOC to register R1. The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

R3 ← [R1]+[R2]	Add R1, R2, R3	Add the contents of registers R1 and R2, and places their sum into register R3.

LOC ← [R1]	Store R1,LOC	Transfer data from register R1 to memory location LOC. The contents of LOC are overwritten by the execution of this instruction.
	SUM EQU 200	Informs assembler that the name SUM should be replaced by the value 200.
	ADDI 5,R3	suffix I in the mnemonic ADDI states that the given operand is in the Immediate addressing mode.

Number Notation:

Most assemblers allow numerical values to be specified in different ways, using conventions that are defined by the assembly-language syntax.

Example: If the number 93, which is represented by the 8-bit binary number 01011101 is to be used as an immediate operand, it can be given as a decimal number, as in the instruction

ADD #93, R1 (or)

ADDI 93, R1

or as a binary number identified by an assembler-specific prefix symbol such as a percent sign, as in **ADDI %01011101,R1**

Binary numbers can be written more compactly as hexadecimal, or hex, numbers, in which four bits are represented by a single hex digit. The first ten patterns 0000, 0001, . . . , 1001, referred to as binary-coded decimal (BCD), are represented by the digits 0, 1, . . . , 9. The remaining six 4-bit patterns, 1010, 1011, . . . , 1111, are represented by the letters A, B, . . . , F. In hexadecimal representation, the decimal value 93 becomes 5D.

In assembly language, a hex representation is often identified by the prefix 0x (as in the C language) or by a dollar sign prefix. Thus, we would write

ADDI 0x5D, R1 (or)

ADDI $5D, R1

Q16. Discuss briefly about execution of Assembly language program.

Assembly and execution of programs:

The process involved in executing assembly language program is as follows:

Assembler:

An ***Assembler*** is used to translate the assembly language mnemonics into machine language(i.e binary codes) for the instructions and information about the addresses of the instructions.

- i.e., it 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.

The main task of assembler is ***determining the values that replace the names***.

- ***Case 1:*** the value of a name is specified by an ***EQU*** directive, this is a straightforward task.
- ***Case 2:*** A name is defined in the ***Label*** field of a given instruction, the value represented by the name is determined by the location of this instruction in the assembled object program. Hence, the assembler must keep track of addresses as it generates the machine code for successive instructions.

  For example, the names LOOP and SUM in the program of Figure 2.13 will be assigned the values 112 and 200, respectively.

- ***Case 3:*** the assembler does not directly replace a ***name representing an address*** with the actual value of this address.

For example, in a branch instruction, the name that specifies the location to which a branch is to be made (the branch target address) is not replaced by the Actual address.

A branch instruction is usually implemented in machine code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter.

Two Pass Assembler:

Two Pass Assembler has 2 passes:

First Pass:

- It creates a complete symbol table. At the end of this pass, all *names will have been assigned numerical values.*
- As the assembler scans through a source-program, it keeps track of all names and numerical- values that correspond to them in a symbol-table.
- When a name appears a second time in the source-program, it is replaced with its value from the table.
- When a name appears as an operand, before it is given a value, the assembler will not be able to determine the branch target because the name referred to has not yet been recorded in symbol table.

Second Pass:

- The assembler then scans through a source-program second time and substituting values from the table.

Loader:

- 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.
- A Loader Program is used to transfer object program from disk to memory.
- After loading object program, loader starts execution by branching to first instruction to be executed.
- The address of this instruction START has been included as an operand for END directive.

Debugger:

- A debugger is a program which is used to help the user find the programming errors.
- The debugger allows to look into the contents of registers and memory locations after the program runs. We can also change the contents of registers and memory locations and rerun the program.
- Some debuggers allows to stop the program after each instruction so that you can check or alter memory and register contents. This is called single step debug.
- A debugger also allows to set a breakpoint at any point in the program. If we insert a break point, the debugger will run the program up to the instruction where the breakpoint is put and then stop the execution.

Q17. Discuss briefly about basic input and output operations.

BASIC INPUT/OUTPUT OPERATIONS:

- Consider a task that reads characters typed on a keyboard, stores these data in the memory, and displays the same characters on a display screen.
- A simple way of implementing this task is to write a program that performs all functions needed to realize the desired action. This method is known as ***program-controlled I/O.***
- Difference in speed between processor (faster device) and I/O device (slower device) creates the need for mechanisms to synchronize the transfer of data.
- A solution: on output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character. Input is sent from the keyboard in a similar way.

- If same memory space is used for accessing both I/O data and memory, then it is called as ***memory mapped I/O.***
- Address values issued by processor used to access instructions and operands always refers to memory locations.
- In computers that use memory-mapped I/O, some address values are used to refer to peripheral device buffer-registers such as DATAIN & DATAOUT. The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a device interface.
- 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.
- To perform I/O transfer, machine instructions check the state of the status flags and transfers data between I/O devices and processor.

<u>Process that takes place when moving a character from the keyboard to the processor.</u>

- For this transfer, buffer-register DATAIN & a status control flags(SIN) are used.

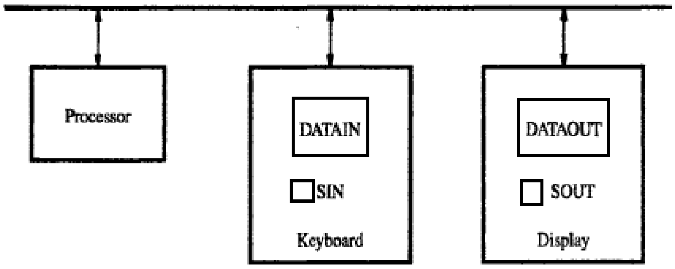


Figure 2.19 Bus connection for processor, keyboard, and display.

- 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, SIN informs the processor that a valid character is in DATAIN.

 SIN=0 → When the character is transferred to the processor.

- Machine instructions to transfer input to the processor.

```
READWAIT    Read the SIN flag
            Branch to READWAIT
   Transfer data from DATAIN to R1
```

- The Read operation described above may be implemented by the CISC-style instructions as follows:

```
READWAIT    Testbit #3, INSTATUS
            Branch=0 READWAIT
            MoveByte DATAIN,R1
```

- Testbit checks the status of one bit in INSTATUS flag in keyboard interface. If the bit tested is equal to zero, then Branch instruction that checks the state of the Z flag is true and then be used to cause a branch to the beginning of the wait loop.
- If the bit tested is equal to one, the data is read from input buffer.

<u>Process that takes place when characters are transferred from the processor to the display.</u>

- 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.
- Machine instructions to transfer output to the display.

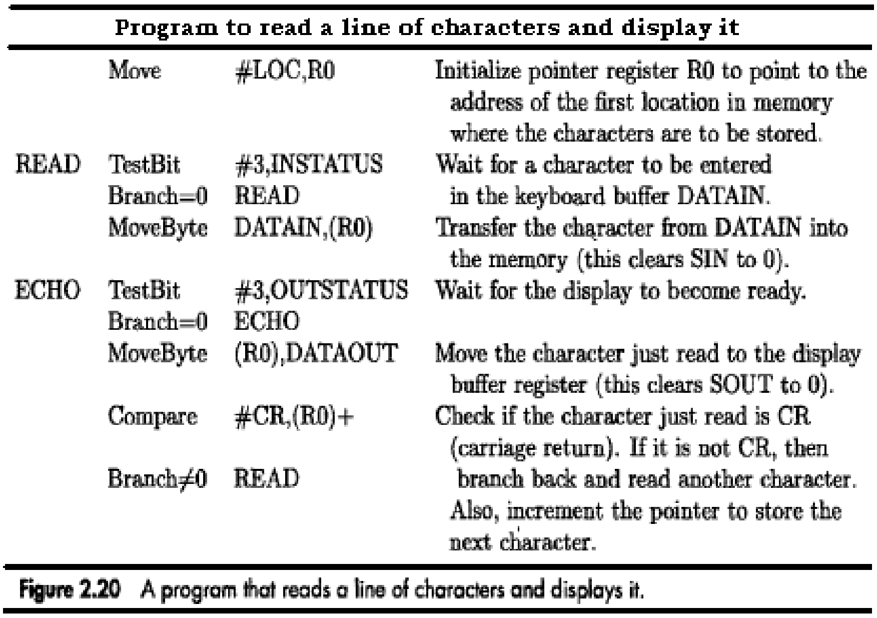
```
WRITEWAIT     Read the SOUT flag
              Branch to WRITEWAIT if SOUT = 0
         Transfer data from R1 to DATAOUT
```

- The Read operation described above may be implemented by the CISC-style instructions as follows:

```
WRITEWAIT     Testbit #3, OUTSTATUS
              Branch=0 WRITEWAIT
              MoveByte R1, DATAOUT
```

- Testbit checks the status of one bit in OUTSTATUS flag in display interface. If the bit tested is equal to zero, then Branch instruction that checks the state of the Z flag is true and then be used to cause a branch to the beginning of the wait loop.
- If the bit tested is equal to one, the data is written into output buffer.

CISC style Program to read a line of characters and display it



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.

Q18. Explain the role of stacks in computer programming equation.

- A stack is a special type of data structure where elements are inserted and deleted from the same end. This end is called the top of the stack (Figure: 2.14).
- Data are stored in and retrieved from a stack on a LIFO (Last In First Out) basis.
- One end of the stack is fixed while the other end rises and falls as data are pushed and popped.
- a stack would continuously move through the memory of a computer in the direction of decreasing addresses.
- 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).
- Assume a byte-addressable memory with a 32-bit word length,

	can be implemented as	Can be written as	Register transfer notation	

PUSH	Subtract #4, SP Move NEWITEM, (SP)	Move NEWITEM, -(SP)	SP ← SP – 4 [SP] ← NEWITEM	decrement the stack pointer by 4 so that it points to the new top element and then store(push) the top value from the stack into location NEWITEM.

POP	Move (SP),ITEM
	Add #4, SP
		Move (SP)+, ITEM	ITEM ← [SP]
			SP ← SP + 4
				load (pop) the top value from the stack into register location ITEM and then increment the stack pointer by 4 so that it points to the new top element.

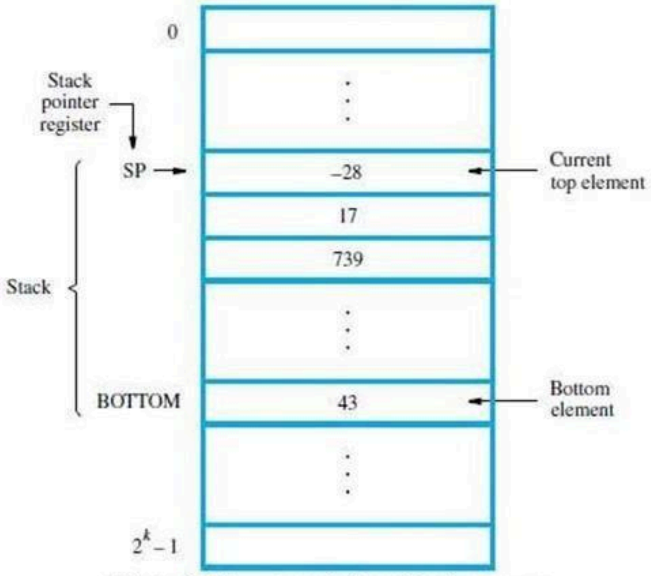
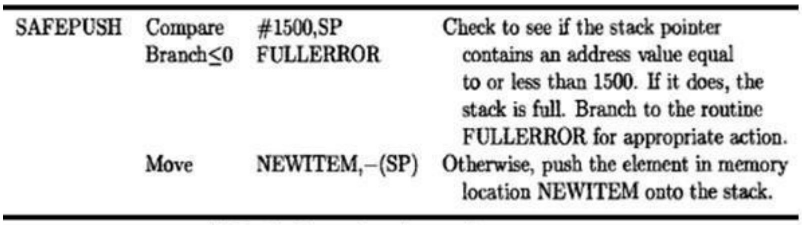


Figure 2.14 A stack of words in the memory.

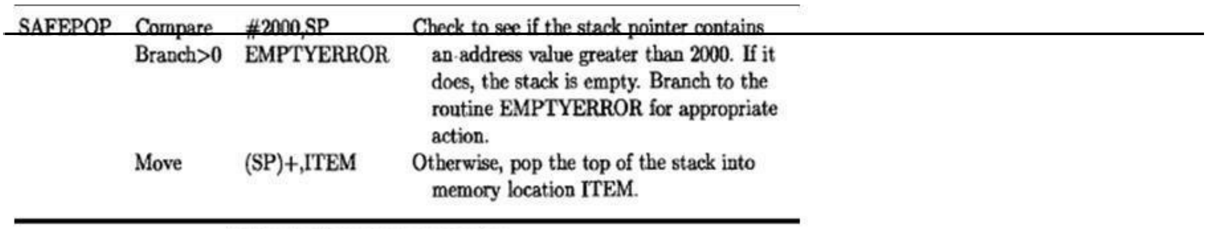
- Routine for a safe push operation as follows:



```
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

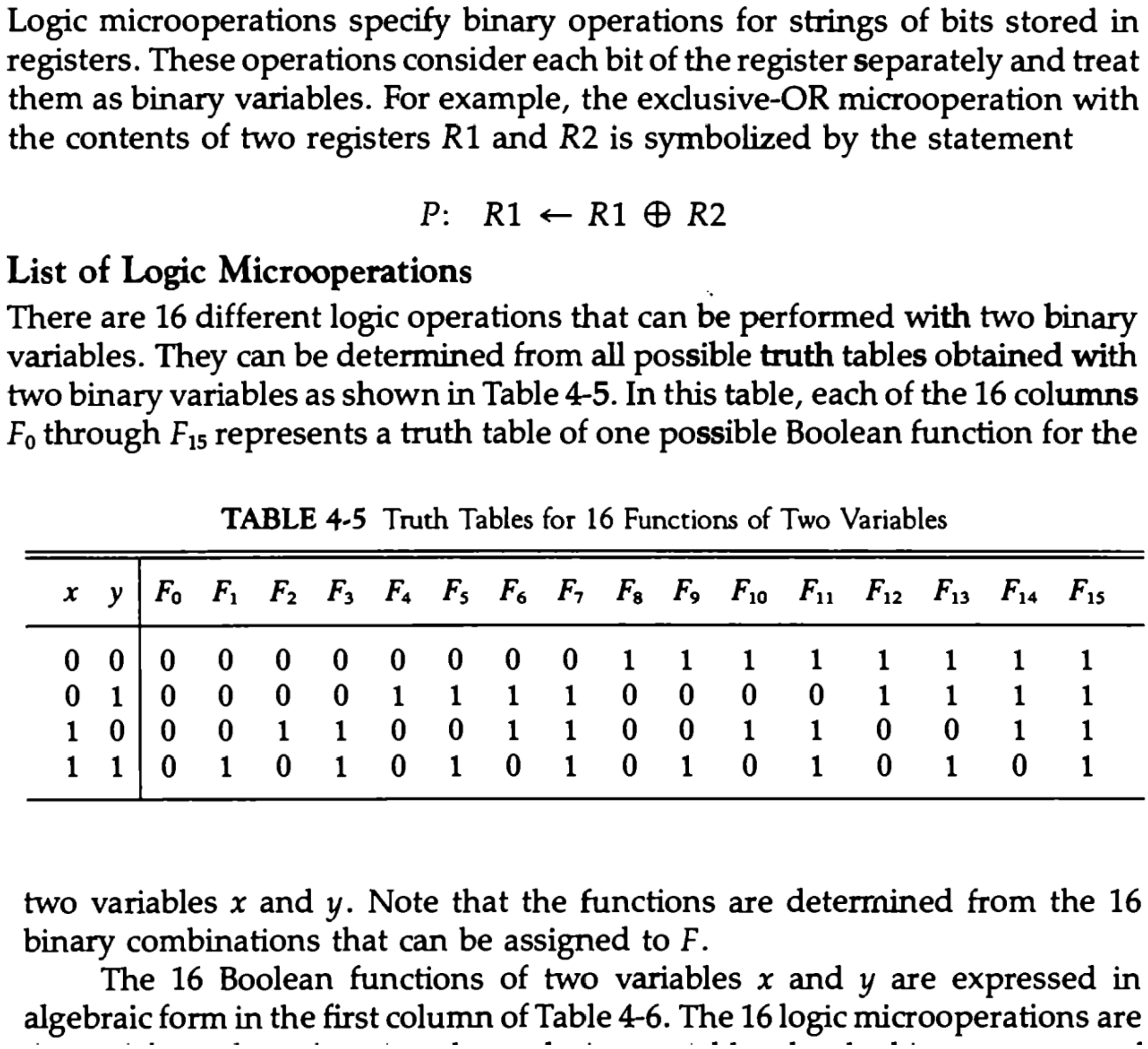
- Routine for a safe pop operation as follows:



```
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

LOGIC INSTRUCTIONS



Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers $R1$ and $R2$ is symbolized by the statement

$$P: \quad R1 \leftarrow R1 \oplus R2$$

List of Logic Microoperations

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table 4-5. In this table, each of the 16 columns F_0 through F_{15} represents a truth table of one possible Boolean function for the two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

TABLE 4-5 Truth Tables for 16 Functions of Two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 4-6. The 16 logic microoperations are

TABLE 4-6 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

The *selective-set* operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

1010	A before
$\underline{1100}$	B (logic operand)
1110	A after

The two leftmost bits of B are 1's, so the corresponding bits of A are set to 1. One of these two bits was already set and the other has been changed from 0 to l. The two bits of A with corresponding 0's in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and previous values of A. Therefore, the OR microoperation can be used to selectively set bits of a register.

The *selective-complement* operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

$$\begin{array}{ll} 1010 & A \text{ before} \\ \underline{1100} & B \text{ (logic operand)} \\ 0110 & A \text{ after} \end{array}$$

Again the two leftmost bits of B are 1's, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register.

The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1's in B. For example:

$$\begin{array}{ll} 1010 & A \text{ before} \\ \underline{1100} & B \text{ (logic operand)} \\ 0010 & A \text{ after} \end{array}$$

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB'. The corresponding logic microoperation is

$$A \leftarrow A \wedge \overline{B}$$

The *mask* operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B. The mask operation is an AND micro operation as seen from the following numerical example:

$$\begin{array}{ll} 1010 & A \text{ before} \\ \underline{1100} & B \text{ (logic operand)} \\ 1000 & A \text{ after masking} \end{array}$$

The two rightmost bits of A are cleared because the corresponding bits of B are 0's. The two leftmost bits are left unchanged because the corresponding bits of B are 1's. The mask operation is more convenient to use than the selective-clear operation because most computers provide an AND instruction, and few provide an instruction that executes the microoperation for selective-clear.

The *insert* operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an *A* register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

0110 1010	*A* before
0000 1111	*B* (mask)
0000 1010	*A* after masking

and then insert the new value:

0000 1010	*A* before
1001 0000	*B* (insert)
1001 1010	*A* after insertion

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

The *clear* operation compares the words in *A* and *B* and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as shown by the following example:

1010	A
1010	B
0000	$A \leftarrow A \oplus B$

When *A* and *B* are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

SHIFT AND ROTATE INSTRUCTIONS

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.

TABLE 4-7 Shift Microoperations

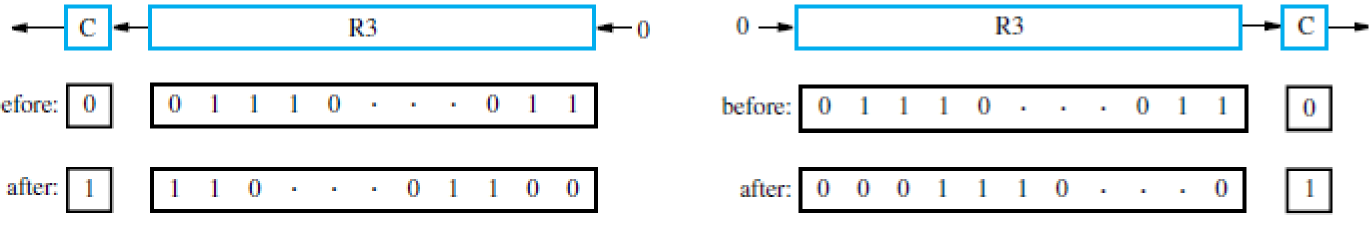
Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

A *logical* shift is one that transfers 0 through the serial input. We will adopt the symbols *shl* and *shr* for logical shift-left and shift-right microoperations. For example:

$$R1 \leftarrow \text{shl } R1$$

$$R2 \leftarrow \text{shr } R2$$

are two microoperations that specify a 1-bit shift to the left of the content of register $R1$ and a 1-bit shift to the right of the content of register $R2$. The register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.



(a) Logical shift left LShiftL R3, #2

(b) Logical shift right LShiftR R3, #2

An *arithmetic* shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Figure 4-11 shows a typical register of n bits. Bit R_{n-1} in the leftmost position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1}, and so on for the other bits in the register. The bit in R_0 is lost.

The arithmetic shift-left inserts a 0 into R_0, and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}. A sign

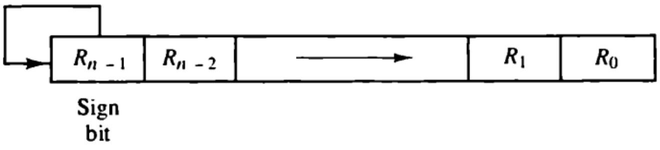
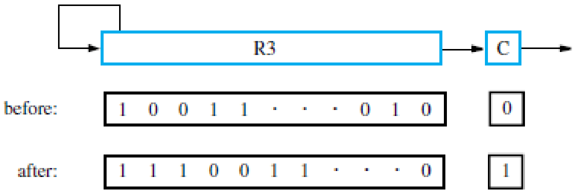


Figure 4-11 Arithmetic shift right.



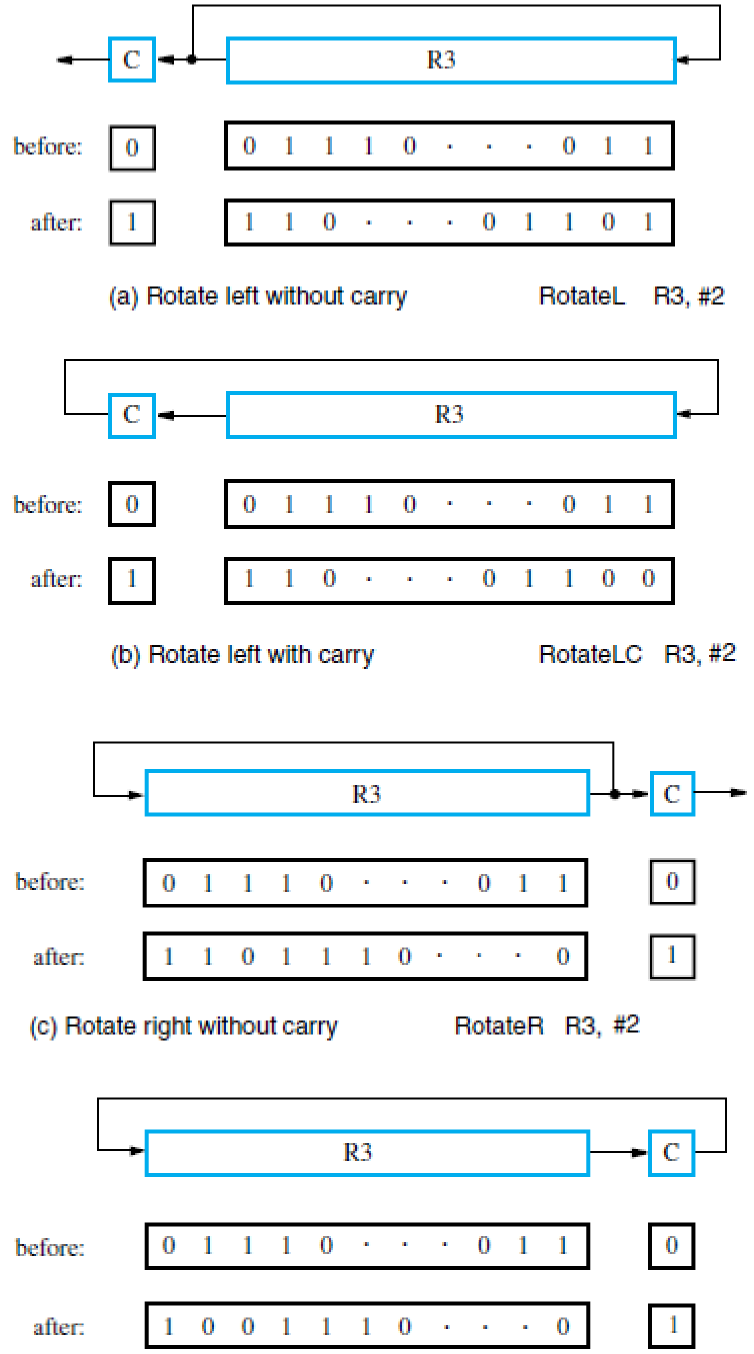
(c) Arithmetic shift right AShiftR R3, #2

Figure 2.23 Logical and arithmetic shift instructions.

ROTATE OPERATIONS

- In shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry-flag C.
- To preserve all bits, a set of rotate instructions can be used.

The *circular* shift (also known as a *rotate* operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols *cil* and *cir* for the circular shift left and right, respectively. The symbolic notation for the shift microoperations is shown in Table 4-7.



(a) Rotate left without carry RotateL R3, #2

(b) Rotate left with carry RotateLC R3, #2

(c) Rotate right without carry RotateR R3, #2

(d) Rotate right with carry RotateRC R3, #2

Figure 2.25 Rotate instructions.