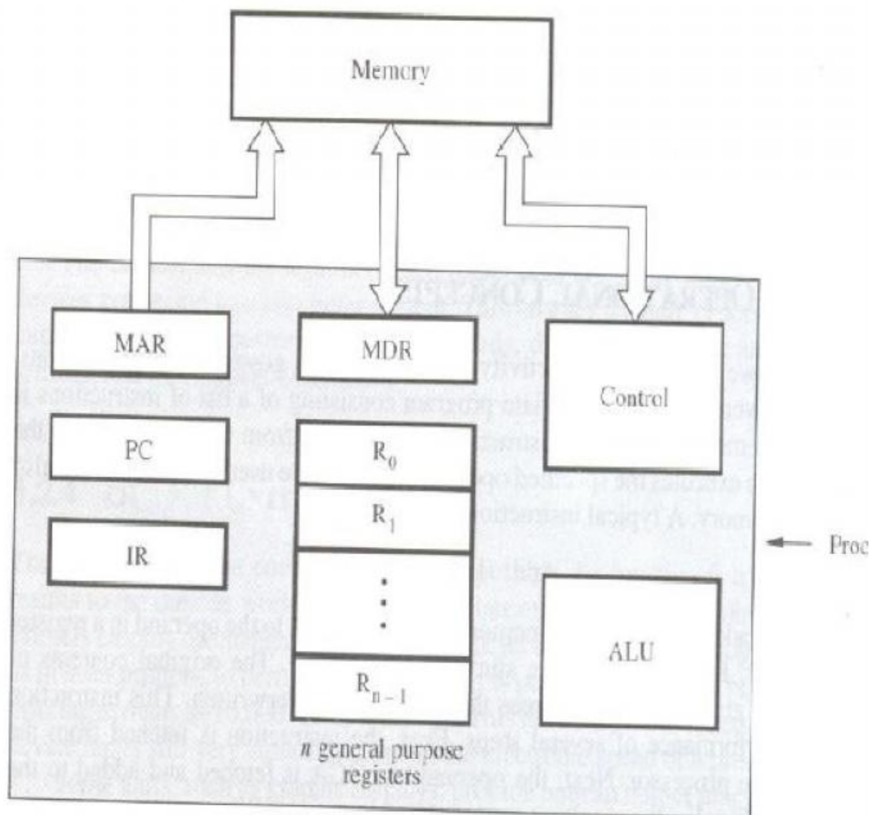
Q1. (a) List the steps needed to execute the machine instruction given below in terms of transfer between the components of processor, memory & some commands ADD LOCA, R0. Assume the instruction is stored in memory location 'INSTR'. (6)

- _ Transfer the contents of register PC to register MAR
- _ Issue a Read command to memory, and then wait until it has transferred the requested word into register MDR
- _ Transfer the instruction from MDR into IR and decode it
- _ Transfer the address LOCA from IR to MAR
- _ Issue a Read command and wait until MDR is loaded
- _ Transfer contents of MDR to the ALU
- _ Transfer contents of R0 to the ALU
- _ Perform addition of the two operands in the ALU and transfer result into R0
- _ Transfer contents of PC to ALU
- _ Add 1 to operand in ALU and transfer incremented address to PC.



(b) Brief about performance & its evaluation process. (4)

Basic Performance Equation:

T = (N*S)/R

Where, T→Performance Parameter
R→Clock Rate in cycles/sec
N→Actual number of instruction execution
S→Average number of basic steps needed to execute one machine instruction.
To achieve high performance, N,S<R

Performance Measurement:
The Performance Measure is the time it takes a computer to execute a given benchmark.
A non-profit organization called SPEC (System Performance Evaluation Corporation) selects and publishes representative application program.

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

The Overall SPEC rating for the computer is given by,

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

Q2. (a)Write the syntax for basic instruction types and assembly language for the F=ax^2+bx+c. (8)

The general instruction format is: **Operation Source1,Source2,Destination**
Symbolic add instruction: **ADD A,B,C**

The general instruction format is: **Operation Source,Destination**
Symbolic add instruction: **MOVE B,C**
ADD A,C

The general instruction format is: **Operation operand**
Symbolic add instruction: **LOAD A**
ADD B
STORE C

```
LOAD A          MUL X, X        MUL X, X, R0
MUL   X         MUL X, A        MUL R0, A, R1
MUL   X         MUL X, B        MUL X, B, R2
STORE R0        ADD A, B        ADD R1, R2, R3
LOAD B          ADD B, C        ADD R3, C, F
MUL   X         MOV C, F
ADD   C
ADD   R0
MOV F
```

(b) Define Bus. **(2)**

A group of lines that serves as the connection path to several devices is called a Bus.
A Bus may be lines or wires or one bit per line.
The lines carry data or address or control signal.

There are 2 types of Bus structures. They are
Single Bus Structure
Multiple Bus Structure

Q3. Define Addressing modes and explain any 5 of its types. **(10)**

1. Register addressing mode - The operand is the contents of a processor register; the name (address) of the register is given in the instruction.

Example: **MOVE R1,R2**

This instruction copies the contents of register R2 to R1.

2. Absolute addressing mode - The operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called Direct.)

Example: **MOVE LOC,R2**

This instruction copies the contents of memory location of LOC to register R2.

3. Immediate addressing mode The operand is given explicitly in the instruction.

Example: **MOVE #200 , R0**

The above statement places the value 200 in the register R0. A common convention is to use the sharp sign (#) in front of the value to indicate that this value is to be used as an immediate operand.

INDIRECTION AND POINTERS

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which the memory address of the operand can be determined. We refer to this address as the effective address (EA) of the operand.

4. Indirect addressing mode - The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

Example **Add (R2),R0**

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 Rl and uses the Immediate addressing mode to place the address value NUM 1, which is the address of the first number in the list, into R2.

INDEXING AND ARRAY

It is useful in dealing with lists and arrays.

5. Index mode - The effective address of the operand is generated by adding a constant value to the contents of a register.

Where X denotes the constant value contained in the instruction and Ri is the name of the register involved. The effective address of the operand is given by **EA = X + [Ri].** The contents of the index register are not changed in the process of generating the effective address.

Example : EA= 20+1000=1020

6. Relative mode - The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

This mode can be used to access data operands. But, its most common use is to specify the target address in branch instructions. An instruction such as **Branch>O LOOP** causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

7.

Auto-increment mode:

- The Effective Address of the operand is the contents of a register in the instruction.
- After accessing the operand, the contents of this register is automatically incremented to point to the next item in the list.

Mode	Assembler syntax	Addressing Function
Auto-increment	(Ri)+	EA=[Ri]; Increment Ri

Auto-decrement mode:

- The Effective Address of the operand is the contents of a register in the instruction.
- After accessing the operand, the contents of this register is automatically decremented to point to the next item in the list.

Mode	Assembler Syntax	Addressing Function
Auto-decrement	-(Ri)	EA=[Ri]; Decrement Ri

Q4. (a) Define Assembler directives & list the assembler directives used in assembly language.(6)

There are some instructions in the assembly language program which are not a part of processor instruction set. These instructions are instructions to the assembler, linker, and loader. These are referred to as **pseudo-operations** or as **assembler directives.** The assembler directives enable we to control the way in which a program assembles and lists. They act during the assembly of a program and do not generate any executable machine code.

The assemble language requires assembler directives for performing following basic functions.

- To indicate starting location of the memory where the data block is stored and starting location of the memory where code is stored.
- To define different types of variables or to set aside one or more storage locations of corresponding data type in memory.
- To indicate the assembler about the values of the variables.
- To indicate start and end of subroutine program.

ORIGIN : This assembler directive tells assembler that where to place the data block in the memory or where to start loading of object program in the memory. In short, the ORIGIN directive specifies the starting memory locations for data and object code.

DB, DW, DD, DQ, and DT : These directives are used to define different types of variables, or to set aside one or more storage locations of corresponding data type in memory. These are known as **data control directives**. Their definitions are as follows :

DB - Define Byte

DW - Define Word

DD - Define Doubleword

DQ - Define Quadword

DT - Define Ten Bytes

Example :

```
AMOUNT DB 10H, 20H, 30H, 40H    ; Declare array of 4 bytes named
                                ; AMOUNT
```

DUP : The DUP directive can be used to initialize several locations and to assign values to these locations.

Format : Name Data_Type Num DUP (value)

Example : TABLE DW 10 DUP (0)   ; Reserve an array of 10

EQU : The EQU directive is used to redefine a data name or variable with another data name, variable, or immediate value. The directive should be defined in a program before it is referenced.

Formats :

Numeric Equate	: name EQU expression
String Equate	: name EQU <string>

Example :

NUM EQU 200	; It defines NUM = 200
ST EQU <'This is string'>	; It defines as string

PROC : The procedures in the programs can be defined by PROC directive. The procedure name must be present, must be unique, and must follow naming conventions for the language. After the PROC directive the term NEAR or FAR are issued to specify the type of the procedure.

ENDP : ENDP directive is used along with the PROC directive. ENDP defines the end of the procedure.

(b) Explain about Nested Subroutines. (4)

Ans: 1. Subroutine nesting is a procedure in which a subroutine calls another subroutine. It can go in any depth.

2. It is essential to save the contents of link register in some other location before calling another subroutine.

3. If it is not done, the return address of the first subroutine is lost.

4. Therefore the addresses need to be saved in last in first out or First-in-last out manner.

5. This suggest they should be pushed onto a stack.

6. A particular register is designed as the stack pointer, SP this pointer points to a stack called processor stack.

7. The call instruction pushes the PC contents into the Processor stack.

8. The return address pops the address from this stack.

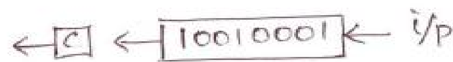
Thus about the subroutine nesting & processor stack.

Q5. (a) Illustrate Shift Instructions with an example. (5)

(a) Logical shifts

* Logical Shift Left
  syntax: LShiftL count, dst
* Logical Shift Right
  syntax: LShiftR count, dst

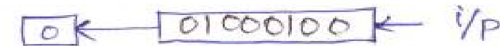
eg: LogicalShift Left LShiftL #2, R0



(1 shift) 1 00100010 — replace by zero for vacant bit

(2nd shift) 0 01000100 — "

o/p



LogicalShift Right LShift R #2, R0 (6)

1st shift

replace zero for vacant bit

2nd shift

→ Arithmetic Shift :

* ArithmeticShift left
  syntax: AShiftL #2, R0
* ArithmeticShift left
  syntax : AShiftR #2, R0

eg:

Arithmetic Shift left   AShiftL #2, R0

C ← 01000100 ← i/p

0 ← 10001000

1 ← 00010000

vacant bit is replaced by zero (* specifically for left shift)

Arithmetic Shift Right   A shiftR #1, R0

i/p → 01000100 → C

→ 00100010 → 0

MSB of sign bit is (+ve) which is 0 it is replaced in the vacant bit pos

AshiftR #1, R0

R0

→ 10001000 → C

→ 1 1000100 → 0

MSB of sign bit is (-ve) 1, which is replaced here.

(b) Write an ALP for that reads one line of character from keyboard, stores in memory & echoes it back to display. (5).

Label	Instruction	Operands	Comment
	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.

Q6. Explain design of fast adders with necessary diagrams. ***(10)***

A **carry-look ahead adder** (CLA) or **fast adder** is a type of adder used in digital logic. A carry-look ahead adder improves speed by reducing the amount of time required to determine carry bits.

The carry-look ahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits.

One method of speeding up this process by eliminating inter stage carry delay is called **lookahead-carry addition.** This method utilizes logic gates to look at the lower-order bits of the augend and addend to see if a higher-order carry is to be generated. It uses two functions : carry generate and carry propagate.

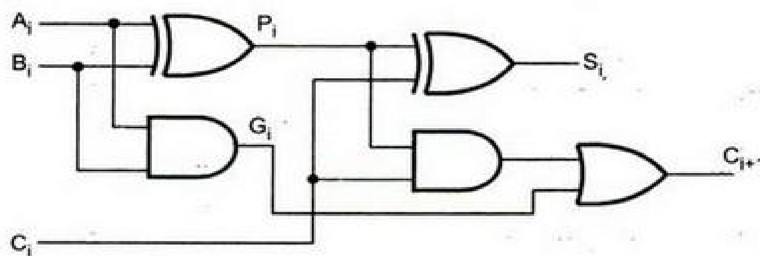


Fig. 2.8 Full adder circuit

Consider the circuit of the full adder shown in Fig. 2.8. Here, we define two functions : **carry generate** and **carry propagate.**

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$ (Refer Appendix-A for details.)

Inputs			Outputs	
A	B	C_{in}	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 2.1 Truth table for full-adder

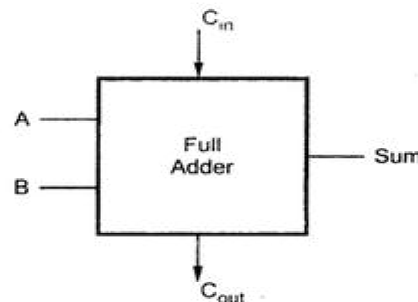


Fig. 2.1 Block schematic of full-adder

Sum $= \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$ **Or Sum** $= A \oplus B \oplus C_{in}$

$C_{out} = AB + A\,C_{in} + B\,C_{in}$

Consider the circuit of the full adder shown in Fig. 2.8. Here, we define two functions : carry **generate** and carry **propagate.**

$$P_i = A_i \oplus B_i$$

$$G_i = A_i\,B_i \text{ (Refer Appendix-A for details.)}$$

The output sum and carry can be expressed as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i\,C_i$$

G_i is called a carry generate and it produces on carry when both A_i and B_i are one, regardless of the input carry. P_i is called a carry propagate because it is term associated with the propagation of the carry from C_i to C_{i+1}. Now C_{i+1} can be expressed as a sum of products function of the P and G outputs of all the preceding stages. For example, the carriers in a four stage carry-lookahead adder are defined as follows :

$$C_1 = G_0 + P_0\,C_{in}$$

$$C_2 = G_1 + P_1\,C_1 = G_1 + P_1G_0 + P_1\,P_0\,C_{in}$$

$$C_3 = G_2 + P_2\,C_2 = G_2 + P_2\,G_1 + P_2\,P_1\,G_0 + P_2\,P_1\,P_0\,C_{in}$$

$$C_4 = G_3 + P_3\,C_3$$

$$= G_3 + P_3\,G_2 + P_3\,P_2\,G_1 + P_3\,P_2\,P_1\,G_0 + P_3\,P_2\,P_1\,P_0\,C_{in}$$

Taking the equation (2)

$$C_{i+1} = x_i y_i + x_i C_i + y_i C_i$$

$$= x_i y_i + (x_i + y_i) C_i$$

$$= G_i + P_i C_i$$

$G_i = x_i y_i$ And $P_i = (x_i + y_i)$

G_i = generate propagation delay.

P_i = Propagation carry delay [delayed caused by the addition of previous carry)

As we considered 4bit - parallel addition in carry lookahead logic adder.

$C_{i+1} = G_i + P_i C_i$

sub $i = 0$

$C_1 = G_0 + P_0 C_0$ — (3)

$C_2 = G_1 + P_1 C_1$ — (4) sub (3) in (4)

$= G_1 + P_1 (G_0 + P_0 C_0)$

$= G_1 + P_1 G_0 + P_1 P_0 C_0$

sub C_2 value in eqr

$C_3 = G_2 + P_2 C_2$

$= G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0)$

$= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

sub C_3 value. in eqr

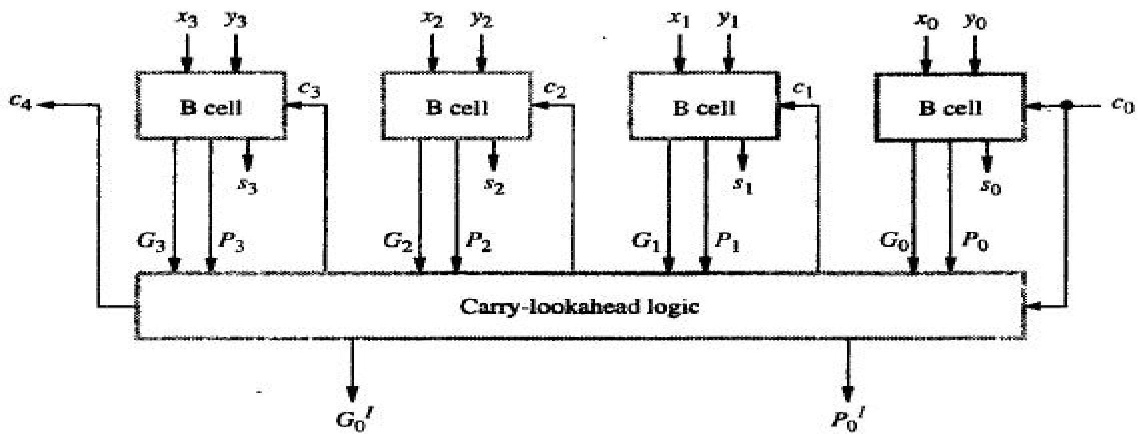
$C_4 = G_3 + P_3 C_3$

$= G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0)$

$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$.

The number of gate delays in every case is uniformly 3 for C1, C2, C3, C4 and sum is obtaintd after 7 gate delays after 4 gate delays.

It means S4 is obtained



(b) 4-bit adder

Q7. Perform Non restoring division of 1001 / 11. **(10)**

Step	A	Q	Working
R1	00000	1001	
∵ $A_0=0$ LS by 1	00001	001_	A = 00001
		001	−B = 11101
A−B	11110		11110
∵ $A_0=1$, $Q_0=0$	11110	0010	
R2	11110	0010	A = 11100
∵ $A_0=1$	11110	0010	+B = 00011
LS by 1	11100	010_	
A+B	11111	010	
∵ $A_0=1$, $Q_0=0$	11111	0100	
R3	11111	0100	
LS by 1	11110	100_	A = 11110
A+B	00001	100_	+B = 00011
∵ $A_0=0$	00001	1001	00001
R4	00001	1001	
LS by 1	00011	001_	A = 00011
A−B	00000	001_	−B = 11101
∵ $A_0=0$	00000	0011	00000
	R	Q	

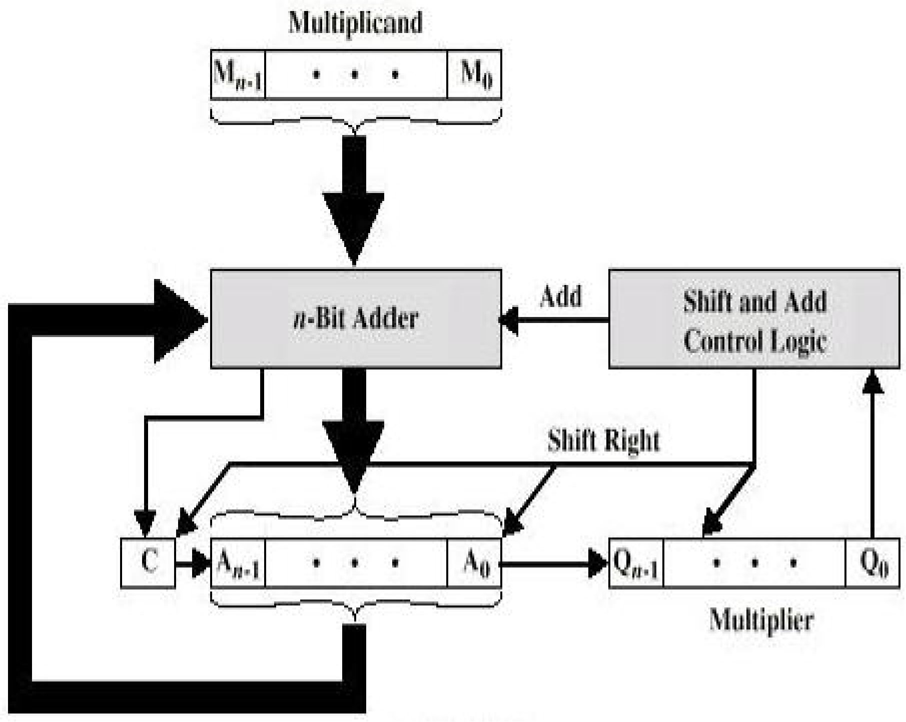
Q8. (a) Draw the circuitry diagram & perform sequential multiplication 13 & 6. (5)

(b) Perform Bit pair recoding multiplication process of 13 & -7. (5)

C (0) **A(0000)** **q(0111)**

	C	A	Q	
		0000	0110	
R1	0	0000 0000	0110 0011	
R2	0	1101 0110	0011 1001	1101 0000 1101
R3	1	0011 1001	1001 1100	1101 0110 1]0011
	0	1001 0100	1100 1110	

01001110



b) 13 – 01101

−7 – 11001

1 | 1 1 1 0 0 1 | 0

0 0/−1 0/1 −1

0 −2 1

i	i−1	value
0	0	0
1	0	−1
0	1	1
1	1	0

0 1 1 0 1

0 −2 1

12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	0	0	1	1	0	1
1	1	1	1	1	0	0	1	1	0		
0	0	0	0	0	0	0	0				
1	1	1	1	1	0	1	0	0	1	0	1

−(91)