1

COMPUTER ABSTRACTIONS AND TECHNOLOGY

Unit Structure

1.0 OBJECTIVES

In this chapter you will learn about:

- A brief history of computer development
- The different types of computers
- The basic structure of a computer and its operation
- Number and character representations

1.1 INTRODUCTION

The computer organization is the function and designs of various units of digital computers that store and process information. It also deals with input units of computer which receive information from external sources and the output units which send computed results to external destinations. The input, storage, processing and output operations are governed by a list of instructions that constitute a program.

Computer hardware consists of electronic circuits, magnetic and optical storage devices, displays, electromechanical devices, and communication facilities. Computer architecture encompasses the specification of an instruction set and the functional behavior of the hardware units that implement the instructions.

1.2 FUNCTIONAL UNITS AND THEIR INTERACTION

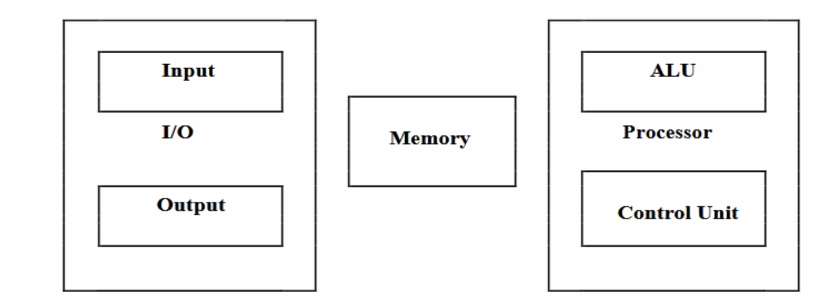


Fig a : Functional units of computer

Figure above shows the functional units of the computer. A computer consists of five functionally independent main parts input, memory, arithmetic logic unit (ALU), and output and control unit.

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.

Examples of input devices:
Keyboard, Joysticks, trackballs, mouse, scanners etc are other input devices.

Memory unit:

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

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 semiconductor storage cells. Each capable of storing one bit of

information. These are processed in a group of fixed sized memory unit called as ‘word’.

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 many 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.

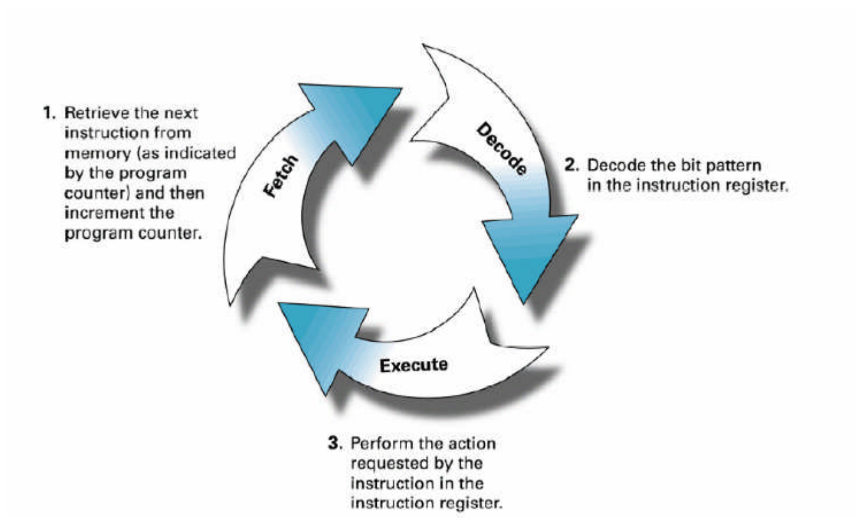
Examples:-Printer, speakers, monitor etc

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.

1.3 BASIC STRUCTURE AND OPERATION

Computer is a fast electronic calculating machine which accepts digital input, processes it according to the internally stored instructions (Programs) and produces the result on the output device. The internal operation of the computer can be as depicted in the following diagram


1. Retrieve the next instruction from memory (as indicated by the program counter) and then increment the program counter.
Fetch
Decode
2. Decode the bit pattern in the instruction register.
Execute
3. Perform the action requested by the instruction in the instruction register.

Basic operation of an instruction

An Instruction consists of two parts, an Operation code and operand/s as shown below:

Opcode operand

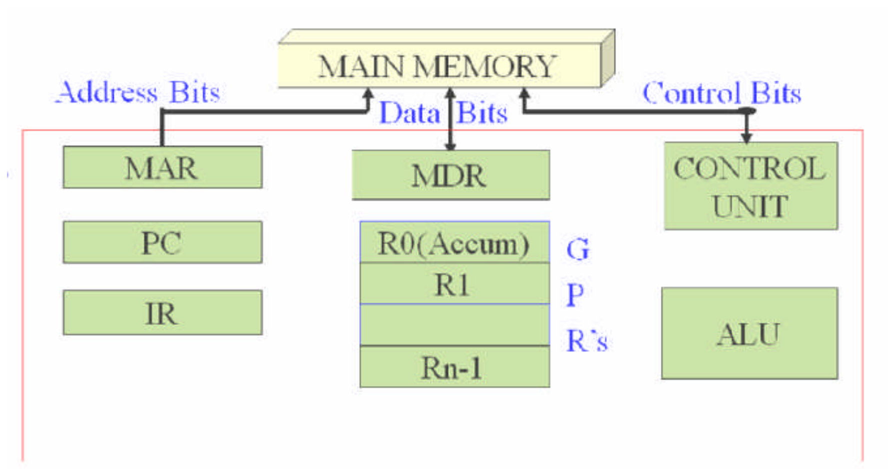
Let us take an instruction

ADD LOCA, R0

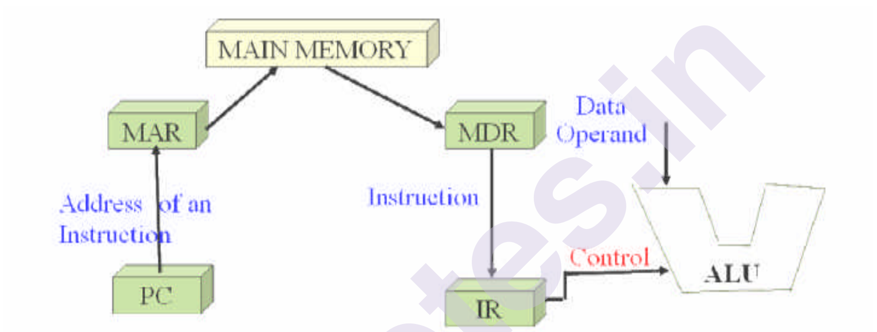
Step to execute instruction

1. Fetch the instruction from main memory into the processor
2. Fetch the operand at location LOCA from main memory into the processor Register R1
3. Add the content of Register R1 and the contents of register R0
4. Store the result (sum) in R0.

The following diagram shows how the memory and the processor are connected. As shown in the diagram, in addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The instruction register holds the instruction that is currently being executed. The program counter keeps track of the execution of the program. It contains the memory address of the next instruction to be fetched and executed. There are n general purpose registers R_0 to R_{n-1} which can be used by the programmers while writing the programs.



The interaction between the processor and the memory and the direction of flow of information is as shown in the diagram below:



1.4 REPRESENTATION OF NUMBERS AND CHARACTERS

1.4.1 Number representation

a) Integer representation

The binary numbers used in digital computers must be represented by using binary storage devices such as Flip-Flops (FF). Each device represent one bit. The most direct number system representation for binary valued storage devices is an integer representation system. Simply writing the value or states of the flip-flops gives the number in integer form.

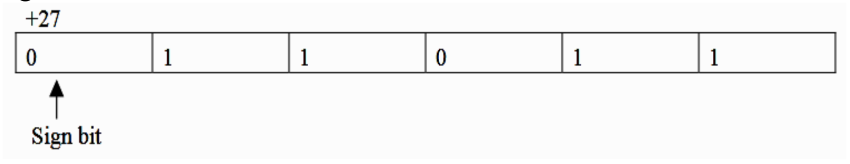
For example, a 6-bit FF register could store binary numbers ranging from 000000 to 111111 (0 to 63 in decimal). Since digital computers handle +ve as well as –ve numbers, some means is required for representing the sign of the number (+ or -). This is usually done by placing another bit called sign bit to the left of the magnitude bits. A'0' in sign bit position represent a +ve number while a '1' in sign bit position represent a –ve number.

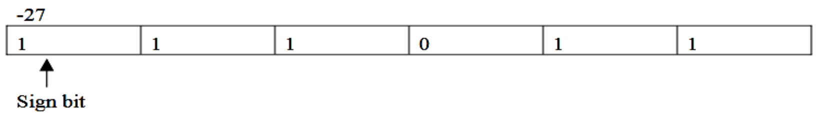
a. Unsigned Integer

Simply writing the values of the number in binary form gives the magnitude of the number in the Unsigned Integer form.

b. Signed Integer

0 in the leftmost bit represents positive and 1 in the sign bit represents negative.





The above example explains representation of +27 and -27 in sign-magnitude representation

Sign magnitude numbers are used only when we do not add or subtract the data. They are used in analog to digital conversions. They have limited use as they require complicated arithmetic circuits.

b)Fixed point and Floating point representation:

A real number or floating point number has integer part and fractional part separated by a decimal.

It is either positive or negative. e.g. 0.345, -121.37 etc.

Fixed Point Representation:

One method of representing real numbers would be to assume a fixed position for the decimal point. e.g. in a 8-bit fixed point representation, where 1 bit is used for sign (+ve or –ve) and 5 bits are used for integral part and two bits are used for fractional part:

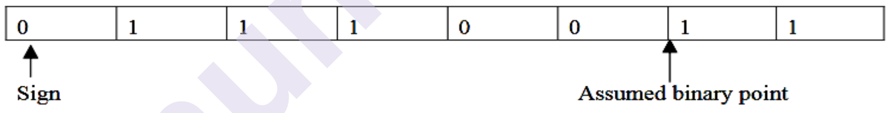


Figure: Representation of fixed point number in memory

The above diagram represents binary number +11100.11

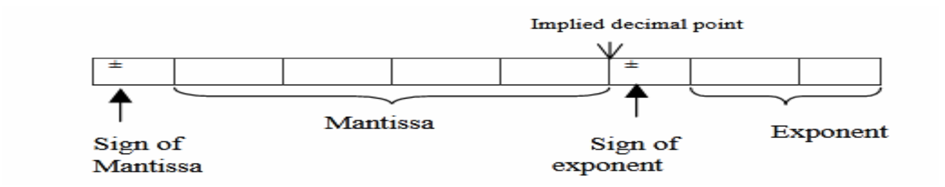
Largest positive number which can be stored is 11111.11 while smallest positive number which can be stored is 00000.01. This range is quite inadequate even for simple arithmetic calculations. To increase the range we use floating point representation.

Floating Point Representation:

In floating point representation, the number is represented as a combination of a mantissa (m), and an exponent (e). In such a representation it is possible to float a decimal point within number towards left or right side.

For example: 53436.256 $= 5343.6256 \times 10^1$
$= 534.36256 \times 10^2$
$=53.436256 \times 10^3$
$=5.3436256 \times 10^4$ and so on
$= 534362.56 \times 10^{-1}$
$= 5343625.6 \times 10^{-2}$
$=53436256.0 \times 10^{-3}$ and so on

Representation of floating point number in computer memory (with four digit mantissa) Let us assume we have hypothetical 8 digit computer out of which four digits are used for mantissa and two digits are used for exponent with a provision of sign of mantissa and sign of exponent.



1.4.2 Character Representation

In computer memory, character are "encoded" (or "represented") using a chosen "character encoding schemes".

For example, in ASCII:

- Code numbers 65D (41H) to 90D (5AH) represents 'A' to 'Z', respectively.
- Code numbers 97D (61H) to 122D (7AH) represents 'a' to 'z', respectively.
- Code numbers 48D (30H) to 57D (39H) represents '0' to '9', respectively.

It is important to note that the representation scheme must be known before a binary pattern can be interpreted.

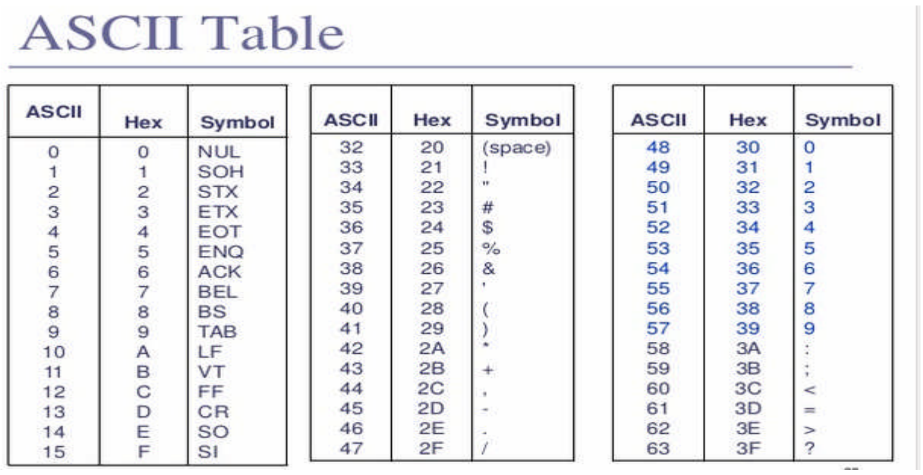
For E.g., the 8-bit pattern "0100 0010B" could represent anything under the sun known only to the person encoded it.

The most commonly-used character encoding schemes are: 7-bit ASCII (ISO/IEC 646) and 8-bit Latin-x (ISO/IEC 8859-x) for western European characters, and Unicode (ISO/IEC 10646) for internationalization (i18n).

A 7-bit encoding scheme (such as ASCII) can represent 128 characters and symbols. An 8-bit character encoding scheme (such as Latin-x) can represent 256 characters and symbols; whereas a 16-bit encoding scheme (such as Unicode UCS-2) can represents 65,536 characters and symbols.

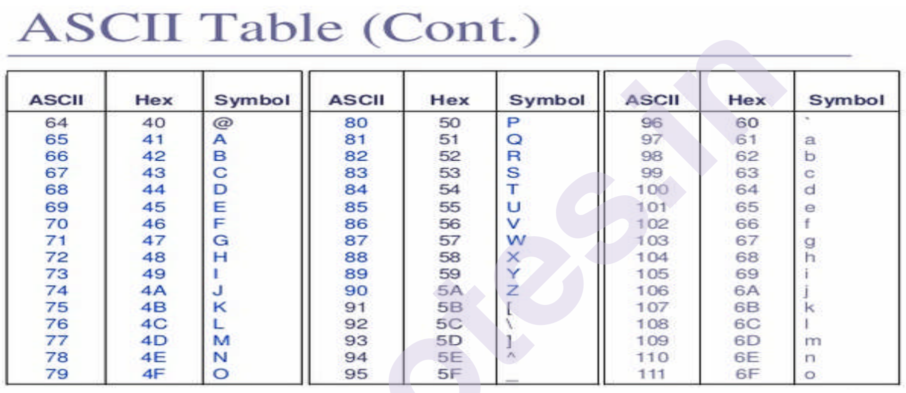
In this representation some symbols are non graphic and some are graphics means some cannot be printed or displayed, for example “line feed”, “null”, “escape” etc. and some are printed which includes alphabets, digits , punctuation mark and other symbols etc.

For example the ASCII code table



ASCII Table

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	32	20	(space)	48	30	0
1	1	SOH	33	21	!	49	31	1
2	2	STX	34	22	"	50	32	2
3	3	ETX	35	23	#	51	33	3
4	4	EOT	36	24	$	52	34	4
5	5	ENQ	37	25	%	53	35	5
6	6	ACK	38	26	&	54	36	6
7	7	BEL	39	27	'	55	37	7
8	8	BS	40	28	(	56	38	8
9	9	TAB	41	29	)	57	39	9
10	A	LF	42	2A	*	58	3A	:
11	B	VT	43	2B	+	59	3B	;
12	C	FF	44	2C	,	60	3C	<
13	D	CR	45	2D	-	61	3D	=
14	E	SO	46	2E	.	62	3E	>
15	F	SI	47	2F	/	63	3F	?



ASCII Table (Cont.)

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
64	40	@	80	50	P	96	60	`
65	41	A	81	51	Q	97	61	a
66	42	B	82	52	R	98	62	b
67	43	C	83	53	S	99	63	c
68	44	D	84	54	T	100	64	d
69	45	E	85	55	U	101	65	e
70	46	F	86	56	V	102	66	f
71	47	G	87	57	W	103	67	g
72	48	H	88	58	X	104	68	h
73	49	I	89	59	Y	105	69	i
74	4A	J	90	5A	Z	106	6A	j
75	4B	K	91	5B	[	107	6B	k
76	4C	L	92	5C	\	108	6C	l
77	4D	M	93	5D	]	109	6D	m
78	4E	N	94	5E	^	110	6E	n
79	4F	O	95	5F	_	111	6F	o

1.4.3 Number system

The technique to represent and work with numbers is called number system. Decimal number system is the most common number system. Other popular number systems include binary number system, octal number system, hexadecimal number system, etc.

1. Decimal Number System

It consists of total 10 digits to represent number given as 0,1,2,3,4,5,6,7,8,9.

Thus the base of the number system is 10. Value of any digit in the decimal number will be decided depending on its position. For example take a number 2,345. Here value of 2 is 2000. As its position is thousand positions. The above number can be expressed as

2*1000+3*100+4*10+5*1=2,345

Thus in above representation the power of 10 increases from 0 to 3 from right to left.

2. Binary Number system:

It consists of only two digits to represent i.e. 0 and 1. So the base of the number system is 2. Base is also called as **radix.** The value of each bit is depends on the position of the bit. So position plays role of power to the base to decide value of bit.

3. **Octal Number System:**
   It consists of 8 digits starting from 0 up to 7. So base or radix of number system is 8.

4. **Hexadecimal Number System**
   It consists of 16 symbols first ten digits starting from 0 up to 9 and remaining 6 symbols A,B,C,D,E,F. So base or radix of number system is 16.

Number System Relationship

HEXADECIMAL	DECIMAL	OCTAL	BINARY
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111

1.4.4 Conversion of Number System

Conversions Related to Decimal System
Two types of conversions

1. **Conversion from any Radix r to Decimal**

   Steps to follow

   (a) Note down given number

   (b) Write down weights for different positions.

   (c) Multiply each digit in given number with corresponding weight to obtain product numbers.

   (d) Add all the product numbers to get the decimal equivalent.

For example (Binary to Decimal)

10100011= $(1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) +$ $(1 \times 2^1) + (1 \times 2^0)$

= 128 + 0 + 32 + 0 + 0 + 0 + 2 + 1

= 163

Therefore, binary number **$(10100011)_2$ = $(163)_{10}$** decimal number

Similarly for other radix take the power of the respective radix and obtain the equivalent number

2. Conversion from Decimal to Other System

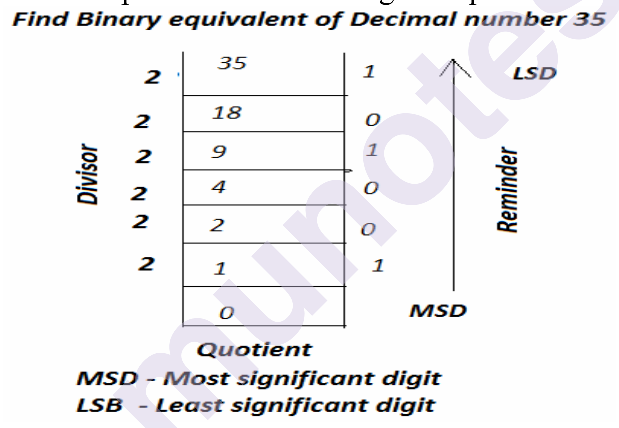
Number in any other radix can be converted to decimal steps to be followed is as follows

(a) Divide integer part of given decimal by the base, note the reminder

(b) Continue to divide the quotient by the base (r) until there is nothing left, keeping track of reminders from each step. (Select the quotient such that the remainder is always less than the base)

(c) List reminder values in reverse order to the equivalent.

For example take the following example


Find Binary equivalent of Decimal number 35
Divisor
2
2
2
2
2
2
35
18
9
4
2
1
0
1
0
1
0
0
1
LSD
MSD
Reminder
Quotient
MSD - Most significant digit
LSB - Least significant digit

Thus the $(35)_{10}$= $(101001)_2$

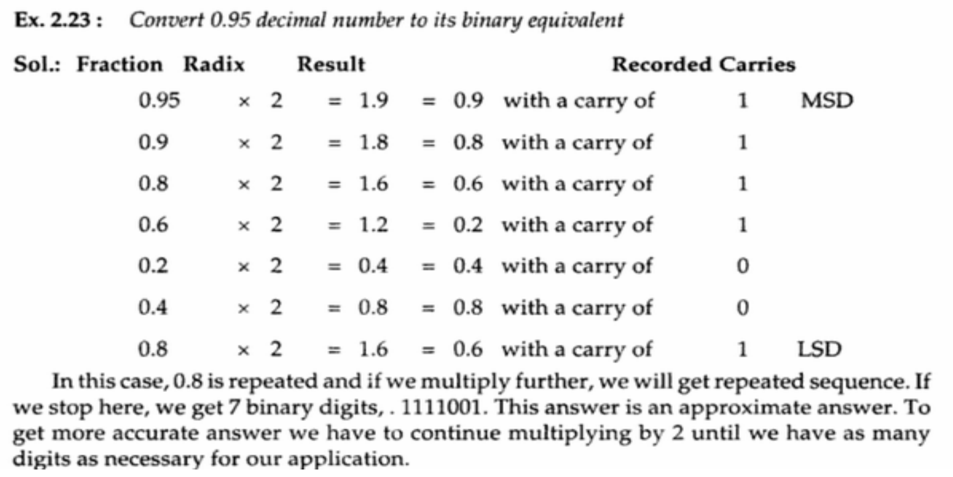
In the same steps we can obtain decimal number from any radix. The divisor will be changed by the radix.

3. Fractional part conversion:

Steps to convert decimal to any other radix

1. Multiply given fractional decimal number by the base
2. Record carry in the result i.e. integer part
3. Multiply fractional part by radix
4. Repeat step 2 and 3 up to end
5. First carry will be treated as MSD and last will be treated as LSD

Following figure shows the example of same



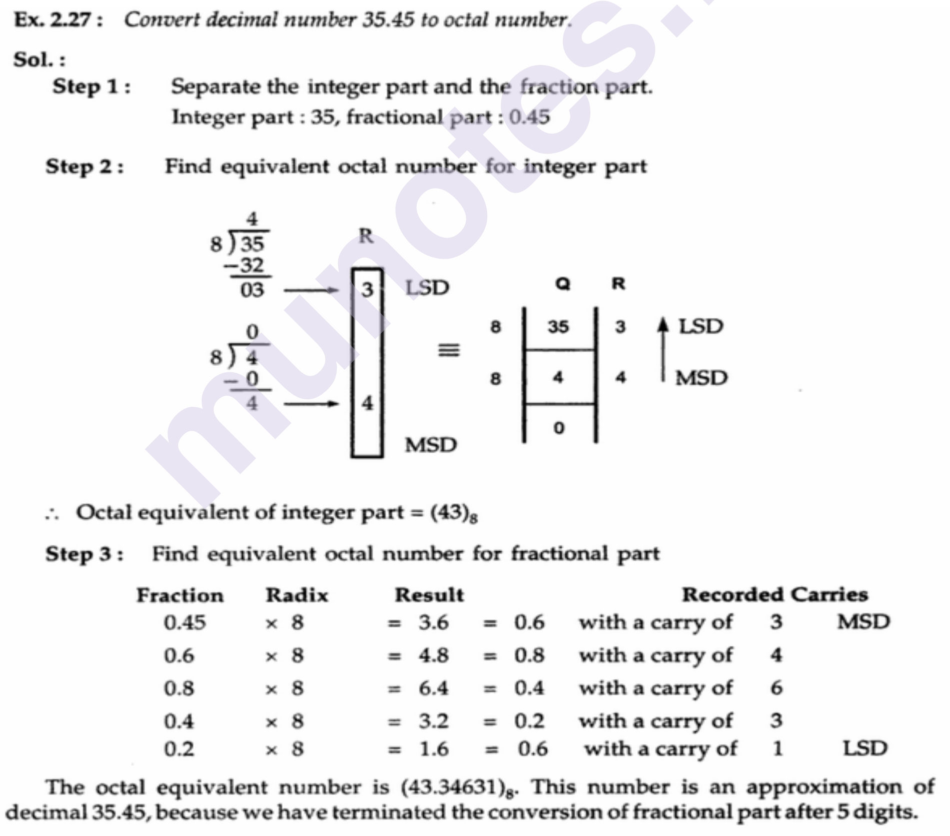
Ex. 2.23 : *Convert 0.95 decimal number to its binary equivalent*

Sol.:

Fraction	Radix	Result			Recorded Carries	
0.95	× 2	= 1.9	= 0.9	with a carry of	1	MSD
0.9	× 2	= 1.8	= 0.8	with a carry of	1	
0.8	× 2	= 1.6	= 0.6	with a carry of	1	
0.6	× 2	= 1.2	= 0.2	with a carry of	1	
0.2	× 2	= 0.4	= 0.4	with a carry of	0	
0.4	× 2	= 0.8	= 0.8	with a carry of	0	
0.8	× 2	= 1.6	= 0.6	with a carry of	1	LSD

In this case, 0.8 is repeated and if we multiply further, we will get repeated sequence. If we stop here, we get 7 binary digits, . 1111001. This answer is an approximate answer. To get more accurate answer we have to continue multiplying by 2 until we have as many digits as necessary for our application.

In case of mixed part i.e. integer and fraction part number we have to separate out the integer part and fractional part and carry out the conversion process mentioned in above procedures. Following figure shows the example.

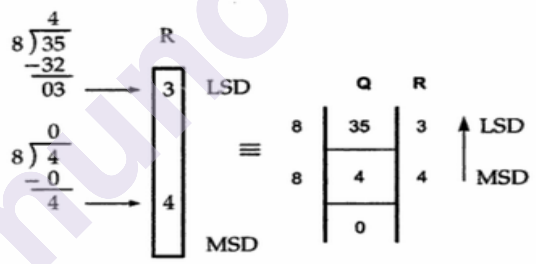


Ex. 2.27 : *Convert decimal number 35.45 to octal number.*

Sol. :

Step 1 : Separate the integer part and the fraction part.
Integer part : 35, fractional part : 0.45

Step 2 : Find equivalent octal number for integer part



∴ Octal equivalent of integer part = $(43)_8$

Step 3 : Find equivalent octal number for fractional part

Fraction	Radix	Result			Recorded Carries	
0.45	× 8	= 3.6	= 0.6	with a carry of	3	MSD
0.6	× 8	= 4.8	= 0.8	with a carry of	4	
0.8	× 8	= 6.4	= 0.4	with a carry of	6	
0.4	× 8	= 3.2	= 0.2	with a carry of	3	
0.2	× 8	= 1.6	= 0.6	with a carry of	1	LSD

The octal equivalent number is $(43.34631)_8$. This number is an approximation of decimal 35.45, because we have terminated the conversion of fractional part after 5 digits.

1.4.5 Conversion from Binary to other system

1 Binary to Octal Conversion

Steps to follow

a. Mark 3-bit from LSB to make Group of 3 bits
b. Convert each group into its equivalent octal number

Use the table

Octal Number	Binary Equivalent		
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Example : Convert $(111011011)_2$ into its equivalent octal
Solution : Given number is 111011011
Grouping 111 011 011
Starting from LSB group 1 = 011 = 3
Group 2= 011 = 3
Group 3 = 111 =7
So equivalent no is $(733)_8$

2 Binary to Hex Conversion
Steps to follow

a. Mark 4-bit from LSB to make Group of 4 bits
b. Convert each group into its equivalent hex equivalent.

Example : Convert $(111011011)_2$ into its equivalent hex.
Solution : Given number =111011011
Grouping = 000111011011 (3 extra zeros added to LSB)
Starting from LSB Group 1 =1011 = B
Group 2 = 1101 = D
Group 3 = 0001 = 1
So equivalent no is $(1DB)_{16}$

1.4.6 Conversion from other to Binary System

1 Octal to Binary Conversion

Steps to follow

a. Convert each octal digit into its equivalent binary using 3 bit

b. Write the binary equivalent bits in order to obtain binary no.

For example : Convert $(765)_8$ number in equivalent binary

Solution : binary equivalent of 5=101

binary equivalent of 6=110

binary equivalent of 7=111

so binary equivalent will be $(111110101)_2$

2 Octal to Hex Conversion

Steps to follow

a. Convert each hex digit into its equivalent binary using 3-bit

b. Write the binary equivalent bits in order to obtain binary no.

For example : Convert $(732)_{18}$ number in equivalent binary

Solution : binary equivalent of 7=111
binary equivalent of 3=011
binary equivalent of 1=001
so binary equivalent will be $(111\ 011\ 001)_2$

c. Convert the binary to hexadecimal using group of 4

111011001 = 0001 1101 1001
= 1 D 9

1.4.7 Two's Complement Number System

This scheme is most popularly used for number representation.

2's complement of binary number can be obtained by adding 1 to LSB of 1's complement of that number.

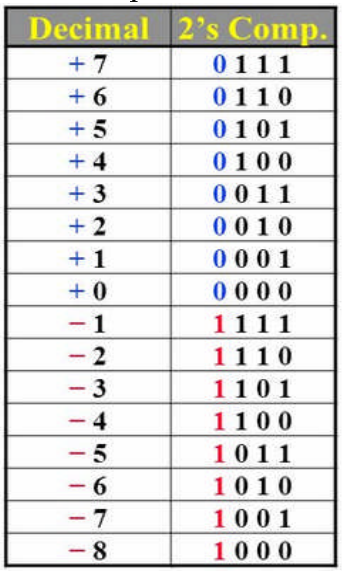
So 2's complement = 1's complement+1

Representation of Positive and Negative Numbers using 2's Complement

Positive numbers in 2's complement is same as that of signed number representation. E.g.(+5) is represented as 0101 in 2's complement form.

Negative numbers are 2's complement of corresponding positive numbers. E.g. (-6) is represented in 2's complement form as follows

Number given = -6
Binary equivalent of 6 is =0110
2's complement =1001+1= 1010



Decimal	2's Comp.
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
+0	0000
−1	1111
−2	1110
−3	1101
−4	1100
−5	1011
−6	1010
−7	1001
−8	1000

The figure lists the 4 bit representation of signed number.

1.5 LET US SUM UP

Thus, we have studied the basic concepts about the structure of computers and their operation. Machine instructions and programs have been described briefly. The addition and
Subtraction of binary numbers has been explained.

1.6 LIST OF REFERENCES

- Carl Hamacher et al., Computer Organization and Embedded Systems, 6 ed., McGraw-Hill 2012
- Patterson and Hennessy, Computer Organization and Design, Morgan Kaufmann, ARM Edition, 2011
- R P Jain, Modern Digital Electronics, Tata McGraw Hill Education Pvt. Ltd. , 4th Edition, 2010

1.7 UNIT END EXERCISES

1) Write a note on computer number system.
2) Explain basic functional units of computer.

❖❖❖❖

2

LOGIC CIRCUITS AND FUNCTIONS

Unit Structure

2.0 OBJECTIVES

In this chapter you will learn about:

- Machine instructions and program execution
- Addressing methods for accessing register and memory operands
- Assembly language for representing machine instructions, data, and programs
- Stacks and subroutines

2.1 INTRODUCTION

Logic gates are the basic building block of digital electronics. A gate is an electronic device which is used to compute a function on a two valued signal.

Combinational circuits are defined as the time independent circuits which do not depends upon previous inputs to generate any output are termed as combinational circuits. **Sequential circuits** are that which are dependent on clock cycles and depends on present as well as past inputs to generate any output.

A multiplexer is a circuit that accepts many input but give only one output. A demultiplexer function exactly in the reverse of a multiplexer, that is a demultiplexer accepts only one input and gives many outputs. Generally multiplexer and demultiplexer are used together, because of the communication systems are bi directional.

2.2 STUDY OF BASIC LOGIC GATES

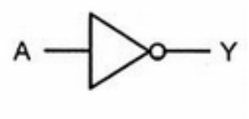
Logic gates are the basic building blocks of any digital system. It is an electronic circuit having one or more than one input and only one output. The relationship between the input and the output is based on certain logic. Based on this, logic gates are named as AND gate, OR gate, NOT gate etc. All the gates have graphical symbol, mathematical equation, truth table which describes the behavior of the each gate.

Classification of logic gates

1. Basic Gates

a) NOT (Inverter) gate

i. Symbol :



Here input to gate is one named as A. and one output Y

ii. Equation:

$Y = \overline{A}$

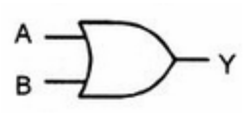
So the value at input will be inverted at output as shown in truth table.

iii. Truth table:

Input A	Output $Y=\overline{A}$
0	1
1	0

b) OR gate

i. Symbol:



Here A and B are two inputs and Y is one output, the output of the OR gate is HIGH when at least one input is HIGH

ii. Equation:

$$Y = A+B$$

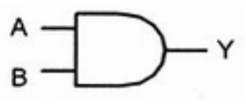
iii. Truth table

Input		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

The truth table shows that the output of OR gate is high at least one input is high.

c) AND gate

i. Symbol :



Here A and B are two inputs and Y is an output terminal.Equation:

$$Y = A.B$$

ii. Truth table

Input		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

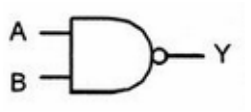
The truth table shows that AND gate output will be high only when both the inputs are high. Otherwise the output is low

2. Universal gates

A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The NAND and NOR gates are universal gates. In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families.

1. NAND gate :

i. Symbol :



Here A and B are two inputs and Y is one output.

ii. Equation :

$Y=\overline{A.B}$

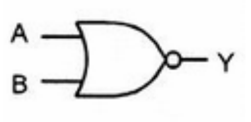
iii. Truth table :

Input		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

The truth table shows that NAND gate output will be high when at least one of the inputs is low.

2. NOR gate :

i. Symbol :



Here A and B are two inputs and Y is one output.

ii. Equation :

$Y=\overline{A + B}$

iii. Truth table :

Input		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

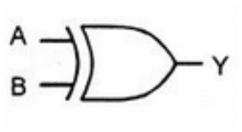
The truth table shows that output ofNOR gate output will behigh when all the inputs are low.

3. Exclusive gates :

There are two types of exclusives gates present as stated and explained below:

a. Exclusive OR (EX-OR)

i. Symbol :



Here A and B are two inputs and Y is one output.

Equation :

Y=$A \oplus$ B

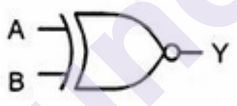
ii. Truth table :

Input		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

The truth table shows that EX-OR gate output will be high when odd numbers of inputs are low. When even number of inputs are high then output is low.

b. Exclusive NOR

i. Symbol :



Here A and B are two inputs and Y is one output.

ii. Equation :

Y=$\overline{A \oplus B}$

iii. Truth table :

Input		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

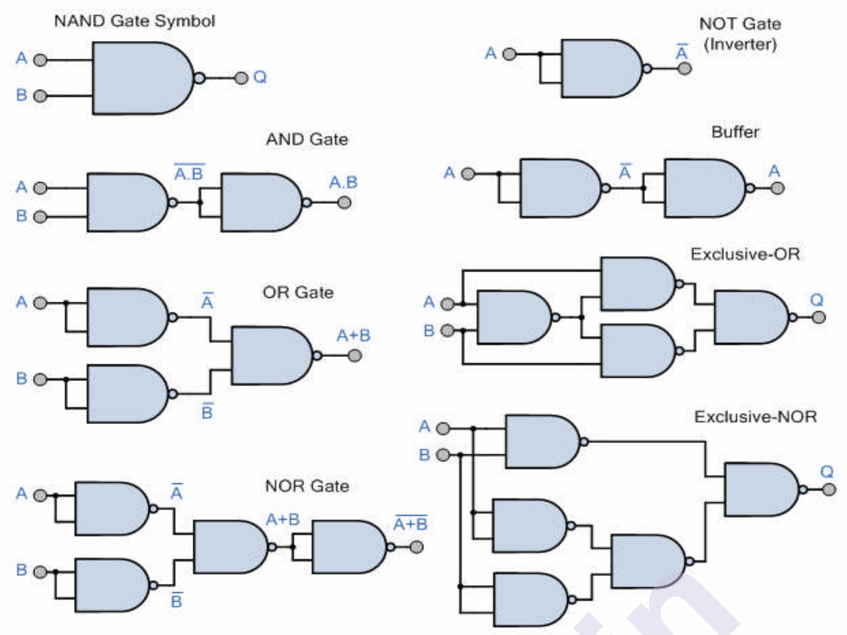
The truth table shows that EX-OR gate output will be high when even numbers of inputs are low. When even number of inputs are high then output is low.

NOR and NAND as Universal Gate.

NOR is OR gate with inverter

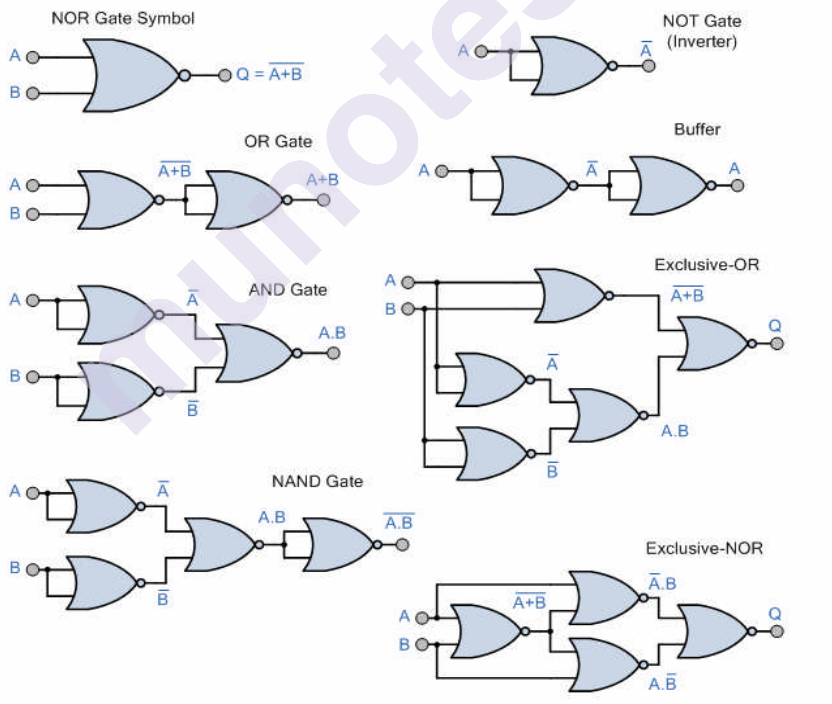
NAND is AND gate with inverter

Logic Gates using only NAND Gates



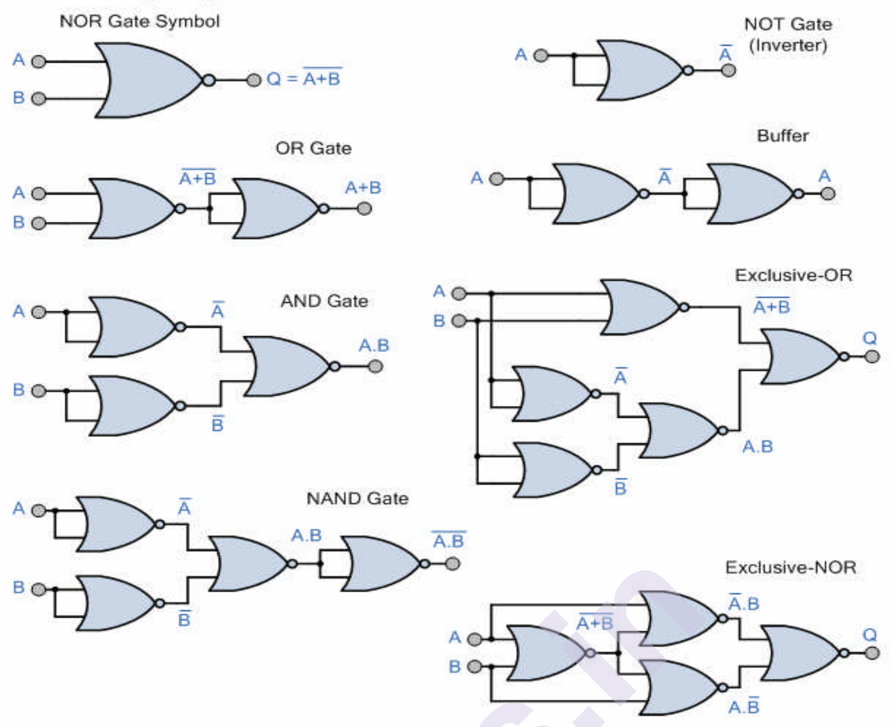
Thus ALL other logic gate functions can be created using only NAND gates making it a universal logic gate.

Logic Gates using only NOR Gates



Thus ALL other logic gate functions can be created using only NOR gates making it also a universal logic gate.

Logic Gates using only NOR Gates



Thus ALL other logic gate functions can be created using only NOR gates making it also a universal logic gate.

2.3 LAWS OF BOOLEAN ALGEBRA

1. The basic Laws of Boolean Algebra can be stated as follows:
   1. Commutative Law states that the interchanging of the order of operands in a Boolean equation does not change its result. For example:
      1. OR operator → A + B = B + A
      2. AND operator → A . B = B . A

   2. Associative Law of multiplication states that the AND operation are done on two or more than two variables. For example: A . (B . C) = (A .B) . C

   3. Distributive Law states that the multiplication of two variables and adding the result with a variable will result in the same value as multiplication of addition of the variable with individual variables. For example:
      A + B.C = (A + B) .(A + C).

   4. Annulment law:
      A . 0 = 0
      A + 1 = 1
   5. Identity law:
      A.1 = A
      A + 0 = A

6. Idempotent law:
   $A + A = A$
   $A.A = A$
7. Complement law:
   $A + \overline{A} = 1$
   $A \,.\, \overline{A} = 0$
8. Double negation law:
   $\overline{\overline{A}} = A$
9. Absorption law:
   $A.(A+B) = A$
   $A + A.B = A$

De Morgan's Law:

De Morgan's Law is also known as De Morgan's theorem, works depending on the concept of Duality. Duality states that interchanging the operators and variables in a function, such as replacing 0 with 1 and 1 with 0, AND operator with OR operator and OR operator with AND operator.

De Morgan stated 2 theorems, which will help us in solving the algebraic problems in digital electronics. The De Morgan's statements are:

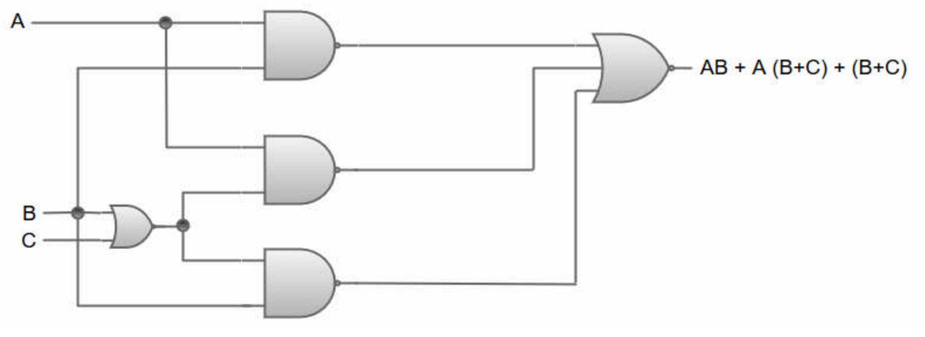
1. "The negation of a conjunction is the disjunction of the negations", which means that the complement of the product of 2 variables is equal to the sum of the complements of individual variables.
   For example,
   $\overline{A \cdot B} \equiv \overline{A} + \overline{B}$

2. "The negation of disjunction is the conjunction of the negations", which means that complement of the sum of two variables is equal to the product of the complement of each variable.
   For example,
   $\overline{A + B} \equiv \overline{A} \cdot \overline{B},$

2.3.1 Simplification using Boolean algebra

Let us consider an example of a Boolean function:
AB+A (B+C) + B (B+C)

The logic diagram for the Boolean function AB+A (B+C) + B (B+C) can be represented as:



We will simplify this Boolean function on the basis of rules given by Boolean algebra.

AB + A (B+C) + B (B+C)

AB + AB + AC + BB + BC {Distributive law; A (B+C) = AB+AC, B (B+C) = BB+BC}

AB + AB + AC + B + BC {Idempotent law; BB = B}

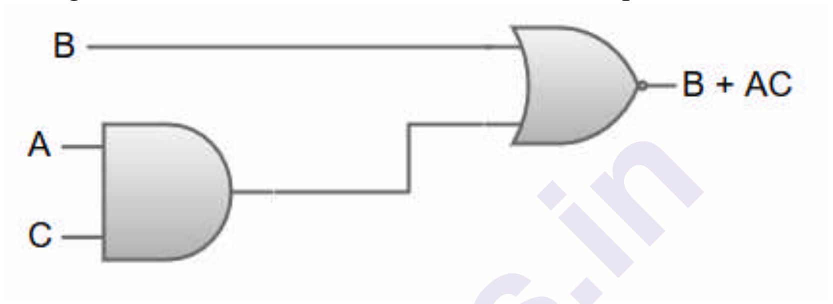
AB + AC + B + BC {Idempotent law; AB+AB = AB}

AB + AC +B {Absorption law; B+BC = B}

B + AC {Absorption law; AB+B = B}

Hence, the simplified Boolean function will be B + AC.

The logic diagram for Boolean function B + AC can be represented as:



Boolean Function Representation

The use of switching devices like transistors give rise to a special case of the Boolean algebra called as switching algebra. In switching algebra, all the variables assume one of the two values which are 0 and 1.

In Boolean algebra, 0 is used to represent the 'open' state or 'false' state of logic gate. Similarly, 1 is used to represent the 'closed' state or 'true' state of logic gate.

A Boolean expression is an expression which consists of variables, constants (0-false and 1-true) and logical operators which results in true or false.

A Boolean function is an algebraic form of Boolean expression. A Boolean function of n-variables is represented by f(x1, x2, x3….xn). By using Boolean laws and theorems, we can simplify the Boolean functions of digital circuits. A brief note of different ways of representing a Boolean function is as follows.

- Sum-of-Products (SOP) Form
- Product-of-sums (POS) form
- Canonical forms

There are two types of canonical forms:

- Sum-of-min terms or Canonical SOP
- Product-of- max terms or Canonical POS

Boolean functions can be represented by using NAND gates and also by using K-map (Karnaugh map) method. We can standardize the Boolean expressions by using by two standard forms.

SOP form – Sum Of Products form
POS form – Product Of Sums form

Standardization of Boolean equations will make the implementation, evolution and simplification easier and more systematic.

2.3.2 Sum of Product (SOP) Form

The sum-of-products (SOP) form is a method (or form) of simplifying the Boolean expressions of logic gates. In this SOP form of Boolean function representation, the variables are operated by AND (product) to form a product term and all these product terms are ORed (summed or added) together to get the final function.

A sum-of-products form can be formed by adding (or summing) two or more product terms using a Boolean addition operation. Here the product terms are defined by using the AND operation and the sum term is defined by using OR operation.

The sum-of-products form is also called as Disjunctive Normal Form as the product terms are ORed together and Disjunction operation is logical OR. Sum-of-products form is also called as Standard SOP.

SOP form representation is most suitable to use them in FPGA (Field Programmable Gate Arrays).

Examples

$F(A,B,C,D,E)=AB + ABC + CDE$

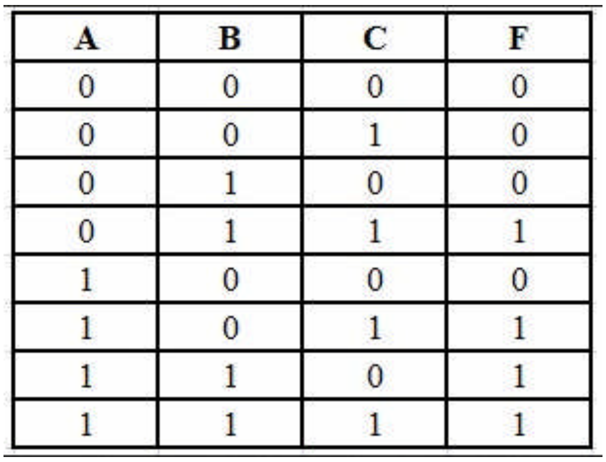
$F(A,B,C,D,E) =(AB)\,\overline{} + ABC + CD\ E\,\overline{}$

SOP form can be obtained by

- Writing an AND term for each input combination, which produces HIGH output.
- Writing the input variables if the value is 1, and write the complement of the variable if its value is 0.
- OR the AND terms to obtain the output function.

Ex: Boolean expression for majority function F = A’BC + AB’C + ABC ‘ + ABC

Truth table:



A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Now write the input variables combination with high output. F = AB + BC + AC.

Checking
By Idempotence law, we know that
([ABC + ABC)] + ABC) = (ABC + ABC) = ABC
Now the function F = A'BC + AB'C + ABC ' + ABC
= A'BC + AB'C + ABC' + ([ABC + ABC)] + ABC)
= (ABC + ABC ') + (ABC + AB'C) + (ABC + A'BC)
= AB (C + C ') + A (B + B') C + (A + A') BC
= AB + BC + AC.

2.3.2 Product of Sums (POS) Form

The product of sums form is a method (or form) of simplifying the Boolean expressions of logic gates. In this POS form, all the variables are ORed, i.e. written as sums to form sum terms.

All these sum terms are ANDed (multiplied) together to get the product-of-sum form. This form is exactly opposite to the SOP form. So this can also be said as "Dual of SOP form".

Here the sum terms are defined by using the OR operation and the product term is defined by using AND operation. When two or more sum terms are multiplied by a Boolean OR operation, the resultant output expression will be in the form of product-of-sums form or POS form.

The product-of-sums form is also called as Conjunctive Normal Form as the sum terms are ANDed together and Conjunction operation is logical AND. Product-of-sums form is also called as Standard POS.

Examples
F(A,B,C,D,E)= (A+B) . (A + B + C) . (C +D)
F(A,B,C,D,E)= (A+B) ¯¯ . (C + D + E ¯¯)

POS form can be obtained by

- Writing an OR term for each input combination, which produces LOW output.
- Writing the input variables if the value is 0, and write the complement of the variable if its value is 1.
- AND the OR terms to obtain the output function.

Ex: Boolean expression for majority function F = (A + B + C) (A + B + C ‘) (A + B’ + C) (A’ + B + C)

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Now write the input variables combination with high output. F = AB + BC + AC.

Checking
By Idempotence law, we know that
[(A + B + C) (A + B + C)] (A + B + C) = [(A + B + C)] (A + B + C) = (A + B + C)
Now the function
F = (A + B) (B + C) (A + C)
= (A + B + C) (A + B + C ‘) (A + B’ + C) (A’ + B + C)
= [(A + B + C) (A + B + C)] (A + B + C) (A + B + C ‘) (A + B’ + C) (A’ + B + C)
= [(A + B + C) (A + B + C ‘)] [(A + B + C) (A’ + B + C)] [(A + B + C) (A + B’ + C)]
= [(A + B) + (C . C ‘)] [(B + C) + (A . A’)] [(A + C) + (B . B’)]
= [(A + B) + 0] [(B + C) + 0] [(A + C) + 0] = (A + B) (B + C) (A + C)

2.3.3 Canonical Form (Standard SOP and POS Form)

Any Boolean function that is expressed as a sum of minterms or as a product of max terms is said to be in its “canonical form”.

It mainly involves in two Boolean terms, “minterms” and “maxterms”.

When the SOP form of a Boolean expression is in canonical form, then each of its product term is called ‘minterm’. So, the canonical form of sum of products function is also known as "minterm canonical form" or Sum-of-minterms or standard canonical SOP form.

Similarly, when the POS form of a Boolean expression is in canonical form, then each of its sum term is called 'max term'. So, the canonical form of product of sums function is also known as "maxterm canonical form or Product-of sum or standard canonical POS form".

2.3.4 Min terms

A min term is defined as the product term of n variables, in which each of the n variables will appear once either in its complemented or un-complemented form. The min term is denoted as mi where i is in the range of $0 \leq i < 2^n$.

A variable is in complemented form, if its value is assigned to 0, and the variable is un-complimented form, if its value is assigned to 1.

For a 2-variable (x and y) Boolean function, the possible minterms are:
x'y', x'y, xy' and xy.

For a 3-variable (x, y and z) Boolean function, the possible minterms are:
x'y'z', x'y'z, x'yz', x'yz, xy'z', xy'z, xyz' and xyz.

- 1 – Minterms = minterms for which the function F = 1.
- 0 – Minterms = minterms for which the function F = 0.

Any Boolean function can be expressed as the sum (OR) of its 1-min terms. The representation of the equation will be

- F(list of variables) = Σ(list of 1-min term indices)

Ex: F (x, y, z) = Σ (3, 5, 6, 7)

The inverse of the function can be expressed as a sum (OR) of its 0- min terms. The representation of the equation will be

- F(list of variables) = Σ(list of 0-min term indices)

Ex: F' (x, y, z) = Σ (0,1, 2, 4)

Examples of canonical form of sum of products expressions (min term canonical form):
i) Z = XY + XZ'
ii) F = XYZ' + X'YZ + X'YZ' + XY'Z + XYZ

In standard SOP form, the maximum possible product terms for n number of variables are given by 2^n. So, for 2 variable equations, the product terms are 22 = 4. Similarly, for 3 variable equations, the product terms are 23 = 8.

2.3.5 Max terms

A max term is defined as the product of n variables, within the range of $0 \leq i < 2^n$. The max term is denoted as Mi. In max term, each variable is complimented, if its value is assigned to 1, and each variable is un-complimented if its value is assigned to 0.

For a 2-variable (x and y) Boolean function, the possible max terms are:
x + y, x + y’, x’ + y and x’ + y’.

For a 3-variable (x, y and z) Boolean function, the possible maxterms are:
x + y + z, x + y + z’, x + y’ + z, x + y’ + z’, x’ + y + z, x’ + y + z’, x’ + y’ + z and x’ + y’ + z’.

- 1 – Max terms = max terms for which the function F = 1.
- 0 – max terms = max terms for which the function F = 0.

Any Boolean function can be expressed the product (AND) of its 0 – max terms. The representation of the equation will be

- F(list of variables) = Π (list of 0-max term indices)

Ex: F (x, y, z) = Π (0, 1, 2, 4)

The inverse of the function can be expressed as a product (AND) of its 1 – max terms. The representation of the equation will be

- F(list of variables) = Π (list of 1-max term indices)

Ex: F’ (x, y, z) = Π (3, 5, 6, 7)

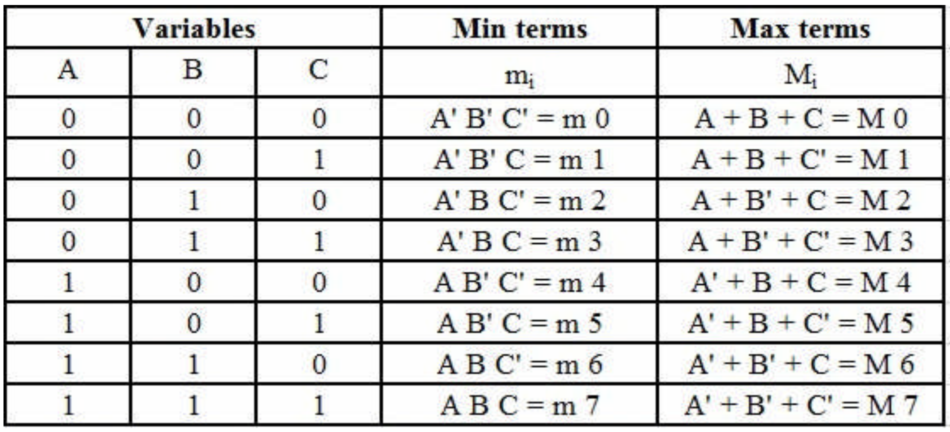
Examples of canonical form of product of sums expressions (max term canonical form):
i. Z = (X + Y) (X + Y')
ii. F = (X' + Y + Z') (X' + Y + Z) (X' + Y' + Z')

In standard POS form, the maximum possible sum terms for n number of variables are given by 2^n. So, for 2 variable equations, the sum terms are $2^2 = 4$.

Similarly, for 3 variable equations, the sum terms are $2^3 = 8$.

Table for 2^3 min terms and 2^3 max terms

The below table will make you understand about the representation of the mean terms and max terms of 3 variables.



Variables			Min terms	Max terms
A	B	C	m_i	M_i
0	0	0	A' B' C' = m 0	A + B + C = M 0
0	0	1	A' B' C = m 1	A + B + C' = M 1
0	1	0	A' B C' = m 2	A + B' + C = M 2
0	1	1	A' B C = m 3	A + B' + C' = M 3
1	0	0	A B' C' = m 4	A' + B + C = M 4
1	0	1	A B' C = m 5	A' + B + C' = M 5
1	1	0	A B C' = m 6	A' + B' + C = M 6
1	1	1	A B C = m 7	A' + B' + C' = M 7

2.4 K-MAP

Karnaugh Map or K-map is introduced by a telecom engineer, Maurice Karnaugh at Bell labs in 1953, as a refined technique of 'Edward Veitch's Veitch diagram' and it is a method to simplify or reduce the complexities of a Boolean expression.

Karnaugh map method or K-map method is the pictorial representation of the Boolean equations and Boolean manipulations are used to reduce the complexity in solving them. These can be considered as a special or extended version of the 'Truth table'.

Karnaugh map can be explained as "An array containing 2^k cells in a grid like format, where k is the number of variables in the Boolean expression that is to be reduced or optimized". As it is evaluated from the truth table method, each cell in the K-map will represent a single row of the truth table and a cell is represented by a square.

The cells in the k-map are arranged in such a way that there are conjunctions, which differ in a single variable, are assigned in adjacent rows. The K-map method supports the elimination of potential race conditions and permits the rapid identification.

By using Karnaugh map technique, we can reduce the Boolean expression containing any number of variables, such as 2-variable Boolean expression, 3-variable Boolean expression, 4-variable Boolean expression and even 7-variable Boolean expressions, which are complex to solve by using regular Boolean theorems and laws.

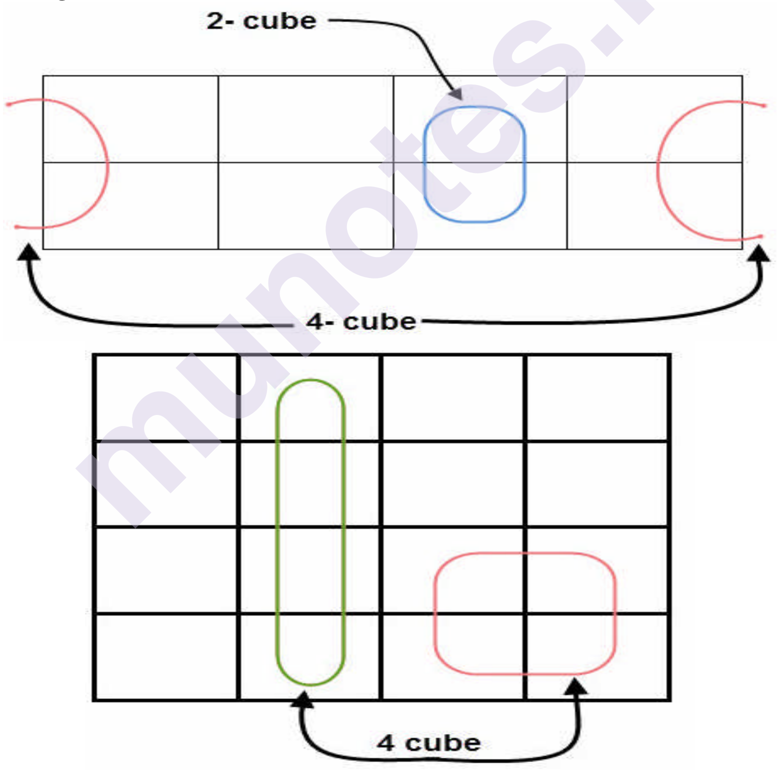
2.4.1 Minimization with Karnaugh Maps and advantages of K-map

- K-maps are used to convert the truth table of a Boolean equation into minimized SOP form.
- Easy and simple basic rules for the simplification.
- The K-map method is faster and more efficient than other simplification techniques of Boolean algebra.

- All rows in the K-map are represented by using a square shaped cells, in which each square in that will represent a minterm.
- It is easy to convert a truth table to k-map and k-map to Sum of Products form equation.

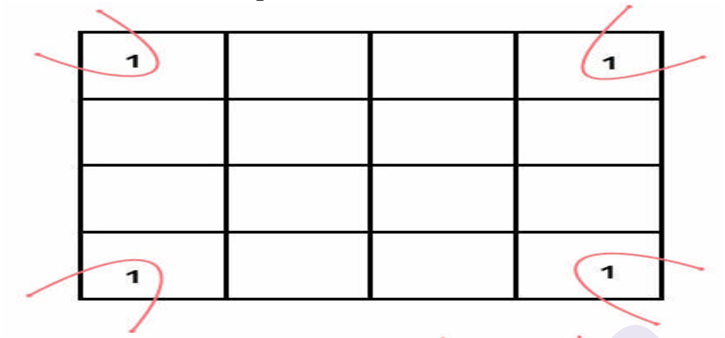
2.4.2 Grouping of K-map variables (SOP case)

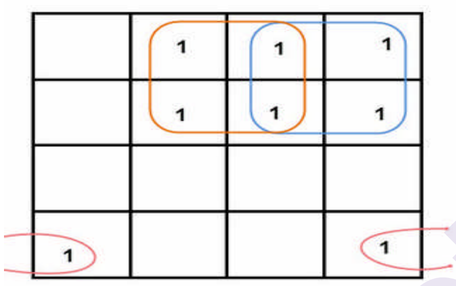
- There are some rules to follow while we are grouping the variables in K-maps. They are
- The square that contains '1' should be taken in simplifying, at least once.
- The square that contains '1' can be considered as many times as the grouping is possible with it.
- Group shouldn't include any zeros (0).
- A group should be the as large as possible.
- Groups can be horizontal or vertical. Grouping of variables in diagonal manner is not allowed.

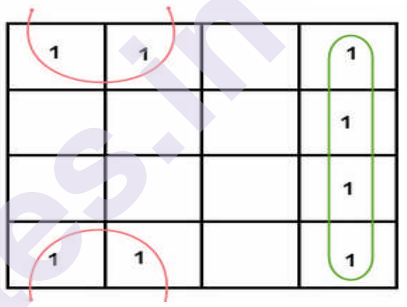


- If the square containing '1' has no possibility to be placed in a group, then it should be added to the final expression.
- Groups can overlap.
- The number of squares in a group must be equal to powers of 2, such as 1, 2, 4, 8 etc.

- Groups can wrap around. As the K-map is considered as spherical or folded, the squares at the corners (which are at the end of the column or row) should be considered as they adjacent squares.
- The grouping of K-map variables can be done in many ways, so they obtained simplified equation need not to be unique always.
- The Boolean equation must be in must be in canonical form, in order to draw a K-map.

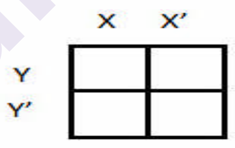




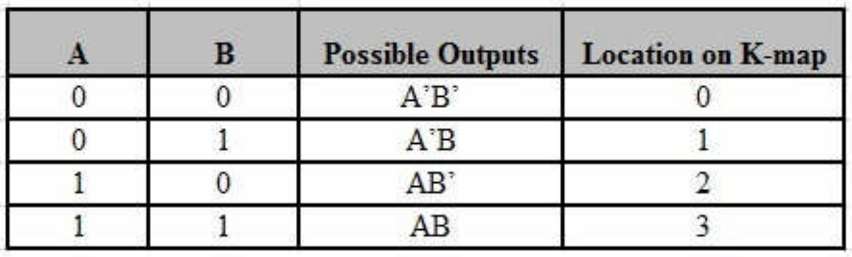


2 variable K-maps

There are 4 cells (2^2)in the 2-variable k-map. It will look like (see below image)

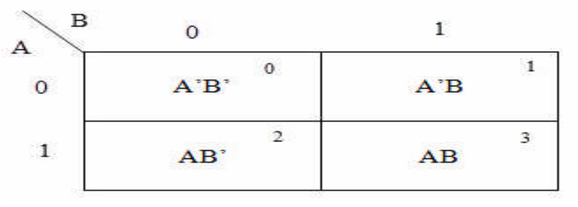


The possible min terms with 2 variables (A and B) are A.B, A.B', A'.B and A'.B'. The conjunctions of the variables (A, B) and (A', B) are represented in the cells of the top row and (A, B') and (A', B') in cells of the bottom row. The following table shows the positions of all the possible outputs of 2-variable Boolean function on a K-map.



A	B	Possible Outputs	Location on K-map
0	0	A'B'	0
0	1	A'B	1
1	0	AB'	2
1	1	AB	3

A general representation of a 2 variable K-map plot is shown below.



When we are simplifying a Boolean equation using Karnaugh map, we represent the each cell of K-map containing the conjunction term with 1. After that, we group the adjacent cells with possible sizes as 2 or 4. In case of larger k-maps, we can group the variables in larger sizes like 8 or 16.

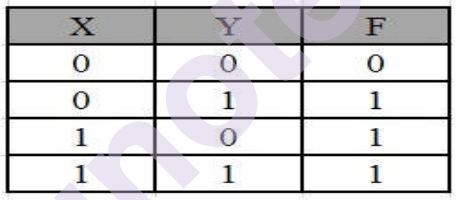
The groups of variables should be in rectangular shape, that means the groups must be formed by combining adjacent cells either vertically or horizontally. Diagonal shaped or L-shaped groups are not allowed. The following example demonstrates a K-map simplification of a 2-variable Boolean equation.

Example

Simplify the given 2-variable Boolean equation by using K-map.

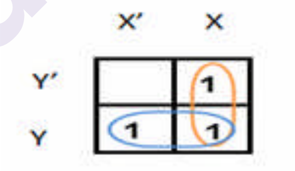
F = X Y' + X' Y + X'Y'

First, let's construct the truth table for the given equation,



X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

We put 1 at the output terms given in equation.



In this K-map, we can create 2 groups by following the rules for grouping, one is by combining (X', Y) and (X', Y') terms and the other is by combining (X, Y') and (X', Y') terms. Here the lower right cell is used in both groups. After grouping the variables, the next step is determining the minimized expression.

By reducing each group, we obtain a conjunction of the minimized expression such as by taking out the common terms from two groups, i.e. X' and Y'. So the reduced equation will be X' +Y'.

3 variable K-maps

For a 3-variable Boolean function, there is a possibility of 8 output min terms. The general representation of all the min terms using 3-variables is shown below.

A	B	C	Output Function	Location on K-map
0	0	0	A’B’C’	0
0	0	1	A’B’C	1
0	1	0	A’BC’	2
0	1	1	A’BC	3
1	0	0	AB’C’	4
1	0	1	AB’C	5
1	1	0	ABC’	6
1	1	1	ABC	7

A typical plot of a 3-variable K-map is shown below. It can be observed that the positions of columns 10 and 11 are interchanged so that there is only change in one variable across adjacent cells. This modification will allow in minimizing the logic.

A \ BC	00	01	11	10
0	A’B’C’ 0	A’B’C 1	A’BC 3	A’BC’ 2
1	AB’C’ 4	AB’C 5	ABC 7	ABC’ 6

Up to 8 cells can be grouped in case of a 3-variable K-map with other possibilities being 1,2 and 4.

Example

Simplify the given 3-variable Boolean equation by using k-map.

F = X’ Y Z + X’ Y’ Z + X Y Z’ + X’ Y’ Z’ + X Y Z + X Y’ Z’

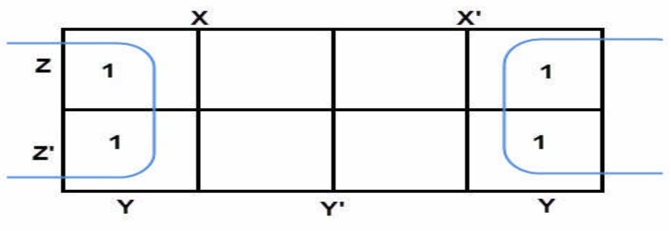
First, let’s construct the truth table for the given equation,

x	y	z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

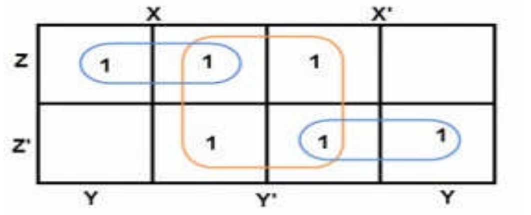
We put 1 at the output terms given in equation.

There are 8 cells (2^3) in the 3-variable k-map. It will look like (see below image).

The largest group size will be 8 but we can also form the groups of size 4 and size 2, by possibility. In the 3 variable Karnaugh map, we consider the left most column of the k-map as the adjacent column of rightmost column. So the size 4 group is formed as shown below.



And in both the terms, we have 'Y' in common. So the group of size 4 is reduced as the conjunction Y. To consume every cell which has 1 in it, we group the rest of cells to form size 2 group, as shown below.



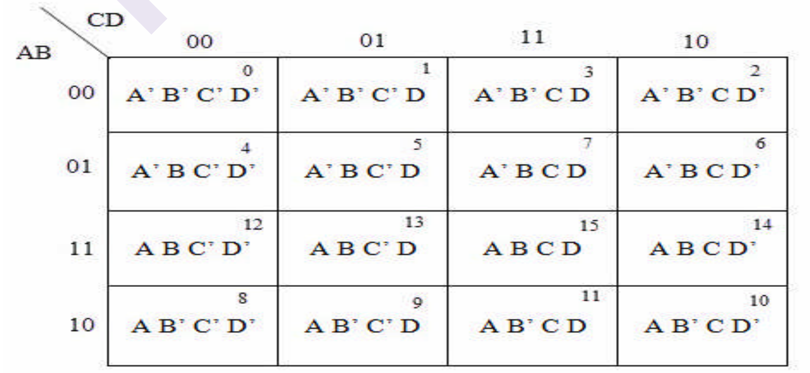
The 2 size group has no common variables, so they are written with their variables and its conjugates. So the reduced equation will be X Z' + Y' + X' Z. In this equation, no further minimization is possible.

4 variable K-maps

There are 16 possible min terms in case of a 4-variable Boolean function. The general representation of minterms using 4 variables is shown below.

A	B	C	D	Output function	K-map location
0	0	0	0	A' B' C' D'	0
0	0	0	1	A' B' C' D	1
0	0	1	0	A' B' C D'	2
0	0	1	1	A' B' C D	3
0	1	0	0	A' B C' D'	4
0	1	0	1	A' B C' D	5
0	1	1	0	A' B C D'	6
0	1	1	1	A' B C D	7
1	0	0	0	A B' C' D'	8
1	0	0	1	A B' C' D	9
1	0	1	0	A B' C D'	10
1	0	1	1	A B' C D	11
1	1	0	0	A B C' D'	12
1	1	0	1	A B C' D	13
1	1	1	0	A B C D'	14
1	1	1	1	A B C D	15

A typical 4-variable K-map plot is shown below. It can be observed that both the columns and rows of 10 and 11 are interchanged.



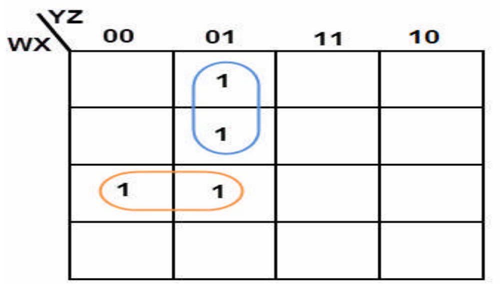
AB \ CD	00	01	11	10
00	0 A' B' C' D'	1 A' B' C' D	3 A' B' C D	2 A' B' C D'
01	4 A' B C' D'	5 A' B C' D	7 A' B C D	6 A' B C D'
11	12 A B C' D'	13 A B C' D	15 A B C D	14 A B C D'
10	8 A B' C' D'	9 A B' C' D	11 A B' C D	10 A B' C D'

The possible number of cells that can be grouped together are 1, 2, 4, 8 and 16.

Example

Simplify the given 4-variable Boolean equation by using k-map. F (W, X, Y, Z) = (1, 5, 12, 13)

Sol: F (W, X, Y, Z) = (1, 5, 12, 13)



By preparing k-map, we can minimize the given Boolean equation as

F = W Y’ Z + W ‘Y’ Z

2.5 COMBINATIONAL CIRCUITS

A digital logic circuit is defined as the one in which voltages are assumed to be having a finite number of distinct value. Types of digital logic circuits are combinational logic circuits and sequential logic circuits. These are the basic circuits used in most of the digital electronic devices like computers, calculators, mobile phones.

Digital logic circuits are often known as switching circuits, because in digital circuits the voltage levels are assumed to be switched from one value to another value instantaneously. These circuits are termed as logic circuits, as their operation obeys a definite set of logic rules.

Classification of logical circuits:

Combinational Circuit :

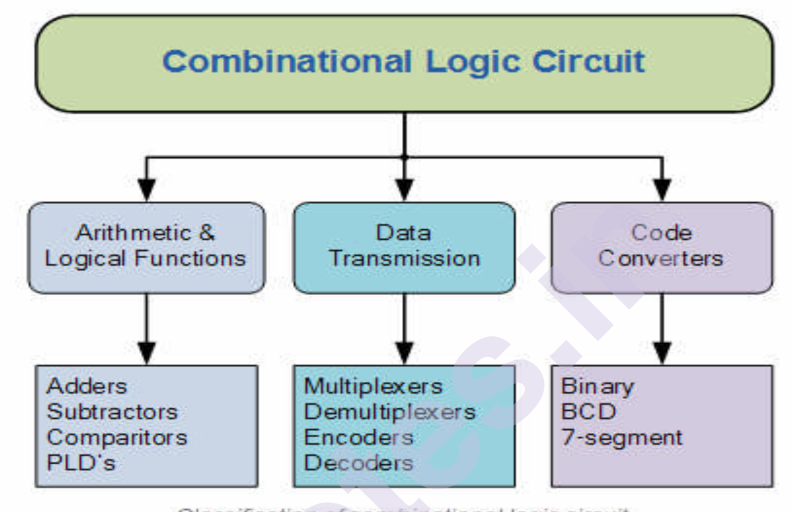
- Combinational digital logic circuits are basically made up of digital logic gates like AND gate, OR gate, NOT gate and universal gates (NAND gate and NOR gate).
- All these gates are combined together to form a complicated switching circuit. The logic gates are building blocks of combinational logic circuits. In a combinational logic circuit, the output at any instant of time depends only on present input at that particular instant of time and combinational circuits do not have any memory devices.
- Encoders and Decoders are examples of combinational circuit. A decoder converts the binary coded data at its present input into a number of different output lines. Other examples of combinational switching circuits are half adder and full adder, encoder, decoder, multiplexer, de-multiplexer, code converter etc.

- Combinational circuits are used in microprocessor and microcontroller for designing the hardware and software components of a computer.

Classification of combinational digital logic circuits

Combinational digital logic circuits are classified into three major parts – arithmetic or logical functions, data transmission and code converter.

The following chart will elaborate the further classifications of combinational digital logic circuit.



Classification of combinational logic circuit

Application of combinational circuit:

Adder:

An **Adder** is a device that can add two binary digits. It is a type of digital circuit that performs the operation of additions of two numbers. It is mainly designed for the addition of binary number, but they can be used in various other applications like binary code decimal, address decoding, table index calculation, etc. There are two types of Adder. One is **Half Adder**, and another one is known as **Full Adder**. The detail explanation of the two types of the adder are as follows

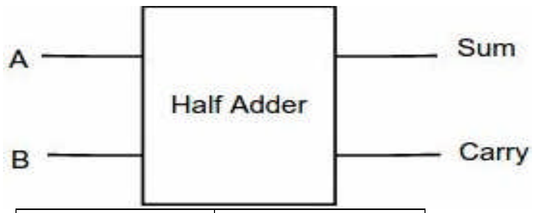
2.5.1 Design Half Adder Circuit

There are two inputs and two outputs in a Half Adder. Inputs are named as A and B, and the outputs are named as Sum (S) and Carry (C). Half adder, is designed to add two one bit number with the help of logic gates. The binary addition as shown below.

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10

Here the output “1” of “10” becomes the carry-out. **SUM** is the normal output and the **CARRY** is the carry-out.

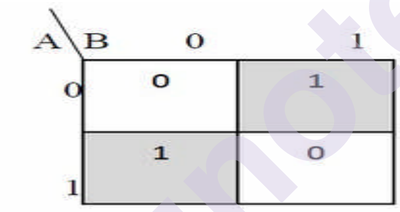
Block diagram of half adder



Inputs		Outputs	
A	**B**	**S**	**C**
0	**0**	**0**	**0**
0	**1**	**1**	**0**
1	**0**	**1**	**0**
1	**1**	**0**	**1**

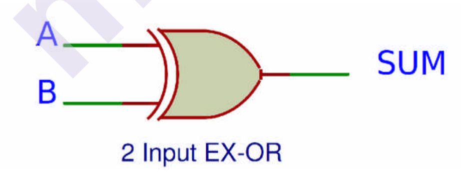
From above truth table we know that we have two k-maps one for Sum and other for Carry

1. K-Map for Sum is as shown below

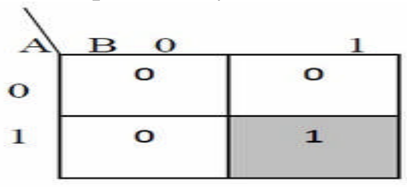


S = A’B+AB’

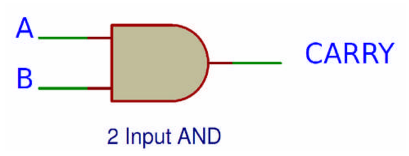
Circuit diagram for this output is EX-OR gate as shown below



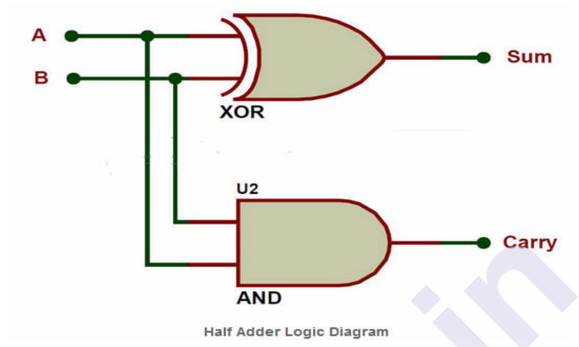
2. K-Map for Carry is as shown below



From kmap we get the output as
C = A.B   The logic diagram for the same is



If A and B are binary inputs to the half adder, then the logic function to calculate sum S is Ex – OR of A and B and logic function to calculate carry C is AND of A and B. Combining these two, the logical circuit to implement the combinational circuit of Half Adder is shown below.



Limitation of Half Adder-

- Half adders have no scope of adding the carry bit resulting from the addition of previous bits.This is a major drawback of half adders.
- This is because real time scenarios involve adding the multiple number of bits which can not be accomplished using half adders.

2.5.2 Design Full Adder

Full Adder-

- Full Adder is a combinational logic circuit.
- It is used for the purpose of adding two single bit numbers with a carry.
- Thus, full adder has the ability to perform the addition of three bits.
- Full adder contains 3 inputs and 2 outputs (sum and carry) as shown

Step 1 : Identify the input and output variables-

- Input variables = A, B, C_{in} (either 0 or 1)
- Output variables = S, C_{out} (where S = Sum and C_{out} = Carry out)

Step 2 : Truth table for the full adder:

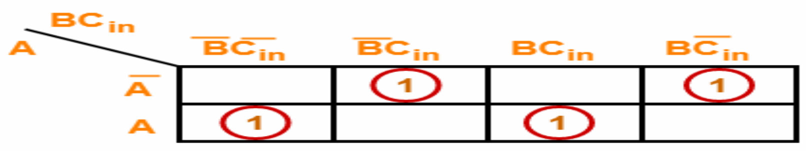
Inputs			**Outputs**	
A	**B**	**Cin**	**S**	**C**
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**0**
0	**1**	**0**	**1**	**0**
0	**1**	**1**	**0**	**1**
1	**0**	**0**	**1**	**0**

1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Step-03:

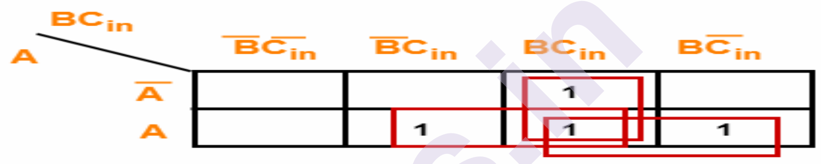
Draw K-maps using the above truth table and determine the simplified Boolean expressions-

For S:



$\boxed{S = A \oplus B \oplus C_{in}}$

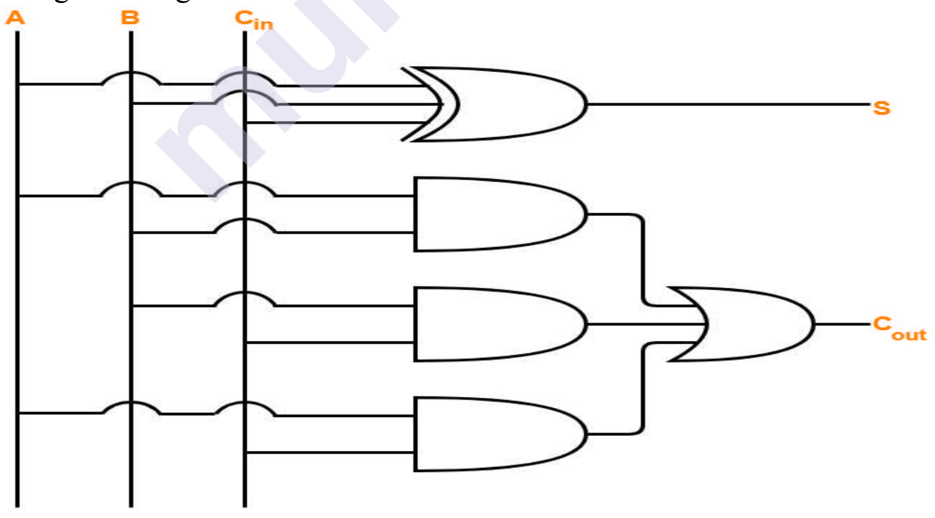
For C_{in}:



$\boxed{C_{out} = AB + BC_{in} + C_{in}A}$

Step-04: Draw the logic diagram.

The implementation of full adder using 1 XOR gate, 3 AND gates and 1 OR gate is as given



Full Adder Logic Diagram

2.5.3 Design Full adder using two half adder.

The full adder can be constructed using two half adder. As shown in the figure below.

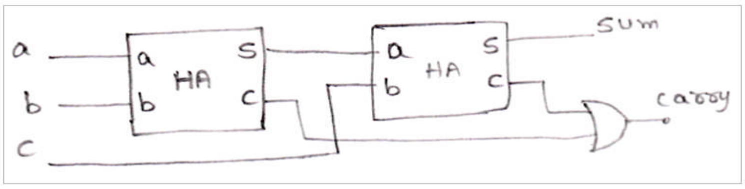


Figure a: Block diagram of full adder using two half adder

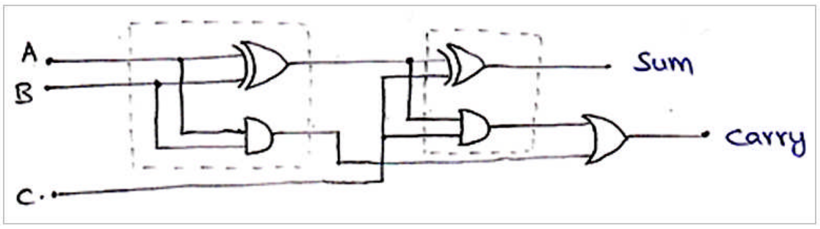


Figure b: Circuit diagram of full adder using two half adder

The proof of how the two half adder is working as full adder for output sum and carry.

$$\begin{aligned}
\text{Sum} &= A \oplus B \oplus C \\
\text{Carry} &= AB + (A \oplus B).\ C \\
&= AB + (\bar{A}.\ B + A.\ \bar{B}).\ C \\
&= AB + \bar{A}.\ BC + A.\ \bar{B}.\ C \\
&= B\ (A + \bar{A}.\ C) + A.\ \bar{B}.\ C \\
&= B\ [(A + \bar{A})\ (A + C)] + A.\ \bar{B}.\ C \\
&= AB + AC + A.\ \bar{B}.\ C \\
&= AB + C\ (B + A.\ \bar{B}) \\
&= AB + C\ [(B + A)\ (B + \bar{B})] \\
&= AB + BC + AC
\end{aligned}$$

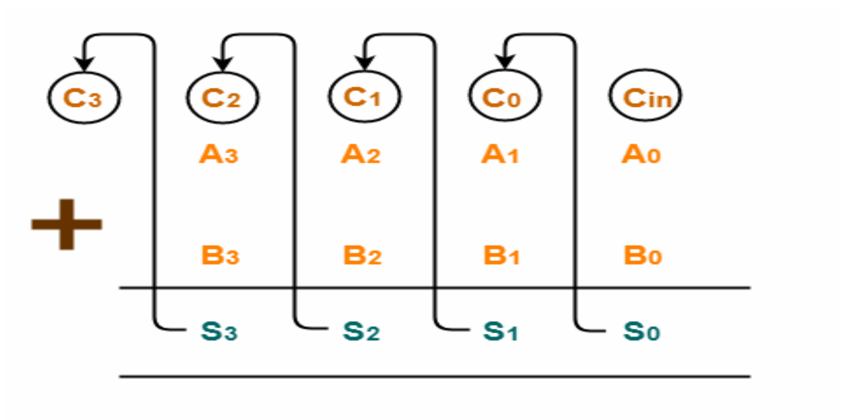
2.5.4 Ripple Carry Adder

- Ripple Carry Adder is a combinational logic circuit.
- It is used for the purpose of adding two n-bit binary numbers.
- It requires n full adders in its circuit for adding two n-bit binary numbers.
- It is also known as **n-bit parallel adder**.

4-bit Ripple Carry Adder-

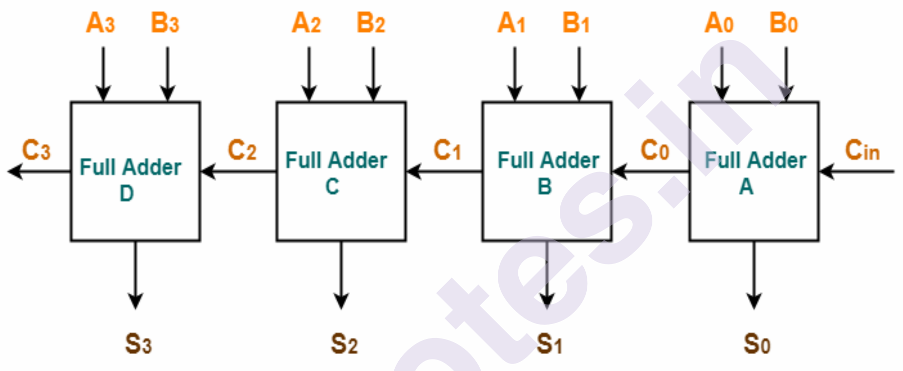
4-bit ripple carry adder is used for the purpose of adding two 4-bit binary numbers.

In Mathematics, any two 4-bit binary numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are added as shown below-



Adding two 4-bit Numbers

Using ripple carry adder, this addition is carried out as shown by the following logic diagram-



4-bit Ripple Carry Adder

As shown-

- Ripple Carry Adder works in different stages.
- Each full adder takes the carry-in as input and produces carry-out and sum bit as output.
- The carry-out produced by a full adder serves as carry-in for its adjacent most significant full adder.
- When carry-in becomes available to the full adder, it activates the full adder.
- After full adder becomes activated, it comes into operation.

Working Of 4-bit Ripple Carry Adder-

Let-

- The two 4-bit numbers are 0101 ($A_3A_2A_1A_0$) and 1010 ($B_3B_2B_1B_0$).
- These numbers are to be added using a 4-bit ripple carry adder.

4-bit Ripple Carry Adder carries out the addition as explained in the following stages-

Stage-01:

- When C_{in} is fed as input to the full Adder A, it activates the full adder A.
- Then at full adder A, $A_0 = 1$, $B_0 = 0$, $C_{in} = 0$.

Full adder A computes the sum bit and carry bit as-

Calculation of S0–

$S0 = A0 \oplus B0 \oplus Cin$

$S0 = 1 \oplus 0 \oplus 0$

$S0 = 1$

Calculation of C0–

$C0 = A0B0 \oplus B0Cin \oplus CinA0$

$C0 = 1.0 \oplus 0.0 \oplus 0.1$

$C0 = 0 \oplus 0 \oplus 0$

$C0 = 0$

Stage-02:

- When C_0 is fed as input to the full adder B, it activates the full adder B.
- Then at full adder B, $A_1 = 0$, $B_1 = 1$, $C_0 = 0$.

Full adder B computes the sum bit and carry bit as-

Calculation of S1–

$S1 = A1 \oplus B1 \oplus C0$

$S1 = 0 \oplus 1 \oplus 0$

$S1 = 1$

Calculation of C1–

$C1 = A1B1 \oplus B1C0 \oplus C0A1$

$C1 = 0.1 \oplus 1.0 \oplus 0.0$

$C1 = 0 \oplus 0 \oplus 0$

$C1 = 0$

Stage-03:

- When C_1 is fed as input to the full adder C, it activates the full adder C.
- Then at full adder C, $A_2 = 1$, $B_2 = 0$, $C_1 = 0$.

Full adder C computes the sum bit and carry bit as-

Calculation of S2–

$S2 = A2 \oplus B2 \oplus C1$

$S2 = 1 \oplus 0 \oplus 0$

$S2 = 1$

Calculation of C2–

$C2 = A2B2 \oplus B2C1 \oplus C1A2$

$C2 = 1.0 \oplus 0.0 \oplus 0.1$

$C2 = 0 \oplus 0 \oplus 0$

$C2 = 0$

Stage-04:

- When C_2 is fed as input to the full adder D, it activates the full adder D.
- Then at full adder D, $A_3 = 0$, $B_3 = 1$, $C_2 = 0$.

Full adder D computes the sum bit and carry bit as-

Calculation of S_3–

$S3 = A3 \oplus B3 \oplus C2$

$S3 = 0 \oplus 1 \oplus 0$

$S3 = 1$

Calculation of C3–

$C3 = A3B3 \oplus B3C2 \oplus C2A3$

$C3 = 0.1 \oplus 1.0 \oplus 0.0$

$C3 = 0 \oplus 0 \oplus 0$

$C3 = 0$

Thus finally,

- Output Sum = $S_3S_2S_1S_0$ = 1111
- Output Carry = C_3 = 0

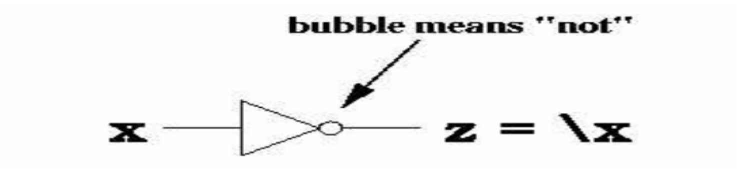
Disadvantages of Ripple Carry Adder-

- Ripple Carry Adder does not allow to use all the full adders simultaneously.
- Each full adder has to necessarily wait until the carry bit becomes available from its adjacent full adder.
- This increases the propagation time.
- Due to this reason, ripple carry adder becomes extremely slow.
- This is considered to be the biggest disadvantage of using ripple carry adder.

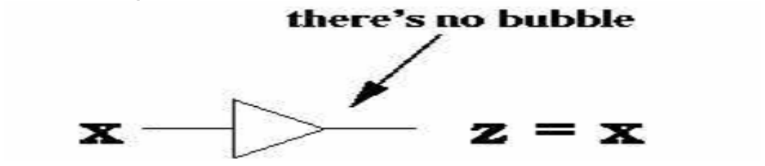
2.5.5 Tristate Buffer

Introduction

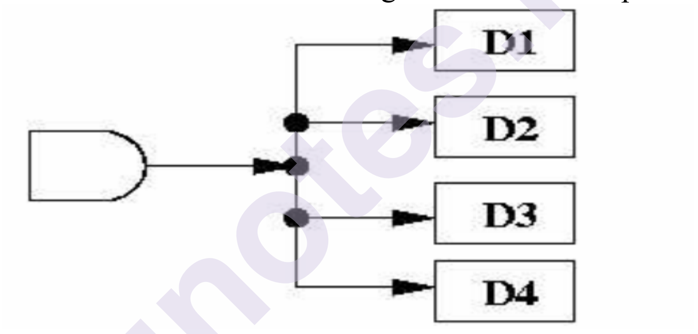
Before we talk about tri-state buffers, let's talk about an inverter. You can read about inverters in the notes about Logic Gates. However, we'll repeat it here for completeness. An inverter is called a NOT gate, and it looks like:



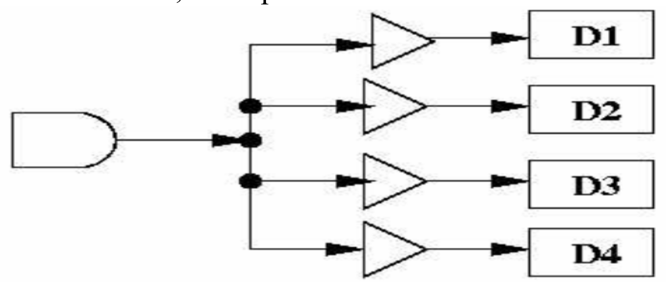
The inverter is a triangle, followed by a circle/bubble. That circle sometimes appears by itself, and means negation. What if we remove the circle? What kind of gate would we have? We'd have a buffer



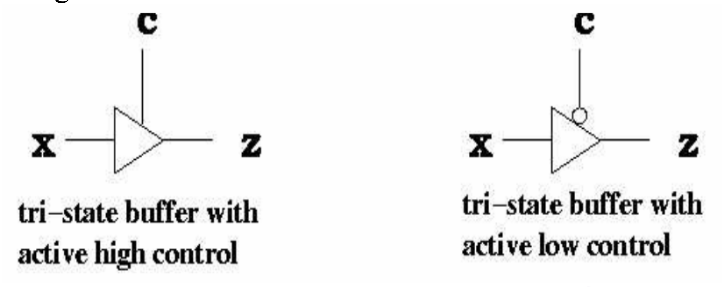
You might think that a buffer is useless. After all, the output is exactly the same as the input. What's the point of such a gate? The answer is a practical issue from real circuits. As you may know, logic gates process 0's and 1's. 0's and1's are really electric current at certain voltages. If there isn't enough current, it's hard to measure the voltage. The current can decrease if the fan out is large. Here's an example:



The "fan out" is the number of devices that an output is attached to. Thus, the AND gate above is attached to the inputs of four other devices. It has a fan out of 4.If the current coming out of the AND gate is i, then assuming each of the four devices gets equal current, then each device gets i / 4 of the current. However, if we put in a buffer:



Then the current can be "boosted" back to the original strength. Thus, a buffer (like all logic gates) is an active device. It requires additional inputs to power the gate, and provide it voltage and current. You might wonder "Do I really need to know this? Isn't this just EE stuff?" That's true, it is. The point of the discussion was to motivate the existence of a plain buffer. Tri-state buffer: It's a Valve A buffer's output is defined as z = x. Thus, if the input, x is 0, the output, z is 0. If the input, x is 1, the output, z is 1. It’s a common misconception to think that 0 is nothing, while 1 is something. In both cases, they’re something. If you read the discussion in What’s a Wire, you’ll see that a wire either transmits a 0, a 1, or "Z", which is really what’s nothing. It’s useful to think of a wire as a pipe and 0 as "red kool aid" and 1 as "green kool aid" and "Z" as "no kool aid". A tri-state buffer is a useful device that allows us to control when current passes through the device, and when it doesn’t. Here are two diagrams of the tri-state buffer.


c
x
z
tri–state buffer with active high control
c
x
z
tri–state buffer with active low control

A tri-state buffer has two inputs: a data input x and a control input c. The control input acts like a valve. When the control input is active, the output is the input. That is, it behaves just like a normal buffer. The "valve" is open. When the control input is not active, the output is "Z". The "valve" is open, and no electrical current flows through. Thus, even if x is 0 or 1, that value does not flow through. Here’s a truth table describing the behavior of a active-high tri-state buffer.

c	x	z
0	0	Z
0	1	Z
1	0	0
1	1	1

In this case, when the output is Z, that means it’s high impedance, neither 0, nor 1, i.e., no current. As usual, the condensed truth table is more enlightening.

c	z
0	Z
1	x

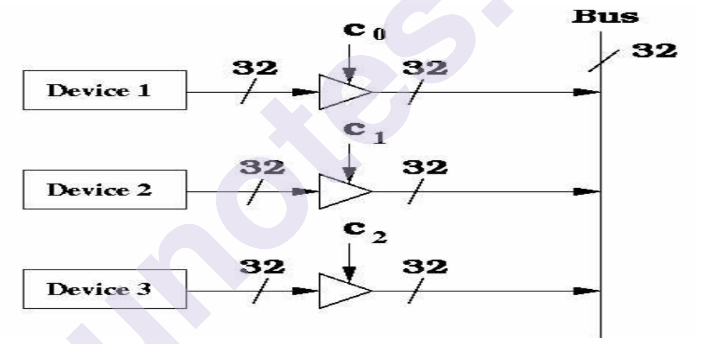
As you can see, when c = 1 the valve is open, and z = x. When c = 0 the valve is closed, and z = Z (e.g., high impedance/no current).Active-low tri-state buffers Some tri-state buffers are active low. In an active-low tri-state buffer, c = 0 turns open the valve, while c = 1turns it off. Here's the condensed truth table for an active-low tri-state buffer.

c	z
0	x
1	Z

As you can see, when c = 0 the valve is open, and z = x. When c = 1 the valve is closed, and z = Z (e.g., high impedance/no current). Thus, it has the opposite behavior of a tri-state buffer.

Why Tri-State Buffers?

We've had a long discussion about what a tri-state buffer is, but not about what such a device is good for.A common way for many devices to communicate with one another is on a bus, andthat a bus should only have one device writing to it, although it can have many devices reading from it. Since many devices always produce output (such as registers) and these devices are hooked to a bus, we need away to control what gets on the bus, and what doesn't. A tri state buffer is good for that. Here's an example:



2.5.6 Fan In and Fan Out

Fan In and Fan Out are the characteristics of digital IC. Digital IC's are complete functional network.

Fan in:

The term fan in is defined as maximum number of inputs that a logic gate can accept. If number of input exceeds, the output will be undefined or incorrect. It is specified by manufacturer and is provided in the data sheet. e.g. for 2 input OR gate fan in = 2

Fan-out:

The fan out term is defined as the maximum number of inputs (load) that can be connected to the output of a gate without degrading the normal operation. Fan Out is calculated from the amount of current available in the output of a gate and the amount of current needed in each input of the connecting gate. It is specified maximum load may cause a malfunction because the circuit will not be able to supply the demand power.

For Eg :If output of an X-OR gate is connected to 3 other external gate without degrading output performance of the IC then Fan Out =3.

2.6 MULTIPLEXER

Multiplexer means many to one. A multiplexer (MUX) is a combinational circuit which is often used when the information from many sources must be transmitted over long distances and it is less expensive to multiplex data onto a single wire for transmission.

Multiplexer can be considered as multi-position or rotary switch as shown in fig. 1. There are n – inputs and one output. The switch position is controlled by the selector lines. The select inputs decide which input is connected to the output.

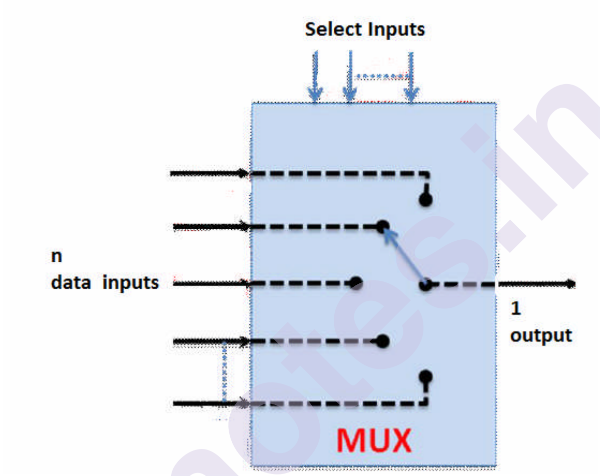
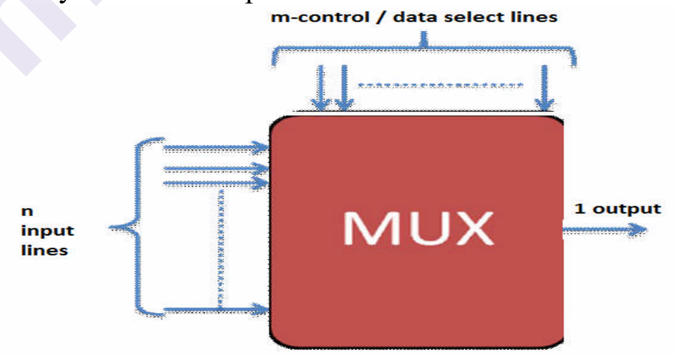


Figure 1 : Multiplexer as multi-position or rotary switch

The basic operation of multiplexer is controlled by a selector lines that routes one of many input signals to the output. Fig.1 shows the logic symbol of general symbol of multiplexer.



Multiplexer are also called as DATA Selector or router because it accepts several data inputs and allows only one of them to get through to the output at a time. The basic multiplexer has n input lines and single output line. It also has m – select or control lines. The relation between number of select lines and number of data inputs are

$$2^m=n$$

As multiplexer selects one out of many, it is often called as 2^m to 1 line converter.

Types of Multiplexer

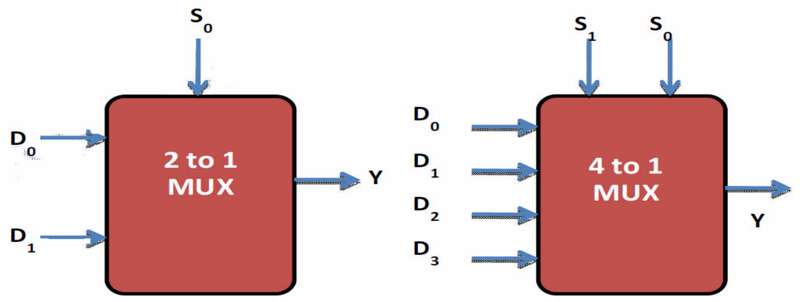


Figure-3(a) : Logic symbols of 2 to 1 and 4 to 1 multiplexers

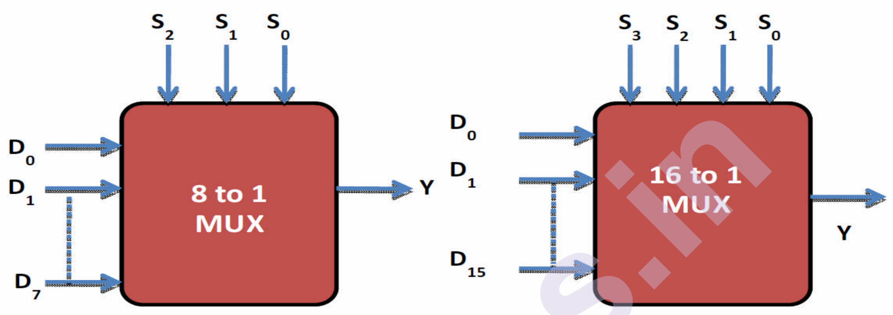


Figure-3(b) : Logic symbols of 8 to 1 and 16 to 1 multiplexers

Similarly we can extend the idea to 8 to 1 multiplexer and 16 to 1 multiplexer as shown in Fig. 3(b). For example 8 to 1 multiplexer with 8 inputs namely D0, D1,… D7, 3 select line S2, S1, So are the select line and Y as the single output. Similarly, the 16 to 1 multiplexer has 16 inputs D0, D1, … D15 , 4 select input S3,S2, S1 and S0 and Y as the output.

2.7 DE-MULTIPLEXER

De-multiplexer has a single input and n output lines. De-multiplexer can be visualized as reverse multi-position switch. The select lines permit input data from single line to be switched to any one of the many output lines as shown in fig.

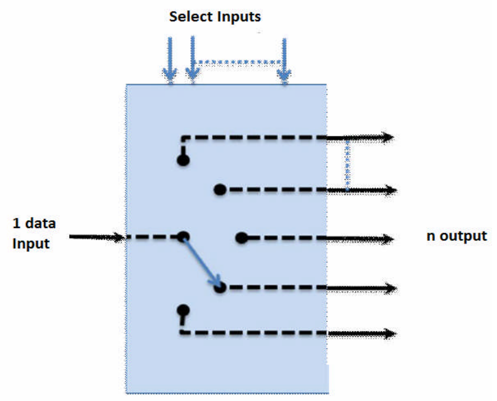


Fig : Multi-position switch as De-multiplexer

Thus the de-multiplexer takes one data input source and selectively distributes it to 1 of N output channels just like multi-position switch. It also has '**m**' select lines for selecting the desired output for the input data as shown in fig. The mathematical relation between select lines and '**n**' output are:

$$2^m = n$$

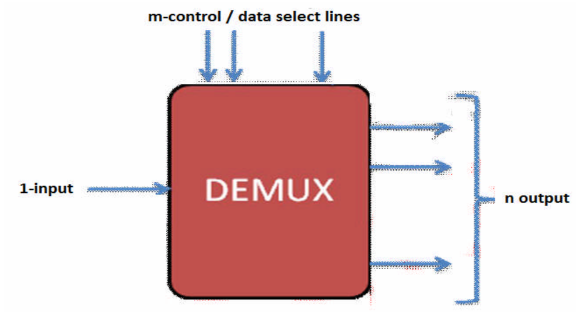


Figure: Logic symbol of basic de-multiplexer

As a de-multiplexer takes data from one input line and distributes over a 2m output line, hence it is often referred to as 1 to 2^m **line converter**. There are four basic types de-multiplexers: 1 to 2demultiplexer, 1 to 4 de-multiplexer, 1 to 8 de-multiplexer and 1 to 16 de-multiplexer as shown in fig. . Number of select lines decides this classification.

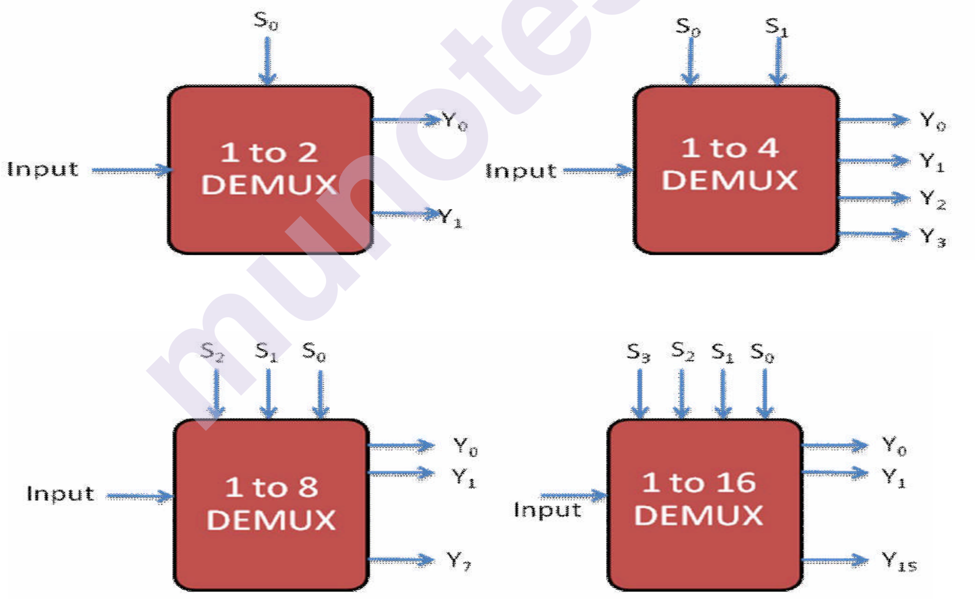
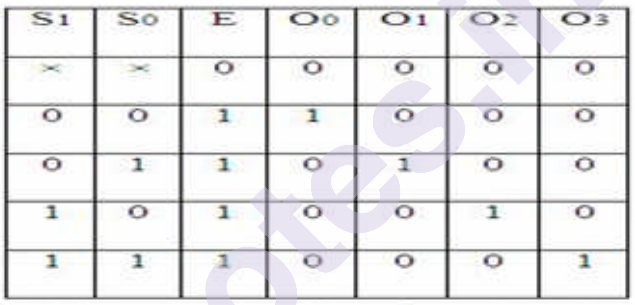


Fig : Types of Demux

2.8 DECODER

In digital electronics, a decoder can take the form of a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different e.g. n-to-2^n , binary-coded decimal decoders. Decoding is necessary in applications such as data multiplexing, 7 segment display and memory address decoding. The example decoder circuit would be an AND gate because the output of an AND gate is "High" (1) only when all its inputs are "High." Such output is called as "active High output". If instead of AND gate, the NAND gate is connected the output will be "Low" (0) only when all its inputs are "High". Such output is called as "active low output". A slightly more complex decoder would be the n-to-2n type binary decoders. These types of decoders are combinational circuits that convert binary information from 'n' coded inputs to a maximum of 2^n unique outputs. In case the 'n' bit coded information has unused bit combinations, the decoder may have less than 2^n outputs. 2-to-4 decoder, 3-to-8 decoder or 4-to-16 decoder are other examples. The input to a decoder is parallel binary number and it is used to detect the presence of a particular binary number at the input. The output indicates presence or absence of specific number at the decoder input.

Let us suppose that a logic network has 2 inputs S1 and S0. They will give rise to 4 states S1, S1', S0, S0'. The truth table for this decoder is shown below:



S1	S0	E	O0	O1	O2	O3
x	x	0	0	0	0	0
0	0	1	1	0	0	0
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1

Table 1: Truth Table of 2:4 decoder

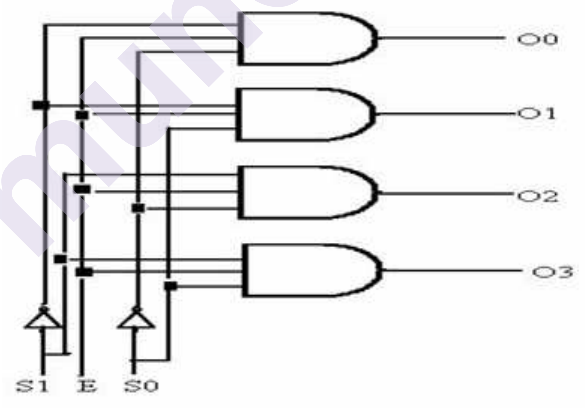


Fig 1: Logic Diagram of 2:4 decoder

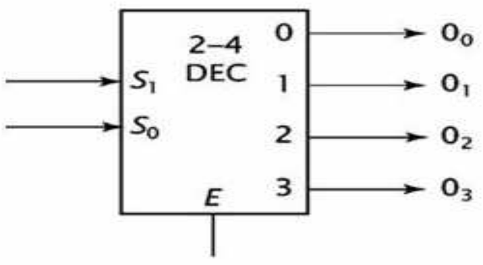


Fig 2: Representation of 2:4 decoder

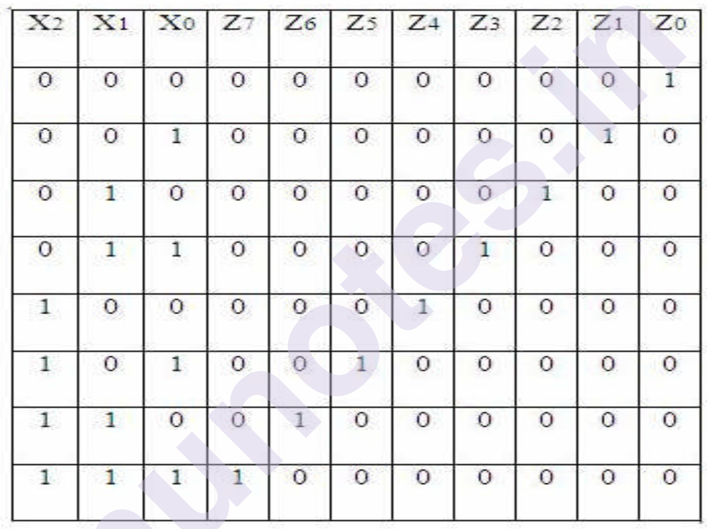
For any input combination only one of the outputs is low and all others are high. The low value at the output represents the state of the input.

Decoder expansion

We can combine two or more small decoders with enable inputs to form a larger decoder e.g. 3-to-8-line decoder constructed from two 2-to-4-line decoders. Decoder with enable input can function as de-multiplexer.

2.9 DECODER

It uses all AND gates, and therefore, the outputs are active- high. For active- low outputs, NAND gates are used. It has 3 input lines and 8 output lines. It is also called as binary to octal decoder it takes a 3-bit binary input code and activates one of the 8(octal) outputs corresponding to that code. The truth table is as follows:



X_2	X_1	X_0	Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Table 2: Truth Table of 3:8 decoder

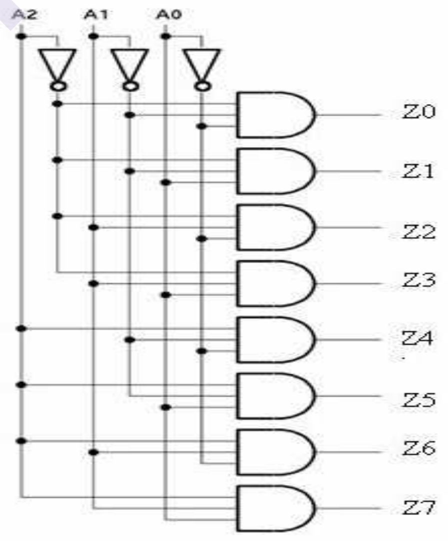


Fig 3: Logic Diagram of 3:8 decoder

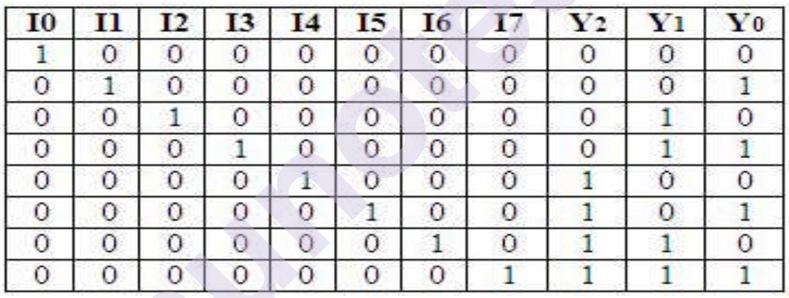
2.10 ENCODER

An encoder is a device, circuit, transducer, software program, algorithm or person that converts information from one format or code to another. The purpose of encoder is standardization, speed, secrecy, security, or saving space by shrinking size. Encoders are combinational logic circuits and they are exactly opposite of decoders. They accept one or more inputs and generate a multi bit output code.

Encoders perform exactly reverse operation than decoder. An encoder has M input and N output lines. Out of M input lines only one is activated at a time and produces equivalent code on output N lines. If a device output code has fewer bits than the input code has, the device is usually called an encoder.

Octal to binary encoder

Octal-to-Binary take 8 inputs and provides 3 outputs, thus doing the opposite of what the 3-to-8 decoder does. At any one time, only one input line has a value of 1. The figure below shows the truth table of an Octal-to-binary encoder.



I0	I1	I2	I3	I4	I5	I6	I7	Y2	Y1	Y0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Table 3: Truth Table of octal to binary encoder

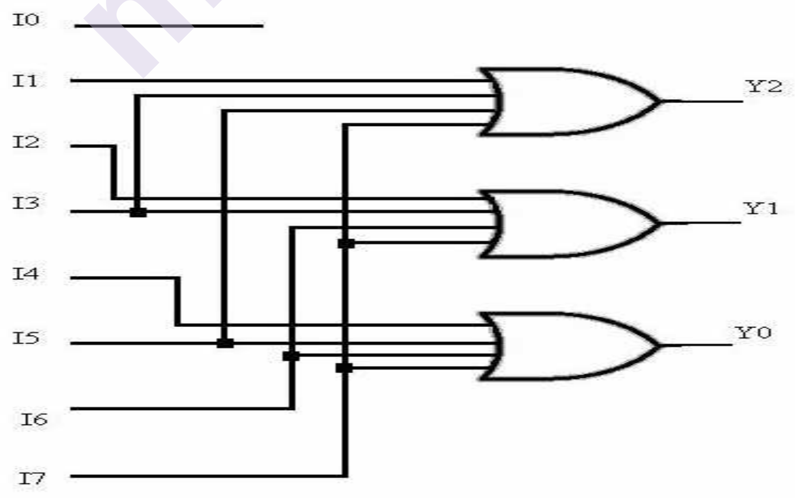


Fig 4: Logic Diagram of octal to binary encoder

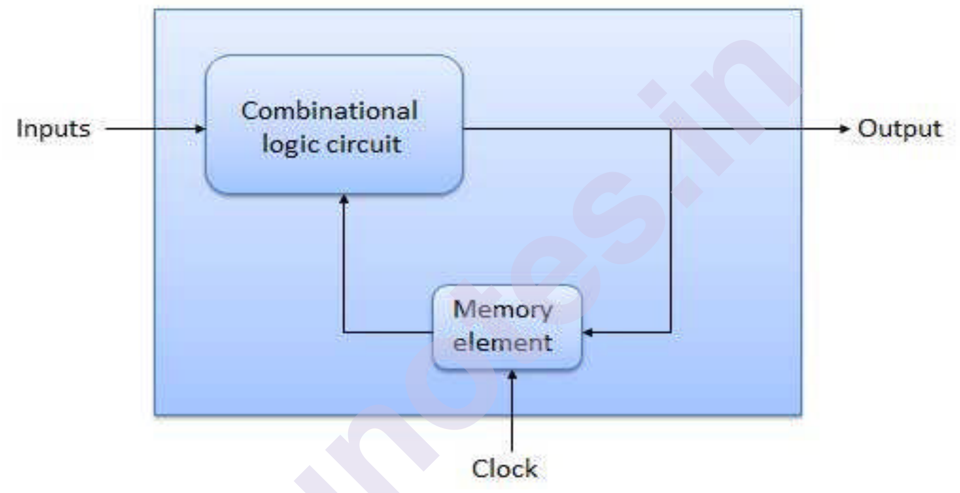
2.11 SEQUENTIAL CIRCUIT

Sequential circuit

A Sequential digital logic circuit is different from combinational logic circuits. In sequential circuit the output of the logic device is not only dependent on the present inputs to the device, but also on past inputs. In other words output of a sequential logic circuit depends on present input as well as present state of the circuit. So the sequential circuits have memory devices in order to store the past outputs. In fact sequential digital logic circuits are nothing but combinational circuit with memory. These types of digital logic circuits are designed using finite state machine.

Block diagram of sequential circuit

S



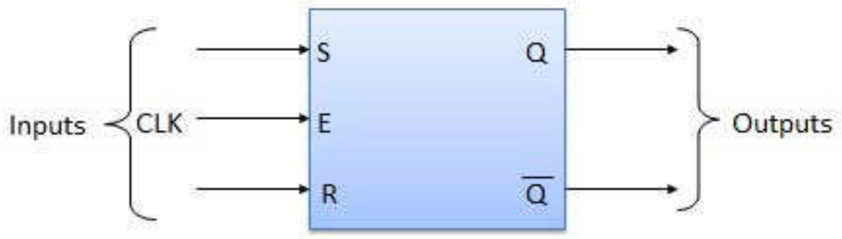
2.12 FLIP FLOP

Flip flop is a sequential circuit which generally samples its inputs and changes its outputs only at particular instants of time and not continuously. Flip flop is said to be edge sensitive or edge triggered rather than being level triggered like latches.

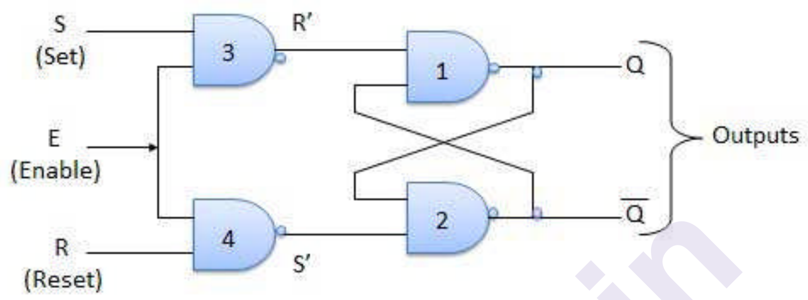
2.12.1 S-R Flip Flop

It is basically S-R latch using NAND gates with an additional **enable** input. It is also called as level triggered SR-FF. For this, circuit in output will take place if and only if the enable input (E) is made active. In short this circuit will operate as an S-R latch if E = 1 but there is no change in the output if E = 0.

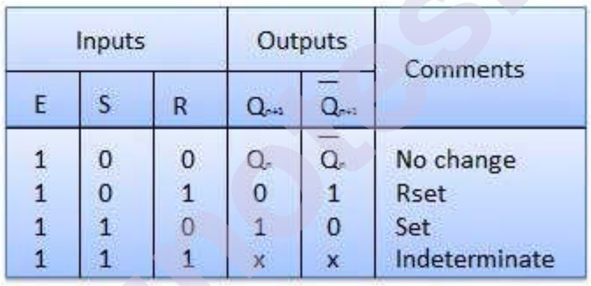
Block Diagram



Circuit Diagram



Truth Table



Inputs			Outputs		Comments
E	S	R	Q_{n+1}	$\overline{Q}_{n+1}$	
1	0	0	Q_n	$\overline{Q}_n$	No change
1	0	1	0	1	Rset
1	1	0	1	0	Set
1	1	1	x	x	Indeterminate

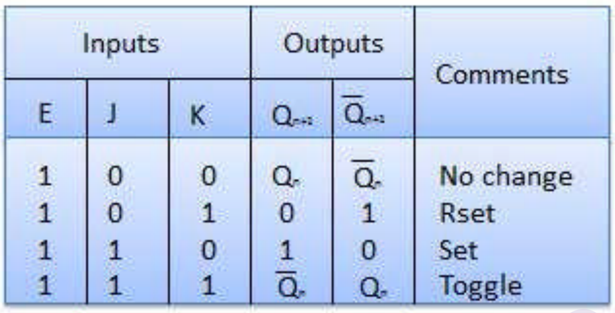
Operation

S.N.	Condition	Operation
1	**S = R = 0** : No change	If S = R = 0 then output of NAND gates 3 and 4 are forced to become 1. Hence R' and S' both will be equal to 1. Since S' and R' are the input of the basic S-R latch using NAND gates, there will be no change in the state of outputs.
2	**S = 0, R = 1, E = 1**	Since S = 0, output of NAND-3 i.e. R' = 1 and E = 1 the output of NAND-4 i.e. S' = 0. Hence Q_{n+1} = 0 and Q_{n+1} bar = 1. This is reset condition.
3	**S = 1, R = 0, E = 1**	Output of NAND-3 i.e. R' = 0 and output of NAND-4 i.e. S' = 1. Hence output of S-R NAND latch is Q_{n+1} = 1 and Q_{n+1} bar = 0. This is the reset condition.
4	**S = 1, R = 1, E = 1**	As S = 1, R = 1 and E = 1, the output of NAND gates 3 and 4 both are 0 i.e. S' = R' = 0. Hence the **Race** condition will occur in the basic NAND latch.

2.12.2 Master Slave JK Flip Flop

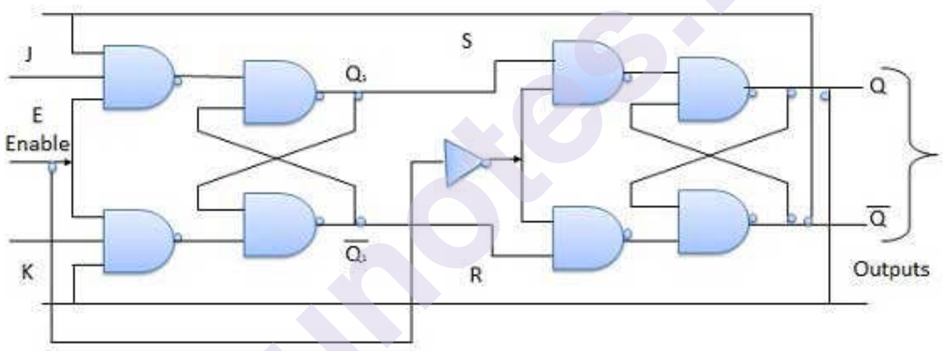
Master slave JK FF is a cascade of two S-R FF with feedback from the output of second to input of first. Master is a positive level triggered. But due to the presence of the inverter in the clock line, the slave will respond to the negative level. Hence when the clock = 1 (positive level) the master is active and the slave is inactive. Whereas when clock = 0 (low level) the slave is active and master is inactive.

Truth Table



Inputs			Outputs		Comments
E	J	K	Q_{n+1}	$\overline{Q}_{n+1}$	
1	0	0	Q_n	$\overline{Q}_n$	No change
1	0	1	0	1	Rset
1	1	0	1	0	Set
1	1	1	$\overline{Q}_n$	Q_n	Toggle

Circuit Diagram



Operation

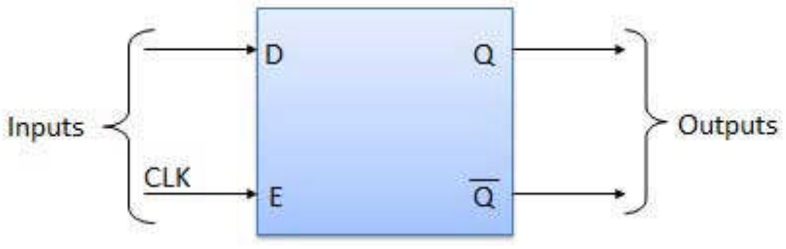
S.N.	Condition	Operation
1	**J = K = 0 (No change)**	When clock = 0, the slave becomes active and master is inactive. But since the S and R inputs have not changed, the slave outputs will also remain unchanged. Therefore outputs will not change if J = K =0.
2	**J = 0 and K = 1 (Reset)**	Clock = 1 − Master active, slave inactive. Therefore outputs of the master become Q_1 = 0 and Q_1 bar = 1. That means S = 0 and R =1. Clock = 0 − Slave active, master inactive. Therefore outputs of the slave become Q = 0 and Q bar = 1. Again clock = 1 − Master active, slave inactive. Therefore even with the changed outputs Q = 0 and Q bar = 1 fed back to master, its output will be Q1 = 0 and Q1 bar = 1. That means S = 0 and

		R = 1. Hence with clock = 0 and slave becoming active the outputs of slave will remain Q = 0 and Q bar = 1. Thus we get a stable output from the Master slave.
3	**J = 1 and K = 0 (Set)**	Clock = 1 − Master active, slave inactive. Therefore outputs of the master become Q_1 = 1 and Q_1 bar = 0. That means S = 1 and R =0. Clock = 0 − Slave active, master inactive. Therefore outputs of the slave become Q = 1 and Q bar = 0. Again clock = 1 − then it can be shown that the outputs of the slave are stabilized to Q = 1 and Q bar = 0.
4	**J = K = 1 (Toggle)**	Clock = 1 − Master active, slave inactive. Outputs of master will toggle. So S and R also will be inverted. Clock = 0 − Slave active, master inactive. Outputs of slave will toggle. These changed output are returned back to the master inputs. But since clock = 0, the master is still inactive. So it does not respond to these changed outputs. This avoids the multiple toggling which leads to the race around condition. The master slave flip flop will avoid the race around condition.

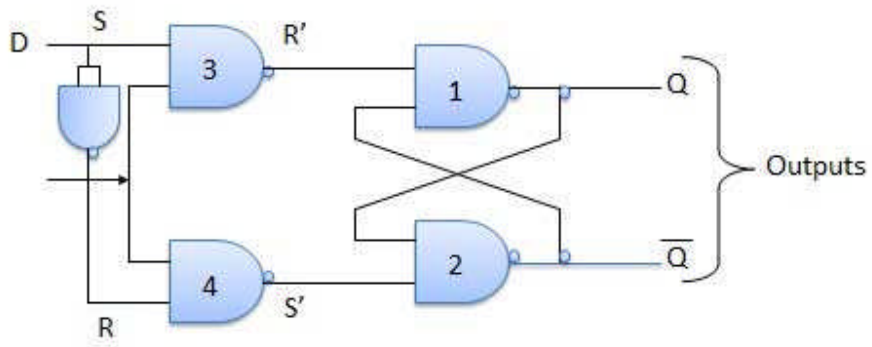
2.12.3 Delay Flip Flop / D Flip Flop

Delay Flip Flop or D Flip Flop is the simple gated S-R latch with a NAND inverter connected between S and R inputs. It has only one input. The input data is appearing at the output after some time. Due to this data delay between i/p and o/p, it is called delay flip flop. S and R will be the complements of each other due to NAND inverter. Hence S = R = 0 or S = R = 1, these input condition will never appear. This problem is avoided by SR = 00 and SR = 1 conditions.

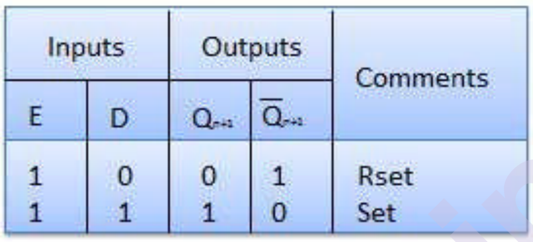
Block Diagram



Circuit Diagram



Truth Table



Inputs		Outputs		Comments
E	D	Q_{n+1}	$\overline{Q}_{n+1}$	
1	0	0	1	Rset
1	1	1	0	Set

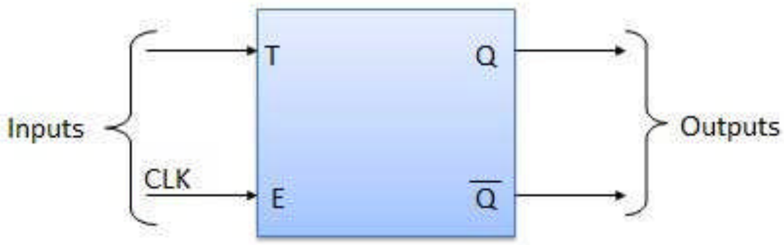
Operation

S.N.	Condition	Operation
1	**E = 0**	Latch is disabled. Hence no change in output.
2	**E = 1 and D = 0**	If E = 1 and D = 0 then S = 0 and R = 1. Hence irrespective of the present state, the next state is Q_{n+1} = 0 and Q_{n+1} bar = 1. This is the reset condition.
3	**E = 1 and D = 1**	If E = 1 and D = 1, then S = 1 and R = 0. This will set the latch and Q_{n+1} = 1 and Q_{n+1} bar = 0 irrespective of the present state.

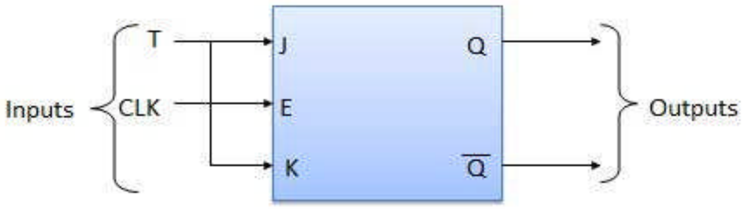
2.12.4 Toggle Flip Flop / T Flip Flop

Toggle flip flop is basically a JK flip flop with J and K terminals permanently connected together. It has only input denoted by **T** as shown in the Symbol Diagram. The symbol for positive edge triggered T flip flop is shown in the Block Diagram.

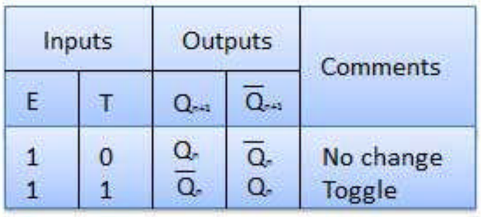
Symbol Diagram



Block Diagram



Truth Table



Inputs		Outputs		Comments
E	T	Q_{n+1}	$\overline{Q}_{n+1}$	
1 1	0 1	Q_n $\overline{Q}_n$	$\overline{Q}_n$ Q_n	No change Toggle

Operation

S.N.	Condition	Operation
1	**T = 0, J = K = 0**	The output Q and Q bar won't change
2	**T = 1, J = K = 1**	Output will toggle corresponding to every leading edge of clock signal.

2.13 STATE DIAGRAMS AND STATE TABLES

State table:

State diagrams are used to give an abstract description of the behavior of a system. This behavior is analyzed and represented by a series of events that can occur in one or more possible states. Hereby "each diagram usually represents objects of a single class and tracks the different states of its objects through the system.

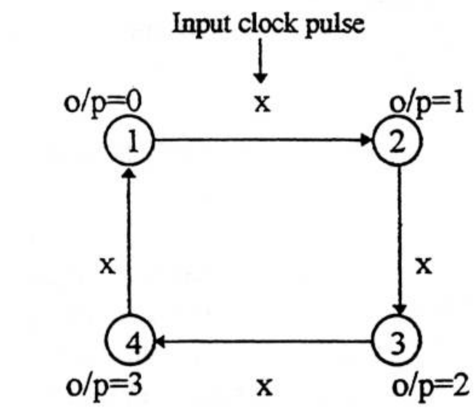
Fundamental to the synthesis of sequential circuits is the concept of internal states. At the start of a design the total number of states required is determined. This is achieved by drawing a state diagram, which shows the internal states and the transitions between them.

State diagram representation:

All states are stable (steady) and transitions from one state to another are caused by input (or clock) pulses. Each internal state is represented in the state diagram by a circle containing an arbitrary number or letter ; transitions are shown by arrows labeled with the particular input causing the change of state. In the case of pulse outputs the transition arrows are also labeled with the output associated with the input pulse. This will be made clear by examples given below.

As a simple example, consider a basic counter circuit that is driven by clock pulses (x) and counts in the following decimal sequence: 0,1,2,3,0,1,2,3,0,1,2, etc.

It follows that there are four unique states yielding the following state diagram:



The corresponding state table is derived directly from the above:

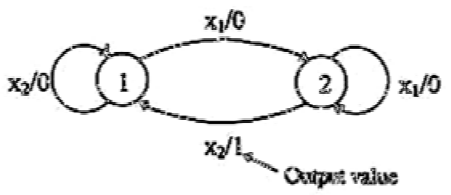
Present State	Next State (after application of clock pulse)	o/p Z_1Z_2
①	2	0 0
②	3	0 1
③	4	1 0
④	1	1 1

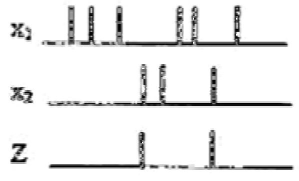
It follows that since there are 4 unique states then two flip-flops are required in the design. Each flip-flop output can take on the value 0 or 1, giving four possible combinations. It should be pointed out at the outset that once the state diagram and corresponding state table are derived from the given specification, the design procedure that follows is relatively straightforward.

State Diagrams and State Table Examples

Example

In a circuit having input pulses $\mathbf{x_1}$ and $\mathbf{x_2}$ the output z is said to be a pulse occurring with the first $\mathbf{x_2}$ pulse immediately following an $\mathbf{x_1}$ pulse.





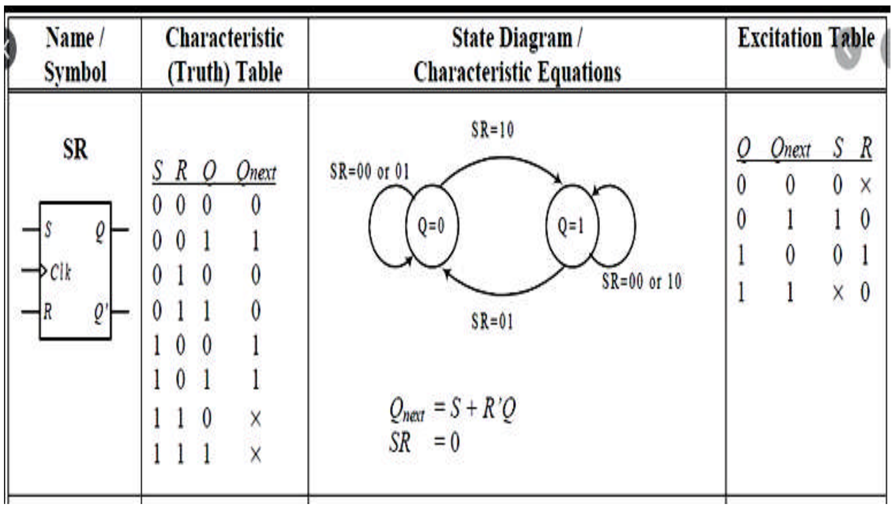
State Table:

Present State	Next State x_1	Next State x_2
①	2/0	1/0
②	2/0	1/1

Alternatively:

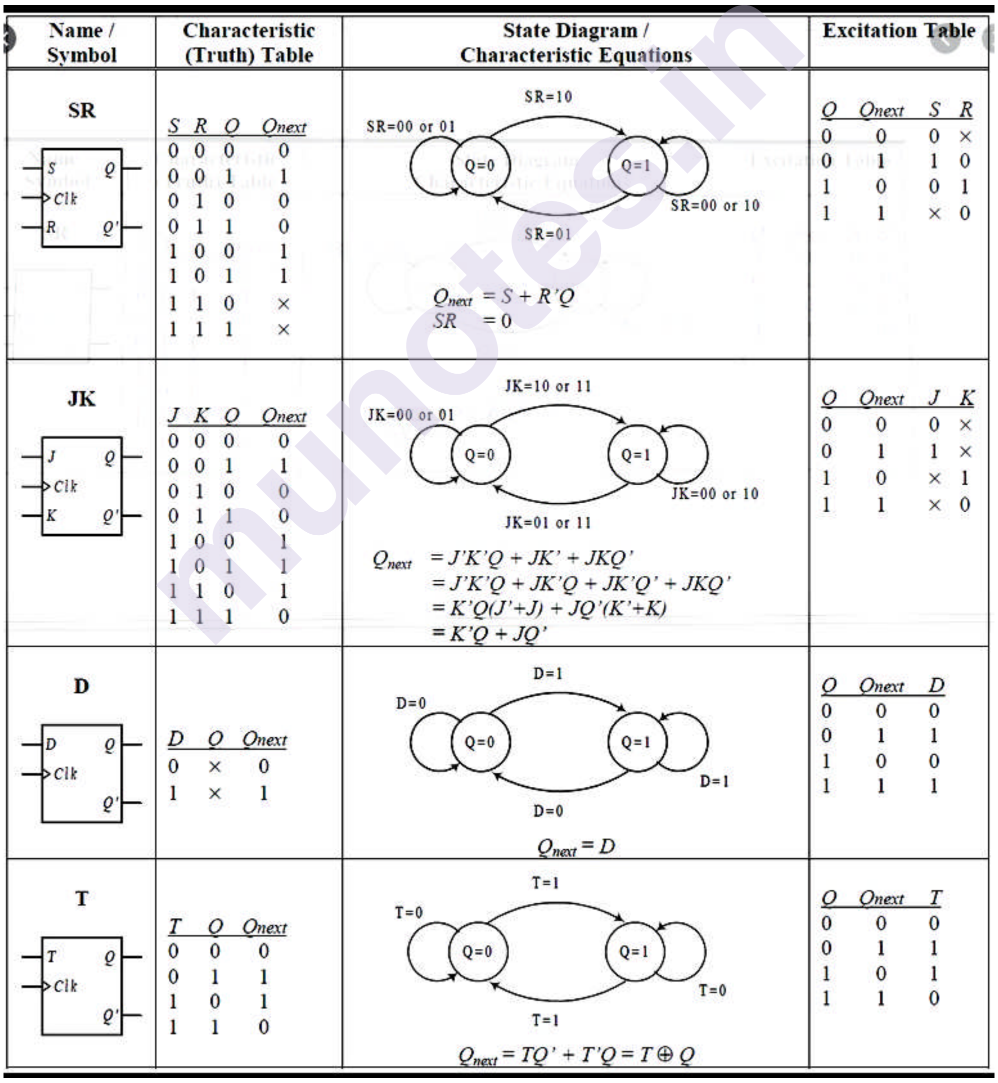
Present State	Next State x_1	Next State x_2
①	2	1
②	2	1,Z

Example state table and state diagram for SR flip flop



Name / Symbol	Characteristic (Truth) Table	State Diagram / Characteristic Equations	Excitation Table
SR	S R Q Qnext 0 0 0 0 0 0 1 1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 1 1 1 1 0 × 1 1 1 ×	$Q_{next} = S + R'Q$ $SR = 0$	Q Qnext S R 0 0 0 × 0 1 1 0 1 0 0 1 1 1 × 0

S
Q
Clk
R
Q'
SR=10
SR=00 or 01
Q=0
Q=1
SR=00 or 10
SR=01



Name / Symbol	Characteristic (Truth) Table	State Diagram / Characteristic Equations	Excitation Table
SR	S R Q Qnext 0 0 0 0 0 0 1 1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 1 1 1 1 0 × 1 1 1 ×	$Q_{next} = S + R'Q$ $SR = 0$	Q Qnext S R 0 0 0 × 0 1 1 0 1 0 0 1 1 1 × 0
JK	J K Q Qnext 0 0 0 0 0 0 1 1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1 0	$Q_{next} = J'K'Q + JK' + JKQ'$ $= J'K'Q + JK'Q + JK'Q' + JKQ'$ $= K'Q(J'+J) + JQ'(K'+K)$ $= K'Q + JQ'$	Q Qnext J K 0 0 0 × 0 1 1 × 1 0 × 1 1 1 × 0
D	D Q Qnext 0 × 0 1 × 1	$Q_{next} = D$	Q Qnext D 0 0 0 0 1 1 1 0 0 1 1 1
T	T Q Qnext 0 0 0 0 1 1 1 0 1 1 1 0	$Q_{next} = TQ' + T'Q = T \oplus Q$	Q Qnext T 0 0 0 0 1 1 1 0 1 1 1 0

2.14 LET US SUM UP

Thus, we have studied basic concepts of logic gates, truth table and logic circuits functions and categories of combinational circuit and Sequential circuit, K-map and minimization of k-map. Also you learned what is multiplexer and demultiplexer as well.

2.15 LIST OF REFERENCES

- Carl Hamacher et al., Computer Organization and Embedded Systems, 6 ed., McGraw-Hill 2012
- Patterson and Hennessy, Computer Organization and Design, Morgan Kaufmann, ARM Edition, 2011
- R P Jain, Modern Digital Electronics, Tata McGraw Hill Education Pvt. Ltd. , 4th Edition, 2010

2.16 UNIT END EXERCISES

1) Explain the concept of universal gate.
2) Design and explain full adder circuit
3) Compare multiplexer and De-multiplexer
4) Draw the circuit for half-adder using k-map reduction technique.
5) Explain tristate buffer.

❖❖❖❖

3

INSTRUCTION SET ARCHITECTURES

Unit Structure

3.0 OBJECTIVES

In this chapter you will learn about:

- Machine instructions and program execution
- Addressing methods for accessing register and memory operands
- Assembly language for representing machine instructions, data, and programs
- Stacks and subroutines

3.1 INTRODUCTION

Programs and the data that processor operate are held in the main memory of the computer during execution, and the data with high storage requirement is stored in the secondary memories such as floppy disk, etc. In this chapter, we discuss how this vital part of the computer operates.

The dominant architecture in the PC market, was the Intel IA-32, belongs to the complex instruction set computing (CISC) design. The CISC instruction set architecture is too complex in nature and developers develop it very complex so to use with higher level languages which supports complex data structures

For variety of reasons, in the early 1980's designers started looking at simple Instruction set architectures, as these ISAs tend to produce instruction sets with less number of instructions known as Reduced Instruction Set Computer(RISC)

3.2 MEMORY LOCATIONS AND ADDRESSES

The memory consists of many millions of storage *cells*, each of which can store a *bit* of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For this purpose, the memory is organized so that a group of *n* bits can be stored or retrieved in a single, basic operation. Each group of *n* bits is referred to as a *word* of information, and *n* is called the *word length*. The memory of a computer can be schematically represented as a collection of words, as shown in Figure 3.1.

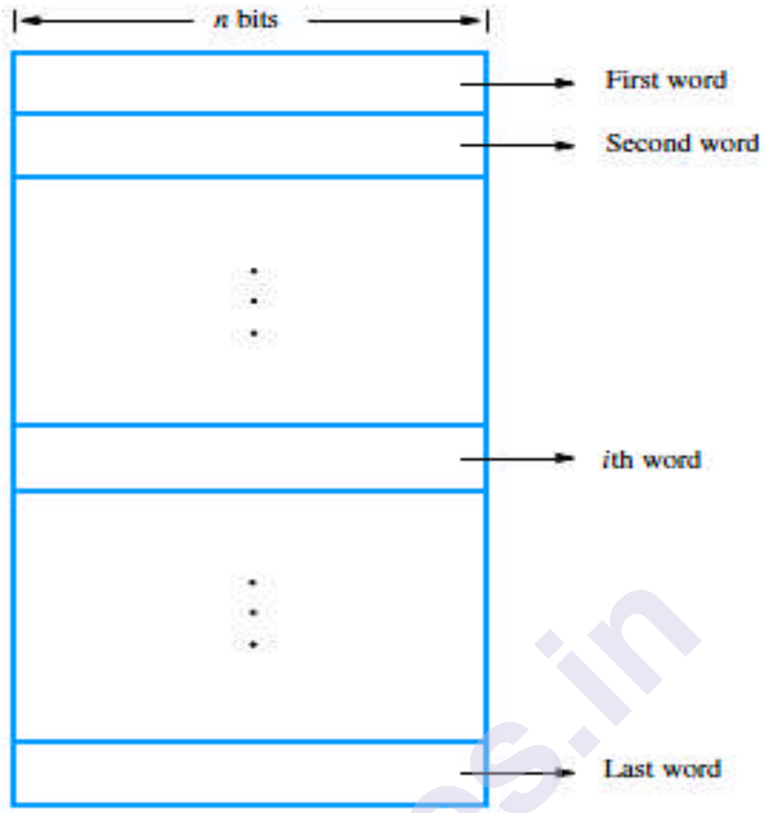


Fig 3.1 Memory words

Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits, as shown in Figure 3.2. A unit of 8 bits is called a *byte*. Machine instructions may require one or more words for their representation. We will discuss how machine instructions are encoded into memory words in a later section, after we have described instructions at the assembly-language level.

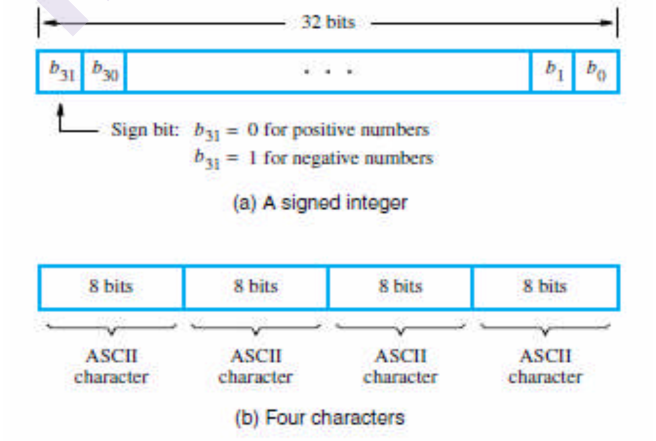


Fig 3.2 Examples of encoded information in a 32- bit word

Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or *addresses* for each location. It is customary to use numbers from 0 to $2k - 1$, for some suitable value of *k*, as the addresses of successive locations in the memory. Thus, the memory can have up to $2k$ addressable locations. The $2k$ addresses constitute the *address space* of the computer.

For example, a 24-bit address generates an address space of 2^{24} (16,777,216) locations. This number is usually written as 16M (16 mega), where 1M is the number 2^{20} (1,048,576). A 32-bit address creates an address space of 2^{32} or 4G (4 giga) locations, where 1G is 2^{30}. Other notational conventions that are commonly used are K (kilo) for the number 2^{10} (1,024), and T (tera) for thenumber 2^{12}.

3.2.1 Byte Addressability

There are three basic information quantities to deal with: bit, byte, and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign distinct addresses to individual bit locations in the memory. The most practical assignment is to have successive addresses refer to successive byte locations in the memory. This is the assignment used in most modern computers. The term *byte-addressable memory* is used for this assignment. Byte locations have addresses 0, 1, 2, ...Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8, . . . ,with each word consisting of four bytes.

3.2.2 Big-Endian and Little-Endian Assignments

There are two ways that byte addresses can be assigned across words, as shown in Figure 3.3. The name *big-endian* is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name *little-endian* is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word.

The words "more significant" and "less significant" are used in relation to the weights (powers of 2) assigned to bits when the word represents a number. Both little-endian and big-endian assignments are used in commercial machines. In both cases, byte addresses 0, 4, 8, . . . ,are taken as the addresses of successive words in the memory of a computer with a 32-bit word length.

These are the addresses used when accessing the memory to store or retrieve a word. In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The most common convention, and the one we will use in this book, is shown in Figure 3.2*a*. It is the most natural ordering for the encoding of numerical data. The same ordering is also used for labeling bits within a byte, that is, *b*7, *b*6, . . . , *b*0, from left to right

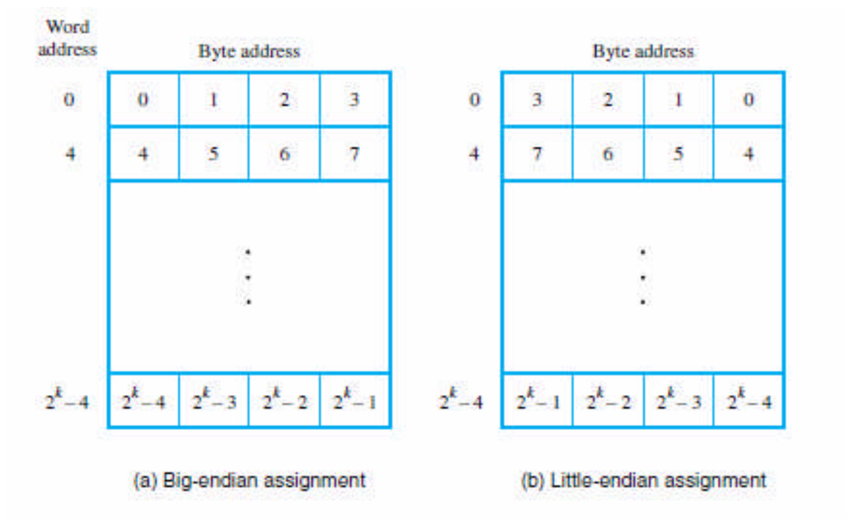

Word address
Byte address
0
4
2^k-4
0 1 2 3
4 5 6 7
2^k-4 2^k-3 2^k-2 2^k-1
(a) Big-endian assignment
Byte address
0
4
2^k-4
3 2 1 0
7 6 5 4
2^k-1 2^k-2 2^k-3 2^k-4
(b) Little-endian assignment

Fig 3.3 Byte and word addressing

3.2.3 Word Alignment

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, . . . ,as shown in Figure 3.3. We say that the word locations have *aligned* addresses if they begin at a byte address that is a multiple of the number of bytes in a word. For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4, . . . ,and for a word length of 64 (23 bytes), aligned words begin at byte addresses 0, 8, 16, . .

3.3 MEMORY OPERATIONS

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, *Read* and *Write*.

The Read operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The Write operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location. To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

3.4 INSTRUCTIONS AND INSTRUCTION SEQUENCING

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

3.4.1 Register Transfer Notation

The transfer of information from one location in a computer to another it is an possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify such locations symbolically with convenient names.

For example, names that represent the addresses of memory locations may be LOC, PLACE, A, or VAR2. Predefined names for the processor registers may be R0 or R5. Registers in the I/O subsystem may be identified by names such as DATAIN or UTSTATUS. To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name. Thus, the expression means that the contents of memory location LOC are transferred into processor register R2.

$$R2 \leftarrow [LOC]$$

As another example, consider the operation that adds the contents of registers R2 and R3, and places their sum into register R4. This action is indicated as

$$R4 \leftarrow [R2] + [R3]$$

This type of notation is known as *Register Transfer Notation* (RTN). Note that the right hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

In computer jargon, the words “transfer” and “move” are commonly used to mean “copy.” Transferring data from a *source* location A to a *destination* location B means that the contents of location A are read and then written into location B. In this operation, only the contents of the destination will change. The contents of the source will stay the same.

3.4.2 Assembly-Language Notation

Another type of notation to represent machine instructions and programs is the Assembly-language notation.

For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R2, is specified by the statement

Load R2, LOC

The contents of LOC are unchanged by the execution of this instruction, but the old contentsof register R2 are overwritten. The name Load is appropriate for this instruction, because the contents read from a memory location are *loaded* into a processor register.

The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

Add R4, R2, R3

In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

An *instruction* specifies an operation to be performed and the operands involved. In the above examples, we used the English words Load and Add to denote the required operations. In the assembly-language instructions of actual (commercial) processors, such operations are defined by using *mnemonics*, which are typically abbreviations of the words describing the operations.

For example, the operation Load may be written as LD, while the operation Store, which transfers a word from a processor register to the memory, may be written as STR or ST. Assembly languages for different processors often use different mnemonics for a given operation. To avoid the need for details of a particular assembly language at this early stage, we will continue the presentation in this chapter by using English words rather than processor-specific mnemonics.

3.4.3 RISC and CISC Instruction Sets

One of the most important characteristics that distinguish different computers is the nature of their instructions. There are two fundamentally different approaches in the design of instruction sets for modern computers. One popular approach is based on the premise that higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers. This approach is conducive to an implementation of the processing unit in which the various operations needed to process a sequence of instructions are performed in "pipelined" fashion to overlap activity and reduce total execution time of a program.

The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer. Such computers are called *Reduced Instruction Set Computers* (RISC). An alternative to the RISC approach is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations.

This approach was prevalent prior to the introduction of the RISC approach in the 1970s. Although the use of complex instructions was not originally identified by any particular label, computers based on this idea have been subsequently called *Complex Instruction Set Computers* (CISC).

3.4.4 Introduction to RISC Instruction Sets

Two key characteristics of RISC instruction sets are:

- Each instruction fits in a single word.
- A *load/store architecture* is used, in which
  - – Memory operands are accessed only using Load and Store instructions.
  - – All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.

At the start of execution of a program, all instructions and data used in the program are stored in the memory of a computer. Processor registers do not contain valid operands at that time. If operands are expected to be in processor registers before they can be used by an instruction, then it is necessary to first bring these operands into the registers. This task is done by Load instructions which copy the contents of a memory location into a processor register. Load instructions are of the form

Load destination, source

or more specifically

Load processor_register, memory_location

The memory location can be specified in several ways. The term *addressing modes* is used to refer to the different ways in which this may be accomplished. Let us now consider a typical arithmetic operation. The operation of adding two numbers is a fundamental capability in any computer. The statement in a high-level language program instructs the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C.

$$\mathbf{C = A + B}$$

When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. For simplicity, we will refer to the addresses of these locations as A, B, and C, respectively. The contents of these locations represent the values of the three variables.

$$C \leftarrow [A] + [B]$$

Hence, the above high-level language statement requires the action to take place in the computer. To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

The required action can be accomplished by a sequence of simple machine instructions. We choose to use registers R2, R3, and R4 to perform the task with four instructions:

```
Load     R2, A
Load     R3, B
Add      R4, R2, R3
Store    R4, C
```

It Add is a *three-operand*, or a *three-address*, instruction of the form

```
Add    destination, source1, source2
```

The Store instruction is of the form

```
Store    source, destination
```

where the source is a processor register and the destination is a memory location. Observe that in the Store instruction the source and destination are specified in the reverse order from the Load instruction; this is a commonly used convention.

Note that we can accomplish the desired addition by using only two registers, R2 and R3, if one of the source registers is also used as the destination for the result. In this case the addition would be performed as

```
Add    R3, R2, R3
```

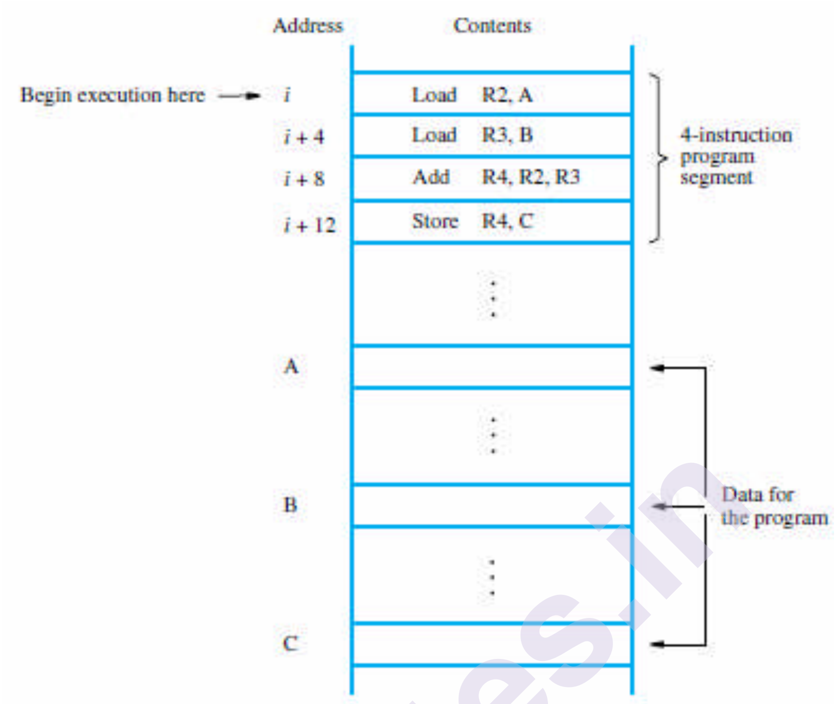
and the last instruction would become

```
Store    R3, C
```

3.4.5 Instruction Execution and Straight-Line Sequencing

In the preceding subsection, we used the task C = A + B, implemented as C←[A] + [B], as an example. Figure 3.4 shows a possible program segment for this task as it appears in the memory of a computer. We assume that the word length is 32 bits and the memory is byte-addressable. The four instructions of the program are in successive word locations, starting at location *i*. Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses *i* + 4, *i* + 8, and *i* +

12. For simplicity, we assume that a desired memory address can be directly specified in Load and Store instructions, although this is not possible if a full 32-bit address is involved.



A program for C←[A] + [B]

Let us consider how this program is executed. The processor contains a register called the *program counter* (PC), which holds the address of the next instruction to be executed. To begin executing a program, the address of its first instruction (*i* in our example) must be placed into the PC.

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, the PC is incremented by 4 to point to the next instruction.

Thus, after the Store instruction at location *i* + 12 is executed, the PC contains the value *i* + 16, which is the address of the first instruction of the next program segment. Executing a given instruction is a two-phase procedure. In the first phase, called *instruction fetch*, the instruction is fetched from the memory location whose address is in the PC.

This instruction is placed in the *instruction register* (IR) in the processor. At the start of the second phase, called *instruction execute*, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor.

This involves a small number of steps such as fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

3.4.6 Branching

Consider the task of adding a list of *n* numbers. The program outlined in Figure 3.5 is a generalization of the program in Figure 3.4. The addresses of the memory locations containing the *n* numbers are symbolically given as NUM1, NUM2, . . . ,NUM*n*, and separate Load and Add instructions are used to add each number to the contents of register R2. After all the numbers have been added, the result is placed in memory location SUM.

Instead of using a long list of Load and Add instructions, as in Figure 3.5, it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum. To add all numbers, the loop has to be executed as many times as there are numbers in the list. Figure 3.6 shows the structure of the desired program. The body of the loop is a straight-line sequence of instructions executed repeatedly. It starts at location LOOP and ends at the instruction Branch_if_[R2]>0. During each pass through this loop, the address of the next list entry is determined, and that entry is loaded into R5 and added to R3. The address of an operand can be specified in various ways, as will be described in Section 2.4. For now, we concentrate on how to create and control a program loop.

Assume that the number of entries in the list, *n*, is stored in memory location N, as shown. Register R2 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R2 at the beginning of the program.

```
Subtract    R2, R2, #1
```

Then, within the body of the loop, the instruction reduces the contents of R2 by 1 each time through the loop. (We will explain the significance of the number sign '#' in Section 3.4.1.) Execution of the loop is repeated as long as the contents of R2 are greater than zero.

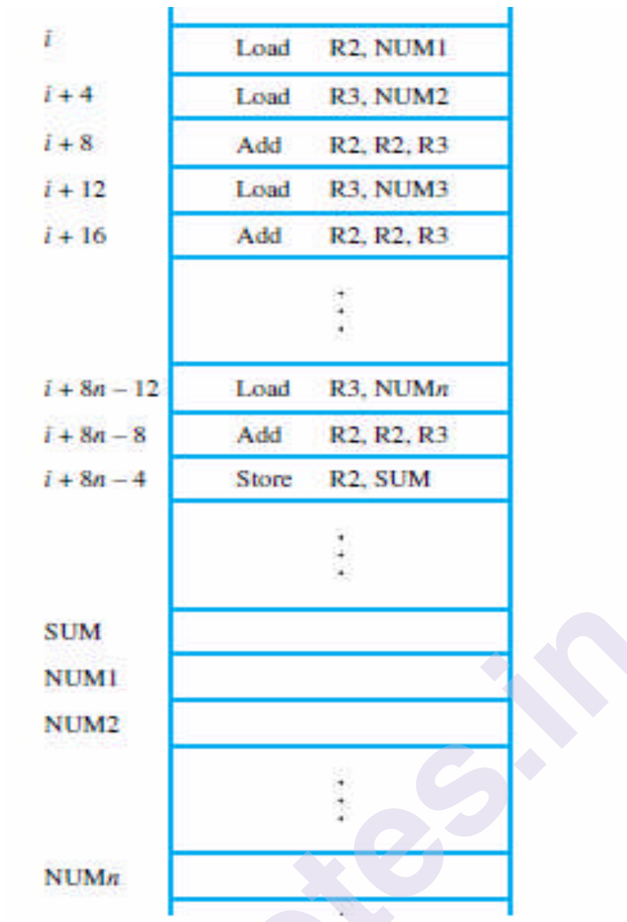


fig. 3.5 A program for odding n numbers

We now introduce *branch* instructions. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the *branch target*, instead of the instruction at the location that follows the branch instruction in sequential address order. A *conditional branch* instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

Branch_if_[R2]>0 LOOP

In the program in Figure 3.6, the instruction is a conditional branch instruction that causes a branch to location LOOP if the contents of register R2 are greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R3. At the end of the *n*th pass through the loop, the Subtract instruction produces a value of zero in R2, and, hence, branching does not occur. Instead, the Store instruction is fetched and executed. It moves the final result from R3 into memory location SUM.

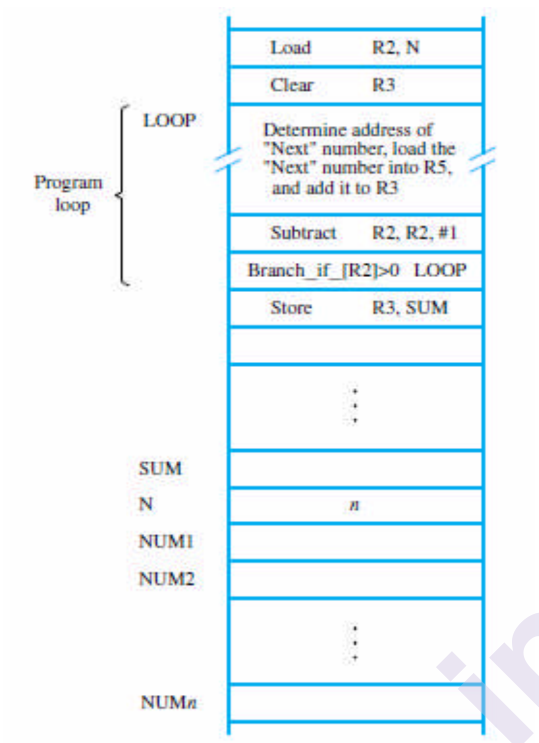


Fig. 3.6 Using a loop to odd n numbers

The capability to test conditions and subsequently choose one of a set of alternative ways to continue computation has many more applications than just loop control. Such a capability is found in the instruction sets of all computers and is fundamental to the programming of most nontrivial tasks.

One way of implementing conditional branch instructions is to compare the contents of two registers and then branch to the target instruction if the comparison meets the specified requirement. For example, the instruction that implements the action

Branch_if_[R4]>[R5] LOOP

may be written in generic assembly language as

Branch_greater_than R4, R5, LOOP

or using an actual mnemonic as

BGT R4, R5, LOOP

It compares the contents of registers R4 and R5, without changing the contents of either register. Then, it causes a branch to LOOPif the contents of R4 are greater than the contents of R5.

3.4.7 Generating Memory Addresses

The purpose of the instruction block starting at LOOP is to add successive numbers from the list during each pass through the loop. Hence, the Load instruction in that block must refer to a different address during each pass. How are the addresses specified? The memory operand address cannot be given directly in a single Load instruction in the loop. Otherwise, it would need to be modified on each pass through the loop. As one possibility, suppose that a processor register, R*i*, is used to hold the memory address of an operand. If it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop, it can provide the needed capability.

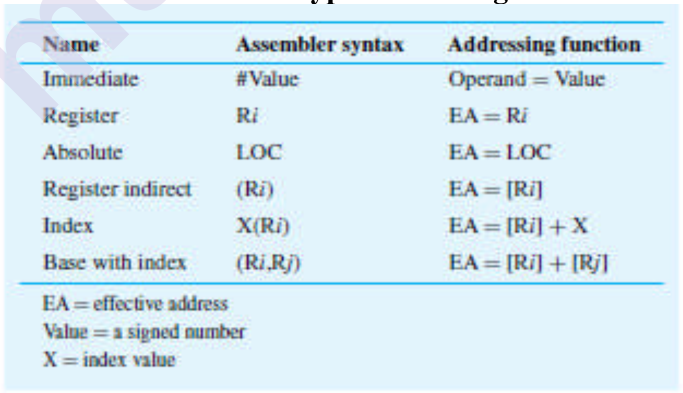
This situation, and many others like it, give rise to the need for flexible ways to specify the address of an operand. The instruction set of a computer typically provides a number of such methods, called *addressing modes*. While the details differ from one computer to another, the underlying concepts are the same.

3.5 ADDRESSING MODES

In general, a program operates on data that reside in the computer's memory. These data can be organized in a variety of ways that reflect the nature of the information and how it is used. Programmers use *data structures* such as lists and arrays for organizing the data used in computations.

Programs are normally written in a high-level language, which enables the programmer to conveniently describe the operations to be performed on various data structures. When translating a high-level language program into assembly language, the compiler generates appropriate sequences of low-level instructions that implement the desired operations. The different ways for specifying the locations of instruction operands are known as *addressing modes*.

Table 3.1 RISC type addressing



Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute	LOC	EA = LOC
Register indirect	(R*i*)	EA = [R*i*]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]

EA = effective address
Value = a signed number
X = index value

modes

In this section we present the basic addressing modes found in RISC-style processors. A summary is provided in Table 3.1, which also includes the assembler syntax we will use for each mode.

3.5.1 Implementation of Variables and Constants

Variables are found in almost every computer program. In assembly language, a variable is represented by allocating a register or a memory location to hold its value. This value can be changed as needed using appropriate instructions. The program in Figure 3.5 uses only two addressing modes to access variables. We access an operand by specifying the name of the register or the address of the memory location where the operand is located. The precise definitions of these two modes are:

Register mode—The operand is the contents of a processor register; the name of the register is given in the instruction.

Absolute mode—The operand is in a memory location; the address of this location is given explicitly in the instruction.

The instruction

```
Add R4, R2, R3
```

uses the Register mode for all three operands. Registers R2 and R3 hold the two source operands, while R4 is the destination. The Absolute mode can represent global variables in a program. A declaration such as

```
Integer NUM1, NUM2, SUM;
```

in a high-level language program will cause the compiler to allocate a memory location to each of the variables NUM1, NUM2, and SUM. Whenever they are referenced later in the program, the compiler can generate assembly-language instructions that use the Absolute mode to access these variables.

The Absolute mode is used in the instruction

```
Load R2, NUM1
```

which loads the value in the memory location NUM1 into register R2. Constants representing data or addresses are also found in almost every computer program. Such constants can be represented in assembly language using the Immediate addressing mode.

Immediate mode—The operand is given explicitly in the instruction.

For example, the instruction

```
Add R4, R6, 200 immediate
```

adds the value 200 to the contents of register R6, and places the result into register R4. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the number sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form

```
Add R4, R6, #200
```

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an *effective address* (EA) can be derived by the processor when the instruction is executed. The effective address is then used to access the operand.

3.5.2 Indirection and Pointers

The program in Figure 3.6 requires a capability for modifying the address of the memory operand during each pass through the loop. A good way to provide this capability is to use a processor register to hold the address of the operand. The contents of the register are then changed (incremented) during each pass to provide the address of the next number in the list that has to be accessed. The register acts as a *pointer* to the list, and we say that an item in the list is accessed *indirectly* by using the address in the register. The desired capability is provided by the indirect addressing mode.

Indirect mode—The effective address of the operand is the contents of a register that is specified in the instruction.

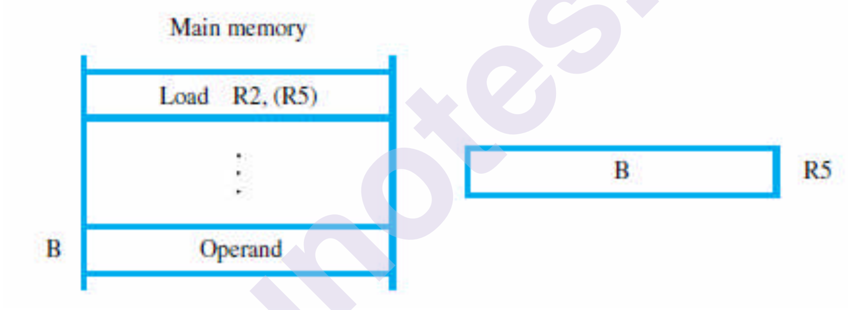


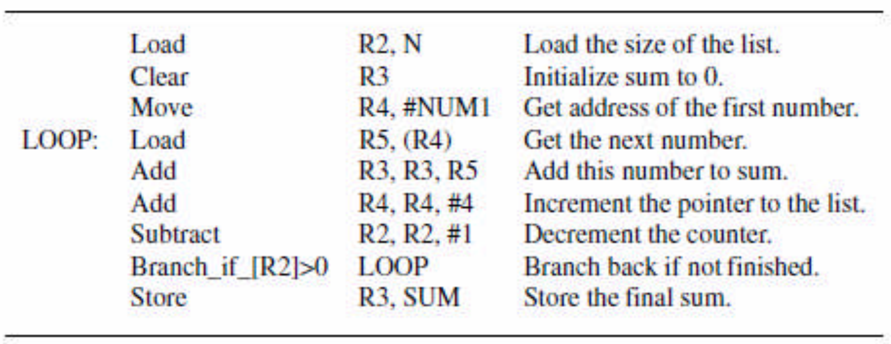
Fig 3.7 Register indirect addressing

To execute the Load instruction in Figure 3.7, the processor uses the value B, which is in register R5, as the effective address of the operand. It requests a Read operation to fetch the contents of location B in the memory. The value from the memory is the desired operand, which the processor loads into register R2. Indirect addressing through a memory location is also possible, but it is found only in CISC-style processors.

Let us now return to the program in Figure 3.6 for adding a list of numbers. Indirect addressing can be used to access successive numbers in the list, resulting in the program shown in Figure 3.8. Register R4 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R4. The initialization section of the program loads the counter value *n* from memory location N into R2.

Then, it uses the Clear instruction to clear R3 to 0. The next instruction uses the Immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R4. Observe that we cannot use the Load instruction to load the desired immediate value, because the Load instruction can operate only on memory source operands. Instead, we use the Move instruction

Move R4, #NUM1



```
        Load            R2, N       Load the size of the list.
        Clear           R3          Initialize sum to 0.
        Move            R4, #NUM1   Get address of the first number.
LOOP:   Load            R5, (R4)    Get the next number.
        Add             R3, R3, R5  Add this number to sum.
        Add             R4, R4, #4  Increment the pointer to the list.
        Subtract        R2, R2, #1  Decrement the counter.
        Branch_if_[R2]>0 LOOP       Branch back if not finished.
        Store           R3, SUM     Store the final sum.
```

Fig. 3.8 Use of indirect addressing in the program of Fig 3.6

In many RISC-type processors, one general-purpose register is dedicated to holding a constant value zero. Usually, this is register R0. Its contents cannot be changed by a program instruction. We will assume that R0 is used in this manner in our discussion of RISC-style processors. Then, the above Move instruction can be implemented as

Add R4, R0, #NUM1

It is often the case that Move is provided as a *pseudo instruction* for the convenience of programmers, but it is actually implemented using the Add instruction. The first three instructions in the loop in Figure 3.8 implement the unspecified instruction block starting at LOOP in Figure 3.6. The first time through the loop, the instruction

Load R5, (R4)

fetches the operand at location NUM1 and loads it into R5. The first Add instruction adds this number to the sum in register R3. The second Add instruction adds 4 to the contents of the pointer R4, so that it will contain the address value NUM2 when the Load instruction is executed in the second pass through the loop.
As another example of pointers, consider the C-language statement

A = *B;

where B is a pointer variable and the '*' symbol is the operator for indirect accesses. This statement causes the contents of the memory location pointed to by B to be loaded into memory location A. The statement may be compiled into

```
Load R2, B
Load R3, (R2)
Store R3, A
```

Indirect addressing through registers is used extensively. The program in Figure 3.8 shows the flexibility it provides.

3.5.3 Indexing and Arrays

The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays.

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

For convenience, we will refer to the register used in this mode as the *index register*. Typically, this is just a general-purpose register. We indicate the Index mode symbolically as

$$X(R\mathit{i})$$

where X denotes a constant signed integer value contained in the instruction and R*i* is the name of the register involved. The effective address of the operand is given by

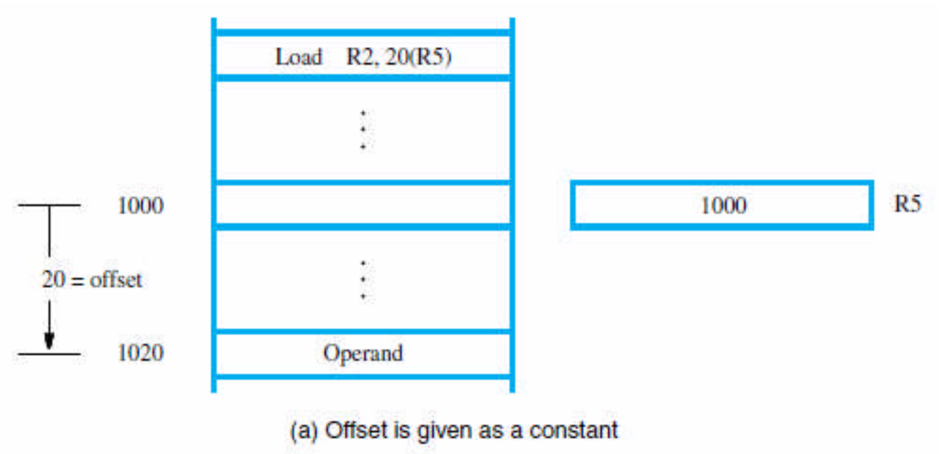
$$EA = X + [R\mathit{i}]$$

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

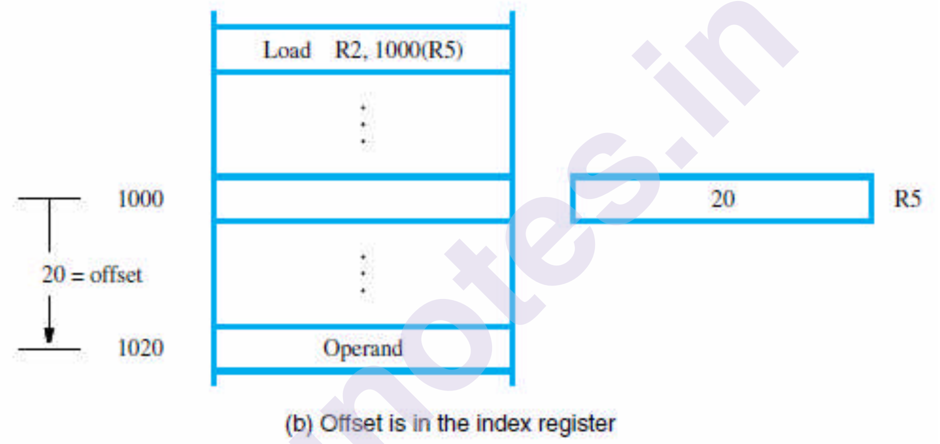
Figure3.9 illustrates two ways of using the Index mode. In Figure 3.9*a*, the index register, R5, contains the address of a memory location, and the value X defines an *offset* (also called a *displacement*) from this address to the location where the operand is found. An alternative use is illustrated in Figure 3.9*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 held in a register.

The usefulness of indexed addressing, consider a simple example 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 3.10. 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. We should note that the list in Figure 2.10 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



(a) Offset is given as a constant



(b) Offset is in the index register

Fig 3.9 Indexed addressing

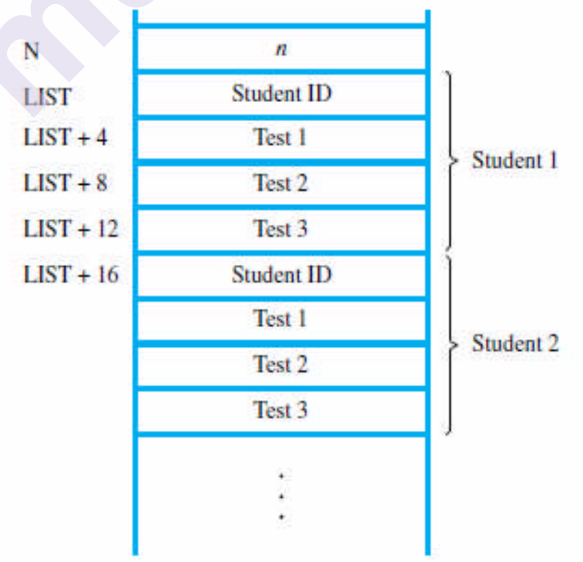
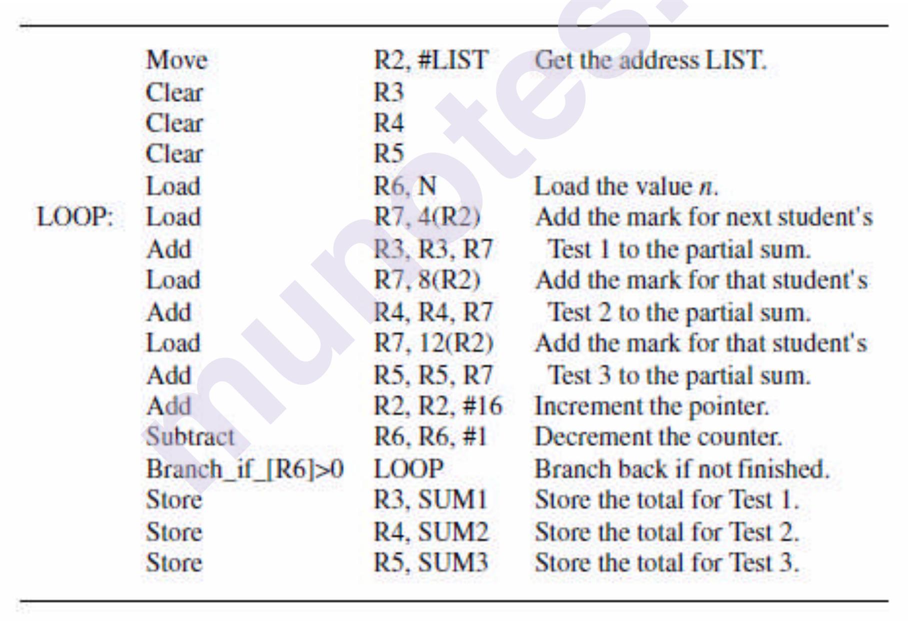


Fig. 3.10 A list of students marks

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. A possible program for this task is given in Figure 3.11. In the body of the loop, the program uses the

Index addressing mode in the manner depicted in Figure 3.9*a* to access each of the three scores in a student's record. Register R2 is used as the index register. Before the loop is entered, R2 is set to point to the ID location of the first student record which is the address LIST.

On the first pass through the loop, test scores of the first student are added to the running sums held in registers R3, R4, and R5, which are initially cleared to 0. These scores are accessed using the Index addressing modes 4(R2), 8(R2), and 12(R2). The index register R2 is then incremented by 16 to point to the ID location of the second student. Register R6, initialized to contain the value *n*, is decremented by 1 at the end of each pass through the loop. When the contents of R6 reach 0, all student records have been accessed, and the loop terminates. Until then, the conditional branch instruction transfers control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from registers R3, R4, and R5, into memory locations SUM1, SUM2, and SUM3, respectively.



```
        Move              R2, #LIST     Get the address LIST.
        Clear             R3
        Clear             R4
        Clear             R5
        Load              R6, N         Load the value n.
LOOP:   Load              R7, 4(R2)     Add the mark for next student's
        Add               R3, R3, R7      Test 1 to the partial sum.
        Load              R7, 8(R2)     Add the mark for that student's
        Add               R4, R4, R7      Test 2 to the partial sum.
        Load              R7, 12(R2)    Add the mark for that student's
        Add               R5, R5, R7      Test 3 to the partial sum.
        Add               R2, R2, #16   Increment the pointer.
        Subtract          R6, R6, #1    Decrement the counter.
        Branch_if_[R6]>0  LOOP          Branch back if not finished.
        Store             R3, SUM1      Store the total for Test 1.
        Store             R4, SUM2      Store the total for Test 2.
        Store             R5, SUM3      Store the total for Test 3.
```

Fig. 3.11 indexed addressing used in accessing test scores in the list in Fig. 3.10

3.6 ASSEMBLY LANGUAGE

Machine instructions are represented by patterns of 0s and 1s. Such patterns are awkward to deal with when discussing or preparing programs. Therefore, we use symbolic names to represent the patterns. So far, we have used normal words, such as Load, Store, Add, and Branch, for the instruction operations to represent the corresponding binary code patterns.

When writing programs for a specific computer, such words are normally replaced by acronyms called *mnemonics*, such as LD, ST, ADD, and BR. A shorthand notation is also useful when identifying registers, such as R3 for register 3. Finally, symbols such as LOC may be defined as needed to represent particular memory locations.

A complete set of such symbolic names and rules for their use constitutes a programming language, generally referred to as an *assembly language*. The set of rules for using the mnemonics and for 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 assembler program is one of a collection of utility programs that are a part of the system software of a computer.

The assembler, like any other program, is stored as a sequence of machine instructions in the memory of the computer. A user program is usually entered into the computer through a keyboard and stored either in the memory or on a magnetic disk.

At this point, the user program is simply a set of lines of alphanumeric characters. When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine language program. The latter contains patterns of 0s and 1s specifying instructions that will be executed by the computer. The user program in its original alphanumeric text format is called a *source program*, and the assembled machine-language program is called an *object program*.

The assembly language for a given computer may or may not be case sensitive, that is, it may or may not distinguish between capital and lower-case letters. In this section, we use capital letters to denote all names and labels in our examples to improve the readability of the text. For example, we write a Store instruction as

```
ST R2, SUM
```

The mnemonic ST represents the binary pattern, or *operation (OP) code*, for the operation performed by the instruction. The assembler translates this mnemonic into the binary OP code that the computer recognizes.

The OP-code mnemonic is followed by at least one blank space or tab character. Then the information that specifies the operands is given. In the Store instruction above, the source operand is in register R2. This information is followed by the specification of the destination operand, separated from the source operand by a comma. The destination operand is in the memory location that has its binary address represented by the name SUM.

Since there are several possible addressing modes for specifying operand locations, an assembly-language instruction must indicate which mode is being used. For example, a numerical value or a name used by itself, such as SUM in the preceding instruction, may be used to denote the Absolute mode. The number sign usually denotes an immediate operand. Thus, the instruction

```
ADD R2, R3, #5
```

adds the number 5 to the contents of register R3 and puts the result into register R2. The number sign is not the only way to denote the Immediate addressing mode. In some assembly languages, the immediate addressing mode is indicated in the OP-code mnemonic.

For example, the previous Add instruction may be written as

```
ADDI R2, R3, 5
```

The suffix I in the mnemonic ADDI states that the second source operand is given in the Immediate addressing mode. Indirect addressing is usually specified by putting parentheses around the name or symbol denoting the pointer to the operand. For example, if register R2 contains the address of a number in the memory, then this number can be loaded into register R3 using the instruction

```
LD R3, (R2)
```

3.6.1 Assembler Directives

In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program. We have already mentioned that we need to assign numerical values to any names used in a program. Suppose that the name TWENTY is used to represent the value 20. This fact may be conveyed to the assembler program through an *equate* statement such as

```
TWENTY EQU 20
```

This statement does not denote an instruction that will be executed when the object program is run; in fact, it will not even appear in the object program. It simply informs the assembler that the name TWENTY should be replaced by the value 20 wherever it appears in the program. Such statements, called *assembler directives* (or *commands*), are used by the assembler while it translates a source program into an object program.

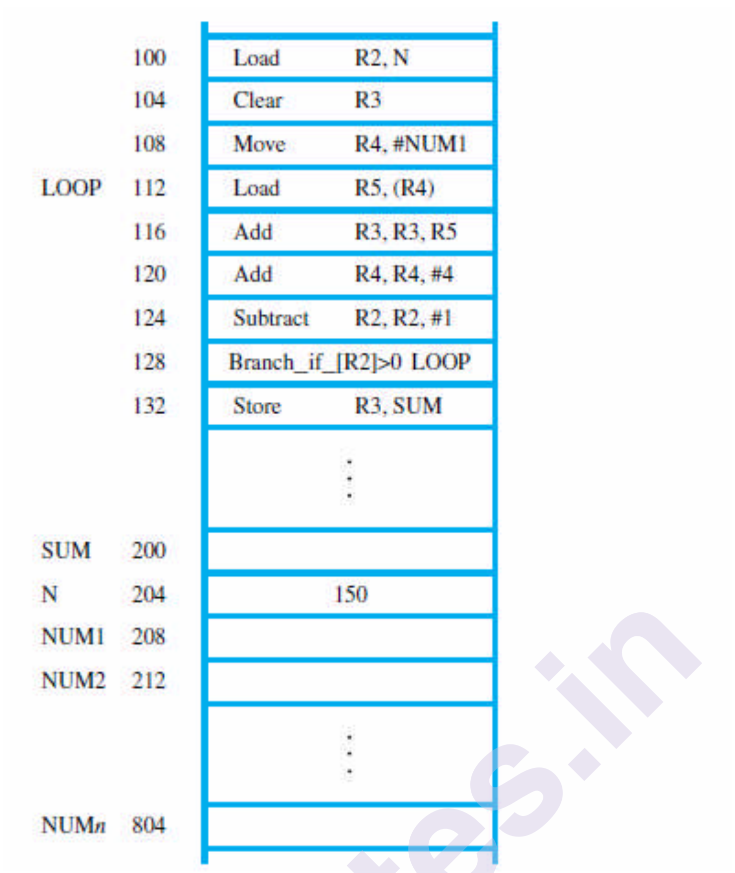


fig. 3.12 Memory arrangement for the program in fig. 3.8

Of the object program are to be loaded in the memory starting at address 100. It is followed by the source program instructions written with the appropriate mnemonics and syntax. Note that we use the statement

BGT R2, R0, LOOP

to represent an instruction that performs the operation

Branch_if_[R2]>0 LOOP

The second ORIGIN directive tells the assembler program where in the memory to place the data block that follows. In this case, the location specified has the address 200. This is intended to be the location in which the final sum will be stored. A 4-byte space for the sum is reserved by means of the assembler directive RESERVE. The next word, at address 204, has to contain the value 150 which is the number of entries in the list.

	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

Fig. 3.13 Assembly Language representation for the program in fig. 3.12

The DATAWORD directive is used to inform the assembler of this requirement. The next RESERVE directive declares that a memory block of 600 bytes is to be reserved for data. This directive does not cause any data to be loaded in these locations. The last statement in the source program is the assembler directive END, which tells the assembler that this is the end of the source program text.

A different way of associating addresses with names or labels is illustrated in Figure 3.13. Any statement that results in instructions or data being placed in a memory location may be given a memory address label. The assembler automatically assigns the address of that location to the label. For example, in the data block that follows the second ORIGIN directive, we used the labels SUM, N, and NUM1. Because the first RESERVE statement after the ORIGIN directive is given the label SUM, the name SUM is assigned the value 200. Whenever SUM is encountered in the program, it will be replaced with this value. Using SUM as a label in this manner is equivalent to using the assembler directive

SUM EQU 200

Similarly, the labels N and NUM1 are assigned the values 204 and 208, respectively, because they represent the addresses of the two word locations immediately following the word location with address 200.

Most assembly languages require statements in a source program to be written in the form Label: Operation Operand(s) Comment These four *fields* are separated by an appropriate delimiter, perhaps one or more blank or tab characters. The Label is an optional name associated with the memory address where the machine-language instruction produced from the statement will be loaded. Labels may also be associated with addresses of data items. In Figure 3.13 there are four labels: LOOP, SUM, N, and NUM1.

3.7 STACKS

A *stack* is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a pushdown stack. Imagine a pile of trays in a cafeteria; customers pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile. Another descriptive phrase, last-in–first-out (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms push and pop are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

In modern computers, a stack is implemented by using a portion of the main memory for this purpose. One processor register, called the stack pointer (SP), is used to point to a particular stack structure called the processor stack.

Data can be stored in a stack with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations. We use a stack that grows in the direction of decreasing memory addresses in our discussion, because this is a common practice.

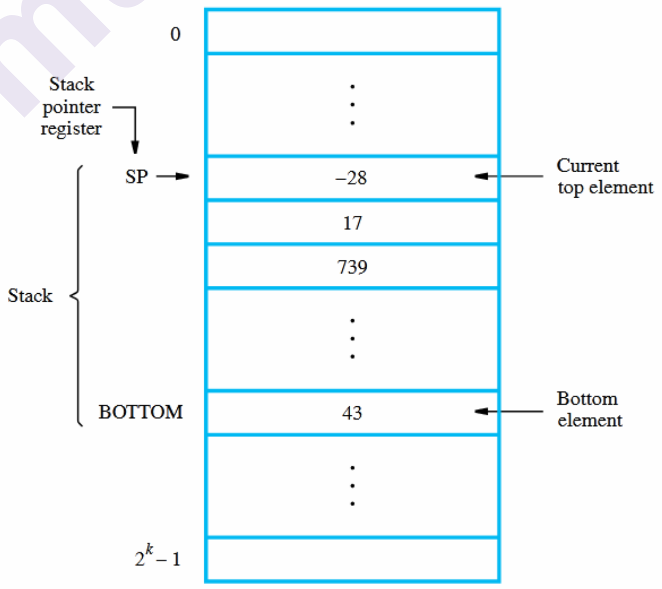


Figure 3.14: A stack of words in the memory.

Figure 3.14 shows an example of a stack of word data items. The stack contains numerical values, with 43 at the bottom and−28 at the top. The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time. If we assume a byte-addressable memory with a 32-bit word length, the push operation can be implemented as

```
Subtract    SP, SP, #4
Store       Rj, (SP)
```

where the Subtract instruction subtracts 4 from the contents of SP and places the result in SP. Assuming that the new item to be pushed on the stack is in processor register Rj, the Store instruction will place this value on the stack. These two instructions copy the word from Rj onto the top of the stack, decrementing the stack pointer by 4 before the store (push) operation. The pop operation can be implemented as

```
Load    Rj, (SP)
Add     SP, SP, #4
```

These two instructions load (pop) the top value from the stack into register Rj and then increment the stack pointer by 4 so that it points to the new top element. Figure15 shows the effect of each of these operations on the stack in Figure 14

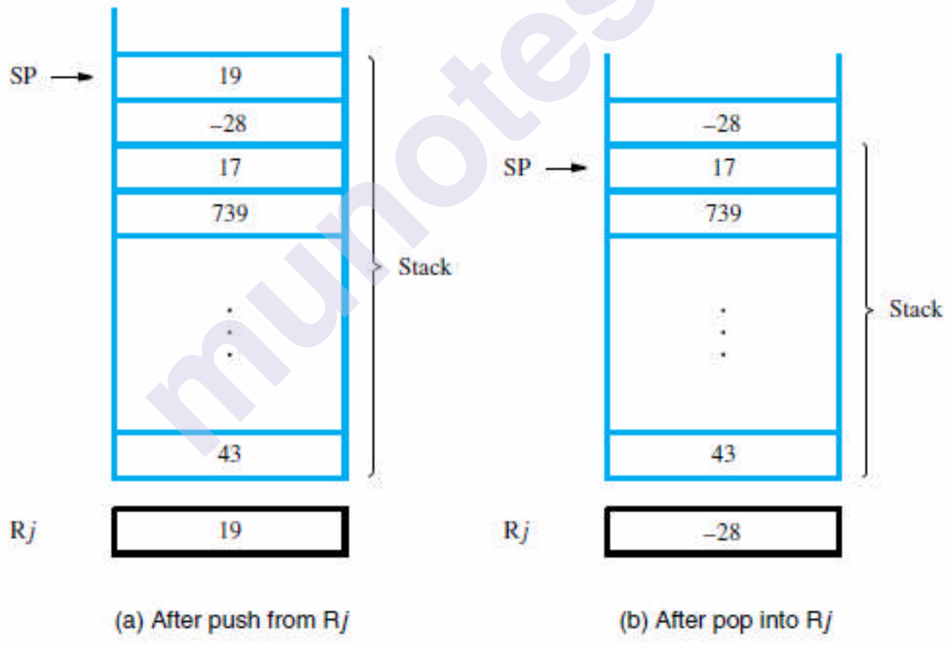


Fig. 3.15 Effect of stock operations on the stock in fig 3.14

3.8 SUBROUTINES

In a given program, it is often necessary to perform a particular task many times on different data values. It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a subroutine. For example, a subroutine may evaluate a mathematical function, or it may sort a list of values into increasing or decreasing order.

It is possible to reproduce the block of instructions that constitute a subroutine at everyplace where it is needed in the program. However, to save space, only one copy of this block is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is calling the subroutine. The instruction that performs this branch operation is named a Call instruction.

After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to return to the program that called it, and it does so by executing a Return instruction. Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location. The location where the calling program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed. Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.

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 just a special branch instruction that performs the following operations:

1. Store the contents of the PC in the link register
2. Branch to the target address specified by the Call instruction.

The Return instruction is a special branch instruction that performs the operation

- Branch to the address contained in the link register

Figure 3.16 illustrates how the PC and the link register are affected by the Call and Return instructions.

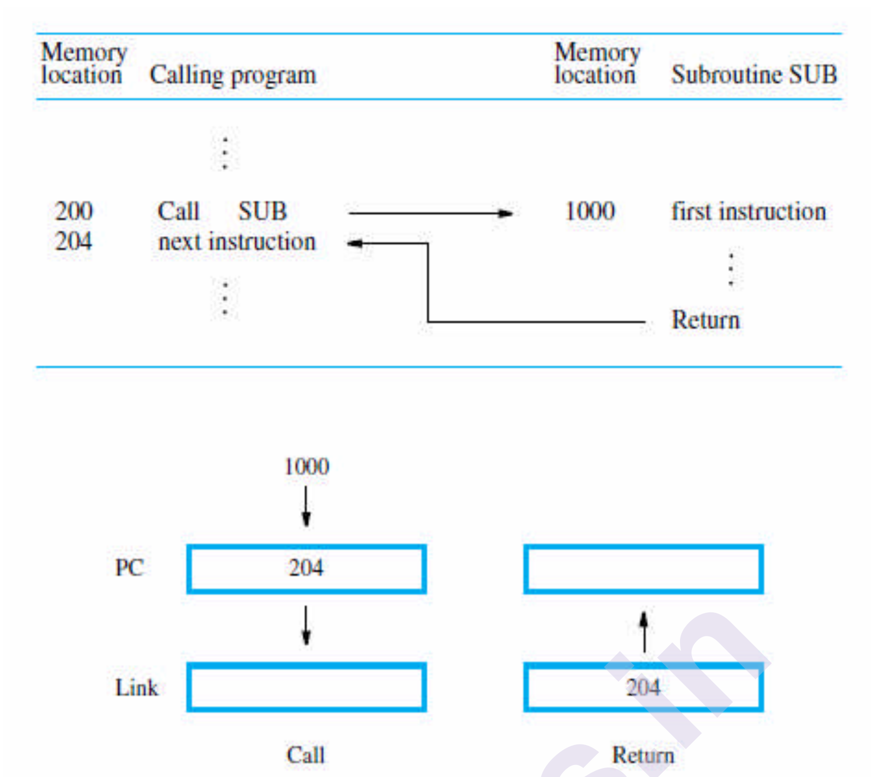


Fig. 3.16 subroutine linkage using a link register

3.8.1 Subroutine Nesting and the Processor Stack

A common programming practice, called subroutine nesting, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, overwriting 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 last-in–first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto the processor stack.

Correct sequencing of nested calls is achieved if a given subroutine SUB1 saves there turn address currently in the link register on the stack, accessed through the stack pointer, SP, before it calls another subroutine SUB2. Then, prior to executing its own Return instruction, the subroutine SUB1 has to pop the saved return address from the stack and load it into the link register.

3.8.2 Parameter Passing

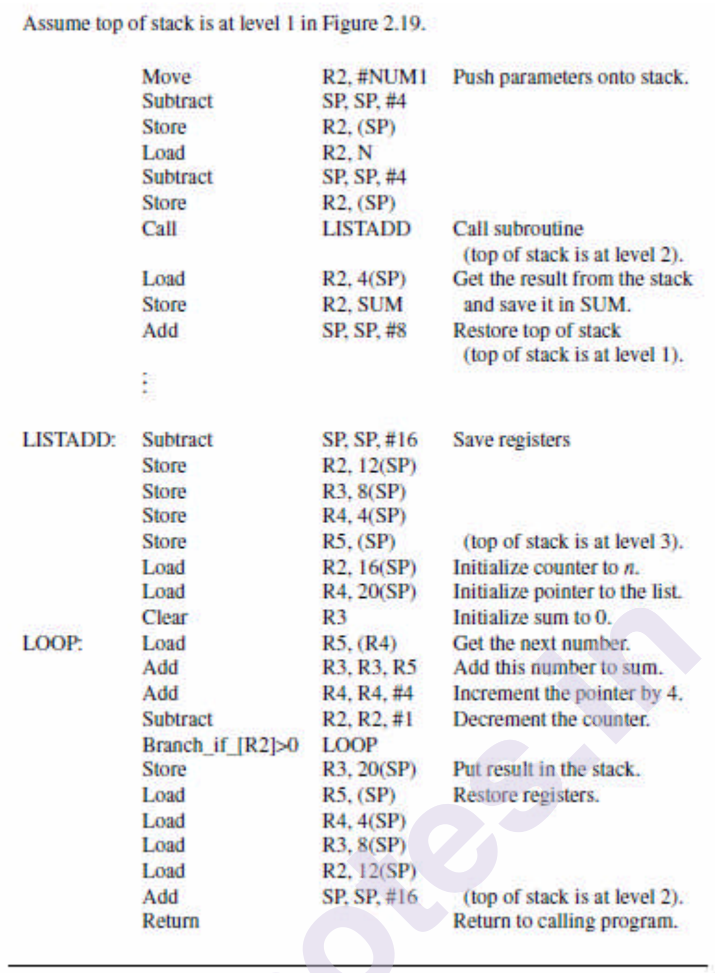
When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, which are the results of the computation. This exchange of information between a calling program and a subroutine is referred to as parameter passing. Parameter passing may be accomplished in several ways. The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack.

```
Calling program

            Load               R2, N          Parameter 1 is list size.
            Move               R4, #NUM1      Parameter 2 is list location.
            Call               LISTADD        Call subroutine.
            Store              R3, SUM        Save result.
            :

Subroutine

LISTADD:    Subtract           SP, SP, #4     Save the contents of
            Store              R5, (SP)         R5 on the stack.
            Clear              R3             Initialize sum to 0.
LOOP:       Load               R5, (R4)       Get the next number.
            Add                R3, R3, R5     Add this number to sum.
            Add                R4, R4, #4     Increment the pointer by 4.
            Subtract           R2, R2, #1     Decrement the counter.
            Branch_if_[R2]>0   LOOP
            Load               R5, (SP)       Restore the contents of R5.
            Add                SP, SP, #4
            Return                            Return to calling program.
```

Fig. 3.17 Program of fig. 3.8 written as a subroutine parameters passed through registers

Passing parameters through processor registers is straightforward and efficient. Figure 3.17 shows how the program in Figure 8 for adding a list of numbers can be implemented as a subroutine, LISTADD, with the parameters passed through registers. The size of the list, n, contained in memory location N, and the address, NUM1, of the first number, are passed through registers R2 and R4. The sum computed by the subroutine is passed back to the calling program through register R3. The first four instructions in Figure 17 constitute the relevant part of the calling program. The first two instructions load *n* and NUM1 into R2 and R4. The Call instruction branches to the subroutine starting at location LIST ADD. This instruction also saves the return address (i.e., the address of the Store instruction in the calling program) in the link register. The subroutine computes the sum and places it inR3. After the Return instruction is executed by the subroutine, the sum in R3 is stored in memory location SUM by the calling program.

In addition to registers R2, R3, and R4, which are used for parameter passing, the subroutine also uses R5. Since R5 may be used in the calling program, its contents are saved by pushing them onto the processor stack upon entry to the subroutine and restored before returning to the calling program.



Assume top of stack is at level 1 in Figure 2.19.

```
            Move              R2, #NUM1   Push parameters onto stack.
            Subtract          SP, SP, #4
            Store             R2, (SP)
            Load              R2, N
            Subtract          SP, SP, #4
            Store             R2, (SP)
            Call              LISTADD     Call subroutine
                                            (top of stack is at level 2).
            Load              R2, 4(SP)   Get the result from the stack
            Store             R2, SUM       and save it in SUM.
            Add               SP, SP, #8  Restore top of stack
                                            (top of stack is at level 1).
            :

LISTADD:    Subtract          SP, SP, #16 Save registers
            Store             R2, 12(SP)
            Store             R3, 8(SP)
            Store             R4, 4(SP)
            Store             R5, (SP)      (top of stack is at level 3).
            Load              R2, 16(SP)  Initialize counter to n.
            Load              R4, 20(SP)  Initialize pointer to the list.
            Clear             R3          Initialize sum to 0.
LOOP:       Load              R5, (R4)    Get the next number.
            Add               R3, R3, R5  Add this number to sum.
            Add               R4, R4, #4  Increment the pointer by 4.
            Subtract          R2, R2, #1  Decrement the counter.
            Branch_if_[R2]>0  LOOP
            Store             R3, 20(SP)  Put result in the stack.
            Load              R5, (SP)    Restore registers.
            Load              R4, 4(SP)
            Load              R3, 8(SP)
            Load              R2, 12(SP)
            Add               SP, SP, #16   (top of stack is at level 2).
            Return                        Return to calling program.
```

Figure 2.18 Program of Figure 2.8 written as a subroutine; parameters passed on the stack.

Figure 19 shows the stack entries for this example. Assume that before the subroutine is called, the top of the stack is at level 1. The calling program pushes the address NUM1and the value onto the stack and calls subroutine LISTADD. The top of the stack is now at level 2. The subroutine uses four registers while it is being executed. Since these registers may contain valid data that belong to the calling program, their contents should be saved at the beginning of the subroutine by pushing them onto the stack. The top of the stack is now at level 3. The subroutine accesses the parameters n and NUM1 from the stack using indexed addressing with offset values relative to the new top of the stack (level 3). Note that it does not change the stack pointer because valid data items are still at the top of the stack. The value n is loaded into R2 as the initial value of the count, and the address NUM1is loaded into R4, which is used as a pointer to scan the list entries. At the end of the computation, register R3 contains the sum. Before the subroutine returns to the calling program, the contents of R3 are inserted into the stack, replacing the parameter NUM1, which is no longer needed. Then the contents of the four registers used by the subroutine are restored from the stack. Also, the stack pointer is incremented to point to the top of the stack that existed when the subroutine was called, namely the parameter n at level 2. After the subroutine returns, the calling program stores the result in location SUM and lowers the top of the stack to its original level by incrementing the SP by 8.Observe that for subroutine LISTADD in Figure 2.18, we did not use a pair of instructions

```
Subtract    SP,    SP, #4
Store   Rj,    (SP)
```

to push the contents of each register on the stack. Since we have to save four registers, this would require eight instructions. We needed only five instructions by adjusting SP immediately to point to the top of stack that will be in effect once all four registers are saved. Then, we used the Index mode to store the contents of registers. We used the same optimization when restoring the registers before returning from the subroutine.

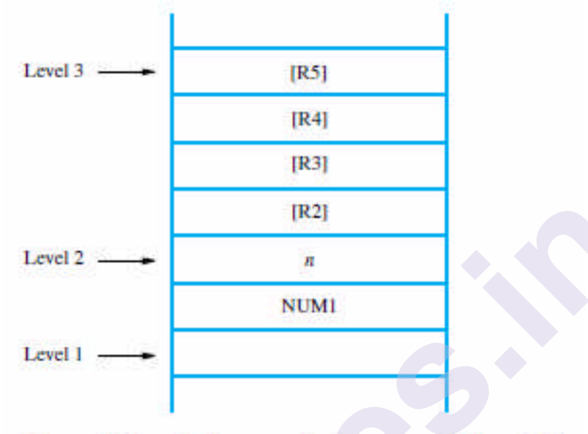


Figure 2.19 Stack contents for the program in Figure 2.18.

Parameter Passing by Value and by Reference

Note the nature of the two parameters, NUM1 and n, passed to the subroutines in Figures17 and 18. The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list. This technique is called passing by reference. The second parameter is passed by value, that is, the actual number of entries, n, is passed to the subroutine

3.8.3 The Stack Frame

Now, observe how space is used in the stack in the example in Figures 18 and 19.During execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private work space for the subroutine, allocated at the time the subroutine is entered and deallocated when the subroutine returns control to the calling program. Such space is called a stack frame.

If the subroutine requires more space for local memory variables, the space for these variables can also be allocated on the stack. Figure 20 shows an example of a commonly used layout for information in a stack frame.

In addition to the stack pointer SP, it is useful to have another pointer register, called the frame pointer (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine. In the figure, we assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R2, R3, and R4 need to be saved because they will also be used within the subroutine. When nested subroutines are used, the stack frame of the calling subroutine would also include the return address, as we will see in the example that follows.

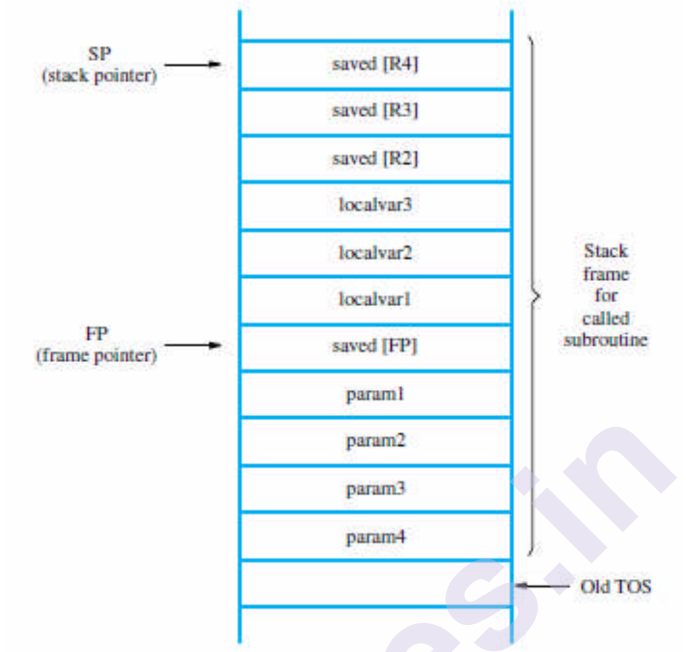


Figure 2.20 A subroutine stack frame example.

With the FP register pointing to the location just above the stored parameters, as shown in Figure 20, we can easily access the parameters and the local variables by using the Index addressing mode. The parameters can be accessed by using addresses 4(FP), 8(FP),....The local variables can be accessed by using addresses−4(FP),−8(FP),....The contents of FP remain fixed throughout the execution of the subroutine, unlike the stack pointer SP, which must always point to the current top element in the stack.

Now let us discuss how the pointers SP and FP are manipulated as the stack frame is allocated, used, and deallocated for a particular invocation of a subroutine. We begin by assuming that SP points to the old top-of-stack (TOS) element in Figure 20. Before the subroutine is called, the calling program pushes the four parameters onto the stack. Then the Call instruction is executed. At this time, SP points to the last parameter that was pushed on the stack. If the subroutine is to use the frame pointer, it should first save the contents of FP by pushing them on the stack, because FP is usually a general-purpose register and it may contain information of use to the calling program. Then, the contents of SP, which now points to the saved value of FP, are copied into FP.

Thus, the first three instructions executed in the subroutine are

```
Subtract     SP,     SP,     #4
Store   FP,     (SP)
Move    FP,      SP
```

The Move instruction copies the contents of SP into FP. After these instructions are executed, both SP and FP point to the saved FP contents. Space for the three local variables is now allocated on the stack by executing the instruction

```
Subtract          SP,      SP,      #12
```

Finally, the contents of processor registers R2, R3, and R4 are saved by pushing them onto the stack. At this point, the stack frame has been set up as shown in Figure 2.20.The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R4, R3, and R2 back into those registers, deallocates the local variables from the stack frame by executing the instruction

```
Add     SP,       SP,      #12
```

and pops the saved old value of FP back into FP. At this point, SP points to the last parameter that was placed on the stack. Next, the Return instruction is executed; transferring control back to the calling program. The calling program is responsible for deallocating the parameters from the stack frame, some of which may be results passed back by the subroutine. After deallocation of the parameters, the stack pointer points to the old TOS, and we are back to where we started.

3.9 TYPES OF MACHINE INSTRUCTION

3.9.1 Logical Instruction

Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits, as described in Appendix A. It is also useful to be able to perform logic operations in software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel. For example, the instruction

```
And R4, R2, R3
```

computes the bit-wise AND of operands in registers R2 and R3, and leaves the result in R4.An immediate form of this instruction may be

```
And R4, R2, #Value
```

where Value is a 16-bit logic value that is extended to 32 bits by placing zeros into the 16most-significant bit positions.

Consider the following application for this logic instruction. Suppose that four ASCII characters are contained in the 32-bit register R2. In some task, we wish to determine if the rightmost character is Z. If it is, then a conditional branch to FOUND Z is to be made. which is expressed in hexadecimal notation as 5A. The three-instruction sequence

```
AndR2, R2, #0xFF
MoveR3, #0x5A
Branch_if_[R2]=[R3]FOUNDZ
```

implements the desired action. The And instruction clears all bits in the leftmost three character positions of R2 to zero, leaving the rightmost character unchanged. This is the result of using an immediate operand that has eight 1s at its right end, and 0s in the 24 bits to the left. The Move instruction loads the hex value 5A into R3. Since both R2 and R3have 0s in the leftmost 24 bits, the Branch instruction compares the remaining character at the right end of R2 with the binary representation for the character Z, and causes a branch to FOUNDZ if there is a match.

3.9.2 Shift and Rotate

There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use a logical shift. For a signed number, we use an arithmetic shift, which preserves the sign of the number. Logical Shifts Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (L Shift R). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction. The general form of a Logical-shift-left instruction is

LShiftL Ri,Rj, count

which shifts the contents of register Rj left by a number of bit positions given by the count operand, and places the result in register Ri, without changing the contents of Rj. The count operand may be given as an immediate operand, or it may be contained in a processor register. To complete the description of the shift left operation, we need to specify the bit values brought into the vacated positions at the right end of the destination operand, and to determine what happens to the bits shifted out of the left end. Vacated positions are filled with zeros. In computers that do not use condition code flags, the bits shifted out are simply dropped. In computers that use condition code flags, these bits are passed through the Carry flag, C, and then dropped. Involving the C flag in shifts is useful in performing arithmetic operations on large numbers that occupy more than one word. Figure 3.23ashows an example of shifting the contents of register R3left by two bit positions. The Logical-shift-right instruction, L Shift R, works in the same manner except that it shifts to the right. Figure 21 billustrates this operation.

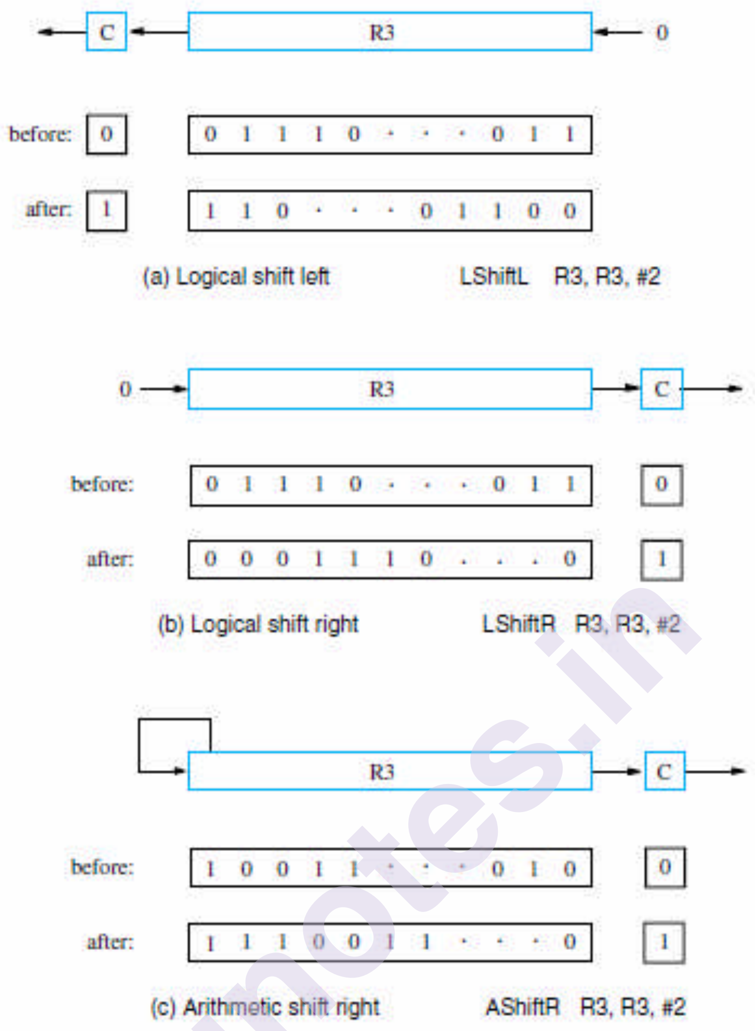


Figure 2.23 Logical and arithmetic shift instructions.

In an arithmetic shift, the bit pattern being shifted is interpreted as a signed number. A study of the 2's-complement binary number representation in Figure 3.3 reveals that shifting a number one bit position to the left is equivalent to multiplying it by 2, and shifting it to the right is equivalent to dividing it by 2. Of course, overflow might occur on shift in g left, and the remainder is lost when shifting right. Another important observation is that on a right shift the sign bit must be repeated as the fill-in bit for the vacated position as a requirement of the 2's-complement representation for numbers. This requirement when shifting right distinguishes arithmetic shifts from logical shifts in which the fill-in bit is always 0. Otherwise, the two types of shifts are the same. An example of an Arithmetic-shift-right instruction, A Shift R, is shown in Figure 21c. The Arithmetic-shift-left is exactly the same as the Logical-shift-left. Rotate Operations In the 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. For situations where it is desirable to preserve all of the bits, rotate instructions may be used instead. These are instructions that move the bits shifted out of one end of the operand into the other end. Two versions of both the Rotate-left and Rotate-right instructions are often provided. In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag. Figure21 shows the left and right rotate operations with and without the C flag being included in the rotation. Note that when the C flag is not included in the rotation, it still retains the last bit shifted out of the end of the register. The OP codes Rotate L, Rotate LC, Rotate R, and Rotate RC, denote the instructions that perform the rotate operations.

3.9.3 Multiplication and Division

Two signed integers can be multiplied or divided by machine instructions with the same format as we saw earlier for an Add instruction. The instruction

Multiply Rk,Ri,Rj

performs the operation

$$Rk \leftarrow [Ri] \times [Rj]$$

The product of two n-bit numbers can be as large as 2nbits. Therefore, the answer will not necessarily fit into register Rk. A number of instruction sets have a Multiply instruction that computes the low-order n bits of the product and places it in register Rk, as indicated. This is sufficient if it is known that all products in some particular application task will fit into n bits. To accommodate the general 2n-bit product case, some processors produce the product in two registers, usually adjacent registers Rk and R(k+ 1), with the high-order half being placed in register R(k+ 1).

An instruction set may also provide a signed integer Divide instruction

Divide Rk,Ri,Rj

which performs the operation

$$Rk \leftarrow [Rj]/[Ri]$$

placing the quotient in Rk. The remainder may be placed in R(k+ 1), or it may be lost.

3.10 CISC INSTRUCTION SET

Key difference from RISC

1. We can operate directly on operands. Don’t require to *load/store architecture*
2. Instruction can of different length

Instructions in modern CISC processor typically do not use three-address format. Most arithmetic and logic instruction user the *two-address* format

Syntax of instruction

Operation   destination, source

For example Add instruction of this type is

Add   B,A

Which performs the operation B←[A] + [B] on memory operands. When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that memory location B is both a source and a destination.

Consider again the task of adding two numbers

C=A+B

where all three operands may be in memory locations. Obviously, this cannot be done with a single two-address instruction. The task can be performed by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is

Move C,B

Which performs the operation C←[B], leaving the contents of location B unchanged. The operation C←[A]+[B] can now be performed by the two-instruction sequence

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

Observe that by using this sequence of instructions the contents of neither A nor B locations are overwritten.

3.10.1 Additional Addressing Modes

Most CISC processors have all of the five basic addressing modes—Immediate, Register, Absolute, Indirect, and Index. Three additional addressing modes are often found in CISC processors.

Auto increment and Auto decrement Modes

These are two modes that are particularly convenient for accessing data items in successive locations in the memory and for implementation of stacks.

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 point to the next operand in memory.

We denote the Auto increment mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Auto increment mode is written as (Ri)+

To access successive words in a byte-addressable memory with a 32-bit word length, the increment amount must be 4. Computers that have the Auto increment mode automatically increment the contents of the register by a value that corresponds to the size of the accessed operand. Thus, the increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for32-bit operands. Since the size of the operand is usually specified as part of the operation code of an instruction, it is sufficient to indicate the Auto increment mode as (Ri)+.

As a companion for the Auto increment mode, another useful mode accesses the memory locations in the reverse order:

Auto decrement mode—The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

We denote the Auto decrement mode by putting the specified register in parentheses, pre-ceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write −(Ri)

In this mode, operands are accessed in descending address order.

The address is decremented before it is used in the Auto decrement mode and incremented after it is used in the Auto increment mode. The main reason for this is to make it easy to use these modes together to implement a stack structure.Instead of needing two instructions

```
Subtract      SP, #4
Move(SP), NEWITEM
```

to push a new item on the stack, we can use just one instruction

```
Move−(SP), NEWITEM
```

Similarly, instead of needing two instructions

```
MoveITEM, (SP)
AddSP, #4
```

to pop an item from the stack, we can use just

```
MoveITEM, (SP)+
```

3.10.2 Relative Mode

We have defined the Index mode by using general-purpose processor registers. Some computers have a version of this mode in which the program counter, PC, is used instead of a general-purpose register. Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter. Since the addressed location is identified relative to the program counter, which always identifies the current execution point in a program, the name Relative mode is associated with this type of addressing.

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

3.11 CISC AND RISC STYLES

RISC style is characterized by:

1. Simple addressing modes.
2. All instructions fitting in a single word.
3. Fewer instructions in the instruction set, as a consequence of simple addressing modes.
4. Arithmetic and logic operations that can be performed only on operands in processor registers.
5. Load/store architecture that does not allow direct transfers from one memory location to another; such transfers must take place via a processor register.
6. Simple instructions those are conducive to fast execution by the processing unit using techniques such as pipelining.
7. Programs that tend to be larger in size, because more, but simpler instructions are needed to perform complex tasks

CISC style is characterized by:

1. More complex addressing modes.
2. More complex instructions, where an instruction may span multiple words.
3. Many instructions that implement complex tasks.
4. Arithmetic and logic operations that can be performed on memory operands as well as operands in processor registers.
5. Transfers from one memory location to another by using a single Move instruction.
6. Programs that tend to be smaller in size, because fewer, but more complex instructions are needed to perform complex tasks.

Before the 1970s, all computers were of CISC type. An important objective was to simplify the development of software by making the hardware capable of performing fairly complex tasks, that is, to move the complexity from the software level to the hardware level. This is conducive to making programs simpler and shorter, which was important when computer memory was smaller and more expensive to provide. Today, memory is inexpensive and most computers have large amounts of it.

RISC-style designs emerged as an attempt to achieve very high performance by making the hardware very simple, so that instructions can be executed very quickly in pipelined fashion. This results in moving complexity from the hardware level to the software level. Sophisticated compilers were developed to optimize the code consisting of simple instructions. The size of the code became less important as memory capacities increased.

While the RISC and CISC styles seem to define two significantly different approaches, today's processors often exhibit what may seem to be a compromise between these approaches. For example, it is attractive to add some non-RISC instructions to a RISC processor in order to reduce the number of instructions executed, as long as the execution of these new instructions is fast.

3.12 LET US SUM UP

Thus, we have studied the representation and execution of instructions and programs at the assembly and machine level as seen by the programmer. The discussion emphasized the basic principles of addressing techniques and instruction sequencing. The programming examples illustrated the basic types of operations implemented by the instruction set of any modern computer.

3.13 LIST OF REFERENCES

➢ Carl Hamacher et al., Computer Organization and Embedded Systems, 6 ed., McGraw-Hill 2012

➢ Patterson and Hennessy, Computer Organization and Design, Morgan Kaufmann, ARM Edition, 2011

➢ R P Jain, Modern Digital Electronics, Tata McGraw Hill Education Pvt. Ltd. , 4th Edition, 2010

3.14 UNIT END EXERCISES

1) Explain how memory is used to read write operations.
2) Explain Big-Endian and Little Endian Assignment.
3) Explain characteristics of RISC instruction set.
4) State and explain the ways of byte address assignment.
5) What is pointer? Explain its use in indirection operation.

❖❖❖❖

4.1

BASIC PROCESSOR UNIT

Unit Structure

4.1.1 OBJECTIVES

At the end, the learners will be able to

- Describe the various components of a processor
- Illustrate the concept of Datapath.
- Compare and Constrast between various types of instruction
- Differentiate between the RISC and CISC processor

4.1.2 INTRODUCTION

1. At a highest level, a computer consists of CPU(central processing unit),memory and Input-output components with one or more module of each type.

2. These components are interconnected in some fashion to achieve the basic function of the computer, which is to execute programs.

3. Thus at a top level we can characterize a computer system by describing a)The external behavior of each component, that is the data and control signals that it exchanges with other components and b)The interconnection structure and the contrls required to manage the use of the interconnection structure.

4. This top-level view of structure and function is important because of its explanatory power in understanding the nature of a computer.

5. Equally important is its use to understand the increasingly complex issues of performance evaluation.

6. A grasp of the top-level structure and function offers insight in to system bottlenecks, alternate pathways, the magnitude of system failures if a component fails and the ease of adding performance enhancements.

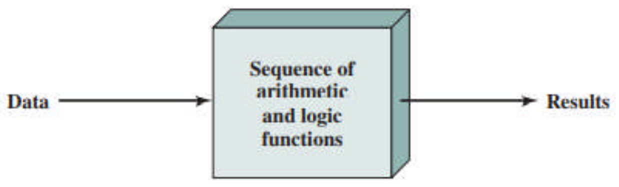
7. In many cases, requirements for greater system power and fail safe capabilities are being met by changing the design rather than merely increasing the speed and reliability of individual components.

8. Thus, this unit focuses on major components of computer, instruction fetch and RISC and CISC.

4.1.3 MAIN COMPONENTS

1. All computer designs are based on concepts developed by john von Neumann at the institute of Advanced studies, Princeton. Such a design is referred to as the Von Neumann architecture and is based on three key concepts.
   - 1.1 Data and instructions are stored in a single read-write memory.
   - 1.2 The contents of this memory are addressable by location, without regard to the type of data contained there.
   - 1.3 Execution occurs in a sequential fashion from one instruction to the next.
2. There is a small set of basic logic components that can be combined in various ways to store binary data and perform arithmetic and logical operations on that data.
3. If there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed.
4. One can think of the process of connecting the various components in the desired configuration as a form of programming.

5. The resulting "program" is in the form of hardware and is termed a hardwired program.
6. Suppose we construct a general-purpose configuration of arithmetic and logic functions. This set of hardware will perform various functions on data, depending on the control signals applied to the hardware.
7. In the original case of customized hardware, the system accepts data and control signals and produces results.
8. But with general-purpose hardware, the system accepts data and control signals and produces results.
9. Thus instead of rewriting the hardware for each new program, the programmer merely needs to supply a new set of control signals.
10. At each step, some arithmetic logical operation is performed on some data. For each step, a new set of control signals is needed.
11. Let us provide a unique code for each possible set of control signals, and let us add to the general-purpose hardware a segment that can accept a code and generate control signals as shown in figure given below.
12. Programming is now much easier. Instead of rewriting the hardware for each new program, all we need to do is provide a new sequence of codes.
13. The two major components of the system, an instruction interpreter and a module of general-purpose arithmetic and logic functions. These two constitute the CPU.
14. An input device will bring instructions and data in sequentially.
15. Operations on data may require access to more than just one element at a time in a predetermined sequence. Thus there must be a place to temporarily store both instructions and data. That module is called memory or main memory to distinguish it from other peripheral devices.
16. The CPU exchanges data with memory, for this purpose a two types of registers is required such as Memory address register(MAR) which specifies the address in memory for the next read or write and a memory buffer register (MBR) which contains the data to be written in to memory or receives the data read from memory.
17. A Memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a binary number that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to CPU and memory and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.



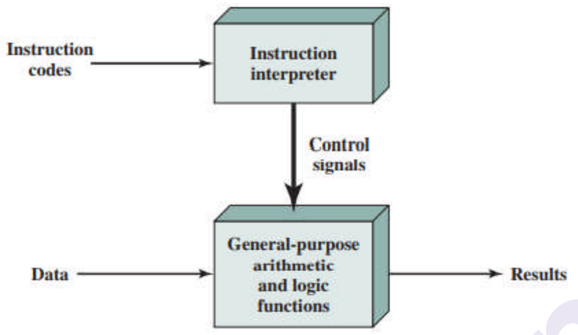
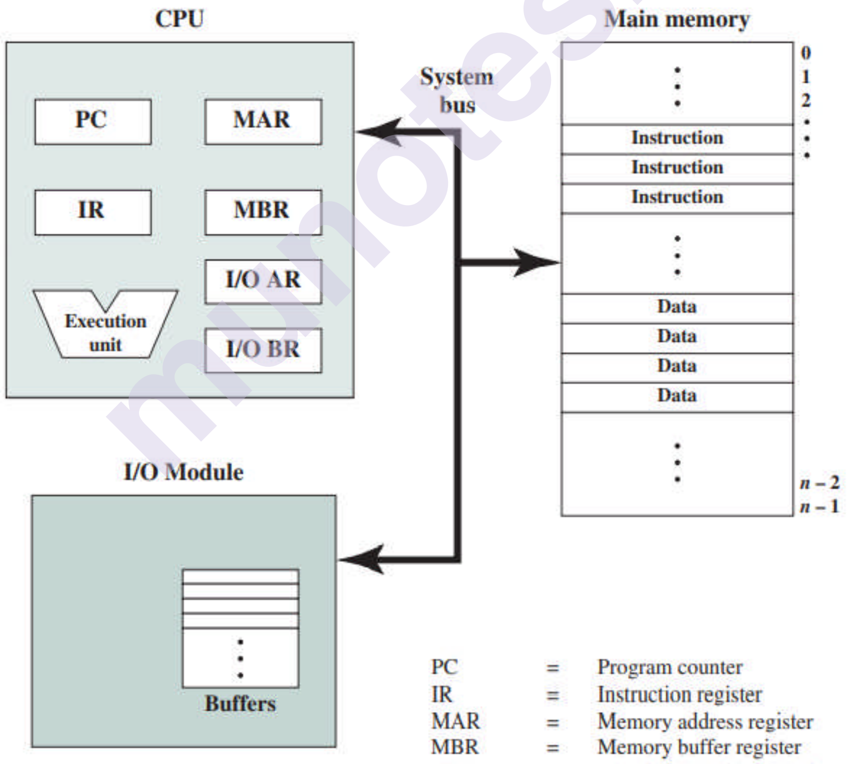


Fig 1 (Programming in Hardware) and Fig 2 (Programming in Software)



PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

Fig 2 Different Components of a Computer

4.1.3.1 Register and Register Files

1. Computers compute. The component that performs computation is CPU or more concretely it is the ALU(Arithmetic logical unit) in CPU that do the computation.

2. To compute, we need first prepared input, however ALU cannot access memory directly, instead a set of registers are provided as a cache that is faster but smaller than main memory.

3. Not only the General purpose registers are involved in computation, but CPU also has some registers in the purpose of control and recording status.

4. Data Registers - for eg MOV AX, 1234H here this instruction explains data is moved from address 1234H to register H & L.

5. Address registers:-To access some location of memory, we simply use an address register to contain the address of that location, but the actual practice is kind of much more complex. One popular addressing method is segmented addressing. With this method, memory is divided in to segments and each segment is of variable-length, blocks of words. To refer to a location in such a memory system, we need to give two pieces of information. One is the segment number and second the location of data in that segment. That is the address consists of two parts, segment address and the offset within the segment. For example CPU 8086 shifts the content of CS to the left by 4 bits and then adds up the result and the content of IP. Finally the sum is used as the effective address.

   CS:IP

   DS:DI

   DS:SI

It should b made clear that segment is just a logical concept, not a physically existing entity in memory. We may simply write to CS to change the segment it point to.

Stack Pointers

Due to the popularity of stack in program execution, computer systems provide registers to access memory segment in the way of accessing stacks. For eg in 8086, we have SS:SP where SS gives the stack segment and SP always points to the top of the stack. Thus the following two sets of instructions have the same effect.

```
PUSH AX

SUB Sp,2

Mov(SS:SP), AX
```

6. Control Registers- All the registers discussed above are related to data access, but there are some control registers which are used for executing instructions in the machine.

6.1 Program Counter(PC)- Contains the address of an instruction to be fetched from memory.

6.2 Instruction Register(IR)- contains the instruction most recently fetched. The execution of an instruction is actually to interpret the operation code in the instruction and generate signals for ALU or other components in CPU. For example when xy==00, ALU does A+B==>C ad when xy=01, ALU does A-B==>C, etc.

7. Status Register- CPU also includes registers, that contain status information. Thy are known as Program Status word(PSW). PSW typically contain condition codes with other status information.

7.1 Condition codes are bits set by the processor hardware as the result of operations. For example an arithmetic operation may produce a positive, negative, zero or overflow result. The code may subsequently b tested as part of conditional branch operation, Lets say

```
CMP AX,BX
JGE Exit
```

Generally, the condition codes cannot b altered by explicit referece becae they are intended for feedback regarding the execution of an instruction and are updated automatically whenever a related instruction is executed. There are a number of factors that have to be taken in to account. One is operating system support and another key factor is the allocation of control information between registers and memory. As registers are much faster, but due to the price reason a computer system doesn’t have many registers so atleast part of control information has to be put in memory.

8. Register File- There are two set of registers called “General Purpose” and “Special Purpose”.

8.1 The origin of the register set is simply the need to have some sort of memory on the computer and the inability to build what we now call “Main Memory”.

8.2 When reliable technologies such as magnetic cores, became available for main memory, the concept of CPU registers was retained.

8.3 Registers are now implemented as a set of flip-flops physically located on the CPU chip. These are used because access time for registers are two orders of magnitude faster than access times for main memory (1 nanosecond Vs. 80 nanoseconds).

8.4 General Purpose registers- These are mostly used to store intermediate results of computation. The count of such registers is often a power of 2 say 2^4=16 and so on, as N bit address 2^N items.

8.5 Special Purpose registers- These are often used by the control unit in its execution of the program.

8.5.1 PC- The Program counter- It is also called as the Instruction pointer(IP) which points to the memory location of the instruction to be executed next.

8.5.2 IR(Instruction Register)- This holds the machine language version of the instruction currently being executed.

8.5.3 MAR(Memory address register)- This holds the address of the memory word being referenced. All execution steps begin with PC, MAR.

8.5.4 MBR(Memory Buffer Register)- also called MDR(Memory Data Register) which holds the data being read from memory o written to memory.

8.5.5 PSR(Program status Register)- often called the PSW (Program status word) contains a collection of logical bits that characterize the status of the program executed lastly in memory.

8.5.6 PSR(Program Status Register) is actually a collection of bits that describe the running status of the process. The PSR is generally divided in to two parts

ALU Result Bits: The carry–out from the last arithmetic computation.

V set if the last arithmetic operation resulted in overflow.

N set if the last arithmetic operation gave a negative number.

Z set it the last arithmetic operation resulted in a 0.

Control Bits: Is set if interrupts are enabled. When I = 1, an I/O device can raise an interrupt when it is ready for a data transfer.

Priority : A multi–bit field showing the execution priority of the CPU; e.g., a 3–bit field for priorities 0 through 7.This facilitates management of I/O devices that have different priorities associated with data transfer rates.

Access Mode :The privilege level at which the current program is allowed to execute. All operating systems require atleast two modes: Kernel and User.

4.1.3.2 ALU (Arithmetic Logic Unit)

1. The heart of every computer is an Arithmetic Logic Unit(ALU). This is the part of the computer which performs arithmetic operations on numbers eg addition, substraction, etc.

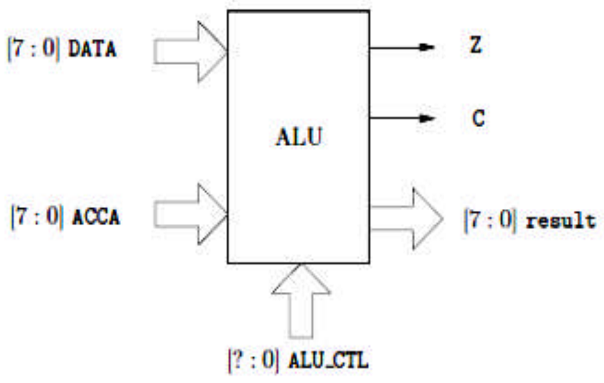


Fig 3 Simple Block Diagram of ALU

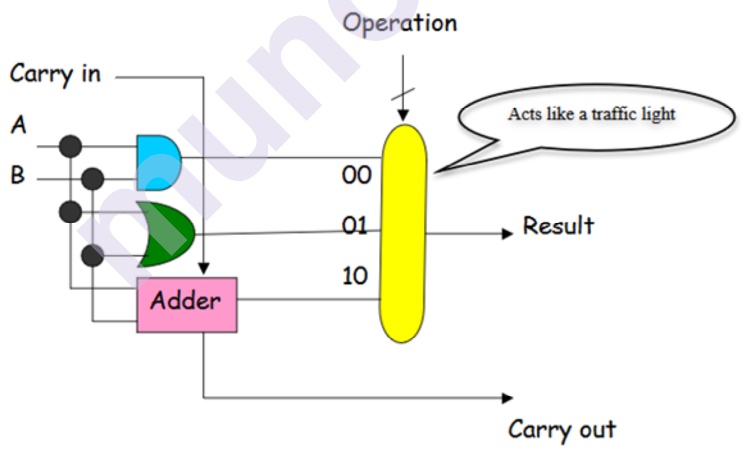
2. Above figure is describing about ALU which will perform 10 functions on 8-bit inputs. So this ALU will generate an 8bit result, one bit carry(c), and a one bit zero-bit(Z). So ALU uses control lines for selection of particular function among these 10 functions.

3. The following Table describes these instructions which will be executed by ALU shown in above figure

Sr.No	Mnemonic	Description
1	LOAD	(Load DATA into RESULT) DATA => RESULT C is a don't care 1-> Z if RESULT == 0, 0-> Z otherwise
2	ADDA	(Add DATA to ACCA) ACCA + DATA => RESULT C is carry from addition 1-> Z if RESULT == 0, 0-> Z otherwise
3	SUBA	(Subtract DATA from ACCA) ACCA – DATA => RESULT C is borrow from subtraction 1 ->Z if RESULT == 0, 0-> Z otherwise
4	ANDA	(Logical AND DATA with ACCA) ACCA & DATA => RESULT C is a don't care 1-> Z if RESULT == 0, 0 -> Z otherwise
5	ORAA	(Logical OR DATA WITH ACCA) ACCA | DATA => RESULT C is a don't care 1-> Z if RESULT == 0, 0 -> Z otherwise
6	COMA	(Compliment with ACCA) 1-> C 1->Z if RESULT == 0, 0 Z otherwise

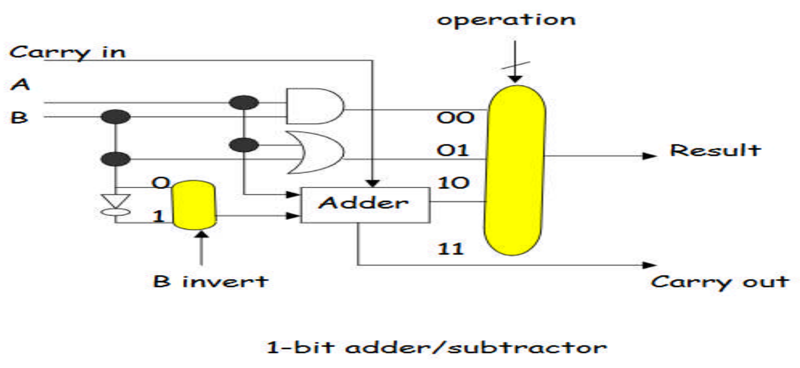
7	INCA	(Increment ACCA by 1) ACCA + 1 = RESULT C is a don’t care 1 if RESULT == 0, 0-> Z otherwise
8	LSRA	(Logical shift right of ACCA) Shift all bits of ACCA one place to the right: 0 RESULT[7], ACCA[7:1] RESULT[6:0] ACCA[0] C 1 ->Z if RESULT == 0, 0 Z otherwise
9	LSLA	(Logical shift left of ACCA) Shift all bits of ACCA one place to the left: 0-> RESULT[0], ACCA[6:0] RESULT[7:1] ACCA[7] ->C 1 ->Zif RESULT == 0, 0-> Z otherwise
10	ASRA	(Arithmetic shift right of ACCA) Shift all bits of ACCA one place to the right: ACCA[0] RESULT[7], ACCA[7:1]->RESULT[6:0] ACCA[0] ->C 1 -> Z if RESULT == 0, 0 -> Z otherwise

For example here we are going to see the designing of one bit ALU which does operation like AND, OR, ADD, where 00 implies the AND operation, 01 implies OR, and 10 implies ADD operation



One bit ALU

Now Same ALU can be designed for one bit substraction in which A-B operation is performed by method of A+2’s Complement of B and A+1’s Complement of B+1



For subtraction, B invert = 1 and Carry in = 1

One bit ALU for subtraction

4.1.3.3 Control Unit

1. A machine instruction set goes a long way towards defining the processor.
2. But before the execution of instruction, one must know the machine instruction set along with the effect of each opcode, addressing modes, set of available registers and also along with the functions that the processor has to do the execution, but this is the not the actual case as to execute functions , also there is a requirement of external interfaces, usually through a buses and how interrupts are handled, so to handle all these operations altogether in an controlled manner, there is need to for a separate unit, known as Control unit.
3. So control unit is one which is going to handle the functions like

   3.1 Operation (opcodes)

   3.2 Addressing Modes

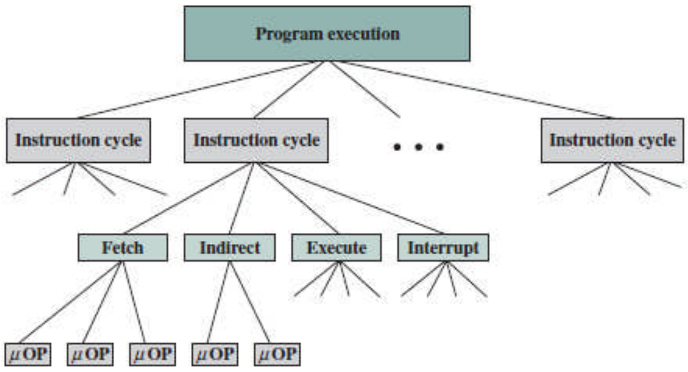
   3.3 Registers

   3.4 I/O Module interface

   3.5 Memory Module Interface

   3.6 Interrupts
4. Items from 3.1 to 3.3 are defined by the instruction set. Item 3.4 and 3.5 are typically defined by specifying the system bus. Item 3.6 is defined partially by the system bus and partially by the type of support the processor offers to the operating system.
5. The operation of a computer, in executing a program, consists of a sequence of instruction cycles, with one machine instruction per cycle.
6. Each instruction cycle is made up of a number of smaller units. I.e fetch, indirect, execute and interrupt.
7. Each instructions is executed during an instruction cycle made up of shorter sub cycles (eg fetch, indirect, execute, interrupt).

8. The execution of each sub cycle involves one or more shorter operations, that is micro-operations.



Elements of a Program Execution

9. Fetch Cycle- It is the cycle which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. Four registers are involved

9.1 Memory Address Register(MAR)-It specifies the address in memory for a read or write operation.

9.2 Memory Buffer Register(MIBR)-It contains the value to be stored in memory or the last value read from memory.

9.3 Program Counter(PC)-Holds the address of the next instruction to be fetched.

9.4 Instruction Register-Holds the last instruction fetched.

10. The Interrupt Cycle- At the completion of the execute cycle, a test is made to determine whether any enabled interrupt have occurred. If yes then the interrupt cycle occurs. For eg

t_1: MBR ← (PC)
t_2: MAR ← Save_Address
PC ← Routine_Address
t_3: Memory ← (MBR)

In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved and the PC is loaded with the address of the start of the interrupt-processing routine.

10. The Execute Cycle- The fetch, indirect and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and in each ease, the same micro-operations are repeated each time around. This is not true of the execute cycle, because of the variety of opcodes there are number of different sequences of micro-operations that can occur. For example instruction ADD R1,X adds the contents of the location X to register R1..

The following sequence of micro-operation might occur

1. MAR(IR address), 2.MBR(memory), 3. R1-(R)+(MBR). These three complete set of operation is required to complete the instruction ADD.

12. The Instruction Cycle- Each phase of the instruction cycle can be decomposed in to a sequence of elementary micro-operations.

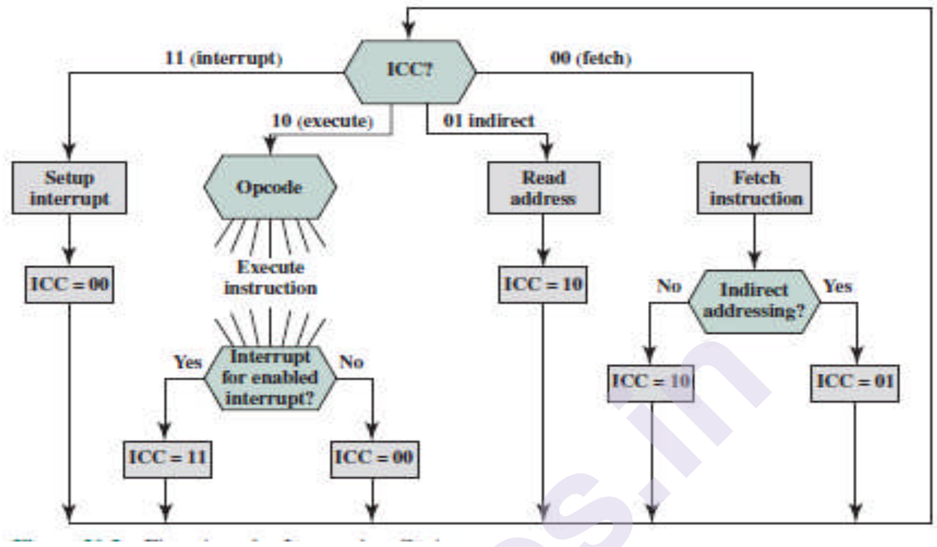


Fig 6 Flow chart for Instruction Cycle

4.1.3.4 Interfaces to instruction and data memories

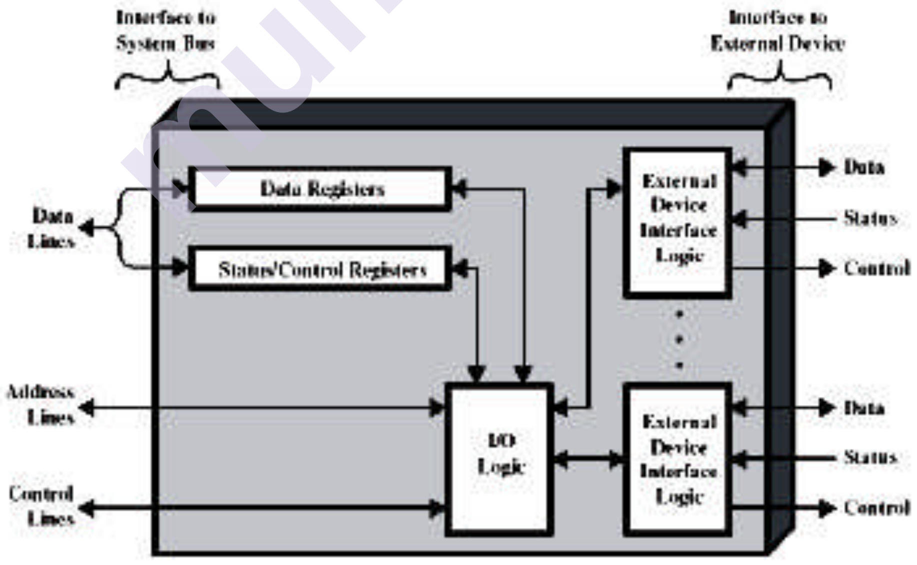


Fig 7 Block Diagram of an I/O Module

1. Module connects to the computer through a set of signal lines-system bus.

2. Data transferred to and from the module are buffered with data registers.

3. Status registers are useful for providing status to the signal lines and also act as a control registers.

4. Set of control lines are used by module logic for interaction purpose.

5. To issue commands to the I/O Module, processor uses control signals lines.

6. To control an device module must generate and recognize addresses for an device.

7. The function of an I/O module to allow an processor to view devices.

8. I/O module may hide device details from the processor so that only the processor will be incharge of performing the read ad write operations.

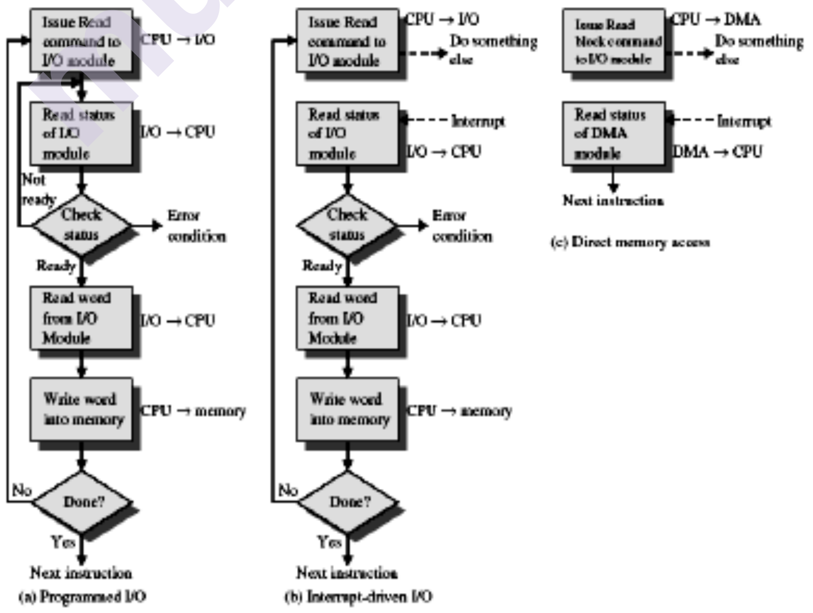
9. I/O Commands- The processor issues an address, specifying I/O Module and device , and an I/O command. The commands are as follows

9.1 Control- Activate a peripheral and tell it what to do.

9.2 Test- Test various status conditions associated with an I/O module ad its peripherals.

9.3 Read- Causes the I/O module to obtain an item of data from the peripheral and place it in to an internal register.

9.4 Write- Causes the I/O module to take a unit of data from the data bus and transmit it to the peripheral



Three techniques for input of a Block of Data

The above figure depicts three techniques by means of which block of data is fetched to a processor. The first technique is programmed I/O where processor executes an I/O instruction by issuing command to appropriate I/O module. The second technique is Interrupt-Driven I/O where the processor does not have to repeatedly check the I/O module status, in order to overcomes the processor having to wait long periods of time for I/O modules. The third techniques is Direct Memory access which is used to eliminate the drawback of programmed and Interrupt-Driven I/O as drawback is I/O transfer rate limited to speed that processor ca test and service devices and processor tied up managing I/O transfers, where as DMA module only uses system bus when processor does not need it to fasten the process of serving an request.

4.1.4 DATA PATH

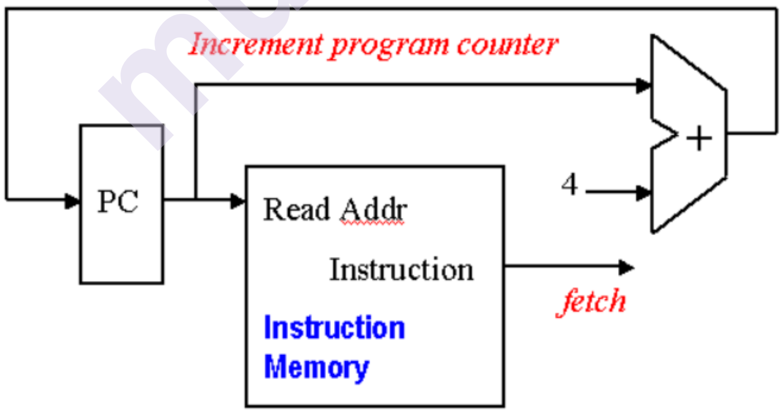
1. The data path is the "brawn" of a processor, since it implements the fetch-decode-execute cycle. The general discipline for data path design is to

   1) Determine the instruction classes and formats in the ISA.

   2) Design datapath components and interconnections for each instruction class or format and

   3) Compose the data path segments designed in step 2 to produce a composite datapath.

2. Datapath comprises of following components like memory(for storing the current instruction), Program Counter (stores the address of current instruction) and ALU for executing the current instruction).



Interaction between components to form a basic data path

3. Types of Data Path

   3.1 R Format- To implement R format instructions only the register file and ALU I required. The ALU accepts its input from the Data Read ports of the register file and the register file is the output of the ALU.for example opcode r1, r2,r3 is the R-format data path instruction

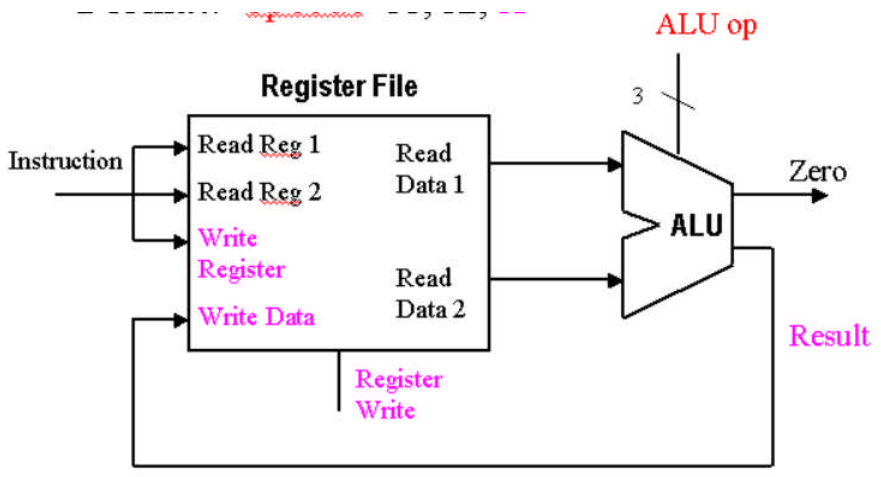
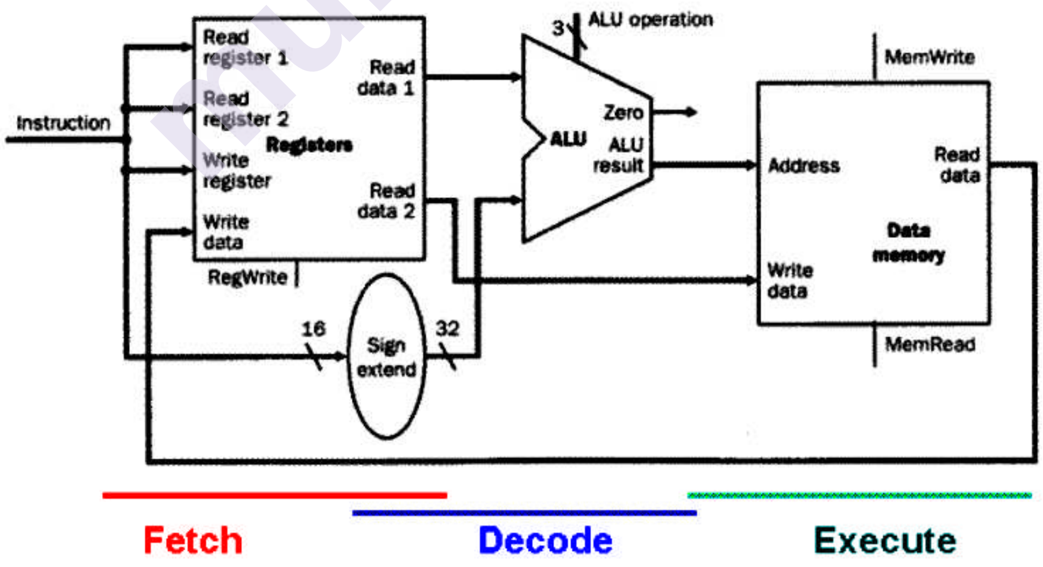


Fig 10 R-format Data Path

3.2 Load/Store Data Path- It performs the following actions in the order given as 1) Register Access takes input from the register file, to implement the instruction data or address fetch of the fetch-decode-execute cycle. 2) Memory Address Calculation- decodes the base address and offset, combining them to produce the actual memory address. This step uses the sign extender and ALU. 3) Read/Write from memory takes data or instruction from the data memory and implements the first part of the execute step of the fetch/decode/execute cycle. 4) Write in to Register File puts data or instructions in to the data memory implementing the second part of the execute step of the fetch/decode/execute cycle.



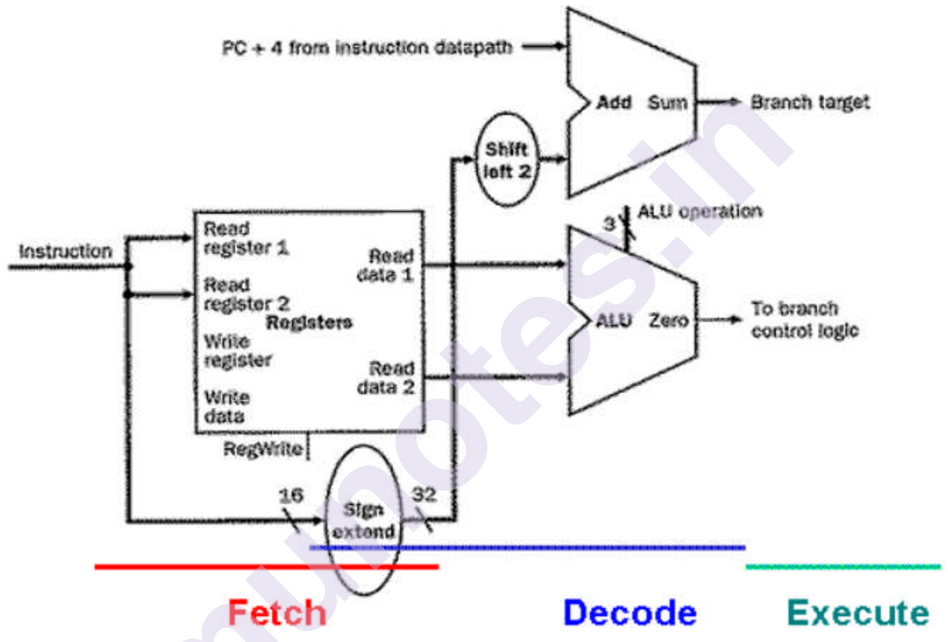
Schematic diagram of Load/Store data path

3.3 Branch/Jump Data-path performs the following actions in the order given-

1) Register Access takes input from the register file, to implement the instruction fetch or data fetch step of the fetch-decode-execute cycle.

2) Calculate Branch Target- It evaluates the branch condition along with it also calculates the branch target address to be ready for the branch if it is taken. This completes the decode step of the fetch-decode-execute cycle.

3) Evaluate Branch Condition and Jump- It determines whether or not the branch should be taken. This effectively changes the PC to the branch target address and completes the executes step of the fetch-decode-execute cycle.



Schematic diagram of the branch instruction data path

4.1.5 INSTRUCTION FETCH AND EXECUTE; EXECUTING ARITHMETIC/LOGIC, MEMORY ACCESS AND BRANCH INSTRUCTIONS

CPU repeatedly performs the following operations such as

1.1 Fetch- The next instruction from memory in to the instruction register.

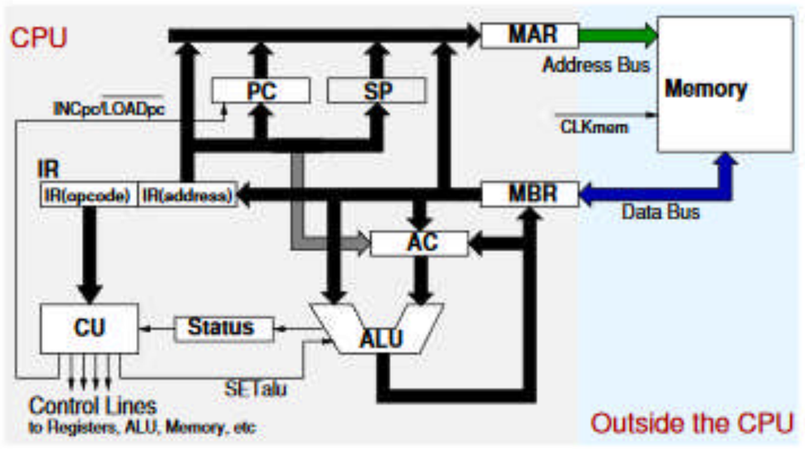
1.2 Decode - The instruction (that is work out which it is).

1.3 Execute the instruction

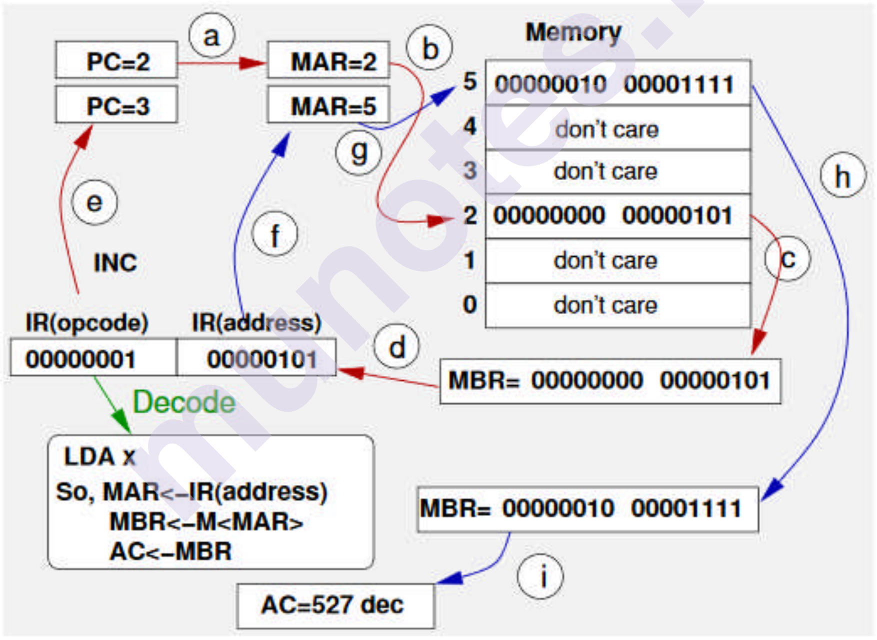
2. Fetching and an Executing an instruction simply require the CPU's Control section to issue levels and pulses which set up pathways and fire register transfers so that. Data is moved from memory to registers

and between registers, Data is passed through the ALU and Data is stuffed back in to the memory.

3. For eg Instruction fetch MAR <- PC, MBR<-(MAR), IR<-MBR, PC+1



Schematic Diagram of Fetch cycle



Fetch and Execute of LDA X Instruction with X=5 and PC=2

4. The above figure shows how fetch and execute of LDA x is executed by CPU and its Registers like Arithmetic logic unit.

4.1 During the fetch MAR<-PC

4.2 Addressing Location 2

4.3 Reading the memory MBR<-(MAR)

4.4 Now the MBR is transferred to the IR

4.5 The last part of the fetch is to increment the PC.

4.6 Decode then first step of execute is MAR<-IR(operand)

4.7 Now addressing location 5

4.8 Reading the memory MBR <- (MAR) again.

4.9 Now transfer to the Accumulator AC <- MBR

5. Execution of Branch Instruction-

For eg JMP X- Branch unconditionally to new location for next instruction. The PC is always incremented during the fetch cycle on the assumption that the next instruction is in the next memory location. This instruction allows an unconditional branching to a non-consecutive instruction.

4.1.6 HARDWIRED AND MICRO-PROGRAMMED CONTROL FOR RISC AND CISC

1. To execute an instruction, there are two types of control units hardwired control unit and micro-programmed control unit.
2. Hardwired control unit are generally faster than micro-programmed designs.
3. A micro-programmed control unit is a relatively simple logic circuit that is capable of 1)Sequencing through micro-instructions and 2) Generating control signals to execute each micro -instructions.
4. The Following Table shows the difference between the Hardwired and Micro-Programmed control unit

Table 1 Difference about RISC and CISC

Sr.No	RISC	CISC
1	Hardwired control unit generates the control signals needed for the processor using logic circuits	Micro-programmed control unit generates the control signals with the help of micro instructions stored in control memory
2	Hardwired control unit is faster when compared to micro programmed control unit as the required control signals are generated with the help of hardwares	This is slower than the other as micro instructions are used for generating signals here
3	Difficult to modify as the control signals that need to be generated are hard wired	Easy to modify as the modification need to be done only at the instruction level

4	More costlier as everything has to be realized in terms of logic gates	Less costlier than hardwired control as only micro instructions are used for generating control signals
5	It cannot handle complex instructions as the circuit design for it becomes complex	It can handle complex instructions
6	Only limited number of instructions are used due to the hardware implementation	Control signals for many instructions can be generated
7	Used in computer that makes use of Reduced Instruction Set Computers(RISC)	Used in computer that makes use of Complex Instruction Set Computers(CISC)

Miscellaneous Questions

Q1. Explain the main components of a processor?

Q2. Difference between Hardwired and Micro-Programmed instructions set?

Q3. Illustrate the concept of Data Path?

Q4. Explain the procedure for instruction fetch and execute?

Q5. Explain the Branch Instruction?

❖❖❖❖

4.2

BASIC INPUT/OUTPUT

Unit Structure

4.2.1 OBJECTIVES

At the end of this unit, the student will be able to

- Describe about the different I/O devices
- Elaborate the concept of data transfer between processor and I/O devices
- Illustrate the concept of Interrupts and Exceptions

4.2.2 INTRODUCTION

1. The I/O subsystem of a computer provides an efficient mode of communication between the central system and the outside environment. It handles all the input-output operations of the computer system.

2. I/O devices that are connected to computer are called peripheral devices. These devices are designed to read information in to or out of the memory unit upon command from the CPU and are designed to be the part of computer system.

3. These devices are also called as peripherals. Below is the three types of peripherals discussed here

   3.1 Input Peripherals-Allows user input, from the outside world to the computer. Eg Keyboard, mouse etc.

3.2 Output Peripherals- Allows information output, from the computer to the outside world. Example printer,monitor etc.

3.3 Input-Output Peripherals- Allows both input(from outside world to computer) as well as output(from computer to the outside world) Example Touch Screen etc.

4. Interface is a shared boundary between two separate components of the computer system which can be used to attach two or more components to the system for communication purposes.

5. So there are two types of interface a)CPU Interface b)I/O Interface.

6. Peripherals connected to a computer need special communication links for interfacing with CPU. In computer system, there are special hardware components between the CPU and peripherals to control or manage the input-output transfers.

7. These components are called input-output interface units because they provide communication links between processor bus and peripherals.

8. They provide a method for transferring information between internal system and input-output devices.

4.2.3 ACCESSING I/O DEVICES

1. Storage is only one of many types of I/O devices within a computer.

2. A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

3. A general purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.

4. Each device controller is in charge of a specific type of device maintains two types of buffer commonly known as Local Buffer Storage and set of special Purpose registers.

5. Typically, operating systems have a device driver for each device controller.

6. This device driver understands the device controller and presents a uniform interface to the device to the rest of the operating system.

7. The following diagram (Fig 1) explains about how an device driver will request for an particular I/O device. To start an I/O operation, the

device driver loads the appropriate registers within the device controller.

7.1 The device controller, in turn examines the contents of these registers to determine what action to take.

7.2 The controller starts the transfer of data from the device to its local buffer.

7.3 Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation.

7.4 The device driver then returns control to the operating system. This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement.

7.5 To solve this problem, Direct Memory Access (DMA) is used. After setting up buffers, pointers and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory with no interruption by the CPU.

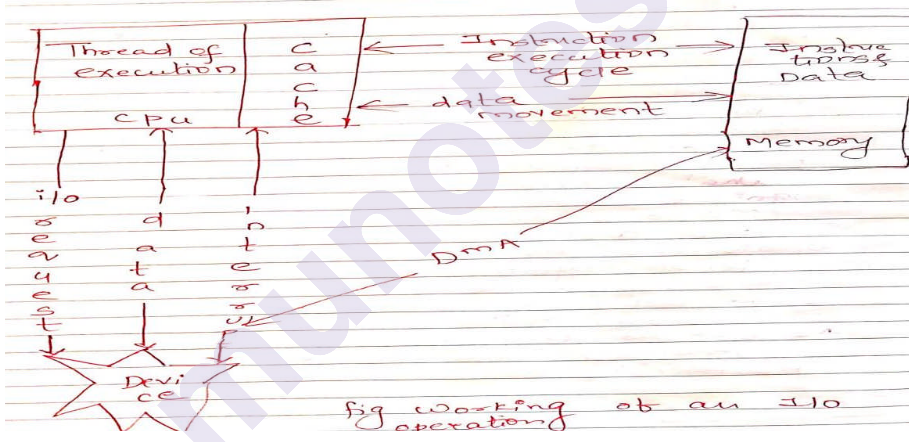


Fig 1 Working of an I/O operation

8. Single-Bus Structure- The bus enables all the devices connected to it to exchange information. Typically, the bus consists of three sets of lines used to carry address, data and control signals. Each I/O device is assigned a unique set of addresses.

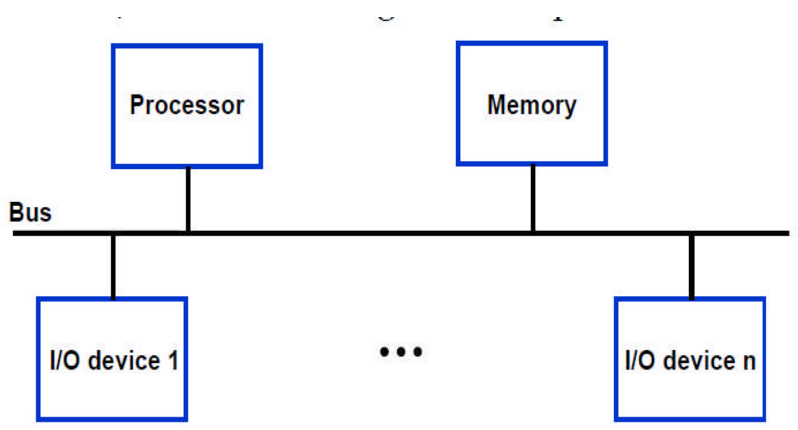


Fig 2 Single-Bus Structure.

9. To access the device appropriately, three data transfer techniques are required such as memory mapped I/O, Programmed I/O, Interrupt Driven and Direct Memory Access(DMA).

4.2.4 DATA TRANSFERS BETWEEN PROCESSOR AND I/O DEVICES

A. Memory Mapped I/O

1 When I/O devices and the memory share the same address space, the arrangement is called memory-mapped I/O.

2. With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device.

3. Most Computer systems use memory-mapped I/O.

4. Some processors have special IN and OUT instructions to perform I/O transfers. When building a computer system based on these processors, the designer has the option of connecting I/O devices to use the Special I/O address space or simply incorporating them as part of the memory address space.

5. The address decoder, the data and the status registers, and the control circuitry required to coordinate I/O transfers constitute the device's Interface circuit.

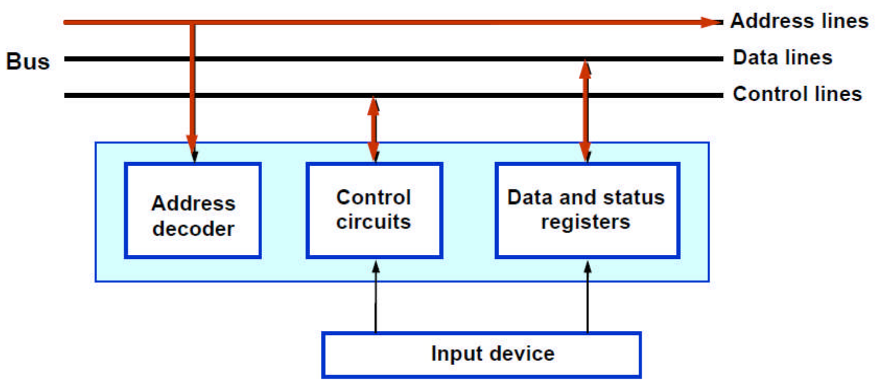


Fig 3 I/O Interface for an Input Device

6. Above figure explains how processor access I/O devices. First of all processor places a particular address on the address lines, it is examined by the address decoders of all devices on the bus. The device that recognizes this address responds to the commands issued on the control lines. The processor uses the control lines to request either a Read or a Write operation, and the requested data are transferred over the data lines. Intel processor uses I/O mapped I/O.

B. Programmed-Controlled I/O

1. Consider a simple example of I/O operations involving a keyboard and a display device in a computer system. The four registers shown below are used in the data transfer operations. The two flags KIRQ and DIRQ in status register are used in conjunction with interrupts.

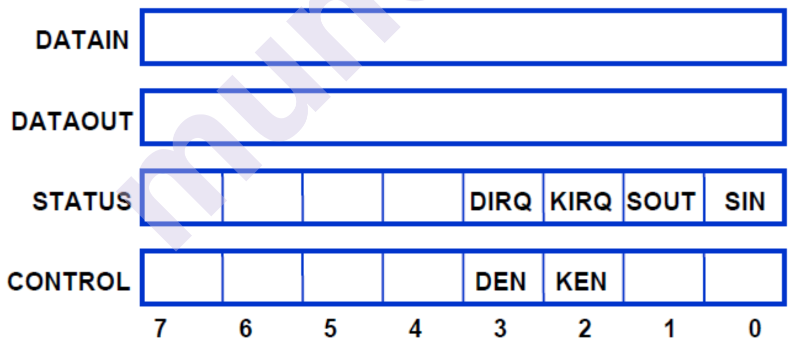


Fig 4 Programmed-Controlled I/O for Keyboard

2. For example A Program that reads on line from the keyboard, stores it in memory buffer and echoes it back to the display

```
        Move       #LINE, R0        Initialize memory pointer
WAITK   TestBit    #0,STATUS        Test SIN
        Branch=0   WAITK            Wait for character to be entered
        Move       DATAIN,R1        Read character
WAITD   TestBit    #1,STATUS        Test SOUT
        Branch=0   WAITD            Wait for display to become ready
        Move       R1,DATAOUT       Send character to display
        Move       R1,(R0)+         Store character and advance pointer
        Compare    #$0D,R1          Check if Carriage Return
        Branch≠0   WAITK            If not, get another character
        Move       #$0A,DATAOUT     Otherwise, send Line Feed
        Call       PROCESS          Call a subroutine to process the
                                    input line
```

Fig 5 Program Snippet about Keyboard Interface

3. The example described above illustrates programmed-controlled I/O, in which the processor repeatedly checks a status flag to achieve the required synchronization between the processor and an input or output device. We say that the processor polls the devices.

4. There are two other commonly used mechanisms for implementing I/O operations: Interrupts and Direct Memory Access.

4.1 Interrupts: Synchronization is achieved by having the I/O device send a special signal over the bus whenever it is ready for a data transfer operation.

4.2 Direct Memory Access: It involves having the device interface transfer data directly to or from the memory.

C Interrupt Driven I/O

1. To avoid the processor being not performing any useful computation, a hardware signal called an interrupt to the processor can do it. At least one of the bus control lines, called an interrupt-request line, is usually dedicated for this purpose.

2. An interrupt- service routine usually is needed and is executed when an interrupt request is issued.

3. On the other hand, the processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal. An interrupt-acknowledge signal serve this function.

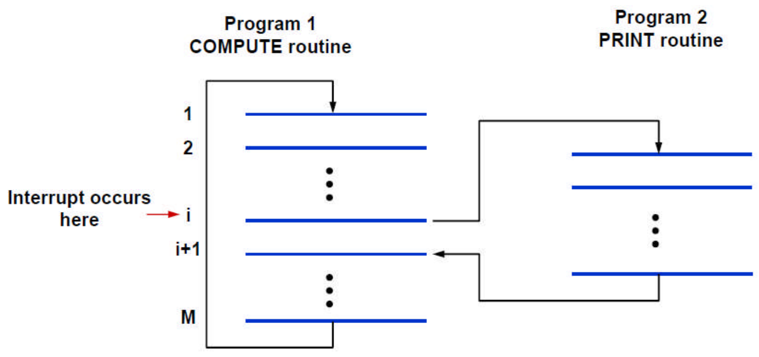


Fig 5 Example of Interrupt Driven I/O

4. Treatment of an interrupt-service routine is very similar to that of subroutine. An important departure from the similarity should be noted. A subroutine performs a function required by the program from which it is called. The interrupt-service routine may not have anything in common with the program being executed at the time the interrupt request is received. In fact, the two programs often belong to different users.
5. Before executing the interrupt-service routine, any information that may be altered during the execution of that routine must be saved. This information must be restored before the interrupted program is resumed.
6. The information that needs to be saved and restored typically includes the condition code flags and the contents of any registers used by both the interrupted program and the interrupt-service routine.
7. Saving registers also increases the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine. The delay is called interrupt latency.
8. Typically, the processor saves only the contents of the program counter and the processor status register. Any additional information that needs to be saved must be saved by program instruction at the beginning of the interrupt-service routine and restored at the end of the routine.
9. An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

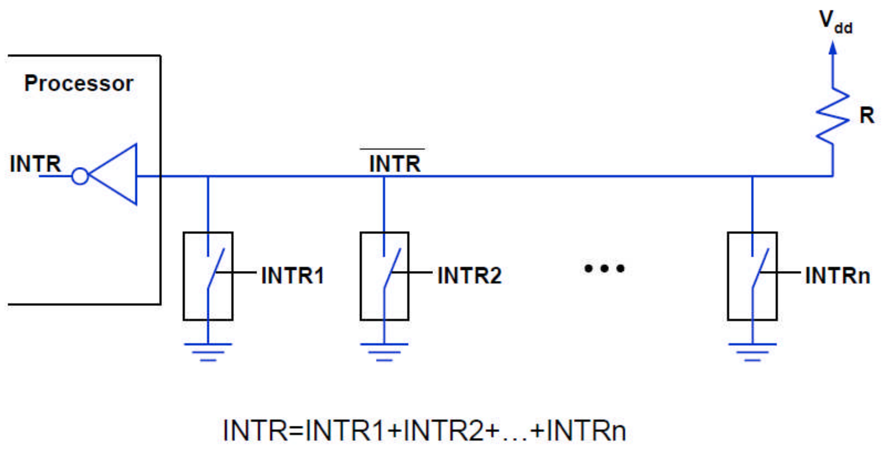


Fig 6 Interrupt Hardware Design

10. Handling Multiple Devices gives rise to a many questions

10.1 How can the processor recognize the device requesting an interrupt?

10.2 Given that different devices are likely to require different interrupt-service routines, how can the processor obtain the starting address of the appropriate routine in each case?

10.3 Should a device be allowed to interrupt request be handled?

11. The information needed to determine whether a device is requesting an interrupt is available in its status register. When a device raises an interrupt request, it sets to one of the bits in its status register, which is called IRQ bit.

12. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all the I/O devices connected to the bus. The polling scheme is easy to implement. Its main disadvantages is the time spent interrogating all the devices.

13. A device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine. This is called vectored interrupts.

14. An interrupt request from a high-priority device should be accepted while the processor is servicing another request from a lower-priority device.

15. The processor's priority is usually encoded in a few bits of the processor status word. It can be changed by program instructions that write in to the program status register(PS). These are privileged instructions which can be executed only while the processor is running in the supervisor mode.

16. The processor is in the supervisor mode only when executing operating system routines. It switches to the user mode before beginning to execute application program,

17. An attempt to execute a privileged instruction while in the user mode leads to a special type of interrupt called a privilege exception.

18. An example of the implementation of a multiple-priority scheme

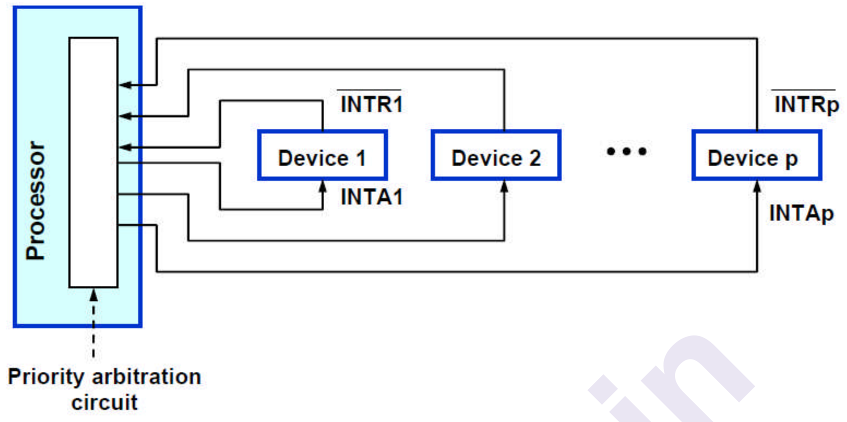


Fig 7 Interrupt Priority arbitration circuit

19. Above figure illustrates when there is arrival of two or more request from different devices for interrupt so priority arbitration circuit will decide which request to serve first.

20. To serve request more accurately priority arbitration circuit is used with daisy chain

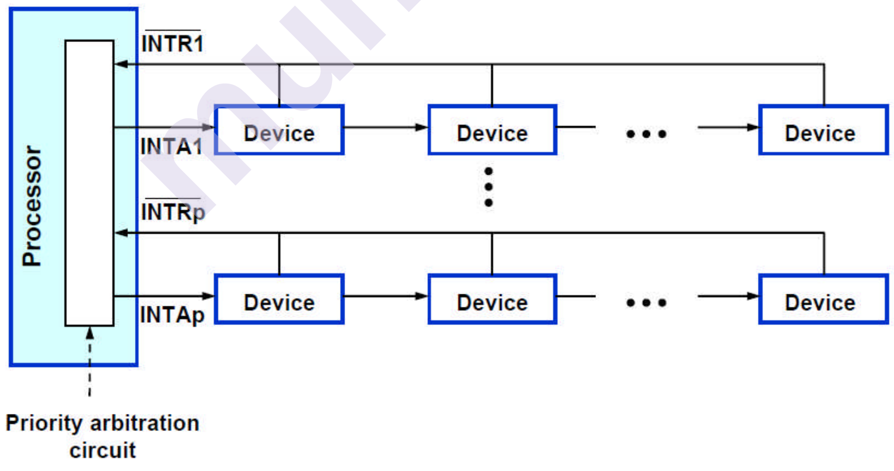


Fig 7 Interrupt Priority with daisy chain

D. Direct Memory Access

1. To transfer large blocks of data at high speed, a special control unit may be provided between an external device and the main memory, without continuous intervention by the processor. This approach is called direct memory access (DMA).

2. DMA transfers are performed by a control circuit that is part of the I/O device interface. We refer to this circuit as a DMA controller.

3. Since it has to transfer blocks of data, the DMA controller must increment the memory address for successive words and keep track of the number of transfers.

4. Although a DMA controller can transfer data without intervention by the processor, its operation must be under the control of a program executed by the processor.

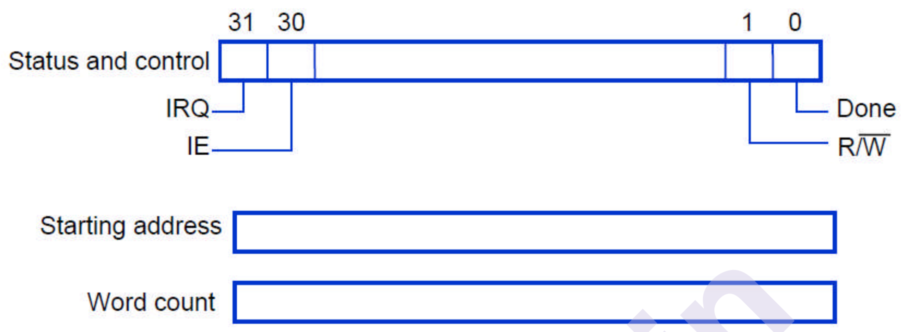


Fig 8 DMA Controller

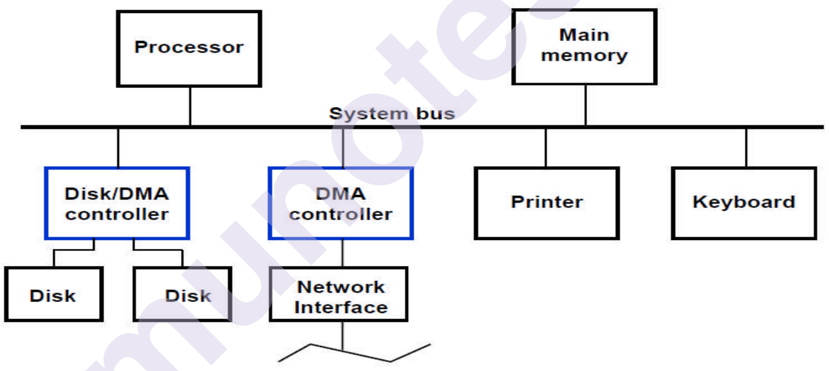


Fig 9 DMA controller in a Computer system

5. Memory accesses by the processor and the DMA controllers are interwoven. Request by DMA devices for using the bus are always given higher priority than processor requests.

6. Among different DMA devices, top priority is given to high-speed peripherals such as disk, a high-speed network interface etc.

7. Since the processor originates most memory access cycles, the DMA controller can be said to "steal" memory cycles from the processor. Hence, this interweaving technique is usually called cycle stealing.

8. The DMA controller may transfer a block of data without interruption. This is called block/burst mode.

9. A conflict may arise if both the processor and a DMA controller or two DMA controller try to use the bus at the same time to access the main

memory. To resolve this problem, an arbitration procedure on bus is needed.

10. The device that is allowed to initiate data transfer on the bus at any given time is called the bus master. When the current master releases control of the bus, another device can acquire this status.

11. Bus arbitration is the process by which the next device to become the bus master take in to account the needs of various devices by establishing a priority system for gaining access to the bus.

12. There are two approaches to bus arbitration- Centralized and Distributed

13. In centralized arbitration, a single bus arbiter performs the required arbitration.

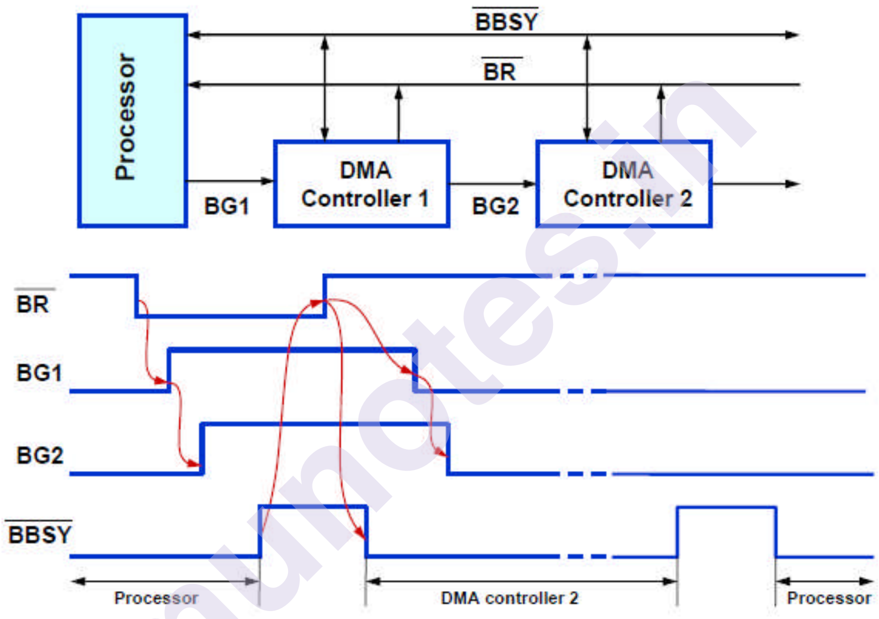


Fig 10 Centralized Arbitration

14. In distributed arbitration, all devices participate in the selection of the next bus master.

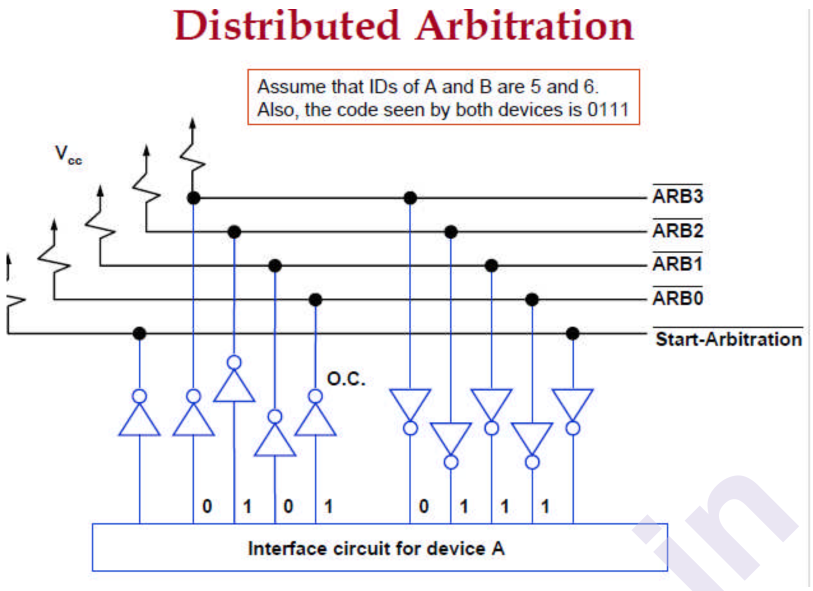

Distributed Arbitration
Assume that IDs of A and B are 5 and 6.
Also, the code seen by both devices is 0111
Vcc
ARB3
ARB2
ARB1
ARB0
Start-Arbitration
O.C.
0
1
0
1
0
1
1
1
Interface circuit for device A

Fig 11 Distributed Arbitration

15. A bus protocol is the set of rules that govern the behavior of various devices connected to the bus as to when to place information on the bus, assert control signals, and so on.

16. In a synchronous bus, all devices derive timing information from a common clock line. Equal spaced pulses on this line define equal time intervals.

17. In the simplest form of a synchronous bus, each of these intervals, constitutes a bus cycle during which one data transfer can take place.

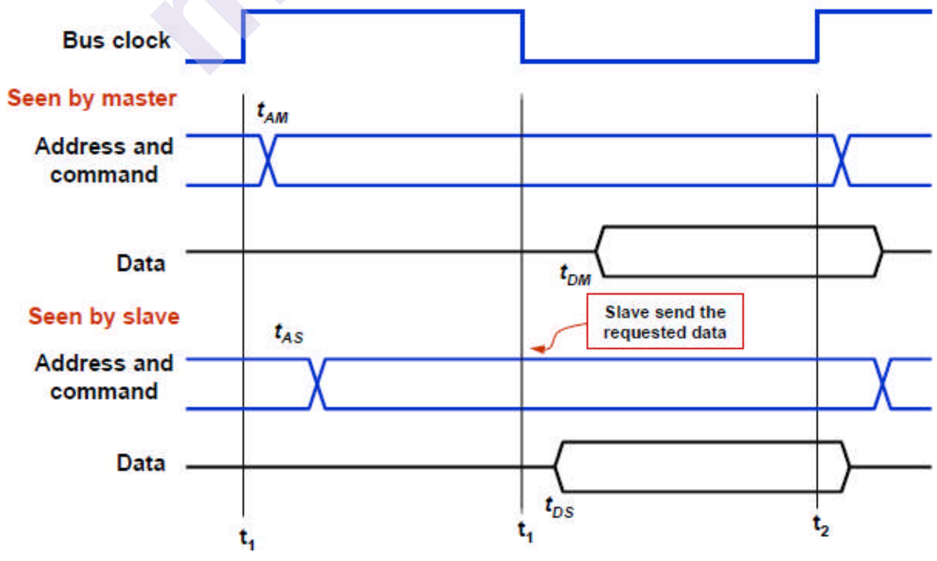


Fig 12 Synchronous Bus Example

18. An alternative scheme for controlling data transfers on the bus is based on the use of a handshake between the master and slave.

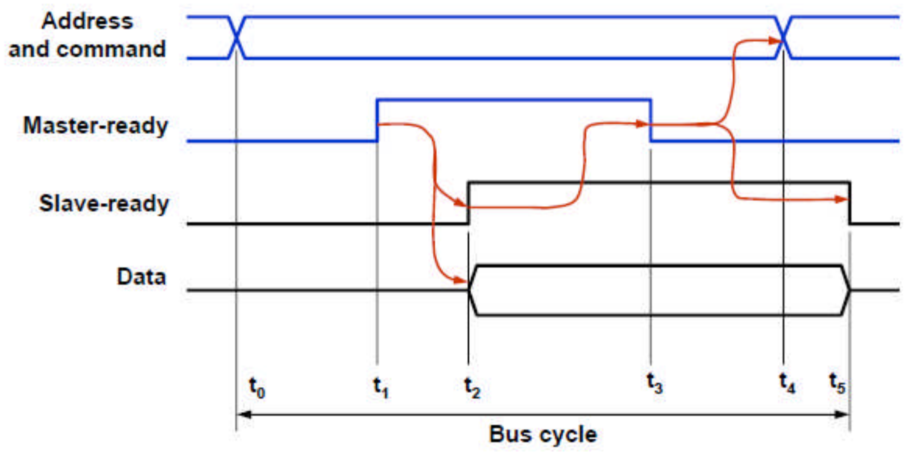


Fig 13 Asynchronous bus example

19. The choice of a particular design involves trade-offs among factors such as simplicity of the device interface, ability to accommodate device interfaces that introduce different amounts of delay, total time required for bus transfer , ability to detect errors results from addressing a non-existent device or from an interface malfunction.

20. Asynchronous bus- The handshake process eliminates the need for synchronization of the sender and receiver clock, thus simplifying timing design

21. Synchronous bus-Clock Circuitry must be designed carefully to ensure proper synchronization and delays must be kept within strict bounds.

4.2.5 INTERRUPTS AND EXCEPTIONS: INTERRUPT REQUESTS AND PROCESSING

1. Exceptions and interrupts are unexpected events that disrupt the normal flow of instruction. An exception is an unexpected event from within the processor. An interrupt is an unexpected event from outside the processor.

2. When an exception or interrupt occurs, the hardware begins executing code that performs an action in response to the exception. This action may involve killing a process, outputting a error message, communicating with an eternal device or horribly crashing the entire computer system by initiating a “Blue screen of Death” and halting the CPU. The instruction responsible for this action reside in the operating system kernel and the code that performs this action is called the interrupt handler code.

3. Exception Types

Sr.No	Exception Type	Explanation
1	Arithmetic Overflow	Occurs during the execution of an add or sub instruction. If the result of the computation is too large or too small to hold in the result register, the overflow output of the ALU will become high during the execute state. This event triggers an exception.
2	Undefined instruction	Occurs when an unknown instruction is fetched. This exception is caused by an instruction in the IR that has an unknown opcode or an R-type instruction that has an unknown function code.
3	System Call	Occurs when the processor executes a syscall instruction. Syscall instructions are used to implement operating system services(functions)

4. Interrupt Request (IRQ) Pin is the first pin which will allow an external device to interrupt to the processor. Since the processor don't want to service any external interrupts before it is finished executing the current instruction, we may have to make the external device wait for several clock cycles. Because of this, we need a way to tell the eternal device that we have serviced this interrupt. So this problem be solved by adding a second pin known as IACK(Interrupt acknowledge), that will be an output.

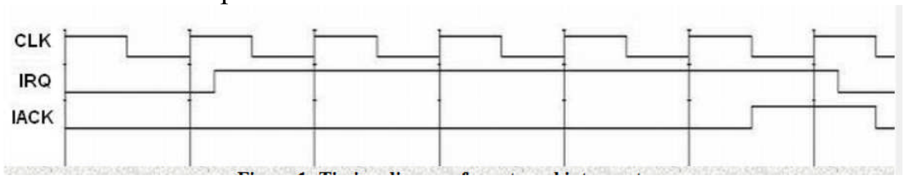


Fig 14 Timing Diagram for external interrupt

5. When an exception or interrupt occurs, the processor may perform the following actions:

5.1 Move the current PC in to another register, call the EPC.

5.2 Record the reason for the exception in the cause register.

5.3 Automatically disable further interrupts or exceptions from occuring, by left-shifting the status register.

5.4 Change control (jump) to a hardwired exception handler address.

5.5 To return from a handler, the processor may perform the following actions:

Move the contents of the EPC register to the PC.

Re-enable interrupts and exceptions, by right-shifting the status register.

6. When multiple types of exceptions and interrupts can occur, there must be a mechanism in place where different handler code can be executed for different types of events. So there are two methods to handle this problem

6.1 Polled interrupt- The processor can branch to a certain address that begins a sequence of instructions that check the cause of the exception and branch to handler code for the type of exception encountered.

6.2 Vectored interrupt-The processor can branch to a different address for each type of exception. Each exception address is separated by only one word. A jump instruction is placed at each of these addresses that forces the processor to jump to the handler code for each type of exception.

Miscellaneous Questions

Q1. Explain the operation of DMA

Q2. Explain the Input-operation

Q3. Illustrate about the different data transfer techniques used in CPU

Q4. Illustrate the concept of Exception and Interrupt Request

Q5. Explain the concept of Single-bus Structure

❖❖❖❖