COMPUTER ORGANIZATION

(DMCA103)

(MCA)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

Chapter 1

Digital Logic Circuits

Many scientific, industrial and commercial advances have been made possible by the advent of computers. Digital Logic Circuits form the basis of any digital (computer) system. In this topic, we will study the essential features of digital logic circuits, which are at the heart of digital computers. Digital Logic circuits may be subdivided into Combinational Logic Circuits and Sequential Logic Circuits.

1.1 Logic Gates

1.1.1 AND gate

AND Operation:AND operation is represented by C = A • B

Its associated TRUTH TABLE is shown below. A truth table gives the value of output variable (here C) for all combinations of input variable values (here A and B). Thus in an AND operation, the output will be 1 (True) only if all of the inputs are 1 (True).

2-input AND gate

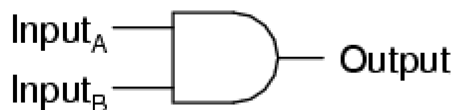


Fig. 1.1 AND Gate

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Table: 1.1 Truth table for AND gate

The following relationships can be easily derived from this circuit:

A.A = A
1.A = A
0.A = 0
A.Ā = 0
A.B = B.
A A.(B.C) = (A.B).C = A.B.C

1.1.2 OR gate

OR Operation: OR operation is represented by C = A + B

Here A, B & C are logical (Boolean) variables and the + sign represents the logical addition, called an 'OR' operation. The symbol for the operation (called an OR gate) is shown in Fig. 4. Its associated TRUTH TABLE is shown below. Thus in an OR operation, the output will be 1 (True) if either of the inputs is 1 (True). If both inputs are 0 (False), only then the output will be 0 (False). Notice that though the symbol + is used, the logical

addition described above does not follow the rules of normal arithmetic addition.

2-input OR gate

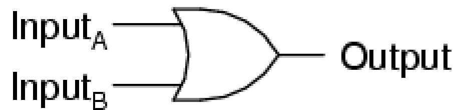


Fig.1.2 OR Gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Table: 1.2 Truth table for OR gate

The following relationships can be easily derived from this circuit:

$$A+\bar{A} = 1$$
$$A+A = A$$
$$0+A = A$$
$$1+A = 1$$

$$(A+B)+C=A+(B+C)=A+B+C$$

1.1.3 NOT Operation

NOT operation is represented by $C = \bar{A}$

The NOT gate has only one input which is then inverted by the gate. Here, *A* is the 'complement' of A. The symbol and truth table for the operation are shown below:

Inverter, or NOT gate

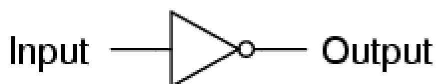


Fig.1.3 NOT Gate

Truth table

Input	Output
0	1
1	0

Table: 1.3 Truth table for NOT gate

1.1.4 NAND Gate:

We could combine AND and NOT operations together to form a NAND gate. Thus the logical expression for a NAND gate is

$$C = (A \cdot B)'$$

The symbol and truth table are given in the following figure. The NAND gate symbol is given by an AND gate symbol with a circle at the output to indicate the inverting operation.

2-input NAND gate

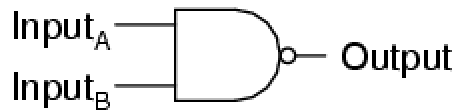


Fig.1.4 NAND Gate

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Table: 1.4 Truth Table for NAND gate

1.1.5 NOR Gate:

Similarly, OR and NOT gates could be combined to form a NOR gate.

2-input NOR gate

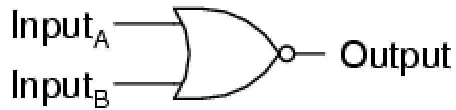


Fig 1.5 NOR Gate

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Table: 1.5 Truth Table for NOR gate

1.1.6 Exclusive OR Operation:

In logic circuits, exclusive OR operation is represented as shown below.

$$C = A \oplus B$$

Fig 1.6 Exclusive OR Gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table: 1.6 Truth Table for Exclusive-OR gate

1.2 Boolean Algebra

The operation of almost all modern digital computers is based on two-valued or binary systems. Binary systems were known in the ancient Chinese civilization and by the classical Greek philosophers who created a well-structured binary system, called propositional logic. Propositions may be TRUE or FALSE, and are stated as functions of other propositions which are connected by the three basic logical connectives: AND, OR, and NOT. For example the statement:

"I will take an umbrella with me if it is raining or the weather forecast is bad"

connects the proposition I will take an umbrella with me functionally to the two propositions it is raining and the weather forecast is bad. We can see that the umbrella proposition can be fully determined by the raining and weather ones. In functional terms we can be consider the truth value of the umbrella proposition as the output of the truth values of the other two. We can represent this by means of a simple block diagram

The meaning of the OR connective is that the corresponding output is TRUE if either one of the input propositions is TRUE, otherwise it is FALSE. Since there are only two possible values for any proposition, we can easily calculate a truth value for I will take an umbrella for all possible input conditions. This produces the

Truth Table of the basic OR function:

Raining	Bad Forecast	Umbrella
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

Table: 1.7

We can make the propositions as complex as we require. For example, if we want to include the proposition I will take the car, we may make a statement such as: "If I do not take the car then I will take the umbrella if it is raining or the weather forecast is bad". However, to find the correct block diagram we have to state the

proposition in a well-structured way using brackets to indicate how the proposition is composed. The correct representation is:

(Take Umbrella) = (NOT (Take Car)) AND ((Bad Forecast) OR (Raining))

Notice that we have changed the IF verbal construction into an equation with binary variables. The block diagram is shown in Figure 2. To simplify the handling of complex binary connectives, the mathematician George Boole developed Boolean Algebra in the last century, using

ordinary algebraic notation, and 1 for TRUE and 0 for FALSE. In this course we will use the symbol _ for the AND and + for the OR connectives which we call Boolean operators. The NOT operator, which is unary, we will denote with a post fix prime, eg A0 means NOT A. (Alternatives that you may see in books are ^ for AND, *V* for OR, and either over-score or prefix ¬ for NOT). Sometimes, when the meaning is clear from the context, we may omit the AND symbol. Using the values 1 for TRUE and 0 for FALSE the truth tables of the three basic operators are as follows.

AND ·

A	B	R
0	0	0
0	1	0
1	0	0
1	1	1

OR +

A	B	R
0	0	0
0	1	1
1	0	1
1	1	1

NOT'

A	R
0	1
1	0

Boolean operations are carried out in a well-defined order or "precedence", which is defined as follows:

Operator	Symbol	Precedence
NOT	'	Highest
AND	·	Middle
OR	+	Lowest

Table: 1.8

Expressions inside brackets are always evaluated first, overriding the precedence order. The Boolean equation of the block diagram (Figure 2) in fully bracketed form is given by:

U = ((C0) _ ((W) + (R)))

By taking advantage of the precedence rules, we can simplify it by removing brackets:

U = C0 _ (W + R)

We can use the basic truth tables for AND, OR and NOT to evaluate the overall truth table of a more complex expression. For example, to find out whether we should take our umbrella or not we can evaluate the overall truth of the proposition given in the above equation for every possible input combination. We shall call this the Truth Table Method. In this case, there are eight possible different combinations of input values since there are three independent inputs and $8 = 2^3$.

R W C	$X_1 = R + W$	$X_2 = C_0$	$U = X_1 _ X_2$
0 0 0	0	1	0
0 0 1	0	0	0
0 1 0	1	1	1
0 1 1	1	0	0
1 0 0	1	1	1
1 0 1	1	0	0
1 1 0	1	1	1
1 1 1	1	0	0

Table: 1.9

Like all algebras, there are rules to manipulate Boolean expressions. The most simple are the rules that concern the unary operator NOT:

$$(A')' = A$$

$$A \cdot A' = 0$$

$$A + A' = 1$$

General rules like the distributive, commutative, and associative rules hold for the AND and OR binary operators as follows.

Associative $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
$(A + B) + C = A + (B + C)$

Commutative $A \cdot B = B \cdot A$
$A + B = B + A$

Distributive $A \cdot (B + C) = A \cdot B + A \cdot C$
$A + (B \cdot C) = (A + B) \cdot (A + C)$ (the weird one!)

In addition, there are simplification rules for Boolean equations. There are three important groups of simplification rules. The first one uses just one variable:

$$A \cdot A = A$$
$$A + A = A$$

The second group uses Boolean constants 0 and 1:

$$A \cdot 0 = 0$$
$$A \cdot 1 = A$$
$$A + 0 = A$$
$$A + 1 = 1$$

The third group involves two or more variables and contains a large number of possible simplification rules (or theorems) such as:

$A + A \cdot (B) = A$ (proof: $A + A \cdot B = A \cdot (1 + B) = A \cdot 1 = A$)

Note that in this expression either A or B may stand for any complex Boolean expression.

There are two important rules which constitute de Morgan's theorem:

$$(A + B)' = A' \cdot B'$$
$$(A \cdot B)' = A' + B'$$

This theorem is widely used in Boolean logic design. The theorem holds for any number of terms, so:

$$(A + B + C)' = ((A + B) + C)' = ((A + B)') \cdot C' = A' \cdot B' \cdot C'$$

and similarly:

$$(A \cdot B \cdot C \cdot \ldots \cdot X) = A' + B' + C' + \ldots + X'$$

You may have noticed by now that rules are often given in pairs. It makes sense that in a binary system there is some kind of symmetry between the two operators. For Boolean algebra this symmetry is called duality. Every equation has its dual which one

can generate by replacing the AND operators with ORs (and vice versa) and the constants 0 with 1s (and vice versa).
For example, the dual equation of the important simplifying rule:

$$A + A \cdot B = A$$

is:

$$A \cdot (A + B) = A \text{ (proof: } A \cdot A + A \cdot B = A + A \cdot B = A \text{)}$$

Do not mix up or get confused between a dual expression which is generated by the above rules and the complement (or inverted) expression which is generated by applying the NOT operator. The rules are similar, but they mean very different things.

Finally, let us simplify the proposition I am not taking an umbrella.

$(U)' = (C' \cdot (W + R))'$

apply de Morgan's theorem $U' = (C')' + (W + R)'$

apply de Morgan's theorem again $U' = (C')' + W' \cdot R'$

and simplify $U' = C + W' \cdot R'$

1.3 Map Simplification

1.3.1 Karnaugh Maps

From the previous examples we can see that rules of Boolean algebra can be applied in order to simplify expressions. Apart from being laborious (and requiring us to remember all the laws) this method can lead to solutions which, though they appear minimal, are not. The Karnaugh map provides a simple and straight-forward method of minimising boolean expressions. With the Karnaugh map Boolean expressions having up to four and even six variables can be simplified easily. The simplified logical expression is then used so that minimum hardware

is employed in the implementation of logical circuits. A Karnaugh map provides a pictorial method of grouping together expressions with common factors andtherefore eliminating unwanted variables. The values inside the squares are copied from the output column

of the truth table, therefore there is one square in the map for every row in the truth table. Around the edge of the Karnaugh map are the values of the two input variable. A is along the top and B is down the left hand side. The diagram below explains this.

Boolean Expressions in Two Variables:

Consider the following truth table.

A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

Table: 1.10

The logical expression X is given by $X=\bar{A}.\bar{B}+A.B$

The Karnaugh map of the above truth table is shown in the following figure. The values inside the squares are copied from the output column of the truth table, therefore there is one square in the map for every row in the truth table. Around the edge of the Karnaugh map are the values of the two input variable A and B and their inverses.

In other words, we may say that Karnaugh map is a graphical representation of the truth table.

	A	$\bar{A}$
B	1	0
$\bar{B}$	0	1

Fig:1.7 Map simplification for two variables

Consider the logical expression $Y = A.B + \bar{A}.B$

Its Karnaugh map is shown below. The two adjacent squares may be combined together as shown by the loop.

	A	$\bar{A}$
B	1	1
$\bar{B}$	0	0

Fig:1.8 Map simplification for two variables

Referring to the map above, the two adjacent 1's are grouped together. Through inspection it can be seen that variable A has its true and false form within the group. This eliminates variable A leaving only variable B which only has its true form. The minimised answer therefore is Y = B.

It simply means that we are combining the two terms of the above expression Y as shown below:

$$Y = B(A + \bar{A}) = B$$

Therefore, as the variable A changes from its normal form to its inverse form ($\bar{A}$) when we move from one square to the adjacent one, the simplified expression of Y will be independent of A.

Taking another example, the expression $Z=\bar{A}.B+A.B+A.\bar{B}$

is simplified as follows.

	A	A
B	1	1
$\bar{B}$	1	0

Fig:1.9 Map simplification for two variables

First, combining the two adjacent squares in row 1, we get *B*.
Next, combining the two adjacent squares in column 1, we get *A*.
Hence, we get *Z* as shown below:

$$Z = B + A$$

Therefore we can easily conclude that, combining two adjacent squares in Karnaugh map eliminates one variable from the resulting Boolean expression of the corresponding squares.

Boolean Expressions in Three Variables:

Consider the following truth table.

A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Table: 1.11

The corresponding Boolean expression using SOP is:

$$X=\bar{A}.\bar{B}.\bar{C}+\bar{A}.\bar{B}.C+\bar{A}.B.\bar{C}+A.B.\bar{C}$$

Figure below shows the Karnaugh map of the above truth table. The expression X may be simplified by combining two adjacent squares as shown.

	AB	$A\bar{B}$	$\bar{A}\bar{B}$	$\bar{A}B$
C	0	0	1	0
$\bar{C}$	1	0	1	1

Fig: 1.10 Map simplification for three variables

The simplified expression of X is: $X=\bar{A}.\bar{B}+B.\bar{C}$

Now, consider another expression *Y* given below:

$$Y=A.B.\bar{C}+A.\bar{B}.\bar{C}+\bar{A}.\bar{B}.\bar{C}+\bar{A}.B.\bar{C}$$

The Karnaugh map of Y is shown below. In this case, we are able to combine four adjacent squares. Note that

	AB	$A\bar{B}$	$\bar{A}\bar{B}$	$\bar{A}B$
C	0	0	0	0
$\bar{C}$	1	1	1	1

Fig: 1.11 Map simplification for three variables

Y can also be obtained as:

$$Y=A.\bar{C}(B+\bar{B})+\bar{A}.\bar{C}(\bar{B}+B)$$

$$=A.\bar{C}+\bar{A}.\bar{C}=\bar{C}$$

Consider another example:

$$Z=A.\bar{B}.C+\bar{A}.\bar{B}.C+A.\bar{B}.\bar{C}+\bar{A}.\bar{B}.\bar{C}$$

The corresponding Karnaugh map is shown below:

	AB	$A\bar{B}$	$\bar{A}\bar{B}$	$\bar{A}B$
C	0	1	1	0
$\bar{C}$	0	1	1	0

Fig: 1.12 Map simplification for three variables

In this case also, we are able to combine 4 adjacent squares. Note that

Combining the two adjacent squares in columns 2 and 3 of row 1, the variable A gets eliminated, and we are left with $Z_1=\bar{B}.C$

Combining the two adjacent squares in columns 2 and 3 of row 2, the variable A gets eliminated and we are left with $Z_2= \bar{B}.\bar{C}$

Combining these two expressions, $Z= \bar{B}.C+ \bar{B}.\bar{C}=B$

Finally, let us consider another expression W below:

$$W=A.B.C+ \bar{A}.B.C+A.B.\bar{C}+\bar{A}.B.\bar{C}$$

The Karnaugh map of *W* is shown below. Note that the resulting expression should be independent of *A* and *C*. So, *W* is simplified as: $W = B$

	AB	$A\overline{B}$	$\overline{A}\overline{B}$	$\overline{A}B$
C	1	0	0	1
$\overline{C}$	1	0	0	1

Fig: 1.13 Map simplification for three variables

Therefore, we can conclude that combining four adjacent squares in Karnaugh map eliminates two variables from the resulting Boolean expression of the corresponding squares.

Boolean Expressions in Four Variables

Knowing how to generate Gray code should allow us to build larger maps. Actually, all we need to do is look at the left to right sequence across the top of the 3-variable map, follow a similar sequence for the other two variables and write it down on the left side of the 4-variable map.

Karnaugh map of four variables *A*, *B*, *C* and *D* is shown in the following figure. As we have shown in the previous examples, we may easily prove that:

Combining eight adjacent squares in Karnaugh map eliminates three variables from the resulting Boolean expression of the corresponding squares.

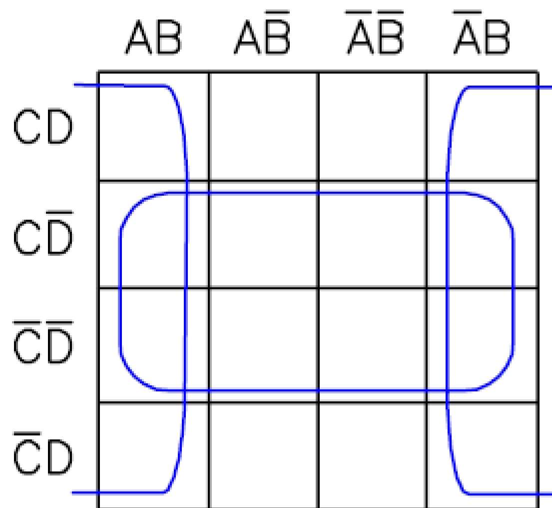


Fig: 1.14 Map simplification for four variables

Example: Simplify the Boolean expression:

$$X = \bar{A}.\bar{B}.\bar{C}.D + \bar{A}.\bar{B}.C.D + \bar{A}.B.\bar{C}.D + \bar{A}.B.C.D$$

Karnaugh map of *X* is shown in Figure 12.24. As we could combine 4 adjacent squares as shown below, the simplified expression should be independent of two variables. Adjacent squares in a row suggest that the resultant expression should be independent of *B*. Similarly, adjacent squares in a column suggest that it should also be independent of *C*. Hence, the simplified expression of *X* is given by : $X = \bar{A}.D$

	AB	$A\bar{B}$	$\bar{A}\bar{B}$	$\bar{A}B$
CD	0	0	1	1
$C\bar{D}$	0	0	0	0
$\bar{C}\bar{D}$	0	0	0	0
$\bar{C}D$	0	0	1	1

Fig: 1.15 Map simplification for four variables

Example: Simplify the following Boolean expression using Karnaugh map.

$$Y=A.B.\bar{C}.D+A.\bar{B}.\bar{C}.D+\bar{A}.\bar{B}.\bar{C}.D+\bar{A}.B.\bar{C}.D+\bar{A}.\bar{B}.\bar{C}.\bar{D}+\bar{A}.B.\bar{C}.\bar{D}+$$
$$\bar{A}.B.C.\bar{D}+\bar{A}.B.C.D+\bar{A}.\bar{B}.C.D$$

Solution: Karnaugh map of *Y* is shown below. There are four loops enclosing 4-adjacent squares. First, consider the loop 1. The resulting expression for these squares should be independent of *C* and *D*. Next, consider loop 2. The resulting expression of these squares should be independent of *B* and *D*. Thirdly, consider loop 3. The resulting expression of these squares should be independent of *B* and *C*. Finally, consider loop 4.

The resulting expression of these squares should be independent of *A* and *B*. Hence, we get

$$Y=\bar{A}.B+\bar{A}.\bar{C}+\bar{A}.D+\bar{C}.D$$

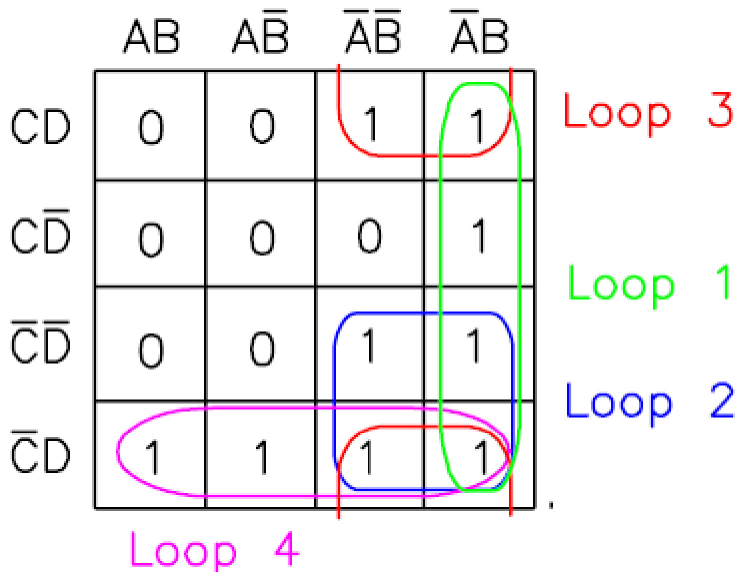


Fig: 1.16 Map simplification for four variables

Reductions could be done with Boolean algebra. However, the Karnaugh map is faster and easier, especially if there are many logic reductions to do.

Karnaugh maps: Complete Simplification Process

1. Draw out the pattern of output 1's and 0's in a matrix of input values

2. Construct the K map and place 1s and 0s in the squares according to the truth table.

3. Group the isolated 1s which are *not* adjacent to any other 1s. (single loops)

4. Group any pair which contains a 1 adjacent to only one other 1. (double loops)

5. Group any quad that contains one or more 1s that have not already been grouped, *making*

sure to use the minimum number of groups.

6. Group any pairs necessary to include any 1s that have not yet been grouped, *making sure to*

use the minimum number of groups.

7. Form the OR sum of all the terms generated by each group.

Compared to the algebraic method, the K-map process is a more orderly process requiring fewer steps and always

producing a minimum expression. It must be noted that the minimum expression is generally **NOT** unique.

1.4 Combinational Logic Circuits

Combinational circuit is a circuit in which we combine the different gates in the circuit, for example encoder, decoder, multiplexer and demultiplexer. Some of the characteristics of combinational circuits are following −

- The output of combinational circuit at any instant of time, depends only on the levels present at input terminals.
- The combinational circuit do not use any memory. The previous state of input does not have any effect on the present state of the circuit.
- A combinational circuit can have an n number of inputs and m number of outputs.

Fig: 1.17 Combinational circuit

1.4.1 Half Adder

Half adder is a combinational logic circuit with two inputs and two outputs. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of two **single** bit numbers. This circuit has two outputs **carry** and **sum.**

Block diagram:

Fig: 1.18 Half Adder

Inputs		Output	
A	B	S	C
0	0	0	0
0	1	1	0
0	1	1	0
1	1	0	1

Table: 1.12 Truth table for half adder

Circuit Diagram

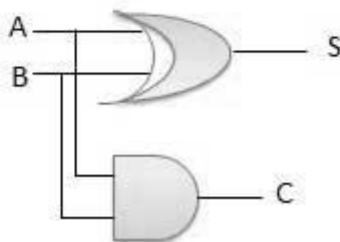


Fig: 1.19 Circuit diagram for half adder

1.4.2 Full Adder

Full adder is developed to overcome the drawback of Half Adder circuit. It can add two one-bit numbers A and B, and carry c. The full adder is a three input and two output combinational circuit.

Block diagram:

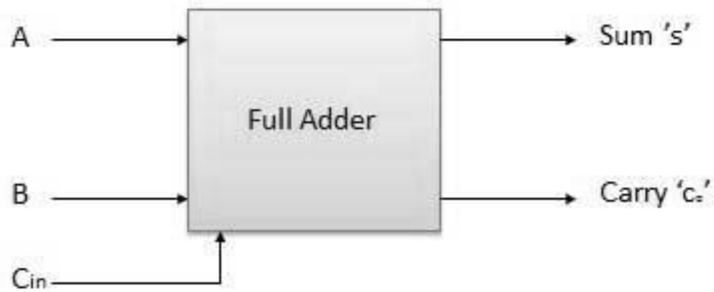


Fig: 1.20 Full Adder

Inputs			Output	
A	B	C_{in}	S	C_o
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

Table: 1.13 Truth table for Full adder

Circuit Diagram

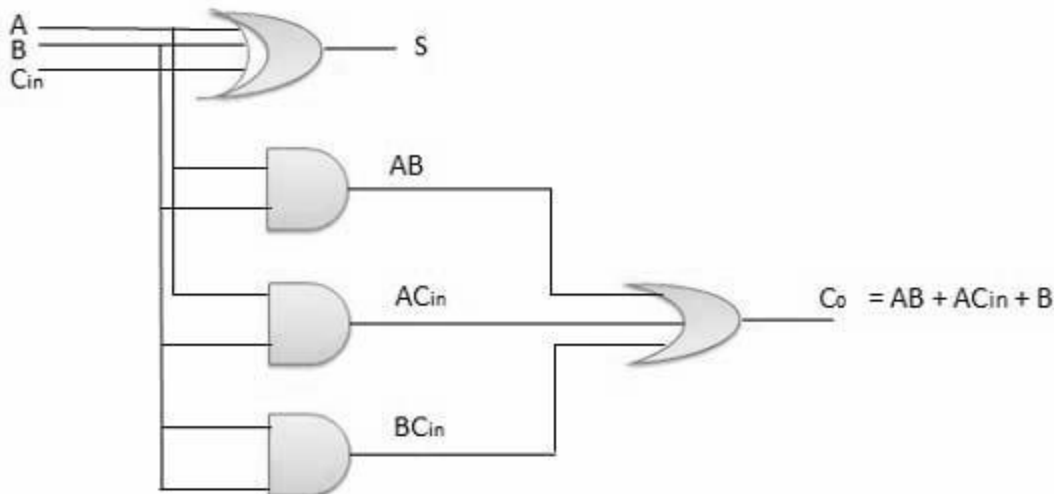


Fig: 1.21 Circuit diagram for Full adder

1.4.3 N-Bit Parallel Adder

The Full Adder is capable of adding only two single digit binary number along with a carry input. But in practical we need to add binary numbers which are much longer than just one bit. To add two n-bit binary numbers we need to use the n-bit parallel adder. It uses a number of full adders in cascade. The carry output of the previous full adder is connected to carry input of the next full adder.

1.4.4 A 4 Bit Parallel Adder

In the block diagram, A_0 and B_0 represent the LSB of the four bit words A and B. Hence Full Adder-0 is the lowest stage. Hence its C_{in} has been permanently made 0. The rest of the connections are exactly same as those of n-bit parallel adder is shown in fig. The four bit parallel adder is a very common logic circuit.

Block diagram:

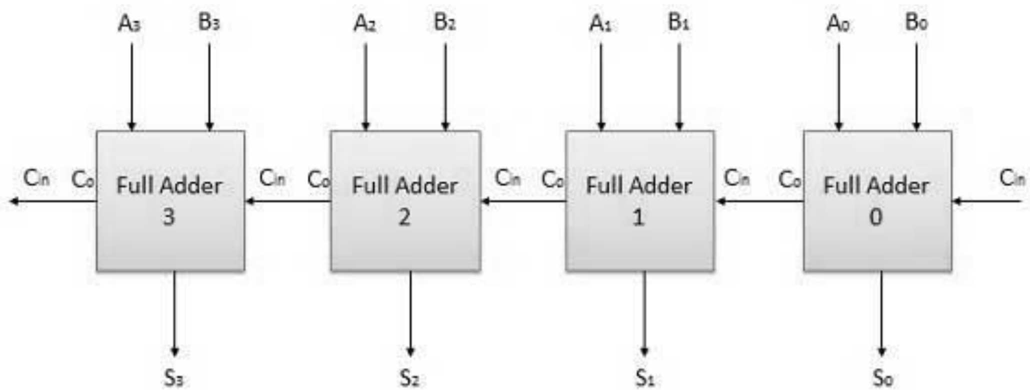


Fig: 1.22 A 4bit parallel adder

1.4.5 Multiplexers

Multiplexer is a special type of combinational circuit. There are n-data inputs, one output and m select inputs with 2m = n. It is a digital circuit which selects one of the n data inputs and routes it to the output. The selection of one of the n inputs is done by the selected inputs. Depending on the digital code applied at the selected inputs, one out of n data sources is selected and transmitted to the single output Y. E is called the strobe or enable input which is useful for the cascading. It is generally an active low terminal that means it will perform the required operation when it is low.

Block diagram:

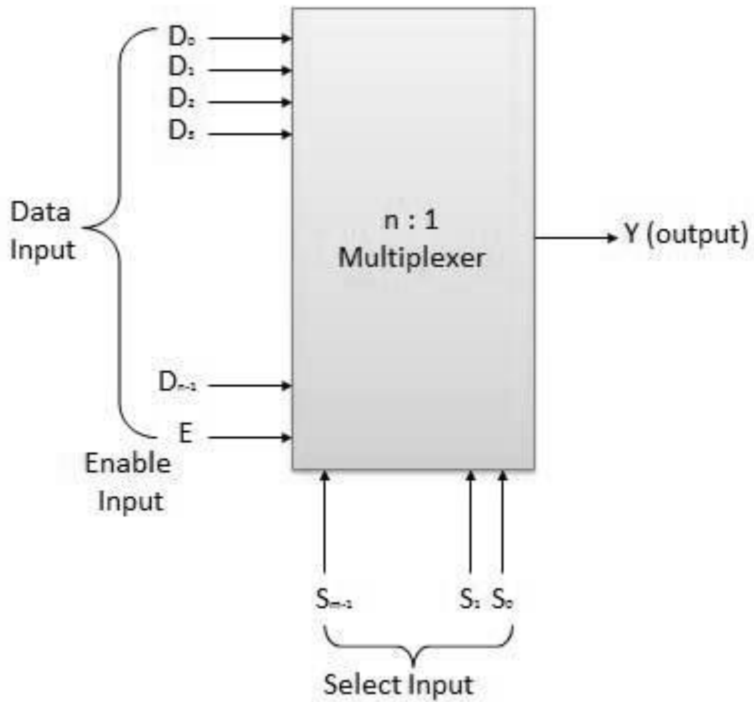


Fig: 1.23 An n:1 multiplexer

Block Diagram:

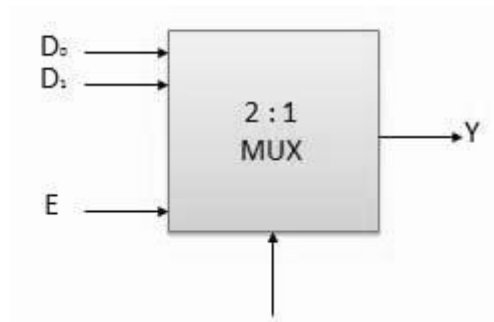


Fig:1.24 A 2:1 multiplexer

Enable	Select	Output
E	S	Y
0	x	0
1	0	D_0
1	1	D_1

x = Don't care

Table: 1.14

1.4.6 Demultiplexers:

A demultiplexer performs the reverse operation of a multiplexer i.e. it receives one input and distributes it over several outputs. It has only one input, n outputs, m select input. At a time only one output line is selected by the select lines and the input is transmitted to the selected output line. A de-multiplexer is equivalent to a single pole multiple way switch as shown in fig.

Block diagram:

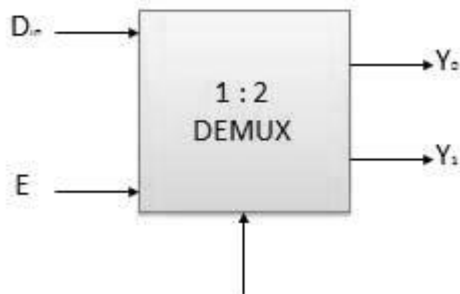


Fig:1.25 A 1:2 demultiplexer

Truth Table

Enable	Select	Output	
E	S	Y0	Y1
0	x	0	0
1	0	0	D_{in}
1	1	D_{in}	0

x = Don't care

Table: 1.15

1.5 Flip flops

In electronics, a **flip-flop** or **latch** is a circuit that has two stable states and can be used to store state information. A flip-flop is a bistable multivibrator. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in sequential logic. Flip-flops and latches are a fundamental building block of digital electronics systems used in computers, communications, and many other types of systems.

Flip-flops and latches are used as data storage elements. A flip-flop stores a single *bit* (binary digit) of data; one of its two states represents a "one" and the other represents a "zero". Such data storage can be used for storage of *state*, and such a circuit is described as sequential logic. When used in a finite-state machine, the output and next state depend not only on its current input, but also on its current state (and hence, previous inputs). It can also be used for counting of pulses, and for synchronizing variably-timed input signals to some reference timing signal.

Flip-flops can be either simple (transparent or opaque) or clocked (synchronous or edge-triggered). Although the term flip-flop has historically referred generically to both simple and clocked circuits, in modern usage it is common to reserve the term *flip-flop* exclusively for discussing clocked circuits; the simple ones are commonly called *latches*.

1.5.1 SR Flip-Flop

The **SR flip-flop**, also known as a *SR Latch*, can be considered as one of the most basic sequential logic circuit possible. This simple flip-flop is basically a one-bit memory bistable device that has two inputs, one which will "SET" the device (meaning the output = "1"), and is labelled S and another which will "RESET" the device (meaning the output = "0"), labelled R.

Then the SR description stands for "Set-Reset". The reset input resets the flip-flop back to its original state with an output Q that will be either at a logic level "1" or logic "0" depending upon this set/reset condition.

A basic NAND gate SR flip-flop circuit provides feedback from both of its outputs back to its opposing inputs and is commonly used in memory circuits to store a single data bit. Then the SR flip-flop actually has three inputs, Set, Reset and its current output Q relating to it's current state or history. The term "Flip-flop" relates to the actual operation of the device, as it can be "flipped" into one logic Set state or "flopped" back into the opposing logic Reset state.

1.5.2 The NAND Gate SR Flip-Flop

The simplest way to make any basic single bit set-reset SR flip-flop is to connect together a pair of cross-coupled 2-input NAND gates as shown, to form a Set-Reset Bistable also known as an active LOW SR NAND Gate Latch, so that there is feedback from each output to one of the other NANDgate inputs. This device consists of two inputs, one called the *Set*, S and the

other called the *Reset*,R with two corresponding outputs Q and its inverse or complement Q (not-Q) as shown below.

The Basic SR Flip-flop

Fig:1.27 SR Flip-flop

The Set State

Consider the circuit shown above. If the input R is at logic level "0" (R = 0) and input S is at logic level "1" (S = 1), the NAND gate *Y* has at least one of its inputs at logic "0" therefore, its output Q must be at a logic level "1" (NAND Gate principles). Output Q is also fed back to input "A" and so both inputs to NAND gate *X* are at logic level "1", and therefore its output Q must be at logic level "0".

Again NAND gate principals. If the reset input R changes state, and goes HIGH to logic "1" with Sremaining HIGH also at logic level "1", NAND gate *Y* inputs are now R = "1" and B = "0". Since one of its inputs is still at logic level "0" the output at Q still remains HIGH at logic level "1" and there is no change of state. Therefore, the flip-flop circuit is said to be "Latched" or "Set" with Q = "1" and Q = "0".

Reset State

In this second stable state, Q is at logic level “0”, (not Q = “0”) its inverse output at Q is at logic level “1”, (Q = “1”), and is given by R = “1” and S = “0”. As gate *X* has one of its inputs at logic “0” its outputQ must equal logic level “1” (again NAND gate principles). Output Q is fed back to input “B”, so both inputs to NAND gate *Y* are at logic “1”, therefore, Q = “0”.

If the set input, S now changes state to logic “1” with input R remaining at logic “1”, output Q still remains LOW at logic level “0” and there is no change of state. Therefore, the flip-flop circuits “Reset” state has also been latched and we can define this “set/reset” action in the following truth table.

It can be seen that when both inputs S = “1” and R = “1” the outputs Q and Q can be at either logic level “1” or “0”, depending upon the state of the inputs S or R BEFORE this input condition existed. Therefore the condition of S = R = “1” does not change the state of the outputs Q and Q.

Truth Table for this Set-Reset Function

State	S	R	Q	Q	Description
Set	1	0	0	1	Set Q » 1
	1	1	0	1	no change
Reset	0	1	1	0	Reset Q » 0
	1	1	1	0	no change
Invalid	0	0	1	1	Invalid Condition

Table: 1.16

However, the input state of S = "0" and R = "0" is an undesirable or invalid condition and must be avoided. The condition of S = R = "0" causes both outputs Q and Q to be HIGH together at logic level "1" when we would normally want Q to be the inverse of Q. The result is that the flip-flop looses control of Q and Q, and if the two inputs are now switched "HIGH" again after this condition to logic "1", the flip-flop becomes unstable and switches to an unknown data state based upon the unbalance as shown in the following switching diagram.

1.5.2 The D-type Flip Flop

One of the main disadvantages of the basic SR NAND Gate bistable circuit is that the indeterminate input condition of "SET" = logic "0" and "RESET" = logic "0" is forbidden. This state will force both outputs to be at logic "1", over-riding the feedback latching action and whichever input goes to logic level "1" first will lose control, while the other input still at logic "0" controls the resulting state of the latch.

But in order to prevent this from happening an inverter can be connected between the "SET" and the "RESET" inputs to produce another type of flip flop circuit known as a Data Latch, Delay flip flop, D-type Bistable, D-type Flip Flop or just simply a **D Flip Flop** as it is more generally called.

The D Flip Flop is by far the most important of the Clocked Flip-flops as it ensures that ensures that inputs S and R are never equal to one at the same time. The D-type flip flop are constructed from a gated SR flip-flop with an inverter added between the S and the R inputs to allow for a single D(data) input.

Then this single data input, labelled D, is used in place of the "set" signal, and the inverter is used to generate the complementary "reset" input thereby making a level-sensitive D-type flip-flop from a level-sensitive RS-latch as now S = D and R = not D as shown.

D-type Flip-Flop Circuit

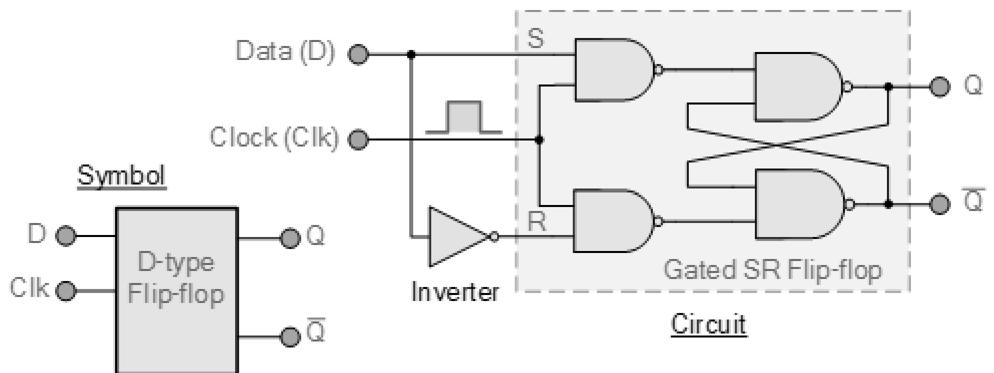


Fig: 1.28 D-type Flip-flop

We remember that a simple SR flip-flop requires two inputs, one to “SET” the output and one to “RESET” the output. By connecting an inverter (NOT gate) to the SR flip-flop we can “SET” and “RESET” the flip-flop using just one input as now the two input signals are complements of each other. This complement avoids the ambiguity inherent in the SR latch when both inputs are LOW, since that state is no longer possible.

Thus this single input is called the “DATA” input. If this data input is held HIGH the flip flop would be “SET” and when it is LOW the flip flop would change and become “RESET”. However, this would be rather pointless since the output of the flip flop would always change on every pulse applied to this data input.

To avoid this an additional input called the “CLOCK” or “ENABLE” input is used to isolate the data input from the flip flop’s latching circuitry after the desired data has been stored. The effect is that D input condition is only copied to the output Q when the clock input is active. This then forms the basis of another sequential device called a **D Flip Flop**.

The "D flip flop" will store and output whatever logic level is applied to its data terminal so long as the clock input is HIGH. Once the clock input goes LOW the "set" and "reset" inputs of the flip-flop are both held at logic level "1" so it will not change state and store whatever data was present on its output before the clock transition occurred. In other words the output is "latched" at either logic "0" or logic "1".

Truth Table for the D-type Flip Flop

Clk	D	Q	Q	Description
↓ » 0	X	Q	Q	Memory no change
↑ » 1	0	0	1	Reset Q » 0
↑ » 1	1	1	0	Set Q » 1

Table: 1.17

Note that: ↓ and ↑ indicates direction of clock pulse as it is assumed D-type flip flops are edge triggered

1.5.3 The JK Flip Flop

This simple JK flip Flop is the most widely used of all the flip-flop designs and is considered to be a universal flip-flop circuit. The sequential operation of the JK flip flop is exactly the same as for the previous SR flip-flop with the same "Set" and "Reset" inputs. The difference this time is that the "JK flip flop" has no invalid or forbidden input states of the SR Latch even when S and R are both at logic "1".

The JK flip flop is basically a gated SR Flip-flop with the addition of a clock input circuitry that prevents the illegal or invalid output condition that can occur when both inputs S and R are equal to logic level “1”. Due to this additional clocked input, a JK flip-flop has four possible input combinations, “logic 1”, “logic 0”, “no change” and “toggle”. The symbol for a JK flip flop is similar to that of an SR Bistable Latch as seen in the previous tutorial except for the addition of a clock input.

The Basic JK Flip-flop

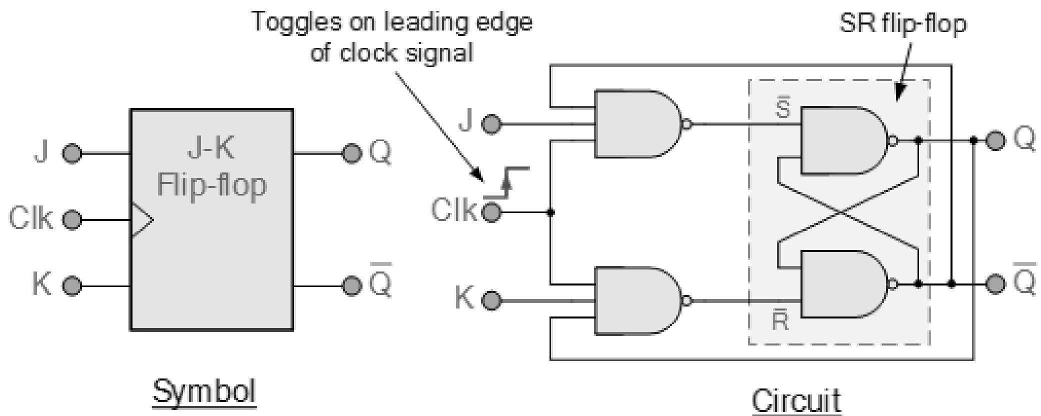


Fig: 1.28 J-K Flip-flop

Both the S and the R inputs of the previous SR bistable have now been replaced by two inputs called the J and K inputs, respectively after its inventor Jack Kilby. Then this equates to: J = S and K = R.

The two 2-input AND gates of the gated SR bistable have now been replaced by two 3-input NAND gates with the third input of each gate connected to the outputs at Q and Q. This cross coupling of the SR flip-flop allows the previously invalid condition of S = “1” and R = “1” state to be used to produce a “toggle action” as the two inputs are now interlocked.

If the circuit is now “SET” the J input is inhibited by the “0” status of Q through the lower NAND gate. If the circuit is “RESET” the K input is inhibited by the “0” status of Q through the upper NAND gate. As Q and Q are always different we can use them to control the input. When both inputs J and K are equal to logic “1”, the JK flip flop toggles as shown in the following truth table.

Table: 1.18 <u>The Truth Table for the JK Function</u>

	Input		Output		Description
	J	K	Q	Q	
same as for the SR Latch	0	0	0	0	Memory no change
	0	0	0	1	
	0	1	1	0	Reset Q » 0
	0	1	0	1	
	1	0	0	1	Set Q » 1
	1	0	1	0	
toggle action	1	1	0	1	Toggle
	1	1	1	0	

Then the JK flip-flop is basically an SR flip flop with feedback which enables only one of its two input terminals, either SET or RESET to be active at any one time thereby eliminating the invalid condition seen previously in the SR flip flop circuit. Also when both the J and the K inputs are at logic level “1” at the same time, and the clock input is pulsed either “HIGH”, the circuit will “toggle” from its SET state to a RESET state, or visa-versa. This results in the JK flip flop acting more like a T-type toggle flip-flop when both terminals are “HIGH”.

1.5.4 T Flip-flop

This Flip-flop is obtained from JK type when J and K are connected to provide a single input designated by T, Hence the T Flip-flop has only two conditions

When T = 0 (J=K=0), the clock transition does not change the state of the Flip-flop.

When T = 1 (J=K=1), the clock transition complements the state of the Flip-flop. And the conditiond can be expressed as:

$$Q(t+1)= Q(t) +o T$$

Edge Triggered Flip-flops

Edge triggered flip flops are most commonly used to synchronizethe state change duringa clock pulse transition. In this type of flip flopsoutput transitions occur at a specific level of the clock pulse. Whenever the pulse input levels exceeds the threshold level the inputs are locked out so that the flip flop is unresponsive to further changes in inputs until clock pulse returns to 0 and another pulse occurs

Master – slave Flip-flop

Another type of flip- flops known as master- slave flip flops are used in some systems. Here the circuits consist of two flip- flops,

the first is the master and the second is the slave. The master responds to positive level of the clock and the slave responds to negative level of the clock. Here the output changes during the 1-to-0 transition of clock signal.

1.6 Sequential Logic Circuits

A circuit with interconnection of flip-flops and gates is called a sequential circuit. The combinational circuit consists of gates but when included with flip-flops the circuit is termed as sequential circuit.

sequential logic is a type of logic circuit whose output depends not only on the present value of its input signals but on the sequence of past inputs, the input history. This is in contrast to combinational logic, whose output is a function of only the present input. That is, sequential logic has *state* (*memory*) while combinational logic does not. Or, in other words, sequential logic is combinational logic with memory.

Sequential logic is used to construct finite state machines, a basic building block in all digital circuitry, as well as memory circuits and other devices. Virtually all circuits in practical digital devices are a mixture of combinational and sequential logic.

In other words, the output state of a “sequential logic circuit” is a function of the following three states, the “present input”, the “past input” and/or the “past output”. *Sequential Logic circuits*remember these conditions and stay fixed in their current state until the next clock signal changes one of the states, giving sequential logic circuits “Memory”.

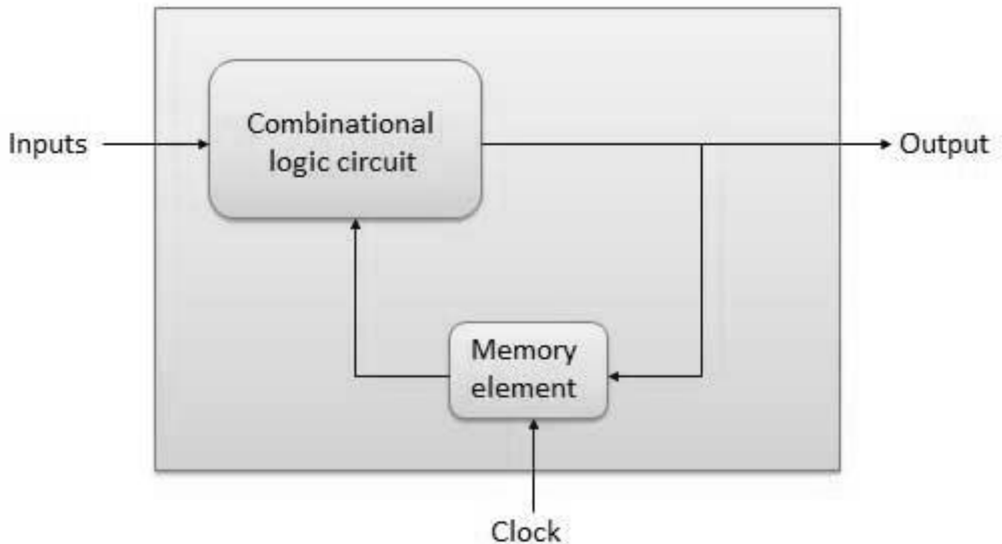


Fig: 1.29 Example sequential circuit

1.6.1 State Tables and State Diagrams

We have examined a general model for sequential circuits. In this model the effect of all previous inputs on the outputs is represented by a state of the circuit. Thus, the output of the circuit at any time depends upon its current state and the input. These also determine the next state of the circuit. The relationship that exists among the inputs, outputs, present states and next states can be specified by either the **state table** or the **state diagram**.

State Table

The state table representation of a sequential circuit consists of three sections labelled *present state*, *next state* and *output*. The present state designates the state of flip-flops before the occurrence of a clock pulse. The next state shows the states of flip-flops after the clock pulse, and the output section lists the value of the output variables during the present state.

State Diagram

In addition to graphical symbols, tables or equations, flip-flops can also be represented graphically by a state diagram. In this diagram, a state is represented by a circle, and the transition between states is indicated by directed lines (or arcs) connecting the circles. An example of a state diagram is shown in Figure 3 below.

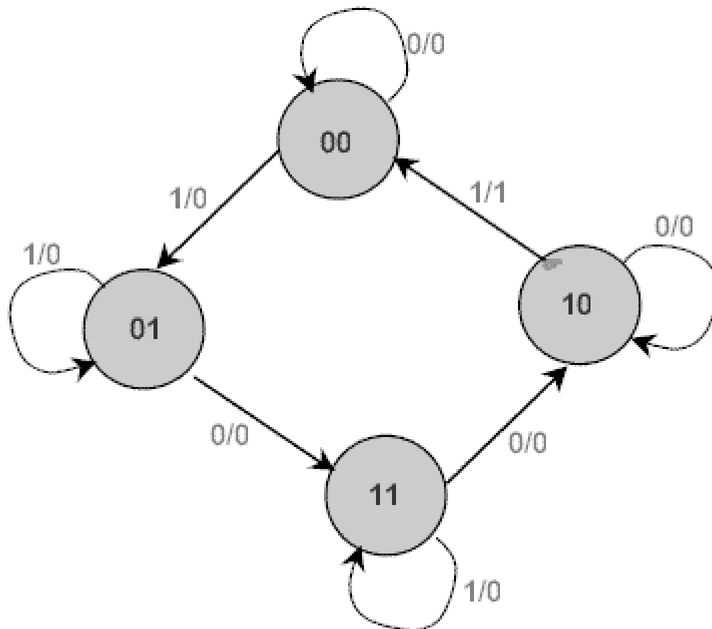


Fig: 1.30 State Diagram

The binary number inside each circle identifies the state the circle represents. The directed lines are labelled with two binary numbers separated by a slash (/). The input value that causes the state transition is labelled first. The number after the slash symbol / gives the value of the output. For example, the directed line from state 00 to 01 is labelled 1/0, meaning that, if the sequential circuit is in a present state and the input is 1, then the next state is 01 and the output is 0. If it is in a present state 00 and the input is 0, it

will remain in that state. A directed line connecting a circle with itself indicates that no change of state occurs. The state diagram provides exactly the same information as the state table and is obtained directly from the state table.

Consider a sequential circuit shown in Figure 4. It has one input x, one output Z and two state variables Q1Q2 (thus having four possible present states 00, 01, 10, 11).

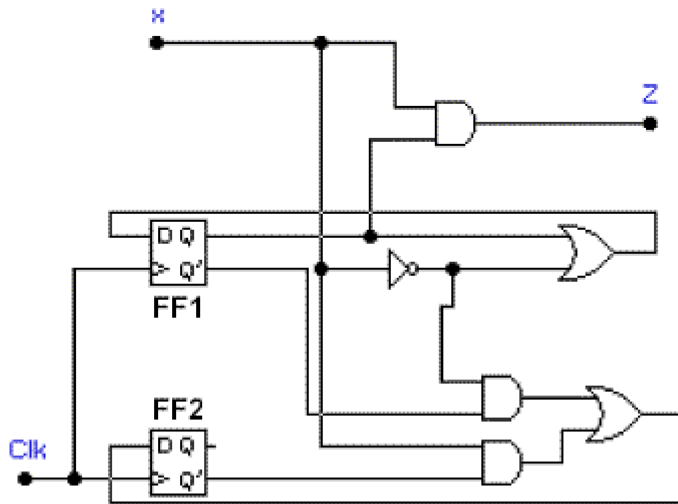


Fig:1.31 A Sequential Circuit

The behaviour of the circuit is determined by the following Boolean expressions:

Z = x*Q1

D1 = x' + Q1

D2 = x*Q2' + x'*Q1'

These equations can be used to form the state table. Suppose the present state (i.e. Q1Q2) = 00 and input x = 0. Under these

conditions, we get Z = 0, D1 = 1, and D2 = 1. Thus the next state of the circuit D1D2 = 11, and this will be the present state after the clock pulse has been applied. The output of the circuit corresponding to the present state Q1Q2 = 00 and x = 1 is Z = 0. This data is entered into the state table as shown in Table

Present State Q1Q2	Next State x = 0	Next State x = 1	Output x = 0	Output x = 1
0 0	1 1	0 1	0	0
0 1	1 1	0 0	0	0
1 0	1 0	1 1	0	1
1 1	1 0	1 0	0	1

Table: 1.19 State table for the sequential circuit in Figure 1.31s

The state diagram for the sequential circuit in Figure 1.31 is

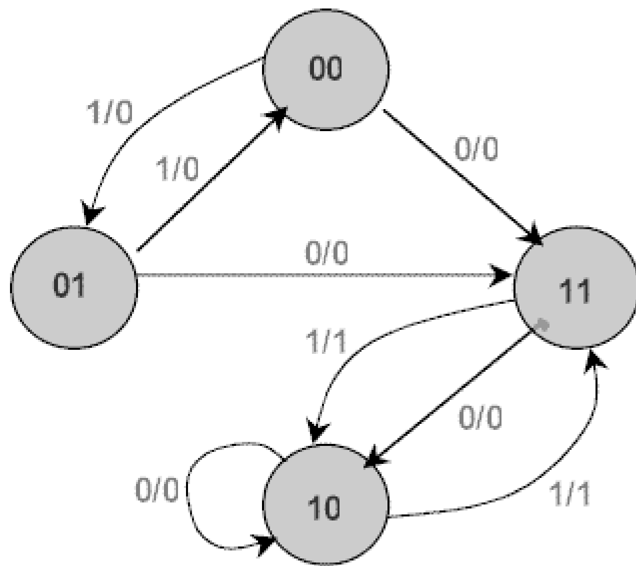


Fig: 1.32. State Diagram of circuit in Fig: 1.31

State Diagrams of Various Flip-flops:

NAME	STATE DIAGRAM
SR	S,R=0,0 · S,R=1,0 · S,R=0,0 · Q = 0 · Q = 1 · S,R=0,1
JK	J,K=1,0 or 1,1 · J,K=0,0 · J,K=0,0 · Q = 0 · Q = 1 · J,K=0,1 or 1,1
D	D = 1 · D = 1 · D = 1 · Q = 0 · Q = 1 · D = 0
T	T = 0 · T = 1 · T = 0 · Q = 0 · Q = 1 · T = 1

Table: 1.20. State diagrams of the four types of flip-flops.

You can see from the table that all four flip-flops have the same number of states and transitions. Each flip-flop is in the set state when Q=1 and in the reset state when Q=0. Also, each flip-flop can move from one state to another, or it can re-enter the same state. The only difference between the four types lies in the values of input signals that cause these transitions.

A state diagram is a very convenient way to visualise the operation of a flip-flop or even of large sequential components.

Chapter 2
Digital Components

2.1 Integrated Circuits

An integrated circuit (IC), sometimes called a *chip* or microchip, is a semiconductor wafer on which thousands or millions of tiny resistors, capacitors, and transistors are fabricated. An IC can function as an amplifier, oscillator, timer, counter, computer memory, or microprocessor. A particular IC is categorized as either linear (analog) or digital, depending on its intended application.

Linear ICs have continuously variable output (theoretically capable of attaining an infinite number of states) that depends on the input signal level. As the term implies, the output signal level is a linear function of the input signal level. Ideally, when the instantaneous output is graphed against the instantaneous input, the plot appears as a straight line. Linear ICs are used as audio-frequency (AF) and radio-frequency (RF) amplifiers. The *operational amplifier*(op amp) is a common device in these applications.

Digital ICs operate at only a few defined levels or states, rather than over a continuous range of signal amplitudes. These devices are used in computers, computer networks, modems, and frequency counters. The fundamental building blocks of digital ICs are logic gates, which work with binary data, that is, signals that have only two different states, called low (logic 0) and high (logic 1).

Integrated circuits are used in virtually all electronic equipment today and have revolutionized the world of electronics. Computers, mobile phones, and other digital home appliances are now inextricable parts of the structure of modern societies, made possible by the low cost of integrated circuits.

In the early days of simple integrated circuits, the technology's large scale limited each chip to only a few transistors, and the low degree of integration meant the design process was relatively simple. Manufacturing yields were also quite low by today's standards. As the technology progressed, millions, then billions[17] of transistors could be placed on one chip, and good designs required thorough planning, giving rise to new design methods.

The first integrated circuits contained only a few transistors. Early digital circuits containing tens of transistors provided a few logic gates, and early linear ICs had as few as two transistors. The number of transistors in an integrated circuit has increased dramatically since then. The term "large scale integration" (LSI) was first used by IBM scientist Rolf Landauer when describing the theoretical concept that term gave rise to the terms "small-scale integration" (SSI), "medium-scale integration" (MSI), "very-large-scale integration" (VLSI), and "ultra-large-scale integration" (ULSI). The early integrated circuits were SSI.

SSI circuits were crucial to early aerospace projects, and aerospace projects helped inspire development of the technology. Integrated circuits began to appear in consumer products by the turn of the decade, a typical application being FM inter-carrier sound processing in television receivers.

The first MOS chips were small-scale integrated chips for NASA satellites.

The next step in the development of integrated circuits, taken in the late 1960s, introduced devices which contained hundreds of transistors on each chip, called "medium-scale integration" (MSI).

In 1964, Frank Wanlass demonstrated a single-chip 16-bit shift register he designed, with an incredible (for the time) 120 transistors on a single chip.

Name	Signification	Year	Number of transistors	Number of logic gates
SSI	*small-scale integration*	1964	1 to 10	1 to 12
MSI	*medium-scale integration*	1968	10 to 500	13 to 99
LSI	*large-scale integration*	1971	500 to 20,000	100 to 9,999
VLSI	*very large-scale integration*	1980	20,000 to 1,000,000	10,000 to 99,999
ULSI	*ultra-large-scale integration*	1984	1,000,000 and more	100,000 and more

Table: 2.1

MSI devices were attractive economically because while they cost little more to produce than SSI devices, they allowed more complex systems to be produced using smaller circuit boards, less assembly work (because of fewer separate components), and a number of other advantages.

Further development, driven by the same economic factors, led to "large-scale integration" (LSI) in the mid-1970s, with tens of thousands of transistors per chip.

SSI and MSI devices often were manufactured by masks created by hand-cutting Rubylith; an engineer would inspect and verify the completeness of each mask. LSI devices contain so many transistors, interconnecting wires, and other features that it is considered impossible for a human to check the masks or even do the original design entirely by hand; the engineer depends on computer programs and other hardware aids to do most of this work.[24]

Integrated circuits such as 1K-bit RAMs, calculator chips, and the first microprocessors, that began to be manufactured in moderate quantities in the early 1970s, had under 4000 transistors. True LSI circuits, approaching 10,000 transistors, began to be produced around 1974, for computer main memories and second-generation microprocessors.

2.1.1 VLSI

The final step in the development process, starting in the 1980s and continuing through the present, was "very-large-scale integration" (VLSI). The development started with hundreds of thousands of transistors in the early 1980s, and continues beyond several billion transistors as of 2009.

Multiple developments were required to achieve this increased density. Manufacturers moved to smaller design rules and cleaner fabrication facilities, so that they could make chips with more transistors and maintain adequate yield. The path of process improvements was summarized by the International Technology Roadmap for Semiconductors (ITRS). Design tools improved enough to make it practical to finish these designs in a reasonable time. The more energy-efficient CMOS replaced NMOS and PMOS, avoiding a prohibitive increase in power consumption.

In 1986 the first one-megabit RAM chips were introduced, containing more than one million transistors. Microprocessor chips passed the million-transistor mark in 1989 and the billion-transistor mark in 2005. The trend continues largely unabated, with chips introduced in 2007 containing tens of billions of memory transistors.

2.1.2 ULSI, WSI, SOC and 3D-IC

To reflect further growth of the complexity, the term *ULSI* that stands for "ultra-large-scale integration" was proposed for chips of more than 1 million transistors.[27]

Wafer-scale integration (WSI) is a means of building very large integrated circuits that uses an entire silicon wafer to produce a single "super-chip". Through a combination of large size and reduced packaging, WSI could lead to dramatically reduced costs for some systems, notably massively parallel supercomputers. The name is taken from the term Very-Large-Scale Integration, the current state of the art when WSI was being developed.[28]

A system-on-a-chip (SoC or SOC) is an integrated circuit in which all the components needed for a computer or other system are included on a single chip. The design of such a device can be complex and costly, and building disparate components on a single piece of silicon may compromise the efficiency of some elements. However, these drawbacks are offset by lower manufacturing and assembly costs and by a greatly reduced power budget: because signals among the components are kept on-die, much less power is required (see Packaging).[29]

A three-dimensional integrated circuit (3D-IC) has two or more layers of active electronic components that are integrated both vertically and horizontally into a single circuit. Communication between layers uses on-die signaling, so power consumption is much lower than in equivalent separate circuits. Judicious use of short vertical wires can substantially reduce overall wire length for faster operation.

2.2 Decoders

The Binary Decoder is another combinational logic circuit constructed from individual logic gates and is the exact opposite to that of an "Encoder" we looked at in the last tutorial. The name "Decoder" means to translate or decode coded information from one format into another, so a digital decoder transforms a set of digital input signals into an equivalent decimal code at its output.

Binary Decoders are another type of Digital Logic device that has inputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines, so a decoder that has a set of two or more bits will be defined as having an *n*-bit code, and therefore it will be possible to represent 2^n possible values. Thus, a decoder generally decodes a binary value into a non-binary one by setting exactly one of its *n* outputs to logic "1".

If a binary decoder receives n inputs (usually grouped as a single Binary or Boolean number) it activates one and only one of its 2^n outputs based on that input with all other outputs deactivated.

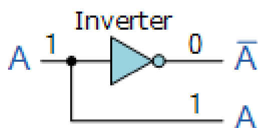


Fig: 2.1

So for example, an inverter (NOT-gate) can be classed as a 1-to-2 binary decoder as 1-input and 2-outputs (2^1) is possible because with an input Ait can produce two outputs A and A (not-A) as shown.

Then we can say that a standard combinational logic decoder is an n-to-mdecoder, where $m \leq 2^n$, and whose

output, Q is dependent only on its present input states. In other words, a binary decoder looks at its current inputs, determines which binary code or binary number is present at its inputs and selects the appropriate output that corresponds to that binary input.

A *Binary Decoder* converts coded inputs into coded outputs, where the input and output codes are different and decoders are available to "decode" either a Binary or BCD (8421 code) input pattern to typically a Decimal output code. Commonly available BCD-to-Decimal decoders include the TTL 7442 or the CMOS 4028. Generally a decoders output code normally has more bits than its input code and practical "binary decoder" circuits include, 2-to-4, 3-to-8 and 4-to-16 line configurations.

An example of a 2-to-4 line decoder:

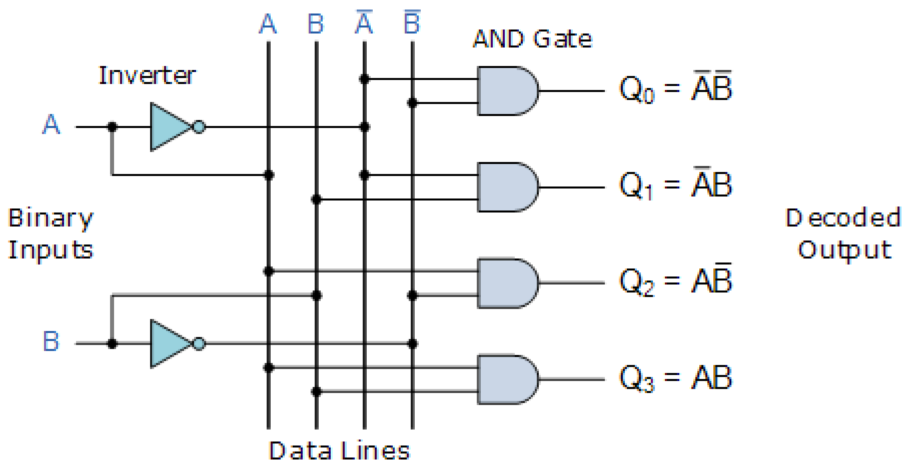
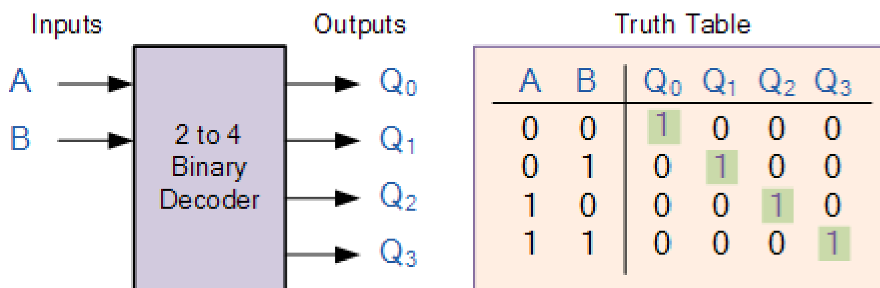


Fig: 2.2



A	B	Q_0	Q_1	Q_2	Q_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Fig: 2.3 **Table: 2.2**

This simple example above of a 2-to-4 line binary decoder consists of an array of four AND gates. The 2 binary inputs labelled A and B are decoded into one of 4 outputs, hence the description of 2-to-4 binary decoder. Each output represents one of the miniterms of the 2 input variables, (each output = a miniterm).

The binary inputs A and B determine which output line from Q0 to Q3 is "HIGH" at logic level "1" while the remaining outputs are held "LOW" at logic "0" so only one output can be active (HIGH) at any one time. Therefore, whichever output line is "HIGH" identifies the binary code present at the input, in other words it "de-codes" the binary input.

Some binary decoders have an additional input pin labelled "Enable" that controls the outputs from the device. This extra input allows the decoders outputs to be turned "ON" or "OFF" as required. These types of binary decoders are commonly used as "memory address decoders" in microprocessor memory applications.

We can say that a binary decoder is a demultiplexer with an additional data line that is used to enable the decoder. An alternative way of looking at the decoder circuit is to regard inputs A, B and C as address signals. Each combination of A, B or C defines a unique memory address.

We have seen that a 2-to-4 line binary decoder (TTL 74155) can be used for decoding any 2-bit binary code to provide four outputs, one for each possible input combination. However, sometimes it is required to have a Binary Decoder with a number of outputs greater than is available, so by adding more inputs, the decoder can potentially provide 2^n more outputs.

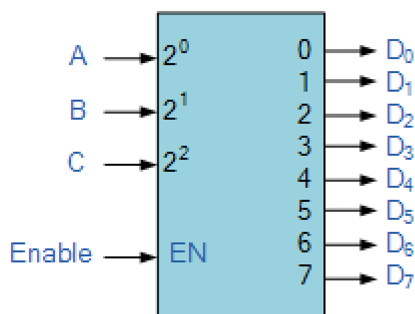
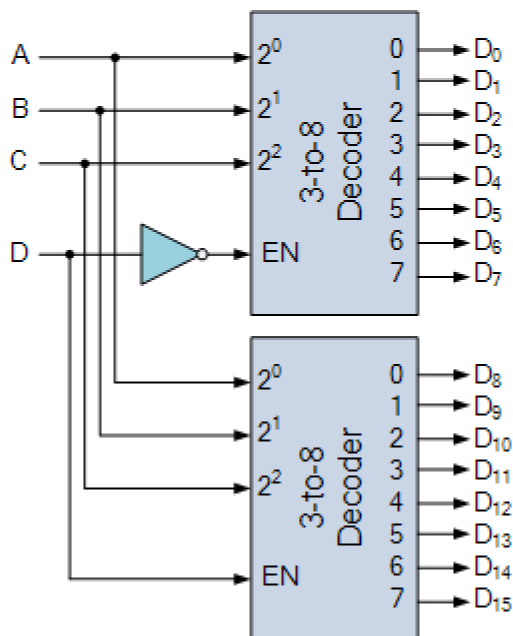


Fig: 2.4

So for example, a decoder with 3 binary inputs (n = 3), would produce a 3-to-8 line decoder (TTL 74138) and 4 inputs (n = 4) would produce a 4-to-16 line decoder (TTL 74154) and so on. But a decoder can also have less than 2^n outputs such as the BCD to seven-segment decoder (TTL 7447) which has 4 inputs and only 7 active outputs to drive a display rather than the full 16 (2^4) outputs as you would expect.

Here a much larger 4 (3 data plus 1 enable) to 16 line binary decoder has been implemented using two smaller 3-to-8 decoders.

A 4-to-16 Binary Decoder Configuration.



4-to-16 Line Decoder Implemented
with two 3-to-8 Decoders

Fig: 2.5

Inputs A, B, C are used to select which output on either decoder will be at logic "1" (HIGH) and input D is used with the enable input to select which encoder either the first or second will output the "1".

However, there is a limit to the number of inputs that can be used for one particular decoder, because as *n* increases, the number of AND gates required to produce an output also becomes larger resulting in the fan-out of the gates used to drive them becoming large.This type of active-"HIGH" decoder can be implemented using just Inverters, (NOT Gates) and AND gates. It is convenient to use an AND gate as the basic decoding element for the output because it produces a "HIGH" or logic "1" output

only when all of its inputs are logic “1”.But some binary decoders are constructed using NAND gates instead of AND gates for their decoded output, since NAND gates are cheaper to produce than AND’s as they require fewer transistors to implement within their design.The use of NAND gates as the decoding element, results in an active-“LOW” output while the rest will be “HIGH”. As a NAND gate produces the AND operation with an inverted output, the NAND decoder looks like this with its inverted truth table.

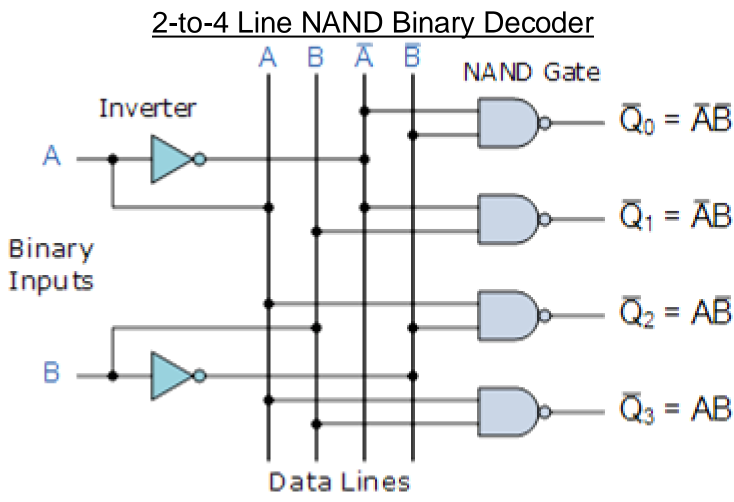


Fig: 2.6

Truth Table

A	B	Q_0	Q_1	Q_2	Q_3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

Table: 2.3

Then for the NAND decoder, only one output can be LOW and equal to logic "0" at any given time, with all the other outputs being HIGH at logic "1".

Decoders are also available with an additional "Enable" input pin which allows the decoded output to be turned "ON" or "OFF" by applying a logic "1" or logic "0" respectively to it. So for example, when the enable input is at logic level "0", (EN = 0) all outputs are "OFF" at logic "0" (for AND gates) regardless of the state of the inputs A and B.

Generally to implement this enabling function the 2-input AND or NAND gates are replaced with 3-input AND or NAND gates. The additional input pin represents the enable function.

2.3 Multiplexers

Multiplexing is the generic term used to describe the operation of sending one or more analogue or digital signals over a common transmission line at different times or speeds and as such, the device we use to do just that is called a **Multiplexer**.

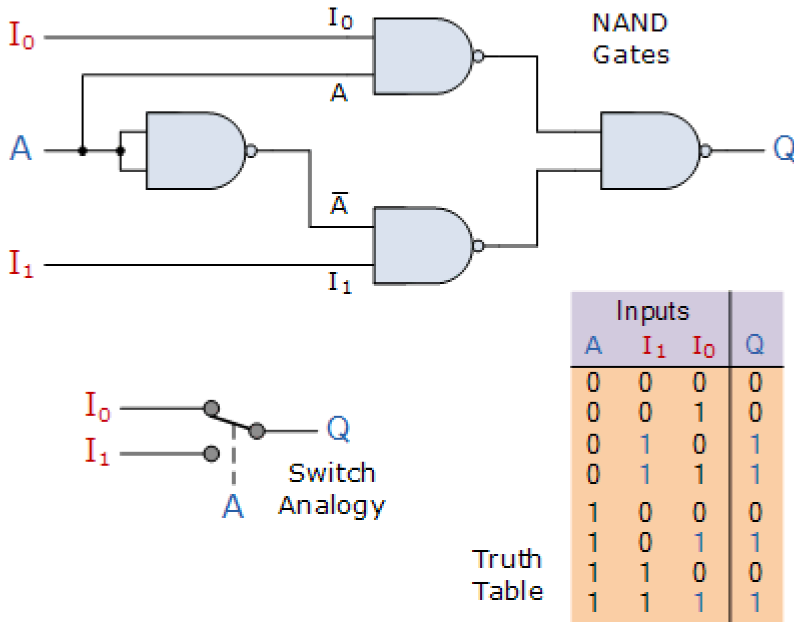
The *multiplexer*, shortened to "MUX" or "MPX", is a combinational logic circuit designed to switch one of several input lines through to a single common output line by the application of a control signal. Multiplexers operate like very fast acting multiple position rotary switches connecting or controlling multiple input lines called "channels" one at a time to the output.

Multiplexers, or MUX's, can be either digital circuits made from high speed logic gates used to switch digital or binary data or they can be analogue types using transistors, MOSFET's or relays to switch one of the voltage or current inputs through to a single output.

The most basic type of multiplexer device is that of a one-way rotary switch as shown.

Generally, the selection of each input line in a multiplexer is controlled by an additional set of inputs called *control lines* and according to the binary condition of these control inputs, either “HIGH” or “LOW” the appropriate data input is connected directly to the output. Normally, a multiplexer has an even number of 2^N data input lines and a number of “control” inputs that correspond with the number of data inputs.

Note that multiplexers are different in operation to *Encoders*. Encoders are able to switch an n-bit input pattern to multiple output lines that represent the binary coded (BCD) output equivalent of the active input. We can build a simple 2-line to 1-line (2-to-1) multiplexer from basic logic NAND gates as shown.



Inputs			
A	I1	I0	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Fig: 2.7 **Table: 2.4**

The input A of this simple 2-1 line multiplexer circuit constructed from standard NAND gates acts to control which input (I_0 or I_1) gets passed to the output at Q.

From the truth table above, we can see that when the data select input, A is LOW at logic 0, input I_1passes its data through the NAND gate multiplexer circuit to the output, while input I_0 is blocked. When the data select A is HIGH at logic 1, the reverse happens and now input I_0 passes data to the output Q while input I_1 is blocked.

So by the application of either a logic "0" or a logic "1" at A we can select the appropriate input, I_0 orI_1 with the circuit acting a bit like a single pole double throw (SPDT) switch. Then in this simple example, the 2-input multiplexer connects one of two 1-bit sources to a common output, producing a 2-to-1-line multiplexer and we can confirm this in the following Boolean expression.

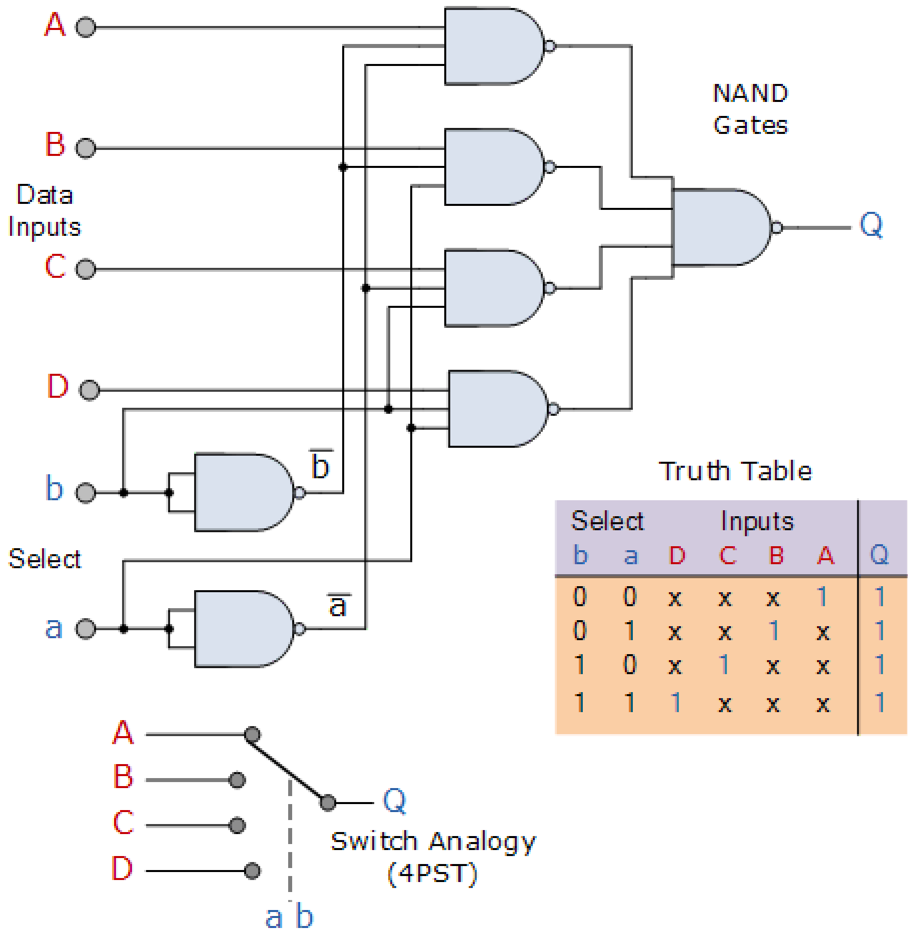
$$Q = A.I_0.I_1 + A.I_0.I_1 + A.I_0.I_1 + A.I_0.I_1$$

and for our 2-input multiplexer circuit above, this can be simplified too:

$$Q = A.I_1 + A.I_0$$

We can increase the number of data inputs to be selected further simply by following the same procedure and larger multiplexer circuits can be implemented using smaller 2-to-1 multiplexers as their basic building blocks. So for a 4-input multiplexer we would therefore require two data select lines as 4-inputs represents 2^2 data control lines give a circuit with four inputs, I_0, I_1, I_2, I_3 and two data select lines A and B as shown.

4-to-1 Channel Multiplexer



Select		Inputs				
b	a	D	C	B	A	Q
0	0	x	x	x	1	1
0	1	x	x	1	x	1
1	0	x	1	x	x	1
1	1	1	x	x	x	1

Fig: 2.8 **Table: 2.4**

The Boolean expression for this 4-to-1 Multiplexer above with inputs A to D and data select lines a, b is given as:

$$Q = abA + abB + abC + abD$$

In this example at any one instant in time only ONE of the four analogue switches is closed, connecting only one of the input lines A to D to the single output at Q. As to which switch is closed depends upon the addressing input code on lines “a” and “b“, so for this example to select input Bto the output at Q, the binary input address would need to be “a” = logic “1” and “b” = logic “0”.

Then we can show the selection of the data through the multiplexer as a function of the data select bits as shown.

<u>Multiplexer Input Line Selection</u>

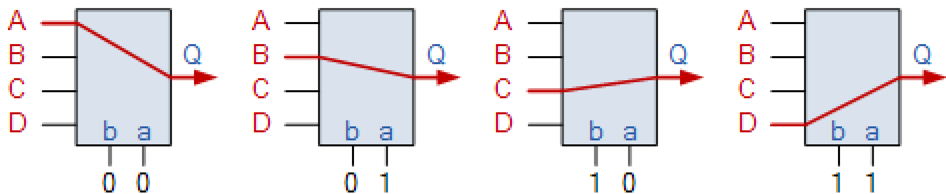


Fig: 2.9

Adding more control address lines will allow the multiplexer to control more inputs but each control line configuration will connect only ONE input to the output.

Then the implementation of the Boolean expression above using individual logic gates would require the use of seven individual gates consisting of AND, OR and NOT gates as shown.

<u>4 Channel Multiplexer using Logic Gates</u>

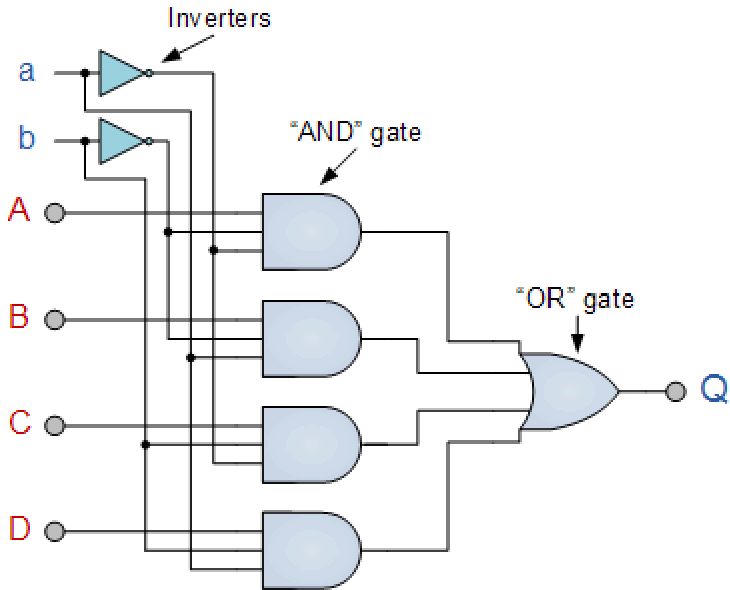


Fig: 2.10

The symbol used in logic diagrams to identify a multiplexer is as follows.

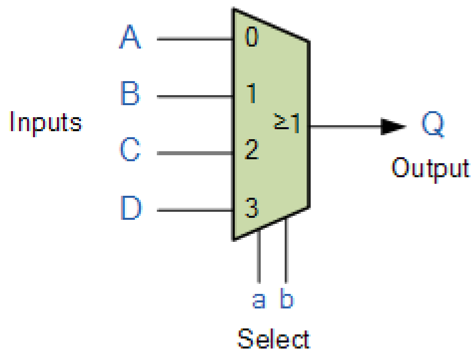


Fig: 2.11

2.4 Registers

Flip-flop is a 1 bit memory cell which can be used for storing the digital data. To increase the storage capacity in terms of number of bits, we have to use a group of flip-flop. Such a group of flip-flop is known as a Register. The n-bit register will consist of n number of flip-flop and it is capable of storing an n-bit word. Other than flip-flops registers can also have combinational gates which perform certain data processing tasks. The flip-flops hold the binary information and the combinational gates control when and how new information is transferred into the registers. To store N bits, a register must have N flip-flops, one for each bit to be stored.

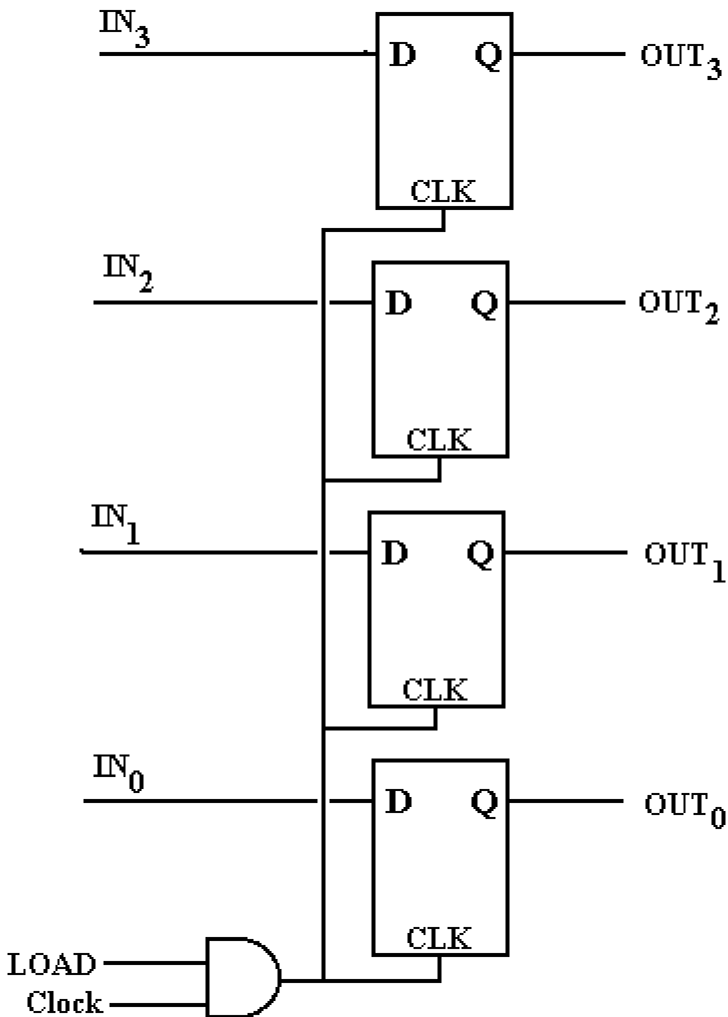


Fig: 2.13 a four bit register using D Flip-flops

State Diagram of circuit inIn this example, the 4-bit register is implemented by four D flip-flops. Note the input CLK comes from an AND gate that puts out the logical AND of the system clock (Clock) and the LOAD signal. When LOAD is 0, the flip-flops are cut off from the input and do not change state in response to the

input. The design calls for LOAD to be 1 for almost one clock pulse, so that the system clock and LOAD are both high for 1/2 clock cycle. At this time, the register is loaded.

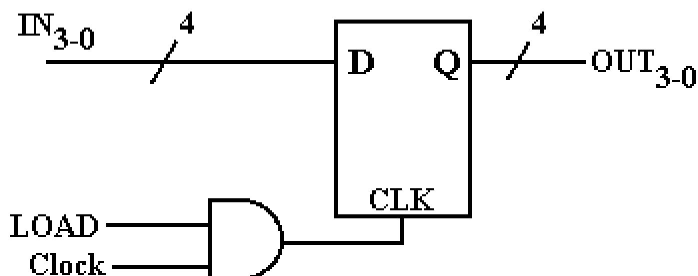


Fig: 2.14 a four bit register using

State Diagram of circuit in the figure at right shows a short-hand notation used when drawing registers that contain a number of flip-flops identically configured. It should be obvious that the figure represents a 4-bit register.

2.5 Shift Registers

The Shift Register is another type of sequential logic circuit that can be used for the storage or the transfer of data in the form of binary numbers. This sequential device loads the data present on its inputs and then moves or “shifts” it to its output once every clock cycle, hence the name “shift register”.

A shift register basically consists of several single bit “D-Type Data Latches”, one for each data bit, either a logic “0” or a “1”, connected together in a serial type daisy-chain arrangement so that the output from one data latch becomes the input of the next latch and so on.

Data bits may be fed in or out of a shift register serially, that is one after the other from either the left or the right direction, or all together at the same time in a parallel configuration.

The number of individual data latches required to make up a single Shift Register device is usually determined by the number of bits to be stored with the most common being 8-bits (one byte) wide constructed from eight individual data latches.

Shift Registers are used for data storage or for the movement of data and are therefore commonly used inside calculators or computers to store data such as two binary numbers before they are added together, or to convert the data from either a serial to parallel or parallel to serial format. The individual data latches that make up a single shift register are all driven by a common clock (Clk) signal making them synchronous devices.

Shift register IC's are generally provided with a *clear* or *reset* connection so that they can be "SET" or "RESET" as required. Generally, shift registers operate in one of four different modes with the basic movement of data through a shift register being:

- Serial-in to Parallel-out (SIPO) - the register is loaded with serial data, one bit at a time, with the stored data being available at the output in parallel form.
- Serial-in to Serial-out (SISO) - the data is shifted serially "IN" and "OUT" of the register, one bit at a time in either a left or right direction under clock control.
- Parallel-in to Serial-out (PISO) - the parallel data is loaded into the register simultaneously and is shifted out of the register serially one bit at a time under clock control.
- Parallel-in to Parallel-out (PIPO) - the parallel data is loaded simultaneously into the register, and transferred together to their respective outputs by the same clock pulse.

The effect of data movement from left to right through a shift register can be presented graphically as:

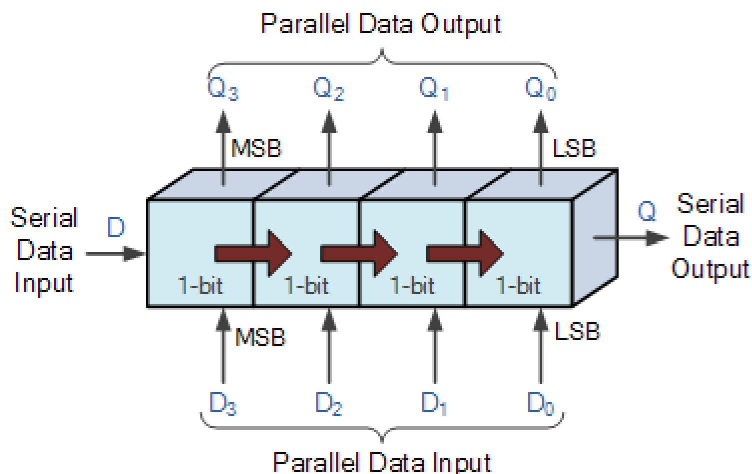


Fig: 2.15.a shift register

Also, the directional movement of the data through a shift register can be either to the left, (left shifting) to the right, (right shifting) left-in but right-out, (rotation) or both left and right shifting within the same register thereby making it *bidirectional*. In this tutorial it is assumed that all the data shifts to the right, (right shifting).

2.5.1 Serial-in to Parallel-out (SIPO) Shift Register

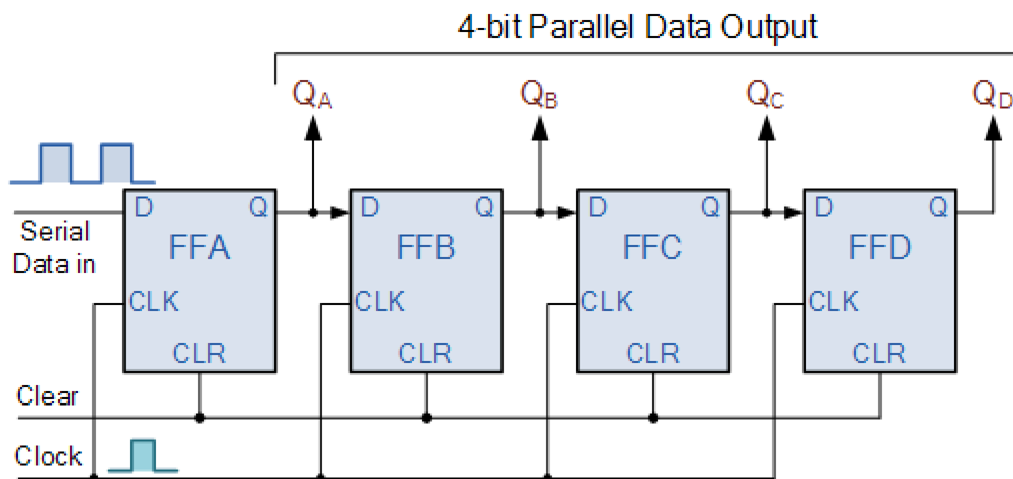


Fig: 2.16 a 4-bit Serial-in to Parallel-out Shift Register

The operation is as follows. Lets assume that all the flip-flops (FFA to FFD) have just been RESET (CLEAR input) and that all the outputs Q_A to Q_D are at logic level "0" ie, no parallel data output.

If a logic "1" is connected to the DATA input pin of FFA then on the first clock pulse the output of FFA and therefore the resulting Q_A will be set HIGH to logic "1" with all the other outputs still remaining LOW at logic "0". Assume now that the DATA input pin of FFA has returned LOW again to logic "0" giving us one data pulse or 0-1-0.

The second clock pulse will change the output of FFA to logic "0" and the output of FFB and Q_B HIGH to logic "1" as its input D has the logic "1" level on it from Q_A. The logic "1" has now moved or been "shifted" one place along the register to the right as it is now at Q_A.

When the third clock pulse arrives this logic "1" value moves to the output of FFC (Q_C) and so on until the arrival of the fifth clock pulse which sets all the outputs Q_A to Q_D back again to logic level "0" because the input to FFA has remained constant at logic level "0".

The effect of each clock pulse is to shift the data contents of each stage one place to the right, and this is shown in the following table until the complete data value of 0-0-0-1 is stored in the register. This data value can now be read directly from the outputs of $Q_A$to Q_D.

2.5.2 Serial-in to Serial-out (SISO) Shift Register

This shift register is very similar to the SIPO above, except were before the data was read directly in a parallel form from the outputs Q_A to Q_D, this time the data is allowed to flow straight through the register and out of the other end. Since there is only one output, the DATA leaves the shift register one bit at a time in a serial pattern, hence the name Serial-in to Serial-Out Shift Register or SISO**.**The SISO shift register is one of the simplest of the four configurations as it has only three connections, the serial input (SI) which determines what enters the left hand flip-flop, the serial output (SO) which is taken from the output of the right hand flip-flop and the sequencing clock signal (Clk). The logic circuit diagram below shows a generalized serial-in serial-out shift register.

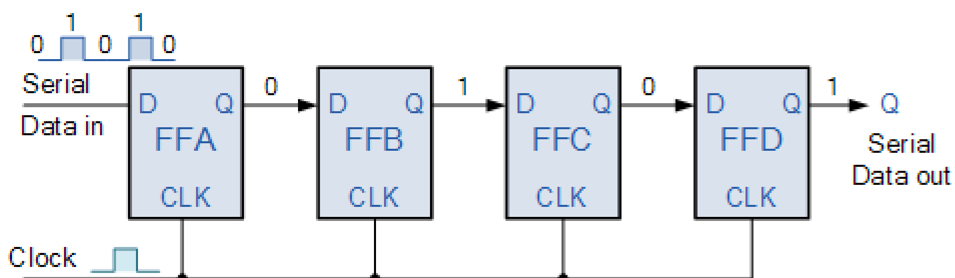


Fig: 2.17 a 4-bit Serial-in to Serial-out Shift Register

You may think what's the point of a SISO shift register if the output data is exactly the same as the input data. Well this type of Shift Register also acts as a temporary storage device or it can act as a time delay device for the data, with the amount of time delay being controlled by the number of stages in the register, 4, 8, 16 etc or by varying the application of the clock pulses. Commonly available IC's include the 74HC595 8-bit Serial-in to Serial-out Shift Register all with 3-state outputs.

2.5.3 Parallel-in to Serial-out (PISO) Shift Register

The Parallel-in to Serial-out shift register acts in the opposite way to the serial-in to parallel-out one above. The data is loaded into the register in a parallel format in which all the data bits enter their inputs simultaneously, to the parallel input pins P_A to P_D of the register. The data is then read out sequentially in the normal shift-right mode from the register at Q representing the data present at P_A to P_D.

This data is outputted one bit at a time on each clock cycle in a serial format. It is important to note that with this type of data register a clock pulse is not required to parallel load the register as it is already present, but four clock pulses are required to unload the data.

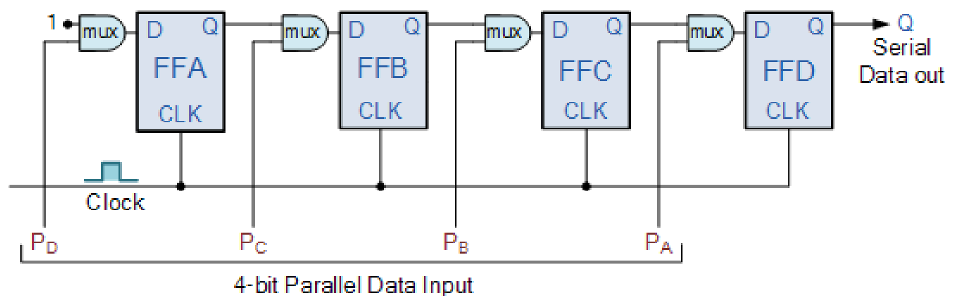


Fig: 2.18 a 4-bit Parallel-in to Serial-out Shift Register

As this type of shift register converts parallel data, such as an 8-bit data word into serial format, it can be used to multiplex

many different input lines into a single serial DATA stream which can be sent directly to a computer or transmitted over a communications line. Commonly available IC's include the 74HC166 8-bit Parallel-in/Serial-out Shift Registers.

2.5.4 Parallel-in to Parallel-out (PIPO) Shift Register

The final mode of operation is the Parallel-in to Parallel-out Shift Register. This type of shift register also acts as a temporary storage device or as a time delay device similar to the SISO configuration above. The data is presented in a parallel format to the parallel input pins P_A to P_D and then transferred together directly to their respective output pins Q_A to Q_A by the same clock pulse. Then one clock pulse loads and unloads the register. This arrangement for parallel loading and unloading is shown below.

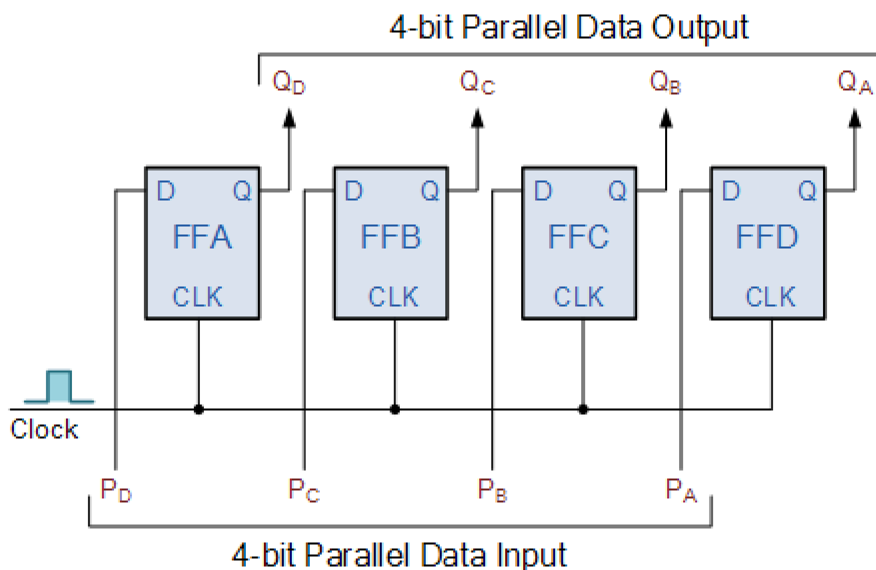


Fig: 2.19 a 4-bit Parallel-in to Parallel-out Shift Register

The PIPO shift register is the simplest of the four configurations as it has only three connections, the parallel input

(PI) which determines what enters the flip-flop, the parallel output (PO) and the sequencing clock signal (Clk).

Similar to the Serial-in to Serial-out shift register, this type of register also acts as a temporary storage device or as a time delay device, with the amount of time delay being varied by the frequency of the clock pulses. Also, in this type of register there are no interconnections between the individual flip-flops since no serial shifting of the data is required.

2.6 Binary Counters

A counter is a device which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal. The most common type is a sequential digital logic circuit with an input line called the "clock" and multiple output lines. The values on the output lines represent a number in the binary or BCD number system. Each pulse applied to the clock input increments or decrements the number in the counter.

A counter circuit is usually constructed of a number of flip-flops connected in cascade. Counters are a very widely-used component in digital circuits, and are manufactured as separate integrated circuits and also incorporated as parts of larger integrated circuits.

2.6.1 Asynchronous (ripple) counter

An asynchronous (ripple) counter is a single d-type flip-flop, with its J (data) input fed from its own inverted output. This circuit can store one bit, and hence can count from zero to one before it overflows (starts over from 0). This counter will increment once for every clock cycle and takes two clock cycles to overflow, so every cycle it will alternate between a transition from 0 to 1 and a transition from 1 to 0. Notice that this creates a new clock with a 50% duty cycle at exactly half the frequency of the input clock. If

this output is then used as the clock signal for a similarly arranged D flip-flop (remembering to invert the output to the input), one will get another 1 bit counter that counts half as fast. Putting them together yields a two-bit counter:

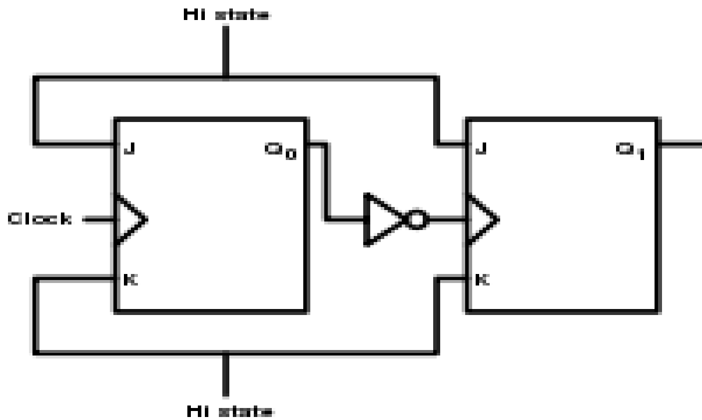


Fig: 2.20 An Asynchronous ripple counter

Cycle	Q1	Q0	(Q1:Q0) dec
0	0	0	0
1	0	1	1
2	1	0	2
3	1	1	3
4	0	0	0

Table: 2.5

You can continue to add additional flip-flops, always inverting the output to its own input, and using the output from the previous flip-flop as the clock signal. The result is called a ripple counter, which can count to $2^n - 1$ where n is the number of bits (flip-flop stages) in the counter. Ripple counters suffer from unstable outputs as the overflows "ripple" from stage to stage, but they do find frequent application as dividers for clock signals, where the instantaneous count is unimportant, but the division ratio overall is (to clarify this, a 1-bit counter is exactly equivalent to a divide by two circuit; the output frequency is exactly half that of the input when fed with a regular train of clock pulses).

The use of flip-flop outputs as clocks leads to timing skew between the count data bits, making this ripple technique incompatible with normal synchronous circuit design styles.

2.6.2 Synchronous counter

In synchronous counters, the clock inputs of all the flip-flops are connected together and are triggered by the input pulses. Thus, all the flip-flops change state simultaneously (in parallel). The circuit below is a 4-bit synchronous counter. The J and K inputs of FF0 are connected to HIGH. FF1 has its J and K inputs connected to the output of FF0, and the J and K inputs of FF2 are connected to the output of an AND gate that is fed by the outputs of FF0 and FF1. A simple way of implementing the logic for each bit of an ascending counter (which is what is depicted in the image to the right) is for each bit to toggle when all of the less significant bits are at a logic high state. For example, bit 1 toggles when bit 0 is logic high; bit 2 toggles when both bit 1 and bit 0 are logic high; bit 3 toggles when bit 2, bit 1 and bit 0 are all high; and so on.

Synchronous counters can also be implemented with hardware finite-state machines, which are more complex but allow for smoother, more stable transitions.Hardware-based counters are of this type.A simple way of implementing the logic for each bit of an ascending counter (which is what is depicted in the image to

the right) is for each bit to toggle when all of the less significant bits are at a logic high state

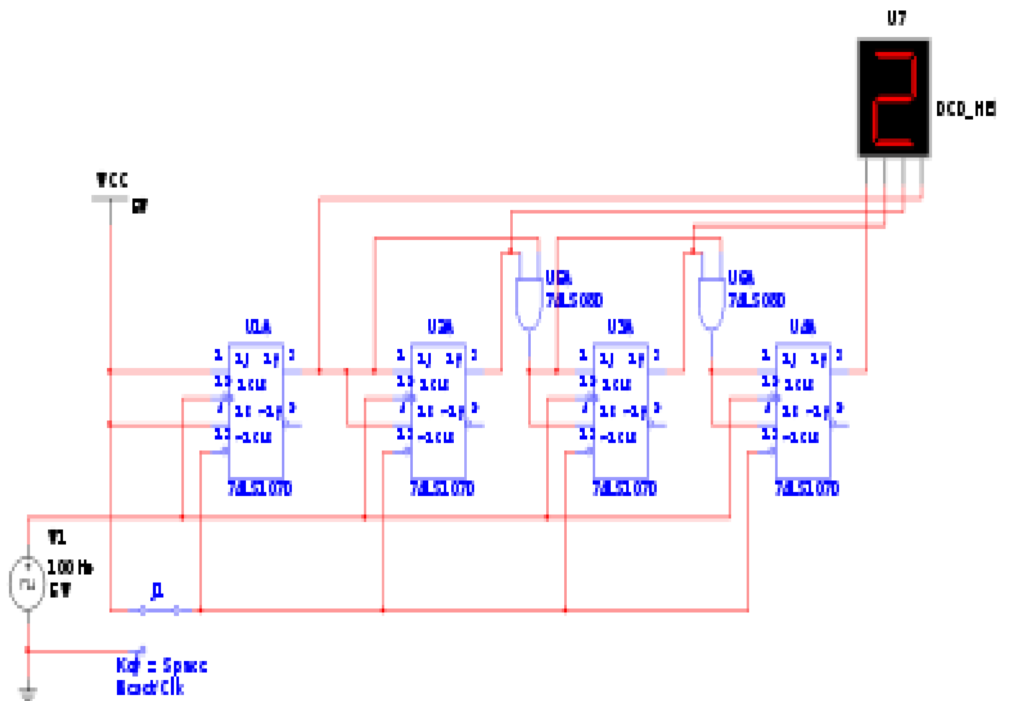


Fig: 2.21 A 4-bit synchronous counter using JK flip-flop

Chapter 3

Data Representation

3.1 Data types

Information that a Computer is dealing with

* Data
  - Numeric Data: Numbers(Integer, real)
  - Non-numeric: Data Letters, Symbols

* Relationship between data elements
  - Data Structures: Linear Lists, Trees, Rings, etc

* Program(Instruction)

Data
Numeric data - numbers(integer, real)
Non-numeric data - symbols, letters

3.1.1 Number System

Nonpositional number system
  - Roman number system

Positional number system
  - Each digit position has a value called a *weight* associated with it
  - Decimal, Octal, Hexadecimal, Binary

Base (or radix) R number
  - Uses R distinct symbols for each digit
  - Example AR = an-1 an-2 ... a1 a0 .a-1…a-m

$$V(A_R) = \sum_{i=-m}^{n-1} aiR^R$$

R = 10 Decimal number system,

R = 2 Binary number system

R = 8 Octal number system

R = 16 Hexadecimal

3.1.2 WHY POSITIONAL NUMBER SYSTEM IN THE DIGITAL COMPUTERS ?

Major Consideration is the *COST* and *TIME*

- Cost of building *hardware*

Arithmetic and Logic Unit, CPU,Communications

- Time to processing

Arithmetic - Addition of Numbers - *Table for Addition*

* Non-positional Number System

- Table for addition is infinite

--> Impossible to build, very expensive even

if it can be built

* Positional Number System

- Table for Addition is finite

--> Physically realizable, but cost wise

the smaller the table size, the less

expensive --> Binary is favorable to Decimal

3.1.3 CONVERSION OF BASES

Base R to Decimal Conversion

$V(A) = \Sigma a_k R^k$

$A = a_{n-1} a_{n-2} a_{n-3} \ldots a_0 . a_{-1} \ldots a_{-m}$

(736.4)8 = 7 x 82 + 3 x 81 + 6 x 80 + 4 x 8-1

= 7 x 64 + 3 x 8 + 6 x 1 + 4/8 = $(478.5)_{10}$

$(110110)_2 = ... = (54)_{10}$

$(110.111)_2 = ... = (6.785)_{10}$

$(F3)_{16}$ = ... = (243)10 (0.325)6 = ... = $(0.578703703 ....)_{10}$

Decimal to Base R number

Separate the number into its integer and fraction parts and convert each part

separately.

- Convert integer part into the base R number

--> successive divisions by R and accumulation of the remainders.

- Convert fraction part into the base R number

--> successive multiplications by R and accumulation of integer digits

COMPLEMENT OF NUMBERS

Complements - to convert positive to negative or vice versa

Two types of complements for base R number system: - R's complement and (R-1)'s

Complement

The (R-1)'s Complement

Subtract each digit of a number from (R-1)

Example - 9's complement of 83510 is 16410 - 1's complement of 10102 is 01012(bit by bit

complement operation)

The R's Complement

Add 1 to the low-order digit of its (R-1)'s complement

Complements - to convert positive to negative or vice versa

Example

- 10's complement of 83510 is 16410 + 1 = 16510

- 2's complement of 10102 is 01012 + 1 = 01102

FIXED POINT NUMBERS

Numbers: Fixed Point Numbers and Floating Point Numbers

Binary Fixed-Point Representation

$X = X_n X_{n-1}\ X_{n-2} \ldots X_1 X_0 .\ X_{-1} X_{-2} \ldots X_{-m}$

Sign Bit(x_n): 0 for positive - 1 for negative

Remaining Bits($X_n X_{n-1}\ X_{n-2} \ldots X_1 X_0 .\ X_{-1} X_{-2} \ldots X_{-m}$)

Following 3 representations:

Signed magnitude representation

Signed 1's complement representation

Signed 2's complement representation

Example: Represent +9 and -9 in 7 bit-binary number

Only one way to represent +9 ==> 0 001001

Three different ways to represent -9:

In signed-magnitude: 1 001001

In signed-1's complement: 1 110110

In signed-2's complement: 1 110111

Numbers: Fixed Point Numbers and Floating Point Numbers In general, in computers, fixed point numbers are represented either integer part only or fractional part only.

CHARACTERISTICS OF 3 DIFFERENT REPRESENTATIONS

Complement

Signed magnitude: Complement only the sign bit

Signed 1's complement: Complement all the bits including sign bit

Signed 2's complement: Take the 2's complement of the number, including its sign bit.

Maximum and Minimum Representable Numbers and Representation of Zero

$X = x_n \, x_{n-1} \ldots x_0 \, . \, x_{-1} \ldots x_{-m}$

Signed Magnitude

Max: $2_n - 2_{-m}$ 011 ... 11.11 ... 1

Min: $-(2_n - 2_{-m})$ 111 ... 11.11 ... 1

Zero: +0 000 ... 00.00 ... 0

-0 100 ... 00.00 ... 0

Signed 1’s Complement

Max: $2_n - 2_{-m}$ 011 ... 11.11 ... 1

Min: $-(2_n - 2_{-m})$ 100 ... 00.00 ... 0

Zero: +0 000 ... 00.00 ... 0

-0 111 ... 11.11 ... 1

Signed 2’s Complement

Max: 2_n - 2_{-m} 011 ... 11.11 ... 1

Min: -2_n 100 ... 00.00 ... 0

Zero: 0 000 ... 00.00 ... 0

When we type some letters or words, the computer translates them in numbers as computers can understand only numbers. A computer can understand positional number system where there are only a few symbols called digits and these symbols represent different values depending on the position they occupy in the number.

The data types found in memory of digital computers may be classified as beibg one of the following categories:

- Numbers used for arithmetic computations
- Alphabetical letters used in data processing
- Other discrete symbols used for special purposes

A value of each digit in a number can be determined using

- The digit
- The position of the digit in the number
- The base of the number system (where base is defined as the total number of digits available in the number system).

3.1.4 Decimal Number System

The number system that we use in our day-to-day life is the decimal number system. Decimal number system has base 10 as it uses 10 digits from 0 to 9. In decimal number system, the successive positions to the left of the decimal point represent units, tens, hundreds, thousands and so on.

Each position represents a specific power of the base (10). For example, the decimal number 1234 consists of the digit 4 in the units position, 3 in the tens position, 2 in the hundreds position, and 1 in the thousands position, and its value can be written as

```
(1x1000)+ (2x100)+ (3x10)+ (4xl)

(1x10^3)+ (2x10^2)+ (3x10^1)+ (4xl0^0)

1000 + 200 + 30 + 4

1234
```

As a computer programmer or an IT professional, you should understand the following number systems which are frequently used in computers.

S.N.	Number System and Description
1	**Binary Number System** Base 2. Digits used : 0, 1
2	**Octal Number System** Base 8. Digits used : 0 to 7
3	**Hexa Decimal Number System** Base 16. Digits used : 0 to 9, Letters used : A- F

3.1.5 Binary Number System

Characteristics of binary number system are as follows:

- Uses two digits, 0 and 1.
- Also called base 2 number system
- Each position in a binary number represents a 0 power of the base (2). Example 2^0
- Last position in a binary number represents a x power of the base (2). Example $2^x$where x represents the last position - 1.

Example:

Binary Number : 10101_2

Calculating Decimal Equivalent:

Step	Binary Number	Decimal Number
Step 1	10101_2	$((1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	10101_2	$(16 + 0 + 4 + 0 + 1)_{10}$

Step 3	10101_2	21_{10}

Note : 10101_2 is normally written as 10101.

3.1.6 Octal Number System

Characteristics of octal number system are as follows:

- Uses eight digits, 0,1,2,3,4,5,6,7.
- Also called base 8 number system
- Each position in an octal number represents a 0 power of the base (8). Example 8^0
- Last position in an octal number represents a x power of the base (8). Example $8^x$where x represents the last position - 1.

Example:

Octal Number : 12570_8

Calculating Decimal Equivalent:

Step	Octal Number	Decimal Number
Step 1	12570_8	$((1 \times 8^4) + (2 \times 8^3) + (5 \times 8^2) + (7 \times 8^1) + (0 \times 8^0))_{10}$
Step 2	12570_8	$(4096 + 1024 + 320 + 56 + 0)_{10}$

Step 3	12570_8	5496_{10}

Note : 12570_8 is normally written as 12570.

3.1.7 Hexadecimal Number System

Characteristics of hexadecimal number system are as follows:

- Uses 10 digits and 6 letters, 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.
- Letters represents numbers starting from 10. A = 10. B = 11, C = 12, D = 13, E = 14, F = 15.
- Also called base 16 number system
- Each position in a hexadecimal number represents a 0 power of the base (16). Example 16^0
- Last position in a hexadecimal number represents a x power of the base (16). Example 16^x where x represents the last position - 1.

Example:

Hexadecimal Number : $19FDE_{16}$

Calculating Decimal Equivalent:

Step	Binary Number	Decimal Number
Step 1	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (F \times 16^2) + (D \times 16^1) + (E \times 16^0))_{10}$

Step 2	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (15 \times 16^2) + (13 \times 16^1) + (14 \times 16^0))_{10}$
Step 3	$19FDE_{16}$	$(65536 + 36864 + 3840 + 208 + 14)_{10}$
Step 4	$19FDE_{16}$	106462_{10}

REPRESENTATION OF NUMBERS

Binary	Decimal	Octal	Hexadecimal
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Table: 3.1

3.2 Complements

In mathematics and computing, the method of complements is a technique used to subtract one number from another using only addition of positive numbers. This method was commonly used in mechanical calculators and is still used in modern computers.

3.2.1 Decimal number complements:

9’s complement of the decimal number N = (10n – 1) – N

= n (9’s) – N

i.e. {subtract each digit from 9}

Example -> 9’s complement of 134795 is 865204

Similarly

1’s complement of the binary number N = (2n -1) – N = n (1’s) – N

Example -> 1’s complement of 110100101 is 001011010

which can be obtained by replacing each one by a zero and each zero by one.

3.2.2 r’s complement:

10’s complement of the decimal number N = 10n – N = (r-1)’s complement + 1

Example -> 10’s complement of 134795 is 865205

Example -> find the 9’s and 10’s complements of 314700.

Answer -> 9’s complement = 685299

10’s complement=685300

Rule: To find the 10’s complement of a decimal number leave all leading zeros unchanged. Then subtract the first non-zero digit from 10 and all the remaining digits from 9’s.

The 2’s complement of a binary number is defined in a similar way.

Example: Find the 1’s and 2’s complements of the binary number 1101001101

Answer -> 1’s complement is 0010110010

2’s complement is 0010110011

Example: Find the 1’s and 2’s complements of 100010100

Answer -> 1’s complement is 011101011

2’s complement is 011101100

Subtraction using r’s complement

To find M-N in base r, we add M + r’s complement of N

Result is M + (rn – N)

1) If M > N then result is M – N + rn (rn is an end carry and can be neglected.

2) If M < N then result is rn –(N-M) which is the r’s complement of the result.

Example: Subtract (76425 – 28321) using 10’s complements.

Answer -> 10’s complement of 28321 is 71679

```
Then add -> 7 6 4 2 5
          + 7 1 6 7 9
          1 4 8 1 0 4 (Discard 1)
```

Therefore the difference is 48104 after discarding the end carry

Example: subtract (28531 – 345920)

Answer -> It is obvious that the difference is negative. We also have to work with the same number of digits, when dealing with complements.

10's complement of 345920 is 654080

```
        Then add -> 0 2 8 5 3 1
                  + 6 5 4 0 8 0
               ---------------------
  (No end carry)    6 8 2 6 1 1
```

Therefore the difference is negative and is equal to the 10's complement of the answer.

Difference is -> - 317389

The same rules apply to binary.

Example: subtract (11010011 – 10001100)

Answer -> 2's complement of 10001100 is 01110100

```
  Then add -> 1 1 0 1 0 0 1 1
            + 0 1 1 1 0 1 0 0
        --------------------------
  (Discard) 1 0 1 0 0 0 1 1 1
```

The difference is positive and is equal to 01000111

The same rules apply to subtraction using the (r-1)'s complements. The only difference is that when an end carry is generated, it is not discarded but added to the least significant digit of the result. Also, if no end carry is generated, then the answer is negative and in the (r-1)'s complement form.

Example: Subtract (76425 – 28321) using 9's complements.

Answer -> 9's complement of 28321 is 71678

```
Then add -> 7 6 4 2 5
          + 7 1 6 7 8
          ------------
          1 4 8 1 0 3
                    1
          ------------
(Difference) 4 8 1 0 4
```

Example: subtract (11010011 – 10001100) using 1's complement.

Answer -> 1's complement of 10001100 is 01110011

```
Then add -> 1 1 0 1 0 0 1 1
          + 0 1 1 1 0 0 1 1
          -----------------
          1 0 1 0 0 0 1 1 0
                          1
          -----------------
(Difference) 1 0 1 0 0 0 1 1 1
```

3.2.3 1's Complement

1's complement of a binary number is obtained simply by replacing each 1 by 0 and each 0 by 1. Alternately, 1's complement of a binary can be obtained by subtracting each bit from 1.

Example. Find 1's complement of (i) 011001 (ii) 00100111

Solution. (i) Replace each 1 by 0 and each 0 by 1

```
0 1  1  0  0  1
↓ ↓  ↓  ↓  ↓  ↓
1 0  0  1  1  0
```

So, 1's complement of 011001 is 100110.

(ii) Subtract each binary bit from 1.

```
   11 1 1 1 1 1 1
  –0 0 1 0 0 1 1 1
  ----------------
   1 1 0 1 1 0 0 0    ← 1's complement
```

one can see that both the method gives same result.

3.2.4 2's Complement

2's complement of a binary number can be obtained by adding 1 to its 1's complement.

Example. Find 2's complement of (i) 011001 (ii) 0101100

Solution. (i)

```
01 1 0 0 1 ← Number
10 0 1 1 0 ← 1's complement
       + 1 ← Add 1 to 1's complement
10 0 1 1 1 ← 2's complement
```

(ii)

```
01 0 1 1 0 0 ← Number
10 1 0 0 1 1 ← 1's complement
         + 1 ← Add 1 to 1's complement
10 1 0 1 0 0 ← 2's complement
```

3.3 Fixed and Floating point representation

3.3.1 Fixed point representation

Unsigned numbers are used to represent positive number, but to represent negative numbers we need negative notation. In normal arithmetic representation a negative number is indicated by minus sign, but in computers everything can be represented only by 1's and 0's. in addition to sign a number can also have decimal point, the position of the binary number is necessary to represent fraction, integer etc..

3.3.2 Integer representation

When an integer binary number is positive its represented by 0, and when the number is negative the sign is represented by 1. The number can be represented in three different ways:

(i) Signed –Magnitude representation
(ii) Signed – 1's complement representation
(iii) Signed – 2's complement representation

For example consider a signed number 14 stored in 8-bit register

+14 is represented as 00001110, the left most bit is '0' representing a positive number, but whereas -14 can be represented three different ways:

10001110 in Signed magnitude

11110001 in signed 1's complement representation

11110010 in signed 2's complement representation

3.3.2 Floating point representation

Scientific Notation:

- Science deals regularly with very large and very small numbers.
- To do so it adopts *Floating Point Notation*. Below are some examples:

e.g. 1. the distance between the Earth and Sun:

1.496×10^{11} meters = 149600000000

e.g. 2. the distance between an atomâ€™s nucleus and an electron:

0.529×10^{-10} meters = 0.0000000000529

Since the base of the number system can be inferred, the "x10" part is required.

But a way was needed to distinguish the "mantissa" (e.g. 1.496 and 0.529 above) from the "exponent" (10^{11} and 10^{-10} above) .

So these are alternately expressed in the form: +1.496E11, and +.529E-10

Floating-Point notation in Binary consists of 3 parts:

1. a *Sign* bit ["0" is non-negative (+), "1" is negative (-)
2. the *Exponent* and
3. the *Mantissa*.

- In an 8-bit pattern example below, the most significant (the right-most) bit is the sign-bit, followed by a 3-bit exponent (expressed in excess notation) followed by a 4-bit mantissa.

Important Note: in a ***normalized*** floating point notation, the mantissa must begin, i.e., the most significant bit of the mantissa must be a "1" and the *radix* point is assumed to be at the left of the mantissa. In this course, we will always use the normalized system.

A. Decoding Binary Floating Point Notation

1. Analyse bit pattern according to the 3 field patterns (sign, exponent, mantissa)

2. Extract the mantissa and place the radix point on its left side. E.G., .1001

3. Extract the contents of the exponent field and interpret it using the *Excess* notation. (This 3-bit example is excess (4) notation so it represents 5-4 or +1.)

4. Move the radix the same number of positions as was determined from Step 3 above.

Move the radix <u>right</u> the number of bit position indicated by the exponent value if the exponent is positive value.

Move the radix <u>left</u> the number of bit position indicated by the exponent value (add 0's as necessary as placeholders) if exponent is negative value.

5. Using the original sign bit, represent the decoded number (in decimal.)

E.G. 01011001

Sign Exponent Mantissa

Bit in Excess(4) (Normalized)

The Sign bit is 0 so the number represented is a non-negative (+) number

Next, the number 101 in excess (4) notation is 5-4 that is +1; a positive exponent moves the radix to the right and a negative exponent moves the radix to the left.

The (normalized) mantissa 1001 is assumed to be .1001; after applying the exponent by moving the radix 1 position to the right, it becomes 1.001 or 1 and 1/8th.

Therefore the number 01011001 (in *normalized* floating point notation) represents the value +1 1/8

3.4 Other binary codes

In the coding, when numbers, letters or words are represented by a specific group of symbols, it is said that the number, letter or word is being encoded. The group of symbols is called as a code. The digital data is represented, stored and transmitted as group of binary bits. This group is also called as binary code. The binary code is represented by the number as well as alphanumeric letter.

Advantages of Binary Code

Following is the list of advantages that binary code offers.

- Binary codes are suitable for the computer applications.
- Binary codes are suitable for the digital communications.
- Binary codes make the analysis and designing of digital circuits if we use the binary codes.
- Since only 0 & 1 are being used, implementation becomes easy.

Classification of binary codes

The codes are broadly categorized into following four categories.

- Weighted Codes
- Non-Weighted Codes
- Binary Coded Decimal Code
- Alphanumeric Codes
- Error Detecting Codes
- Error Correcting Codes

Weighted Codes

Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight. Several systems of the codes are

used to express the decimal digits 0 through 9. In these codes each decimal digit is represented by a group of four bits.

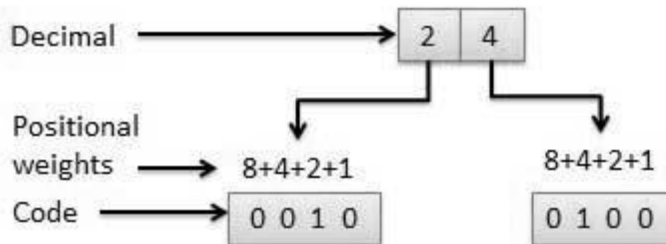


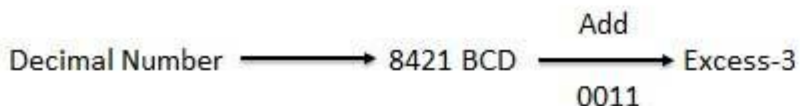
Fig: 3.1

Non-Weighted Codes

In this type of binary codes, the positional weights are not assigned. The examples of non-weighted codes are Excess-3 code and Gray code.

Excess-3 code

The Excess-3 code is also called as XS-3 code. It is non-weighted code used to express decimal numbers. The Excess-3 code words are derived from the 8421 BCD code words adding (0011)2 or (3)10 to each code word in 8421. The excess-3 codes are obtained as follows −



Example

Decimal	BCD 8 4 2 1	Excess-3 BCD + 0011
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

Table 3.1

Gray Code

It is the non-weighted code and it is not arithmetic codes. That means there are no specific weights assigned to the bit position. It has a very special feature that, only one bit will change each time the decimal number is incremented as shown in fig. As only one bit changes at a time, the gray code is called as a unit distance code. The gray code is a cyclic code. Gray code cannot be used for arithmetic operation.

Decimal	BCD	Gray
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1

Table 3.2

Application of Gray code

- Gray code is popularly used in the shaft position encoders.
- A shaft position encoder produces a code word which represents the angular position of the shaft.

Binary Coded Decimal (BCD) code

In this code each decimal digit is represented by a 4-bit binary number. BCD is a way to express each of the decimal digits with a binary code. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111). But in BCD code only first ten of these are used (0000 to 1001). The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD.

Decimal	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Table 3.3

Advantages of BCD Codes

- It is very similar to decimal system.
- We need to remember binary equivalent of decimal numbers 0 to 9 only.

Disadvantages of BCD Codes

- The addition and subtraction of BCD have different rules.
- The BCD arithmetic is little more complicated.
- BCD needs more number of bits than binary to represent the decimal number. So BCD is less efficient than binary.

Alphanumeric codes

A binary digit or bit can represent only two symbols as it has only two states '0' or '1'. But this is not enough for communication between two computers because there we need many more symbols for communication. These symbols are required to represent 26 alphabets with capital and small letters, numbers from 0 to 9, punctuation marks and other symbols.

The alphanumeric codes are the codes that represent numbers and alphabetic characters. Mostly such codes also represent other characters such as symbol and various instructions necessary for conveying information. An alphanumeric code should at least represent 10 digits and 26 letters of alphabet i.e. total 36 items. The following three

alphanumeric codes are very commonly used for the data representation.

- American Standard Code for Information Interchange (ASCII).
- Extended Binary Coded Decimal Interchange Code (EBCDIC).
- Five bit Baudot Code.

ASCII code is a 7-bit code whereas EBCDIC is an 8-bit code. ASCII code is more commonly used worldwide while EBCDIC is used primarily in large IBM computers.

3.5 Error Detection codes

What is Error?

Error is a condition when the output information does not match with the input information. During transmission, digital signals suffer from noise that can introduce errors in the binary bits travelling from one system to other. That means a 0 bit may change to 1 or a 1 bit may change to 0.

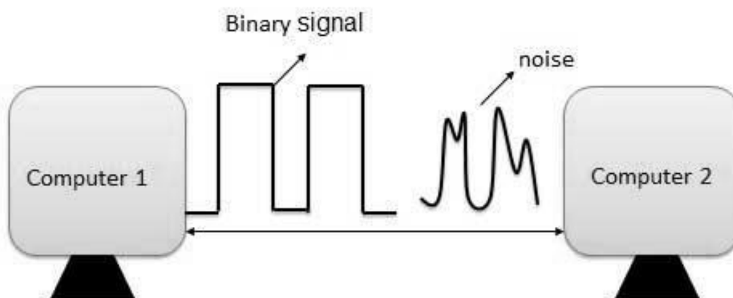


Fig: 3.2

Error-Detecting codes

Whenever a message is transmitted, it may get scrambled by noise or data may get corrupted. To avoid this, we use error-detecting codes which are additional data added to a given digital message to help us detect if an error occurred during transmission of the message. A simple example of error-detecting code is parity check.

Error-Correcting codes

Along with error-detecting code, we can also pass some data to figure out the original message from the corrupt message that we received. This type of code is called an error-correcting code. Error-correcting codes also deploy the same strategy as error-detecting codes but additionally, such codes also detect the exact location of the corrupt bit.

In error-correcting codes, parity check has a simple way to detect errors along with a sophisticated mechanism to determine the corrupt bit location. Once the corrupt bit is located, its value is reverted (from 0 to 1 or 1 to 0) to get the original message.

How to Detect and Correct Errors?

To detect and correct the errors, additional bits are added to the data bits at the time of transmission.

- The additional bits are called **parity bits**. They allow detection or correction of the errors.
- The data bits along with the parity bits form a **code word**.

Parity Checking of Error Detection

It is the simplest technique for detecting and correcting errors. The MSB of an 8-bits word is used as the parity bit and the

remaining 7 bits are used as data or message bits. The parity of 8-bits transmitted word can be either even parity or odd parity.

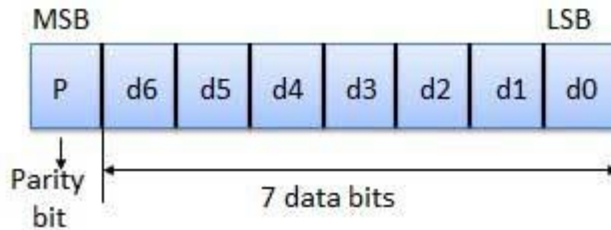


Fig: 3.3

Even parity -- Even parity means the number of 1's in the given word including the parity bit should be even (2,4,6,....).

Odd parity -- Odd parity means the number of 1's in the given word including the parity bit should be odd (1,3,5,....).

Use of Parity Bit

The parity bit can be set to 0 and 1 depending on the type of the parity required.

- For even parity, this bit is set to 1 or 0 such that the no. of "1 bits" in the entire word is even. Shown in fig. (a).
- For odd parity, this bit is set to 1 or 0 such that the no. of "1 bits" in the entire word is odd. Shown in fig. (b).

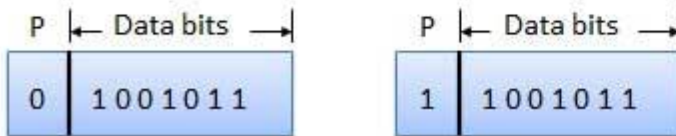


Fig. (a)

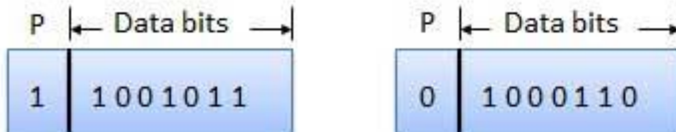


Fig. (b)

Fig: 3.4

How Does Error Detection Take Place?

Parity checking at the receiver can detect the presence of an error if the parity of the receiver signal is different from the expected parity. That means, if it is known that the parity of the transmitted signal is always going to be "even" and if the received signal has an odd parity, then the receiver can conclude that the received signal is not correct. If an error is detected, then the receiver will ignore the received byte and request for retransmission of the same byte to the transmitter.

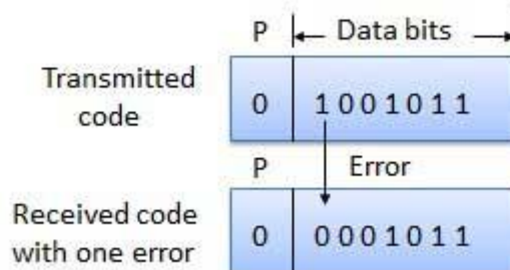


Fig: 3.4

Chapter 4

Register Transfer and Micro operations

4.1 Register Transfer language

In computer science, register transfer language (RTL) is a kind of intermediate representation (IR) that is very close to assembly language, such as that which is used in acompiler. Academic papers and textbooks also often use a form of RTL as an architecture-neutral assembly language.

Microoperations

• Digital systems are modular in nature, with modules containing registers, decoders, arithmetic elements, control logic, etc.

• These digital components are defined by the registers that they contain and the operations performed on their data. These operations are called microoperations.

• Microoperations are elementary operations performed on the information stored in one or more registers.

Hardware Organization

• The hardware organization of a digital computer is best defined by specifying:

– The set of register that it contains and their function.

– The sequence of microoperations performed on the

binary information stored in the registers.

– The control signals that initiates the sequence of

microoperations.

Register Transfer Language

• A register transfer language is a notation used to describe the microperation transfers between registers.

• It is a system for expressing in symbolic form the microoperation sequences among register that are used to implement machine-language instructions.

Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic

- The modules are interconnected with common data and control paths to form a digital computer system
- The operations executed on data stored in registers are called *microoperations*
- A microoperation is an elementary operation performed on the information stored in one or more registers
- Examples are shift, count, clear, and load
- Some of the digital components from before are registers that implement microoperations
- The internal hardware organization of a digital computer is best defined by specifying
  - The set of registers it contains and their functions

- The sequence of microoperations performed on the binary information stored
- The control that initiates the sequence of microoperations

• Use symbols, rather than words, to specify the sequence of microoperations

• The symbolic notation used is called a *register transfer language*

• A programming language is a procedure for writing symbols to specify a given computational process

• Define symbols for various types of microoperations and describe associated hardware that can implement the microoperations

4.2 Register Transfer

Registers are denoted by capital letters and are

sometimes followed by numerals, e.g.,

– MAR – Memory Address Register (holds addresses for the memory unit)

– PC – Program Counter (holds the next instruction's address)

– IR – Instruction Register (holds the instruction being executed)

– R1 – Register 1 (a CPU register)

• We can indicate individual bits by placing them in parentheses, e.g., PC(8-15), R2(5), etc.

Block Diagrams of Registers

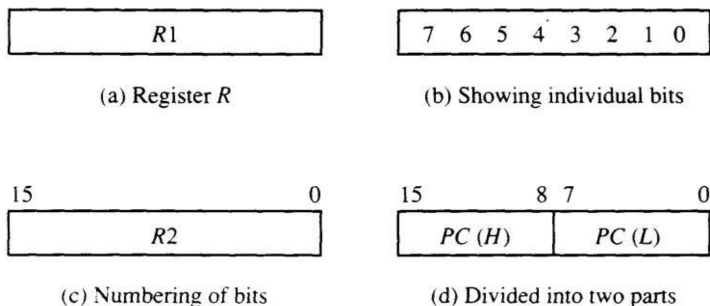


Fig: 4.1 Block diagram for Registers

Designate information transfer from one register to another by

$$R2 \leftarrow R1$$

This statement implies that the hardware is available

- o The outputs of the source must have a path to the inputs of the destination
- o The destination register has a parallel load capability

If the transfer is to occur only under a predetermined control condition, designate it by

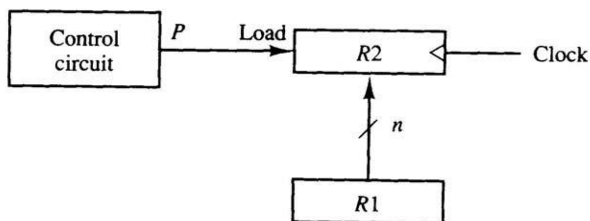
If (P = 1) *then* (R2 ← R1)

or,

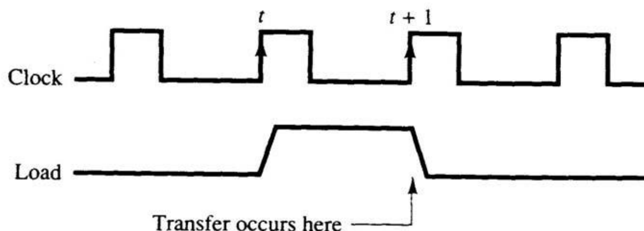
P:R2← R1,

where P is a control function that can be either 0 or 1

Every statement written in register transfer notation implies the presence of the required hardware construction



(a) Block diagram



(b) Timing diagram

Fig: 4.2 Transfer from R1 to R2 when P=1

It is assumed that all transfers occur during a clock edge transition

All microoperations written on a single line are to be executed at the same time T: R2 ← R1, R1 ← R2 , the above transfers will be done when T = 1.

TABLE Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	MAR, $R2$
Parentheses ( )	Denotes a part of a register	$R2(0-7)$, $R2(L)$
Arrow ←	Denotes transfer of information	$R2 \leftarrow R1$
Comma ,	Separates two microoperations	$R2 \leftarrow R1$, $R1 \leftarrow R2$

Table: 4.1

4.3 Bus and Memory Transfers

A computer consists of many registers and data paths, which are used for transferring the data from one register to another register. Consider that separate wires are used for linking registers, than the huge number of wires in the circuit will create a mess and increase the complexity of the system. This issue is resolved by using common bus system in which, the information is transferred in multiple configuration register. In a bus structure, common lines are used where; each line is used for transferring each bit of the binary data at a time.

Multiplexers are used for constructing a common bus and are allowed to select the source register whose information will be placed on the bus. Consider an example in which there are four registers A, B, C, and D. All these register contains 4 bits numbered from 0 to 3. The bus consist four multiplexers and two selection lines namely, S0 and S1, which are connected to the selection input of all four multiplexers. The following table represents the working of bus system.

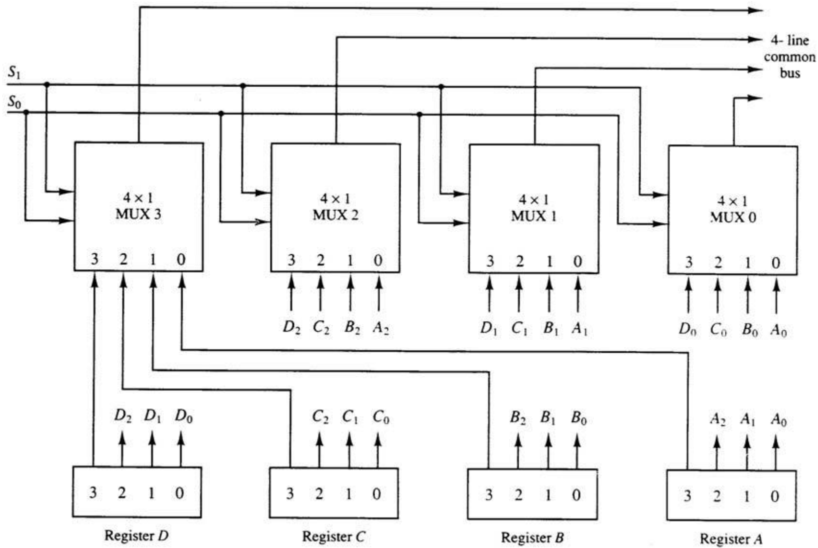


Fig: 4.3 Bus system for four registers

S0	S1	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Table: 4.2 Function table for bus in fig: 4.3

When 0 is selected in both selection lines i.e. S0S1 = 00, then we get A0 as output from the MUX0. Similarly we will get A1 from MUX1, A2 from MUX2, and A3 from MUX3. Considering these outputs from the four multiplexers, we will get A3A2A1A0 at the common bus. Hence, register A will be selected for the transfer.

Similarly, when S0S1 = 01, then the output from the multiplexer will be B3B2B1B0 and register B will be selected for the data transfer. When S0S1 = 10, then the output from the multiplexer will be C3C2C1C0 and register C will be selected for the data transfer. When S0S1 = 11, then the output from the multiplexer will be D3D2D1D0 and register D will be selected for the data transfer.

When the data is transferred from the bus to register or from the register to bus, following notations is used.

BUS <- A, B <- BUS

In the above notation, the data from the register A is being transferred to the BUS and the data from the BUS is loaded to the register B. This notation can also be represented as B <- A, if the bus exist in the system.

Find all the help you need for your homework help and assignment help at Transtutors.com. Our team of experts is capable of providing homework help and assignment help for all levels ranging from school level to undergraduate and graduate level. With us you can be rest assured that all the resource for the homework help and assignment help provided will be original and plagiarism free.

- Rather than connecting wires between all registers, a common bus is used
- A bus structure consists of a set of common lines, one for each bit of a register
- Control signals determine which register is selected by the bus during each transfer
- Multiplexers can be used to construct a common bus
- Multiplexers select the source register whose binary information is then placed on the bus
- The select lines are connected to the selection inputs of the multiplexers and choose the bits of one register

In general, a bys system will multiplex *k* registers of *n* bits each to produce an *n*-line common bus. This requires *n* multiplexers – one for each bit. The size of each multiplexer must be *k* x 1. The number of select lines required is *log k.* To transfer information from the bus to a register, the bus lines are connected to the inputs of all destination registers and the corresponding load control line must be activated . Rather than listing each step as

$$BUS \leftarrow C,\ R1 \leftarrow BUS,$$

use R1 ← C, since the bus is implied

Three State Buffers

In digital electronics three-state, tri-state, or 3-state logic allows an output port to assume a high impedance state in addition to the 0 and 1 logic levels, effectively removing the output from the circuit. This allows multiple circuits to share the same output line or lines Three-state buffers can also be used to implement efficient multiplexers, especially those with large numbers of inputs.

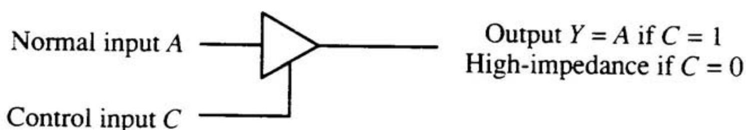


Fig: 4.4 Graphic symbols for three state buffers

The three-state buffer gate has a normal input and a control input which determines the output state

- With control 1, the output equals the normal input
- With control 0, the gate goes to a high-impedance state

• This enables a large number of three-state gate outputs to be connected with wires to form a common bus line without endangering loading effects

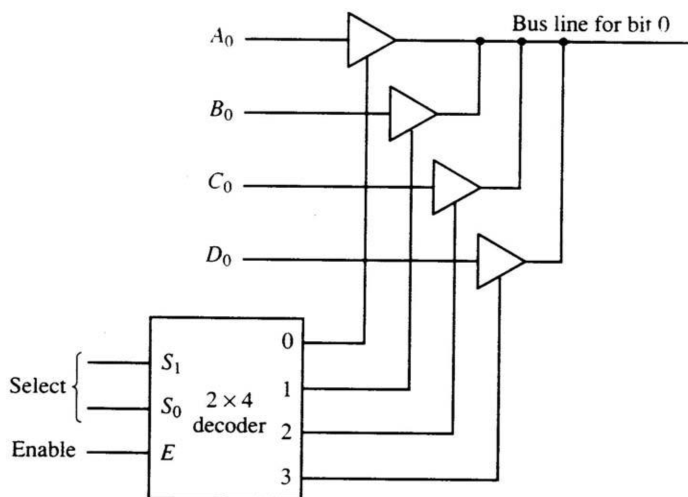


Fig: 4.5 Bus lines with three-state buffers

Decoders are used to ensure that no more than one control input is active at any given time. This circuit can replace the multiplexer in Figure 4.3. To construct a common bus for four registers of *n* bits each using three-state buffers, we need *n* circuits with four buffers in each. Only one decoder is necessary to select between the four registers

Memory Transfer

The transfer of information from a memory word to the outside environment is called a read operation. The transfer of a new information which has to be stored into the memory is called a write operation. A symbol M will be used to represent a memory word. It is also necessary to specify the address of M when writing

memory transfer operations; this is indicated by enclosing the address in square brackets following the letter M.

The read operation can be stated as:

$$\text{Read: } DR \leftarrow M[AR]$$

The write operation can be stated as:

$$\text{Write: } M[AR] \leftarrow R1$$

Microoperations

Microoperations are classified into four categories:

– Register transfer microoperations (data moves from register to register)

– Arithmetic microoperations (perform arithmetic on data in registers)

– Logic microoperations (perform bit manipulation on data in registers)

– Shift microoperations (perform shift on data in registers)

4.4 Arithmetic Microoperations

Unlike register transfer microoperations, arithmetic microoperations change the information content.

The basic arithmetic microoperations are:

– addition

– subtraction
– increment
– decrement
– shift

The Register Transfer Language statement R3 <- R1 + R2 indicates an add microoperation. We can similarly specify the other arithmetic microoperations. Multiplication and division are not considered. Multiplication is implemented by a sequence of adds and shifts. Division is implemented by a sequence of substracts and shifts.

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

Table: 4.3 Arithmetic microoperations

Binary Adder

We implement a binary adder with registers to hold the data and a digital circuit to perform the addition (called a *binary adder*). The binary adders is constructed using full adders connected in cascade so that the carry produced by one full adder becomes an input for the next.

Adding two *n*-bit numbers requires *n* full adders. The *n* data bits for *A* and *B* might come from *R1*and *R2* respectively

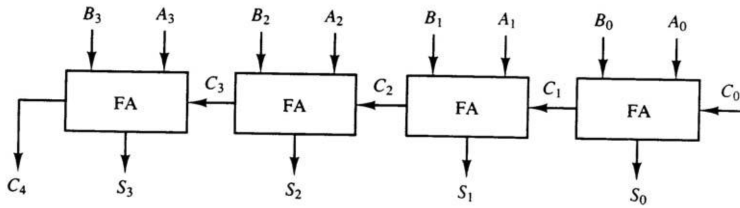


Fig: 4.6 4-bit binary adder

Adder-Subtracter

Subtracting ***A – B*** is most easily done by adding ***B'*** to ***A*** and then adding 1. This makes it convenient to combine both addition and subtraction into one circuit, called an addersubtracter.

• M is the mode indicator

– M = 0 indicates addition (B is left alone and C0 is 0)

– M = 1 indicates subtraction (B is complement and C0 is 1).

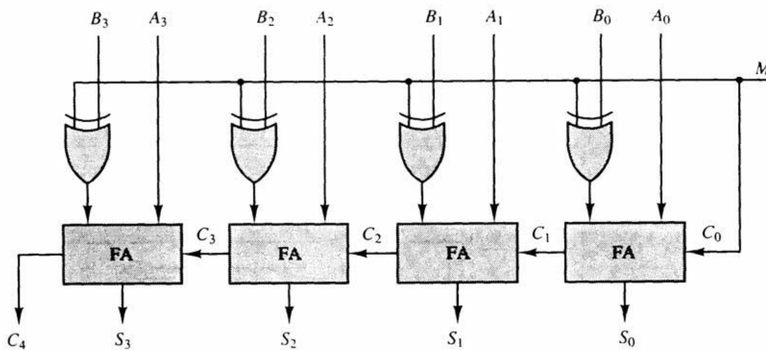


Fig: 4.7 4-bit adder-subtractor

Binary Incrementer

The binary incrementer adds 1 to the contents of a register, e.g., a register storing 0101 would have 0110 in it after being incremented. • There are times when we want incrementing done independent of a register. We can accomplish this with a series of cascading half-adders.

The increment microoperation adds one to a number in a register This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one. If the increment is to be performed independent of a particular register, then use half-adders connected in cascade. An *n*-bit binary incrementer requires *n* half-adders

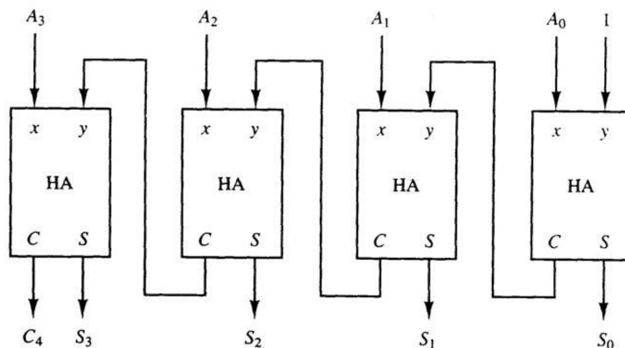


Fig: 4.8 4-bit binary incrementer

Arithmetic Circuit

Each of the arithmetic microoperations can be implemented in one composite arithmetic circuit

- The basic component is the parallel adder

- Multiplexers are used to choose between the different operations

- The output of the binary adder is calculated from the

  following sum: $D = A + Y + C_{in}$

We can implement 7 arithmetic microoperations (add, add with carry, subtract, subtract with borrow, increment, decrement and transfer) with one circuit. • We provide a series of cascading full adders with Ai and the output of a 4x1 multiplexer. The multiplexers' inputs are two selects, Bi, Bi', logical 0 and logical 1. Which of these four values we provide (together with the carry) determines which microoperation is performed.

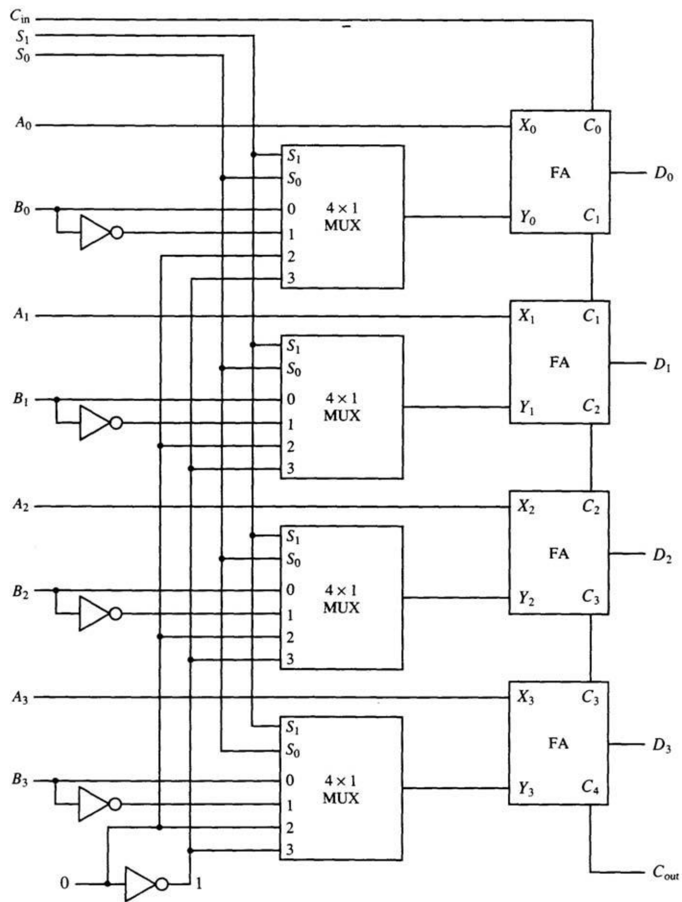


Fig: 4.9 4-bit Arithmetic Circuit

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Table: 4.4 Arithmetic circuit function table

When S1S0 = 00, the MUX provides B. The result is Add (for Cin = 0) or Add With Carry (for Cin =1).

When S1S0 = 01, the MUX provides B'. The result is Subtract with Borrow (for Cin = 0) or Subtract (for Cin = 1).

When S1S0 = 10, the MUX provides 0. The result is Transfer (for Cin = 0) or Increment (for Cin =1).

When S1S0 = 11, the MUX provides 1. The result is Decrement (for Cin = 0) or Transfer (for Cin =1).

4.5 Logic Microoperations

Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately

Example: the XOR of R1 and R2 is symbolized by

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

Example: R1 = 1010 and R2 = 1100

1010 Content of R1

1100 Content of R2
0110 Content of R1 after P = 1

Symbols used for logical microoperations:

o OR: ∨

o AND: ∧

o XOR: ⊕

The + sign has two different meanings: logical OR and summation

- When + is in a microoperation, then summation
- When + is in a control function, then OR
  - Example: P + Q: R1 ← R2 + R3, R4 ← R5 ∨ R6
- There are 16 different logic operations that can be performed with two binary variables

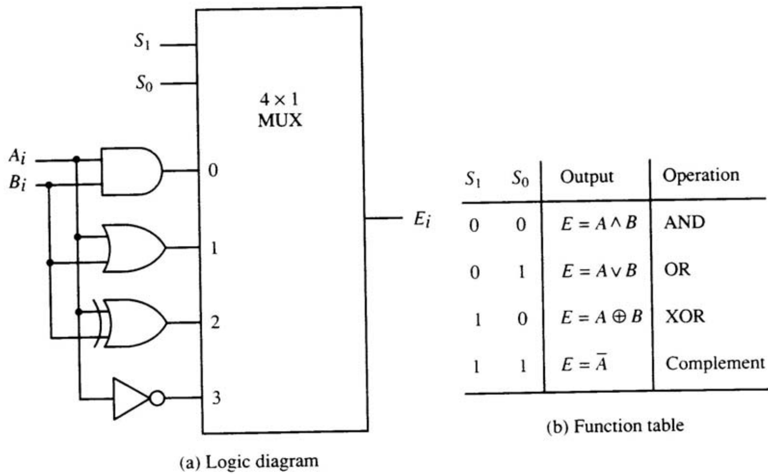
x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table: 4.5 Truth tables for 16 functions of two variables

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

Table: 4.5 Sixteen Logic Microoperations

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers. All 16 microoperations can be derived from using four logic gates



S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

Fig: 4.10 One stage of logic circuit

Logic microoperations can be used to change bit values, delete a group of bits, or insert new bit values into a register

Logic Operations allow us to manipulate

individual bits in ways that we could not do

otherwise.

• These applications include:

– selective set

– selective complement

– select clear

– mask

– insert

– clear

- **selective-set**

  The *selective-set* operation sets to 1 the bits in A where there are corresponding 1's in B

  1010 A before
  1100 B (logic operand)
  1110 A after

  $A \leftarrow A \vee B$

- **selective-complement**

  The *selective-complement* operation complements bits in A where there are corresponding 1's in B

  1010 A before
  1100 B (logic operand)
  0110 A after
  $A \leftarrow A \oplus B$

- **selective-clear**

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

  1010 A before
  1100 B (logic operand)
  0010 A after
  $A \leftarrow A \wedge \overline{B}$

- **mask**

The *mask* operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0's in B

1010 A before
1100 B (logic operand)
1000 A after

$A \leftarrow A \wedge B$

- **insert**

The *insert* operation inserts a new value into a group of bits. This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

0110 1010 A before
0000 1111 B (mask)
0000 1010 A after masking
0000 1010 A before
1001 0000 B (insert)
1001 1010 A after insertion

- **clear**

The *clear* operation compares the bits in A and B and produces an all 0's result if the two number are equal

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

If A& B are both 1 or both 0, this produces 0.This is done using the logical-AND operation and ***B***.

4.6 Shift Microoperations

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations.

There are three types of shifts: logical, circular, and arithmetic

- A *logical shift* is one that transfers 0 through the serial input

- The symbols *shl* and *shr* are for logical shift-left and shift-

  Right by one position R1 ← shl R1

- The *circular shift* (aka rotate) circulates the bits of the register around the two ends without loss of information

- The symbols *cil* and *cir* are for circular shift left and right

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

Table: 4.6 Shift Microoperations

- The *arithmetic shift* shifts a signed binary number to the left or right
- To the left is multiplying by 2, to the right is dividing by 2
- Arithmetic shifts must leave the sign bit unchanged
- A sign reversal occurs if the bit in R_{n-1} changes in value after the shift
- This happens if the multiplication causes an overflow
  - An overflow flip-flop V_s can be used to detect the overflow $V_s = R_{n-1} \oplus R_{n-2}$

If $V_s = 0$, there is no overflow, but if $V_s = 1$, there is an overflow and a sign reversal after the shift. V_s must be transferred into the overflow flio-flop with the same clock pulse that shifts the register.

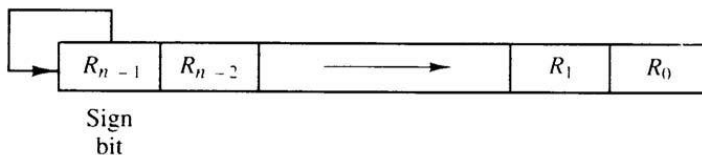
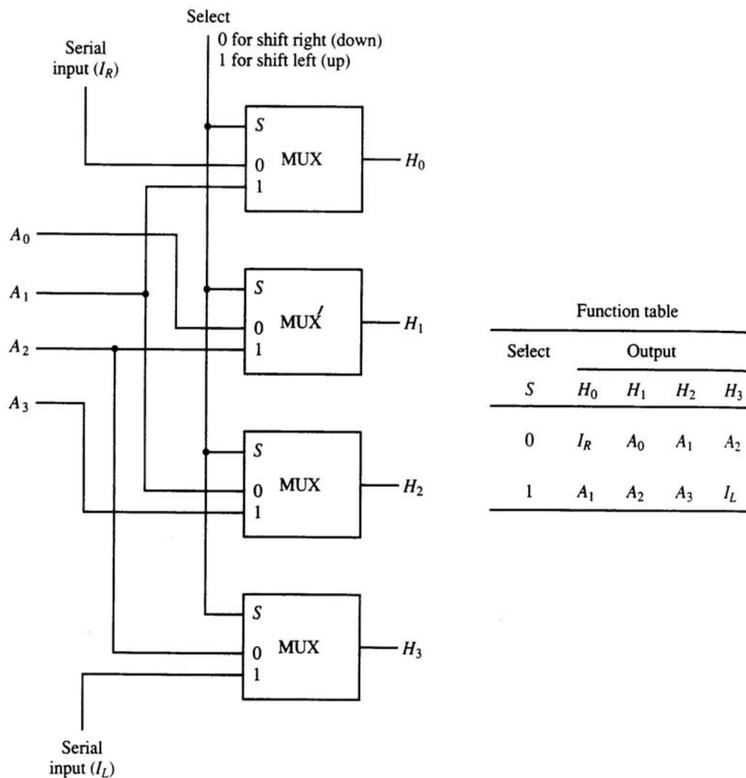


Fig: 4.11 Arithmetic right shift

- A bi-directional shift unit with parallel load could be used to implement this
- Two clock pulses are necessary with this configuration: one to load the value and another to shift
- In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit

• The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register

• This can be constructed with multiplexers



Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Fig: 4.12 Combinational circuit shifter

4.7 Arithmetic Logic Shift Unit

The *arithmetic logic unit (ALU)* is a common operational unit connected to a number of storage registers. To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU. The ALU performs an operation and the result is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period

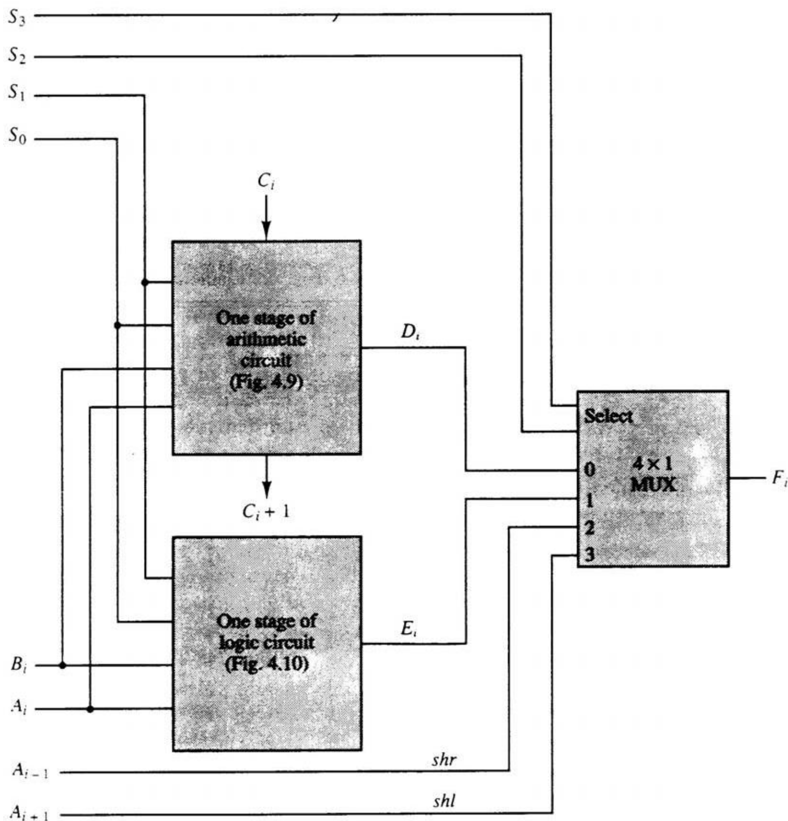


Fig: 4.13 One stage of Arithmetic logic shift unit

Operation select					Operation	Function
S_3	S_2	S_1	S_0	C_{in}		
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F =$ shr A	Shift right A into F
1	1	×	×	×	$F =$ shl A	Shift left A into F

Table: 4.7 Function table for arithmetic logic shift unit

Chapter 5

Basic Computer Organization and Design

Here we start our discussion definitions and a review of the basic organization of a computer and of a CPU. We also specify the internal components of the CPU, i.e. the registers and data paths. We next describe the instruction formats and instructions for this CPU. The Basic Computer has three different formats for its instructions; each of the 25 instructions follows only one of these three formats. We will review these formats and the overall function of each instruction. Next we look at the control signals used in this design. These signals are used to trigger micro-operations and coordinate data manipulation within the computer. We also show the hardware to generate these signals. We then get to the heart of th e design: the machine cycles which fetch, decode and execute these instructions. By using the control signals to enable micro-operations properly, the CPU realizes its instruction set. We first look at the fetch and indirect cycles. Then we review the individual execute cycles. We next look at input/output operations and interrupts. The Basic Computer has one input port and output port, so we don't have to worry about port addressing in our design. Inputs and outputs are used to trigger interrupts in this computer. We examine the I/O hardware and the interrupt cycle code. We then present an example of the hardware design for one of the internal components to further illustrate the design process. Finally, concluding remarks are presented.

5.1 Instruction Codes

Instruction code can be defined as a group of bits that tell the computer to perform a specific operation

The instruction code is an opcode plus additional information, such as a memory address. It is not the micro-operations. In terms

of programming, it is closest to a single assembly language instruction.

A program can be defined as a set of instructions that specify the *operations*, *operands*, and the *sequence* by which processing has to occur.

The computer reads each instruction from memory and places it in a control register. The control then interprets the binary codeof the instruction and proceeds to execute itby issuing a sequence of microoperations.

Instruction Code :

- A group of bits that instruct the computer to perform a Specific operation
- It is usually divided into parts(refer to Fig. 5-1 instruction format)

Operation Code :

- The most basic part of an instruction code
- A group of bits that define such operations as add, subtract, multiply, shift, and complement

Stored Program Organization

A stored program concept is one in which first the program and data are stored in the main memory and then the processor fetches instructions and executes them, one after another. A stored-program computer is one which stores program instructions in electronic memory. Often the definition is extended with the requirement that the treatment of programs and data in memory be interchangeable or uniform.

The CPU coordinates data transfers between itself and memory or I/O devices. The paths shown here not only carry data, but also the control signals which cause data to be transferred. They also carry address information which is used to select the correct memory location or I/O port address.

The control unit issues signals to coordinate functions of the ALU, the registers and external hardware. By issuing these signals in the proper order, they cause a sequence of operations to occur. By performing this sequence, an instruction is fetched, decoded and executed.

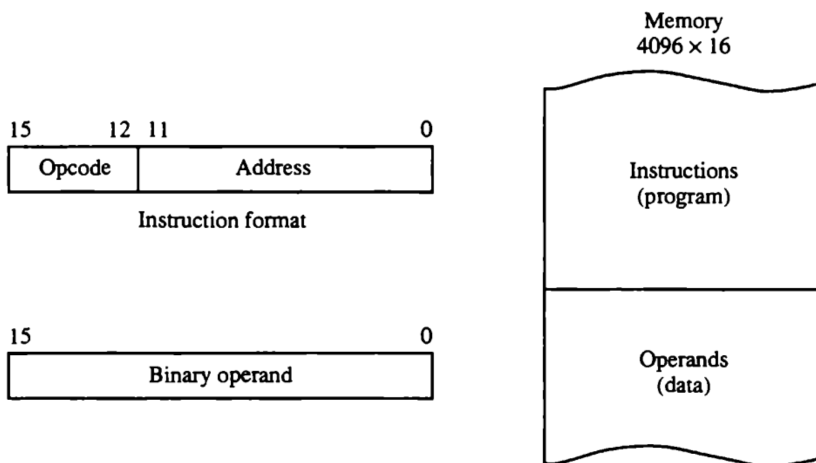


Fig: 5.1 Stored Program Organization

The simplest way to organize a computer is

»One processor register : AC (Accumulator)

- The operation is performed with the memory operand and the content of AC

» Instruction code format with two parts : Op. Code + address

- Op. Code : specify 16 possible operations *(4 it)*

- Address : specify the address of an operand *(12 bit)*

- If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction(*address field*) can be used for other purpose

» Memory : 12 bit = 4096 word(Instruction and Data are stored)

- Store each instruction code(*program*) and operand (*data*) in 16-bit memory word

Addressing Modes

- Immediate operand :

  » the second part of an instruction code(*address field*) specifies an *operand*

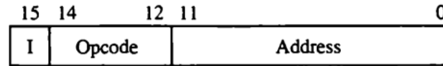
- Direct address operand :

  » the second part of an instruction code specifies the *address of an operand*

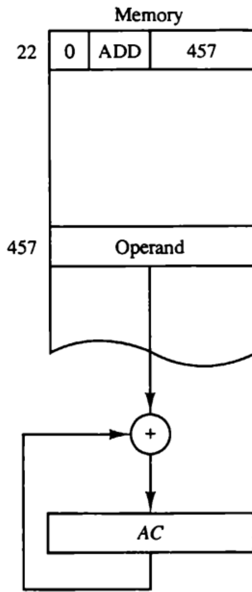
- Indirect address operand :

  » the bits in the second part of the instruction designate an *address of a memory word in which the address of the operand is found*

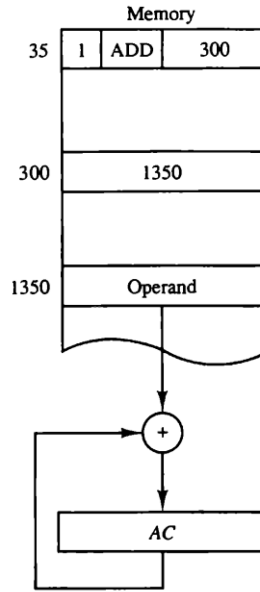
- One bit of the instruction code is used to distinguish between A direct and an indirect



(a) Instruction format



(b) Direct address



(c) Indirect address

Fig: 5.2 Demonstration of direct and indirect address

Effective Address:

- The operand address in computation-type instruction or the target address in a branch-type instruction

5.2 Computer Registers

In a computer, a register is one of a small set of data holding places that are part of a computer processor . A register may hold a computer instruction , a storage address, or any kind of data (such as a bit sequence or individual characters). Some instructions specify registers as part of the instruction. For

example, an instruction may specify that the contents of two defined registers be added together and then placed in a specified register. A register must be large enough to hold an instruction - for example, in a 32-bit instruction computer; a register must be 32 bits in length. In some computer designs, there are smaller registers - for example, half-registers - for shorter instructions. Depending on the processor design and language rules, registers may be numbered or have arbitrary names.

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Table: 5.1 List of registers for basic computer

Since memory is 4K in size, it requires 12 address bits. Each word of memory contains 16 bits of data. The address register (AR) is 12 bits wide, since this system requires that many bits in order to access memory. Similarly, the program counter (PC) is also 12 bits wide. Each data word is 16 bits wide. The Data Register (DR) must also be 16 bits wide, since it receives data from and sends data to memory. The accumulator (AC) acts on 16 bits of data. The Instruction Register (IR) receives instruction codes from memory which are 16 bits wide. Of note: TR is a temporary register. Only the CPU can cause it to be accessed. The programmer cannot directly manipulate the contents of TR. Most CPU's have one or more temporary registers which it uses to perform instructions.

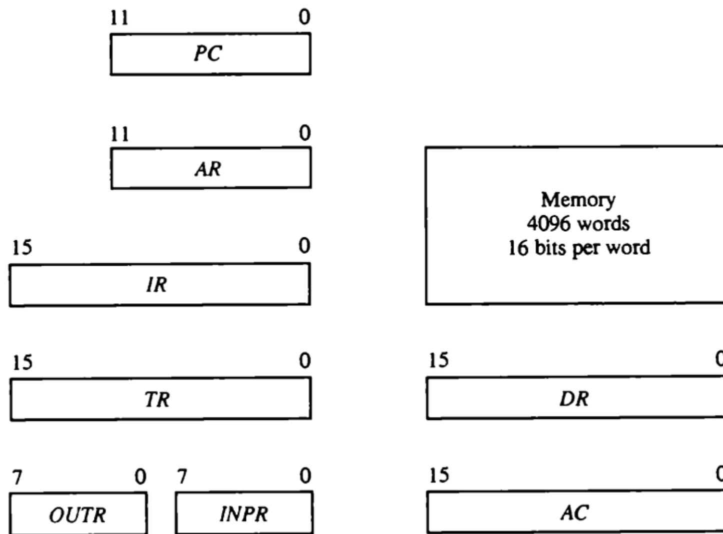


Fig: 5.3 Basic computer registers and memory

The input and output registers (INPR and OUTR) are 8 bits wide each. For this CPU, I/O instructions only transfer 8 bits of data at a time. The 3-bit sequence counter (SC) is used to generate the correct timing (T) states. Other 1-bit registers are the carry out (E), the indirect register (I), the interrupt enable (IEN) and the input and output flags (FGI and FGO).

Basic computer registers and memory :

- Data Register(**DR**) : hold the operand(Data) read from memory
- Accumulator Register(**AC**) : general purpose processing register

- Instruction Register(**IR**) : hold the instruction read from memory

- Temporary Register(**TR**) : hold a temporary data during processing
- Address Register(**AR**) : hold a memory address, 12 bit width
- Program Counter(**PC**) :

  »hold the address of the next instruction to be read from memory after the current instruction is executed
  »Instruction words are read and executed in sequence unless a branch instruction is encountered
  »A branch instruction calls for a transfer to a nonconsecutive instruction in the program
  »The address part of a branch instruction is transferred to PC tobecome the address of the next instruction
  »To read instruction, memory read cycle is initiated, and PC is incremented by one(next instruction fetch)

- Input Register(*INPR*) : receive an 8-bit character from an input device
- Output Register(*OUTR*) : hold an 8-bit character for an output device

Common Bus System

A wire or a collection of wires that carry some multi-bit information is known as bus. Main purpose of bus is to transfer information form one system to another. The basic computer has eight registers (AC, PC, DR, AC, IR, TR, INPR, OUTR), a memory unit and a control unit. Path must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the output of each register and input of other registers. A

more efficient scheme is to use a common bus. Thus common bus provides a path between memory unit and registers.

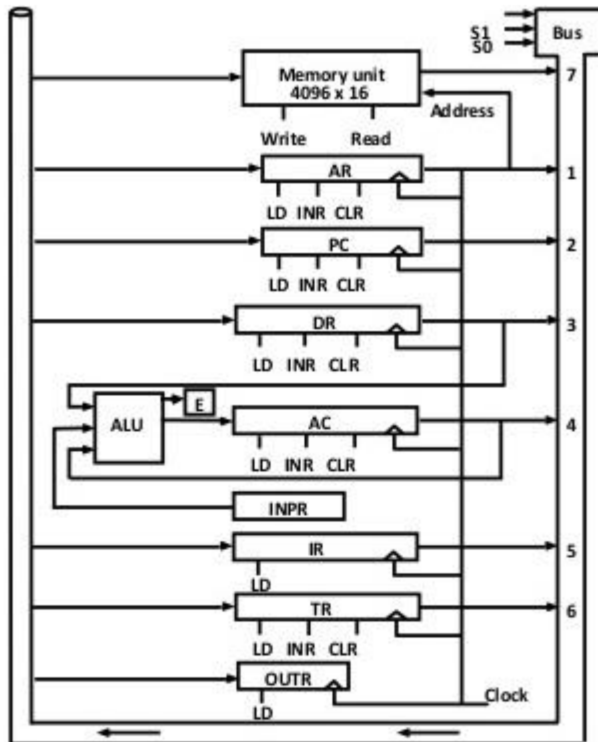


Fig: 5.4 Basic computer registers connected to a common bus

This is the internal design of the CPU for the Basic Computer. The CPU is designed around an internal common bus with a common clock. Each register can place its data onto the bus, and has internal tri-state buffers on the outputs. The control unit must make sure that at most one register (or memory unit) places

data onto the bus at one time.

The memory unit is external to the CPU. It always receives its address from the address register (AR) and makes its data

available to the CPU bus. It receives data from the CPU bus as well. Read and write signals are supplied by the control unit.

The address registers, program counter (PC) and data register (DR) each load data onto and receive data from the system bus. Each has a load, increment and clear signal derived from the control unit. These signals are sychronous; each register combines these signals with the system clock to activate the proper function. Since AR and PC are only 12-bits each, they use the low order 12 bits of the bus.

The accumulator makes its data available on the bus but does not receive data from the bus. Instead, it receives data solely from an ALU, labeled “Adder and Logic” in the diagram. To load data into AC, place it onto the bus via DR and pass it directly through the ALU. The synchronous load, increment and clear signals act as in the previous registers. Note that E, the 1-bit carry flag, also receives its data from the ALU.

The input register, INPR, receives data from an external input port, not shown here, and makes its data available only to AC. The output register makes its data available to the output port via hardware not shown here. We will examine these two components in more detail later in this module.

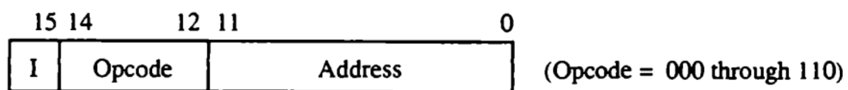
The instruction register, IR, can only be loaded; it cannot be incremented nor cleared. Its output is used to generate Di and Ti. We will look at that hardware later in this module.

TR is a temporary register. The CPU uses this register to store intermediate results of operations. It is not accessible by the external programs. It is loaded, incremented and cleared like the other registers.

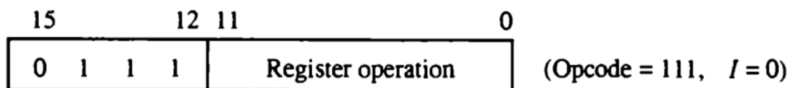
5.3 Computer Instructions

The basic computer has three instruction formats. Each format has 16 bits. The operation code part of the instruction contains

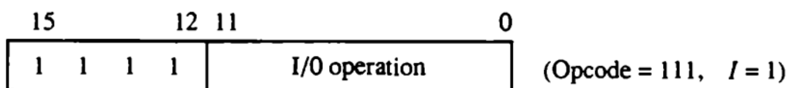
three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address. The register-reference instructions are recognized by the operation code 111and with a 0 in the leftmost bit of the instruction. A register-reference instruction specifies an operation on or a test of the AC register. An operand from memory is not needed; therefore the other 12 bits are used to specify the operation or test to be executed. Similarly, an input-output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the left most bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.



(a) Memory – reference instruction



(b) Register – reference instruction



(c) Input – output instruction

Fig: 5.5 Basic computer instruction formats

Symbol	Hexadecimal code $I = 0$	$I = 1$	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear *AC*
CLE	7400		Clear *E*
CMA	7200		Complement *AC*
CME	7100		Complement *E*
CIR	7080		Circulate right *AC* and *E*
CIL	7040		Circulate left *AC* and *E*
INC	7020		Increment *AC*
SPA	7010		Skip next instruction if *AC* positive
SNA	7008		Skip next instruction if *AC* negative
SZA	7004		Skip next instruction if *AC* zero
SZE	7002		Skip next instruction if *E* is 0
HLT	7001		Halt computer
INP	F800		Input character to *AC*
OUT	F400		Output character from *AC*
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Table: 5.2 Basic computer instructions

Instruction Set Completeness

Before investigating te operations performed by the instructions, let us discuss the type of instructions that must be included in a computer. A computer should have a set of instructions so that the

user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical and shift instructions
2. Instructions for moving information to and from memory and processor registers.
3. Program control instructions together with instructions that check status conditions.
4. Input and output instructions

5.4 Timing and Control

All sequential circuits in the Basic Computer CPU are driven by a master clock, with the exception of the INPR register.

At each clock pulse, the control unit sends control signals to control inputs of the bus, the registers, and the ALU.

The control unit (CU) is a component of a computer's central processing unit (CPU) that directs operation of the processor. It tells the computer's memory, arithmetic/logic unit and input and output devices how to respond to a program's instructions.

It directs the operation of the other units by providing timing and control signals Most computer resources are managed by the CU, It directs the flow of data between the CPU and the other devices. John von Neumann included the control unit as part of the von Neumann architecture. In modern computer designs, the control unit is typically an internal part of the CPU with its overall role and operation unchanged since its introduction.

Control unit design and implementation can be done by two general methods:

Hardwired control units are implemented through use of sequential logic units, featuring a finite number of gates that can generate specific results based on the instructions that were used to invoke those responses Hardwired control units are generally faster than microprogrammed designs.

Their design uses a fixed architecture—it requires changes in the wiring if the instruction set is modified or changed. This architecture is preferred in reduced instruction set computers (RISC) as they use a simpler instruction set.

A controller that uses this approach can operate at high speed; however, it has little flexibility, and the complexity of the instruction set it can implement is limited.

The hardwired approach has become less popular as computers have evolved. Previously, control units for CPUs used ad-hoc logic, and they were difficult to design.

The idea of microprogramming is an intermediate level to execute computer program instructions. Microprograms were organized as a sequence of microinstructions and stored in special control memory. The algorithm for the microprogram control unit is usually specified by flowchart description.[4] The main advantage of the microprogram control unit is the simplicity of its structure. Outputs of the controller are organized in microinstructions and they can be easily replaced.

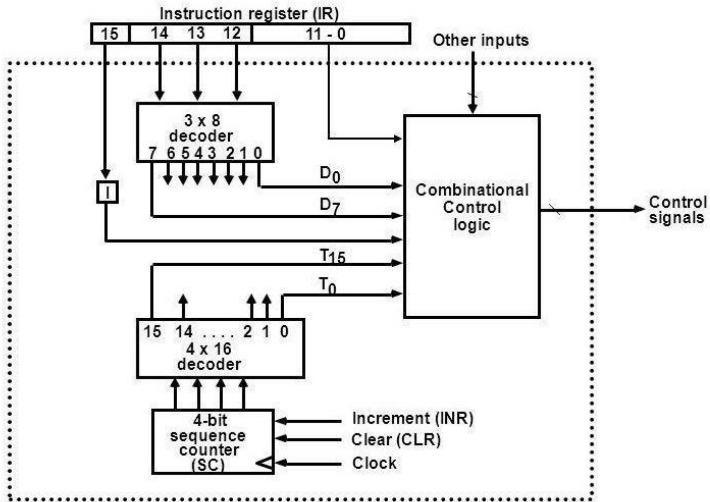


Fig: 5.6 Control unit of a basic computer

The T signals occur in sequence and are never skipped over. The only two options during a T-state are to proceed to the next T-state or to return to Tstate 0. The D signals decode the instruction and are used to select the correct execute routine. I is used to select the indirect routine and also to select the correct execute routine for non-memory reference instructions. R is used for interrupt processing and will be explained later.

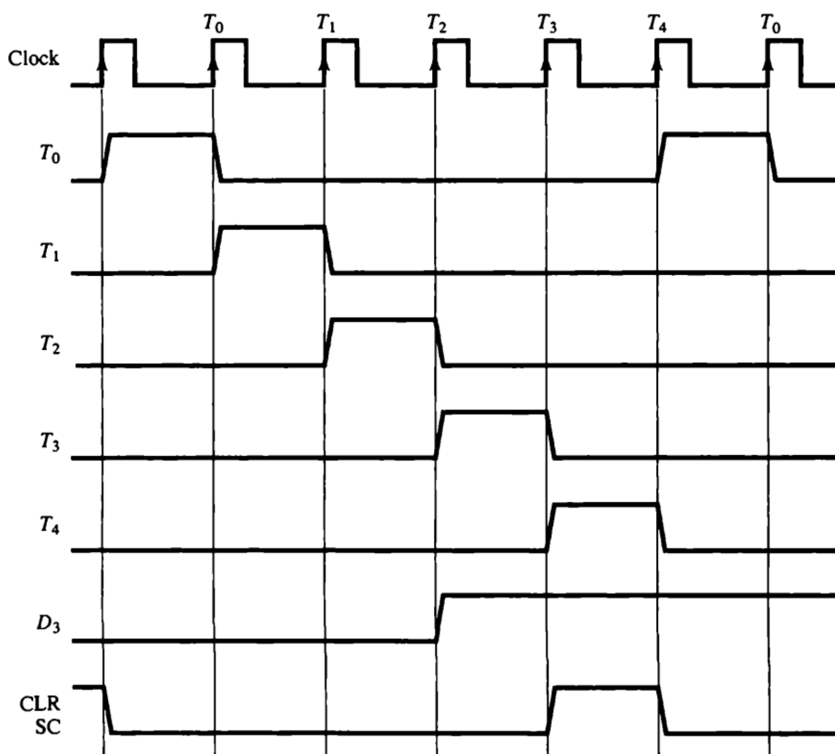


Fig: 5.7 Example of control timing signals

This circuit generates the T signals. The sequence counter, SC, is incremented once per clock cycle. Its outputs are fed into a 3-8 decoder which generates the T signals. Whenever a microoperation sets SC to zero, it resets the counter, causing T0 to be activated during the next clock cycle. The D signals are generated similarly to the T signals. Here the source is IR(14-12) instead of SC. Also note that IR won't change during the instruction execution.

5.5 Instruction Cycle

The time period during which one instruction is fetched from memory and executed when a computer is given an instruction in machine language. There are typically four stages of an instruction cycle that the CPU carries out:

- Fetch the instruction from memory. This step brings the instruction into the *instruction register*, a circuit that holds the instruction so that it can be decoded and executed.
- Decode the instruction.
- Read the effective address from memory if the instruction has an indirect address.
- Execute the instruction.

Steps 1 and 2 are called the *fetch cycle* and are the same for each instruction. Steps 3 and 4 are called the *execute cycle* and will change with each instruction.

Each computer's CPU can have different cycles based on different instruction sets, but will be similar to the following cycle:

1. **Fetching the instruction**: The next instruction is fetched from the memory address that is currently stored in the program counter (PC), and stored in the instruction register (IR). At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.

2. **Decode the instruction**: During this cycle the encoded instruction present in the IR (instruction register) is interpreted by the decoder.

3. **Read the effective address**: In case of a memory instruction (direct or indirect) the execution phase will be in the next clock pulse. If the instruction has an indirect address, the effective address is read from main memory, and any required data is fetched from main memory to be

processed and then placed into data registers (Clock Pulse: T_3). If the instruction is direct, nothing is done at this clock pulse. If this is an I/O instruction or a Register instruction, the operation is performed (executed) at clock Pulse.

4. **Execute the instruction**: The control unit of the CPU passes the decoded information as a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction such as reading values from registers, passing them to the ALU to perform mathematical or logic functions on them, and writing the result back to a register. If the ALU is involved, it sends a condition signal back to the CU. The result generated by the operation is stored in the main memory, or sent to an output device. Based on the condition of any feedback from the ALU, Program Counter may be updated to a different address from which the next instruction will be fetched.

The cycle is then repeated.

Initiating the cycle

The cycle starts immediately when power is applied to the system using an initial PC value that is predefined for the system architecture (in Intel IA-32 CPUs, for instance, the predefined PC value is 0xfffffff0). Typically this address points to instructions in a read-only memory (ROM) (not the random access memory or RAM) which begins the process of loading the operating system. (That loading process is called *booting.*) [1]

Fetch the Instruction

Step 1 of the Instruction Cycle is called the Fetch Cycle. This step is the same for each instruction.

1) The CPU sends PC to the MAR and sends a READ command on the control bus

2) In response to the read command (with address equal to PC), the memory returns the data stored at the memory location indicated by PC on the databus.

3) The CPU copies the data from the databus into its MDR (also known as MBR (see section Circuits Used above))...

4) A fraction of a second later, the CPU copies the data from the MDR to the Instruction Register (IR)

5) The PC is incremented so that it points to the following instruction in memory. This step prepares the CPU for the next cycle.

The Control Unit fetches the instruction's address from the Memory Unit

Decode the Instruction

Step 2 of the instruction Cycle is called the Decode Cycle. The decoding process allows the CPU to determine what instruction is to be performed, so that the CPU can tell how many operands it needs to fetch in order to perform the instruction. The opcode fetched from the memory is decoded for the next steps and moved to the appropriate registers. The decoding is done by the CPU's Control Unit.

Read the effective address

Step 3 is deciding which operation it is. If this is a Memory operation - in this step the computer checks if it's a direct or indirect memory operation:

- Direct memory instruction - Nothing is being done.

- Indirect memory instruction - The effective address is being read from the memory.

If this is a I/O or Register instruction - the computer checks its kind and executes the instruction.

Execute the Instruction

Step 4 of the Instruction Cycle is the Execute Cycle. Here, the function of the instruction is performed. If the instruction involves arithmetic or logic, the Arithmetic Logic Unit is utilized. This is the only stage of the instruction cycle that is useful from the perspective of the end user. Everything else is overhead required to make the execute stage happen.

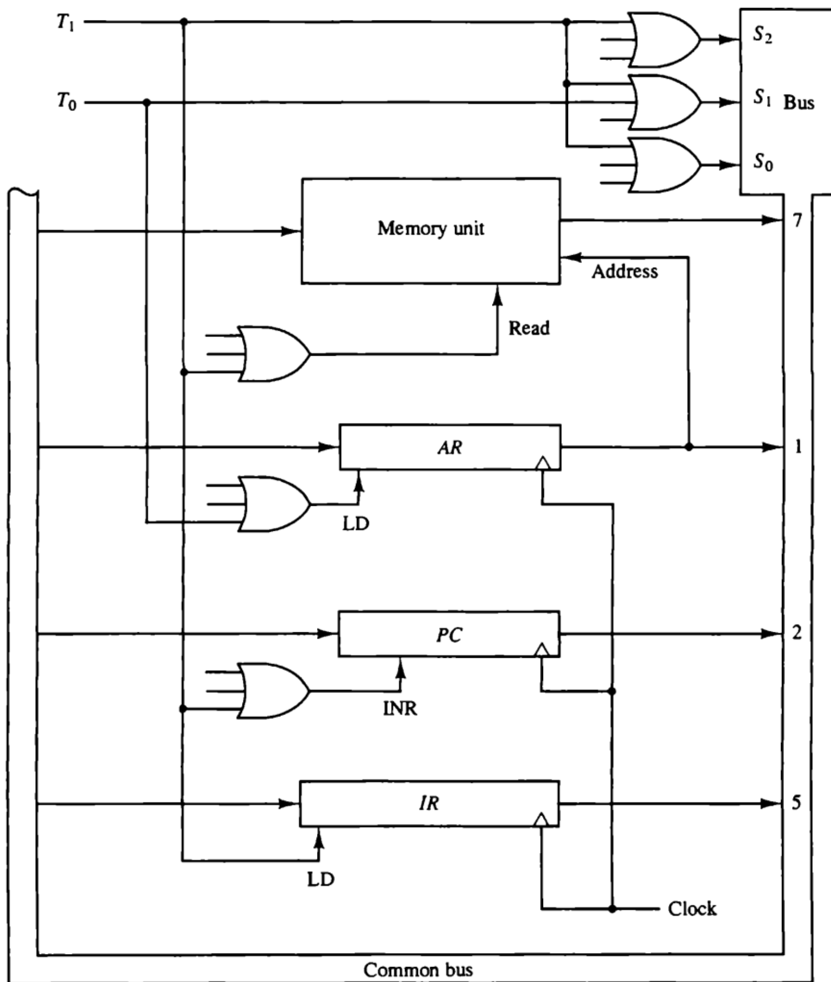


Figure 5.8 Register transfers for the fetch phase

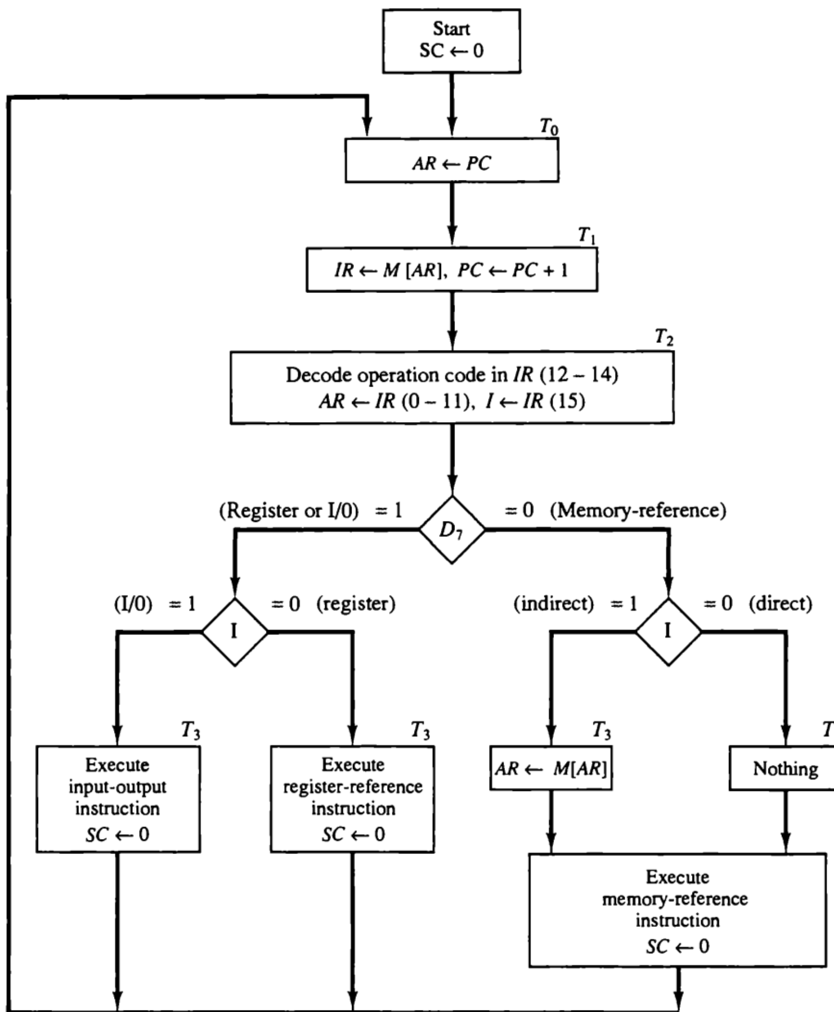


Fig:5.9 Flow chart for instruction cycle

Register Reference Instructions

There are 12 instructions which fall in the category of register reference instructions. These instructions are specified by the

instruction code from 0 to 11 bits. They are also transferred to AR at time T. the control operations and micro operations for the register reference instruction are listed below. These instructions are executed with the clock transition in accordance with the timing variable. The control functions and micro operations are distinguished from one another by just one bit. The seven instructions of the reference register are used for the CLEAR, COMPLEMENT, CIRCULAR SHIFT, and INCREMENT micro operations. Four instructions are for SKIP of next instruction in sequence when some stated condition is satisfied. It can be achieved by incrementing the PC i.e. the program counter.

Explanation of the instructions:

CLEAR- It is used for resetting the register. The value of register after CLEAR is ‘0’.

COMPLEMENT- It is making the complement of the given data. By complement we mean to say 2’s complement.

CIRCULAR SHIFT- circular shift can be defined as the shifting of the bits of data in circular fashion.

INCREMENT- it is for incrementing the value by 1.

SKIP- it is for skipping the instruction.

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

Table: 5.3 Execution of register reference instructions

5.6 Memory Reference Instructions

For the instructions to be carried out in a sequential manner we need the proper definition of the micro operations to be executed under it. So we need a precise defined form of them. As we know that instructions are read from the memory into the registers so the term memory reference instructions came into the picture.
We have around seven memory reference instruction.
Listed below are:

1. AND to AC

This instruction as the name suggests performs the function of ANDing on the bits of AC (accumulator) and the memory word. The result is stored in accumulator (AC). The control function for this instruction uses the decoder. Two timing signals are also needed for this instruction to be carried out. Clock transition is also associated with the next timing signal. The same clock

transition clears the SC and transfers the control to timing signal to start new instruction cycle.

$$D_0T_4: \quad DR \leftarrow M[AR]$$
$$D_0T_5: \quad AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0$$

2. ADD to AC

This instruction adds the content of the memory word to AC which is specified by the effective address. The result i.e. the SUM is transferred to AC and the output is transferred to E flip flop. Here also the same two timing signals are used again but with different decoders. After the instruction is fetched from the memory and decoded, only the single output of the decoder will be active and rest all will be deactivated. And this output will determine the sequence of micro operations that control has followed.

$$D_1T_4: \quad DR \leftarrow M[AR]$$
$$D_1T_5: \quad AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$$

3. LDA: Load to AC

Load means to transfers. This instruction transfers the memory word specified by the effective address to the accumulator or in other words we can say it load the memory word into the accumulator. It is first necessary to read the memory word into another register named DR and then transfer the content of the same into the AC (accumulator). The reason for this is that the delay which is encountered whiles the adding and logic operations.

$$D_2T_4: \quad DR \leftarrow M[AR]$$
$$D_2T_5: \quad AC \leftarrow DR, \quad SC \leftarrow 0$$

4. STA: Store AC

This instruction is just the opposite in functioning of the LOAD. Here the content is stored from the AC into the memory word specified by the effective address. Now the output of the AC is directly connected to the bus, so we can expect only one micro operation for this entire instruction.

$$D_3T_4: \quad M[AR] \leftarrow AC, \quad SC \leftarrow 0$$

5. BUN: Branch Unconditionally

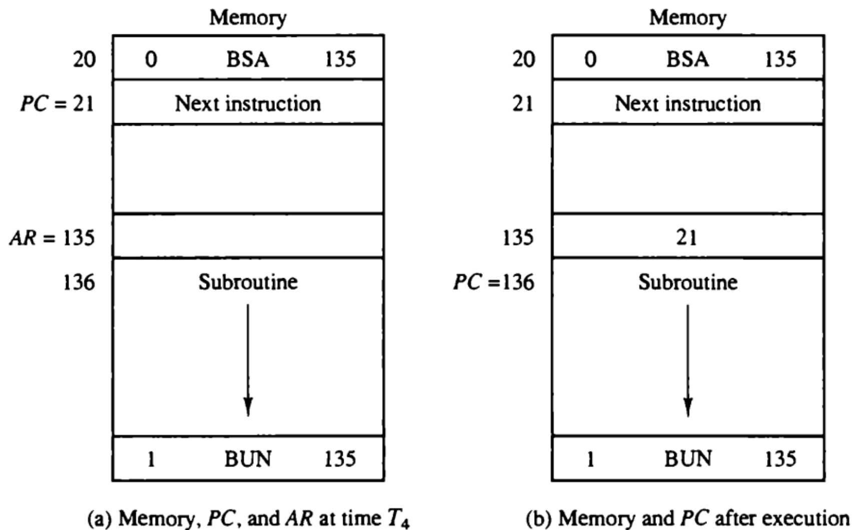
This instruction has the responsibility to transfer the entire program to the instruction which is specified by the effective address. We now know that program counter (PC) holds the address of the instruction to be read from the memory and program is a set of instructions to be carried out to accomplish the particular task. BUN instruction allows the programmer to specify an instruction out of the program and modify the program.

$$D_4T_4: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

6. BSA: Branch and Save Return Address

As the name tells the function of this **instruction**, it allows the branching in the execution of instruction. By branching we mean

that the instructions can have sub routine or procedure. When this instruction is executed, it stores the address of the next instruction to be executed as PC (Program counter).



(a) Memory, *PC*, and *AR* at time T_4 (b) Memory and *PC* after execution

Fig: 5.10 Example of BSA instruction execution

$$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$$
$$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

7. ISZ: Increment and Skip if Zero

This is a increment instruction which increments the word specified by the effective address, and if by any chance it founds that its value is zero then the value of PC is incremented by 1. It is in general practice to store a negative number in the memory word.

D_6T_4: $DR \leftarrow M[AR]$
D_6T_5: $DR \leftarrow DR + 1$
D_6T_6: $M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$

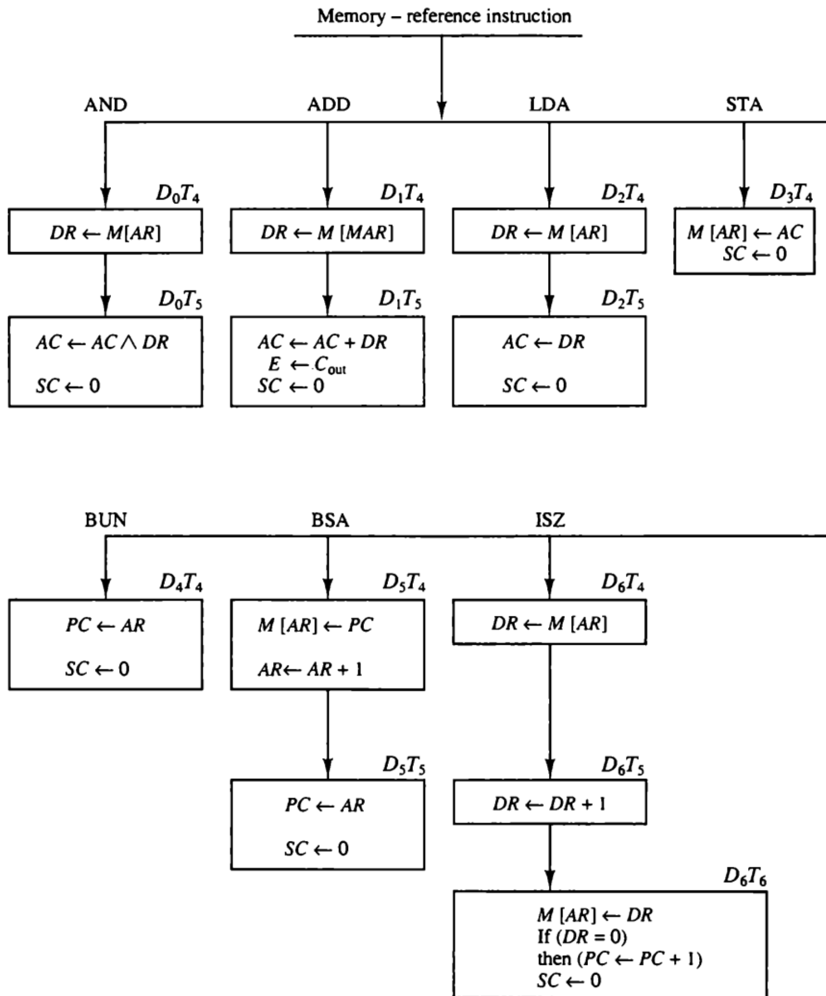


Fig 5.9 Flow chart for memory reference instructions

5.7 Input-Output and Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input output devises

Input Output Configration

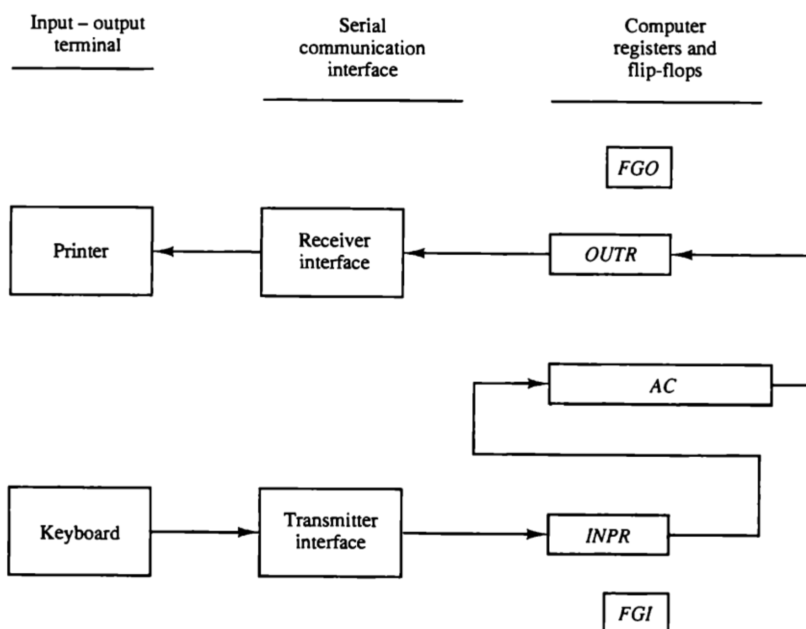


Fig 5.12 Input Output Configuration

Input Register(*INPR*) and Output Register(*OUTR*) are two registers communicate with a communication interface serially and with the AC in parallel. The Basic Computer has one 8-bit input port and one 8-bit output port. Each port interface is modeled as an 8-bit register which can send data to or receive data from

AC(7-0). Whenever input data is to be made available, the external input port writes the data to INPR and sets FGI to 1. When the output port requests data, it sets FGO to 1.

As will be shown shortly, the FGI and FGO flags are used to trigger interrupts (if interrupts are enabled by the IEN flag).

Input Output Instructions

Input and Output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-Output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and $I = 1$.

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Table: 5.5 Input Output Instructions

Once data is made available to the CPU, it can be read in using the INP instruction. Note that this not only reads the data into the accumulator, but also resets FGI to zero. This tells the input port that it may send more data. In a similar manner, the OUT instruction writes data to OUTR and resets FGO to zero, notifying the output port that data is available. The SKI and SKO instructions skip an instruction if there is a pending input or output request. This is useful in determining the I/O request which caused an interrupt to occur. ION and IOF enable and disable interrupts. Interrupts will be explained more fully shortly.

In the Basic Computer, I/O requests are processed as interrupts. This process is followed for input requests. The input will only be processed if interrupts are enabled. It will be ignored, but will remain pending, if interrupts are disabled.

Outputs are handled similarly to inputs. Note that both input and output interrupts call an interrupt service routine at location 0. There is only one routine for both input and output, so it must distinguish between the two. This is where the SKI and SKO instructions become useful.

Program Interrupt

An interrupt occurs if the interrupt is enabled (IEN = 1) AND an interrupt is pending (FGI or FGO = 1). u Before processing the interrupt, complete the current instruction. u Call the interrupt service routine at address 0 and disable interrupts. It is of the utmost importance to complete the current instruction, otherwise the CPU will not perform properly. The interrupt service routine is called by the CPU in a manner similar to the execution of the BSA instruction.

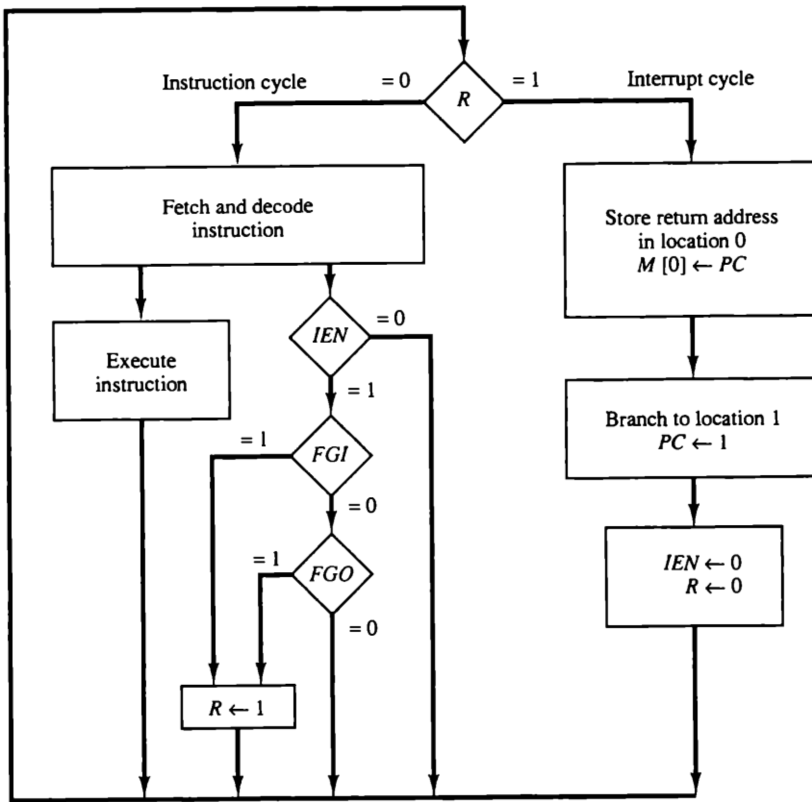


Fig: 5.13 Flowchart for Interrupt cycle

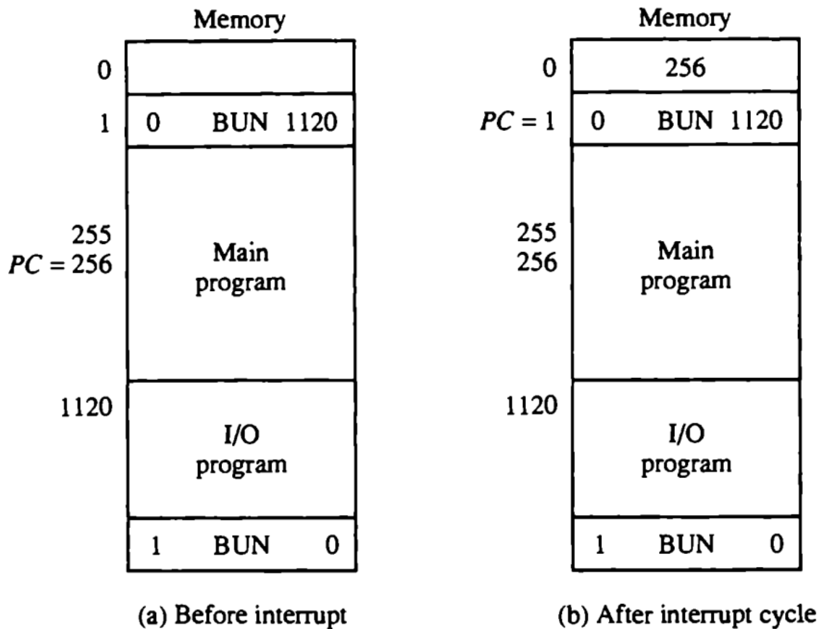


Fig: 5.14 Demonstration of interrupt cycle

Interrupt Cycle

An interrupt is asserted by setting R to 1. This occurs when interrupts are enabled (IEN) and there is either an input or output request (FGI+FGO). We must also have completed the current fetch cycle (T0'T1'T2'). When we look at the code to implement the interrupt cycle, we see why we must wait until after T2 to set R to 1. If we set R to 1 during T0, for example, the next micro-instruction would be RT1, right in the middle of the interrupt cycle. Since we want to either perform an entire opcode fetch or an entire interrupt cycle, we don't set R until after T2. The interrupt cycle acts like a BSA 0 instruction. During T0 we write a 0 into AR and copy the contents of PC, the return address, to TR. We then store the return address to location 0 and clear the program counter during T1. In T2, we increment PC to 1, clear the interrupt enable, set R to zero (because we've finished the interrupt cycle) and clear SC to bring us back to T0. Note that IEN is set to 0.

Activating an interrupt request:

T0’ T1’ T2’(IEN)(FGI + FGO): R <- 1

Interrupt cycle:

RT0: AR <- 0, TR <- PC
RT1: M[AR] <- TR, PC <- 0
RT2: PC <- PC + 1, IEN <- 0, R <- 0, SC <- 0

This disables further interrupts. If another interrupt occurred while one was being serviced, the second interrupt would write its return address into location 0, overwriting the interrupt return address of the original routine. Then it would not be possible to return to the program properly.

Modified fetch phase

R’T0: AR <- PC
R’T1: IR <- M[AR], PC <- PC+1
R’T2: AR <- IR(11-0), I <- IR15,D0, D1, ... D7 <- Decode IR(14-12)