RCS-353: COMPUTER ORGANIZATION LAB

EXPERIMENTS:-

1. Implement different logic gates and verify their truth table
2. Implementing HALF ADDER, FULL ADDER using basic logic gates
3. Implementing Binary -to -Gray, Gray -to -Binary code conversions.
4. Implementing 3-8 line DECODER and Implementing 4x1 and 8x1 MULTIPLEXERS.
5. Verify the excitation tables of various FLIP-FLOPS.
6. Design of an 8-bit Input/ Output system with four 8-bit Internal Registers.
7. Design of an 8-bit ARITHMETIC LOGIC UNIT.
8. Design the data path of a computer from its register transfer language description. — shifter 116 M/M
9. Design the control unit of a computer using either hardwiring or microprogramming based on its register transfer language description.
10. Write an algorithm and program to perform matrix multiplication of two n * n matrices on the 2-D mesh SIMD model, Hypercube SIMD Model or multiprocessor system.

PROGRAM-1

Implement different logic gates and verify their truth table.

Logic Gates: A logic gate is an elementary building block of a digital circuit. Most **logic gates** have two inputs and one output. At any given moment, every terminal is in one of the two binary conditions low (0) or high (1), represented by different voltage levels.

Example:
AND
OR
NOT
NAND
NOR
EXOR, and EXNOR.

AND gate

The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B. Bear in mind that this dot is sometimes omitted i.e. AB

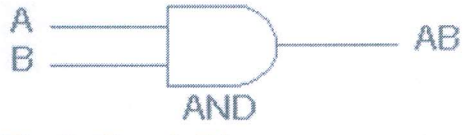


Fig-1: Circuit Diagram

2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

Fig2: Truth Table

OR gate

The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high. A plus (+) is used to show the OR operation.

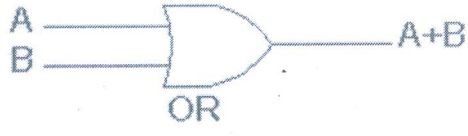


Fig-1: Circuit Diagram

2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Fig2: Truth Table

NOT gate

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an *inverter*. If the input variable is A, the inverted output

is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways that the NAND logic gate can be configured to produce a NOT gate. It can also be done using NOR logic gates in the same way.

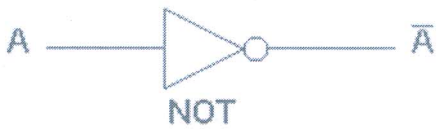
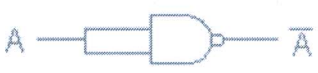


Fig-1: Circuit Diagram

NOT gate	
A	$\overline{A}$
0	1
1	0

Fig2: Truth Table



NAND gate

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if **any** of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

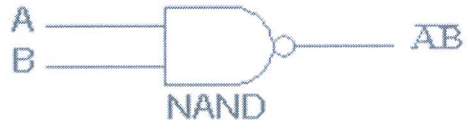


Fig-1: Circuit Diagram

2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

Fig2: Truth Table

NOR gate

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if **any** of the inputs are high. The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

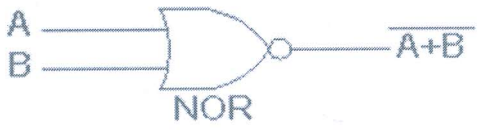


Fig-1: Circuit Diagram

2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

Fig2: Truth Table

EXOR gate

The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high. An encircled plus sign (⊕) is used to show the EOR operation.

Fig-1: Circuit Diagram

2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Fig2: Truth Table

EXNOR gate

The '**Exclusive-NOR**' gate circuit does the opposite to the EOR gate. It will give a low output if **either, but not both**, of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents inversion.

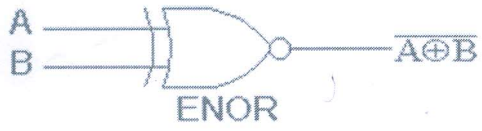


Fig-1: Circuit Diagram

2 Input EXNOR gate		
A	B	$\overline{A⊕B}$
0	0	1
0	1	0
1	0	0
1	1	1

Fig2: Truth Table

Table 1: Logic gate symbols

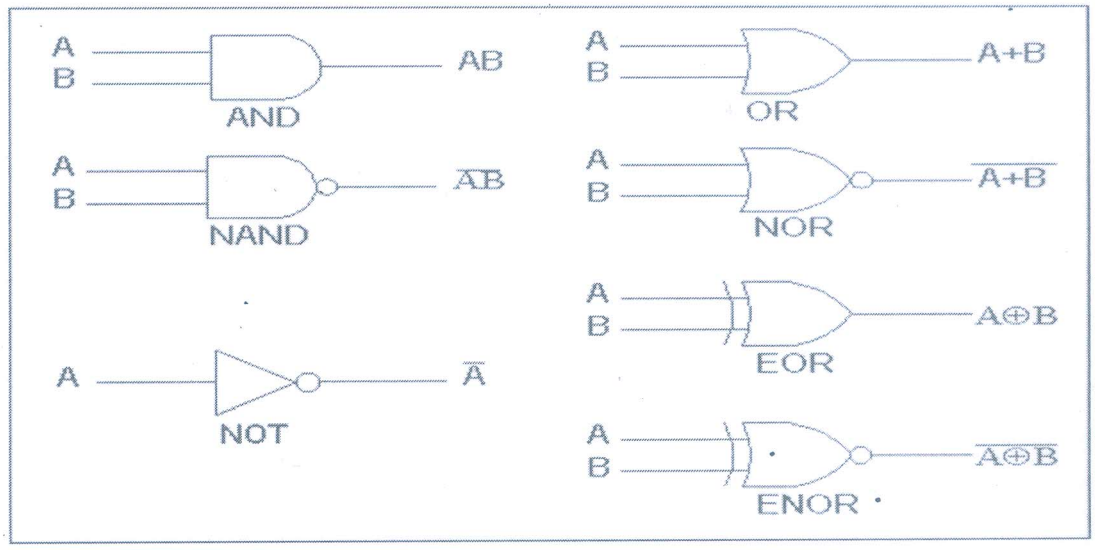


Table 2: Logic gates representation using the Truth table

NOT gate	
A	$\overline{A}$
0	1
1	0

INPUTS		OUTPUTS					
A	B	AND	NAND	OR	NOR	EXOR	EXNOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

Implementation of XNOR with the help of basic gates.

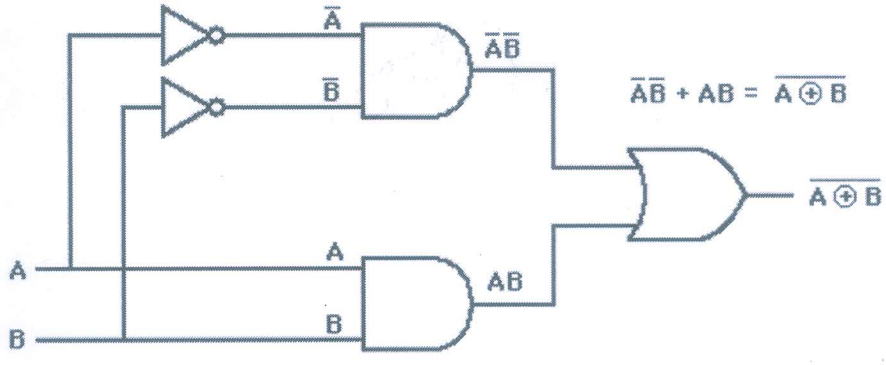


Fig: XNOR

Implementation of XOR with the help of basic gates.

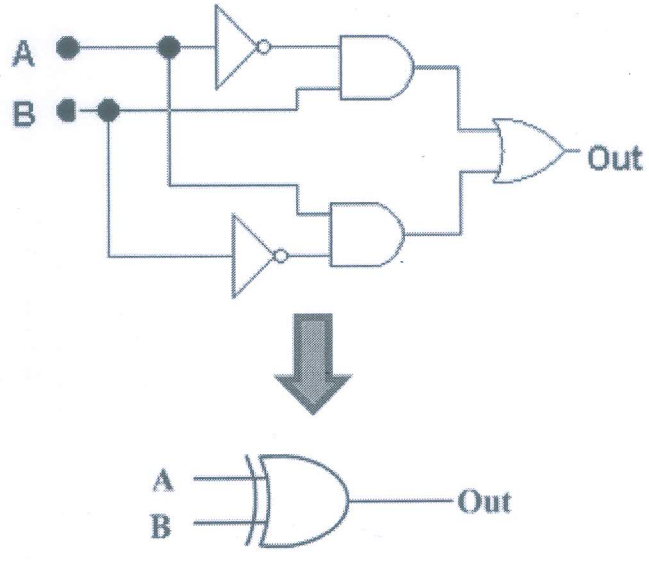
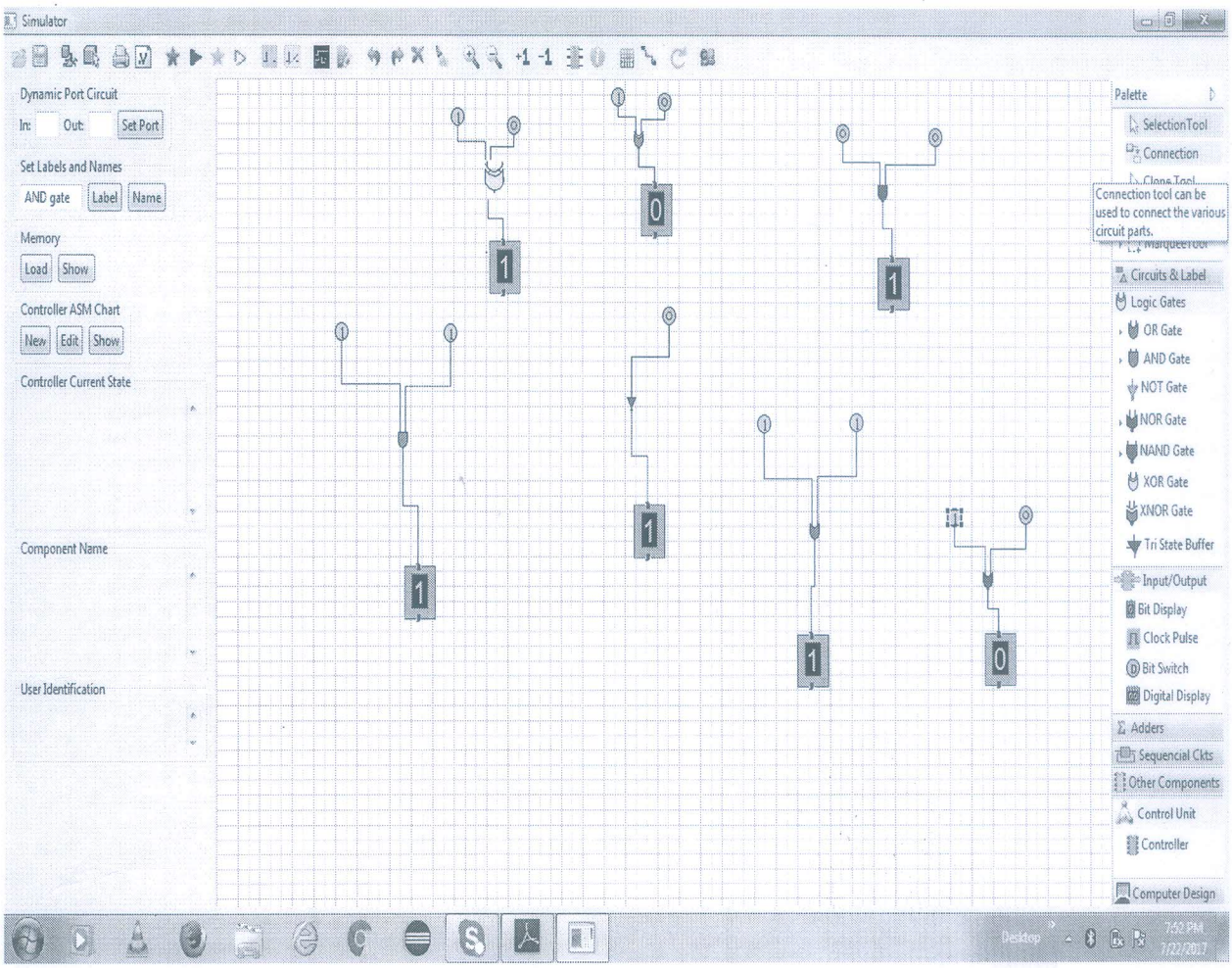


Fig: XOR

Implementation using simulator:



Program-2

Implementing Falf Adder and Full Adder using logic gates.

Half Adder:

With the help of half adder, we can design circuits that are capable of performing simple addition with the help of logic gates.

Let us first take a look at the addition of single bits.

0+0 = 0

0+1 = 1

1+0 = 1

1+1 = 10

These are the least possible single-bit combinations. But the result for 1+1 is 10. Though this problem can be solved with the help of an EXOR Gate, if you do care about the output, the sum result must be re-written as a 2-bit output.

Thus the above equations can be written as

0+0 = 00

0+1 = 01

1+0 = 01

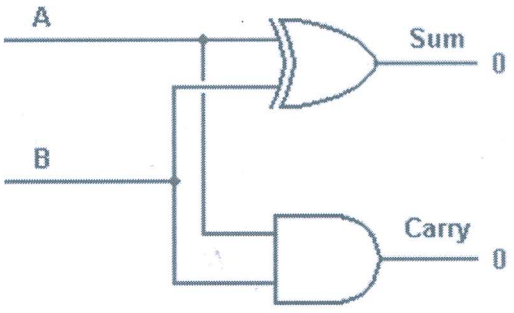
1+1 = 10

Truth Table:

INPUTS		OUTPUTS	
A	**B**	**SUM**	**CARRY**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

From the equation it is clear that this 1-bit adder can be easily implemented with the help of EXOR Gate for the output 'SUM' and an AND Gate for the carry. Take a look at the implementation below.

Circuit Diagram:

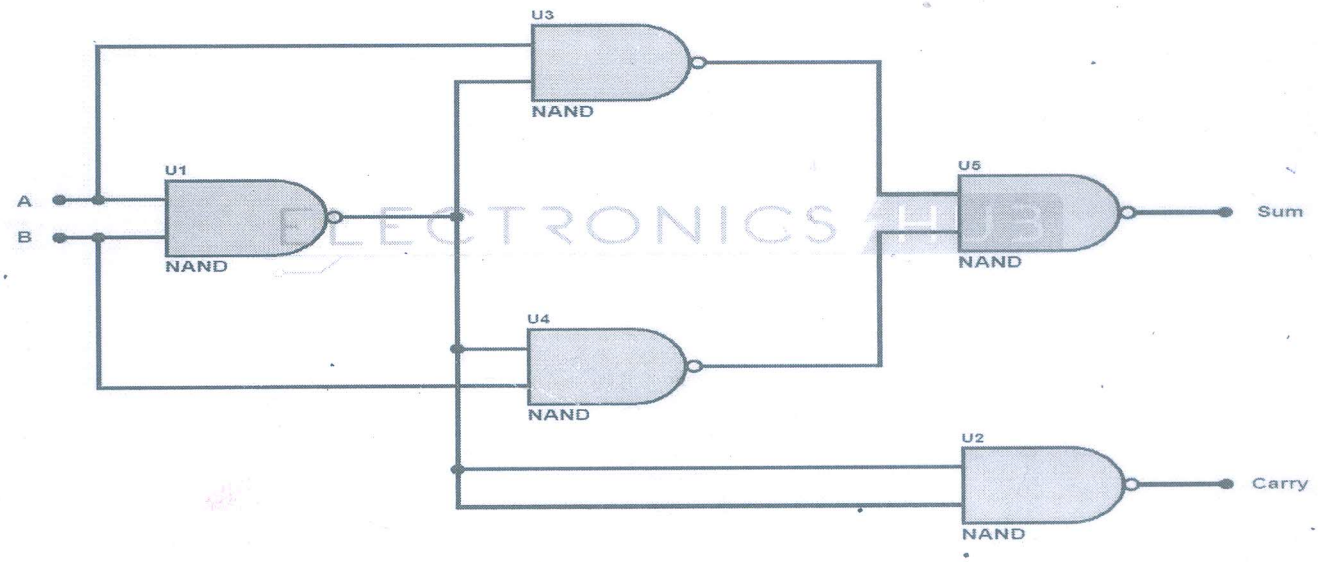


Half Adder Circuit

For complex addition, there may be cases when you have to add two 8-bit bytes together. This can be done only with the help of full-adder logic.

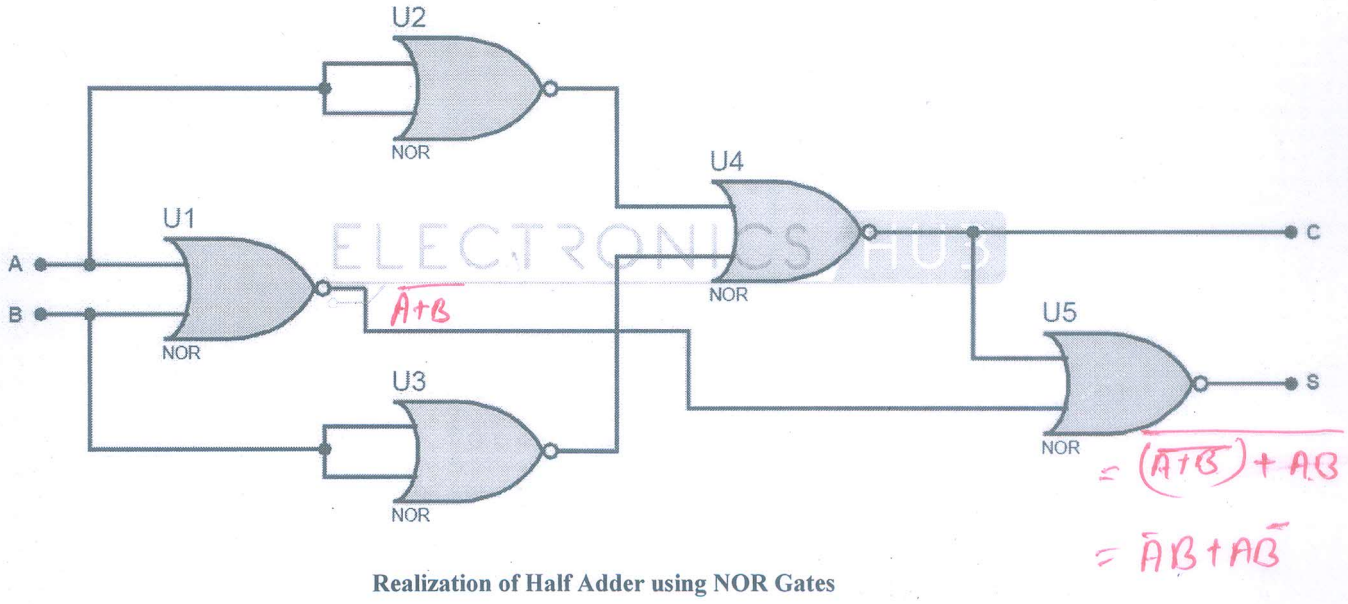
Half Adder using NAND Gates:

Five NAND gates are required in order to design a half adder. The circuit to realize half adder using NAND gates is shown below.



Half Adder using NOR Gates:

Five NOR gates are required in order to design a half adder. The circuit to realize half adder using NOR gates is shown below



Realization of Half Adder using NOR Gates

Limitations of Half Adder :

The reason these simple binary adders are called Half Adders is that there is no scope for them to add the carry bit from previous bit. This is a major limitation of half adders when used as binary adders especially in real time scenarios which involves addition of multiple bits. To overcome this limitation, full adders are developed.

Full Adder :

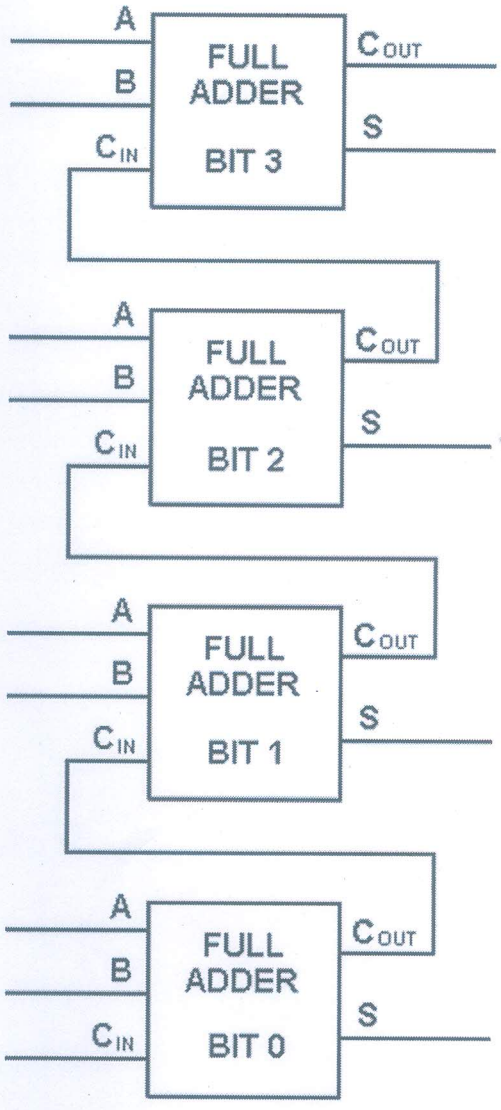
This type of adder is a little more difficult to implement than a half-adder. The main difference between a half-adder and a full-adder is that the full-adder has three inputs and two outputs. The first two inputs are A and B and the third input is an input carry designated as CIN. When a full adder logic is designed we will be able to string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next.

The output carry is designated as COUT and the normal output is designated as S. Take a look at the truth-table.

INPUTS		**OUTPUTS**		
A	**B**	**CIN**	**COUT**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

From the above truth-table, the full adder logic can be implemented. We can see that the output S is an EXOR between the input A and the half-adder SUM output with B and CIN inputs. We must also note that the COUT will only be true if any of the two inputs out of the three are HIGH.

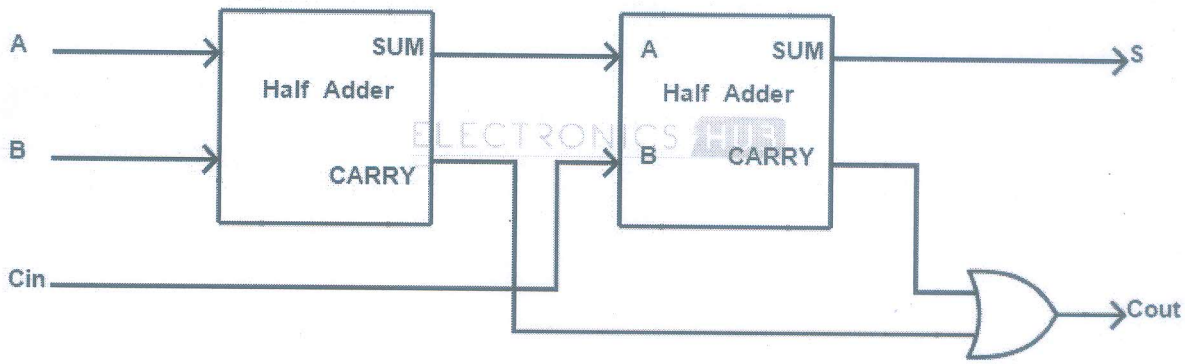
Thus, we can implement a full adder circuit with the help of two half adder circuits. The first will half adder will be used to add A and B to produce a partial Sum. The second half adder logic can be used to add CIN to the Sum produced by the first half adder to get the final S output. If any of the half adder logic produces a carry, there will be an output carry. Thus, COUT will be an OR function of the half-adder Carry outputs. Take a look at the implementation of the full adder circuit shown below

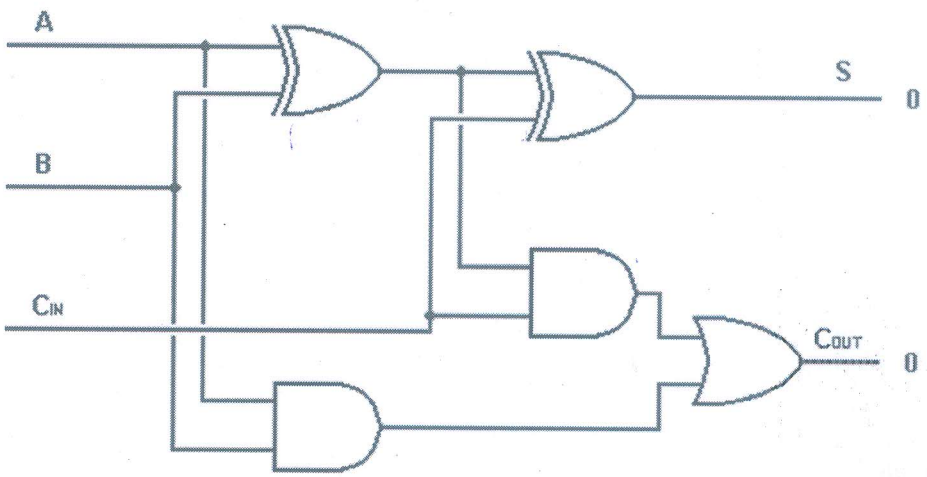


Multi-Bit Addition using Full Adder.

Implementation of Full Adder using Half Adders

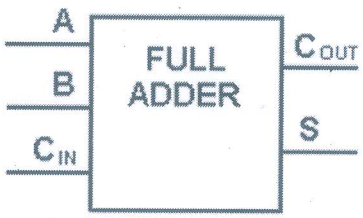
A full adder can be formed by logically connecting two half adders. The block diagram that shows the implementation of a full adder using two half adders is shown below.





Full Adder Circuit

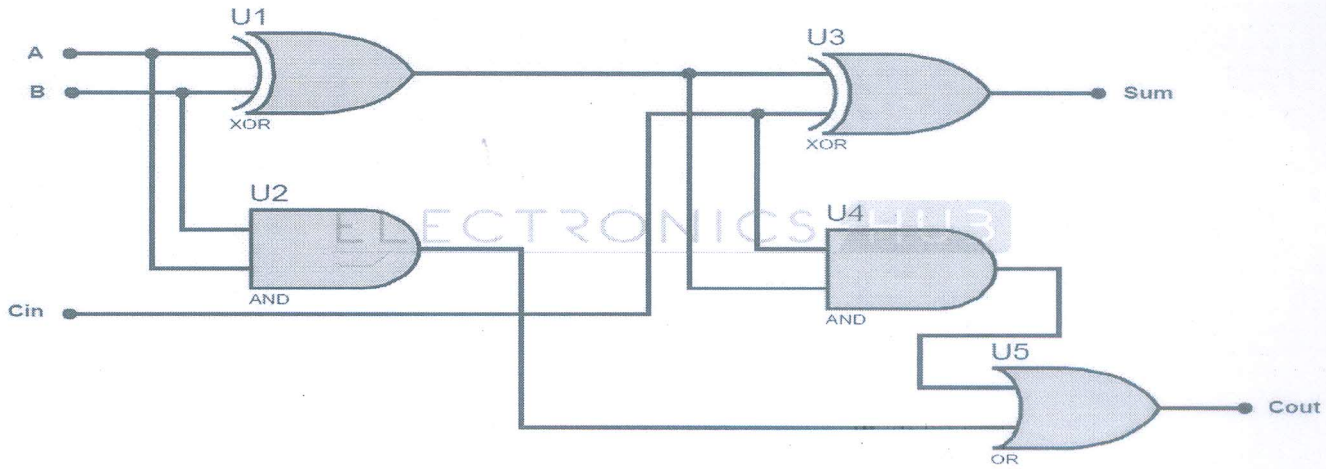
Though the implementation of larger logic diagrams is possible with the above full adder logic a simpler symbol is mostly used to represent the operation. Given below is a simpler schematic representation of a one-bit full adder.



Single-bit Full Adder

With this type of symbol, we can add two bits together taking a carry from the next lower order of magnitude, and sending a carry to the next higher order of magnitude. In a computer, for a multi-bit operation, each bit must be represented by a full adder and must be added simultaneously. Thus, to add two 8-bit numbers, you will need 8 full adders which can be formed by cascading two of the 4-bit blocks. The addition of two 4-bit numbers is shown below.

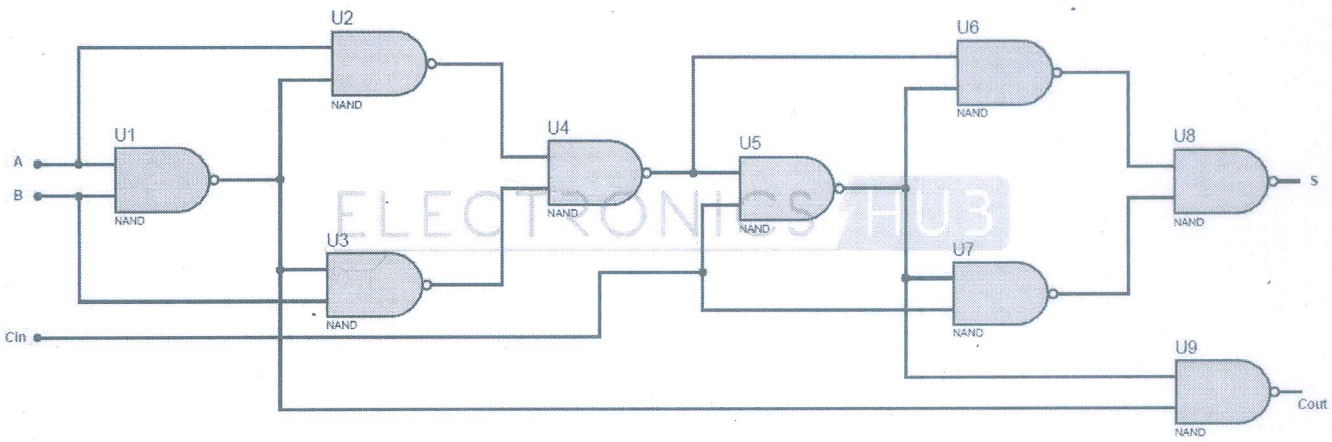
adders and an OR gate. The implementation of full adder using two half adders is show below.



Implementation of Full Adder with 2 Half Adders

Full Adder using NAND Gates:

As mentioned earlier, a NAND gate is one of the universal gates and can be used to implement any logic design. The circuit of full adder using only NAND gates is shown below.



Full Adder using NAND Gates

Full adder is a simple 1 – bit adder. If we want to perform n – bit addition, then n number of 1 – bit full adders should be used in the form of a cascade connection.

Program No-3

Conversion from Binary Code to Gray Code and Vice Versa

Gray Code : The **reflected binary code** (**RBC**), also known as **Gray code.** The reflected binary code was originally designed to prevent spurious output from electromechanical switches. Today, Gray codes are widely used to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems.

Binary Code to Gray Code:

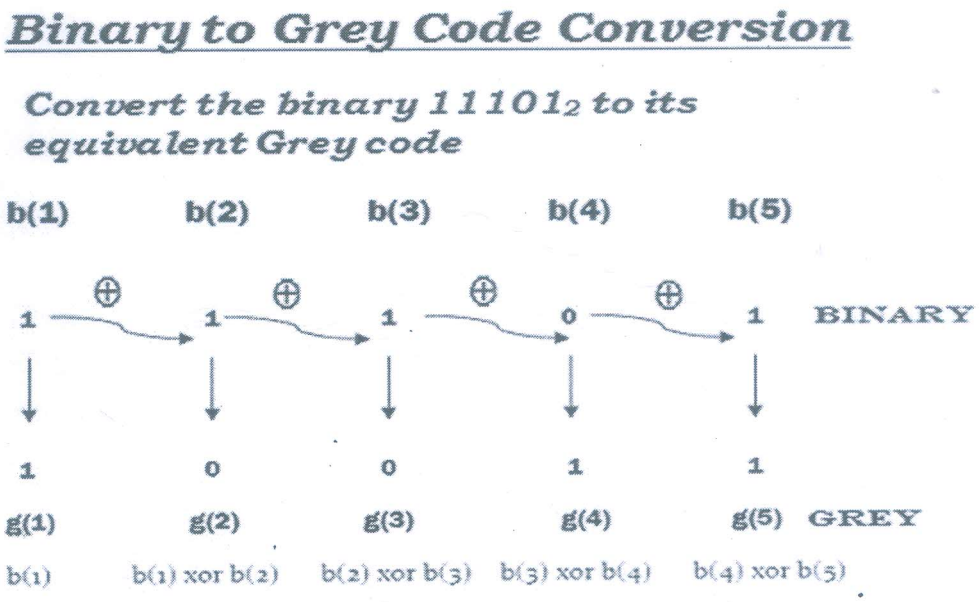
The logical circuit which converts binary code to equivalent gray code is known as **binary to gray code converter**. The gray code is a non weighted code. The successive gray code differs in one bit position only that means it is a unit distance code. It is also referred as cyclic code. It is not suitable for arithmetic operations. It is the most popular of the unit distance codes. It is also a reflective code.

Example:

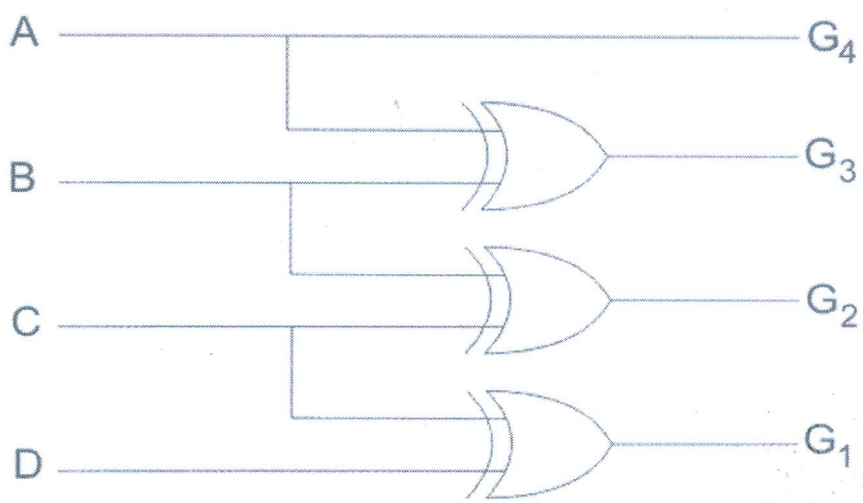
Problem

Find the equivalent grey code for the binary 11101_2.

Solution :



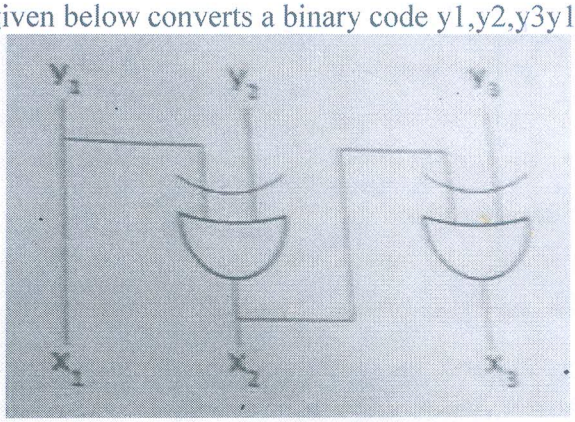
Circuit Diagram :



Logic Circuit for Binary to Gray Code Converter

Questions:

1) Convert the binary number 1100 to Gray code.
2) The binary number 11101011000111010 can be written in hexadecimal as _____.
3) The logic circuit given below converts a binary code y1,y2,y3y1,y2,y3 into .....



(i)Excess-3 code
(ii)Gray code
(iii)BCD code

Gray to Binary Code Converter :

The conversion of gray to binary code also requires XOR'ing .but this time bits of gray code is XOR'ed with output binary code bits .The M.S.B is written as it is . then the output M.S.B in binary is XOR'ed with the adjacent bit in the gray code . and then the next adjacent bit if gray code is XOR'ed with last obtained binary bit .

Example :

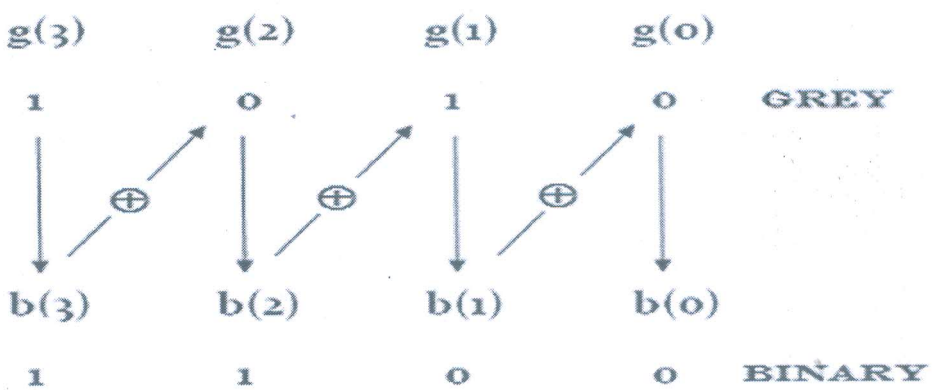
Problem

Find the equivalent binary number for the grey code 1010.

Solution :

Grey Code to Binary Conversion

Convert the Grey code 1010 to its equivalent Binary



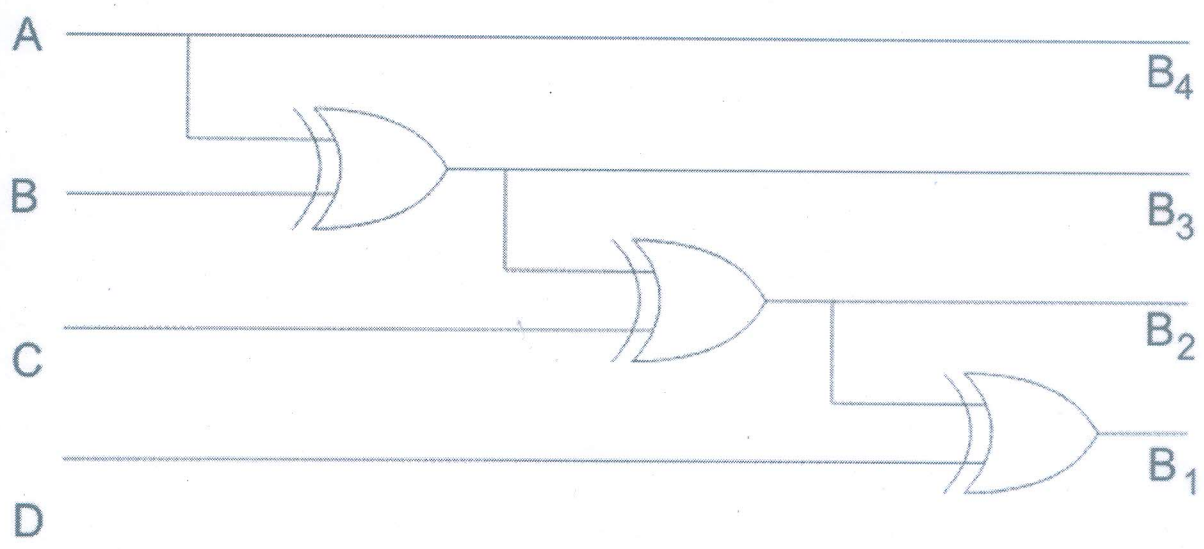
i.e $b(3) = g(3)$

$b(2) = b(3) \oplus g(2)$

$b(1) = b(2) \oplus g(1)$

$b(0) = b(1) \oplus g(0)$

Circuit Diagram:



Logic Circuit for Gray to Binary Code Converter

Questions:

1. Perform the following conversion (1029)10 to gray.
2. What is the feature of gray code? What are its applications?
3. Convert the gray code number 11011 to binary.

Program-4

Implementing 3-8 line DECODER and Implementing 4x1 and 8x1 MULTIPLEXERS

ENCODER

An encoder is an electronic device used to convert an analogue signal to a digital signal such as a BCD code. It has a number of input lines, but only one of the inputs is activated at a given time and produces an N-bit output code that depends on the activated input. The encoders and decoders are used in many electronics projects to compress the multiple number of inputs into smaller number of outputs. The encoder allows 2 power N inputs and generates N-number of outputs. For example, in 4-2 encoder, if we give 4 inputs it produces only 2 outputs.

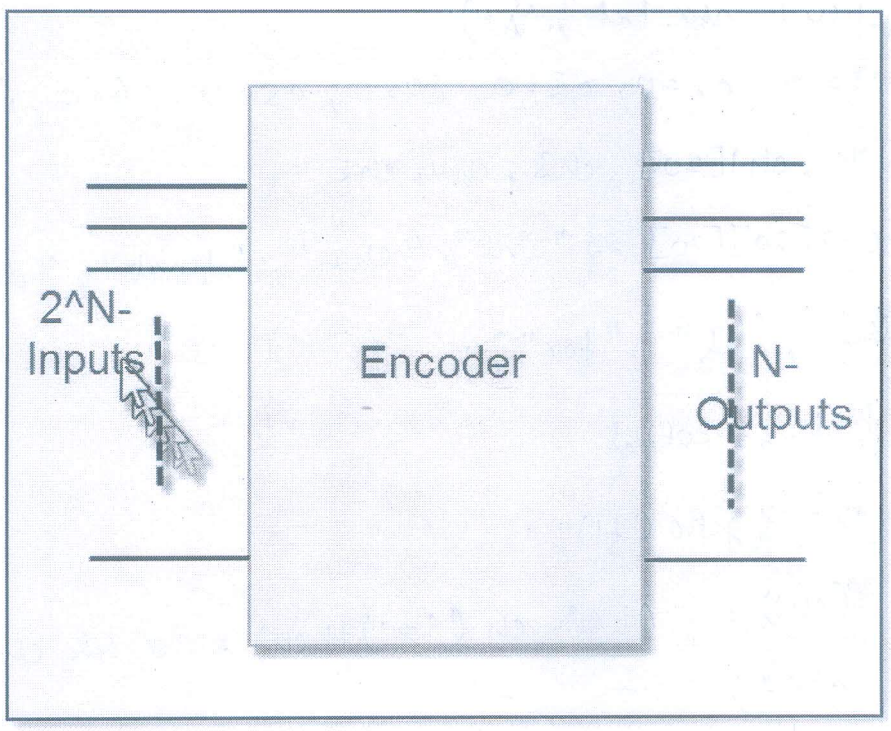
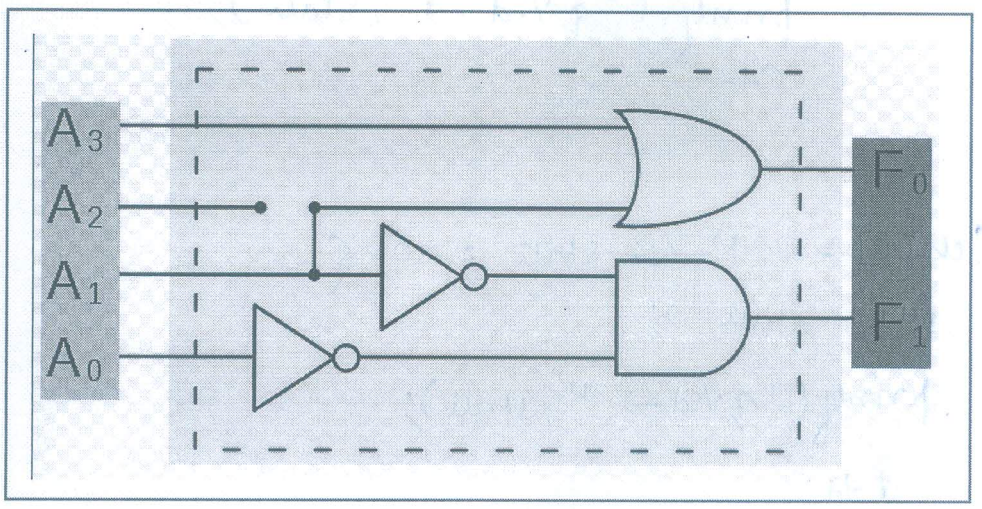


Fig 1:Encoder

Truth Table of The Encoder

The decoders and encoders are designed with logic gate such as an OR-gate. There are different types of encoders and decoders like 4, 8, and 16 encoders and the truth table of encoder depends upon a particular encoder chosen by the user. Here, a 4-bit encoder is being explained along with the truth table. The four-bit encoder allows only four inputs such as A0, A1, A2, A3 and generates the two outputs F0, F1, as shown in below diagram.



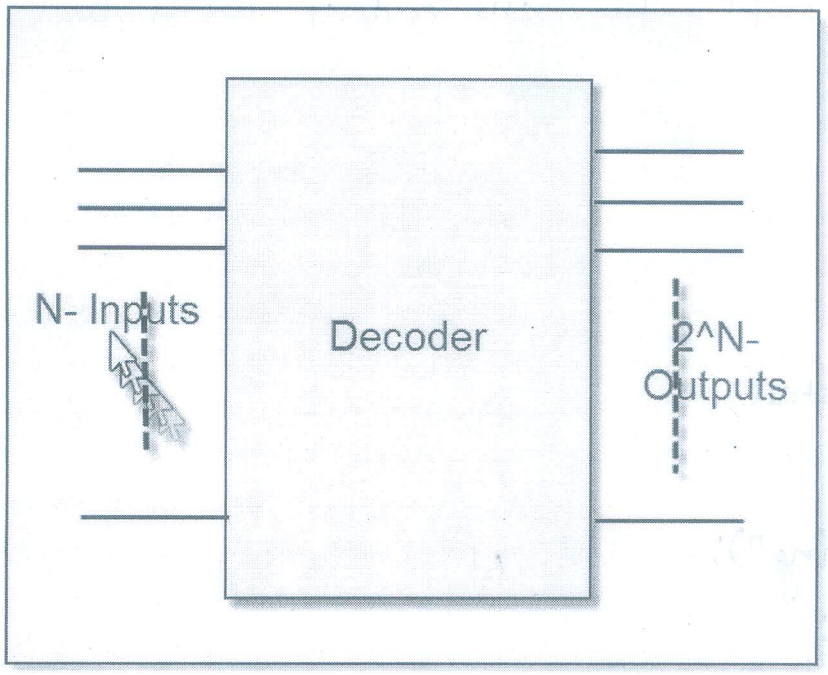
Simple Encoder

I_3	I_2	I_1	I_0	O_1	O_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Encoder Truth Table

Introduction of Decoder

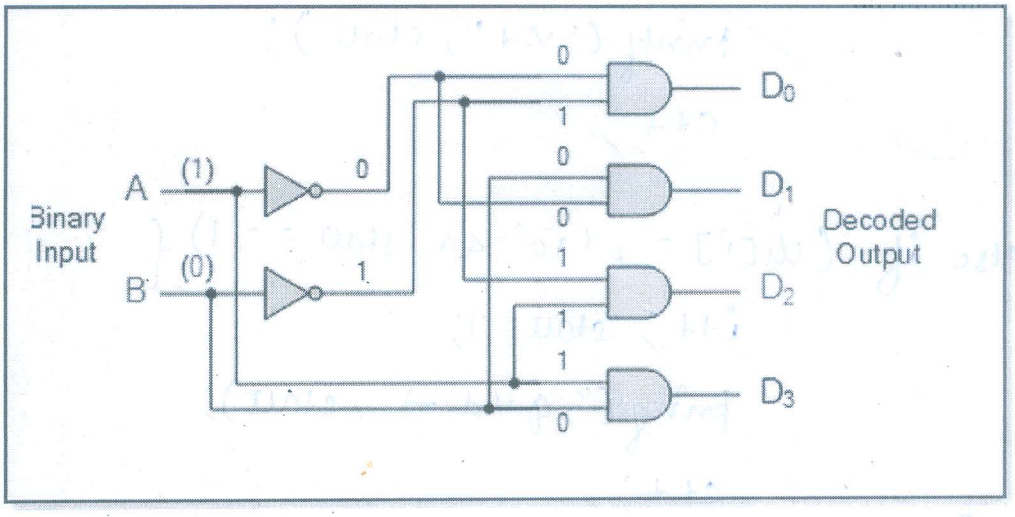
The decoder is an electronic device that is used to convert digital signal to an analogue signal. It allows single input line and produces multiple output lines. The decoders are used in many communication projects that are used to communicate between two devices. The decoder allows N- inputs and generates 2 power N-numbers of outputs. For example, if we give 2 inputs that will produce 4 outputs by using 4 by 2 decoder.



Decoder

Truth Table of The Decoder

The encoders and decoders are designed with logic gates such as AND gate. There are different types of decoders like 4, 8, and 16 decoders and the truth table of decoder depends upon a particular decoder chosen by the user. The subsequent description is about a 4-bit decoder and its truth table. The four bit decoder allows only four outputs such as A0, A1, A2, A3 and generates two outputs F0, F1, as shown in the below diagram.



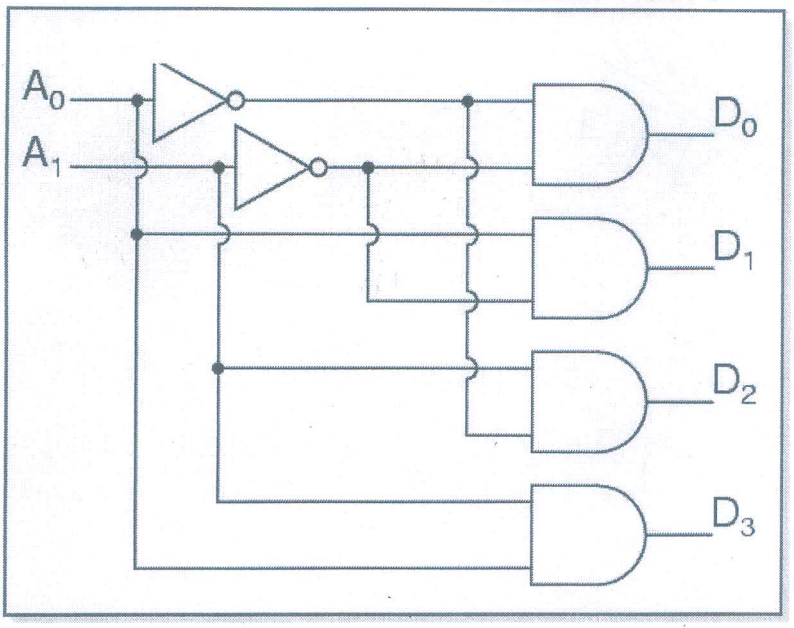
Decoder Circuit

2-to-4 line Decoder

In this type of encoders and decoders, decoders contain two inputs A0, A1, and four outputs represented by D0, D1, D2, and D3. As you can see in the truth table – for each input combination, one output line is activated.

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Decoder Truth T

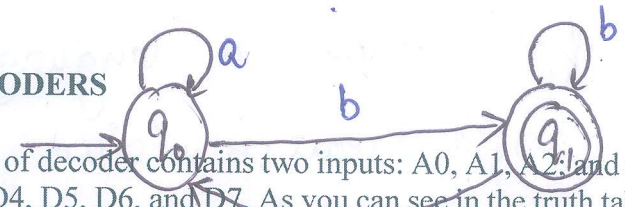


2-to-4 Decoder

In this example, you can notice that, each output of the decoder is actually a minterm, resulting from a certain inputs combination, that is:

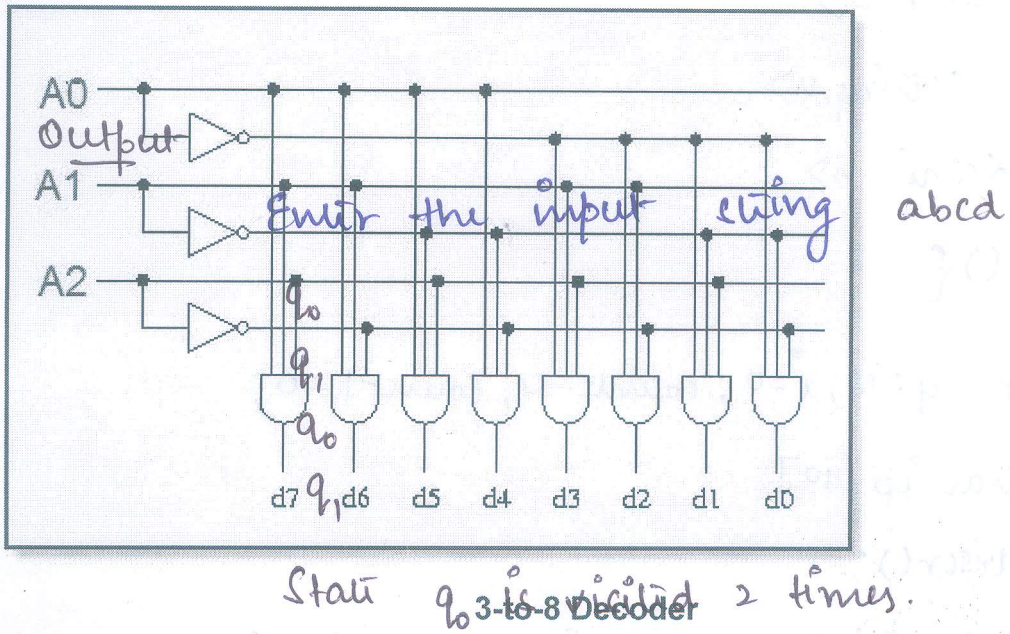
- D0 =A1 A0, (minterm m0) which corresponds to input 00
- D1 =A1 A0, (minterm m1) which corresponds to input 01
- D2 =A1 A0, (minterm m2) which corresponds to input 10
- D3 =A1 A0, (minterm m3) which corresponds to input 11

The circuit is implemented with AND gates, as shown in the figure. In this circuit, the logic equation for D0 is A1/A0, and so on. Thus, each output of the decoder will be generated to the input combination.



3-8 DECODERS

This type of decoder contains two inputs: A0, A1, A2, and four outputs represented by D0, D1, D2, D3, D4, D5, D6, and D7. As you can see in the truth table, for each input combination, one output line is activated. For example, an input will activate the line A0, A1, A3 as 01 at the input has activated line D1, and so on.



3-to-8 Decoder

In this example, you can notice that each output of the decoder is actually a minterm, resulting from a certain inputs combination, that is;

- D0 =A2 A1 A0, (minterm m0) which corresponds to input 000
- D1 = A2 A1 A0, (minterm m1) which corresponds to input 001
- D2 = A2 A1 A0, (minterm m2) which corresponds to input 010
- D3 = A2 A1 A0, (minterm m3) which corresponds to input 011
- D4 = A2 A1 A0, (minterm m0) which corresponds to input 100
- D5 = A2 A1 A0, (minterm m1) which corresponds to input 101
- D6 = A2 A1 A0, (minterm m2) which corresponds to input 110
- D7 = A2 A1 A0, (minterm m3) which corresponds to input 111

The circuit is implemented with AND gates, as shown in the figure. In this circuit, the logic equation for D0 is A2/A1/A0/, and so on. Thus, each output of the decoder will be generated to the input combination.

Implementing Multiplexers

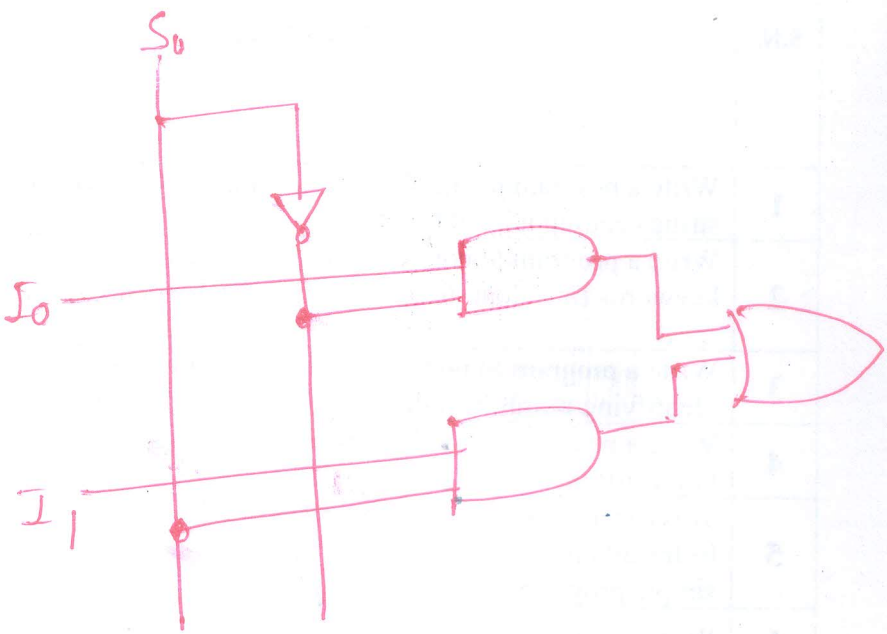
Multiplexer

In electronics, a **multiplexer** (or **mux**) is a device that selects one of several analog or digital input signals and forwards the selected input into a single line. A multiplexer of 2^n inputs has *[n]* select lines, which are used to select which input line to send to the output. Multiplexers are mainly used to increase the amount of data that can be sent over the network within a certain amount of time and bandwidth.A multiplexer is also called a **data selector**. Multiplexers can also be used to implement Boolean functions of multiple variables.

A 2-to-1 multiplexer has a boolean equation where and are the two inputs, is the selector input, and is the output:

Which can be expressed as a truth table:

S	A	B	Z
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



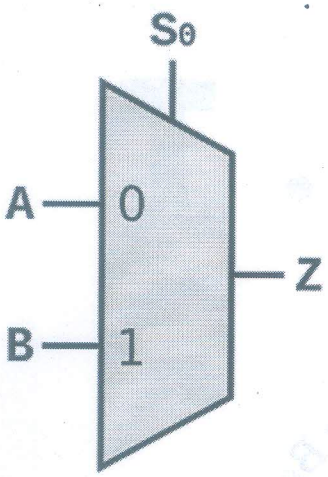
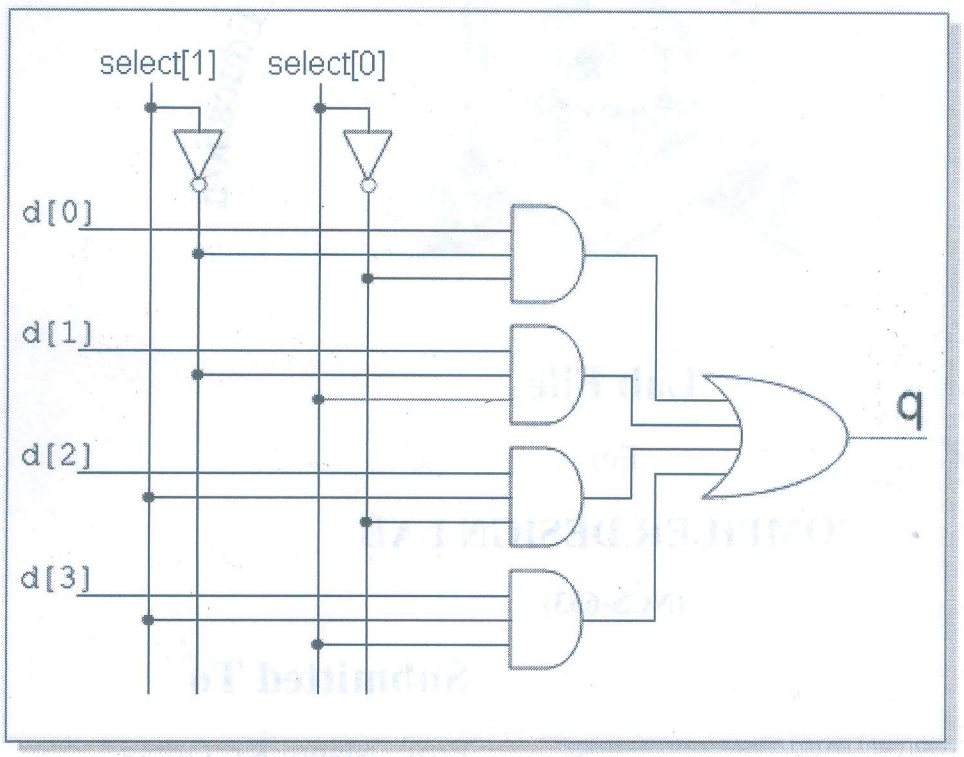
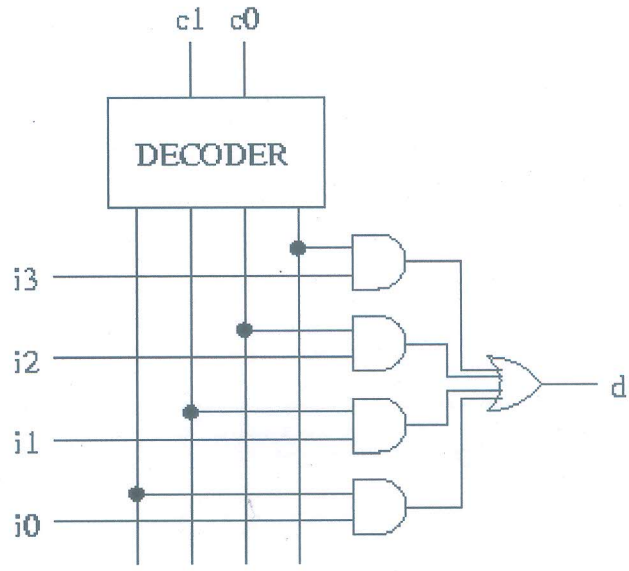


Fig: A 2-to-1 mux



The implementation of a multiplexer is straightforward, and uses a decoder. Here is a 4x1 multiplexer.



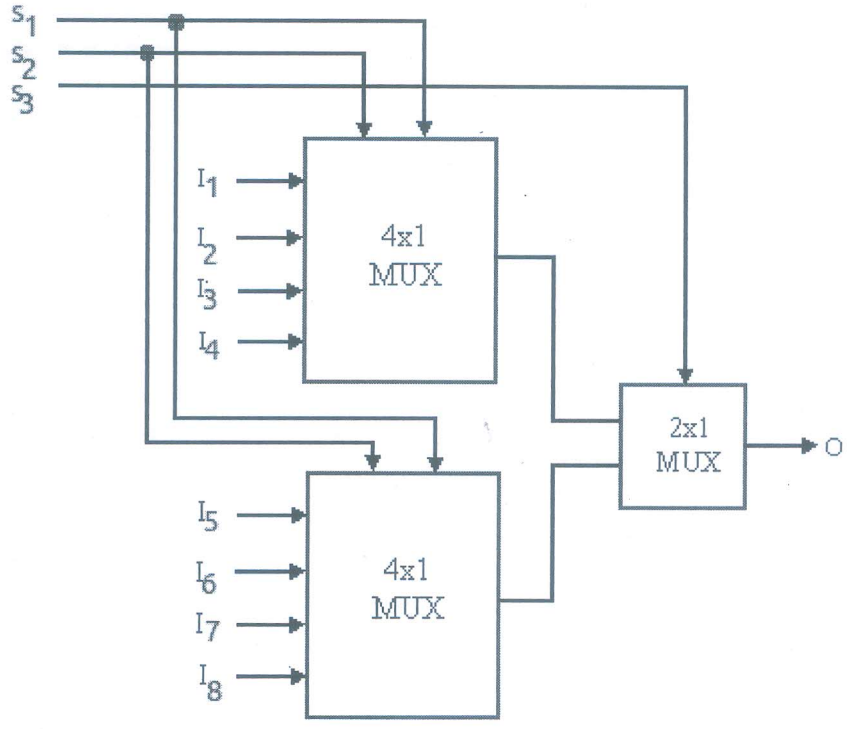
All the outputs of the decoder are 0, apart from one. The inputs c_1c_0 determine which of the outputs is non-zero.

All but one of the AND gates have 0 on one input and therefore output 0. The remaining AND gate has 1 on one input and i_n (where n is represented in binary by c_1c_0) on the other input. The output of this AND gate is the value of i_n.

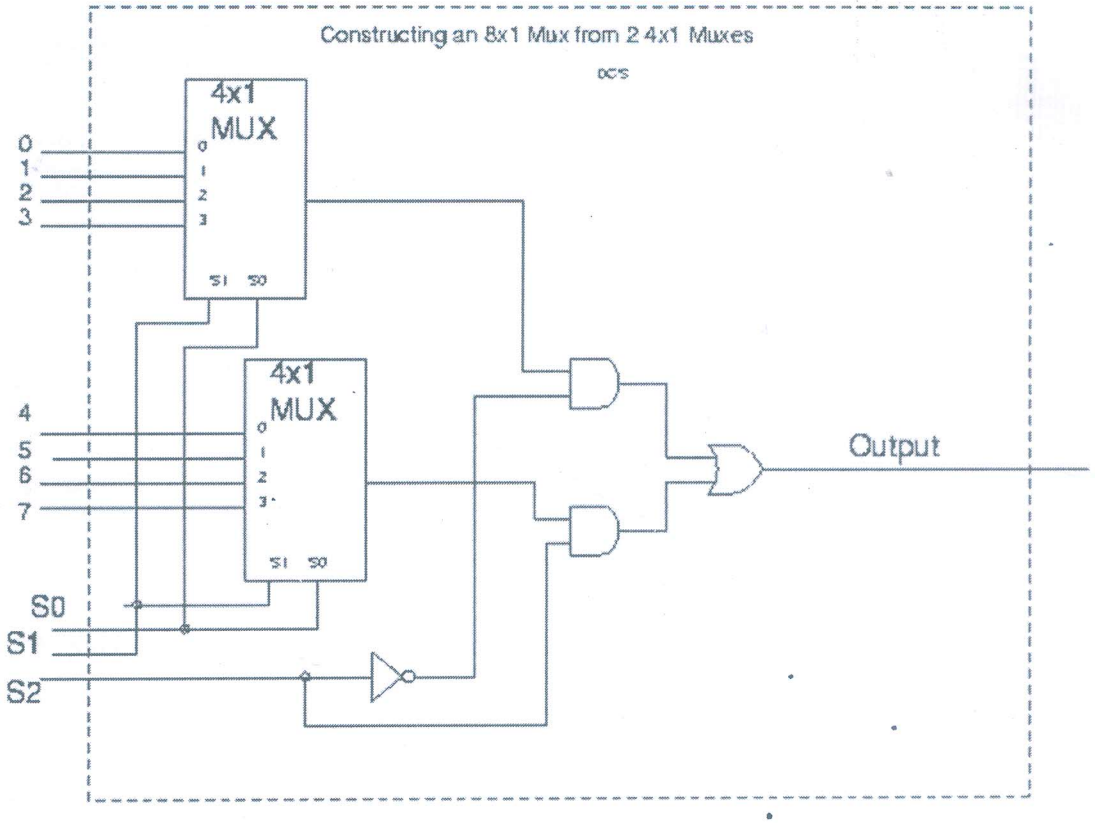
The OR gate has 0 on all of its inputs apart from one, and has the value of i_n on the remaining input. The output of the OR gate is therefore the value of i_n.

Larger multiplexers can be implemented in the same way.

8x1 Multiplexer

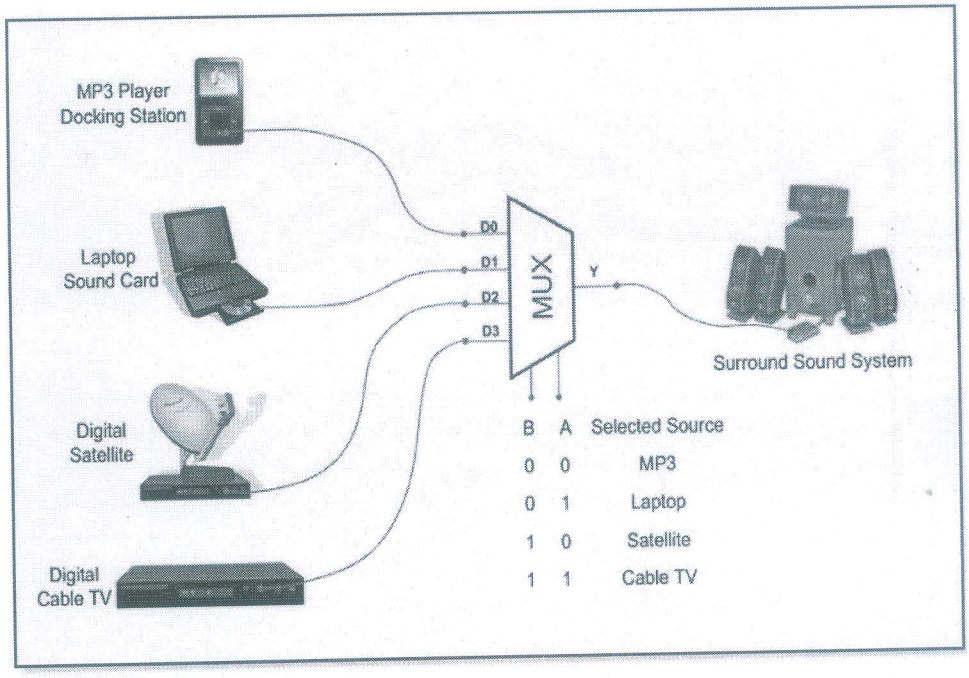


Implementation of 8x1 MUX using 4x1 and 2x1 MUXs



Multiplexer Applications

The multiplexers and demultiplexers are digital electronic devices that are used to control applications. A multiplexer is a device that allows multiple input signals and produces a single output signal. For example, sometimes we need to produce a single output from multiple input :lines. Electronic multiplexer can be considered as a multiple input and single output lines. In this case, the multiplexer used selects the input line to be sent to the output. The digital code is applied to the selected inputs to generate respective output. The digital code is applied to the selected inputs to generate respective output A common application of multiplexing occurs when several embedded system devices share a single transmission line or bus line while communicating with the device. Each device in succession has a brief time to send and receive the data. This is the special advantage of using this MUX.



Multiplexer

7. Latches and Flip-Flops

Latches and *flip-flops* are the basic elements for storing information. One latch or flip-flop can store one bit of information. The main difference between latches and flip-flops is that for latches, their outputs are constantly affected by their inputs as long as the enable signal is asserted. In other words, when they are enabled, their content changes immediately when their inputs change. Flip-flops, on the other hand, have their content change only either at the rising or falling edge of the enable signal. This enable signal is usually the controlling clock signal. After the rising or falling edge of the clock, the flip-flop content remains constant even if the input changes.

There are basically four main types of latches and flip-flops: SR, D, JK, and T. The major differences in these flip-flop types are the number of inputs they have and how they change state. For each type, there are also different variations that enhance their operations. In this chapter, we will look at the operations of the various latches and flip-flops.

7.1 Bistable Element

The simplest sequential circuit or storage element is a *bistable element*, which is constructed with two inverters connected sequentially in a loop as shown in Figure 1. It has no inputs and two outputs labeled Q and Q'. Since the circuit has no inputs, we cannot change the values of Q and Q'. However, Q will take on whatever value it happens to be when the circuit is first powered up. Assume that $Q = 0$ when we switch on the power. Since Q is also the input to the bottom inverter, Q', therefore, is a 1. A 1 going to the input of the top inverter will produce a 0 at the output Q, which is what we started off with. Similarly, if we start the circuit with $Q = 1$, we will get $Q' = 0$, and again we get a stable situation.

A bistable element has memory in the sense that it can remember the content (or state) of the circuit indefinitely. Using the signal Q as the state variable to describe the state of the circuit, we can say that the circuit has two stable states: $Q = 0$, and $Q = 1$; hence the name "bistable."

An analog analysis of a bistable element, however, reveals that it has three equilibrium points and not two as found from the digital analysis. Assuming again that $Q = 1$, and we plot the output voltage (V_{out1}) versus the input voltage (V_{in1}) of the top inverter, we get the solid line in Figure 2. The dotted line shows the operation of the bottom inverter where V_{out2} and V_{in2} are the output and input voltages respectively for that inverter.

Figure 2 shows that there are three intersection points, two of which corresponds to the two stable states of the circuit where Q is either 0 or 1. The third intersection point labeled *metastable*, is at a voltage that is neither a logical 1 nor a logical 0 voltage. Nevertheless, if we can get the circuit to operate at this voltage, then it can stay at that point indefinitely. Practically, however, we can never operate a circuit at precisely a certain voltage. A slight deviation from the metastable point as cause by noise in the circuit or other stimulants will cause the circuit to go to one of the two stable points. Once at the stable point, a slight deviation, however, will not cause the circuit to go away from the stable point but rather back towards the stable point because of the feedback effect of the circuit.

An analogy of the metastable behavior is a ball on top of a symmetrical hill as depicted in Figure 3. The ball can stay indefinitely in that precarious position as long as there is absolutely no movement whatsoever. With any slight force, the ball will roll down to either of the two sides. Once at the bottom of the hill, the ball will stay there until an external force is applied to it. The strength of this external force will cause the ball to do one of three things. If a

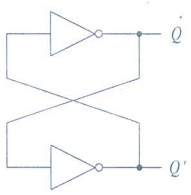


Figure 1. Bistable element.

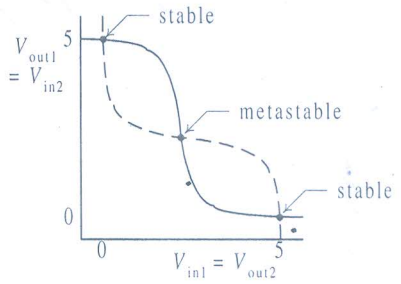


Figure 2. Analog analysis of bistable element.

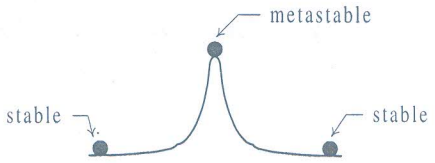


Figure 3. Ball and hill analogy for metastable behavior.

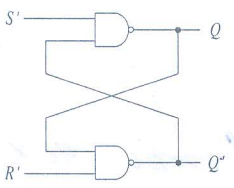
small force is applied to the ball, it will go partly up the hill and then rolls back down to the same side. If a big enough force is applied to it, it will go over the top and down the other side of the hill. We can also apply a force that is just strong enough to push the ball to the top of the hill. Again at this precarious position, it can roll down either side.

We will find that all latches and flip-flops have this metastable behavior. In order for the element to change state, we need to apply a strong enough pulse satisfying a given minimum width requirement. Otherwise, the element will either remain at the current state or go into the metastable state in which case unpredictable results can occur.

7.2 SR Latch

The bistable element is able to remember or store one bit of information. However, because it does not have any inputs, we cannot change the information bit that is stored in it. In order to change the information bit, we need to add inputs to the circuit. The simplest way to add inputs is to replace the two inverters with two NAND gates as shown in Figure 4(a). This circuit is called a *SR latch*. In addition to the two outputs Q and Q', there are two inputs S' and R' for *set* and *reset* respectively. Following the convention, the prime in S and R denotes that these inputs are active low. The SR latch can be in one of two states: a set state when $Q = 1$, or a reset state when $Q = 0$.

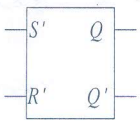
To make the SR latch go to the set state, we simply assert the S' input by setting it to 0. Remember that 0 NAND anything gives a 1, hence $Q = 1$ and the latch is set. If R' is not asserted ($R' = 1$), then the output of the bottom NAND gate will give a 0, and so $Q' = 0$. This situation is shown in Figure 4 (d) at time t_0. If we de-assert S' so that $S' = R' = 1$, the latch will remain at the set state because Q', the second input to the top NAND gate, is 0 which will keep $Q = 1$ as shown at time t_1. At time t_2 we reset the latch by making $R' = 0$. Now, Q' goes to 1 and this will force Q to go to a 0. If we de-assert R' so that again we have $S' = R' = 1$, this time the latch will remain at the reset state as shown at time t_3. Notice the two times (at t_1 and t_3) when both S' and R' are de-asserted. At t_1, Q is at a 1, whereas, at t_3, Q is at



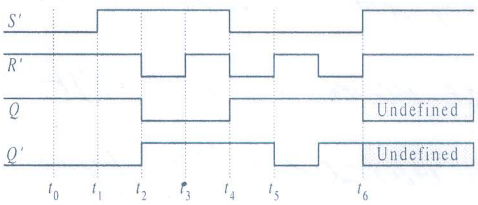
(a)

S	R	Q	Q_{next}	Q_{next}'
0	0	×	1	1
0	1	×	1	0
1	0	×	0	1
1	1	0	0	1
1	1	1	1	0

(b)

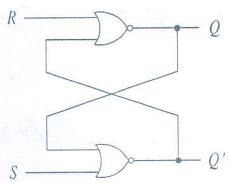


(c)



(d)

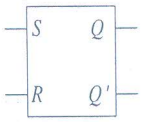
Figure 4. SR latch: (a) circuit using NAND gates; (b) truth table; (c) logic symbol; (d) timing diagram.



(a)

S	R	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	×	0	1
1	0	×	1	0
1	1	×	0	0

(b)



(c)

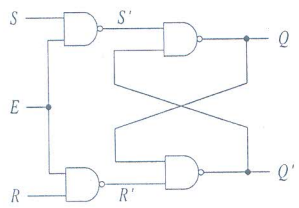
Figure 5. SR latch: (a) circuit using NOR gates; (b) truth table; (c) logic symbol.

a 0. When both inputs are de-asserted, the SR latch maintains its previous state. Previous to t_1, Q has the value 1, so at t_1, Q remains at a 1. Similarly, previous to t_3, Q has the value 0, so at t_3, Q remains at a 0.

If both S' and R' are asserted, then both Q and Q' are equal to 1 as shown at time t_4. If one of the input signals is de-asserted earlier than the other, the latch will end up in the state forced by the signal that was de-asserted later as shown at time t_5. At t_5, R' is de-asserted first, so the latch goes into the normal set state with $Q = 1$ and $Q' = 0$.

A problem exists if both S' and R' are de-asserted at exactly the same time as shown at time t_6. If both gates have exactly the same delay then they will both output a 0 at exactly the same time. Feeding the zeros back to the gate input will produce a 1, again at exactly the same time, which again will produce a 0, and so on and on. This oscillating behavior, called the *critical race*, will continue forever. If the two gates do not have exactly the same delay then the situation is similar to de-asserting one input before the other, and so the latch will go into one state or the other. However, since we do not know which is the faster gate, therefore, we do not know which state the latch will go into. Thus, the latch's next state is undefined.

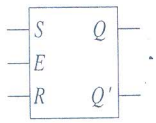
In order to avoid this indeterministic behavior, we must make sure that the two inputs are never de-asserted at the same time. Note that both of them can be de-asserted, but just not at the same time. In practice, this is guaranteed by not having both of them asserted. Another reason why we do not want both inputs to be asserted is that when they are both asserted, Q is equal to Q', but we usually want Q to be the inverse of Q'.



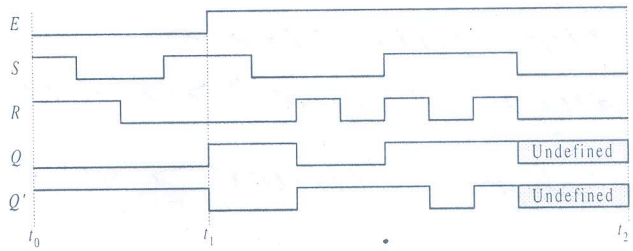
(a)

E	S	R	Q	Q_{next}	Q_{next}'
0	×	×	0	0	1
0	×	×	1	1	0
1	0	0	0	0	1
1	0	0	1	1	0
1	0	1	×	0	1
1	1	0	×	1	0
1	1	1	×	1	1

(b)

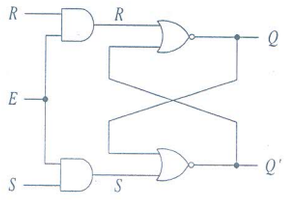


(c)



(d)

Figure 6. SR latch with enable: (a) circuit using NAND gates; (b) truth table; (c) logic symbol; (d) timing diagram.



(a)

E	S	R	Q	Q_{next}	Q_{next}'
0	×	×	0	0	1
0	×	×	1	1	0
1	0	0	0	0	1
1	0	0	1	1	0
1	0	1	×	0	1
1	1	0	×	1	0
1	1	1	×	0	0

(b)

Figure 7. SR latch with enable: (a) circuit using NOR gates; (b) truth table.

From the above analysis, we obtain the truth table in Figure 4(b) for the NAND implementation of the SR latch. Q is the current state or the current content of the latch and Q_{next} is the value to be updated in the next state. Figure 4(c) shows the logic symbol for the SR latch.

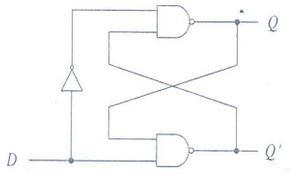
The SR latch can also be implemented using NOR gates as shown in Figure 5(a). The truth table for this implementation is shown in Figure 5(b). From the truth table, we see that the main difference between this implementation and the NAND implementation is that for the NOR implementation, the S and R inputs are active high, so that setting S to 1 will set the latch and setting R to 1 will reset the latch. However, just like the NAND implementation, the latch is set when $Q = 1$ and reset when $Q = 0$. The latch remembers its previous state when $S = R = 0$. When $S = R = 1$, both Q and Q' are 0. The logic symbol for the SR latch using NOR implementation is shown in Figure 5(c).

7.3 SR Latch with Enable

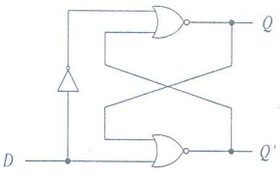
The SR latch is sensitive to its inputs all the time. It is sometimes useful to be able to disable the inputs. The *SR latch with enable* (also known as a *gated SR latch*) accomplishes this by adding an enable input, E, to the original implementation of the latch that allows the latch to be enabled or disabled. The circuit for the SR latch with enable using NAND gates is shown in Figure 6(a), its truth table in Figure 6(b), and logic symbol in Figure 6(c). When $E = 1$, the circuit behaves like the normal NAND implementation of the SR latch except that the S and R inputs are active high rather than low. When $E = 0$, the latch remains in its previous state regardless of the S and R inputs. In actual circuits, the enable input can either be active high or low, and may be named *ENABLE*, *CLK*, or *CONTROL*. A typical operation of the latch is shown in the timing diagram in Figure 6(d). Between t_0 and t_1, $E = 0$ so changing the S and R inputs do not affect the output. Between t_1 and t_2, $E = 1$ and the trace is similar to the trace of Figure 4(d) except that the input signals are inverted.

The SR latch with enable can also be implemented using NOR gates as shown Figure 7.

7.4 D Latch



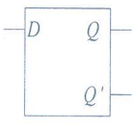
(a)



(b)

D	Q	Q_{next}	Q_{next}'
0	×	0	1
1	×	1	0

(c)

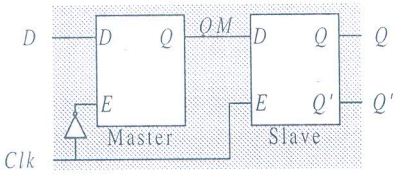


(d)

Figure 8. D latch: (a) circuit using NAND gates; (b) circuit using NOR gates; (c) truth table; (d) logic symbol.

slave latch will not change. The circuit of Figure 10(a) is called a *positive* edge-triggered flip-flop because the output Q on the slave latch changes only at the rising edge of the clock. If the slave latch is enabled when the clock is low, then it is referred to as a *negative* edge-triggered flip-flop. The circuit of Figure 10(a) is also referred to as a *master-slave* D flip-flop because of the two latches used in the circuit. Figure 10(b) and (c) show the truth table and the logic symbol respectively. Figure 10(d) shows the timing diagram for the D flip-flop.

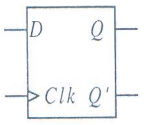
Another way of constructing a positive-edge-triggered flip-flop is to use three interconnected SR latches rather than a master and slave D latch with enable. The circuit is shown in Figure 11. The advantage of this circuit is that it uses only 6 NAND gates (26 transistors) as opposed to 10 gates (46 transistors) for the master-slave D flip-flop of Figure 10(a). The operation of the circuit is as follows. When $E = 0$, the outputs of gates 2 and 3 are high (0 NAND x = 1). Thus $n_2 = n_3 = 1$, which maintains the output latch, comprising gates 5 and 6, in its current state. At the same time $n_4 = D'$ since one input to gate 4 is n_3 which is a 1 (1 NAND $x = x'$). Similarly, $n_1 = D$. When E changes to 1, n_2 will be equal to $n_1' = D'$, while n_3 will be equal to D. So if $D = 0$, then n_3 will be 0, thus asserting R' and resetting the output latch Q to 0. On the other hand, if $D = 1$, then n_2 will be 0, thus asserting S' and setting the output latch Q to 1. Once $E = 1$, changing D will not change n_2 or n_3, so Q will remain stable during the remaining time that E is asserted.



(a)

Clk	D	Q	Q_{next}	Q_{next}'
0	×	0	0	1
0	×	1	1	0
1	×	0	0	1
1	×	1	1	0
↓	0	×	0	1
↓	1	×	1	0

(b)



(c)

Figure 10. Master-slave positive-edge-triggered D flip-flop: (a) circuit using D latches; (b) truth table; (c) logic symbol; (d) timing diagram.

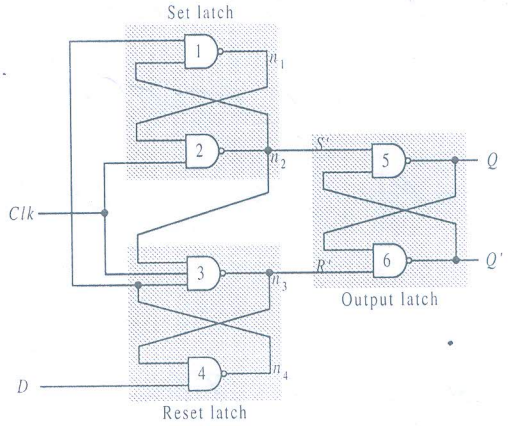


Figure 11. Positive-edge-triggered D flip-flop.

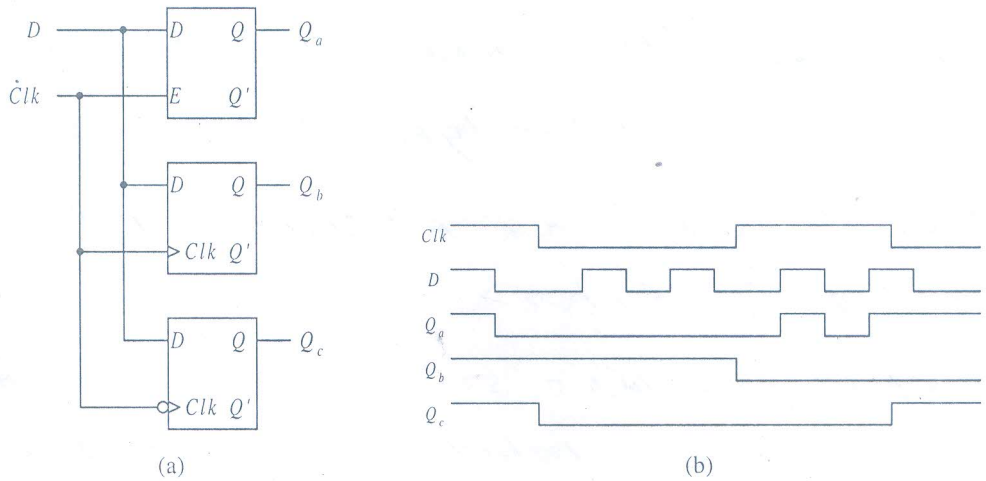
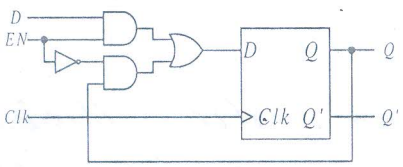


Figure 12. Comparison of a gated latch, a positive-edge-triggered flip-flop, and a negative-edge-triggered flip-flop: (a) circuit; (b) timing diagram.

Figure 12 compares the different operations between a latch and a flip-flop. In (a), we have a gated D latch, a positive-edge-triggered D flip-flop and a negative-edge-triggered D flip-flop, all having the same *D* input and controlled by the same clock signal. (b) shows a sample trace of the circuit's operations. Notice that the gated D latch Q_a follows the *D* input as long as the clock is high. The positive-edge-triggered flip-flop Q_b responds to the *D* input only at the rising edge of the clock while the negative-edge-triggered flip-flop Q_c responds to the *D* input only at the falling edge of the clock.

7.7 D Flip-Flop with Enable

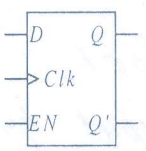
A commonly desired function in D flip-flops is the ability to hold the last value stored rather than load in a new value at the clock edge. This is accomplished by adding an enable input called *EN* or *CE* (clock enable) through a multiplexer as shown in Figure 13(a). When *EN* = 1, the primary *D* signal will pass to the *D* input of the flip-flop, thus updating the content of the flip-flop. When *EN* = 0, the bottom AND gate is enabled and so the current content of the flip-flop, *Q*, is passed back to the input, thus, keeping its current value. Notice that changes to the flip-flop value occur only at the rising edge of the clock. The truth table and the logic symbol for the D flip-flop with enabled is shown in (b) and (c) respectively.



(a)

Clk	*EN*	*D*	*Q*	Q_{next}	Q_{next}'
0	×	×	0	0	1
0	×	×	1	1	0
1	×	×	0	0	1
1	×	×	1	1	0
↑	0	×	0	0	1
↑	0	×	1	1	0
↑	1	0	×	0	1
↑	1	1	×	1	0

(b)



(c)

Figure 13. D flip-flop with enable: (a) circuit; (b) truth table; (c) logic symbol.

7.8 Asynchronus Inputs

Flip-flops, as we have seen so far, change states at the edge of a synchronizing clock signal. Many circuits require the initialization of flip-flops to a known state independent of the clock signal. Sequential circuits that change states whenever a change in input values occurs independent of the clock are referred to as *asynchronous sequential circuits*. *Synchronous sequential circuits*, on the other hand, change states only at the edge of the clock signal. Asynchronous inputs are usually available for both flip-flops and latches, and they are used to either set or clear the storage element's content independent of the clock.

Figure 14(a) shows a D latch with asynchronous *PRESET'* and *CLEAR'* inputs, and (b) is the logic symbol for it. (c) is the circuit for the D edge-triggered flip-flop with asynchronous *PRESET'* and *CLEAR'* inputs, and (d) is the logic symbol for it. When *PRESET'* is asserted (set to 0) the content of the storage element is set to a 1 immediately, and when *CLEAR'* is asserted (set to 0) the content of the storage element is set to a 0 immediately.

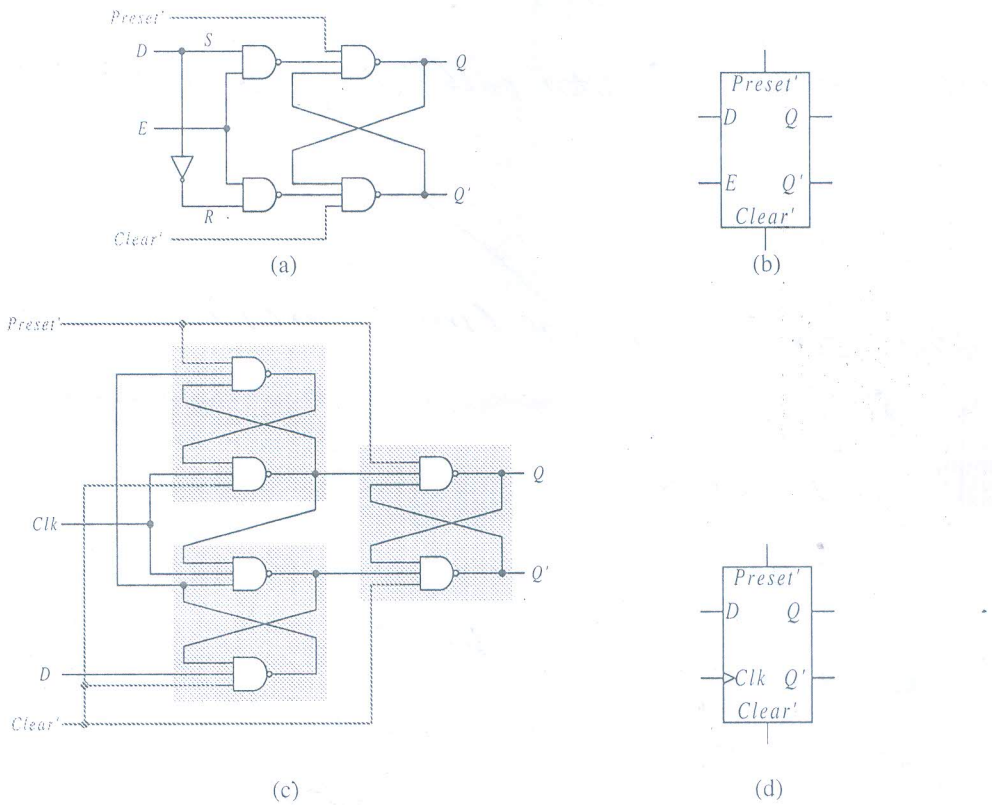


Figure 14. Storage elements with asynchronous inputs: (a) D latch with preset and clear; (b) logic symbol for (a); (c) D edge-triggered flip-flop with preset and clear; (d) logic symbol for (c).

7.9 Flip-Flop Types

There are basically four main types of flip-flops: SR, D, JK, and T. The major differences in these flip-flop types are in the number of inputs they have and how they change state. Each type can have different variations such as active high or low inputs, whether they change state at the rising or falling edge of the clock signal, and whether they have asynchronous inputs or not. The flip-flops can be described fully and uniquely by its logic symbol, characteristic table, characteristic equation, state diagram, or excitation table, and are summarized in Figure 15.

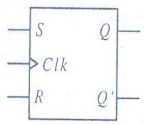
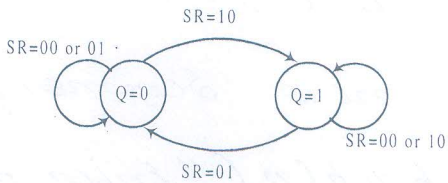
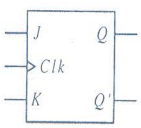
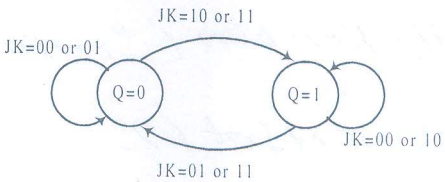
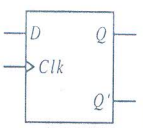
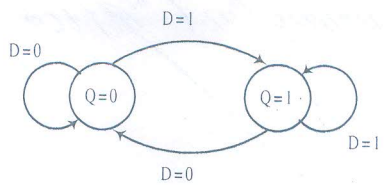
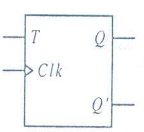
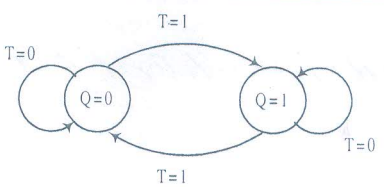
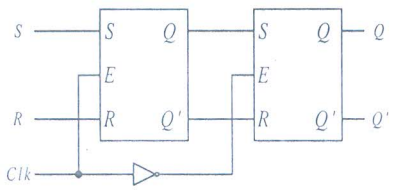
Name / Symbol	Characteristic (Truth) Table	State Diagram / Characteristic Equations	Excitation Table
SR 	S R Q Qnext 0 0 0 0 0 0 1 1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 1 1 1 1 0 × 1 1 1 ×	 $Q_{next} = S + R'Q$ $SR = 0$	Q Qnext S R 0 0 0 × 0 1 1 0 1 0 0 1 1 1 × 0
JK 	J K Q Qnext 0 0 0 0 0 0 1 1 0 1 0 0 0 1 1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

Figure 15. Flip-flop types.

7.9.1 SR Flip-Flop

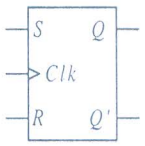
We can replace the D latches in the D flip-flop of Figure 10(a) with SR latches to get a master-slave SR flip-flop shown in Figure 16. Like SR latches, SR flip-flops are useful in control applications where we want to be able to set or reset the data bit. However, unlike SR latches, SR flip-flops change their content only at the active edge of the clock signal. Similar to SR latches, SR flip-flops can enter an undefined state when both inputs are asserted simultaneously.



(a)

S	R	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	×	×
1	1	1	×	×

(b)

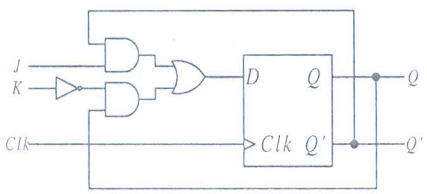


(c)

Figure 16. SR flip-flop: (a) circuit; (b) truth table; (c) logic symbol.

7.9.2 JK Flip-Flop

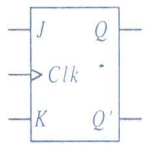
JK flip-flops are very similar to SR flip-flops. The J input is just like the S input in that when asserted, it sets the flip-flop. Similarly, the K input is like the R input where it clears the flip-flop when asserted. The only difference is when both inputs are asserted. For the SR flip-flop, the next state is undefined, whereas, for the JK flip-flop, the next state is the inverse of the current state. In other words, the JK flip-flop toggles its state when both inputs are asserted. The circuit, truth table and the logic symbol for the JK flip-flop is shown in Figure 17.



(a)

J	K	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

(b)

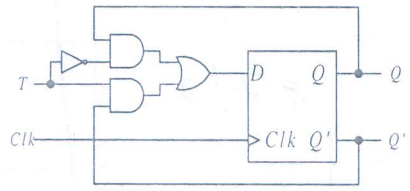


(c)

Figure 17. JK flip-flop: (a) circuit; (b) truth table; (c) logic symbol.

7.9.3 T Flip-Flop

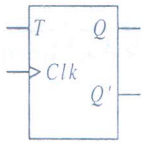
The T flip-flop has one input in addition to the clock. T stands for toggle for the obvious reason. When T is asserted ($T = 1$), the flip-flop state toggles back and forth, and when T is de-asserted, the flip-flop keeps its current state. The T flip-flop can be constructed using a D flip-flop with the two outputs Q and Q' feedback to the D input through a multiplexer that is controlled by the T input as shown in Figure 18.



(a)

T	Q	Q_{next}	Q_{next}'
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

(b)

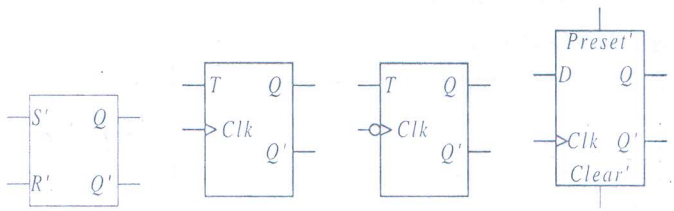


(c)

Figure 18. T flip-flop: (a) circuit; (b) truth table; (c) logic symbol.

7.9.4 Logic Symbol

The *logic* or *graphical symbol* describes the flip-flop's inputs and outputs, the names given to these signals, and whether they are active high or low. All the flip-flops have Q and Q' as their outputs. All of them also have a *CLK* input. The small triangle at the clock input indicates that the circuit is a flip-flop and so it is triggered by the edge of the clock signal; if there is a circle in front, then it is the falling edge, otherwise, it is the rising edge of the clock



(a) (b) (c) (d)

Figure 19. Various logic symbols: (a) Active low SR latch; (b) positive-edge-triggered active high T flip-flop; (c) negative-edge-triggered T flip-flop; (d) positive-edge-triggered D flip-flop with asynchronous active low preset and clear.

signal. Without the small triangle, the circuit is a latch. In addition, the flip-flops have one or two more inputs that characterize the flip-flop and give it its name. Figure 19 shows several sample logic symbols for various memory elements.

7.9.5 Characteristic Table

The *characteristic table* is just the truth table but usually written in a shorter format. For example, compare the characteristic table for the JK flip-flop in Figure 20 with the truth table in Figure 17(b). The truth table, as we have seen, simply lists all possible combinations of the input signals, the current state (or content) of the flip-flop, and the next state that the flip-flop will go to at the next active edge of the clock signal. The characteristic table answers the question of what is the next state when given the inputs and the current state, and is used in the analysis of sequential circuits.

J	K	Q_{next}
0	0	Q
0	1	0
1	0	1
1	1	Q'

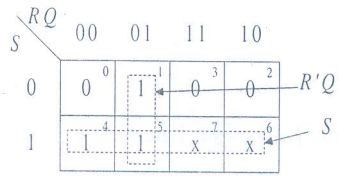
Figure 20. JK flip-flop characteristic table.

7.9.6 Characteristic Equation

The *characteristic equation* is the functional Boolean equation that is derived from the characteristic table. This equation formally describes the functional behavior of the flip-flop. Like the characteristic table, it specifies the flip-flop's next state as a function of its current state and inputs. For example, the characteristic equation for the JK flip-flop can be derived from the truth table as follows:

$$\begin{aligned} Q_{next} &= J'K'Q + JK'Q + JK'Q' + JKQ' \\ &= K'Q(J'+J) + JQ'(K'+K) \\ &= K'Q + JQ' \end{aligned}$$

The characteristic equation can also be obtained from the truth table using the K-map method as follows for the SR flip-flop:



Thus, the characteristic equation for the SR flip-flop is

$$Q_{next} = S + R'Q.$$

7.9.7 State Diagram

A *state diagram* is a graph that shows the flip-flop's operations in terms of how it transitions from one state to another. The nodes are labeled with the states and the directed arcs are labeled with the input signals that cause the transition to go from one state to the next. Figure 21 shows the state diagram for the SR flip-flop. For example, to go from state $Q = 0$ to the state $Q = 1$, the two inputs S and R have to be 1 and 0 respectively. Similarly, if the current state is $Q = 0$ and we want to remain in that state, then SR need to be 00 or 01.

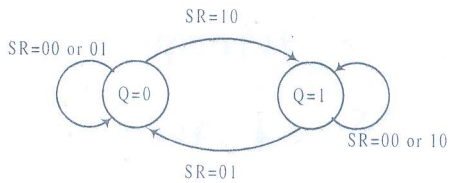


Figure 21. State diagram for the SR flip-flop.

7.9.8 Excitation Table

The *excitation table* gives the value of the flip-flop's inputs that are necessary to change the flip-flop's current state to the desired next state at the next active edge of the clock signal. The excitation table answers the question of what should the inputs be when given the current state that the flip-flop is in and the next state that we want the flip-flop to go to. This table is used in the synthesis of sequential circuits.

Program- 7

Aim: Design of an 8-bit ARITHMETIC LOGIC UNIT.

1. Introduction to Arithmetic Logic Unit

In ECL, TTL and CMOS, there are available integrated packages which are referred to as arithmetic logic units (ALU). The logic circuitry in this units is entirely combinational (i.e. consists of gates with no feedback and no flip-flops).The ALU is an extremely versatile and useful device since, it makes available, in single package, facility for performing many different logical and arithmetic operations.

Arithmetic Logic Unit (ALU) is a critical component of a microprocessor and is the core component of central processing unit.

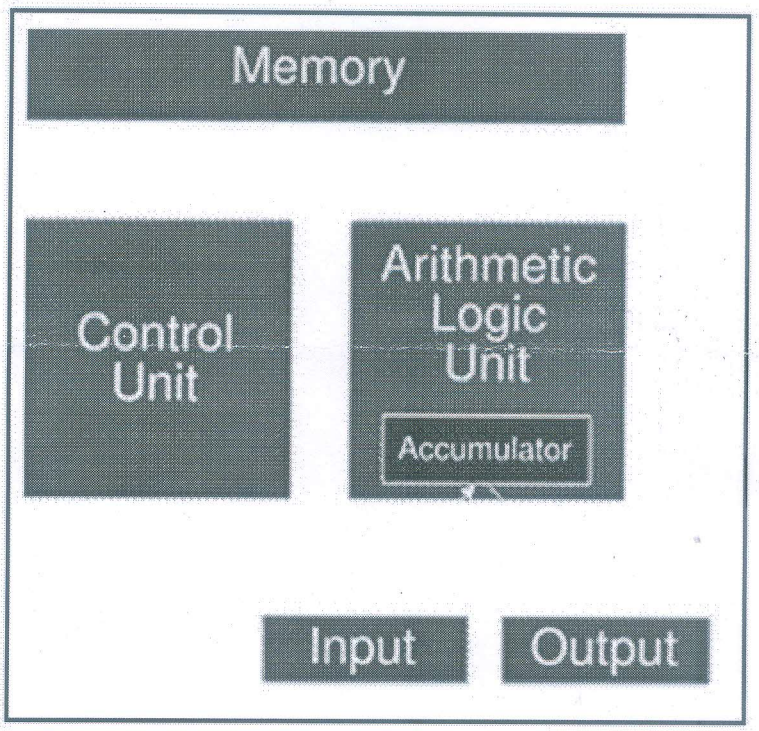


Fig.1 Central Processing Unit (CPU)

ALU's comprise the combinational logic that implements logic operations such as AND, OR and arithmetic operations, such as ADD, SUBTRACT.

Functionally, the operation of typical ALU is represented as shown in diagram below,

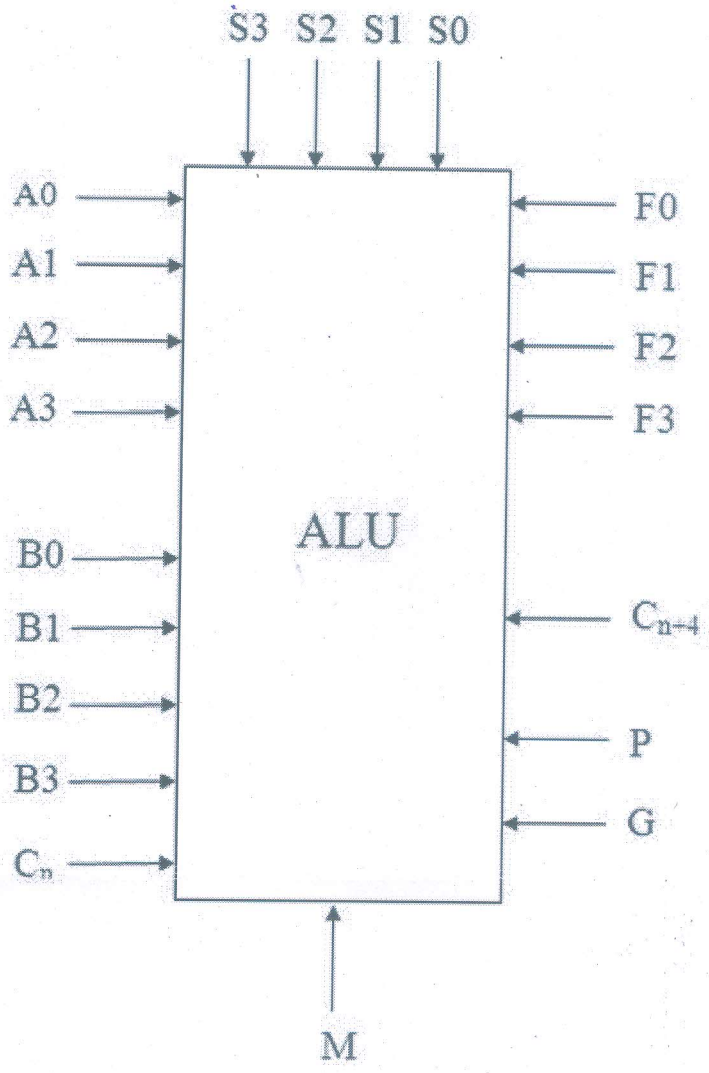


Fig.2 Functional representation of Arithmetic Logic Unit

2. Functional Description of 4-bit Arithmetic Logic Unit

Controlled by the four function select inputs (S0 to S3) and the mode control input (M), ALU can perform all the 16 possible logic operations or 16 different arithmetic operations on active HIGH or active LOW operands.

When the mode control input (M) is HIGH, all internal carries are inhibited and the device performs logic operations on the individual bits. When M is LOW, the carries are enabled and the ALU performs arithmetic operations on the two 4-bit words. The ALU incorporates full internal carry look-ahead and provides for either ripple carry between devices using the Cn+4 output, or for carry look-ahead between packages using the carry propagation (P) and carry generate (G) signals. P and G are not affected by carry in.

For high-speed operation the device is used in conjunction with the ALU·carry look-ahead circuit. One carry look-ahead package is required for each group of four ALU devices. Carry look-ahead can be provided at various levels and offers high-speed capability over extremely long word lengths. The comparator output (A=B) of the device goes HIGH when all four

function outputs (F0 to F3) are HIGH and can be used to indicate logic equivalence over 4 bits when the unit is in the subtract mode. A=B is an open collector output and can be wired-AND with other A=B outputs to give a comparison for more than 4 bits. The open drain output A=B should be used with an external pull-up resistor in order to establish a logic HIGH level. The A=B signal can also be used with the Cn+4 signal to indicate A > B and A < B.

The function table lists the arithmetic operations that are performed without a carry in. An incoming carry adds a one to each operation. Thus, select code LHHL generates A minus B minus 1 (2s complement notation) without a carry in and generates A minus B when a carry is applied.

Because subtraction is actually performed by complementary addition (1s complement), a carry out means borrow; thus, a carry is generated when there is no under-flow and no carry is generated when there is underflow.

As indicated, the ALU can be used with either active LOW inputs producing active LOW outputs (Table 1) or with active HIGH inputs producing active HIGH outputs (Table 2).

MODE SELECT INPUTS				ACTIVE HIGH INPUTS AND OUTPUTS	
S_3	S_2	S_1	S_0	LOGIC (M=H)	ARITHMETIC[2] (M=L; C_n=H)
L	L	L	L	$\overline{A}$	A
L	L	L	H	$\overline{A + B}$	A + B
L	L	H	L	$\overline{A}B$	$A + \overline{B}$
L	L	H	H	logical 0	minus 1
L	H	L	L	$\overline{AB}$	A plus $A\overline{B}$
L	H	L	H	$\overline{B}$	(A + B) plus $A\overline{B}$
L	H	H	L	$A \oplus B$	A minus B minus 1
L	H	H	H	$A\overline{B}$	$A\overline{B}$ minus 1
H	L	L	L	$\overline{A} + B$	A plus AB
H	L	L	H	$\overline{A \oplus B}$	A plus B
H	L	H	L	B	$(A + \overline{B})$ plus AB
H	L	H	H	AB	AB minus 1
H	H	L	L	logical 1	A plus A[1]
H	H	L	H	$A + \overline{B}$	(A + B) plus A
H	H	H	L	A + B	$(A + \overline{B})$ plus A
H	H	H	H	A	A minus 1

MODE SELECT INPUTS				ACTIVE HIGH INPUTS AND OUTPUTS	
S_3	S_2	S_1	S_0	LOGIC (M=H)	ARITHMETIC[2] (M=L; C_n=H)
L	L	L	L	$\overline{A}$	A
L	L	L	H	$\overline{A + B}$	A + B
L	L	H	L	$\overline{A}B$	$A + \overline{B}$
L	L	H	H	logical 0	minus 1
L	H	L	L	$\overline{AB}$	A plus $A\overline{B}$
L	H	L	H	$\overline{B}$	(A + B) plus $A\overline{B}$
L	H	H	L	$A \oplus B$	A minus B minus 1
L	H	H	H	$A\overline{B}$	$A\overline{B}$ minus 1
H	L	L	L	$\overline{A} + B$	A plus AB
H	L	L	H	$\overline{A \oplus B}$	A plus B
H	L	H	L	B	$(A + \overline{B})$ plus AB
H	L	H	H	AB	AB minus 1
H	H	L	L	logical 1	A plus A[1]
H	H	L	H	$A + \overline{B}$	(A + B) plus A
H	H	H	L	A + B	$(A + \overline{B})$ plus A
H	H	H	H	A	A minus 1

3. Logic Diagram

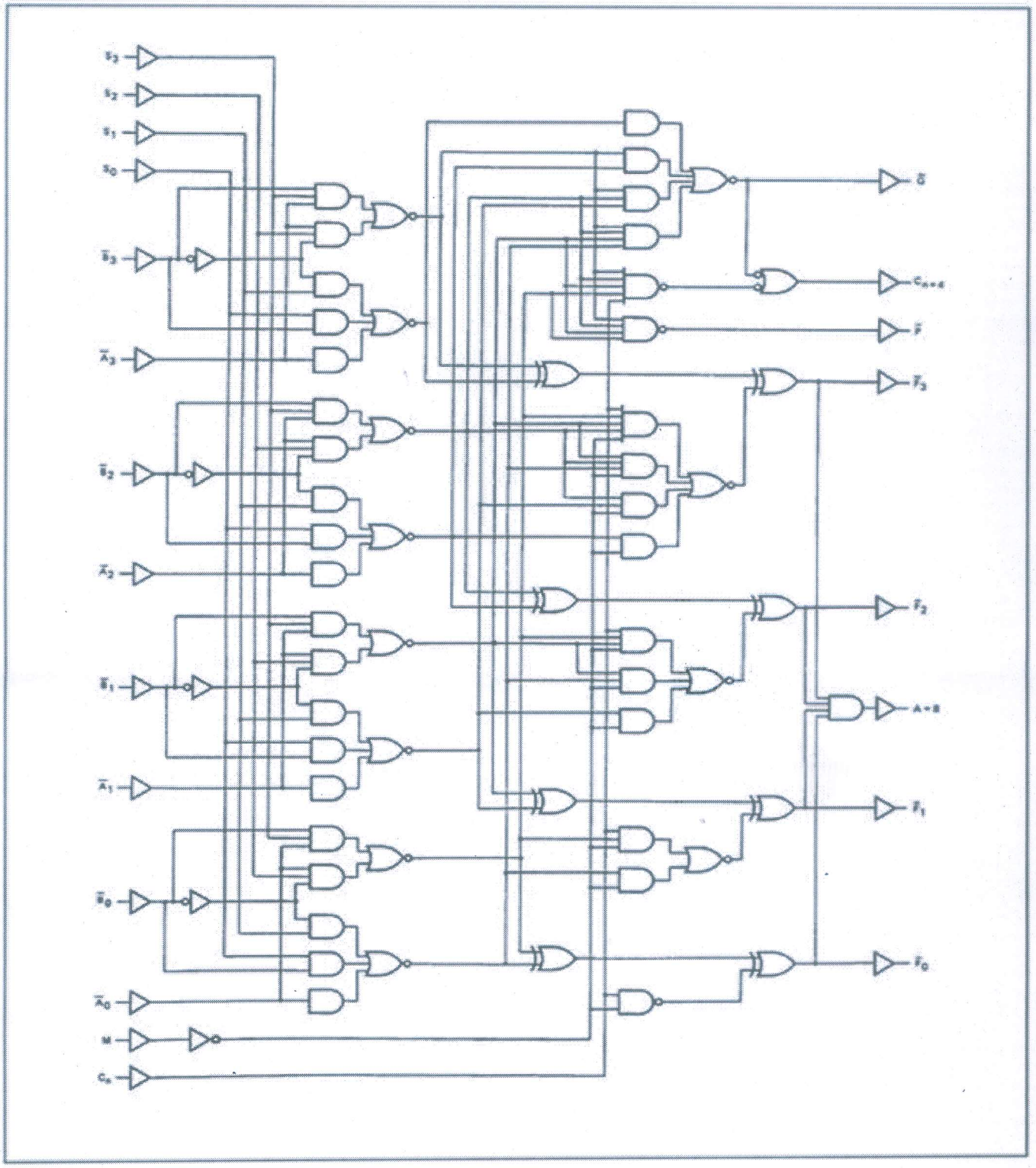


Fig.3 Logic Diagram of Arithmetic Logic Unit

4. Examples for arithmetic operations in ALU

4.1 Binary Adder-Subtractor

The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations. 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 10. The first three operations produce sum of one digit, but when the both augends and addend bits are equal 1, the binary sum consists of two digits. The higher significant bit of the result is called carry .When the augends and addend number contains more significant digits, the carry obtained from the addition of the two bits is called half adder. One that performs the addition of three bits (two significant bits and a previous carry) is called half adder. The name of circuit is from the fact that two half adders can be employed to implement a full adder.
A binary adder-subtractor is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers. Connecting n full adders in cascade produces a binary adder for two n-bit numbers.

The subtraction circuit is included by providing a complementing circuit.

4.1.1 Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adder connected in cascade, the output carry from each full adder connected to the input carry of the next full adder in the chain. Fig. 4 shows the interconnection of four full adder (FA) circuits to provide a 4-bit binary ripple carry adder. The augends bits of A and addend bits of B are designated by subscript numbers from right to left, with subscript 0denoting the least significant bit. The carries are connected in the chain through the full adders. The input carry to the adder is C0 and it ripples through the full adder to the output carry C4.The S output generate the required sum bits. An n-bit adder requires n full adders with each output connected to the input carry of the next higher order full adder.

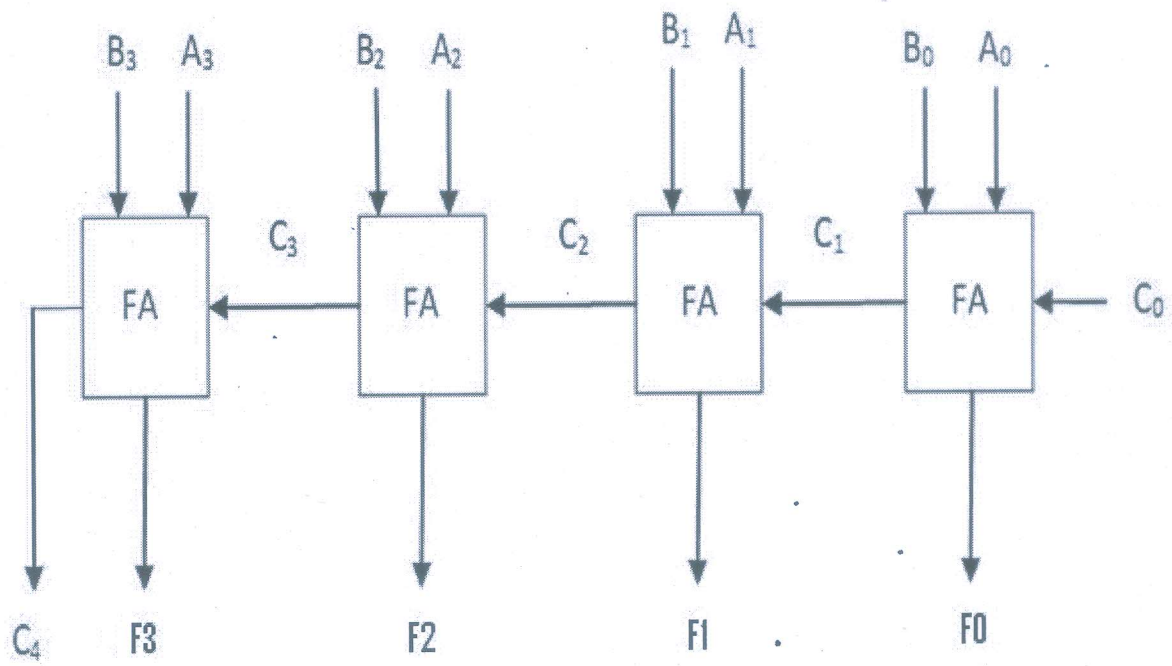


Fig 4: 4-Bit Adder

The bits are added with full adders, starting from the position to form the sum bit and carry. The input carry C0 in the least significant position must be 0. The value of Ci+1 in a given significant position is the output carry of the full adder. This value is transferred into the input carry of the full adder that adds the bits one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available for the correct sum bits to appear at the outputs.

The 4 bit adder is a typical example of a standard component. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the classical method would require a truth table with 29 = 512 entries, since there are nine inputs to the circuit. By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation.

4.1.2 Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complement. Subtraction A–B can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with the inverters and a one can be added to the sum through the input carry.

The circuit for subtracting, A–B, consists of an adder with inverter placed between each data input B and the corresponding input of the full adder. The input carry C0 must be equal to 1when performing subtraction. The operation thus performed becomes A, plus the 1's complement of B, plus 1.This is equal to A plus 2's complement of B. For unsigned numbers this gives A–B if $A \geq B$ or the 2's complement of (B–A) if $A < B$. for signed numbers, the result is A – B, provided that there is no overflow.

The addition and subtraction operations can be combined into one circuit with one common binary adder. This is done by including an EX-OR gate with each full adder. A 4-bit adder-subtractor circuit is shown in fig 5. The mode input M controls the operation. When M = 0, the circuit is an adder, and when M = 1, the circuit becomes a subtractor. Each EX-OR gate receives input M and one of the inputs of B. when M = 0, we have B (Ex-OR) 0 = B. the full adder receive the value of B, the input carry is 0, and the circuit performs A plus B. when M = 1, we have B (Ex-OR) 1= B' and C0 = 1. The B inputs are complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B. (The EX-OR with output is for detecting an overflow.)

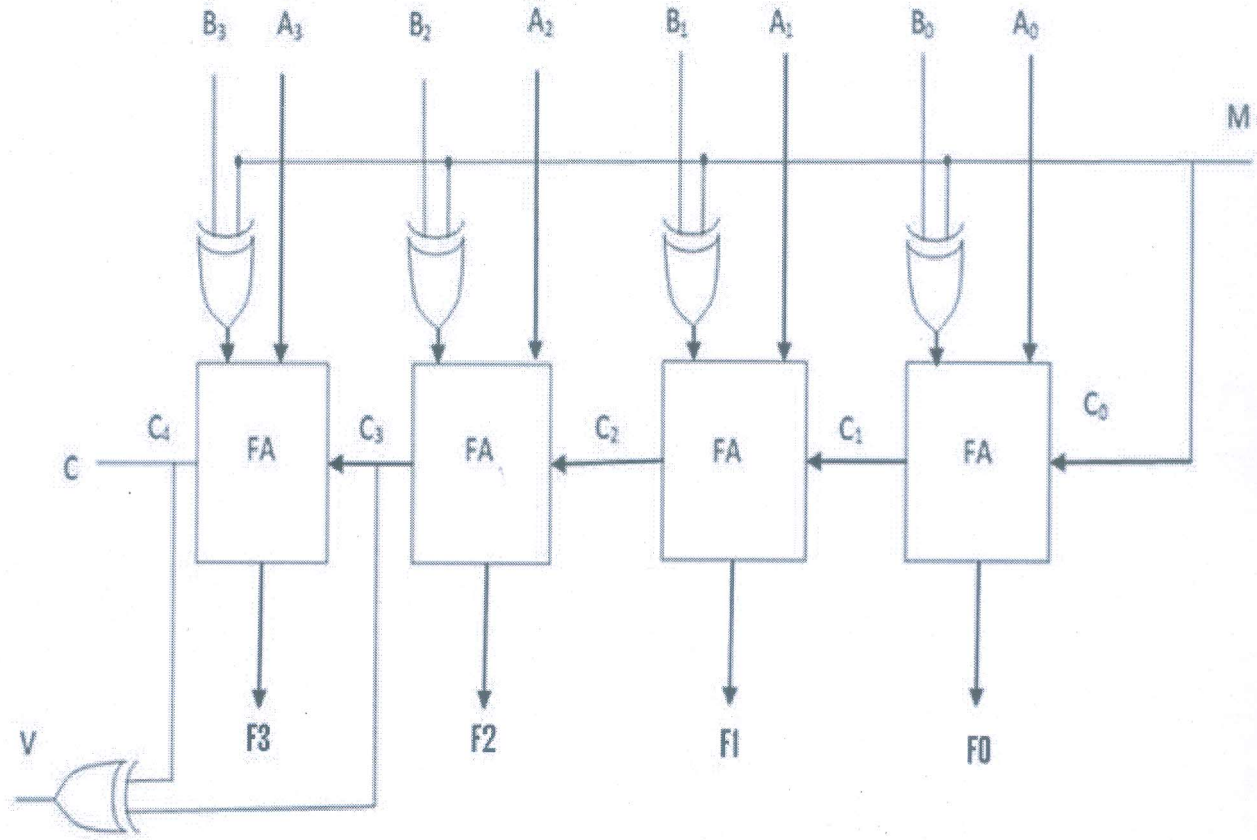


Fig 5: 4-Bit Adder Subtractor

It is worth noting that binary numbers in the signed-complemented system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, computers need only one common hardware circuit to handle both type of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently, depending on whether it is assumed that the numbers are signed or unsigned.

5. Examples for Logical operations in ALU

In a 4-bit Arithmetic Logic Unit, logical operations are performed on individual bits.

5.1 EX-OR

In a 4 bit ALU, the inputs given are A0, A1, A2, A3 and B0, B1, B2, B3. Operations are performed on individual bits. Thus, as shown in a fig.6, inputs, A0 and B0 will give output F0.

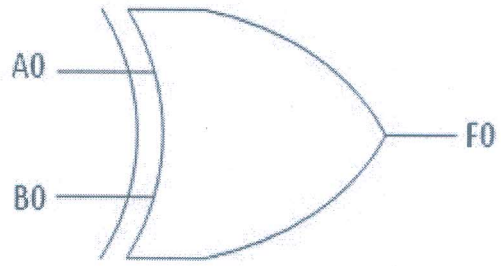


Fig.6 Ex-OR Gate

Inputs		Output
A0	B0	F0
0	0	0
0	1	1
1	0	1
1	1	0

Table 3: Truth table for Ex-OR Gate

Similarly, for other inputs (A1, A2, A3), outputs (F1, F2, F3) are given.

Also, when active low inputs (A0', A1', A2', A3'and B0', B1', B2', B3') are taken, logical operation (here Ex-OR) can be done as shown in fig.7.

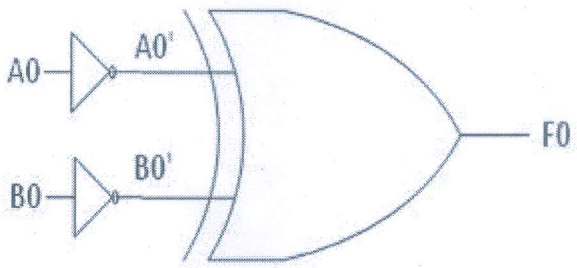


Fig.7 Ex-OR Gate with active low inputs

Inputs		Output
A0'	B0'	F0
1	1	0
1	0	1
0	1	1
0	0	0

Table 4: Truth Table for Ex-OR Gate with active low inputs

Program-8

Aim: Data Path Design of a computer from its register transfer language description.

Datapath is the hardware that performs all the required operations, for example, ALU, registers, and internal buses.

- *Control* is the hardware that tells the datapath what to do, in terms of switching, operation selection, data movement between ALU components, etc.

The processor represented by the shaded block in Figure 4.1 is organized as shown in Figure 1. Observe that the ALU performs I/O on data stored in the register file, while the Control Unit sends (receives) control signals (resp. data) in conjunction with the register file.

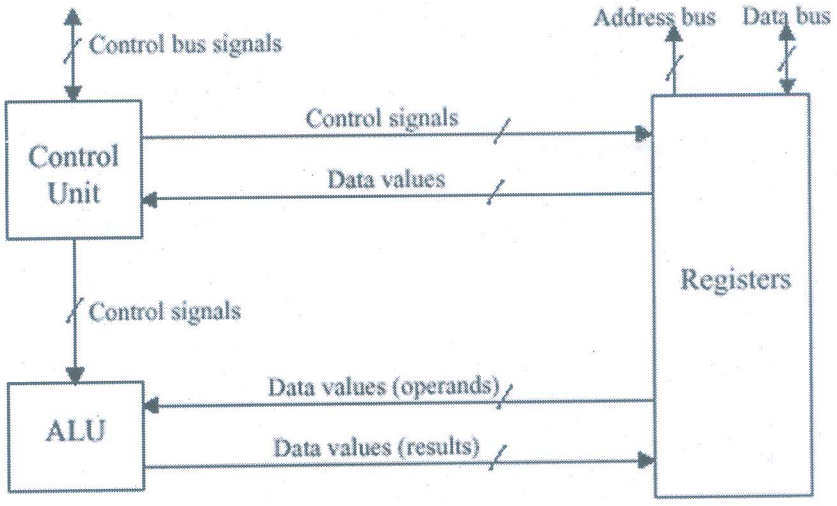


Figure 1. Schematic diagram of the processor in Figure 4.1, adapted from [Maf01].

In MIPS, the ISA determines many aspects of the processor implementation. For example, implementational strategies and goals affect clock rate and CPI. These implementational constraints cause parameters of the components in Figure 2 to be modified throughout the design process.

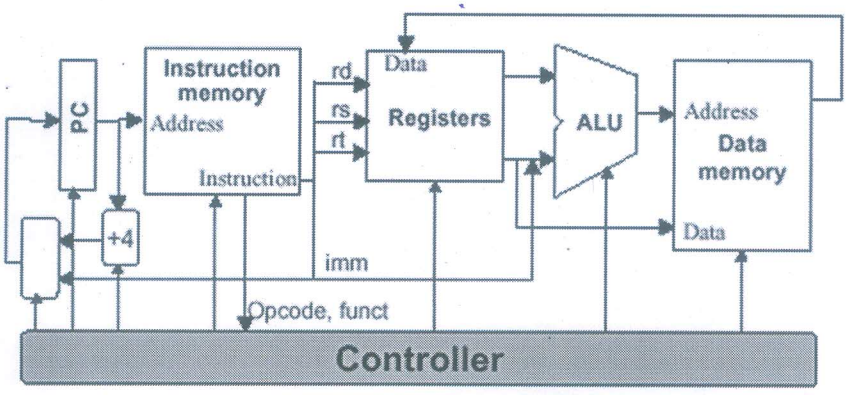
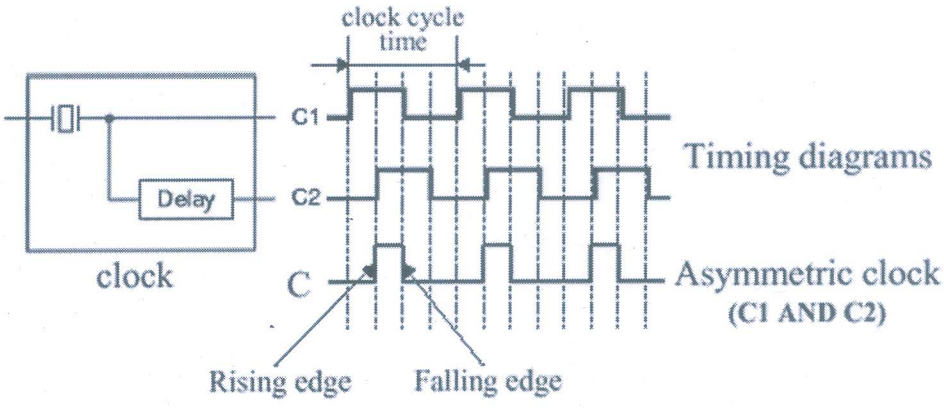


Figure 2. Schematic diagram of MIPS architecture from an implementational perspective, adapted from [Maf01].

Such implementational concerns are reflected in the use of logic elements and clocking strategies. For example, with combinational elements such as adders, multiplexers, or shifters, outputs depend only on current inputs. However, sequential elements such as memory and registers contain state information, and their output thus depends on their inputs (data values and clock) as well as on the stored state. The clock determines the order of events within a gate, and defines when signals can be converted to data to be read or written to processor components (e.g., registers or memory). For purposes of review, the following diagram of clocking is presented:

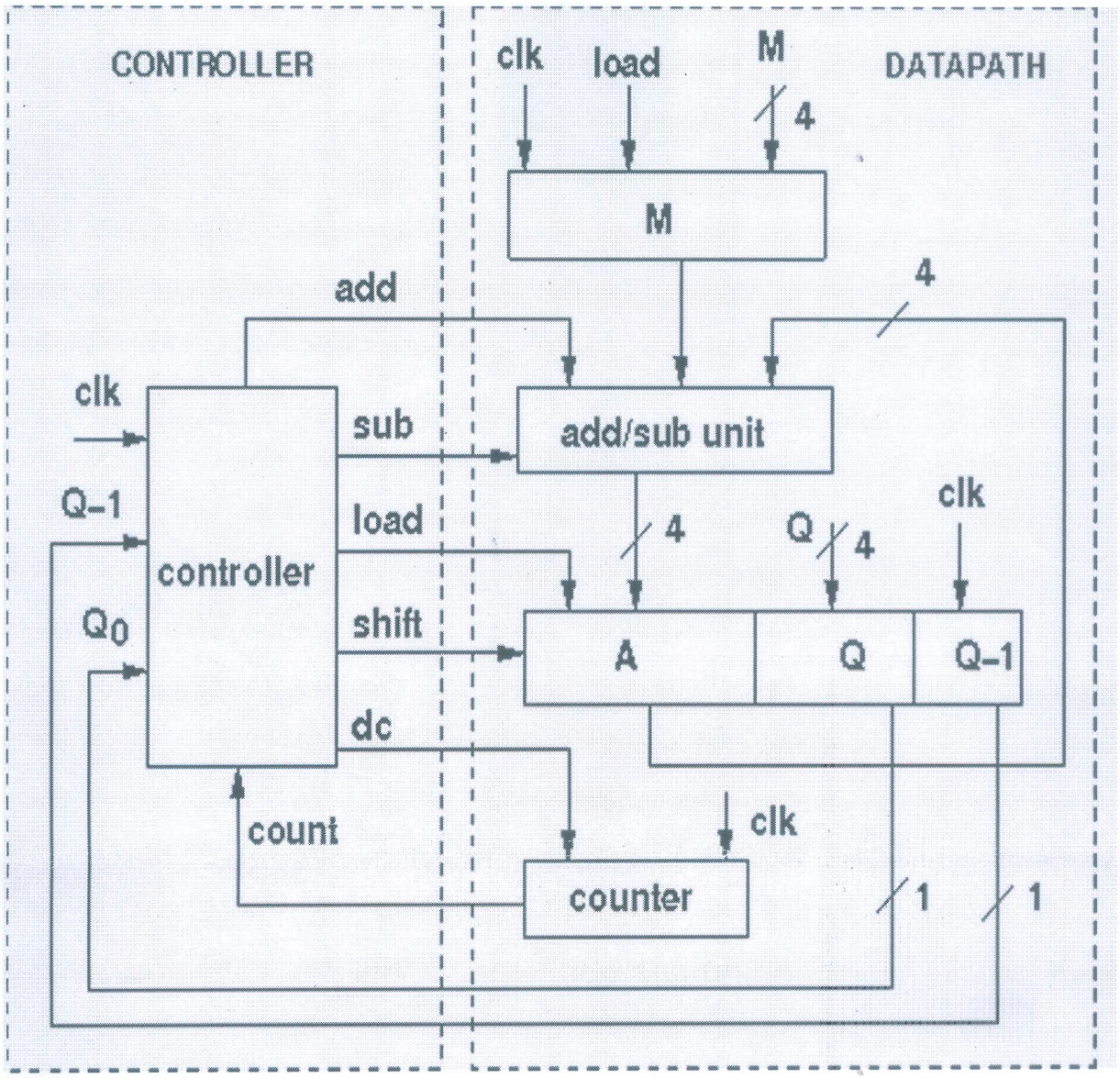


Here, a signal that is held at logic high value is said to be *asserted*. In Section 1, we discussed how edge-triggered clocking can support a precise state transition on the active clock pulse edge (either the rising or falling edge, depending on what the designer selects). We also reviewed the SR Latch based on *nor* logic, and showed how this could

be converted to a clocked SR latch. From this, a clocked D Latch and the D flip-flop were derived. In particular, the D flip-flop has a falling-edge trigger, and its output is initially deasserted (i.e., the logic low value is present).

Design Issues :

Booth's algorithm can be implemented in many ways. This experiment is designed using a controller and a datapath. The operations on the data in the datapath is controlled by the control signal received from the controller. The datapath contains registers to hold multiplier, multiplicand, intermediate results, data processing units like ALU, adder/subtractor etc., counter and other combinational units. Following is the schemetic diagram of the Booth's multiplier which multiplies two 4-bit numbers in 2's complement of this experiment. Here the adder/subtractor unit is used as data processing unit.M, Q, A are 4-bit and Q-1 is a 1-bit rigister. M holds the multiplicand, Q holds the multiplier, A holds the results of adder/subtractor unit. The counter is a down counter which counts the number of operations needed for the multiplication. The data flow in the data path is controlled by the five control signals generated from the controller. these signals are *load* (to load data in registers), *add* (to initiate addition operation), *sub* (to initiate subtraction operation), *shift* (to initiate arithmetis right shift operation), *dc* (this is to decrement counter). The controller generates the control signals according to the input received from the datapath. Here the inputs are the least significant Q0 bit of Q register, Q-1 bit and count bit from the down counter.


CONTROLLER
DATAPATH
clk
load
M
4
M
add
4
clk
sub
add/sub unit
Q–1
load
4
Q
4
clk
controller
shift
A
Q
Q–1
Q0
dc
count
clk
counter
1
1

Program – 9

Aim : Design the Control unit of a computer using either hardwiring or microprogrammoing based on its register transfer language description

Control unit

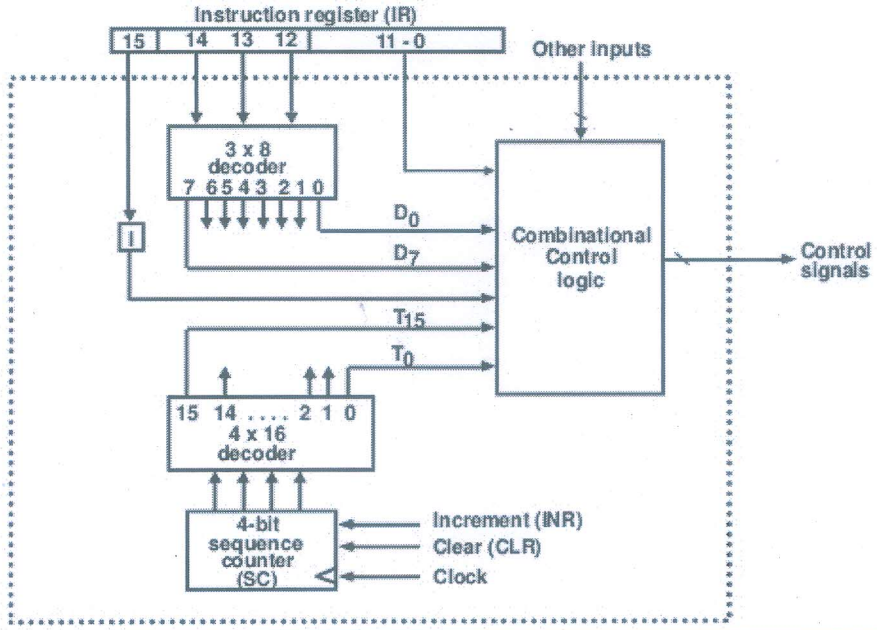
A processor is composed of datapath and control unit. Datapath of a processor is the execution unit such as ALU, shifter, registers and their interconnects. Control unit is considered to be the most complex part of a processor. Its function is to control various units in the datapath. Control unit realises the behaviour of a processor as specified by its micro-operations. The performance of control unit is crucial as it determines the clock cycle of the processor.

Control unit can be implemented by hardwired or by microprogram. A computer designer strives to optimise three aspects of control unit design:

1. the complexity (hence cost) of the control unit
2. the speed of control unit
3. the engineering cost of the design (time, correctness etc.)

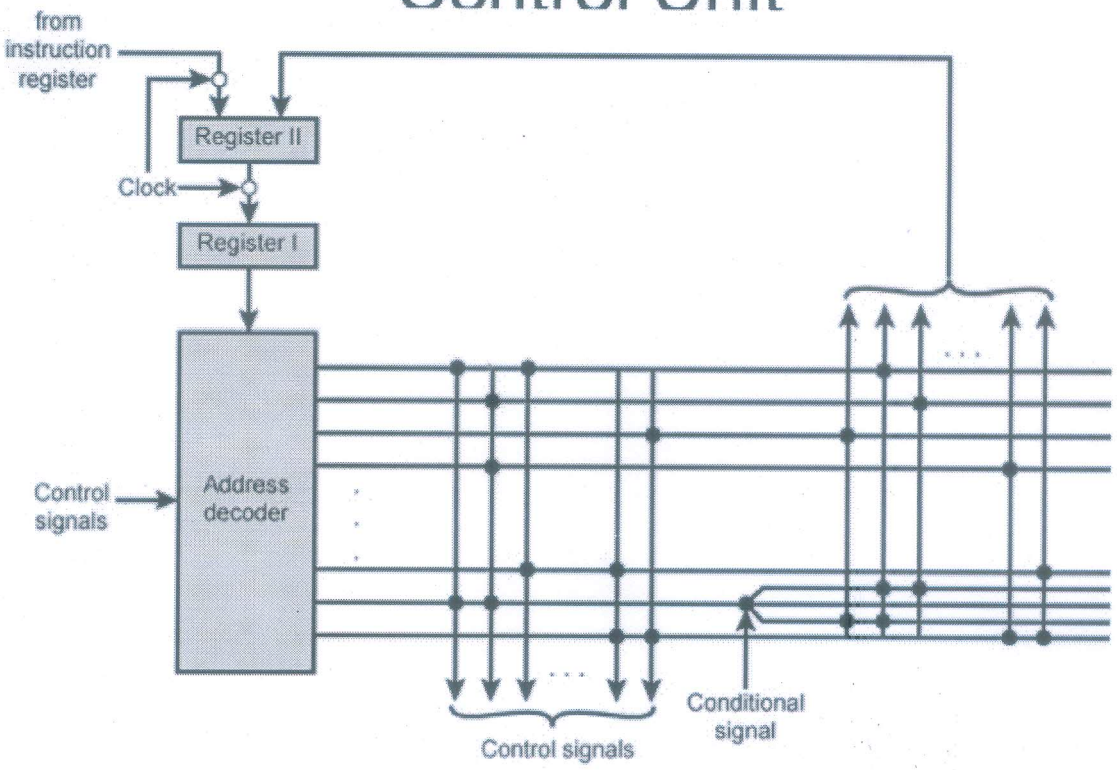
TIMING AND CONTROL

Control unit of Basic Computer



Control unit using microprogramming

Wilkes's Microprogrammed Control Unit



Hardwired control unit

In the past, hardwired control unit is very difficult to design hence its engineering cost is very high. Presently, the emphasis of computer design is the performance therefore hardwired design is the choice. Also the CAD tools for logic design have improved to the point that a complex design can be mostly automated. Therefore almost all processors of today use hardwired control unit.

Starting with a behavioural description of the control unit, the *state diagram* of micro-operations is constructed. Most states are simply driven by clock and only transition to the next state. Some states branch to different states depend on conditions such as testing conditional codes or decoding the instruction.

A Block diagram of the Basic Computer's Hard-wired Control unit

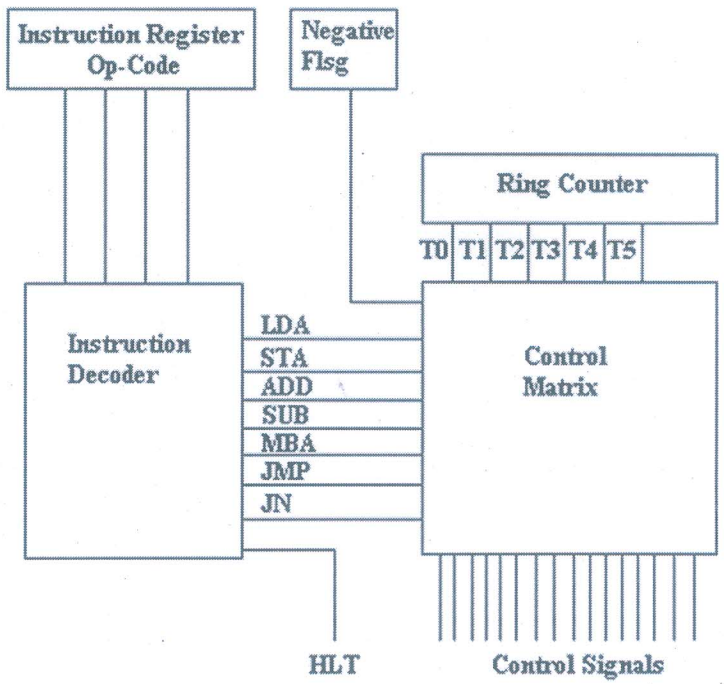


Figure 3. The Internal Organization of the Ring Counter

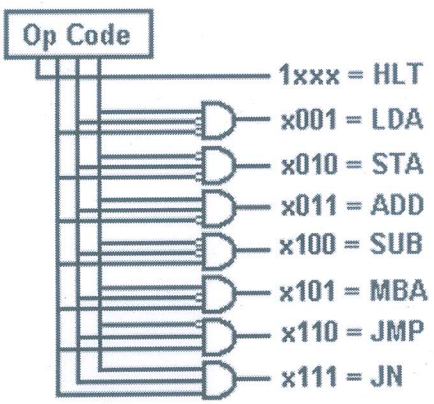


Figure 4. The Internal Organization of the Hard-wired Instruction Decoder

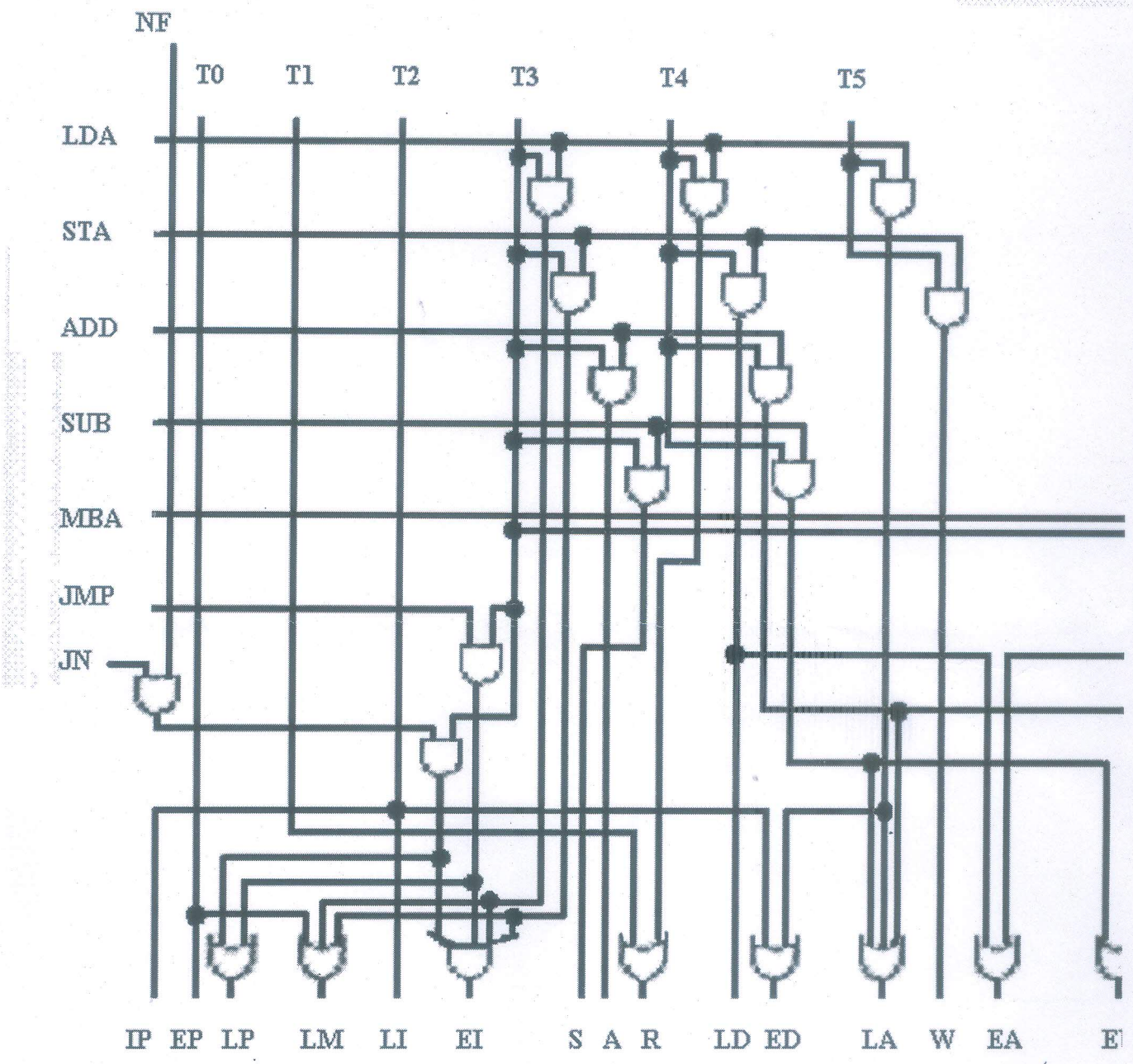


Figure 6. The logical equations required for each of the hardwired control signals on the basic computer. The machine's control matrix is designed from these equations.

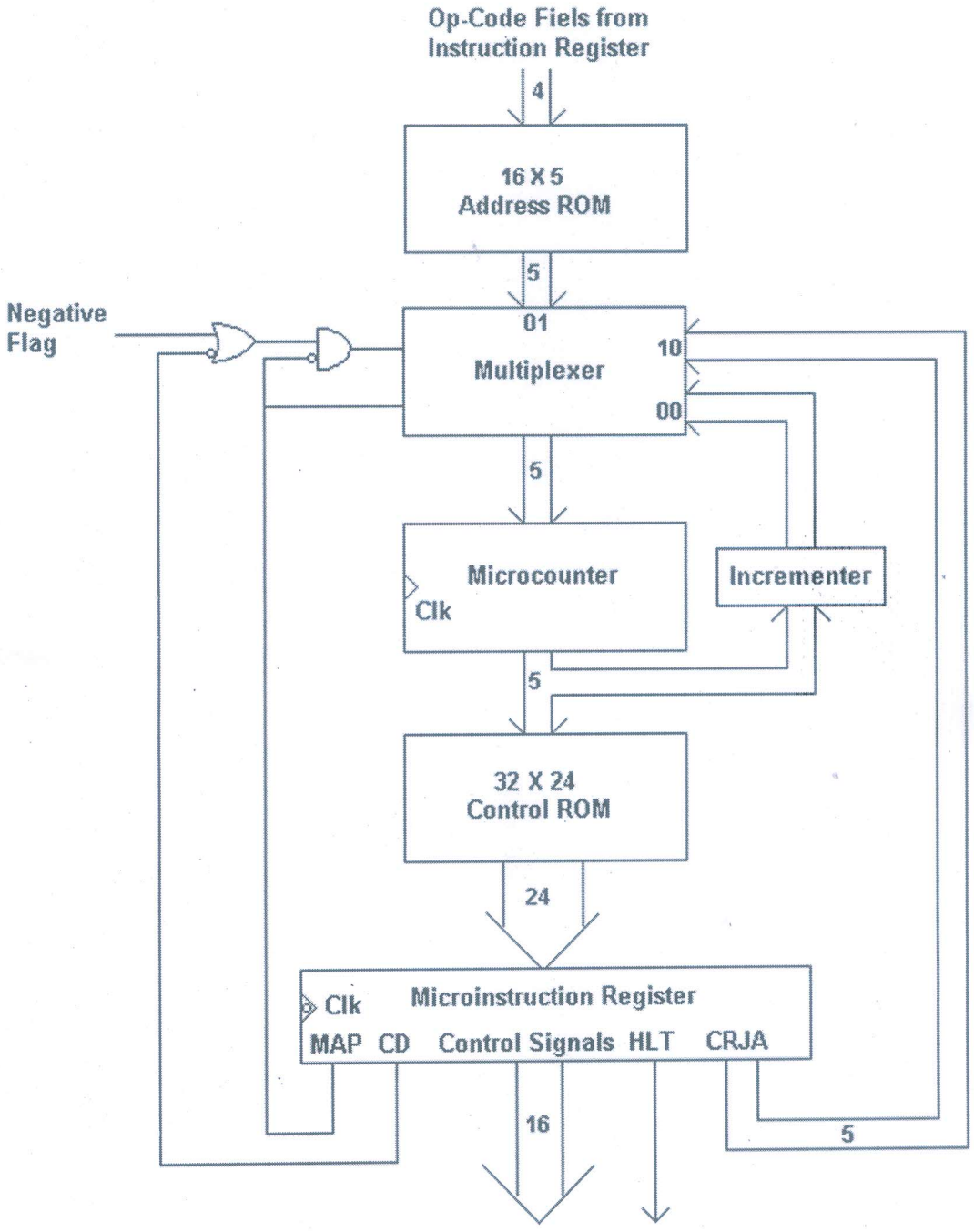


Figure 7. A Microprogrammed Control Unit for the Simple Computer

1	1	1	5
CD	MAP	HLT	CRJA

Figure 8. Next address field of the microinstruction register. CD is the condition bit, MAP causes the address of the next microinstruction to be obtained from the address ROM, HLT stops the clock, and CRJA is the control ROM jump address field.