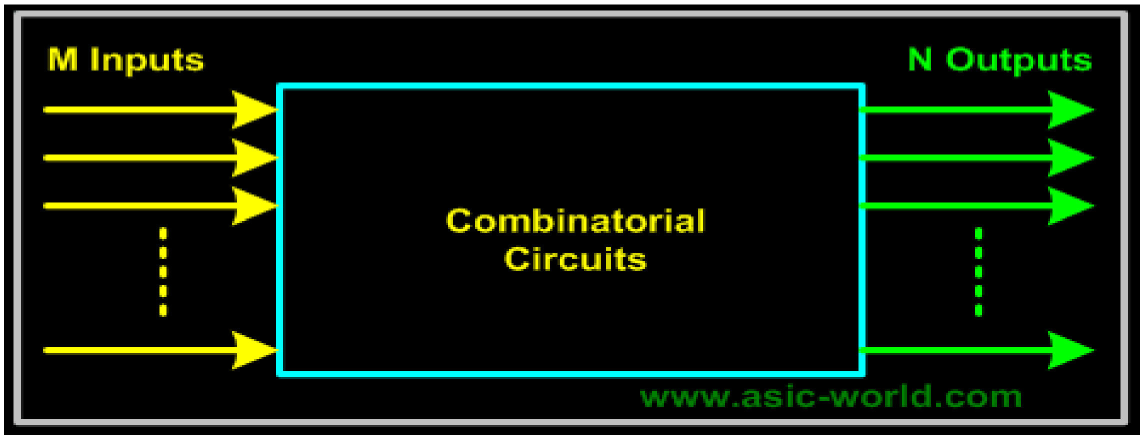
UNIT-III

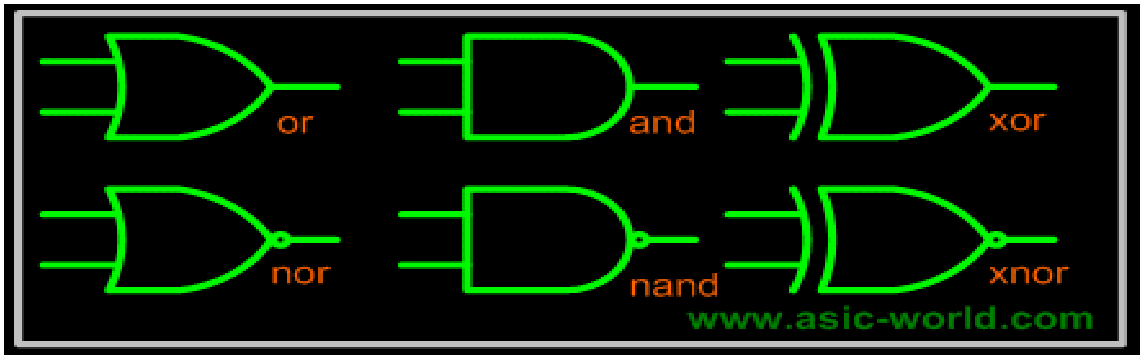
Combinational Logic

Introduction

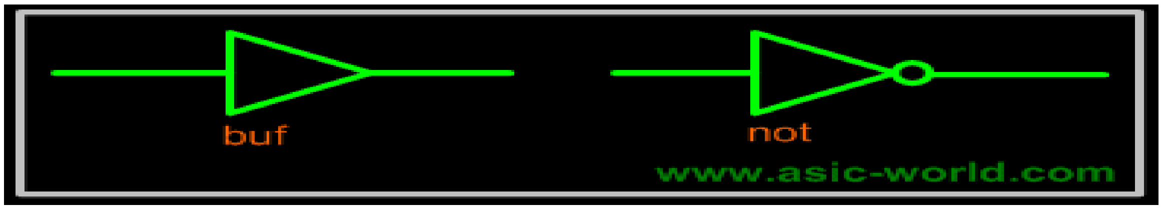
Combinatorial Circuits are circuits which can be considered to have the following generic structure.



Whenever the same set of inputs is fed in to a combinatorial circuit, the same outputs will be generated. Such circuits are said to be stateless. Some simple combinational logic elements that we have seen in previous sections are "Gates".



All the gates in the above figure have 2 inputs and one output; combinational elements simplest form are "not" gate and "buffer" as shown in the figure below. They have only one input and one output.

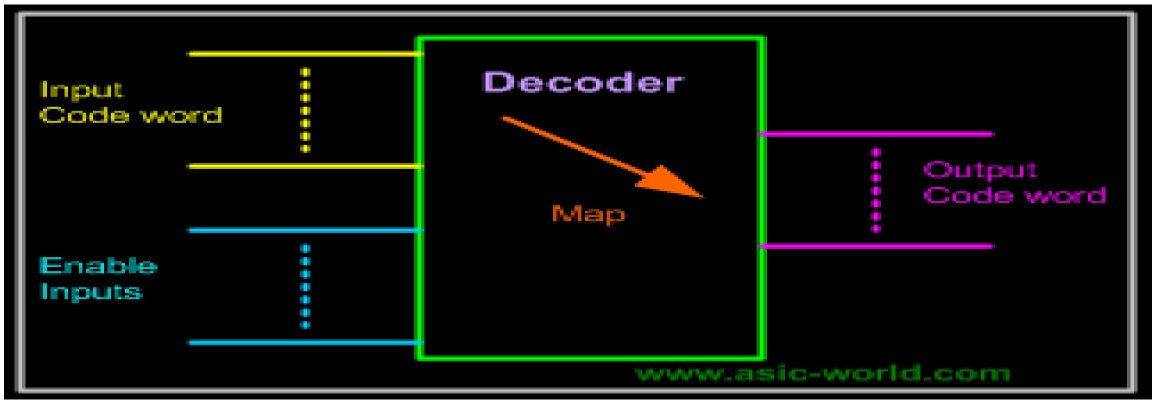


Decoders

A decoder is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different; e.g. n-to-2n, BCD decoders.

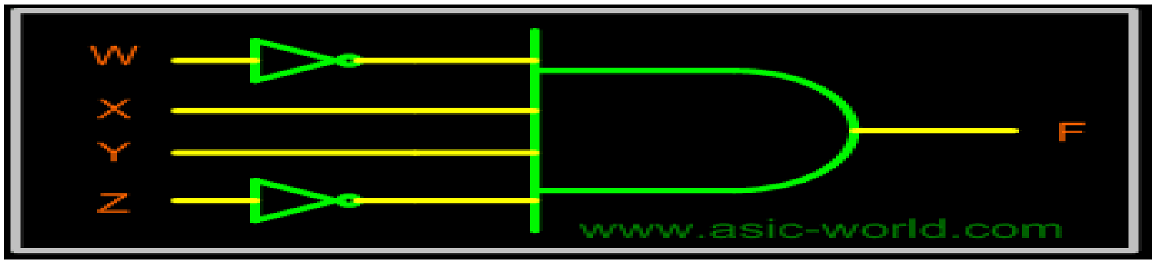
Enable inputs must be on for the decoder to function, otherwise its outputs assume a single "disabled" output code word.

Decoding is necessary in applications such as data multiplexing, 7 segment display and memory address decoding. Figure below shows the pseudo block of a decoder.



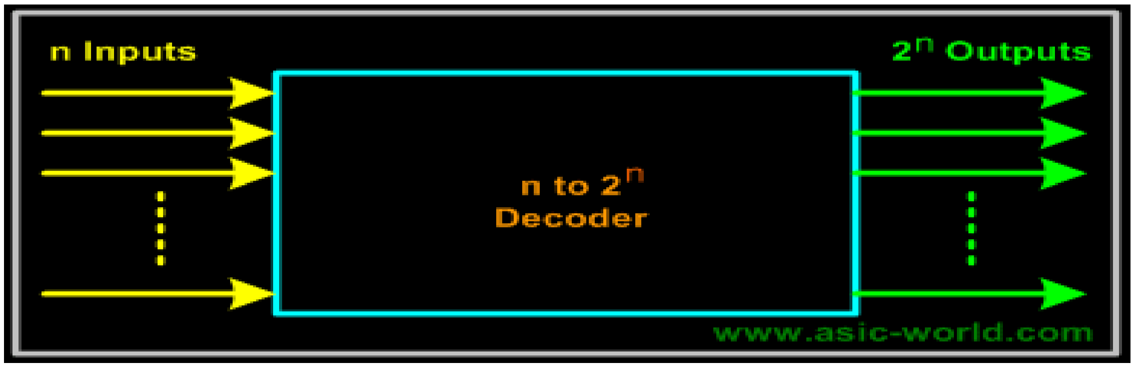
Basic Binary Decoder

AND gate can be used as the basic decoding element, because its output is HIGH only when all its inputs are HIGH. For example, if the input binary number is 0110, then, to make all the inputs to the AND gate HIGH, the two outer bits must be inverted using two inverters as shown in figure below.



Binary n-to-2^n Decoders

A binary decoder has n inputs and 2^n outputs. Only one output is active at any one time, corresponding to the input value. Figure below shows a representation of Binary n-to-2^n decoder.



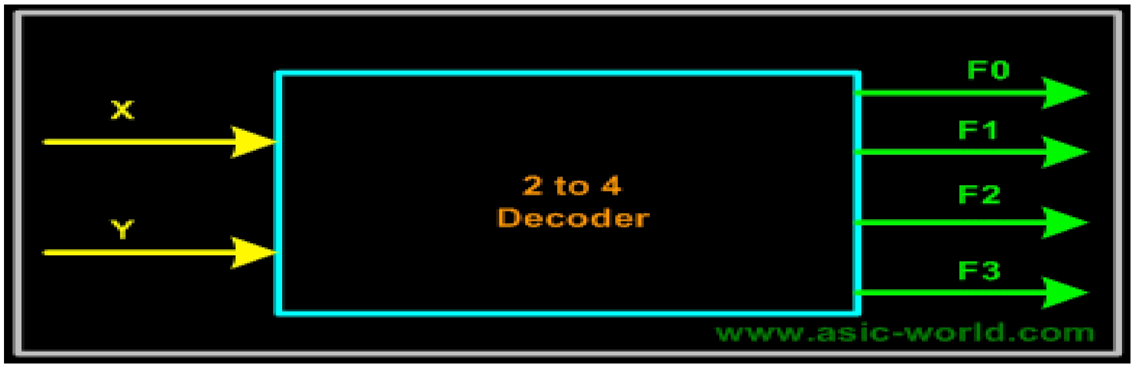
✦Example - 2-to-4 Binary Decoder

A 2 to 4 decoder consists of two inputs and four outputs, truth table and symbols of which is shown below.

Truth Table

X	Y	F0	F1	F2	F3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

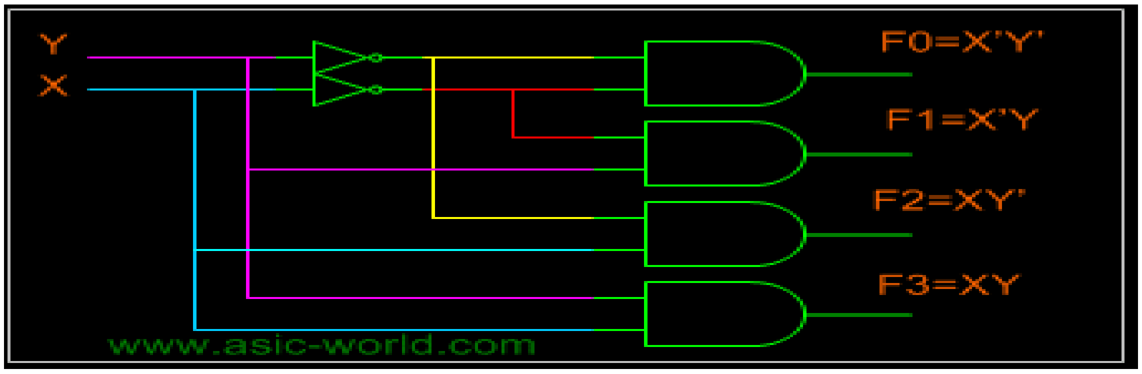
Symbol



To minimize the above truth table we may use kmap, but doing that you will realize that it is a waste of time. One can directly write down the function for each of the outputs. Thus we can draw the circuit as shown in figure below.

Note: Each output is a 2-variable minterm (X'Y', X'Y, XY', XY)

Circuit



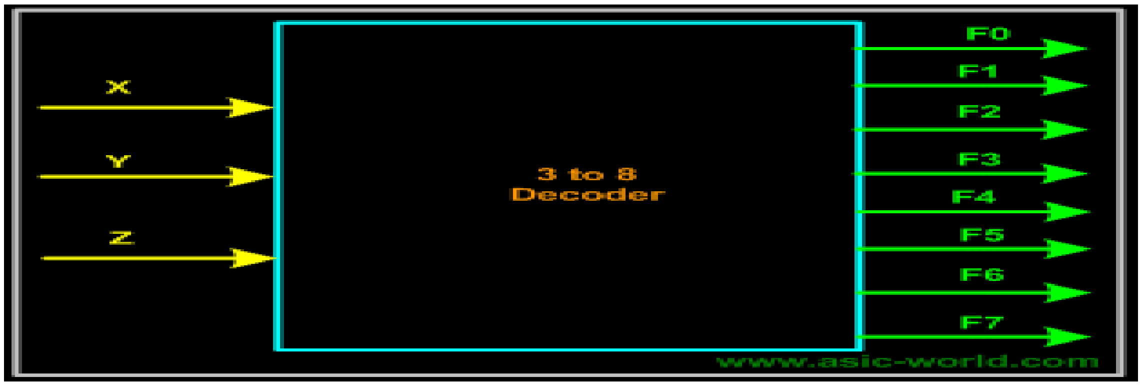
Example - 3-to-8 Binary Decoder

A 3 to 8 decoder consists of three inputs and eight outputs, truth table and symbols of which is shown below.

Truth Table

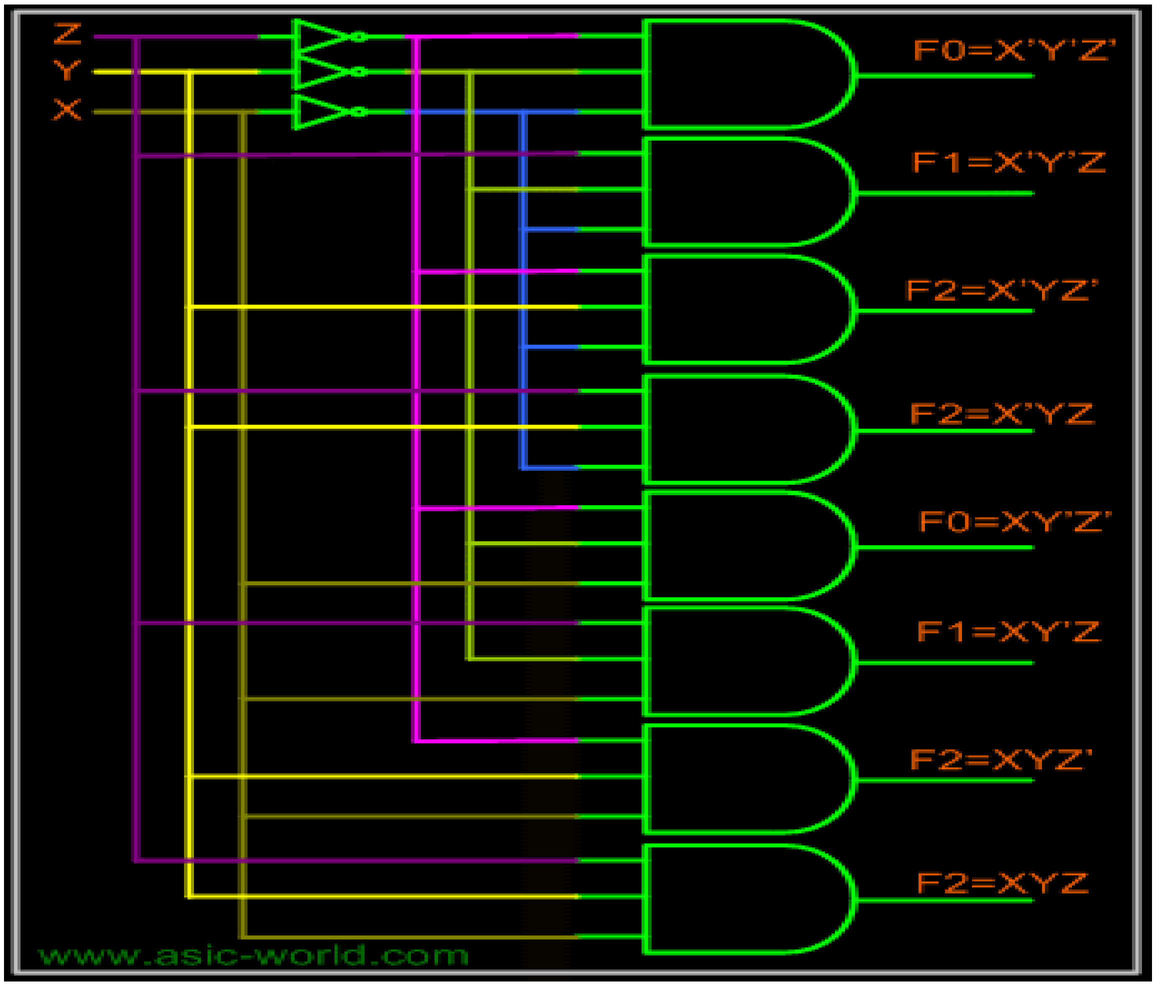
X	Y	Z	F0	F1	F2	F3	F4	F5	F6	F7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Symbol



From the truth table we can draw the circuit diagram as shown in figure below.

Circuit



Implementing Functions Using Decoders

- Any n-variable logic function, in canonical sum-of-minterms form can be implemented using a single n-to-2^n decoder to generate the minterms, and an OR gate to form the sum.
  - The output lines of the decoder corresponding to the minterms of the function are used as inputs to the or gate.
- Any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n decoder with m OR gates.
- Suitable when a circuit has many outputs, and each output function is expressed with few minterms.

✦Example - Full adder

Equation

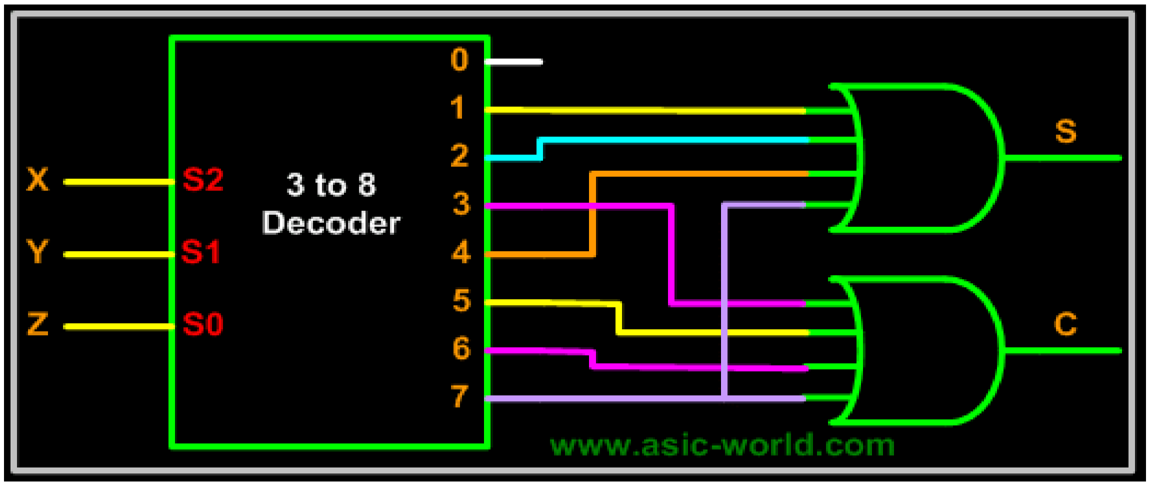
$S(x, y, z) = \Sigma(1,2,4,7)$
$C(x, y, z) = \Sigma(3,5,6,7)$

Truth Table

X	Y	Z	C	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 truth table we know the values for which the sum (s) is active and also the carry (c) is active. Thus we have the equation as shown above and a circuit can be drawn as shown below from the equation derived.

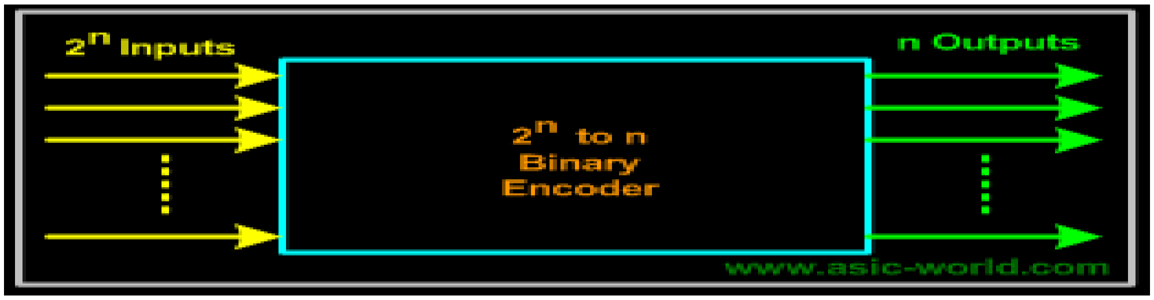
Circuit



Encoders

An encoder is a combinational circuit that performs the inverse operation of a decoder. If a device output code has fewer bits than the input code has, the device is usually called an encoder. e.g. 2^n-to-n, priority encoders.

The simplest encoder is a 2^n-to-n binary encoder, where it has only one of 2^n inputs = 1 and the output is the n-bit binary number corresponding to the active input.



Example - Octal-to-Binary Encoder

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

Truth Table

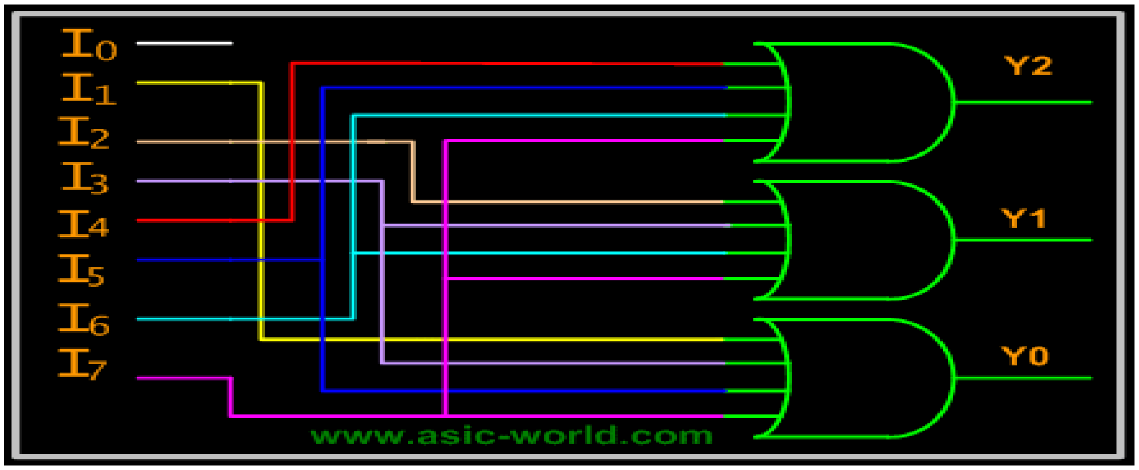
I0	I1	I2	I3	I4	I5	I6	I7	Y2	Y1	Y0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

For an 8-to-3 binary encoder with inputs I0-I7 the logic expressions of the outputs Y0-Y2 are:

Y0 = I1 + I3 + I5 + I7
Y1= I2 + I3 + I6 + I7
Y2 = I4 + I5 + I6 +I7

Based on the above equations, we can draw the circuit as shown below

Circuit



Example - Decimal-to-Binary Encoder

Decimal-to-Binary take 10 inputs and provides 4 outputs, thus doing the opposite of what the 4-to-10 decoder does. At any one time, only one input line has a value of 1. The figure below shows the truth table of a Decimal-to-binary encoder.

Truth Table

I0	I1	I2	I3	I4	I5	I6	I7	I8	I9	Y3	Y2	Y1	Y0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	0	1	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	0	0	1

From the above truth table , we can derive the functions Y3, Y2, Y1 and Y0 as given below.

Y3 = I8 + I9
Y2 = I4 + I5 + I6 + I7
Y1 = I2 + I3 + I6 + I7
Y0 = I1 + I3 + I5 + I7 + I9

Priority Encoder

If we look carefully at the Encoder circuits that we got, we see the following limitations. If more then two inputs are active simultaneously, the output is unpredictable or rather it is not what we expect it to be.

This ambiguity is resolved if priority is established so that only one input is encoded, no matter how many inputs are active at a given point of time.

The priority encoder includes a priority function. The operation of the priority encoder is such that if two or more inputs are active at the same time, the input having the highest priority will take precedence.

Example - 4to3 Priority Encoder

The truth table of a 4-input priority encoder is as shown below. The input D3 has the highest priority, D2 has next highest priority, D0 has the lowest priority. This means output Y2 and Y1 are 0 only when none of the inputs D1, D2, D3 are high and only D0 is high.

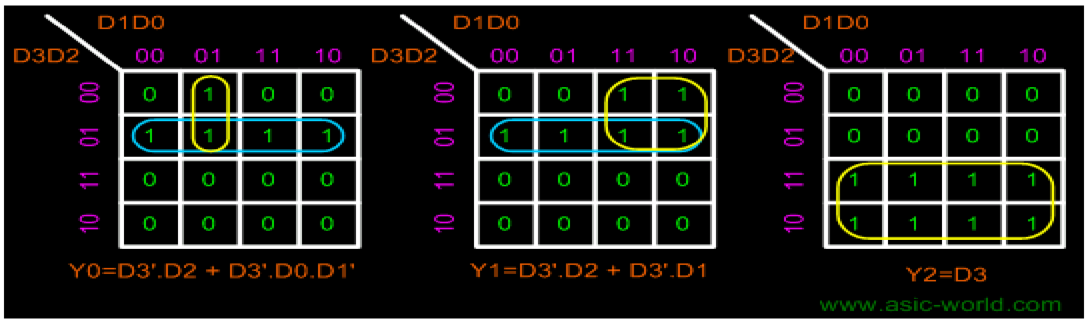
A 4 to 3 encoder consists of four inputs and three outputs, truth table and symbols of which is shown below.

Truth Table

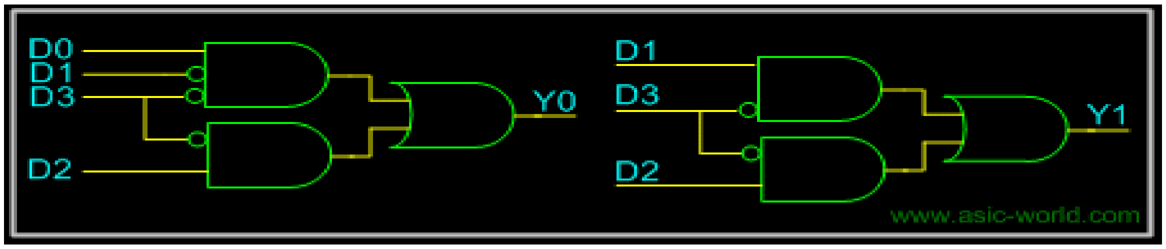
D3	D2	D1	D0	Y2	Y1	Y0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	x	0	1	0
0	1	x	x	0	1	1
1	x	x	x	1	0	0

Now that we have the truth table, we can draw the Kmaps as shown below.

Kmaps



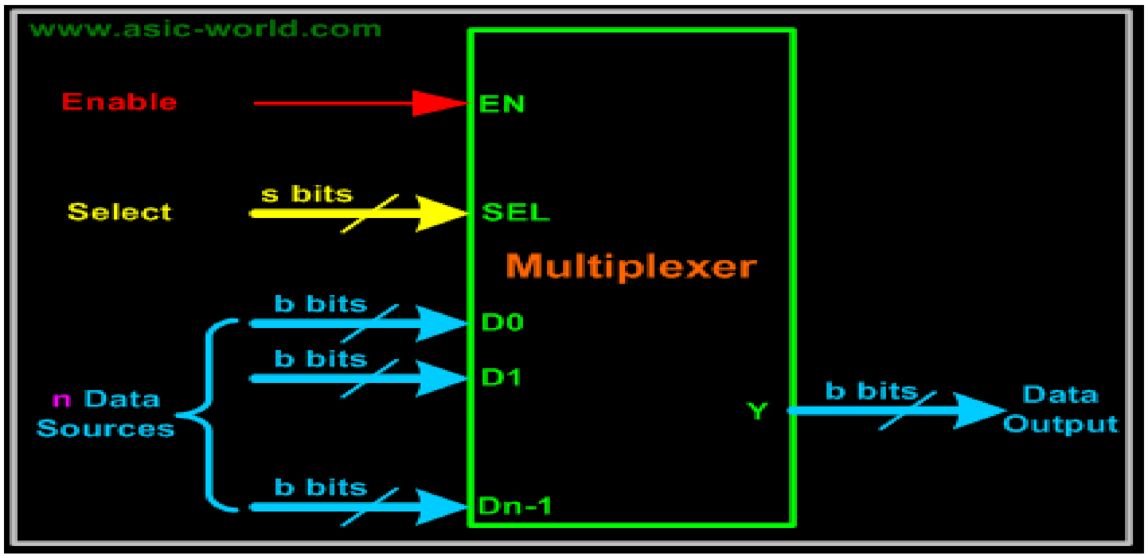
From the Kmap we can draw the circuit as shown below. For Y2, we connect directly to D3.



We can apply the same logic to get higher order priority encoders.

Multiplexer

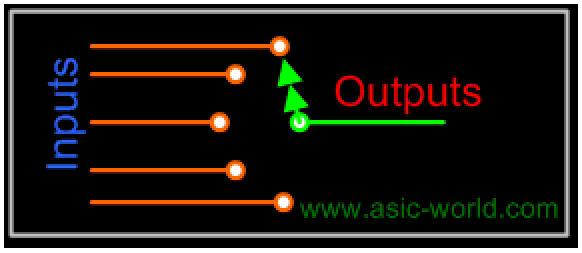
A multiplexer (MUX) is a digital switch which connects data from one of n sources to the output. A number of select inputs determine which data source is connected to the output. The block diagram of MUX with n data sources of b bits wide and s bits wide select line is shown in below figure.



MUX acts like a digitally controlled multi-position switch where the binary code applied to the select inputs controls the input source that will be switched on to the output as shown in the figure below. At any given point of time only one input gets selected and is connected to output, based on the select input signal.

Mechanical Equivalent of a Multiplexer

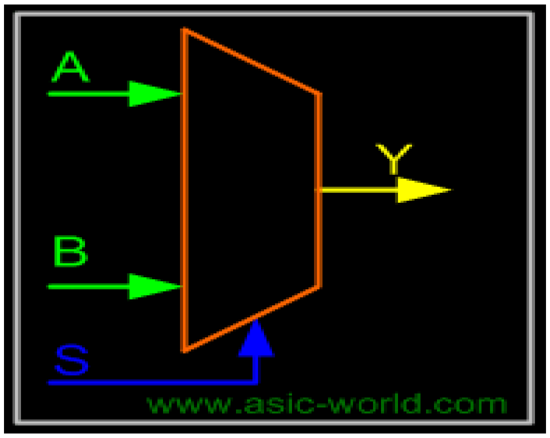
The operation of a multiplexer can be better explained using a mechanical switch as shown in the figure below. This rotary switch can touch any of the inputs, which is connected to the output. As you can see at any given point of time only one input gets transferred to output.



Example - 2x1 MUX

A 2 to 1 line multiplexer is shown in figure below, each 2 input lines A to B is applied to one input of an AND gate. Selection lines S are decoded to select a particular AND gate. The truth table for the 2:1 mux is given in the table below.

Symbol



Truth Table

S	Y
0	A
1	B

Design of a 2:1 Mux

To derive the gate level implementation of 2:1 mux we need to have truth table as shown in figure. And once we have the truth table, we can draw the K-map as shown in figure for all the cases when Y is equal to '1'.

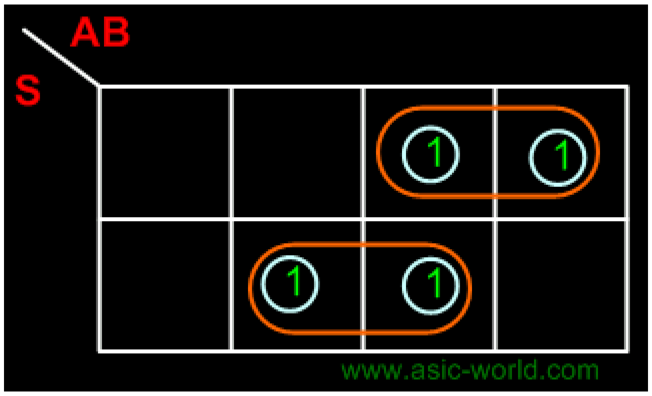
Combining the two 1' as shown in figure, we can drive the output y as shown below

Y = A.S' + B.S

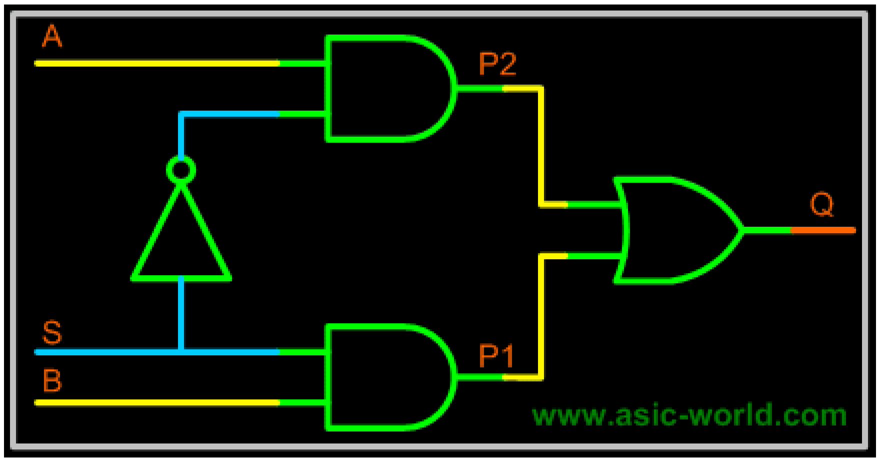
Truth Table

B	A	S	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Kmap



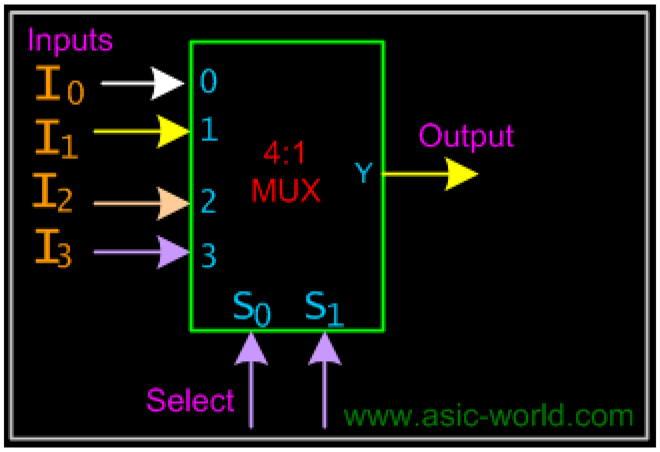
Circuit



Example : 4:1 MUX

A 4 to 1 line multiplexer is shown in figure below, each of 4 input lines I0 to I3 is applied to one input of an AND gate. Selection lines S0 and S1 are decoded to select a particular AND gate. The truth table for the 4:1 mux is given in the table below.

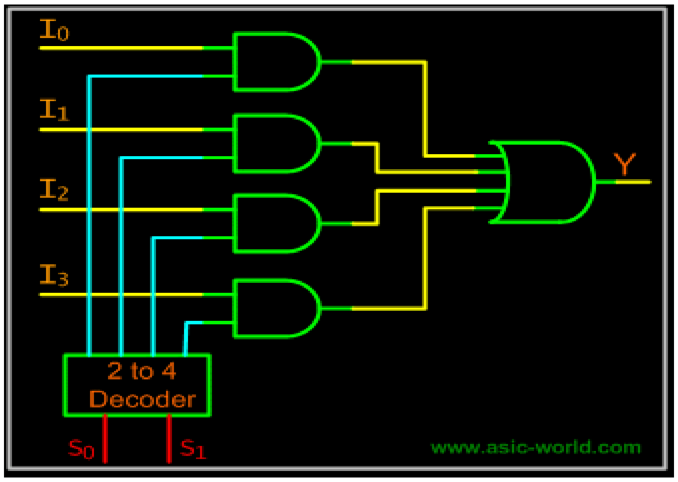
Symbol



Truth Table

S1	S0	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Circuit



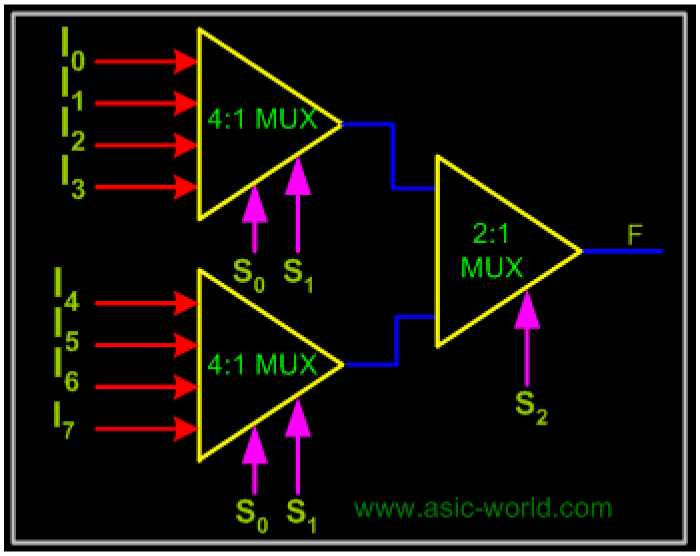
Larger multiplexers can be constructed from smaller ones. An 8-to-1 multiplexer can be constructed from smaller multiplexers as shown below.

Example - 8-to-1 multiplexer from Smaller MUX

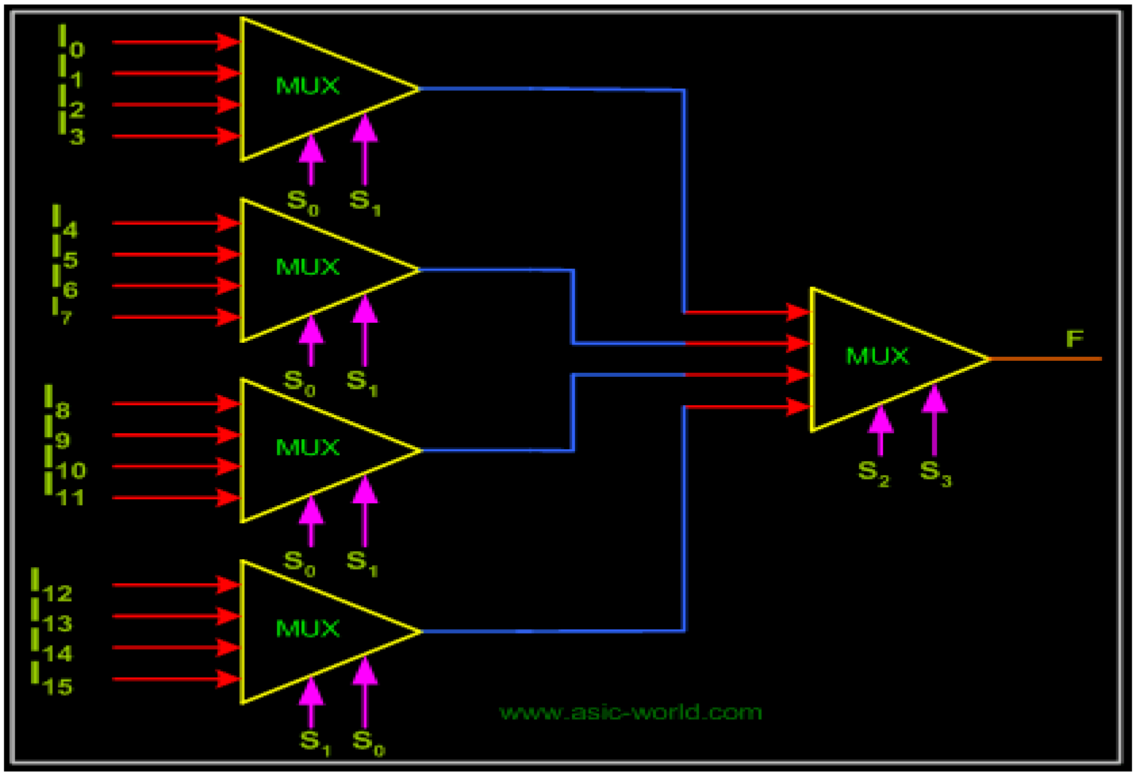
Truth Table

S2	S1	S0	F
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

Circuit



✦Example - 16-to-1 multiplexer from 4:1 mux

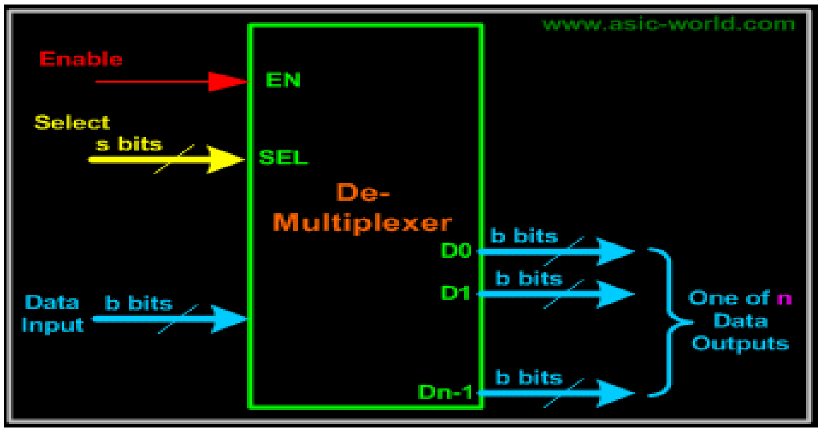


De-multiplexers

They are digital switches which connect data from one input source to one of n outputs.

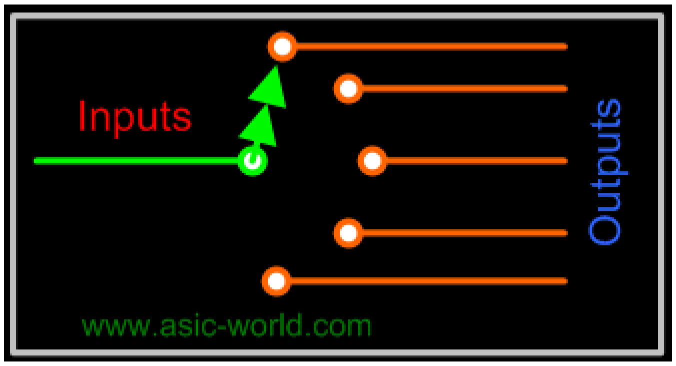
Usually implemented by using n-to-2^n binary decoders where the decoder enable line is used for data input of the de-multiplexer.

The figure below shows a de-multiplexer block diagram which has got s-bits-wide select input, one b-bits-wide data input and n b-bits-wide outputs.

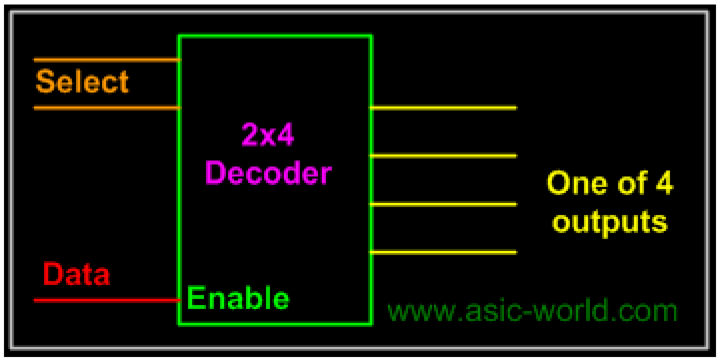


Mechanical Equivalent of a De-Multiplexer

The operation of a de-multiplexer can be better explained using a mechanical switch as shown in the figure below. This rotary switch can touch any of the outputs, which is connected to the input. As you can see at any given point of time only one output gets connected to input.

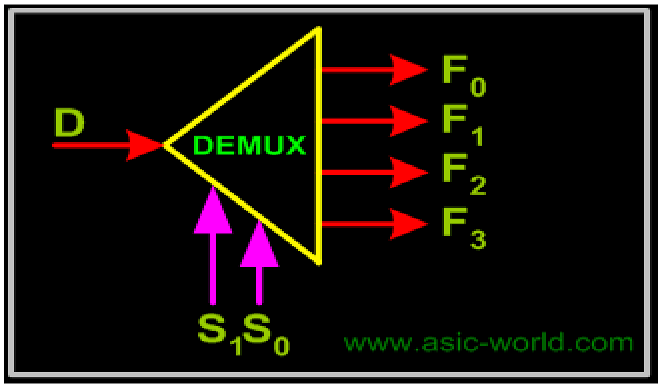


1-bit 4-output de-multiplexer using a 2x4 binary decoder.



Example: 1-to-4 De-multiplexer

Symbol



Truth Table

S1	S0	F0	F1	F2	F3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

Boolean Function Implementation

Earlier we had seen that it is possible to implement Boolean functions using decoders. In the same way it is also possible to implement Boolean functions using muxers and de-muxers.

Implementing Functions Multiplexers

Any n-variable logic function can be implemented using a smaller 2^{n-1}-to-1 multiplexer and a single inverter (e.g 4-to-1 mux to implement 3 variable functions) as follows.

Express function in canonical sum-of-minterms form. Choose n-1 variables as inputs to mux select lines. Construct the truth table for the function, but grouping inputs by selection line values (i.e select lines as most significant inputs).

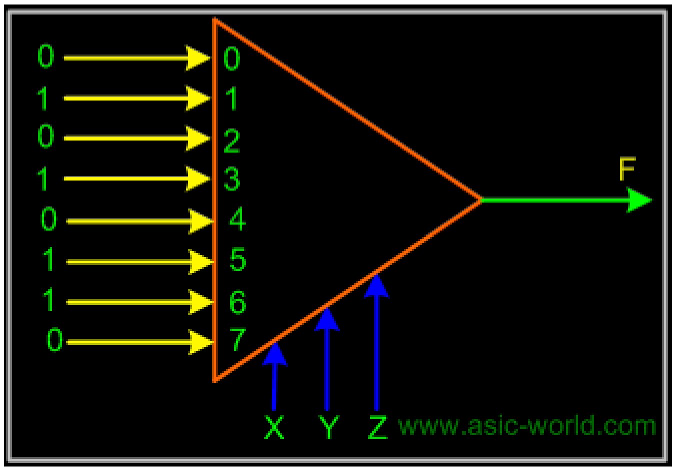
Determine multiplexer input line i values by comparing the remaining input variable and the function F for the corresponding selection lines value i.

We have four possible mux input line i values:

- Connect to 0 if the function is 0 for both values of remaining variable.
- Connect to 1 if the function is 1 for both values of remaining variable.
- Connect to remaining variable if function is equal to the remaining variable.
- Connect to the inverted remaining variable if the function is equal to the remaining variable inverted.

Example: 3-variable Function Using 8-to-1 mux

Implement the function F(X,Y,Z) = S(1,3,5,6) using an 8-to-1 mux. Connect the input variables X, Y, Z to mux select lines. Mux data input lines 1, 3, 5, 6 that correspond to the function minterms are connected to 1. The remaining mux data input lines 0, 2, 4, 7 are connected to 0.



Example: 3-variable Function Using 4-to-1 mux

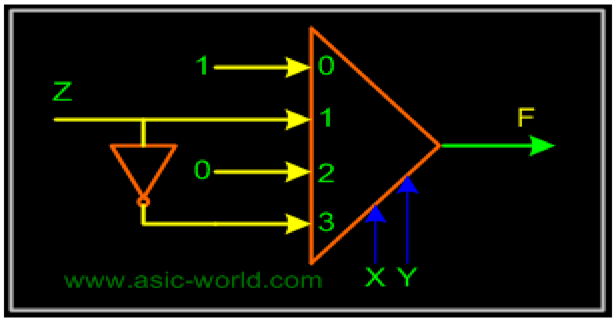
Implement the function F(X,Y,Z) = S(0,1,3,6) using a single 4-to-1 mux and an inverter. We choose the two most significant inputs X, Y as mux select lines.

Construct truth table.

Truth Table

Select i	X	Y	Z	F	Mux Input i
0	0	0	0	1	1
0	0	0	1	1	1
1	0	1	0	0	Z
1	0	1	1	1	Z
2	1	0	0	0	0
2	1	0	1	0	0
3	1	1	0	1	Z'
3	1	1	1	0	Z'

Circuit



We determine multiplexer input line i values by comparing the remaining input variable Z and the function F for the corresponding selection lines value i

- when XY=00 the function F is 1 (for both Z=0, Z=1) thus mux input0 = 1
- when XY=01 the function F is Z thus mux input1 = Z
- when XY=10 the function F is 0 (for both Z=0, Z=1) thus mux input2 = 0
- when XY=11 the function F is Z' thus mux input3 = Z'

1) **Half Adder**

```
entity HALF_ADDER is
        port (A, B: in BIT; SUM, CARRY: out BIT);
   end HALF_ADDER;
architecture HA_STRUCTURE of HALF_ADDER is
                component XOR2
                port (X, Y: in BIT; Z: out BIT);
                end component;
                component AND2
                port (L, M: in BIT; N: out BIT);
                end component;
                begin
X1: XOR2 port map (A, B, SUM);
A1: AND2 port map (A, B, CARRY);
end HA_STRUCTURE;
```

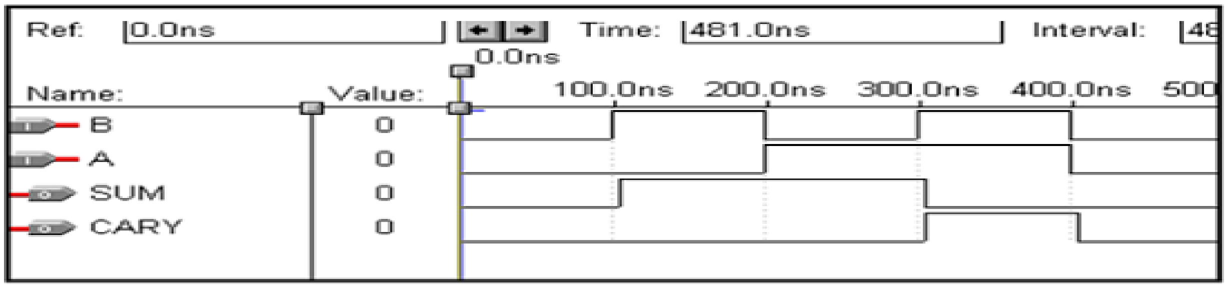
2) **Decoder**

```
entity DECODER2x4 is
port (A, B, ENABLE: in SIT: Z: out BIT_VECTOR(0 to 3));
end DECODER2x4;

architecture DEC_STR of DECODER2x4 is

                component INV
                port (A: in BIT; Z: out BIT);
                end component;
                component NAND3
                port (A, B, C: in BIT; Z: out BIT);
                end component;
                signal ABAR, BBAR: BIT;
        begin
                I0: INV port map (A, ABAR);
                I1: INV port map (B, BBAR);
                N0: NAND3 port map (ABAR, BBAR, ENABLE, Z(0));
                N1: NAND3 port map (ABAR, B, ENABLE, Z(1));
                N2: NAND3 port map (A, BBAR, ENABLE, Z(2));
                N3: NAND3 port map (A, B, ENABLE, Z(3));
end DEC_STR;
```

3) Half adder Dataflow

```
Library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
   port(a,b:in bit; sum,carry:out bit);
end half_adder;
architecture data of half_adder is
begin
   sum<= a xor b;
   carry <= a and b;
end data;
```

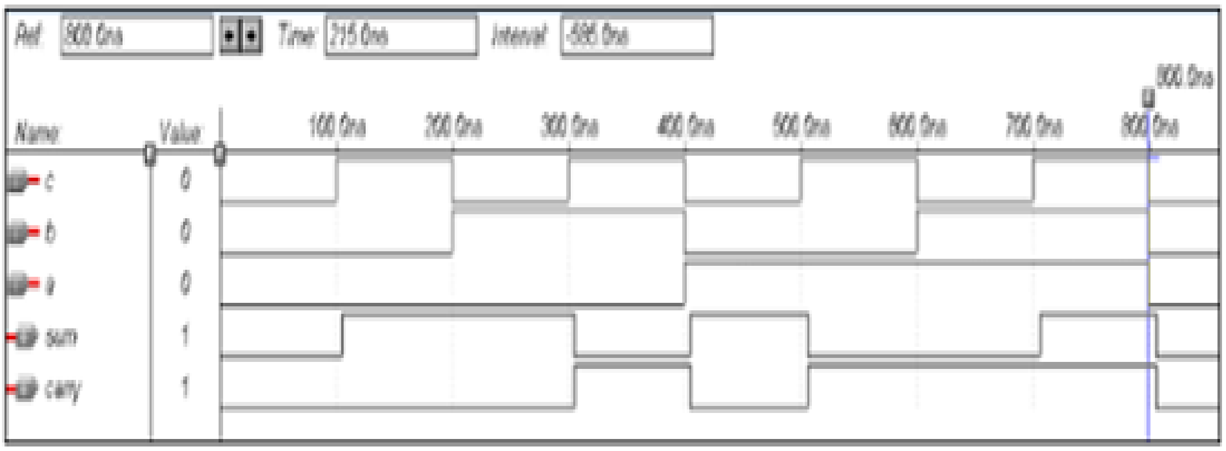
Waveforms



4) VHDL Code for a Full Adder

```
Library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
port(a,b,c:in bit; sum,carry:out bit);
end full_adder;
architecture data of full_adder is
begin
   sum<= a xor b xor c;
   carry <= ((a and b) or (b and c) or (a and c));
end data;
```

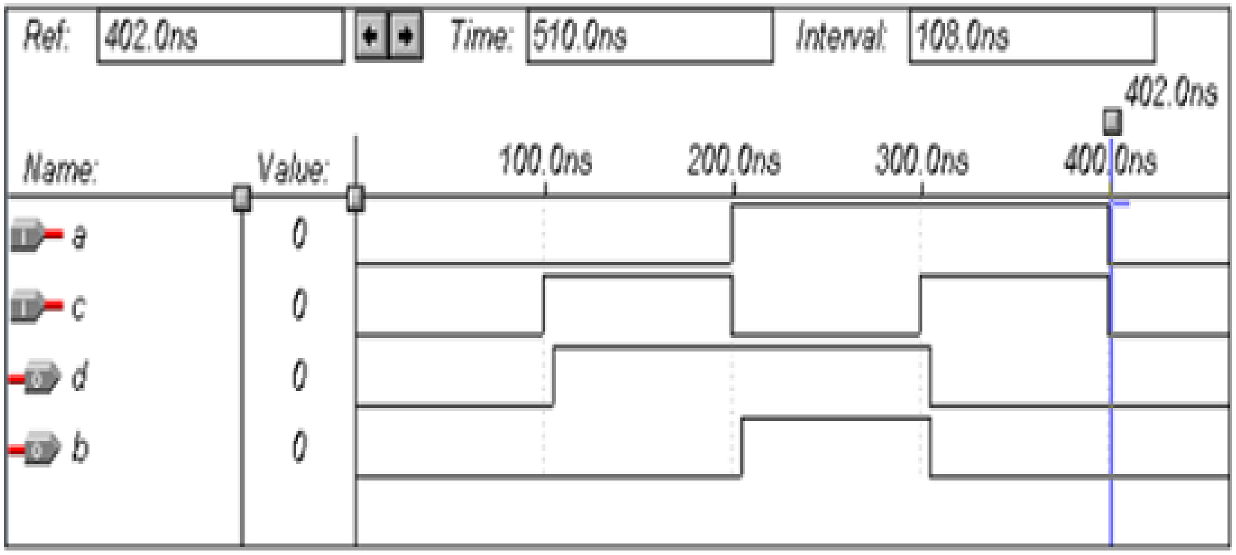
Waveforms



5) Half-Subtractor

```
Library ieee;
use ieee.std_logic_1164.all;
  entity half_sub is
   port(a,c:in bit; d,b:out bit);
end half_sub;
architecture data of half_sub is
begin
   d<= a xor c;
   b<= (a and (not c));
end data;
```

Waveforms

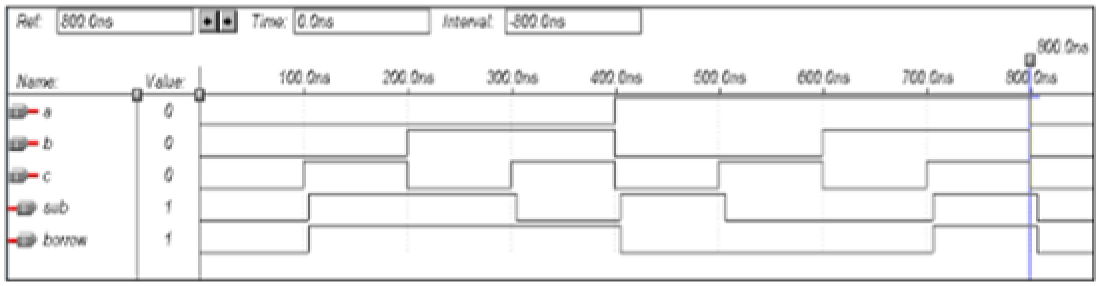


6) VHDL Code for a Full Subtractor

```
Library ieee;
```

```
use ieee.std_logic_1164.all;
entity full_sub is
   port(a,b,c:in bit; sub,borrow:out bit);
end full_sub;
architecture data of full_sub is
begin
   sub<= a xor b xor c;
   borrow <= ((b xor c) and (not a)) or (b and c);
end data;
```

Waveforms

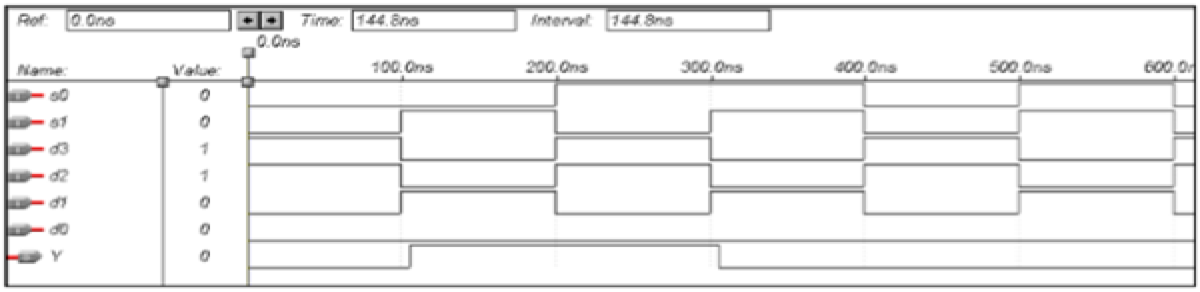


7) VHDL Code for a Multiplexer

```
Library ieee;
```

```
use ieee.std_logic_1164.all;
  entity mux is
   port(S1,S0,D0,D1,D2,D3:in bit; Y:out bit);
end mux;
  architecture data of mux is
begin
   Y<= (not S0 and not S1 and D0) or
      (S0 and not S1 and D1) or
      (not S0 and S1 and D2) or
      (S0 and S1 and D3);
end data;
```

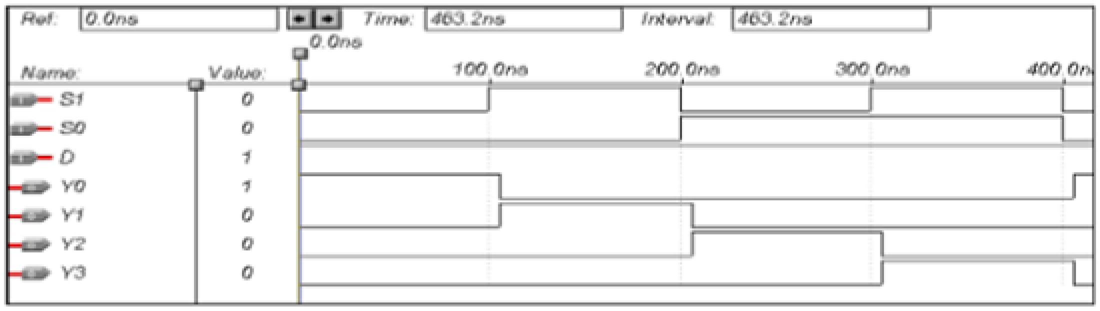
Waveforms



```
8) vhdl code for Demux
Library ieee;
use ieee.std_logic_1164.all;
```

```
  entity demux is
   port(S1,S0,D:in bit; Y0,Y1,Y2,Y3:out bit);
end demux;
  architecture data of demux is
begin
   Y0<=  ((Not S0) and (Not S1) and D);
   Y1<=  ((Not S0) and S1 and D);
   Y2<=  (S0 and (Not S1) and D);
   Y3<=  (S0 and S1 and D);
end data;
```

Waveforms

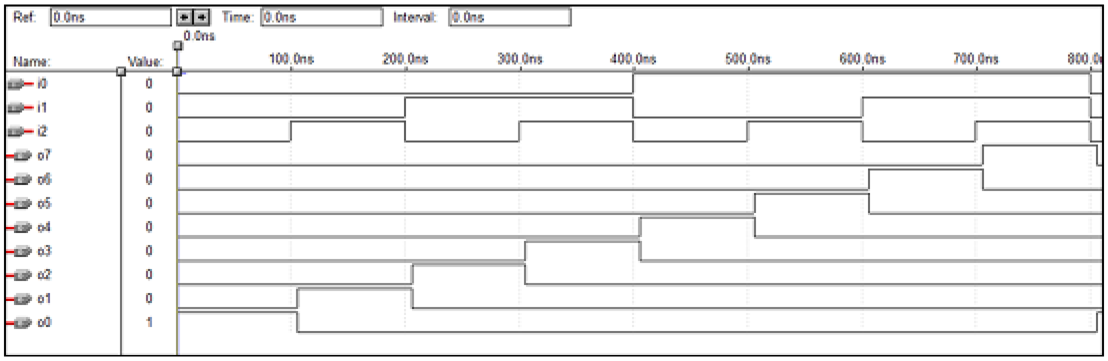


9)VHDL Code for a 8 x 3 Encoder

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity enc is
   port(i0,i1,i2,i3,i4,i5,i6,i7:in bit; o0,o1,o2: out bit);
end enc;

architecture vcgandhi of enc is
begin
   o0<=i4 or i5 or i6 or i7;
   o1<=i2 or i3 or i6 or i7;
   o2<=i1 or i3 or i5 or i7;
end vcgandhi;
```

Waveforms

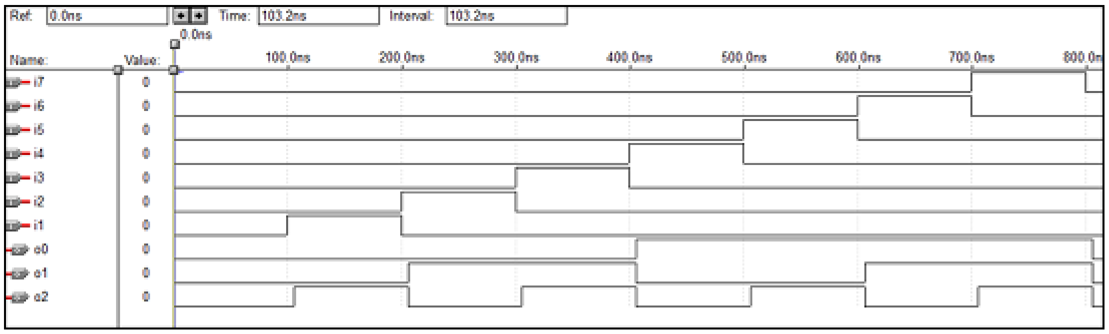


VHDL Code for a 3 x 8 Decoder

```
library ieee;
use ieee.std_logic_1164.all;

entity dec is
```

```
  port(i0,i1,i2:in bit; o0,o1,o2,o3,o4,o5,o6,o7: out bit);
end dec;

architecture vcgandhi of dec is
begin
  o0<=(not i0) and (not i1) and (not i2);
  o1<=(not i0) and (not i1) and i2;
  o2<=(not i0) and i1 and (not i2);
  o3<=(not i0) and i1 and i2;
  o4<=i0 and (not i1) and (not i2);
  o5<=i0 and (not i1) and i2;
  o6<=i0 and i1 and (not i2);
  o7<=i0 and i1 and i2;
end vcgandhi;
```

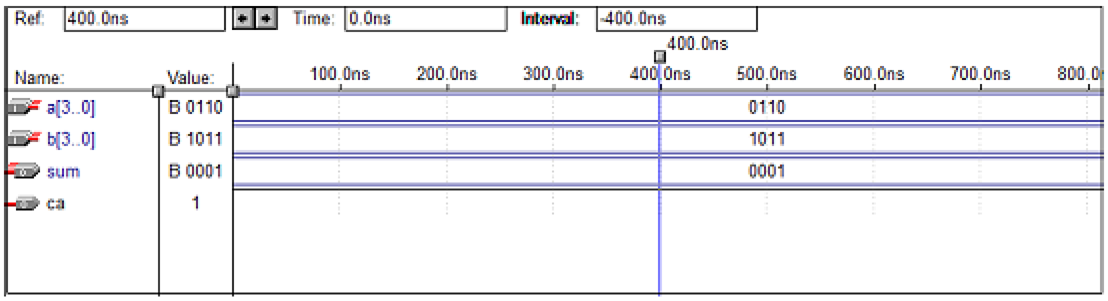
Waveforms



VHDL Code – 4 bit Parallel adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity pa is
  port(a : in STD_LOGIC_VECTOR(3 downto 0);
    b : in STD_LOGIC_VECTOR(3 downto 0);
    ca : out STD_LOGIC;
    sum : out STD_LOGIC_VECTOR(3 downto 0)
  );
end pa;

architecture vcgandhi of pa is
  Component fa is
    port (a : in STD_LOGIC;
      b : in STD_LOGIC;
      c : in STD_LOGIC;
      sum : out STD_LOGIC;
      ca : out STD_LOGIC
    );
  end component;
  signal s : std_logic_vector (2 downto 0);
  signal temp: std_logic;
begin
  temp<='0';
  u0 : fa port map (a(0),b(0),temp,sum(0),s(0));
  u1 : fa port map (a(1),b(1),s(0),sum(1),s(1));
  u2 : fa port map (a(2),b(2),s(1),sum(2),s(2));
  ue : fa port map (a(3),b(3),s(2),sum(3),ca);
end vcgandhi;
```

Waveforms



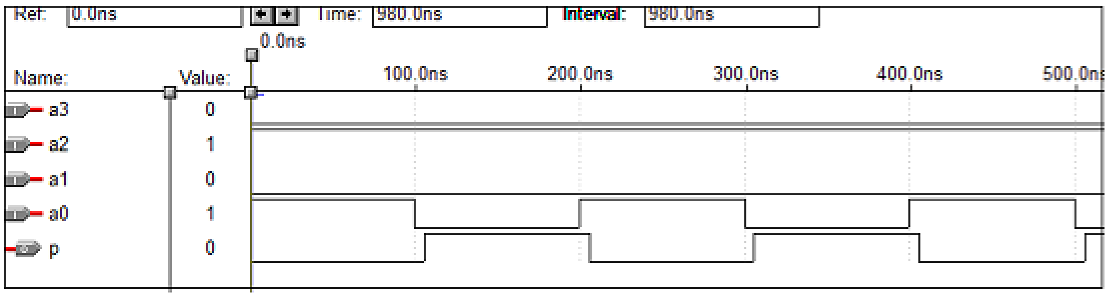
```
Ref: 400.0ns    Time: 0.0ns    Interval: -400.0ns

Name:     Value:
a[3..0]   B 0110    0110
b[3..0]   B 1011    1011
sum       B 0001    0001
ca        1
```

VHDL Code – 4 bit Parity Checker

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity parity_checker is
   port (a0,a1,a2,a3 : in std_logic;
      p : out std_logic);
end parity_checker;

architecture vcgandhi of parity_checker is
begin
   p <= (((a0 xor a1) xor a2) xor a3);
end vcgandhi;
```

Waveforms

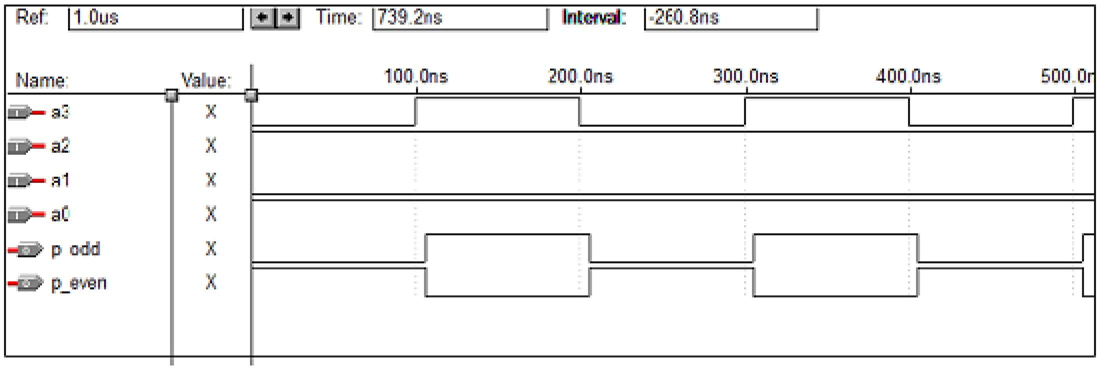


VHDL Code – 4 bit Parity Generator

```
library ieee;
use ieee.std_logic_1164.all;

entity paritygen is
   port (a0, a1, a2, a3: in std_logic; p_odd, p_even: out std_logic);
end paritygen;

architecture vcgandhi of paritygen is
begin
   process (a0, a1, a2, a3)

   if (a0 ='0' and a1 ='0' and a2 ='0' and a3 ='0')
      then odd_out <= "0";
```

```
    even_out <= "0";
  else
    p_odd <= (((a0 xor a1) xor a2) xor a3);
    p_even <= not(((a0 xor a1) xor a2) xor a3);
end vcgandhi
```

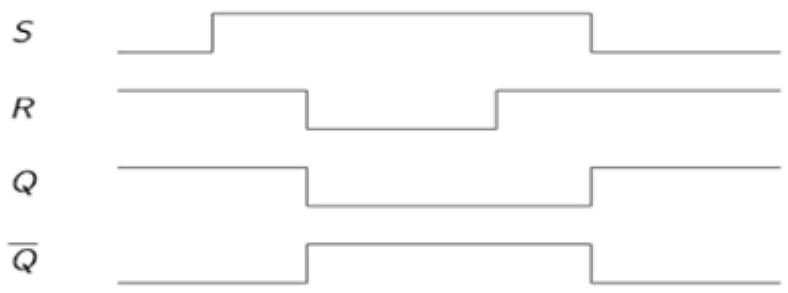
Waveforms



VHDL Code for an SR Latch

```
library ieee;
use ieee.std_logic_1164.all;

entity srl is
  port(r,s:in bit; q,qbar:buffer bit);
end srl;

architecture virat of srl is
  signal s1,r1:bit;
begin
  q<= s nand qbar;
  qbar<= r nand q;
end virat;
```

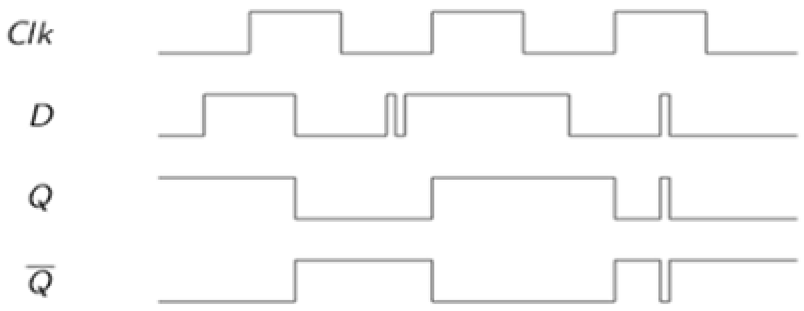
Waveforms



VHDL Code for a D Latch

```
library ieee;
use ieee.std_logic_1164.all;

entity Dl is
   port(d:in bit; q,qbar:buffer bit);
end Dl;

architecture virat of Dl is
   signal s1,r1:bit;
begin
   q<= d nand qbar;
   qbar<= d nand q;
end virat;
```

Waveforms

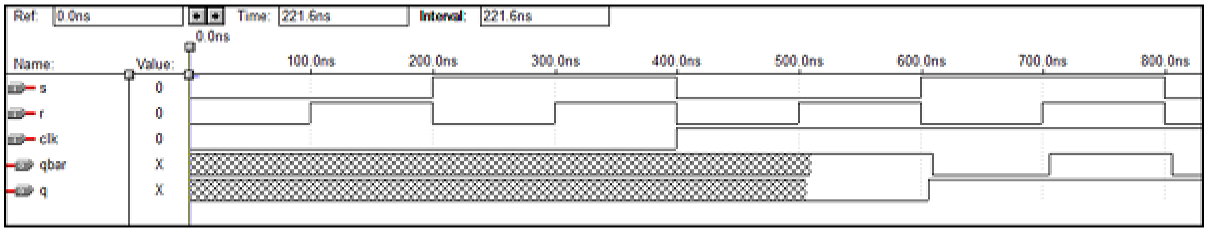


VHDL Code for an SR Flip Flop

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity srflip is
   port(r,s,clk:in bit; q,qbar:buffer bit);
end srflip;

architecture virat of srflip is
   signal s1,r1:bit;
begin
   s1<=s nand clk;
   r1<=r nand clk;
   q<= s1 nand qbar;
   qbar<= r1 nand q;
end virat;
```

Waveforms

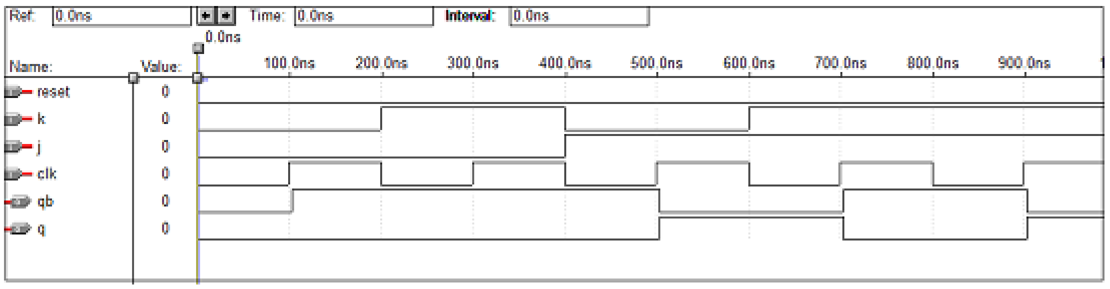


VHDL code for a JK Flip Flop

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity jk is
   port(
      j : in STD_LOGIC;
      k : in STD_LOGIC;
      clk : in STD_LOGIC;
      reset : in STD_LOGIC;
      q : out STD_LOGIC;
      qb : out STD_LOGIC
   );
```

```
end jk;

architecture virat of jk is
begin
  jkff : process (j,k,clk,reset) is
  variable m : std_logic := '0';

  begin
    if (reset = '1') then
      m : = '0';
    elsif (rising_edge (clk)) then
      if (j/ = k) then
        m : = j;
      elsif (j = '1' and k = '1') then
        m : = not m;
      end if;
    end if;

    q <= m;
    qb <= not m;
  end process jkff;
end virat;
```

Waveforms

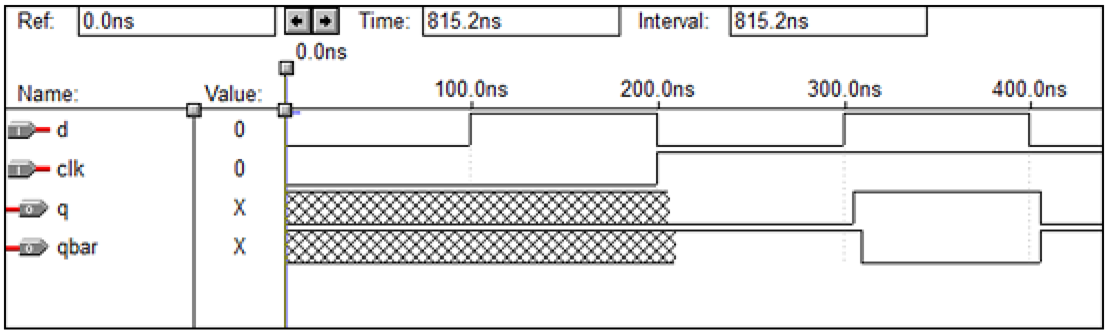


VHDL Code for a D Flip Flop

```
Library ieee;
```

```
use ieee.std_logic_1164.all;

entity dflip is
  port(d,clk:in bit; q,qbar:buffer bit);
end dflip;

architecture virat of dflip is
  signal d1,d2:bit;
begin
  d1<=d nand clk;
  d2<=(not d) nand clk;
  q<= d1 nand qbar;
  qbar<= d2 nand q;
end virat;
```

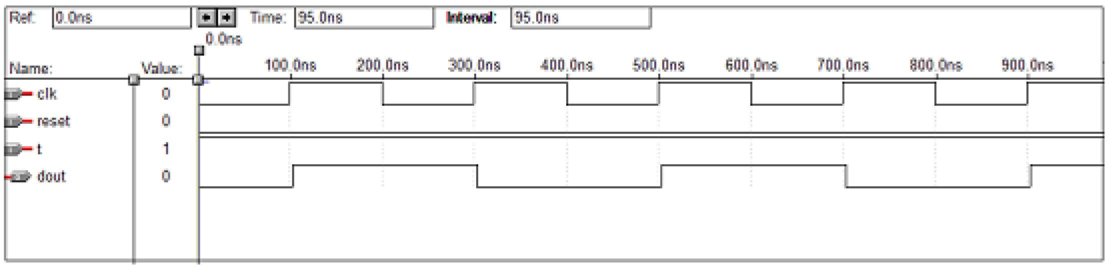
Waveforms



VHDL Code for a T Flip Flop

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Toggle_flip_flop is
  port(
    t : in STD_LOGIC;
    clk : in STD_LOGIC;
    reset : in STD_LOGIC;
    dout : out STD_LOGIC
  );
end Toggle_flip_flop;

architecture virat of Toggle_flip_flop is
begin
  tff : process (t,clk,reset) is
  variable m : std_logic : = '0';

  begin
    if (reset = '1') then
      m : = '0';
    elsif (rising_edge (clk)) then
      if (t = '1') then
        m : = not m;
      end if;
    end if;
    dout < = m;
  end process tff;
end virat;
```

Waveforms

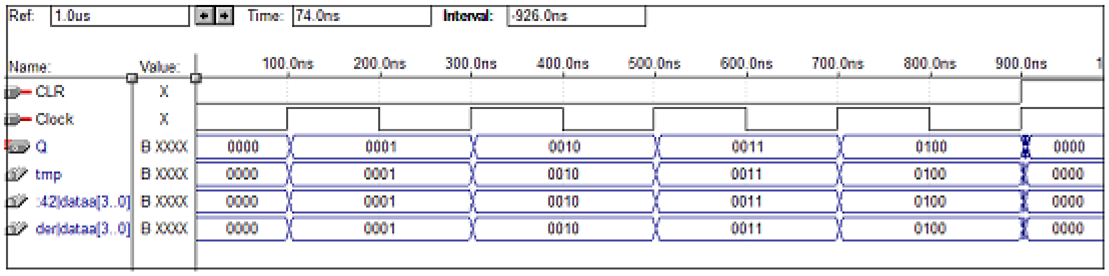


VHDL Code for a 4 - bit Up Counter

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
entity counter is
  port(Clock, CLR : in std_logic;
    Q : out std_logic_vector(3 downto 0)
  );
end counter;

architecture virat of counter is
  signal tmp: std_logic_vector(3 downto 0);
begin
  process (Clock, CLR)

  begin
    if (CLR = '1') then
      tmp < = "0000";
    elsif (Clock'event and Clock = '1') then
      mp <= tmp + 1;
    end if;
  end process;
  Q <= tmp;
end virat;
```

Waveforms



Ref: 1.0us Time: 74.0ns Interval: -926.0ns

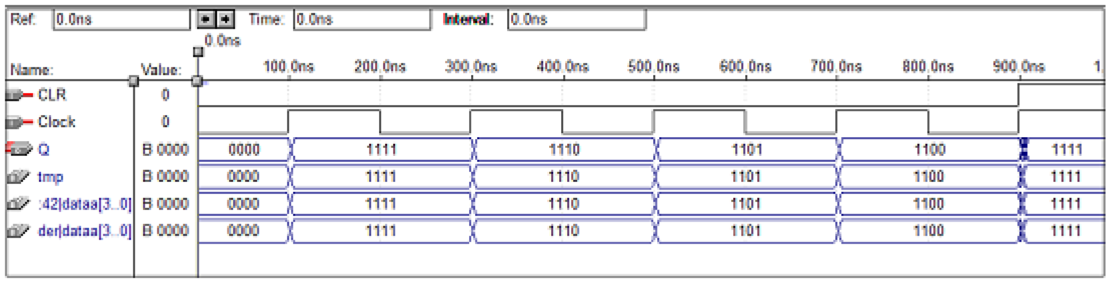
Name:	Value:						
Q	B XXXX	0000	0001	0010	0011	0100	0000
tmp	B XXXX	0000	0001	0010	0011	0100	0000
:42|dataa[3..0]	B XXXX	0000	0001	0010	0011	0100	0000
der|dataa[3..0]	B XXXX	0000	0001	0010	0011	0100	0000

VHDL Code for a 4-bit Down Counter

```
library ieee;
use ieee.std_logic_1164.all;
```

```vhdl
use ieee.std_logic_unsigned.all;

entity dcounter is
  port(Clock, CLR : in std_logic;
    Q : out std_logic_vector(3 downto 0));
end dcounter;

architecture virat of dcounter is
  signal tmp: std_logic_vector(3 downto 0);

begin
  process (Clock, CLR)
  begin
    if (CLR = '1') then
      tmp <= "1111";
    elsif (Clock'event and Clock = '1') then
      tmp <= tmp - 1;
    end if;
  end process;
  Q <= tmp;
end virat;
```

Waveforms


Ref: 0.0ns
Time: 0.0ns
Interval: 0.0ns
0.0ns
Name:
Value:
100.0ns
200.0ns
300.0ns
400.0ns
500.0ns
600.0ns
700.0ns
800.0ns
900.0ns
CLR 0
Clock 0
Q B 0000
tmp B 0000
:42|dataa[3..0] B 0000
der|dataa[3..0] B 0000
0000
1111
1110
1101
1100
1111