UNIT - 5
VHDL

5.1 INTRODUCTION TO HARDWARE DESCRIPTION LANGUAGE (HDL)

A hardware description language (HDL) describes the hardware of digital system in a textual form. It is used to represent logic diagrams, Boolean expresions and other digital circuits. HDL textual form can be read by both humans and computers. So the design can be exchanged easily between the designers using HDL. The main applications of HDL processing are

1. Logic simulation
2. Logic synthesis
3. Timing Analysis
4. Post Synthesis Simulation
5. Hardware generation

5.1.1 Logic simulation

Simulation for design validation is done before a design is synthesized. This simulation pass is also referred to as behavioral simulation or RT (Register transfer) level simulation or Pre-synthesis simulation. Logic simulation is the process of representing the structure and behavior of a digital logic system through the use of computer. A simulator interprets the HDL description given by the designer and produces readable output, such as a timing diagram, that predicts how the hardware will behave before it is actually fabricated.

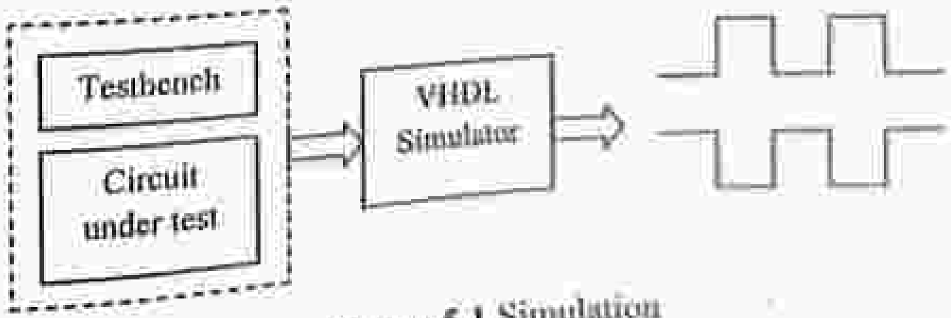

Figure 5.1 Simulation

During the simulation process, the errors in the HDL statements are detected and corrections can be made by modifying the appropriate HDL statements. At the RT level a design includes clock level timing, but no gate delays and wire delays are included. Simulation at this level is accurate to the clock level. Timing of RT level simulation does not consider hazards, glitches, race conditions and other timing issues. The flow chart shown in figure 5.2 gives the sequence of tasks.

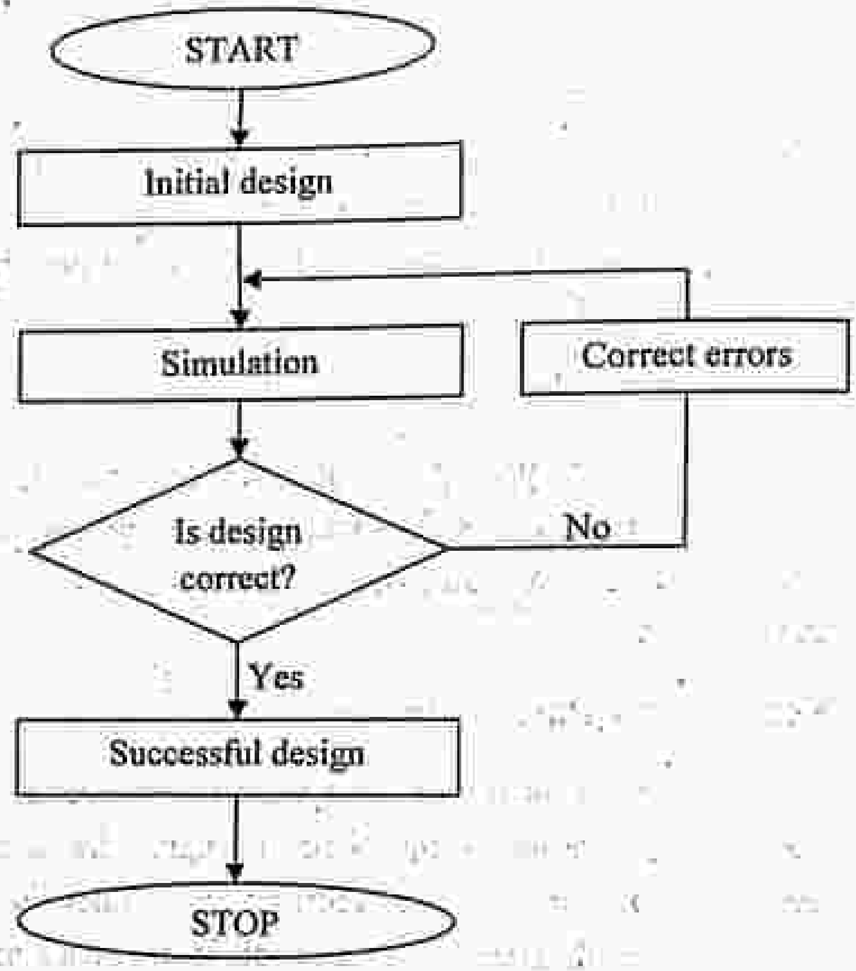

Figure 5.2

5.1.2 Logic synthesis

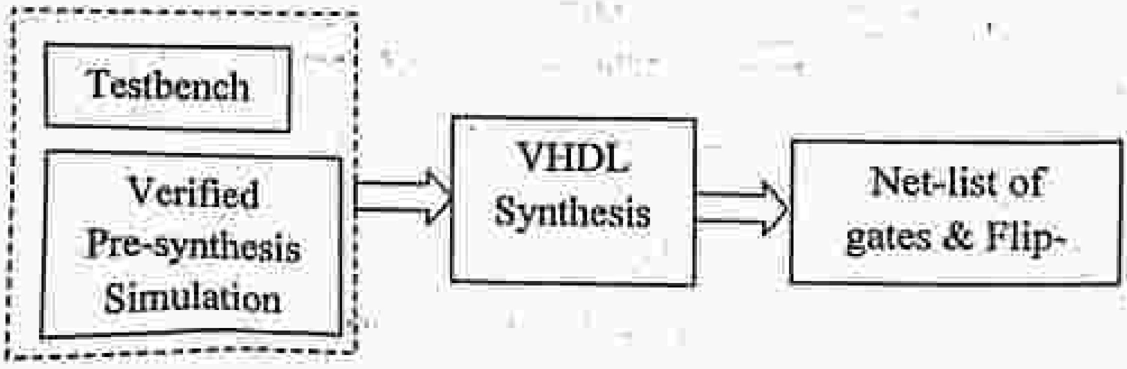

Figure 5.3 Synthesis

Logic synthesis is the process of deriving a list of components and their interconnections from the model of a digital system described in HDL. The logic synthesis produces a database with instructions on how to fabricate a physical piece of digital hardware. In the design process, after a design is successfully entered and its pre-synthesis simulation results have been verified by the designer, it must be compiled to make one step closer to an actual hardware on silicon. The compilation process translates various ports of the design to an intermediate format, links all ports together, generates the corresponding logic, places and route components of the target hardware and generates timing details. The input of the compilation phase is a hardware description that consists of various levels of VHDL and its output is a detailed hardware for programming an FPLD (Field programmable logic devices) or manufacturing an ASIC.

5.1.3 Timing analysis

The timing analysis phase generates worst-case delays, clocking speed, delays from one gate to another as well as required setup and hold times. Results of timing analysis appear in tables or graphs. Designers use this information to decide the speed of their circuits.

5.1.4 Post synthesis simulation

After synthesis is done, the synthesis tool generates a complete net-list of target hardware components and their timings. The details of gates used for the implementation of the design are described in this net-list. The net-list also includes wiring delays and load effects on gates used in the post synthesis design.

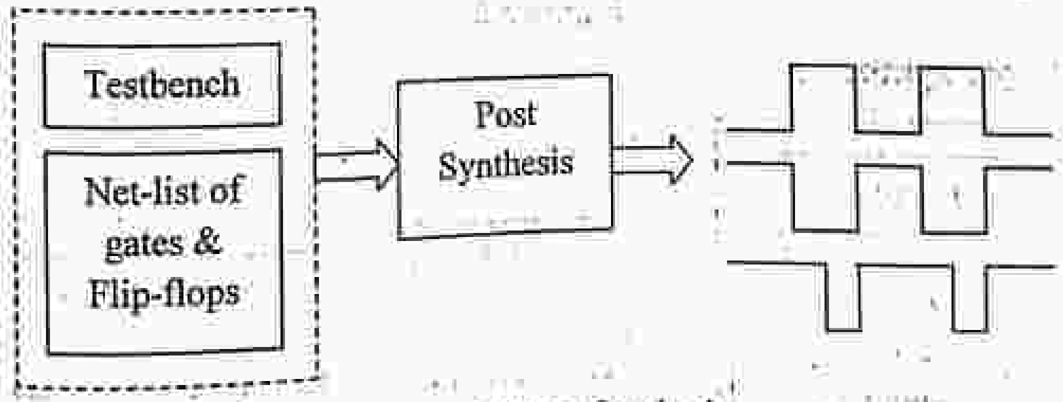

Figure 5.4 Post Synthesis

5.1.5 Hardware generation

The last stage in an automated VHDL based design is hardware generation. This stage generates a net-list for ASIC manufacturing, a program for programming PLD's or layout of custom IC cells.

There are two standard HDLs that are supported by IEEE:

1. VHDL (VHSIC HDL) or Very High Speed Integrated Circuits Hardware Description Language.
2. Verilog HDL.

5.2 RTL DESIGN (REGISTER TRANSFER LEVEL)

For the design of a digital system using an automated design environment, the RTL design begins with the specification of the design and ends with generating net-list for an ASIC (Application specific Integrated Circuits), or layout for a custom IC, or a program for PLD (Programmable logic devices). The various steps followed in the design flow are given below.

1. Write the VHDL program. Save it with a file name (filename.vhd). the name is given to entity also.
2. Next is the compilation process. High level VHDL Language is converted into net-list at the gate level.
3. Next is optimization, where the optimized net-list is obtained.
4. Simulate the program
5. Finally, place and route software generate physical layout for PLD/FPGA chip.

5.2.1 VHDL Entry

The first step in the design of a digital system is the design entry. The design is described in VHDL. A complete design may consist of components at the gate or transistor level or behavioral ports describing the functionality or components described by their bussing structure.

VHDL designs are usually described at the level that specifies system registers and transfer of data between register through busses. This level of system description is referred to as register transfer level (RTL).

VHDL sequential statements are used for high level behavioral description. A system or component is described in a sequential fashion similar to software language. VHDL signal assignments are statements for representing, logic block, bus assignments and bus and input/output interconnect specifications. VHDL instantiation statements are used to describe lower level components. These components can be small as a gate or a transistor, or as large as a complete processor.

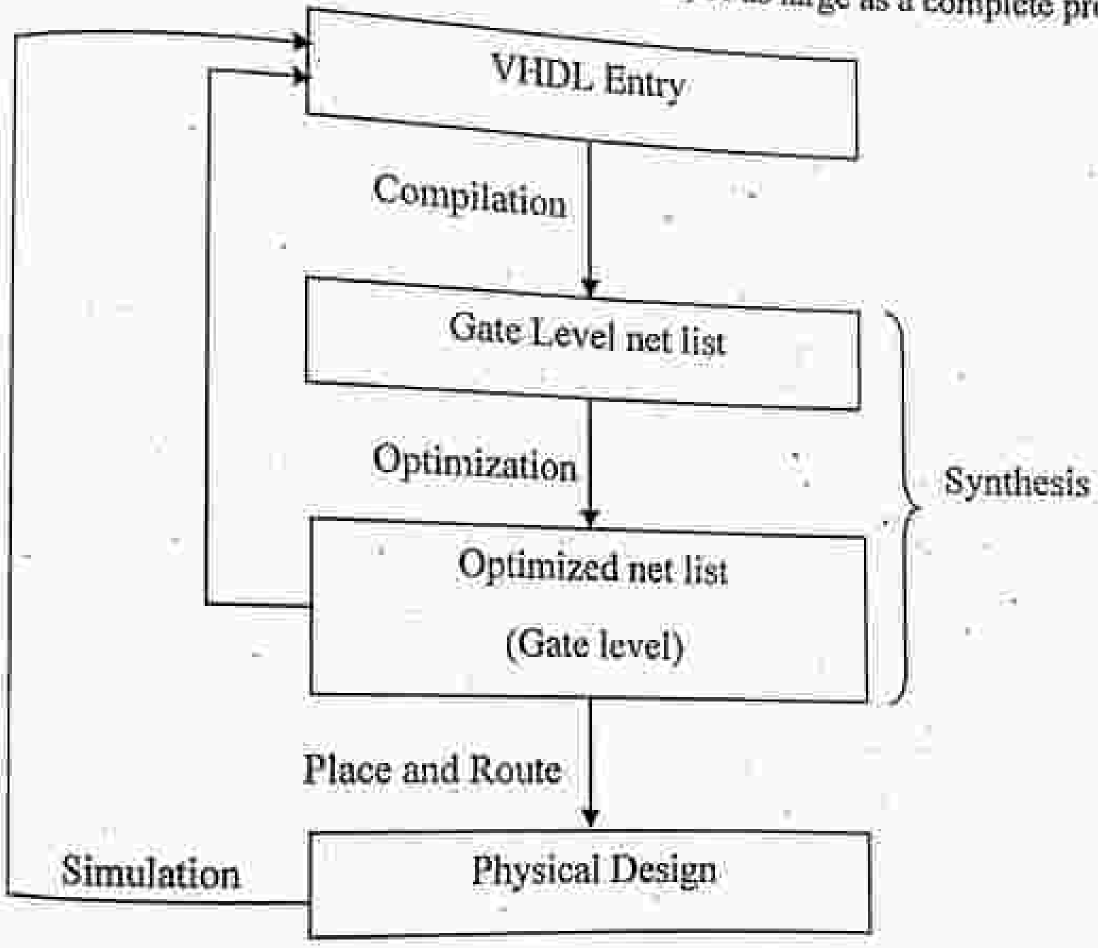

Figure 5.5 RTL Design flow

Code structure

VHDL is not case sensitive. Representing 'a' and 'A' both are same.

Library declarations

It contains a list of all libraries which are used in the design.

Eg. IEEE, std,

Entity

It is used to specify the inputs/output pins of the circuit.

(eg)

```
entity or1 is
port (a,b,c: in STD_LOGIC;
        y : out STD_LOGIC);
end
```

Here a,b, c represents the input pins, y represents the output pin.

Architecture

It is used to specify the behaviour of the circuit.

```
(eg) architecture or1 of or2 is
        begin
        Y<=a or b;
        end or2
```

Mode:

The mode of the signal can be IN, OUT, INOUT, BUFFER.

Type:

The type of the signal can be BIT, STD_LOGIC, INTEGER

Name:

The Name of the entity can be any name except VHDL reserved words.

5.2.2 Data Types

HDL uses the following data types

1. Scalar type
    a. Enumeration
    b. Integer
    c. Physical
    d. Floating point

2. Composite types
    a. Array type
    b. Record type
3. Access types
4. File types

1. **Scalar type**

Enumeration types

It defines a type that has a set of user defined values consisting of identifiers and character literals. Values of an enumeration type are called enumeration literals.

(eg) type MUL is ('u', 'o', 'l', 'z');

Integer types

It defines a type whose set of values fall within a specific integer range.

(eg) type INDEX is range 0 to 15;

Floating point type

It has a set of values in a given range of real numbers.

(eg) type REAL_DATA is range 0.0 to 31.9;

Physical types

It contains values that represent measurement of some physical quantity like time, length, voltage or current.

(eg) type CURRENT is range 0 to 1E9;

2. **Composite types**

These are composed of elements of a single type (array type) or elements that have the different type (record type)

Array types

An object of an array type consists of elements that have the same type.

(eg) type ADDRESS_WORD is array (0 to 63) of BIT;

Record types

An object of record type is composed of elements of same or different types.

(eg) type PIN_TYPE is range 0 to 10;

3. **Access types**

These provide access to objects of a given type. Values belonging to an access type are pointers to a dynamically allocated object of some other type.

(eg) type PTR is access MODULE;

4. **File types**

These provide access to objects that contain a sequence of values of a given type. It provides a mechanism by which VHDL design communicates with the host environment.

Syntax: type file_type_name is file of type_name;

5.3 VHDL MODELING APPROACHES

VHDL can be used in three different approaches for describing the hardware. These three different approaches are

1. Data flow modeling
2. Structural modeling
3. Behavioral modeling

5.3.1 Data flow modeling

To make an effective design with VHDL, the design is typically decomposed into several blocks. These blocks are then interconnected together to form a complete design. Therefore a VHDL design may be completely described in a single block or it may be represented in several blocks. The entity describes how that block operates. An example for entity declaration is given below.

```
entity orgate is                    - declaration of entity
 port(a,b : in std_logic;           - declaration of input pins
        y: out std_logic);          - declaration of output pins
end orgate;                         - end of entity
```

Here, orgate is the name of the entity. The list to the left of the colon contains the name of the signals (a,b,y). The right side of the colon represents the mode of the signals. The mode may be input (in), output (out) or both input and output (inout). In the above example a and b are the input signals and y be the output signal. After declaring the entity the architecture declaration for the orgate entity can be written as follows.

```
architecture arch_or of orgate is          - architecture declaration
begin
y<= a or b;
end  arch_or;                                    - End of architecture
```

Here the name of the architecture is arch_or having the entity named as orgate. The lines between begin and end describes the operation of the design. Here <= operator describes how the data flows from the signals on the right side to the signal on the left side. The dataflow approach indicates how the data flows from input to the output. An example of dataflow approach is shown in example 5.1

Example 5.1: Write the VHDL program for AND gate using dataflow approach

Solution:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity andgate is
   Port ( a : in  STD_LOGIC;
         b : in  STD_LOGIC;
         y : out  STD_LOGIC);
end andgate;
architecture gates of andgate is
begin
y <= a and b;
end gates;
```

Example 5.2: Write the VHDL program for OR gate using dataflow approach

Solution:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity orgate is
  Port ( a : in  STD_LOGIC;
       b : in  STD_LOGIC;
       y : out  STD_LOGIC);
end orgate;
architecture gates of orgate is
begin
y <= a or b;
end gates;
```

Example 5.3: Write the VHDL program for XOR gate using dataflow approach

Solution:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity xorgate is
  Port ( a : in  STD_LOGIC;
       b : in  STD_LOGIC;
       y : out  STD_LOGIC);
end xorgate;
architecture gates of xorgate is
begin
y <= a xor b;
end gates;
```

Example 5.4: Write the VHDL program for half adder using dataflow approach

Solution:

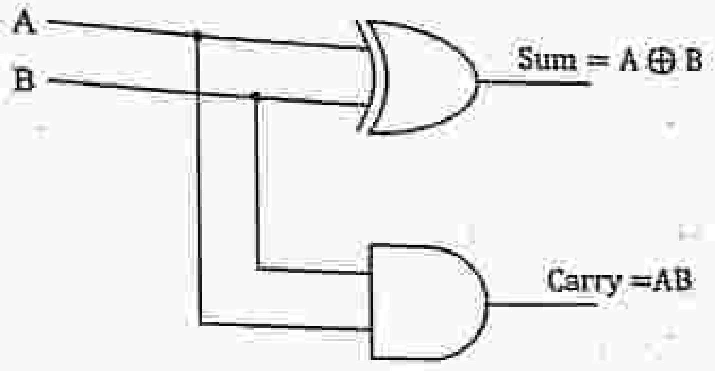

Figure 5.6 Logic diagram of Half adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity HA1 is
   Port ( a : in  STD_LOGIC;
        b : in  STD_LOGIC;
        sum : out  STD_LOGIC;
        carry : out  STD_LOGIC);
end HA1;
architecture adder of HA1 is
begin
sum<= a xor b;
carry<= a and b;
end adder;
```

5.3.2 Structural modeling

Ones the basic building blocks are defined using entities they can be combined together to form the complete design. The structural modeling is a textual description of the schematic.

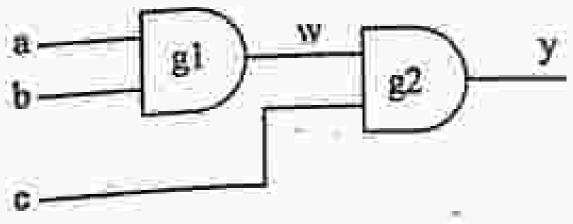

Figure 5.7

The entity declaration for the above circuit is shown below.

```
entity circuit is
port(a,b,c:in std_logic;
        y: out std_logic);
end circuit;
```

The architecture declaration of the structural description is shown below

```
architecture Behavioral of circuit is
signal w: std_logic;
component andgate is
port(i,j: in std_logic;
k: out std_logic);
end component;
begin
g1: andgate port map(a,b,w);
g2: andgate port map(w,c,y);
end;
```

where the VHDL for AND gate (andgate) is shown in example 5.1

Example 5.5: Write the VHDL code for Full adder using structural description

Solution:

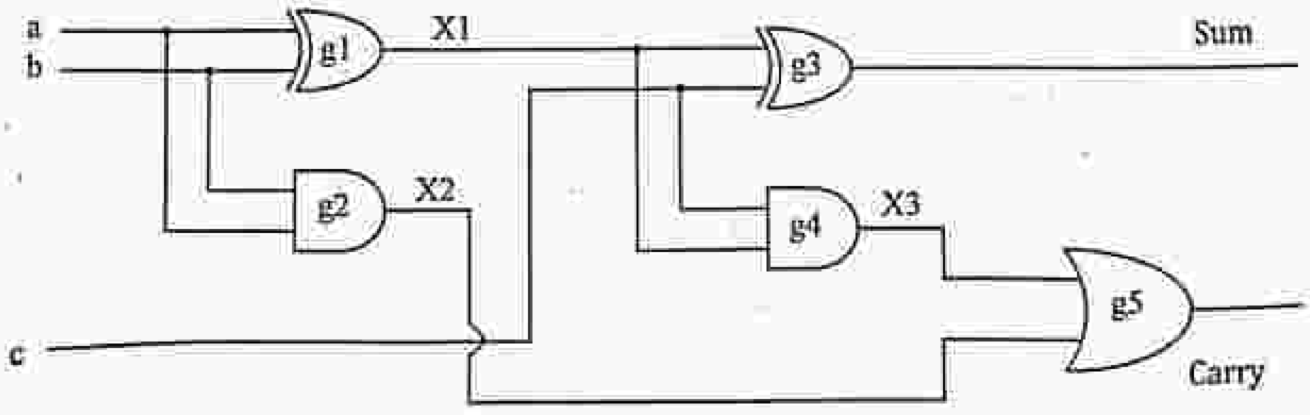

Figure 5.8 Logic diagram of Full adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity fulladder is
  Port ( a,b,c : in  STD_LOGIC;
         sum,carry : out  STD_LOGIC);
end fulladder;
architecture adder of fulladder is
signal X1,X2,X3: std_logic;
component andgate is
port(i,j: in std_logic;
        k: out std_logic);
end component;
component orgate is
port(i,j: in std_logic;
k: out std_logic);
end component;
component xorgate is
port(i,j: in std_logic;
k: out std_logic);
end component;
begin
          g1: xorgate port map(a,b,X1);
          g2: andgate port map(a,b,X2);
          g3: xorgate port map(X1,c,sum);
          g4: andgate port map(X1,c,X3);
          g5: orgate port map(X2,X3,carry);
end adder;
```

where the VHDL for AND gate (andgate), XOR gate (xorgate), OR gate (orgate) is shown in example 5.1, example 5.3, example 5.2 respectively.

5.3.3 Behavioral modeling

The behavioral modeling approach differs from the dataflow approach and structural approach. This method models, how the input is guided towards the output. It does not deal with the working of the internal circuit. Behavioral model can be used in complex components design. It is more powerful and more convenient for such complex design. An example for VHDL design using behavioral modeling is shown below. This shows the design of XNOR gate using behavioral modeling.

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

Table 5.1 Truth table of XNOR gate

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity xnorgate is
Port ( a,b : in  STD_LOGIC;
       y : out  STD_LOGIC);
end xnorgate;
architecture gates of xnorgate is
begin
process(a,b)
begin
if (a='0' and b='0') then
y<='1';
elsif (a='0' and b='1') then
y<='0';
elsif (a='1' and b='0') then
y<='0';
elsif (a='1' and b='1') then
y<='1';
end if;
end process;
end gates;
```

Example 5.6: Write the VHDL code for full adder using behavioral modeling

Solution

Inputs			Outputs	
a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 5.2 Truth table of full adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity fulladder1 is
   Port ( a,b,c : in  STD_LOGIC;
        sum, carry : out  STD_LOGIC);
end fulladder1;
architecture adder of fulladder1 is
begin
process(a,b,c)
begin
if (a='0' and b='0' and c='0') then
sum<='0';
carry<='0';
elsif (a='0' and b='0' and c='1') then
sum<='1';
carry<='0';
elsif (a='0' and b='1' and c='0') then
sum<='1';
carry<='0';
```

```
elsif (a='0' and b='1' and c='1') then
sum<='0';
carry<='1';
elsif (a='1' and b='0' and c='0') then
sum<='1';
carry<='0';
elsif (a='1' and b='0' and c='1') then
sum<='0';
carry<='1';
elsif (a='1' and b='1' and c='0') then
sum<='0';
carry<='1';
elsif (a='1' and b='1' and c='1') then
sum<='1';
carry<='1';
end if;
end process;
end adder;
```

5.4 VHDL FOR COMBINATIONAL LOGIC CIRCUITS

Example 5.7: Write the VHDL program for full adder using dataflow modeling

Solution

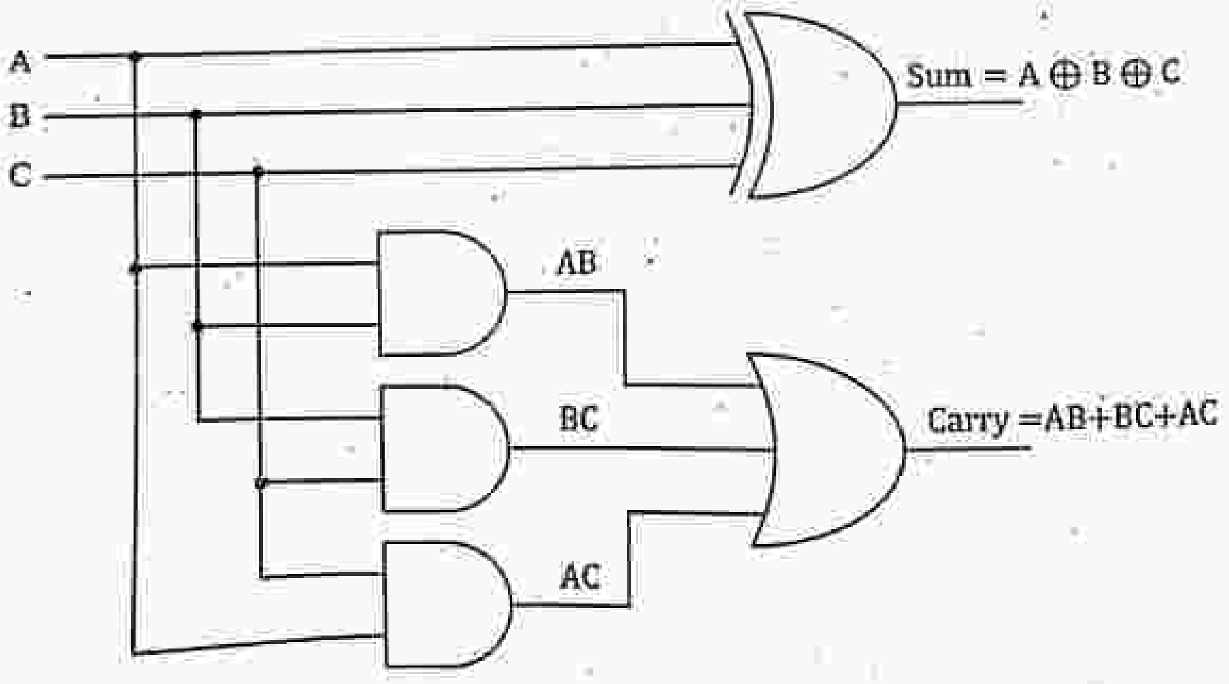

Figure 5.9 Logic diagram of full adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity FA1 is
  Port ( a : in  STD_LOGIC;
       b : in  STD_LOGIC;
         C : in  STD_LOGIC;
       sum : out  STD_LOGIC;
       carry : out  STD_LOGIC);
end FA1;
architecture adder of FA1 is
begin
sum<= a xor b xor c;
carry<= (a and b) or (b and c) or (a and c);
end adder;
```

Example 5.8: Write the VHDL program for Half subtractor using dataflow modeling

Solution

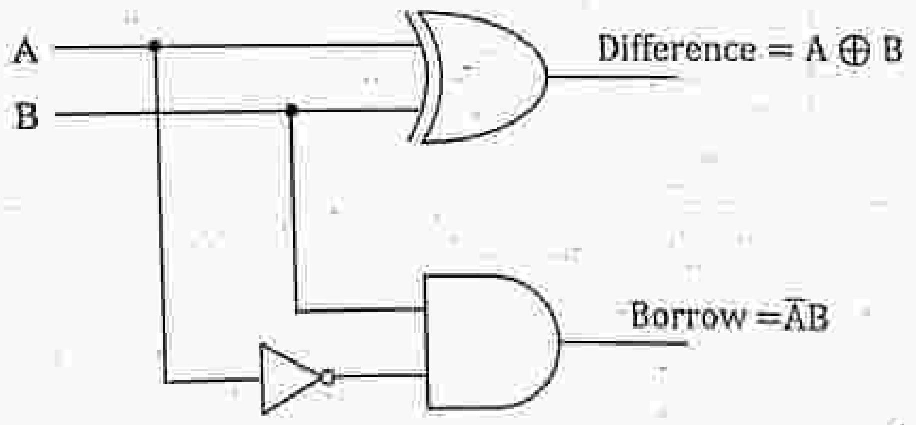

Figure 5.10 Logic diagram of half Subtractor

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity HS1 is
  Port ( A : in  STD_LOGIC;
       B : in  STD_LOGIC;
       Borrow : out  STD_LOGIC;
       Difference : out  STD_LOGIC);
```

```
end HS1;
architecture Behavioral of HS1 is
begin
Difference<= A xor B;
Borrow<= (not A) and B;
end Behavioral;
```

Example 5.9: Write the VHDL program for full subtractor using dataflow modeling

Solution

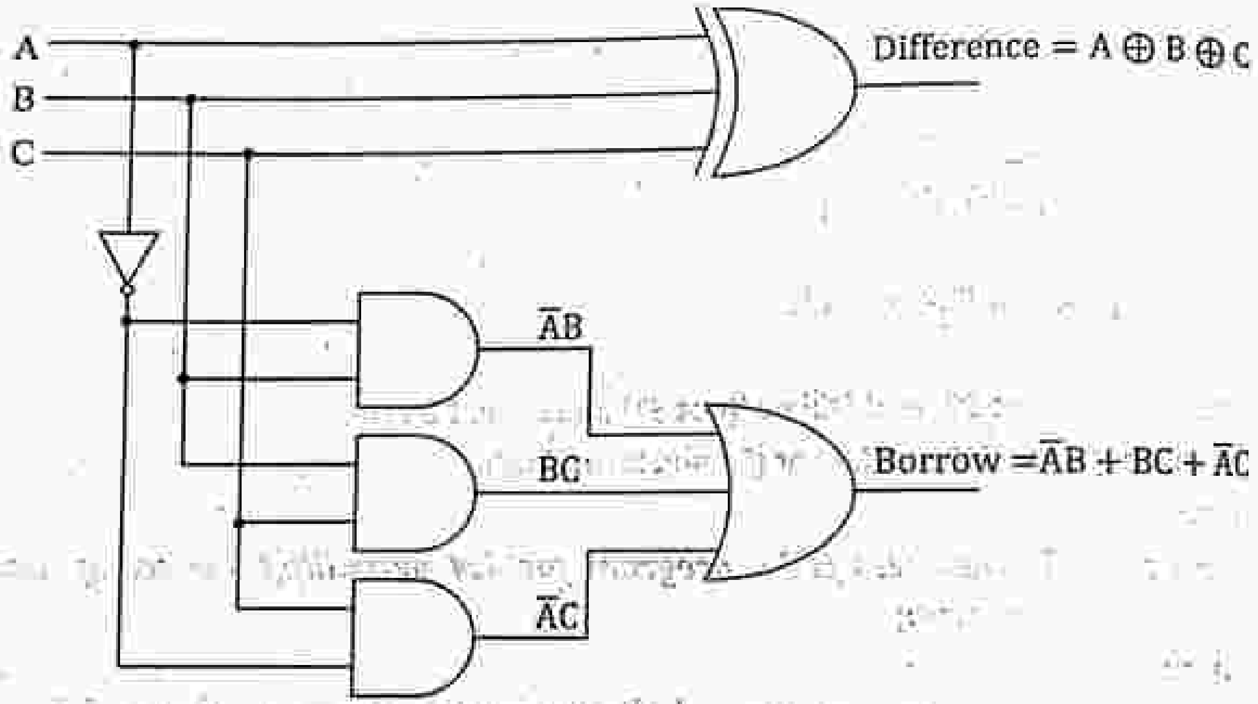

Figure 5.11 Logic diagram of full subtractor

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FS1 is
  Port ( A : in  STD_LOGIC;
      B : in  STD_LOGIC;
      C : in  STD_LOGIC;
      Borrow : out  STD_LOGIC;
      Difference : out  STD_LOGIC);
end FS1;
architecture Behavioral of FS1 is
begin
Difference<= A xor B xor C;
Borrow<= ((not A) and B) or (B and C) or ((not A) and C);
end Behavioral;
```

Example 5.10: Write the VHDL program for 4:1 multiplexer using dataflow modeling

Solution:

The dataflow description for the 4:1 multiplexer shown in figure 2.174 is as follows,

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux1 is
  Port ( D0 : in  STD_LOGIC;
      D1 : in  STD_LOGIC;
      D2 : in  STD_LOGIC;
      D3 : in  STD_LOGIC;
        S0 : in  STD_LOGIC;
        S1 : in  STD_LOGIC;
      Y : out  STD_LOGIC);
end mux1;
architecture Behavioral of mux1 is
begin
Y<=((not S1) and (not S0) and D0) or ((not S1) and S0 and D1) or
      (S1 and (not S0) and D2) or (S1 and S0 and D3);
end Behavioral;
```

Example 5.11: Write the VHDL program for 1:4 de-multiplexer using dataflow modeling

Solution:

The dataflow description for the 1:4 de-multiplexer shown in figure 2.195 is as follows,

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity demux1 is
port(D,S1,S0:in std_logic;
        Y3,Y2,Y1,Y0:out std_logic);
end demux1;
architecture Behavioral of demux1 is
begin
Y0<=((not S1) and (not S0) and D);
Y1<=((not S1) and S0 and D);
Y2<=(S1 and (not S0) and D);
Y3<=(S1 and S0 and D);
end Behavioral;
```

Example 5.12: Write the VHDL program for binary to gray code convertor using dataflow modeling

Solution:

The dataflow description for the binary to gray code convertor shown in figure 2.228 is shown below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity binary2gray is
   Port ( b3,b2,b1,b0 : in  STD_LOGIC;
        g3,g2,g1,g0 : out  STD_LOGIC);
end binary2gray;
architecture code_converter of binary2gray is
begin
g0<=b1 xor b0;
g1<=b2 xor b1;
g2<=b3 xor b2;
g3<=b3;
end code_converter;
```

Example 5.13: Write the VHDL program for 4-bit binary parallel adder using structural modeling

Solution

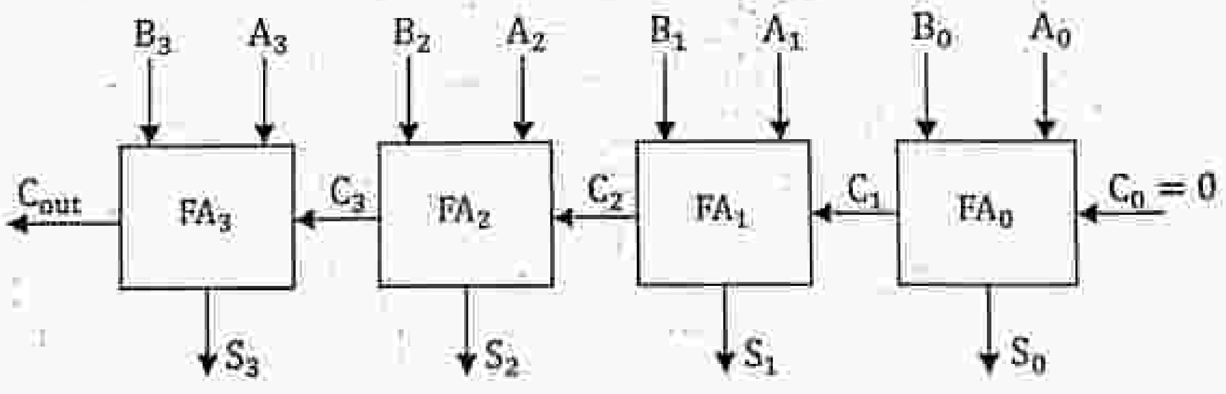

Figure 5.12 4-bit adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity adder4bit is
  port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
      B : in  STD_LOGIC_VECTOR (3 downto 0);
                          Cin:in  STD_LOGIC;
```

```
        S : out  STD_LOGIC_VECTOR (3 downto 0);
        Cout : out  STD_LOGIC);
end adder4bit;
architecture addition of adder4bit is
signal C1,C2,C3 : STD_LOGIC;
component fulladder is
port(a,b,c: in std_logic;
        sum,carry: out std_logic);
end component;
begin
FA1: fulladder port map(A(0),B(0),Cin,S(0),C1);
FA2: fulladder port map(A(1),B(1),C1,S(1),C2);
FA3: fulladder port map(A(2),B(2),C2,S(2),C3);
FA4: fulladder port map(A(3),B(3),C3,S(3),Cout);
end addition;
```

where the VHDL program for fulladder is shown in example 5.7.

Example 5.14: Write the VHDL program for 4×4 bit array multiplier using structural modeling

Solution:

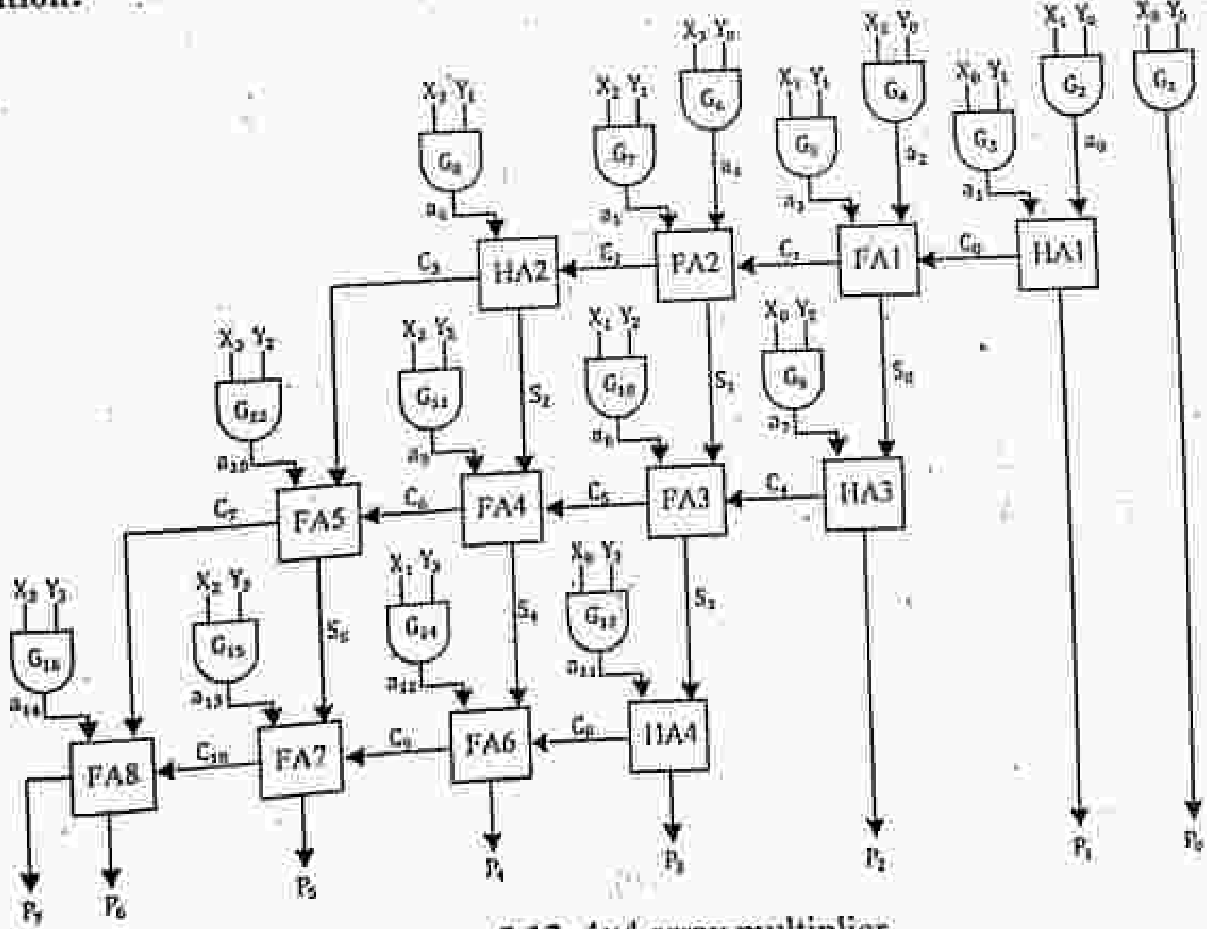

Figure 5.13 4×4 array multiplier

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity multiplier is
  Port ( X : in  STD_LOGIC_VECTOR (3 downto 0);
       Y : in  STD_LOGIC_VECTOR (3 downto 0);
       P : out  STD_LOGIC_VECTOR (7 downto 0));
end multiplier;
architecture circuit of multiplier is
signal a: STD_LOGIC_VECTOR (14 downto 0);
signal c: STD_LOGIC_VECTOR (10 downto 0);
signal s: STD_LOGIC_VECTOR (5 downto 0);
component andgate2 is
port(a,b: in std_logic;
   y: out std_logic);
end component;
component halfadder is
port(a,b: in std_logic;
   sum,carry: out std_logic);
end component;
component fulladder is
port(a,b,c: in std_logic;
   sum,carry: out std_logic);
end component;
begin
G1:andgate2 port map(X(0),Y(0),P(0));
G2:andgate2 port map(X(1),Y(0),a(0));
G3:andgate2 port map(X(0),Y(1),a(1));
```

```
G4:andgate2 port map(X(2),Y(0),a(2));
G5:andgate2 port map(X(1),Y(1),a(3));
G6:andgate2 port map(X(3),Y(0),a(4));
G7:andgate2 port map(X(2),Y(1),a(5));
G8:andgate2 port map(X(3),Y(1),a(6));
G9:andgate2 port map(X(0),Y(2),a(7));
G10:andgate2 port map(X(1),Y(2),a(8));
G11:andgate2 port map(X(2),Y(2),a(9));
G12:andgate2 port map(X(3),Y(2),a(10));
G13:andgate2 port map(X(0),Y(3),a(11));
G14:andgate2 port map(X(1),Y(3),a(12));
G15:andgate2 port map(X(2),Y(3),a(13));
G16:andgate2 port map(X(3),Y(3),a(14));
HA1:halfadder port map(a(0),a(1),P(1),c(0));
FA1:fulladder port map(a(3),a(2),c(0),s(0),c(1));
FA2:fulladder port map(a(5),a(4),c(1),s(1),c(2));
HA2:halfadder port map(a(6),c(2),s(2),c(3));
HA3:halfadder port map(s(0),a(7),P(2),c(4));
FA3:fulladder port map(a(8),s(1),c(4),s(3),c(5));
FA4:fulladder port map(a(9),s(2),c(5),s(4),c(6));
FA5:fulladder port map(a(10),c(3),c(6),s(5),c(7));
HA4:halfadder port map(s(3),a(11),P(3),c(8));
FA6:fulladder port map(a(12),s(4),c(8),P(4),c(9));
FA7:fulladder port map(a(13),s(5),c(9),P(5),c(10));
FA8:fulladder port map(a(14),c(7),c(10),P(6),P(7));
end circuit;
```

where the VHDL for 2-input AND gate (andgate2), Half adder (halfadder), and Full adder (fulladder) are shown in example 5.1, example 5.4 and example 5.7 respectively.

Example 5.15: Write the VHDL program for 3 bit magnitude comparator using structural modeling

Solution:

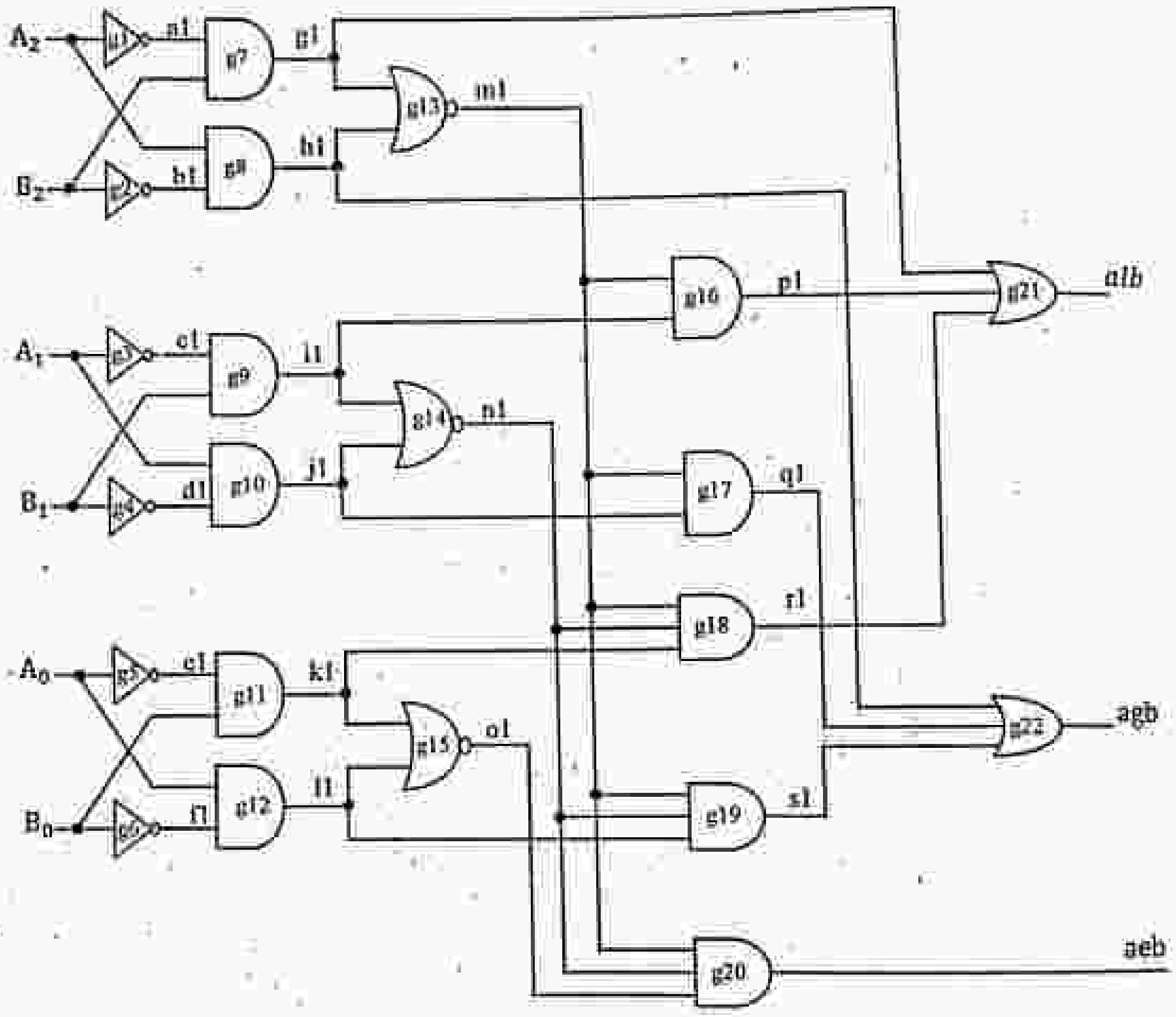

Figure 5.14 Logic diagram of 3-bit magnitude comparator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity magcomp is
  Port ( A : in  STD_LOGIC_VECTOR (2 downto 0);
        B : in  STD_LOGIC_VECTOR (2 downto 0);
        agb : out  STD_LOGIC;
        alb : out  STD_LOGIC;
        aeb : out  STD_LOGIC);
```

```
end magcomp;
architecture combinational of magcomp is
signal a1,b1,c1,d1,e1,f1,g1,h1,i1,j1,k1,l1,m1,n1,o1,p1,q1,r1,s1: STD_LOGIC;
component notgate is
port(a: in std_logic;
    y: out std_logic);
end component;
component andgate2 is
port(a,b: in std_logic;
    y: out std_logic);
end component;
component andgate3 is
port(a,b,c: in std_logic;
    y: out std_logic);
end component;
component orgate3 is
port(a,b,c: in std_logic;
    y: out std_logic);
end component;
component norgate2 is
port(a,b: in std_logic;
    y: out std_logic);
end component;
begin
ga1:notgate port map(A(2),a1);
ga2:notgate port map(B(2),b1);
ga3:notgate port map(A(1),c1);
ga4:notgate port map(B(1),d1);
ga5:notgate port map(A(0),e1);
ga6:notgate port map(B(0),f1);
ga7:andgate2 port map(a1,B(2),g1);
ga8:andgate2 port map(b1,A(2),h1);
```

```
ga9:andgate2 port map(c1,B(1),i1);
ga10:andgate2 port map(d1,A(1),j1);
ga11:andgate2 port map(e1,B(0),k1);
ga12:andgate2 port map(f1,A(0),l1);
ga13:norgate2 port map(g1,h1,m1);
ga14:norgate2 port map(i1,j1,n1);
ga15:norgate2 port map(k1,l1,o1);
ga16:andgate2 port map(m1,i1,p1);
ga17:andgate2 port map(m1,j1,q1);
ga18:andgate3 port map(m1,n1,k1,r1);
ga19:andgate3 port map(m1,n1,l1,s1);
ga20:andgate3 port map(m1,n1,o1,aeb);
ga21:orgate3 port map(g1,p1,r1,alb);
ga22:orgate3 port map(h1,q1,s1,agb);
end combinational;
```

where the VHDL for NOT gate (notgate), 2-input AND gate (andgate2), 3-input AND gate (andgate3), 2-input NOR gate (norgate2) and 3-input OR gate (orgate3) are shown in example 5.16, example 5.1, example 5.19, example 5.17 and example 5.18 respectively.

Example 5.16: Write the VHDL program for not gate using dataflow modeling

Solution

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity notgate is
  Port ( a : in  STD_LOGIC;
       y : out  STD_LOGIC);
end notgate;
architecture gates of notgate is
begin
y<= (not a);
end gates;
```

Example 5.17: Write the VHDL program for 2 input NOR gate using dataflow modeling

Solution

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity norgate2 is
  Port ( a,b : in  STD_LOGIC;
       y : out  STD_LOGIC);
end norgate2;
architecture gates of norgate2 is
begin
y<=(a nor b);
end gates;
```

Example 5.18: Write the VHDL program for 3 input OR gate using dataflow modeling

Solution

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity orgate3 is
  Port ( a,b,c : in  STD_LOGIC;
       y : out  STD_LOGIC);
end orgate3;
architecture gates of orgate3 is
begin
y<=a or b or c;
end gates;
```

Example 5.19:* *Write the VHDL program for 3 input AND gate using dataflow modeling

Solution

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity andgate3 is
   Port ( a,b,c : in  STD_LCGIC;
        y : out  STD_LOGIC);
end andgate3;
architecture gates of andgate3 is
begin
y<= a and b and c;
end gates;
```

Example 5.20:* *Write the VHDL program for CMOS AND gate.

Solution

VHDL for CMOS AND gate

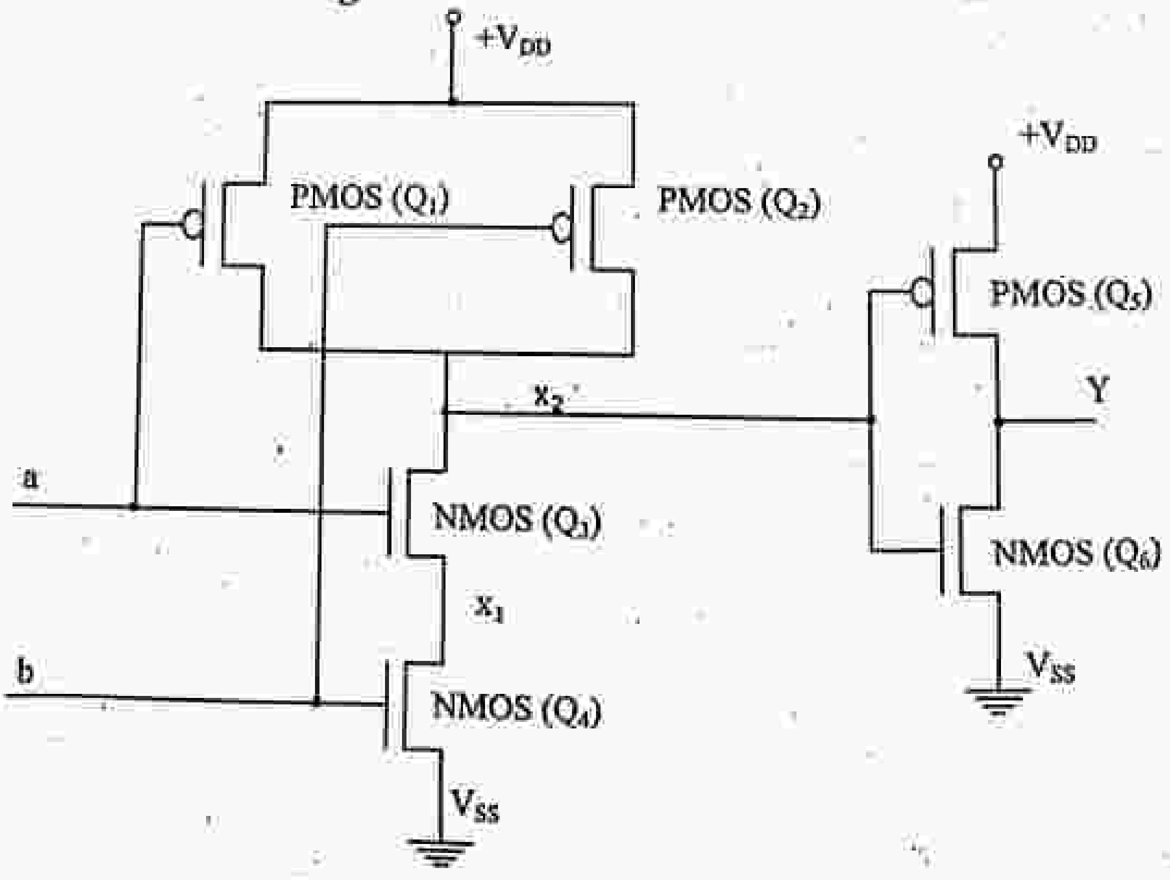

Figure 5.15 CMOS AND gate

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity cmosand is
  Port ( a,b : in  STD_LOGIC;
       Y : out  STD_LOGIC);
end cmosand;
architecture cmos of cmosand is
signal vcc1,gnd1,x1,x2:std_logic;
component pmos is
port(S1,G1: in std_logic;
    D1: out std_logic);
end component;
component nmos is
port(S1,G1: in std_logic;
    D1: out std_logic);
end component;
begin
vcc1<='1';
gnd1<='0';
Q1:pmos port map(vcc1,a,x2);
Q2:pmos port map(vcc1,b,x2);
Q4:nmos port map(gnd1,b,x1);
Q3:nmos port map(x1,a,x2);
Q5:pmos port map(vcc1,x2,Y);
Q6:nmos port map(gnd1,x2,Y);
end cmos;
```

VHDL for PMOS

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity pmos is
Port ( S1,G1 : in  STD_LOGIC;
     D1 : out  STD_LOGIC);
end pmos;
architecture Behavioral of pmos is
begin
D1<=S1 when G1='0' else 'Z';
end Behavioral;
```

VHDL for NMOS

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity nmos is
Port ( S1,G1 : in  STD_LOGIC;
        D1 : out  STD_LOGIC);
end nmos;
architecture Behavioral of nmos is
begin
D1<=S1 when G1='1' else 'Z';
end Behavioral;
```

Example 5.21: *Write the VHDL program for 4 to 2 Encoder using dataflow modeling.*

Solution:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity encoder4to2 is
   Port ( I0,I1,I2,I3 : in  STD_LOGIC;
        E0,E1 : out  STD_LOGIC);
end encoder4to2;
architecture encoder of encoder4to2 is
begin
E0<= I1 or I3;
E1<= I2 or I3;
end encoder;
```

Example 5.22: *Write the VHDL program for 2 to 4 decoder using dataflow modeling.*

Solution:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity decoder2to4 is
  Port ( I1,I0 : in  STD_LOGIC;
       D0,D1,D2,D3 : out  STD_LOGIC);
end decoder2to4;
architecture decoder of decoder2to4 is
begin
D0<= (not I1) and (not I0);
D1<= (not I1) and I0;
D2<= I1 and (not I0);
D3<= I1 and I0;
end decoder;
```

5.5 VHDL FOR SEQUENTIAL LOGIC CIRCUITS

Example 5.23: Write the VHDL code for SR flip-flop using behavioral modeling

Solution:

S	R	Q_{n+1}	State
0	0	Q_n	No change
0	1	0	Reset
1	0	1	Set
1	1	X	Indeterminate

Table 5.3 Truth table of SR flip-flop

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity srff is
Port ( S,R,CLK : in  STD_LOGIC;
       Q : inout  STD_LOGIC);
end srff;
architecture Behavioral of srff is
begin
process(CLK,S,R)
begin
```

```
if(CLK'event and CLK='1') then
if(S='0' and R='0')then
Q<=Q;
elsif(S='0' and R='1') then
Q<='0';
elsif(S='1' and R='0') then
Q<='1' ;
else null;
end if;
end if;
end process;
end Behavioral;
```

Example 5.24: Write the VHDL code for JK flip-flop using behavioral modeling

Solution:

J	K	Q_{n+1}	State
0	0	Q_n	No change
0	1	0	Reset
1	0	1	Set
1	1	$\overline{Q}_n$	Toggles

Table 5.4 Truth table of JK flip-flop

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity jkff is
Port ( J,K,CLK : in  STD_LOGIC;
       Q: inout  STD_LOGIC);
end jkff;
architecture Behavioral of jkff is
begin
process(CLK,J,K)
begin
if(CLK'event and CLK='1') then
if(J='0' and K='0')then
Q<=Q;
elsif(J='0' and K='1') then
Q<='0';
elsif(J='1' and K='0') then
Q<='1' ;
elsif(J='1' and K='1') then
Q<=(not Q) ;
end if;
end if;
end process;
end Behavioral;
```

Example 5.25: Write the VHDL code for D flip-flop using behavioral modeling

Solution:

D	Q_{n+1}	State
0	0	Reset
1	1	Set

Table 5.5 Truth table of D flip-flop

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity dff is
   Port ( D,CLK : in  STD_LOGIC;
          Q : out  STD_LOGIC);
end dff;
architecture Behavioral of dff is
begin
process(CLK,D)
begin
if(CLK' event and CLK='1')then
if(D='0') then
Q<='0';
else
Q<='1';
end if;
end if;
end process;
end Behavioral;
```

Example 5.26: Write the VHDL code for T flip-flop using behavioral modeling

Solution:

T	Q_{n+1}	State
0	Q_n	No change
1	$\overline{Q}_n$	Toggles

Table 5.6 Truth table of T flip-flop

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity tff is
Port ( T,CLK : in  STD_LOGIC;
       Q : inout  STD_LOGIC);
end tff;
architecture Behavioral of tff is
begin
process(CLK,T)
begin
if(CLK' event and CLK='1')then
if(T='0') then
Q<=Q;
else
Q<= (not Q);
end if;
end if;
end process;
end Behavioral;
```

Example 5.27: *Write the VHDL code for 4-bit up-counter using behavioral modeling*

Solution:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity upcount1 is
  Port ( clk : in  STD_LOGIC;
       count : out  STD_LOGIC_VECTOR (3 downto 0));
end upcount1;
architecture Behavioral of upcount1 is
signal temp: STD_LOGIC_VECTOR (3 downto 0):="0000";
begin
process(clk)
begin
if(clk='1') then
temp<=temp+ "0001";
end if;
count<=temp;
end process;
end Behavioral;
```

Example 5.28: *Write the VHDL code for 4-bit down counter using behavioral modeling*

Solution:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity downcount1 is
  Port ( clk : in  STD_LOGIC;
       count : out  STD_LOGIC_VECTOR (3 downto 0));
end downcount1;
architecture Behavioral of downcount1 is
signal temp: STD_LOGIC_VECTOR (3 downto 0):="1111";
begin
```

```
process(clk)
begin
if(clk='1') then
temp<=temp- "0001";
end if;
count<=temp;
end process;
end Behavioral;
```

Example 5.29: Write the VHDL code for decade counter using behavioral modeling

Solution:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity decade_counter is
port(
      reset : in STD_LOGIC;
      clk : in STD_LOGIC;
      Qout : out STD_LOGIC_VECTOR(3 downto 0)
      );
end decade_counter;
architecture counter of decade_counter is
begin
count : process (reset,clk) is
  variable m : std_logic_vector (3 downto 0) := "0000";
  begin
    if (reset='1') then
      m := "0000";
    elsif (rising_edge (clk)) then
      m := m + 1;
    end if;
    if (m="1010") then
      m := "0000";
    end if;
    Qout <= m;
  end process count;
end counter;
```

Example 5.30: Write the VHDL code for serial-in serial-out shift register using structural modeling

Solution:

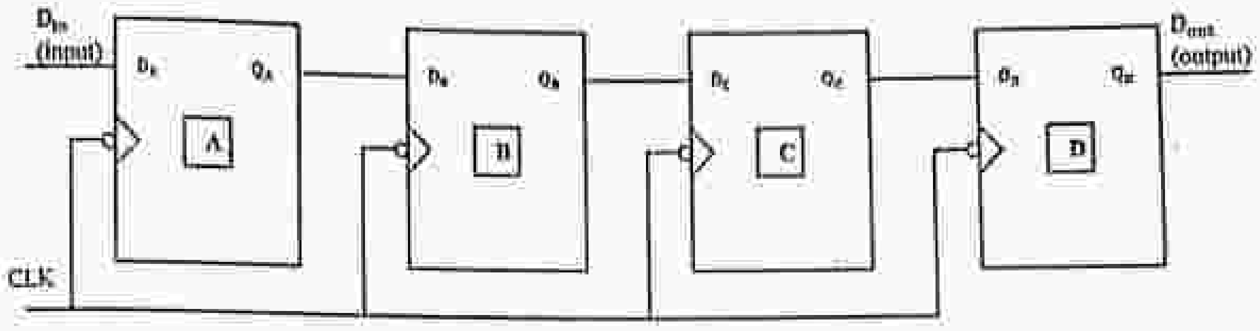

Figure 5.16 Serial-in Serial-Out shift register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity siso is
port(Din,CLK:in std_logic;
Dout:out std_logic);
end siso;
architecture register of siso is
signal QA,QB,QC:std_logic;
component dff is
port(D,CLK: in std_logic;
Qn:out std_logic);
end component;
begin
ff1: dff port map(Din,CLK,QA);
ff2:dff port map(QA,CLK,QB);
ff3:dff port map(QB,CLK,QC);
ff4:dff port map(QC,CLK,Dout);
end register;
```

where the VHDL for D flip-flop (dff) is shown in example 5.25.

5.5.1 Finite state Machine (FSM)

The finite state machine is used to design a sequential logic circuit. The finite state machine have finite number of user defined states. The machine is in only one state at a time. The state at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition. The state that occurs after triggering event or condition is called next state. The following example shows the VHDL program for a state machine.

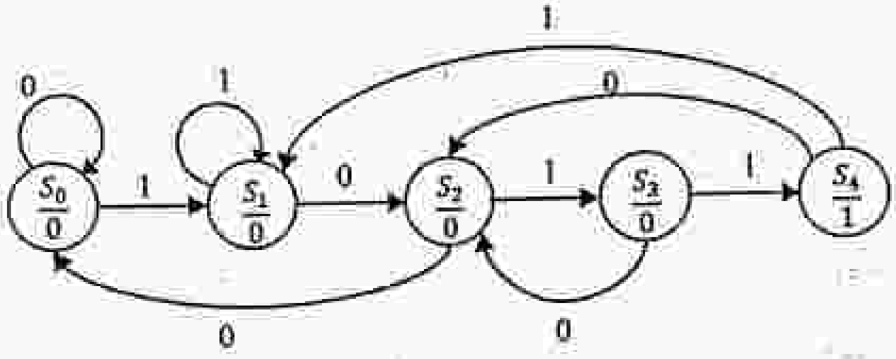

Figure 5.17

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity seq1 is
   Port ( clk : in  STD_LOGIC;
        s_in : in  STD_LOGIC;
        s_out : out  STD_LOGIC);
end seq1;
architecture Behavioral of seq1 is
type state_type is (s0,s1,s2,s3,s4);
signal current_state,Next_state: state_type;
begin
process(clk)
begin
if(clk='1')then
          current_state<=next_state;
end if;
end process;
process (current_state)
begin
case current_state is
when s0=>
          s_out<='0';
        if(s_in='0') then
        Next_state <=s0;
        else
        Next_state<=s1;
        end if;
when s1=>
        s_out<='0';
        if(s_in='0') then
        Next_state <=s2;
        else
        Next_state<=s1;
        end if;
when s2=>
        s_out<='0';
        if(s_in='0') then
        Next_state <=s3;
        else
        Next_state<=s0;
        end if;
when s3=>
        if(s_in='0') then
        Next_state <=s2;
        s_out<='0';
        else
        Next_state<=s4;
        s_out<='1';

        end if;
when s4=>
        s_out<='0';
        if(s_in='0') then
        Next_state <=s2;
        else
        Next_state<=s1;
        end if;
when others=>
        NULL;
end case;
end process;
end Behavioral;
```

Example 5.31: ***Write the VHDL code for the given state diagram, using behavioral modeling. Design it using one-hot state assignment and implement it using programmable Array logic (PAL).***

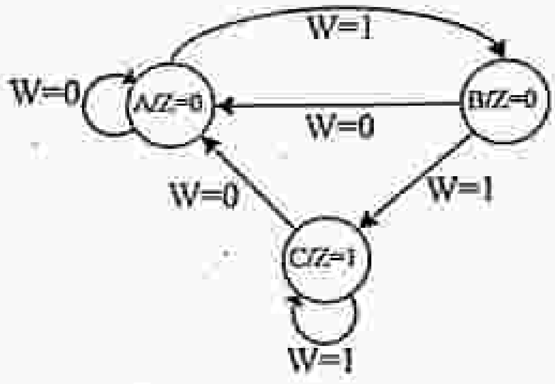

Figure 5.18

Solution:

VHDL code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity seq1 is
   Port ( clk : in  STD_LOGIC;
          W : in  STD_LOGIC;
          Z : out STD_LOGIC);
end seq1;
architecture Behavioral of seq1 is
type state_type is (A,B,C);
signal current_state,Next_state: state_type;
begin
process(clk)
begin
if(clk='1')then
current_state<=next_state;
end if;
end process;
process (current_state)
begin
case current_state is
when A=>
if(W='0') then
Next_state <=A;
Z<='0';
else
Next_state<=B;
Z<='0';
end if;
when B=>
if(W='0') then
Next_state <=A;
Z<='0';
else
Next_state<=C;
Z<='1';
end if;
when C=>
if(W='0') then
Next_state <=A;
Z<='0';
else
Next_state<=C;
Z<='1';
end if;
when others=>
NULL;
end case;
end process;
end Behavioral;
```

Implementation using PAL

Step 1: Draw the state table

Present state	Next state		Output	
	W=0	W=1	W=0	W=1
A	A	B	0	0
B	A	C	0	1
C	A	C	0	1

Table 5.7 State table

Step 2: Reduce the number of states if possible

Here states 'B' and 'C' have same next states and Output. So the state 'B' and 'C' can be reduced to a single state 'B'.

Present state	Next state		Output	
	W=0	W=1	W=0	W=1
A	A	B	0	0
B	A	B	0	1

Table 5.8 Reduced State table

Step 3: Assign binary values to the states and plot the transition table by choosing the type of Flip-flop. Use the following T Flip-flop excitation table to find the value of T

A	A^+	T
0	0	0
0	1	1
1	0	1
1	1	0

Table 5.9 Excitation table of T flip-flop

Present state	Input	Next state	Flip-flop inputs	Output
a	W	a^+	T	Z
0	0	0	0	0
0	1	1	1	0
1	0	0	1	0
1	1	1	0	1

Table 5.10 Transition table

Step 4: Derive the Flip-flop input equations and output equations using K-map

K-map for T

a \ W	$\overline{W}$ 0	W 1
$\bar{a}$ 0	0 $_0$	1 $_1$
a 1	1 $_2$	0 $_3$

$T = \bar{a}W + a\overline{W}$

K-map for Z

a \ W	$\overline{W}$ 0	W 1
$\bar{a}$ 0	0 $_0$	0 $_1$
a 1	0 $_2$	1 $_3$

$Z = aW$

PAL program table

Product term	AND inputs		Outputs
	W	a	
$\bar{a}W$	1	0	$T = \bar{a}W + a\overline{W}$
$a\overline{W}$	0	1	
aW	1	1	$Z = aW$
-	-	-	

Table 5.11

Logic diagram

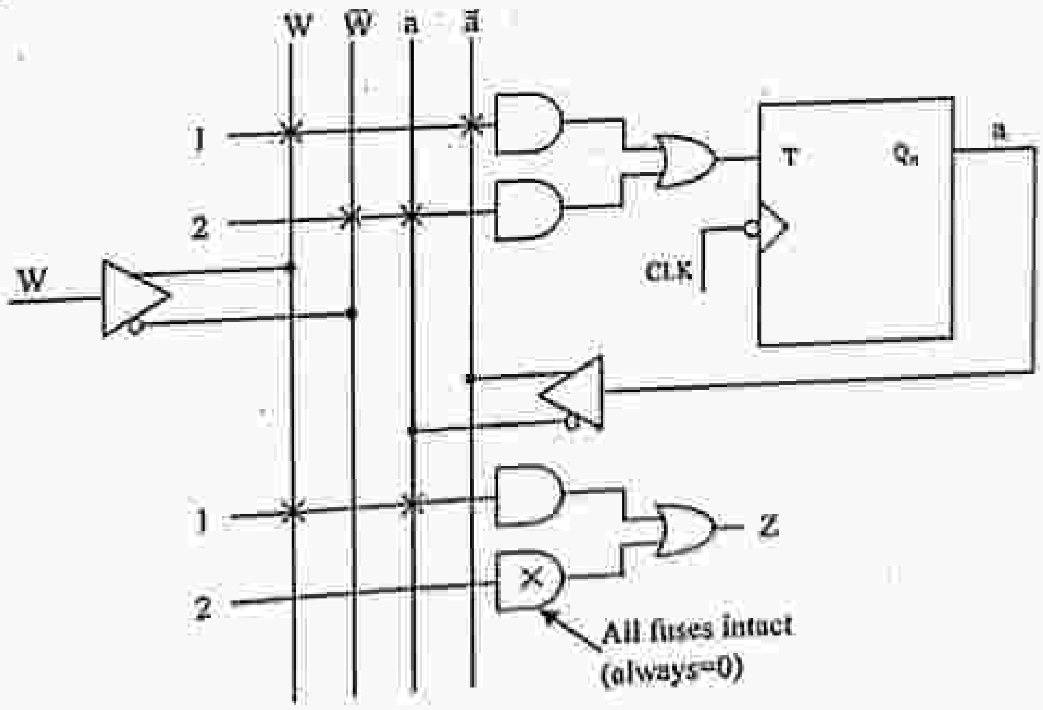

Figure 5.19 PAL

5.6 OPERATORS

There are various built-in operators in VHDL. The different types of operators are given below.

1. Logical operators
2. Relational operators
3. Shift operators.
4. Arithmetic operators
5. Concatenation operator
6. Miscellaneous operators.

5.6.1 Logical operators

The logical operators NOT, AND, OR, NAND, NOR and XOR can be used with any bit type or bit_vectors. When these operators are used on bits, they have their usual meaning. For example, if a=1 and b=0, then a xor b will give a value '1'. If logical operators are used as bit_vectors, the bit vectors must have the same number of elements and the operation is performed bitwise. For example if a=1010 and b=1111, then a xor b will be '0101'

(eg) and , or, not, nand, nor, xor, xnor

```
y<=a and b
y<=a or b
y<=(not  b)
y<=a nand b
y<=a nor b
y<=a xor b
y<=a xnor b
```

5.6.2 Relational operators

The relational operators =, /=, <, <= ,> , >= have their usual meanings. The result of all these operator is a Boolean value (TRUE or FALSE). The arguments to the = and /= operators may be of any type. But the arguments of the <,<=,> and >= operators may be any scalar type (integer, real and physical) or the bit_vector type. If the argument are bit_vectors, then the arguments must be the same length and result is TRUE only if the relation is true for each corresponding elements of the array.

=	Equal to
/=	Not equal to
<	Less than
>	Greater than
>=	Greater than or equal to

Table 5.12

5.6.3 Shift operators

Shift operator performs shift operation such as left shift, right shift, rotate right and rotate left. The operations of shift operations are described below.

"1001" sll 2 => 0100
"1001" srl 2 => 0010
"0101" sla 2 => 0111
"1010" sra 2 => 1110
"0111" rol 2 => 1101
"1011" ror 2 => 1110

sll	Shift left logical
srl	Shift right logical
sla	Shift left arithmetic
sra	Shift right arithmetic
rol	Rotate left
ror	Rotate right

Table 5.13

An example for shift left operation is shown below

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
entity shifting is
   Port ( a : in  unsigned(3 downto 0);
          b : out  unsigned(3 downto 0));
end shifting;
architecture Behavioral of shifting is
begin
b<= a sll 2;
end Behavioral;
```

5.6.4 Arithmetic operators

The typical arithmetic operators are + (addition), - (subtraction), * (multiplication) and / (division). Although these operators are not built-in for bit_vectors, they are used with bit_vectors by interpreting them as a binary representation of integers, which may be added, subtracted, multiplied or divided

+	Addition
-	Subtraction
*	Multiplication
/	Division

Table 5.14

An example for arithmetic addition operation is shown below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity addition is
 Port ( a,b : in  unsigned(3 downto 0);
       c : out  unsigned(3 downto 0));
end addition;
architecture Behavioral of addition is
begin
c<=a+b;
end Behavioral;
```

5.6.5 Concatenation operator

The concatenation operator is a built in VHDL operator that performs the concatenation of bit_vectors. For example if a=1010 and b=1101, then the concatenation operation will give a 8 bit number '10101101' such that 'a' is on the left half and 'b' is on the right half.

An example for concatenation operator is shown below

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity addition is
 Port ( a,b : in  unsigned(3 downto 0);
       c : out  unsigned(7 downto 0));
end addition;
architecture Behavioral of addition is
begin
c<=a & b;
end Behavioral;
```

5.6.6 Miscellaneous operators

The typical miscellaneous operator includes, abs (absolute), ** Exponentiation, mod (Modulus), rem (remainder) etc.

abs	Absolute
**	Exponentiation
mod	Modulus
rem	remainder

Table 5.15

5.7 PACKAGES

A package is used to provide a convenient method to store and share declarations that are common for many design units.

It is represented by

- A package declaration and
- A package body

5.7.1 Package declarations

- It contains a set of declarations that may be shared by various design units. It defines items which are made visible to other design units.
- A package body contains the hidden details of a package.

Syntax of package declaration

```
Package package name is
Package item declaration
    -   Sub program declaration
    -   Type declaration
    -   Sub type declaration
    -   Constant declaration
    -   Signal declaration
    -   Variable declaration
    -   File declaration
    -   Component declaration
    -   Attribute declaration
    -   Attribute specifications
    -   Disconnection specifications
    -   Use clauses
end [package] [package name];
```

(eg)

```
        Package example is
                Constant L2H : TIME : =60NS
                type output is (ADD,SUB);
                ______
                ______
                ______
            Component NOR2
                Port (A,B; in; c; out);
            end component;
        end example;
```

These can be accessed by other design units by using the library and use clauses.

Use work.example.all; → It includes all declarations from package example

```
package prog1 is
constant delay: Time; deferred constant
    ______
    ______
    ______
end package prog1;
```

5.7.2 Package Body

It contains the behavior of the subprograms and the values of the deferred constants which are declared in a package declaration. The syntax of a package body is given below.

```
Package body package name is
        Package body item declaration
            -   Subprogram bodies
            -   Complete constant declarations
            -   Subprogram declarations
            -   Type and subtype declarations
            -   File declaration
            -   Use clauses
        end [package body] [package name];
```

The package name must be the same name of its corresponding package declaration. Prog1 is the name given in the last example.

```
        Package body prog1 is
        Use WORK.TABLES.all'
        constant Delay :-Time:=20ns
        function
        begin
        =
        end "and";
```

The package body is used to store private declarations.

A package declaration is used to store public declarations that can be accessed by other units.

5.8 SUBPROGRAMS

Procedures and function are the two types of subprograms.

Functions :

These are used for computing a single value. It executes in zero simulation time.

Procedures:

These are used to partition large behavioral description. Procedures can return zero or more values. It may or may not execute in zero simulation time. It depends whether the wait statement is used or not.

The format for subprogram is

```
Subprogram specify is
Subprogram item declarations
begin
subprogram statements
        ______
        ______
        ______
end [function/procedure] [subprogram name];
```

subprogram specification is used to specify the name of a subprogram and it defines the format parameter names, their class. (eg signal, variable, constant) and their mode (in, out, inout).

Parameters of 'in' mode are read only parameters, parameters of 'out' mode are write only parameters, parameters of 'inout' mode can be read and write.

Actuals

Actual is a subprogram call is used to pass the values from and to a subprogram

If the parameters of 'variable' or 'constant' class are used, values are passed to the subprogram by value.

Arrays may or may not be passed by reference.

Files are passed by reference.

Subprogram item declaration

It contains a set of declarations that are accessible for used within a subprogram. Variables are created and initialized every time, when the subprogram is called.

Subprogram statements

It contains sequential statements which can define the computation to be performed by the subprogram.

Return statement is allowed only within the subprogram

return [expression]

Return statement is used to terminate the subprogram and control is returned to the calling program.

Subprogram Name

It is available at the end of a subprogram body.

Functions:

These are used o describe the sequential algorithms which are frequently used. Its syntax is

[pure/impure] function function name [parameter list, return return_type]

Pure function

The function that returns the same value each time it is called as pure function

Impure function

The function which potentially returns different values each time it is called as impure function.

Parameter list

It is used to describe the list of formal parameters

Function call

function name (list of actuals)

Procedures

It permits decomposition of large behaviors into modular sections. It can return zero or more values using parameters of out, inout mode. Its syntax is

Procedure procedure name (parameter list)

The syntax of a procedure call is

[Label :] procedure name (list of actual)

5.9 TESTBENCH

The stimulus that tests the functionality of the design is called a testbench. The testbench is also written in HDL. Thus after simulating the design, the functionality of the design is tested using a testbench. The operation of digital circuit that has been designed can be verified fast and accurately by functional and timing simulation. In the simulation stage the design errors and incompatibility of components can be detected. Simulating a design requires generation of test data and observation of simulation results. This process can be done by use of VHDL module that is referred as a testbench.

In functional simulation, the circuit logical operation is studied by deriving the truth table of the circuit independent of timing considerations.

In timing simulation, the circuit operation is studied by considering the timing behavior of the circuit.

- A testbench is used to verify the functionality of a design.
- A testbench is at the highest level in the hierarchy of the design. So a testbench is a model which is used to exercise and verify the correctness of a hardware model.
- The testbench provides the necessary input stimulus to the (Design under test) DUT and examines the output from DUT.

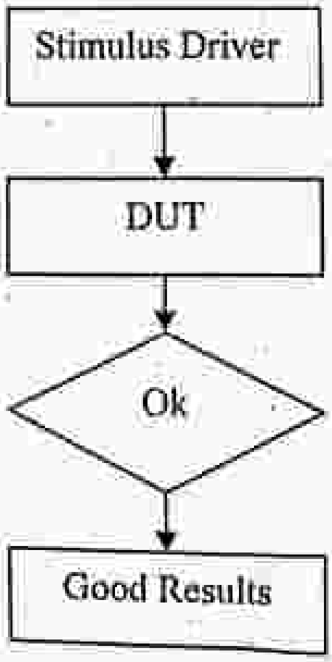

Figure 5.20

The testbench is used for

- Generating stimulus for simulation.
- Applying the stimulus to the DUT and collect the output.
- Comparing obtained output with expected output.

5.9.1 Types of Testbenches

The testbenches are classified into

1. Stimulus only
2. Full testbench
3. Simulator specific
4. Hybrid testbench
5. Fast testbench

1. Stimulus only

- The stimulus only testbench contains the stimulus driver and DUT blocks of a testbench.
- The testbench won't verifies the results.

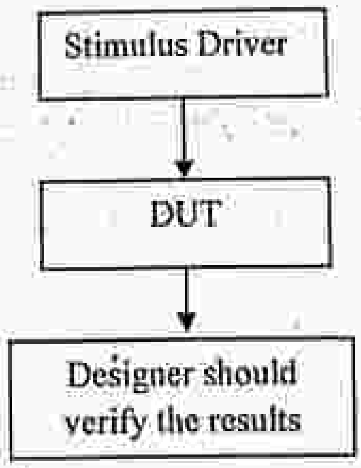

Figure 5.21

2. Full testbench

- A full testbench is similar to a stimulus only testbench except that the full testbench also includes the capability to check the output of the DUT.
- Full testbench generate stimulus to DUT and then verify the results.

3. Simulator specific

- The simulator specific testbench is written specifically for one brand of simulator. Most simulators include a command language that allows the designer to control the simulator.

4. Hybrid testbenches

- Hybrid testbenches do not utilize only one technique, but a combination of a number of techniques.

5. Fast testbench

- Speed of simulation is an important factor in all testbench. This is because how fast a simulation can run. The testbenches consumes more time to read data from vector files, because those files are very large. To avoid such problems a designer can use fast testbenches.

The testbench format is shown below

```
entity testbench1 is
end;
architecture testbench2 of testbench1 is
component test
port (port names and modes)
end component;
local signal declarations;
begin
____

____
Port map (port associations);
end testbench2;
```

5.9.2 Waveform generation

Two methods can be followed to generate waveforms

1. Repetitive pattern
2. Non-Repetitive pattern or vector pattern

rand
that

t a

s is
nes
ery

1. **Repetitive pattern**

Repetitive pattern waveform can be generated by using the following statement.

If clk<=not clk after 10 ns

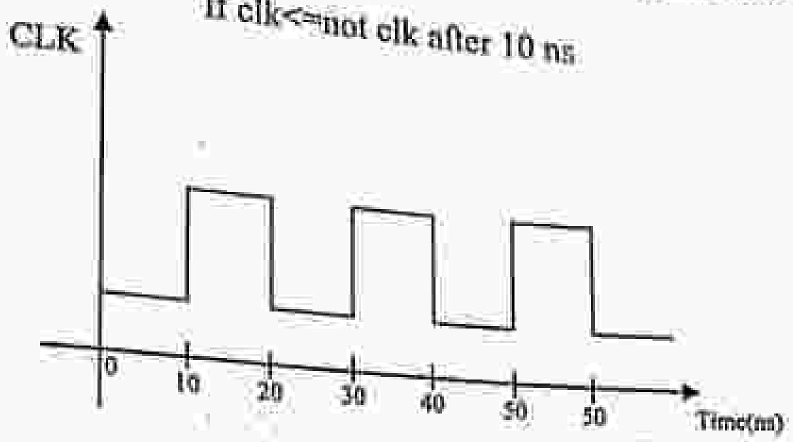

Figure 5.22

Here ON period and OFF period are same

(eg)

```
Process
constant OFF period;  TIME; =10ns;
constant ON period; TIME; =5ns;
begin
wait for OFF period;
clk <='1';
wait for ON period;
clk<='0';
end process
```

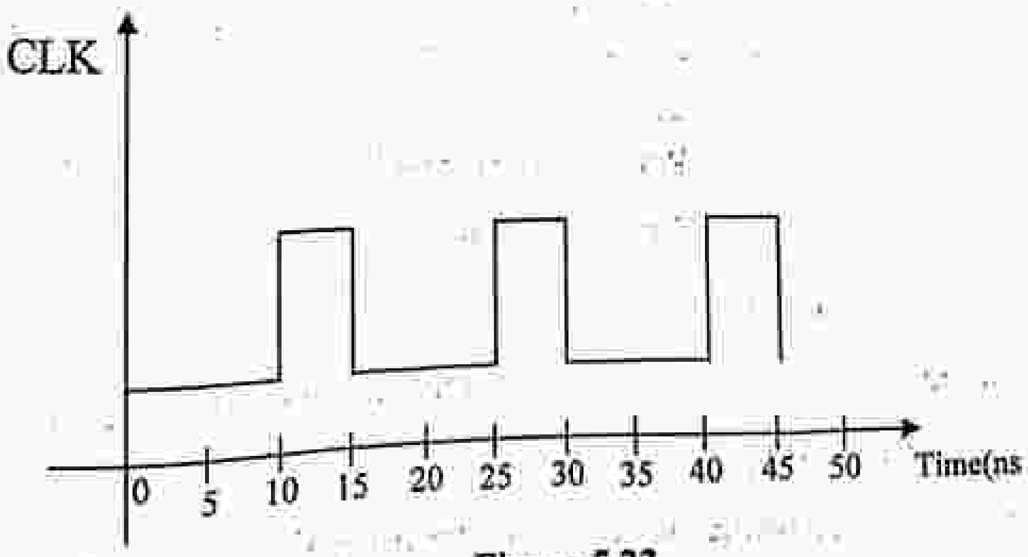

Figure 5.23

2. Non repetitive waveform

➢ Non repetitive waveform can be generated by using following statement.

Clk<= '0', '1' after 50ns, '0' after 80ns, '1' after 100 ns

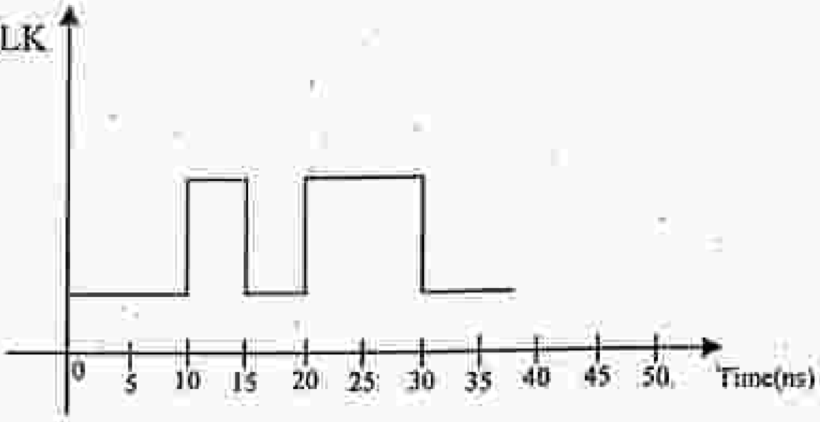

Figure 5.24

The testbench program of half adder is shown below.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
ENTITY HA_testbench_vhd IS
END HA_testbench_vhd;
ARCHITECTURE behavior OF HA_testbench_vhd IS
        -- Component Declaration for the Unit Under Test (UUT)
        COMPONENT halfadder
        PORT(
                a : IN std_logic;
                b : IN std_logic;
                sum : OUT std_logic;
                carry : OUT std_logic
                );
        END COMPONENT;
        --Inputs
        SIGNAL a :  std_logic := '0';
        SIGNAL b :  std_logic := '0';
        --Outputs
        SIGNAL sum : std_logic;
        SIGNAL carry : std_logic;
BEGIN
        -- Instantiate the Unit Under Test (UUT)
        uut: halfadder PORT MAP(
                a => a,
                b => b,
                sum => sum,
                carry => carry
        );
        tb : PROCESS
        BEGIN
        a<='0';
        b<='0';
                wait for 100 ns;
        a<='0';
        b<='1';
                wait for 100 ns;
        a<='1';
        b<='0';
                wait for 100 ns;
        a<='1';
        b<='1';
                wait for 100 ns;
                wait;
        END PROCESS;
END;
```

Example 5.32: Write the testbench program for 4-bit binary parallel adder

Solution:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
ENTITY adder_test_vhd IS
END adder_test_vhd;
ARCHITECTURE behavior OF adder_test_vhd IS
```

```
        COMPONENT adder4bit
        PORT(
                A : IN std_logic_vector(3 downto 0);
                B : IN std_logic_vector(3 downto 0);
                Cin : IN std_logic;
                S : OUT std_logic_vector(3 downto 0);
                Cout : OUT std_logic
                );
        END COMPONENT;
        SIGNAL Cin :  std_logic := '0';
        SIGNAL A :  std_logic_vector(3 downto 0) := (others=>'0');
        SIGNAL B :  std_logic_vector(3 downto 0) := (others=>'0');
        --Outputs
        SIGNAL S :  std_logic_vector(3 downto 0);
        SIGNAL Cout :  std_logic;
BEGIN
        uut: adder4bit PORT MAP(
                A => A,
                B => B,
                Cin => Cin,
                S => S,
                Cout => Cout
        );
        tb : PROCESS
        BEGIN
        a<="0000";
        b<="0101";
        wait for 100 ns;
        a<="1100";
        b<="0100";
        wait for 100 ns;
        a<="1111";
        b<="0000";
        wait for 100 ns;
        a<="0010";
        b<="1101";
        wait for 100 ns;
        END PROCESS;
END;
```