LAB MANUAL

D.S.D.

DIGITAL SYSTEM DESIGN LAB

L T P	**CLASS WORK : 25**
0 0 2	**EXAM :25**
	TOTAL :50
	DURATION OF EXAM :3 Hrs

LIST OF EXPERIMENTS

1. Design all the gates using VHDL.

2. Write VHDL programs for the following circuits, check the waveforms generated.
   (a) Half-adder
   (b) Full-adder
3. Write VHDL programs for the following circuits, check the waveforms generated.
   (a) Multiplexer
   (b) Demultiplexer
4. Write VHDL programs for the following circuits, check the waveforms generated.
   (a) Decoder
   (b) Encoder
5. Write VHDL programs for the comparator, check the waveforms generated.

6. Write VHDL programs for the code converter, check the waveforms generated.

7. Write VHDL programs for the Flip-flop, check the waveforms generated.

8. Write VHDL programs for the counter, check the waveforms generated.

9. Write VHDL programs for the following circuits, check the waveforms generated.
   (a) Register
   (b) Shift register

10. Implement any three (given above) on FPGA/ CPLD kit.

Rational behind VHDL Lab

VHDL is VHSIC (Very High Speed Integrated Circuits) Hardware Description Language .VHDL is designed to describe the behavior of the digital systems It is a design entry language .VHDL is concurrent .Using VHDL test benches, we can verify our design. VHDL integrates nicely with low level design tools. It is IEEE standard (1076 and 1164).VHDL has hierarchical design units. Learning VHDL is easy.

A structure level description defines the circuit in terms of a collection of components. It is a language which provides us a mechanism to model the digital circuits without the requirement of IC's and hardware. It enables us to write the code for the digital circuits and useful in designing of the complex VLSI's. VHDL supports behavioral, structural descriptions, thus supporting various levels of abstraction.

Hardware Requirements

Pentium IV Processor
128MB RAM

Software Requirements

Operating System: Windows 95 Onwards
Language: Active_HDL 7.2

An Introduction and Background

VHDL is an acronym which stands for VHSIC Hardware Description Language. VHSIC is yet another acronym which stands for Very High Speed Integrated Circuits. If you can remember that, then you're off to a good start. The language has been known to be somewhat complicated, as its title. The acronym does have a purpose, though; it is supposed to capture the entire theme of the language, that is to describe hardware much the same way we use schematics.

VHDL can wear many hats. It is being used for documentation, verification, and synthesis of large digital designs. This is actually one of the key features of VHDL, since the same VHDL code can theoretically achieve all three of these goals, thus saving a lot of effort. In addition to being used for each of these purposes, VHDL can be used to take three different approaches to describing hardware. These three different approaches are the structural, data flow, and behavioral methods of hardware description. Most of the time a mixture of the three methods is employed. The following sections introduce you to the language by examining its use for each of these three methodologies. There are also certain guidelines that form an approach to using VHDL for synthesis, which is not addressed by this tutorial.

VHDL is a standard (VHDL-1076) developed by IEEE (Institute of Electrical and Electronics Engineers). The language has been through a few revisions, and you will come across this in the VHDL community. Currently, the most widely used version is the 1987 (std 1076-1987) version, sometimes referred to as VHDL'87, but also just VHDL. However, there is a newer revision of the language referred to as VHDL'93. VHDL'93 (adopted in 1994 of course) is fairly new and is still in the process of replacing VHDL'87.

Descriptions

To make designs more understandable and maintainable, a design is typically decomposed into several blocks. These blocks are then connected together to form a complete design. Using the schematic capture approach to design, this might be done with a block diagram editor. Every portion of a VHDL design is considered a block. A VHDL design may be completely described in a single block, or it may be decomposed in several blocks. Each block in VHDL is analogous to an off-the-shelf part and is called an entity. The *entity* describes the interface to that block and a separate part associated with the entity describes how that block operates. The interface description is like a pin description in a data book, specifying the inputs and outputs to the block. The description of the operation of the part is like a schematic for the block. For the remainder of the

tutorial we will refer to a block as a design, even though a complete design may be a collection of many blocks interconnected.

The following is an example of an *entity declaration* in VHDL.

```
entity latch is
  port (s,r: in bit;
        q,nq: out bit);
end latch;
```

The first line indicates a definition of a new entity, whose name is *latch*. The last line marks the end of the definition. The lines in between, called the *port clause*, describe the interface to the design. The port clause contains a list of interface declarations. Each *interface declaration* defines one or more signals that are inputs or outputs to the design.

The second part of the description of the latch design is a description of how the design operates. This is defined by the architecture declaration. The following is an example of an *architecture declaration* for the latch entity.

```
architecture dataflow of latch is
  signal q0 : bit := '0';
  signal nq0 : bit := '1';
begin
  q0<=r nor nq0;
  nq0<=s nor q0;

  nq<=nq0;
  q<=q0;
end dataflow;
```

The first line of the declaration indicates that this is the definition of a new architecture called *dataflow* and it belongs to the entity named latch. So this architecture describes the operation of the latch entity. The lines in between the begin and end describe the latch's operation..

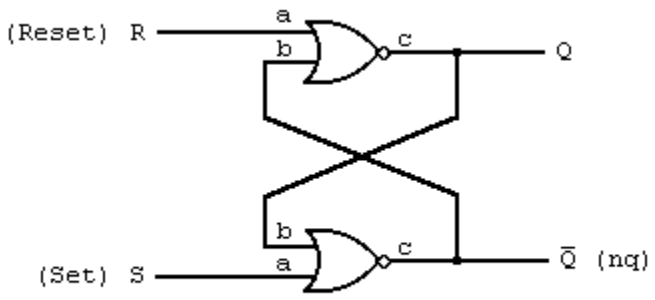
Connecting Blocks

Once we have defined the basic building blocks of our design using entities and their associated architectures, we can combine them together to form other designs. This section describes how to combine these blocks together in a structural description.

Let's specify the operation of the latch entity used in the previous section by connecting some previously defined entities. The entity declaration for the latch was:

```
entity latch is
  port (s,r: in bit;
        q,nq: out bit);
end latch;
```

We will declare an architecture different from the one in the last section that demonstrates the structural approach. To do so, we assume that an entity named *nor_gate* has been defined that will be used in the design. The schematic for the latch might be



SR Latch

We can specify the same connections that occur in the schematic using VHDL with the following architecture declaration:

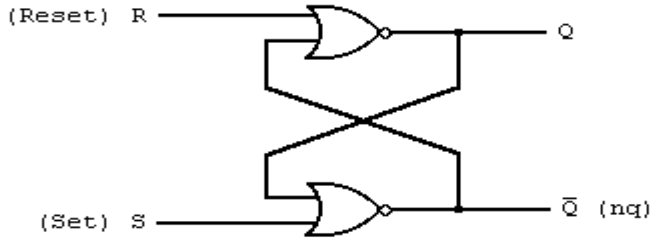
```
architecture structure of latch is
  component nor_gate
    port (a,b: in bit;
          c: out bit);
  end component;
begin
  n1: nor_gate
    port map (r,nq,q);
  n2: nor_gate
    port map (s,q,nq);
end structure;
```

The lines between the first and the keyword *begin* are a *component declaration*. It describes the interface of the entity *nor_gate* that we would like to use as a component in (or part of) this design. Between the *begin* and *end* keywords, the first two lines and second two lines define two *component instances*.

A First Example

In the data flow approach, circuits are described by indicating how the inputs and outputs of built-in primitive components (ex. an *and* gate) are connected together. In other words we describe how signals (data) flow through the circuit. Let's look at the first example.

Suppose we were to describe the following SR latch using VHDL as in the following



SR Latch

schematic.

We might build an entity like the one that follows.

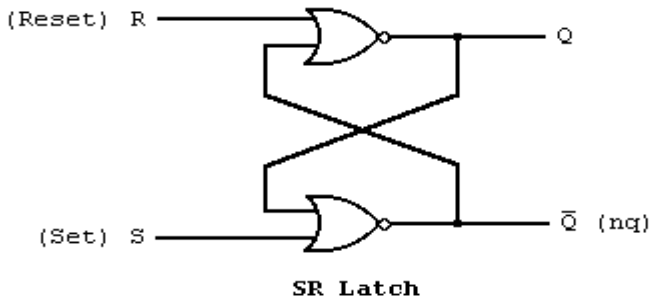
```
entity latch is
  port (s,r : in bit;
        q,nq : out bit);
end latch;

architecture dataflow of latch is
begin
  q<=r nor nq;
  nq<=s nor q;
end dataflow;
```

As we saw in the last section, the entity describes the interface to the design. There are four signals *s,r,q,* and *nq* that are accessible externally to the design. Again we model the signals in our design with the VHDL data type *bit*, which can represent two level logic values. A signal assignment statement describes how data flows from the signals on the right side of the <= operator to the signal on the left side. The right side of the <= operator is called an expression..

How it Works

The scheme used to model a VHDL design is called *discrete event time simulation*. When the value of a signal changes, we say an *event* has occurred on that signal. If data flows from signal A to signal B, and an event has occurred on signal A (i.e. A's value changes), then we need to determine the possibly new value of B. This is the foundation of the discrete event time simulation. The values of signals are only updated when certain events occur and events occur at discrete instances of time.

The following is a schematic version of the SR latch.



SR Latch

The internal operation of the latch was essentially captured using the following two statements.

```
q<=r nor nq;
nq<=s nor q;
```

Since data flows from *r* and *nq* to *q*, we say that *q* depends on *r* and *nq*. In general, given any signal assignment statement, the signal on the left side of the <= operator depends on all the signals appearing on the right side. to update *q*.

Other Types

In the previous sections all of the signals in the examples have been of the type bit. VHDL provides several other types, some of which are described here. Often times we use several bit signals together to represent a binary number in a design. VHDL provides a mechanism for defining new types which represent a collection of several data items of the same type. These kinds of types are called *arrays*. There is a predefined array type called *bit_vector* which represents a collection of bits. The following example demonstrates how the bit_vector type can be used to define a 1-to-4-line demultiplexer.

```
entity demux is
  port (e: in bit_vector (3 downto 0);     -- enables for each output
        s: in bit_vector (1 downto 0);     -- select signals
        d: out bit_vector (3 downto 0));   -- four output signals
end demux;

architecture rtl of demux is
  signal t : bit_vector(3 downto 0);       -- an internal signal
begin
  t(3)<=s(1) and s(0);
  t(2)<=s(1) and not s(0);
  t(1)<=not s(1) and s(0);
  t(0)<=not s(1) and not s(0);
  d<=e and t;
end rtl;
```

Other Operators

The previous sectioned mentioned a few different types that are available in VHDL. There are also several built-in operators that can be used with those types. This section mentions some of these.

The logical operators NOT, AND, OR, NAND, NOR, and XOR can be used with any bit type or bit_vector. When used as operators on bits they have their usual meaning. When used with bit_vectors, the bit_vectors must have the same number of elements, and the operation is performed bitwise. For example, "00101001" xor "11100101" results in "11001100".

The typical algebraic operators are available for integers, such as +,-,* (multilication), and / (division). Although these operations are not built-in for bit_vectors, they are often provided in libraries that come with your VHDL software. They are used with bit_vectors by interpreting them as a binary representation of integers, which may be added, subtracted, multiplied, or divided.

Also predefined are the normal relational operators. They are =, /=, <, <=, > and >= and have their usual meanings (/= denotes the not equal operator). The result of all these operators is a boolean value (TRUE or FALSE). The arguments to the = and /= operators may be of any type. The arguments of the <, <=, > and >= operators may be any scalar type (integer, real, and physical types) or the bit_vector type. If the arguments are bit_vectors, then the arguments must be the same length and the result is TRUE only if the relation is true for each corresponding element of the array arguments.

The & operator is a built-in VHDL operator that performs the concatenation of bit_vectors. For example, with the following declarations:

```
signal a: bit_vector (1 to 4);

signal b: bit_vector (1 to 8);
```

The following statement would connect *a* to the right half of *b* and make the left half of *b* constant '0'.

```
b<="0000" & a;
```

The & appends the *a* to the end of the "0000" to form a result that contains 8 bits.

The Process Statement

The behavioral approach to modeling hardware components is different from the other two methods in that it does not necessarily in any way reflect how the design is implemented. It is basically the black box approach to modeling. It accurately models what happens on the inputs and outputs of the black box, but what is inside the box (how it works) is irrelevant.

Behavioral descriptions are supported with the process statement. The process statement can appear in the body of an architecture declaration just as the signal assignment statement does. The contents of the process statement can include sequential statements like those found in software programming languages. These statements are used to compute the outputs of the process from its inputs. Sequential statements are often more powerful, but sometimes have no direct correspondence to a hardware implementation.

The process statement can also contain signal assignments in order to specify the outputs of the process.

Our first example of the process statement is trivial and would not normally be done in a process statement. However, it allows us to examine the process statement without learning any sequential statements first.

```
compute_xor: process (b,c)
begin
  a<=b xor c;
end process;
```

The first part

```
compute_xor:
```

is used to name the process. Next is the keyword *process* that starts the definition of a process. Following that is a list of signals in parenthesis, called the *sensitivity list*. The signal sensitivity list is used to specify which signals should cause the process to be re-evaluated. Whenever any event occurs on one of the signals in the sensitivity list, the process is re-evaluated. A process is evaluated by performing each statement that it contains. These statements (the body of the process) appear between the *begin* and *end* keywords.

Using Variables

There are two major kinds of objects used to hold data. The first kind, used mostly in structural and data flow descriptions, is the signal. The second, which can only be used in processes is called a variable. A variable behaves like you would expect in a software programming language, which is much different than the behavior of a signal.

Although variables represent data like the signal, they do not have or cause events and are modified differently. Variables are modified with the variable assignment. For example,

```
a:=b;
```

assigns the value of *b* to *a*. The value is simply copied to *a* immediately. Since variables may only be used in processes, the assignment statement may only appear in a process. The assignment is performed when the process is executed, as explained in the last section.

The following example shows how a variable is used in a process.

```
count: process (x)
  variable cnt : integer := -1;
begin
  cnt:=cnt+1;
end process;
```

Variable declarations appear before the *begin* keyword of a process statement as in the example. The *variable declaration* is the same as the signal declaration except the key word *variable* is used instead of *signal*. The declaration in this example includes an optional part, which specifies the initial value of the variable, when a simulation begins.

The initialization part is included by adding the := and some constant expression after the type part of the declaration. This initialization part may also be included in signal declarations. The variable *cnt* is declared to be of the type integer. The integer type represents negative and positive integer values.

Sequential Statements

There are several statements that may only be used in the body of a process. These statements are called *sequential statements* because they are executed sequentially. That is, one after the other as they appear in the design from the top of the process body to the bottom. In this section we will examine some of these statements.

The first example illustrates the *if* statement and a common use of the VHDL *attribute*.

```
count: process (x)
  variable cnt : integer :=0 ;
begin
  if (x='1' and x'last_value='0') then
    cnt:=cnt+1;
  end if;
end process;
```

This if statement has two main parts, the condition and the statement body. A *condition* is any boolean expression (an expression that evaluates to TRUE and FALSE, such as expressions using relational operators). The condition in the example uses the attribute *last_value*, which is used to determine the last value that a signal had.

The execution of the if statement begins by evaluating the condition. If the condition evaluates to the value TRUE then the statements in the statement body will be executed. Otherwise, execution will continue after the *end if* and the statement body of the if statement is skipped. Thus, the assignment statement in this example is executed every time there is a rising edge on the signal *x*, counting the number of rising edges.

An example of another common form of the if statement is

```
...
if (inc='1') then
  cnt:=cnt+1;
else
  cnt:=cnt-1;
end if;
...
```

This form has two statement bodies. If the condition is TRUE, the first list of statements is executed (between the *then* and the *else*) and the second list of statements (between the *else* and the *end if*) is not. Otherwise, the second statement list is executed and the first is not. Thus, this example will increment *cnt* if *inc* is '1' and decrement it otherwise.

The last statement we will look at is the *loop statement*. We will explain just one form of the loop statement, often called a *for statement*. The for statement is used to execute a list

of statements several times. The following example uses a loop statement to compute the even parity of a bit vector.

```
signal x : bit_vector (7 downto 0);
...
process (x)
  variable p : bit;
begin
  p:='0'
  for i in 7 downto 0 loop
    p:=p xor x(i);
  end loop;
end process;
```

The signal *x* is an 8 bit signal representing a byte. The variable *p* is used to compute the parity of this byte. The first part of the for loop *i in 7 downto 0* is called the *parameter specification.* It specifies how many times the loop body will be executed and creates a temporary variable. It begins with the name of the temporary variable that will be created, in this case it is *i*. This is followed by the key word *in* and then a range of values as we have seen before. The body of the loop is executed once for every value in the range specified. The value of the temporary variable is assigned one of the values in the range each time the loop body is executed. In this example, the assignment will be executed first with i=7 then again with i=6, and again with i=5, and so on down to 0. This loop statement behaves the same as the following statements.

```
p:='0';
p:=p xor x(7);
p:=p xor x(6);
p:=p xor x(5);
p:=p xor x(4);
p:=p xor x(3);
p:=p xor x(2);
p:=p xor x(1);
p:=p xor x(0);
```

Signals and Processes

This section is short, but contains important information about the use of signals in the process statement. The issue of concern is to avoid confusion about the difference between how a signal assignment and variable assignment behave in the process statement. Remember a signal assignment, if anything, merely schedules an event to occur on a signal and does not have an immediate effect. When a process is resumed, it executes from top to bottom and no events are processed until after the process is complete. This means, if an event is scheduled on a signal during the execution of a process, that event can be processed after the process has completed at the earliest. Let's examine an example of this behavior. In the following process two events are scheduled on signals *x* and *z*.

```
...
signal x,y,z : bit;
...
process (y)
begin
```

```
  x<=y;
  z<=not x;
end process;
```
If the signal *y* changes then an event will be scheduled on *x* to make it the same as *y*.

This is pointed out because this is not necessarily the intuitive behavior and because variables operate differently. For example, in

```
process (y)
variable x,z : bit;
begin
  x:=y;
  z:=not x;
end process;
```
The value of the variable *z* would be the opposite of the value of *y* because the value of the variable *x* is changed immediately.

.

Program Output

In most programming languages there is a mechanism for printing text on the monitor and getting input from the user through the keyboard. Even though your simulator will let you monitor the value of signals and variables in your design, it is also nice to be able to output certain information during simulation. It is not provided as a language feature in VHDL, but rather as a standard library that comes with every VHDL language system. In VHDL, common code can be put in a separate file to be used by many designs. This common code is called a library. In order to use the library that provides input and output capabilities you must add the statement
```
use textio.all;
```
immediately before every architecture that uses input and output. The name of the library is textio and this statement indicates that you wish to use everything or all of the textio library. Once you have done that, you may use any of the features discussed in this section. Note that although it is not part of the language, the library is **standard** and will be the same regardless of the VHDL tools you are using.
```
use textio.all;
architecture behavior of check is
begin
  process (x)
    variable s : line;
    variable cnt : integer:=0;
  begin
    if (x='1' and x'last_value='0') then
      cnt:=cnt+1;
      if (cnt>MAX_COUNT) then
        write(s,"Counter overflow - ");
        write(s,cnt);
        writeline(output,s);
      end if;
    end if;
  end process;
```

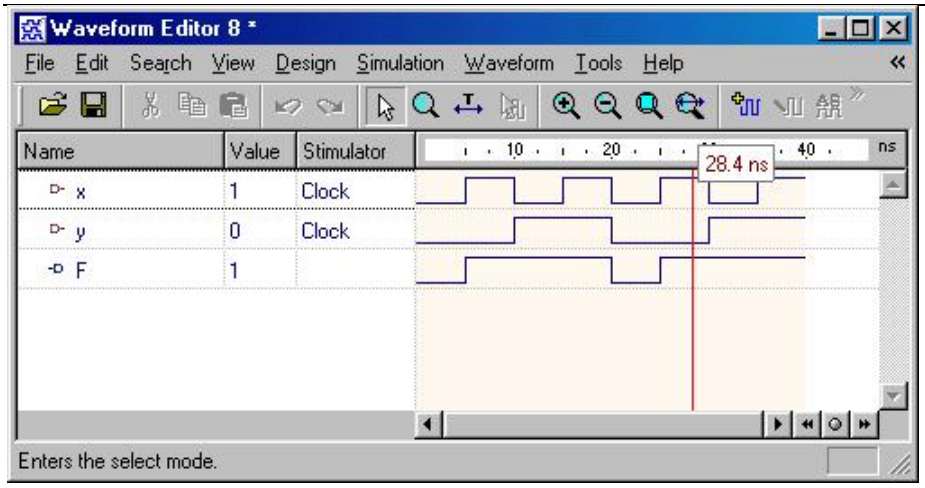
```
end behavior;
```
The *write* function is used to append text information at the end of a line variable which is empty when the simulator is initialized. The function takes two arguments, the first is the name of the line to append to, and the second is the information to be appended. In the example, *s* is set to "Counter overflow - ", and then the current value of *cnt* is converted to text and added to the end of that. The *writeline* function outputs the current value of a line to the monitor, and empties the line for re-use. The first argument of the writeline function just indicates that the text should be output to the screen. If MAX_COUNT were a constant equal to 15 and more than 15 rising edges occur on the signal *x*, then the message
```
Counter overflow - 16
```
would be printed on the screen.

The write statement can also be used to append constant values and the value of variables and signals of the types bit, bit_vector, time, integer, and real. Keyboard input is more complex than output, and is not discussed in this tutorial.

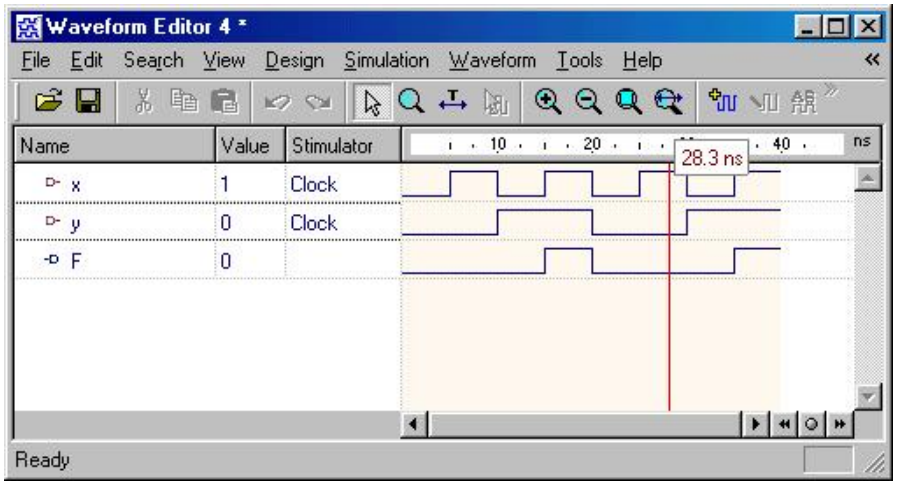
Design all the gates using VHDL.

```
----------------------------------------
                      --- OR gate
----------------------------------------

Library ieee;
use ieee.std_logic_1164.all;

--------------------------------------

entity OR_ent is
port(   x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end OR_ent;

--------------------------------------

architecture OR_arch of OR_ent is
begin

    process(x, y)
    begin
        -- compare to truth table
        if ((x='0') and (y='0')) then
            F <= '0';
        else
            F <= '1';
        end if;
    end process;

end OR_arch;

architecture OR_beh of OR_ent is
begin
```

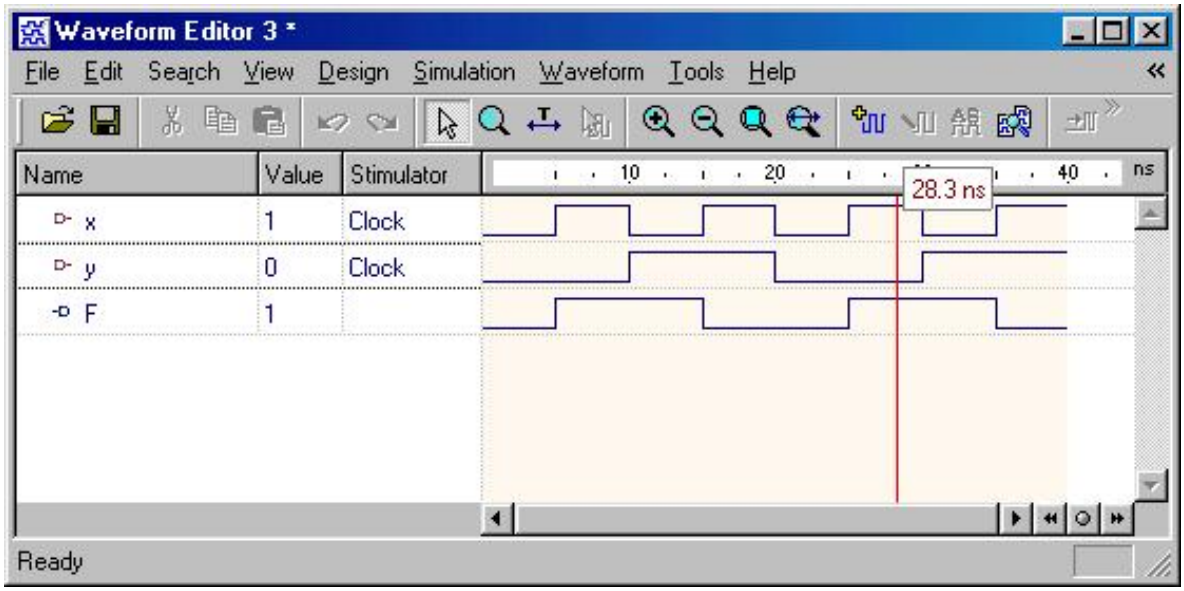
```
    F <= x or y;

end OR_beh;
```




```
-------------------------------------------------
                    -- AND gate
-------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-------------------------------------------------
entity AND_ent is
port(   x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end AND_ent;
-------------------------------------------------

architecture behav1 of AND_ent is
begin

    process(x, y)
    begin
        -- compare to truth table
        if ((x='1') and (y='1')) then
            F <= '1';
        else
            F <= '0';
        end if;
    end process;

end behav1;

architecture behav2 of AND_ent is
begin
    F <= x and y;
end behav2;
-------------------------------------------------
```

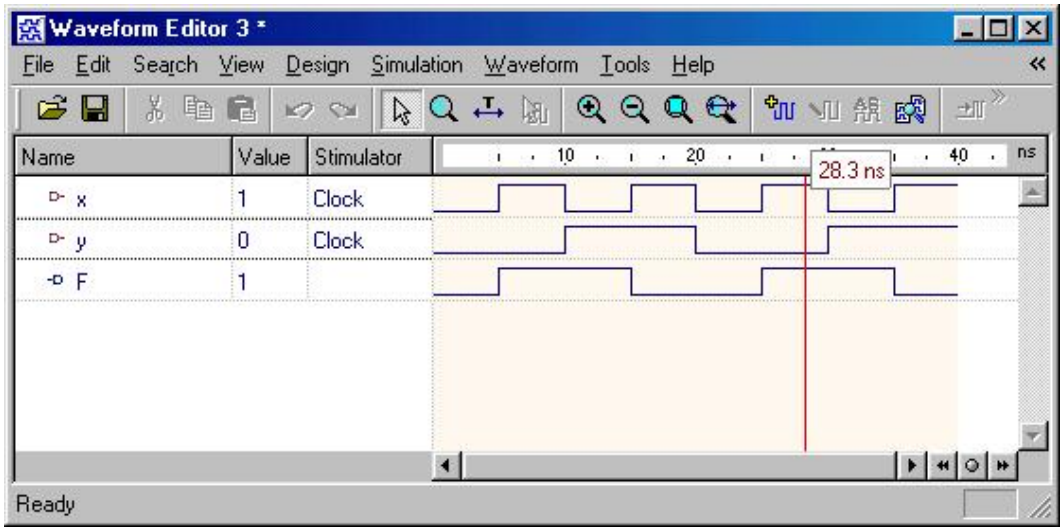



```
---------------------------------------
                        -- XOR gate
---------------------------------------
library ieee;
use ieee.std_logic_1164.all;
---------------------------------------
entity XOR_ent is
port(   x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end XOR_ent;

---------------------------------------

architecture behv1 of XOR_ent is
begin
    process(x, y)
    begin
        -- compare to truth table
        if (x/=y) then
            F <= '1';
        else
            F <= '0';
        end if;
    end process;

end behv1;

architecture behv2 of XOR_ent is
begin
    F <= x xor y;
end behv2;
```

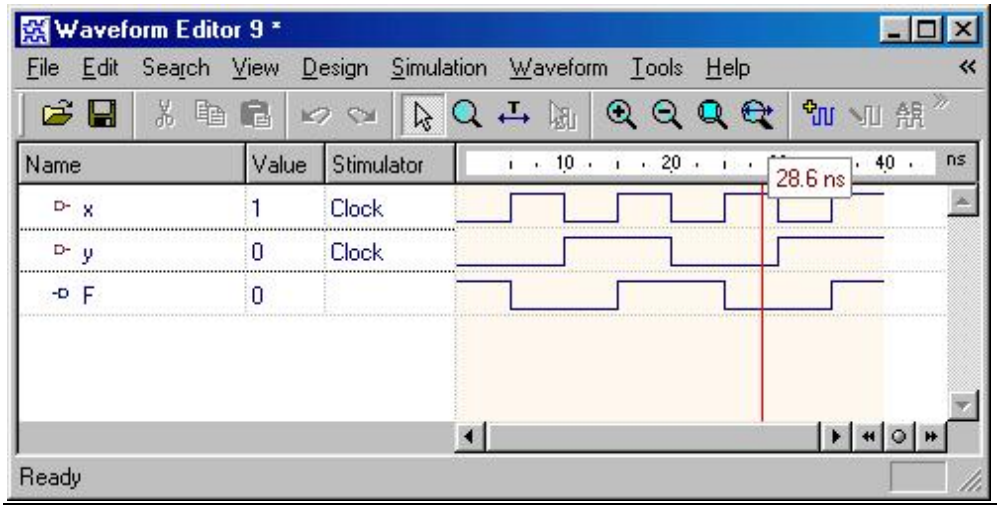



```
----------------------------------------
                                    -- NOR gate
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
----------------------------------------
entity NOR_ent is
port(   x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end NOR_ent;

----------------------------------------

architecture behv1 of NOR_ent is
begin

    process(x, y)
    begin
        -- compare to truth table
        if (x='0' and y='0') then
            F <= '1';
        else
            F <= '0';
        end if;
    end process;

end behv1;

architecture behv2 of NOR_ent is
begin

    F <= x nor y;
```

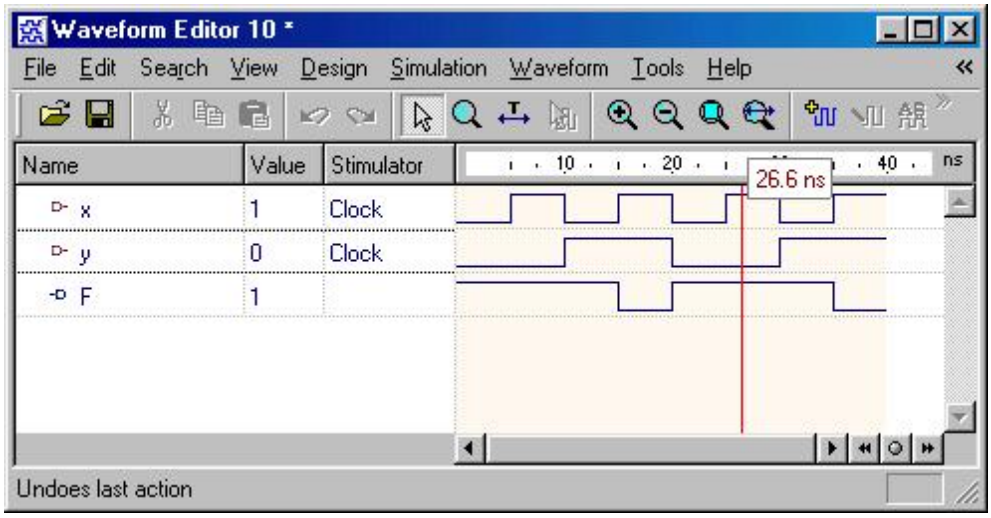
```
end behv2;
```




```
-------------------------------------
                        -- XOR gate
-------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-------------------------------------
entity XNOR_ent is
port(   x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end XNOR_ent;
-------------------------------------

architecture behv1 of XNOR_ent is
begin

    process(x, y)
    begin
        -- compare to truth table
        if (x/=y) then
            F <= '0';
        else
            F <= '1';
        end if;
    end process;

end behv1;

architecture behv2 of XNOR_ent is
begin

    F <= x xnor y;

end behv2;
```



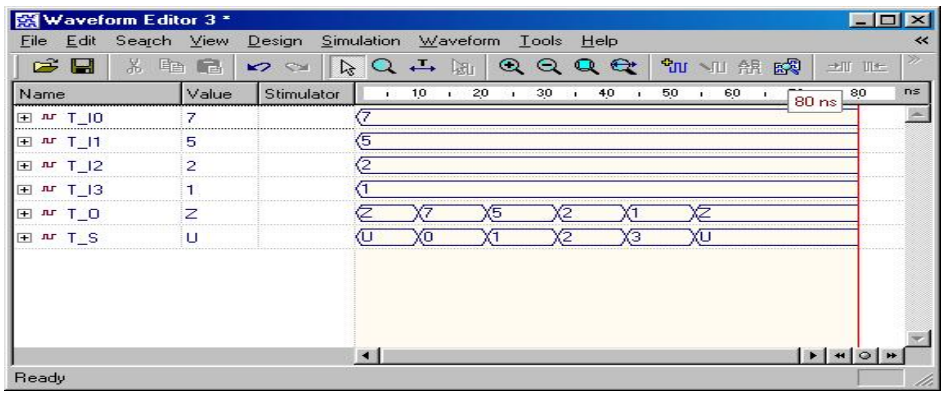

```
-----------------------------------------
-- NAND gate
-----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------
entity NAND_ent is
port(   x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end NAND_ent;

-----------------------------------------

architecture behv1 of NAND_ent is
begin

    process(x, y)
    begin
        -- compare to truth table
        if (x='1' and y='1') then
            F <= '0';
        else
            F <= '1';
        end if;
    end process;

end behv1;
-----------------------------------------
architecture behv2 of NAND_ent is
begin
        F <= x nand y;
end behv2;
```



Write VHDL programs for the following circuits, check the waveforms generated. (a) Multiplexer (b) Demultiplexer

```
-------------------------------------------------
-- VHDL code for 4:1 multiplexer
-------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-------------------------------------------------
entity Mux is
port(   I3:     in std_logic_vector(2 downto 0);
        I2:     in std_logic_vector(2 downto 0);
        I1:     in std_logic_vector(2 downto 0);
        I0:     in std_logic_vector(2 downto 0);
        S:      in std_logic_vector(1 downto 0);
        O:      out std_logic_vector(2 downto 0)
);
end Mux;
-------------------------------------------------
architecture behv1 of Mux is
begin
    process(I3,I2,I1,I0,S)
    begin
            -- use case statement
        case S is
            when "00" =>        O <= I0;
            when "01" =>        O <= I1;
            when "10" =>        O <= I2;
            when "11" =>        O <= I3;
            when others =>      O <= "ZZZ";
        end case;
    end process;
end behv1;
architecture behv2 of Mux is
begin
```

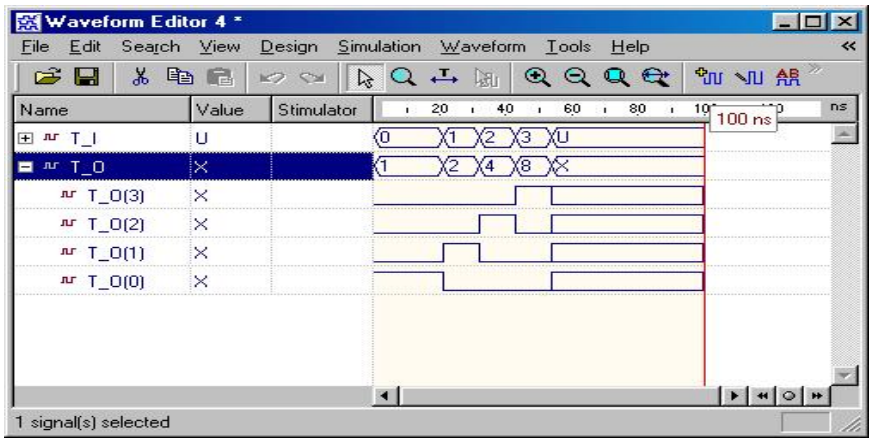
```
    -- use when.. else statement
    O <=        I0 when S="00" else
                I1 when S="01" else
                I2 when S="10" else
                I3 when S="11" else
                "ZZZ";
end behv2;
```



Write VHDL programs for the following circuits, check the waveforms generated. (a) Decoder (b) Encoder

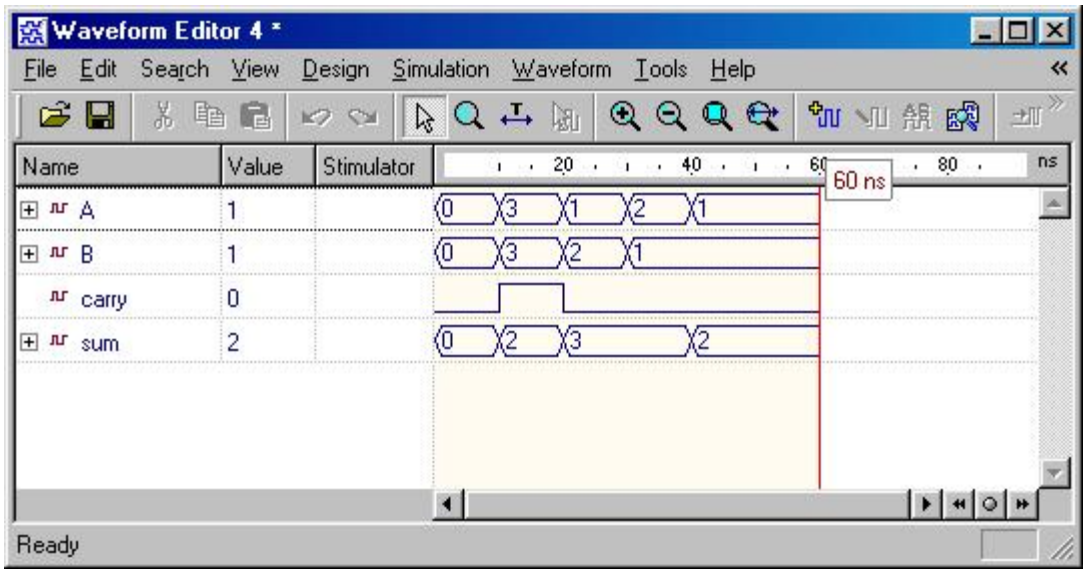
```
-------------------------------------------------
   2:4 Decoder
   -------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-------------------------------------------------
entity DECODER is
port(   I:      in std_logic_vector(1 downto 0);
        O:      out std_logic_vector(3 downto 0)
);
end DECODER;
-------------------------------------------------
architecture behv of DECODER is
begin
        process (I)
    begin
                    case I is
            when "00" => O <= "0001";
            when "01" => O <= "0010";
            when "10" => O <= "0100";
            when "11" => O <= "1000";
            when others => O <= "XXXX";
        end case;
    end process;
end behv;
architecture when_else of DECODER is
begin
    -- use when..else statement

    O <=        "0001" when I = "00" else
                "0010" when I = "01" else
                "0100" when I = "10" else
```

```
                "1000" when I = "11" else
                "XXXX";
end when_else;
-------------------------------------------------
```



Write VHDL programs for the following circuits, check the waveforms generated. (a) Half-adder (b) Full-adder

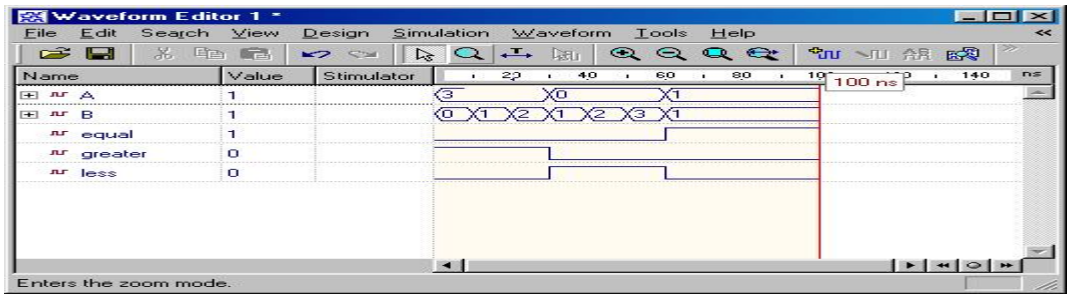
```
 ----------------------------------------------------
-- VHDL code for n-bit adder
-----------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-----------------------------------------------------------
entity ADDER is
generic(n: natural :=2);
port(   A:      in std_logic_vector(n-1 downto 0);
        B:      in std_logic_vector(n-1 downto 0);
        carry:  out std_logic;
        sum:    out std_logic_vector(n-1 downto 0)
);

end ADDER;
-----------------------------------------------------------
architecture behv of ADDER is
signal result: std_logic_vector(n downto 0);
begin

    result <= ('0' & A)+('0' & B);
    sum <= result(n-1 downto 0);
    carry <= result(n);
end behv;
-----------------------------------------------------------
```



Write VHDL programs for the comparator, check the waveforms generated.

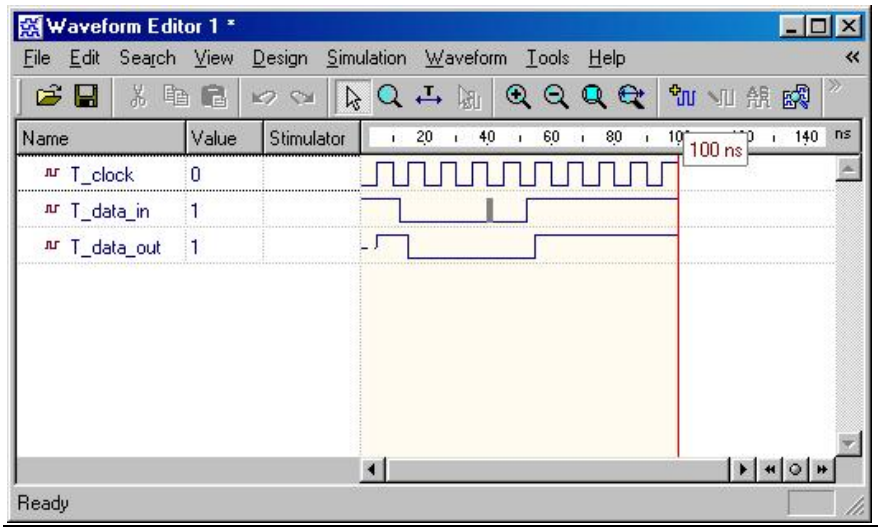
```
-----------------------------------------------------
-- n-bit Comparator
-----------------------------------------------------
 library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------------------
entity Comparator is
generic(n: natural :=2);
port(   A:      in std_logic_vector(n-1 downto 0);
        B:      in std_logic_vector(n-1 downto 0);
        less:           out std_logic;
        equal:          out std_logic;
        greater:        out std_logic
);
end Comparator;
-----------------------------------------------------
architecture behv of Comparator is

begin

    process(A,B)
    begin
        if (A<B) then
            less <= '1';
            equal <= '0';
            greater <= '0';
        elsif (A=B) then
            less <= '0';
            equal <= '1';
            greater <= '0';
        else
```

```
            less <= '0';
            equal <= '0';
            greater <= '1';
        end if;
    end process;

end behv;
-------------------------------------------------
```



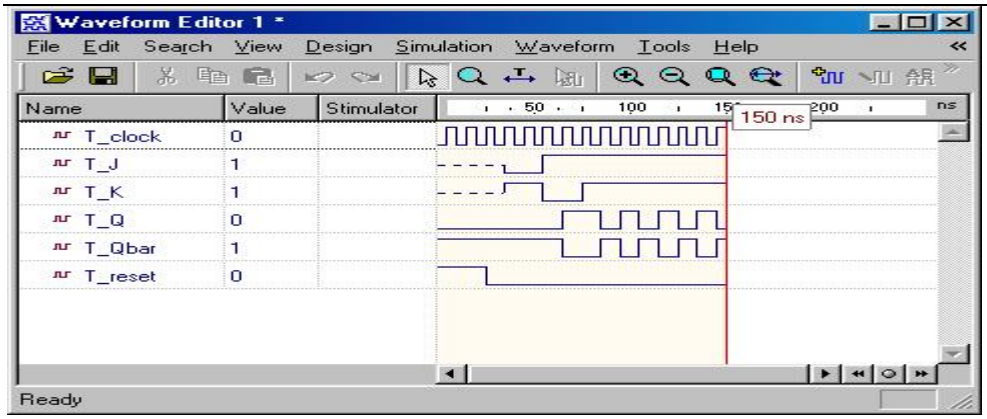
Write VHDL programs for the Flip-flop, check the waveforms generated.

```
-------------------------------------------
-- D Flip-Flop
-------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
use work.all;
-------------------------------------------
entity dff is
port(   data_in:        in std_logic;
        clock:          in std_logic;
        data_out:       out std_logic
);
end dff;
-------------------------------------------
architecture behv of dff is
begin

    process(data_in, clock)
    begin

        -- clock rising edge

        if (clock='1' and clock'event) then
            data_out <= data_in;
        end if;

    end process;

end behv;
```



```
JK Flip-Flop with reset
-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------------
entity JK_FF is
port (  clock:          in std_logic;
        J, K:           in std_logic;
        reset:          in std_logic;
        Q, Qbar:        out std_logic
);
end JK_FF;
-----------------------------------------------
architecture behv of JK_FF is
     signal state: std_logic;
    signal input: std_logic_vector(1 downto 0);
begin

    input <= J & K;
    p: process(clock, reset) is
    begin
                if (reset='1') then
            state <= '0';
        elsif (rising_edge(clock)) then

             -- compare to the truth table
            case (input) is
                when "11" =>
                    state <= not state;
                when "10" =>
```

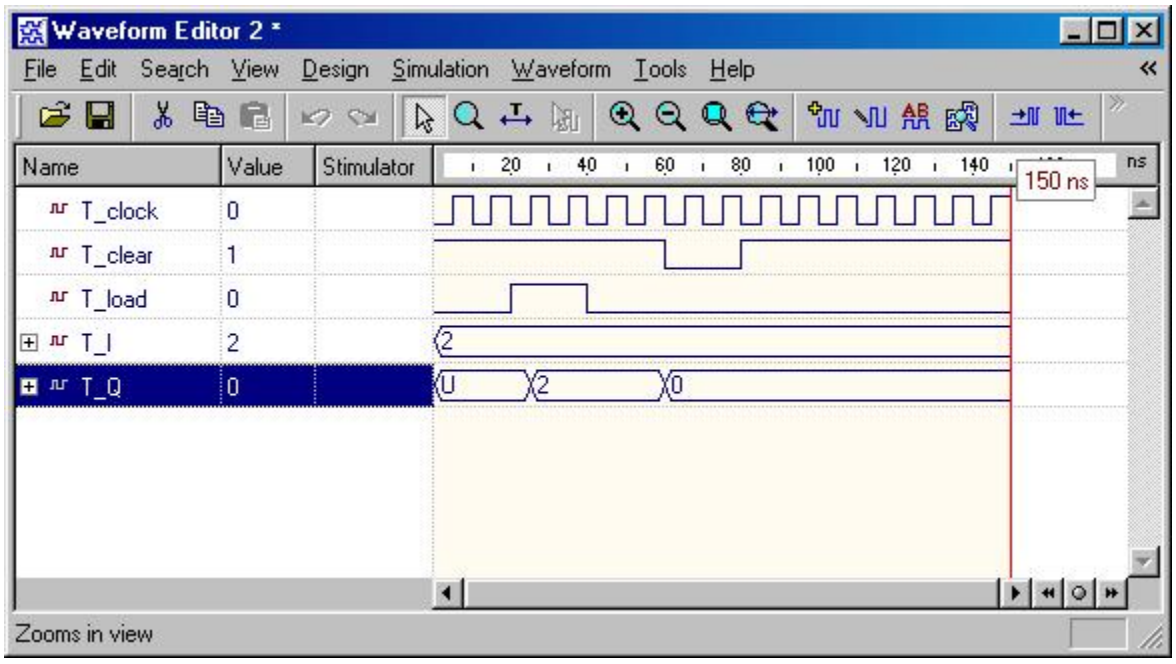
```
                state <= '1';
            when "01" =>
                state <= '0';
            when others =>
                null;
            end case;
        end if;
    end process;
        Q <= state;
    Qbar <= not state;
end behv;
```



Write VHDL programs for the following circuits, check the waveforms generated.
(a) Register
(b) Shift register

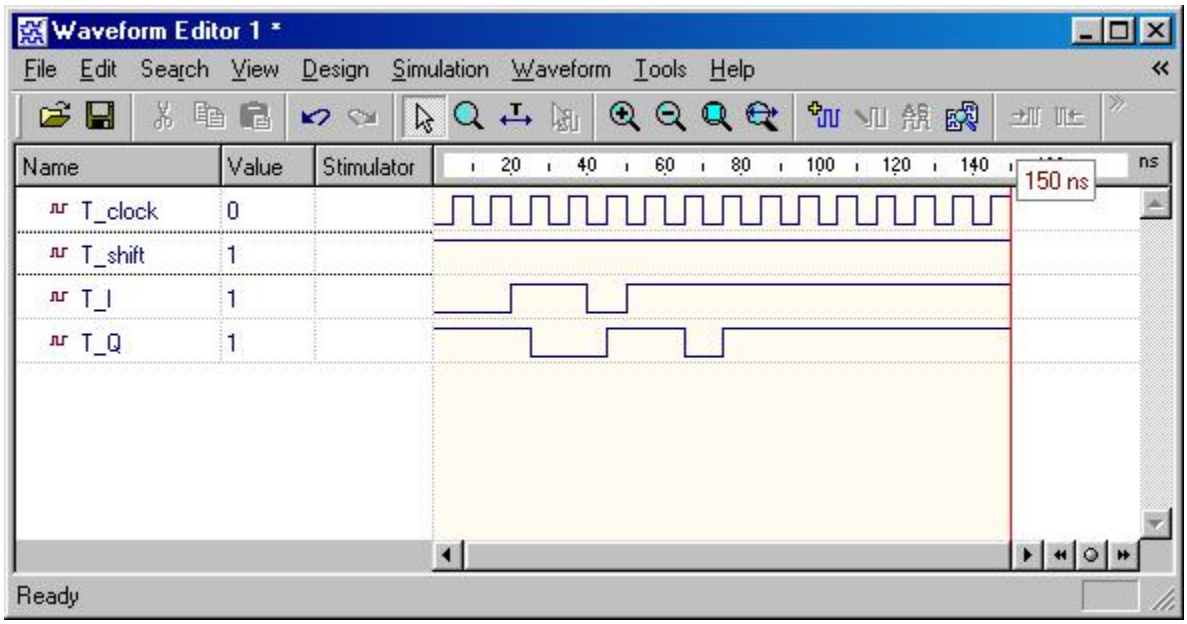
```
--------------------------------------------------
-- n-bit Register
--------------------------------------------------

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

--------------------------------------------------

entity reg is

generic(n: natural :=2);
port(   I:      in std_logic_vector(n-1 downto 0);
        clock:  in std_logic;
        load:   in std_logic;
        clear:  in std_logic;
        Q:      out std_logic_vector(n-1 downto 0)
);
end reg;

--------------------------------------------------

architecture behv of reg is
```

```
    signal Q_tmp: std_logic_vector(n-1 downto 0);

begin

    process(I, clock, load, clear)
    begin

        if clear = '0' then
            -- use 'range in signal assigment
            Q_tmp <= (Q_tmp'range => '0');
        elsif (clock='1' and clock'event) then
            if load = '1' then
                Q_tmp <= I;
            end if;
        end if;

    end process;

    -- concurrent statement
    Q <= Q_tmp;

end behv;

-------------------------------------------------
```



```
-------------------------------------------------
-- 3-bit Shift-Register/Shifter
-------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
-------------------------------------------------
entity shift_reg is
port(   I:              in std_logic;
        clock:          in std_logic;
        shift:          in std_logic;
        Q:              out std_logic
);
end shift_reg;
-------------------------------------------------
architecture behv of shift_reg is

    -- initialize the declared signal
    signal S: std_logic_vector(2 downto 0):="111";

begin

    process(I, clock, shift, S)
    begin

        -- everything happens upon the clock changing
        if clock'event and clock='1' then
            if shift = '1' then
                S <= I & S(2 downto 1);
            end if;
        end if;
    end process;
    -- concurrent assignment
```

```
    Q <= S(0);
end behv;
-------------------------------------------------
```



Write VHDL programs for the counter, check the waveforms generated.

```
-------------------------------------------------
-- VHDL code for n-bit counter
-------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
-------------------------------------------------
entity counter is
generic(n: natural :=2);
port(   clock:  in std_logic;
        clear:  in std_logic;
        count:  in std_logic;
        Q:      out std_logic_vector(n-1 downto 0)
);
end counter;
-------------------------------------------------
architecture behv of counter is
           signal Pre_Q: std_logic_vector(n-1 downto 0);

begin

        process(clock, count, clear)
    begin
        if clear = '1' then
            Pre_Q <= Pre_Q - Pre_Q;
        elsif (clock='1' and clock'event) then
            if count = '1' then
                Pre_Q <= Pre_Q + 1;
            end if;
        end if;
```

```
    end process;

       Q <= Pre_Q;
end behv;
-------------------------------------------------------
```