MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that Ms./Mr. …………………...…………………………………… Reg. No. …..…………………… Section: ……………… Roll No: ………………... has satisfactorily completed the lab exercises prescribed for Digital System Design Lab [CSE 2162] of Second Year B. Tech. Degree in Computer Science and Engg. at MIT, Manipal, in the academic year 2023-2024.

Date: ……...................................

Signature
Faculty in Charge

CONTENTS

Course Objectives

- To develop the skills of implementing logic circuits using Verilog.
- Simplify the logical expressions and implement using logic gates.
- Design and analyze the combinational and sequential circuits, simple systems.
- Relate theoretical concepts to practical applications like a multiplexer, encoder, decoder, code converter, counter, shift register applications.

Course Outcomes

At the end of this course, students will be able to

- Demonstrate the simulation of simple logical circuits in Verilog and design arithmetic circuits
- Develop the Verilog code for multiplexers, encoders, decoders and utilize them in the hierarchical code to build various practical applications.
- Design and simulate sequential circuits and simple processors using Verilog.

Evaluation plan

- Internal Assessment Marks: 60%
    - ✓ Continuous evaluation component (for each evaluation):10 marks
    - ✓ The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce.
    - ✓ Total marks of the continuous evaluations will be scaled to 40.
    - ✓ Midterm exam will be in the 6^{th} lab for 20 marks.
- End semester assessment of 2-hour duration: 40 marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should have a separate observation book and the required stationery for every lab session.
2. Be on time and follow the institution dress code.
3. Must sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance.
5. Adhere to the rules and maintain the decorum.

In- Lab Session Instructions

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record.
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The observation book should be complete with proper design, logical diagrams, truth tables and waveforms related to the experiment they perform.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalties in evaluation.
- The exercises for each week are divided into three sets:
    - Solved exercise.
    - Lab exercises - to be completed during lab hours.
    - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill.
- Questions for lab tests and examinations are not necessarily limited to the questions in the manual but may involve some variations and/or combinations of the questions.
- A sample note preparation is given as a model for observation.

THE STUDENTS SHOULD NOT

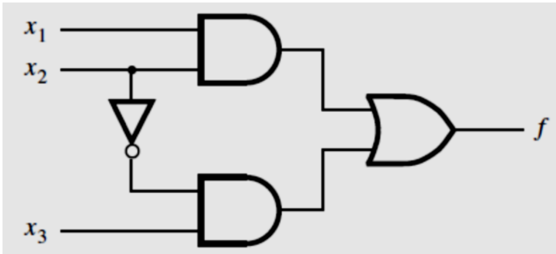
- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

SAMPLE LAB OBSERVATION NOTE PREPARATION

LAB NO: **Date:**

Title: Introduction to Verilog

1. Write Verilog code to implement the following circuit using the continuous assignment.



Aim: To write Verilog code, Testbench code, Truth table, and waveform for the above circuit.

Verilog code:

```
module example1(x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    assign f = (x1 & x2) | (~x2 & x3);
endmodule
```

TestBench Code:

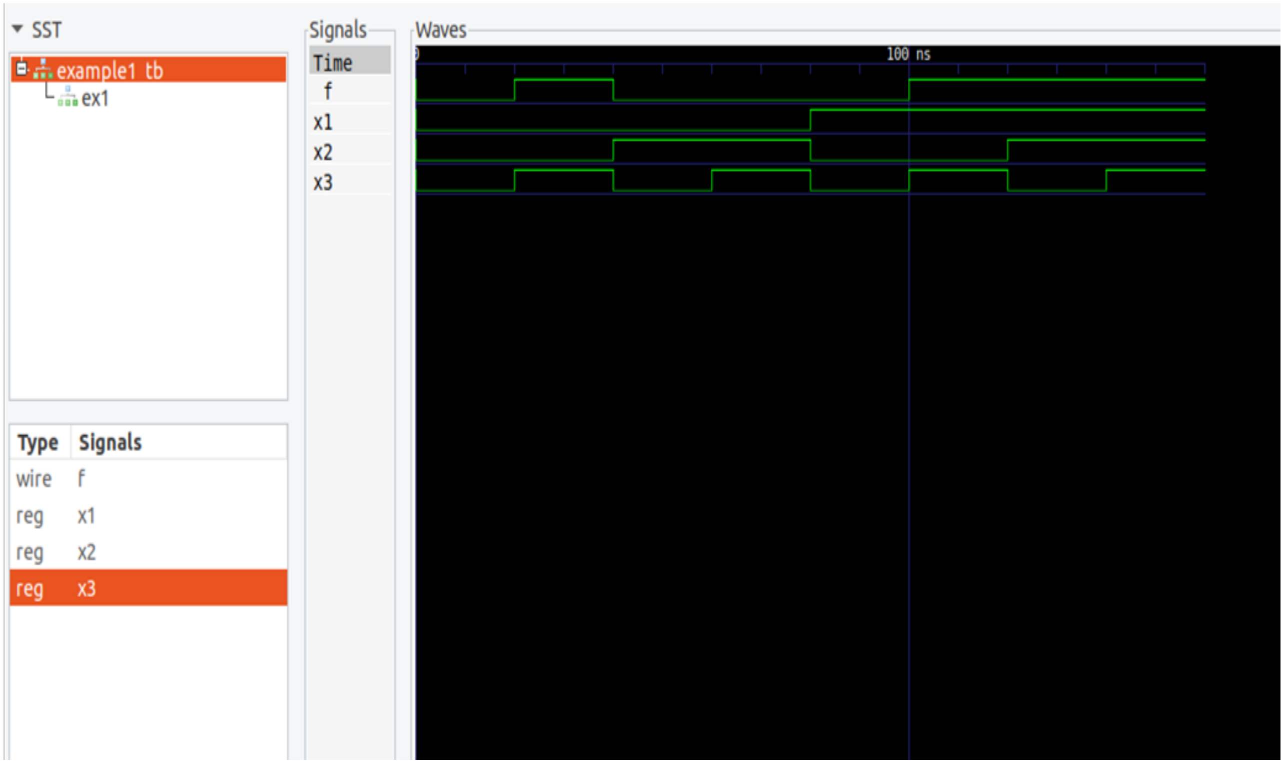
```
`timescale 1ns/1ns
`include "example1.v"    //Name of the Verilog file

module example1_tb();
reg x1, x2, x3;          //Input
wire f;                  //Output

example1 ex1(x1, x2, x3, f);   //Instantiation of the module
initial
begin

        $dumpfile("example1_tb.vcd");
        $dumpvars(0, example1_tb);

        x1=1'b0; x2=1'b0; x3=1'b0;
        #20;

        x1=1'b0; x2=1'b0; x3=1'b1;
        #20;
```

```
        x1=1'b0; x2=1'b1; x3=1'b0;
        #20;

        x1=1'b0; x2=1'b1; x3=1'b1;
        #20;

        x1=1'b1; x2=1'b0; x3=1'b0;
        #20;

        x1=1'b1; x2=1'b0; x3=1'b1;
        #20;

        x1=1'b1; x2=1'b1; x3=1'b0;
        #20;

        x1=1'b1; x2=1'b1; x3=1'b1;
        #20;

        $display("Test complete");
end

endmodule
```

Truth table:

x1	x2	x3	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Waveform:



SST
example1_tb
ex1
Type
Signals
wire
f
reg
x1
reg
x2
reg
x3
Signals
Time
f
x1
x2
x3
Waves
100 ns

Lab No 1: **Date:**

INTRODUCTION TO VERILOG AND K-MAP

Objectives:

In this lab, student will be able to

1. Learn the basic concepts of logic circuits and analyze the logic network with the aid of truth table and timing diagrams.
2. Learn the different tools available in the CAD system.
3. Write and simulate different representation of logic circuits using Verilog.
4. Design minimum cost circuit by applying the steps for optimization using K-map .
5. Write Verilog code for the simplified expression.

I. Basic concepts of Logic Circuits

Logic Circuits

- Perform operations on digital signals.
- Signal values are restricted to a few discrete values.
- In binary logic circuits, there are only two values, 0 and 1.

Logic Gates and Networks

- Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a logic gate.
- It has one or more inputs and one output that is a function of its inputs.
- It is often convenient to describe a logic circuit by drawing a circuit diagram, or schematic, consisting of graphical symbols representing the logic gates.

The graphical symbols for the AND, NOT and OR gates are shown in Fig. 1.1, 1.2 and 1.3 respectively.

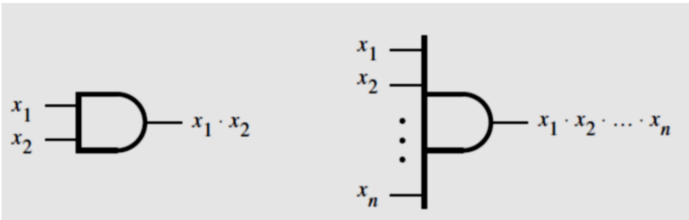


Figure 1.1

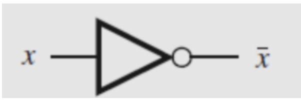


Figure 1.2

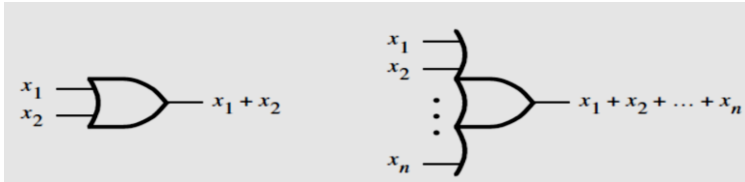


Figure 1.3

- A larger circuit is implemented by a network of gates, as shown in Fig. 1.4

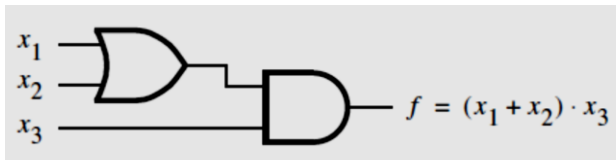


Figure 1.4

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

Figure 1.5 **Truth Table**

- The operations AND, OR etc can also be defined in the form of a table as shown in Figure 1.5.
- The first two columns (to the left of the heavy vertical line) give all four possible combinations of logic values that the variables x1 and x2 can have.
- The next column defines the AND operation for each combination of values of x1 and x2, and the last column defines the OR operation.
- In general, for n input variables the truth table has 2^n rows.

II. Analysis of a Logic Network

- Determining the function performed by an existing logic network is referred to as the Analysis process.
- The reverse task of designing a new network that implements a desired functional behavior is referred to as the Synthesis process.
- To determine the functional behavior of the network in Fig. 1.6, we can consider what happens if we apply all possible values to input signals x1 and x2. The analysis of these input values at various intermediate points is shown in Fig. 1.7.

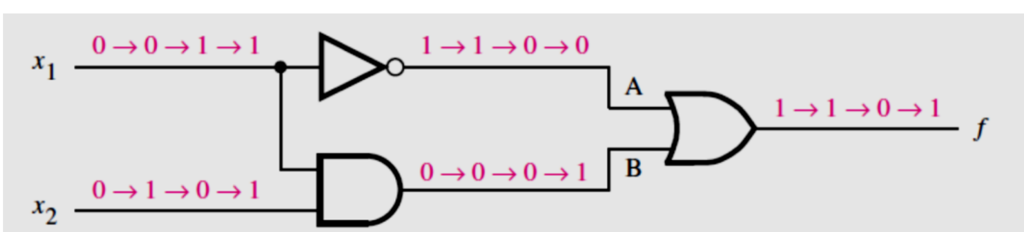


Figure 1.6

x1	x2	A	B	f
0	0	1	0	1
0	1	1	0	1
1	0	0	0	0
1	1	0	1	1

Figure 1.7

Timing Diagram

- The information in Fig. 1.7 can be presented in graphical form, known as a timing diagram, as shown in Fig. 1.8.
- The figure shows the waveforms for the inputs and output of the network, as well as for the internal signals at the points labeled A and B.

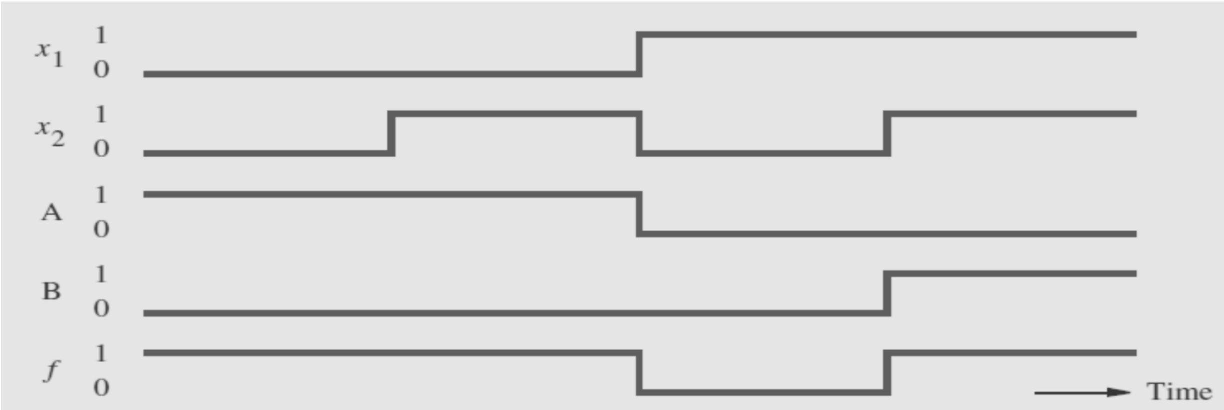


Figure 1.8

Functionally Equivalent Networks

- Going through the same analysis procedure, we find that the output '*g'* in Fig. 1.9, changes in exactly the same way as f does in Fig. 1.6.
- Therefore, g(x1, x2) = f (x1, x2), which indicates that the two networks are functionally equivalent.

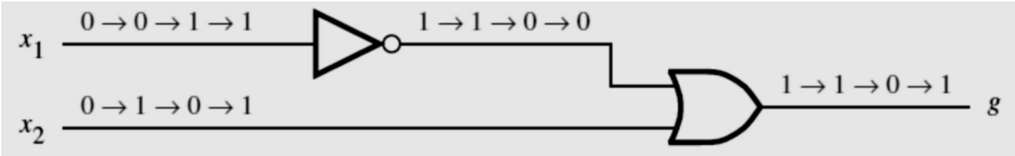


Figure 1.9

III. Introduction to CAD Tools

- Logic circuits are designed using CAD tools that automatically implement synthesis techniques.

- CAD system includes tools for design entry, synthesis and optimization, simulation and physical design.

Design Entry

- The starting point in the process of designing a logic circuit is the conception of what the circuit is supposed to do and the formulation of its general structure.
- The first stage of this process involves entering into the CAD system a description of the circuit being designed. This stage is called design entry.
- For design entry, we are writing source code in a hardware description language.

Hardware Description Languages

- A hardware description language (HDL) is similar to a typical computer programming language except that an HDL is used to describe hardware rather than a program to be executed on a computer.
- Two HDLs are IEEE standards: Verilog HDL and VHDL.

Why use Verilog

- Supported by most companies that offer digital hardware technology.
- Verilog provides design portability. A circuit specified in Verilog can be implemented in different types of chips and with CAD tools provided by different companies, without changing the Verilog specification.
- Both small and large logic circuit designs can be efficiently represented in Verilog code.

Functional Simulation

- The functional simulator tool verifies that the designed circuit functions as expected.
- It uses two types of information.
    - ➢ First, the user's initial design is represented by the logic equations generated during synthesis.
    - ➢ Second, the user specifies the valuations of the circuit's inputs that should be applied to these equations during the simulation.
- For each valuation, the simulator evaluates the outputs produced by the expressions.
- The results of simulations are usually provided in the form of a timing diagram that the user can examine to verify that the circuit operates as required.

Timing Simulation

- When the values of inputs to the circuit change it takes a certain amount of time before a corresponding change occurs at the output. This is called a propagation delay of the circuit.

IV. Representation of Digital Circuits in Verilog

- **Structural representation-** A larger circuit is defined by writing code that connects simple circuit elements together.
- **Behavioral representation-** Describing a circuit by using logical expressions and programming constructs that define the behavior of the circuit but not its actual structure in terms of gates.

Structural Specification of Logic Circuits

- A gate is represented by indicating its functional name, output, and inputs. Different logic gates are shown in Table 1.1

For example,

- A two-input AND gate, with inputs x1 and x2 and output y, is denoted as

  and (y, x1, x2);
- A four-input OR gate is specified as

  or (y, x1, x2, x3, x4);
- The NOT gate is given by **not** (y, x); implements y = x'.

Table 1.1

Name	Description	Usage
and	$f = (a \cdot b \cdot \cdots)$	**and** $(f, a, b, \ldots)$
nand	$f = \overline{(a \cdot b \cdot \cdots)}$	**nand** $(f, a, b, \ldots)$
or	$f = (a + b + \cdots)$	**or** $(f, a, b, \ldots)$
nor	$f = \overline{(a + b + \cdots)}$	**nor** $(f, a, b, \ldots)$
xor	$f = (a \oplus b \oplus \cdots)$	**xor** $(f, a, b, \ldots)$
xnor	$f = (a \odot b \odot \cdots)$	**xnor** $(f, a, b, \ldots)$
not	$f = \overline{a}$	**not** (f, a)

Verilog Module

- It is a circuit or subcircuit described with Verilog code.
- The module has a name, ***module_name***, which can be any valid identifier, followed by a list of ports.
- The term port refers to an input or output connection in an electrical circuit. The ports can be of type **input**, **output**, or **inout** (bidirectional), and can be either scalar or vector.

The General Form of a Module

```
module module name [(port name{, port name})];
        [parameter declarations]
        [input declarations]
        [output declarations]
        [inout declarations]
        [wire or tri declarations]
        [reg or integer declarations]
        [function or task declarations]
        [assign continuous assignments]
        [initial block]
        [always blocks]
        [gate instantiations]
        [module instantiations]
endmodule
```

Documentation in Verilog Code

- Documentation can be included in Verilog code by writing a comment. A short comment begins with the double slash, **//**, and continues to the end of the line. A long comment can span multiple lines and is contained inside the delimiters **/*** and ***/.**

White Space

- White space characters, such as SPACE and TAB, and blank lines are ignored by the Verilog compiler.
- Multiple statements can be written on a single line.
- Placing each statement on a separate line and using indentation within blocks of code, such as an **if-else** statement are good ways to increase the readability of code.

Signals in Verilog Code

- A signal in a circuit is represented as a net or a variable with a specific type.
- A net or variable declaration has the form

  type [range] signal_name{, signal_name};
- The signal_name is an identifier
- The range is used to specify vectors that correspond to multi-bit signals

Signal Values and Numbers

- Verilog supports scalar nets and variables that represent individual signals and vectors that correspond to multiple signals.
- Each individual signal can have four possible values:

  0 = logic value 0 1 = logic value 1

  z = tri-state (high impedance) x = unknown value

- The value of a vector variable is specified by giving a constant of the form [size][’radix]constant where size is the number of bits in the constant, and radix is the number base. Supported radices are

  d = decimal   b = binary   h = hexadecimal   o = octal
- Some examples of constants include

  0 the number 0

  10 the decimal number 10

  ’b10 the binary number 10 = $(2)_{10}$

  ’h10 the hex number 10 = $(16)_{10}$

  4’b100 the binary number 0100 = $(4)_{10}$

Nets

Verilog defines a number of types of nets.

- A net represents a node in a circuit.
- For synthesis purposes, the only important nets are of **wire** type.
- For specifying signals that are neither inputs nor outputs of a module, which are used only for internal connections within the module, Verilog provides the **wire** type.

Identifier Names

- Identifiers are the names of variables and other elements in Verilog code.
- The rules for specifying identifiers are simple: any letter or digit may be used, as well as the _ underscore and $ characters.
- An identifier must not begin with a digit and it should not be a Verilog keyword.
  - Examples of legal identifiers are f, x1, x, y, and Byte.
  - Some examples of illegal names are 1x, +y, x*y, and 258
- Verilog is case sensitive, hence k is not the same as K, and BYTE is not the same as Byte.

Verilog Operators

- Verilog operators are useful for synthesizing logic circuits.
- Table 1.2 lists these operators in groups that reflect the type of operation performed.

Table 1.2

Operator type	**Operator Symbols**	**Operation Performed**	**Number of operands**
Bitwise	~	1’s complement	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
Logical	!	NOT	1

	&& ||	AND OR	2 2
Reduction	& ~& | ~| ^ ~^ or ^~	Reduction AND Reduction NAND Reduction OR Reduction NOR Reduction XOR Reduction XNOR	1 1 1 1 1 1
Arithmetic	+ - - * /	Addition Subtraction 2’s complement Multiplication Division	2 2 1 2 2
Relational	> < >= <=	Greater than Lesser than Greater than or equal to Lesser than or equal to	2 2 2 2
Equality	== !=	Logical equality Logical inequality	2 2
Shift	>> <<	Right shift Left shift	2 2
Concatenation	{,}	Concatenation	Any number
Replication	{{}}	Replication	Any number
Conditional	?:	Conditional	3

Running a sample Verilog code

Let us look at a Verilog implementation in the following steps:

1. Create a directory with section followed by roll number (to be unique); e.g. A21
2. Open any text editor in Ubuntu (say, **gedit**) and type the source code.

```
module example2(x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    and (g, x1, x2);
    not (k, x2);
    and (h, k, x3);
    or (f, g, h);
endmodule
```

3. Save the source code with file name same as module name but with **'.v'** extension in the required directory.
4. Type the following testbench code.

```
`timescale 1ns/1ns
`include "example2.v"    //Name of the Verilog file

module example2_tb();
reg x1, x2, x3;            //Input
wire f;                    //Output
example2 ex2(x1, x2, x3, f);   //Instantiation of the module
initial
begin
        $dumpfile("example2_tb.vcd");
        $dumpvars(0, example2_tb);

        x1=1'b0; x2=1'b0; x3=1'b0;
        #20;

        x1=1'b0; x2=1'b0; x3=1'b1;
        #20;

        x1=1'b0; x2=1'b1; x3=1'b0;
        #20;

        x1=1'b0; x2=1'b1; x3=1'b1;
        #20;

        x1=1'b1; x2=1'b0; x3=1'b0;
        #20;

        x1=1'b1; x2=1'b0; x3=1'b1;
        #20;

        x1=1'b1; x2=1'b1; x3=1'b0;
        #20;

        x1=1'b1; x2=1'b1; x3=1'b1;
        #20;

        $display("Test complete");
end
endmodule
```

5. Save the source code with file name same as module name but with **'.v'** extension in the same directory.
6. Go to the terminal. Type the following commands for compilation.

- **iverilog -o example2_tb.vvp example2_tb.v**

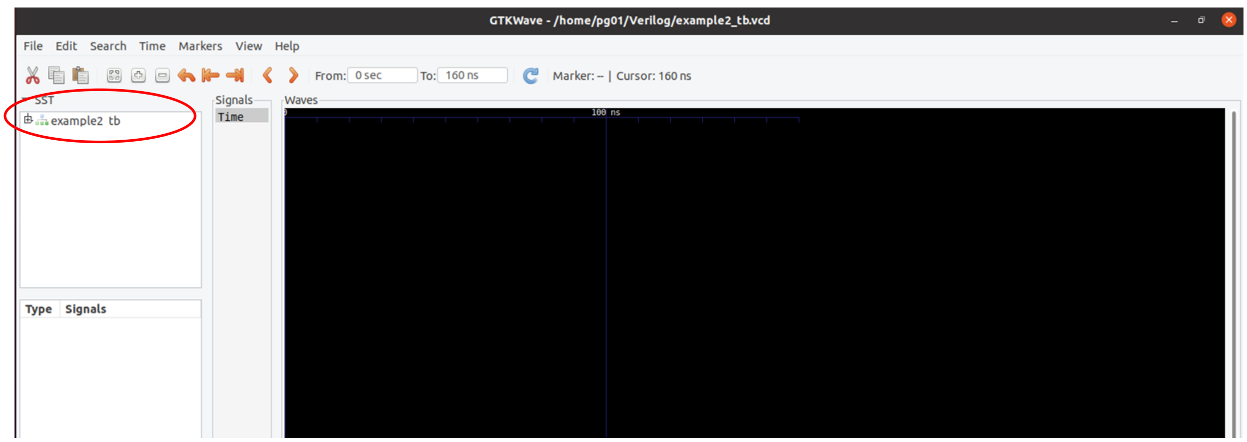
On successful execution, file example2_tb.vvp is created.

- **vvp example2_tb.vvp**

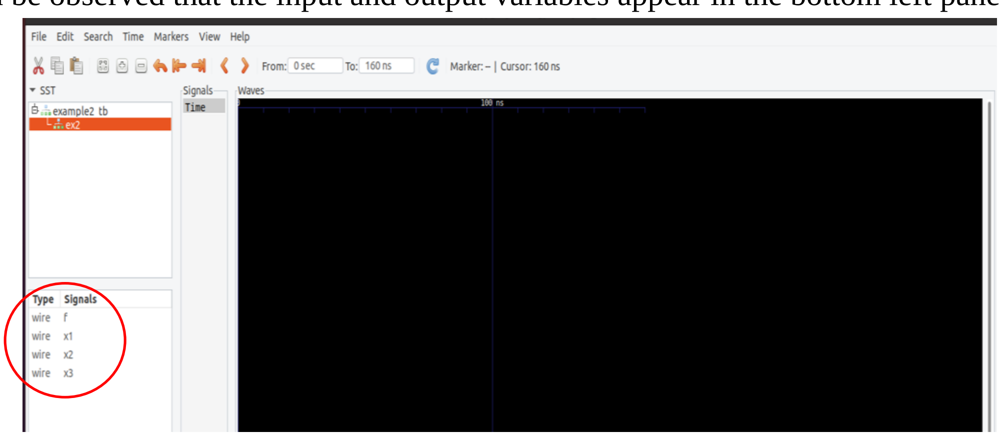
On successful execution, file example2_tb.vcd is created and the following messages are displayed in the terminal.

VCD info: dumpfile example2_tb.vcd opened for output.
Test complete

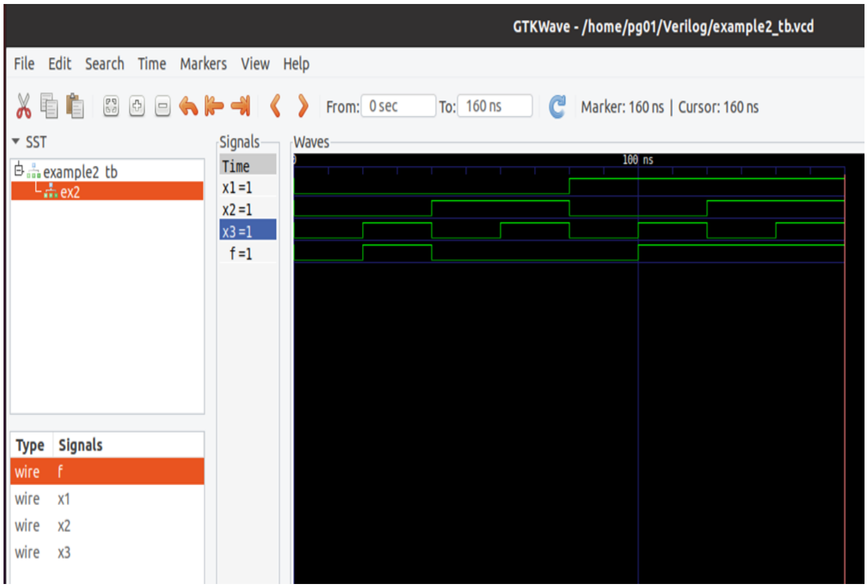
7. Open **gtkwave** by double clicking the example2_tb**.vcd** file available in your directory or type **gtkwave** in the terminal and open the example2_tb**.vcd** file in gtkwave. A window appears as shown below.



8. Expand example2_tb that appears in the left pane. The screen appears as shown below. It can be observed that the input and output variables appear in the bottom left pane.



9. Drag and drop the variables that appears in the *Signals* tab of the bottom left pane into **Signals** column that appears in the middle of the window. The output appears as shown below. Cross-verify the output with the truth table.



10. Generated outputs are with respect to the values for input variables in the test bench.

V. Behavioral Specification of Logic Circuits

- Gate level primitives can be tedious when large circuits have to be designed.
- Abstract expressions and programming constructs are used to describe the behavior of a digital circuit.
- To define the circuit using logic expressions. The AND and OR operations are indicated by the '&' and '|' signs, respectively.
- The **assign** keyword provides a continuous assignment for the output signal.
- Whenever any signal on the right-hand side changes its state, the value of output will be re-evaluated.

```
module example2 (x1, x2, x3, f);
        input x1, x2, x3;
        output f;
        assign f = (x1 & x2) | (x2 & x3);
endmodule
```

VI. Steps for Optimization using K-map:

K-map

- ➢ A systematic way of performing optimization.
- ➢ Finds a minimum-cost expression for a given logic function by reducing the number of product (or sum) terms needed in the expression, by applying the combining property.

- **Implicant-** A product term that indicates the input valuation(s) for which a given function is equal to 1.
- **Prime Implicant-** An implicant is called a prime implicant if it cannot be combined into another implicant that has fewer literals.
- **Essential prime implicant-**If a prime implicant includes a minterm for which $f = 1$ that is not included in any other prime implicant, then it must be included in the cover and is called as an essential prime implicant.
- The process of finding a minimum-cost circuit involves the following steps:
  1. Generate all prime implicants for the given function *f*.
  2. Find the set of essential prime implicants.
  3. If the set of essential prime implicants covers all valuations for which $f = 1$, then this set is the desired cover of *f*. Otherwise, determine the nonessential prime implicants that should be added to form a complete minimum-cost cover.

VII. Incompletely Specified Functions

- A function that has don't-care condition(s).
- Using the shorthand notation, the function f is specified as

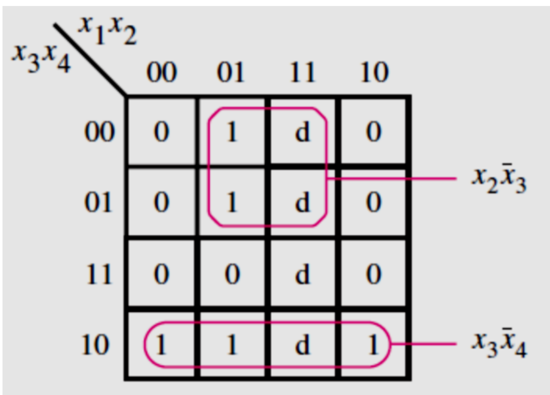
  f (x1,…….., x4) =∑m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)

  where D is the set of don't cares.

Example:

Simplify the following function using K-map and write Verilog code to implement this.

f (x1,…….., x4) =∑m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)



$f = x2\overline{x3} + x3\overline{x4}$

Lab Exercises

1. Simplify the following functions using K-map and implement the circuit using logic gates.
   a) $f(A,B,C,D) = \sum m(2,3,6,7,9,10,11,13,14,15)$
   b) $f(A,B,C,D) = \sum m(2,4,5,6,10) + D(12,13,14,15)$
2. Simplify the following functions using K-map and implement the circuit using logic gates.
   a) $f(A,B,C,D) = \prod M(0,1,4,8,9,12,15)$
   b) $f(A,B,C,D) = \prod M(2,5,6,8,10) + D(9,12,14)$

Additional Exercises

1. Write Verilog code to describe the following functions
   $f1 = ac' + bc + b'c'$
   $f2 = (a + b' + c)(a + b + c')(a' + b + c')$
   Check whether f1 and f2 in question 1 are functionally equivalent or not.
2. Simulate a circuit that has four inputs, x1, x2, x3, and x4, which produces an output value of 1 whenever three or more of the input variables have the value 1; otherwise, the output has to be 0.

Lab No 2: **Date:**

MULTILEVEL SYNTHESIS AND ARITHMETIC CIRCUITS

Objectives:

In this lab, student will be able to

1. Design multilevel NAND and NOR circuits.
2. Write Verilog code for multilevel circuits.
3. Use functional decomposition for the synthesis of multilevel circuits.

- **Multilevel NAND and NOR Circuits**
  - Multilevel AND-OR circuits can be realized by a circuit that contains only NAND gates or only NOR gates.
  - Each AND gate is converted to a NAND by inverting its output.
  - Each OR gate is converted to a NAND by inverting its inputs.
  - Each AND gate is converted to a NOR by inverting its inputs.
  - Each OR gate is converted to a NOR by inverting its output.
  - Inversions that are not a part of any gate can be implemented as two-input NAND/NOR gates, where the inputs are tied together.
- **Functional decomposition-** Complex logic circuit can be reduced by decomposing a two-level circuit into sub circuits, where one or more sub circuits implement functions that may be used in several places to construct the final circuit.

Solved Exercise

Apply functional decomposition for the following function to obtain a simplified circuit and simulate using Verilog.

$$f = \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 x_4 + \overline{x}_1 \overline{x}_2 x_4$$

Factoring *x*3 from the first two terms and *x*4 from the last two terms, this expression becomes

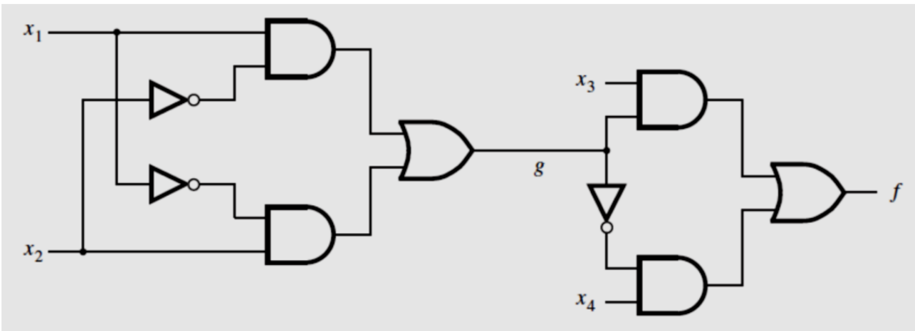
$$f = (\bar{x}_1 x_2 + x_1 \bar{x}_2)x_3 + (x_1 x_2 + \bar{x}_1 \bar{x}_2)x_4$$

Now let $g(x_1, x_2) = \bar{x}_1 x_2 + x_1 \bar{x}_2$ and observe that

$$\begin{aligned}
\bar{g} &= \overline{\bar{x}_1 x_2 + x_1 \bar{x}_2} \\
&= \overline{\bar{x}_1 x_2} \cdot \overline{x_1 \bar{x}_2} \\
&= (x_1 + \bar{x}_2)(\bar{x}_1 + x_2) \\
&= x_1 \bar{x}_1 + x_1 x_2 + \bar{x}_2 \bar{x}_1 + \bar{x}_2 x_2 \\
&= 0 + x_1 x_2 + \bar{x}_1 \bar{x}_2 + 0 \\
&= x_1 x_2 + \bar{x}_1 \bar{x}_2
\end{aligned}$$

Then f can be written as

$$f = g x_3 + \bar{g} x_4$$



Verilog code:

```
module example5(x1, x2, x3, x4, f);
        input x1, x2, x3, x4;
        output f;
        assign g = (x1 & ~x2) | (~x1 & x2);
        assign f = (g & x3) | (~g & x4);
endmodule
```

Lab Exercises

1. Find the minimum cost circuit for the function $F = A\bar{C}D + AC\bar{D} + B\bar{C}\bar{D} + BCD$ using functional decomposition. Write the Verilog code to simulate the same.
2. Write behavioral Verilog code to implement full adder.
3. Write behavioral Verilog code to implement 4-bit adder/subtractor.
4. Design 3*3 multiplier using adders and basic gates. Write the Verilog code to verify the design

Additional Exercises

1. Design and write Verilog code for a 2 digit BCD adder.
2. Write Verilog code to implement full adder. Perform AND, NAND, NOR, OR, XOR, XNOR operations using a full adder circuit.

Lab No 3: **Date:**

MULTIPLEXERS, COMPARATORS AND CODE CONVERTERS

Objectives:

In this lab, student will be able to

1. Learn more about the behavioral style of Verilog programming.
2. Understand the concept of multiplexers.
3. Design and implement simple multiplexers, large multiplexers using small multiplexers.
4. Learn the concept and write Verilog code to simulate comparators and code converters.

I. Multiplexers

- The multiplexer has a number of data inputs, one or more select inputs, and one output.
- It passes the signal value on one of the data inputs to the output.
- A multiplexer that has N data inputs, $w_0, \ldots, w_{N-1}$, requires $\log_2 N$ select inputs.
- Fig. 3.1a shows the graphical symbol for a 2-to-1 multiplexer.
- The functionality of a multiplexer can be described in the form of a truth table. Fig. 3.1b shows the functionality of a 2-to-1 multiplexer.

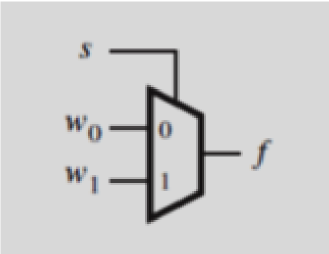


Figure 3.1a Graphical symbol

s	f
0	w_0
1	w_1

Figure 3.1b Truth table

The Conditional Operator

- In a logic circuit, it is often necessary to choose between several possible signals or values based on the state of some condition.
- Verilog provides a conditional operator (**?:**) which assigns one of the two values depending on a conditional expression.

  Syntax of conditional operator

  ```
  conditional_expression ?  true_expression : false_expression
  ```

- If the conditional expression evaluates to 1 (true), then the value of true_expression is chosen; otherwise, the value of false_expression is chosen.

The case Statement

- The general form of a **case** statement is given below.

```
case (expression)
        alternative1: begin
                statement;
                end
        alternative2: begin
                statement;
                end
        [default: begin
                statement;
              end]
endcase
```

- The bits in expression, called as controlling expression, are checked for a match with each alternative.
- The first successful match causes the associated statements to be evaluated.
- Each digit in each alternative is compared for an exact match of the four values 0, 1, x, and z.
- A special case is the **default** clause, which takes effect if no other alternative matches.
- The **casex** statement reads all z and x values as don't cares.

II. Arithmetic Comparison Circuits

- Compare the relative magnitude of two binary numbers.

III. Code Converters

- Convert from one type of input representation to a different output representation.

Parameters

- A parameter associates an identifier name with a constant.
- Using the following declaration, the identifier n can be used in place of the number 4.

  parameter n = 4;

Verilog Procedural Statements

- Rather than using gates or logic expressions, circuits can be specified in terms of their behavior.
- Also called sequential statements.
- Procedural statements are evaluated in the order in which they appear in the code whereas concurrent statements are executed in parallel.
- Verilog syntax requires that procedural statements should be inside an **always** block.

Always Block

- An **always** block is a construct that contains one or more procedural statements.
- It has the form

```
always @(sensitivity_list)
[begin]
    [procedural assignment statements]
    [if-else statements]
    [case statements]
    [while, repeat, and for loops]
    [task and function calls]
[end]
```

- The ***sensitivity_list*** is a list of signals that directly affect the output results generated by the **always** block.
- If the value of a signal in the sensitivity list changes, then the statements inside the **always** block are evaluated in the order presented.

Variables:

- A variable can be assigned a value and this value is retained until it is overwritten in a subsequent assignment statement.
- There are two types of variables, **reg,** and an **integer**.
    - The keyword **reg** does not denote a storage element or register. In Verilog code, **reg** variables can be used to model either combinational or sequential parts of a circuit.
    - **Integer** variables are useful for describing the behavior of a module, but they do not directly correspond to nodes in a circuit.

The if-else Statement

- The general form of the **if-else** statement is given below.

```
if (expression1)
begin
    statement;
end
else if (expression2)
begin
    statement;
end
else
begin
    statement;
```

```
    end
```

- If expression1 is true, then the first statement is evaluated.
- When multiple statements are involved, they have to be included inside a **begin-end** block.
- The **else if** and **else** clauses are optional.
- Verilog syntax specifies that when **else if** or **else** are included, they are paired with the most recent unfinished **if** or **else if**.

for Loop

- the general form of **for** loop

```
for (initial_index; terminal_index; increment)
  begin
      statement;
  end
```

- The ***initial_index*** is evaluated once, before the first loop iteration, and typically performs the initialization of the **integer** loop control variable.
- In each loop iteration, the begin-end block is performed, and then the increment statement is evaluated.
- Finally, the ***terminal_index*** condition is checked, and if it is True (1), then another loop iteration is done.

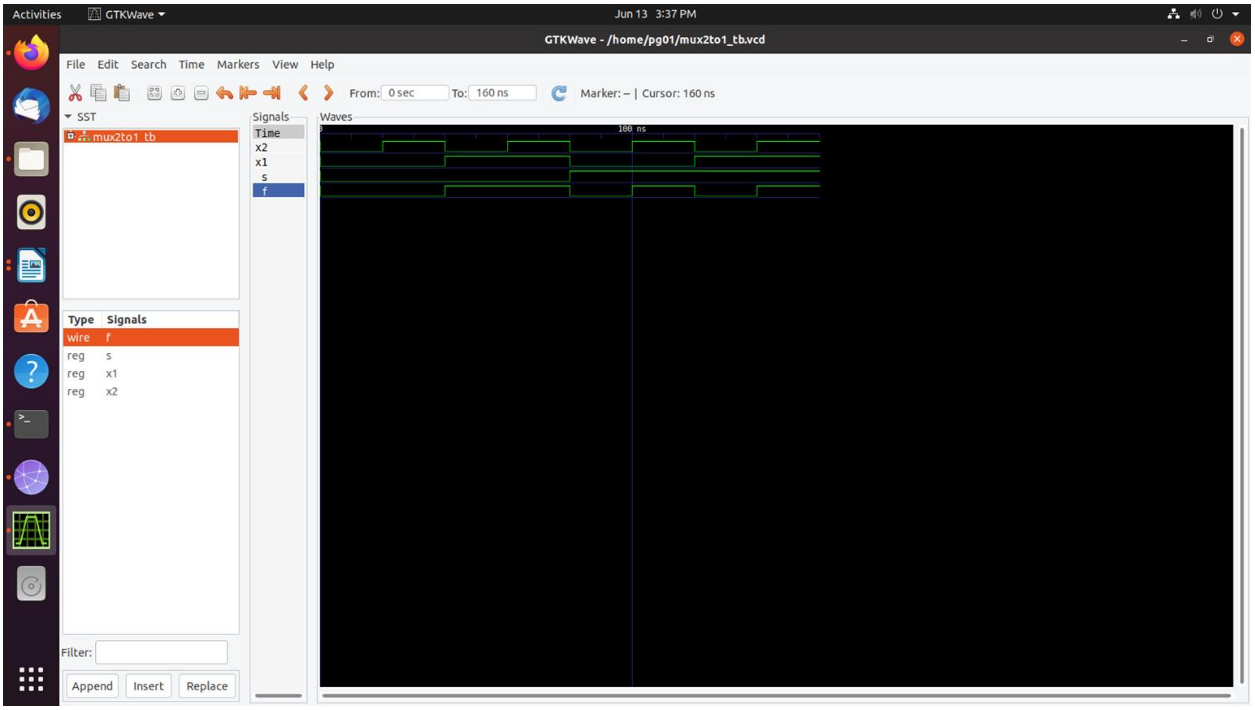
Solved Exercise

1. Write behavioral Verilog code for 2 to 1 multiplexer using **always** and **conditional** operators.

Verilog code:

```
module mux2to1 (w0, w1, s, f);
                input w0, w1, s;
                output f;
                reg f;
                always @(w0 or w1 or s)
                 f = s ? w1 : w0;
endmodule
```

```
`timescale 1ns/1ns
`include "mux2to1.v"
module mux2to1_tb();
reg x1,x2,s;
wire f;
mux2to1 ex2(x1,x2,s,f);
initial
begin
$dumpfile("mux2to1_tb.vcd");
$dumpvars(0,mux2to1_tb);
s=1'b0;x1=1'b0;x2=1'b0;
#20;
s=1'b0;x1=1'b0;x2=1'b1;
#20;
s=1'b0;x1=1'b1;x2=1'b0;
#20;
s=1'b0;x1=1'b1;x2=1'b1;
#20;
s=1'b1;x1=1'b0;x2=1'b0;
#20;
s=1'b1;x1=1'b0;x2=1'b1;
#20;
s=1'b1;x1=1'b1;x2=1'b0;
#20;
s=1'b1;x1=1'b1;x2=1'b1;
#20;
$display("test Complete");
end
endmodule
```



Lab Exercises

1. Write behavioral Verilog code for a 4 to 1 multiplexer using the **if-else** statement.
2. Write behavioral Verilog code for an 8 to 1 multiplexer using **case** statement. Use this along with a 2 to 1 multiplexer to write the hierarchical code for a 16 to 1 multiplexer.
3. Design a 3-bit comparator using only logic gates. Write behavioral Verilog code to simulate the design.
4. Using **for** loop, write behavioral Verilog code to convert an N bit grey code into equivalent binary code.

Additional Exercises

1. Write Verilog code to perform BCD to excess-3 code conversion. Use K-map to derive the simplified expressions for excess-3 code.
2. Write and simulate the Verilog code for a 4-bit comparator using 2-bit comparators.

Lab No : 4 **Date:**

DECODERS AND ENCODERS

Objectives:

In this lab, student will be able to

1. Learn the concept of decoders, encoders and priority encoders.
2. Write Verilog code for decoders, decoder trees and encoders.

Decoders

- Decoder circuits are used to decode the encoded information.
- A binary decoder, depicted in Fig. 4.1, is a logic circuit with n inputs and 2^n outputs.
- Each output corresponds to one valuation of the inputs, and only one output is asserted at a time.
- The decoder also has an enable input En, that is used to disable the outputs; if En = 0, then none of the decoder outputs is asserted.
- If En = 1, the valuation of $w_{n-1} \cdots w_1 w_0$ determines which of the outputs is asserted.
- Larger decoders can be built using smaller decoders referred to as decoder tree.

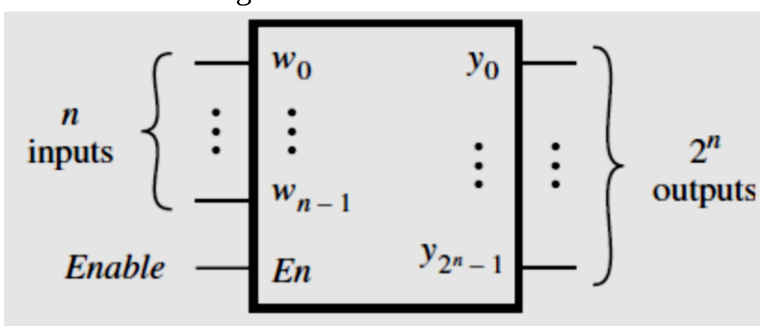


Figure 4.1 Decoder

Binary Encoders

- A binary encoder encodes information from 2^n inputs into an n-bit code, as indicated in Fig. 4.2.
- Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1.

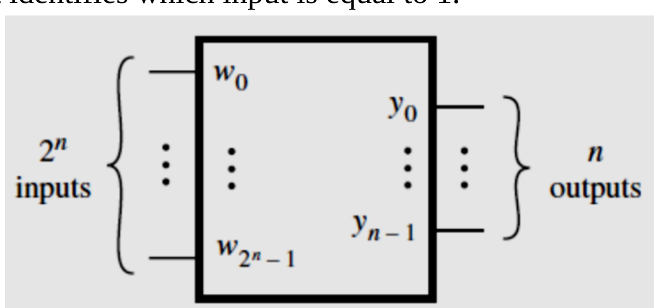


Figure 4.2 Encoder

Priority Encoder

- In a priority encoder, each input has a priority level associated with it.
- When an input with a high priority is asserted, the other inputs with lower priority are ignored. Since it is possible that none of the inputs is equal to 1, an output, z, is provided to indicate this condition.
- The truth table for a 4-to-2 priority encoder is shown in Fig. 4.3.

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Figure 4.3 Truth table for a 4 to 2 priority encoder

Casex Statement:

- Verilog provides variants of the **case** statement that treat the *z* and *x* values in a different way.
- The **casez** statement treats all *z* values as don't cares.
- The **casex** statement treats all *z* and *x* values as don't cares.

Solved exercise

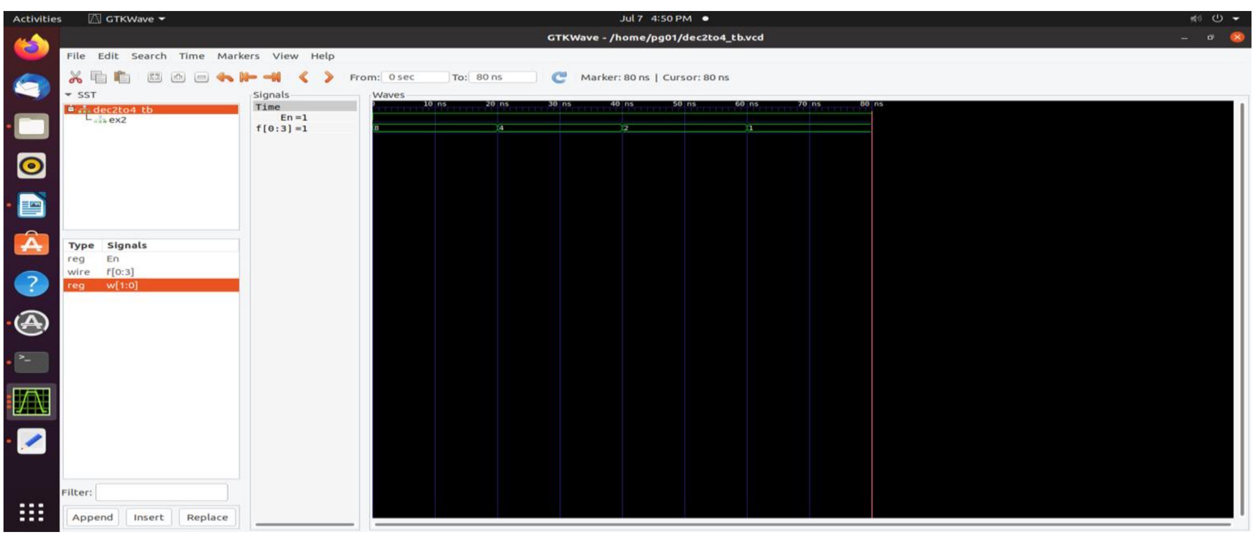
Write behavioral Verilog code for 2 to 4 binary decoder using **for** loop.

Verilog code:

```
module dec2to4 (W, Y, En);
    input [1:0] W;
    input En;
    output [0:3] Y;
    reg [0:3] Y;
    integer k;
    always @(W or En)
    for (k = 0; k <= 3; k = k+1)
    if ((W == k) && (En == 1))
    Y[k] = 1;
    else
    Y[k] = 0;
endmodule
```

Test bench

```
timescale 1ns/1ns
`include "dec2to4.v"
module dec2to4_tb();
reg [1:0]w;
reg En;
wire [0:3]f;
dec2to4 ex2(w,f,En);
initial
begin
$dumpfile("dec2to4_tb.vcd");
$dumpvars(0,dec2to4_tb);
w=0;En=1;
#20;
w=1;En=1;
#20;
w=2;En=1;
#20;
w=3;En=1;
#20;
$display("test Complete");
end
endmodule
```



Lab Exercises

1. Write behavioral Verilog code for a 2 to 4 decoder with active-high enable input and active high output using the **if-else** statement.
2. Using the 2 to 4 decoders designed in Q1 above, design a 4 to 16 decoder with active-high enable input and active high output and write the Verilog code for the same.
3. Write behavioral Verilog code for a 4 to 2 priority encoder using **casex** statement.
4. Write behavioral Verilog code for 16 to 4 priority encoder using **for** loop.

Additional Exercises

1. Write behavioral Verilog code for a 2 to 4 decoder with active-high enable input and active high output using **case** statement. Using the 2 to 4 decoders above, design a 4 to 16 decoder with active-low enable input and active high output and write the Verilog code for the same.
2. Write behavioral Verilog code for a 3 to 8 decoder with active-high enable input and active low output using the **if-else** statement. Using 3 to 8 decoders and a 2 to 4 decoder, design a 5 to 32 decoder with active-high enable input and active low output and write the Verilog code for the same.

Lab No : 5 **Date:**

MULTIPLEXER AND DECODER APPLICATIONS

Objectives:

In this lab, student will be able to

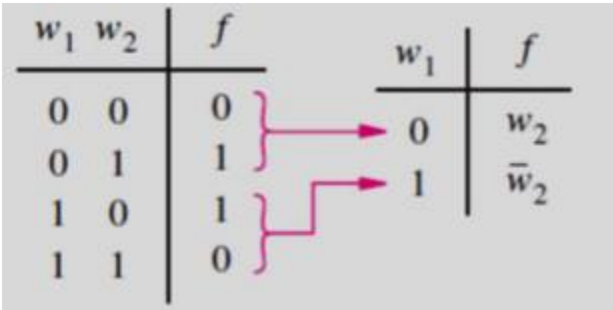
1. Understand the concept of multiplexers.
2. Learn how to synthesis logic functions using multiplexers.
3. Learn the concept of decoders, encoders and priority encoders.
4. Implement logic functions using decoders and other necessary gates.

Synthesis of Logic Functions Using Multiplexers

- Procedure to synthesis a logic function is as shown in Fig. 5.1
- One of the input signals, w_1 in this function, is chosen as the select input to the 2 to 1 multiplexer
- When $w_1 = 0$, f has the same value as input w_2, and when $w_1 = 1$, f has the value of w_2'.

Solved Exercise

Realize the function f= $w1 \oplus w2$ using a 2 to 1 multiplexer and other necessary gates. Write a Verilog code to implement the design.



w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	$\bar{w}_2$

Figure 5.1a Truth table

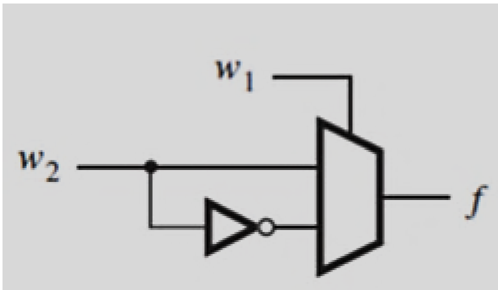
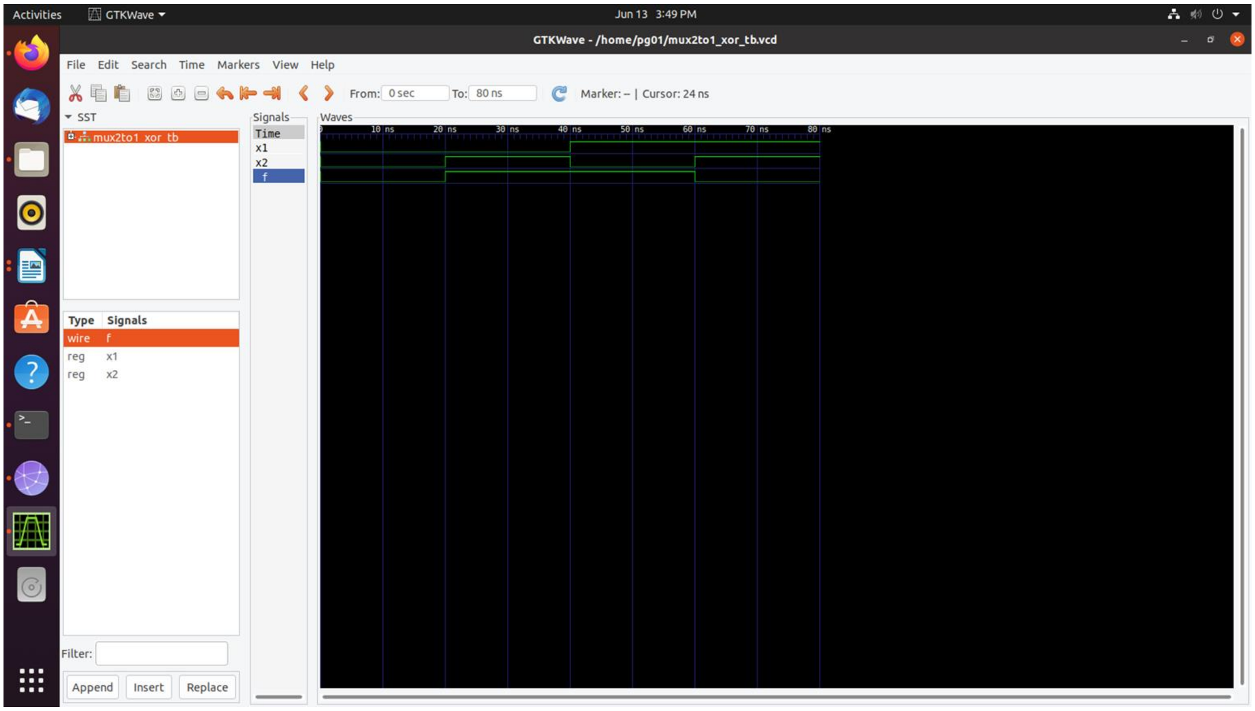


Figure 5.1b Circuit

Verilog code:

```
module mux2to1 (w1, w2, f);
        input w1, w2;
        output f;
        reg f;
        always @(w1 or w2)
        f = w1?~w2:w2;
endmodule
```

```
`timescale 1ns/1ns
`include "mux2to1_xor.v"
module mux2to1_xor_tb();
reg x1,x2;
wire f;
mux2to1_xor ex2(x1,x2,f);
initial
begin
$dumpfile("mux2to1_xor_tb.vcd");
$dumpvars(0,mux2to1_xor_tb);
x1=1'b0;x2=1'b0;
#20;
x1=1'b0;x2=1'b1;
#20;
x1=1'b1;x2=1'b0;
#20;
x1=1'b1;x2=1'b1;
#20;
$display("test Complete");
end
endmodule
```



Lab Exercises

1. Implement the expression F(A,B,C,D)=∑m (0,1,2,4,6,9,12,14) using 4:1 multiplexer and write the Verilog code for the same.
2. Write Verilog code to convert 4-bit gray code to binary code using 8:1 multiplexers.
3. Implement the function, F(a, b, c, d) = Σm (1,4,6,8,9,13,15) using a 4 to 16 binary decoder and an OR gate.
4. Write Verilog code to implement 3 input majority function using 2 to 4 decoders and other necessary gates.

Additional Exercises

1. Design and write the Verilog code for a BCD to 2421 code converter using 4 to 1 multiplexers and other necessary gates.
2. Design and implement a full adder using 2 to 4 decoder(s) and other gates.

Lab No 6: **Date:**

FLIP FLOPS

Objectives:

In this lab. student will be able to

1. Learn different types of Flip Flops.
2. Understand the concept of triggering of flip flops and different types of reset.
3. Write Verilog code for Flip Flops.

I. Flip Flops

- A flip flop circuit can maintain a binary state until directed by an input signal to switch the state.
- Major differences among various types of flip flops are in the number of inputs they process and in the manner in which the inputs affect the binary state.

Triggering of Flip-Flops:

- The state of a flip flop is switched by a momentary change in the input signal which is called triggering the flip flop.

Positive Edge and Negative Edge:

- A positive clock source remains at 0 during the interval between pulses and goes to 1 during the occurrence of a pulse.
- The pulse transition from 0 to 1 is called a **positive edge** and return from 1 to 0 is called a **negative edge.**

II. Verilog Constructs for Storage Elements

- The Verilog keywords **posedge** and **negedge** are used to implement edge-triggered circuits.
- The keyword **posedge** specifies that a change may occur only on the positive edge of Clock.
- The keyword **negedge** specifies that a change may occur only on the negative edge of Clock.

Blocking and Non-Blocking Assignments

Blocking

- A Verilog compiler evaluates the statements in an **always** block in the order in which they are written.
- If a variable is given a value by a blocking assignment statement, then this new value is used in evaluating all subsequent statements in the block.
- Denoted by the ‘=’ symbol

- Example

```
Q1 = D;
Q2 = Q1;
```

- The above statement results in Q2=Q1=D

Non-Blocking

- Verilog also provides a non-blocking assignment, denoted with ‘**<=**’.
- All non-blocking assignment statements in an **always** block are evaluated using the values that the variables have when the **always** block is entered.
- Thus, a given variable has the same value for all statements in the block.
- The meaning of non-blocking is that the result of each assignment is not seen until the end of the **always** block.
- Example

```
Q1 <= D;
Q2 <= Q1;
```

- The variables Q1 and Q2 have some value at the start of evaluating the **always** block, and then they change to a new value concurrently at the end of the **always** block.

Flip-Flops with Clear Capability

- By using a particular sensitivity list and a specific style of an **if-else** statement, it is possible to include clear (or preset) signals on flip-flops.

Solved Exercise:

Write behavioral Verilog code for positive edge-triggered D FF with synchronous reset.

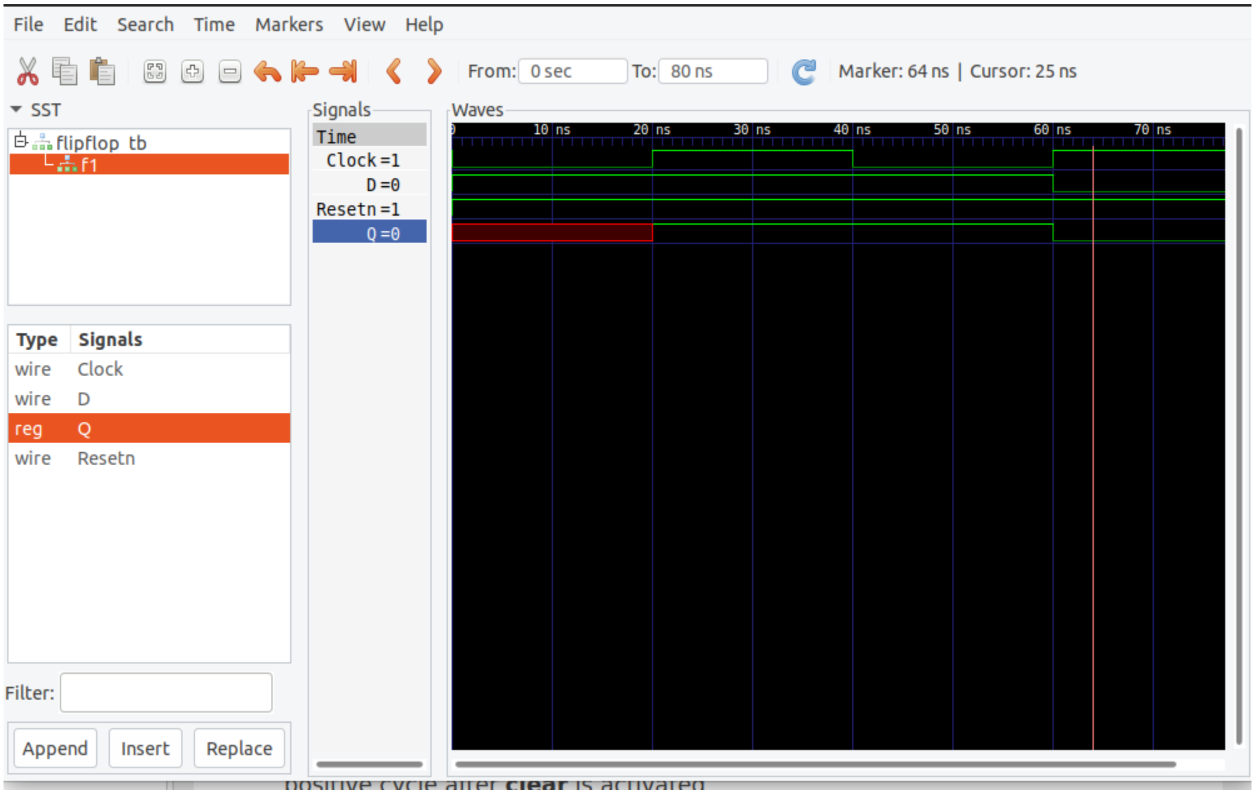
Verilog Code:

```
module flipflop (D, Clock, Resetn, Q);
        input D, Clock, Resetn;
        output Q;
        reg Q;
        always @(posedge Clock)
        if (!Resetn)
        Q <= 0;
        else
        Q <= D;
endmodule
```

TestBench Code:

```
``timescale 1ns/1ns
`include "flipflop.v"    //Name of the Verilog file

module flipflop_tb1();
reg D, Clock, Resetn;     //Input
wire Q;                   //Output

flipflop f1(D, Clock, Resetn, Q);//Instantiation of the module
initial
begin

        $dumpfile("flipflop_tb1.vcd");
        $dumpvars(0, flipflop_tb1);

  Clock=0;
    forever #20 Clock = ~Clock;
end
initial begin

        D=1;  Resetn=1;
        #20;
        D=1; Resetn=1;
        #20;
        D=1;  Resetn=1;
        #20;
        D=0; Resetn=1;
        #20;
        $display("Test complete");
end
endmodule
```

The output waveform appears as shown below.



Lab Exercises

1. Write behavioral Verilog code for a positive edge-triggered D FF with asynchronous active high reset.
2. Write behavioral Verilog code for a negative edge triggered T FF with asynchronous active low reset.
3. Write behavioral Verilog code for a positive edge-triggered JK FF with synchronous active high reset.
4. Write behavioral Verilog code for a negative edge-triggered SR FF with synchronous active low reset.

Additional Exercises

1. Write behavioral Verilog code for a negative edge triggered T FF with synchronous active high reset.
2. Write behavioral Verilog code for a positive edge-triggered JK FF with asynchronous active low reset.

Lab No 7: **Date:**

REGISTERS

Objectives:

In this lab. student will be able to

1. Understand the concept of registers and shift registers.
2. Write Verilog code for Flip Flops and registers.
3. Design and simulate the Verilog code for shift registers with parallel load capabilities .

Registers

- A register is a group of binary cells suitable for holding binary information.

Shift Registers:

- A register capable of shifting its binary information either to the right or to the left is called a shift register.
- The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next.
- The simplest possible shift register is one that uses only flip-flops, as shown below.

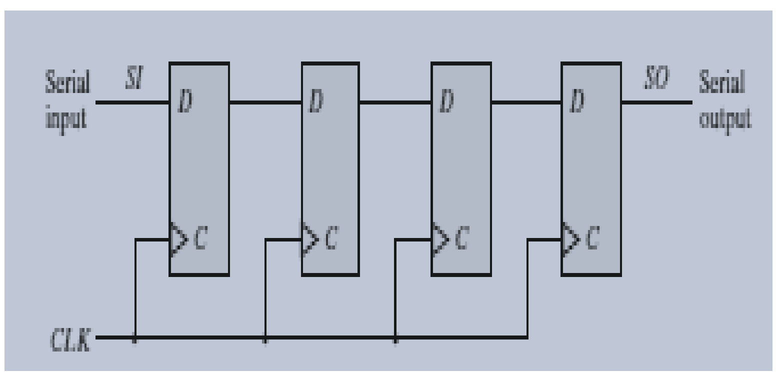


Figure 7.1 Four-bit Shift Register

- The output of a given flip-flop is connected to the D input of the flip-flop at its right.
- This shift register is unidirectional (left-to-right). Each clock pulse shifts the contents of the register one-bit position to the right.
- The *serial input* determines what goes into the leftmost flip-flop during the shift. The *serial*
- *output* is taken from the output of the rightmost flip-flop.
- Shift register that can shift in both directions is called as *bidirectional* shift register.

Shift Registers with Parallel Load Capabilities:

- If a parallel load capability is added to a shift register, then data entered in parallel can be taken out in serial fashion by shifting the data stored in the register.

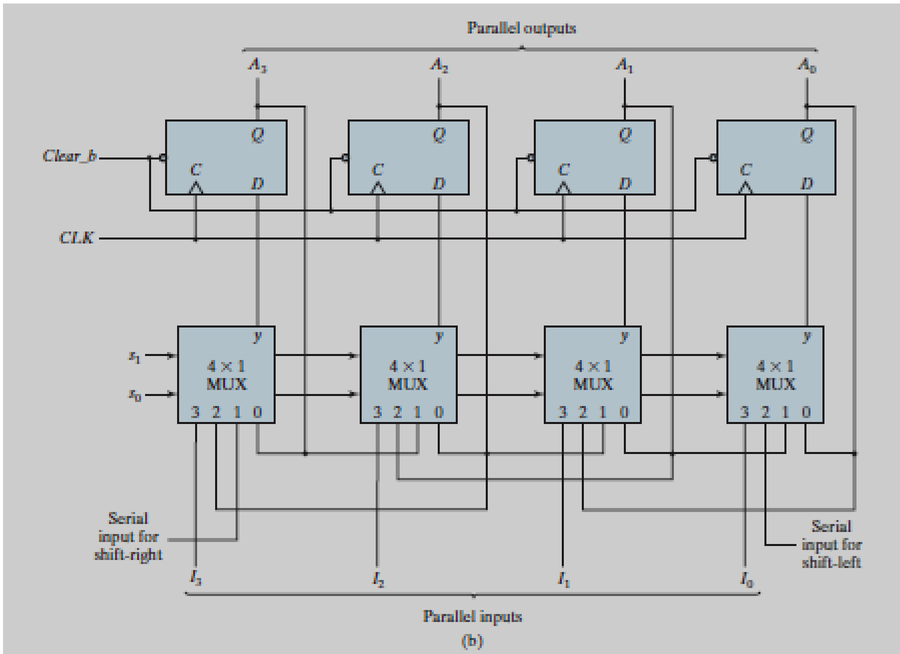


Figure 7.2 Four-bit Shift Register with Parallel Load

- As shown in Fig. 7.2, the most general shift register with parallel load has the following capabilities:
  - A *clear* control to clear the register to 0.
  - A *clock* input to synchronize the operations.
  - A *shift - right* control to enable the shift-right operation and the *serial input* and
  - *output* lines associated with the shift right.
  - A *shift - left* control to enable the shift-left operation and the *serial input* and *output*
  - lines associated with the shift left.
  - A *parallel - load* control to enable a parallel transfer and the *n* input lines associated with the parallel transfer.
  - *n* parallel output lines.
  - A control state that leaves the information in the register unchanged in response to the clock. Other shift registers may have only some of the preceding functions, with at least one shift operation.

Solved Exercise:

Write structural Verilog code for a 4-bit register.

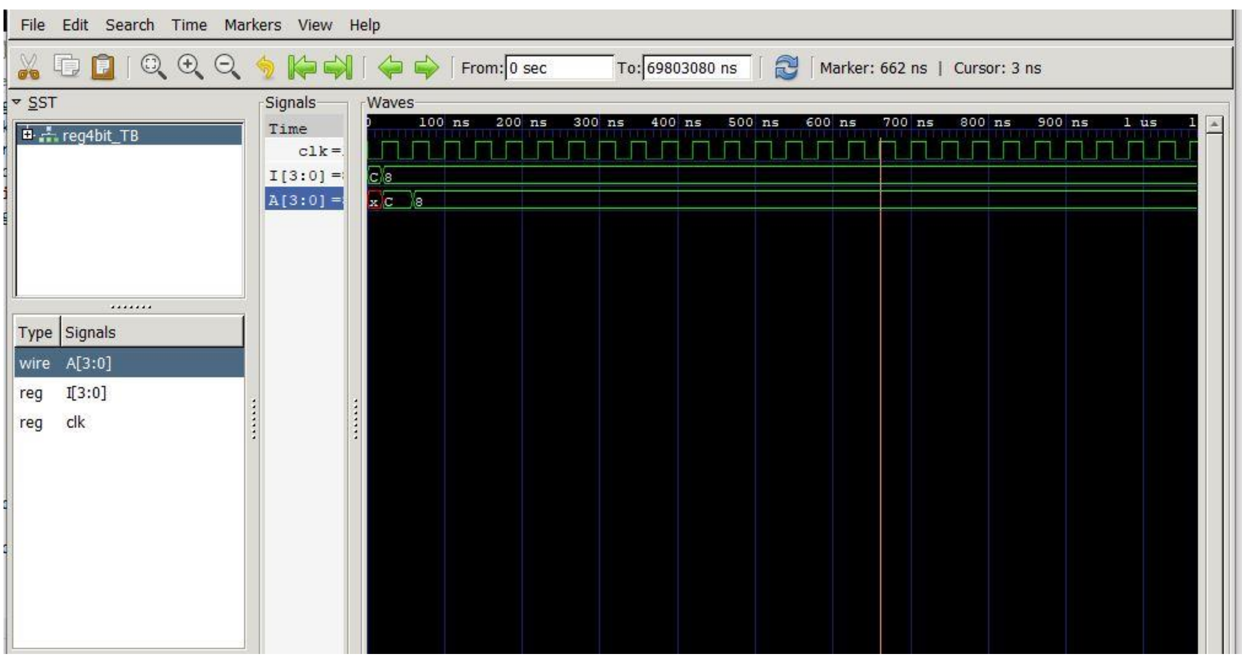
Verilog Code:

```
module reg4bit(I,clk,A);
input [3:0]I;
input clk;
output [3:0]A;

dffp s1(I[3],clk,A[3]);
dffp s2(I[2],clk,A[2]);
dffp s3(I[1],clk,A[1]);
dffp s4(I[0],clk,A[0]);
endmodule

module dffp (D, Clock, Q);
input D, Clock;
output Q;
reg Q;
always @(posedge Clock)
Q<= D;
endmodule
```

TestBench Code

```
`timescale 1ns/1ns
`include "reg4bit.v"
module reg4bit_TB();
reg [3:0]I;
reg clk;
wire [3:0]A;
reg4bit a1(I,clk, A);
initial
begin
clk=0;
forever #20 clk = ~clk;
end
initial
begin
        $dumpfile("reg4bit_TB.vcd");
        $dumpvars(0,reg4bit_TB);
        I = 12;  #20;
        I = 8;    #20;
        $display("Test complete");
end
endmodule
```

The output waveform appears as shown below.



Lab Exercises

1. Write structural Verilog code for a 6-bit shift register.
2. Write Verilog code for an N bit register.

3. Design and simulate a shift register with the parallel load that operates according to the following function table:

Shift	Load	Register Operation
0	0	Shift left
0	1	Load parallel data
1	X	No change

Additional Exercises

1. Write Verilog code for an N bit shift register.
2. Design and simulate a shift register with the parallel load that operates according to the following function table:

Shift	Load	Register Operation
0	0	Shift left
0	1	Load parallel data
1	0	Complement
1	1	No change

Lab No: 8 **Date:**

DESIGN OF SEQUENTIAL CIRCUITS

Objectives:

In this lab, student will be able to

1. Describe the behavior of the sequential circuit using state tables and state diagrams.
2. Apply the state reduction techniques to simplify the design of sequential circuits.
3. Perform the state assignment techniques to design the sequential circuits.
4. Design sequential circuits using various flip-flops to meet different applications.

State Table

- The time sequence of inputs, outputs, and flip-flop states can be enumerated in a state table.
- The sample state table for a sequential circuit is shown in Table 8.1.

Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Table 8.1 State Table for a Sequential Circuit.

- The table consists of four sections labelled present state, input, next state, and output .
  - ➢ The present-state section shows the states of flip-flops A and B at any given time t .
  - ➢ The input section gives a value of x for each possible present state.
  - ➢ The next-state section shows the states of the flip-flops one clock cycle later, at time t + 1.
  - ➢ The output section gives the value of y at time t for each present state and input condition.
- The derivation of a state table requires listing all possible binary combinations of present states and inputs.

State Diagram

- The information available in a state table can be represented graphically in the form of a state diagram.
- In this type of diagram, a state is represented by a circle, and the (clock-triggered) transitions between states are indicated by directed lines connecting the circles.
- The state diagram provides the same information as the state table. The Fig. 9.1 shows the state diagram for the state table mentioned in Table 8.1.

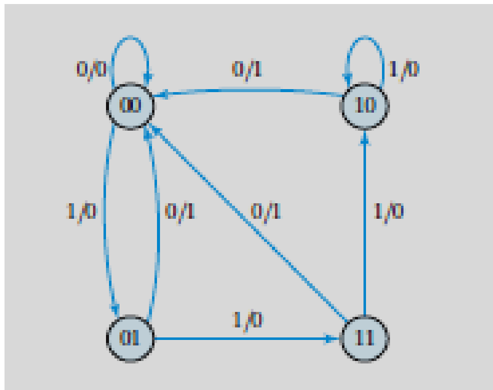


Figure 8.1 State Diagram of the Sequential Circuit.

State Reduction

- The reduction in the number of flip-flops in a sequential circuit is referred to as the state-reduction problem.
- State-reduction algorithms are concerned with procedures for reducing the number of states in a state table, while keeping the external input–output requirements unchanged.
- Since m flip-flops produce 2m states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops.
- An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates to realize its next state and output logic.

State Assignment

- In order to design a sequential circuit with physical components, it is necessary to assign unique coded binary values to the states.
- For a circuit with m states, the codes must contain n bits, where $2^n \geq$ m. For example, with three bits, it is possible to assign codes to eight states, denoted by binary numbers 000 through 111.
- Unused states are treated as don't-care conditions during the design.

Design Procedure:

- Design procedures or methodologies specify hardware that will implement a desired behaviour.
- The design of a clocked sequential circuit starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which the logic diagram can be obtained.
- In contrast to a combinational circuit, which is fully specified by a truth table, a sequential circuit requires a state table for its specification.
- The first step in the design of sequential circuits is to obtain a state table or an equivalent representation, such as a state diagram.
- A synchronous sequential circuit is made up of flip-flops and combinational gates. The design of the circuit consists of choosing the flip-flops and then finding a combinational gate structure that, together with the flip-flops, produces a circuit which fulfils the stated specifications.
- The number of flip-flops is determined from the number of states needed in the circuit and the choice of state assignment codes. The combinational circuit is derived from the state table by evaluating the flip-flop input equations and output equations.
- In fact, once the type and number of flip-flops are determined, the design process involves a transformation from a sequential circuit problem into a combinational circuit problem. In this way, the techniques of combinational circuit design can be applied.
- The procedure for designing synchronous sequential circuits can be summarized by a list of recommended steps:

  1. From the word description and specifications of the desired operation, derive a state diagram for the circuit.

  2. Reduce the number of states if necessary.

  3. Assign binary values to the states.

  4. Obtain the binary-coded state table.

  5. Choose the type of flip-flops to be used.

  6. Derive the simplified flip-flop input equations and output equations.

  7. Draw the logic diagram.

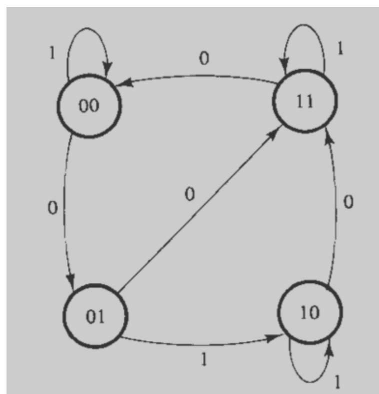
Lab Exercises

1. Analyze the given state table of a synchronous sequential circuit and perform state reduction. Using the reduced state table, design and simulate the sequential circuit using D flip-flops.

Present State	**Next State**		**Output**	
	x=0	**x=1**	**x=0**	**x=1**
a	f	b	0	0

b	d	c	0	0
c	f	e	0	0
d	g	a	1	0
e	d	c	0	0
f	f	b	1	1
g	g	h	0	1
h	g	a	1	0

2. Design the synchronous sequential circuit for the following state diagram using T FF. Write the Verilog code for the derived circuit.

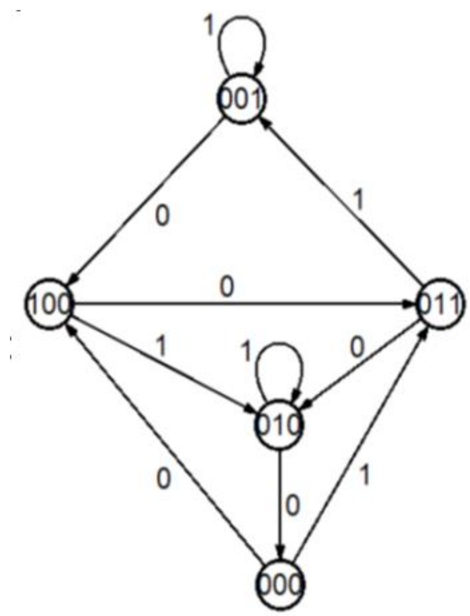


Additional Exercises

1. Design and simulate a sequential circuit with two *JK* flip-flops *A* and *B* and two inputs *E* and *x*. If $E = 0$, the circuit remains in the same state regardless of the value of x .

- When $E = 1$ and $x = 1$, the circuit goes through the state transitions from 00 to 01, to 10, to 11, back to 00, and repeats.
- When $E = 1$ and $x = 0$, the circuit goes through the state transitions from 00 to 11, to 10, to 01, back to 00, and repeats.

2. Design the synchronous sequential circuit for the following state diagram using JK FF. Write the Verilog code for the derived circuit.


1
001
0
1
100
0
011
1
1
0
010
0
0
1
000

Lab No: 9 **Date:**

COUNTERS

Objectives:

In this lab, student will be able to

1. Learn the concept of synchronous/asynchronous up/down counters.
2. Learn the concept of ring and Johnson counters.
3. Write Verilog code for different types of counters.

Counters:

- A counter is essentially a register that goes through a predetermined sequence of states upon the application of input pulses.
- A binary counter with a reverse count is called a binary down counter.

Ripple Counters

- ➢ Also called as asynchronous counters.
- ➢ The CP inputs of all flip flops (except the first) are triggered not by the incoming pulses, but by the transition that occurs in other flip flops.
- ➢ Four-bit binary ripple counter is shown in Fig. 9.1

Synchronous Counters

- ➢ The input pulses are applied to the CP input of all the flip flops.
- ➢ The common pulse triggers all flip flops simultaneously.
- ➢ The change of state of a particular flip flop is dependent on the present state of other flip flops.
- ➢ For synchronous sequential circuits the design procedure is as follows:
  2. From the given information (word description/state diagram/timing diagram/other pertinent information) about the circuit, obtain the state table.
  3. Determine the number of flip flops needed.
  4. From the state table, derive the circuit excitation and output tables.
  5. Derive the circuit output functions and the flip flop input functions by simplification.
  6. Draw the logic diagram.

Ring Counter

- ➢ Circular shift register with only one flip flop being set at any particular time, all others are cleared.
- ➢ N bit ring counter will have N states and requires N flip flops.
- ➢ Fig. 9.2 shows a 4-bit ring counter using decoder and counter.

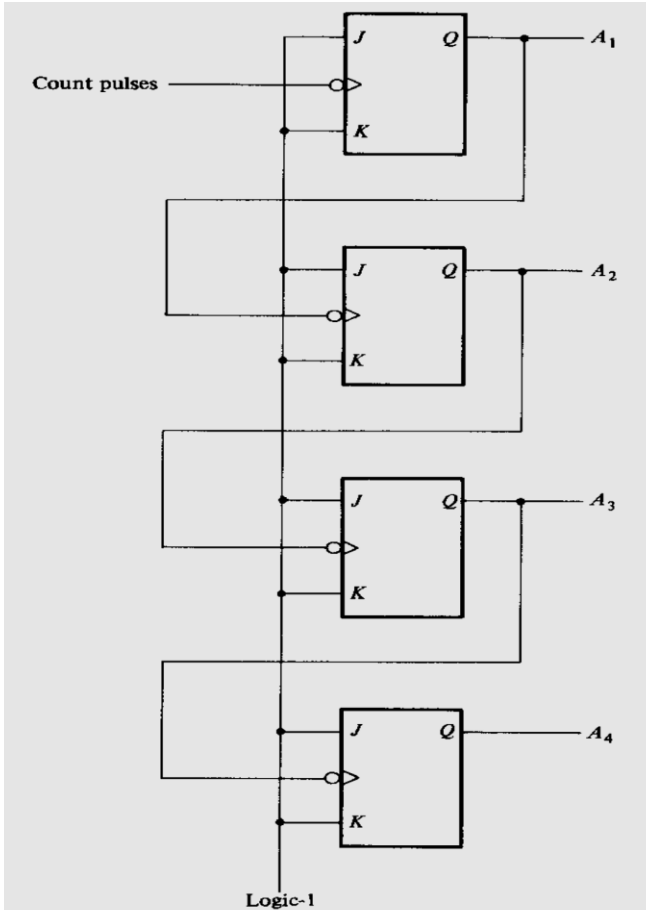


Figure 9.1: 4-bit binary ripple counter

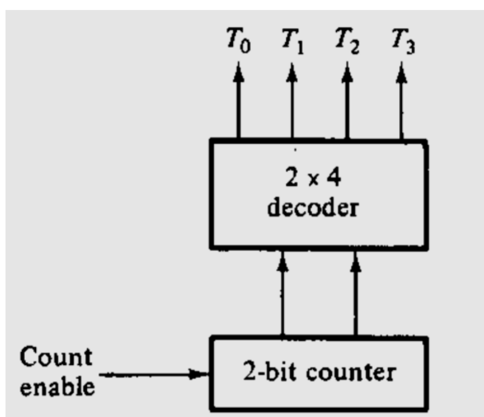


Figure 9.2: 4-bit ring counter

Johnson counter or Switch tail ring counter

- Circular shift register with the complement output of the last flip flop connected to the input of the first flip flop.
- k-bit switch tail ring counter with 2k decoding gates provide outputs for 2k timing signals.

- k- bit Johnson counter requires k flip flops.
- Fig. 9.3 shows a 4-bit Johnson counter

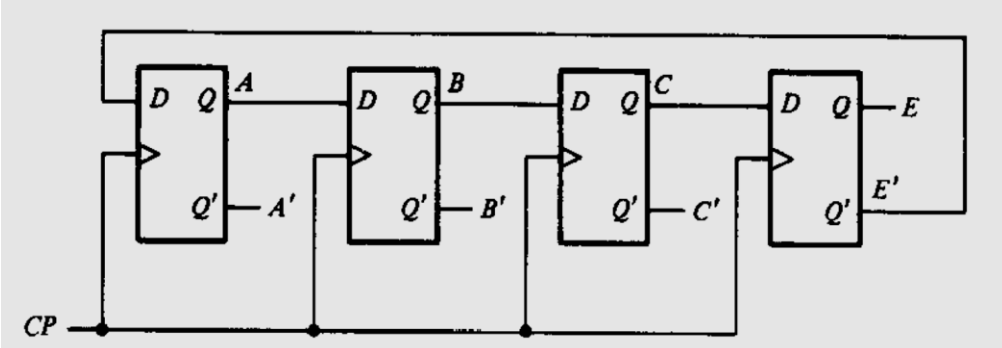


Figure 10.3 4-bit Johnson counter

Solved Exercise

Write Verilog code in structural style for a 3-bit synchronous counter using T flip-flops.

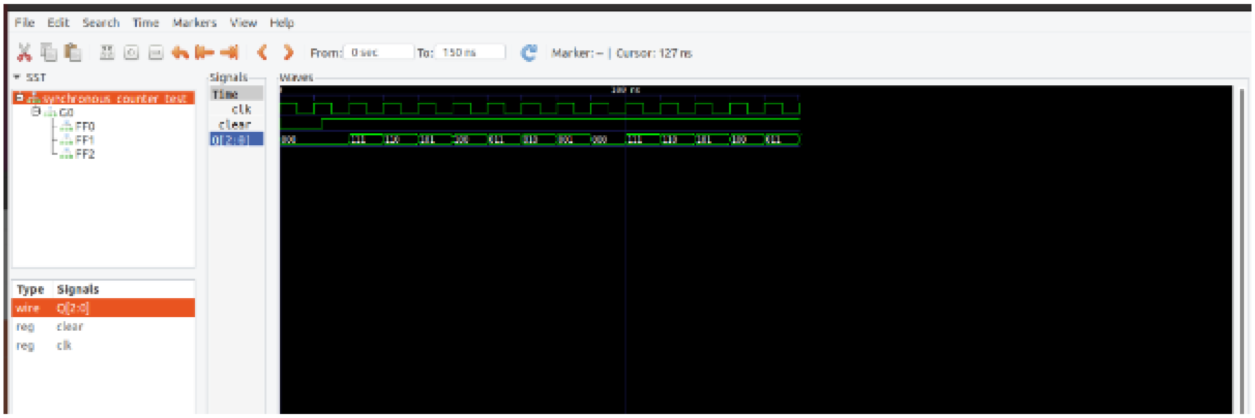
```
// Module for positive edge triggered T FF
module tff (Q, T, clear, clk);
input T,clear, clk;
output reg Q;
always @ (posedge clk or negedge clear)
begin
if ( clear==0)
Q<=0;
else if ( T==1)
Q<=~Q;
else Q<=Q;
end
endmodule

module synchronous_counter (clear, clk, Q);
input clear, clk;
output [2:0] Q;
wire w1, w2, w3;
tff FF0 (Q[0], 1'b1, clear, clk);
tff FF1 (Q[1], w1 , clear, clk);
tff FF2 (Q[2], w3, clear, clk);
nand G0 (w1,1'b1, Q[0]);
nand G1 (w2,1'b1, Q[1]);
and G2 (w3, w1, w2);
```

```
endmodule
```

TestBench Code

```
`timescale 1ns/1ns
`include "synchronous_counter.v"    //Name of the Verilog file


module synchronous_counter_test ;
reg clear, clk;
wire [2:0] Q;
// Instanstiation
synchronous_counter G0 (clear, clk, Q);
always #5 clk=~clk;
initial
begin
clear=0;
clk=1;
#150 $finish;
end
initial
begin
$dumpfile("synchronous_counter.vcd");
$dumpvars(0,synchronous_counter_test);
$monitor($time, " clear=%b, Q=%3b", clear, Q );
#12 clear =1;
end
endmodule
```

The output waveform appears as shown below.



Lab Exercises

1. Write behavioral Verilog code to implement 4-bit ring counter.
2. Design and simulate 5-bit Johnson counter.
3. Write Verilog code to implement 4-bit asynchronous up counter.

Additional Exercises

1. Write Verilog code for the 4-bit synchronous counter using JK flipflops and control input "W". When W=1 counter should count up or else when W=0 counter should count down.
2. Design and simulate a sequential circuit which produces a high output for every 5th clock pulse.
3. Implement two-digit BCD counter by including two modulo-10 counters. Validate the design by writing behavioral Verilog code.

Lab No: 10 **Date:**

COUNTERS WITH NON-BINARY SEQUENCE

Objectives:

In this lab, student will be able to

1. Learn the concept of synchronous up/down counters.
2. Learn the concept of designing non-binary sequence counters using various flip-flops.
3. Write Verilog code for different types of counters.

Counters with non-binary Sequence:

A counter with *n* flip-flops may have a binary sequence of less than 2" states. Counters may follow an arbitrary sequence that may not be the straight binary sequence. In any case, the design procedure is the same. The state table is obtained from the count sequence and the counter is designed using sequential-circuit design techniques. The unused states are treated as don't cares.

Solved Exercise

Design and write Verilog code for a three-bit counter that counts in the sequence 0, 1, 2, 4, 5, 6, 0 using JK flip-flops.

Excitation Table for Counter

Present State			Next State			Flip-Flop Inputs					
A	*B*	*C*	*A*	*B*	*C*	*JA*	*KA*	*JB*	*KB*	*JC*	*KC*
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	0	0	1	X	X	1	0	X
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X

In the above table, the count has a repeated sequence of six states, with flip-flops *B and* C repeating the binary count 00, 01, 10, while flip-flop *A* alternates between 0 and 1 every three counts. The count sequence is not straight binary and two states, 011 and 111, are not included in the count. Inputs *KB* and *KC* have only 1's and X's in their columns, so these inputs are always equal to 1. The other flip-flop input functions can be simplified using min terms 3 and 7 as don't-care conditions. The simplified functions are:

$JA = B \qquad KA = B$

$JB = C \qquad KB = 1$

$JC = B' \qquad KC = 1$

The logic diagram of the counter is shown in Fig. 10.1.

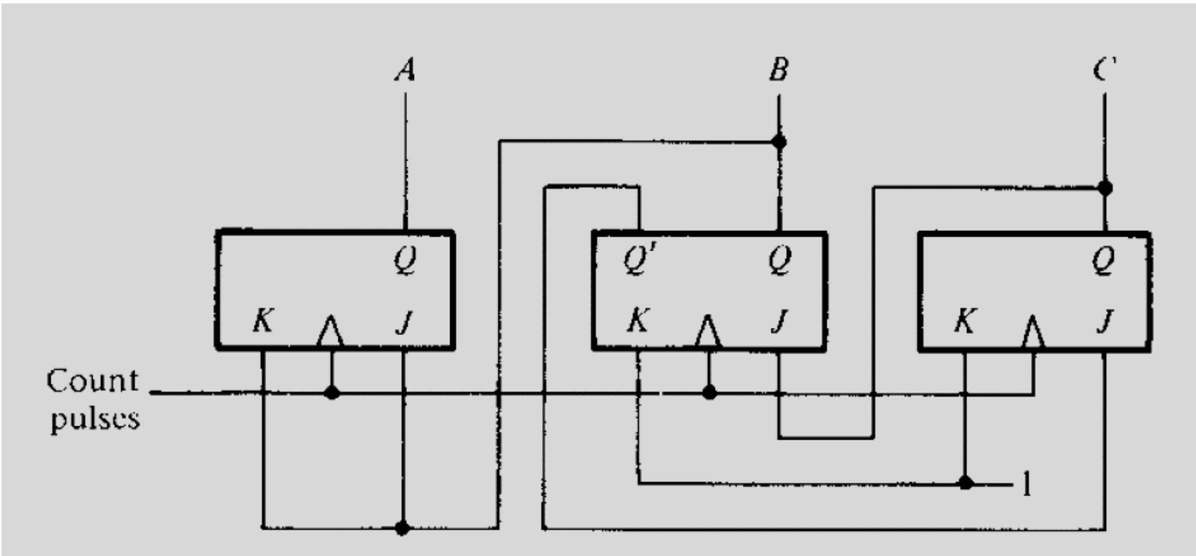


Figure 10.1 Logic diagram of counter

Verilog Code:

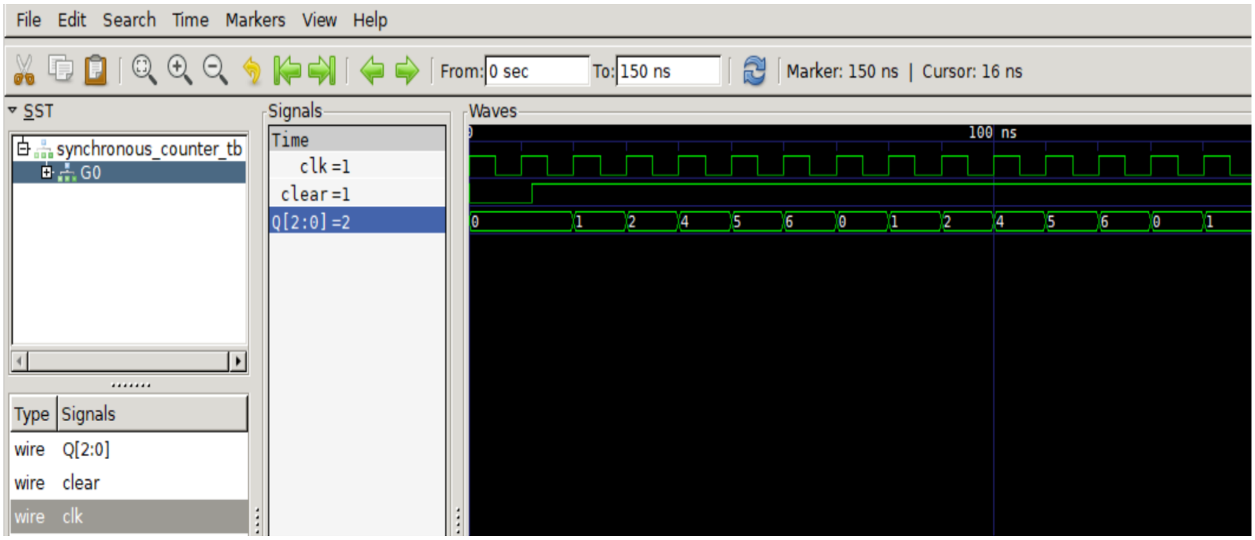
```
// Module for positive edge triggered JK FF
// Module for positive edge triggered JK FF
module jkff (Q, J, K, clear, clk);
input J,K,clear, clk;
output reg Q;
always @ (posedge clk or negedge clear)
begin
if ( clear==0 )
Q<=0;
else if (J==0 && K==1)
Q<=0;
else if(J==1 && K==0)
Q<=1;
else if ( J==1 && K==1)
Q<=~Q;
else Q<=Q;
end
```

```
endmodule

module synchronous_counter (clear, clk, Q);
input clear, clk;
output [2:0] Q;
jkff FF0 (Q[0],~Q[1], 1'b1, clear, clk);
jkff FF1 (Q[1], Q[0], 1'b1, clear, clk);
jkff FF2 (Q[2], Q[1],Q[1],clear, clk);
endmodule
```

TestBench Code:

```
`timescale 1ns/1ns
`include "synchronous_counter.v"    //Name of the Verilog file
module synchronous_counter_tb ;
reg clear, clk;
wire [2:0] Q;
// Instanstiation
synchronous_counter G0 (clear, clk, Q);
always #5 clk=~clk;
initial
begin
clear=0;
clk=1;
#150 $finish;
end
initial
begin
$dumpfile("synchronous_counter_tb.vcd");
$dumpvars(0,synchronous_counter_tb);
$monitor($time, " clear=%b, Q=%3b", clear, Q );
#12 clear =1;
end
endmodule
```

The output waveform appears as shown below.



Lab Exercises

1. Design a counter with the following repeated binary sequence: 0, 1, 2, 4, 5, 6, 7. Use JK flip-flops.
2. Derive the expressions for the design of a counter with the following repeated binary sequence: 1, 2, 3, 0, 7, 6, 5. Use SR flip-flops.
3. Design a counter with *T* flip-flops that goes through the following binary repeated sequence: 0, 1, 3, 7, 6, 4.

Additional Exercises

1. Design a counter with the following repeated binary sequence: 0, 1, 2, 5, 4, 6, 8, 9, 12, 11, 13, 15. Use T flipflops.
2. Design and simulate a synchronous counter using T flipflops with a control input w which operates in the manner given. When w=0, the counter should follow the repeated binary sequence: 0, 2, 3, 4, 6 and when w=1, the counter should follow the repeated binary sequence: 6, 4, 3, 2, 0.

References:

1. Stephen Brown and Zvonko Vranesic, “Fundamentals of digital logic with Verilog design”, Tata MGH publishing Co.Ltd., 3rd edition, 2014.
2. M.Morris Mano, “Digital design”, PHI Pvt. Ltd., 2nd edition, 2000