

**MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL ACADEMY OF HIGHER EDUCATION
Manipal – 576 104**

SCHOOL OF COMPUTER ENGINEERING

CERTIFICATE

This is to certify that Ms./Mr.
Reg. No. Section: Roll No: has
satisfactorily completed the lab exercises prescribed for Digital Systems Lab [CSS XXX] of
Second Year B. Tech. Degree in Computer Science and Engg. at MIT, Manipal, in the academic
year 2025-2026.

Date:

Signature
Faculty in Charge

CONTENTS

LAB NO.	TITLE	PAGE NO.	REMARKS
	Course Objectives and Outcomes	i	
	Evaluation plan	i	
	Instructions to the Students	i	
	Sample Lab Observation note preparation	iii	
1	Introduction to Verilog	1	
2	Simplification using K-maps	13	
3	Multilevel Synthesis	15	
4	Arithmetic circuits	16	
5	Comparators and Multiplexers	21	
6	Decoders and encoders	27	
7	Flip flops	31	
8	Registers	35	
9	Design of Sequential Circuits	41	
10	Counters	44	
13	References	49	

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 10 evaluations reduced to marks out of 60.
- End semester assessment of 2-hour duration: 40 %

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.

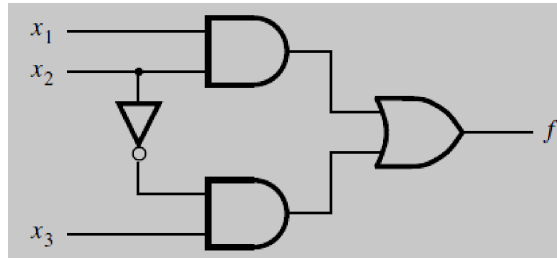
SAMPLE LAB OBSERVATION NOTE PREPARATION

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

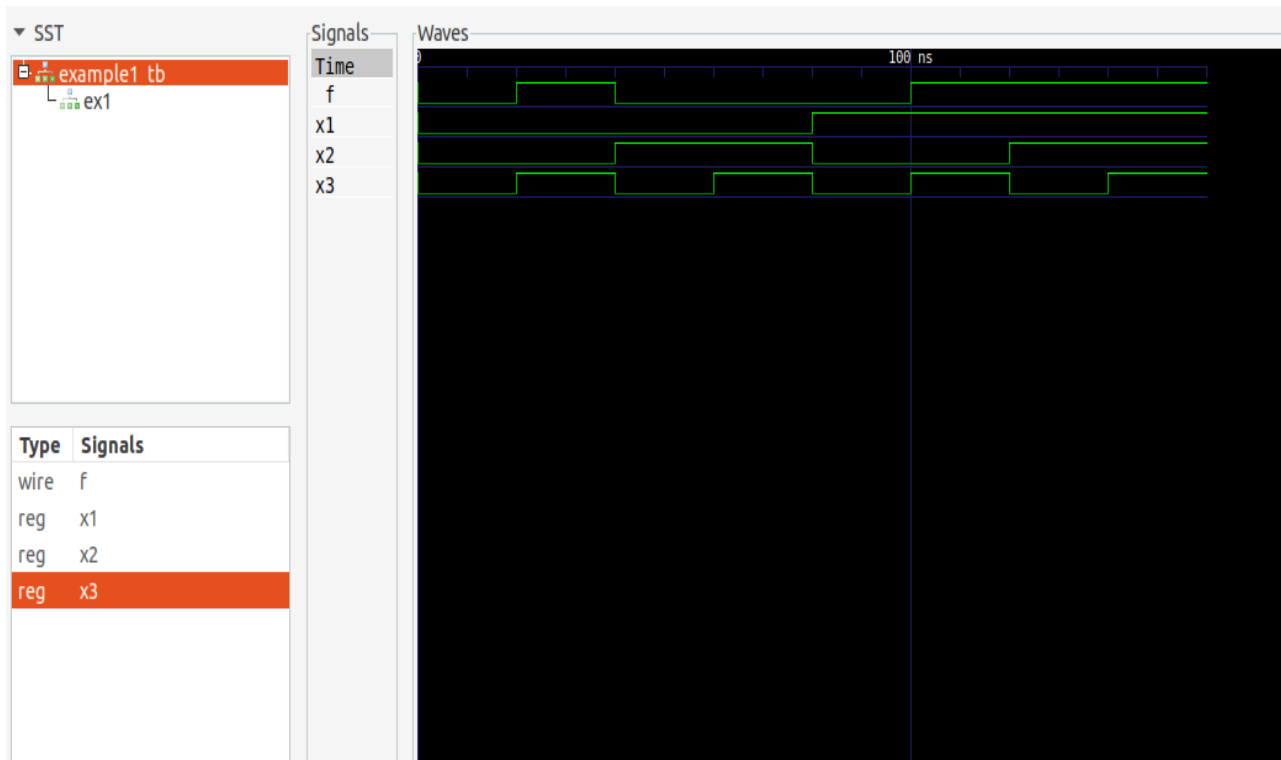
endmodule

```

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:



INTRODUCTION TO VERILOG

Objectives:

In this lab, student will be able to

1. Learn the basic concepts of logic circuits and analyze the logic network.
2. Write the Truth table and Timing diagram.
3. Understand different representation of logic circuits in Verilog.
4. Learn the different tools available in the CAD system.
5. Write and simulate logic circuits using Verilog.

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.

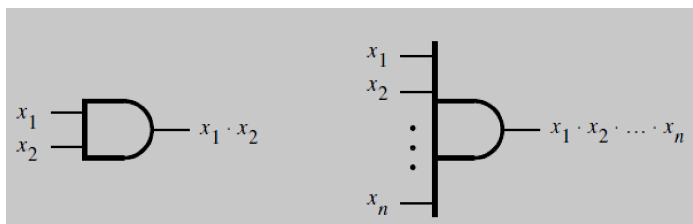


Figure 1.1

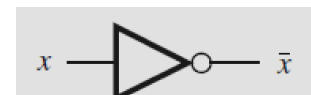


Figure 1.2

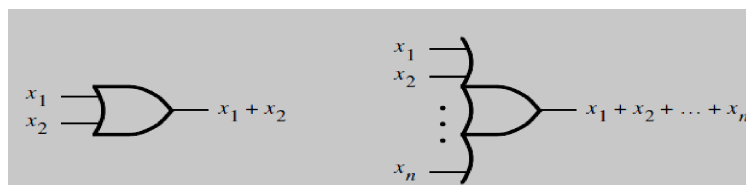


Figure 1.3

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

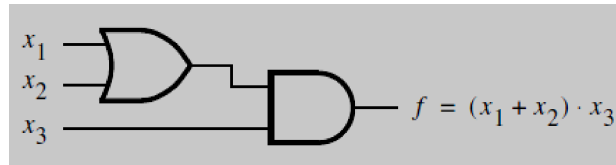


Figure 1.4

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 x_1 and x_2 can have.
- The next column defines the AND operation for each combination of values of x_1 and x_2 , 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 x_1 and x_2 . The analysis of these input values at various intermediate points is shown in Fig. 1.7.

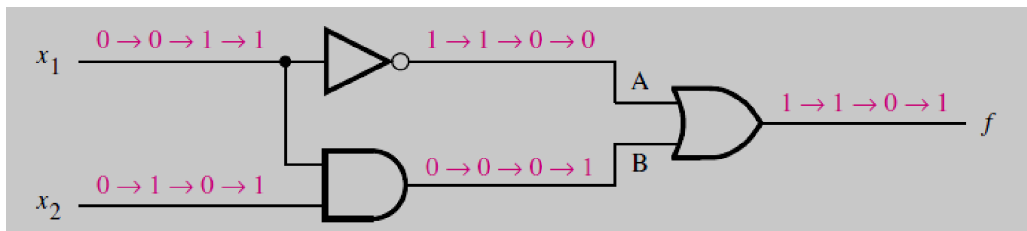


Figure 1.6

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.

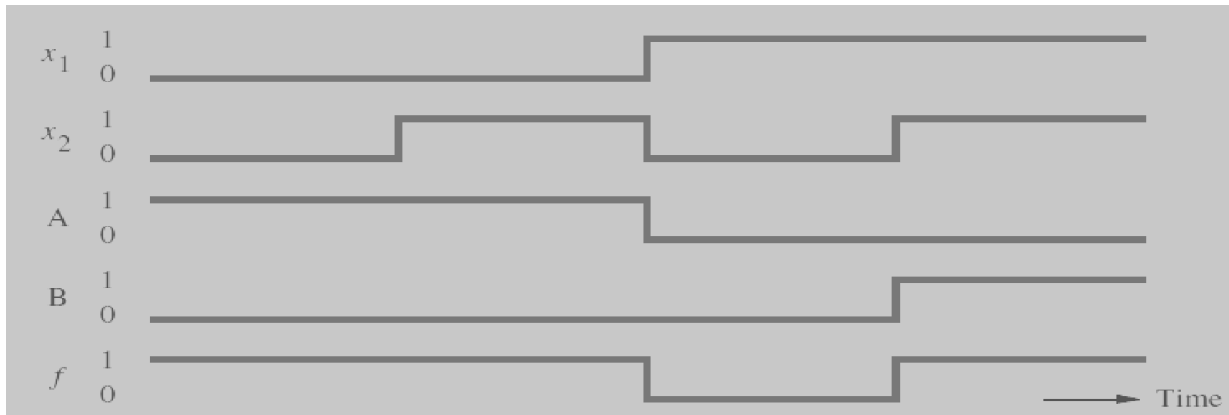


Figure 1.8

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, $g(x_1, x_2) = f(x_1, x_2)$, which indicates that the two networks are functionally equivalent.

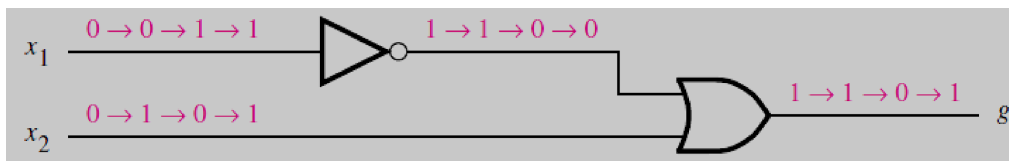


Figure 1.9

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	$f = (a \cdot b \cdot \dots)$	and (f, a, b, ...)
nand	$f = \overline{(a \cdot b \cdot \dots)}$	nand (f, a, b, ...)
or	$f = (a + b + \dots)$	or (f, a, b, ...)
nor	$f = \overline{(a + b + \dots)}$	nor (f, a, b, ...)
xor	$f = (a \oplus b \oplus \dots)$	xor (f, a, b, ...)
xnor	$f = (a \odot b \odot \dots)$	xnor (f, a, b, ...)
not	$f = \bar{a}$	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	2
		OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
		Reduction OR	1
	~	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1
Arithmetic	+	Addition	2
	-	Subtraction	2
	-	2's complement	1
	*	Multiplication	2
	/	Division	2
Relational	>	Greater than	2
	<	Lesser than	2
	>=	Greater than or equal to	2
	<=	Lesser than or equal to	2
Equality	==	Logical equality	2
	!=	Logical inequality	2
Shift	>>	Right shift	2
	<<	Left shift	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;
```

```

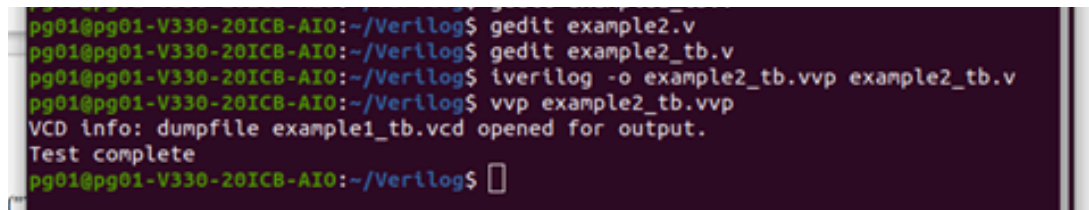
    $display("Test complete");
end

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

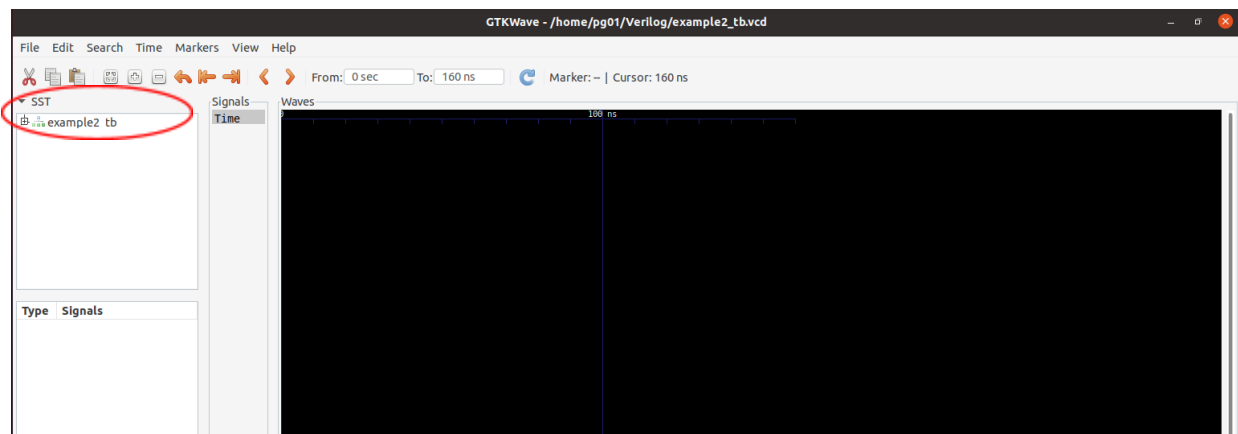


```

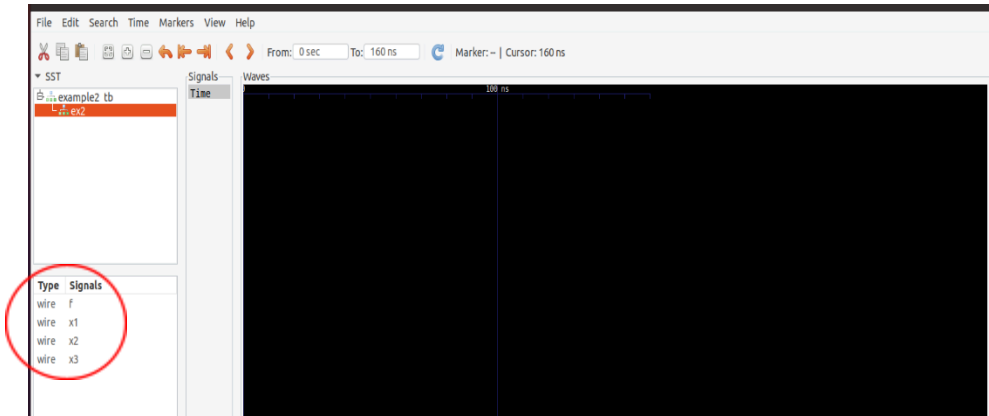
pg01@pg01-V330-20ICB-AIO:~/Verilog$ gedit example2.v
pg01@pg01-V330-20ICB-AIO:~/Verilog$ gedit example2_tb.v
pg01@pg01-V330-20ICB-AIO:~/Verilog$ iverilog -o example2_tb.vvp example2_tb.v
pg01@pg01-V330-20ICB-AIO:~/Verilog$ vvp example2_tb.vvp
VCD info: dumpfile example1_tb.vcd opened for output.
Test complete
pg01@pg01-V330-20ICB-AIO:~/Verilog$

```

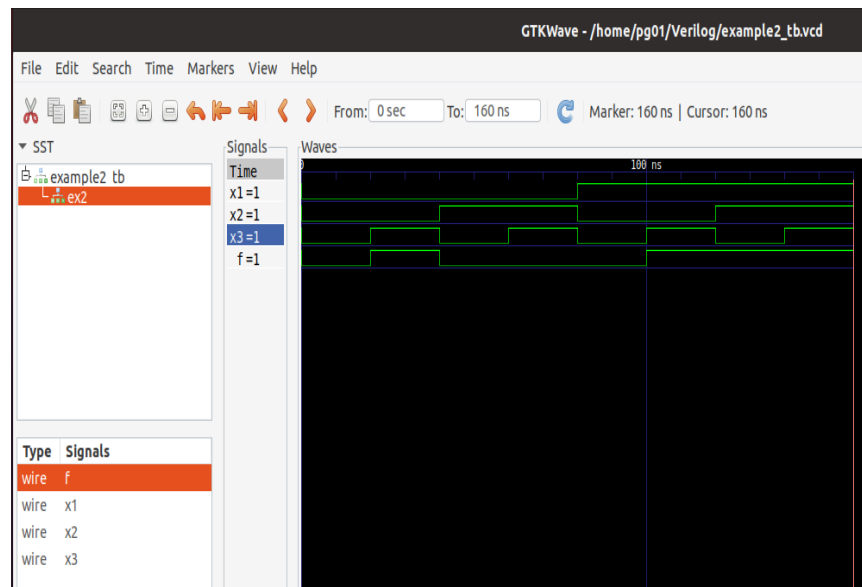
7. Open **gtkwave** by double clicking the example2_tb.vcd file available in your directory or type **gtkwave** in the terminal and open the example2_tb.vcd file in gtkwave. A window appears as shown below.



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.

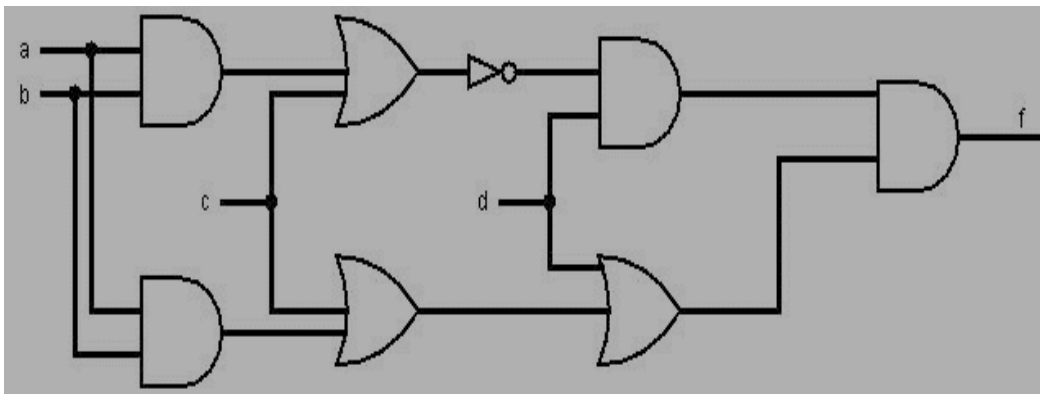
```

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

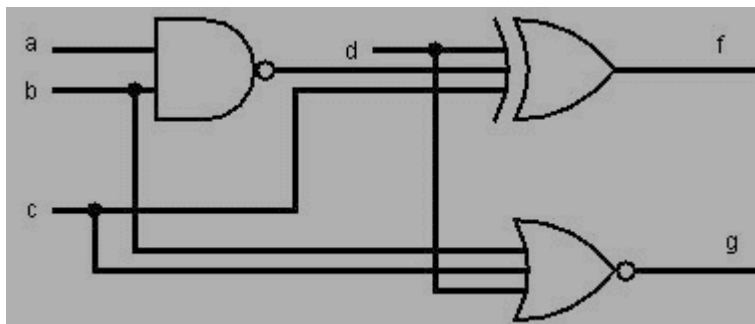
```

Lab Exercises

1. Write the Verilog code to implement the circuit in the following figure.



- i. Using gate-level primitives
 - ii. Using continuous assignment statements.
2. Write the Verilog code to implement the circuit in the following figure.



- i. Using gate-level primitives
- ii. Using continuous assignment statements.

Additional Exercises

1. Write Verilog code to describe the following functions

$$f1 = ac' + bc + b'c'$$

$$f2 = (a + b' + c)(a + b + c')(a' + b + c')$$

2. Check whether f1 and f2 in question 1 are functionally equivalent or not.

Lab No 2:

Date:

SIMPLIFICATION USING K-MAP

Objectives:

In this lab, student will be able to

1. Understand the steps for optimization using K-map.
2. Design minimum cost circuit.
3. Write Verilog code for the simplified expression.

I. 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.

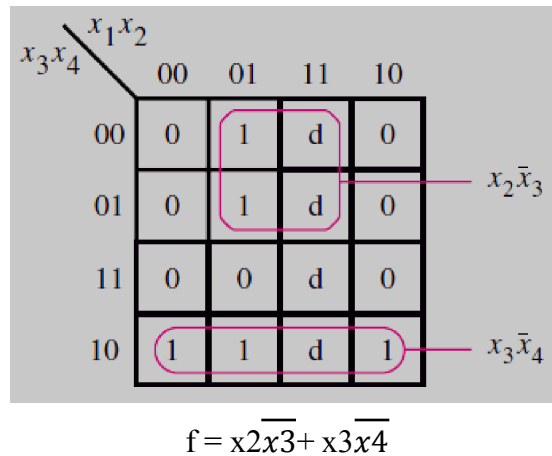
II. Incompletely Specified Functions

- A function that has don't-care condition(s).
- Using the shorthand notation, the function f is specified as
$$f(x_1, \dots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$
where D is the set of don't cares.

Solved Exercise:

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

$$f(x_1, \dots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$



Verilog code:

```

module example4(x2, x3, x4, f);
    input x2, x3, x4;
    output f;
    assign f = (x2 & ~x3) | (x3 & ~x4);
endmodule

```

Lab Exercises

- Simplify the following functions using K-map and implement the circuit using logic gates.
 - $f(A,B,C,D) = \sum m(2,3,4,5,6,7,10,11,12,15)$
 - $f(A,B,C,D) = \sum m(1,3,4,9,10,12) + D(0,2,5,11)$
- Simplify the following functions using K-map and implement the circuit using logic gates.
 - $f(A,B,C,D) = \prod M(0,1,4,6,8,9,12,14)$
 - $f(A,B,C,D) = \prod M(6,9,10,11,12) + D(2,4,7,13)$
- Simulate a circuit that has four inputs, x_1 , x_2 , x_3 , and x_4 , 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.

Additional Exercises:

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

Lab No 3:

Date:

MULTILEVEL SYNTHESIS

Objectives:

In this lab, student will be able to

1. Design multilevel NAND and NOR circuits.
2. Write Verilog code for 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.

Lab Exercises

1. Minimize the following expression using K-map and simulate using only NAND gates.
 $f(A, B, C, D) = \pi M(2,6,8,9,10,11,14)$
2. Minimize the following expressions using K-map and simulate using only NOR gates.
 $f(A, B, C, D) = \sum m(0,1,2,5,8,9,10)$
3. Minimize the following expressions using K-map and simulate using NOR gates only.
 $f(A, B, C, D) = \sum m(1,3,5,7,9) + D(6,12,13)$
4. Minimize the following expression using K-map and simulate using only NAND gates.
 $f(A, B, C, D) = \pi M(4,9,12,13,14) + D(5,6,11,15)$

Additional Exercises

1. Minimize the following expression using K-map and simulate using only NAND gates.

$$f(A, B, C, D) = \prod(1,3,5,8,9,11,15)+D(2,13)$$
2. Find the minimum cost SOP implementation for the following function f using K-map.

$$f = A'C'D' + A'C + AB'C' + ACD'$$

Lab No 4:

Date:

ARITHMETIC CIRCUITS

Objectives:

In this lab, student will be able to

1. Design arithmetic circuits using combinational logic.
2. Simulate arithmetic circuits using Verilog.

I. Adder circuit:

- **Half adder-** a circuit that implements the addition of only two single bit inputs.
- **Full adder-** a circuit that implements the addition of two single bit inputs and one carry bit.
- **Ripple-carry adder**
 - For each bit position, we can use a full-adder circuit, connected as shown in Fig. 4.1.
 - Carries that are produced by the full-adders propagate to the left.

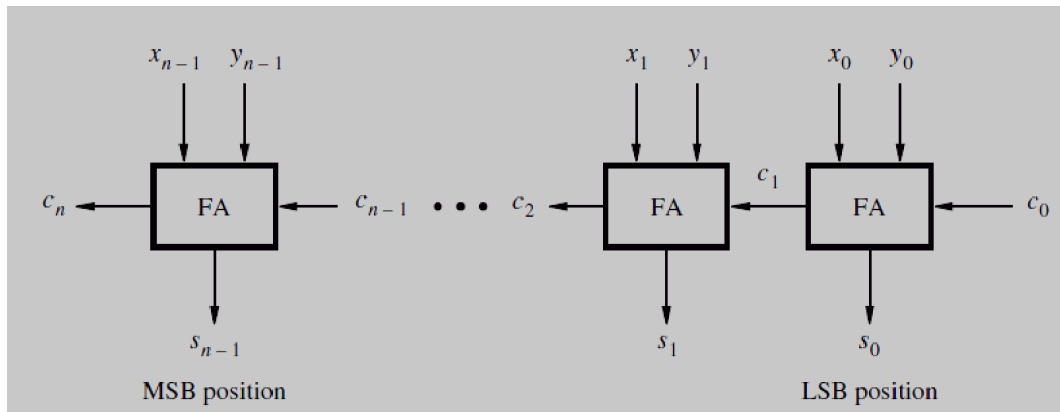


Figure 4.1 An n-bit ripple carry adder

- **Adder/Subtractor unit**

- The only difference between performing addition and subtraction is that for subtraction it is necessary to use the 2's complement of one operand.
- Add/Sub control signal chooses whether addition or subtraction is to be performed.
- Outputs of the XOR gates represent Y if Add/Sub = 0, and they represent the 1's complement of Y if Add/Sub = 1.
- Add/Sub is also connected to the carry-in c_0 . This makes $c_0 = 1$ when subtraction is to be performed, thus adding the 1 that is needed to form the 2's complement of Y .
- When the addition operation is performed, we will have $c_0 = 0$.
- The circuit is shown in Fig. 4.2

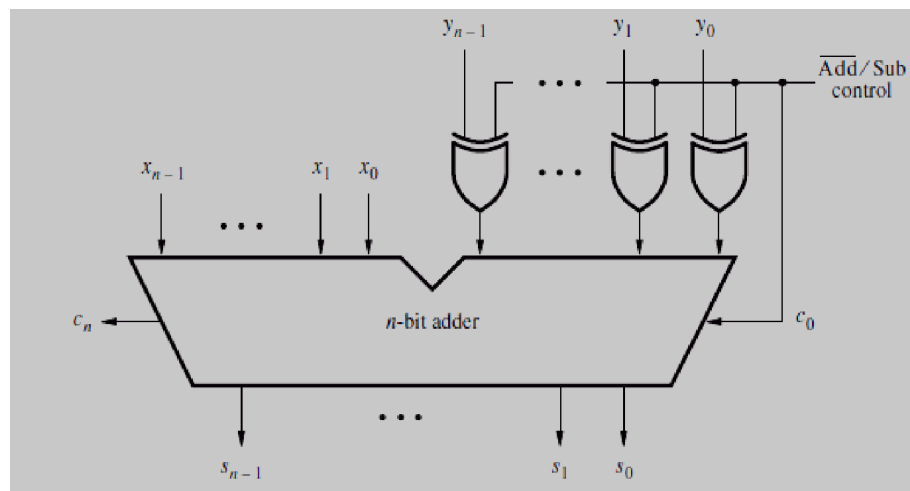


Figure 4.2 Adder/subtractor unit

• Binary multiplier

- Multiplication of binary numbers is performed in the same way as in decimal numbers. The multiplicand is multiplied by each bit of the multiplier starting from the least significant bit.
- Each such multiplication forms a partial product. Successive partial products are shifted one position to the left. The final product is obtained from the sum of the partial products.

• BCD Addition

In Binary Coded Decimal (BCD) representation each digit of a decimal number is represented by 4-bit binary. When 2 BCD numbers are added,

- If $X + Y \leq 9$, then the addition is the same as the addition of 2 four-bit unsigned binary numbers.

- A correct decimal digit can be generated by adding 6 to the result of a four-bit addition whenever the result exceeds 9 or when the carry is generated.

II. Designing sub circuits in Verilog

- A Verilog module can be included as a sub circuit in another module.
- Both modules must be defined in the same file.
- The general form of a module instantiation statement is given below.

```
module_name [#(parameter overrides)] instance_name (
    .port_name ( [expression] ) {, .port_name ( [expression] )} );
```

- The **instance_name** can be any legal Verilog identifier and the port connections specify how the module is connected to the rest of the circuit.
- The same module can be instantiated multiple times in a given design provided that each instance name is unique.
- The #(parameter overrides) can be used to set the values of parameters defined inside the **module_name** module.
- Each **port_name** is the name of a port in the sub circuit, and each expression specifies a connection to that port.
- **Named port connections** -The syntax **.port_name** is provided so that the order of signals listed in the instantiation statement does not have to be the same as the order of the ports given in the **module** statement of the sub circuit.
- **Ordered port connections**-If the port connections are given in the same order as in the sub circuit, then **.port_name** is not needed.

Using Vectored Signals

- Multibit signals are called *vectors*.
- An example of an input vector is

```
input [3:0] W;
```
- This statement defines *W* to be a four-bit vector. Its individual bits can be referred to using an index value in square brackets.
- The most-significant bit (MSB) is referred to as *W*[3] and the least-significant bit (LSB) is *W*[0].

Solved Exercise

Write the Verilog code to implement a 4-bit adder.

Verilog code:

```

module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, carryout;
    fulladd stage0 (carryin, x0, y0, s0, c1);
    fulladd stage1 (c1, x1, y1, s1, c2);
    fulladd stage2 (c2, x2, y2, s2, c3);
    fulladd stage3 (c3, x3, y3, s3, carryout);
endmodule

module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;
    assign s = x ^ y ^ Cin;
    assign Cout = (x & y) | (x & Cin) | (y & Cin);
endmodule

```

TestBench Code:

```

`timescale 1ns/1ns
`include "adder4.v"           //Name of the Verilog file

module adder4_TB();
reg carryin, x3, x2, x1, x0, y3, y2, y1, y0;
wire s3, s2, s1, s0, carryout;

adder4 a1(carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
initial
begin

    $dumpfile("adder4_TB.vcd");
    $dumpvars(0, adder4_TB);

    carryin=1'b0;
    x0=1'b1; x1=1'b0; x2=1'b1; x3=1'b0; //x0 is LSB and x3 is MSB
    y0=1'b1; y1=1'b0; y2=1'b1; y3=1'b0; //y0 is LSB and y3 is MSB
    #20;

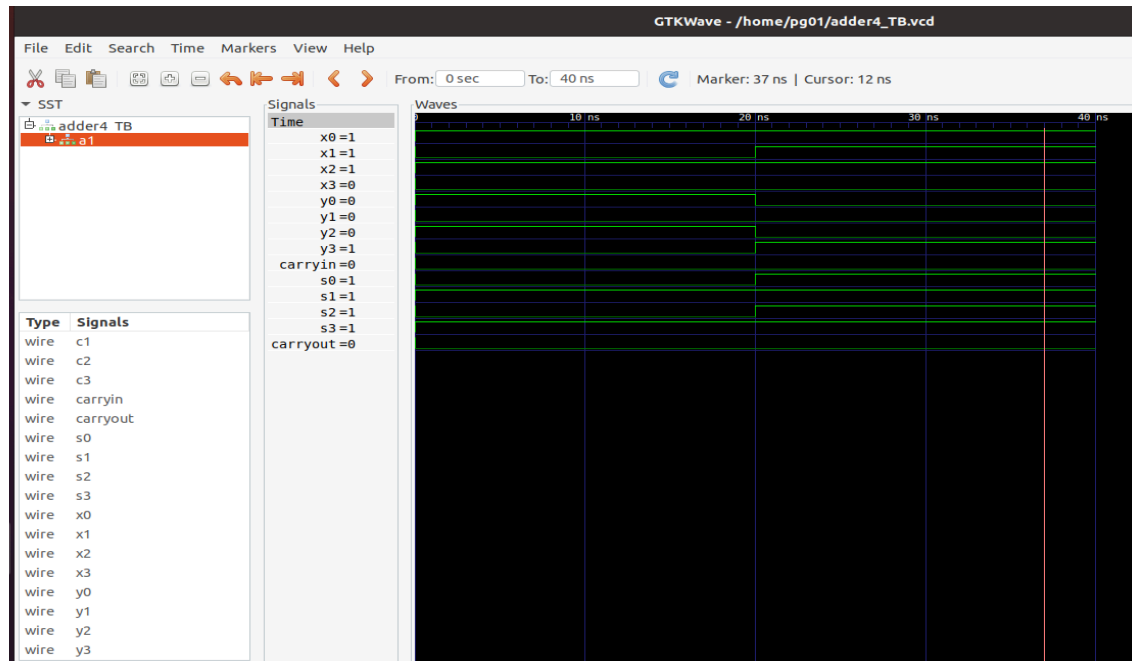
    carryin=1'b0;
    x0=1'b1; x1=1'b1; x2=1'b1; x3=1'b0;
    y0=1'b0; y1=1'b0; y2=1'b0; y3=1'b1;
    #20;

    $display("Test complete");
end

endmodule

```

The output waveform appears as shown below.



- Separate Verilog module for the ripple carry adder instantiate the fulladd module as a sub circuit.

Lab Exercises

Write behavioral Verilog code to implement the following and simulate

1. Half adder and a full adder
2. Four-bit adder
3. Four-bit adder/subtractor using a four-bit adder.
4. 2-bit multiplier using half adders and basic gates.
5. Single-digit BCD adder using four-bit adder(s).

Additional Exercises

1. Design and simulate a circuit that determines how many bits in a six-bit unsigned number are high.
2. Design and write Verilog code for a 2 digit BCD adder.

COMPARATORS AND MULTIPLEXERS

Objectives:

In this lab, student will be able to

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

I. Arithmetic Comparison Circuits

- Compare the relative magnitude of two binary numbers.

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.

II. 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, \dots, w_{N-1} , requires $\log_2 N$ select inputs.
- Fig. 5.1a 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. 5.1b shows the functionality of a 2-to-1 multiplexer.

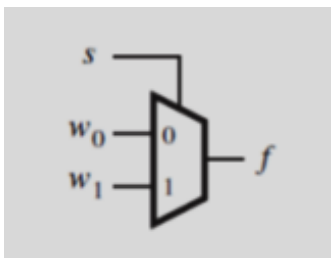


Figure 5.1a Graphical symbol

s	f
0	w_0
1	w_1

Figure 5.1b 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.

Solved Exercise

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

Verilog code:

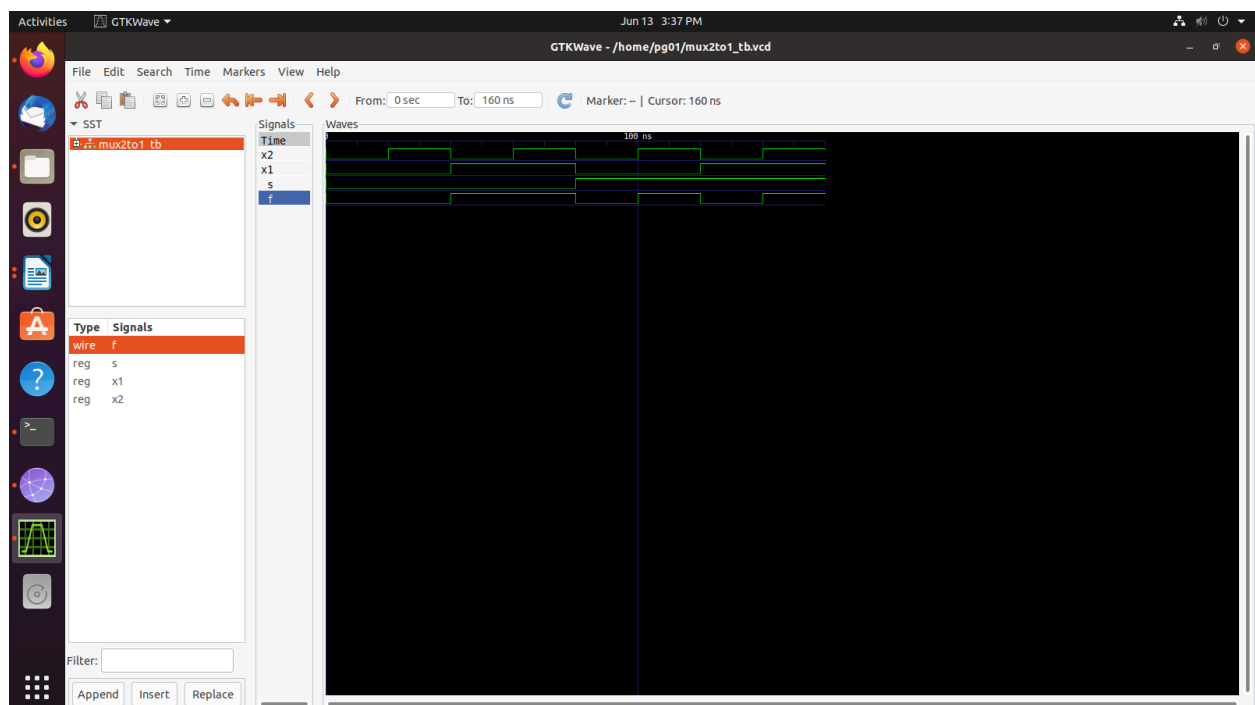
```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output f;
    reg f;
    always @(w0 or w1 or s)
        f = s ? w1 : w0;
endmodule
```

```
`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 and simulate the Verilog code for a 4-bit comparator.
2. Write behavioral Verilog code for a 2 to 1 multiplexer using the **if-else** statement. Use this to write the hierarchical code for a 4 to 1 multiplexer.
3. Write behavioral Verilog code for a 4 to 1 multiplexer using **conditional** operator. Use this to write the hierarchical code for a 16 to 1 multiplexer.
4. 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.

Additional Exercises

1. Write behavioral Verilog code for an 8 to 1 multiplexer using an **if-else** statement. Use this to write the hierarchical code for a 32 to 1 multiplexer.
2. Write behavioral Verilog code to simulate a 4-bit comparator using 2-bit comparators.

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. 6.1, 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.

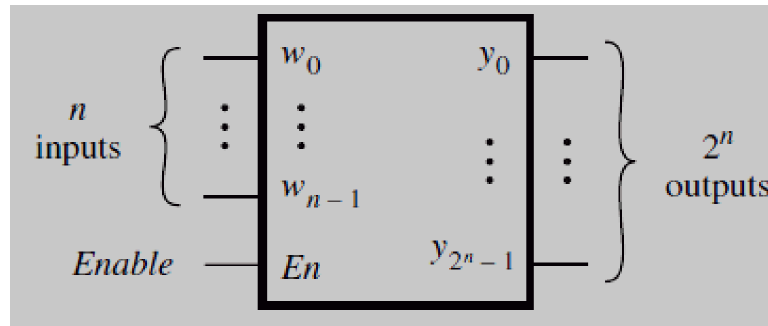


Figure 6.1 Decoder

Binary Encoders

- A binary encoder encodes information from 2^n inputs into an n -bit code, as indicated in Fig. 6.2.
- 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.

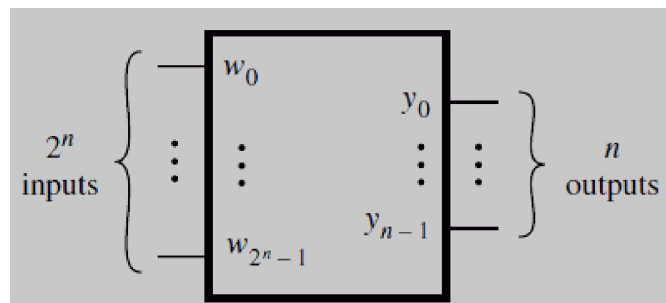


Figure 6.2 Encoder

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. 6.3.

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 6.3 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

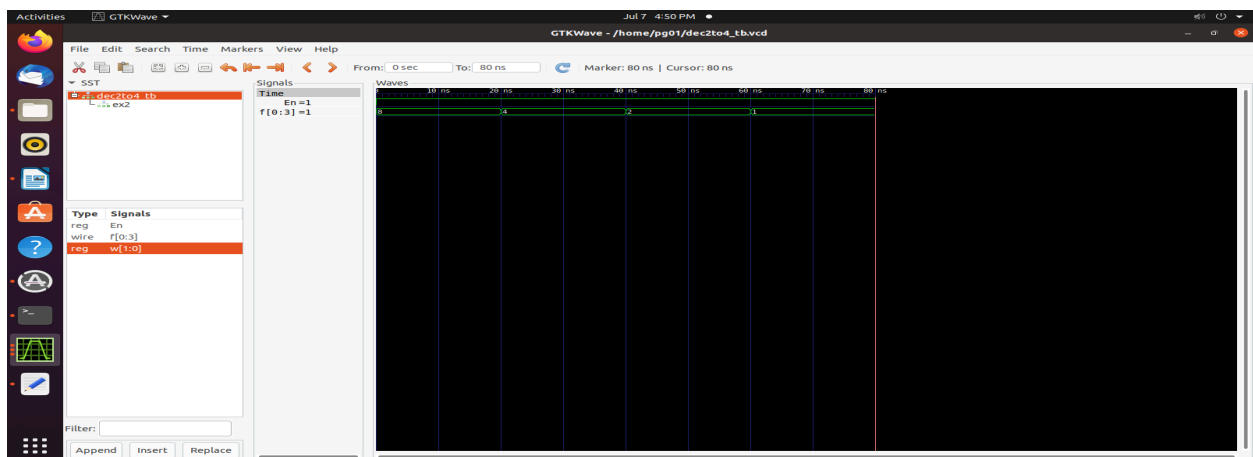
Write behavioral Verilog code for 2 to 4 binary decoder using **for** loop.

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. Using the 2 to 4 decoders above, design a 3 to 8 decoder 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 high output using **for** loop. Using the 3 to 8 decoders above, design a 4 to 16 decoder and write the Verilog code for the same.
3. Write behavioral Verilog code for a 2 to 4 decoder with active-high enable input and active low output using **case** statement. Using the 2 to 4 decoders above, design a 4 to 16 decoder with active-high enable input and active low output and write the Verilog code for the same.
4. Write behavioral Verilog code for a 4 to 2 priority encoder using **casex** statement.
5. 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 7:

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

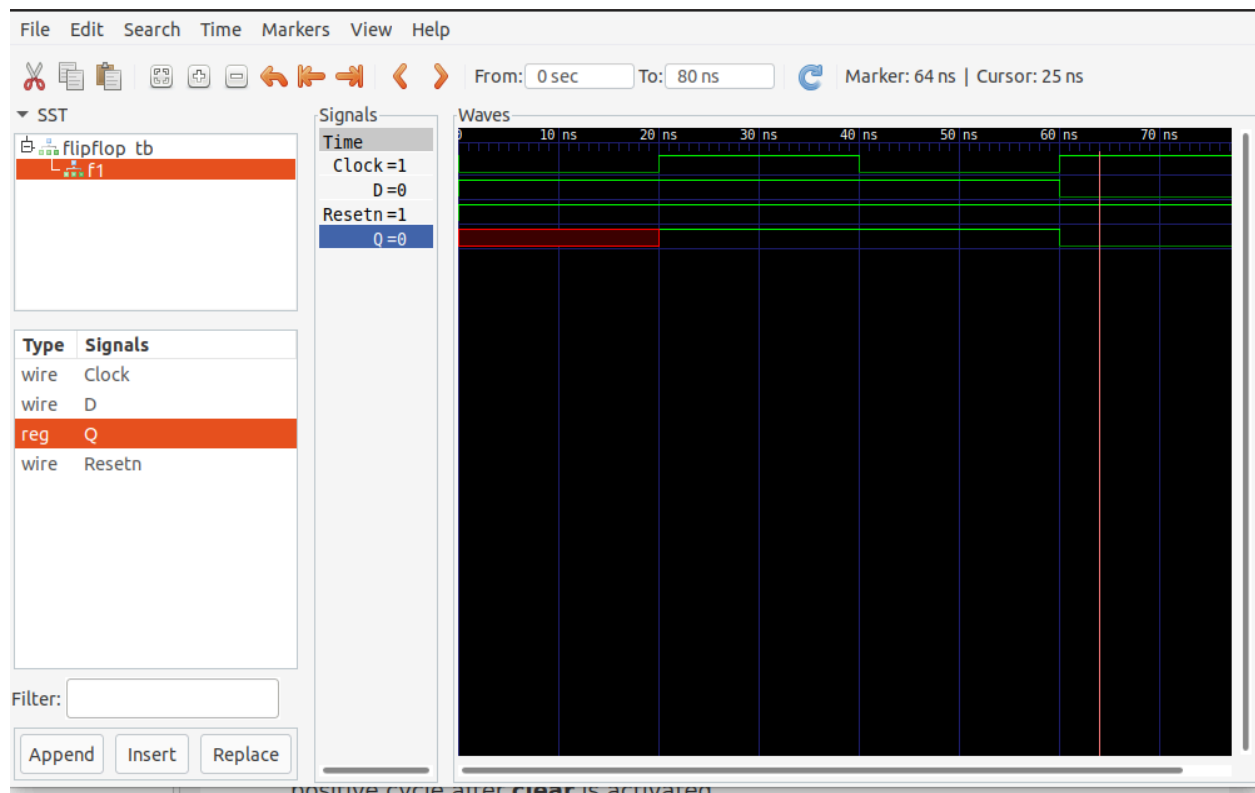
    Sdumpfile("flipflop_tb1.vcd");
    Sdumpvars(0, flipflop_tb1);

    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;
    Sdisplay("Test complete");
end
endmodule

a1(carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
initial
begin
```

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.

REGISTERS

Objectives:

In this lab. student will be able to

1. Understand the concept of registers and shift registers.
2. Write Verilog code for 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.

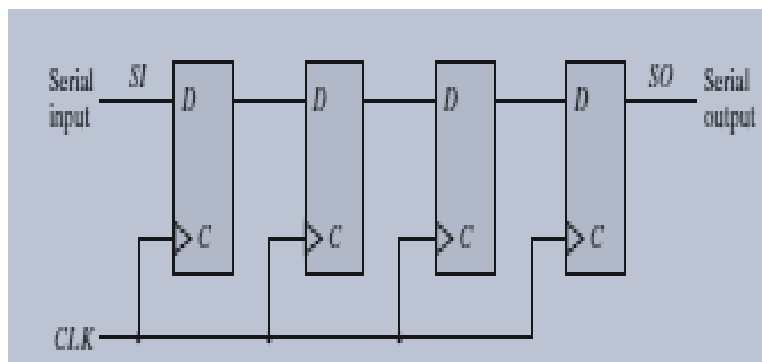


Figure 8.1 Four-bit Shift Register

- 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.

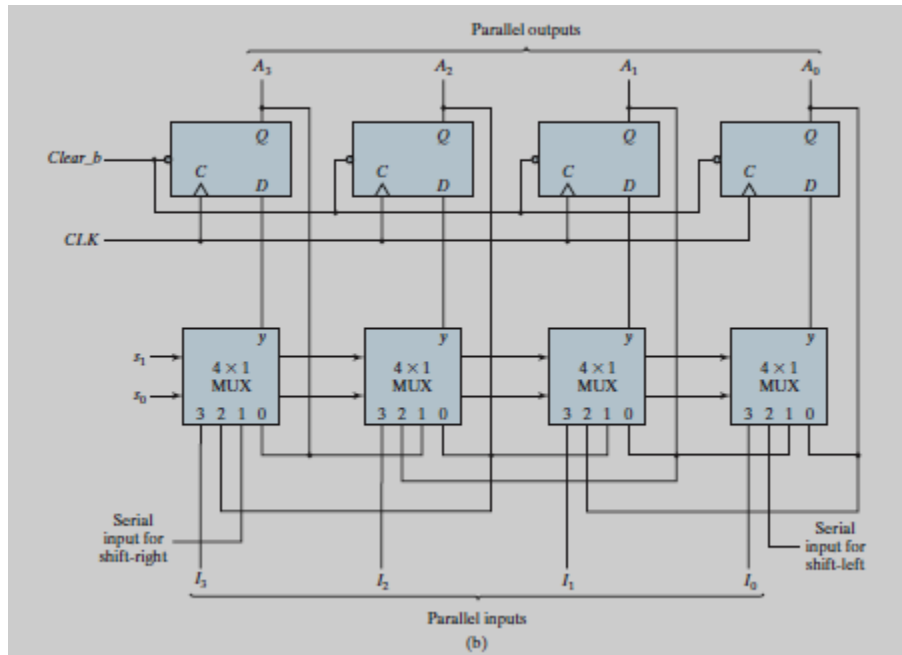


Figure 8.2 Four-bit Shift Register with Parallel Load

- As shown in Fig. 8.2, 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

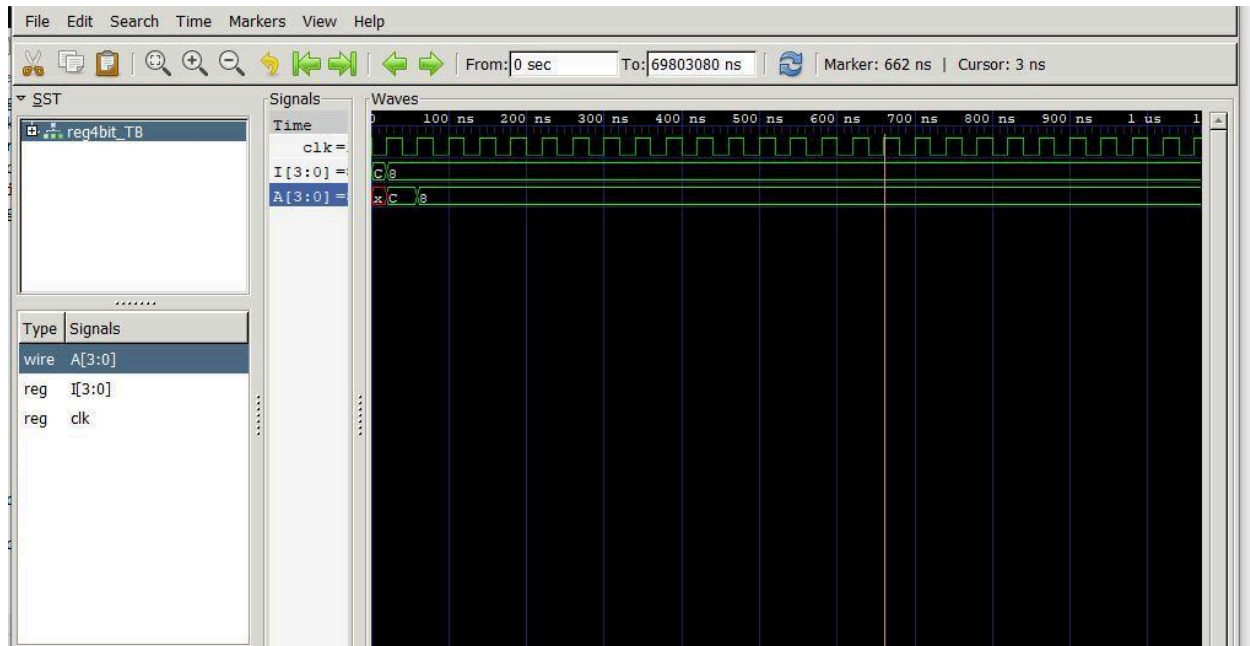
```

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
    Sdumpfile("reg4bit_TB.vcd");
    Sdumpvars(0,reg4bit_TB);
    I = 12; #20;
    I = 8;  #20;
    Sdisplay("Test complete");
end
endmodule
```

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: 9

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. 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 9.1.

Table 9.1 State Table for a Sequential Circuit.

Present State		Input	Next State		Output
A	B		A	B	
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

- 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 9.1.

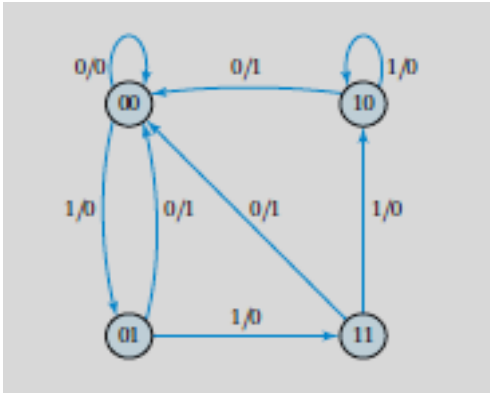


Figure 9.1 State Diagram of the Sequential Circuit.

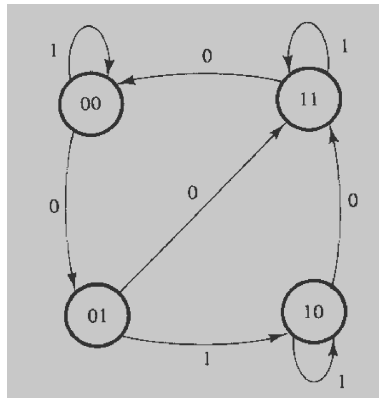
Design Procedure:

- 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.

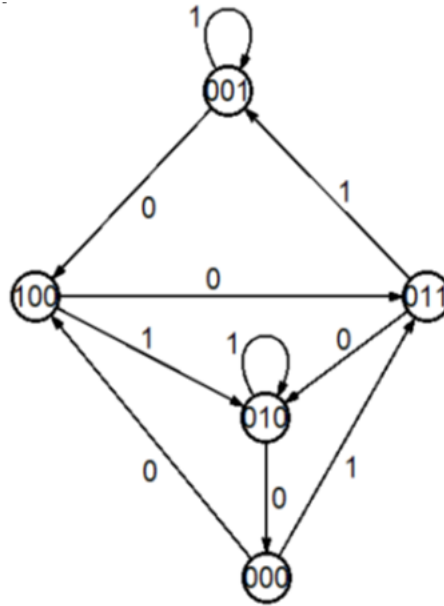
Lab Exercises

1. 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 T FF. Write the Verilog code for the derived circuit.

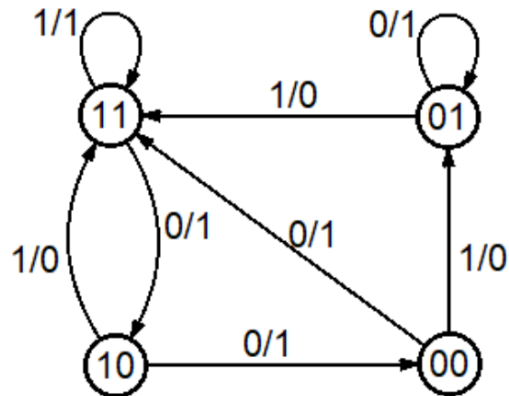


Additional Exercises

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



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



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. 10.1

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:
 1. From the given information (word description/state diagram/timing diagram/other pertinent information) about the circuit, obtain the state table.
 2. Determine the number of flip flops needed.
 3. From the state table, derive the circuit excitation and output tables.
 4. Derive the circuit output functions and the flip flop input functions by simplification.
 5. 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. 10.2 shows a 4-bit ring counter using decoder and counter.

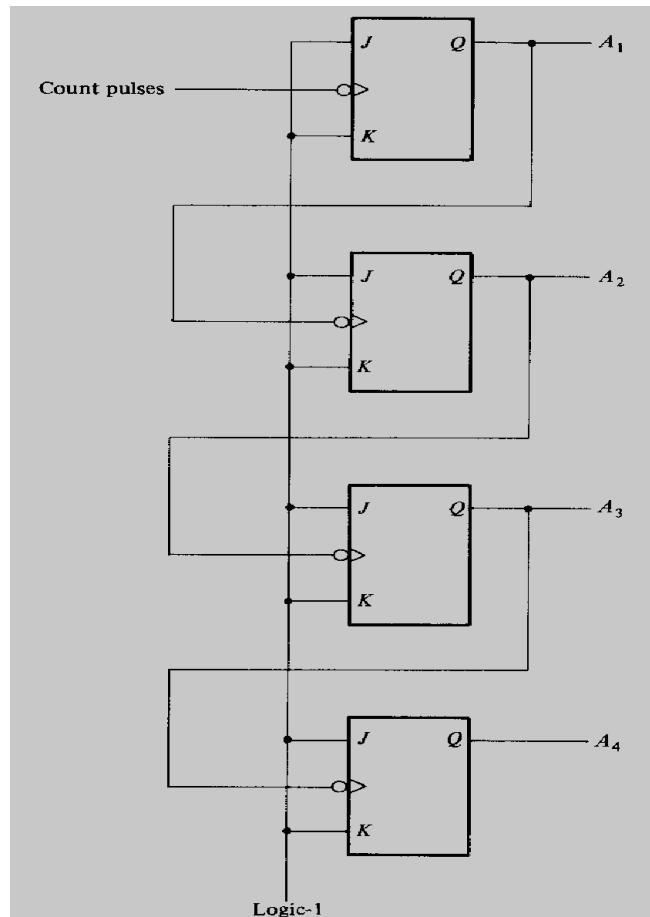


Figure 10.1: 4-bit binary ripple counter

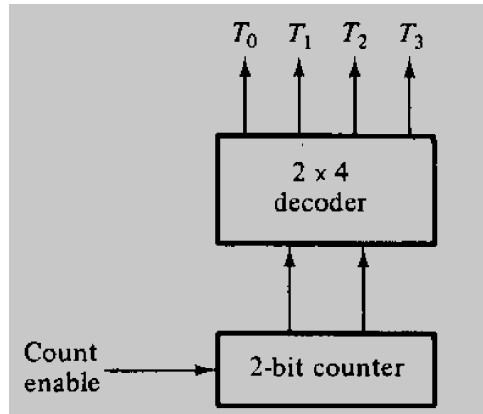


Figure 10.2: 4-bit ring counter

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. 10.3 shows a 4-bit Johnson counter

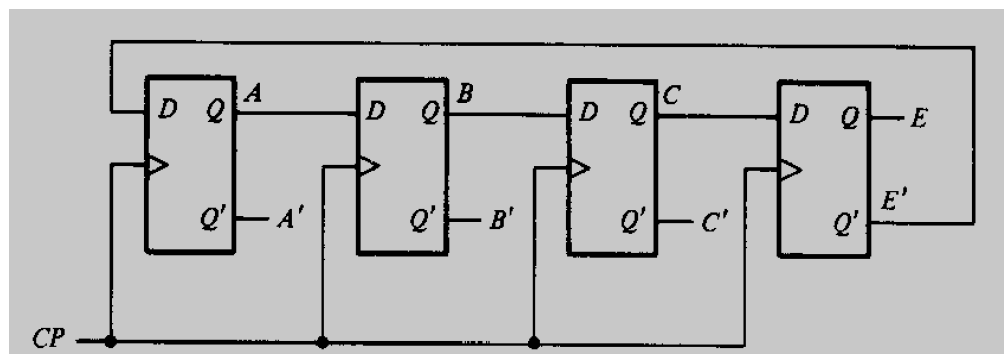


Figure 10.3 4-bit Johnson counter

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

```

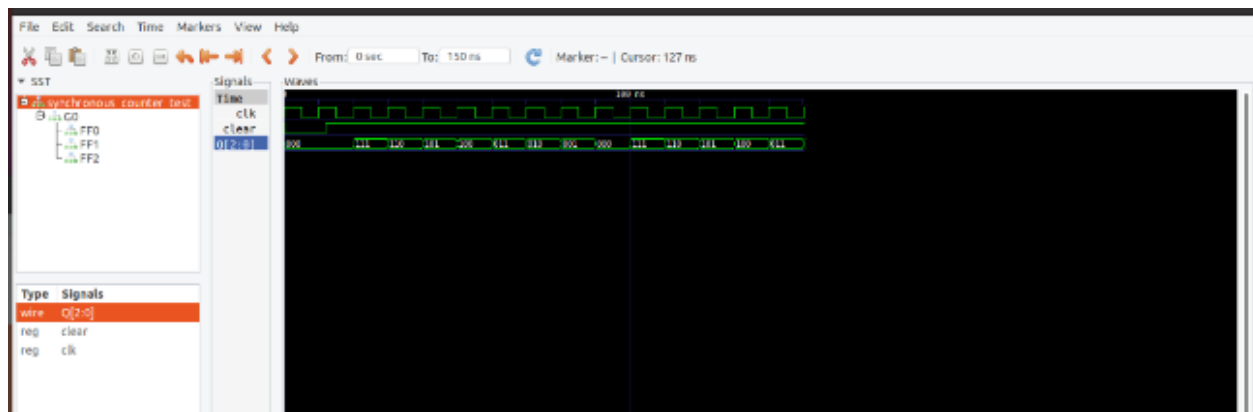
`timescale 1ns/1ns
`include "synchronous_counter.v" //Name of the Verilog file

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);  
$monitor($stime, " 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. Implement two-digit BCD counter by including two modulo-10 counters. Validate the design by writing behavioral Verilog code.
3. Assume a 4-bit signal A and a clock as inputs and a 4-bit signal Y as output. Design a circuit which repeats the operations cyclically with a clock as follows:

Clock(input)	Y(output)
1 st clock cycle	rotate A by 1 bit to the left
2 nd clock cycle	rotate A by 2 bit to the left
3 rd clock cycle	rotate A by 3 bit to the left
4 th clock cycle	complement of A

And then repeats in the same pattern.

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

