

Post Lab Report-03

Date: 6 April 2021	Course Code: CSE345
Experiment 3	Course Title: Digital Logic Design
Name: Adri Saha	Id: 2019-1-60-024
Course instructor: Touhid Ahmed	Section: 4

Experiment Name: Behavioral Verilog Simulation of a Combinational Logic Circuit

Abstract: Digital Logic Design is the one of the introductory courses for Electrical and Electronics Engineering students. It can introduce students to circuit design, problem solving, testing, and feature verification. In this experiment, students were asked to design some combinational logic circuits by using **Behavioral Verilog Simulation** from Boolean expressions. And they can also learn behavioral Verilog coding of a combinational logic circuits using procedural model and continuous assign statement. Here, they also learn how to reduce a Verilog code by simplification of these statements. They will also be able to design circuits both abstractly and hierarchically using behavioral and structural modeling. Hardware description language (HDL) is used to explain behavioral modeling. And that is how they can design a complete logic circuits and can get to know about the uses of these. The digital logic design lab is a learning experience that most students enjoy because it gives them their first hands-on experience designing and constructing miniature structures.

Introduction:

Most of our examples use if-else statements, case statements, for loops, and other procedural statements to create behavioral code. It's possible to write behavioral code in a way that looks like a computer program, with a complicated control flow and several loops as well as branches. It can be difficult to connect such code, also known as high-level behavioral code, to the final hardware implementation; it can also be difficult to predict what circuit a high-level synthesis tool would produce. Behavioral models in Verilog contain procedural statements, which control the simulation and manipulate variables of the data types. There, the

main difference between behavioral and structural model in Verilog is that behavioral model describes the system in an algorithmic manner, while structural model describes the system using basic components such as logic gates

Combinational Logic Circuits: Digital logic circuits may be combinational or sequential. A combinational circuit consists of input variables, logic gates, and output variables. Combinational designs are those in which the system's output is solely determined by the inputs' current values. Since the outputs are directly dependent on the current inputs, these designs do not need any memory.

The logic gates accept signals from the input and generate signals to the outputs. The principle of operation is that the circuit operates on just two voltage levels, called logic 0(low) and logic 1(high).

For n input variables, there are 2^n possible combinations of binary input values.

Uses:

- Use in computer circuits to perform Boolean algebra on input signals and on stored data.
- Practical computer circuits normally contain a mixture of combinational and sequential logic.

Behavioral Verilog Simulation:

In Verilog, behavioral models have procedural statements that control the simulation and manipulate data type variables. Many of these statements can be found in the procedures. There is an operation flow associated with and process. At simulation time 'zero,' all the flows identified by the 'always' and 'initial' statements start together during behavioral model simulation.

Uses:

- Contain procedural statements for controlling the simulation and manipulating data type variables.
- It specifies how the circuit should work. As a result, behavioral modeling is regarded as the highest level of abstraction as compared to dataflow or structural models.
- It represents digital circuits at a functional and algorithmic level.

- Behavioral simulation allows for the verification of syntax and functionality without the use of timing data.
- It is used for the most of verification during design development.

Operators: There are a lot of operators in Verilog. The product of logical operators is a one-bit value. The consequence is x if an operand is ambiguous (contains an ambiguous x). The symbols for reduction operators are the same as for other bitwise operators, but they only have one operand. The reduction &A yields the AND of all the bits in A, whereas the reduction &A perennial yields the NAND. In most cases, logical operators are used.

The bitwise operators are 1's complement (\sim), unary plus (+), 2's complement ($-$), AND (&), OR (|), XOR (\wedge), and XNOR ($\sim\wedge$ or $\wedge\sim$). The logical operators generate a one-bit result. They are NOT (!), AND (&&), and OR (||).

Continuous Assignments:

Continuous assignment is the name given to the assign sentence. Any changes in the RHS (right-hand side) expression are continuously assigned/updated to the LHS (left-hand side) net by continuous assignment. A scalar or vector wire net must be used for LHS. Just use the continuous assignment statement outside of process statements like initial or always. As an example, assign a = b & c; This is known as a continuous assign since the wire on the left side of the assignment operator is continuously driven with the value of the expression on the right side.

Uses:

- It is used to assign values to scalar and vector nets and occurs whenever the RHS changes.
- It makes it easier to drive the net with logical expressions by allowing one to model combinational logic without specifying an interconnection of gates.
- It is very reduced & simple easy code.

```
// Continuous model specification Verilog code
module expt3_2 (input A, B, output S);
    assign S = (~A & B) | (A & ~B);
endmodule
```

Procedural Assignments: The purpose of procedural assignments is to update variables such as reg, integer, time, and memory. Continuous assignments drive net

variables and are evaluated and modified whenever an input operand changes value, as mentioned below. Procedural assignments drive net variables and are evaluated and updated whenever an input operand changes value.

Under the supervision of the process, procedural assignments update the value of register variables.

Uses:

- Used for updating register data types and memory data types.
- The if-else statement is an example of a Verilog *procedural* statement which is very useful for Verilog simulation. There are other procedural statements, such as loop statements.

```
// Procedural model specification Verilog code
module expt3_1 (input A, B, output reg S);
always @(A, B) begin
S=0;
if(~A & B) S=1;
if(A & ~B) S=1;
end
endmodule
```

In loop it is important to define register before write output.

Always Block: Procedural statements must be enclosed in a construct known as an always block in Verilog syntax. A single statement or multiple statements may be included in an always block. The sensitivity is the part of the always block in parentheses after the @ symbol. The simulator only executes the statements within an always block when one or more of the signals in the sens are active.

If-else Statement: Between one 'if' and one 'else' comment, there can be several 'else if' statements in an if-else block. It also can be attached with or operator. Here, in line 3-6 'begin - end' is inserted, which is used to identify multiple statements within a 'if', 'else if', or 'else' block.

// if else statement Verilog code using or operator

```
module Experiment3_4 (input A,B,C, output reg S);
```

```

always @(A,B,C)
begin S=0;
if((~A&~B&~C)|(~A&B&C)|(A&~B&C)|(A&B&~C)) begin S=1;
end
end
endmodule

```

Case Statement: Lines 3-9 represent the case statement. The value of the output y is determined by the value of 's,' for example, if 's' is '1,' then line 4-8 is real. The 'default' keyword can be used to provide the output for undefined-cases if we don't specify all of the possible cases in the 'case-statement.'

```

// Verilog code using case statement
module Experiment3_3(input A,B,C, output reg S);
always @*
begin
    case ({A,B,C})
        3'b001: S=1;
        3'b010: S=1;
        3'b100: S=1;
        3'b111: S=1;
        default: S=0;
    endcase
end
endmodule

```

Moreover, instead of listing the relevant signals in the sensitivity list, it is possible to write simply always @* if the compiler will figure out which signals need to be considered.

Objectives: Our main purpose is to design logic circuits using Behavioral Verilog Simulation. Students can utilize behavioral code by assign statements, if-else statements, case statements, for loops and other procedural statements. They can also reduce their Verilog codes using them.

- Using a procedural model, learn behavioral Verilog coding of a combinational logic circuit.
- To learn how to code a combinational logic circuit in behavioral Verilog using the continuous assign statement.
- Learn to design behavioral Verilog simulation from a Boolean expression.

Theory and experiments results:

A. 2 Input XOR Gate: The logic function implemented by a 2-input Ex-OR is given as either: “A OR B but NOT both” will give an output at Q. In general, an Ex-OR gate will give an output value of logic “1” ONLY when there are an ODD number of 1’s on the inputs to the gate, if the two numbers are equal, the output is “0”.

Uses:

- It is used in simple digital addition circuits which calculate the sum and carry of two (half-adder) or three (full adder) bit numbers.
- XOR gates are also used to determine the parity of a binary number, i.e., if the total number of 1's in the number is odd or even.
- The **XOR** gate is achieved by combining standard logic gates together to form more complex gate functions that are used extensively in building arithmetic logic circuits, computational logic comparators and error detection circuits.
- The two-input “Exclusive-OR” gate is basically a modulo two adder, since it gives the sum of two binary numbers and as a result are more complex in design than other basic types of logic gate.

Boolean expression is:

$$S = (A \oplus B) = A.B' + A'.B$$

The truth table of 2 input XOR gate is:

Here, A & B is input & output S.

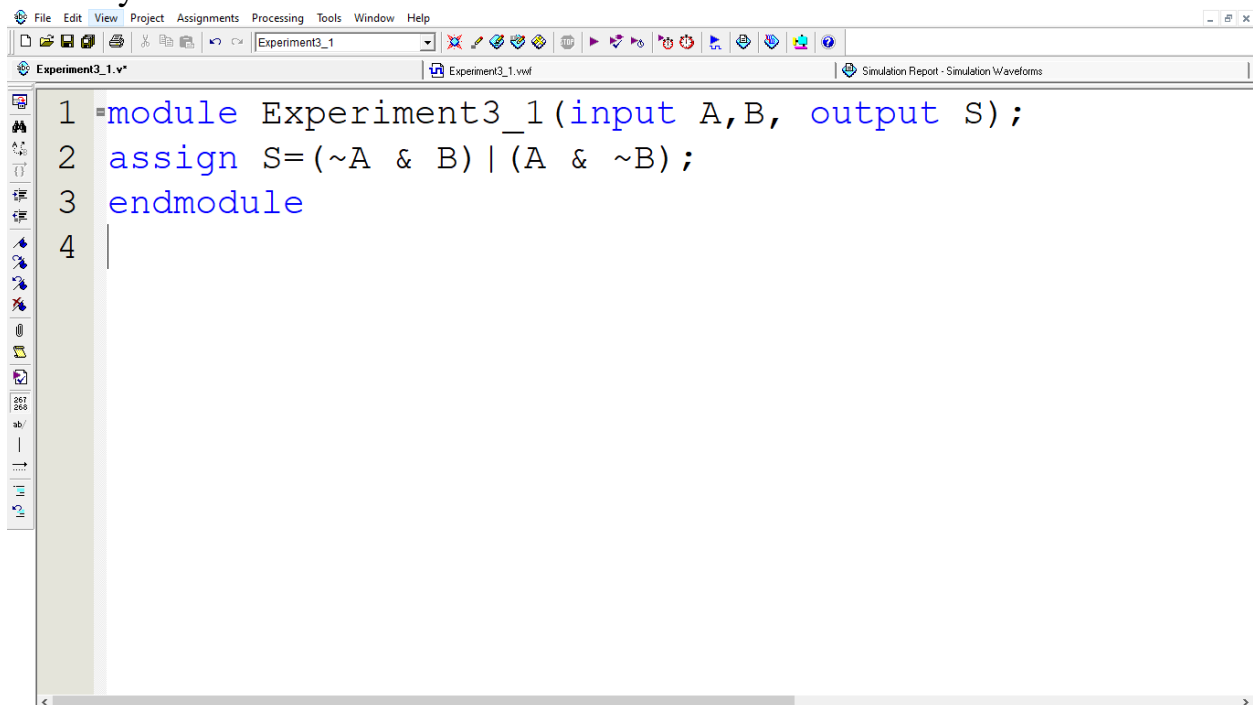
Input		Output
A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1: Truth table of 2 input XOR Gate

Verilog Codes: Verilog is a Hardware Description Language; a textual format for describing electronic circuits and systems.

Uses:

- Applied to electronic design.
- Verilog is intended to be used for verification through simulation, for timing analysis, for test analysis (testability analysis and fault grading) and for logic synthesis.

The image shows a screenshot of a Verilog code editor window. The window has a menu bar with 'File', 'Edit', 'View', 'Project', 'Assignments', 'Processing', 'Tools', 'Window', and 'Help'. Below the menu bar is a toolbar with various icons for file operations, editing, and simulation. The main text area contains the following Verilog code:

```
1 module Experiment3_1(input A,B, output S);  
2   assign S=(~A & B) | (A & ~B);  
3 endmodule  
4
```

The code is written in a blue monospaced font. The line numbers 1, 2, 3, and 4 are on the left side of the code area. The editor also shows a status bar at the bottom with the text 'Experiment3_1.v' and 'Experiment3_1.vwf'.

Figure 2: Verilog code using continuous assignment in 2 input X-OR gate

Waveform Simulation:

In this simulation, we have taken end time 20 nano seconds for 2 input A, B. So, time period of A & B is 20 & 10 nano seconds in sequent. As there are 2 inputs. We see, output is “high” or 1 after 5 ns and again low after 15ns. So, it is verified with the truth table of 2 input XOR gate.

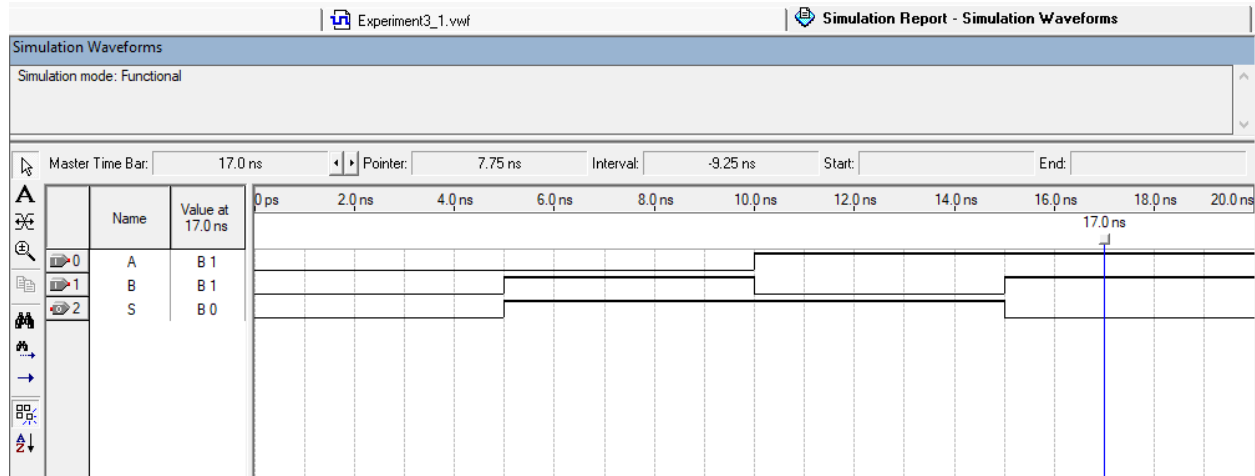


Figure 3: Simulation of Verilog code using continuous assignment for 2 input X-OR gate

B. 2 Input XNOR Gate

The **Exclusive-NOR Gate**, also written as: “Ex-NOR” or “XNOR”, function is achieved by combining standard gates together to form more complex gate functions.

Uses:

- The XNOR logic gates are used in error detecting circuits which are to detect
- Odd parity or even parity bits in digital data transmission circuits.
- XNOR gate is mainly used in arithmetic and encryption circuits.

Here, A & B is input & output S.

Boolean expression of 2 input exclusive nor gate:

$$Q = (A \oplus B) = A'.B' + A.B$$

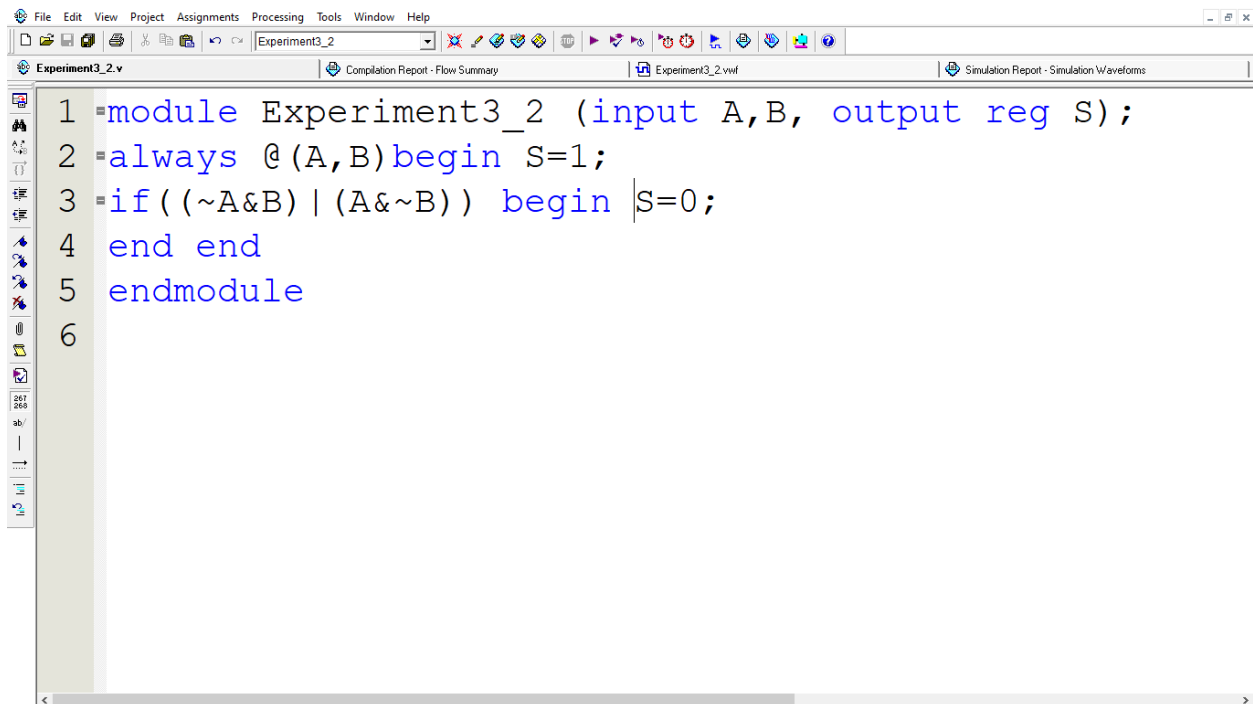
The truth table of 2 input XNOR gate is:

Here, A & B is input & output S.

Input		Output
A	B	S
0	0	1
0	1	0
1	0	0
1	1	1

Figure 4: Truth Table of 2 input XNOR Gate

Verilog Codes:



```
1 module Experiment3_2 (input A,B, output reg S);
2   always @(A,B) begin S=1;
3   if ((~A&B) | (A&~B)) begin S=0;
4   end end
5 endmodule
6
```

Figure 5: Verilog code using procedural statement in 2 input X-NOR gate

Verilog Simulation: In this simulation, we have taken end time 40 nano seconds. So, time period of A & B is 40 & 20 nano seconds in sequent. As there are 2 inputs.

We see, output is “low” or 0 after 10 ns and again high after 30 ns. So, it is verified with the truth table of 2 input XNOR gate.

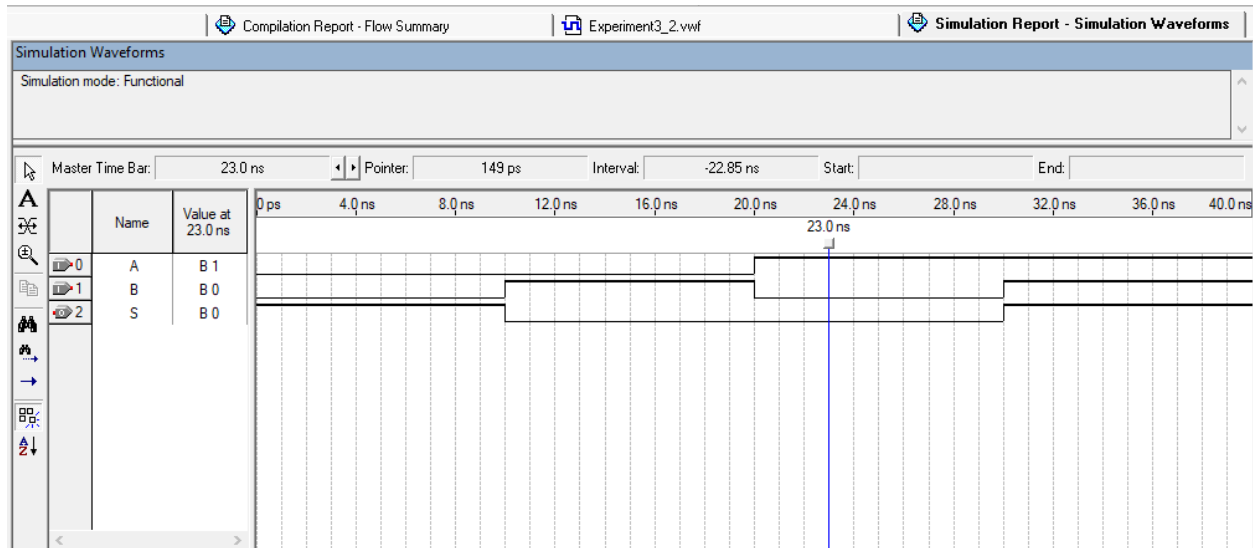


Figure 6: Simulation of Verilog code using procedural assignment for 2 input X-NOR gate

C. 3 Input XOR Gate

Here, A, B, C is input & output S.

Boolean expression: $S = A'B'C + A'BC' + AB'C' + ABC$

The truth table of 3 input XOR gate is:

Here, A, B, C is input & output S.

Input			Output
A	B	C	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 7: Truth table of 3 input XOR Gate

Verilog Codes:

There are total $2^3=8$ output for 3 input. So, there are 8 cases, for 4 cases output is high & for rest 4 cases output is low. So, we write 4 cases condition in binary for high voltage output. And default the rest output is low.

```

1 module Experiment3_3(input A,B,C, output reg S);
2   always @*
3   begin
4     case ({A,B,C})
5       3'b001: S=1;
6       3'b010: S=1;
7       3'b100: S=1;
8       3'b111: S=1;
9       default: S=0;
10    endcase
11  end
12 endmodule
13

```

Figure 8: Verilog code using case statement in 3 input X-OR gate

Simulation:

In this simulation, we have taken end time 80 nano seconds. So, time period of A, B & C is 80, 40 & 20 nano seconds in sequent. As there are 3 inputs. All of the output are matched with truth table of X-OR gate.

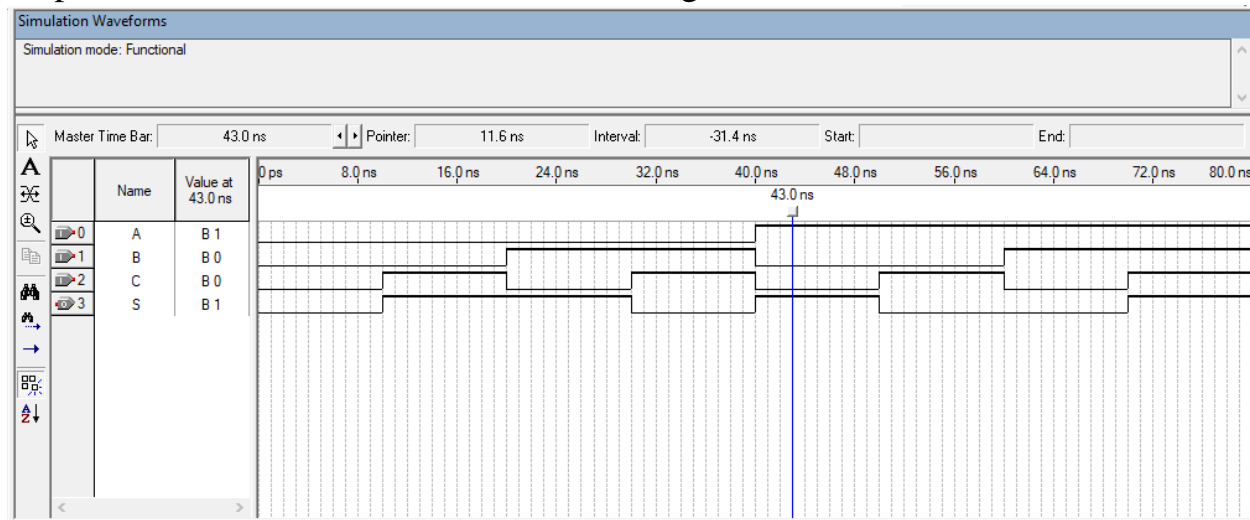


Figure 9: Simulation of Verilog code using case assignment for 3 input X-OR gate.

D. 3 Input XNOR Gate:

Here, A, B & C is input & output S.

$$\text{Boolean expression of: } S = AB'C + ABC' + A'BC + A'B'C'$$

The truth table of 3 input XNOR gate is:

Here, A, B, C is input & output S.

Input			Output
A	B	C	S
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Figure 10: Truth table of 3 input XNOR Gate

Verilog Code:

There is total $2^3=8$ output for 3 input. So, there are 8 cases, for 4 cases output is high & for rest 4 cases output is low. So, we combine 4 if condition with AND operator for high voltage output. And declare “begin 0” first, so rest of the output is 0. As we create loops here, declared output in register.

```

1 module Experiment3_4 (input A,B,C, output reg S);
2   always @ (A,B,C) begin S=0;
3   if ( (~A&~B&~C) | (~A&B&C) | (A&~B&C) | (A&B&~C) ) begin S=1;
4   end end
5 endmodule

```

Figure 11: Verilog code using if statement in 3 input X-NOR gate

Verilog Simulation:

In this simulation, we have taken end time 80 nano second. So, time period of A, B & C is 80, 40 & 20 nano seconds in sequent. As there are 3 inputs. All of the output are matched with truth table of X-NOR gate.

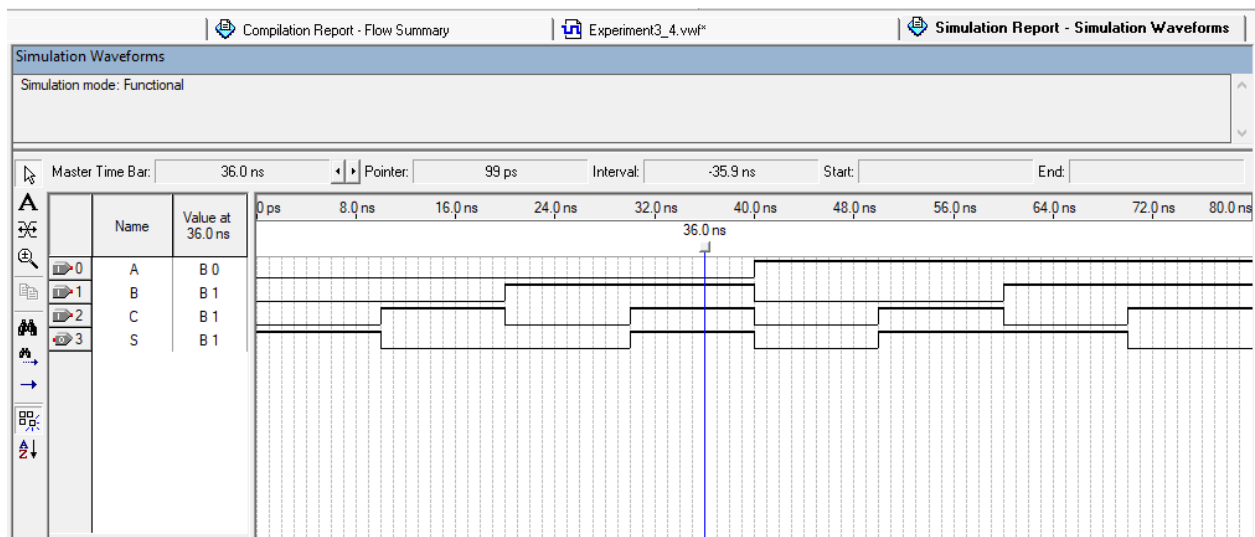


Figure 12: Simulation of Verilog code using if assignment for 3 input X-NOR gate

Conclusion: From this experiment, students learned to simulate combinational logic circuit using behavioral Verilog simulation. They also learn to design a combinational circuit from descriptive problem specification such as a Boolean expression. They also know about uses of behavioral Verilog simulation such as procedural model, continuous assign statement etc. They also can reduce the Verilog codes using loops. And also simulate the design and verify his/her hands-on experience designing by Quartus Software.

Reference:

1. Book: Fundamentals of Digital Logic with Verilog Design by Stephen Brown, Zvonko Vranesic-McGraw-Hill Science_Engineering_Math (2013)
2. https://www.tutorialspoint.com/vlsi_design/behavioural_modelling_timing_control_in_verilog.htm#:~:text=Advertisements,activity%20flow%20associated%20with%20it.
3. <https://verilogguide.readthedocs.io/en/latest/verilog/procedure.html>