Department of Computer and Software Engineering – ITU
SE201L: Digital Logic Design Lab

Course Instructor: Ms Aqsa Khalid	**Dated: 11 – 09 – 2023**
Lab Engineer: Muhammad Kashif	**Semester: Fall 2024**
Batch: BSSE 23 A	

LAB 4 Introduction to Verilog HDL, and NAND/NOR Implementation of Logic Circuits Using Verilog Structural Modeling

Name	Roll Number	Lab Marks

Checked on: _____

Signature: _____

ntroduction to Verilog HDL, and NAND/NOR Implementation of Logic Circuits Using Verilog Structural Modeling

4.1. Introduction

This lab exercise aims to familiarize students with a Hardware Description Language, namely, Verilog. Students are introduced to one of the three styles of modeling in Verilog: Verilog structural modeling. The other two modeling styles, i.e. the dataflow and the behavioral, will be covered in later labs. Students are also introduced to the NAND and NOR equivalents of logic functions, which they implement in Verilog using structural modeling.

4.2. Objectives

This lab will enable students to achieve the following:

• Understand structural modeling style of Verilog, and familiarize with ModelSim and its features • Develop Verilog modules to implement Boolean functions and Verilog test benches to test their operation, and display the simulation results in timing diagrams

- Perform conversions of a Boolean expression, comprising AND, OR and NOT operations, to NAND-only and NOR only expressions
- Verify the logical equivalence of the NAND and NOR implementations of a logic function to its standard form using Verilog

4.3. Conduct of Lab

1. This lab experiment has to be performed using the ModelSim PE Student Edition, installed in Embedded Lab PCs.
2. Bring printout of this lab manual when you come to perform the lab.
3. You can work and get evaluated in groups of two. However, manual submission has to be separate.
4. If there is difficulty in understanding any aspect of the lab, please seek help from the lab engineer or the TA.
5. If a lab task contains an instruction to show working to lab engineer, make sure that the lab engineer evaluates and marks on your manual for that task. If your manual is unmarked for this task, it can result in marks deduction.
6. Complete the lab within the allocated time. Late submissions will be marked zero.
7. Print the codes and screenshots of simulation results of all tasks and attach these to your lab manual. Submit complete manual to the lab engineer no later than 24 hours after the lab.

4.4. Theory and Background

4.4.1. Introduction to Verilog HDL

Hardware Description Languages (HDLs) are textual computer programming languages that are used to describe the structure and behavior of electronic circuits, specifically digital logic circuits. HDLs are used in design, manufacture, testing and simulation of digital logic systems. Verilog, SystemVerilog and VHDL are the most popular currently used HDLs.

The Verilog HDL is an IEEE standard - number 1364. The first version of the IEEE standard for Verilog was published in 1995. A revised version was published in 2001; this is the version used by most Verilog users. The IEEE Verilog standard document is known as the Language Reference Manual, or LRM. This is the complete authoritative definition of the Verilog HDL.

Verilog HDL programming has three modeling styles: structural, dataflow and behavioral. In this lab, you will learn to use structural modeling in Verilog. The other two styles are the topics of later labs.

4.4.1.1. Structural modeling in Verilog

Structural modeling style is at the lowest level of abstraction among the different modeling styles of Verilog. It is basically a gate-level textual description of the circuit schematic. Let's look at the following example to understand the syntax and the different components of a Verilog module based on structural modeling style:

```
module FirstVerilogModule (A, B, X);
    input A, B;
    output X;
    wire NOT_A, NOT_B, A_AND_NOT_B, NOT_A_AND_B;
    not n1 (NOT_A, A);
    not n2 (NOT_B, B);
    and a1 (A_AND_NOT_B, A, NOT_B);
    and a2 (NOT_A_AND_B, NOT_A, B);
    or o1 (X, A_AND_NOT_B, NOT_A_AND_B);
endmodule
```

This code implements the logic circuit of the following figure:

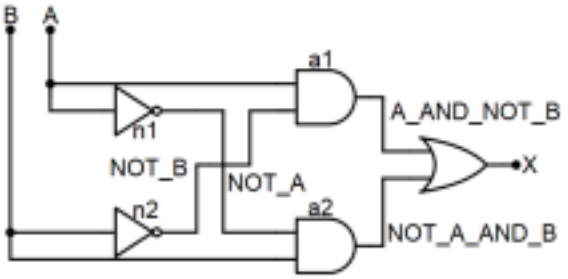


Figure 4.1: Schematic of logic circuit implemented by FirstVerilogModule

Let us now look at each line of the FirstVerilogModule code and understand what the syntax means:

1. *module FirstVerilogModule (A, B, X);* This line declares the module name, FirstVerilogModule, and the names of its associated ports. At this point, it is not declared whether these ports are going to be inputs, outputs or inouts. Notice that there is a semi-colon at the end of this code line. It is part of Verilog syntax to end each line of code with a semi-colon except in a few cases, which will be discussed as they are encountered.
2. *input A, B;* This line declares that the ports A and B of FirstVerilogModule are inputs.
3. *output X;* Port X is declared an output.
4. *wire NOT_A, NOT_B, A_AND_NOT_B, NOT_A_AND_B;* This line creates wires with the mentioned names. In Verilog, there are two broad data types: *nets* and *registers*. *wire* is a *net* data type that symbolizes the data transmission elements in a hardware circuit and it cannot in itself store any data, but is used to connect different elements together. In our example, NOT_A will be used as the connection of the output of NOT gate n1 to the input of AND gate a2. Similarly, other wires also act as connections between different components as evident from Figure 4.1.
5. *not n1 (NOT_A, A);* This line of code inverts input A and outputs it to NOT_A. *not* is a gate primitive in Verilog. It represents its namesake gate, the *NOT* gate. There are several other basic primitives in Verilog which will be introduced in coming labs when needed. 'n1' is the name of this instance of *not*. It is not mandatory to define an instance name for most of Verilog primitives, so, *not (NOT_A, A);* without *n1*, would have also worked fine. Note that the order of input and output must be as shown, i.e., the output comes first and input(s) come(s) after that.
6. *not n2 (NOT_B, B);* This line of code inverts input B and outputs it to NOT_B.
7. *and a1 (A_AND_NOT_B, A, NOT_B);* This line of code uses the *and* primitive of Verilog and ANDs the inputs A and NOT_B and outputs it to the wire A_AND_NOT_B. Here again, the instance name 'a1' can be skipped. 8. *and a2 (NOT_A_AND_B, NOT_A, B);* This line of code ANDs NOT_A and B and outputs the result to wire NOT_A_AND_B.
9. *or o1 (X, A_AND_NOT_B, NOT_A_AND_B);* This line of code ORs A_AND_NOT_B and NOT_A_AND_B and outputs the result to the wire X. Note that X is an output port, however, it, in fact, acts as a wire that can be accessed from outside this module. All wires that we declared are internal to this module and cannot be accessed from outside of this module. However all ports act as wires that can be accessed from outside of the module.
10. *endmodule* This line of code marks the end of the module. This line will be written at the end of all Verilog modules. Note that there is no semi-colon at the end of this line. Here, it is not necessary to put a semi-colon; however, putting one will not generate an error.

4.4.1.2. Test bench in Verilog

The module created above needs to be simulated in time. At the moment, this module's inputs have not been assigned any values and it does not have any notion of time. To do so, we will write its test bench as shown below:

```
module first_testbench;
    reg x, y;
    wire z;
    FirstVerilogModule F1 (.A(x), .B(y), .X(z));
    initial begin
          x = 0;
          y = 0;
          #100
          x = 0;
```

```
            y = 1;
            #100
            x = 1;
            y = 0;
            #100
            x = 1;
            y = 1;
        end
    endmodule
```

Now, let us look at each line of the above code and understand what the syntax means:

1. *module first_testbench;* This line initializes the Verilog module first_testbench. Note that this module does not have any port, as it is only being used to test our FirstVerilogModule and it is not expected to be used inside any other module. 2. *reg x, y;* This line declares two variables, x and y, with register data types. These will be used as inputs to our FirstVerilogModule. Contrary to a *wire*, *reg* data type can be used to store values. It can also be assigned inside an initial block which will be explained when its line of code is discussed below.
3. *wire z;* This line declares a wire z. This will be used as an output of FirstVerilogModule.
4. *FirstVerilogModule F1 (.A(x), .B(y), .X(z));* This line of code instantiates FirstVerilogModule inside this test bench. To test a module, it must be instantiated inside its test bench. In this case, giving an instance name, such as *F1*, is mandatory. Please note the syntax and the order of each term. The ports of FirstVerilogModule have been connected to the registers x and y and wire z that had been declared previously.
5. *initial begin* This line starts the *initial* procedural block in Verilog. The term *initial* marks the beginning of simulation time at time 0. The term *begin* in Verilog is the equivalent of opening brace ({) in C. The *initial* block is terminated by the term *end* which is the equivalent of closing brace (}) in C. The code inside an *initial* block is executed sequentially and only once.
6. *x = 0;* This line loads binary value 0 in reg x.
7. *y = 0;* This line loads binary value 0 in reg y. After both x and y have been loaded with certain values, it is expected that in simulation, the output z is updated according to the code of FirstVerilogModule.
8. *#100* This line introduces a delay of 100 time units in simulation before the next input change can occur. The time unit depends on the setting in the simulation software. In ModelSim, one time unit is equal to 1 picosecond by default. 9. Lines 9 to 16 load different binary values in x and y to check the output z for all possible combinations. Each input combination is held constant for some time by using 100 time units delay.
10. *end* This terminates the *initial* block.
11. *endmodule* This line marks the end of the module first_testbench.

4.4.2. ModelSim

ModelSim is an HDL simulation software, manufactured by Mentor Graphics. It can be used to simulate, debug and verify the operation of HDL programs. We will use ModelSim to verify our Verilog modules for all Verilog-based labs. Please refer to this lab's video tutorial to learn how to set up and run your first Verilog module in simulation using ModelSim. Figure 4.2 shows the timing diagrams obtained by simulating first_testbench module from the example above.

Figure 4.2: Timing diagrams obtained by simulating first_testbench

4.4.3. NAND/NOR logic

NAND gate is a logic gate whose output is the complement of the output of an AND gate. NAND operation is an important operation in digital logic because it has *functional completeness*, a property that implies that all Boolean operations (AND, OR and NOT) can be implemented using only NAND operation. Figure 4.3(a) illustrates the use of NAND gate as a universal gate. The equivalent NAND-only Boolean expressions for AND, OR and NOT are listed in Table 4.1.

NOR gate is the *dual* of NAND gate and its output is the complement of the output of an OR gate. NOR operation also has

functional completeness. Figure 4.3(b) illustrates how a NOR gate can be used to implement basic logic gates in a circuit. The equivalent NOR-only expressions for AND, OR and NOT are listed in Table 4.1.

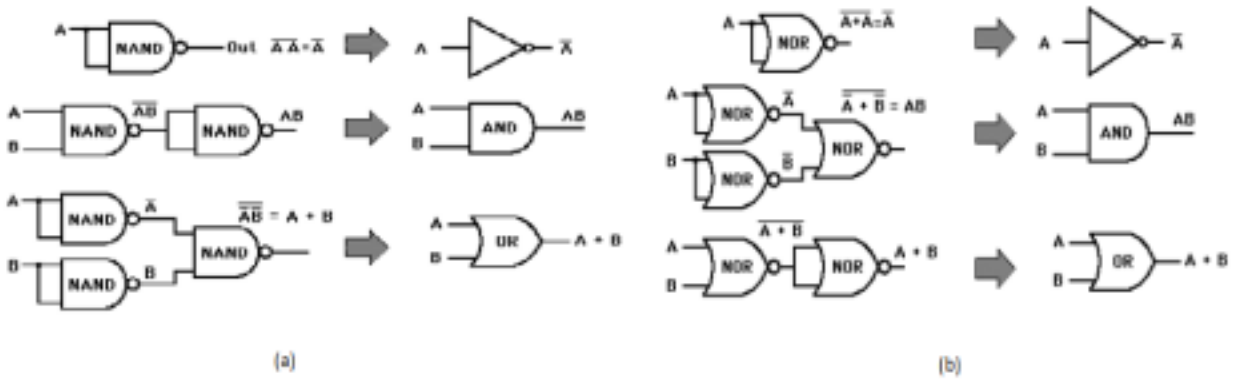


Figure 4.3: Implementation of basic logic gates using (a) NAND and (b) NOR

Table 4.1: Boolean expressions of NAND and NOR logic implementations of NOT, AND and OR

Standard form	NAND equivalent	NOR equivalent
$\bar{A}$	$\overline{A.A}$	$\overline{A+A}$
A. B	$\overline{\overline{A.B}}$ $\overline{\overline{A.B}}.\overline{(A.A)}.\overline{(B.B)}$	$\overline{\overline{A}+\overline{A}}+\overline{\overline{B}+\overline{B}}$
A + B	$\overline{\overline{A+B}}$ $\overline{\overline{A+B}}.\overline{(A.A)}.\overline{(B.B)}$	$\overline{\overline{A}+\overline{B}}+\overline{\overline{A}+\overline{B}}$

NAND and NOR gates are frequently used in digital logic design because of their property of functional completeness and ease of fabrication with electronic components.

4.5. Equipment and Software

8. Lab PC or personal laptop
9. ModelSim PE Student Edition
10. MS Word

4.6. Lab Tasks

4.6.1. Task 1: Implementing a logic function using Verilog structural modeling [Marks: 15]

11. Using the skeleton code given below, implement the following 2-variable function using structural modeling in Verilog: [5]

$$F = A.B + A\bar{B}$$

```
module my_func (A, B, F);
input A, B;
output F;
wire NOT_A, NOT_B, A_AND_B, NOT_A_AND_NOT_B;
```

```
// Write your code here;

module my_func(A,B,F);

input A,B;

output F;

wire NOT_A, NOT_B, A_AND_B, NOT_A_AND_NOT_B;

not G1(NOT_A, A);

not G2(NOT_B, B);

and G3(A_AND_B,A,B);

and G4(NOT_A_AND_NOT_B, NOT_A, NOT_B);

or G5(F, A_AND_B, NOT_A_AND_NOT_B);

endmodule
```

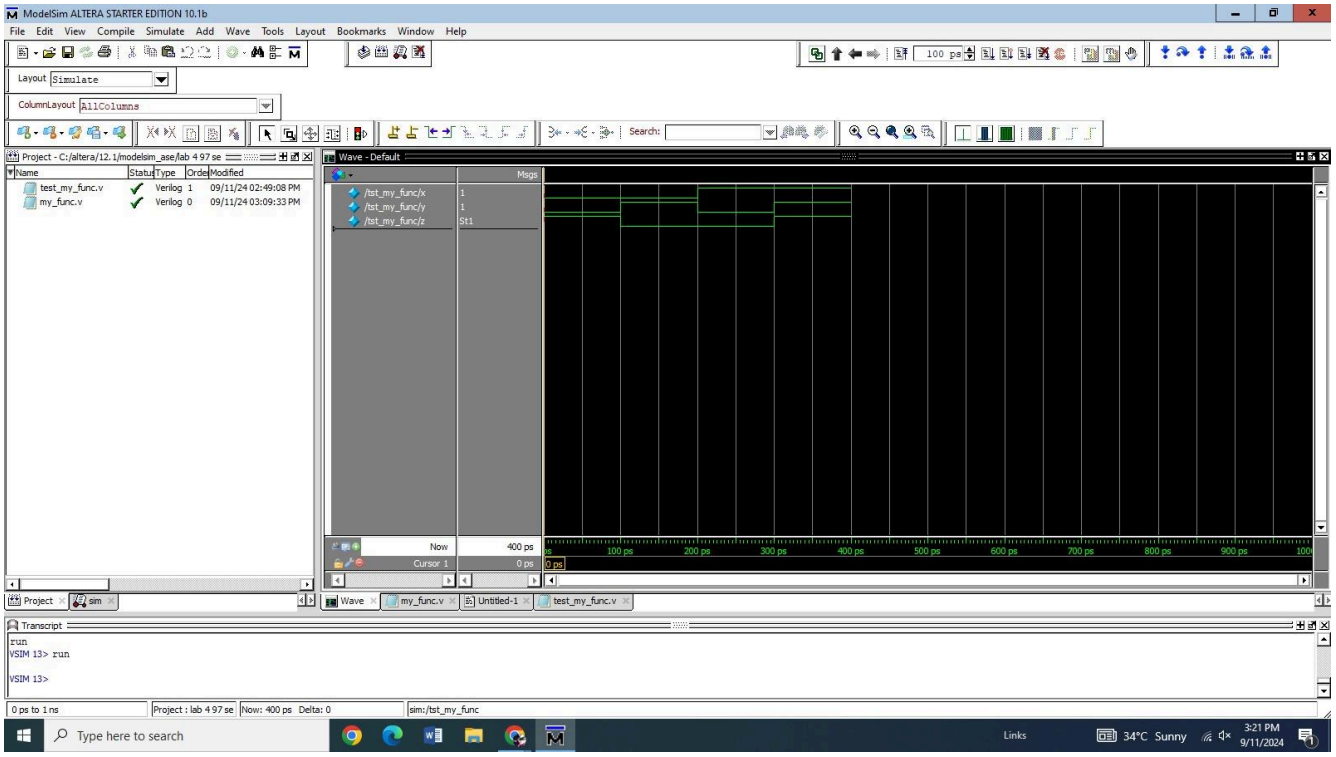
12. Write a test bench module to test the my_func module created above for all possible combinations of A and B. Use the skeleton code provided below: [5]

```
module test_my_func;
reg x, y;
wire z;
my_func func_1 (.A(x), .B(y), .F(z));
initial begin

// Write your code here;

   x = 0;

    y = 0;

    #100

    x = 0;

    y = 1;

    #100

    x = 1;

    y = 0;

    #100

    x = 1;
```

```
        y = 1;

    end
    endmodule
```

☐ Run simulation and verify the output of F from the timing waveforms. Show simulation results to the lab engineer to obtain credit. [2] ☐ Answer question 1 of Analysis.

☐ Take clear screenshots of code of all modules of this task and the simulation timing waveforms and paste them in a WORD file such that all fit on a single page. Make sure that screenshots are legible. [3]



4.6.2. Task 2: NAND implementation [Marks: 15]

☐ Consider a 4-variable function G:

$$G(A, B, C, D) = \sum m(0, 1, 2, 3, 5, 9, 13, 15)$$

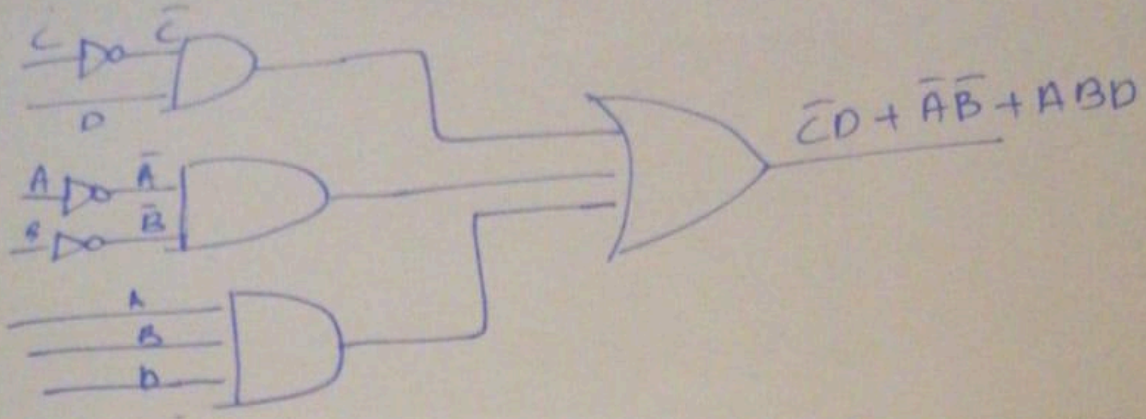
☐ Simplify this function either by use of Boolean algebra identities, or K-map. Show the simplification process clearly in the space given below: [4]

☐ Draw the logic circuit of the simplified expression of G in the space below: [2]

Replace all gates with their equivalent NAND implementations given in Figure 4.3. Next, remove any redundant pairs of NAND gates (pairs that cancel each other) in the circuit thus obtained and re-draw the final NAND-only circuit. Use the space given below: [4]

$\bar{C}D + \bar{A}\bar{B} + ABD$

18. Draw the logic circuit of the simplified expression of G in the space below. [2]



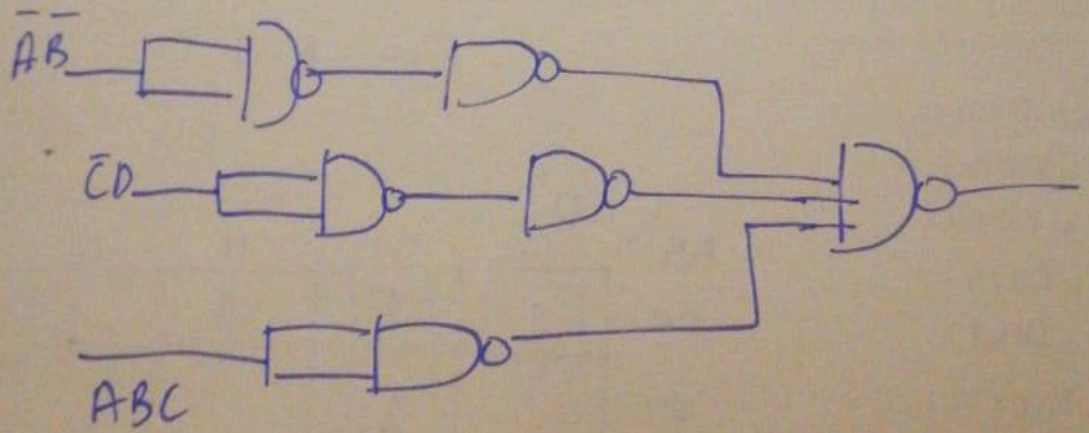
19. Replace all gates with their equivalent NAND implementations given in Figure 4.3. Next, remove any redundant pairs of NAND gates (pairs that cancel each other) in the circuit thus obtained and re-draw the final NAND-only circuit. Use the space given below. [4]

$\bar{C}D + \bar{B}\bar{A} + ABD$

$\overline{\bar{A}B} \cdot \overline{\bar{C}D} \cdot \overline{A\bar{B}D}$

$\overline{\overline{\bar{A}B}} + \overline{\overline{\bar{C}D}} + \overline{\overline{A\bar{B}D}}$

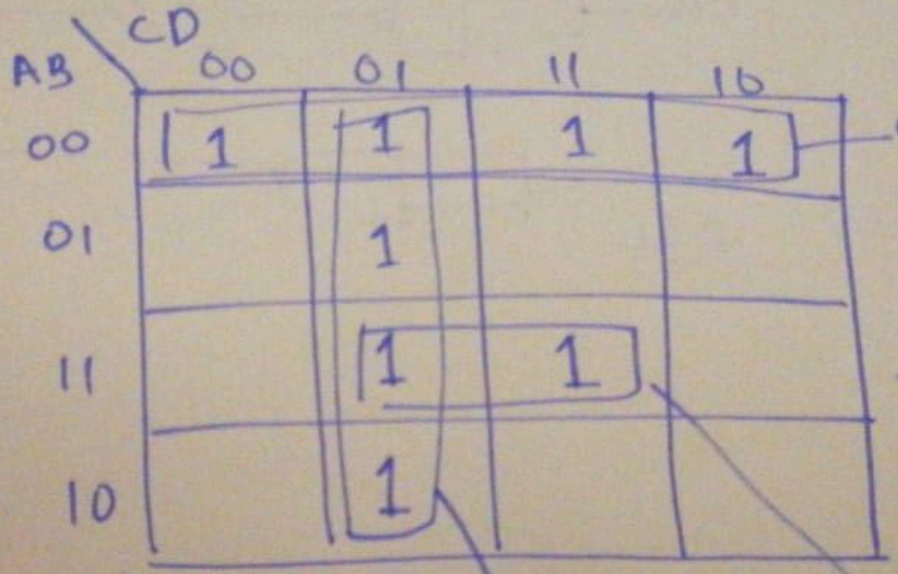
$\bar{A}B + \bar{C}D + A\bar{B}D$



17. Simplify this function either by use of Boolean algebra identities, or K-map. Show the simplification process clearly in the space given below: [4]

$\bar{A} = 0 \quad A = 1$

$m_0 = 0000$
$m_1 = 0001$
$m_2 = 0010$
$m_3 = 0011$
$m_5 = 0101$
$m_9 = 1001$
$m_{13} = 1101$
$m_{15} = 1111$



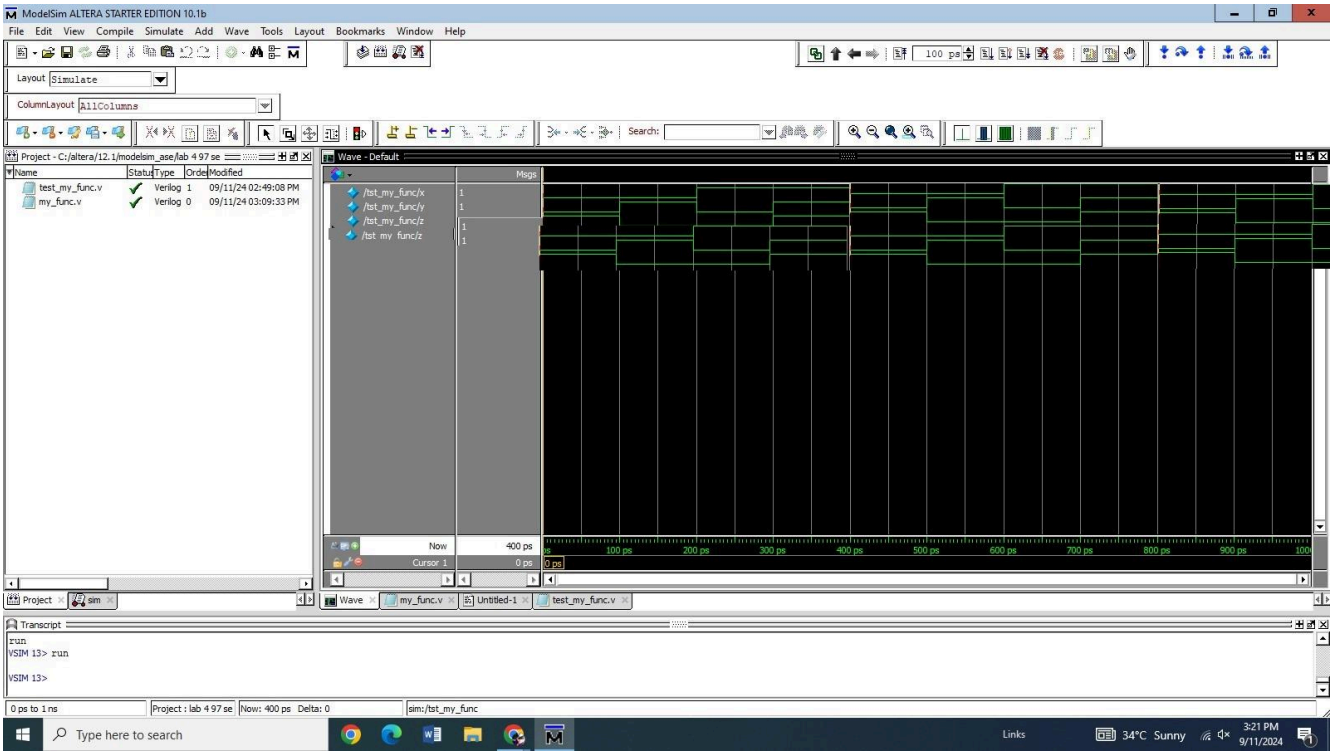
$\bar{C}D + \bar{A}\bar{B} + ABD$

Using structural modeling, write code for a Verilog module that uses only the *nand* gate primitive to implement the NAND-only circuit derived in step 4. [2] Write code for a test bench module that tests the above module for all combinations of A, B, C and D. [1] Fill in the truth table of G in Table 4.2 according to the timing waveforms obtained after running the simulation. Verify that the truth table corresponds to the minterms of G. Show simulation results to the lab engineer to obtain credit. [2]

Take clear screenshots of code of all modules of this task and the simulation timing waveforms and paste them in your WORD file such that all fit on no more than two pages. Make sure that screenshots are legible.

```
module my_func (A, B, C, D, F);
input A, B, C, D;
output F;
wire NOT_A, NOT_B, NOT_C, NOT_D;
wire term1, term2, term3;

// Inverting inputs
not (NOT_A, A);
not (NOT_B, B);
not (NOT_C, C);
not (NOT_D, D);

// Terms of the expression
and (term1, NOT_C, D);
and (term2, NOT_A, NOT_B);
and (term3, A, B, D);

// ORing the terms to get the final output
or (F, term1, term2, term3);
```

```
endmodule

module test_my_func;
reg A, B, C, D;
wire F;
my_func func_1 (.A(A), .B(B), .C(C), .D(D), .F(F));

initial begin
  // Test all combinations of A, B, C, and D
  $monitor("A=%b, B=%b, C=%b, D=%b, F=%b", A, B, C, D, F);
  A = 0; B = 0; C = 0; D = 0; #10;
  A = 0; B = 0; C = 0; D = 1; #10;
  A = 0; B = 0; C = 1; D = 0; #10;
  A = 0; B = 0; C = 1; D = 1; #10;
  A = 0; B = 1; C = 0; D = 0; #10;
  A = 0; B = 1; C = 0; D = 1; #10;
  A = 0; B = 1; C = 1; D = 0; #10;
  A = 0; B = 1; C = 1; D = 1; #10;
  A = 1; B = 0; C = 0; D = 0; #10;
  A = 1; B = 0; C = 0; D = 1; #10;
  A = 1; B = 0; C = 1; D = 0; #10;
  A = 1; B = 0; C = 1; D = 1; #10;
  A = 1; B = 1; C = 0; D = 0; #10;
  A = 1; B = 1; C = 0; D = 1; #10;
  A = 1; B = 1; C = 1; D = 0; #10;
  A = 1; B = 1; C = 1; D = 1; #10;
end
endmodule
```



12. **Table 4.2: Truth table of G**

A	B	C	D	G
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

4.6.3. Analysis [Marks: 5]

Draw the truth table of F. Does the simulation result correspond to this truth table? [1] How is Verilog

different from other computer programming languages such as C, C++, Java, etc.? [2]

$G) + \bar{A}\bar{B} + ABD$

Table 4.2: Truth table of G

A	A'	B	B'	C	C'	D	G
0	1	0	1	0	1	0	1
0	1	0	1	0	1	1	1
0	1	0	1	1	0	0	1
0	1	0	1	1	0	1	1
0	1	1	0	0	1	0	1
0	1	1	0	0	1	1	0
0	1	1	0	1	0	0	0
0	1	1	0	1	0	1	0
1	0	0	1	0	1	0	0
1	0	0	1	0	1	1	1
1	0	0	1	1	0	0	0
1	0	0	1	1	0	1	1
1	0	1	0	0	1	0	0
1	0	1	0	0	1	1	1
1	0	1	0	1	0	0	0
1	0	1	0	1	0	1	1

4.1.1. Analysis [Marks: 5]

24. Draw the truth table of F. Does the simulation result correspond to this truth table? [1]

A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

25. How is Verilog different from other computer programming languages such as C, C++, Java, etc.? [2]

Verilog is used to model and simulate digital circuits. whereas C/C++ such are software programming languages. used to write programs that run on a processor.

Assessment Rubric

Method:
Lab report evaluation and instructor observation during lab sessions.

Performance	CLO	Able to complete the tasks over 80% (4-5)	Able to complete the tasks 50 – 80% (2-3)	Tasks completion below 50% (0-1)	Marks
1. Teamwork	1	Actively engages and cooperates with other group members in an effective manner	Cooperates with other group members in a reasonable manner	Distracts or discourages other group members from conducting the experiments	
2. Laboratory safety and disciplinary rules	1	Observes lab safety rules; handles the development board and other components with care and adheres to the lab disciplinary guidelines aptly	Observes safety rules and disciplinary guidelines with minor deviations	Disregards lab safety and disciplinary rules	
3. Realization of experiment	3	Conceptually understands the topic under study and develops the experimental setup accordingly	Needs guidance to understand the purpose of the experiment and to develop the required setup	Incapable of understanding the purpose of the experiment and consequently fails to develop the required setup	
4. Conducting experiment	3	Sets up hardware/software properly according to the requirement of experiment and examines the output carefully	Makes minor errors in hardware/software setup and observation of output	Unable to set up experimental setup, and perform the procedure of experiment	
5. Data collection	3	Completes data collection from the experiment setup by giving proper inputs and observing the outputs, complies with the instructions regarding data entry in manual	Completes data collection with minor errors and enters data in lab report with slight deviations from provided guidelines	Fails at collecting data by giving proper inputs and observing output states of experiment setup, unable to fill the lab report properly	
6. Data analysis	3	Analyzes the data obtained from experiment thoroughly and accurately verifies it with theoretical understanding, accounts for any discrepancy in data from theory with sound explanation, where asked	Analyzes data with minor error and correlates it with theoretical values reasonably. Attempts to account for any discrepancy in data from theory	Unable to establish the relationship between practical and theoretical values and lacks the theoretical understanding to explain any discrepancy in data	

7. Computer use	3	Successfully uses lab PC and internet to look for relevant datasheets, carry out calculations, or verify results using simulation	Requires assistance in looking for IC datasheets and carrying out calculation and simulation tasks	Does not know how to use computer to look up datasheets or carry out calculation and simulation tasks	
				Total (out of 35)	