

Digital Logic Design

EEE241

Lab Manual



Name	
Registration Number	
Class	
Instructor's Name	

Introduction

This is the Lab Manual for EEE – 241 Digital Logic Design. The labs constitute 25 % of the total marks for this course.

During the labs you will work in groups (no more than three students per group). You are required to complete the ‘Pre-Lab’ section of the lab before coming to the lab. You will be graded for this and the ‘In-Lab’ tasks during the in-lab viva. You will complete the ‘Post-Lab’ section of each lab before coming to the next week’s lab.

You are not allowed to wander in the lab or consult other groups when performing experiments. Similarly, the lab reports must contain original efforts. CIIT has a zero tolerance anti-plagiarism policy.

Apart from these weekly labs you will complete two projects. Lab Sessional I and Lab Sessional II will be conducted and graded as 10% and 15% marks, respectively. Final Project / Final Exam which will be graded as Lab Terminal. The grading policy is already discussed in the Course Description File.

Acknowledgement

The labs for EEE-241 Digital Logic Design were designed and prepared by Dr. M. Faisal Siddiqui. The manuals were updated by Mr. Ali Raza Shahid, Mr. Sikender Gul and Mr. Shahid Mehmood. The first version was completed in Session Spring 2014, The second version was completed during the summer break of 2016. Third version is completed in session Spring 2016-17. Fourth version is completed in session Spring 2018. Open ended labs were introduced in fifth version and completed in Summer 2018. Typesetting and formatting of this version was supervised by Dr. Omar Ahmad and was carried out by Mr. Abdul Rehman, Mr. Suleman & Mr. Baqir Hussain.

History of Revision

Version and Date of Issue	Team	Comments
Version 1. May 2014	Dr. M. Faisal Siddiqui.	This is the first editable draft of EEE – 241 lab manual. Labs were designed by Dr. M. Faisal Siddiqui. For comments and suggestions please contact: faisal_siddiqui@comsats.edu.pk
Version 2. September 2016	Dr. Riaz Hussain. Mr. Ali Raza Shahid. Mr. Sikender Gul. Ms. Ayesha Hameed. Mr. Nouman Riaz.	This is the second editable draft of EEE – 241 lab manual. The manual was remodeled according to the new OBE format. Some of the labs were improved.
Version 3. January 2017	Dr. Ahmad Naseem Alvi. Dr. M. Faisal Siddiqui. Mr. Ali Raza Shahid. Mr. Sikender Gul.	This is the third editable draft of EEE – 241 lab manual. Some amendments had been made after the feedback from the instructor during Fall-2016.

Version 4. January 2018	Dr. M. Faisal Siddiqui	<p>This is the fourth editable draft of EEE – 241 lab manual. Some of the labs were improved. Pre-Lab, In-Lab and Post-Lab tasks were improved. Sequential circuit labs were re-modelled. Amendments had been made after the feedback from the DLD stream course and lab Instructors.</p> <p>For comments and suggestions please contact: faisal_siddiqui@comsats.edu.pk</p>
Version 5. August 2018	Dr. M. Faisal Siddiqui	<p>This is the fifth editable draft of EEE – 241 lab manual. Open ended labs were introduced in this version.</p> <p>For comments and suggestions please contact: faisal_siddiqui@comsats.edu.pk</p>

Safety Precautions

- Be calm and relaxed, while working in lab.
- First check your measuring equipment.
- When working with voltages over 40 V or current over 10 A, there must be at least two people in the lab at all time.
- Keep the work area neat and clean.
- Be sure about the locations of fire extinguishers and first aid kit.
- No loose wires or metals pieces should be lying on the table or near the circuit.
- Avoid using long wires, that may get in your way while making adjustments or changing leads.
- Be aware of bracelets, rings, and metal watch bands (if you are wearing any of them). Do not wear them near an energized circuit.
- When working with energized circuit use only one hand while keeping rest of your body away from conducting surfaces.
- Always check your circuit connections before power it ON.
- Always connect connection from load to power supply.
- Never use any faulty or damaged equipment and tools.
- If an individual comes in contact with a live electrical conductor.
 - Do not touch the equipment, the cord, the person.
 - Disconnect the power source from the circuit breaker and pull out the plug using insulated material.

Table of Contents

Introduction.....	2
Acknowledgement	3
History of Revision.....	3
Safety Precautions.....	5
Table of Contents	6
LAB #01: Introduction to Basic Logic Gate ICs on Digital Logic Trainer and Proteus Simulation.....	9
Objective	9
Pre-Lab:.....	9
In-Lab:.....	14
Post-Lab Tasks:.....	22
Critical Analysis/Conclusion	24
LAB #02: Boolean Function Implementation using Universal Gates	25
Objectives	25
Pre-Lab:.....	25
In lab:	25
Post-Lab:	32
Critical Analysis/Conclusion	34
LAB #03: Introduction to Verilog and Simulation using XILINX ISE	35
Objective	35
Pre-Lab:.....	35
In-Lab:.....	39
In-Lab Task 2:.....	46
Post-Lab:	46
Critical Analysis/Conclusion	49
LAB #04: Design and Implementation of Boolean Functions by Standard Forms using ICs/Verilog.....	50
Objective	50
Pre-Lab:.....	50
Pre-Lab Tasks:	54
In-Lab Tasks:	55
Post-Lab Tasks:.....	60

Critical Analysis/Conclusion	62
LAB #05: Logic Minimization of Complex Functions using Automated Tools	63
Objective	63
Pre-Lab:.....	63
In-Lab:.....	64
Post-Lab Tasks:	70
Critical Analysis/Conclusion	72
LAB #06: Xilinx ISE Design Flow with FPGA.....	73
Objective	73
Pre-Lab:.....	73
In-Lab:.....	77
In-Lab Task:	82
Post-Lab Task:	82
Critical Analysis/Conclusion	83
7. LAB #07: Design and Implementation of $n - bit$ Adder/Subtractor on FPGA.....	84
Objectives	84
Pre-Lab:.....	84
In-Lab Tasks:	85
Post-Lab:	89
Critical Analysis/Conclusion	91
LAB #08: Design and Implementation of $n - bit$ Binary Multiplier on FPGA	92
Objective	92
Pre-Lab:.....	92
Pre-Lab Task	94
In-Lab Task 1: Implement $4 - bit$ by $2 - bit$ binary multiplier using ICs.....	94
In-Lab Task 2: Implement $4 - bit$ by $3 - bit$ binary multiplier on FPGA.....	94
Post-Lab Tasks:.....	95
Critical Analysis/Conclusion	96
LAB #09: Design and Implementation of BCD to 7-Segment Decoder on FPGA	97
Objective	97
Pre-Lab:.....	97
In-Lab Task 1: Test the functionality of a BCD to 7-Segment decoder IC (CD4511) with common cathode 7-Segment display	100

In-Lab Task 2: Design and implementation of a BCD to 7-Segment decoder on FPGA (Nexys2).....	101
Post-Lab Task:	102
Critical Analysis/Conclusion	104
Lab #10: Design and Implementation of a Sequence Detector using Mealy/Moore Machine.....	105
Objective	105
Pre-Lab:.....	105
In-Lab: Design and Implementation of a Sequence Detector using Mealy Machine	107
Sequence: _____	107
Post-Lab Tasks:.....	111
Critical Analysis/Conclusion	114
LAB #11: Implementation of a BCD Counter with Control Inputs on FPGA	115
Objective	115
Pre-Lab:.....	115
In-Lab: Implementation of a BCD Counter with control inputs on FPGA.....	118
Post-Lab Task:	122
Critical Analysis/Conclusion	123
Lab #12: Implementation of a Special Shift Register on FPGA	125
Objective	125
Pre-Lab:.....	125
In-Lab Tasks:	131
Post-Lab Task:	136
Critical Analysis/Conclusion	137

LAB #01: Introduction to Basic Logic Gate ICs on Digital Logic Trainer and Proteus Simulation

Objective

Part 1

To know about the basic logic gates, their truth tables, input-output characteristics and analyzing their functionality. Introduction to logic gate ICs, Integrated Circuits pin configurations and their use.

Part 2

Learn to use Proteus Software for Simulation of Digital Logic Circuits.

Pre-Lab:

Background Theory:

The Digital Logic Circuits can be represented in the form of **(1) Boolean Functions**, **(2) Truth Tables**, and **(3) Logic Diagram**. Digital Logic Circuits may be practically implemented by using electronic gates. The following points are important to understand.

- Electronic gates are available in the form of Integrated Circuits (ICs) and they require a power.
- Supply Gate INPUTS are driven by voltages having two nominal values, e.g. 0V and 5, 12V representing logic 0 and logic 1 respectively.
- The OUTPUT of a gate provides two nominal values of voltage only, e.g. 0V and 5, 12V representing logic 0 and logic 1 respectively. In general, there is only one output to a logic gate except in some special cases.
- Truth tables are used to help show the function of a logic gate in terms of input values combination with the desired output.
- Logic Diagram is used to represent the Digital Logic Circuit in the form of symbols connected with each other.
- Digital Logic Circuits can be simulated in the virtual environment called simulation software

The basic operations are described below with the aid of Boolean function, logic symbol, and truth table.

AND gate:

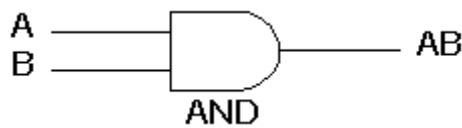


Figure 1.1: AND gate

Table 1.1: Truth Table of 2 input AND gate

A	B	$F = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. $A \cdot B$. Bear in mind that this dot is sometimes omitted i.e. AB .

OR gate:

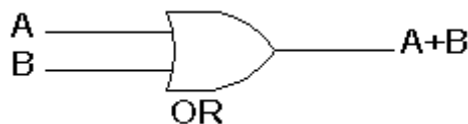


Figure 1.2: OR gate

Table 1.2: Truth Table of 2 input OR gate

A	B	$F = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high. A plus (+) is used to show the OR operation.

NOT gate

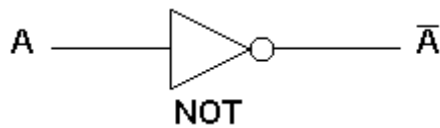


Figure 1.3: NOT gate

Table 1.3: Truth Table of NOT gate

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

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an *inverter*.

NAND gate



Figure 1.4: NAND gate

Table 1.4: Truth Table of 2 input NAND gate

A	B	$F = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The output of NAND gate is **high** if **any** of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

NOR gate

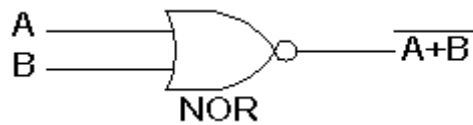


Figure 1.5: NOR gate

Table 1.5: Truth Table of 2 input NOR gate

A	B	$F = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The output of NOR gate is **low** if **any** of the inputs are high. The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

XOR gate

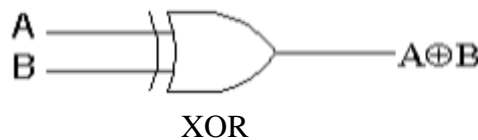


Figure 1.6: XOR gate

Table 1.6: Truth Table of 2 input XOR gate

A	B	$F = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

The '**Exclusive-OR**' gate is a circuit which will give a high output if **odd** number of inputs are high. An encircled plus sign " \oplus " is used to show the EOR operation.

XNOR gate

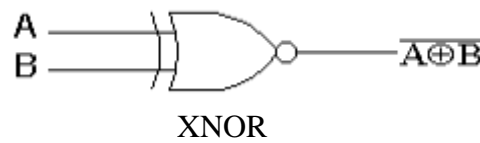


Figure 1.7: XNOR gate

Table 1.7: Truth Table of 2 input XNOR gate

<i>A</i>	<i>B</i>	$F = \overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

The '**Exclusive-NOR**' gate circuit does the opposite to the XOR gate. It will give a high output if **even** number of inputs are high. The symbol is an XOR gate with a small circle on the output. The small circle represents inversion.

Digital systems are said to be constructed by using logic gates. These gates are AND, OR, NOT, NAND, NOR, XOR and XNOR. Logic gate ICs are available in different packages and technologies. Two main classifications are as below:

1. 74 Series TTL Logic ICs
2. 4000 Series CMOS Logic ICs

74 series is TTL (*Transistor-Transistor Logic*) based integrated circuits family. Power rating for 74 series is 5 to 5.5Volts. This circuitry has fast speed but requires more power than later families. The Pin configuration of basic gates 2-input ICs for 74 Series is given in Figure 1.8:

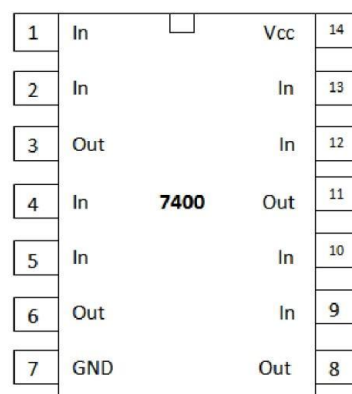


Figure 1.8: TTL ICs' pin configuration

Figure 1.8 shows the **4000 series** is CMOS (complementary metal oxide semiconductors) based integrated circuits. Power ratings are 3V to 15 Volts. CMOS circuitry consumes low power, but it is not fast as compared to TTL.

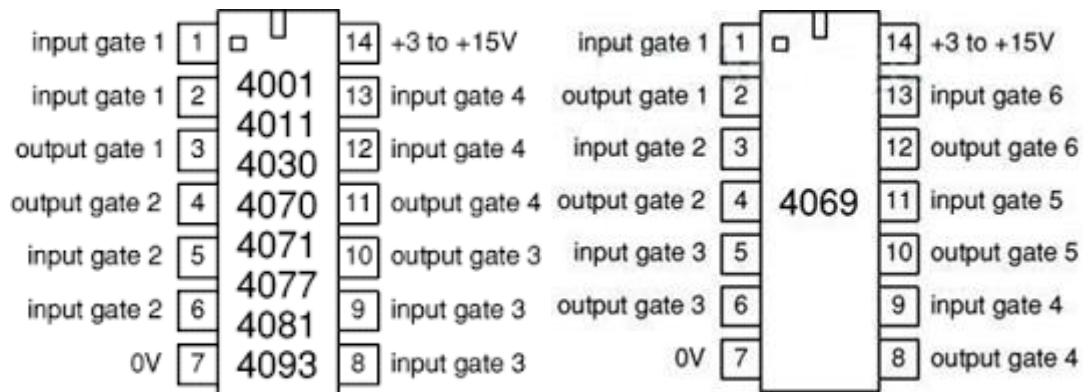


Figure 1.9: Different CMOS ICs' pin configuration

Quad 2-input gates

The ICs available in Lab to perform the Tasks are listed below:

In-Lab:

Part 1: Basic Logic Gate Integrated Circuits (ICs)

Equipment Required

- KL-31001 Digital Logic Lab
- Logic gates ICs
 - 4001 quad 2-input NOR
 - 4011 quad 2-input NAND
 - 4070 quad 2-input XOR
 - 4071 quad 2-input OR
 - 4077 quad 2-input XNOR
 - 4081 quad 2-input AND
 - 4069 Six Inverting Buffer NOT

Procedure

1. Place the IC on the breadboard as shown in the Figure 1.10;
2. Using the power supply available at KL-31001 Digital Logic Lab trainer, connect pin7 (Ground) and pin14 (Vcc) to power up IC.

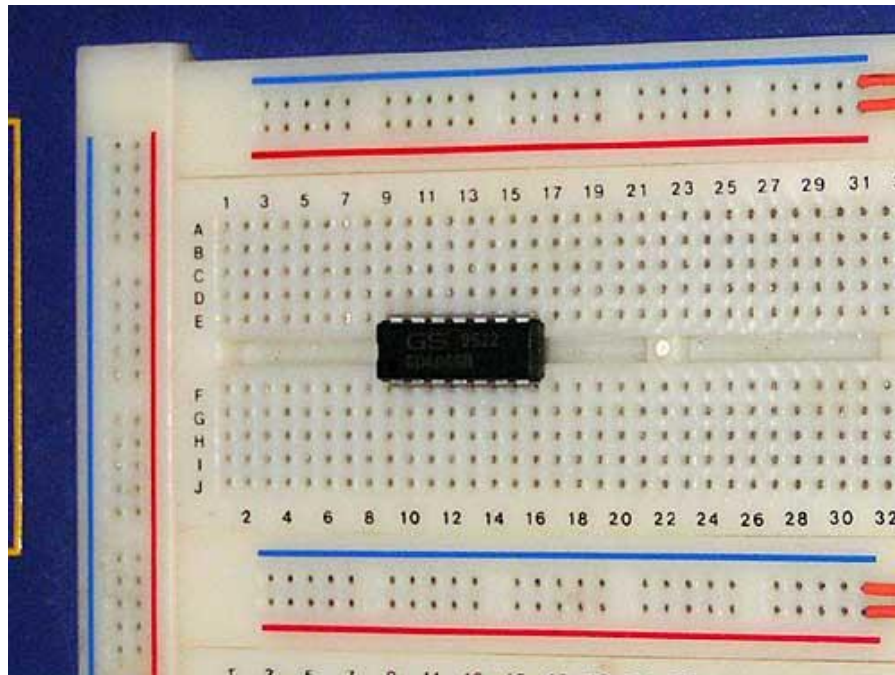


Figure 1.10: IC placement on the breadboard

3. Select number of possible combinations of inputs using the slide switches SW0-SW3 (as shown in Tables 1.8 & 1.9) and note down the output with the help of LED for all gate ICs. (You can use LD0-LD14 located on KL-31001 Digital Logic Lab)
(Note: Please make sure the Trainer board is off during the setup of circuit)

In-lab Task 1:

Verify all gates using their ICs on KL-31001 Digital Logic Lab trainer

Table 1.8: Observation Table for different gates

INPUTS		OUTPUTS					
<i>A</i>	<i>B</i>	<i>AND</i>	<i>OR</i>	<i>XOR</i>	<i>NAND</i>	<i>NOR</i>	<i>XNOR</i>
0	0						
0	1						
1	0						
1	1						

Table 1.9: Observation Table for NOT gate

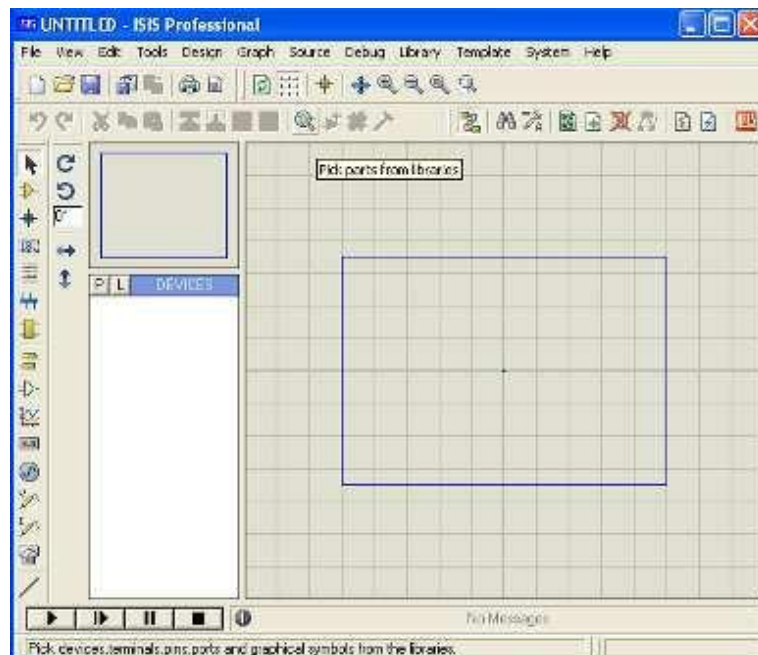
INPUT	OUTPUT
<i>A</i>	<i>B</i>
0	
1	

Part 2 - Proteus (Simulation Software)

Proteus has many features to generate both analog and digital results over a virtual environment. However, this lab will focus on tools that will be used in digital schematic designs and verification of basic logic gates.

Procedure

The Proteus software for simulation is installed in Digital Design Lab. Please follow the details below to figure out the usage of Proteus tools and process of simulation.

*Figure 1.11: Interface of Proteus software window*

Parts Browsing:

Proteus has many models of electronic equipment such as logic gates, many kinds of switches and basic electronic devices. The equipment can be placed by clicking on it and then a new window will pop-up as shown in Figure 1.12.

Finding Steps:

1. Type information of device such as “OR gate” in “*Keywords*’ box.
2. If some specific category is known, the device can narrow on focusing by selecting catalogue in the “*Category*” box.
3. After the information is entered, the list of related devices will appear in the “*Results*” window, so that needed device can be chosen and then click “OK” button to confirm selection in Figure 1.13.

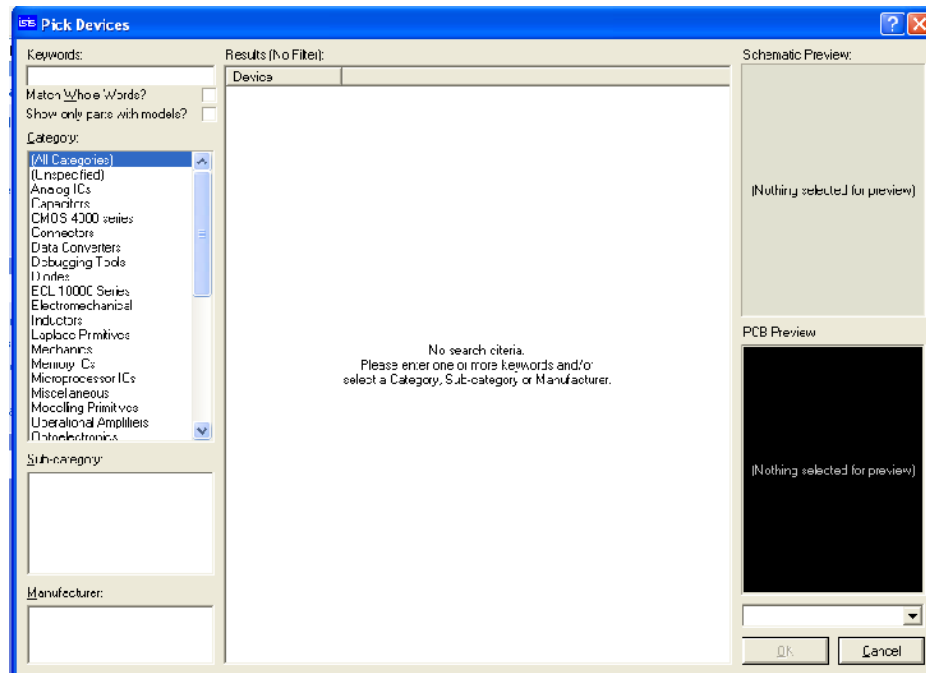


Figure 1.12: Pick Devices window in Proteus

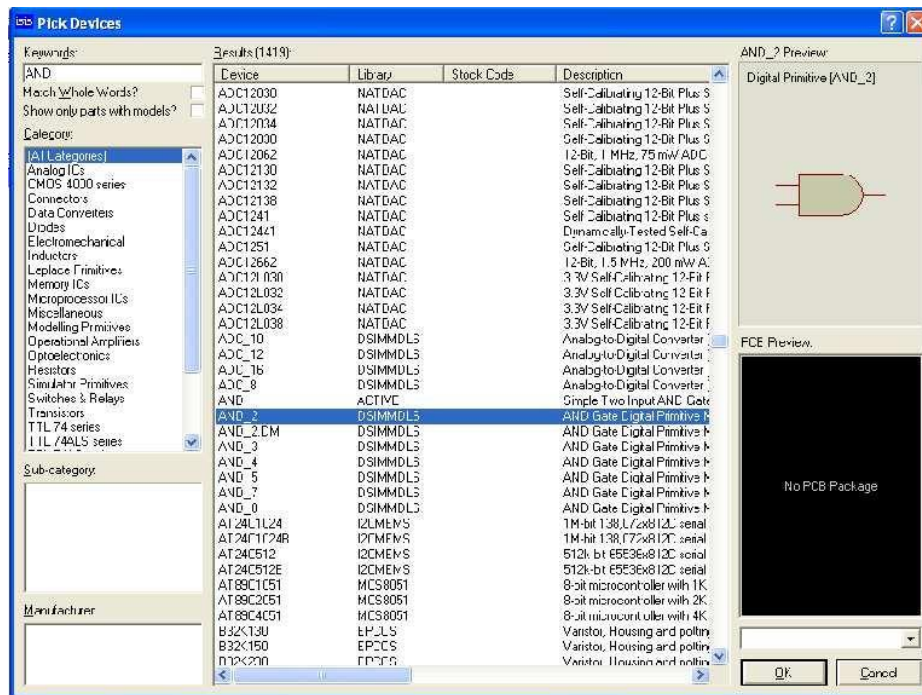


Figure 1.13: Pick selected Devices window in Proteus

Power supply and input signal Generator

All the electrical circuits require power supplies. The power supplies for logic circuits are represented in digital system design on Proteus because the schematic may be too complicated to understand for simulation section. Therefore, power supplies will be needed as input power for a system. Moreover, all the input generators, such as AC generator, DC and pulse, are contained in this category and it will be shown when clicked. In addition, “Ground” will not be available in this group. Because it is not an input signal it is just a terminal junction. Therefore, it will be grouped in the terminal category as shown in Figures 1.14 & 1.15.

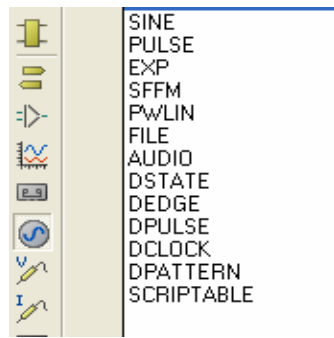


Figure 1.14: Power supplies window in Proteus

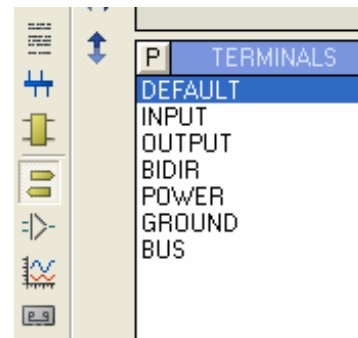


Figure 1.15: Terminals window in Proteus

Logic State:

In addition, there is another input that usually used in the digital circuit, but it does not exist in the real world as an equipment it is called as “**LOGIC STATE**”. It can be found in the picking part section (*type logic state and pick it as shown in Figure 1.16*).

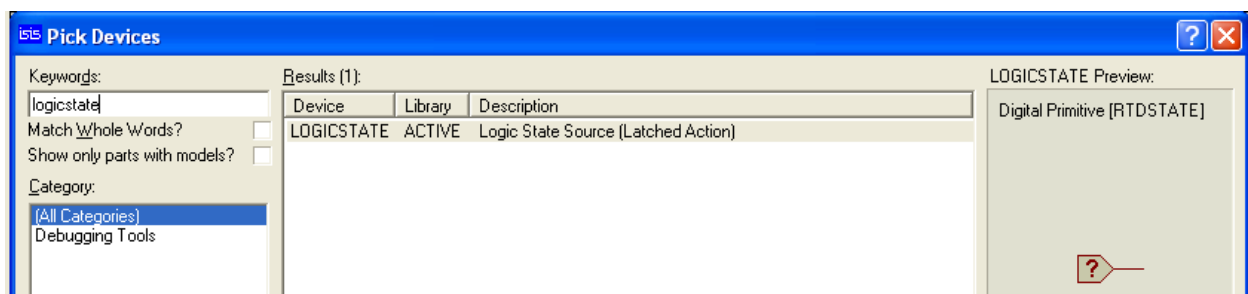


Figure 1.16: Logic State in Proteus

Placing Equipment:

Selecting all devices needed to be placed on the circuit window (*Gray window*) and make the required connections. It can be done by following steps:

1. Click on and select the first device that will be placed.
2. Place mouse wherever the device is preferred to place and then click the left button of the mouse. The device will be placed, if it is needed to be moved, click the right button of the mouse on the device symbol to select the mouse. Then hold this device with the left mouse button and move it to any desired place (Figure 1.17).

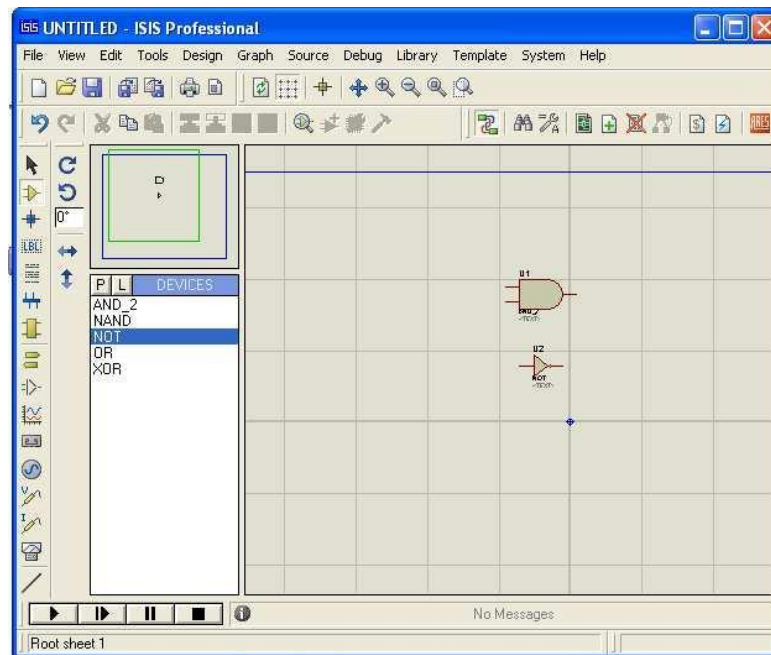


Figure 1.17: Placing the devices in Proteus

To make the connections between the devices, click on the source pin of a device and then move the cursor to destination pin of a device. In this step, the pink line will appear, and it will be a wire of the circuit after clicking the mouse on the destination pin of the circuit (as shown in Figure 1.18).

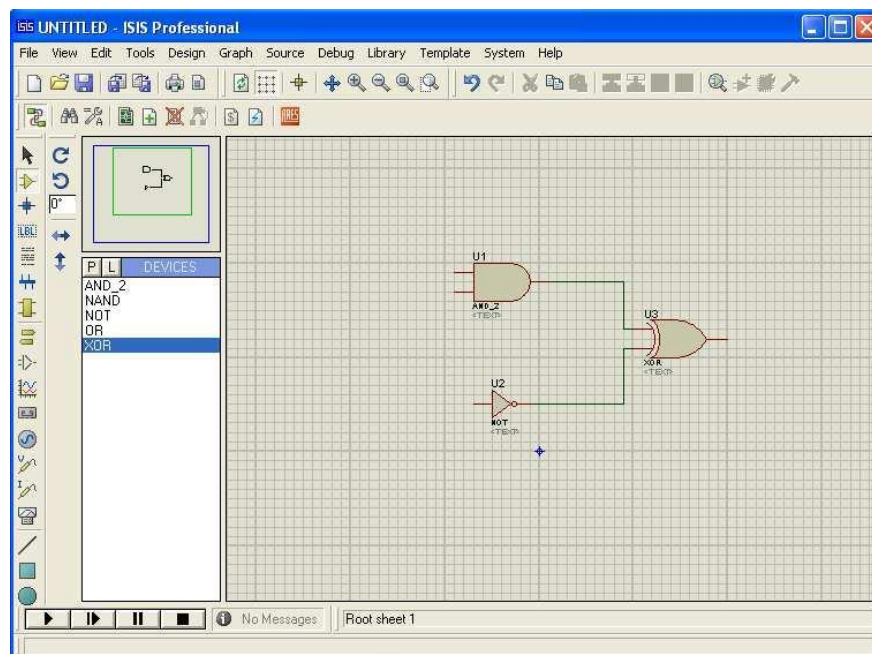


Figure 1.18: Making connection between devices to make a circuit

After wiring all devices and connect all inputs according to the circuit, the simulation is ready to run by clicking on Play button and stop button is used to stop the simulation.

3. Logic probe or LED can be used to observe the output state.

NOTE: The digital result on Proteus can be seen also in Small Square Box at the pin of the equipment & state can be shown in four colors. (Red= Logic 1, Blue = Logic 0, Gray= Unreadable and Yellow= Logic Congestion)

In-Lab Task 2:

Verify all the basic logic gates using the Proteus simulation tool and note down the values in the Tables 1.10 & 1.11 with the corresponding logic symbol and Boolean function. Then show the simulated logic circuit diagrams to your Lab Instructor.

Table 1.10: Observation Table for different gates

INPUTS		OUTPUTS					
<i>A</i>	<i>B</i>	<i>AND</i>	<i>OR</i>	<i>XOR</i>	<i>NAND</i>	<i>NOR</i>	<i>XNOR</i>
0	0						
0	1						
1	0						
1	1						

Table 1.11: Observation Table for NOT gate

INPUT	OUTPUT
<i>A</i>	<i>B</i>
0	
1	

Post-Lab Tasks:

1. Make a list of logic gate ICs of TTL family and CMOS family along with the ICs names. *(Note: at least each family should contain 15 ICs)*

	7400 Series	4000 Series
<i>1</i>		
<i>2</i>		
<i>3</i>		
<i>4</i>		
<i>5</i>		
<i>6</i>		
<i>7</i>		
<i>8</i>		
<i>9</i>		
<i>10</i>		
<i>11</i>		
<i>12</i>		
<i>13</i>		
<i>14</i>		
<i>15</i>		

2. What is Fan-In and Fan-Out?

Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

LAB #02: Boolean Function Implementation using Universal Gates

Objectives

- This lab is designed to simulate and implement any logic function using universals gates (NAND/NOR).
- To build the understanding of how to construct any combinational logic function using NAND or NOR gates only.

Pre-Lab:

Background theory:

Digital circuits are more frequently constructed with universal gates. NAND and NOR gate are called universal gates. Any Boolean logic function can be implemented using NAND only or NOR only gates. NAND and NOR gates are easier to fabricate with electronic components than basic gates. Because of the prominence of universal gates in the design of digital circuits, rules and procedures have been developed for conversion from Boolean function given in terms of AND, OR, and NOT into its equivalent NAND and NOR logic diagram.

Read and understand the universal gates. List the truth tables of AND, OR, NOT, NAND, NOR and XOR gates. Identify the NAND and NOR ICs and their specification for CMOS and TTL families.

In lab:

This lab has two parts. In the first part, simulation and implementation of any logic expression by using only NAND gates are done. In the second part, the same procedure is done by using NOR gates only.

Part 1 - Implementing any logic expression by using only NAND gates

If we can show that the logical operations AND, OR, and NOT can be implemented with NAND gates, then it can be safely assumed that any Boolean function can be implemented with NAND gates.

Procedure

- Simulate NOT, AND, OR, XOR and XNOR gates in Proteus software, by using only NAND gates. Verify their truth tables.
- Insert the IC on the trainer's breadboard.
- Use any one or more of the NAND gates of the IC for this experiment.

- One or more Logic Switches of the trainer (S1 to S9) can be used for input to the NAND gate.
- For output indication, connect the output pin of the circuit to any one of the LEDs of the trainer (L0 to L15).

In-Lab Tasks-Part-1

In-Lab Task 1.1: Verification of NOT function

- Connect the circuit as shown in Figure 2.1.
- Connect +5V to pin 14 (Vcc) and Ground to pin 7 (GND) of the IC.
- By setting the switches to 1 and 0, verify that the output (F) of the circuit conforms to that of a NOT gate. Record your observations in the Table 2.1 below.

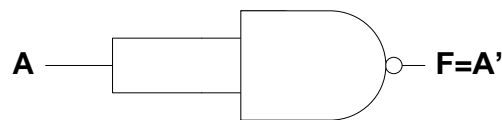


Figure 2.1: NOT gate using NAND gate

Table 2.1: Observation Table for NOT gate

INPUT	OUTPUT
<i>A</i>	<i>F</i>
0	
1	

In-Lab Task 1.2: Verification of AND function

- Connect the circuit as shown in Figure 2.2.
- Connect +5V to pin 14 (Vcc) and Ground to pin 7 (GND) of the IC.
- By setting the switches to 1 and 0, verify that the output (F) of the circuit conforms to that of an AND gate. Record your observations in Table 2.2 below.

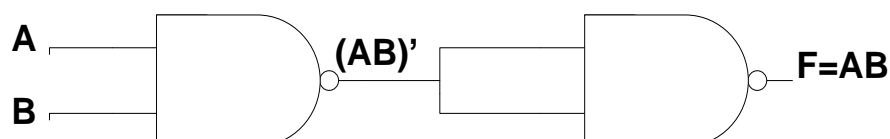


Figure 2.2: AND gate using NAND gates

Table 2.2: Observation Table for AND gate

INPUTS		OUTPUT
<i>A</i>	<i>B</i>	<i>F</i>
0	0	
0	1	
1	0	
1	1	

In-Lab Task 1.3: Verification of OR function

- Connect the circuit as shown in Figure 2.3.
- Connect +5V to pin 14 (Vcc) and Ground to pin 7 (GND) of the IC.
- By setting the switches to 1 and 0, verify that the output (F) of the circuit conforms to that of an OR gate. Record your observations in Table 2.3 below.

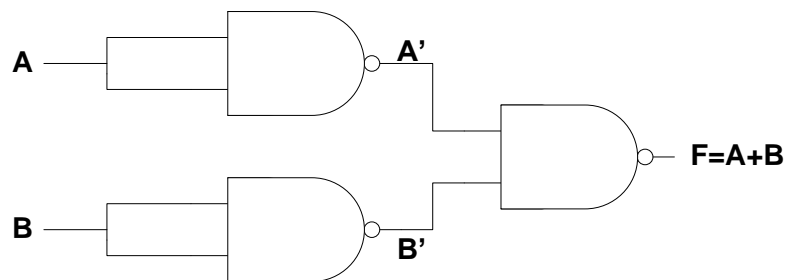


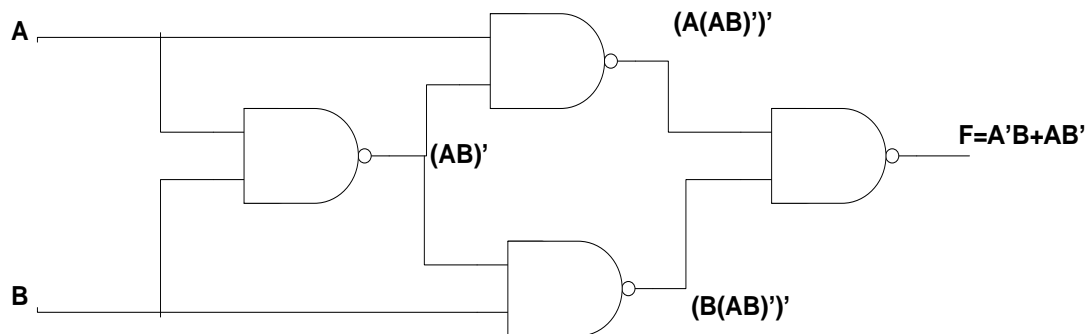
Figure 2.3: OR gate using NAND gates

Table 2.3: Observation Table for OR gate

INPUTS		OUTPUT
<i>A</i>	<i>B</i>	<i>F</i>
0	0	
0	1	
1	0	
1	1	

In-Lab Task 1.4: Verification of XOR function

- Connect the circuit as shown in Figure 2.4.
- Connect +5V to pin 14 (Vcc) and Ground to pin 7 (GND) of the IC.
- By setting the switches to 1 and 0, verify that the output (F) of the circuit conforms to that of an XOR gate. Record your observations in Table 2.4 below.

*Figure 2.4: XOR gate using NAND gates**Table 2.4: Observation Table for XOR gate*

INPUTS		OUTPUT
A	B	F
0	0	
0	1	
1	0	
1	1	

In-Lab Task 1.5: Verification of XNOR function

- Connect the circuit as shown in Figure 2.5.
- Connect +5V to pin 14 (Vcc) and Ground to pin 7 (GND) of the IC.
- By setting the switches to 1 and 0, verify that the output (F) of the circuit conforms to that of an XNOR gate. Record your observations in Table 2.5 below.

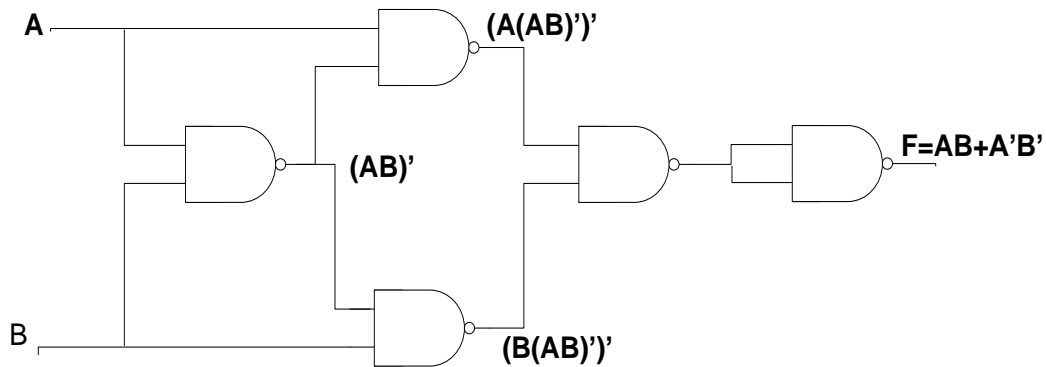


Figure 2.5: XNOR gate using NAND gates

Table 2.5: Observation Table for XNOR gate

INPUTS		OUTPUT
<i>A</i>	<i>B</i>	<i>F</i>
0	0	
0	1	
1	0	
1	1	

In-Lab Task 1.6: Implementation of any Boolean function (2-variables) using only NAND gates

$F(A, B) =$ _____

(Note: Boolean function will be specified by Lab Instructor)

Table 2.6: Observation Table for the given Boolean function

Inputs		Outputs	
<i>A</i>	<i>B</i>	Calculated <i>F_C</i>	Observed <i>F_O</i>
0	0		
0	1		
1	0		
1	1		

Part 2 - Implementing any logic expression by using only NOR gates

If we can show that the logical operations AND, OR, and NOT can be implemented with NOR gates, then it can be safely assumed that any Boolean function can be implemented with NOR gates.

Procedure

- Simulate NOT, AND and OR gates in Proteus software, by using only NOR gates. Verify their truth tables.
- Insert the IC on the trainer's breadboard.
- Use any one or more of the NOR gates of the IC for this experiment.
- One or more Logic Switches of the trainer (S1 to S9) can be used for input to the NOR gate.
- For output indication, connect the output pin of the circuit to any one of the LEDs of the trainer (L0 to L15).

In-Lab Tasks-Part-2

In-Lab Task 2.1: Verification of NOT function

- Connect the circuit as shown in Figure 2.6.
- Connect +5V to pin 14 (Vcc) and Ground to pin 7 (GND) of the IC.
- By setting the switches to 1 and 0, verify that the output (F) of the circuit conforms to that of an NOT gate. Record your observations in Table 2.7 below.



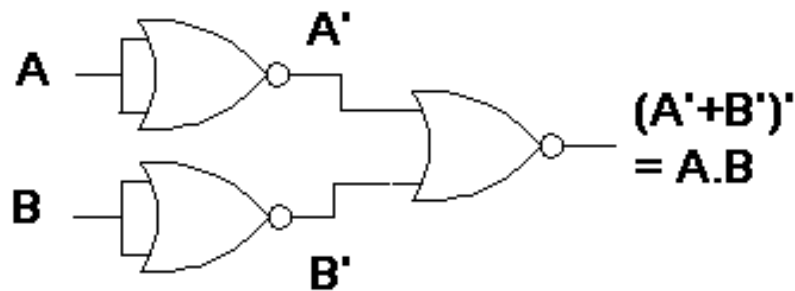
Figure 2.6: NOT gate using NOR gate

Table 2.7: Observation Table for NOT gate

INPUT	OUTPUT
<i>A</i>	<i>F</i>
0	
1	

In-Lab Task 2.2: Verification of AND function

- Connect the circuit as shown in Figure 2.7.
- Connect +5V to pin 14 (Vcc) and Ground to pin 7 (GND) of the IC.
- By setting the switches to 1 and 0, verify that the output (F) of the circuit conforms to that of an AND gate. Record your observations in Table 2.8 below.

*Figure 2.7: AND gate using NOR gates**Table 2.8: Observation Table for AND gate*

INPUTS		OUTPUT
A	B	F
0	0	
0	1	
1	0	
1	1	

In-Lab Task 2.3: Verification of OR function

- Connect the circuit as shown in Figure 2.8.
- Connect +5V to pin 14 (Vcc) and Ground to pin 7 (GND) of the IC.
- By setting the switches to 1 and 0, verify that the output (F) of the circuit conforms to that of an OR gate. Record your observations in Table 2.9 below.

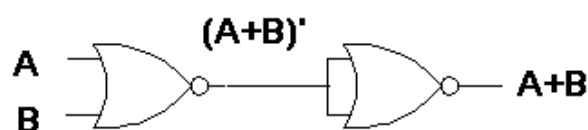
*Figure 2.8: OR gate using NOR gates*

Table 2.9: Observation Table for OR gate

INPUTS		OUTPUT
<i>A</i>	<i>B</i>	<i>F</i>
0	0	
0	1	
1	0	
1	1	

Post-Lab:

Task 01: Simulate NAND, XOR and XNOR gates in Proteus software, by using only NOR gates. Verify their truth tables.

Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

LAB #03: Introduction to Verilog and Simulation **using XILINX ISE**

Objective

Part 1

In this lab, Verilog (Hardware Description Language) is introduced with Xilinx ISE. Verilog is used to model digital systems. It is most commonly used in the design and verification of digital circuits.

Part 2

Xilinx ISE is a verification and simulation tool for Verilog, VHDL, System Verilog, and mixed- language designs.

Pre-Lab:

Background Theory:

Verilog HDL has evolved as a standard hardware description language. Verilog HDL offers many useful features:

- Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use.
- It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.
- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates/structural, RTL, dataflow or behavioral code.
- Also, a designer needs to learn only one language for stimulus and hierarchical design.
- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
- All fabrication vendors provide Verilog HDL libraries for postlogic synthesis simulation.
- Thus, designing a chip in Verilog HDL allows the widest choice of vendors.
- The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

3.1 Level of Abstraction provided by Verilog

3.1.1 Switch level

This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details.

3.1.2 Gate (Structural) level

The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level/structure-level logic diagram.

3.1.3 Dataflow level

At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

3.1.4 Behavioral or algorithmic level

This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.

3.2 Syntax

3.2.1 Comments

Verilog comments are the same as in C++. Use `//` for a single line comment or `/* ... */` for a multiline comment.

3.2.2 Punctuation

White spaces are ignored in Verilog. A semicolon is used to indicate the end of a command line and commas are typically used to separate elements in a list. Like C++, Verilog is case sensitive.

3.2.3 Identifiers

An identifier is usually a variable. You can use any letter, digit, the underscore, or `$`. Identifiers may not begin with a digit and may not be the same as a Verilog *keyword*. As in C++ variable names should be chosen to assist in documentation.

3.2.4 Signal values

Signals in Verilog have one of four values. These are 0 (logic 0), 1 (logic 1), ?, X, or x (don't care or unknown), and Z or z for high impedance tri-state.

3.2.5 Constants

The generic declaration for a constant in Verilog is

```
[size][ 'radix] constant_value
```

In this declaration size indicates the number of bits and 'radix gives the number base (d = decimal, b = binary, o = octal, h = hex). The default radix is decimal.

Examples:

```
16          //The number 16 base 10
4'b1010     //The binary number 1010
8'bx       //An 8-bit binary number of unknown value
12'habc     //The hex number abc = 1010 1011 1100 in binary
8'b10       //The binary number 0000 0010
```

3.3 Structure

3.3.1 Module

A module in Verilog is used to define a circuit or a subcircuit. The module is the fundamental circuit building block in Verilog. Modules have the following structure: Note that the module declaration ends with a semicolon but the keyword `endmodule` does not.

```
module module_name (port_name list);
[declarations]
[assign statements]
[initial block]
[always block]
[gate instantiations]
[other module instantiations]
endmodule
```

3.3.2 Ports

Ports in Verilog can be of type **input**, **output**, or **inout**. The module ports are given in the port name list and are declared in the beginning of the module. Here is a sample module with input and output ports.

```
module MyModule(yOut, aIn);  
output yOut;  
input aIn;  
  
...  
endmodule
```

The port names **input** and **output** default to type **wire**. Either can be a vector and the output variables can be re-declared to type **reg**. The output and input variables in a module are typical names for the output and input pins on the implementation chip.

3.3.3 Signals

A signal is represented by either a net type or a variable type in Verilog. The net type represents a circuit node, and these can be of several types. The two net types most often used are **wire** and **tri**. Type nets do not have to be declared in Verilog since Verilog assumes that all signals are nets unless they are declared otherwise. Variables are either of type **reg** or **integer**. Integers are always 32-bits where the reg type of variables may be of any length. Typically, we use integers as loop counters and reg variables for all other variables. The generic form for representing a signal in Verilog is:

type [range] signal_name

The net types are typically used for input signals and for intermediate signals within combinational logic. Variables are used for sequential circuits or for outputs which are assigned a value within a sequential always block.

Examples:

```
wire w;                //w is a single net of type wire  
wire [2:0] wVect;      //Declares wVect[2], wVect[1], wVect[0]  
tri [7:0] bus;         //An 8-bit tri state bus  
integer i;             //i is a 32-bit integer  
reg r;                //r is a 1-bit register  
reg [7:0] buf;         //buf is an 8-bit register  
reg [3:0] r1, r2;      //r1 and r2 are both 4-bit registers
```

3.3.4 Most often used Operators

	Symbol	Comments
Bitwise	~	Ones complement unary operator
	&	AND
		OR
	^	XOR
	~&	NAND
	~	NOR
	~^	XNOR
Logical	!	Not unary operator also called logical negation
	&&	Logical AND
		Logical OR
Arithmetic	+	Add
	-	Subtract
	*	Multiply
	/	Divide
	%	Mod operator
Relational	>	Greater than
	<	Less than
	>=	Greater than or equals
	<=	Less than or equals
Misc	>>	Shift right
	<<	Shift left
	?:	(cond)?(statements if true):(statements if false)
	{ , }	Concatenation
	{ m { } }	Repetition where m is repetition number

In-Lab:

3.4 Xilinx ISE 13.2

Start ISE from the Start menu by selecting:

Start → All Programs → Xilinx ISE Design Suite 13.2 → ISE Design Tool → Project Navigator or by double-clicking on Xilinx ISE Design Suite 13.2 icon

3.4.1 Creating a Project

- File → New Project (Figure 3.1)

- Select the location for the project (Should be your dld_lab folder)
- Enter the name of the project
- Then click next

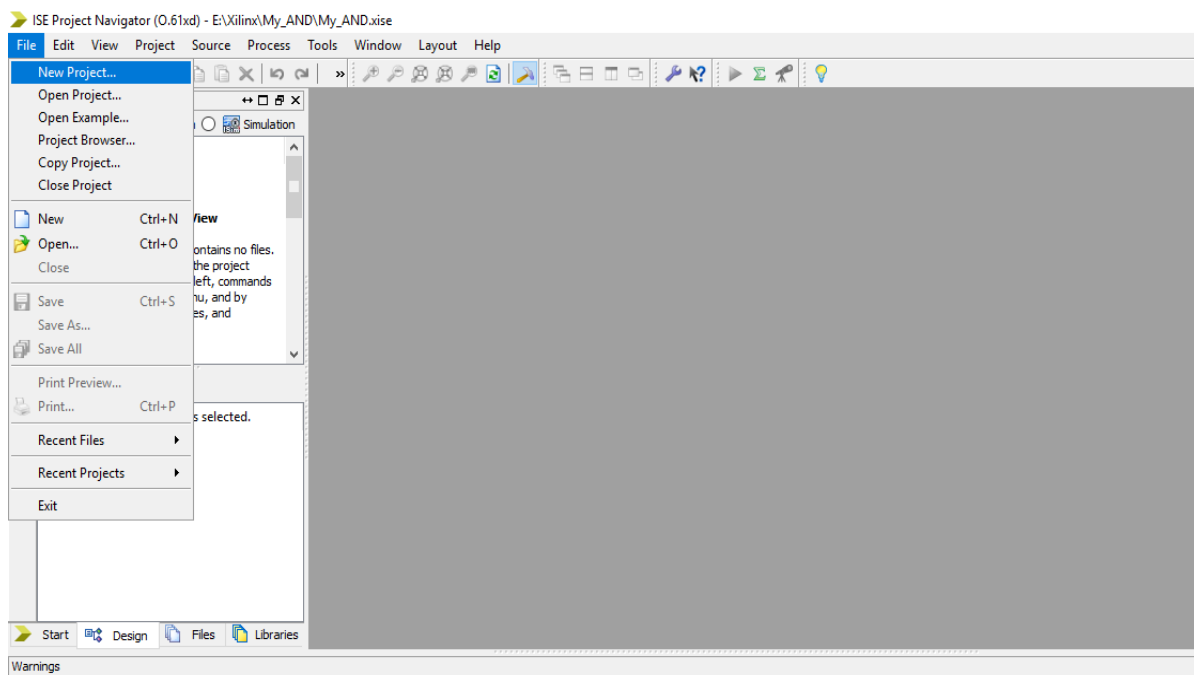


Figure 3.1: XILINX ISE 13.2 Interface

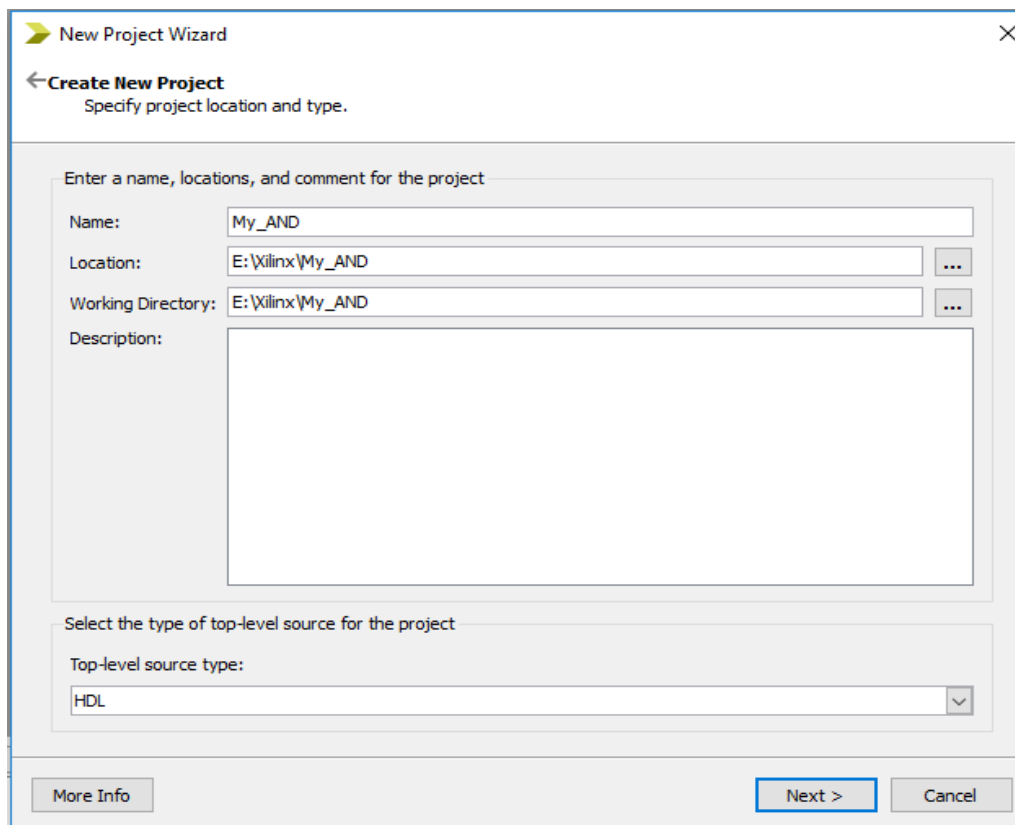


Figure 3.2: Create New project window

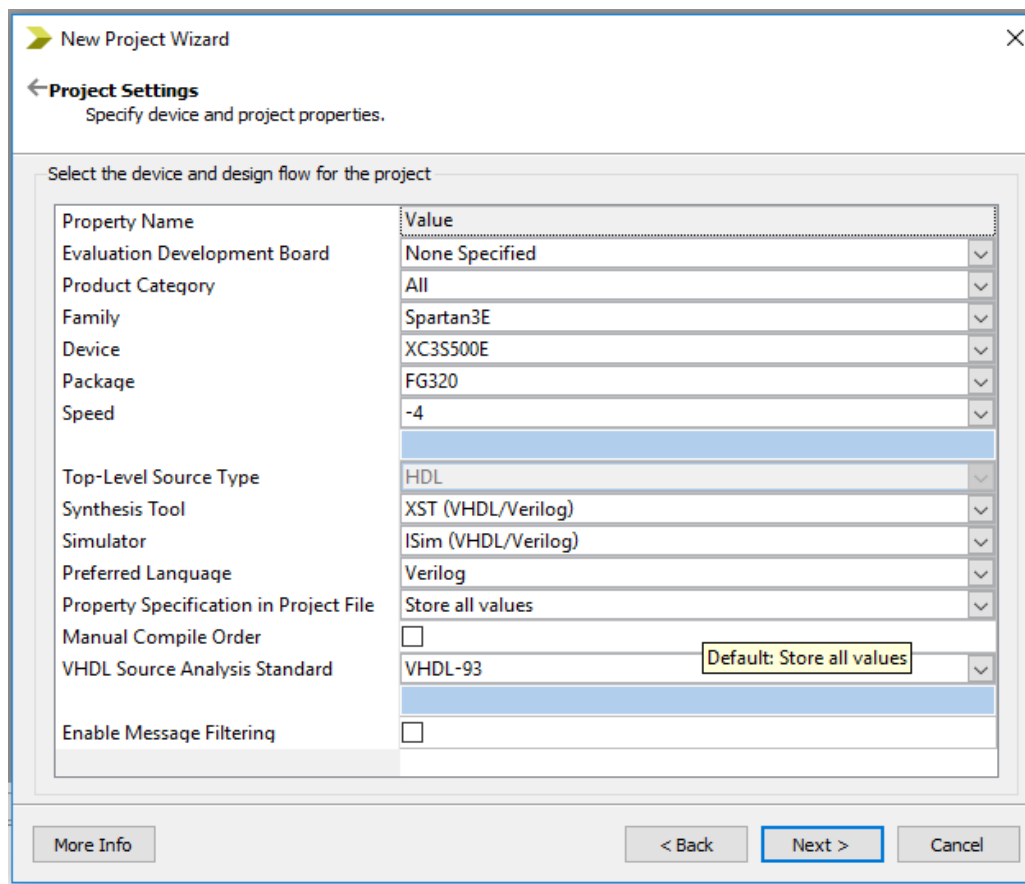


Figure 3.3: Project Settings Window

- Set the project settings:
 - Family to Spartan3E
 - Device to XC3S500E
 - Package to FG320
 - Speed to -4
- Then click to Next
- Project Summary
- Then click finish

3.4.2 Adding a file to the project

Right click on Design Pane and select a New Source or Add Source (If file already exists).

- For New Source:
- Select Source Type (e.g. Verilog Module) and enters the file name
- Then click Next
- Give Port Names and direction of the module
- Then click Next
- Then click Finish

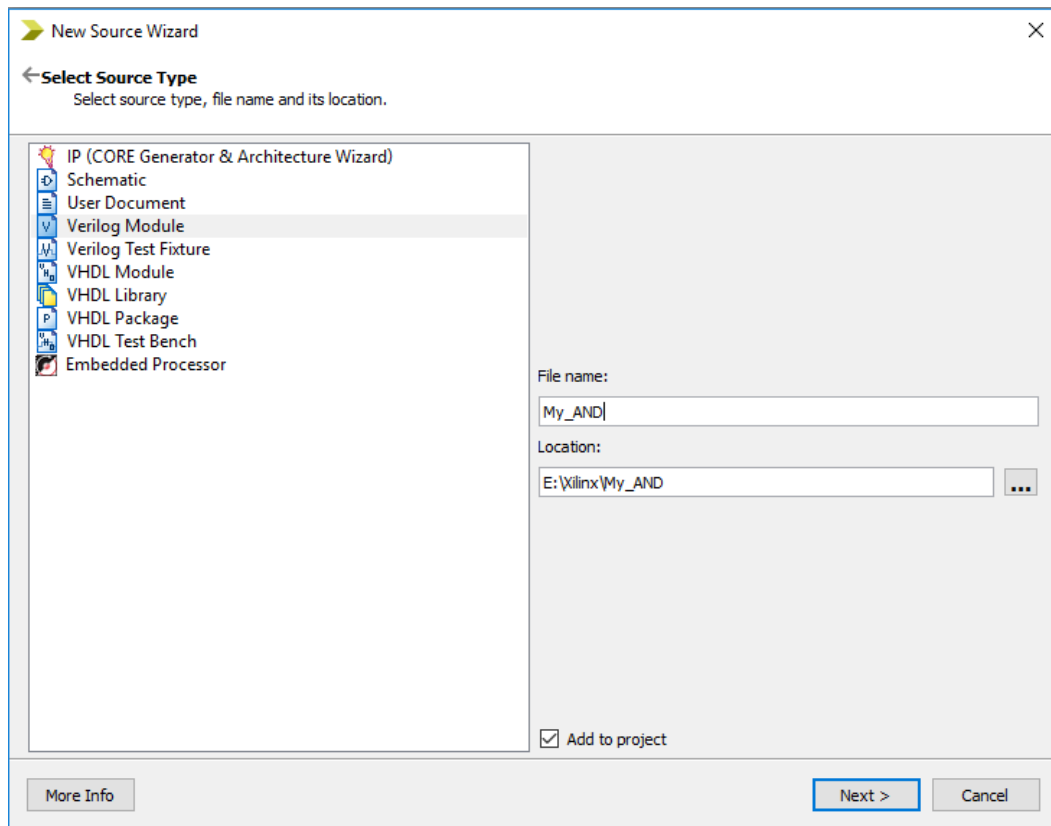


Figure 3.4: New Source Wizard window

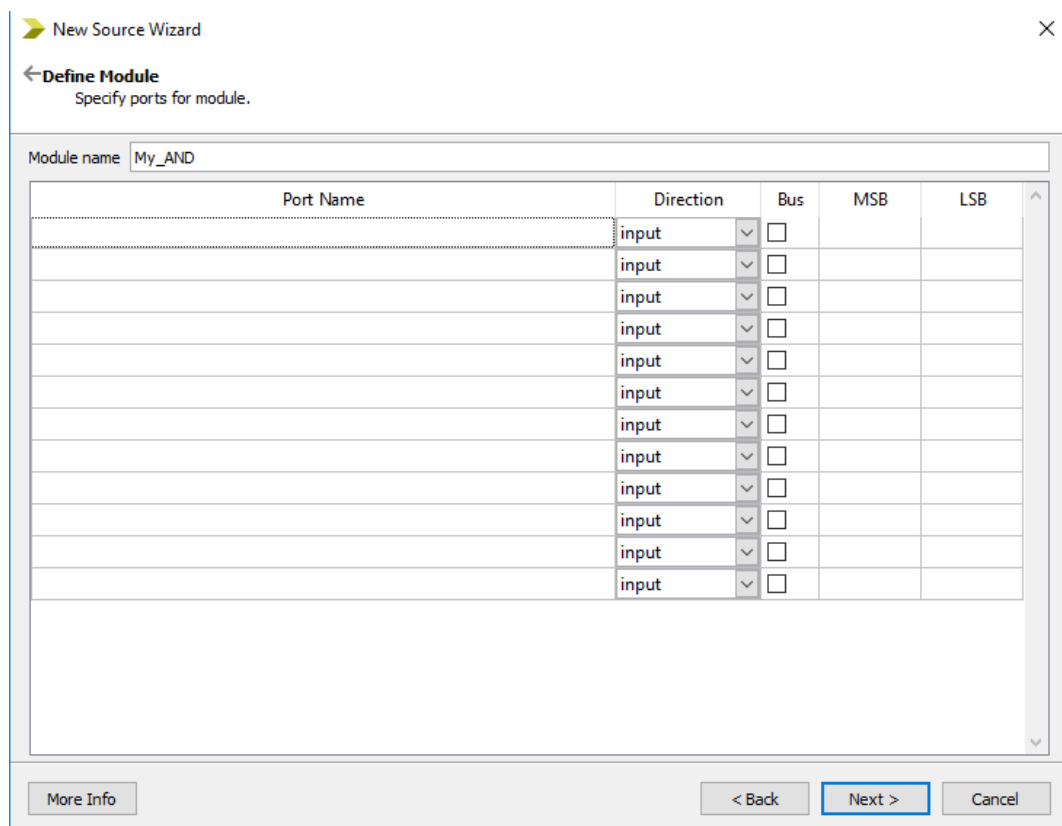


Figure 3.5: Define Module window

- For Add Source:
- Right click on Design Pane and select an Add Source
- Browse the existing HDL file

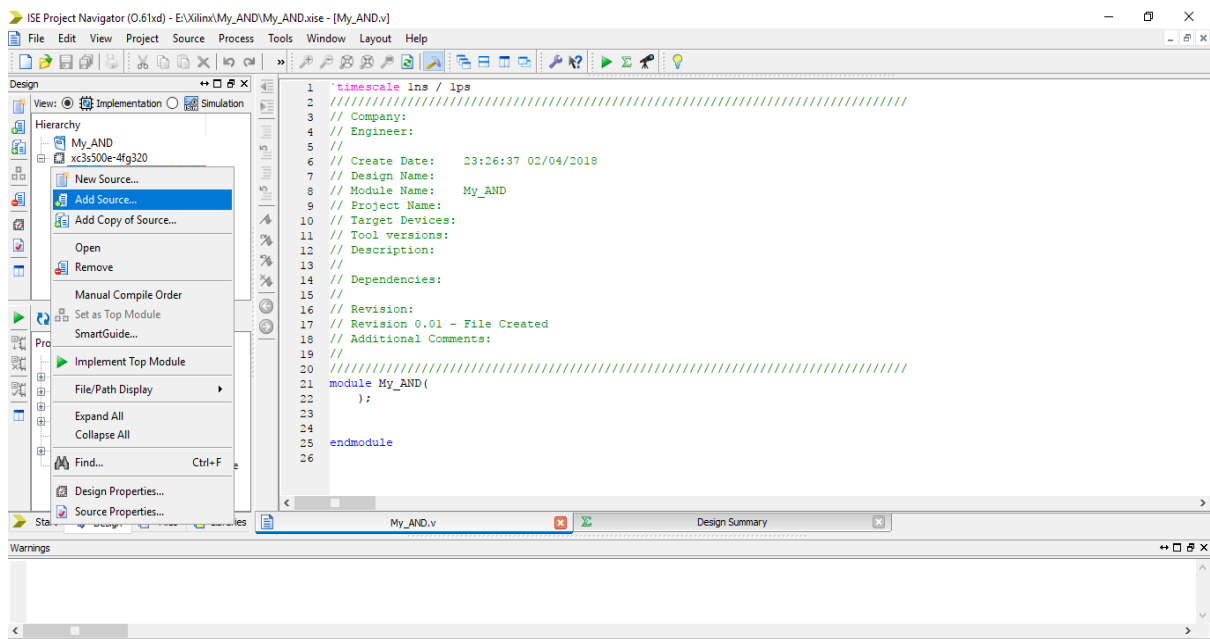


Figure 3.6: Adding a Source to the project

Example: Gate-Level

```
module My_AND(yOut, aIn, bIn);
output yOut;
input aIn, bIn;

and G1(yOut,aIn,bIn);    //AND gate instantiation

endmodule
```

Example: Dataflow

```
module My_AND(yOut, aIn, bIn);
output yOut;
input aIn, bIn;

assign yOut = aIn & bIn; //AND gate in dataflow description

endmodule
```

3.5 Simulation result

- Click on Simulation icon as shown in Figure 3.7.

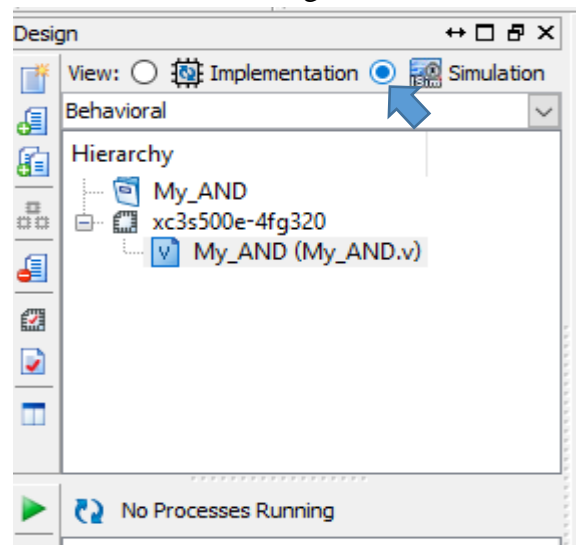


Figure 3.7: Simulation Window

3.5.1 Add test bench file

- Select Source Type (e.g. Verilog test fixture)
- Enter the file name (e.g. TB_Module_Name)
- Then click Next
- Then click Next
- Then click Finish
- Now add various value of input ports (e.g., A and B) as in Figure 3.8

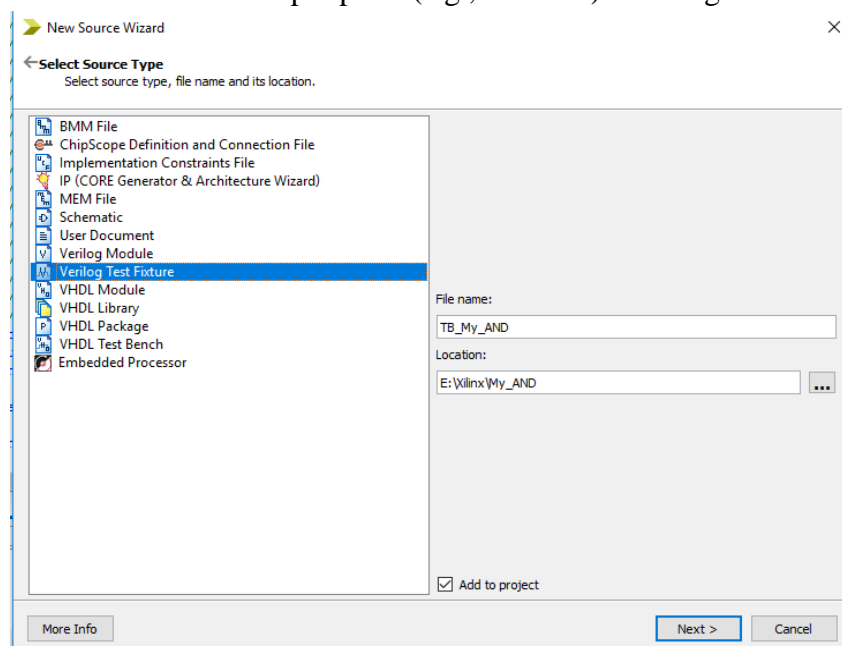


Figure 3.8: Add new source file as a Verilog Test Fixture

Stimulus Example:

```
module TB_My_AND;    // Test bench has no inputs and outputs

    // Inputs
    reg aIn;
    reg bIn;

    // Outputs
    wire yOut;

    // Instantiate the Unit Under Test (UUT)
    My_AND uut (yOut,aIn,bIn);

    initial begin
        // Initialize Inputs
        aIn = 0;      bIn = 0;

        #10; // Wait 10 ns
        aIn = 0;      bIn = 1;

        #10; // Wait 10 ns
        aIn = 1;      bIn = 0;

        #10; // Wait 10 ns
        aIn = 1;      bIn = 1;

        #10 $finish;           //To finish the simulation
    end
endmodule
```

3.5.2 ISE Simulator

- Click on ISE simulator it will open two options as shown in Figure 3.9
- Then double-click on Behavior check syntax and wait to verify by tick sign
- Then double-click on simulate behavior model, which will open a new window

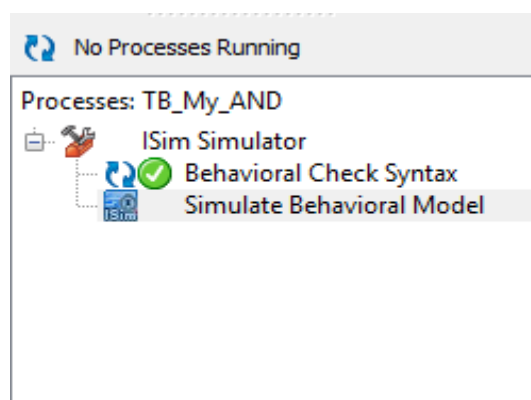


Figure 3.9: ISE Simulator for simulation

3.5.3 Simulation windows

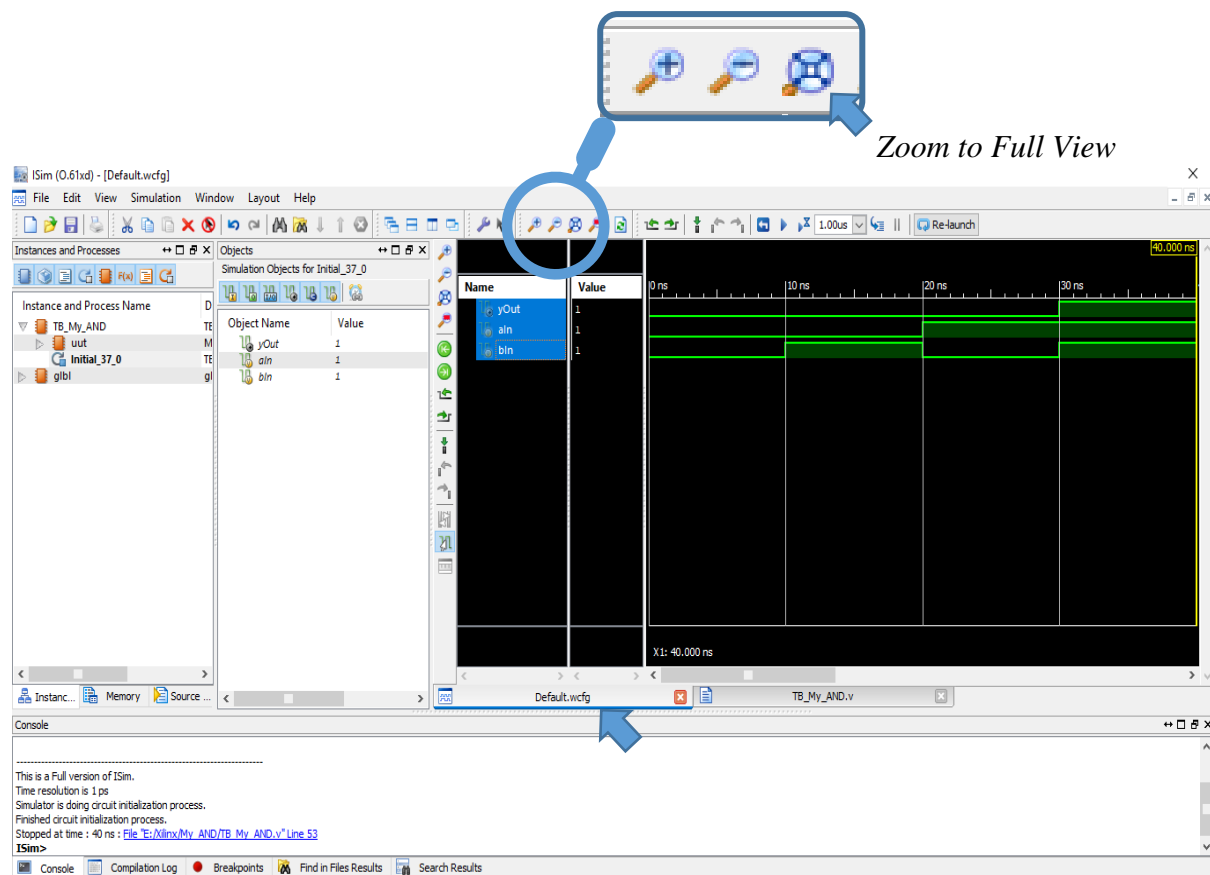


Figure 3.10: ISIM Simulation WAVE window

In-Lab Task 2:

Verify all the basic logic gates using the Xilinx ISE simulation tool and verify your waveform with logic gates truth table.

Task 01: Write a Verilog code (Gate-Level) for NOT, OR, NOR, NAND, XOR and XNOR.

Task 02: Write a stimulus/test bench for Task 01 and show the simulation results.

Post-Lab:

Task 01: Write a Verilog code for the given Boolean function* (e.g. $F = x + \bar{x}y + y\bar{z}$):

- Using Gate-Level model (Provide Gate Level diagram and Truth Table)
- Using Dataflow model

Task 02: Write a stimulus/test bench for Task 01 and show the simulation results.

* (Note: Every student should opt different Boolean function)

Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

LAB #04: Design and Implementation of Boolean Functions by Standard Forms using ICs/Verilog

Objective

In this lab, we implement Boolean functions by using SoP (sums of product) and PoS (products of sum).

Equipment Required

KL-31001 Digital Logic Lab Trainer, Breadboard, Logic gate ICs (NAND & NOR).

Pre-Lab:

Boolean Algebra

Boolean algebra is algebra for the manipulation of objects that can take on only two values, typically true and false.

- It is common to interpret the digital value:
 - 0 as false
 - 1 as true

Boolean Function

A Boolean function typically has one or more input values and yields a result, based on these input value, in the range $\{0, 1\}$. A Boolean operator can be completely described using a table that lists inputs, all possible values for these inputs, and the resulting values of the operation.

A Boolean function can be represented in a truth table and it can be transformed from an algebraic expression into a circuit diagram composed of logic gates. In evaluating Boolean equations AND operation is performed before OR operation unless OR operation is enclosed with in brackets.

Basic Theorems

Table 4.1: Basic Boolean algebra theorems

Identity Name	AND form	OR form
Identity Law	$x \cdot 1 = x$	$x + 0 = x$
Null (or Dominance) Law	$x \cdot 0 = 0$	$x + 1 = 1$
Idempotent Law	$x \cdot x = x$	$x + 1 = x$
Inverse Law	$x \cdot \bar{x} = 0$	$x + \bar{x} = 1$
Commutative Law	$x \cdot y = y \cdot x$	$x + y = y + x$
Associative Law	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$(x + y) + z = x + (y + z)$
Distributive Law	$x + (y \cdot z) = (x + y) \cdot (x + z)$	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
Absorption Law	$x \cdot (x + y) = x$	$x + (x \cdot y) = x$
DeMorgan's Law	$\overline{(x \cdot y)} = \bar{x} + \bar{y}$	$\overline{(x + y)} = \bar{x} \cdot \bar{y}$
Double Complement Law	$\bar{\bar{x}} = x$	

Canonical Forms

Any Boolean function that is expressed as a sum of *minterms* or as a product of *maxterms* is said to be in its canonical form.

In general, the unique algebraic expression for any Boolean function can be obtained from its truth table by using an OR operator to combine all *minterms* for which the function is equal to 1.

Minterm

A *minterm* denoted as m_i , where $0 \leq i < 2^n$, is a product (AND) of the n variables (literals) in which each variable is complemented if the value assigned to it is 0, and uncomplemented if it is 1.

- 1-*minterms* = *minterms* for which the function $F = 1$.
- 0-*minterms* = *minterms* for which the function $F = 0$.
- Any Boolean function can be expressed as a sum (OR) of its 1-*minterms*.

A shorthand notation:

$$F(\text{list of variables}) = \sum (\text{list of 1-minterm indices})$$

- The inverse of the function can be expressed as a sum (OR) of its 0-minterms.

A shorthand notation:

$$\bar{F}(\text{list of variables}) = \sum (\text{list of 0-minterm indices})$$

Example:

Table 4.2: 3-variables Minterm example

x	y	z	Minterms	F	\bar{F}
0	0	0	$m_0 = x'y'z'$	0	1
0	0	1	$m_1 = x'y'z$	0	1
0	1	0	$m_2 = x'yz'$	0	1
0	1	1	$m_3 = x'yz$	1	0
1	0	0	$m_4 = xy'z'$	0	1
1	0	1	$m_5 = xy'z$	1	0
1	1	0	$m_6 = xyz'$	1	0
1	1	1	$m_7 = xyz$	1	0

$$F = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$$

or

$$F(x, y, z) = \sum (3, 5, 6, 7)$$

Similarly,

$$\bar{F} = x'y'z' + x'y'z + x'yz' + xy'z' = m_0 + m_1 + m_2 + m_4$$

or

$$\bar{F}(x, y, z) = \sum (0, 1, 2, 4)$$

Maxterm

A *maxterm* denoted as M_i , where $0 \leq i < 2n$, is a sum (OR) of the n variables (literals) in which each variable is complemented if the value assigned to it is 1, and un complemented if it is 0.

- 0-*maxterms* = *maxterms* for which the function $F = 0$.
- 1-*maxterms* = *maxterms* for which the function $F = 1$.
- Any Boolean function can be expressed as a product (AND) of its 0-*maxterms*.

A shorthand notation:

$$F(\text{list of variables}) = \prod (\text{list of 0-maxterm indices})$$

- The inverse of the function can be expressed as a product (AND) of its 1-*maxterms*.

A shorthand notation:

$$\bar{F}(\text{list of variables}) = \prod (\text{list of 1-maxterm indices})$$

Example:

Table 4.3: 3-variables Maxterm example

x	y	z	Maxterms	F	\bar{F}
0	0	0	$M_0 = x + y + z$	0	1
0	0	1	$M_1 = x + y + z'$	0	1
0	1	0	$M_2 = x + y' + z$	0	1
0	1	1	$M_3 = x + y' + z'$	1	0
1	0	0	$M_4 = x' + y + z$	0	1
1	0	1	$M_5 = x' + y + z'$	1	0
1	1	0	$M_6 = x' + y' + z$	1	0
1	1	1	$M_7 = x' + y' + z'$	1	0

$$F = (x + y + z). (x + y + z'). (x + y' + z). (x' + y + z) = M_0 . M_1 . M_2 . M_4$$

or

$$F(x, y, z) = \prod (0, 1, 2, 4)$$

Similarly,

$$\bar{F} = (x + y' + z'). (x' + y + z'). (x' + y' + z). (x' + y' + z') = M_3 + M_5 + M_6 + M_7$$

or

$$\bar{F}(x, y, z) = \prod (3, 5, 6, 7)$$

Standard Forms

SoP (Sum of Products):

The term "**Sum of Products**" or "**SoP**" is widely used for the canonical form that is a disjunction (OR) of *minterms*.

PoS (Product of Sum):

The term "**Product of Sums**" or "**PoS**" for the canonical form that is a conjunction (AND) of *maxterms*.

Pre-Lab Tasks:

1. Express the Boolean function $F = x + yz$ as a sum of *minterms* by using truth table.
2. Express $F' = (x + yz)'$ as a product of maxterms.
3. Given the function as defined in the truth table (Table 4.4), express F using sum of *minterms* and product of *maxterms*.

Table 4.4: Truth Table for F (Pre-Lab Task 3)

x	y	z	Minterms	Maxterms	F	\bar{F}
0	0	0	$m_0 = x'y'z'$	$M_0 = x + y + z$	0	1
0	0	1	$m_1 = x'y'z$	$M_1 = x + y + z'$	1	0
0	1	0	$m_2 = x'yz'$	$M_2 = x + y' + z$	0	1
0	1	1	$m_3 = x'yz$	$M_3 = x + y' + z'$	1	0
1	0	0	$m_4 = xy'z'$	$M_4 = x' + y + z$	1	0
1	0	1	$m_5 = xy'z$	$M_5 = x' + y + z'$	0	1
1	1	0	$m_6 = xyz'$	$M_6 = x' + y' + z$	0	1
1	1	1	$m_7 = xyz$	$M_7 = x' + y' + z'$	0	1

In-Lab Tasks:

Circuit Implementation

1. First, make the circuit diagram of the given Task.
2. Select appropriate logic gate ICs which are needed.
3. Make connections according to the circuit diagram you made.
4. Connect the input to data switches and output to the logic indicator.
5. Follow the input sequence and record the output.

TASK: Implement the circuit for the given function “ F ”. Function’s output is given in Table 4.5. Finds its Boolean expression in SoP and PoS forms.

Table 4.5: Truth Table for F (In-Lab Task)

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Boolean Equations:

Sum of Min-terms equation of F: _____

Reduced SOP form equation of F: _____

Product of Max-terms equation of F: _____

Reduced POS form equation of F: _____

Reduced form calculation:



Circuit Diagrams:

1. Sum of Min-terms form:



2. Reduced SoP form: (Implement circuit by NAND IC(s)):

3. Reduced PoS form: (Implement circuit by using NOR IC(s))

Table 4.6: Observation Table for In-Lab Task

A	B	C	D	F	Observed Outputs		
					F₁	F₂	F₃
0	0	0	0	0			
0	0	0	1	0			
0	0	1	0	0			
0	0	1	1	1			
0	1	0	0	0			
0	1	0	1	0			
0	1	1	0	0			
0	1	1	1	1			
1	0	0	0	1			
1	0	0	1	0			
1	0	1	0	0			
1	0	1	1	0			
1	1	0	0	0			
1	1	0	1	1			
1	1	1	0	1			
1	1	1	1	1			

F₁: Output of sum of Min-terms form circuit (Simulate on Proteus).

F₂: Output of reduced SoP form circuit implemented using NAND gates (can use inverter gates).

F₃: Output of reduced PoS form circuit implemented using NOR gates (can use inverter gates).

Post-Lab Tasks:

1. Write a Verilog code for the sum of *minterms* circuit, F_1 , (Structural Level).
2. Write a Verilog code for the reduced SoP circuit, F_2 , (Structural Level).
3. Write a Verilog code for the reduced PoS circuit F_3 , (Structural Level).
4. Simulate and verify the outputs by making an appropriate stimulus for the above modules.

Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

LAB #05: Logic Minimization of Complex Functions **using Automated Tools**

Objective

Part 1

In this lab, students will learn Karnaugh Map minimization and how to use logic minimization automated tools for an excessive number of variables in a function.

Part 2

Minimized logic function results are verified by using Verilog Structural Level (Gate-Level) description on Xilinx ISE Design tool.

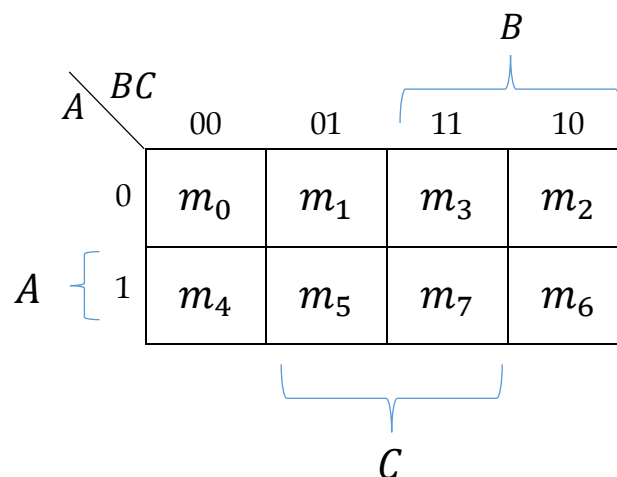
Pre-Lab:

Background Theory:

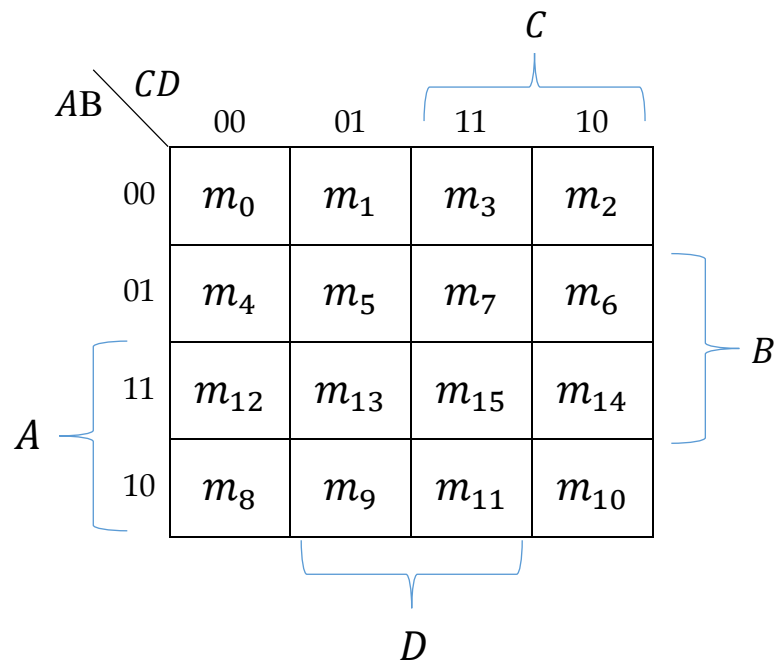
We can simplify the equations and find a Boolean function of any truth table using K-Map.

K-Maps:

- **3-Variables:**



- 4-Variables:



In-Lab:

Part 1: Automated Tool (Karnaugh Map Minimizer) (for variable >4)

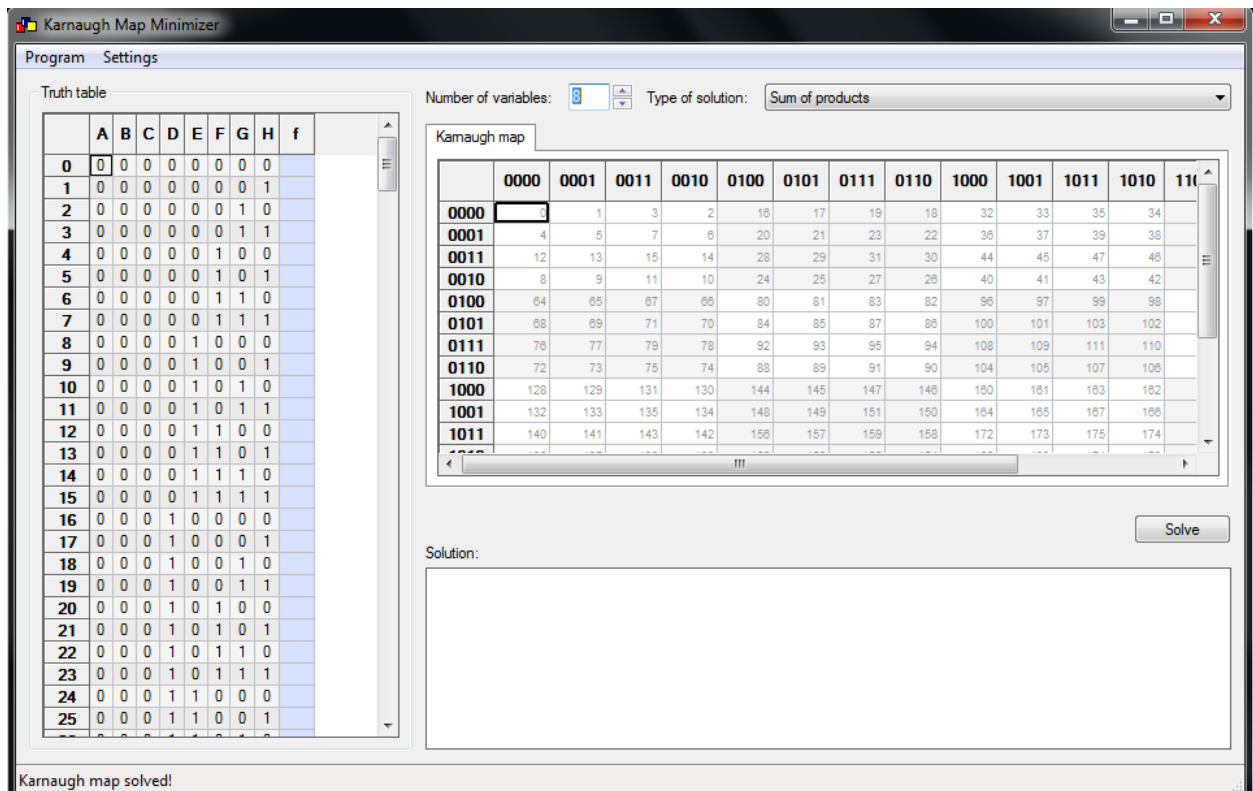


Figure 5.1: Karnaugh Map Minimizer Automated Tool Interface

Equipment/Tools Required:

KL-31001 Digital Logic Lab, Breadboard, Logic gate ICs, K-map Minimizer automated tool (Figure 5.1).

Procedure:

1. First, make the circuit diagram for the desired task or subtask given by the Lab instructor.
2. Using k-map minimizer tool derive the simplified function:
 - a. Set the number of variables
 - b. Set the type of solution
 - c. Set the values of function 'F'. (Function values can be generated randomly) *
 - d. Click on solve button then the solution will come on solution screen.
3. Implement the circuit using logic gate ICs (which is/are needed) for the tasks.
4. Make connections according to the circuit diagram you made.
5. Connect the inputs to data switches and output to the logic indicator.
6. Follow the input sequence and record the outputs in Table 5.1.

In-Lab Task 1:

Implement the minimized function given below using logic gate IC(s).

$$F(A, B, C, D) = \sum (\quad , \quad , \quad , \quad)$$

(Note: minterms will be specified by Lab Instructor)

Function in sum of Min-terms form: _____

Function in a simplified form using K-map: _____

Simplified calculation (K-map):

Circuit Diagram of a min-terms form of the Function:

Number of gates/ICs used: _____

Circuit Diagram of a simplified Function:

Number of gates/ICs used: _____

Truth Table:

Table 5.1: Observation Table for In-Lab Task

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>F</i>	Observed Outputs	
					<i>F</i> ₁	<i>F</i> ₂
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			

*F*₁: Output of sum of Min-terms form circuit.

*F*₂: Output of simplified function circuit.

Part 2: Verilog Design Task

Tools Required

K-map minimizer tool, Xilinx ISE Design tool.

In-lab Task 2:

Using structural model, write a Verilog description for the 8-variable function ' F ':

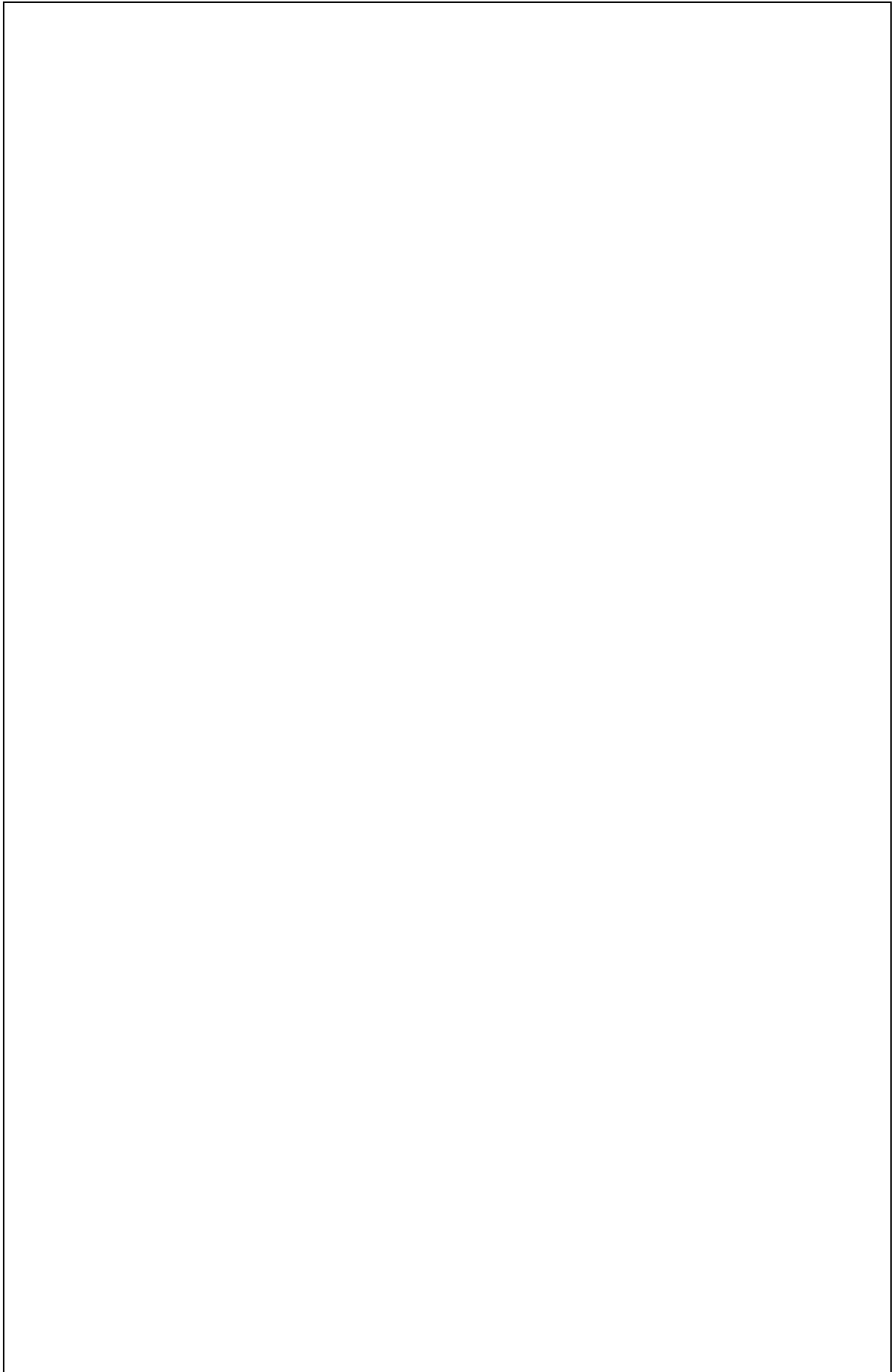
$$F(A, B, C, D, E, F, G, H) = \sum (\quad , \quad , \quad , \quad)$$

Procedure

- Function ' F ' should be generated by Karnaugh Map Optimizer automated tool
- Function ' F ' should be random
- Simulate and verify the output by making an appropriate stimulus on Xilinx ISE tool

Results

(Verilog Code & Simulation Wave Forms)

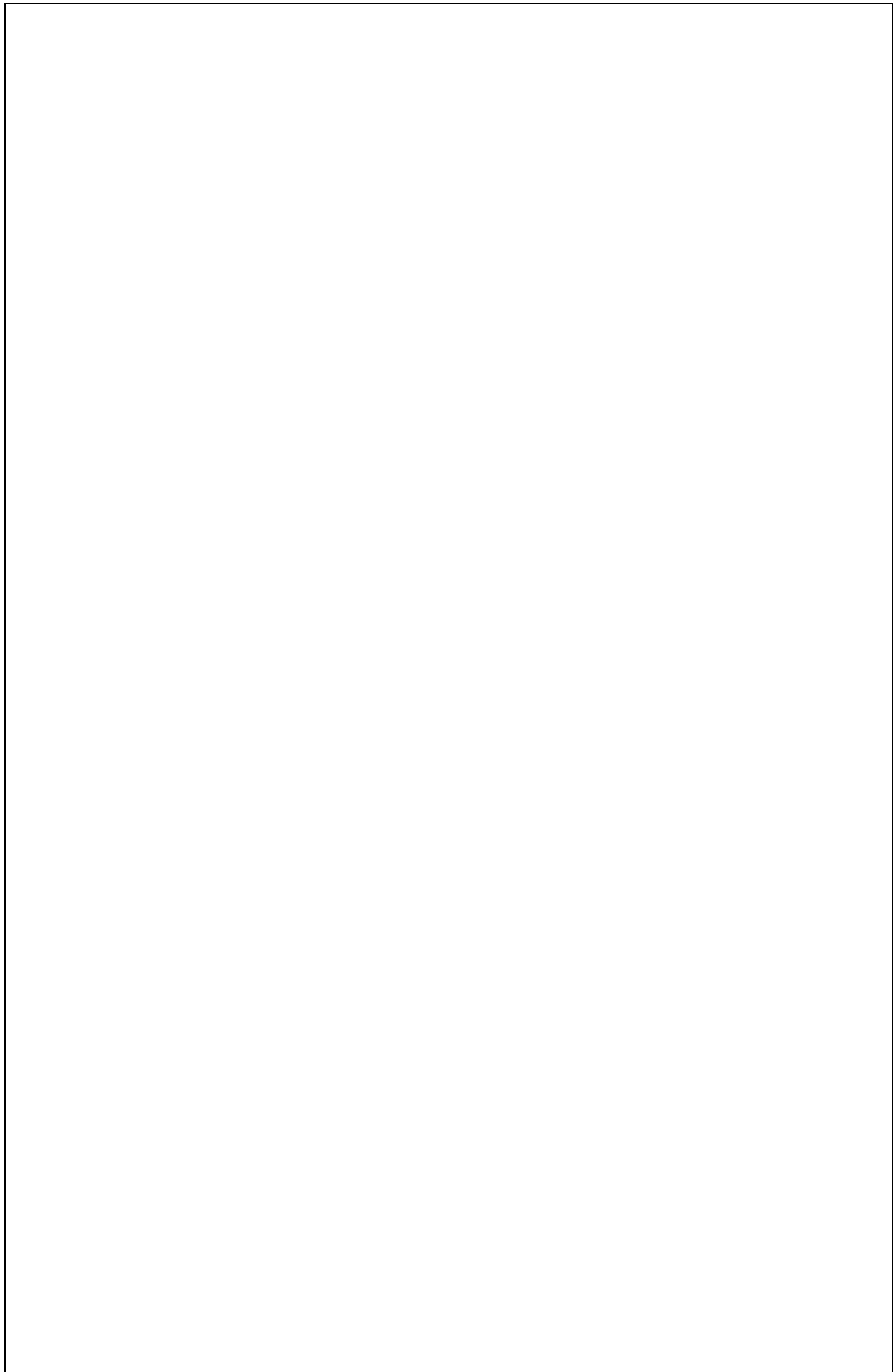


Post-Lab Tasks:

1. Using dataflow model, write a Verilog description for the 8-variable function ' F ' (used in “In-Lab Task 2”):
2. Simulate and verify the output by making an appropriate stimulus on Xilinx ISE tool

Results

(Verilog Code & Simulation Wave Forms)



Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

LAB #06: Xilinx ISE Design Flow with FPGA

Objective

Part 1

In this lab, we learn about the **Xilinx** software design flow with **FPGA**. Introduction of Behavioral modelling with Verilog is also included.

Part 2

Implement different examples on **FPGA**, e.g., Implementation of Gray code on FPGA evaluation Kit (Diligent Nexys2 Board).

Pre-Lab:

Background Theory:

In the coding, when numbers, letters or words are represented by a specific group of symbols, it is said that the number, letter or word is being encoded. The group of symbols is called as a code. The digital data is represented, stored and transmitted as group of binary bits. This group is also called as binary code. The binary code is represented by the number as well as alphanumeric letter e.g., Binary Coded Decimal (BCD), 2421 code, Excess-3 and Gray code.

Gray Code

It is the non-weighted code and it is not arithmetic code. That means there are no specific weights assigned to the bit position. It has a very special feature that, only one bit will change each time the decimal number is incremented as shown in Table 6.1. As only one bit changes at a time, the gray code is called as a unit distance code. The gray code is a cyclic code. Gray code cannot be used for arithmetic operation.

The Gray code is used in many applications in which the normal sequence of binary numbers may produce an ambiguity or error during the transition from one number to the next. For example, from 011 to 100 (which bit will change first?) may produce an intermediate erroneous number 101 if the value of least significant bit takes longer time to change than the other two bits value change. As only one bit changes its value during any transition between two number in Gray code, therefore, this problem is eliminated by the usage of Gray code.

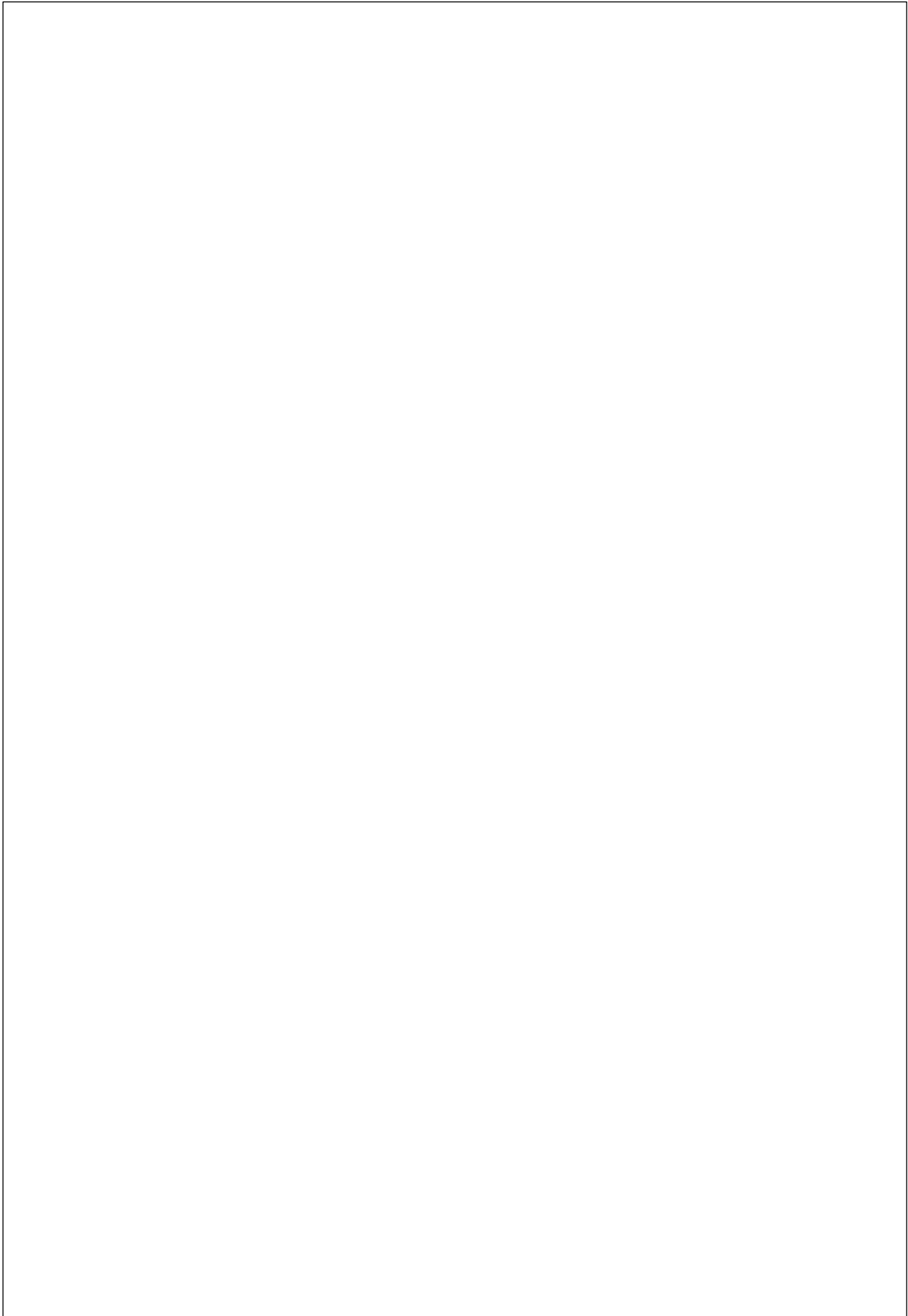
Decimal Equivalent Value in Binary				Gray Code Equivalent			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Pre-Lab Task:

1. Design a code converter that converts a 4-bit binary number to a 4-bit Gray code.
 - a. Provide K-Map solution and Boolean expressions
 - b. Provide gate-level circuit diagram

K-map and Boolean expressions Calculation:

Gate-Level Circuit Diagram of a simplified Function:



In-Lab:

Part 1: Behavioral Modelling

Behavioral modelling represents digital circuits at a functional and algorithmic level. Behavioral descriptions use the keyword **always**, followed by an optional event control expression and a list of procedural assignment statements. The behavior of a digital circuit is described in the body of the **always** block.

```
always @ (sensitivity list)
begin
    //body of an always block
end
```

The sensitivity list specifies which signals should trigger the elements inside the always block to be updated. In combinational circuits, normally all the inputs are the part of sensitivity list. Also “*” symbol is used in the sensitivity list, which includes all the inputs in the sensitivity list. However, for synchronous sequential circuits the sensitivity list consists of edge triggered clock and reset.

Example: Behavioral description of two-to-one line multiplexer

```
module mux_2_1 (Data_out,Ch_A,Ch_B,Sel);

    output reg Data_out;
    input Ch_A,Ch_B,Sel;

    always @ (Ch_A or Ch_B or Sel)
    begin
        if (Sel == 1'b0)
            Data_out = Ch_A;
        else
            Data_out = Ch_B;
        end
    end
endmodule
```

Part 2: Implementation of digital circuits on FPGA

- Create a new project or open an existing project.
- Add a new source file or add an existing file in case of a new project.
- Select the design in implementation mode.
- Then follow these steps:

6.1 Synthesizing a Code

- Double click on Synthesize- XST

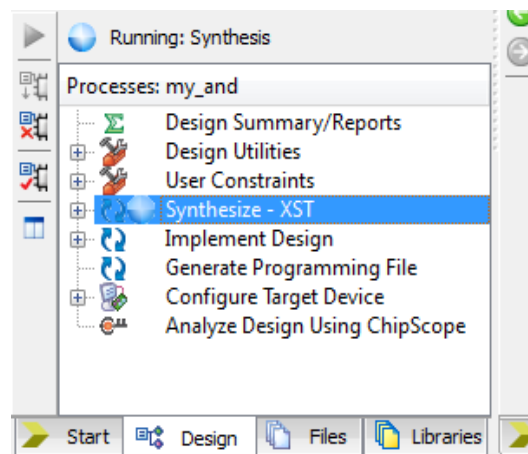


Figure 6.1: Synthesize-XST option Window

- Once it is done (3 possible outcomes will be shown):
 - ✓ : Synthesized-XST completed successfully
 - ✗ : Synthesized-XST failed due to some error(s)
 - ⚠ : Synthesized-XST completed with some warning(s)

6.2 Selecting the User Constraints

- Expanding the User Constraints menu
- Click on I/O Pin Planning (Plan Ahead)

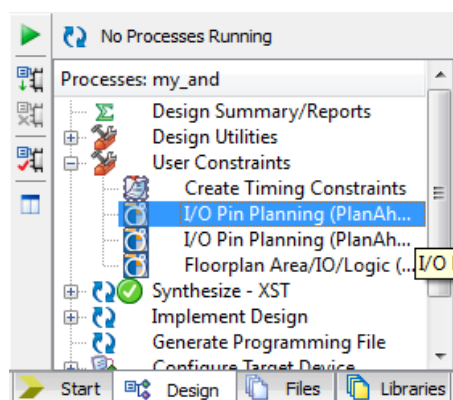


Figure 6.2: User Constraints Option Window

- Click Yes for UCF file
- Plan Ahead 13.2 window will open

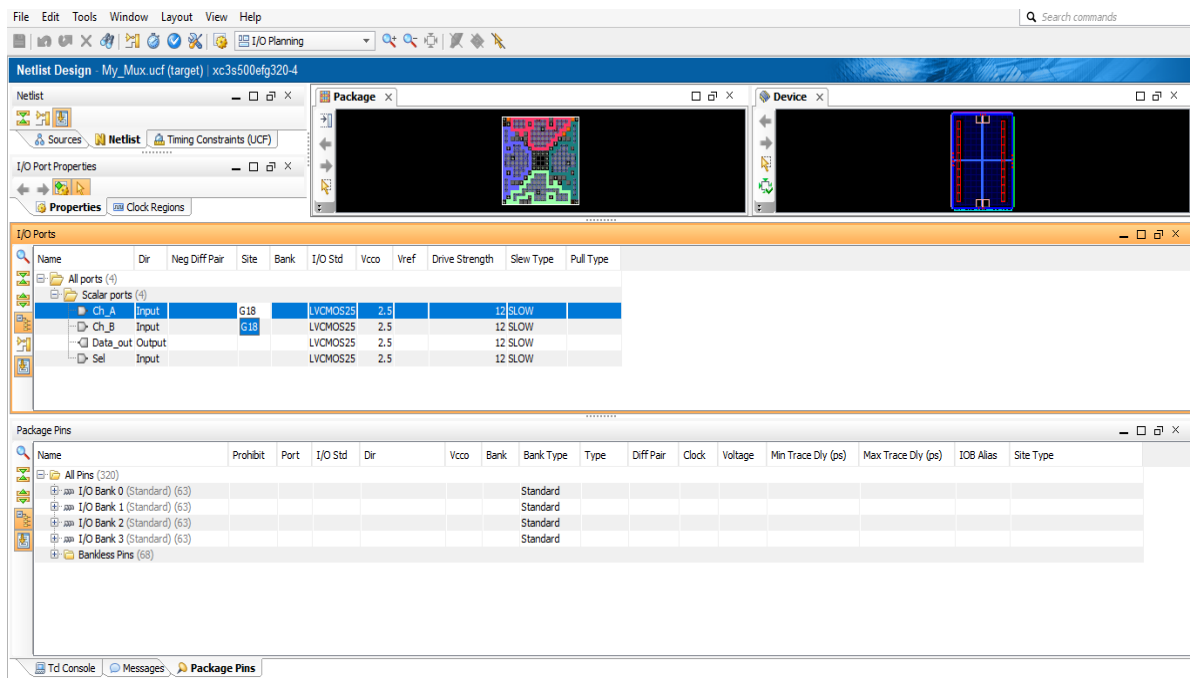


Figure 6.3: Plan Ahead 13.2 Interface

- Set the site for inputs and output(s) for example:
 - Output 'Data_out' to Led0 (select J14)
 - Input "Ch_A" to slide switch "SW0" (select G18)
 - Input "Ch_B" to slide switch "SW1" (select H18)
 - Input "Sel" to slide switch "SW2" (select K18)
- Save the design and exit

6.3 Implementing the Design

- Double click on Implement Design

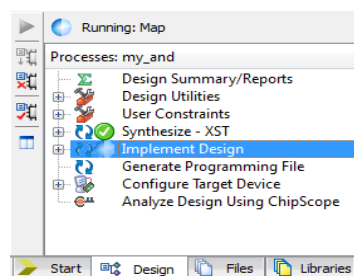


Figure 6.4: Implement Design Option Window

6.3.1 Generating a Programming File

- Double click on Generate Programming File

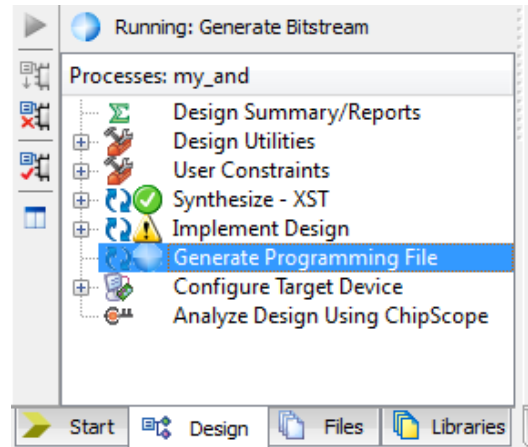


Figure 6.5: Generate Programming File Option Window

- Bit file will be generated after the completion of 'Generate Programming File' process

6.3.2 Generating a Target PROM/ACE File

- Double click on Generate Target Prom/ACE File

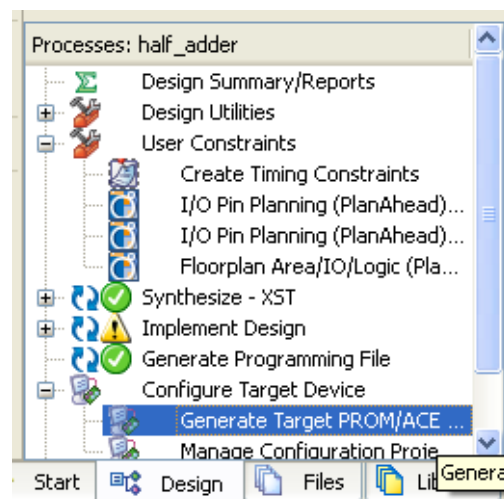


Figure 6.6: Generate Target PROM/ACE File Option Window

- ISE impact window will open (Shown in Figure 6.7)
- Click on Boundary Scan option
- Then right click on the white screen and select initialize chain

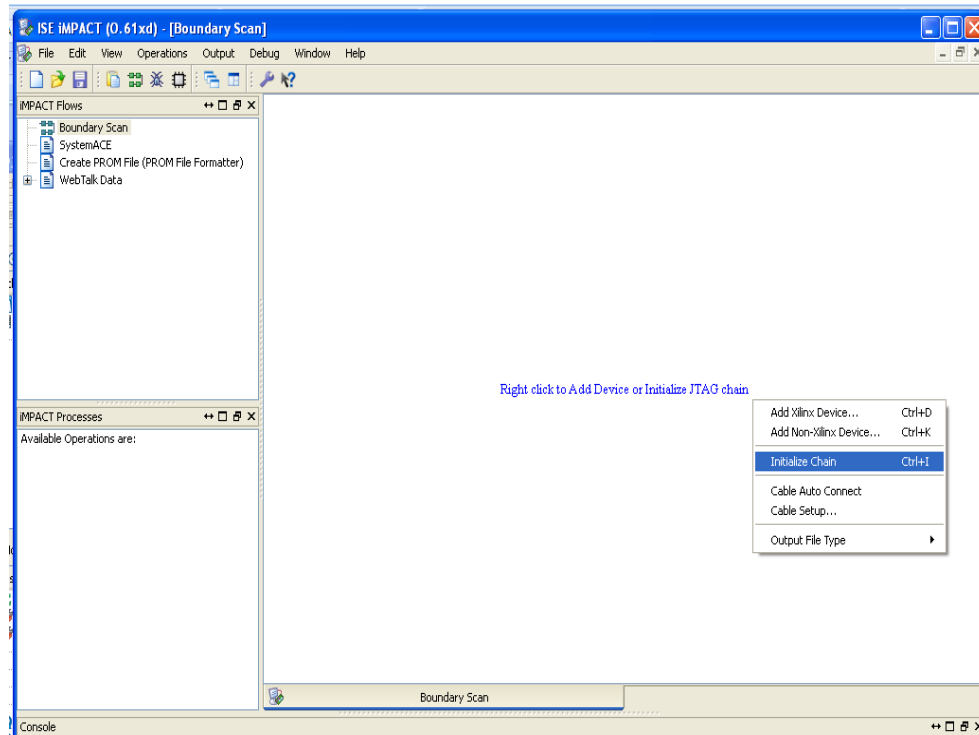


Figure 6.7: ISE Impact Interface

- After identification succeeded; browse the desired project's bit file and click on 'Open' button

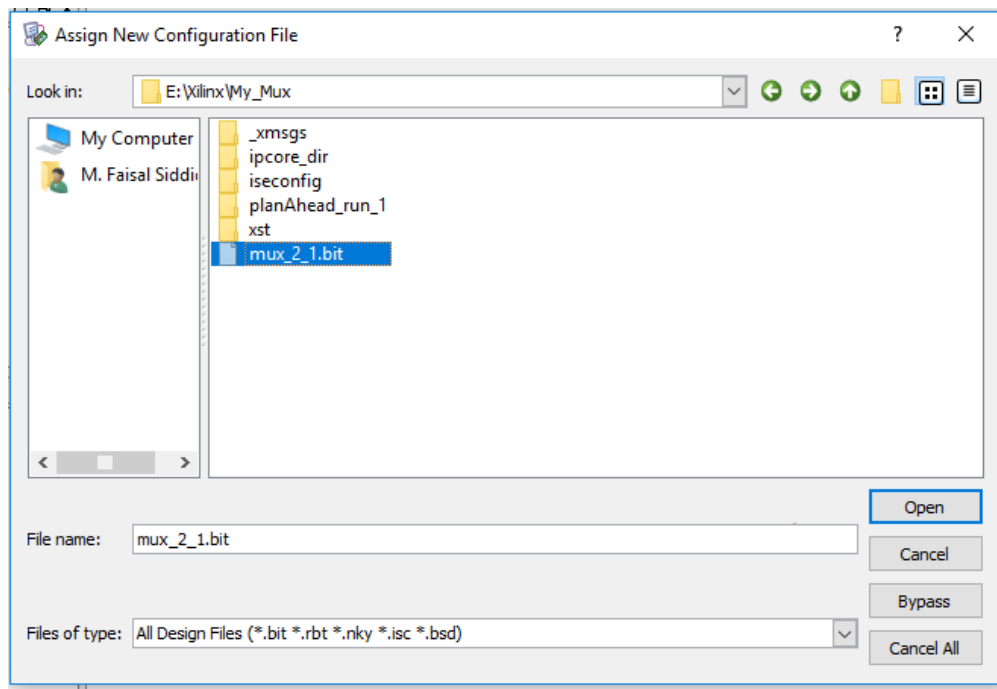


Figure 6.8: Bit file selection

Then bypass the second option (PROM file selection)

- Right click on FPGA chip symbol and click on ‘Program’, to program the FPGA

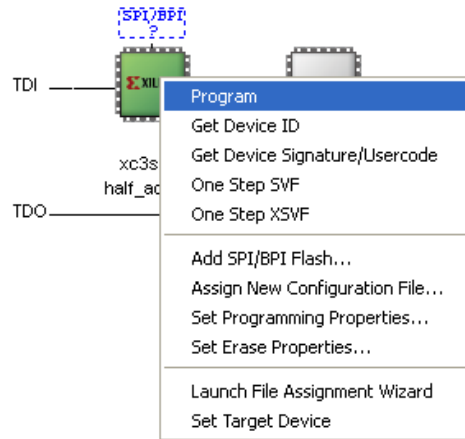


Figure 6.9: Program the FPGA

- After the “Program Succeeded” message comes, test the output on FPGA by changing different input values

In-Lab Task:

1. Using Gate-level model, write a Verilog description of binary to gray converter.
2. Using Behavioral model, write a Verilog description of binary to gray converter.
3. Implement above tasks on FPGA.

Post-Lab Task:

1. Analyse the circuits implementation of In-lab Task 1 and In-lab Task 2 in terms of resource utilization and critical path delay.

Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

LAB #07: Design and Implementation of $n - bit$ Adder/Subtractor on FPGA

Objectives

Part 1

This experiment is to design gate-level hierarchical description of a 4 – *bit* binary Adder / Subtractor in HDL and implement it on FPGA and using ICs.

Part 2

Introduce the different Verilog keyword (**parameter**) and use it to make the module parameterized in Verilog. Parameterized method will allow us to make $n - bit$ binary Adder / Subtractor in Verilog.

Pre-Lab:

Background theory:

Binary Adder is digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain. The process of addition proceeds on a bit-by-bit basis, right to left, beginning with the least significant bit. After the least significant bit, addition at each position adds not only the respective bits of the words, but also consider a possible carry bit from addition at the previous position. Addition of $n - bit$ binary numbers requires the use of a n full adder, or a chain of one-half adder and $n - 1$ full adders. The four-bit adder is an example of a standard component. It can be used in many applications involving arithmetic operations.

Binary Adder/Subtractor can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder. A four-bit adder/subtractor circuit of two binary numbers $A(A_3A_2A_1A_0)$ and $B(B_3B_2B_1B_0)$ is shown in Figure 7.1. The mode input M controls the operation. When $M = 0$, the circuit is an adder. As $B \oplus 0 = B$, $C_0 = 0$, full adders receive the value of B , and the circuit performs $A + B$. when $M = 1$, the circuit becomes a subtractor. Now we have $B \oplus 1 = \bar{B}$ and $C_0 = 1$. The B input is complemented and a 1 is added through the input carry (C_0). The circuit performs the operation A plus the 2's complement of B i.e., $A - B$. The exclusive-OR with output V is for detecting an overflow and C is carry out.

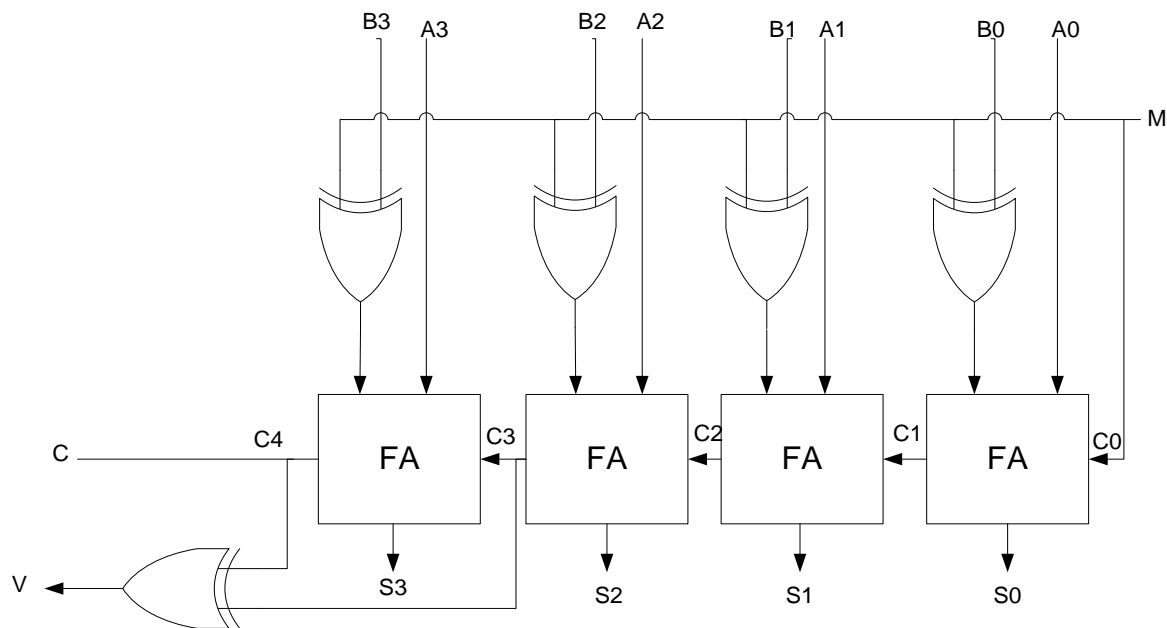


Figure 7.1: Binary adder/subtractor with overflow

Pre-Lab Task:

1. Using Gate-level model, write a Verilog description of a Half-Adder.

In-Lab Tasks:

Part 1 (a) – Implementation of a 4-bit binary Adder / Subtractor without overflow using ICs

Procedure

- Implement a 4 – bit binary adder / Subtractor circuitry as shown in Figure 7.1, without overflow detection.
- Use 4-bit Full Adder IC (7483) (pin description is shown in Figure 7.2) and XOR gates.

16	15	14	13	12	11	10	9
B_3	S_3	C_4	C_0	Gnd	B_0	A_0	S_0
7483 (4-bit Full Adder)							
A_3	A_2	S_2	B_2	Vcc	S_1	B_1	A_1
1	2	3	4	5	6	7	8

Figure 7.2: IC 7483 (4-bit Full Adder) pin configuration

- Connect all the inputs M , A and B (4-bits for each binary number) on slide switches, and the outputs are observed on LEDs.
- Observe the behaviour of Adder /Subtractor by choosing different binary numbers and fill the observation Table given below.

Table 7.1: Observation table for Adder/Subtractor circuit (Implemented using ICs)

		$V_{\text{Calculated}}$	C_{out}	Bit				M
				3	2	1	0	
1	A							
	B							
	Sum							
	Difference							
2	A							
	B							
	Sum							
	Difference							
3	A							
	B							
	Sum							
	Difference							
4	A							
	B							
	Sum							
	Difference							

Part 1 (b) - Implementing hierarchal description of a 4-bit binary Adder / Subtractor

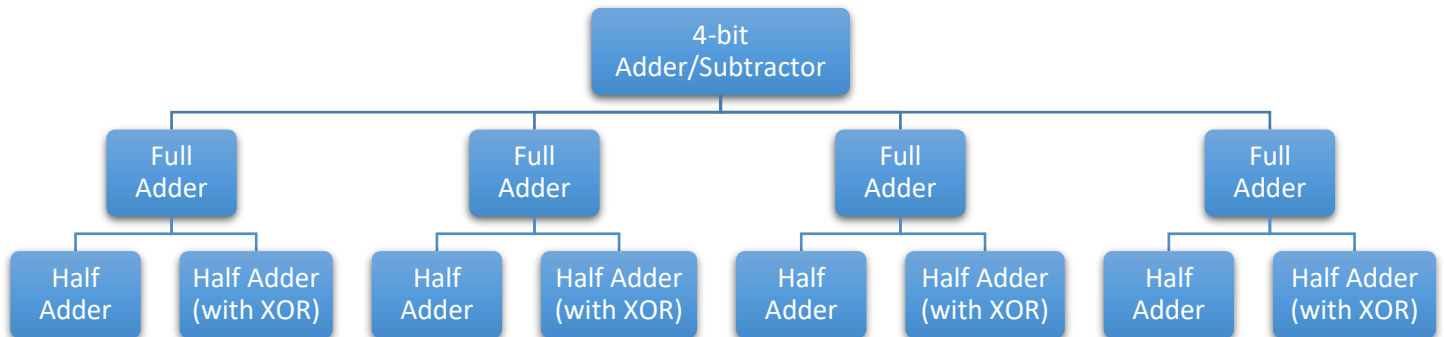


Figure 7.3: Hierarchal description of a 4-bit binary Adder / Subtractor

Procedure

- Write an HDL Verilog code for 4-bit binary Adder/Subtractor using hierarchical approach (Top-Down or Bottom-Up).
 - Use Half Adder modules (made in pre-lab task), by adding existing file in the project, to make a Full Adder module.
 - Then, interconnect Full Adder modules (as shown in Figure 7.3) to make a 4-bit Adder/Subtractor.
 - Finally, add an XOR gate for overflow detection.
- Site all the inputs A and B (4-bits for each binary number) on slide switches, for input M use push switch, and the outputs are observed on LEDs of Nexys2 board.
- Implement the design on FPGA and test the behaviour of Adder /Subtractor by choosing different binary numbers. Fill the observation Table 7.2.

Table 7.2: Observation table for Adder/Subtractor circuit (Implemented on FPGA)

		V	C_{out}	Bit				M
				3	2	1	0	
1	A							
	B							
	Sum							
	$Difference$							

2	<i>A</i>							
	<i>B</i>							
	<i>Sum</i>							
	<i>Difference</i>							
3	<i>A</i>							
	<i>B</i>							
	<i>Sum</i>							
	<i>Difference</i>							
4	<i>A</i>							
	<i>B</i>							
	<i>Sum</i>							
	<i>Difference</i>							

Part 2 – Parameterized modules in Verilog

In this lab task, **parameter** keyword is introduced and use to build the example code.

Parameters: A parameter in Verilog can be any Verilog constant. Parameters are used to generalize a design. For example, a 4 – bit adder becomes more useful as a design if it is put together as an n – bit adder where n is a parameter specified by the user before compilation. Parameter declarations are done immediately after the module declaration.

Here are some typical parameter examples:

```
parameter n = 12;
parameter [3:0]p1 = 4'b1011;
parameter n = 12, m = 32;
```

Parameter Example: n – bit comparator

Comparator is a circuit that computes two inputs (A and B). If A is greater than B , then Greater_Than flag will be high, if A is lesser than B , then Lesser_Than flag will be high, and if A is equal to B , then Equal flag will be high. A and B have any number of bits (i.e., n – bit). The example code of n – bit comparator is given below. Implement it on FPGA for different values of n ($n = 2$ and $n = 4$).


```
module comparator(A_gt_B,A_lt_B,A_eq_B,A,B);

parameter n=4; //Initializing n=4 but can changeable during
                instantiation
output reg A_gt_B,A_lt_B,A_eq_B;
input [n-1:0]A,B; //parameter n is used as number of bits for input
always @(A or B)
begin
    if(A==B)    begin
        A_eq_B = 1;
        A_gt_B = 0;
        A_lt_B = 0;
    end
    else if (A>B)    begin
        A_eq_B = 0;
        A_gt_B = 1;
        A_lt_B = 0;
    end
    else begin
        A_eq_B = 0;
        A_gt_B = 0;
        A_lt_B = 1;
    end
end
endmodule
```

To instantiate parameterized module for different values of n :

```
// <module name> #(v_of_p 1,2,...) <inst. Name.> (port mapping);
// v_of_p = Value of parameter(s)

comparator #(2) C1 (A_gt_B,A_lt_B,A_eq_B,A,B); // Now n=2 so 2-bit
                                                comparator

comparator #(8) C2 (A_gt_B,A_lt_B,A_eq_B,A,B); // Now n=8 so 8-bit
                                                comparator

comparator C3 (A_gt_B,A_lt_B,A_eq_B,A,B);      // No new value so
                                                //default value of n will be used i.e., 4-bit comparator
```

Post-Lab:

1. Using Behavioral model, write a Verilog description of n – bit Adder /Subtractor:
 - Make one stimulus for two different parameter values and show the wave forms results.
2. Analyse the circuits implementation of Structural-Level 4 – bit Adder/Subtractor and Behavioral-Level 4 – bit Adder/Subtractor, in terms of resource utilization and critical path delay.

HDL codes / Results (Wave Forms) / Analysis

Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

LAB #08: Design and Implementation of $n - bit$ Binary Multiplier on FPGA

Objective

In this lab, we will seek how to design $n - bit$ binary multiplier and its implementation on FPGA.

Pre-Lab:

Background Theory:

Binary Multiplier

Binary multiplier is used to multiply two binary numbers. Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers. It is build using binary adders. The common multiplication method is “add and shift” algorithm. If the multiplicand is $N - bits$ and the Multiplier is $M - bits$ then there is $N \times M$ partial product. AND gates are used to generate the Partial Products. Note that in binary multiplication, the processes involve shifting the multiplicand, and adding the shifted multiplicand or zero. Each bit of the multiplier determines whether a 0 is added or a shifter version of the multiplicand (0 implies zero added, 1 implies shifted multiplicand added). Thus, we can infer a basic shift-and-add algorithm to implement unsigned binary multiplication.

General Expression

$$\begin{array}{r}
 \begin{array}{cc}
 & B_1 & B_0 \\
 \times & A_1 & A_0 \\
 \hline
 & A_0 B_1 & A_0 B_0 \\
 A_1 B_1 & A_1 B_0 & \\
 \hline
 C_3 & C_2 & C_1 & C_0
 \end{array}
 \end{array}$$

Page 93

Figure 8.1: 2 – bit by 2 – bit binary multiplier

Pre-Lab Task

1. Using Structural-Level model, write a Verilog description of 2 – bit by 2 – bit multiplier.

In-Lab Task 1: Implement 4 – bit by 2 – bit binary multiplier using ICs

- Implement 4 – bit by 2 – bit binary multiplier using ICs as shown in Figure 8.2.
- Use 4 – bit Full Adder IC (7483) and AND gate ICs.
- Observe the behaviour of a binary Multiplier by choosing different binary numbers and verify the results.

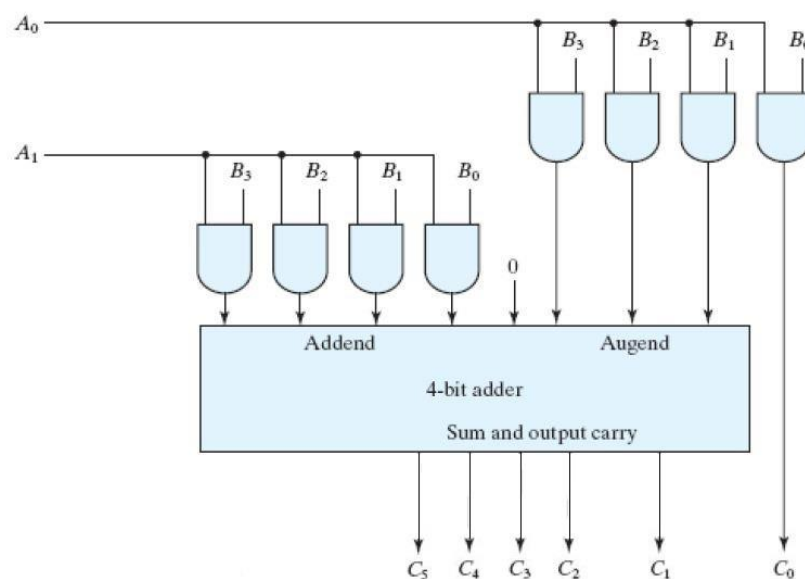


Figure 8.2: 4 – bit by 2 – bit binary multiplier

In-Lab Task 2: Implement 4 – bit by 3 – bit binary multiplier on FPGA

1. Using Structural model, write a Verilog description of 4 – bit by 3 – bit multiplier.
 - a. Use 4 – bit Adder module made in Lab7.
2. Site all the inputs A and B on slide switches and the outputs are observed on LEDs of Nexys2 board.
3. Implement the design on FPGA and test the behaviour of designed binary Multiplier by choosing different binary numbers.

Post-Lab Tasks:

1. Using Behavioral model, write a Verilog description of parameterized Multiplier:
 - Make one stimulus for two different parameter values and show the wave forms results.
2. Analyse the circuits implementation of Structural Binary Multiplier and Behavioral-Level Multiplier, in terms of resource utilization and critical path delay (Input size should be same).

Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

LAB #09: Design and Implementation of BCD to 7-Segment Decoder on FPGA

Objective

In this lab, we design and implement BCD to 7-Segment decoder. Moreover, we learn how to use the 7-segment display of Nexys2 FPGA board.

Pre-Lab:

Background Theory:

Binary to BCD Decoder

In digital, **binary-coded decimal (BCD)** is a class of binary encodings of decimal numbers where each decimal digit is represented by a fixed number of bits, usually four. Table 9.1 gives the four-bit code for one decimal digit. A number with k decimal digits will require $4k$ bits in BCD.

Table 9.1: Binary coded decimal (BCD)

Decimal Value	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

BCD to 7-Segment Decoder

BCD to 7-segment decoder decodes the BCD value on 7-Segment display.

7-Segment Display

A seven-segment display (SSD) is a form of the electronic display device for displaying decimal numerals. The seven elements of the display can be selected in different combinations to represent the decimal numerals. Often the seven segments are arranged in an oblique (slanted) arrangement, which aids readability. Seven-segment displays may use a light-emitting diode (LED) or a liquid crystal display (LCD), for each segment, or other light-generating or controlling techniques. There are two types of simple LED package 7-Segment display:

- Common Anode
- Common Cathode

7-Segment (Common Anode) Displays on Nexys2 board

Nexys2 board contains a four-digit common anode seven-segment LED display. Each of the four digits is composed of seven segments arranged in a pattern (as shown in Figure 9.1), with an LED embedded in each segment. Segment LEDs can be individually illuminated, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark.

The anodes of the seven LEDs forming each digit are tied together into one “common anode” circuit node, but the LED cathodes remain separate. The common anode signals are available as four “digit enable” input signals to the 4-digit display. The cathodes of similar segments on all four displays are connected into seven circuit nodes labeled CA through CG (so, for example, the four “D” cathodes from the four digits are grouped together into a single circuit node called “CD”). These seven cathode signals are available as inputs to the 4-digit display. This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits, but they can only illuminate the segments of the digit whose corresponding anode signal is asserted.

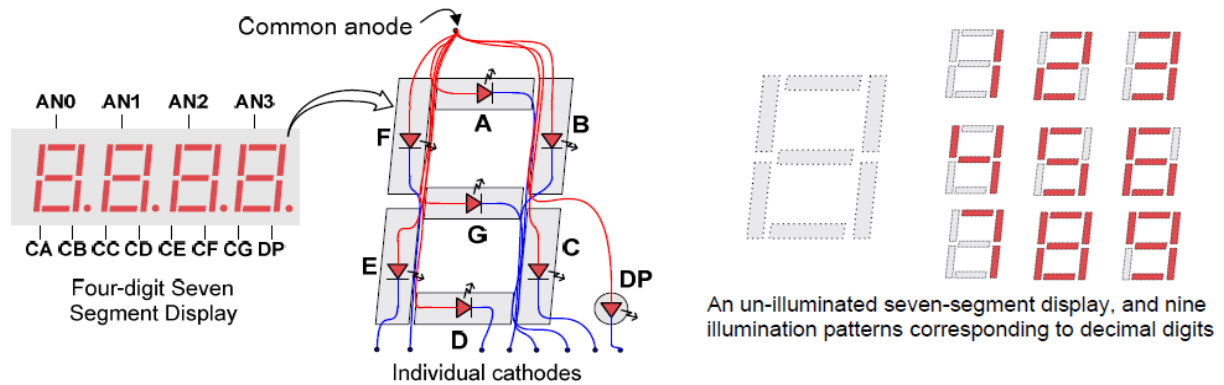


Figure 9.1: Nexys2 Board 7-Segment Displays [Nexys2 reference manual]

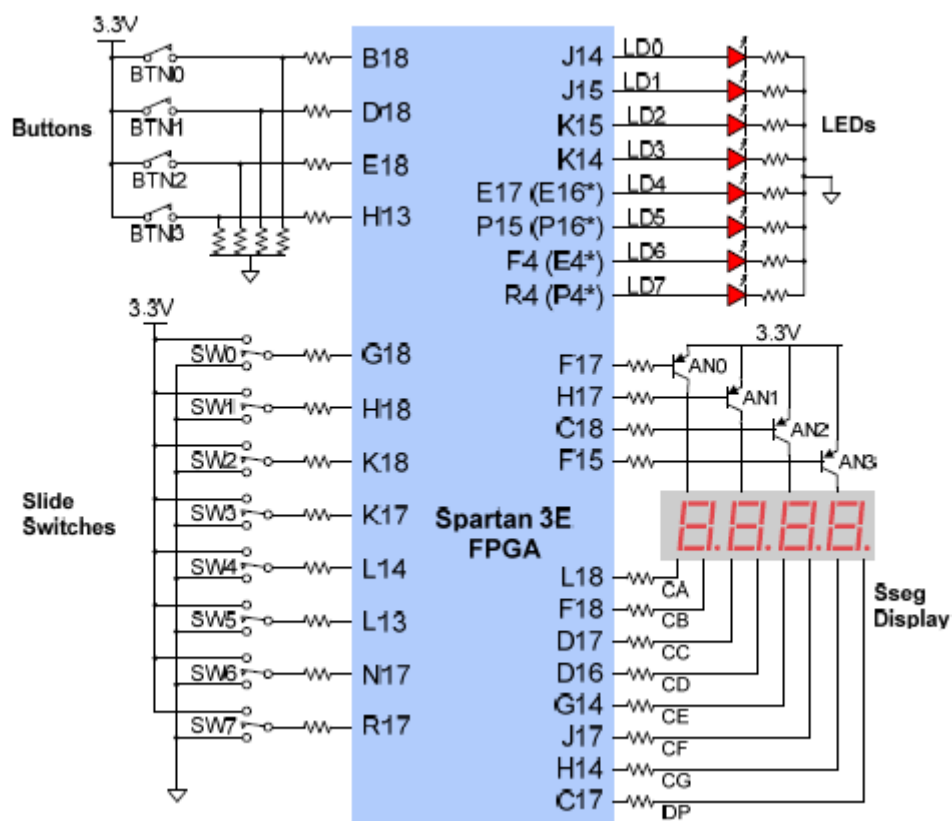


Figure 9.2: Nexys2 Board I/O devices and circuits [Nexys-2 reference manual]

CASE statement in Verilog

```

case (switching variable)
Label Value1: statement(s);
Label Value2: statement(s);

Label Value n: statement(s);
default: statement(s);
endcase
    
```

Example (4-to-1 MUX using CASE statement with parameterized data width)

```
module mux_4_1( Y, I_0, I_1, I_2, I_3, S );

parameter width=4;

output reg [width-1:0]Y;
input [width-1:0]I_0,I_1,I_2,I_3;
input [1:0]S;

always @ ( I_0 or I_1 or I_2 or I_2 or S)
begin
    case (S) //As MUX output is selected by selection line, here
    'S', so it is a switching variable
    2'b00: Y = I_0; // when S=00 then I0 is selected for output
    2'b01: Y = I_1; // when S=01 then I1 is selected for output
    2'b10: Y = I_2; // when S=10 then I2 is selected for output
    2'b11: Y = I_3; // when S=11 then I3 is selected for output
    default: Y = {width{1'b0}}; // when S=zz or xx then output is
    0 (By default)
    endcase
end
endmodule
```

In-Lab Task 1: Test the functionality of a BCD to 7-Segment decoder IC (CD4511) with common cathode 7-Segment display**Procedure**

- Make a circuitry as shown in Figure 9.2.
- Connect all the inputs *A*, *B*, *C* and *D* with the switches.
- Observe the behaviour of a BCD to 7-Segment decoder on 7-Segment by choosing different BCD values.

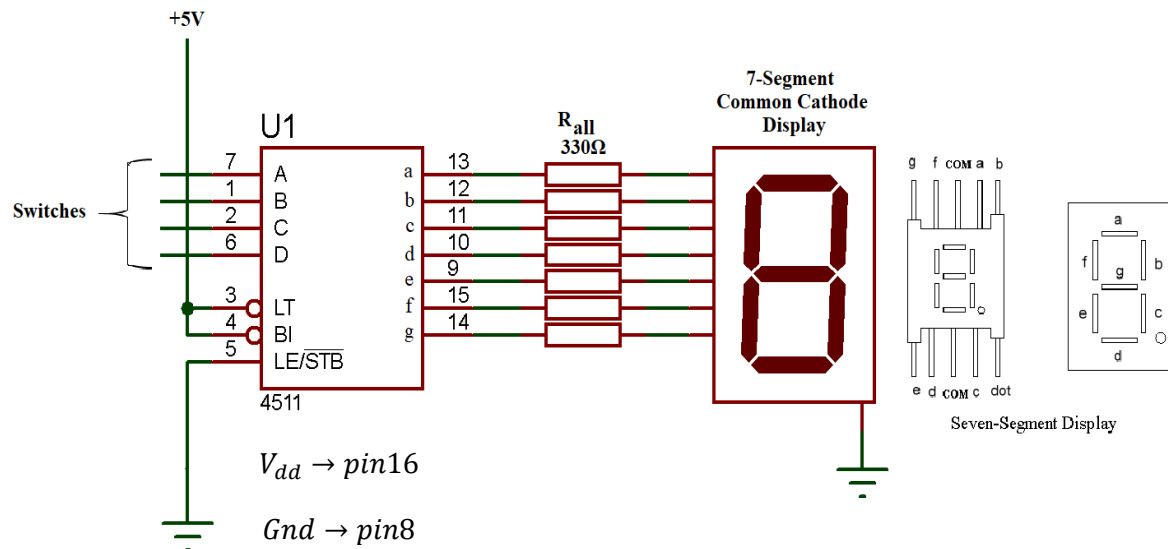


Figure 9.3: BCD to 7-Segment decoder (CD4511) with common cathode circuitry

In-Lab Task 2: Design and implementation of a BCD to 7-Segment decoder on FPGA (Nexys2)

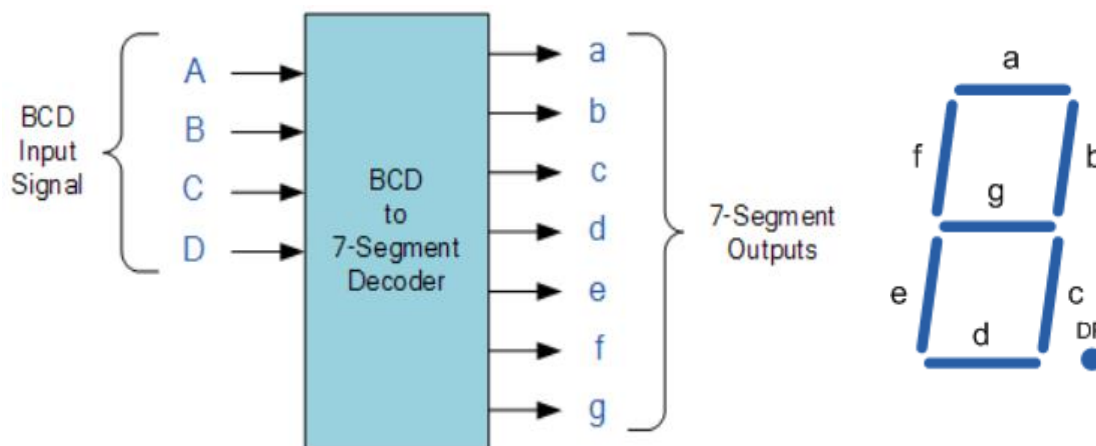


Figure 9.4: BCD to 7-Segment decoder module

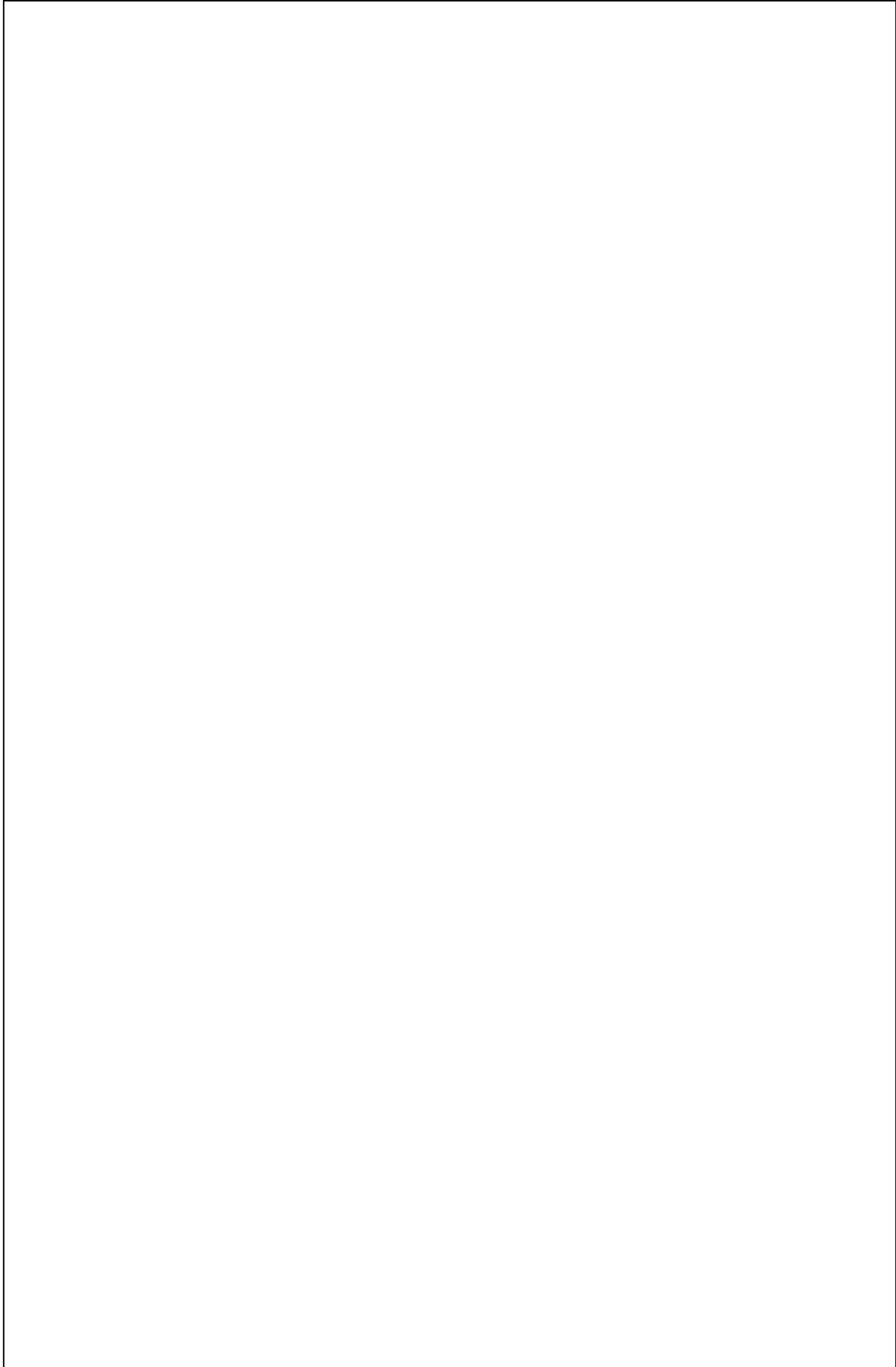
- First, make a truth table for BCD to 7-Segment decoder which has 4-inputs (BCD) and 8-outputs (A to G and DP) of the 7-Segment display (Fill Table 9.2).
- Using Behavioral model, write a Verilog description of BCD to 7-Segment decoder using Case statements.
- Implement it on FPGA (Only one 7-Segment should be on). Remember 7-Segment displays on Nexys2 board are common anode with driver circuitry.
HINT: Activate only one segment anode by **assign** keyword. You have four 7-Segment anodes (AN3, AN2, AN1, AN0) on Nexys2 board.

Table 9.2: Truth table for BCD to 7-Segment decoder

BCD				7-Segment								
<i>D</i>	<i>C</i>	<i>B</i>	<i>A</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>DP</i>	<i>Hex</i>
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									
Decimal Point (DP)												
Error (E)												

Post-Lab Task:

1. Make a stimulus for BCD to 7-Segment decoder (In-Lab Task 2).
2. Report the synthesise key parameters such as resource utilization and critical path delay.



Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

Lab #10: Design and Implementation of a Sequence Detector using Mealy/Moore Machine

Objective

This experiment is to design a Sequence detector using Mealy/Moore models of Finite State Machine (FSM) and implement the design on FPGA.

Pre-Lab:

Background theory:

The most general model of a sequential circuit has inputs, outputs, and internal states. It is customary to distinguish between two models of sequential circuits: the Mealy model and the Moore model. They differ only in the way the output is generated. The two models of a sequential circuit are commonly referred to as a finite state machine, abbreviated FSM.

In the Mealy model, the output is a function of both the present state and the input as shown in Figure 10.1. the outputs may change if the inputs change during the clock cycle. the output of the Mealy machine is the value that is present immediately before the active edge of the clock.

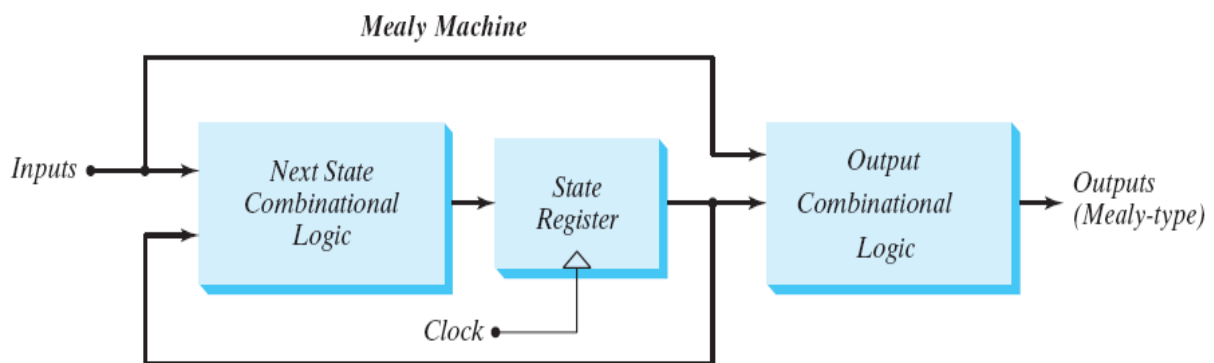


Figure 10.1 Mealy machine model

In the Moore model, the output is a function of only the present state. A circuit may have both types of outputs as shown in Figure 10.2. The outputs of the sequential circuit are synchronized with the clock because they depend only on flip-flop outputs that are synchronized with the clock.

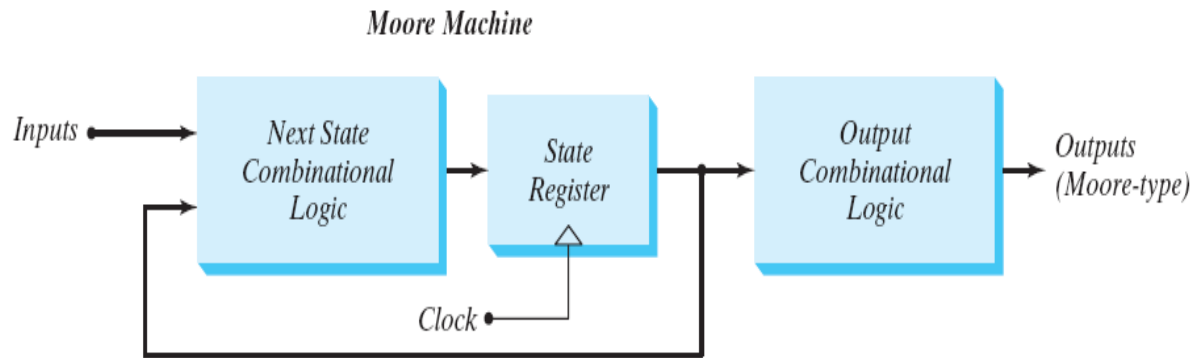


Figure 10.2 Moore machine model

Example 1: Behavioral description of D flip-flop

```
module DFF (Q,D,clk,rst);
    output reg Q;
    input D,clk,rst;

    always @ (posedge clk or posedge rst)
    begin
        if (rst)
            Q <= 1'b0;
        else
            Q <= D;
        end
    endmodule
```

Example 2: Behavioral description of T flip-flop

```
module TFF (Q,T,clk,rst);
    output reg Q;
    input T,clk,rst;
    always @ (posedge clk or posedge rst)
    begin
        if (rst)
            Q <= 1'b0;
        else
            begin
                if (T)
                    Q <= ~Q;
                else
                    Q <= Q;
            end
        end
    endmodule
```

Example 3: Stimulus for D flip-flop

```
module ();  
  parameter PERIOD_H=5;  
  reg clk,rst,D;  
  wire Q;  
  initial begin  
    clk = 1'b0;    rst = 1'b1;    D = 1'b1;  
    #10            rst = 1'b0;    D = 1'b1;  
    #10            D = 1'b0;  
  end  
  always  
    #PERIOD_H clk=~clk;  
  dff DUTT(Q,D,clk,rst );  
endmodule
```

In-Lab: Design and Implementation of a Sequence Detector using Mealy Machine

Sequence: _____

(Note: Sequence will be specified by Lab Instructor)

In-Lab Task 1: State diagram (Mealy), State table, State Equations and Circuit Diagram

In-Lab Task 2: HDL implementation for Mealy machine on FPGA

Step 1: Write the HDL (Verilog) structural description for Mealy based sequence detector for the given sequence. You can use D flip-flop module made in pre-lab task example.

(Insert/Write HDL code here)

Step 2: Instantiate/connect your model according to the given block diagram (as shown in Figure 10.3). The other modules (named: test_pattern and clk_dvsr) are available in your Lab10 folder.

- Add existing files (“test_pattern.v”, “clk_1s.v” and <your sequence detector module>) by Add source option in your project.
- Add a new_source Verilog file (e.g., named as main) and then connect them according to the block diagram below:
- Then implement it on FPGA.

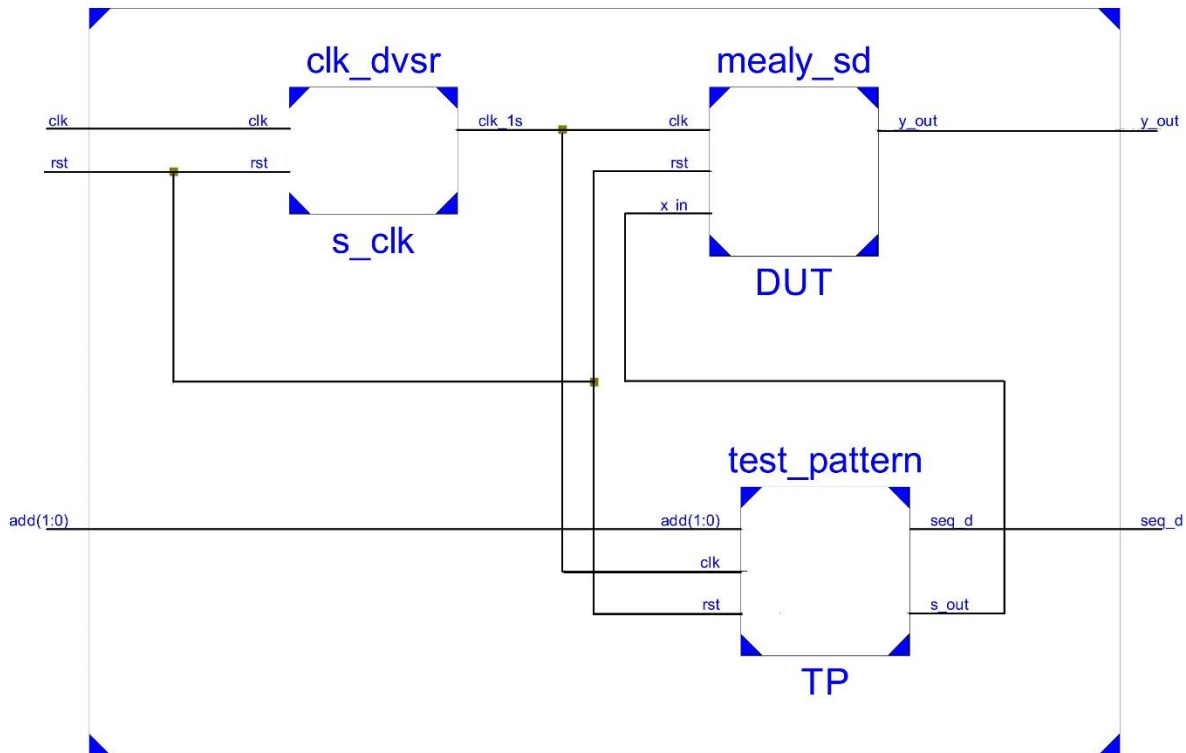


Figure 10.3: Circuit diagram for Lab 10 main module

clk_dvsr (Clock Divisor) Module:

Nexys2 board has a 50 MHz clock. Clock divisor will generate 1 MHz clock. It is used for output verification purposes (B8 is a site number of 50 MHz clock).

test_pattern Module:

test_pattern module will generate four different patterns and anyone of the four patterns is selected by an add (2-bit address) value to test the sequence detector. Output “s_out” will serially out the selected pattern. Output “seq_d” is a flag which indicates the pattern is completed. Table 10.1 shows the pattern selection according to the “add” value.

Table 10.1: Test patterns generated by “test_pattern” module on add value

add	Pattern Number	Pattern/Sequence
00	P1	0101
01	P2	1010
10	P3	0011
11	P4	1100

Post-Lab Tasks:

1. Using State diagram-based Behavioral model, write a Verilog description of Mealy based sequence detector for the given sequence.
 - Make a stimulus for the given task.
 - Record the simulation output waveforms in observations.
2. Using State diagram-based Behavioral model, write a Verilog description of Moore based sequence detector for the given sequence.
 - Make a stimulus for the given task.
 - Record the simulation output waveforms in observations.
3. Analyse the circuits implementation of Structural and Behavioral-Level Mealy sequence detectors, in terms of resource utilization and maximum frequency.

Write HDL code here

(Paste the simulation output waveforms here)

Critical Analysis/Conclusion

Lab Assessment				
Pre-Lab			/1	/10
In-Lab			/5	
Post-Lab	Data Analysis	/4	/4	
	Data Presentation	/4		
	Writing Style	/4		
Instructor Signature and Comments				

LAB #11: Implementation of a BCD Counter with Control Inputs on FPGA

Objective

In this open-ended lab, students need to implement a BCD counter with control inputs on FPGA. The students need to explore:

- Which flip flop is more suitable to design a counter
- Basic control inputs required for a BCD counter
- Available output devices to display the result of a BCD counter on Nexys2 board
- Additional circuitry required to show the result on Nexys2 board
- Comparative analysis of a BCD counter implementation in Structural Level and Behavioral Model

Pre-Lab:

Counter

A counter is essentially a register that goes through a predetermined sequence of binary states. The gates in the counter are connected in such a way as to produce the prescribed sequence of states. Although counters are a special type of register, it is common to differentiate them by giving them a different name. An $n - bit$ binary counter consists of n flip-flops that can count in binary from 0 to $2^n - 1$. The state diagram of a 3 – bit binary counter is shown in Figure 11.1.

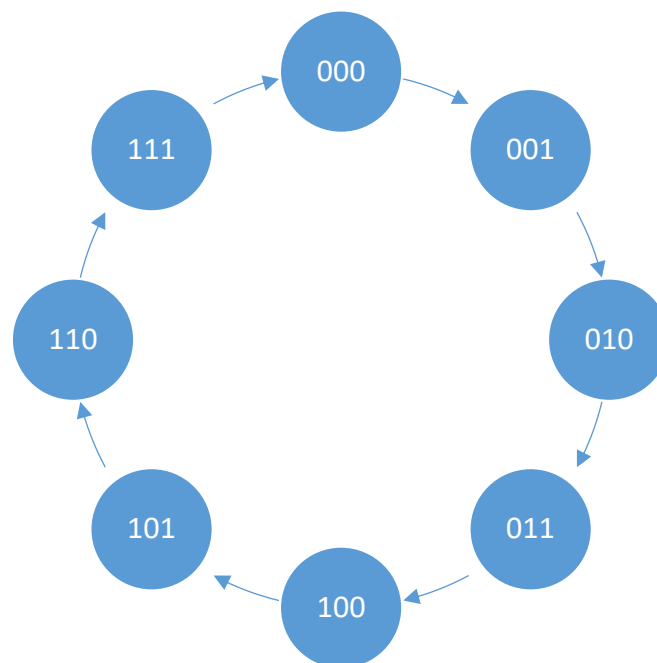
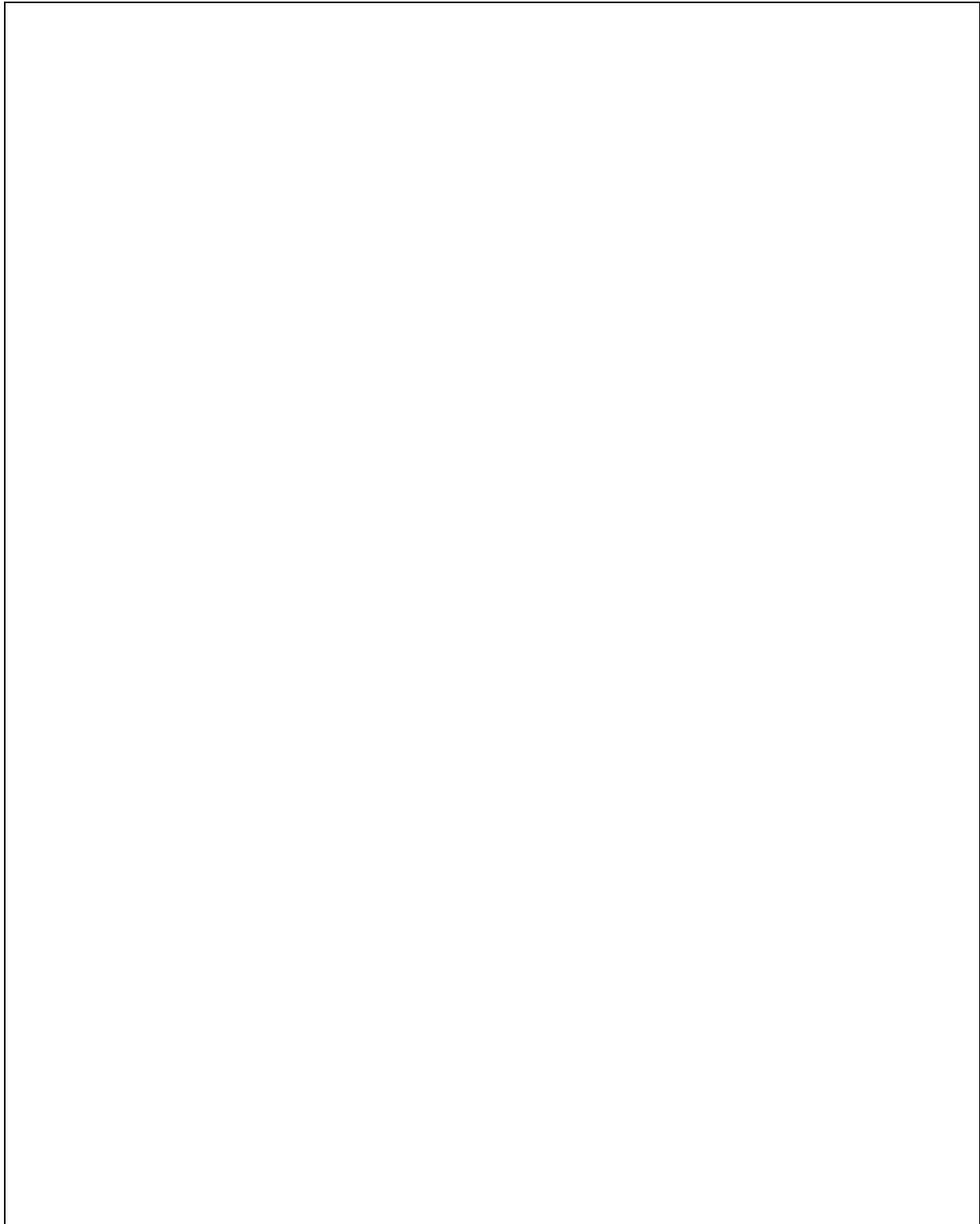


Figure 11.1: State diagram of a 3 – bit binary counter

BCD Counter

A BCD counter follows a sequence of 10 states and returns to 0 after the count of 9. It requires four flip-flops to represent each decimal, as a single BCD digit is represented by at least four bits.

Control Inputs:



Pre-Lab Task: Design a BCD counter using flip-flop: State Diagram, State Table, State Equations and Circuit Diagram

In-Lab: Implementation of a BCD Counter with control inputs on FPGA

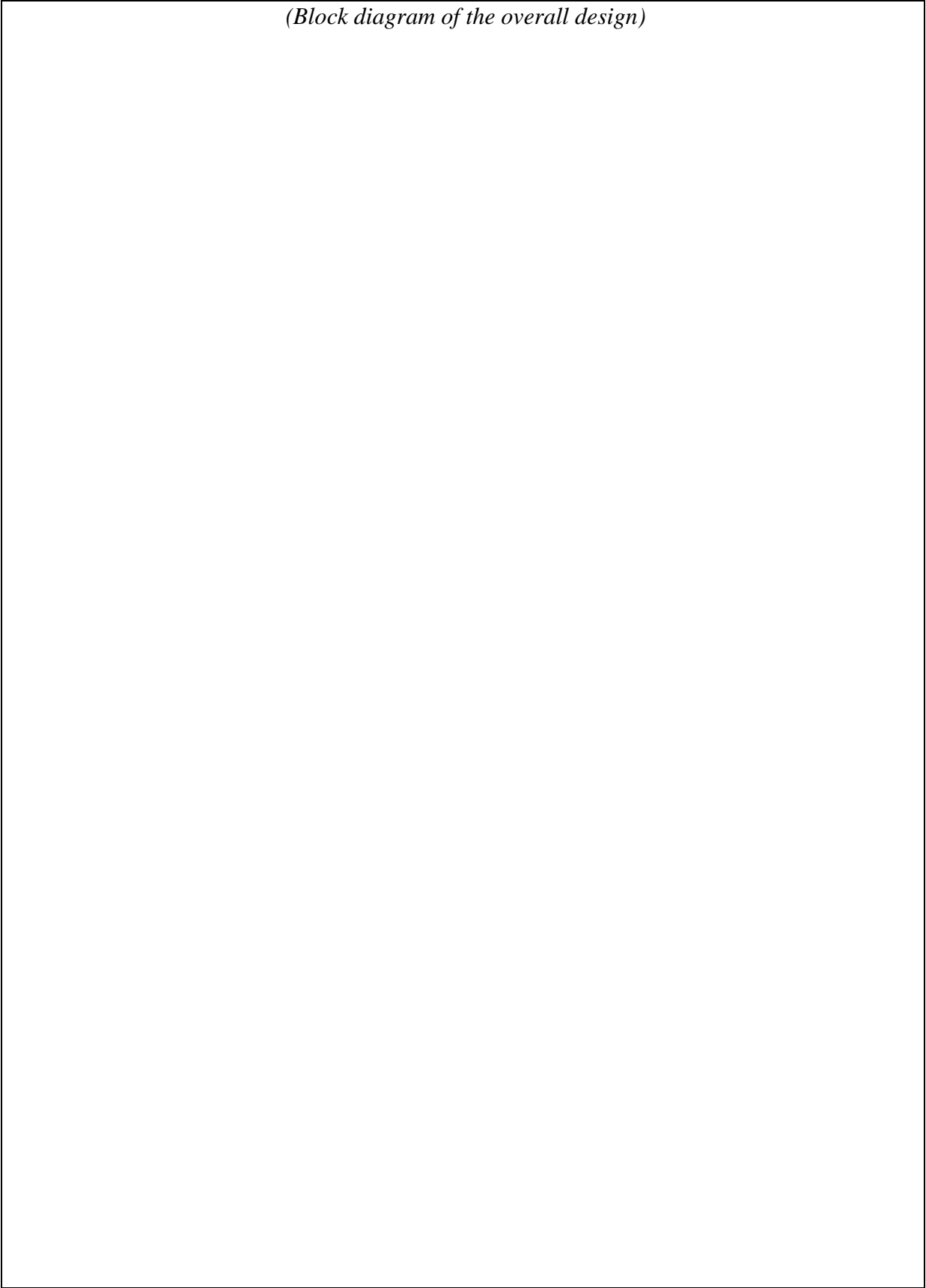
In-Lab Task 1: Structural implementation of a BCD counter and display its result on output device of Nexys2 board

Step 1: Write an HDL (Verilog) structural description of a BCD counter with control inputs:

(Insert/Write HDL code here)

Step 2: Provide a block diagram which shows all the additional modules required to implement the design and shows the result on output device of Nexys2 board:

(Block diagram of the overall design)



Step 3: Write an HDL (Verilog) structural description of a main module (according to the block diagram provided in Step 2) and then implement it on FPGA:

(Insert/Write HDL code here)

In-Lab Task 2: Implementation of a BCD counter with control inputs using state diagram-based HDL behavioral model, and display its result on output device of Nexys2 board

Step 1: Write an HDL code (State diagram-based behavioral description) of the design:

(Insert/Write HDL code here)

Post-Lab Task:

1. Using Behavioral model, write a Verilog description for an $n - bit$ up/down binary counter with control inputs.
 - Make a stimulus for the given task.
 - Record the simulation output waveforms in observations.
2. Analyse the different implementation models for a counter used in this lab.

Critical Analysis/Conclusion

Instructor Signature and Comments

Lab #12: Implementation of a Special Shift Register on FPGA

Objective

In this open-ended lab, students need to implement a special shift register which can perform different shifting operations including universal shift register options. The students need to explore:

- Which flip flop more suits to design a shift register
- Different types of shifting
- Multiple shifting
- Additional circuitry required to show the result on Nexys2 board
- Comparative analysis of a special shift register implementation in Structural Level and Behavioral Model

Pre-Lab:

Register

A register is a digital circuit, which can store data. An $n - bit$ register consists of $n - flip$ flops capable of storing $n - bits$ of binary information. The two common features of registers are:

- Data storage
- Data movement

Shift Register

A shift register is a group of flip-flops set up in a linear fashion with their inputs and outputs connected in such a way that the data is shifted from one flip flop to another when the circuit is active.

Universal Shift Register

A universal shift register (USR) is an integrated logic circuit that can transfer data in three different modes. Like a parallel register: it can load and transmit data in parallel. Like a shift registers: it can load and transmit data in serial fashions, through left shifts or right shifts. In addition, the universal shift register can combine the capabilities of both parallel and shift registers to accomplish tasks that neither basic type of register can perform on its own. For instance, on a particular job a universal register can load data in series (e.g. through a sequence of left/right shifts) and then transmit/output data in parallel.

Depending on the signal values on the select lines of the multiplexers, the register can retain its current state, shift right, shift left or be loaded in parallel. Each operation is the result of an active edge on the clock line.

To operate USR in a specific mode, it must first select the mode. To accomplish mode selection the universal register uses a set of two selector switches, S1 and S0. As shown in Table 12.1, each permutation of the switches corresponds to a loading/input mode.

Table 12.1: Operating mode of USR

Operating Mode	S1	S0
No Change	0	0
Shift-Right	0	1
Shift-Left	1	0
Parallel Load	1	1

No Change Mode:

In the no changed mode ($S_1S_0 = 00$) the register is not admitting any data; so that the content of the register is not affected by whatever is happening at the inputs.

Shift-Right Mode:

In the shift-right mode ($S_1S_0 = 01$) serial inputs are shifted from D_3 to D_0 (in a case of 4 – bit USR). You can confirm this aspect by setting the value of the shift-right switch according to the sequence “0010011”, for each active edge the data will shift right as shown in Table 12.2.

Table 12.2: Data movement in USR for shift-right mode

Clock Cycle	MSB_{in}	D_3	D_2	D_1	D_0
Initial Value	1	0	0	0	0
Cycle 1	1	1	0	0	0
Cycle 2	0	1	1	0	0
Cycle 3	0	0	1	1	0

Cycle 4	1	0	0	1	1
Cycle 5	0	1	0	0	1
Cycle 6	0	0	1	0	0
Cycle 7	×	0	0	1	0

Shift-Left Mode:

In the shift-left mode ($S1S0 = 10$) serial inputs are shifted from D_0 to D_3 (in a case of 4 – bit USB). You can confirm this aspect by setting the value of the shift-right switch according to the sequence “0010011”, for each active edge the data will shift right as shown in Table 12.3.

Table 12.3: Data movement in USB for shift-left mode

Clock Cycle	D_3	D_2	D_1	D_0	LSB_{in}
Initial Value	0	0	0	0	1
Cycle 1	0	0	0	1	1
Cycle 2	0	0	1	1	0
Cycle 3	0	1	1	0	0
Cycle 4	1	1	0	0	1
Cycle 5	1	0	0	1	0
Cycle 6	0	0	1	0	0
Cycle 7	0	1	0	0	×


Parallel-Load mode:

Finally, in the parallel-load mode ($S1S0 = 11$) data is read from the lines I_0 , I_1 , I_2 and I_3 simultaneously and stores it in register D_0 , D_1 , D_2 and D_3 , respectively; after cycling the clock (transition of the active edge) as depicted in Table 12.3 (in a case of 4 – bit USB).

Multiple shifting: (Description with example)

Block Diagram of the design: Special Shift Register

(Block diagram of the design)



In-Lab Tasks:


In-Lab Task 1: Structural implementation of a 4 – *bit* special shift register and display its result on output device of Nexys2 board

Step 1: Write an HDL (Verilog) structural description of a 4 – *bit* special shift register (designed in pre-lab):

(Insert/Write HDL code here)

Step 2: Provide a block diagram of the overall design which shows all the additional modules required to implement the design and shows the result on output device of Nexys2 board:

(Block diagram of the overall design)



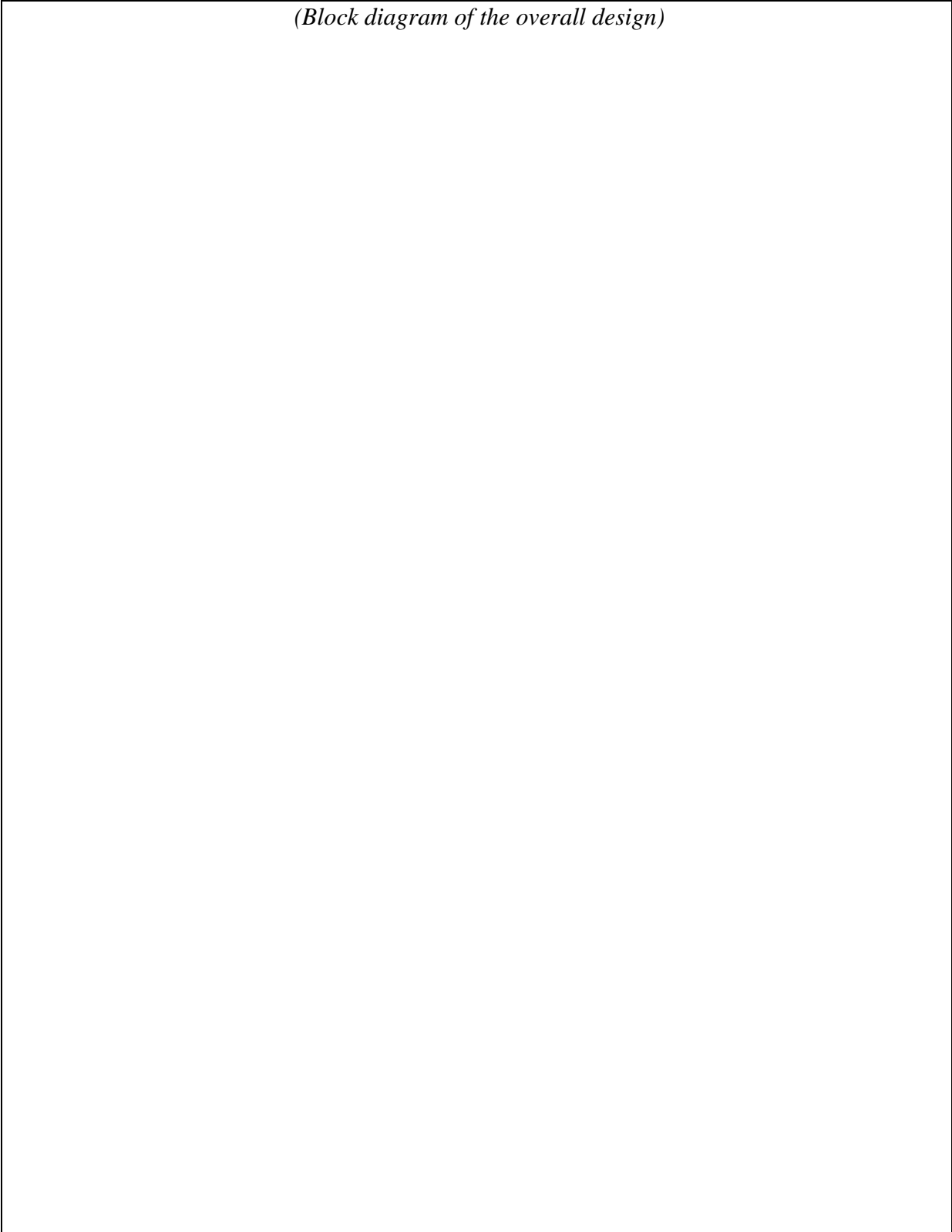
Step 3: Write an HDL (Verilog) structural description of a main module (according to the block diagram provided in Step 2) and then implement it on FPGA:

(Insert/Write HDL code here)

In-Lab Task 2: Repeat the In-lab Task 2 of Lab#10, by using special shift register (designed in this lab) instead of test_pattern module

Step 1: Provide a block diagram of the overall design which shows all the additional modules required to implement the In-Lab Task 2 and shows the result on output device of Nexys2 board:

(Block diagram of the overall design)



Step 2: Write an HDL (Verilog) structural description of a main module (according to the block diagram provided in Step 1) and then implement it on FPGA:

(Insert/Write HDL code here)

Post-Lab Task:

1. Using Behavioral model, write a Verilog description for an $n - bit$ special shift register.
 - Make a stimulus for the given task.
 - Record the simulation output waveforms in observations.
2. Analyse the different implementation models of a special shift register ($4 - bit$) used in this lab.

Critical Analysis/Conclusion

Instructor Signature and Comments