CPE 271L - Final Lab Portfolio

Mustafa Al-Shebeeb

Partner: Jeffrey Thompson

Institution: West Virgina University

Semester: Fall 2023

TA: Michael Barchett

CPE 271L: Course Reflection

Throughout the Fall 2023 semester, CPE 271 Lab proved to be both challenging and intellectually rewarding. The sequence of labs was thoughtfully organized, commencing with a fundamental exploration of logic gates, Quartus Prime Software, and the DE10 Lite Board. Our initial focus on simple binary operations, such as addition and subtraction using logic gates and schematic files, paved the way for a smooth transition into the realm of VHDL coding.

Around week 4, as our logic became more intricate, the introduction of VHDL provided a crucial tool for designing and implementing logic efficiently. This programming language not only expedited the coding process but also facilitated debugging. Additionally, we delved into the use of Waveforms in Quartus Prime, a significant learning outcome. This feature allowed us to visualize and analyze output within the software itself, eliminating the need for a physical connection to the DE10 board.

As the semester progressed, our exploration of synchronous sequential circuits provided insights into memory, including the workings of Read-Only Memory (ROM) and Random Access Memory (RAM). The final lab project, a light controller using PLC, encapsulated our cumulative learning. Noteworthy was the opportunity to enhance our designs for bonus points, encouraging us to explore new and original ideas.

Throughout the course, we honed skills in creating effective lab reports. Topics covered included the creation of port map statements, the design of multiplexers, and other concepts integral to the successful completion of our projects. This course not only deepened our understanding of digital systems but also fostered a spirit of innovation and improvement, exemplified by the chance to go the extra mile in refining our designs.

I believe that Michael Barchett has been a great instructor and very helpful. However, I suggest a slight adjustment in the teaching approach. Instead of lecturing the entire lab in one session and then letting us work independently, it would be more beneficial to go through the lab material step by step. This approach would enable us to pace ourselves better. Additionally, I noticed instances where the lab content seemed ahead of the lecture, making it challenging to grasp the relevance of certain tasks without comprehensive information.

Reflecting on the semester, I am grateful for the guidance provided by Michael Barchett. CPE 271 has fueled my interest in exploring more technical applications within Digital Logic Design. The capabilities of VHDL and FPGA boards appear promising and exciting.

In this comprehensive document, I have compiled the results and analyses from the 10 labs we completed this semester. Each of my lab reports incorporates the suggested edits.

Lab 1: Introduction to Quartus Prime, DE10-Lite Board, & Basic Logic Gates

Date performed: 8/22/2023

Introduction:

This lab focused on introducing the Quartus Prime software and its application in creating and implementing logic circuits using the DE10 Lite Board hardware. Logic gates, the fundamental building blocks of logical circuits, were explored. These gates evaluate inputs to generate outputs based on their logic type, such as NOT, NAND, OR, and AND gates. The lab aimed to model logical circuit behavior by analyzing individual logic gates within the circuit. This process helps achieve desired outputs under specific input conditions. Understanding these gates is valuable as modern computers heavily rely on rapid logic gate evaluations, emphasizing the importance of grasping their functionality.

PART 1

Experiments of Part I

A minterm is an approach for representing inputs as a sum of products. Taking inputs A, B, and C as an example, a minterm is denoted as A' (A bar) when A is 0, and simply as A when A is 1. For instance, if A=1, B=0, and C=1, the corresponding minterm is AB'C. It signifies the summation of input products that yield a high output.

Logic Expression (Visual Analysis):

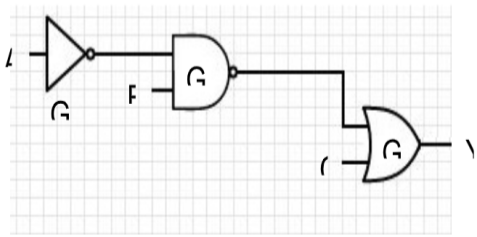


Figure 1.A:
This experiment is done to analyze Figure 1.A and as well as its truth table.

Logic Expression:

Y = (Not A) NAND B) OR C

Results of Part I

Logic Expression: = A'B'C' + A'B'C + A'BC + AB'C' + AB'C +ABC' +ABC

Truth Table:

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

PART II

Experiments of Part II

In this experiment, we designed a circuit in Quartus Prime to assess varying inputs via x1 and x2, resulting in the determination of our output f. Specifically, switch 1 was allocated to input x1, while input x2 was linked to switch 0. The resulting output f was directed to LED 0.

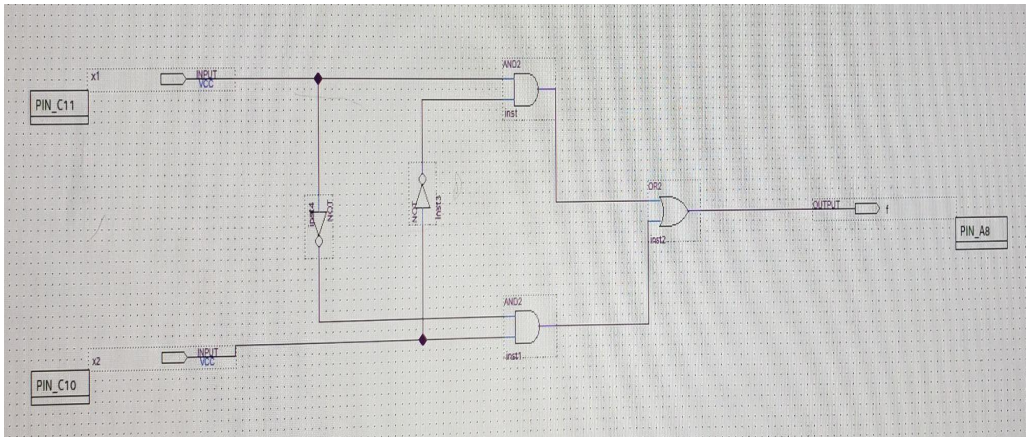


Figure 2.A

Results of Part II

After conducting on-board testing, it became evident that the LED was only activated when one of the inputs was set to a high (1) state. This behavior is clearly illustrated in Figures 2.2 and 2.3. In contrast, when both inputs were either low (0) or high (1), the LED remained unilluminated. This can be observed in Figures 2.4 and 2.5.

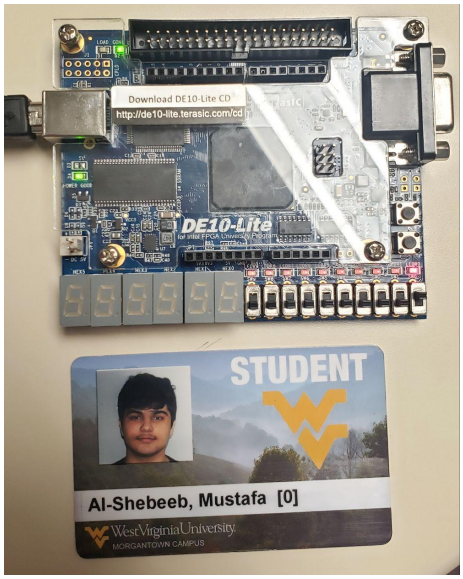


Figure 2.B

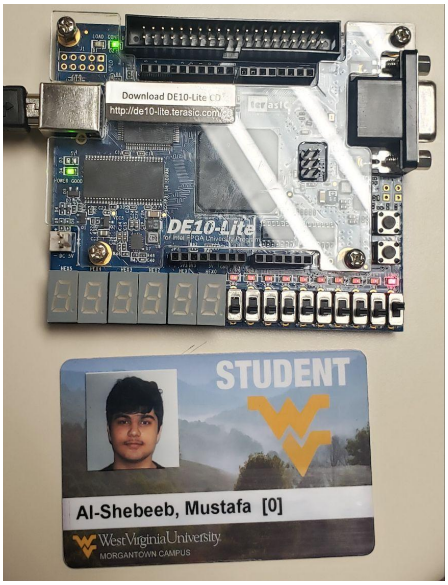


Figure 2.C

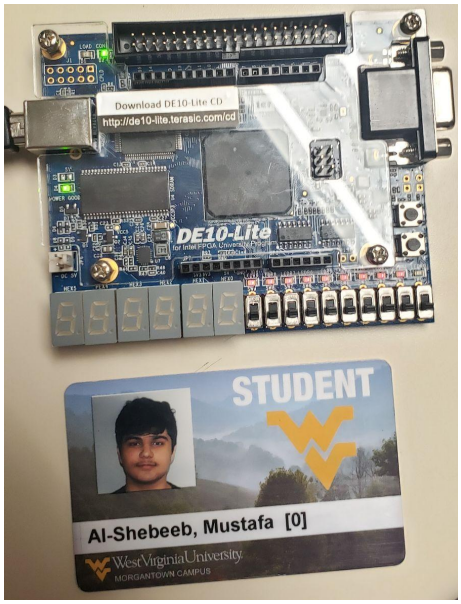


Figure 2.D

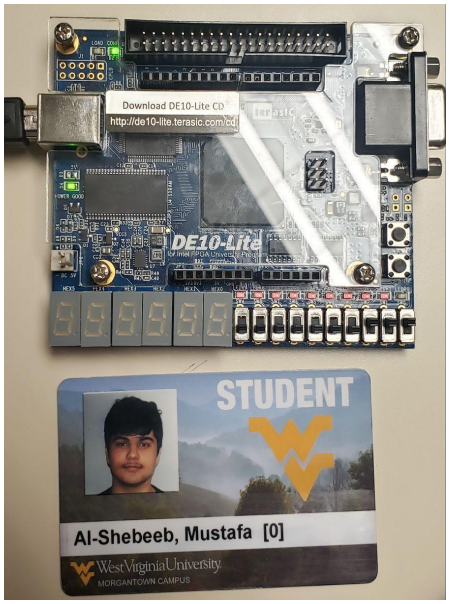


Figure 2.E

Truth Table:

X1	X2	F
0	0	0
0	1	1
1	0	1
1	1	0

F = (X1' AND X2 OR (X2' AND X1')
F = (NOT X1') AND X2) OR (X1 AND)

This expression can be simplified into an AND gate followed by an Inverter

PART III

Experiments of Part III

This experiment involved devising a circuit that produces a high output (illuminates the LED) when a 3-bit binary number (A, B, C, truth table in Figure 3.6) is divisible by three. This circuit is depicted in Figure 3.1.

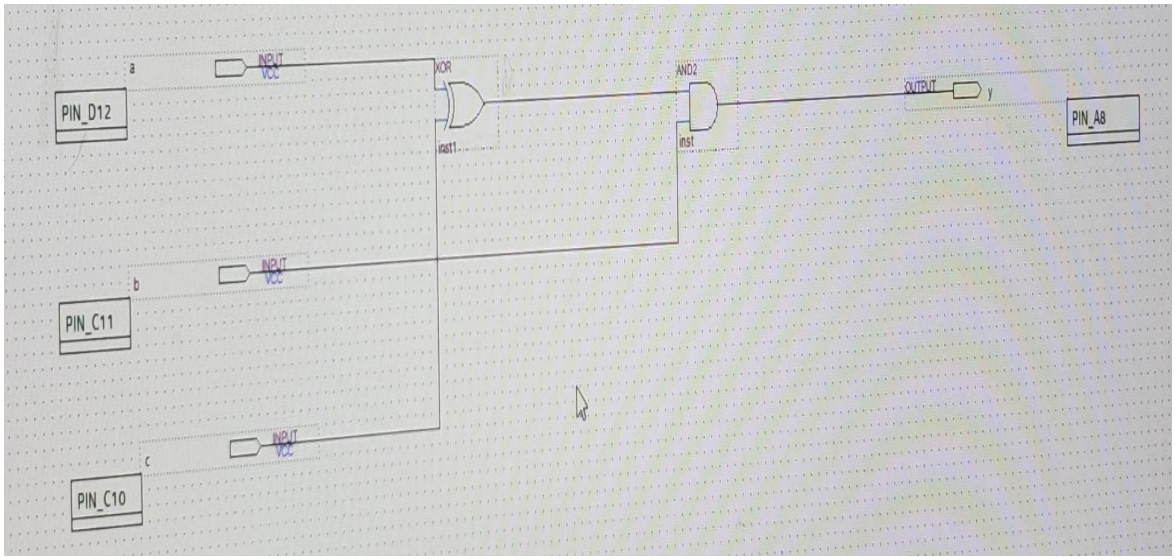


Figure 3.A:

We made the following circuit by using a two-input XOR gate (input A and input C) and connected it to a two-input AND gate.

Results of Part III

The LED illuminated in two scenarios: when inputs A, B, C were 0 1 1 and 1 1 0. This aligns with the correct decimal equivalents of 011 and 110, being 3 and 6 respectively. Both 3 and 6 are divisible by 3, validating our circuit's functionality. Figures 3.2 and 3.3 depict this with the lit LED. However, for inputs 1 0 1 (5 in decimal, shown in Figure 3.4) and 1 1 1 (7 in decimal, shown in Figure 3.5), the LED remains unlit. This is due to 5 and 7 not being divisible by 3.

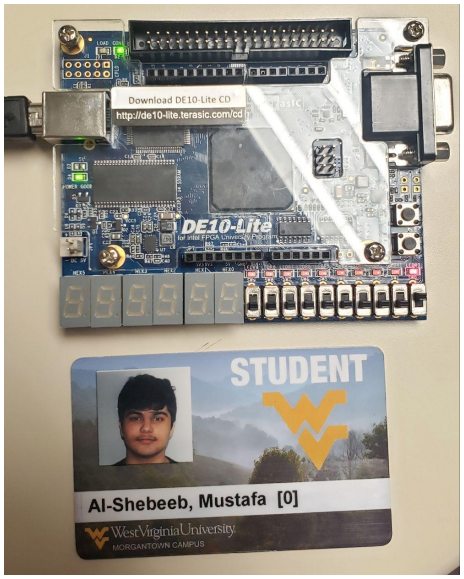


Figure 3.B

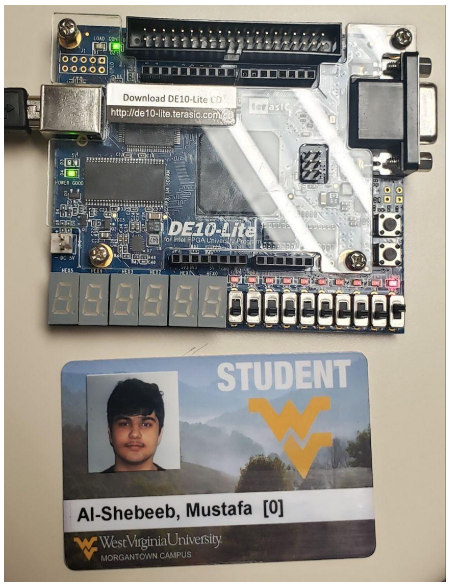


Figure 3.C

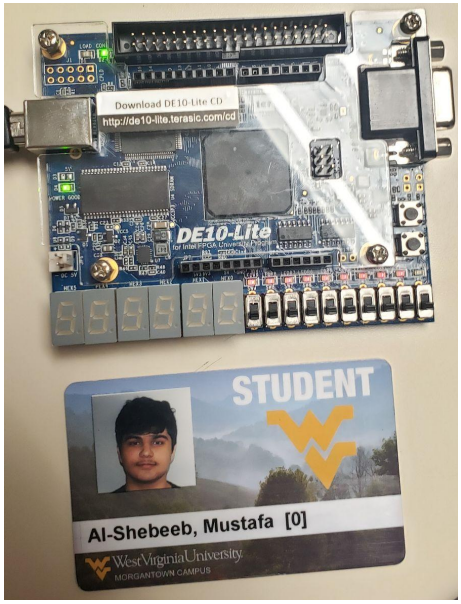


Figure 3.D

Figure 3.E

Decimal #	A	B	C	Y
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	1	0	0	0
4	0	1	1	1
5	1	1	0	1
6	1	0	1	0
7	1	1	1	0

Figure 3.F

The depicted figure illustrates a high output (LED lit) for input binary digits corresponding to 3 and 6. The associated logic expressions are provided below:
Visual: Y = B AND (A XOR C)
Truth Table Minterm Analysis:
Y = A'BC + ABC'
B(AC' + A'C) = B(A + C)

Conclusion

Logic Gates offer an effective way to solve problems based on inputs for desired outputs. The DE10 Lite board is valuable for showcasing inputs through switches and outputs via LEDs. To enhance the learning experience, a sequential approach of covering content, working through it, and then moving forward would be beneficial. This method would improve focus and understanding compared to completing the entire lab all at once.

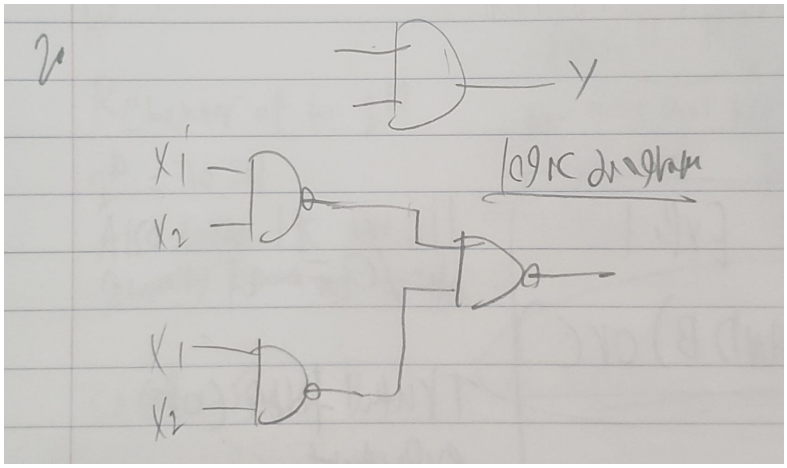
Post Lab Questions for Lab 1

1)

Decimal #	A	B	C	Y
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0

2) In the diagram below the circuit uses 3 NAND gates to perform the function of an AND gate. The two left-most NAND gates use two similar inputs, x1 and x2, the output would be x1'x2'. This output is then channeled through the last NAND gate on the right which produces output x1 x2.

Logic Expression: Y = A'BC' + AB'C' + ABC'



3) In Part 3, it's crucial to utilize the visual expression Y = B AND (A XOR C) rather than the logic expression obtained from minterms. This visual approach aids in building the function on a breadboard, optimizing resource usage due to fewer inputs and terms.

Pre-Lab Questions for Lab 2:

1)A seven-segment display shows numbers (0-9) and characters using 8 LEDs. There are two types: Common Anode and Common Cathode displays. To light up specific segments, provide a logic low for lit segments and 5+ volts for off segments.

2) Hexadecimal is a 16-bit numbering system tht uses decimal numbers and alphabetical letters to represent 0-15. Decimal numbers from 0-9 are represented by 0-9 and 10-15 are in A-F.

Lab 2: PLD's, FPGA's & Seven Segment Displays

Date performed: 8/29/2023

Introduction:

A Field Programmable Gate Array (FPGA) is a programmable hardware device that utilizes an array of logic gates for configuration. The DE10 Lite board was utilized in conjunction with Quartus Prime software to control a seven-segment display unit. The primary function of this software was to convert binary input received from four input switches, resulting in 16 possible input combinations, into corresponding outputs displayed on the seven-segment unit. It is essential to distinguish between two types of Seven Segment Displays: common anode and common cathode. The key distinction lies in how these displays respond to voltage levels. In the case of common anode displays, a logic high (1) signal turns off a segment, while a logic low (0) signal turns it on.

PART I

Experiment of Part I

For this part of the lab we designed a seven-segment decoder. We did this in order to display hexadecimal numbers using switches to force a specific value to the output. Hexadecimal numbers are base 16 numbers and therefore have 16 different symbols to represent one digit. Table 1 maps hexadecimal numbers to their binary equivalent.

Added the table and figures from the lab handout.

Decimal Number	Hexadecimal Number
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e

15	f

Table 1.A: Decimal to Hexadecimal conversions

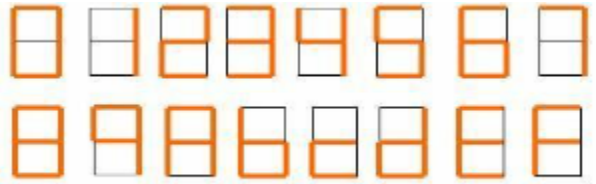


Figure 1.B: 0 through F in Seven Segment Display

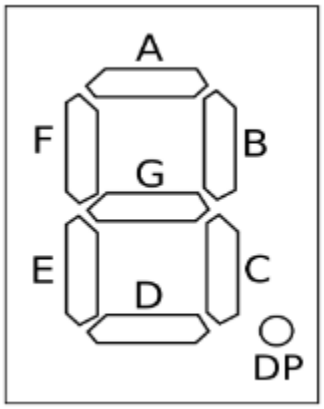


Figure 1.C: Seven Segment Display light segments

In this section, we were tasked with creating a table that has the combination of four inputs (W, X, Y, Z) and the corresponding activation or deactivation of segments, represented in hexadecimal form. We followed the previous diagram in order to correctly write the hexadecimal values using the Seven Segment Display **(0 ⍰ LIGHT SEGMENT ON, 1 ⍰ LIGHT SEGMENT OFF).**

W	X	Y	Z	a	b	c	d	e	f	g	Display
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	1	0	0	1	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0	2
0	0	1	1	0	0	0	0	1	1	0	3
0	1	0	0	1	0	0	1	1	0	0	4
0	1	0	1	0	1	0	0	1	0	0	5
0	1	1	0	0	1	0	0	0	0	0	6
0	1	1	1	0	0	0	1	1	1	1	7

1	0	0	0	0	0	0	0	0	0	0	8
1	0	0	1	0	0	0	1	1	0	0	9
1	0	1	0	0	0	0	1	0	0	0	A
1	0	1	1	1	1	0	0	0	0	0	B
1	1	0	0	1	1	1	0	0	1	0	C
1	1	0	1	1	0	0	0	0	1	0	D
1	1	1	0	0	1	1	0	0	0	0	E
1	1	1	1	0	1	1	1	0	0	0	F

Figure 1.1: Inputs with hex representation on a common anode seven-segment display

Results of Part I

The table is now complete and these are the resulting minterms for each Column

Exclamation(!) before letter represents NOT

a=WXY!Z, !WX!Y!Z, W!XYZ, WX!YZ, WX!YZ

b=!WX!YZ, !WXY!Z, W!XYZ, WX!Y!Z, WXY!Z, WXYZ

c=!W!XY!Z, WX!Y!Z, WXY!Z, WXYZ

d=!W!X!YZ, !WX!Y!Z, !WXYZ, W!X!YZ, W!XY!Z, WXYZ

e=!W!X!YZ, !W!XYZ, !WX!Y!Z, !WX!YZ, !WXYZ, W!X!YZ

f=!W!X!YZ, !W!XY!Z, !W!XYZ, !WXYZ, WX!Y!Z, WX!YZ

g=!W!X!Y!Z, !W!X!YZ, !WXYZ

PART II

Experiments of Part II

Based on the minterms we obtained in Part I, we can now expand the expression and incorporate it into our VHDL file within Quartus Prime. This step enables us to configure the FPGA board to function as desired, with its four inputs correspondingly representing their hexadecimal equivalents.

Results of Part II

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity binary2hex is
5  port (
6   W: in std_logic; --input
7   X: in std_logic; --input
8   Y: in std_logic; --input
9   Z: in std_logic; --input
10  a: out std_logic; --output
11  b: out std_logic; --output
12  c: out std_logic; --output
13  d: out std_logic; --output
14  e: out std_logic; --output
15  f: out std_logic; --output
16  g: out std_logic --output
17 );
18 end binary2hex;
19
20 architecture behavior of binary2hex is
21 begin
22
23  a <= (not W and not X and not Y and Z) or (not W and X and not Y and not Z) or (W and not X and Y and Z) or (W and X and not Y and not Z) or (W and X and not Y and Z);
24  b <= (not W and X and not Y and Z) or (not W and X and Y and not Z) or (W and not X and Y and Z) or (W and X and not Y and not Z) or (W and X and Y and not Z) or ( W and X and Y and Z);
25  c <= (not W and not X and Y and not Z) or (W and X and not Y and not Z) or (W and X and Y and not Z) or (W and X and Y and Z);
26  d <= (not W and not X and not Y and Z) or (not W and X and not Y and not Z) or (not W and X and Y and Z) or (W and not X and not Y and Z) or (W and not X and Y and not Z) or (W and X and Y and Z);
27  e <= (not W and not X and not Y and Z) or (not W and not X and Y and Z) or (not W and X and not Y and not Z) or (not W and X and not Y and Z) or (not W and X and Y and Z) or (W and not X and not Y and Z);
28  f <= (not W and not X and not Y and Z) or (not W and not X and Y and not Z) or (not W and not X and Y and Z) or (not W and X and Y and Z) or (W and X and not Y and not Z) or (W and X and not Y and Z);
29  g <= (not W and not X and not Y and not Z) or (not W and not X and not Y and Z) or (not W and X and Y and Z);
30
31 end behavior;
```

Figure 2.1: Code snippet for seven-segment display.

PART III

Experiments of Part III

In this experiment, we utilized the code from the preceding section to showcase hexadecimal values on the DE10 Lite Board. The pin assignments for the seven-segment display are outlined as follows:

Signal Name	FPGA Pin No.	Description	I/O Standard
SW0	PIN_C10	Slide Switch[0]	3.3-V LVTTL
SW1	PIN_C11	Slide Switch[1]	3.3-V LVTTL
SW2	PIN_D12	Slide Switch[2]	3.3-V LVTTL
SW3	PIN_C12	Slide Switch[3]	3.3-V LVTTL
SW4	PIN_A12	Slide Switch[4]	3.3-V LVTTL
SW5	PIN_B12	Slide Switch[5]	3.3-V LVTTL
SW6	PIN_A13	Slide Switch[6]	3.3-V LVTTL
SW7	PIN_A14	Slide Switch[7]	3.3-V LVTTL
SW8	PIN_B14	Slide Switch[8]	3.3-V LVTTL
SW9	PIN_F15	Slide Switch[9]	3.3-V LVTTL

Table 3: Switch (Input) Pin Map

Signal Name	FPGA Pin No.	Description	I/O Standard
HEX00	PIN_C14	Seven Segment Digit 0[0]	3.3-V LVTTL
HEX01	PIN_E15	Seven Segment Digit 0[1]	3.3-V LVTTL
HEX02	PIN_C15	Seven Segment Digit 0[2]	3.3-V LVTTL
HEX03	PIN_C16	Seven Segment Digit 0[3]	3.3-V LVTTL
HEX04	PIN_E16	Seven Segment Digit 0[4]	3.3-V LVTTL
HEX05	PIN_D17	Seven Segment Digit 0[5]	3.3-V LVTTL
HEX06	PIN_C17	Seven Segment Digit 0[6]	3.3-V LVTTL
HEX07	PIN_D15	Seven Segment Digit 0[7], DP	3.3-V LVTTL

Table 4: 7-Segment Display (Outputs) Pin out of the right-most display

Once we accurately assigned pins for our four inputs (W, X, Y, and Z) to switches 3, 2, 10, and 6, respectively, the seven-segment display successfully illuminated each LED segment, precisely indicating the correct digit as intended.

Results of Part III

The first picture is Binary 1010 input using the switches to display A in hex. The second picture is Binary 1011 input using the switches to represent B in hex.

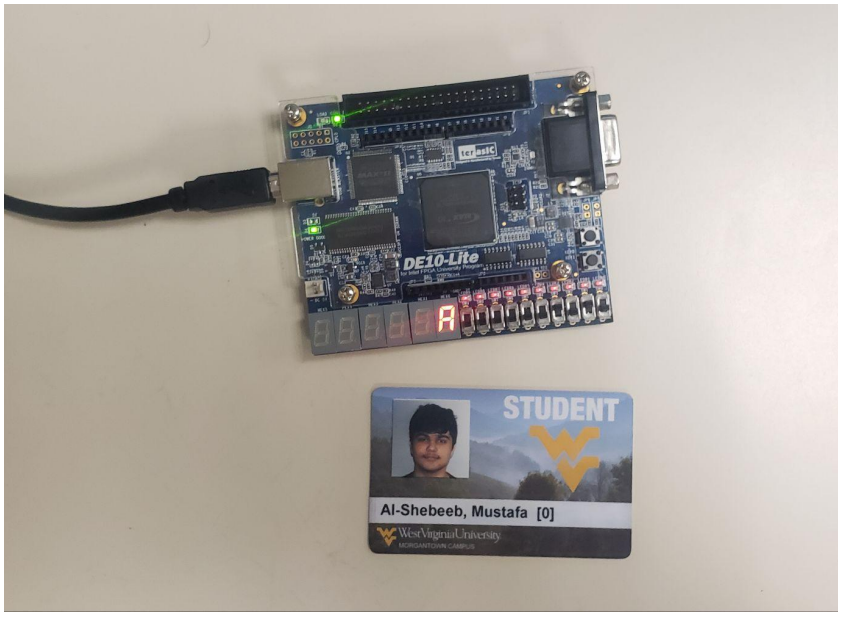


Figure 3.1: W X Y Z = 1010, A in hex representation.

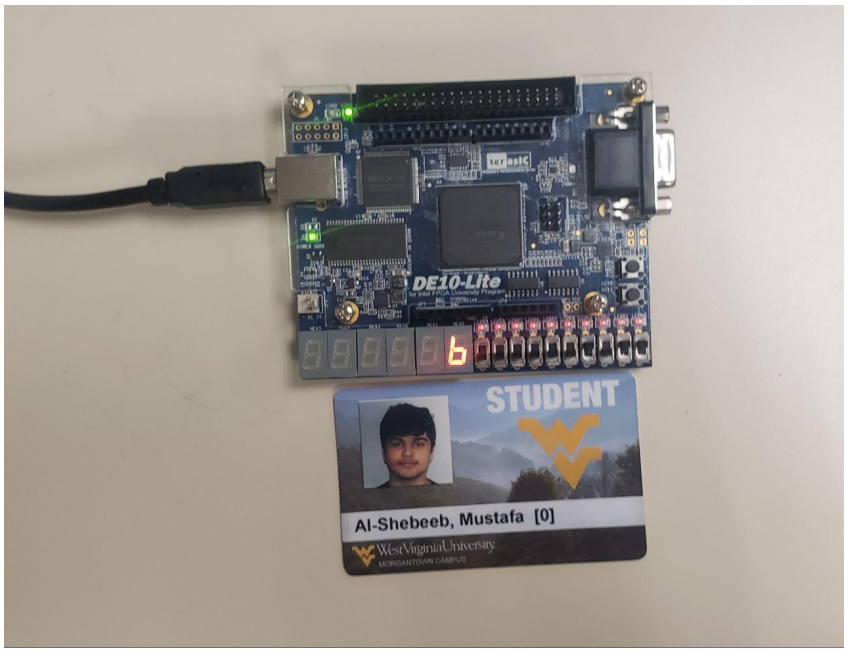


Figure 3.2: W X Y Z = 1011, B in hex representation.

Conclusion

This lab proved to be remarkably engaging and captivating. Witnessing the input switches seamlessly translate into a digital output on the seven-segment display was a truly fascinating experience. With the four input switches symbolizing four binary digits, users could effortlessly convert the binary input into a clear hexadecimal representation displayed on the seven-segment display, showcasing effective functionality.

Post Lab Questions for Lab 2

1)

Common Anode 7-Segment Display:
All the plus (+) sides of the LED segments are connected together. You control each segment by turning on the negative (-) side (the cathode). Common anode displays are easier to connect to microcontrollers and use less power.

Common Cathode 7-Segment Display:
All the minus (-) sides of the LED segments are connected together. You control each segment by turning on the positive (+) side (the anode). Common cathode displays are often used when you want to switch segments rapidly (multiplexing).

2)

W	X	Y	Z	a	b	c	d	e	f	Hex #
0	1	1	1	1	1	1	0	0	0	7

This table illustrates the logic values applied to segments based on inputs (W, X, Y, and Z) along with their corresponding hexadecimal digits. In a common cathode configuration, segments illuminate when supplied with a logic HIGH signal. To display the digit 7, segments a, b, and c must be illuminated.

Pre Lab Questions for Lab 3

1) FPGA stands for "Field-Programmable Gate Array." It's a type of integrated circuit that can be programmed or configured after manufacturing to perform various digital logic tasks. FPGA devices contain an array of configurable logic blocks and interconnects, allowing users to create custom digital circuits and implement specific functions or algorithms.

2) FPGAs (Field-Programmable Gate Arrays) are preferred over GAL (Generic Array Logic) chips for various reasons. FPGAs are incredibly flexible and can handle complex digital tasks, making them suitable for a wide range of applications, from signal processing to system-on-chip designs. They can be reprogrammed multiple times, allowing for adaptation and iterative development. FPGAs offer high performance, operating at high clock

frequencies and supporting parallel processing, which is beneficial for tasks like image processing and hardware acceleration.

Lab 3: Designing a Binary Arithmetic Operator Using Logic Design

Date performed: 9/5/2023

Introduction:

Binary subtraction is a fundamental operation in computer arithmetic. In this lab, we aim to develop two software programs using Quartus Prime, which will function as essential components in binary subtraction: a half subtractor and a full subtractor. A half subtractor is a digital circuit that subtracts two binary bits and provides two outputs: the difference and the borrow. A full subtractor, on the other hand, extends the functionality of the half subtractor by including an additional input, the borrow, and the difference from the previous subtraction.

Part I

Experiment of Part I

The aim of this experiment is to design a half subtractor, a fundamental digital circuit that takes two inputs, where input B is subtracted from input A (A - B). This half subtractor will yield two distinct outputs: The Difference and the Borrow Indicator. The difference output will provide the result of the subtraction operation, which is the numerical difference between input A and input B. The Borrow Indicator serves as a logic high (1) when a borrow operation occurs during the subtraction.

Results of Part I

A		B	Borrow Out (Bout)	Difference (Diff)
0	-	0	0	0
0	-	1	1	1
1	-	0	0	1
1	-	1	0	0

Table 1.A: Half Subtractor Truth Table.

The truth table illustrates the results of various subtraction operations, highlighting the computed differences (Diff) and the presence of a borrow-out (Bout). In three cases, no borrowing is necessary when subtracting smaller numbers from larger ones. However, a crucial scenario arises when subtracting 1 from 0, as it necessitates borrowing due to the absence of negative numbers in our number system. This borrow-out concept is fundamental in binary arithmetic, ensuring accurate subtractions when the subtrahend exceeds the minuend.

Analyzing the truth table allows us to derive equations for Bout and Diff using the minterm method, enabling a systematic representation of these binary outputs in terms of input variables and logical combinations. This method ensures the integrity and consistency of numerical operations within the constraints of our number system.

Bout = NOT A AND B

Diff = (NOT A AND B) OR (A AND NOT B) = A XOR B

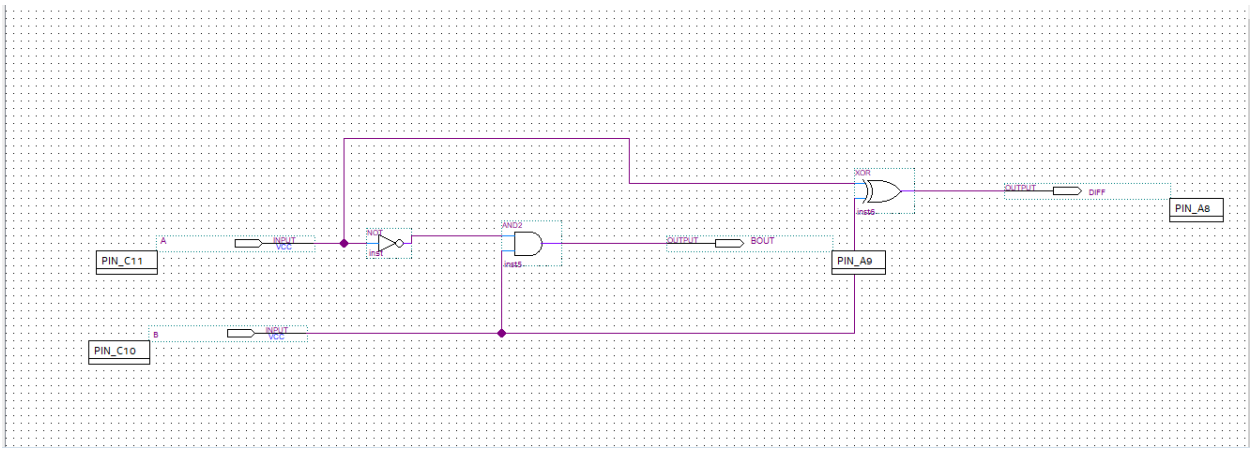


Figure 1.B: Circuit for Half Subtractor.

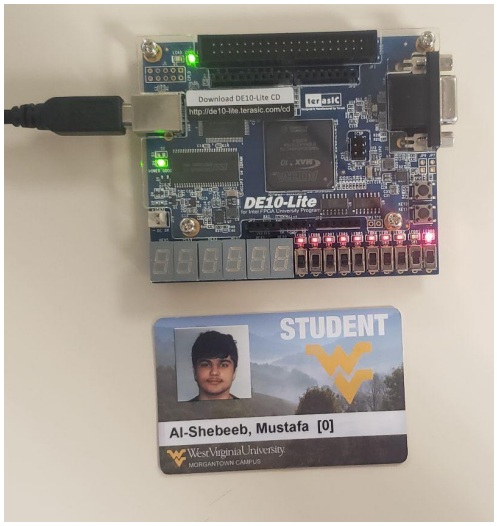


Figure 1.C: 1 - 0 = diff = 1, bout = 0.

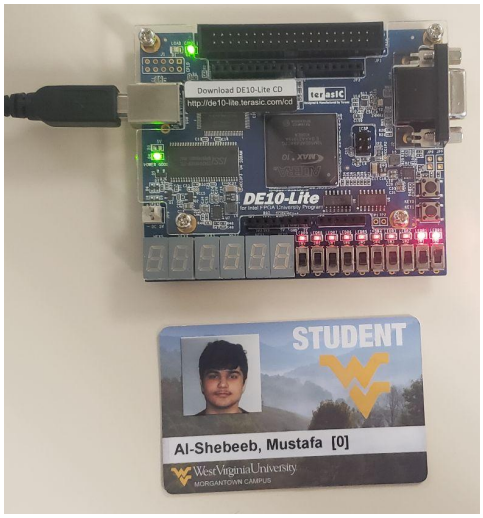


Figure 1.D: 0 - 1 = diff = 1, bout = 1.

Part II

Experiment of Part II

The objective is to design a full subtractor, a digital circuit that accepts three inputs: A, B, and C. The subtraction is performed in two stages. First, A is subtracted from B (A - B), and then the result is further subtracted from C (A - B - C). This full subtractor yields two essential outputs: The Difference and the Borrow Indicator. The difference provides the final result of the subtraction process, which is the difference between A, B, and C (A - B - C). The borrow indicator serves as an indicator, displaying a logic high (1) when a borrow operation occurs during any of the two subtraction stages.

Results of Part II

The truth table provided for the three-input full subtractor showcases its behavior across eight different input combinations, in contrast to the half subtractor with only two inputs. Notably, in four significant cases, a borrow-out (Bout) is necessary to facilitate accurate binary subtraction. These cases involve subtrahends (B and C) that are larger than the minuend (A) for specific bit positions. The activation of the borrow-out mechanism ensures the full subtractor adjusts the subtraction process, crucial in maintaining precision and consistency in binary arithmetic and digital circuit design.

A		B		C	Difference (Diff)	Borrow Out (Bout)
0	-	0	-	0	0	0
0	-	0	-	1	1	0
0	-	1	-	0	1	1
0	-	1	-	1	0	0
1	-	0	-	0	1	1
1	-	0	-	1	0	0
1	-	1	-	0	0	1
1	-	1	-	1	1	1

Figure 2.A: Full Subtractor Truth Table.

By analyzing the provided truth table, we can derive equations for both Bout (borrow-out) and Diff (difference) using the minterm method. This method allows us to express these binary outputs in terms of the input variables and their logical combinations, providing us with a clear and systematic representation of the subtraction process.

Bout = (NOT A AND B AND NOT C) OR (A AND NOT B AND NOT C) OR (A AND B AND NOT C) OR (A AND B AND C)

Diff = (NOT A AND NOT B AND C) OR (NOT A AND B AND NOT C) OR (A AND NOT B AND NOT C) OR (A AND B AND C)

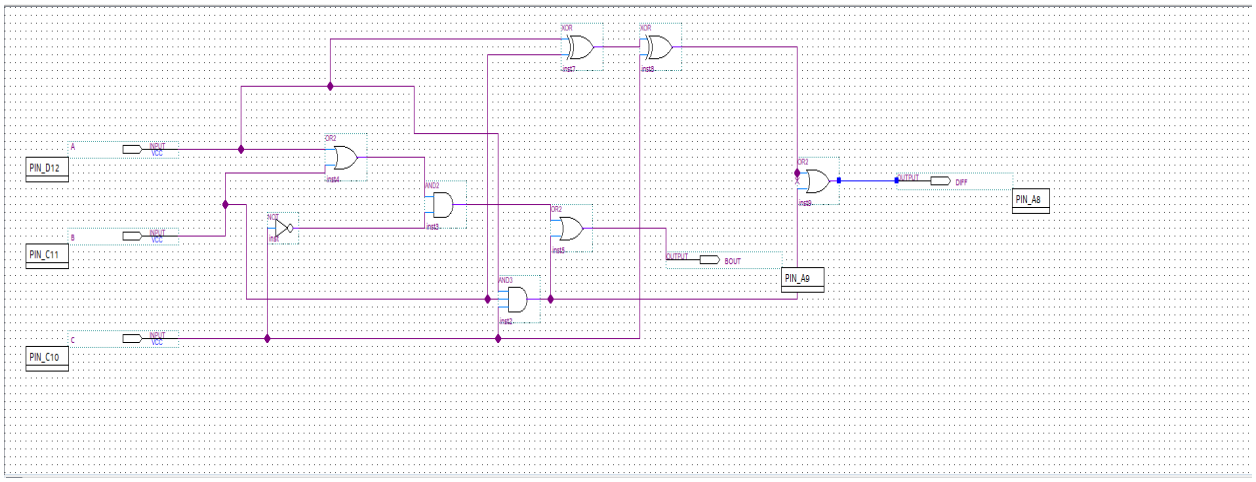


Figure 2.B: Circuit for Full Subtractor.

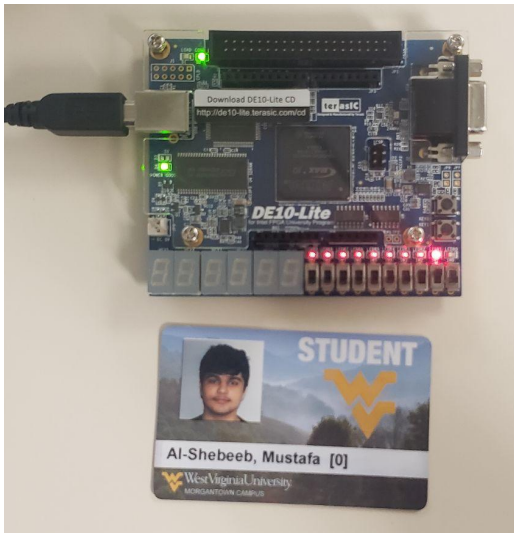


Figure 2.C: 1-1-0 = diff = 0, bout = 1.

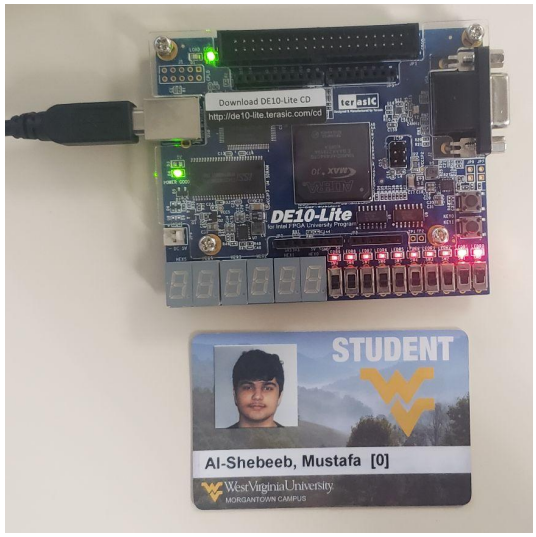


Figure 2.D: 1-1-1 = diff = 1, bout = 1.

Conclusion

I thoroughly enjoyed this lab where we delved into the world of subtractors, both for 2 and 3 bits. It's truly fascinating to witness how FPGAs empower us to implement a wide range of software functions effectively. This experiment vividly demonstrated the versatility and potential of FPGAs in digital logic design.

Post-Lab Questions

1)

100 - 001 = Bout: 011, Diff: 011

111 - 100 = Bout: 000, Diff: 011

Pre-Lab Questions

1) VHDL stands for "VHSIC Hardware Description Language." VHSIC, in turn, stands for "Very High-Speed Integrated Circuit." VHDL is a programming language used for describing the behavior and structure of electronic systems and digital circuits. It is primarily used in the design and simulation of digital systems, especially in the field of electronic design automation (EDA) and FPGA (Field-Programmable Gate Array) programming.

2) Signal Variables: declared as “std_logic”, Variable Variables: declared as “integer(any name) = 0.

3) In VHDL, single quotes (') are used to represent local variables, which may or may not be integers. So, to represent a single bit '0', you would use single quotes like this: '0'. Double quotes ("") are not used to represent single bits in VHDL; they are typically used for string literals or text.

Lab 4: Introduction to VHDL

Date performed: 9/12/2023

Introduction:

Creating schematic files in Quartus Prime is indeed a valuable approach for designing circuits, especially when they are relatively simple. However, as circuits become more intricate, the process of manually placing and connecting individual logic gates can become quite laborious and prone to errors. To address this challenge, this lab introduces a powerful solution: the use of VHDL (Very High-Speed Integrated Circuit Hardware Description Language). VHDL allows us to describe our logical diagrams and circuits using code, which offers several advantages.

By coding our logic expressions with VHDL, we can precisely define the behavior of our circuits using minterms and logical expressions. This method not only simplifies the circuit creation process but also enhances our ability to debug and modify designs efficiently. It's like having a digital blueprint that captures the essence of the circuit's functionality without the need to manually place each gate.

In essence, VHDL empowers us to translate our circuit designs into code, offering a more systematic and scalable approach to complex circuit development. This approach not only saves time but also ensures accuracy and consistency, making it an invaluable tool in the realm of digital design and engineering.

Part I

Experiment of Part I

In this segment of our project, we're focusing on implementing the functionality of an XOR gate. This involves connecting two input switches to our circuit.

Now, let's dive into the XOR gate's behavior. Its distinctive trait is that it produces a high output only when one of its inputs is high. So, when we activate one of the switches, our LED will illuminate. This demonstration effectively illustrates the fundamental concept of an XOR gate, showcasing its ability to detect and respond to a specific input condition, which is crucial in digital logic and circuit design.

Inputs		Output
x1	x2	f
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1.A: Truth Table for Part I

The provided table (Figure 1.1) replicates the behavior of an XOR gate, showcasing the relationship between its inputs (x1 and x2) and its output (f).

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity Part1 is
5
6      port(
7            x1: in std_logic;
8            x2: in std_logic;
9            f: out std_logic
10           );
11
12
13  end part1;
14
15  architecture behavior of Part1 is
16
17  begin
18
19  f <= (not x2 and x1) or (not x1 and x2);
20
21  end behavior;
```

Figure 1.B: VHDL Code for XOR gate

Results of Part I

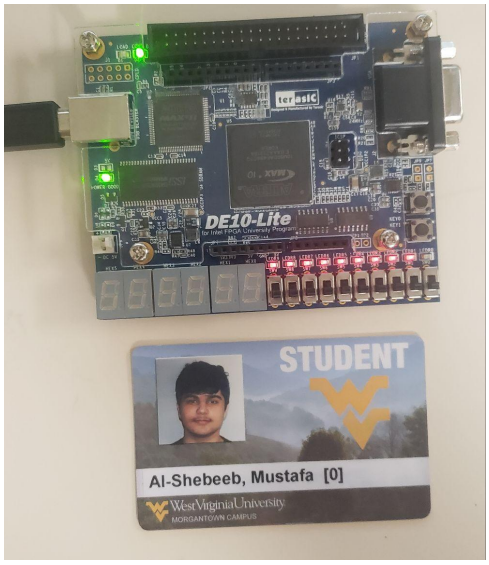


Figure 1.C: x1=0, x2=0, f=0.

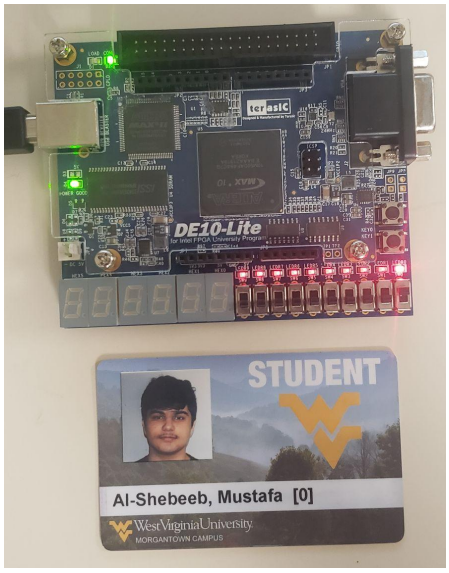


Figure 1.D: x1=0, x2=1, f=1

Part II

Experiment of Part II

In the second phase of our project, our objective was to implement a combinational logic circuit using VHDL. The approach closely mirrored the one we used previously. We initiated the process by extracting minterms from the truth table we had generated. This step was instrumental in precisely defining the logic for our circuit. It allowed us to systematically capture the behavior of the circuit.

Next, we transformed the Sum-of-Products (SOP) expressions into a more human-readable format, aiding our understanding of the circuit's functionality. This was a critical step in articulating how the circuit operates. Finally, armed with this well-defined logic, we transcribed our design into VHDL code, creating a digital representation that could be easily interpreted and executed by Quartus Prime.

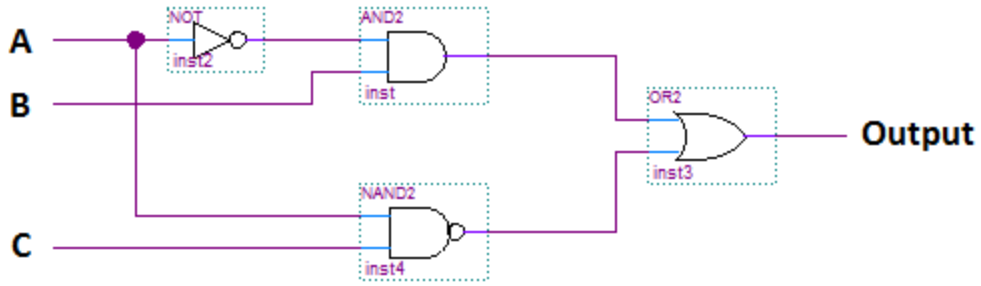


Figure 2.A: Logic Diagram

Inputs			Intermediary			Output
A	**B**	**C**	**G1=**	**G2=**	**G3=**	**Y=**
0	0	0	1	0	1	1
0	0	1	1	0	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	0	0	1	1
1	0	1	0	0	0	0
1	1	0	0	0	1	1
1	1	1	0	0	0	0

Figure 2.B: VHDL Truth Table

y=A'B'C' + A'B'C + A'BC'+A'BC+AB'C' + ABC'

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity Part2 is
5
6      port(
7            A: in std_logic;
8            B: in std_logic;
9            C: in std_logic;
10           Y: out std_logic
11           );
12
13
14  end Part2;
15
16  architecture behavior of Part2 is
17
18  begin
19
20  Y <= (NOT A AND B) OR (A NAND C);
21
22  end behavior;
```

Figure 2.C: VHDL Code

Results of Part II

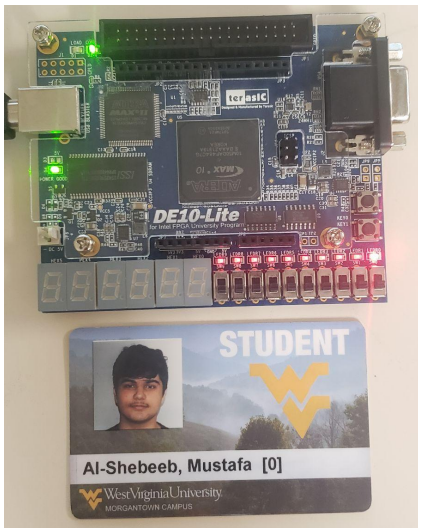


Figure 2.D: 0,0,0=1

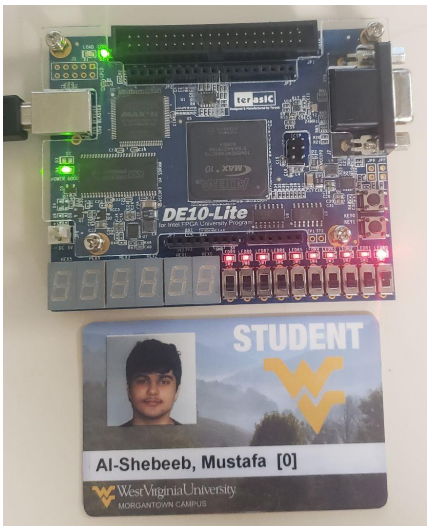


Figure 2.E: 0,0,1=1

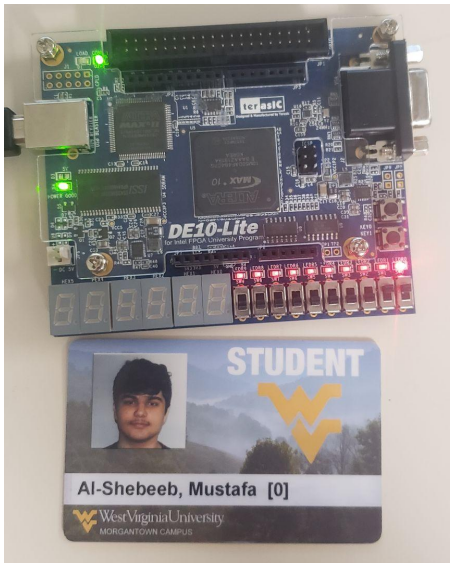


Figure 2.F: 0,1,1=1

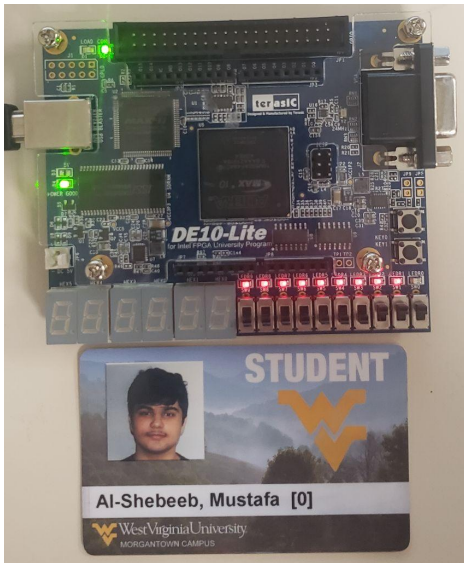


Figure 2.G: 1,1,1=0

Part III

Experiment of Part III

For part three we were asked to develop software that could successfully add two three-bit binary numbers. This was similar to the subtractor in the previous lab in the sense that it has a resulting output and a borrow-out output LED.

Inputs			Outputs	
A	B	C	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 3.A: VHDL Code

Equations:

Sum = (NOT A AND NOT B AND C) OR (NOT A AND B AND NOT C) OR (A AND NOT B AND NOT C) OR (A AND B AND C)

= (A XOR B) XOR C) OR ABC

Carry = (NOT A AND B AND C) OR (A AND NOT B AND C) OR (A AND B AND NOT C) OR (A AND B AND C)

```
library ieee;
use ieee.std_logic_1164.all;

entity Part3 is

    port(
          x1: in std_logic;
          x2: in std_logic;
          x3: in std_logic;
          x4: in std_logic;
          x5: in std_logic;
          x6: in std_logic;
          f1: out std_logic;
          f2: out std_logic;
          f3: out std_logic;
          f4: out std_logic
          );


end Part3;

ArchitEcture behavior of Part3 is
Signal c1: std_logic;
Signal c2: std_logic;

begin

f1 <= (x1 XOR x4);
f2 <= (NOT x2 AND NOT x5 AND c1) OR (NOT x2 AND x5 AND NOT c1) OR (x2 AND NOT x5 AND NOT c1) OR (x2 AND x5 AND c1);
f3 <= (NOT x3 AND NOT x6 AND c2) OR (NOT x3 AND x6 AND NOT c2) OR (x3 AND NOT x6 AND NOT c2) OR (x3 AND x6 AND c2);
f4 <= (NOT x3 AND x6 AND c2) OR (x3 AND NOT x6 AND c2) OR (x3 AND x6 AND NOT c2) OR (x3 AND x6 AND c2);
c1 <= (x1 AND x4);
c2 <= (NOT x2 AND x5 AND c1) OR (x2 AND NOT x5 AND c1) OR (x2 AND x5 AND NOT c1) OR (x2 AND x5 AND c1);

end behavior;
```

Figure 3.B: VHDL Code

Results of Part III

We have achieved the successful addition of two three-bit binary numbers through a systematic process using signals. The addition process involves summing up the values

digit by digit, starting with the first column, which includes X1 and X4. This computation provides us with two significant outcomes:

F1 (Sum): F1 represents the result of adding the first column. It is the outcome of the addition operation between X1 and X4.

C1 (Carry): C1 signifies whether there is a carry-over to the next column during addition. It acts as a crucial indicator for the next step, which involves adding X5 and X2.

The equations governing the calculation of the sum (F) and carry (C) are derived through a systematic process that ensures precision and reliability in our binary addition. This approach allows us to confidently tackle more complex binary arithmetic, laying a solid foundation for advanced digital calculations and designs.

Binary Addition Inputs						**Outputs**			
1st Row Inputs			**2nd Row Inputs**			**Sums**			
X_3	**X_2**	**X_1**	**X_6**	**X_5**	**X_4**	**F_4**	**F_3**	**F_2**	**F_1**
0	1	1	1	0	1	1	0	0	0
0	1	0	1	1	1	1	0	0	1

Figure 3.C: VHDL Code

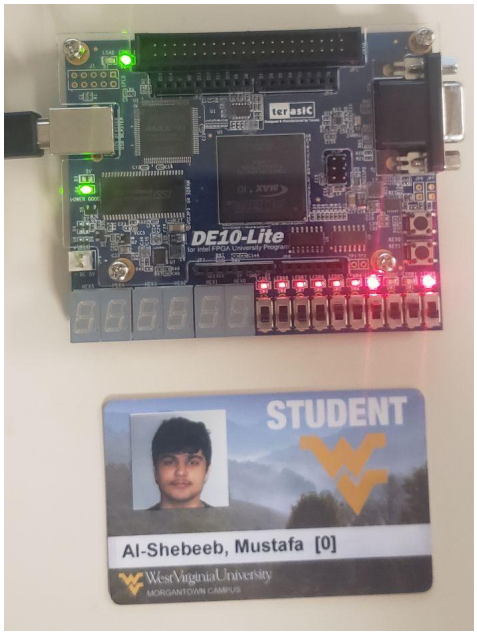


Figure 3.4: 111+010=1001

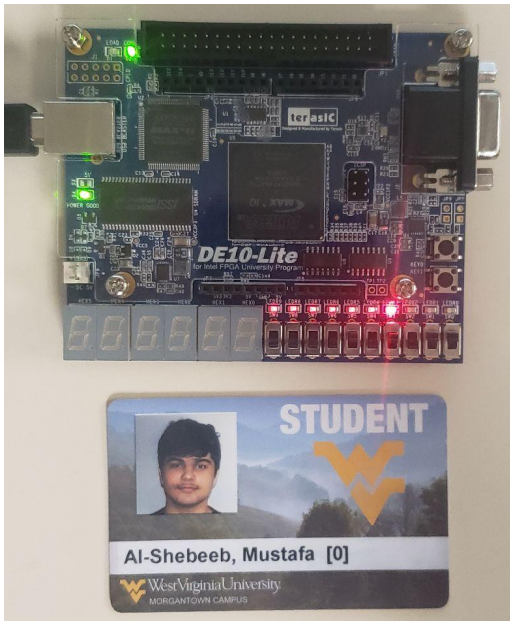


Figure 3.5: 011+101=1000

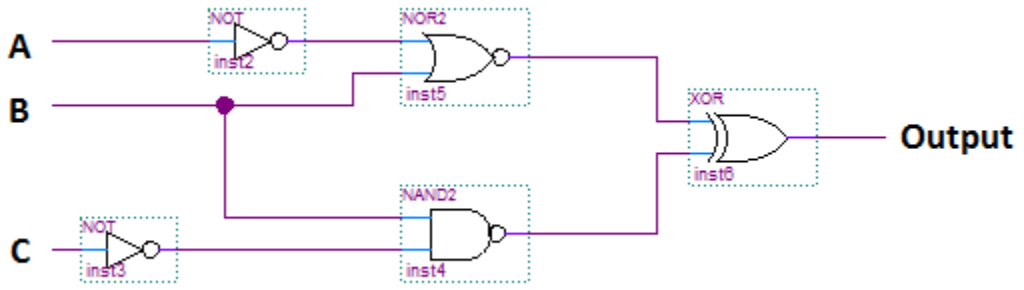
Conclusion

This lab serves as an effective introduction to VHDL programming, a powerful tool in the realm of digital design. What added to its practicality for me was its resemblance to conventional programming languages like Java. VHDL's syntax allows us to declare variables in a manner akin to what I'm accustomed to. Moreover, the lab offered valuable insight into the world of combinational logic, showcasing how VHDL can be harnessed to implement complex logic functions. The process of constructing logical expressions using minterms was particularly illuminating. It demonstrated how we can systematically define and implement logic, bridging the gap between abstract concepts and tangible digital circuitry.

In essence, this lab not only introduces us to VHDL but also highlights its familiarity to conventional programming, making it a valuable tool for both beginners and experienced digital designers. It empowers us to bring precision and structure to complex logic, laying the foundation for more advanced digital endeavors.

Post-Lab Questions

1)



Inputs	Output Steps					
ABC	G1=NOT A	G2=G1 NOR B	G3=NOT C	G4=B NAND G3	G5=G2 XOR G4	Output (y)=G5
000	1	0	1	1	1	1
001	1	0	0	1	1	1
010	1	0	1	0	0	0
011	1	0	0	1	1	1

100	0	1	1	1	0	0
101	0	1	0	1	0	0
110	0	0	1	0	0	0
111	0	0	0	1	1	1

2)

11100	10111
+ 01001	+ 01011
Carry: 10000	Carry: 11110
Sum: 100101	Sum: 100010

Pre-Lab Questions

1) Logic simulation refers to the process of using computer software or hardware to model and analyze the behavior of digital logic circuits. Digital logic circuits are fundamental components of electronic devices and computer systems, composed of logic gates (such as AND, OR, NOT, and others) that process binary signals (0s and 1s) to perform various operations and tasks.

2) Modular design is an approach to designing systems, products, or projects by breaking them down into smaller, self-contained, and interrelated modules or components. Each module serves a specific function or task and can be developed, tested, and maintained independently of the others. These modules are designed to interact with each other through well-defined interfaces, making it easier to build, modify, and maintain complex systems. Here are the key aspects of modular design.

3) To include a lower-level VHDL file in the top-level entity, you can use a "component statement." This statement allows you to define and use the lower-level module within the top-level design. You can also specify new input and output connections to replace those of the lower-level module as needed.

Lab 5: Introduction to Logic Simulation and Modular Design

Date performed: 9/19/2023

Introduction:

Designing circuits in Quartus Prime is a productive approach. However, as circuits grow in complexity, the manual placement of each logic gate can become a laborious and challenging task. To streamline this process, this lab introduces a method of coding logic diagrams and circuits, making their creation and debugging more straightforward. VHDL is the tool of choice for this purpose. Quartus Prime efficiently processes input waveforms and VHDL code, delivering superior results compared to physical FPGA board experimentation.

Part I

Experiment of Part I

This section of the lab familiarizes us with waveform creation. Waveforms illustrate the corresponding output based on inputs. In the case of an XNOR gate, the output registers as a high wave (LOGIC 1) when both inputs yield low readings or when both inputs yield high readings.

Inputs		Output
x1	x2	f
0	0	1
0	1	0
1	0	0
1	1	1

Figure 1.A: Truth Table for Part I

This behavior of the truth table mimicked that of the XNOR gate.

Results of Part I

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity part1 is
5
6     port(
7           x1: in std_logic;
8           x2: in std_logic;
9           f: out std_logic
10          );
11
12
13 end part1;
14
15 architecture behavior of part1 is
16
17 begin
18
19 f <= (x2 and x1) or (not x1 and not x2);
20
21 end behavior;
```

Figure 1.B: VHDL Code for Part I(XNOR)

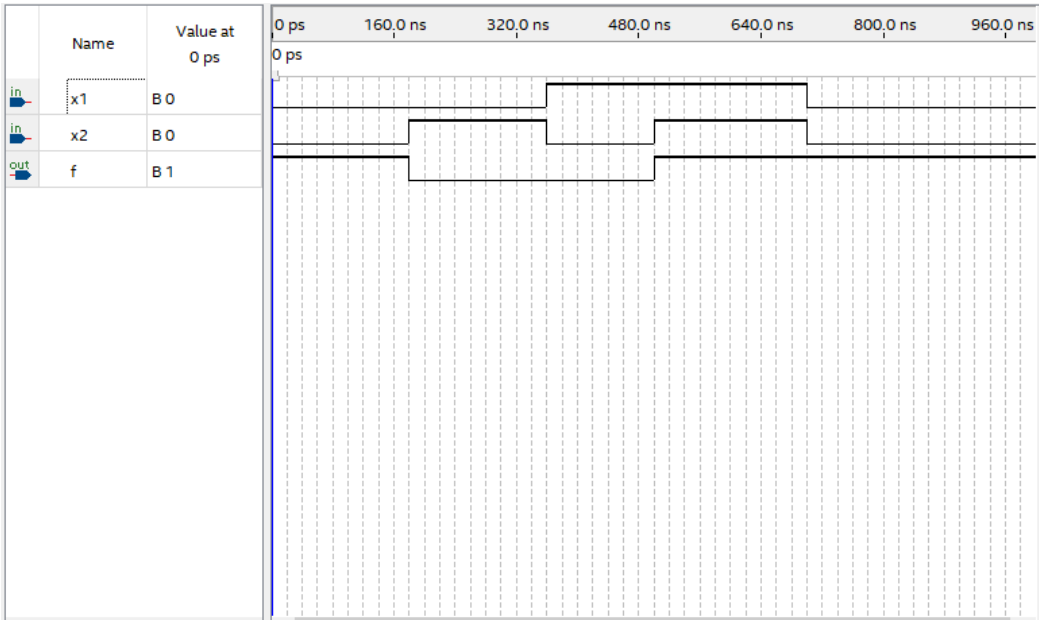


Figure 1.C: Waveform for Part I

We observe that the truth table's accurate behavior is replicated. The waveform output (F) exhibits a high state only when both X1 and X2 are either high or low.

Part II

Experiment of Part II

In this section, we delve into vector notation, which enhances the efficiency of adders. Rather than using a separate variable for each value in a row, we can create a single vector that holds multiple distinct values. For instance, we can have one vector containing all the digits of the first term and another vector containing all the digits of the second term.

```
library ieee;
use ieee.std_logic_1164.all;

entity part2 is

   port(
         x1: in std_logic_vector(2 downto 0);
         x2: in std_logic_vector(2 downto 0);
         f1: out std_logic_vector(3 downto 0)

         );


end part2;

architecture behavior of part2 is
signal c1: std_logic_vector(1 downto 0);

begin

f1(0) <= (x1(0) XOR x2(0));
f1(1) <= (x1(1) AND x2(1) AND c1(0)) OR (NOT x1(1) AND NOT x2(1) AND c1(0)) OR (NOT x1(1) AND x2(1) AND NOT c1(0)) OR (x1(1) AND NOT x2(1) AND NOT c1(0));
f1(2) <= (x1(2) AND x2(2) AND c1(1)) OR (NOT x1(2) AND NOT x2(2) AND c1(1)) OR (NOT x1(2) AND x2(2) AND NOT c1(1)) OR (x1(2) AND NOT x2(2) AND NOT c1(1));
f1(3) <= (NOT x1(2) AND x2(2) AND c1(1)) OR (x1(2) AND NOT x2(2) AND c1(1)) OR (x1(2) AND x2(2) AND NOT c1(1)) OR (x1(2) AND x2(2) AND c1(1));

c1(0) <= (x1(0) AND x2(0));
c1(1) <= (NOT x1(1) AND x2(1) AND c1(0)) OR (x1(1) AND NOT x2(1) AND c1(0)) OR (x1(1) AND x2(1) AND NOT c1(0)) OR (x1(1) AND x2(1) AND c1(0));
end behavior;
```

Figure 2.A: Waveform for Part I

Results of Part II

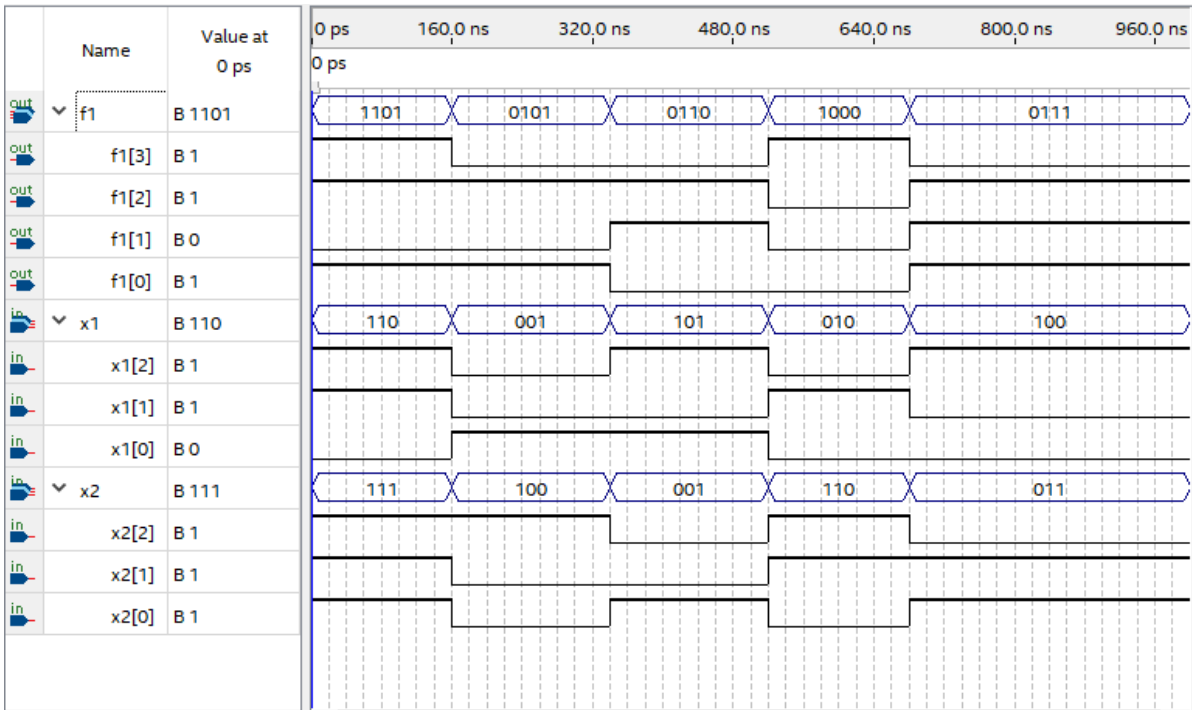


Figure 2.B: Waveform for Part I

The results above show the sums of the terms x1 and x2 with different values and results.

Part III

Experiment of Part III

This section focuses on constructing a one-bit adder, which can be likened to a single column as opposed to the multiple columns in a higher-bit adder. We are limiting ourselves to a single input bit for this purpose. The operation of this adder involves adding A to B and then adding the result to C.

Inputs			Outputs	
A	**B**	**C**	**Sum**	**Carry**

0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Sum = (A XOR B) XOR C) OR ABC

Carry = C AND (A XOR B) or (A and B)

Results and code creation were not necessary for this part, as the findings from part 3 will be used in part 4.

Part IV

Experiment of Part IV

In the concluding section, our task was to develop a two-input binary adder using component statements, building upon the code we created in part 3.

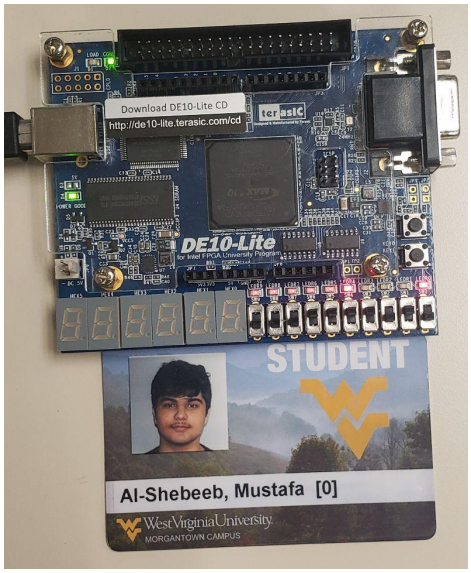
```vhdl
library ieee;
use ieee.std_logic_1164.all;

   entity part4 is
   port(
         a: in std_logic_vector(3 downto 0);
         b: in std_logic_vector(3 downto 0);
         y: out std_logic_vector(4 downto 0)


   );
   end part4;
architecture behavior of part4 is

signal c: std_logic_vector(2 downto 0);

component part3

port(
         x1: in std_logic;
         x2: in std_logic;
         x3: in std_logic;
         f1: out std_logic;
         c1: out std_logic
         );
         end component;

begin

instance_pm1: part3 port map

   (x1 => a(0), x2 => b(0), x3 => '0', f1 => y(0), c1 => c(0));

instance_pm2: part3 port map

   (x1 => a(1), x2 => b(1), x3 => c(0), f1 => y(1), c1 => c(1));

instance_pm3: part3 port map

 (x1 => a(2), x2 => b(2), x3 => c(1), f1 => y(2), c1 => c(2));

instance_pm4: part3 port map

 (x1 => a(3), x2 => b(3), x3 => c(2), f1 => y(3), c1 => y(4));


end behavior;
```

Figure 4.A: VHDL Code for Part IV

We employed the component statement to facilitate the integration of our part 3 VHDL file into part 4. In part 4, which serves as our top-level entity file, part 3 functions as a lower hierarchy file, enabling us to reuse its inputs and outputs. The component statement was instrumental in incorporating the three inputs and two outputs from the part 3 file.

Results of Part IV

Binary Addition Inputs		Output
Inputs (X1)	**Inputs (X2)**	**Output (F)**
1010	0111	10001
0011	1001	01100



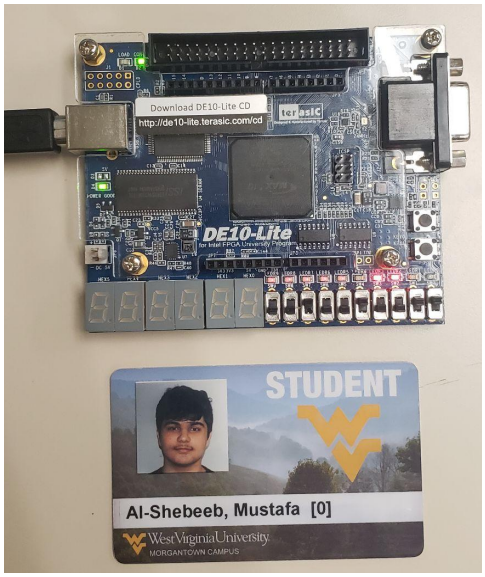


Figure 4.B: x1=1010, x2=0111, f=10001 Figure 4.C: x1=0011, x2=1001, f=01100

We decided to use Lite Board images instead of a waveform simulation due to time constraints.

Conclusion

This lab provided valuable insights into the convenience of using waveforms for logic simulation. It emphasized the efficiency of simulating truth table behaviors through waveforms, as opposed to relying on FPGA configurations, which involve time-consuming tasks like pin assignments. This approach proves to be an effective way to simulate logic, particularly when coded using VHDL.

Post-Lab Questions

1) STD_LOGIC_VECTOR(7 DOWNTO 0);

2) -1st Code: Component Half Adder port must be placed between ArchitEcture and begin statement
   -2nd Code: Missing semicolon at last line of output in first port statement
   -1st Code: End statement should be placed after entity postlab = end postlab;
   -1st Code: Port map statement should be placed in the half adder statement

Lab 6: Behavior Modeling of Combination Logic Circuits using VHDL

Date performed: 9/26/2023

Introduction:

In this lab, we delve into the world of VHDL and explore the application of sequential statements. These statements allow us to execute our code step by step, in a top-to-bottom fashion, as opposed to the concurrent approach we've seen before.

This lab serves as a crucial learning experience as it incorporates various fundamental aspects of typical circuits. We tackle the priority encoder, which generates an output corresponding to the highest-order input. Additionally, we delve into the decoder, where the output reflects the decoded value of the input combination observed on a signal. Lastly, we explore the multiplexer, a device that routes a select input to the output based on selector variables.

Part I

Experiment of Part I

A priority encoder plays a vital role in guiding a computer on which command to prioritize. It accomplishes this by determining and outputting the index of the highest active HIGH input signal.

In this experiment, our task is to design a priority encoder with three inputs and two outputs. The formula to determine the number of inputs for a priority encoder is given by 2^(M) - 1, where M represents the number of outputs.

Inputs			Outputs	
X2	**X1**	**X0**	**F1**	**F0**
0	0	0	0	0

0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Table 1.A: 3 input, 2 output Priority Encoder

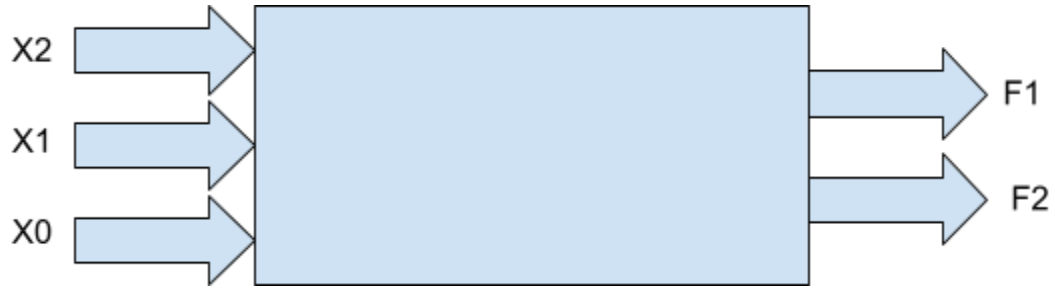


Table 1.B: Block Diagram for 3 input, 2 output Priority Encoder

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity part1 is

    port(
          x0: in std_logic;
          x1: in std_logic;
          x2: in std_logic;
          f0: out std_logic;
          f1: out std_logic
          );


end part1;

architecture behavior of part1 is


begin

process(x0, x1, x2)

 begin

if x2='1' then
    f1<='1';
    f0<='1';
elsif x1='1' then
    f1<='1';
    f0<='0';
elsif x0='1' then
    f1<='0';
    f0<='1';
else
    f1<='0';
    f0<='0';

end if;
 end process;


 end behavior;
```

Table 1.C: VHDL Code for Part I

Results of Part I

Pin assignments were reversed meaning that x0 was sw2

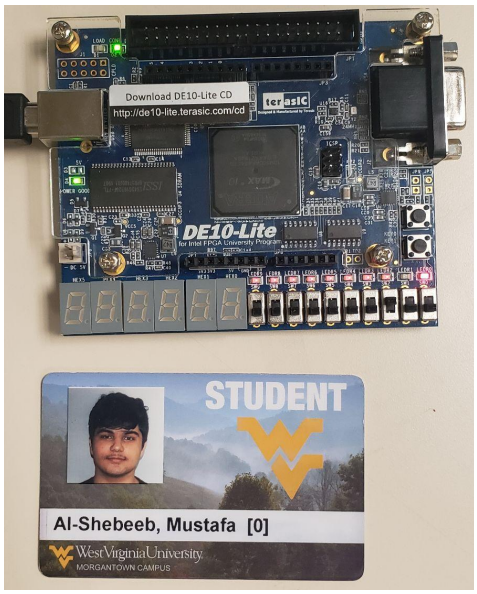


Figure 1.D: 001 = 01

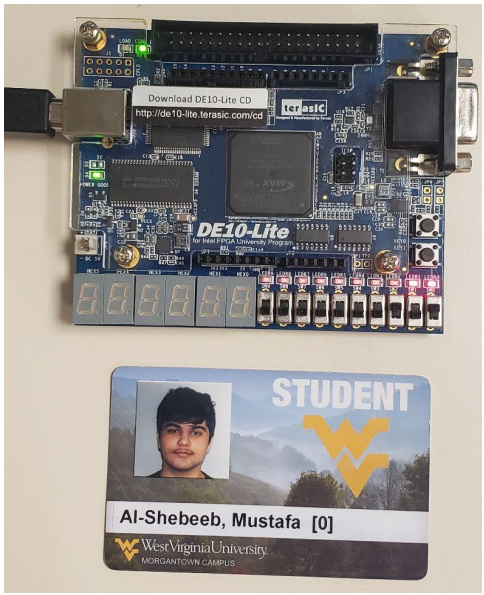


Figure 1.E 101 = 11

Part II

Experiment of Part II

In this section, our objective is to construct a decoder, specifically a 3-to-8 decoder. This decoder takes three inputs, interprets them as binary values, and then activates the corresponding output light. For instance, if the input is '0 1 0', which translates to 2 in decimal, LED 2 will illuminate. It's important to note that an input of '0 0 0' will result in LED 0 lighting up, following a zero-based index.

Inputs			Outputs							
X2	**X1**	**X0**	**Y7**	**Y6**	**Y5**	**Y4**	**Y3**	**Y2**	**Y1**	**Y0**
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Table 2.A: Table 3: 3 to 8 Decoder

Table 2.B: Block Diagram for 3 to 8 Decoder

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity part2 is

   port(
         x: in std_logic_vector(2 downto 0);
         f: out std_logic_vector(7 downto 0)
         );


end part2;

architecture behavior of part2 is


begin

process(x)

 begin

if x="111" then
 f<= "10000000";
elsif x="110" then
 f<= "01000000";
elsif x="101" then
 f<= "00100000";
elsif x="100" then
 f<= "00010000";
elsif x="011" then
 f<= "00001000";
elsif x="010" then
 f<= "00000100";
elsif x="001" then
 f<= "00000010";
else
 f<= "00000001";

end if;
 end process;


 end behavior;
```

Table 2.C: VHDL Code for Part II

Results of Part II

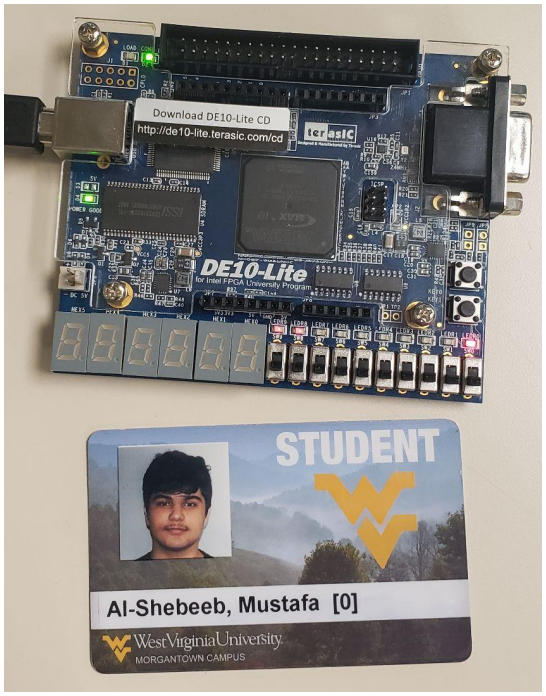


Figure 2.D: Inputs: 000, Outputs: 00000001

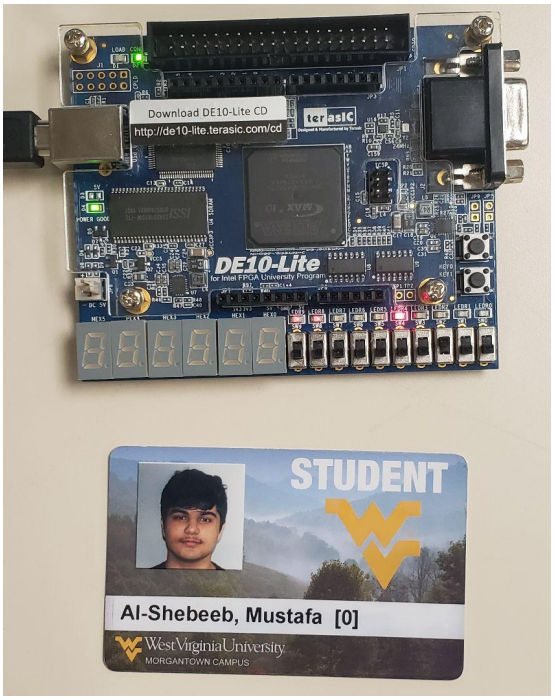


Figure 2.E: Inputs: 100, Outputs: 00010000

Part III

Experiment of Part III

A multiplexer is a versatile device that allows you to choose one specific output from among several input options. Think of it as a selector that picks one input to serve as the output.

In this section, our task is to design a multiplexer equipped with 2 select bits and 4 input bits.

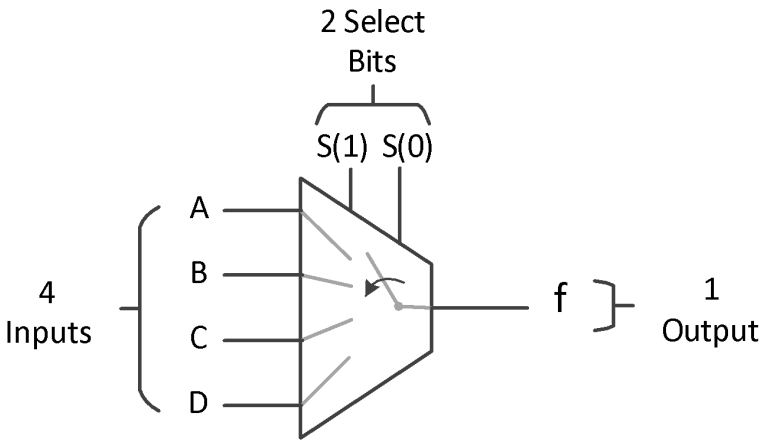


Figure 3.A: Block Diagram for 2 select 4 input mux

Select Bits		Output
S1	**S0**	**F**
0	0	A
0	1	B
1	0	C
1	1	D

Table 3.B: Truth Table for 2 select 4 input mux

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity part3 is

   port(
         S: in std_logic_vector(1 downto 0);
         A: in std_logic;
         B: in std_logic;
         C: in std_logic;
         D: in std_logic;
         F: out std_logic
         );


end part3;

architecture behavior of part3 is


begin

process(S)

begin

if S="11" then
F<= D;
elsif S="10" then
F<= C;
elsif S="01" then
F<= B;
else
F<= A;

end if;
end process;


end behavior;
```

Figure 3.C: VHDL Code for 2 select 4 input mux

Boolean Equation for Mux: S1'S0'A+S1'S0B+S1S0'C+S1S0D

Results of Part III

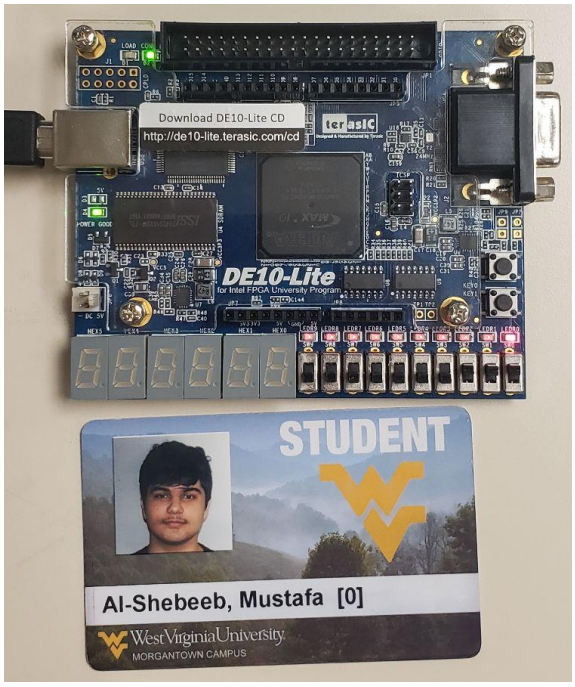


Figure 3.D: D S(0) = 1, S(1) = 1

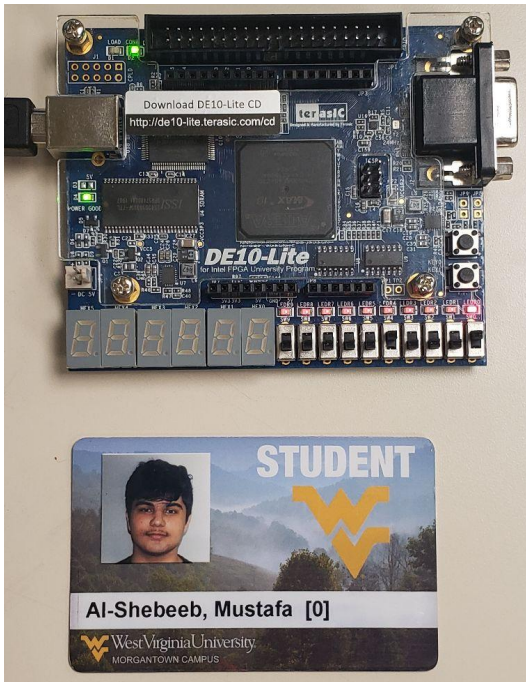


Figure 3.E: B S(0) = 1, S(1) = 0

Part IV

Experiment of Part IV

In Part 4, our objective was to analyze a circuit that featured a multiplexer, which we previously created as a component in Part 3. Notably, two of the inputs connected to the multiplexer were derived from combinational logic.

s(1)	s(0)	x(1)	x(0)	f1 (expression)	f1 (val)
0	0	0	0	x(0) and x(1)	0
0	0	0	1	x(0) and x(1)	0
0	0	1	0	x(0) and x(1)	0
0	0	1	1	x(0) and x(1)	1
0	1	0	0	x(0) or x(1)	0
0	1	0	1	x(0) or x(1)	1
0	1	1	0	x(0) or x(1)	1
0	1	1	1	x(0) or x(1)	1
1	0	0	0	x(0)	0
1	0	0	1	x(0)	1
1	0	1	0	x(0)	0
1	0	1	1	x(0)	1
1	1	0	0	x(1)	0
1	1	0	1	x(1)	0
1	1	1	0	x(1)	1
1	1	1	1	x(1)	1

Table 4.A: Truth Table for Part IV

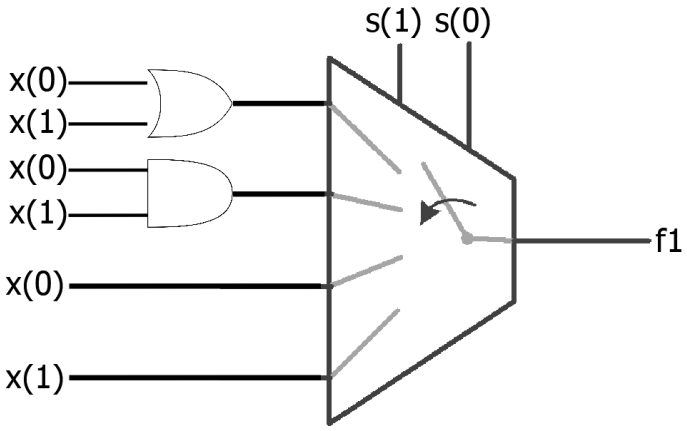


Figure 4.B: Block Diagram for Part IV

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity part4 is
    port(
            s: in std_logic_vector(1 downto 0);
            x: in std_logic_vector(1 downto 0);
            f: out std_logic
            );

end part4;

architecture behavior of part4 is

component part3
port(
            S: in std_logic_vector(1 downto 0);
            A: in std_logic;
            B: in std_logic;
            C: in std_logic;
            D: in std_logic;
            F: out std_logic
            );
            end component;

begin

instance_pm1: part3 port map

   (S(0) => s(0), S(1) => s(1),A => x(0) and x(1), B => x(0) or x(1), C => x(0), D => x(1),  F => f);


end behavior;
```

Figure 4.C: VHDL Code for Part IV

Results of Part IV

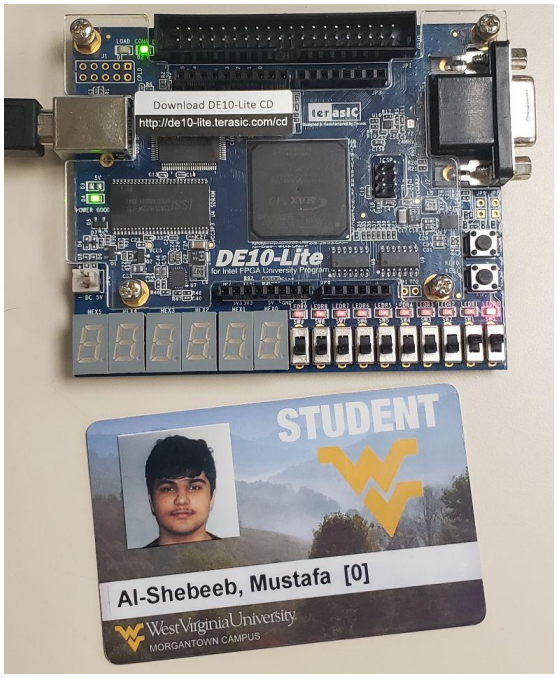


Figure 4.D: inputs: 0011, output: 1

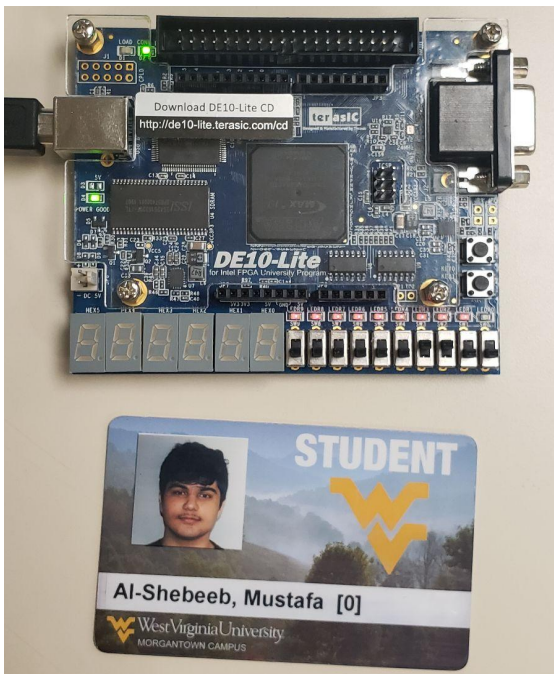


Figure 4.E: inputs: 1100, output: 0

Conclusion

This lab proved to be somewhat challenging and time-intensive. We encountered persistent errors in Part 1 while working with Quartus Prime, which consumed a significant portion of our lab time.

Nevertheless, aside from the initial hurdles, this lab provided a valuable learning experience. It offered a comprehensive understanding of how decoders and multiplexers operate, contributing to a deeper knowledge of complex circuits and their functionality.

Post-Lab Questions

1) 2 ^ (4) - 1 = 15 inputs
2) 2 ^ (4) = 16 inputs
3) Y = A’B’C + A'B 1 + AB’0 + ABC

Pre-Lab Questions

1) A flip-flop is a digital circuit component used for storing binary information in electronic systems. It can exist in one of two stable states, typically representing 0 and 1. Flip-flops

play a critical role in sequential logic circuits, where data needs to be stored and processed over time. The different types are SR, JK, D, and T flipflops

2) The "D" in the D flip-flop stands for "Delay," indicating that its output represents the input value from the previous clock cycle. This output is often referred to as the "Next State."

Lab 7: Sequential Logic Design Part II

Date performed: 10/3/2023

Introduction:

This laboratory exercise focuses on the utilization of memory and the internal clock of the DE10 programming board. It aims to introduce students to the concept of circuits that rely on the DE10 Lite board's clock signal and input signals. Specifically, this lab provides hands-on experience with D-type flip-flops as a fundamental component of digital circuits.

Part I

Experiment of Part I

In this section, we construct a basic D-Flip Flop, sometimes referred to as the Data or Delay Flip Flop. Its operation is straightforward: when the clock signal is in a high state, the input D is duplicated onto the output Q. Consequently, the present state input becomes the output Q during the subsequent clock cycle. This elementary D flip-flop serves as a foundational building block for implementing memory functionalities within digital circuits.

Input	Output
D_n	$Q_{(n+1)}$
0	0
1	1

Table 1.A: D Flip-flop State Table

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity part1 is

   port(
         clock_in: in std_logic;
         d: in std_logic;
         q: out std_logic;
         qnot: out std_logic

      );

end part1;

ArchitEcture behavior of part1 is

begin

process(clock_in)
begin

If (clock_in'event and clock_in ='1') then
   q <= d;
   qnot <= NOT d;


end if;
end process;

end behavior;
```

Figure 1.B: VHDL Code for Part I

Results of Part I

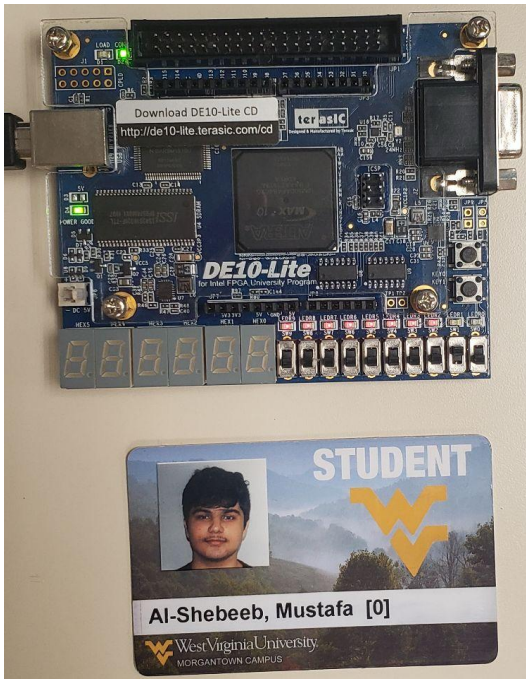


Figure 1.C: Both the clock and D are low.

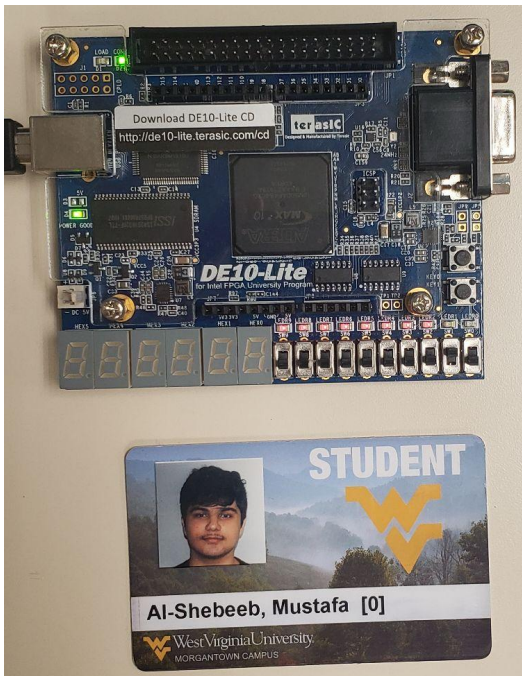


Figure 1.D: Clock-in is on and D is low.

Part II

Experiment of Part II

Given that the built-in clock of the DE10 board operates at a high default frequency of 50MHz, it may be too fast for us to observe logic changes effectively. To address this, we can achieve a slower rate of logic changes by dividing the clock cycles to occur less frequently. This adjustment enables us to clearly visualize and analyze the changes occurring within the circuit.

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3
4   entity part2 is
5
6   PORT(
7
8         clock_in : in std_logic;
9         clock_out: out std_logic
10
11  );
12  end part2;
13
14  architecture behavior of part2 is
15
16  signal clock_tmp: std_logic;
17
18  begin
19
20
21  process(clock_in)
22
23  variable x : integer := 0;
24
25  begin
26
27  if(clock_in'event and clock_in='1') then
28
29     x:=x+1;
30     if x=50000000 then
31        x:=0;
32        clock_tmp<= not clock_tmp;
33        clock_out<= clock_tmp;
34     end if;
35
36  end if;
37  end process;
38  end behavior;
39
```

Figure 2.A: VHDL Code for Part II

The objective of this section was to slow down the clock cycle refresh rate to occur every one second. As a result, the light illuminates for a duration of one second, followed by a one-second interval where it remains off. This alternating pattern continues to repeat, allowing for clear and discernible observation.

Results of Part II

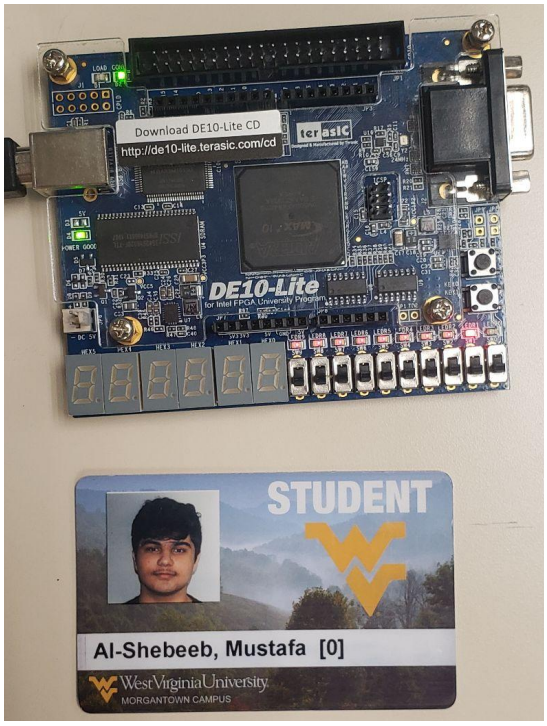


Figure 2.B: Slowed Down Clock, LED0

Part III

Experiment of Part III

In this section, we integrate the D flip-flop introduced in part 1 with the divider from part 2 to engineer a sluggish flip-flop. This particular section focuses on crafting a leisurely D flip-flop that possesses the capability to store information, thanks to a gradual clock cycle. The stored value undergoes alteration with each ascending edge of the clock cycle, contributing to the memory storage process.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity part3 is

PORT(
        clock_in2: in std_logic;
        D: in std_logic;
        Q: out std_logic;
        Qnot: out std_logic


);
end part3;

architecture behavior of part3 is

signal clock_sig: std_logic;


component part2 is

port(
      clock_in: in std_logic;
      clock_out: out std_logic

        );
        end component;

component part1 is

   port(
        clock_in: in std_logic;
        d: in std_logic;
        q: out std_logic;
        qnot: out std_logic

      );

end component;

begin

instance_pm1: part2 port map

   (clock_in => clock_in2, clock_out => clock_sig);


   instance_pm2: part1 port map

   (D => d, Q => q, Qnot => qnot, clock_in => clock_sig);

end behavior;

```

Figure 3.A: VHDL Code for Part III

Results of Part III

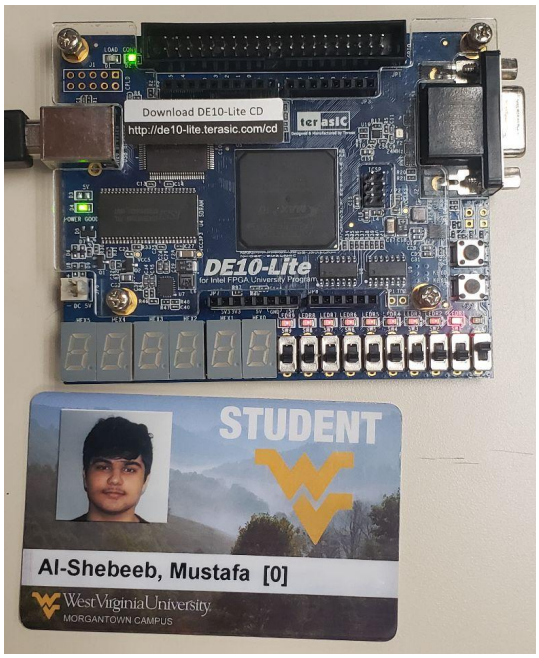


Figure 3.B: Input D is high but Q is low

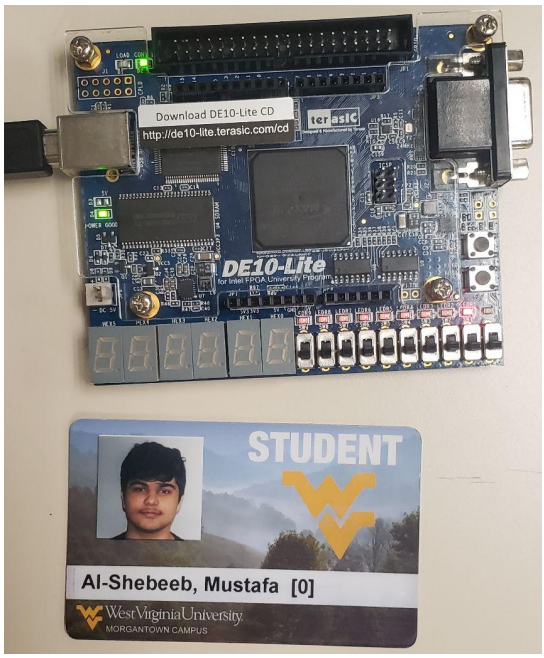


Figure 3.C: Input D is low and Q is low

Part IV

Experiment of Part IV

In this section, we will design a modulo 8 up-down counter. This counter operates based on the input signal: when the input is 0, it counts up, and when the input is 1, it counts down. The count is displayed in binary format. For example, if we're at 101 and the input is 0, the counter will increment to 110 and then 111 before wrapping back to 000. Conversely, if we're at 010 and the input is 1, the counter will decrease to 001, then 000, and finally wrap around to 111, and so forth.

Present State (PS)	**Next State**	
	X = 0 (Count Up)	**X = 1 (Count Down)**
$Q_2\ Q_1\ Q_0$	**$D_2\ D_1\ D_0$**	**$D_2\ D_1\ D_0$**
000	001	111
001	010	000
010	011	001
011	100	010
100	101	011
101	110	100
110	111	101
111	000	110

Figure 4.A: VHDL Code for Part III

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity part4 is

PORT(
    clock_in3: in std_logic;
    x: in std_logic;
    q: inout std_logic_vector(2 downto 0)


);
end part4;

architecture behavior of part4 is

signal clock_sig: std_logic;
signal d: std_logic_vector(2 downto 0);

component part3 is

    port(
          clock_in2: in std_logic;
          d: in std_logic;
          q: out std_logic


        );
end component;

component part2 is

port(
       clock_in: in std_logic;
       clock_out: out std_logic

          );
end component;

component part1 is

    port(
          clock_in: in std_logic;
          d: in std_logic;
          q: out std_logic


        );
end component;
```

```vhdl
begin

    instance_pm1: part3 port map

    (clock_in2 => clock_in3, d => d(0), q => q(0));

    instance_pm2: part3 port map

    (clock_in2 => clock_in3, d => d(1), q => q(1));

    instance_pm3: part3 port map

    (clock_in2 => clock_in3, d => d(2), q => q(2));


    d(0) <= (NOT q(0));

    d(1) <= ((NOT X AND (q(1) XOR q(0))) OR (X AND ((q(1) AND q(0)) OR (NOT q(1) AND NOT q(0)))));

    d(2) <= ((NOT X AND ((NOT q(2) AND q(1) AND q(0)) OR (q(2) AND (q(1) NAND q(0))))) OR (X AND ((NOT q(2) AND NOT q(1) AND NOT q(0))  OR (q(2) AND (q(1) OR q(0))))));


end behavior;
```

Figure 4.B: VHDL Code for Part IV

Results of Part IV

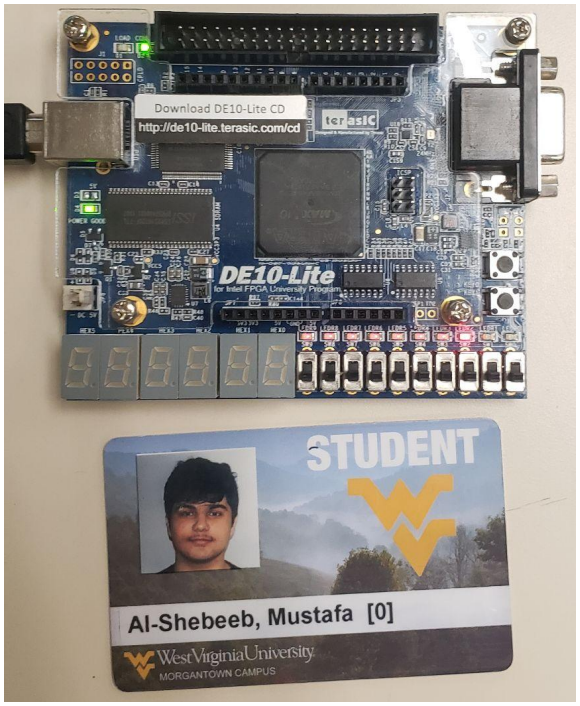


Figure 4.C: Clock is on and counting up. Lights: 4

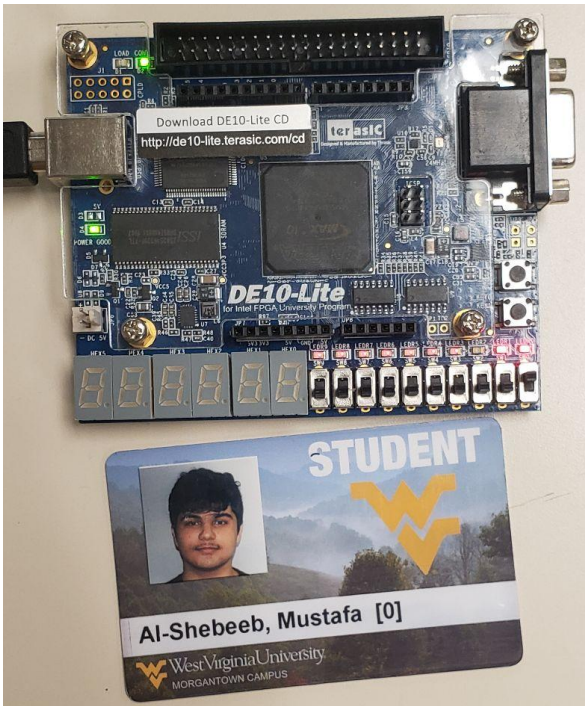


Figure 4.D: Clock is on and counting down. Lights: 3

Conclusion

This lab proved to be quite enjoyable as it allowed us to observe our outputs on the DE10 Lite board. The sequential illumination of lights on the programming board showcased a practical application of memory and clock utilization.

Post-Lab Questions

1) The equation for Kmap in SOP canonical:
   Y = D' + A'B'

Pre-Lab Questions

1)

   In the Mealy model, the output is a function of both the current state and the external input. This model is generally more responsive and can produce outputs that change more frequently.
   In the Moore model, the output is solely a function of the current state. Moore machines produce outputs that are typically more synchronized and predictable compared to Mealy machines.

Lab 8: Sequential Logic Design Part II

Date performed: 10/17/2023

Introduction:

This laboratory exercise provides valuable insights into two distinct categories of state machines. A state machine is essentially a mechanism whose functioning is partially or entirely influenced by the current state of its outputs. In this lab, we explored two primary types of state machines: the Mealy state machine and the Moore state machine.

The Mealy state machine is characterized by its output being contingent upon both the input and the current state. Conversely, the Moore state machine's outputs are solely determined by the present state, with no direct reliance on the input. This dichotomy in output behavior represents a fundamental difference between these two types of state machines.

Part I

Task 1:

Answer the following questions using Figure 6.

1) Does the state diagram in Figure 6 follow the Mealy or Moore model?

   Mealy Model

2) If the circuit is originally at state “11” and the input sequence is “010010”.

   What will the output sequence be(“xxxxxx”)? ____000000_____

   What is the final state that the system lands on after the full input sequence is over? _11_____

3) Finish the following timing diagram by filling out the correct **States** and **Outputs** sections in the empty boxes. **Assume a negative edge triggered system this time**.

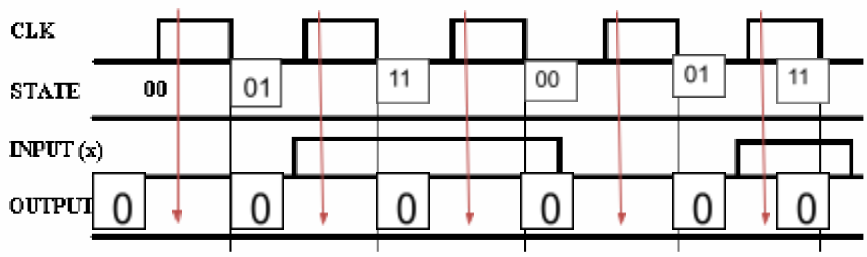


Figure 1.A: Sample Timing Diagram

Task 2:

Consider the sequential circuit shown in Figure 8.

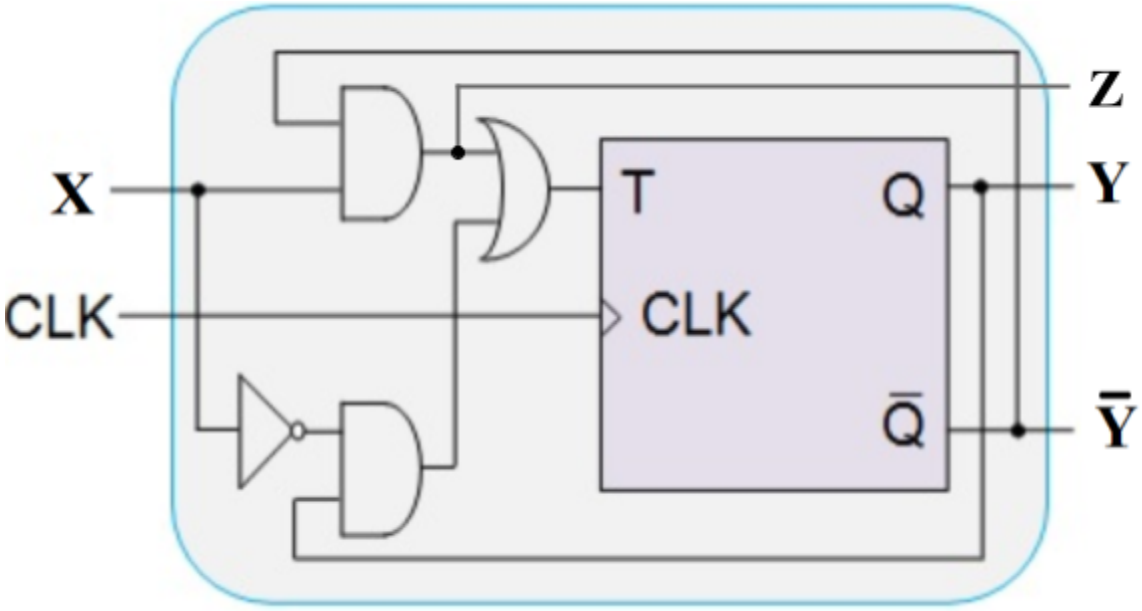


Figure 1.B: Sample Sequential Circuit Implementation

Answer the following questions using Figure 8.

1) What type of Flip-flop is used in Figure 8? ____T-Flip Flop_____

2) How many states does this circuit have? _____2 states___

3) Write the logic expression for T (in terms of x & y):

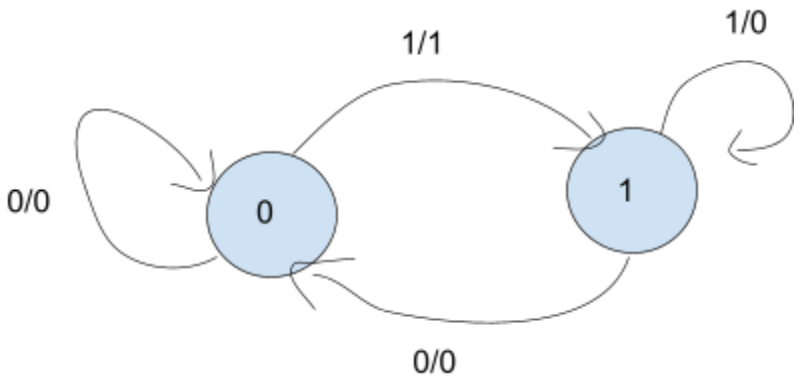
   T = X AND NOT Y OR Y AND NOT X AND Y

4) Write the logic expression for the output z: Z = X AND NOT Y

PS	Present FF Inputs	NS	Outputs

$Q_n = Y_n$	X	T =	$Q_{n+1} = Y_{n+1}$	Z_n =
0	0	**0**	**0**	**0**
0	1	**1**	**1**	**1**
1	0	**1**	**0**	**0**
1	1	**0**	**1**	**0**

Table 1.C: State Transition Table



New improved FSM: Not Hand Drawn

Now determine the output sequence, **Z**, for an **input sequence x=01101000** and an **initial memory state y=0** using the created Mealy Model above.

Z = _____01001000_____

Y = _____01101000_____

Afterwards, analyze both the transition diagram you just drew as well as by referring to the output sequence you got, identify what sort of flip-flop is the sequential circuit in Figure 8 behaving like: _____D Flip-Flop_____

Part II

Experiment 1

Current		**Next State**		**D Flip Flop**	
y_2	y_1	**X=0**	**X=1**	**X=0**	**X=1**
0	**0**	0 d	0 0	0 d	0 0
0	**1**	1 0	0 0	1 0	0 0

1	**0**	d d	0 1	d d	0 1
1	**1**	d d	1 0	d d	1 0
		Y_2,Y_1	**Y_2,Y_1**	**D_2,D_1**	**D_2,D_1**

Table 2.A: D Flip-flop

Equation for D_2:

y_2 y_1 \ x	0	1
00	0	0
01	1	0
11	d	1
10	d	0

Equation for D_1:

y_2 y_1 \ x	0	1
00	d	0
01	0	0
11	d	0
10	d	1

d = 1

Output Equation **D_2** = _____x'y1+y2y1_____

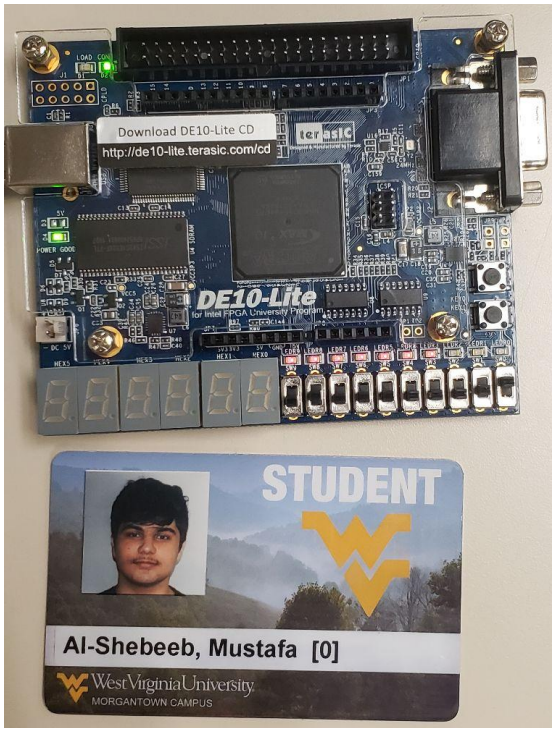
Output Equation **D_1** = _____y2y1'_____

Experiment 2

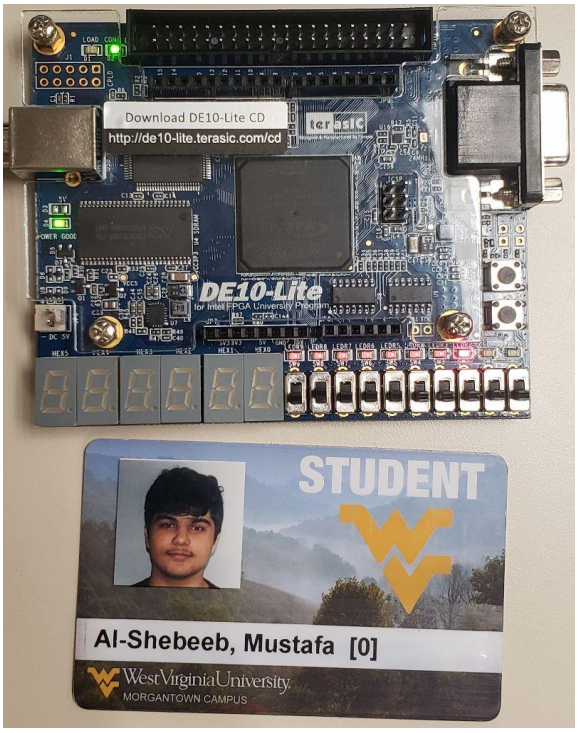
```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity exp2 is
PORT(
    x: in std_logic;
    Clk: in std_logic;
    y2: inout std_logic;
    y1: inout std_logic;
    Z: out std_logic

);
end exp2;
architecture behavior of exp2 is
signal d1: std_logic;
signal d2: std_logic;
component part3 is
PORT(
          clock_in2: in std_logic;
          D: in std_logic;
          Q: out std_logic;
          Qnot: out std_logic


);
end component;



component part2 is
port(
       clock_in: in std_logic;
       clock_out: out std_logic

          );
          end component;
component part1 is
   port(
          clock_in: in std_logic;
          d: in std_logic;
          q: out std_logic;
          qnot: out std_logic

       );
end component;

begin
instance_pm1: part3 port map
(Clock_in2 => Clk, D => d1, Q => y1);
instance_pm2: part3 port map
```

```vhdl
begin
instance_pm1: part3 port map
(Clock_in2 => Clk, D => d1, Q => y1);
instance_pm2: part3 port map
(Clock_in2 => Clk, D => d2, Q => y2);


Z <= (NOT x AND NOT y1) OR (y2 AND NOT y1) OR (x AND y2);
d2 <= (NOT x AND y1) OR (y2 AND y1);
d1 <= (y2 AND NOT y1);


end behavior;
```

Table 2.B: VHDL Code for Experiment 2

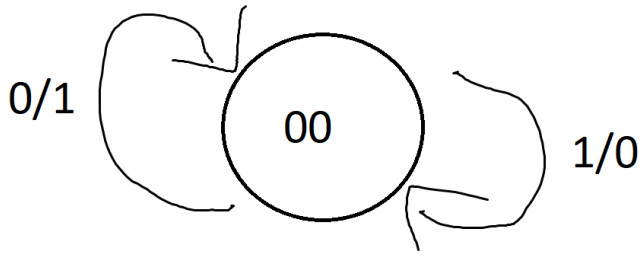


Current State: 0 input: 1 Output: 0



Current State: 0 Input: 0 Output: 0

The output behavior became locked into two specific states due to the presence of "don't care" values. These "don't care" values, lacking explicit assignments, caused the system to become stuck in this state. To rectify this issue, a change in the next state, different from the current state, can be implemented. This resolution is effectively demonstrated in the subsequent experiment. The mealy state diagram provided below illustrates the board's behavior.

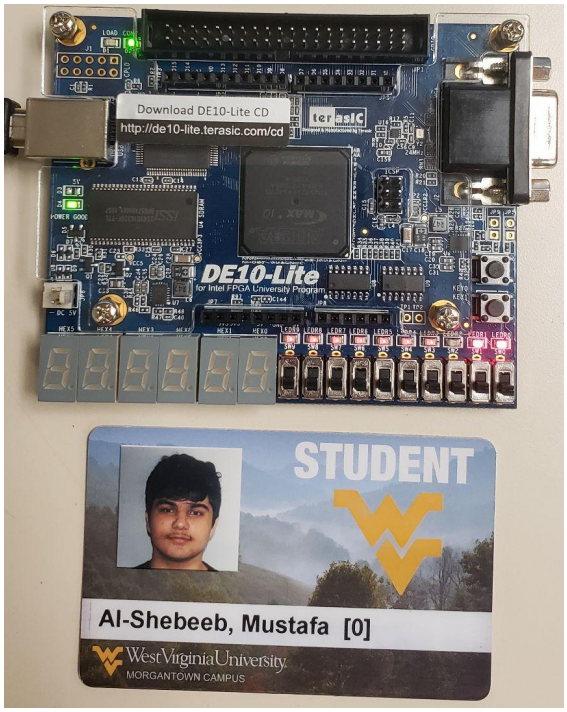


Experiment 3

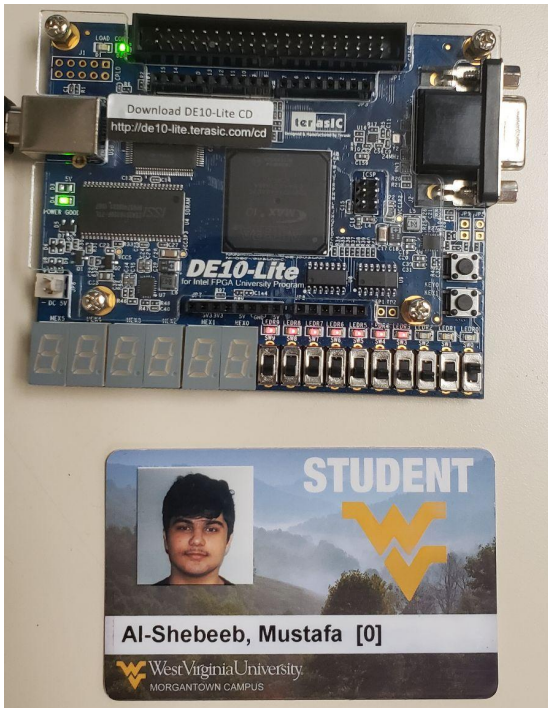
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity exp2 is
PORT(
    x: in std_logic;
    clk: in std_logic;
    y2: inout std_logic;
    y1: inout std_logic;
    Z: out std_logic

);
end exp2;
architecture behavior of exp2 is

signal d1: std_logic;
signal d2: std_logic;

component part3 is
PORT(
          clock_in2: in std_logic;
          D: in std_logic;
          Q: out std_logic;
          Qnot: out std_logic


);
end component;



component part2 is
port(
       clock_in: in std_logic;
       clock_out: out std_logic

          );
          end component;
component part1 is
   port(
          clock_in: in std_logic;
          d: in std_logic;
          q: out std_logic;
          qnot: out std_logic

       );
end component;

begin
instance_pm1: part3 port map
(clock_in2 => clk, D => d1, Q => y1);
instance_pm2: part3 port map
```

```
begin

instance_pm1: part3 port map

(clock_in2 => clk, D => d1, Q => y1);

instance_pm2: part3 port map

(clock_in2 => clk, D => d2, Q => y2);


Z <= (NOT x AND NOT y1) OR (y2 AND NOT y1) OR (x AND y2);
d2 <= (NOT x AND y1) Or (NOT x AND y2) OR (y2 AND y1);
d1 <= (NOT x AND NOT y1) Or (NOT x AND y2) OR (y2 AND NOT y1);


end behavior;
```

Table 3.B: VHDL Code for Experiment 3

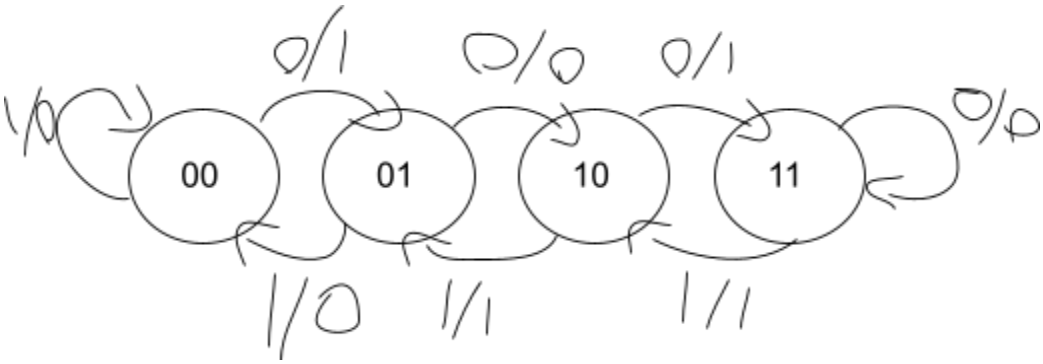


Current State: 0 input: 1 Output: 1



Current State: 0 Input: 0 Output: 0

In the preceding two output figures, we observe a clear depiction of how the current state and output evolve in response to the input. As the input signal registers as "1," the current state undergoes a transition from "11" to "10," and the corresponding output shifts from "0" to "1." This dynamic behavior of the board is effectively represented in the accompanying mealy state diagram displayed below.



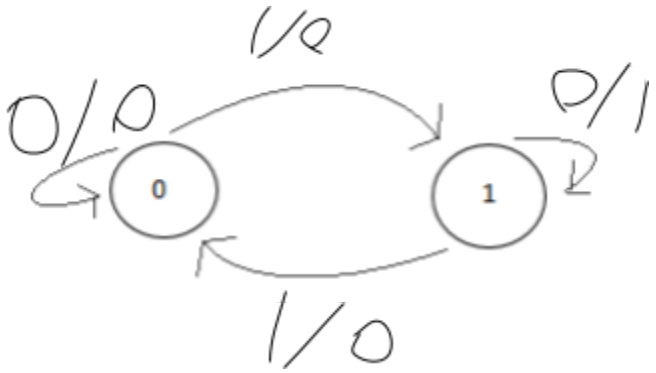
Conclusion

This lab was tough but intriguing. Learning about circuit behavior through various state machines offered a fresh perspective, quite different from what I'm used to. It's clear that I need to study different types of Finite State Machines (FSMs) to better understand these important concepts.

Post-Lab Questions

1)

PS	PS input	NS	Outputs
Y_n	T	Y_{n+1}	Z_n
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0



Pre-Lab Questions

1) Arithmetic Logic Unit - a fundamental component within a computer's central processing unit (CPU). Its primary role is to perform arithmetic and logical operations on data.

2) An enumeration type, often referred to as an enum, is a programming data type that consists of a set of named values, each of which represents a distinct constant. Enumerations are typically used to define a specific set of symbolic names or labels for related values.

3) RAM - Random Access Memory. ROM - Read-Only Memory

   ROM is non-volatile memory, meaning the data stored in ROM is not erased when the computer is powered off. Typically slower to access compared to RAM because it is used for long-term storage of software or firmware.

   RAM is volatile memory, which means it stores data temporarily and loses its content when the computer is turned off. Temporarily stores data that the CPU

needs for processing. It is used for running applications, storing data that is being actively worked on, and for short-term data storage.

Lab 9: Sequential Logic Design Using Behavioral Modeling & Memories and Arithmetic

Date performed: 10/30/2023

Introduction:

In previous labs, we have explored finite state machines, focusing on two specific types: the Mealy state machine and the Moore state machine. This lab extends our understanding by delving into enumerated types and case statements in order to effectively implement a circuit in VHDL. In this lab, we will specifically examine output sequences, an Arithmetic Logic Unit (ALU), and two memory types: Read-Only Memory (ROM) and Random Access Memory (RAM).

Part I

Experiment of Part I

Input Sequence	Final State	Output
1101	1	1
101101	1	1
0101	1	0

Table 1.A: Mealy Sequences Observed

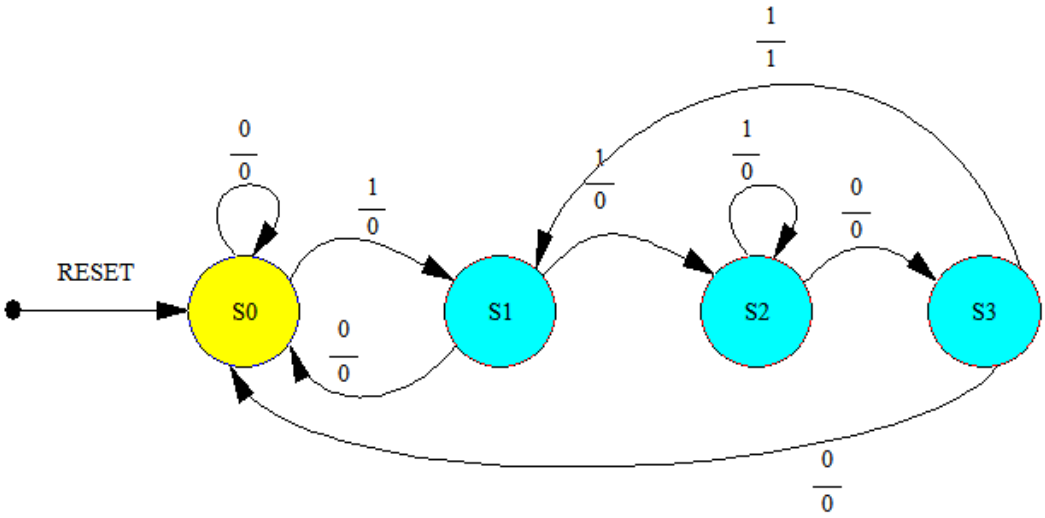


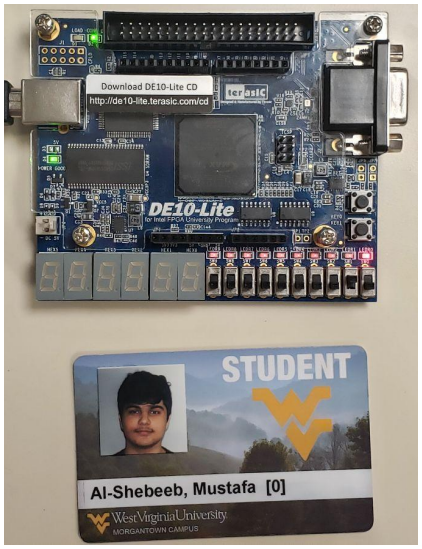
Figure 1.B: Sample Mealy Machine

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3
4
5   entity part1 is
6
7   PORT(
8
9   clk: in std_logic;
10  reset: in std_logic;
11  x: in std_logic;
12  y: out std_logic
13
14  );
15
16  end part1;
17
18
19  architecture behavior of part1 is
20
21  type state_type is (state0, state1, state2, state3);
22
23  signal state: state_type;
24
25  begin
26  process(clk, reset)
27
28  begin
29
30  if reset = '1' then
31  state <= state0;
32  y <= '0';
33
34  elsif clk'event and clk = '1' then
35
36  case state is
37
38
39  when state0 =>
40  if x = '1' then
41  state <= state1;
42  y <= '0';
43
44  elsif x = '0' then
45  state <= state0;
46  y <= '0';
47
48  end if;
49
50  when state1 =>
51  if x = '1' then
52  state <= state2;
53  y <= '0';
54
55  elsif x = '0' then
56  state <= state0;
57  y <= '0';
58
59  end if;
60
61  when state2 =>
62  if x = '0' then
63  state <= state3;
64  y <= '0';
65
66  elsif x = '1' then
```

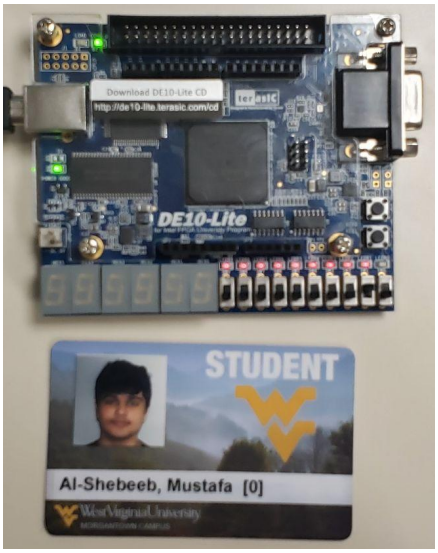
```
66  elsif x = '1' then
67  state <= state2;
68  y <= '0';
69
70  end if;
71
72  when state3 =>
73  if x = '1' then
74  state <= state1;
75  y <= '1';
76
77
78  elsif x = '0' then
79  state <= state0;
80  y <= '0';
81
82  end if;
83
84
85
86  end case;
87  end if;
88  end process;
89
90
91
92
93  end behavior;
```

Figure 1.C: VHDL Code for part I

Results of Part I



Final state: 1 Output: 1



Final State: 1 Output: 0

This represents a Mealy state machine, where the arrows indicate the input and output transitions. We move through states based on the inputs while generating outputs. During calibration on our DE10 board, we utilized a push button as the clock source. Additionally, we employed two switches, with one serving as the input control and the other for resetting the machine. With careful input provided via the board, at the end of the testing sequence, the LED output illuminated, corresponding to the input sequence '1101.'

Part II

Experiment of Part II

In this section, we developed an Arithmetic Logic Unit capable of performing operations on two operands, each consisting of 4-bit inputs. The assigned functions were as follows: "00" for addition of the operands, "01" for subtraction, and "10" for generating the output as the NAND of the two operands.

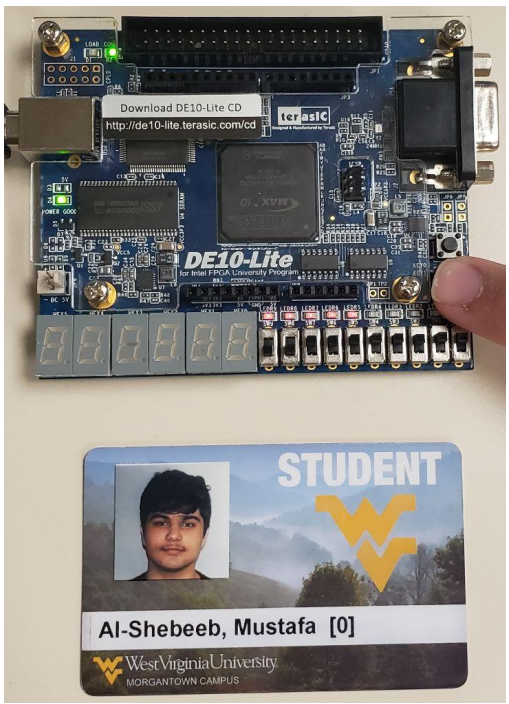
ALUOp Function	Operand A	Operand B	Output
A + B	1111	1111	11110
A - B	1111	1111	00000
A NAND B	1111	1111	10000

Table 2.A: ALU Output

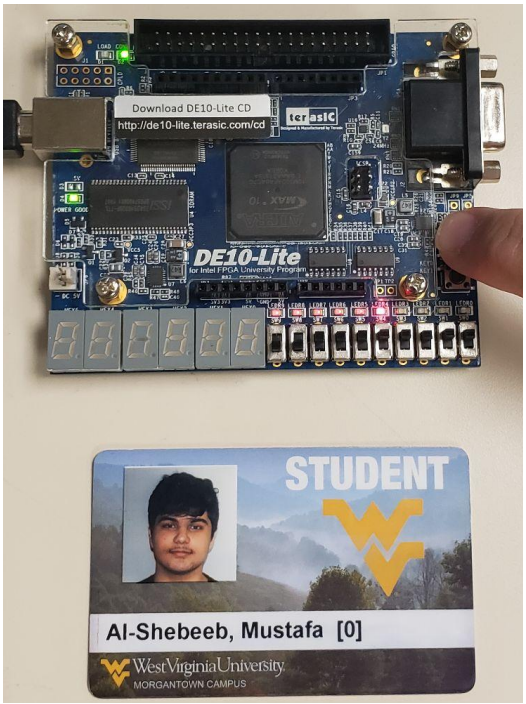
```vhdl
LIBRARY ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
USE ieee.std_logic_1164.all;

entity part2 is

PORT(
a: in std_logic_vector(3 downto 0);
b: in std_logic_vector(3 downto 0);
x: in std_logic_vector(1 downto 0);
y: out std_logic_vector(4 downto 0)

);
end part2;

architecture behavior of part2 is


begin

process(x, a, b)

begin

if x(0) = '0' then
if x(1) = '0' then
y<= ('0'&a)+('0'&b);

elsif x(1) = '1' then
y<= ('0'&a)-('0'&b);

end if;

elsif x(0) = '1' then

if x(1) = '0' then
y<= ('0'&a) NAND ('0'&b);


elsif x(1) = '1' then
y(4) <= '0';
y(3) <= '0';
y(2) <= '0';
y(1) <= '0';
y(0) <= '0';

end if;

end if;


end process;

end behavior;
```

Figure 2.B: VHDL Code for part I

Results of Part II



A - B = 00000



A NAND B = 10000

Part III

Experiment 1 of Part III

In this experiment, we generated a form of memory, namely Read-Only Memory (ROM). This type of memory assigns a unique address to each memory location, and it is structured in the form of a table, as exemplified in the laboratory demonstration. This particular phase of the experiment involves the creation of a 32-entry memory with an 8-bit width. In essence, the memory that we are constructing will be 8 bits wide and 32 locations deep.

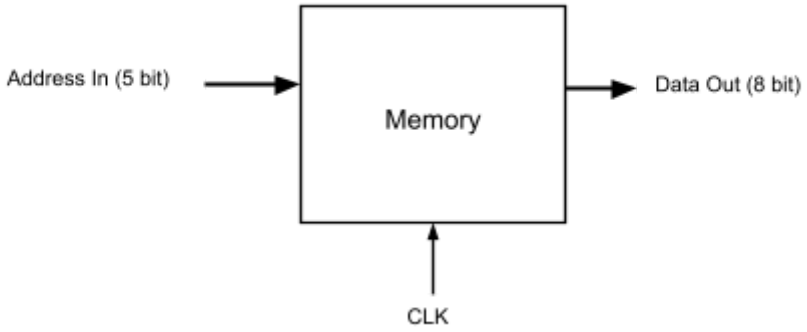
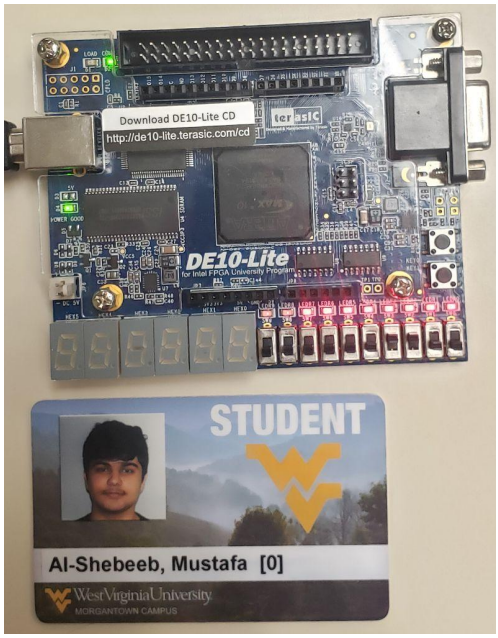


Figure 4: 8 bit * 32 Memory Block Diagram

```
LIBRARY ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
USE ieee.std_logic_1164.all;

entity exp1 is

GENERIC ( width: integer:=8; depth: integer:= 32; addr: integer:=5);
PORT
(
   clk: IN STD_LOGIC;
   read_addr: IN STD_LOGIC_VECTOR(4 downto 0);  -- Address
   data_out: OUT STD_LOGIC_VECTOR(7 downto 0) -- Data
);


end exp1;

architecture behavior of exp1 is
type ram_type is array(0 to 31) of STD_LOGIC_VECTOR(7 downto 0);
signal mem : ram_type:= ("00000111","00000110", "00000101", "00000100", "00000011", "00000010", "00000001", "00000000", others => (others=> '1')); -- Initializing Random

begin

data_out <= mem(conv_integer(read_addr));


end behavior;
```

Figure 3.A: VHDL Code for Part III Experiment 1

Results of Experiment 1



The board is reading all 8 bits of memory,
ROM is functional

Experiment 2 of Part III

In this section, we enhance the memory system that we previously created in part 3 experiment 1. We introduce the capability to access and modify memory in this phase. By examining the block diagram below, it becomes evident that we can alter the memory content at a specific memory address by providing a 4-bit address as an input and specifying the desired data to be entered (referred to as "data in"). The "WE" input indicates whether we intend to read from or edit a memory address. This process is executed successfully when the internal clock reaches its negative edge trigger, and the "WE" input is set to 1.

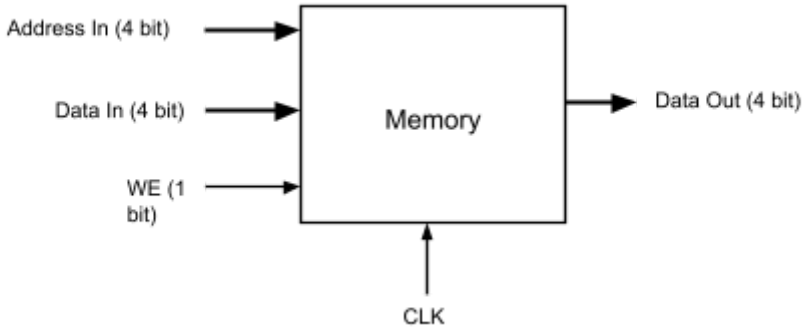
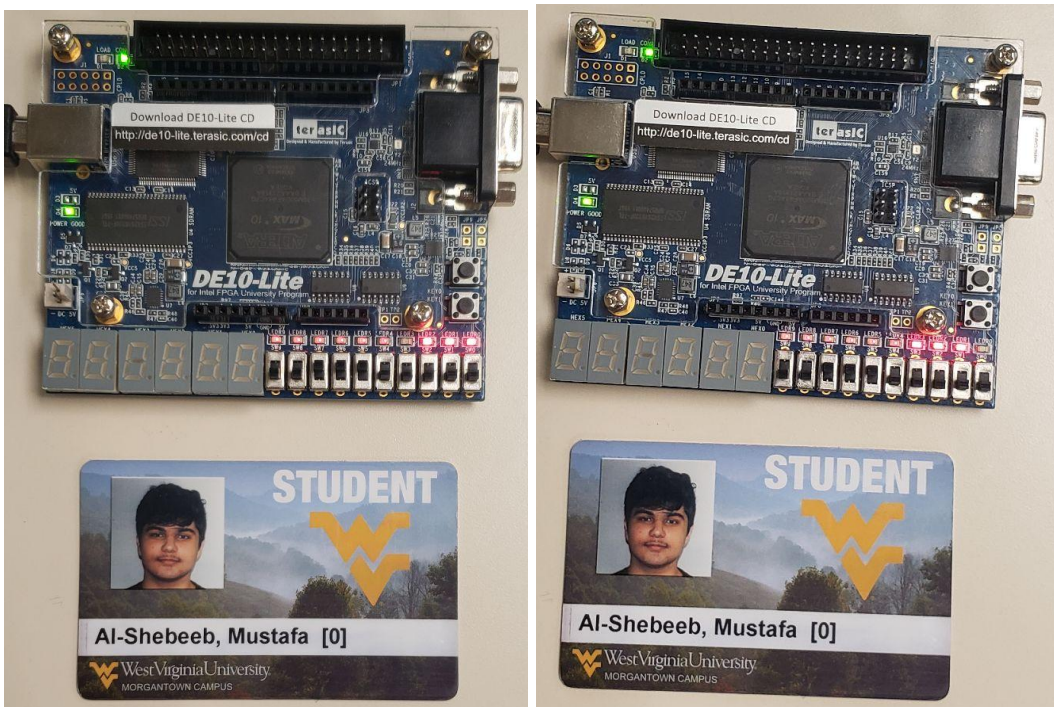


Figure 5 - RAM Block Diagram

```vhdl
LIBRARY ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
USE ieee.std_logic_1164.all;

entity exp2 is

GENERIC ( width: integer:=4; depth: integer:= 16; addr: integer:=4);
PORT
(
   clk: IN STD_LOGIC;
   read_addr: IN STD_LOGIC_VECTOR(3 downto 0);  -- Address
   data_out: OUT STD_LOGIC_VECTOR(3 downto 0); -- Data
   data_in: IN STD_LOGIC_VECTOR(3 downto 0);
   WE: in STD_LOGIC
);


end exp2;

architecture behavior of exp2 is
type ram_type is array(0 to 15) of STD_LOGIC_VECTOR(3 downto 0);
signal mem : ram_type:= ("0111","0110", "0101", "0100", "0011", "0010", "0001", "0000", others => (others=> '1')); -- Initializing Random


begin

process(clk, read_addr, data_in, WE)


begin

if (clk'event and clk = '1') then

if WE = '1' then

mem(conv_integer(read_addr)) <= data_in;


elsif WE = '0' then

data_out <= mem(conv_integer(read_addr));

end if;
end if;

end process;


end behavior;
```

Figure 3.B: VHDL Code for Part III Experiment 1

Results of Experiment 2



The pictures above use a 4-bit memory address input to write a 4-bit output on RAM.

Conclusion

In this lab assignment, we explored various aspects of sequential logic design using behavioral modeling in VHDL and delved into the practical implementation of memories and an Arithmetic Logic Unit (ALU).

Post Lab Questions

1) One-hot encoding is a method of converting categorical data into binary format, where each category is represented by a unique binary value (0 or 1) in a vector, making it easier for machine learning algorithms to work with categorical data.

Lab 10: Introduction to Programmable Logic Controllers & Ladder Logic Design

Date performed: 11/7/2023

Introduction:

A Programmable Logic Controller (PLC) is a robust industrial computer used to automate manufacturing processes, lighting, elevators, and more. PLCs are known for their reliability, real-time capabilities, and resistance to harsh conditions. They replaced unreliable relay logic systems from the 1960s. PLCs use ladder logic for programming, where symbols represent actions and conditions. They make decisions based on inputs, controlling outputs accordingly. Relays play a key role in noise reduction and voltage control. In our project, we used a CLICK Series Micro PLC with a relay to control a stack light with push buttons. The CLICK Software simplifies ladder logic programming and real-time monitoring. The PLC communicates via a standard serial port and retains its program after power cycles.

Part I

Experiment of Part I

In our experiment, we established a fundamental latching circuit by connecting a PLC, push buttons, and a lamp using the provided wires. We carefully followed the instructions to create these connections. The "+24V" terminal on the C0-01AC power supply was linked to the normally open (green) pushbutton and the relay's "5" terminal. Meanwhile, the "0V" terminal was connected to the relay's "7" terminal and the lamp's "X2" port. We also joined the "4" terminal on the normally open pushbutton and the relay's "3" terminal to the "1" terminal on the normally closed (red) pushbutton. Additionally, we established a connection from the "2" terminal on the normally closed pushbutton to the relay's "8" terminal and the lamp's "X1" terminal. This configuration allowed us to control the lamp using the push buttons, turning it on when the normally open pushbutton was pressed and off when the normally closed pushbutton was activated, showcasing the fundamental functionality of a latching circuit.

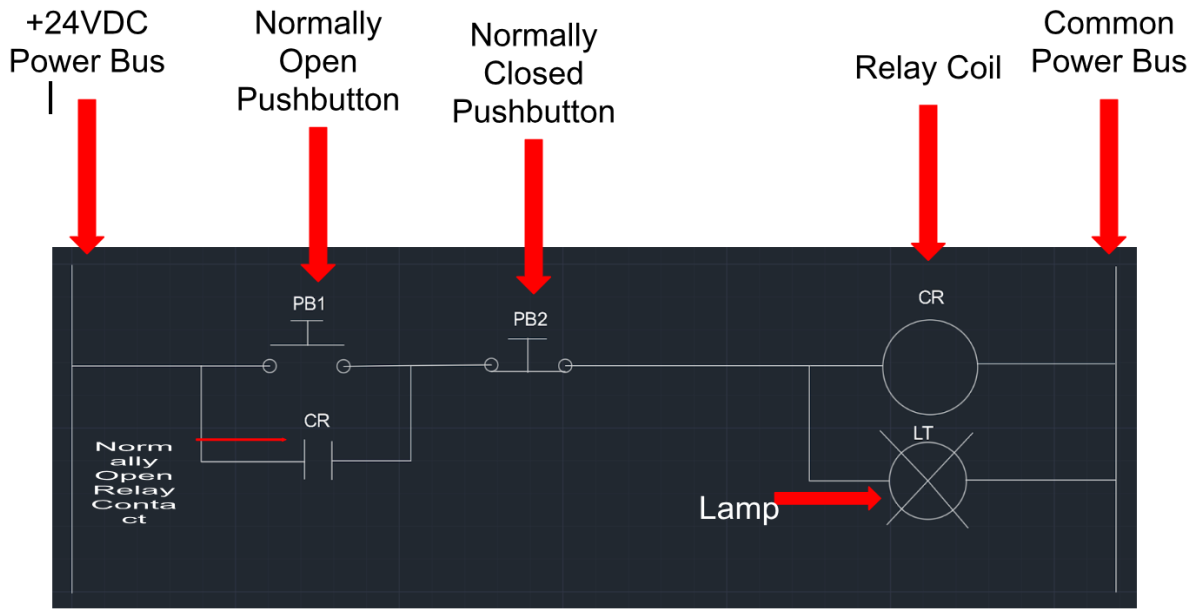
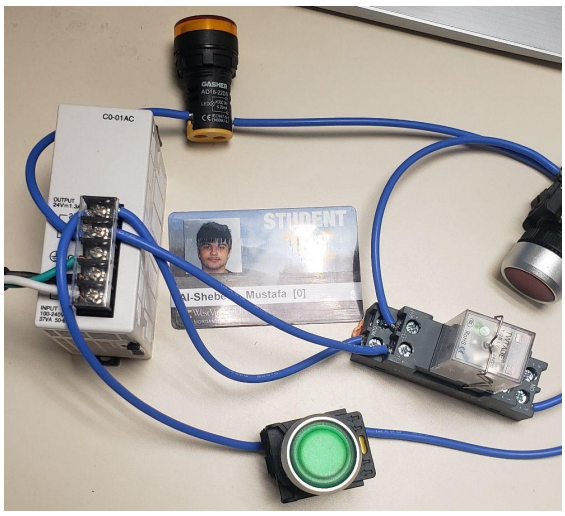
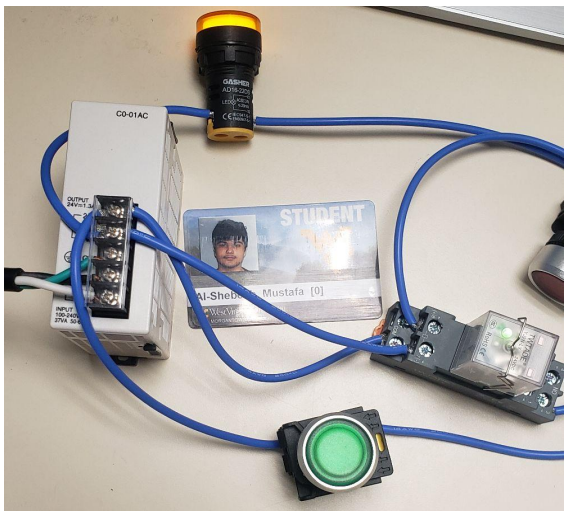


Figure 1.A: PLC Connection Diagram

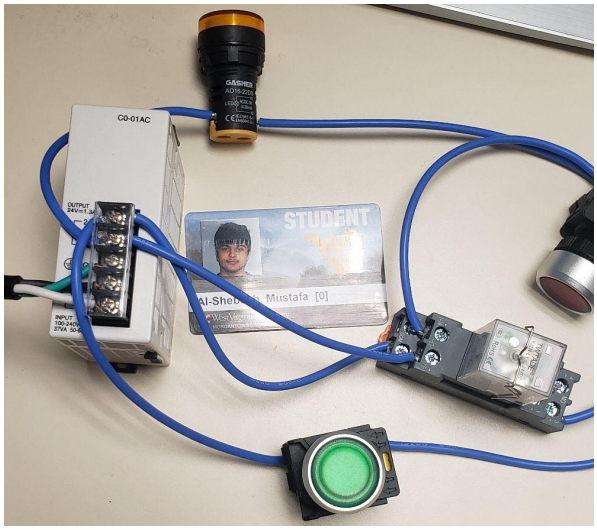
Results of Part I



PLC is powered on, lamp is off



Green Push Putton pressed, Lamp is on



Red push Button Pressed, Lamp is off

Part II

Experiment of Part II

In this part of the lab, we used the CLICK Programming Software to model the latching circuit created in Part 1 as a ladder logic program. We started by selecting the appropriate PLC model and created a ladder logic program with rungs and instructions, mirroring the electrical connections. After wiring the devices to the PLC's input and output points, we conducted a syntax check to ensure there were no errors in our program. Then, we compiled and saved the project.

Next, we powered up the PLC and established a connection with the computer. We transferred the project to the PLC and set the PLC to "RUN" mode. The program behaved similarly to the electrical latching circuit from Part 1, where pressing the green push button turned on the lamp, and pressing the red push button turned it off. The ladder logic symbols resembled the electrical symbols, as ladder logic models relay logic circuits. We documented the working latching program in both on and off states for our lab report. This part of the lab involved coding the PLC and connecting it to the components, demonstrating its ability to control the lamp's operation effectively.

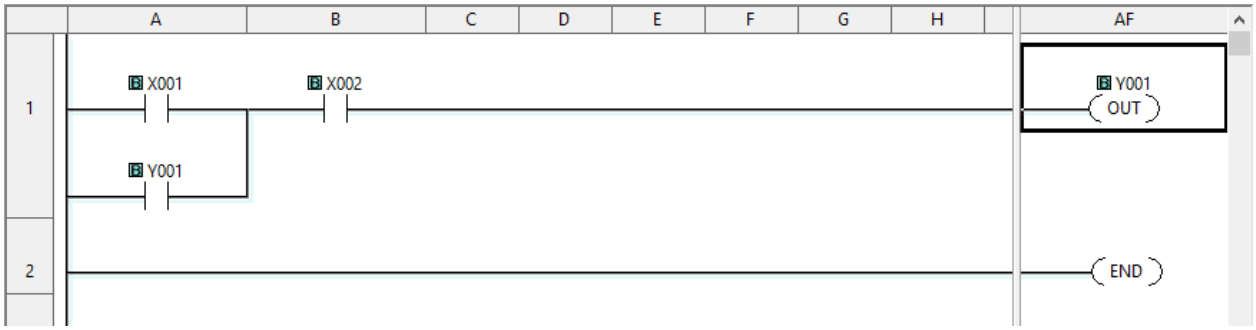
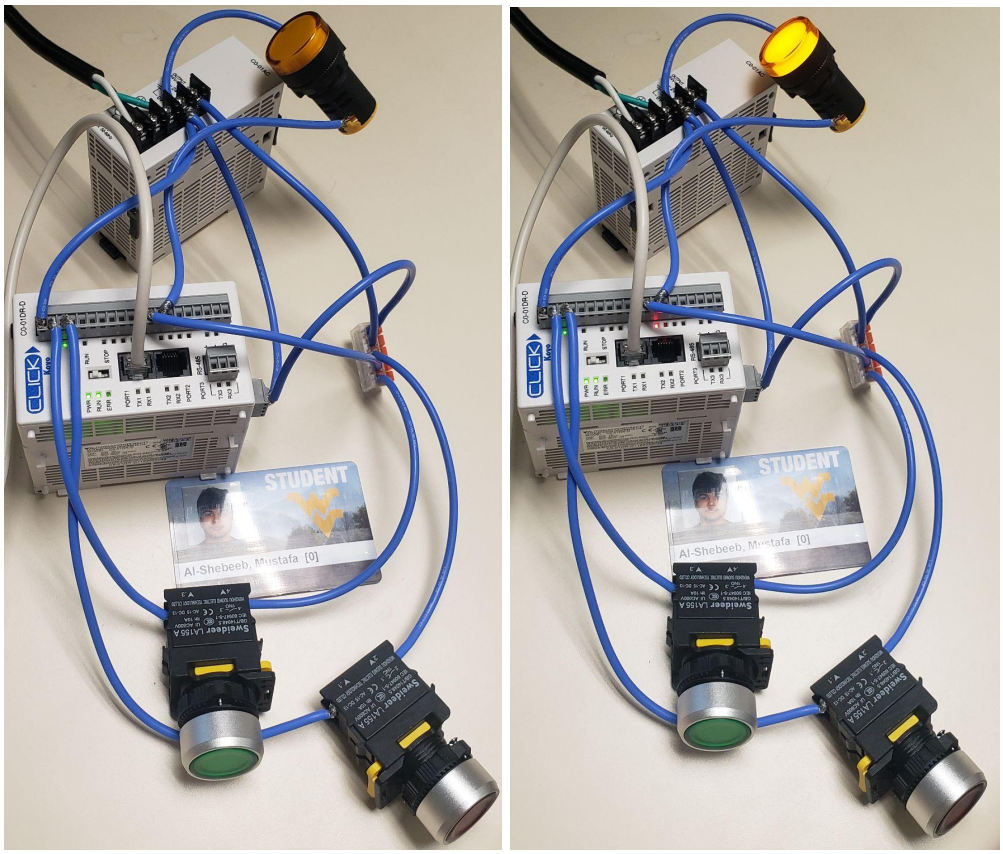
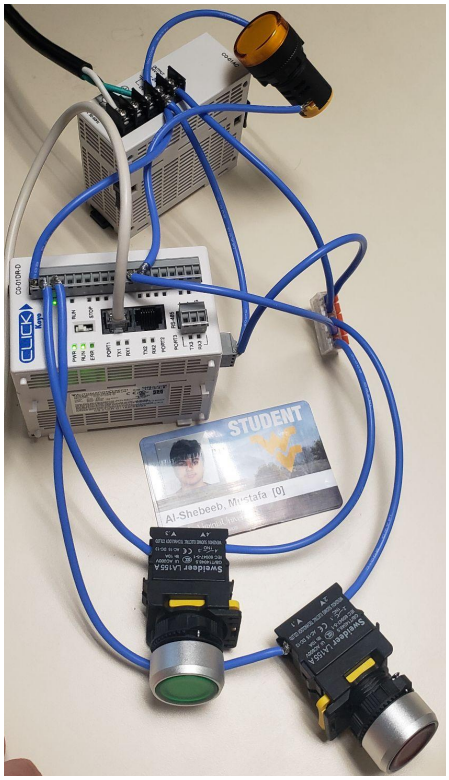


Figure 2.A: Click Program for Part II

Results of Part II



PLC and CPU is powered On, Lamp is Green push button pressed, Lamp is on off.



Red push button pressed, Lamp is off.

Part III

Experiment 1 of Part III

In Experiment 1, we tackled a Boolean Algebraic equation, C1.C2' + C3 = Y1, and translated it into ladder logic within the PLC. To represent this equation, we arranged contacts labeled C1, C2, and C3 in parallel and series configurations to mimic the desired behavior. After writing the program onto the PLC, we employed the Data View Monitor to test the ladder logic program. The setup included ensuring the correct connections, such as linking the "24V" terminal on the Power Supply to the "C3" terminal on the PLC CPU, the "Y1" terminal on the PLC CPU to the "X1" terminal on the lamp, and connecting the "X2" terminal on the lamp to the "0V" terminal on the power supply. By manually overriding the status of the internal C bits using the Data View Monitor, we verified that the program effectively controlled the lamp based on the Boolean logic. We documented the working program with pictures showcasing the PLC and a screenshot of the program, accompanied by our student ID for the lab report.

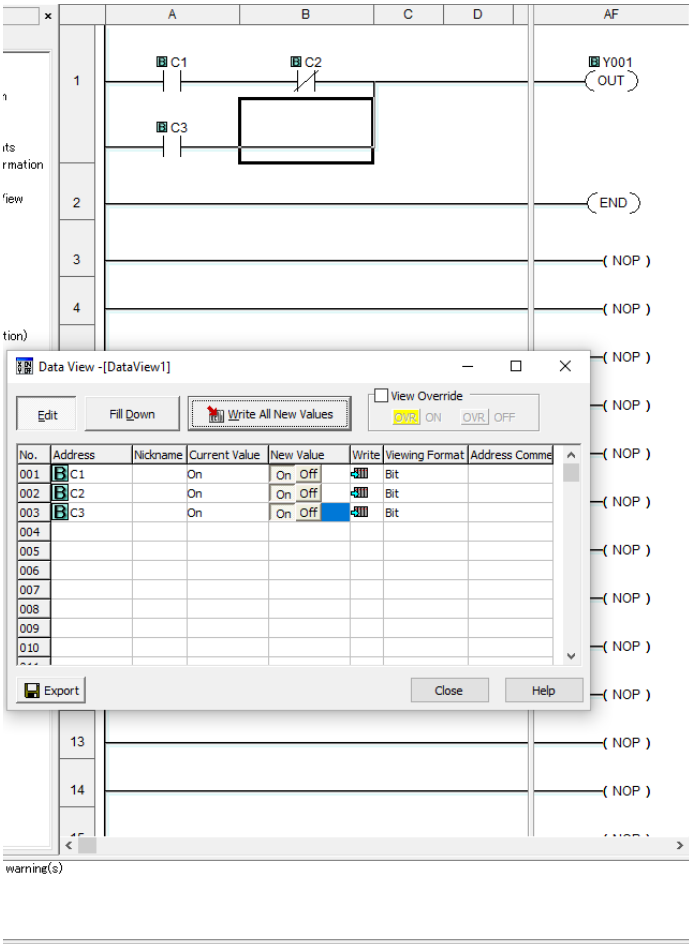
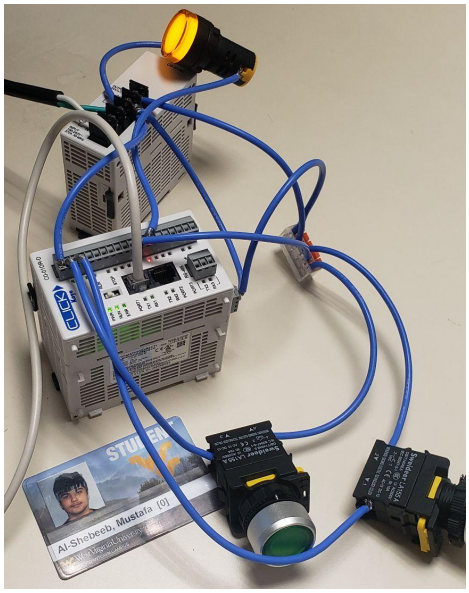


Figure 3.A: Click Program for Experiment 1 of Part III

Results of Experiment 1



PLC using Boolean Logic from the Data View Monitor.

Experiment 2 of Part III

In Experiment 2, addressing concerns about frequent jams in a conveyor system transition point, we took on the role of a controls engineer. To mitigate the issue, we designed a Boolean equation that governs the conditions for the upstream conveyor to run smoothly. The criteria included ensuring the downstream conveyor is operational, allowing the package to move past the new sensor if the pre-existing sensor is unblocked, and prompting the package to "pull up" to the new sensor if the pre-existing sensor is blocked. The designed logic was implemented as a PLC program using the CLICK system. The lamp served as an indicator of the upstream conveyor's status, and we verified the functionality by testing the program on the PLC and CPU setup. Prior to testing, we ensured accurate connections between the power supply, PLC CPU, and the lamp. This experiment addressed the specific challenges in the conveyor system, providing a tailored solution through Boolean logic and PLC programming.

Boolean Equation: Y1 = (C1 AND NOT C2) OR (NOT C3)

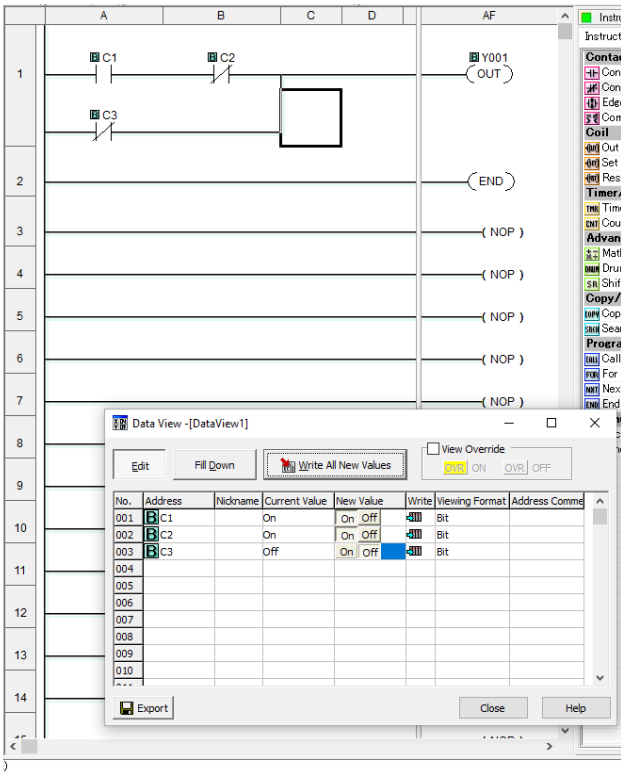
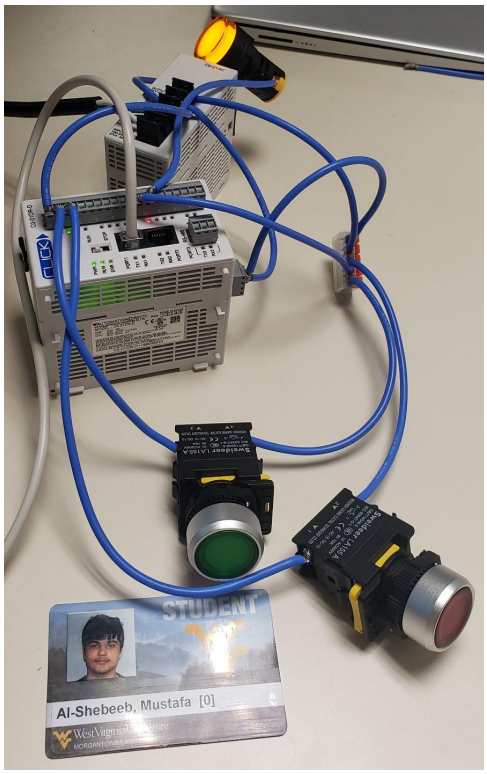


Figure 3.B: Click Program for Experiment 2 of Part III

Results of Experiment 2



PLC using Boolean logic for Conveyor Belt.

Conclusion

This experiment provided an intriguing shift in both software and physical components, particularly in my encounter with Programmable Logic Controllers (PLCs). Being unfamiliar with PLCs initially, the lab significantly enhanced my understanding of their functionalities. The primary challenge involved the meticulous task of accurately connecting devices to designated ports, with occasional struggles to prevent short circuits during wiring setup. However, the CLICK software proved to be comprehensible, making the overall experience insightful. In comparison to previous experiments with the DE10 Lite Board, this lab offered a refreshing change, simplifying program development. Overall, it was an engaging and informative exploration of PLCs, shedding light on their pivotal role in industrial automation.

Post-Lab Questions

1. The Motorola ACE Series is a PLC platform designed for industrial automation, offering a reliable solution for control and monitoring applications. Known for its rugged design and durability, the ACE Series is suitable for harsh environments. This platform provides a range of I/O options, communication protocols, and programming capabilities to meet the demands of diverse industrial settings. With a focus on robust performance, the Motorola ACE Series is a choice for applications where resilience and efficiency are critical.

2. The optimal approach for implementing traffic lights is as a Finite State Machine (FSM). This method, characterized by well-defined states and condition-based transitions, ensures a structured and efficient control system.