CPE 272 Lab Portfolio

Kollin Labowski

Spring 2020

Table of Contents

CPE 272 Course Reflection

This course introduced me to several important concepts that will benefit me in my future classes and in my career. Although I am a computer science major, I believe that understanding the foundations of computer engineering will be very helpful when I am working in my field. Creating programs with Quartus II allowed me to develop skills working with a different programming language. It was interesting to see how VHDL differs from languages I am more familiar with, such as Java and Python, but also how they are similar. One concept I've learned in this course that I find to be particularly intriguing is modular design. It is very convenient that each module in a modular system can be created and designed on its own and can be interchanged with ease if necessary. The concept of modular design in computer engineering is very similar to object-oriented programming, as different objects can be created separately and used when they are needed. It was also fascinating to see how the concept of modular design can be used to create computer systems, both simple and complex. The final project for this course showed just how efficient modular design can be when working on more complicated projects. Another particularly useful concept I learned in this course was sequential logic. Although on the surface it does not appear to be a particularly complicated topic, the applications it has are nearly limitless. Mealy and Moore state diagrams initially proved to be difficult, but by practicing creating and reading them for lab assignments, they became much easier to work with. I like that so many simple computer systems we use in our everyday life can be modeled using state diagrams. This makes their functions so much easier to understand, and it will come in handy if I ever need to program a similar device myself. While these broader computer engineering concepts were helpful, several smaller concepts were also

introduced to me in this lab that are still very useful. Among these are Boolean algebra and K-maps. I had some previous experience with truth tables and logic expressions, but these concepts made using and understanding them much easier. Logic expressions are still important in computer science, so these skills will translate well into my future classes and career. While the labs in this class could be difficult at times, I enjoyed learning about the different type of logic circuits and their applications. When I was looking for colleges to attend last year, I was seriously considering majoring in computer engineering instead of computer science. While I ultimately decided on computer science, this lab has at least rekindled my interest in the subject, and I will have to consider learning more about the field by taking some more high-level classes. Nonetheless, this lab has helped me to learn a lot of important concepts that are significant in the field of computer science as well as computer engineering. Even though most of the programming assignments in this class were simpler in structure than a typical computer science assignment I would complete, they encouraged me to think in a different way than I normally would when completing a computer program. Having a background of completing programs with a wide range of goals will make me more able to tackle assignments and projects in my future. Overall, this lab was incredibly educational, and I will plan to use many of the skills I learned in this class in my future.

Lab Report Revision History

Lab 1: Basic Logic Gates and Seven Segment Displays

- In the Experiment section of Part II, an additional sentence was added to explain the simplification of the Boolean expression

Lab 2: Programmable Logic Devices

- In the Results section of Part II, sentences were revised to accurately describe the confusion between 1's and 0's without incorrectly defining minterms
- Revised Post-Lab Question 1 to refer to the connections in anodes and cathodes as single nodes, not single pins as they were incorrectly defined previously

Lab 3: Field Programmable Gate Array

- Minor changes made to formatting of figures and tables

Lab 4: Introduction to VHDL

- Renamed Part I to Part II and Part II to Part III to more accurately reflect the lab handout
- In the Experiment section of Part II, wording was revised for describing the creation of the truth table to ensure that the report describes the lab steps in the order of completion

Lab 5: Introduction to Logic Simulation and Modular Design

- Replaced figure that was missing portion of code to updated figure containing all necessary code

Lab 6: Behavioral Modeling of Combinational Logic Circuits Using VHDL

- Minor changes made to formatting of figures and tables

Lab 7: Sequential Logic

- Minor changes made to formatting of figures and tables

Lab 8: Sequential Logic Design

- Minor changes made to formatting of figures and tables

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

- Minor changes made to formatting of figures and tables

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

- Minor changes made to formatting of figures and tables

Name: Kollin Labowski

Partner: Matthew Winston

Computer Engineering 272-002

Lab 1: Basic Logic Gates and Seven Segment Displays

Date Completed: 01/21/2020

Introduction

This experiment introduced different types of basic logic gates and how they each function using integrated circuits. These logic integrated circuits were used in conjunction with a breadboard to create circuits representing various Boolean logic expressions. The circuits were created by using truth tables to predict the behavior of each circuit. A seven-segment display was also wired to display a number using the breadboard.

Part I: Building Logic Diagram Circuits

- **Experiment**

  Initially a logic expression was derived from a circuit containing a NOT gate, a NAND gate, and an OR gate, which can be seen in **Figure 1**. A truth table was created to assist the formulation of the logic expression. The circuit was then created on an Altera Board using three integrated circuits, with A, B, and C each being connected to a button. A second logic diagram was analyzed for its truth table and logic expression; however, it was not wired on the breadboard. This circuit can be seen in **Figure 2.**

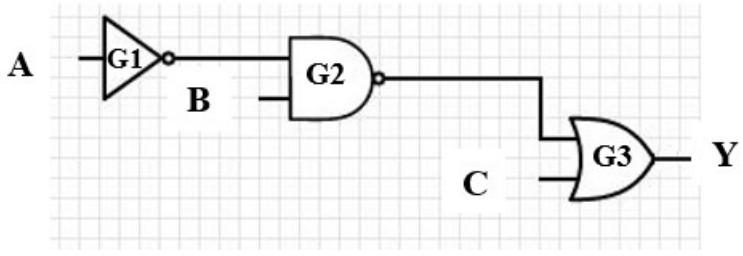


Figure 1: Logic Diagram A

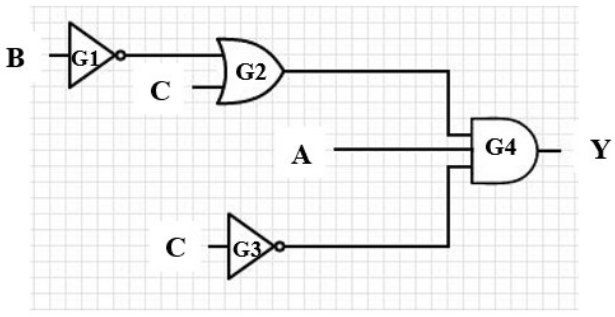


Figure 2: Logic Diagram B

- **Results**

The first expression can be seen in **Equation** 1, and it was used to create the first truth table, seen as **Table 1.** This table predicted a behavior which would return a 0 only in the case that A and C are 0 and B is 1. The LED connected to the end of the logic expression turned on only in this case, indicating the circuit was working correctly. Initially it was believed that the experiment had not produced the correct output due to confusion of the buttons being at a state of 1 by default, but this issue was soon resolved after consulting the teaching assistant. **Equation 2** was derived from the logic diagram in **Figure 2**, and it was used to fill out another truth table, seen in **Table 2**.

Equation 1: (A' * B)' + C = Y

Table 1: Truth Table for Logic Diagram A (Figure 1)

A	B	C	G1=	G2=	G3=Y=
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	0	0
0	1	1	1	0	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	0	1	1

Equation 2: (B' + C) * A * C' = Y

Table 2: Truth Table for Logic Diagram B (Figure 2)

A	B	C	G1=	G2=	G3=	G4=Y=
0	0	0	1	1	1	1
0	0	1	1	1	0	0
0	1	0	0	0	1	0
0	1	1	0	1	0	0
1	0	0	1	1	1	1
1	0	1	1	1	0	0
1	1	0	0	0	1	0
1	1	1	0	1	0	0

Part II: Combinational Logic Networks

- **Experiment**

  A circuit was to be created using a simplified logic expression made from the minterms of a truth table representing numbers divisible by 3. This was done by adding all combinations of the values of A, B, and C which would cause a value of 1 according to the truth table. This expression was then simplified using Boolean algebra and by substituting an XOR symbol. Just as in the first part, this circuit was created on an Altera Board, this time using an AND and an XOR integrated circuit. **Table 3** below shows the given table of binary to decimal conversions used to create the truth table.

Table 3: Binary to Decimal Conversion Table

Binary			Decimal
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

- **Results**

  Table 4 below shows the completed truth table, showing logic highs only for those numbers divisible by 3. **Equation 3** shows the initial expression derived from the minterms, and **Equation 4** shows the simplified form of the equation. The LED connected to the circuit which was wired was only supposed to turn on if the letter combination of A, B, and C formed a binary representation of a number divisible by 3. The LED reacted exactly the way it was predicted to. **Figure 3** displays the diagram of the logic expression which was used as a reference for wiring on the Altera Board.

Table 4: Binary to Decimal Conversion Table Numbers Divisible by Three

Decimal	Binary			Y
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0

Equation 3: A'BC + ABC' = Y

Equation 4: B * (A xor C) = Y

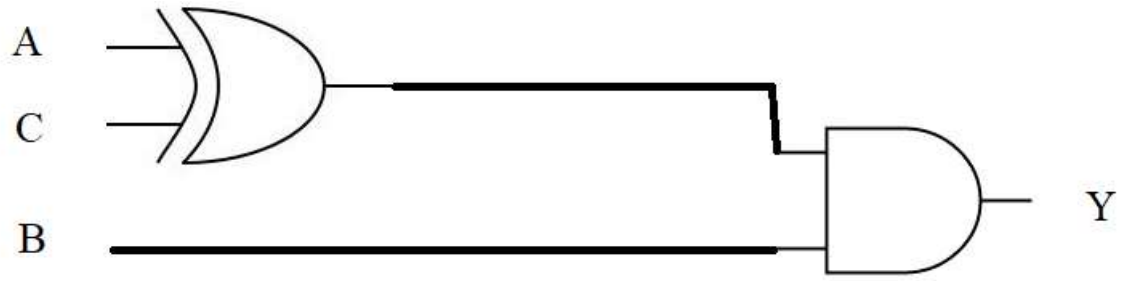


Figure 3: Logic Diagram for Numbers Divisible by Three

Part III: Seven Segment Display

- **Experiment**

  A common anode seven-segment display was provided, and the goal was to hardwire it so that it displayed the number "5." Using a diagram from the internet, the corresponding segments and pins were compared, and the correct pins were connected to ground. Prior to wiring the seven-segment display, a diagram was created on the lab handout to show how each pin should be connected, and it was used as a reference.

- **Results**

Figure 4 shows the diagram of the seven-segment display as was filled out in the lab handout. Initially confusion over the wiring of a breadboard caused the seven-segment display to fail to light up at all. However, after adjusting the design and reviewing the diagram of a seven-segment display again, the correct changes were made to cause the number “5” to be displayed on the seven-segment display. This was in line with what was intended to happen according to the lab project document and can be seen in **Figure 5**.

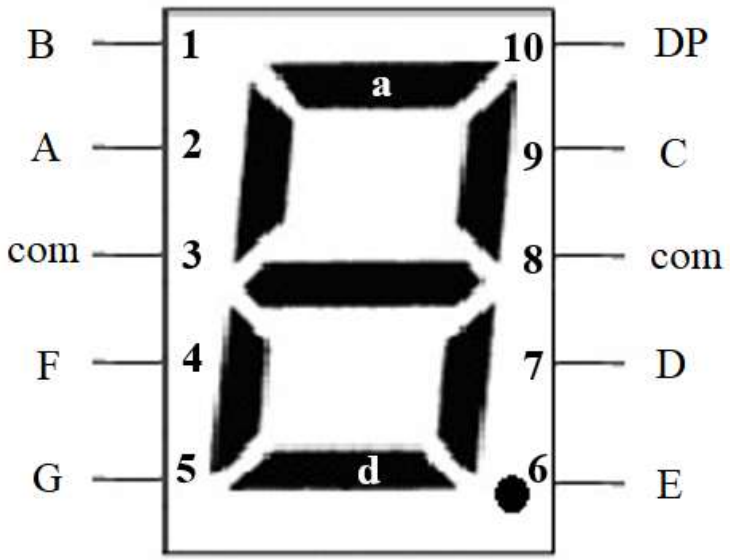


Figure 4: Seven-Segment Display Pin Diagram

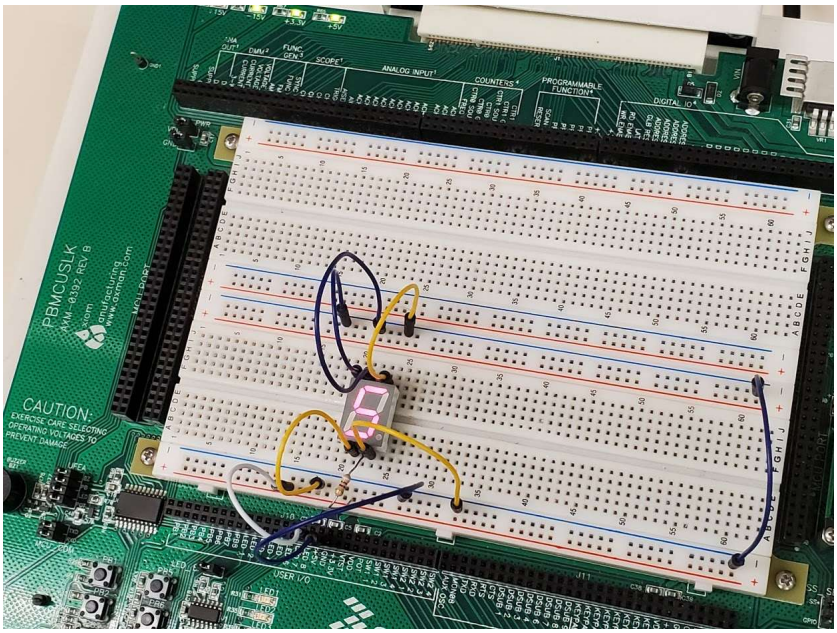


Figure 5: Wired Seven-Segment Display

Conclusions

In this lab, I expanded my knowledge of digital logic, specifically how logic equations, truth tables, and logic diagrams relate to and can be derived from each other. Prior to the lab I had some experience with integrated circuits and breadboards, however I had never worked with an Altera Board. In addition, it has been a few years since my last experience with digital logic design, so it was very helpful to be able to review. The main problems my group ran into were confusion over the states of the buttons, and accidently wiring the seven-segment display incorrectly the first time. Luckily neither of these issues were serious problems, and each of them were learning experience that will ideally prevent the same mistakes from being made in the future.

In terms of feedback, the lab was well structured and was not too difficult to follow. It did, however, seem to be very advanced for the first lab, and although I was able to finish it on time, I feel that it would have been beneficial to all of the students if we had reviewed the content more beforehand and had been taught the information over a longer time period. Nonetheless, the lab was very interesting, and the general structure of the lab was simple enough to understand while still being enriching.

Questions

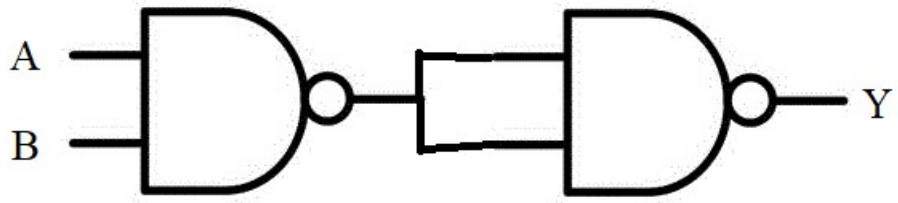
Post-Lab 1 Questions

1. Draw a truth table (similar to the last experiment) to represent number 0-7 divisible by 2. Again, don't include zero. Find the output minterm logic expression.

Decimal	Binary			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

Logic expression: A'BC' + AB'C' + ABC'
Simplified Logic Expression: (AB' + B) * C'

2. Draw a logic diagram to show how you could construct an AND gate using only NAND gates. Label your inputs and outputs



3. Comparing the two expressions of the divisible by 3 experiment (i.e. Part II: 2a and 2b), what was the overall benefit of manipulating the expression if they both give you the same exact output and why the effort was useful? (HINT: Think about if you were given a choice to construct one the expressions into a breadboard circuit. Which one would you choose and why?)

   It was helpful to manipulate the expression because it allowed the circuit to be created using fewer gates. This allowed the circuit to be wired more quickly, and it required fewer integrated circuits. If the circuit needed to be manufactured on a larger scale for any reason, this would greatly reduce the costs of production by reducing the amount of required integrated circuits.

4. When using a common anode seven segment display, does applying a logic high or logic low signal light up a segment? Why does it function that way or why wouldn't it be the opposite logic signal of what you stated?

   Applying a logic low lights up a certain segment in a common anode seven-segment display. This is because once power is connected to the display at any point, the power is transferred to all of the points, so in order to actually light up the segments, you must connect them to ground so that the electricity has a path to travel through the desired pins.

Pre-Lab 2 Questions

1. What is a binary subtractor?

   A binary subtractor is a circuit that allows for the subtraction of one binary number from another and outputs their difference.

Name: Kollin Labowski

Partner: Ryan Stout

Computer Engineering 272-002

Lab 2: Programmable Logic Devices

Date Completed: 01/28/2020

Introduction

This experiment focused on the use of programmable logic devices to simulate circuits using computer code for producing desired outputs on a breadboard. The desired output in this experiment was to be coded in such a way that it would cause the hexadecimal numbers 1 through F to be displayed on a common anode seven-segment display. This was accomplished by producing a truth table that would allow for this pattern of numbers to display on the seven-segment display, and subsequently assigning the minterms to their respective pins on the WinCupl programming software.

Part I: Testing Seven Segment Display

- **Experiment**

  This portion of the experiment involved the testing of a selected seven-segment display on the breadboard, in order to determine whether it was functioning correctly. This was accomplished by hardwiring the number 4 onto the display by connecting the necessary pins to ground. The success of the experiment was to be determined on whether the number 4 was properly displayed on the display.

- **Results**

  After connecting the necessary wires between the seven-segment display and ground, the number 4 was successfully displayed on the display. This indicated that the seven-segment display was functioning correctly, and it was determined that it would be safe to proceed to the next step. Unfortunately, although the display was correctly functioning, it was not actually used during the lab due to an error with the programmable logic devices. Nonetheless, this helped to establish the important practice of monitoring and testing equipment in order to potentially avoid wasting long periods of time debugging.

Part II: Designing a Decoder

- **Experiment**

  The purpose of this portion of the experiment was to develop expressions to represent the logic appropriate for allowing the hexadecimal numbers 1-F to be displayed on a seven-segment display when the input corresponds to its equivalent binary number.

This was accomplished by first creating a truth table that showed which segments would need to be connected to ground at each binary input for the correct hexadecimal number to be displayed. Once the truth table was completed, the minterms were found for each letter segment on the display by adding each combination that produced a logic 0. They were then written out in preparation for programming into the Generic Array Logic device. Shown below in **Table 1** is a given table, representing decimal numbers and their hexadecimal equivalents, which was used to determine which hexadecimal numbers should be displayed on the seven-segment display.

Table 1: Decimal to Hexadecimal Conversions

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

- **Results**

The truth table was created as seen below in **Table 2**. The minterms were found based off this table, and they are shown below in **Figure 1**. In preparation for the programming of the GAL chip, a diagram schematic, shown in **Figure 2**, was consulted. After looking at the minterms and the schematic diagram, a new diagram was created showing where each of the input and output pins would be located on the GAL chip. This is shown below in **Figure 3**. The first time this part of the lab was completed, the 1's and 0's were incorrectly assigned such that they caused the minterm equation to be its opposite. This caused an incorrect output, and it caused a large amount of time and energy to be wasted. Luckily, this issue was eventually resolved after reassigning the values and deriving the minterms again correctly.

Table 2: Binary Input to Seven-Segment Display Output

W	X	Y	Z	a	b	c	d	e	f	g	Display
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	0	0	1	1	9
1	0	1	0	1	1	1	0	1	1	1	A
1	0	1	1	0	0	1	1	1	1	1	B
1	1	0	0	1	0	0	1	1	1	0	C
1	1	0	1	0	1	1	1	1	0	1	D
1	1	1	0	1	0	0	1	1	1	1	E
1	1	1	1	1	0	0	0	1	1	1	F

a = (!W & !X & !Y & Z) # (!W & X & !Y & !Z) # (W & !X & Y & Z) # (W & X & !Y & Z)

b = (!W & X & !Y & Z) # (!W & X & Y & !Z) # (W & !X & Y & Z) # (W & X & !Y & !Z) # (W & X & Y & !Z) # (W & X & Y & Z)

c = (!W & !X & Y & !Z) # (W & X & !Y & !Z) # (W & X & Y & !Z) # (W & X & Y & Z)

d = (!W & !X & !Y & Z) # (!W & X & !Y & !Z) # (!W & X & Y & Z) # (W & !X & !Y & Z) # (W & !X & Y & !Z) # (W & X & Y & Z)

e = (!W & !X & !Y & Z) # (!W & !X & Y & Z) # (!W & X & !Y & !Z) # (!W & X & !Y & Z) # (!W & X & Y & Z) # (W & !X & !Y & Z)

f = (!W & !X & !Y & Z) # (!W & !X & Y & !Z) # (!W & !X & Y & Z) # (!W & X & Y & Z) # (W & X & !Y & Z)

g = (!W & !X & !Y & !Z) # (!W & !X & !Y & Z) # (!W & X & Y & Z) # (W & X & !Y & !Z)

Figure 1: The derived minterms for each segment of the display are shown

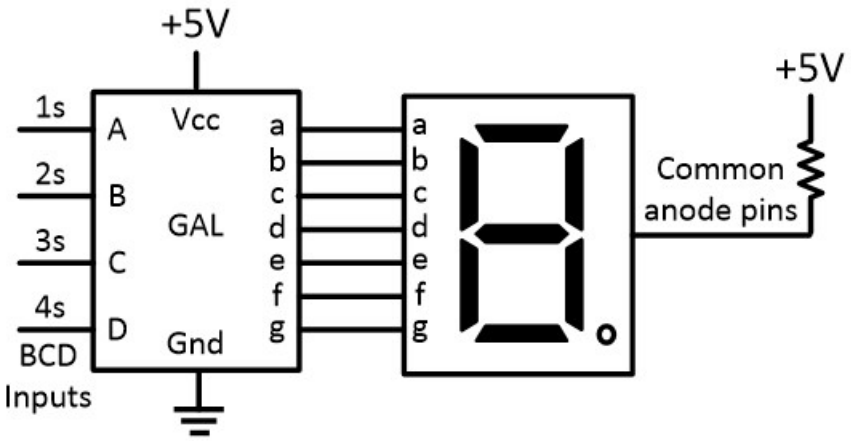


Figure 2: A schematic detailing the wiring of the decoder and seven-segment display

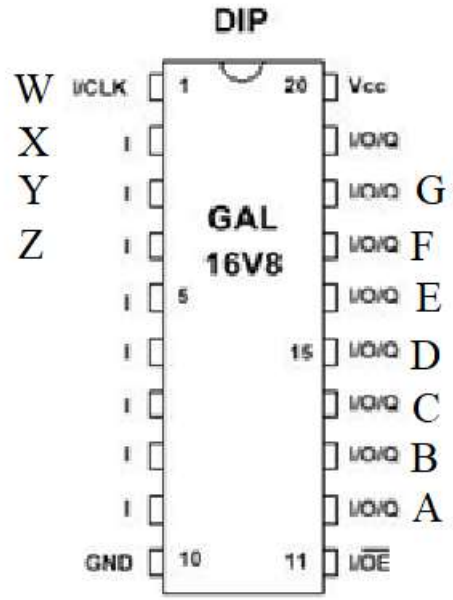


Figure 3: A diagram of the GAL chip with the inputs (W-Z) and outputs (A-G) labeled with the pins they are to be programmed to

Part III: Compiling GAL Code and Demonstration

- **Experiment**

  This portion of the lab involved the compilation of the logic expressions derived in Part II into the WinCupl software for use on the Generic Array Logic device. The input pins were denoted as W, X, Y, and Z in line with the truth table, and they were assigned to pins 1-4 on the GAL. Pins 12-18 were denoted as the output pins, and were labeled a-g, corresponding to the segments on the seven-segment display. As can be seen in **Figure 4** below, each output pin was set equal to its respective logic expression. The symbol "!" was used for NOT, "&" was used for AND, and "#" was used for OR when programming the logic expressions. Once the code was completely written, it was compiled to check for syntax errors. It was then prepared to export to the GAL for testing whether the hexadecimal numbers 1-F were displayed on the display.

- **Results**

  Unfortunately, due to errors with the GAL, the measure of success would come not from the output of 1-F on the seven-segment display, but rather from confirmation from the TA after reviewing the code. After reviewing the logic expressions assigned to each pin, it was determined that the logic expressions were logically and syntactically valid. If the program were able to be exported to the GAL, it would have produced the desired output if it was wired correctly, unless there were some sort of extraneous circumstance. **Figure 4** below shows the final code which was completed and would have been used on the seven-segment display.

```
Name     CPE_LAB_2 ;
PartNo   LTS-5001AWC ;
Date     1/28/2020 ;
Revision 01 ;
Designer Engineer ;
Company  WVU ;
Assembly None ;
Location  ;
Device   g16v8 ;

/* *************** INPUT PINS *********************/
PIN   1  = W                        ; /*                              */
PIN   2  = X                        ; /*                              */
PIN   3  = Y                        ; /*                              */
PIN   4  = Z                        ; /*                              */

/* *************** OUTPUT PINS *********************/
PIN   12  =  a                      ; /*                              */
PIN  13   =  b                      ; /*                              */
PIN   14  =  c                      ; /*                              */
PIN   15  =  d                      ; /*                              */
PIN  16   =  e                      ; /*                              */
PIN  17   =  f                      ; /*                              */
PIN   18  =  g                      ; /*                              */

/** Logic Equations **/


a = (!W & !X & !Y & Z) # (!W & X & !Y & !Z) # (W & !X & Y & Z) # (W & X & !Y & Z);

b = (!W & X & !Y & Z) # (!W & X & Y & !Z) # (W & !X & Y & Z) # (W & X & !Y & !Z) # (W & X & Y & !Z) # (W & X & Y & Z);

c = (!W & !X & Y & !Z) # (W & X & !Y & !Z) # (W & X & Y & !Z) # (W & X & Y & Z);

d = (!W & !X & !Y & Z) # (!W & X & !Y & !Z) # (!W & X & Y & Z) # (W & !X & !Y & Z) # (W & !X & Y & !Z) # (W & X & Y & Z);

e = (!W & !X & !Y & Z) # (!W & !X & Y & Z) # (!W & X & !Y & !Z) # (!W & X & !Y & Z) # (!W & X & Y & Z) # (W & !X & !Y & Z);

f = (!W & !X & !Y & Z) # (!W & !X & Y & !Z) # (!W & !X & Y & Z) # (!W & X & Y & Z) # (W & X & !Y & Z);

g = (!W & !X & !Y & !Z) # (!W & !X & !Y & Z) # (!W & X & Y & Z) # (W & X & !Y & !Z);
```

Figure 4: Displayed is the final code containing all necessary logic expressions for the correct functionality of the seven-segment display in WinCupl.

Conclusion

In this lab, I learned about how programmable logic devices work, and how they can be used to greatly simplify the amount of work necessary for wiring a logic circuit. This was helpful because it showed an alternative to using IC chips that requires less work and is faster. Although we were not able to test the GAL device, it was still interesting to learn about the process of how it is supposed to be used. Unfortunately, my group made the mistake of mixing up the logic 1's and 0's in the truth table, which caused us to waste a large amount of time. We were able to finish the lab the correct way, but it came at the expense of frustration due to the mistake. Luckily, making this mistake should make me more cautious when working with minterms in the future, which should hopefully prevent me from making a similar significant mistake.

Questions

Post-Lab 2 Questions

1) What are the unique characteristics of a common anode and a common cathode 7-segment display?

   Common anode displays have pins that are all electrically connected at one node. The result of this is that segments of the display are lit up by connecting their respective pins to ground. In a common cathode display, the cathodes of each pin are connected to a single node. This means that segments of the display are lit up by connecting their respective pins to power.

2) List the logic values (HIGH or LOW) needed to display a 7 on a common anode 7-segment display.

A = LOW
B = LOW
C = LOW
D = HIGH
E = HIGH
F = HIGH
G = HIGH

Pre-Lab 3 Questions

1) What does FPGA mean?

FPGA stands for “field-programmable gate array,” and it is an integrated circuit that contains programmable logic blocks. It can be programmed and reprogrammed to a specific logic expression.

2) Why are FPGAs used over other programmable Devices such as the GAL chip?

FPGAs can simulate very complicated logic expressions as they contain a very large number of gates. They may be chosen over GAL chips in larger projects for their lower price per gate and greater computational power. However, they are not likely to be used for smaller projects, as they can be expensive as a package, and are not very efficient for less complicated circuits.

Name: Kollin Labowski

Partner: Ethan Eichelberger

Computer Engineering 272-002

Lab 3: Field Programmable Gate Arrays

Date Completed: 2/4/2020

Introduction

This lab focused on the familiarization of the Quartus II circuit-designing software. Learning this tool was accomplished by building a design which was given in the handout and testing its output on an Altera Board. This software was them used to create a circuit which would act as a binary subtractor. Both a binary half subtractor and a full subtractor were created and tested on the Altera Board.

Part I: Getting Started with Quartus II

- **Experiment**

  This portion of the experiment served as the introduction to the Quartus II software and to Altera Boards. The procedure was to follow each step in the process to create a file, construct a circuit by importing different ports and gates, and export the logic to the Altera Board to test its output. A diagram of the circuit was given in the lab handout, so all that was needed in this portion of the experiment was to copy this circuit. The circuit's logic was tested on the Altera Board by setting each of the two inputs to a pin for the switch of the board, and the output to a pin for an LED. The output was then analyzed by noting which input combinations caused the LED to turn on or off.

- **Results**

  The steps of the experiment were followed without any significant issues. The file was successfully created, and the circuit was made to match the diagram in the lab handout. **Figure 1** shows the circuit that was created during lab using the Quartus II software. Once the output was examined on the Altera Board, it was noted that the created circuit was logically equivalent to an XOR gate.

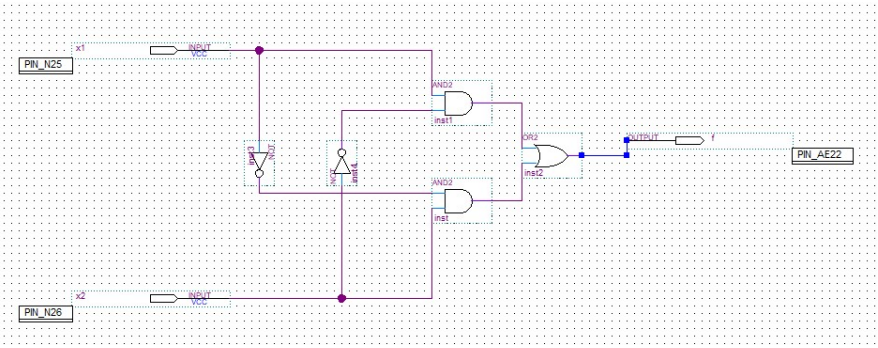


Figure 1: An XOR Gate Equivalent Circuit Created in the Quartus II Software

Part II: Binary Subtractors

- **Experiment**

  The purpose of this portion of the lab was to use the Quartus II software to create circuits which could act as binary subtractors. A binary half subtractor and full subtractor were designed. A truth table was derived for the half subtractor, in which the inputs A and B were used along with the outputs Borrow Out (Bout) and Difference (Diff). The Bout output was evaluated as a 1 only in the case that completing the operation of A – B would result in the need to carry out a number. The Diff output was evaluated as whatever the result of A - B would produce, either a 1 or a 0. The minterms for both Bout and Diff were then found and were simplified using Boolean algebra and XOR equivalences to allow for fewer gate to be required. These new logic expressions were used to create a diagram, and this diagram was then programmed into the Quartus II software. The output of the half subtractor was tested for accuracy. The same process was repeated for the full subtractor; however, the main difference was the presence of a third input, Borrow In (Bin). The outputs were computed in the same way as the first, however the Bin column acted as a third number to be subtracted. Just as with the half subtractor, minterms were created and simplified by using Boolean algebra and XOR equivalences. A new circuit diagram was created using these logic expressions, and the circuit was then created on the Quartus II software. The output was tested by connecting the output pins to two LEDs on the Altera Board and comparing their behavior to the truth table.

- **Results**

  The truth table for the half subtractor, seen in **Table 1**, was created using the method detailed above. **Equation 1** and **Equation 2** show the simplified logic expressions of the outputs Bout and Diff respectively. These logic expressions were used to create the circuit on the Quartus II software, which can be seen in **Figure 2**. **Table 2** shows the truth table, which was created for the binary full subtractor, in which the outputs were evaluated as described in the Experiment section. **Equation 3** and **Equation 4** are the logic expressions for Bout and Diff for the full subtractor. The simplified logic expressions were used to create the circuit in Quartus II, seen in **Figure 3**. It is worth noting that the logic expression in **Equation 4** does not match up with the circuit shown in **Figure 3**. This is because originally it was believed that the necessary logic expression was A(B xnor C) + A'(B xor C), however, after completing the circuit it was determined it could be simplified further using the definition of an XOR gate. Despite this change, the circuit was not updated to show this, and as such it was not created in the most efficient way. Nonetheless, both the half subtractor and full subtractor produced the expected output when tested on the Altera Board, and so they were each determined to have been correctly designed.

Table 1: The Truth Table for the Binary Half Subtractor

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

Equation 1: Bout = A'B
Equation 2: Diff = A xor B

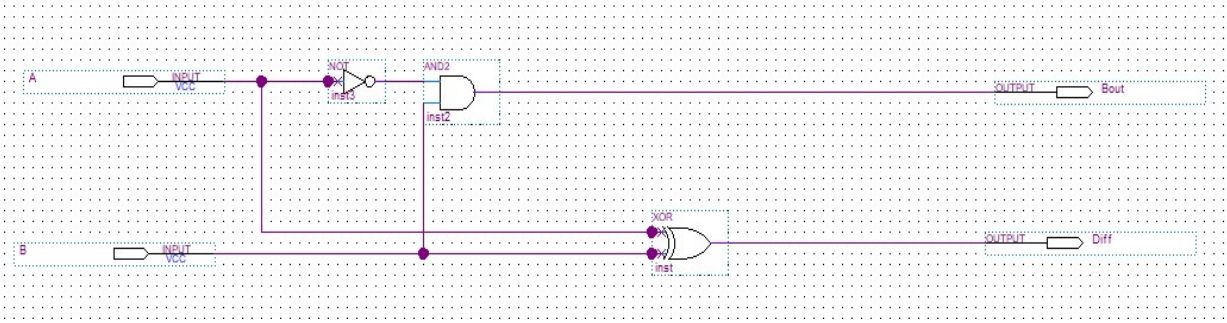


Figure 2: The Binary Half Subtractor Created in the Quartus II Software

Table 2: The Truth Table for the Binary Full Subtractor

A	B	Bin	Bout	Diff
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Equation 3: Bout = A'B + A'C + BC
Equation 4: Diff = A xor (B xor C)

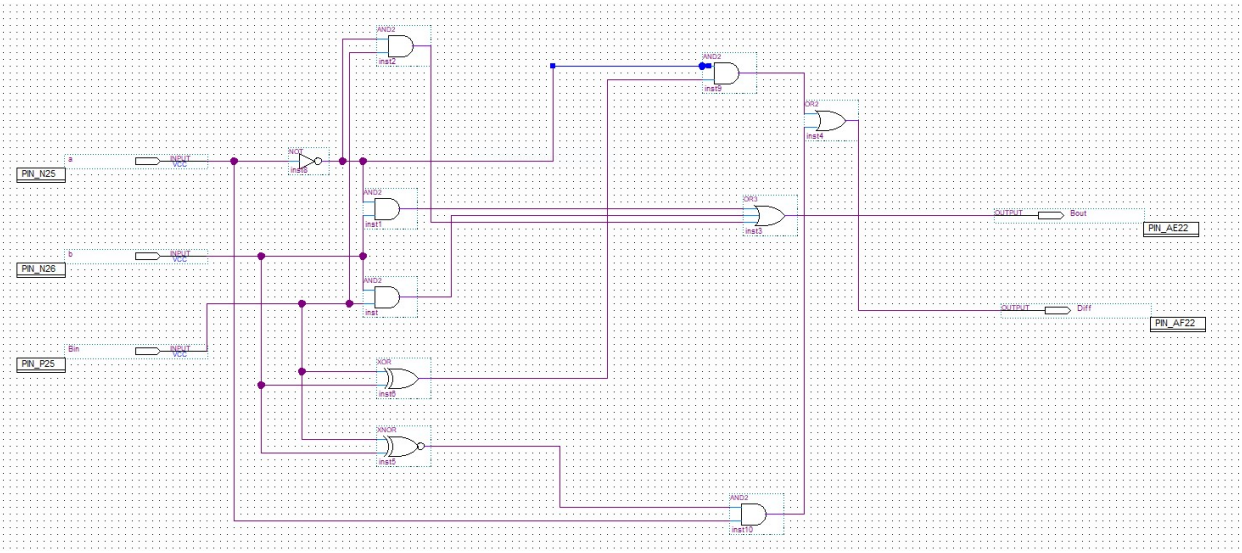


Figure 3: The Binary Full Subtractor Created in the Quartus II Software

Conclusion

In this lab, I learned about how to use the Quartus II software to create and test logic circuits. I have had previous experience with similar software, but it was interesting to see how this software was different. It was also very helpful to learn about circuit designing again, as it has been a long time since I have worked with circuit designing software. For the final part, my partner and I spent more time than we should have need to on the binary subtractor design because we did not simplify the logic expression as much as we could have. We still produced the desired output, but it used more gates than it should have needed to and wasted unnecessary time. Luckily, this situation should cause me to be more thorough with my logic expression simplification in the future, so that I will be able to minimize the amount of work I will have to do and create more efficient designs.

Questions

- **Post-Lab 3 Questions**

1. Does the RUN/PROG switch need to be in RUN or PROG in order to successfully program the DE2 Board?

It needs to be in RUN.

2. Perform the binary subtraction below. Make sure to show both the Borrow out (Bout) and Difference (Diff) output results.

100	111
- 001	- 100
B = **001**	B = **000**
D = **011**	D = **011**

- **Pre-lab 4 Questions**

1. What does VHDL mean?

VHDL stands for **VHSIC Hardware Description Language**

2. What type of variable can be declared in the architecture declaration section of VHDL code?

A **signal** can be declared in the architecture declaration section.

3. When entering a single bit into VHDL are single quotes or double quotes used to contain the bit (Thus, “0” or ‘0’)?

Single quotes are used, so ‘0’ or ‘1’.

Name: Kollin Labowski

Partner: Reed Tuttle

Computer Engineering 272-002

Lab 4: Introduction to VHDL

Date Completed: 2/11/2020

Introduction

The purpose of this experiment was to become familiar with programming logical expressions using VHDL code. It included learning how to create and use VHDL files to be tested on an Altera Board. The Quartus II software was used to program two different logic expressions, one of which was given in a diagram and one of which was to act as a binary adder. Both logic expressions were tested for accuracy on an Altera Board.

Part II: Combinational Logic Design (VHDL)

Experiment

Prior to this portion of the experiment, instructions were reviewed and followed to grow familiar with using Quartus II for coding with VHDL. The purpose of this part was to practice coding with VHDL to get used to how it works. **Figure 1** below shows a logic diagram which was given in the project document. The goal was to code the given logic diagram in VHDL and test its output on an Altera Board. The logic expression was created within a declared architecture, and the keywords “not”, “and”, and “or” were used to simulate the logic. This logic was then exported to the Altera Board by connecting each input to a switch and the output expression to an LED. The output from the Altera Board was used to create a truth table modeling the behavior of the circuit, and the success of the implementation was determined by analyzing the truth table.

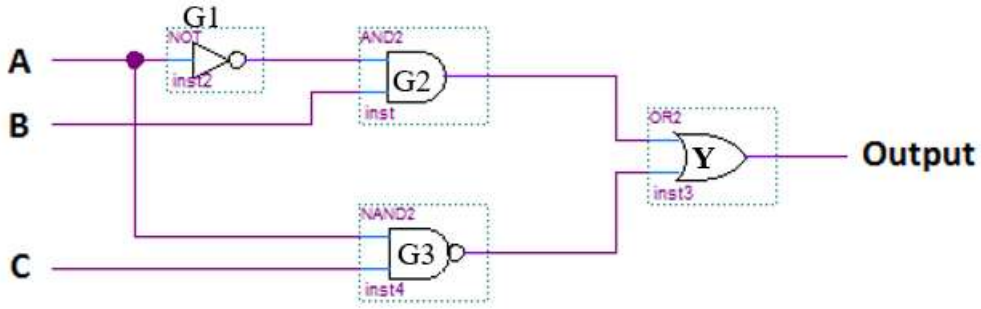
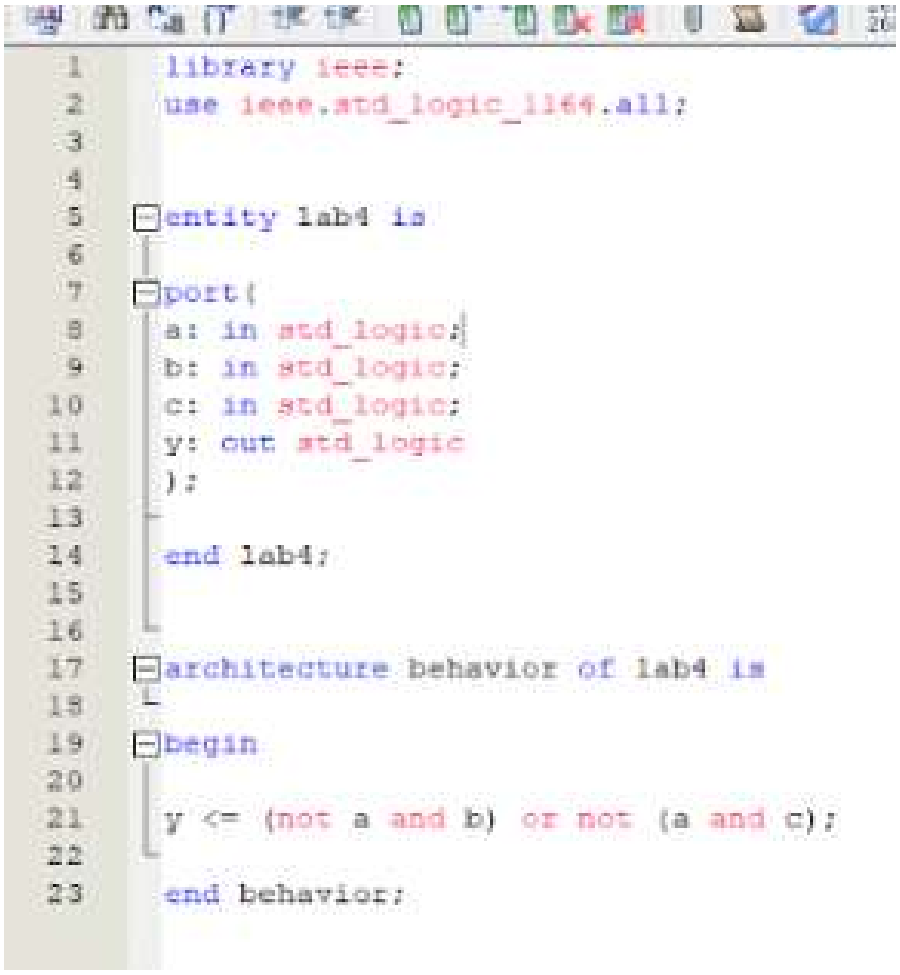


Figure 1: The diagram given in the Lab 4 project document

Results

The code created using VHDL can be seen in **Figure 2** below. Its output was analyzed to create a truth table alongside the VHDL code. This truth table can be seen in **Table 1**. Comparing the behavior of the output to the truth table showed that they were the same. This implied that the code was successfully created and that this portion of the experiment worked as intended.



```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4
5   entity lab4 is
6
7   port(
8    a: in std_logic;
9    b: in std_logic;
10   c: in std_logic;
11   y: out std_logic
12   );
13
14   end lab4;
15
16
17  architecture behavior of lab4 is
18
19  begin
20
21   y <= (not a and b) or not (a and c);
22
23   end behavior;
```

Figure 2: The VHDL code created to act in the behavior of the diagram in Figure 1

Table 1: The truth table created to predict the behavior of Figure 1

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

Part III: VHDL Binary Addition

Experiment

This part of the experiment was like the previous part, as it also required creating a program using VHDL. This required the use of VHDL code to create expressions for adding binary numbers. A truth table was first created to determine what the expected output was. There were two outputs, one for the sum of the 3 bits, and one for the number to be carried out after the addition. The minterms for these two outputs were found by finding their sums of products. VHDL code was then created with six inputs labeled X1-X6, and four outputs labeled F1-F4. In addition, two signals C1 and C2 were created to calculate the value which would be carried between the bits. As with the previous part of the lab, the keywords “not”, “and”, and “or” were used to represent their respective logical operations. Because the first bit to be added for output F1 would never include a carried value, it was given its own special case which did not include the carry in its expression. Likewise, signal C1 also had a simpler special case, as it was not dependent on any prior carried values. Once the VHDL code was completed, it was tested on the Altera Board by setting the inputs to switches and the outputs to adjacent LEDs. The output was determined to work as expected if it matched the truth table and successfully added the two binary numbers.

Results

The truth table which was created and used to derive the logic expressions can be seen in **Table 2**. The minterms which were derived from the outputs of the truth table can be seen in **Equation 1** and **Equation 2**. The complete VHDL code is shown in **Figure 3**. The outputs were tested on the Altera Board, and the specific combinations which were tested can be seen in **Table 3**. Once the output was tested and compared to the expected output, it was determined that the circuit worked correctly. **Figure 4** shows the output on the Altera Board, which corresponds to the second input combination in **Table 3**. This portion of the experiment worked as expected, although the first time it was tested, the output was not correct. After some troubleshooting however, the problem was identified as being a minor error in translation of the expression to VHDL code, and it was resolved relatively easily.

Table 2: The expected inputs and outputs for each singular bit of the binary adder

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

Equation 1: Sum = A'B'C + A'BC' + AB'C' + ABC

Equation 2: Carry = A'BC + AB'C + ABC' + ABC

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


entity lab4_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 lab4_part3;


architecture behavior of lab4_part3 is

Signal c1 : std_logic;
Signal c2 : std_logic;

begin

f1 <= (not x1 and x4) or (x1 and not x4);
c1 <= (x1 and 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);
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);

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);

end behavior;
```

Figure 3: The VHDL code for the binary adder

Table 3: Tested input combinations for binary adder

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

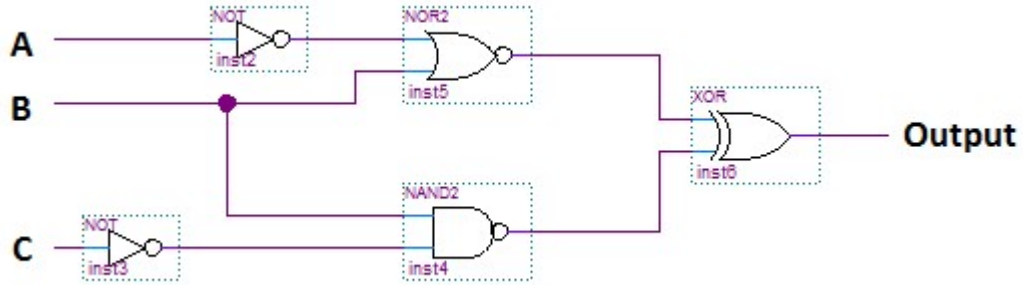
Figure 4: The binary adder adding 101 (5) and 101 (5) to get an output of 1010 (10)

Conclusion

This experiment was useful as it helped me to become familiar with VHDL code. The benefits of VHDL were very apparent as it allowed for you to save time compared to programming the full logic diagrams. A downside of using VHDL code is that it is not as easy to picture what is supposed to happen when looking at the code, and as such it may be more difficult to troubleshoot in certain circumstances. This showed me that it is important to decide how you want to simulate your logic expression, as each of the ways went over in lab have their own benefits and drawbacks. Unlike the previous labs, my partner and I did not run into any significant issues when completing the assignment. This was likely due in part to the fact that this lab was building on the previous one, and so it was a similar process to what we have done before.

Questions

- **Post-Lab 4 Questions**
  1. Find the output of the following combinational logic diagram by using the step by step logic gate approach by hand.



Inputs	Output Steps					
ABC	**A'**	**(A'+B)'**	**C'**	**(BC')'**	**(A'+B)' xor (BC')'**	**Output**
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. Perform the following binary addition (Make sure to show both the carry and sum output results) :

C	C
11100	10111
+ 01001	+ 01011
S	S
S = 100101	**S = 100010**
C = 11000	**C = 11111**

- **Pre-Lab 5 Questions**

1. What does Logic simulation mean?

Logic simulation means using software to predict the behavior of certain models, such as digital circuits.

2. Explain the concept of modular design.

Modular design is a type of design in which a larger structure is divided into smaller modules, which can be independent of one another, often so it can be more easily managed.

3. What type of statement is used to bring a lower hierarchy VHDL file into the top level entity VHDL file?

A “component” statement

Name: Kollin Labowski

Partner: Marlena Schoppert

Computer Engineering 272-002

Lab 5: Introduction to Logic Simulation and Modular Design

Date Completed: 2/18/2020

Introduction

The purpose of this lab was to serve as an introduction to Waveform Simulator as well as the use of component statements in VHDL code. These skills were developed by creating a binary adder, much like in the previous lab, however by using different techniques. These techniques included the use of logic vectors as opposed to normal inputs and outputs, as well as the use of component statements to use code more efficiently.

Part I: XNOR Logic Simulation

Experiment

This portion of the experiment involved the use of Waveform Simulator to simulate the behavior of an XNOR gate. This was accomplished by following the steps in the lab handout to set up the Waveform Simulator. A VHDL file was then created, and it was used to program the logic expression for an XNOR gate, using the keyword “xnor”. Once the Waveform Simulator was set up to use the logic expression, it was run, and different inputs combinations were tested for accuracy. The simulation was determined to be an accurate depiction of an XNOR gate if the output responded according to the behavior of the XNOR gate.

Results

The output observed on the Waveform Simulator was determined to match the expected output of the XNOR gate. Therefore, the logic expression was correctly written and implemented into the Waveform Simulator. **Figure 1** shows the VHDL code which was used to create the XNOR logic expression. **Figure 2** shows the input combinations and their respective outputs as generated in the Waveform Simulator.

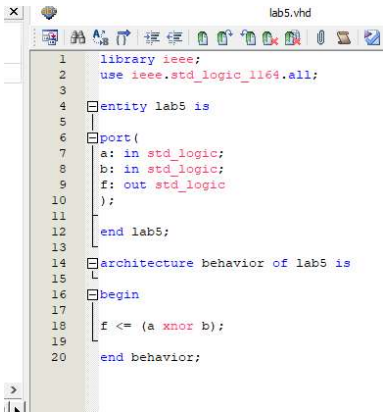


Figure 1: The VHDL code used to simulate the behavior of an XNOR gate

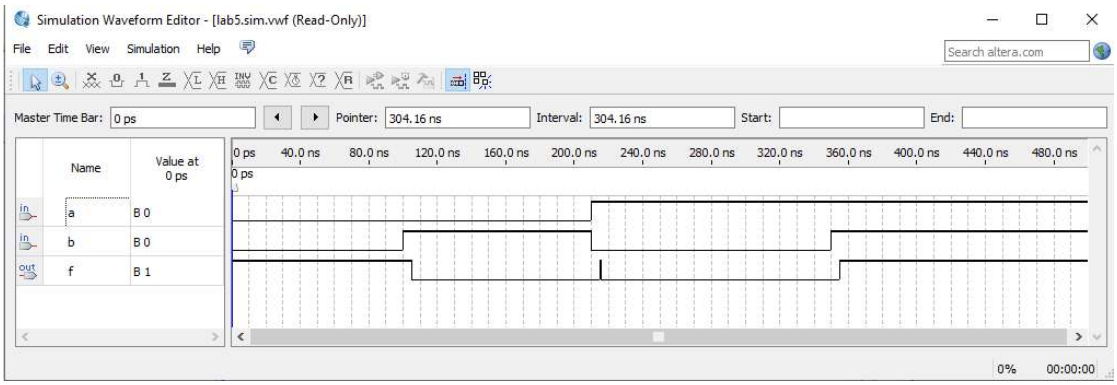


Figure 2: An XNOR gate simulated using Waveform Simulator in Quartus II

Part II: Binary Addition with Vectors/Logic Simulation

Experiment

This portion of the experiment required the creation of a 3-bit binary adder using VHDL code. This was accomplished by reusing the logic found in Lab 4, but by using STD_LOGIC_VECTOR notation as opposed to individual inputs and outputs. Each input (X) and output (F) were replaced with the appropriate portion of the input and output vectors. Other than this, the code was identical to the code created in Lab 4. This code was then simulated using Waveform Simulator, and some different input combinations were tested. Success of this experiment depended on whether the output matched what was expected, which was that the two input combinations could be added together to get the output.

Results

The wave output did match what was expected based on the various inputs. This indicated that the VHDL code was successfully translated into the STD_LOGIC_VECTOR format. **Figure 3** shows the VHDL code for the binary adder using vectors. **Figure 4** shows five input combinations and their respective outputs as they were generated in the Waveform Simulator.

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

entity Lab5P2 is

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

end Lab5P2;

architecture behavior of Lab5P2 is

Signal c : std_logic_vector(1 downto 0);

begin

f(0) <= (not x1(0) and x2(0)) or (x1(0) and not x2(0));
c(0) <= x1(0) and x2(0);

f(1) <= (not x1(1) and not x2(1) and c(0)) or (not x1(1) and x2(1) and not c(0)) or (x1(1) and not x2(1) and not c(0)) or (x1(1) and x2(1) and c(0));
c(1) <= (not x1(1) and x2(1) and c(0)) or (x1(1) and not x2(1) and c(0)) or (x1(1) and x2(1) and not c(0)) or (x1(1) and x2(1) and c(0));

f(2) <= (not x1(2) and not x2(2) and c(1)) or (not x1(2) and x2(2) and not c(1)) or (x1(2) and not x2(2) and not c(1)) or (x1(2) and x2(2) and c(1));
f(3) <= (not x1(2) and x2(2) and c(1)) or (x1(2) and not x2(2) and c(1)) or (x1(2) and x2(2) and not c(1)) or (x1(2) and x2(2) and c(1));


end behavior;
```

Figure 3: The VHDL code which used STD_LOGIC_VECTOR format to simulate a binary adder

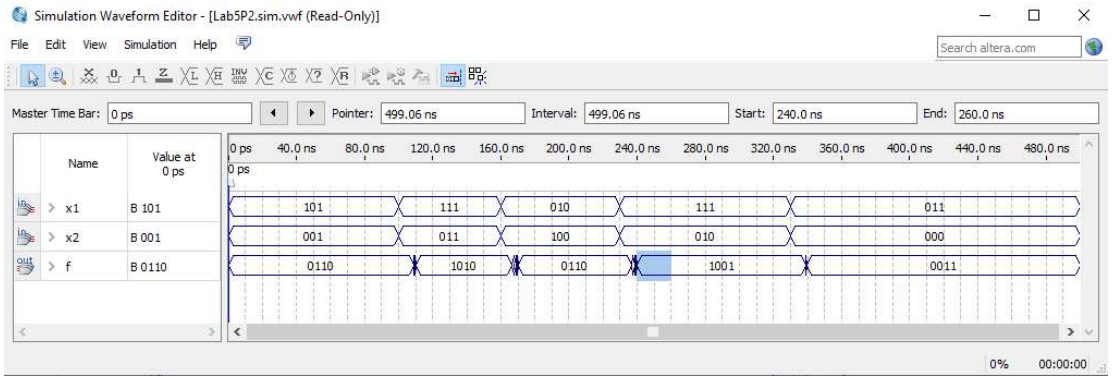


Figure 4: The inputs and output of the binary adder as generated with Waveform Simulator

Part III: VHDL: One-bit Full Adder

Experiment

This portion of the experiment involved the design and programming of a one-bit full adder for use in Part IV. The inputs were labeled A, B, and C, and the outputs were labeled Sum and Carry. Like the previous experiment, Sum was determined by the binary sum of inputs A, B, and C, and the Carry output was 1 if the sum resulted in a carried-out bit. A truth table was created based on this logic, and minterms for the Sum and Carry outputs were derived from this table. These minterms were then programmed into a VHDL file, although the actual logic was not tested in this portion of the lab.

Results

The actual code was not tested in this part, but as it did work in later parts of the experiment, it was found to have been written correctly. **Table 1** Shows the truth table which was derived based on the logic described previously. **Equation 1** and **Equation 2** are the minterms for the Sum and Carry outputs respectively, and they were not simplified. **Figure 5** shows the VHDL code, which includes the minterms coded for the one-bit adder logic.

Table 1: Truth Table for One-bit Binary Full Adder

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

Equation 1: Sum = A'B'C + A'BC' + AB'C' + ABC

Equation 2: Carry = A'BC + AB'C + ABC' + ABC

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity lab5p3 is
5
6  port(
7  a: in std_logic;
8  b: in std_logic;
9  c: in std_logic;
10 sum: out std_logic;
11 carry: out std_logic
12 );
13
14 end lab5p3
15
16 architecture behavior of lab5p3 is
17
18 begin
19
20 sum <= (not a and not b and c) or (not a and b and not c)
21 or (a and not b and not c) or (a and b and c);
22 carry <= (not a and b and c) or (a and not b and c)
23 or (a and b and not c) or (a and b and c);
24
25 end behavior;
```

Figure 5: The VHDL code for the one-bit binary adder

Part IV: Binary Addition with Component Statements

Experiment

This portion of the experiment used the code written in the previous part to simulate a four-bit binary adder. A new VHDL file was created for the four-bit adder, and 8 inputs labeled x0 to x7 as well as 5 outputs labeled f0 to f4 were declared for use in the architecture. Because the code from the previous part was a one-bit adder, this code could be used four times to make a four-bit adder. This was accomplished by declaring a component statement in the architecture that allowed the previously written code to be accessed higher in the hierarchy. Each of the inputs and outputs from the previously written code were declared within the component statement. Then, four instances of the component were made which ran the previously written code but using the values declared at the top of the new VHDL file. Four signals were also used to keep track of the values carried between each bit of the adder. Once all the instances were successfully implemented, the code was compiled, and each input and output were pinned to a place on an Altera Board. Once this was accomplished, the Altera Board was used to test various inputs and determine whether the appropriate output was produced. The experiment was determined to be successful if the output was the binary addition of the two input four-bit binary numbers.

Results

The outputs on the Altera Board did indicate that the logic was programmed correctly. The full VHDL code can be seen in **Figure 6** and **Figure 7**. **Table 2** shows two tested input combinations as well as their expected outputs. **Figure 8** and **Figure 9** show these test inputs and their correct outputs as seen on the Altera Board. Initially, some problems were encountered at this portion of the lab as there was confusion over the syntax and correct location of the component statement and its instances. After some trial and error, as well as some changes to the code, the correct locations were eventually found, and the code compiled and ran correctly.

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  entity lab5p4 is
 5
 6  port(
 7  x0: in std_logic;
 8  x1: in std_logic;
 9  x2: in std_logic;
10  x3: in std_logic;
11  x4: in std_logic;
12  x5: in std_logic;
13  x6: in std_logic;
14  x7: in std_logic;
15  f0: out std_logic;
16  f1: out std_logic;
17  f2: out std_logic;
18  f3: out std_logic;
19  f4: out std_logic
20  );
21  end lab5p4;
22
23
24
25
26  architecture behavior of lab5p4 is
27
28  Signal c0 : std_logic;
29  Signal c1 : std_logic;
30  Signal c2 : std_logic;
31  Signal c3 : std_logic;
32
33  component lab5p3
34
35
36  port(
37  a: in std_logic;
38  b: in std_logic;
39  c: in std_logic;
40  sum: out std_logic;
41  carry: out std_logic
42  );
43
44  end component;
45
```

Figure 6: The first half of the VHDL code for the four-bit full adder

```
40    sum: out std_logic;
41    carry: out std_logic
42    );
43
44    end component;
45
46
47
48    begin
49
50    c0 <= '0';
51
52    instance_label1: lab5p3 port map(
53    a => x0,
54    b => x4,
55    c => c0,
56    sum => f0,
57    carry => c1
58    );
59
60    instance_label2: lab5p3 port map(
61    a => x1,
62    b => x5,
63    c => c1,
64    sum => f1,
65    carry => c2
66    );
67
68    instance_label3: lab5p3 port map(
69    a => x2,
70    b => x6,
71    c => c2,
72    sum => f2,
73    carry => c3
74    );
75
76    instance_label4: lab5p3 port map(
77    a => x3,
78    b => x7,
79    c => c3,
80    sum => f3,
81    carry => f4
82    );
83
84
85    end behavior;
```

Figure 7: The second half of the VHDL code for the four-bit full adder

Table 2: Tested Inputs and Outputs for Binary Full Adder

Binary Addition Inputs		Output
Inputs (X_3, X_2, X_1, X_0)	Inputs (X_7, X_6, X_5, X_4)	Output (F_4, F_3, F_2, F_1, F_0)
0111	0001	01000
1111	0001	10000

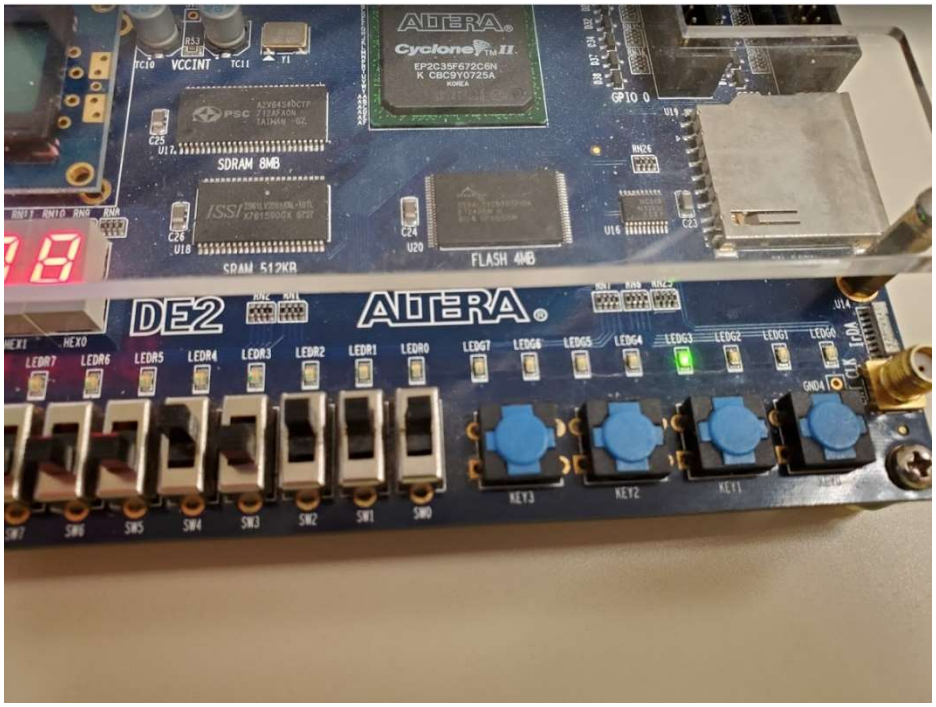


Figure 8: The first of the two input combinations as seen on the Altera Board

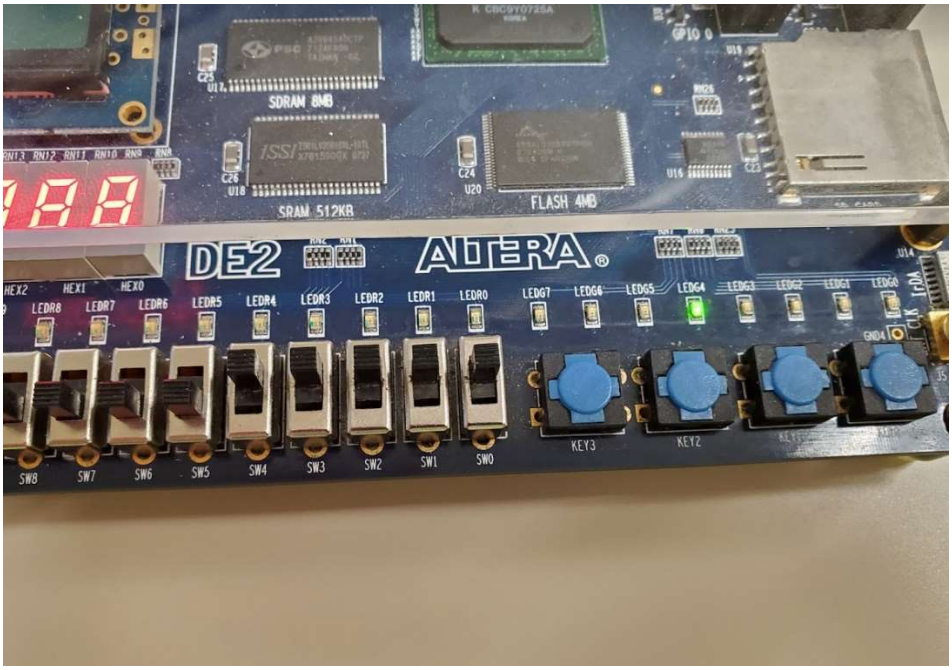


Figure 9: The second of the two input combinations as seen on the Altera Board

Conclusion

This experiment introduced several new, interesting concepts. The Waveform Simulator was an intriguing different way to visualize the results of the logic expression. Learning to use component statements was very helpful, as it allowed my partner and I to program a more complex logic expression without needing to put in too much more work. Luckily, my partner and I did not run into any significant issues during this project, with the only notable one being our confusion of the syntax for the component statement. Now that I understand how to use the component statements correctly, I do not believe I will run into this sort of problem again in the future if I am paying close attention to my work. Overall, this experiment introduced several important concepts I am sure will be useful later in the course.

Questions

- **Post-Lab 4 Questions**

1) Create the declaration statement for a signal named signal_carry that contains 8 bits of information.

Signal signal_carry : std_logic_vector(7 downto 0);

2) Find the errors in the following VHDL code (4 Errors):

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity postlab is
PORT(
topEntityIn_a : in std_logic;
topEntityIn_b : in std_logic;
topEntityOut_1 : out std_logic;
topEntityOut_2 : out std_logic;
);

Architecture behavior of postlab is

begin

Component HalfAdder
PORT(
A : in STD_LOGIC;
B : in STD_LOGIC;
carry : out STD_LOGIC;
sum : out STD_LOGIC
);
end component;

add1: HalfAdder (A => topEntityIn_a, B => topEntityIn _b,
sum => topEntityOut_1, carry => topEntityOut_2);

end behavior;
```

a) There is a semicolon incorrectly placed in the entity statement after topEntityOut_2
b) The component statement is in wrong place (should be before “begin” in architecture)
c) The instance of the component should have the keywords “port map” before the port definition
d) The statement “end postlab;” is missing from the entity declaration

Name: Kollin Labowski

Partner: Jakob Loverde

Computer Engineering 272-002

Lab 6: Behavioral Modeling of Combinational Logic Circuits Using VHDL – I

Date Completed: 2/25/2020

Introduction

The purpose of this lab was to serve as an introduction to common circuits used in computers, including priority encoders, decoders, and multiplexers. Learning about these computer structures was accomplished by writing VHDL code for each structure and testing its output on an Altera Board. Conditional statements were also introduced in this lab, as they were necessary for the coding of the decoder and the multiplexer.

Part I: Priority Encoder

Experiment

This portion of the experiment required the design of a logic circuit to act in the behavior of a priority encoder. According to the lab handout, priority encoders output the index of the most significant bit. In order to understand the structure of an encoder more clearly, a block diagram of a 3 input, 2 output priority encoder was created. Using the inputs and outputs labeled on the diagram, as well as the information provided in the handout, a truth table was created. This truth table was used to code the logic of the encoder into VHDL code. This was accomplished using the maxterms of the logic expression, as they were much shorter than their respective minterms. The code would have been determined to be successfully created in the case that the output matched what was expected in the truth table.

Results

When the code was run on the Altera Board, it was determined that the observed output matched the expected output. **Figure 1** shows the block diagram which was created as described previously. **Table 1** shows the truth table which was derived based on the priority encoder logic. The complete VHDL code, including the use of the maxterms, can be seen in **Figure 2**. It is worth noting that at the time this lab was completed, it was not known that this code was intended to use conditional statements. However, **Figure 3** shows what the body of the architecture would have included had the experiment been completed using conditionals as was intended. Later sections were completed using conditionals as was originally intended.

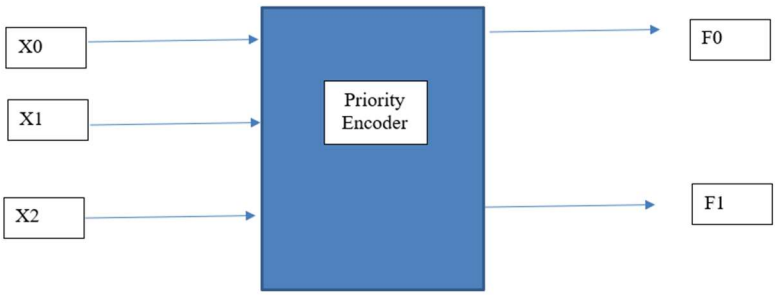


Figure 1: A priority encoder with 3 inputs and 2 outputs

Table 1: Truth Table Output for a 3 Input, 2 Output Priority Encoder

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

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity whatever is
5
6   port(
7   x0: in std_logic;
8   x1: in std_logic;
9   x2: in std_logic;
10  f0: out std_logic;
11  f1: out std_logic
12  );
13
14  end whatever;
15
16  architecture behavior of whatever is
17
18  begin
19
20  f0 <= (x2 or x1 or x0)and(x2 or not x1 or x0)and(x2 or not x1 or not x0);
21  f1 <= (x2 or x1 or x0)and(x2 or x1 or not x0);
22
23  end behavior;
```

Figure 2: The complete VHDL code for the 3 input, 2 output encoder

```
process(x2, x1, x0)
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;
```

Figure 3: The body of the architecture had this section been completed using conditionals

Part II: Decoders

Experiment

This portion of the experiment involved the creation of an active high decoder, which is a type of decoder that uses AND gates as opposed to NAND gates. The logic of the active high decoder was explained in the lab handout. Like the priority encoder, a block diagram was created for a decoder with 3 inputs and 8 outputs. This diagram was then used along with the logic in the project document to create a truth table which would act as a decoder. This part of the experiment involved the use of conditional “if” and “elsif” statements. This was done by creating a process statement and placing the conditional statements within the process. Vectors were used to hold 8 bits each, and this simplified computations greatly. The code was then compiled and run on an Altera Board. As with the encoder, success of this portion of the experiment was determined by comparing the observed and expected outputs.

Results

Because the observed and expected outputs matched, it was determined that the experiment was conducted successfully. **Figure 4** shows the block diagram of the active high decoder which was created. The derived truth table can be seen in **Table 2**. The VHDL code for the decoder can be seen in **Figure 5**. Unlike the first part of the experiment, this section was correctly created using conditionals, as can be seen in the VHDL code.

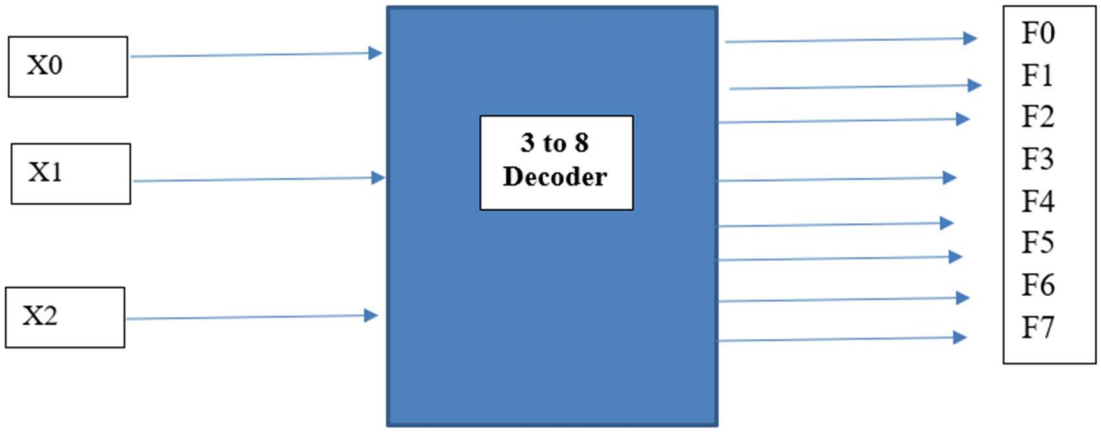


Figure 4: The block diagram for the 3 input, 8 output active high decoder

Table 2: Truth Table Output for a 3 Input, 8 Output Active High Decoder

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

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity whatever2 is
5
6   port(
7   x : in std_logic_vector(2 downto 0);
8   y007 : out std_logic_vector(7 downto 0)
9   );
10
11  end whatever2;
12
13  architecture behavior of whatever2 is
14
15  begin
16
17  process(x)
18  begin
19
20  if (x = "000") then
21     y007 <= "00000001";
22  elsif (x = "001") then
23     y007 <= "00000010";
24  elsif (x = "010") then
25     y007 <= "00000100";
26  elsif (x = "011") then
27     y007 <= "00001000";
28  elsif (x = "100") then
29     y007 <= "00010000";
30  elsif (x = "101") then
31     y007 <= "00100000";
32  elsif (x = "110") then
33     y007 <= "01000000";
34  elsif (x = "111") then
35     y007 <= "10000000";
36
37  end if;
38  end process;
39
40  end behavior;
```

Figure 5: VHDL code for the 3 input, 8 output active high binary decoder

Part III: Multiplexers

Experiment

This portion of the experiment involved the creation of a multiplexer. The logic of a multiplexer was described in the project document, and this was used to create the code for a simple 4-input multiplexer. **Figure 6** shows a block diagram of a 4-input multiplexer that was given in the project document and used as a reference when creating the VHDL code. **Table 3** shows a truth

table that was given in the project document to use as an additional reference, and to show which switch combinations corresponded to each output. These two switches were used to determine which of the four inputs would determine the output. The logic for the multiplexer was implemented to VHDL code using conditional statements. As with the previous structures, the multiplexer logic was compiled and tested on the Altera Board to compare its behavior to the truth table. An additional output table was created including all the output combinations which was more comprehensive than the given truth table. This table was used to generate an output equation, which would provide greater insight into how the underlying structure functions.

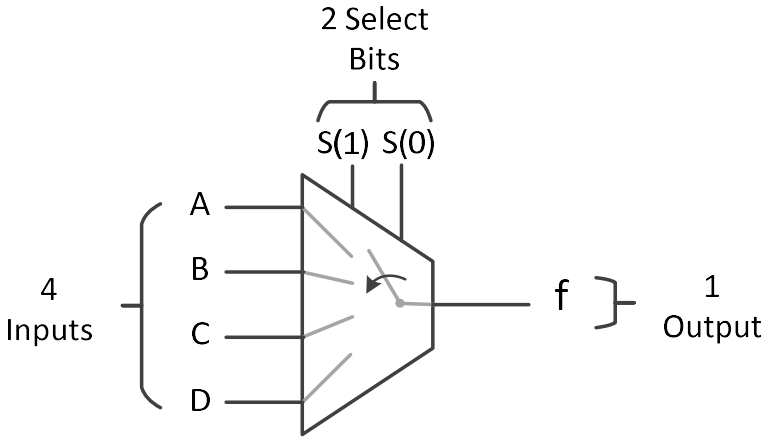


Figure 6: The given block diagram for a 4-input multiplexer

Table 3: The Given Truth Table for the 4-Input Multiplexer

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

Results

Upon testing the circuit, it was determined that the observed and expected output were the same, indicating that the VHDL code was correctly written. **Figure 7** shows the VHDL code which was created for this portion of the project. It should be noted that there was an additional output “FNot” which was added during the next part of the lab. **Table 4** shows the full truth table which lists all possible combinations of inputs for the multiplexer and their respective outputs. **Equation 1** shows the equation that was derived from this table simply through observation.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity whatever3 is
5
6  port(
7  s : in std_logic_vector (1 downto 0);
8  A : in std_logic;
9  B : in std_logic;
10 C : in std_logic;
11 D : in std_logic;
12 f : out std_logic;
13 fNot : out std_logic
14 );
15
16 end whatever3;
17
18 architecture behaveyourrr of whatever3 is
19
20 begin
21
22 process(s)
23 begin
24 if(s = "00") then
25    f <= A;
26    fNot <= not A;
27 elsif(s = "01") then
28    f <= B;
29    fNot <= not B;
30 elsif(s = "10") then
31    f <= C;
32    fNot <= not C;
33 elsif(s = "11") then
34    f <= D;
35    fNot <= not D;
36
37 end if;
38 end process;
39
40 end behaveyourrr;
```

Figure 7: The VHDL code used to act as a multiplexer

Table 4: Truth Table of All Combinations of Multiplexer Inputs

S1	S0	A	B	C	D	F
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	1	1	1	0
0	0	1	0	0	0	1
0	0	1	0	0	1	1
0	0	1	0	1	0	1
0	0	1	0	1	1	1
0	0	1	1	0	0	1
0	0	1	1	0	1	1
0	0	1	1	1	0	1
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	0	0	0	1	0
0	1	0	0	1	0	0
0	1	0	0	1	1	0
0	1	0	1	0	0	1
0	1	0	1	0	1	1
0	1	0	1	1	0	1
0	1	0	1	1	1	1
0	1	1	0	0	0	0
0	1	1	0	0	1	0
0	1	1	0	1	0	0
0	1	1	0	1	1	0
0	1	1	1	0	0	1
0	1	1	1	0	1	1
0	1	1	1	1	0	1
0	1	1	1	1	1	1
1	0	0	0	0	0	0
1	0	0	0	0	1	0
1	0	0	0	1	0	1
1	0	0	0	1	1	1
1	0	0	1	0	0	0
1	0	0	1	0	1	0
1	0	0	1	1	0	1
1	0	0	1	1	1	1

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

Equation 1: F = (S1)'(S0)'(A) + (S1)'(S0)(B) + (S1)(S0)'(C) + (S1)(S0)(D)

Part IV: Multiplexer Circuit

Experiment

This portion of the experiment involved the creation of a more specific multiplexer by reusing code from the previous part. This circuit involved four input combinations using two different variables, and two outputs, with one being the inverse of the other. This was given in the form of a block diagram, which can be seen in **Figure 8**. The block diagram was used to develop a truth table which would be used to code the logic using VHDL. The actual code was created by using a component statement to reuse the code from the previous part. The code was then compiled and run on an Altera Board. Successful completion of this part would be indicated by the equivalence of the expected and observed output.

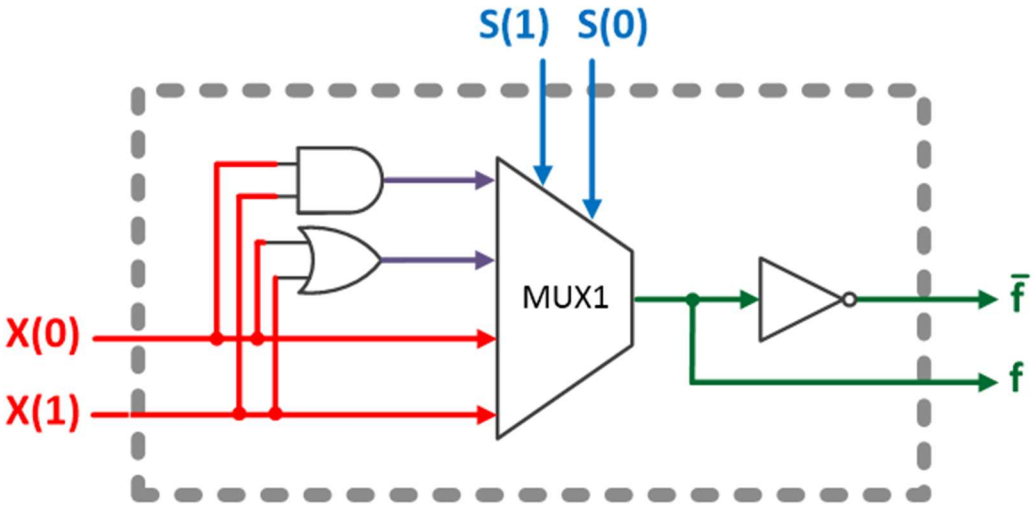


Figure 8: The given block diagram representing the new multiplexer logic to be created

Results

The observed output on the Altera Board matched what was expected according to the truth table. This table can be seen in **Table 5**. The VHDL code can be seen in **Figure 9**, and the component statement is referring to the code shown in the previous part. While the desired output was eventually achieved, there were some issues throughout the process with allowing the correct pins to be set on the Altera Board. This issue was resolved by saving the code and creating a new project to store it.

Table 5: The Truth Table for this Implementation of the Multiplexer

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

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity whateverFINALBOSS is
5
6  port(
7  sFINALBOSS : in std_logic_vector (1 downto 0);
8  x : in std_logic_vector (1 downto 0);
9  f1 : out std_logic;
10 f1Not : out std_logic
11 );
12
13 end whateverFINALBOSS;
14
15 architecture beHAvEyOUrRRr of whateverFINALBOSS is
16
17 component whatever3
18 port(
19 s : in std_logic_vector (1 downto 0);
20 A, B, C, D : in std_logic;
21 f : out std_logic;
22 fNot : out std_logic
23 );
24
25 end component;
26
27 begin
28
29 input1 : whatever3 port map(
30 s => sFINALBOSS,
31 A => x(1) and x(0),
32 B => x(1) or x(0),
33 C => x(0),
34 D => x(1),
35 f => f1,
36 fNot => f1Not
37 );
38
39
40
41 end beHAvEyOUrRRr;
```

Figure 9: The VHDL code for the new multiplexer logic implementation

Conclusion

This lab was very informative about three common computer circuits. Coding them allowed me to better understand how they work, so that they will make more sense when I use them for other projects. Luckily nothing particularly notable went wrong with out experiment, as most issues were resolved relatively quickly. However, my partner and I did miss the part in the beginning where it said we had to program the encoder using conditionals. I added a figure showing how we would have completed that portion of the code had we used conditionals. In future labs, I will be sure to read the instructions more closely to ensure I don't miss any important details.

Questions

- **Post-Lab 6 Questions:**

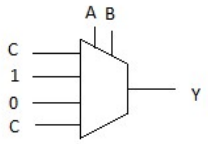
1. How many inputs are in a priority encoder if it has 4 outputs?

   15 Inputs

2. How many inputs are in a multiplexer if it has 4 select bits?

   16 inputs

3. Find the minterm output equation Y for the following multiplexer (equation does not need to be simplified).



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

Y = A'B'C + A'BC' + A'BC + ABC

- **Pre-Lab 7 Questions:**
    1. What is the function of a D Flip-flop?

D Flip-Flops are usually used for their property of causing delays, and they can be used in combination to create delay circuits

Name: Kollin Labowski

Partner: Mohammad Alenezi

Computer Engineering 272-002

Lab 7: Sequential Logic

Date Completed: 3/3/2020

Introduction:

This lab served as an introduction to the circuit building blocks known as flip-flops. In order to learn the functionality and structure of these sequential circuits, D Flip-flops were used to implement a counter based on the internal clock of the Altera Board. This was accomplished by creating portions of necessary code in each part and combining them together at the end to complete the design. Previously covered VHDL topics such as conditions and component statements were used throughout to complete the lab.

Part I: D Flip-Flop

Experiment

This portion of the lab required the simulation of a D Flip-flop using VHDL code. This was accomplished by consulting a given state table which described how the output should react. This table can be seen in **Table 1**. In order to replicate the behavior of the flip-flop, the inputs CLOCK and D were used along with the outputs Q and Qnot. For this portion of the experiment, the CLOCK input was assigned to a switch so that the output could be more easily studied. The input D was assigned to an adjacent switch. Outputs Q and Qnot were each assigned to adjacent LEDs on the Altera Board. According to the given table and information in the lab handout, a D Flip-flop functions such that the value of Q will be set equal to the value of D. However, as the implemented design was to use a positive edge trigger, it would only ever change the output of Q and Qnot in the case that the clock changes from logic '0' to '1'. Qnot was always set to be the opposite of Q . The actual VHDL design was implemented by using a conditional "if" statement which detected when the CLOCK value switched from '0' to '1'. In this case, the values of Q and Qnot were updated. Successful output on the Altera Board would be observed in the case that the values of Q and Qnot respond to changes in D and CLOCK as described previously.

Table 1: A Given State Table Describing the Behavior of a D Flip-Flop

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

Results

Upon testing the output on the Altera Board, it was determined that the code was written correctly. The value of Q would always be set to the value of D, but only after the CLOCK switch was set from '0' to '1'. Therefore, if the CLOCK switch was not changed at all, the value of Q would never actually update, and this was expected behavior as determined previously. **Figure 1** shows the VHDL code which was written in order to implement the behavior of the D Flip-flop.

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

entity lab7p1 is

port(
CLOCK : in std_logic;
D : in std_logic;
Q : out std_logic;
Qnot : out std_logic
);

end lab7p1;

architecture behavior of lab7p1 is

begin

process(CLOCK)
begin

if(CLOCK'event and CLOCK = '1') then
Q <= D;
Qnot <= not D;

end if;
end process;

end behavior;
```

Figure 1: The VHDL code written for use as a D Flip-flop

Part II: Clock Divider

Experiment

This portion of the experiment involved the observation of code given in the lab handout which acted as a clock divider. Essentially, the purpose of this code was to create a form of the CLOCK of the Altera Board which would switch states at a significantly lower frequency. This was accomplished using two conditional "if" statements and an integer variable. Observation of the

code design showed that the integer was created to act as a counter, as it would increase in size every time the input CLOCK changed from '0' to '1'. When the integer reached a constant number of 50000000, then the output CLOCK would change its state and the integer would reset to 0. The need for using such a high number implies that the CLOCK changes states incredibly fast by default and would likely be unusable for creating a counter at its default speed. The given code was copied into a VHDL file and pinned to the Altera Board's internal clock. Observations were then made about its output to determine whether previously made inferences were accurate.

Results

The LED pinned to the output clock turned on and off on the Altera Board at a consistent frequency. The behavior matched what was predicted previously, although the frequency was still notably quick, further implying the very high default clock speed. **Figure 2** shows the VHDL code which was copied and run on the Altera Board.

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

entity lab7p2 is

PORT(
clock_in : in std_logic;
clock_out: out std_logic
);
end lab7p2;

architecture behavior of lab7p2 is
signal clock_tmp: std_logic;
begin

process(clock_in)
variable x : integer := 0;
begin
if(clock_in'event and clock_in='1') then

   x:=x+1;
   if x=50000000 then
      x:=0;
      clock_tmp<= not clock_tmp;
      clock_out<= clock_tmp;
   end if;
end if;
end process;
end behavior;
```

Figure 2: The given VHDL code which acted as a clock divider

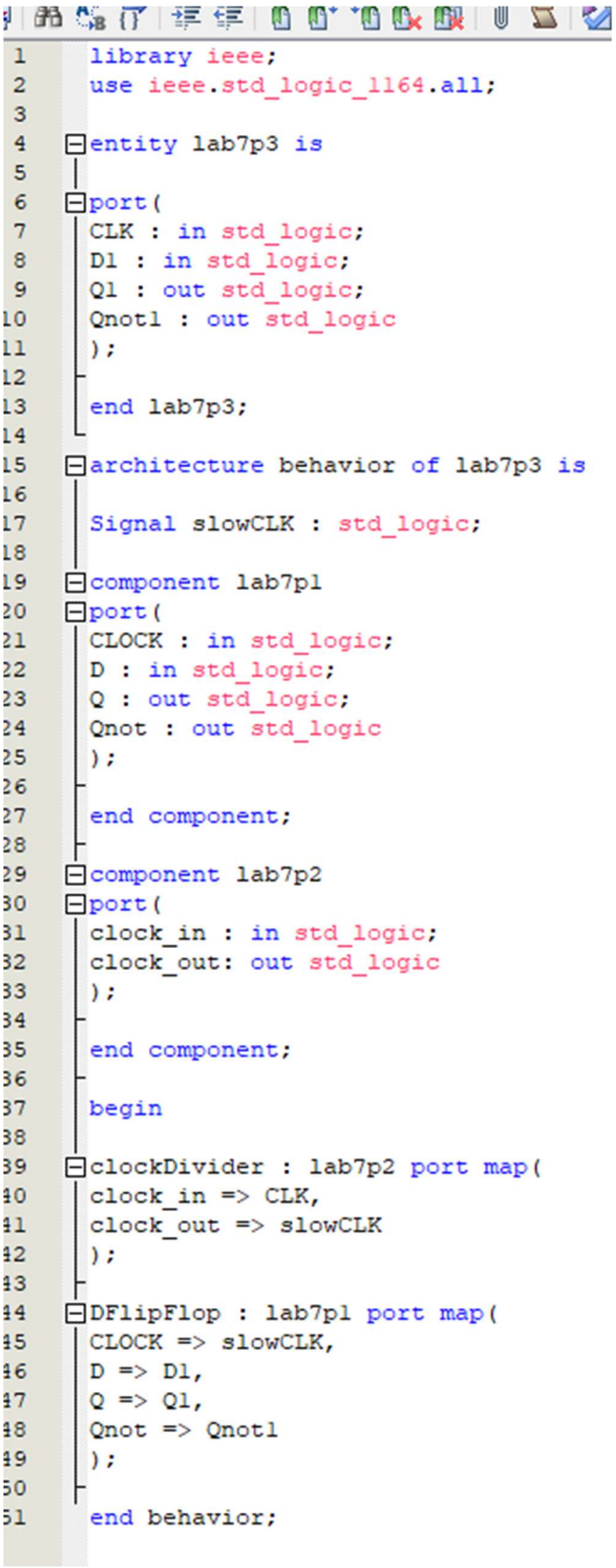
Part III: Building a “Slow” D Flip Flop

Experiment

This portion of the experiment involved the creation of a “Slow” D Flip-flop. In other words, a D Flip-flop was to be created using the slower version of the clock simulated in the second part. This was accomplished by reusing the code from both the first and second part of the experiment in component statements. A diagram was provided in the lab handout to show where inputs and outputs should be connected and used. First, the input clock in the new VHDL file was passed into the clock divider code, and its output clock was stored as a signal. Then, the other inputs D, Q, and Qnot as used in the original D Flip-flop code were passed back into the original code with the slower clock signal. The new inputs and output were then pinned to the Altera Board, and the clock was connected to the Altera’s internal clock. Then, the output was observed to check for accuracy.

Results

The output Q acted in such a way that it was set equal to the input D, but not immediately. The output of Q was only ever updated when the slower clock changed from ‘0’ to ‘1’. In effect, the output Q changed to D with a noticeable delay, and the output Qnot always displayed the opposite of Q. This matched the expected behavior, and as such, it was determined that this portion of the experiment was correctly implemented. **Figure 3** shows the VHDL code used for this section of the lab experiment.



```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity lab7p3 is
5
6   port(
7   CLK : in std_logic;
8   D1 : in std_logic;
9   Q1 : out std_logic;
10  Qnot1 : out std_logic
11  );
12
13  end lab7p3;
14
15  architecture behavior of lab7p3 is
16
17  Signal slowCLK : std_logic;
18
19  component lab7p1
20  port(
21  CLOCK : in std_logic;
22  D : in std_logic;
23  Q : out std_logic;
24  Qnot : out std_logic
25  );
26
27  end component;
28
29  component lab7p2
30  port(
31  clock_in : in std_logic;
32  clock_out: out std_logic
33  );
34
35  end component;
36
37  begin
38
39  clockDivider : lab7p2 port map(
40  clock_in => CLK,
41  clock_out => slowCLK
42  );
43
44  DFlipFlop : lab7p1 port map(
45  CLOCK => slowCLK,
46  D => D1,
47  Q => Q1,
48  Qnot => Qnot1
49  );
50
51  end behavior;
```

Figure 3: The VHDL code used to simulate a “Slow” D Flip-flop

Part IV: Designing a Simple Counter

Experiment

This portion of the experiment involved the use of the code from the previous parts to create a modulo 8 counter. This was accomplished by first filling out an excitation table to set the states of bits D2, D1, and D0 with respect to Q2, Q1, and Q0 both when counting up and when counting down. When counting, 1 was added to the binary value of Q2, Q1, and Q0. Alternatively, 1 was subtracted from the binary value of Q2, Q1, and Q0 when counting down. The calculated values for D2, D1, and D0 with respect to Q2, Q1, and Q0 were used to find equations to use in the VHDL code. This was accomplished by creating 3 K-Maps, one to find an equation for each input D. The equations were derived from these K-Maps by making groups within the maps that would simplify the sum-of-product equation. Then a new VHDL file was created, and a component statement was used to implement the code from the previous part. A signal vector D was used to store the equations calculated from the K-Maps. The inputs included an input x to determine whether the circuit should count up or down, and the clock, which would become slower when passed into the component code. A vector Q was implemented as an inout, so it had properties of both an input and an output, increasing convenience and ease of use. Three instances of the previously made code were created, one for each bit D and value of Q. In the end, the input x was pinned to a switch and the values of vector Q were set to LEDs. Correct output would show the Q vector counting up or down in binary, looping around after reaching either 0 or 7, depending on the direction.

Results

When the program was run on the Altera Board, the output matched what was expected as detailed previously. The created excitation table is shown in **Table 2**. The K-Maps used for simplification are shown in **Table 3**, **Table 4**, and **Table 5**. Their respective derived equations are shown in **Equation 1**, **Equation 2**, and **Equation 3**. The complete VHDL code can be seen in **Figure 4**.

Table 2: The Excitation Table Used for Creating a Modulo 8 Counter

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

Table 3: K-Map for Value D(0)

	x Q2				
		0 0	**0 1**	**1 1**	**1 0**
Q1Q0	**0 0**	1	1	1	1
	0 1				
	1 1				
	1 0	1	1	1	1

Equation 1: D(0) = Q(0)'

Table 4: K-Map for Value D(1)

		x Q2			
		0 0	**0 1**	**1 1**	**1 0**
Q1Q0	**0 0**			1	1
	0 1	1	1		
	1 1			1	1
	1 0	1	1		

Equation 2: D(1) = x xor (Q(1) xor Q(0))

Table 4: K-Map for Value D(2)

		x Q2			
		0 0	**0 1**	**1 1**	**1 0**
Q1Q0	**0 0**		1		1
	0 1		1	1	
	1 1	1		1	
	1 0		1	1	

Equation 3: D(2) = x'Q(2)'Q(1)Q(0) + x'Q(2)Q(1)' + xQ(2)Q(0) + Q(2)Q(1)Q(0)' + xQ(2)'Q(1)'Q(0)'

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

entity lab7p4 is

port(
x : in std_logic;
Clock : in std_logic;
Q : inout std_logic_vector(2 downto 0)
);

end lab7p4;

architecture behavior of lab7p4 is

component lab7p3
port(
CLK : in std_logic;
D1 : in std_logic;
Q1 : out std_logic;
Qnot1 : out std_logic
);

end component;

Signal D : std_logic_vector(2 downto 0);

begin

D(0) <= not Q(0);
D(1) <= x xor (Q(1) xor Q(0));
D(2) <= (not x and not Q(2) and Q(1) and Q(0)) or
(not x and Q(2) and not Q(1)) or
(x and Q(2) and Q(0)) or
(Q(2) and Q(1) and not Q(0)) or
(x and not Q(2) and not Q(1) and not Q(0));

clock1 : lab7p3 port map(
CLK => Clock,
D1 => D(0),
Q1 => Q(0)
);

clock2 : lab7p3 port map(
CLK => Clock,
D1 => D(1),
Q1 => Q(1)
);

clock3 : lab7p3 port map(
CLK => Clock,
D1 => D(2),
Q1 => Q(2)
);
end behavior;
```

Figure 4: VHDL code for the modulo 8 counter

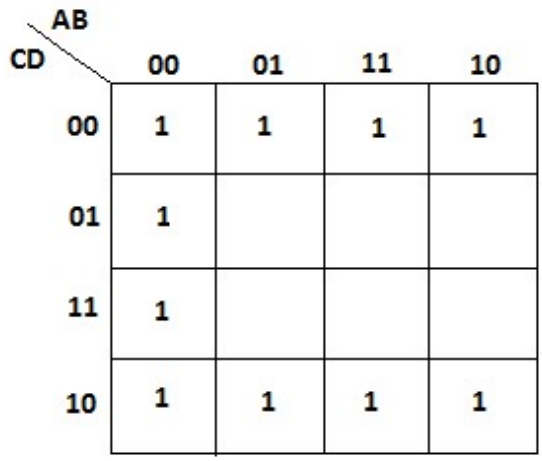
Conclusion

This experiment was the first we have completed in this class which used clocks. I find the clocks to be very interesting, and it appears that they can be applied to a very wide range of applications. The fact that the clock moves so fast by default seems to allow for a lot of customization in terms of the exact speed you would like it to change states. Luckily, my partner and I did not run into any significant difficulties throughout this lab. Our main problems came from failing to read the diagrams as carefully as we should have, but luckily this did not set us back very far at all. Overall, this experiment was very informative, and I look forward to working with similar sequential circuits in the future.

Questions

Post-Lab 7 Questions

1) Given the following K-Map find the **minterm** output equation:



CD \ AB	00	01	11	10
00	1	1	1	1
01	1			
11	1			
10	1	1	1	1

F = D’ + A’B’

Pre-Lab 8 Questions

1) Describe the difference between the Mealy and the Moore model.

   A Mealy model is one in which the output is determined by both the model’s current state and its current inputs. On the other hand, a Moore model is one in which the output is determined solely by the model’s current state.

Name: Kollin Labowski

Partner: Gianfranco Huckaby

Computer Engineering 272-002

Lab 8: Sequential Logic Design

Date Completed: 3/10/2020

Introduction

This lab served as an introduction to sequential logic, and it involved the analysis and creation of state diagrams. It also built upon the topics of the previous lab by allowing the creation of a Mealy State machine which reused the created flip-flop code. The lab also introduced the concepts of Mealy and Moore State Machines and how their outputs are affected by different factors. The final portion of the lab also served as an indication of the problems which can be introduced with the use of "don't care" functions, and how these problems can be avoided.

Part I: Analysis of State Diagrams

Experiment

The first portion of the lab was a series of questions about state diagrams that was intended to show how to read and write the diagrams. The first set of questions were based off a state diagram which can be seen in **Figure 1**. The first question asked whether the diagram was a Mealy or Moore model, and the solution was found by reading the material on the lab handout. The next question provided an initial state of "11" and an input sequence of "010010", and then asked for the corresponding output sequence. The output sequence was found by beginning at the state "11" on the diagram and following the arrow of each input in the sequence from left to right. Each of the respective input's respective outputs were recorded in the same order as the input sequence, and the final state of the input sequence was also noted. The third question of this set provided a blank, negative-edge-triggered timing diagram with a given input combination. The goal of this portion was to fill in the "state" row with the appropriate states, and the "output" row was to be completed based on the given inputs. This question was then followed by a second set of questions revolving around a circuit diagram shown in **Figure 2**. The first question asked was about the type of flip-flop shown in the diagram, and this was determined through inspection of the diagram. The next question asked how many states the circuit had, which was determined using previous knowledge and logic. The output equation for the output z was then determined by inspection to answer the third question of the set. After the two sets of questions, an excitation table was given that was not fully filled out, and the next task was to fill in the information based on the previously generated equation for output z. This excitation table was then observed and used to create a Mealy State Diagram to model the behavior of the circuit. Upon completion of this diagram, a new output sequence was generated from a given input sequence of "01101000" and an initial state of "0". This was accomplished in the same way as the previously generated output sequence.

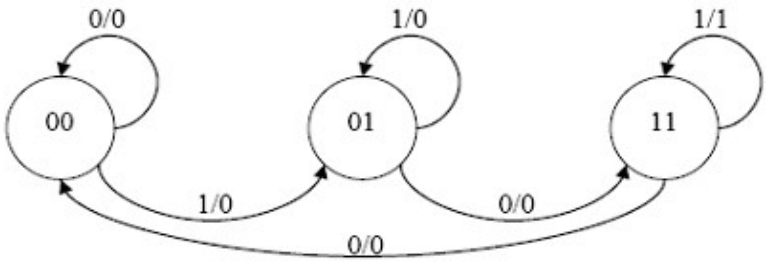


Figure 1: The given state diagram used to answer the first question set

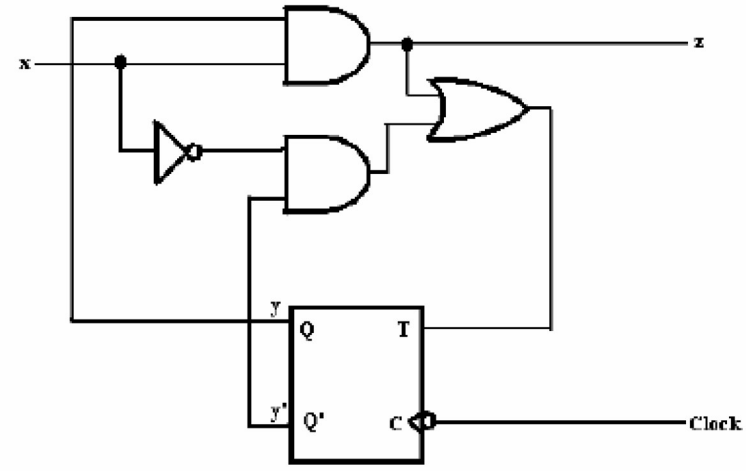


Figure 2: The given sequential circuit used to answer the second set of equations and generate the state diagram

Results

Because the shown state table did have varying behavior based on its input, it was determined to be a Mealy model. The output sequence generated from the input sequence “010010” was “000000”, and the final state was determined to be “11”. The timing diagram filled in to match the state diagram can be seen in **Figure 3**. In the set of questions relating to **Figure 2**, it was determined that the flip-flop used in the diagram was a T Flip-flop. Because the circuit only used a single bit input, the only 2 possible states were “1” and “0”. The logic equation for z in the sequential circuit can be seen in **Equation 1**. The created excitation table can be seen in **Table 1**, and its respective state diagram can be seen in **Figure 4**. The respective output sequence for the input sequence “01101000” in the diagram of **Figure 4** was determined to be “01100000”, with the final state being “1”.

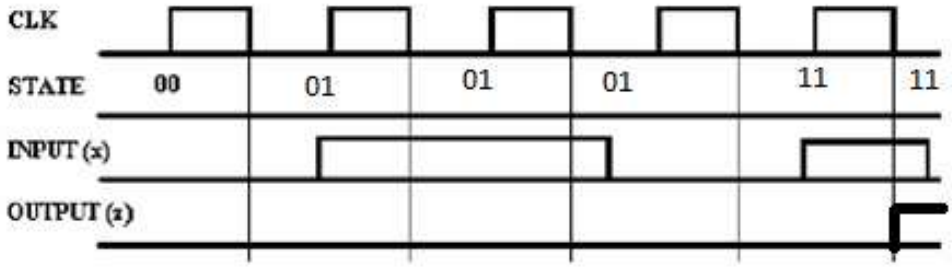


Figure 3: The timing diagram based on **Figure 1**, with "state" and "output" rows filled in

Equation 1: $z = xy$

Table 1: The Excitation Table for the Circuit in **Figure 2**

PS	Present FF inputs		NS	Outputs
$Q_n = y_n$	x	$T = \overline{(X \oplus Q_n)}$	$Q_{n+1} = y_{n+1}$	$z_n = x.Q_n$
0	0	1	1	0
0	1	0	0	0
1	0	0	0	0
1	1	1	1	1

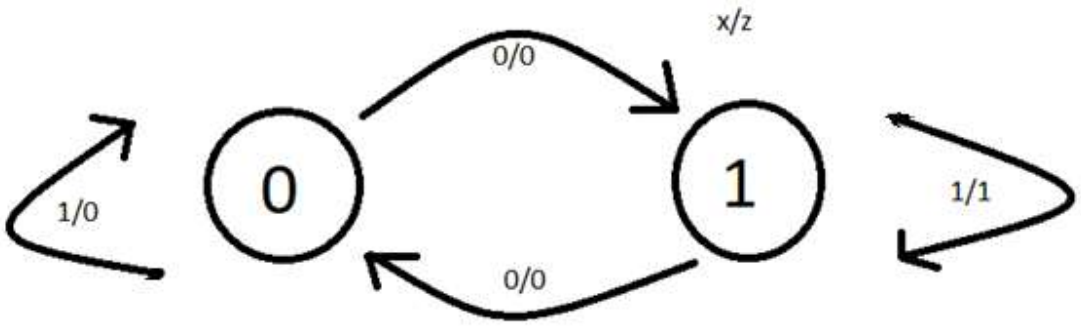


Figure 4: The state diagram created for the circuit in **Figure 2**

Part II

Experiment

This portion of the lab was split into 3 different experiments which built upon each other. The first portion of the experiment involved filling in a binary state table to include logic for a D Flip-flop. This was done by using information already in the table and by analyzing its behavior in comparison to the expected behavior of a D Flip-flop. Once the table was completed, the values of output z were placed into a K-Map, which was used to generate the simplified form of equation z. Notably, the K-Map did involve the use of a "don't care" term, which allowed the equation to be simplified further. The process was then repeated in another table, in which the outputs for values D2 and D1 were determined by comparing the values in the table to the expected behavior of a D Flip-flop. As with z, the values of both D2 and D1 were each placed into a respective K-Map, and the simplified equations for each were determined. As with z, there were multiple "don't care" terms used to simplify the equations. The second experiment built upon the first. For this portion, a VHDL program was created with the inputs X and Clk, the inout variables Y2 and Y1, and the output Z. The slow D Flip-flop code from the previous lab was implemented into this VHDL code with a component statement. Signal variables were used to store the input equations for D2 and D1. Each of the inputs and outputs were passed into the component statement D Flip-flop. Input X was pinned to a switch on the Altera Board, and Y2, Y1, and Z were pinned to LEDs. The output was tested for accuracy, as the expected output would be to have the state either increase or decrease depending on the value of x. In the third and final experiment, it was determined that the values of the "don't care" statements can have a negative effect on the output. In order to fix the issue present in experiment 2, the "don't care" statements in the table were changed to either 1 or 0 in a new table to ensure that the expected behavior was fulfilled. For the new values of D2 and D1, new K-Maps were created, and updated output equations were generated. The VHDL code was then updated with the new equations which lacked the "don't care" functions. The output was then measured again on the Altera Board, and was used to generate another state diagram.

Results

The table generated for the output z can be seen in **Table 2**. The generated output equation for z is in **Equation 2**. **Table 3** shows the table generated for D2 and D1. **Equation 3** and **Equation 4** show the output equations for D2 and D1 respectively. The VHDL code for the second part of the experiment can be seen in **Figure 5**. The observation of this code on the Altera was initially confusing, as it did not change state at all regardless of a change in the input x. The output z changed, but its effect did not appear to be related to the internal clock in any way. It was found that this was caused by the presence of the "don't care" terms, which defaulted to zero and did not allow the state to change as it was supposed to. **Table 4** shows the new table generated that removed the "don't care" terms. **Equation 5** and **Equation 6** show the new equations for D2 and D1 respectively. **Figure 6** shows the newly created VHDL code with the updated terms. When tested this code functioned as expected. The state counted upwards in binary until it reached state "11", at which point it remained at the state. When the input x was toggled, the state counted in the opposite direction until state "00" was reached, at which point the state

remained the same. Output z was measured and used to create the state diagram, which can be seen in **Figure 7**.

Table 2: Binary State Table for D Flip-flop Using Output Z

PS	NS		Outputs	
Q_n=	x=0	x=1	z_n	
$y^-_2y_1$	$y^-_2y_1$	$y^-_2y_1$	x=0	x=1
0 0	0 d	0 0	1	0
0 1	1 0	0 0	0	0
1 0	d d	0 1	d	1
1 1	d d	1 0	0	1

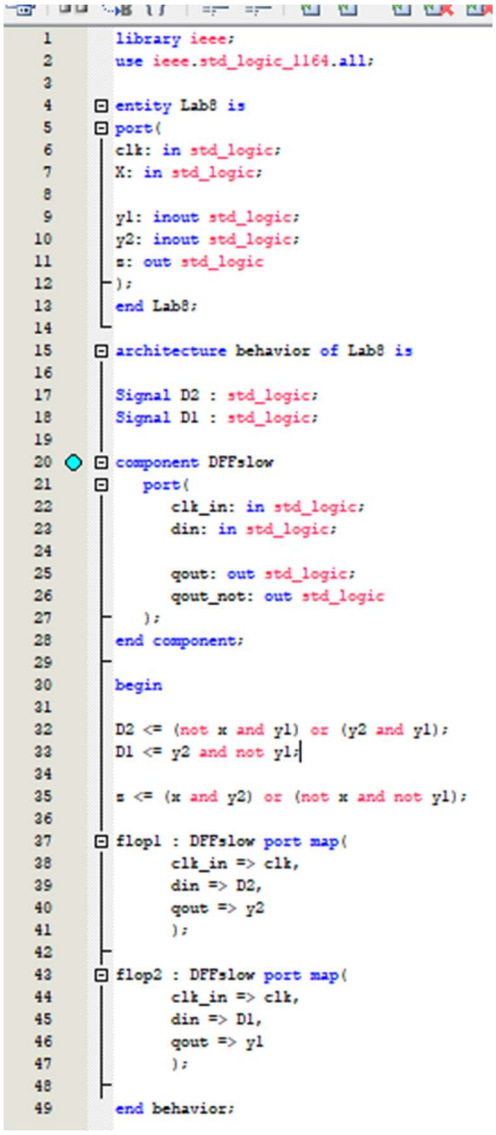
Equation 2: z = x(y2) + x'(y1)'

Table 3: Binary State Table Using D2 and D1

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

Equation 3: D2 = x'(y1) + (y2)(y1)

Equation 4: D1 = (y2)(y1)'



```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity Lab8 is
5   port(
6   clk: in std_logic;
7   X: in std_logic;
8
9   y1: inout std_logic;
10  y2: inout std_logic;
11  z: out std_logic
12  );
13  end Lab8;
14
15  architecture behavior of Lab8 is
16
17  Signal D2 : std_logic;
18  Signal D1 : std_logic;
19
20  component DFFslow
21     port(
22        clk_in: in std_logic;
23        din: in std_logic;
24
25        qout: out std_logic;
26        qout_not: out std_logic
27     );
28  end component;
29
30  begin
31
32  D2 <= (not x and y1) or (y2 and y1);
33  D1 <= y2 and not y1;
34
35  z <= (x and y2) or (not x and not y1);
36
37  flop1 : DFFslow port map(
38        clk_in => clk,
39        din => D2,
40        qout => y2
41        );
42
43  flop2 : DFFslow port map(
44        clk_in => clk,
45        din => D1,
46        qout => y1
47        );
48
49  end behavior;
```

Figure 5: The VHDL code for the second experiment of Part II

Table 4: Binary State Table Using D2 and D1 with "Don't Care" Terms Removed

Current		**Next State**		**D Flip Flop**	
y_2	y_1	**X=0**	**X=1**	**X=0**	**X=1**
0	**0**	0 1	0 0	0 1	0 0
0	**1**	1 0	0 0	1 0	0 0
1	**0**	1 1	0 1	1 1	0 1
1	**1**	1 1	1 0	1 1	1 0
		Y_2,Y_1	Y_2,Y_1	D_2,D_1	D_2,D_1

Equation 5: D2 = x'(y1) + x'(y2) + (y2)(y1)

Equation 6: x'(y1)' + (y2)(y1)' + x'(y2)

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity Lab8 is
5   port(
6   clk: in std_logic;
7   X: in std_logic;
8
9   y1: inout std_logic;
10  y2: inout std_logic;
11  z: out std_logic
12  );
13  end Lab8;
14
15  architecture behavior of Lab8 is
16
17  Signal D2 : std_logic;
18  Signal D1 : std_logic;
19
20  component DFFslow
21     port(
22        clk_in: in std_logic;
23        din: in std_logic;
24
25        qout: out std_logic;
26        qout_not: out std_logic
27     );
28  end component;
29
30  begin
31
32  D2 <= (not x and y1) or (y2 and y1) or (not x and y2);
33  D1 <= (y2 and not y1) or (not x and not y1) or (not x and y2);
34
35  z <= (x and y2) or (not x and not y1);
36
37  flop1 : DFFslow port map(
38        clk_in => clk,
39        din => D2,
40        qout => y2
41        );
42
43  flop2 : DFFslow port map(
44        clk_in => clk,
45        din => D1,
46        qout => y1
47        );
48
49  end behavior;
```

Figure 6: The VHDL code for the final experiment in Part II

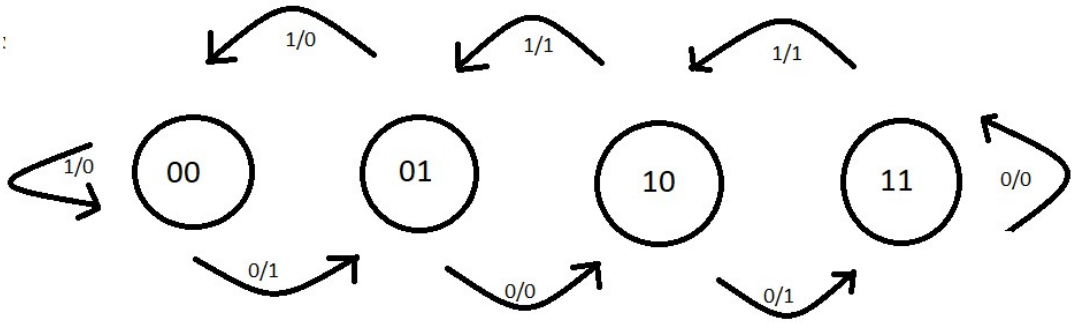


Figure 7: The created state diagram based on the Altera Board output

Conclusion

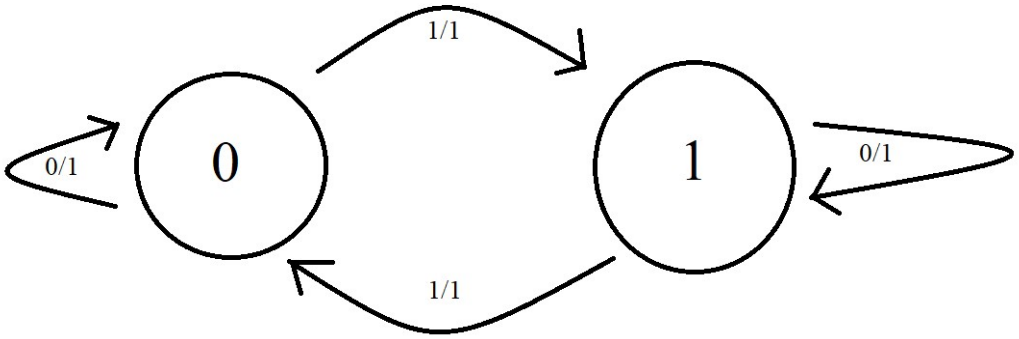
This experiment was more difficult than several of the previous labs, but it was very interesting to learn about state diagrams and sequential logic. Creating the diagrams is an informative way to understand how a circuit works, and it made it less difficult to understand the more complicated nature of sequential logic circuits. My partner and I did not run into any significant difficulties throughout the project, although the output in the second experiment in Part II was initially very confusing. Luckily, the third experiment in Part II helped me to better understand why the circuit was not functioning as intended. Other than initially setting up the K-Maps with the incorrect indexes, there were not any problems that hindered the ability of my partner and I to complete the lab.

Questions

Post-Lab 8 Questions

1) Given the following Transition Table apply the T Flip Flop (just as done in Task 3) to create the State Table. Also, draw the Mealy State Machine Diagram. Excitation table of the T-Flip Flop in the appendix could be useful

PS	PS inputs	NS	Outputs
Y_n	X	Y_{n+1}	Z_n
0	0	0	1
0	1	1	1
1	0	1	1
1	1	0	1



Pre-Lab 9 Questions

1) What is an ALU?

   An ALU is an arithmetic-logic unit, and it is a part of a computer which performs operations of arithmetic, as well as logic operations.

Name: Kollin Labowski

Partner: N/A

Computer Engineering 272-002

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

Date Completed: 4/1/2020

Introduction

This lab served as an introduction to the concepts of enumerated types, case statements, Arithmetic Logic Units (ALU), and memory. Enumerated types and case statements were used to implement a Mealy Machine, similarly to the previous lab. An ALU was also programmed in VHDL using case statements and was based upon a given table of ALU operation codes. Two types of memory, Read-Only Memory (ROM), and Random-Access Memory (RAM), were then created using enumerated types in VHDL. All parts of the project were tested using Waveform Simulator, because Altera Boards were unavailable during the completion of this project.

Part I: Pattern Recognition Using an FSM

Experiment

This portion of the experiment introduced enumerated types and case statements by showing their uses in creating code based on a Mealy Machine. The code to be created with VHDL was based off a given Mealy Machine, seen in **Figure 1**. The behavior of the machine was to detect a logic sequence of "1101" within a larger sequence of logic values which changed randomly with respect to a clock. For the VHDL code, three inputs, labeled clk, reset, and input1, were created along with a single output output1. The clk input represented the clock to base the machine off. The purpose of the reset output was to set the input back to its initial state S0 when it was set equal to 1, as can be seen in **Figure 1**. Input1 was the input logic "1" or "0" that determined the next state to move to according to the Mealy Machine. Output1 represented the corresponding output for each input on the Mealy Machine and was only ever a logic "1" when a sequence of "1101" was detected. An enumerated type was created in the architecture, and it was used to create and track the movement between the four different states of the Mealy Machine. A case statement was then used to determine the function of the code at each of the different states. This case statement was set to run only if the reset input was not "1", and only when the clock changed from a logic low to a logic high. The behavior of each case in the case statement was determined by the Mealy Machine diagram. Because Altera Boards were unavailable for this lab, three input sequences were instead tested by hand and placed into a table. The input sequences were compared with the VHDL code to determine its accuracy.

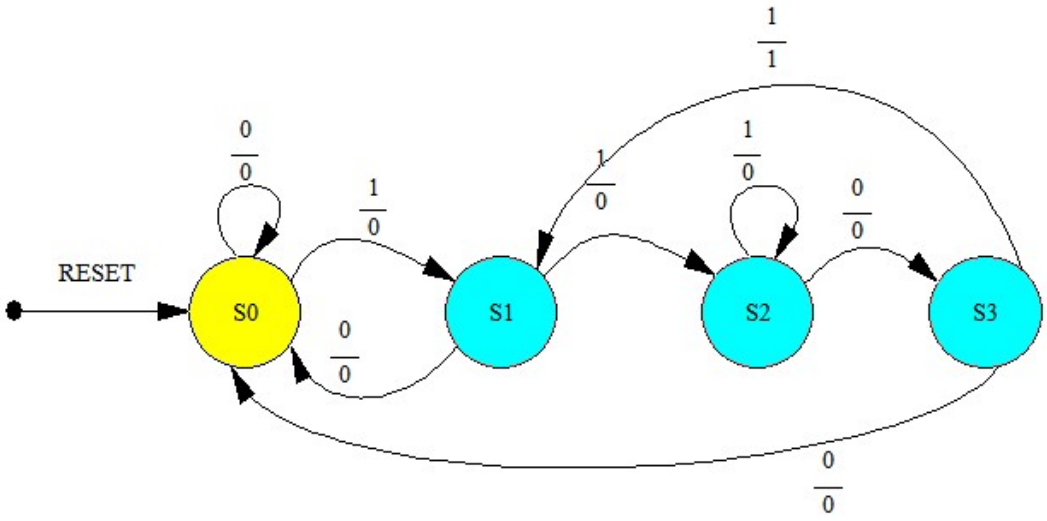


Figure 1: A given Mealy Machine to detect an input sequence of "1101"

Results

The VHDL code, which can be seen in **Figure 2** and **Figure 3**, was determined to be created successfully. Because an Altera Board was unavailable for testing, the output was determined instead by testing each combination in the Waveform Simulator, and the results can be seen in **Table 1**. Although no major problems were encountered during the execution of this part of the lab itself, there was some difficulty in getting the Quartus software to run and function correctly to compile the VHDL code. Eventually it was determined that the Cyclone II device needed to be installed along with the software in order to compile the VHDL code. Once that action had been completed, this portion of the experiment was completed without too much difficulty.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Lab9 is
5  port(
6  clk: in std_logic;
7  reset: in std_logic;
8  input1: in std_logic;
9  output1: out std_logic
10 );
11 end Lab9;
12
13 architecture behavior of Lab9 is
14 type position is (pos0, pos1, pos2, pos3);
15 signal posit: position;
16
17 begin
18 process(clk, reset)
19 begin
20
21 if reset = '1' then
22 posit <= pos0;
23
24 elsif clk' event and clk = '1' then
25
26 case posit is
27 when pos0 =>
28 if input1 = '1' then
29 posit <= pos1;
30 output1 <= '0';
31 else
32 output1 <= '0';
33 end if;
34
```

Figure 2: The first portion of VHDL code for the detection of sequence “1101”

```
32 output1 <= '0';
33 end if;
34
35 when pos1 =>
36 if input1 = '1' then
37 posit <= pos2;
38 output1 <= '0';
39 else
40 posit <= pos0;
41 output1 <= '0';
42 end if;
43
44 when pos2 =>
45 if input1 = '1' then
46 output1 <= '0';
47 else
48 posit <= pos3;
49 output1 <= '0';
50 end if;
51
52 when pos3 =>
53 if input1 = '1' then
54 posit <= pos1;
55 output1 <= '1';
56 else
57 posit <= pos0;
58 output1 <= '0';
59 end if;
60
61 end case;
62 end if;
63 end process;
64
65 end behavior;
```

Figure 3: The second portion of VHDL code for the detection of sequence “1101”

Table 1: Test Input Sequences for the Mealy Machine

Input Sequence	Final State	Output
101101010	S0	000001000
0111001101	S1	0000000001
1011010011	S2	0000010000

Part II: Arithmetic and Logic Units

Experiment

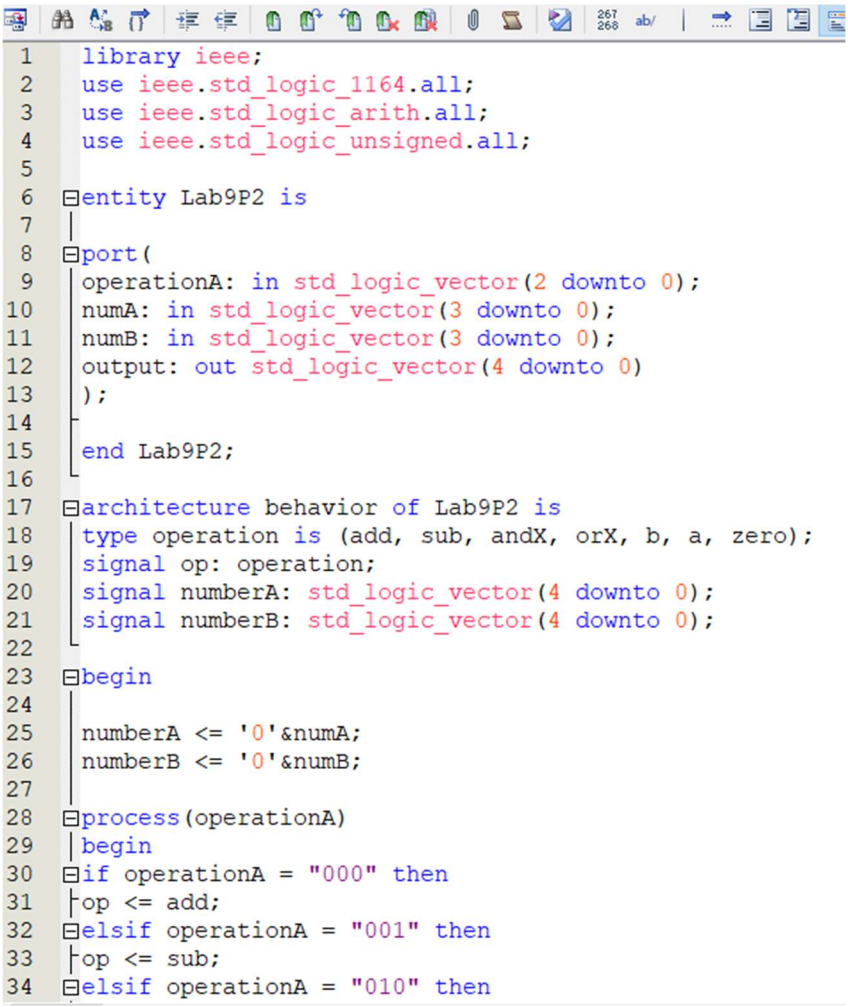
This portion of the experiment involved the coding of an ALU using case statements and enumerated types. The function to be completed by the ALU was determined by a 3-bit sequence, and each of the sequences can be seen in **Table 2**. In order to perform addition and subtraction operations on the bits, the libraries *ieee.std_logic_arith.all* and *ieee.std_logic_unsigned.all* were imported at the top of the VHDL file. The inputs operationA, numA, and numB were used within the code in conjunction with a single output. OperationA was a vector of 3 bits used to determine which operation to perform on the 2 4-bit numbers numA and numB. The output was 5 bits, in order to take the potential of overflow into account when the add operation is performed. Within the architecture of the code, an enumerated type "operation" was created, and represented the type of operation to be performed by the ALU. Two signals numberA and numberB were created which were the same as the numA and numB inputs but with an additional bit of "0" as the most significant bit. A series of conditional statements were then used to determine which operation to perform, or which state to start at. These conditionals were followed by a case statement which performed the operation based on the state. If one of the input operation code was either "110" or "111", then the output was set to "00000". As with the previous portion of the experiment, the ALU was tested manually using the Waveform Simulator, and the results of the test were placed into a table.

Table 2: ALU Operation Codes from Project Document

ALUOp	Operation
000	A+B
001	A-B
010	A and B
011	A or B
100	B
101	A

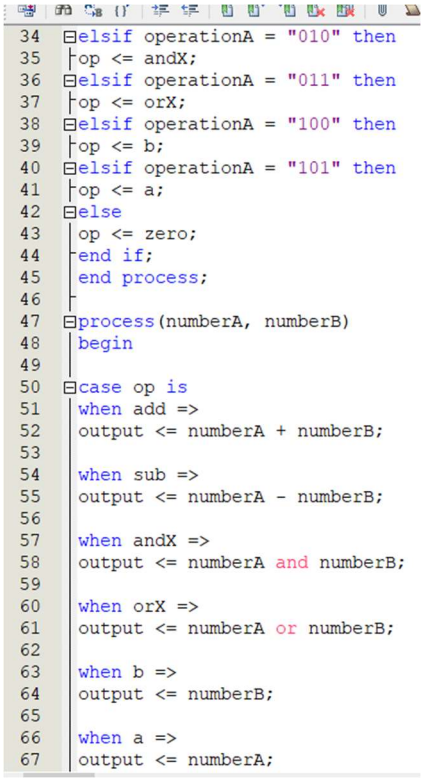
Results

The VHDL code that was tested can be seen in **Figure 4, Figure 5,** and **Figure 6**. The combinations tested in the Waveform Simulator can be seen in **Table 3**. Beyond the table, all operations of the ALU were tested for accuracy, and based on the tests, all operations appeared to function as expected. As such, it was determined that the VHDL code was created successfully. For this portion of the experiment, no significant issues were encountered, although the instructions for the Waveform Simulator from a previous lab were consulted to ensure the process was completed correctly.



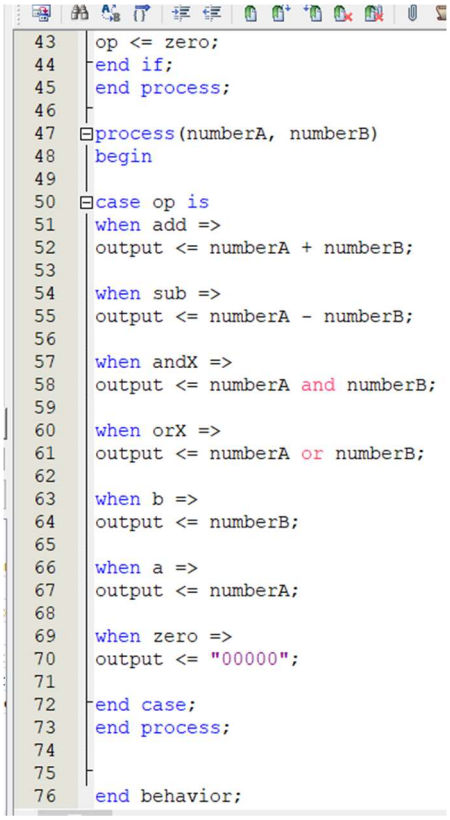
```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.std_logic_arith.all;
 4  use ieee.std_logic_unsigned.all;
 5
 6  entity Lab9P2 is
 7
 8  port(
 9  operationA: in std_logic_vector(2 downto 0);
10  numA: in std_logic_vector(3 downto 0);
11  numB: in std_logic_vector(3 downto 0);
12  output: out std_logic_vector(4 downto 0)
13  );
14
15  end Lab9P2;
16
17  architecture behavior of Lab9P2 is
18  type operation is (add, sub, andX, orX, b, a, zero);
19  signal op: operation;
20  signal numberA: std_logic_vector(4 downto 0);
21  signal numberB: std_logic_vector(4 downto 0);
22
23  begin
24
25  numberA <= '0'&numA;
26  numberB <= '0'&numB;
27
28  process(operationA)
29  begin
30  if operationA = "000" then
31  op <= add;
32  elsif operationA = "001" then
33  op <= sub;
34  elsif operationA = "010" then
```

Figure 4: The first portion of the VHDL code for the ALU



```
34  elsif operationA = "010" then
35  op <= andX;
36  elsif operationA = "011" then
37  op <= orX;
38  elsif operationA = "100" then
39  op <= b;
40  elsif operationA = "101" then
41  op <= a;
42  else
43  op <= zero;
44  end if;
45  end process;
46
47  process(numberA, numberB)
48  begin
49
50  case op is
51  when add =>
52  output <= numberA + numberB;
53
54  when sub =>
55  output <= numberA - numberB;
56
57  when andX =>
58  output <= numberA and numberB;
59
60  when orX =>
61  output <= numberA or numberB;
62
63  when b =>
64  output <= numberB;
65
66  when a =>
67  output <= numberA;
```

Figure 5: The second portion of the VHDL code for the ALU



```
43  op <= zero;
44  end if;
45  end process;
46
47  process(numberA, numberB)
48  begin
49
50  case op is
51  when add =>
52  output <= numberA + numberB;
53
54  when sub =>
55  output <= numberA - numberB;
56
57  when andX =>
58  output <= numberA and numberB;
59
60  when orX =>
61  output <= numberA or numberB;
62
63  when b =>
64  output <= numberB;
65
66  when a =>
67  output <= numberA;
68
69  when zero =>
70  output <= "00000";
71
72  end case;
73  end process;
74
75
76  end behavior;
```

Figure 6: The third portion of the VHDL code for the ALU

Table 3: ALU Code Test Inputs and Outputs

ALUOp Function	Operand A	Operand B	Output
000	1000	1001	10001
001	1110	0100	01010
011	0010	1001	01011

Part III: Introduction to Memory

Experiment

This portion of the experiment involved the simulation of memory using enumerated types. The first experiment in this part involved the creation of ROM with a width of 8 and a depth of 32. In other words, there would be 32 unique memory locations, with 8 bits in each. As with the previous part, the libraries *ieee.std_logic_arith.all* and *ieee.std_logic_unsigned.all* were imported in order to convert binary numbers to integers. The inputs and outputs were determined from a given diagram, seen in **Figure 7**. The inputs used were a clock and an address of the position in memory to read, and the output was the data read from the memory location. A generic class was used to encompass the width, depth, and address of the memory. These values were used to determine how many bits would be used for the read address input and the data output. They were also used to create an enumerated type, which would be used to store the first 8 pieces of data into memory. The change in the memory was based on a positive edge trigger, so whenever the clock changed from "0" to "1" the data at the input position would be output. The output of this code was tested with the Waveform Simulator to determine its accuracy. For the second experiment in this portion, the code from the ROM was altered to act as RAM. The design of this memory type was based on a given diagram, seen in **Figure 8**. This code added 2 inputs; one for data to store in a portion of memory, and one to enable the writing of an input into an output when it was "1". The main alteration to the ROM code was to allow the user to set the output to the input by changing the write_enable variable, and to simply "read" the output otherwise. As with the ROM code, the RAM code was tested with the Waveform Simulator to determine its accuracy.

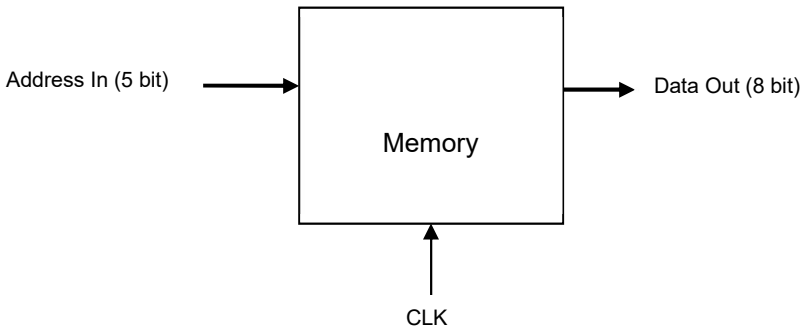


Figure 7: The given diagram of ROM used to create VHDL code

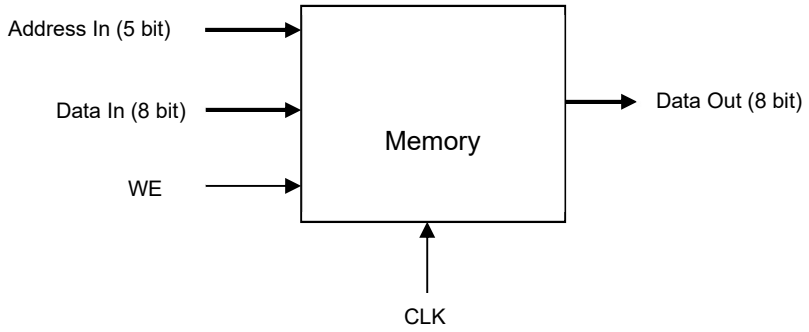


Figure 8: The given diagram of RAM used to alter ROM VHDL code

Results

The VHDL code for the ROM can be seen in **Figure 9**, and the Waveform Simulation of the code can be seen in **Figure 10**. The first 8 positions in the memory were set to the binary numbers 7-0. The simulation shows the first 8 positions in the memory and the output shows the respective values in each. Note that because the design used a positive edge trigger, the changes to the output occur on the rising edge of the clock. Also notable was that the output was set to all 1's in the otherwise, which can be seen when the addresses input continues beyond 8. The VHDL code for the RAM can be seen in **Figure 11**, and its respective Waveform Simulation can be seen in **Figure 12**. As with the previous simulation, this simulation utilized a positive edge trigger. Random positions were used as the addresses to store the data, to show that the positions were not hard coded with the values. It is important to note that the output updates only when the system is in "read" mode, and when in "write" mode it updates the output. As with the previous portion of the lab, no significant difficulties were encountered during this part of the experiment.

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

entity Lab9P3 is

generic (width: integer:= 8; depth: integer:= 32; addr: integer:=5);

port(
clk: in std_logic;
read_addr : in std_logic_vector((addr-1) downto 0);
data_out : out std_logic_vector((width-1) downto 0)
);

end Lab9P3;

architecture behavior of Lab9P3 is
type ram_type is array(0 to (depth-1)) of std_logic_vector((width-1) downto 0);
signal mem : ram_type:=
("00000111", "00000110", "00000101", "00000100",
"00000011", "00000010", "00000001", "00000000",
others => (others=>'1'));

begin
process(clk,read_addr)
begin
if clk' event and clk = '1' then
data_out <= mem(conv_integer(read_addr));
end if;
end process;

end behavior;
```

Figure 9: VHDL code for the simulated ROM

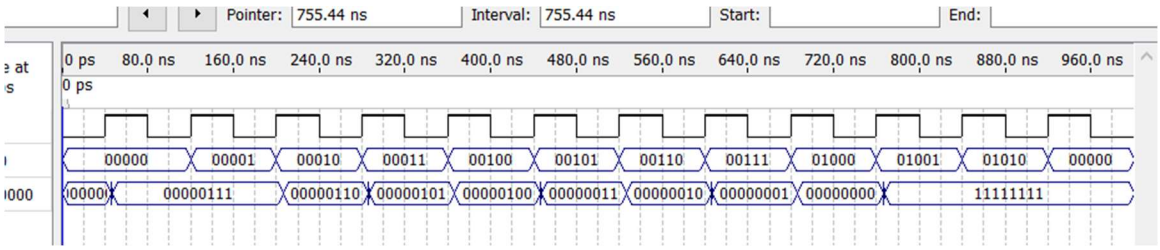


Figure 10: The Waveform Simulation for the ROM

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity Lab9P3 is
7
8  generic (width: integer:= 8; depth: integer:= 32; addr: integer:=5);
9  port(
10 clk: in std_logic;
11 read_addr : in std_logic_vector((addr-1) downto 0);
12 data_out : out std_logic_vector((width-1) downto 0);
13 write_enable : in std_logic;
14 data_in : in std_logic_vector((width-1) downto 0)
15 );
16
17 end Lab9P3;
18
19 architecture behavior of Lab9P3 is
20 type ram_type is array(0 to (depth-1)) of std_logic_vector((width-1) downto 0);
21 signal mem : ram_type:= ("00000111", "00000110", "00000101", "00000100",
22 "00000011", "00000010", "00000001", "00000000", others => (others=>'1'));
23
24 begin
25 process(clk,read_addr, write_enable)
26 begin
27 if clk' event and clk = '1'  and write_enable = '0' then
28 data_out <= mem(conv_integer(read_addr));
29 elsif clk' event and clk = '1' and write_enable = '1' then
30 mem(conv_integer(read_addr)) <= data_in;
31 end if;
32 end process;
33 end behavior;
```

Figure 11: VHDL code for simulated RAM

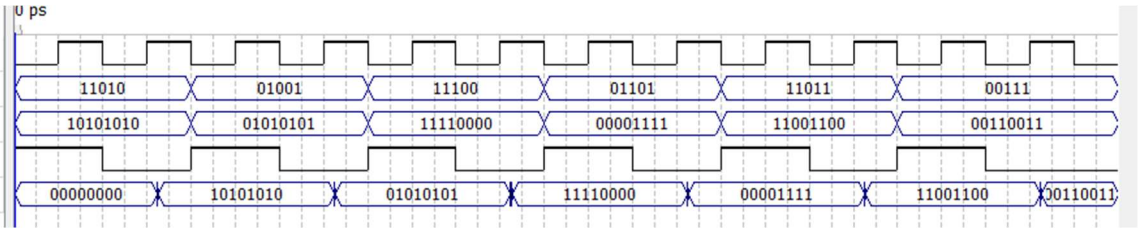


Figure 12: The Waveform Simulation for the RAM

Conclusion

This lab was very interesting because it introduced several new concepts at once. I was already somewhat familiar with memory but learning about ROM and RAM as well as their differences was helpful. This lab did appear to be more difficult than previous labs, but this was likely mostly due to the extraneous circumstances surrounding it. Configuring the Quartus software correctly proved to be more challenging than I anticipated, as I was unaware of the need to download devices. Once I figured it out, however, completing the lab became a much more straightforward process. While it was certainly more difficult due to the inability to ask questions and receive feedback on the spot, it was also a learning experience to figure some of my problems out on my own.

Questions

Pre-Lab 10 Questions

1. What is ladder logic?
   Ladder logic is a language used for programming using ladder diagrams, which are similar to circuit diagrams. It is most frequently used for programming with a programmable logic controller.

Name: Kollin Labowski

Partner: N/A

Computer Engineering 272-002

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

Date Completed: 4/9/2020

Introduction

This lab served as an introduction to programmable logic controllers (PLC) and ladder logic program design. This lab required the use of the CLICK PLC software rather than the Quartus software used for most previous labs. This software was used to first create a sample program as described in the handout. It was then used to create a circuit which would turn an LED on an off in equal time intervals. Unfortunately, this lab was limited by the unavailability of a PLC, and so none of the code was properly tested. Nonetheless, the lab served as an introduction to a new and useful computer component, and a different type of programming method.

Part I: Getting Started with CLICK

Experiment

This first part of the experiment served as a tutorial to the CLICK software and the concept of ladder logic programming. By following instructions of the lab handout, a new CLICK PLC program was created. Three normally open (NO) switches were created by dragging contact objects from the Instruction List located on the right side of the CLICK interface. The first two were placed in parallel on the first rung using the Line Creation tool, and the third was placed separately on the second rung. A timer from the Instruction List was placed on the right side of the first rung and set to the timer number T1. The third switch on the second rung was also labeled T1, connecting the two components. An output coil from the Instruction List was then added to the right side of the second rung. Finally, an end instruction was added to the third rung of the program, indicating where the program stopped. At this point the original intention was that the program be exported to a PLC and tested. However, due to the extraordinary circumstances surrounding the lab, there was no PLC available, and so the code could not be properly tested. Even without testing the code, a figure containing the program created in the tutorial was provided with certain components labeled. This figure, seen in **Figure 1**, was to have each portion of the program identified and its function explained. Therefore, the function of the program was determined by observation rather than through actual testing.

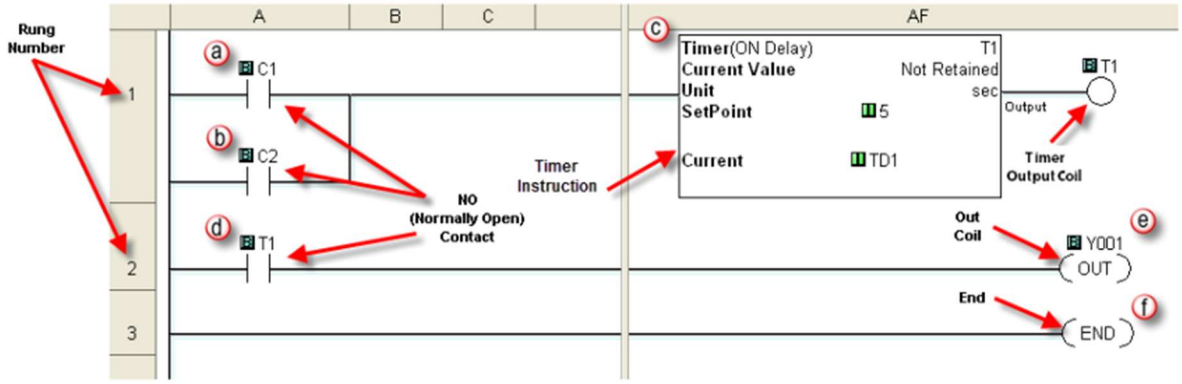


Figure 1: A given figure showing the tutorial program with each component labeled

Results

While the actual program created based on the tutorial instructions, which can be seen in **Figure 2**, was not tested, it was assumed to be correct as it matched the design of the circuit shown in **Figure 1**. The components of **Figure 1** were also labeled based on information provided from the introduction of the lab handout. Components *a* and *b* were both normally open contacts which were wired in parallel. This meant that they were switches that did not allow power to flow through them by default. In the context of the program, if either of the switches were toggled to be closed, power would flow from the power rail on the left to the timer created on the right side of the first rung. This timer was labeled as *c*, and in the case of the tutorial was set to change after 5 seconds. The timer *c* was labeled as T1. Component *d* was also set to T1, connecting its state to the timer *c*. The state of switch *d* was therefore toggled after the timer *c* activated it. The switch *d* then would allow the output coil *e* to connect to the power line and turn on the LED which would have been connected to a PLC. Component *f* was the end instruction, which was necessary to mark the ending point of the program that would have been necessary during compilation. The function of the overall program was to turn on when the switch C1 or C2 was toggled, but only after the timer T1 activated the switch T1. While the function of the program was difficult to understand at first without being able to test the program, it became more straightforward after reading the introduction material more thoroughly.

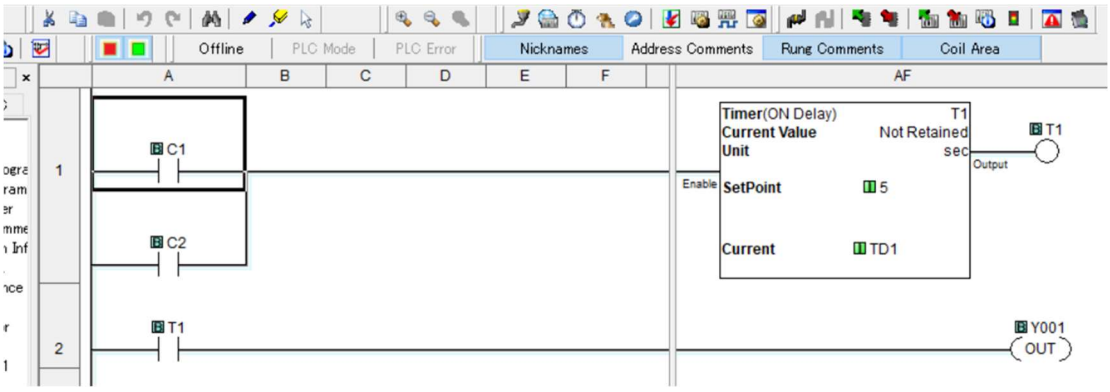


Figure 2: The created ladder logic program based on the tutorial in the lab handout

Part II: CLICK PLC Timer

Experiment

This portion of the experiment involved the creation of a simple ladder logic program which would turn an LED connected to a PLC on and off in intervals of 3 seconds. According to the specifications in the lab handout, the design was to include a normally closed contact, a normally open contact, 2 timers, and an output coil. Additionally, the program in total was to be no longer than 4 rungs. The program was completed by placing the normally closed timer on the first rung and the normally open timer on the second rung. Each of the first two rungs had a timer placed on the right side, labeled T1 and T2 respectively. The contact on the second rung was labeled T1, to connect with the timer T1 on the first rung. Likewise, the timer on the first rung was labeled T2 to connect to the timer on the second rung. The output coil was added to the second rung of the program and would have controlled the LED on the PLC had one been available. Finally, an end instruction was added to the third rung of the program. Unfortunately, the program could not actually be run, so instead, its successful completion was determined by manual inspection.

Results

The program created based on the handout specifications can be seen in **Figure 3**. The intended function of the program was that power would initially flow from the power rail to the first timer T1. This would activate the switch on the second rung and allow power to flow to the output coil, turning on a connected LED. While powering the output coil, the second timer T2 would be activated and open the first switch for 3 seconds. After those 3 seconds, the first switch would close, the second switch would open, the LED would turn back off, and the cycle would repeat. While the code could not be run to determine its effectiveness, through inspection of the connections, it appeared as though it would work properly if a PLC were available.

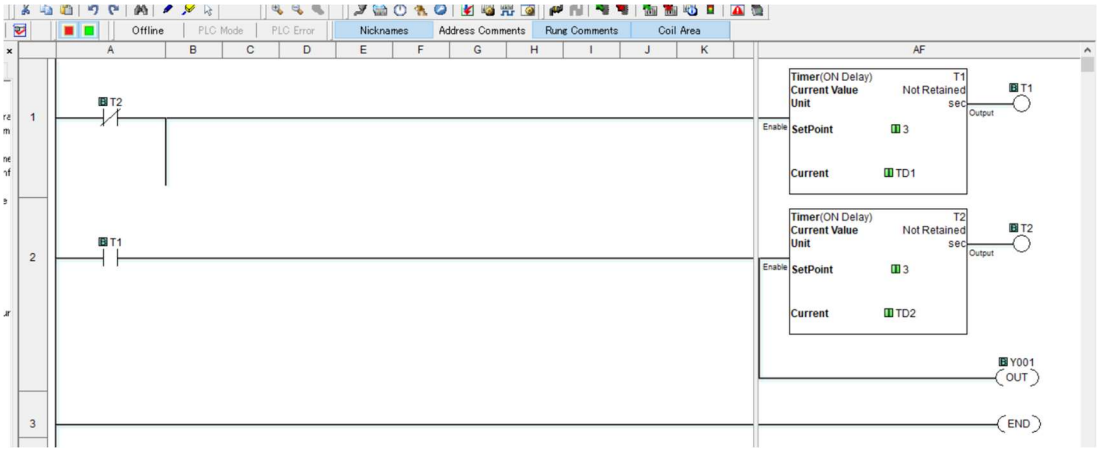


Figure 3: The created circuit to toggle the state of an LED every 3 seconds

Conclusion

This lab was very different from the previous labs, as it used a different type of programming style, as well as a different type of software. It was very interesting to learn about PLCs and the advantages they provide, even if I was unable to test with an actual PLC. The lab was made more difficult by the inability to test code, but luckily the logic was not too different from programs we worked on in previous lab assignments. Once I read about the CLICK PLC programming style and completed the tutorial, it was not too difficult to understand the basic function of the programs. It was not particularly difficult to create the circuit for the second part of the lab, however there is no guarantee that the design is completely free of logical errors as I was unable to run it.

Questions

Post-Lab 10 Questions

1. What makes PLC controllers so popular?
   PLC controllers are so popular because they nearly eliminate the need to wire circuits manually and can be reprogrammed in many different ways to emulate all different sorts of logical behaviors.