Digital Lab Notebook

ECE 3450-001 Digital Electronics

Villanova University

Dr. Jupina

Brian Sakhuja

November 3, 2017

Dedicated to Dr. Caverly's moustache.

Contents

List of Figures

List of Tables

1

Practicum 1: Properties of Digital Circuits

1.1 Part 1: Voltage Transfer Characteristics

1.1.1 Overview

In this part of the practicum, we observe properties of a low-power Shottky Transistor-Transistor Logic (LS-TTL) gate and Complementary Metal-Oxide Semiconductor (CMOS) gate. We used data sheets for both the 74LS04 and CMOS 4069 devices along with oscilloscope readings and manual calculations to evaluate the characteristics of these two different technologies. Using these characteristics, we can determine the advantages and disadvantages of each gate technology.

1.1.2 Purpose

The purpose of this part of the practicum is to observe the input/output characteristics of a LS-TTL inverter (74LS04) and a CMOS Hex inverter (CMOS 4069). We can determine the thresholds for a logic "low" and logic "high" state for these devices by examining their output responses on an oscilloscope. Ideally, we would like an inverter to have two discrete states: low or high, where there is an infinite slope between the two states. However, in reality, we will observe that there is some finite slope.

1.1.3 Procedure

A 0 to 5V sawtooth signal at 1kHz was set up using a function generator. The sawtooth waveform was selected, frequency was set to 1kHz, and amplitude was set at $2.5V_{pp}$ to obtain a 5V amplitude. Next, the offset of the waveform was set to $+1.25V$ to raise the signal by 2.5V. The signal at this point is a 0 to 5V sawtooth at oscillating at 1kHz. Using an oscilloscope, we

checked that the signal is correct. Next, we set a power supply to give 5V. Now, we build the circuit shown in Figure 1.5 using the 74LS04 chip. Using red wires for 5V power and black for 0V (ground), we wire up the circuit and turn the power supply and function generator on. Using the oscilloscope, we then capture the output voltage as well as the input sawtooth signal.

Next, we capture the inverter gate's voltage transfer characteristic (VTC). We press the **Horiz** button on the oscilloscope and then press **Time Mode** in the horizontal menu and hit **XY**. Using the vertical position and scaling knobs, we adjust the channel's position and scaling. This display was captured and imported into Microsoft Word. We add two lines with slope = -1 to determine the values of V_{IL}, V_{OH}, V_{IH}, and V_{OL}. To obtain these values, place the lines on the XY plot where its slope is -1. Then, we draw vertical lines down to the x-axis and horizontal lines to the y-axis. Using the x and y division values in the top left of the oscilloscope display, we determine V_{IL}, V_{OH}, V_{IH}, and V_{OL}. V_{IL} is located along the x-axis and is closest to the y-axis. That is, V_{IL} is smaller in value compared to V_{IH}. V_{OL} is located along the y-axis and is smaller in magnitude compared to V_{OH}. V_{IL} corresponds to V_{OH} and V_{IH} corresponds to V_{OL}. These values were recorded and we repeat the procedure for the CMOS 4069 Hex inverter.

1.1.4 Calculations, Data, and Graphs

Noise margin equations:

$$NM_H = V_{OH} - V_{IH} \tag{1.1}$$

$$NM_L = V_{IL} - V_{OL} \tag{1.2}$$

$$NM = min(NM_H, NM_L) \tag{1.3}$$

Table 1.1: 74LS04

Parameter	Min	Typical	Max	Measured	Units
Low level voltage input V_{IL}	-	-	0.8	0.7	V
High level voltage input V_{IH}	2	-	-	1.0	V
Low level voltage output V_{OL}	-	0.35	0.5	0.2	V
High level voltage output V_{OH}	2.7	3.4	-	4.3	V
Hence calculated noise margin	0.3	-	-	0.5	V

Noise margin calculation for 74LS04

$$NM_H = 4.3 - 1 = 3.3V$$

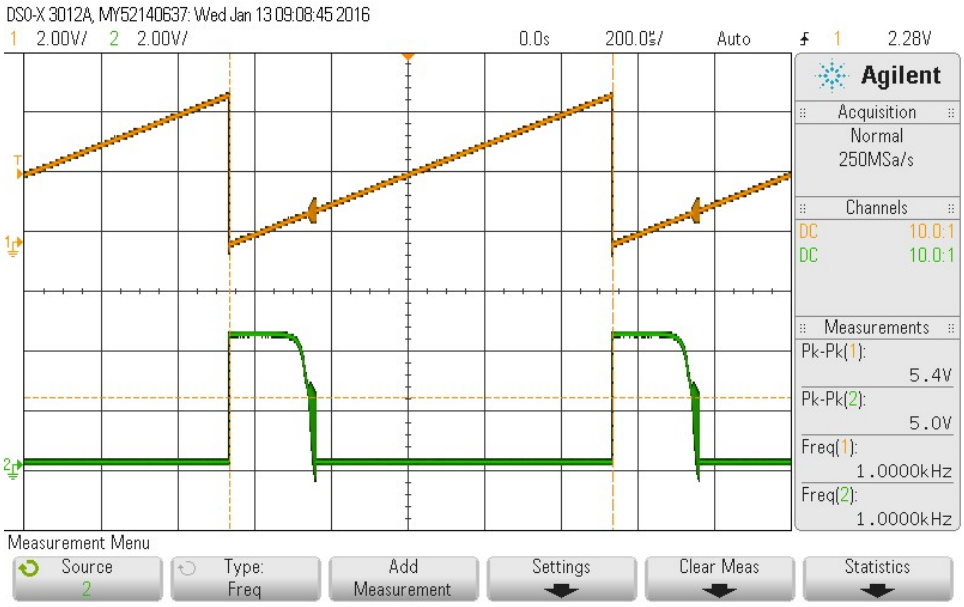

Figure 1.1: TTL Input/Output

$$NM_L = 0.7 - 0.2 = 0.5V$$

$$NM = min(NM_L, NM_H) = 0.5V$$

Table 1.2: CMOS 4069

Parameter	Min	Typical	Max	Measured	Units
Low level voltage input V_{IL}	-	-	2.5	2.6	V
High level voltage input V_{IH}	4	-	-	2.9	V
Low level voltage output V_{OL}	-	-	0.05	0.4	V
High level voltage output V_{OH}	4.95	-	-	4.5	V
Hence calculated noise margin	0.95	-	-	1.6	V

Noise margin calculation for CMOS 4069

$$NM_H = 4.5 - 2.9 = 1.6V$$

$$NM_L = 2.6 - 0.4 = 2.2V$$

$$NM = min(NM_L, NM_H) = 1.6V$$

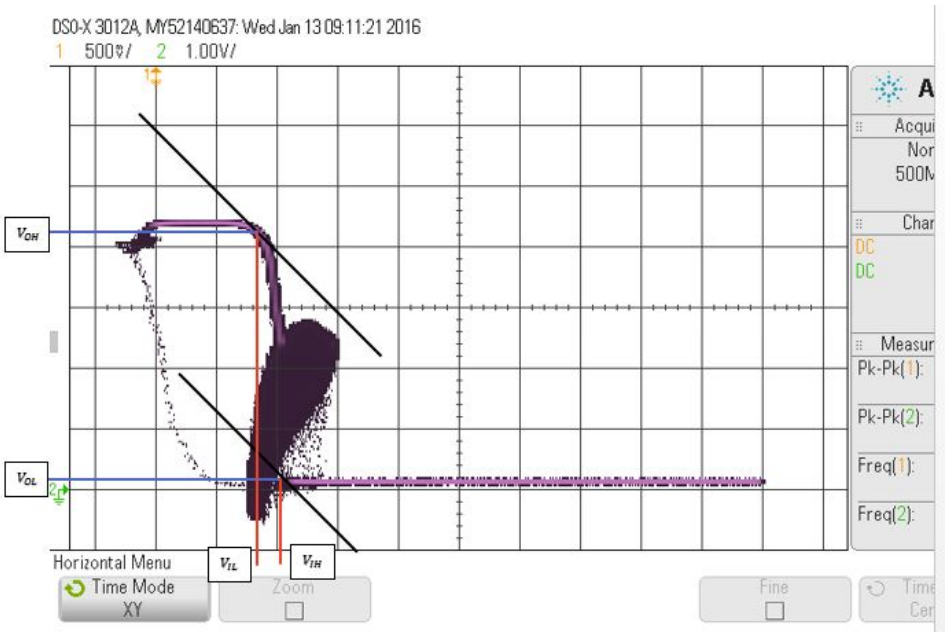

Figure 1.2: TTL XY

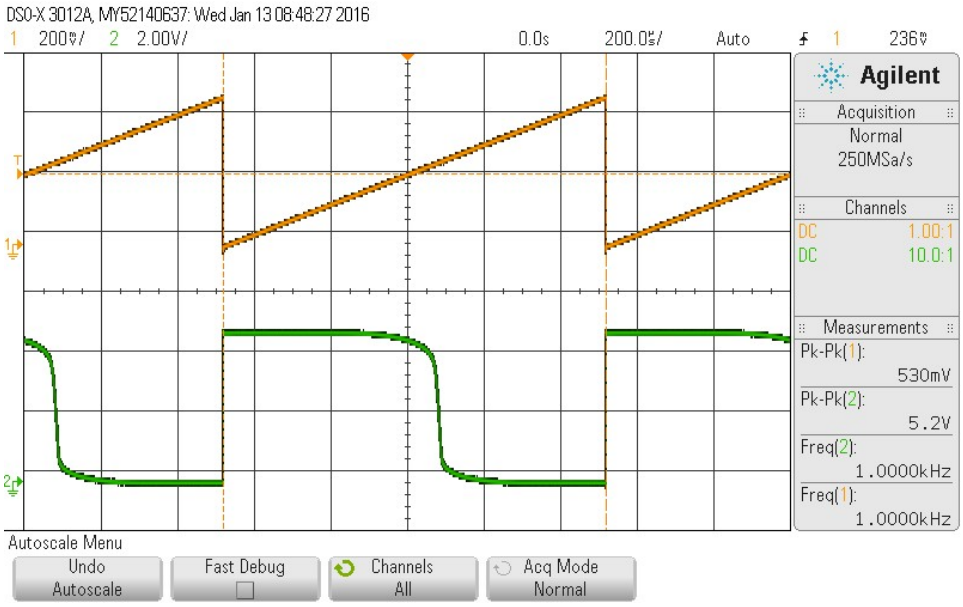

Figure 1.3: CMOS Input/Output

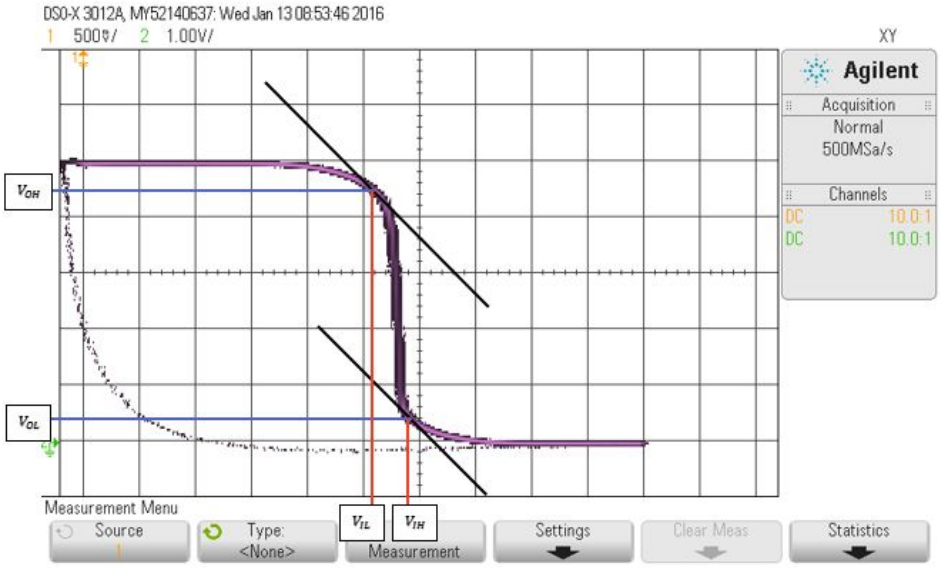

Figure 1.4: CMOS XY

1.1.5 Discussion of Results

For both the 74LS04 and CMOS 4069 technologies we observe the output is what we would expect from an inverter. At low input voltages, we have a logic state 'high' and at high input voltages we observe a 'low' logic state output.

Referring to tables 1.1 and 1.2, we can see that some values come close to the expected, however, some do not. Let us begin by discussing the 74LS04 device in table 1.1. Our measured V_{IL}, V_{IH}, V_{OL}, and noise margin come fairly close to the expected values. Relatively, V_{OH} had the largest discrepancy.

Looking at table 1.2 we now comment on the CMOS 4069 values. V_{IL} and V_{OH} come fairly close to the expected values. However, V_{OL} V_{IH}, and hence the calculated noise margin do not come as close as the other values do.

1.1.6 Conclusions

Errors could have arisen from the fact that the method of obtaining the voltage levels was not very precise. Regardless, the general behavior of these technologies is what we expect. We have observed their logic "low" and "high" output states using an oscilloscope. Hence, we have determined the noise margins. Comparing the 74LS04 and CMOS 4069, we can say that the CMOS performs much better. The LS-TTL device had significantly higher

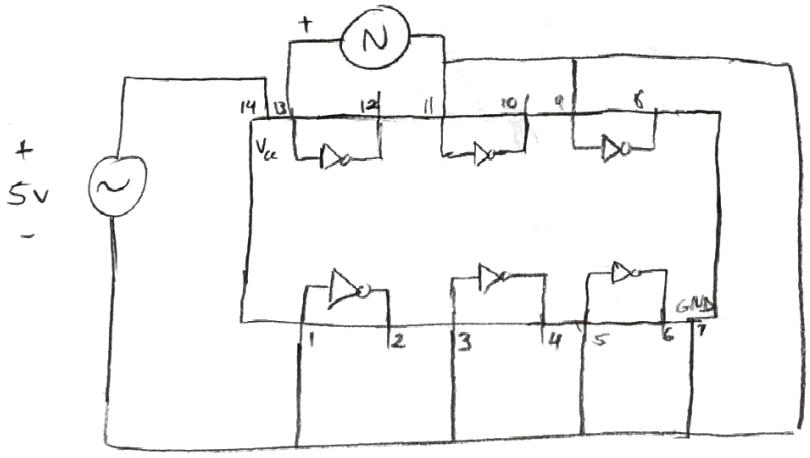

Figure 1.5: Pin Layout

levels of noise, while the CMOS had close to none. Since the amount of noise was smaller, the CMOS device was easier to determine the input/output voltage highs and lows and hence the noise margin.

1.1.7 Prelab

See tables for the values obtained during prelab.

1.2 Part 2: Power Supply Current and Power Dissipation

1.2.1 Overview

To further examine the characteristics of both the Low-power Schottky Transistor-Transistor Logic (74LS04) and Complementary Metal-Oxide Semiconductor (4069 CMOS hex inverter) devices, we examine the power supply current and power dissipation characteristics and compare and contrast their advantages and drawbacks.

1.2.2 Purpose

The purpose of this part of the practicum is to observe the power supply current and power dissipation behavior of both the 74LS04 and CMOS 4069 hex inverter as we increase the frequency. These characteristics are critical

when examining the performance of gate technologies and determining their suited applications.

1.2.3 Procedure

Using the same circuit setup as in Part 1 of this practicum, we wire up a multimeter in series with the 5V power supply such that the power supply's 5V terminal current flows through the meter to V_{CC} of the LS7404 device. The DC/quiescent supply current was measured by tying all inputs to low and recording the supply current. Next, we tie all inputs to high and record the supply current. We then compute the average supply current and then the average supply current per gate. These values are shown in Table 1.3.

The subsequent steps are performed to measure the average supply current when the input is switching as a 50% duty cycle square wave and when the output has a capacitive load. A 47 pF capacitor was added in between the gate output of each inverter and ground. then all inputs were tied together and a 0-5V square wave was added to this common input (the same signal in Part I). We then record the power supply current at frequencies of 1kHz, 10kHz, 100kHz, and 1MHz. The average current and power dissipation per gate were computed and recorded in Table 1.4.

We repeat the procedure with the CMOS 4096 hex inverter.

Then we plot the power dissipated per gate versus frequency for each chip. This is shown in Figure 1.6.

1.2.4 Calculations, Data, and Graphs

Equations

$$P_{AVG} = (\frac{I_{CC}}{6})(V_{CC}) \tag{1.4}$$

$$P_{AVG} = (\frac{I_{DD}}{6})(V_{DD}) \tag{1.5}$$

Table 1.3: 74LS04

Parameter	Typical	Max	Measured	Units
Quiescent IC Supply Current, outputs high, I_{CCH}	1.2	2.4	1.240	mA
Quiescent IC Supply Current, outputs low, I_{CCL}	3.6	6.6	3.865	mA
Average quiescent IC supply current, I_{CC}	2.4	4.5	2.553	mA
Average quiescent supply current per gate, $I_{CC}/6$	0.4	0.75	0.4254	mA
Average quiescent power dissipation per gate	2.0	3.75	2.127	mW

1.2.5 Discussion of Results

We observe that our prelab calculated values for both devices closely match our values that we calculated from our measurements. Specifically for the

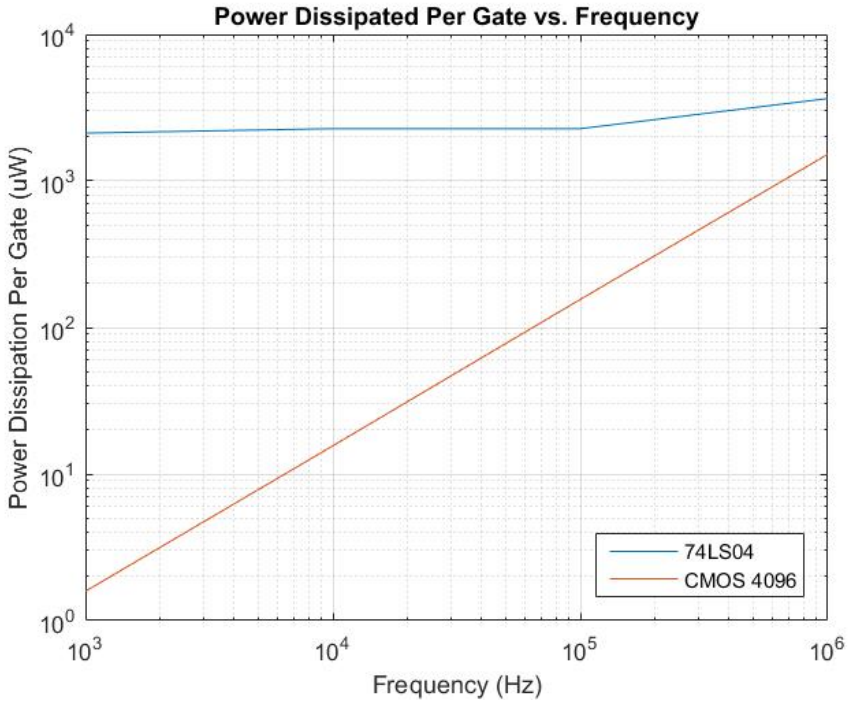

Figure 1.6: Logarithmic Plot of Power Dissipaion Per Gate for Both Devices

Table 1.4: 74LS04: Supply currents when switching at diff. frequencies with a 47pF capacitive load

Parameter	1kHz	10kHz	100kHz	1MHz	Units
Average IC supply current, I_{CC}	2.545	2.563	2.728	4.380	mA
Average supply current per gate, $I_{CC}/6$	0.4242	0.4272	0.4547	0.73	mA
Average power dissipation per gate	2.121	2.274	2.273	3.65	mW
Estimated power dissipation per gate (pre-lab)	2.4	2.45	2.5	3.75	mW

Table 1.5: CMOS 4069 Hex Inverter

Parameter	Typical	Max	Measured	Units
Quiescent IC Supply Current, outputs high, I_{DDH}	-	-	0	mA
Quiescent IC Supply Current, outputs low, I_{DDL}	-	-	0	mA
Average quiescent IC supply current, I_{DD}	0.0005	0.25	0	mA
Average quiescent supply current per gate, $I_{DD}/6$	8.33×10^{-5}	0.0417	0	mA
Average quiescent power dissipation per gate	4.17×10^{-4}	0.2085	0	mW

74LS04 device in Table 1.3, our quiescent IC supply current values were very similar for both high and low outputs (1.2 vs 1.240 mA and 3.6 vs. 3.865 mA). Therefore our average quiescent IC supply current was very similar (2.4 vs. 2.553 mA) as well as our average quiescent supply current per gate and average quiescent power dissipation per gate (0.4 vs 0.4254 mA and 2.0 vs. 2.127 mW).

For the CMOS 4069 hex inverter, we measured no quiescent IC supply current when outputs were either low or high. The average quiescent power dissipation per gate was thus calculated to be 0 as well as shown in Table 1.5.

When we add a capacitive load of 47pF and switch our square wave input at different frequencies, the 74LS04 remained at a relatively constant power dissipation per gate when we increased the frequency. However, the CMOS 4096 hex inverter increased power dissipation per gate when we increased the frequency as shown in Figure 1.6.

1.2.6 Conclusions

Looking at Figure 1.6, we can determine that depending on the application (one frequency versus another), one device might be preferable over another. For example, if we have a low frequency application, the CMOS 4096 would be the choice device as it dissipates less power than the 74LS04 chip. In the end, our prelab values mirror our measured results and we can observe the various drawbacks and advantages of each chip in terms of power dissipation and frequency.

1.2.7 Prelab

See tables for prelab values.

Table 1.6: CMOS 4069: Supply currents when switching at different frequencies with a 47pF capacitive load

Parameter	1kHz	10kHz	100kHz	1MHz	Units
Average IC supply current, I_{CC}	1.9	18.7	186.6	1824	uA
Average supply current per gate, $I_{CC}/6$	0.3167	3.117	31.1	304	uA
Average power dissipation per gate	1.584	15.59	155.5	1520	uW
Estimated power dissipation per gate (pre-lab)	0	25	250	2500	mW

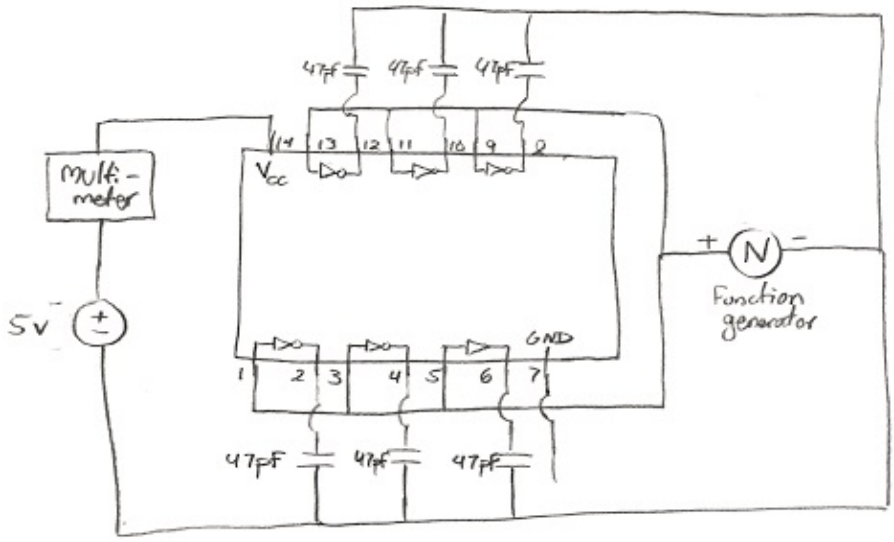

Figure 1.7: Prelab Circuit Diagram

1.3 Part 3: Average Propagation Delay Time and Power-Delay Product

1.3.1 Overview

In the third and final part of the practicum, we observe the average propagation delay time and power-delay product of both devices.

1.3.2 Purpose

Constructing a ring oscillator for the 74LS04 and CMOS 4096 allows us to easily measure the average propagation delay time and period of oscillation. Thus, by obtaining these values, we can calculate the power delay product (PDP), an important property for any gate.

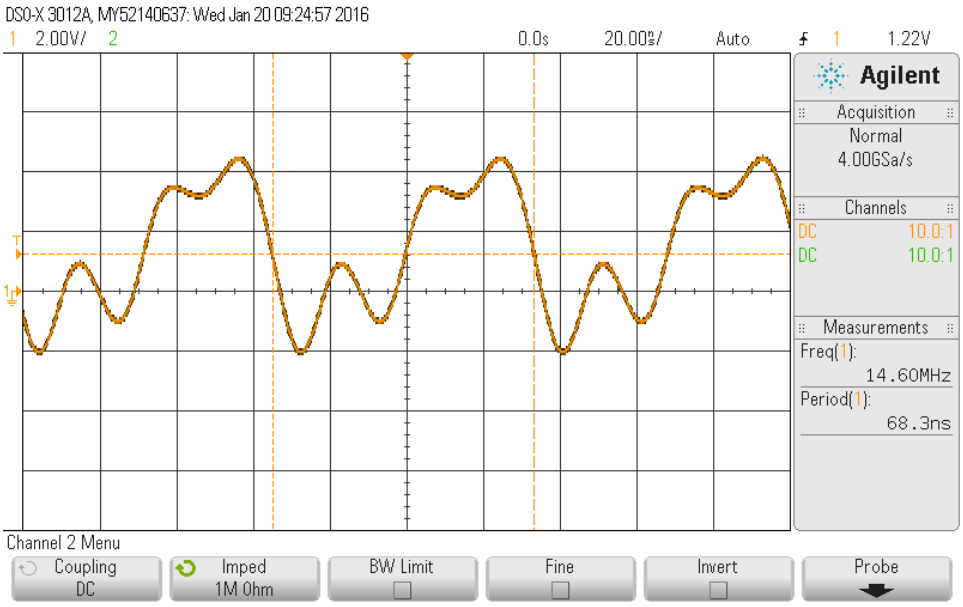

Figure 1.8: TTL Device Ring Oscillator Output

1.3.3 Procedure

We construct the circuit shown in Figure 1.10 using the 74LS04 device. Each capacitor has a value of 47pF. The period of the ring oscillator was measured and recorded using an oscilloscope. A screen capture was taken of the oscilloscope display and is shown in Figure 1.8. Then, Table 1.7 was completed using the measured values. Propagation time delay for a single gate and the time-delay product for the gate needed to be calculated. The procedure was repeated for the CMOS 4096 hex inverter chip.

1.3.4 Calculations, Data, and Graphs

Equations

$$T = 2Nt_p \tag{1.6}$$

1.3.5 Discussion of Results

We observe that the 74LS04 has a smaller period of oscillation of $68.3ns$ versus the CMOS 4096's $383ns$. Thus the period of oscillation per gate is smaller in the 74LS04 ($22.8ns$) than the CMOS 4096 ($127.7ns$). Using equation 1.6, we can determine the expected period of oscillation per gate for each device. For the 74LS04, $T = 2Nt_p = 2(3)(15ns) = 90ns$. Since

Table 1.7: 74LS04

Parameter	Typ/Min	Max	Measured	Units
Period of oscillation	-	-	68.3	ns
Average propagation delay per gate	4	15	22.8	ns
Power dissipation per gate (@1MHz)	3.64	3.64		mW
Power-delay product	14.56	54.6		pW
Energy-delay product	5.824×10^{-8}	81.9×10^{-8}		pJ

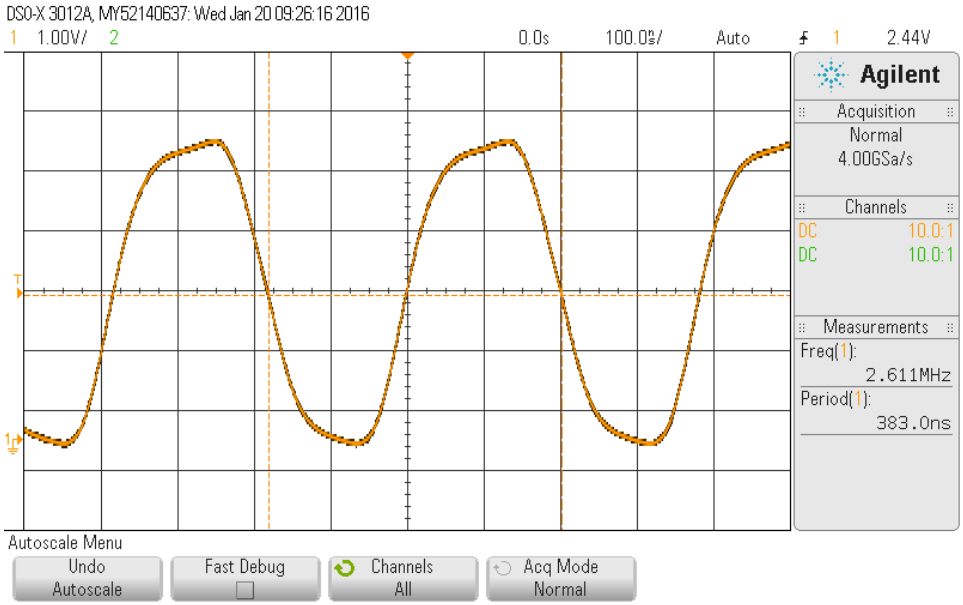

Figure 1.9: CMOS Ring Oscillator Output

we used the maximum t_p value of $15ns$, we obtained the maximum T value. Hence, the period of oscillation is significantly higher than what we measured ($68.3ns$). For the CMOS 4096 device, $T = 2Nt_p = 2(3)(65ns) = 390ns$. Here, we used a typical value of t_p. Thus, our measured value of the period of oscillation is much closer to the calculated one ($390ns$ vs. $383ns$).

1.3.6 Conclusions

Looking at Figures 1.8 and 1.9, we see that the 74LS04 produced a much more rough signal than the CMOS 4096, which produced a smoother sinusoid. The TTL device had much higher frequency components in the output of the ring oscillator. These ripples may indicate that the 74LS04 device is not the best choice in making a ring oscillator since its output is the least sinusoidal of the two devices. Because we did not calculate the PDP or

Table 1.8: CMOS 4096

Parameter	Typ/Min	Max	Measured	Units
Period of oscillation	-	-	383	ns
Average propagation delay per gate	65	125	127.7	ns
Power dissipation per gate (@1MHz)	1.47	1.47		mW
Power-delay product	95.55	183.75		pW
Energy-delay product	621×10^{-8}	2297×10^{-8}		pJ

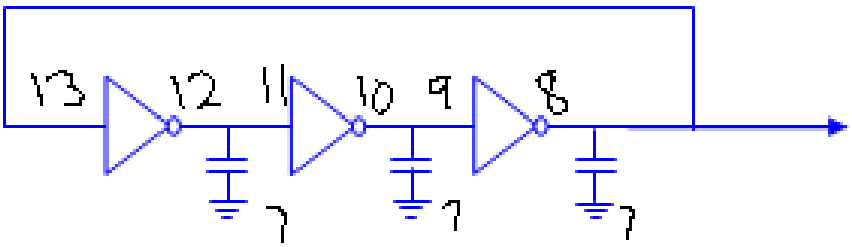

Figure 1.10: Prelab Ring Oscillator Setup with Associated Pin Numbers

EDP using our results, we cannot comment on any discrepancies between the predicted and measured values. However, we can say that the 74LS04 oscillates much faster (nearly 5 times as fast) than the CMOS 4096 device.

1.3.7 Prelab

Refer to Table 1.7 and 1.8 for prelab values (predicted PDP and EDP values).

2

Practicum 2: Layout of a CMOS Circuit

2.1 Overview

In this practicum, we use the Microwind software package to observe how a CMOS logic function could be laid out and implemented in fabrication. CMOS, or complementary metal-oxide semiconductor, technology is ubiquitous in semiconductor technologies. Since the CMOS is **complementary**, the layout of such a logic function contains complementary and symmetrical components of both p-type and n-type metal oxide semi conductor field effect transistors.

2.2 Purpose

This practicum is designed to further understand the layout of a CMOS circuit. In addition, we gain the skill set of going from logic function to layout and layout to logic function. This is an important skill to have so we can understand the behavior of a CMOS device by looking at its layout.

2.3 Procedure

Using the Microwind software package, the logic function $f = \overline{(a+b)\cdot(c+d)}$ was implemented. In Microwind, we entered the Verilog code: $f =\sim ((a|b)\&(c|d))$ in the **Compile** menu. The compile button was pressed and the layout was generated. We capture the layout as shown in Figure 2.2. Next, we used the 2D and 3D tools in the **Simulate** menu to examine the circuit layout in further detail. We capture both NMOS transistors and PMOS transistors in the 2D view (Figures 2.3 and 2.4). Next, we return the **Simulate** menu again and select Process Steps in 3D tool to observe how the logic

device can be fabricated. We capture the 3D cross section in Figure 2.5). Next, we construct the "stick diagram" of the CMOS circuit in MS Paint using Figure 2.2 as a reference. This diagram is shown in Figure 2.6. We then overlay the CMOS gates in Figure 2.7.

2.4 Calculations, Data, and Graphs

See figures.

2.5 Discussion of Results

We have thus shown that the logic function $f = \overline{(a+b)\cdot(c+d)}$ can be built using CMOS technology. In Microwind, we can observe the fabrication steps needed to build such a device. Using MS paint, we construct the stick diagram and locate the transistor locations for this logic function.

2.6 Conclusions

Microwind is a powerful and easy-to-use software that can be used to structure the layout of any logic function in CMOS gate technologies. We successfully implemented the logic function $f = \overline{(a+b)\cdot(c+d)}$ in such a way that we can observe the fabrication steps and construct a stick diagram to make analysis easier.

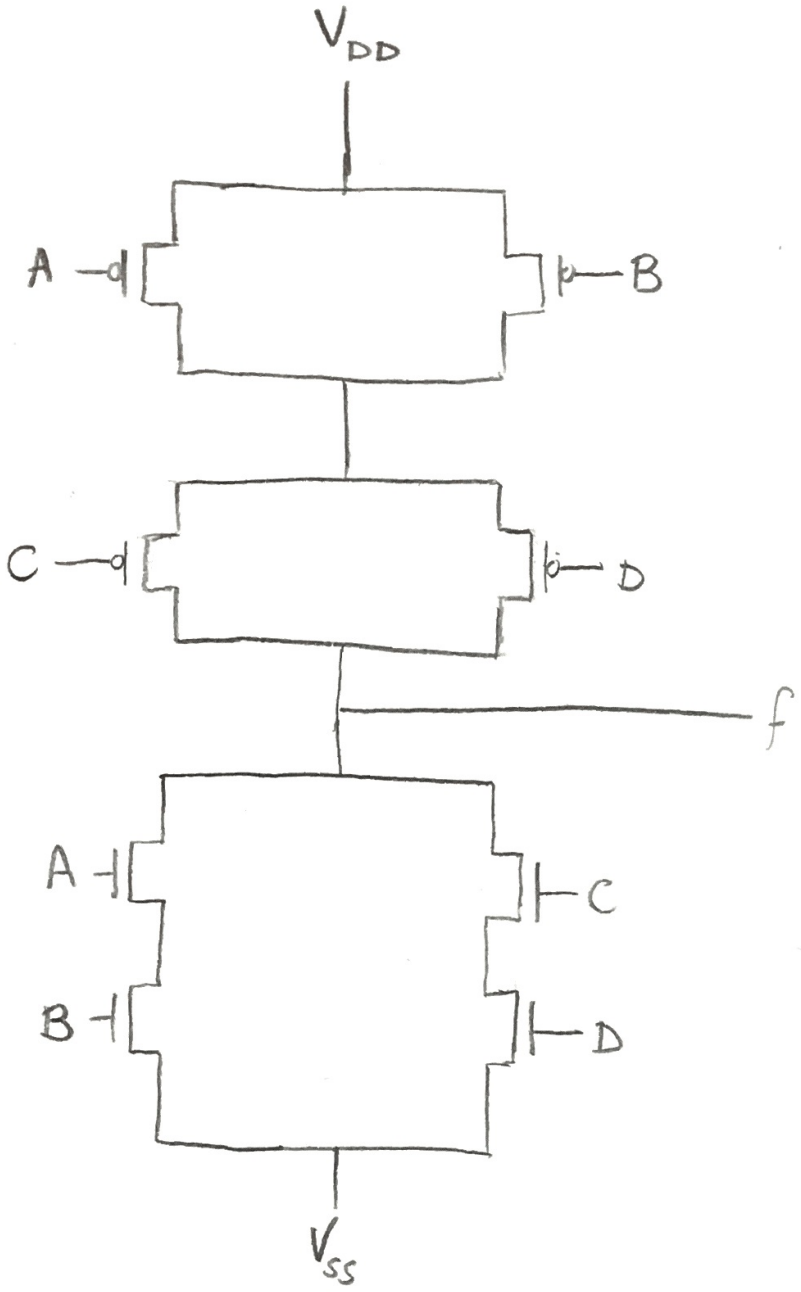

Figure 2.1: CMOS transistor layout of logic function

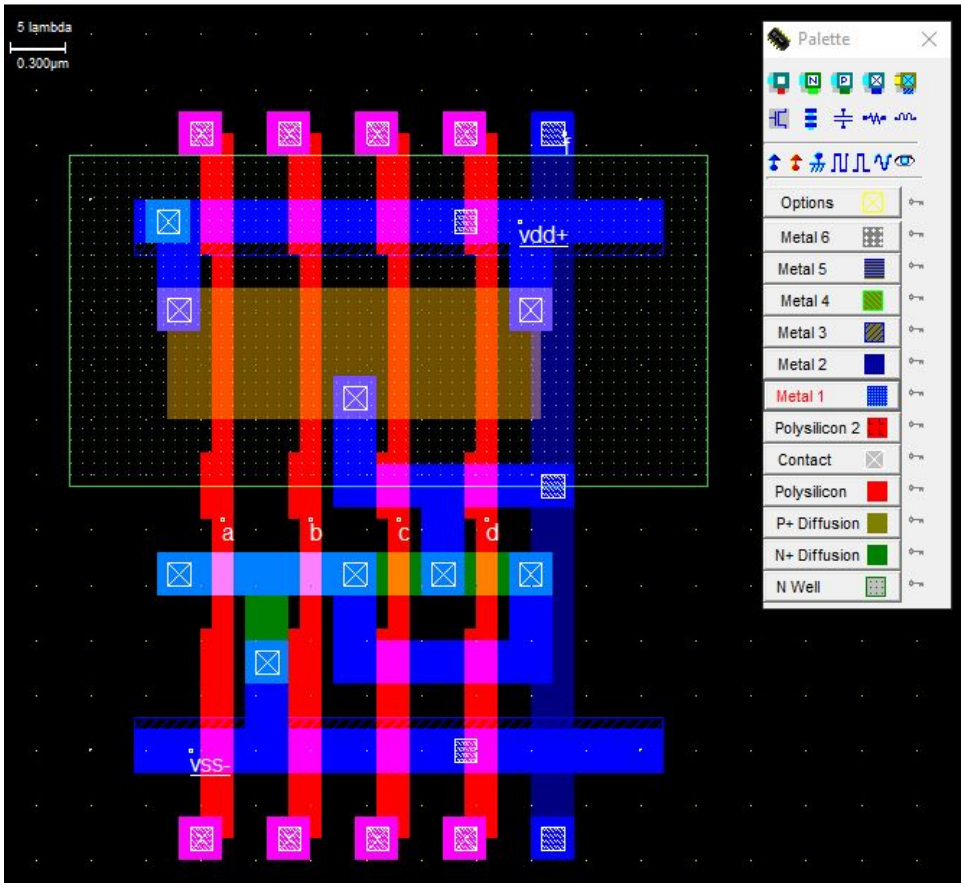

Figure 2.2: Microwind layout of logic function

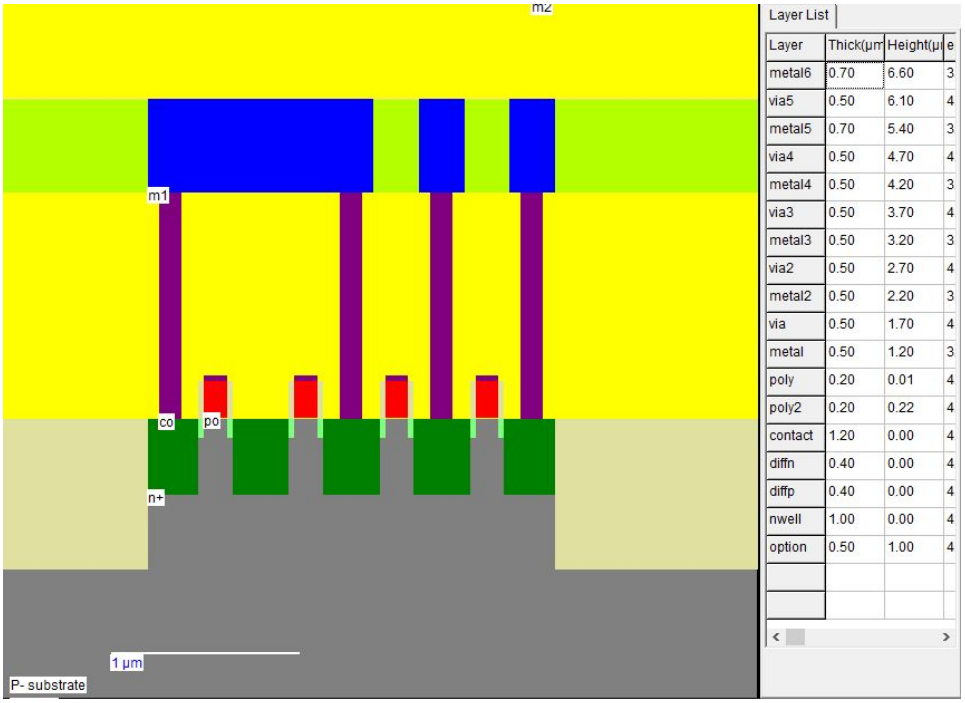

Figure 2.3: 2D cross-sectional view of NMOS transistors

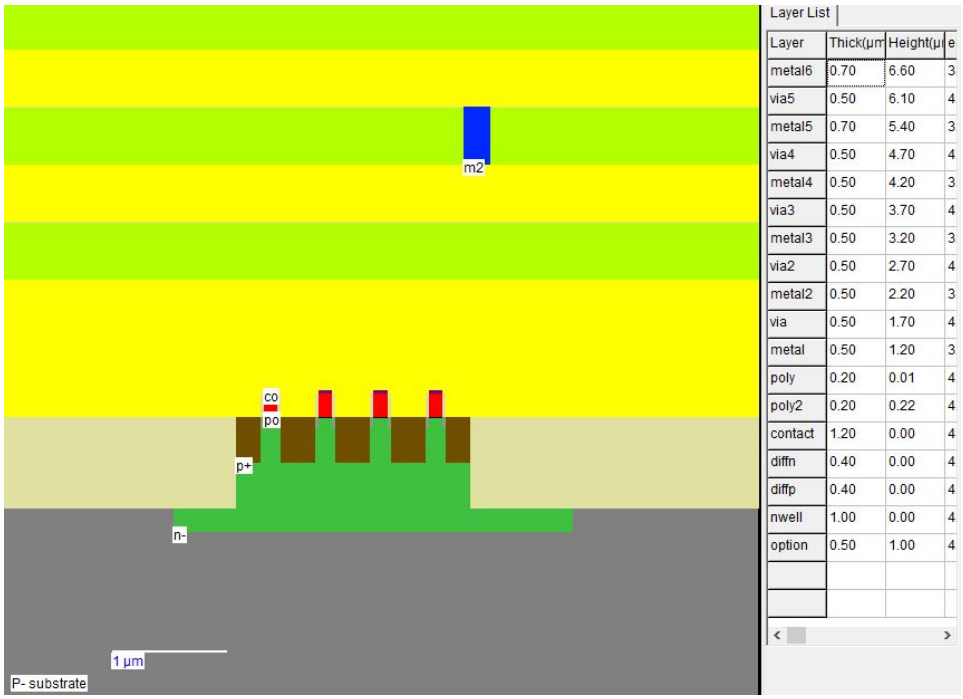

Figure 2.4: 2D cross-sectional view of PMOS transistors

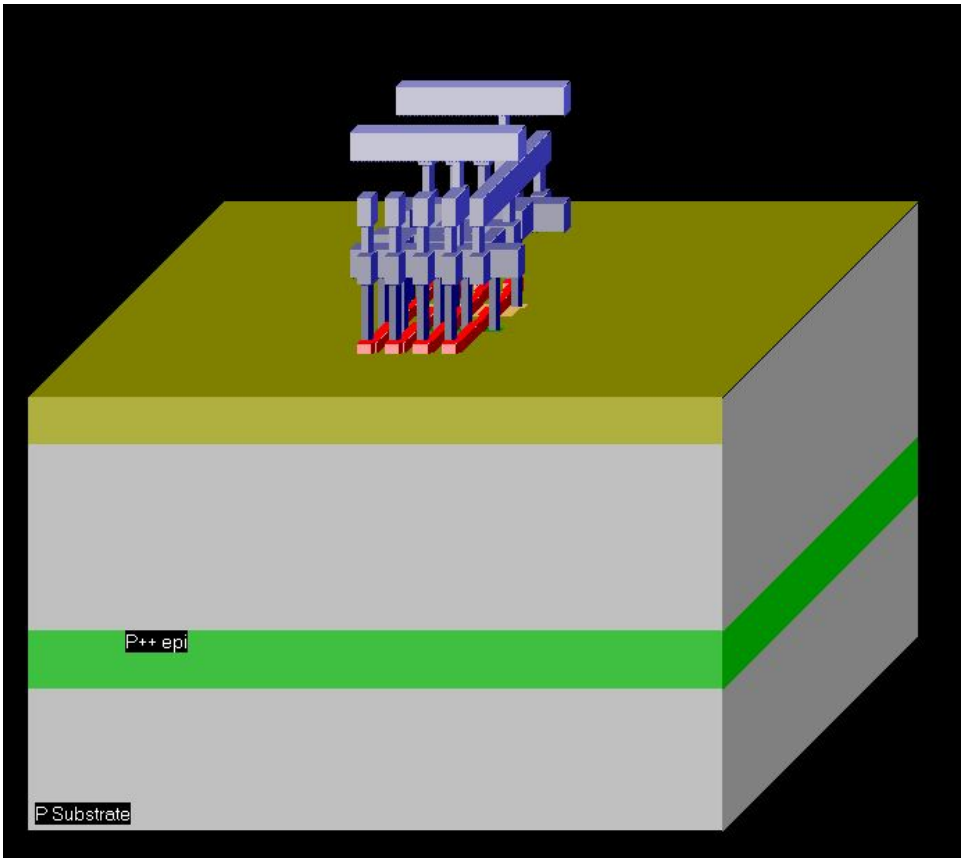

Figure 2.5: 3D view of CMOS circuit

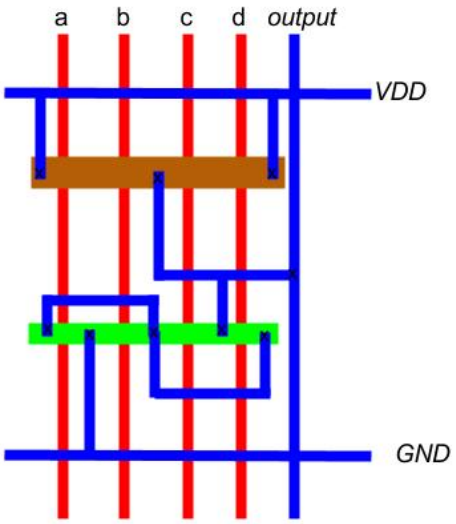

Figure 2.6: Stick diagram of logic function

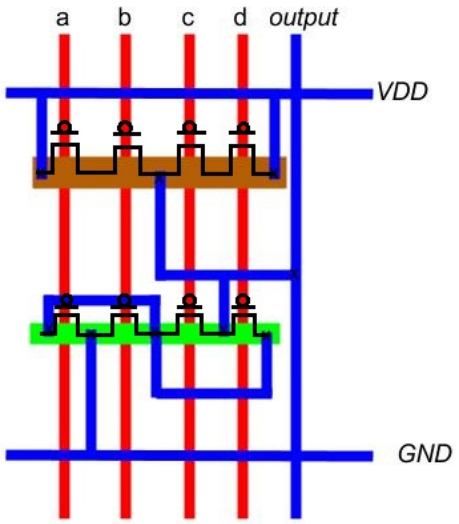

Figure 2.7: Stick diagram of logic function with metal interconnects

3

Practicum 3: DE2 and Quartus Intro

3.1 Overview

Altera Quartus II is a powerful tool to build, model, and test circuits. Schematic capture and VHDL (VHSIC Hardware Description Language) are both important tools required to test characteristics of an ORing gate circuit in this practicum.

3.2 Purpose

The purpose of this practicum is to gain a basic understanding of how to use the Altera Quartus II CAD software package. Using both schematic capture and VHDL, we can observe important characteristics in any circuit we design.

3.3 Procedure

Part 1: Using Schematic Capture to Configure the Altera DE2 Board
A USB cable was connected to the DE2 board and a USB port on a computer with Altera Quartus II CAD tools installed. Then, the 9V DC transformer was connected to the DE2 board and plugged into a 120V AC outlet. The ORing gate circuit was set up for the Cyclone II chip using pates 1-18 of the reference "*Tutorial I: The 15-Minute Design*". Next, the ORing gate circuit was uploaded to the Cyclone II chip on the DE2 board using pages 22-24 of the same reference.

Part 2: Using VHDL to Configure the Altera DE2 Board
The procedures described in Part 1 were repeated using VHDL code using pages 29-33 as a reference.

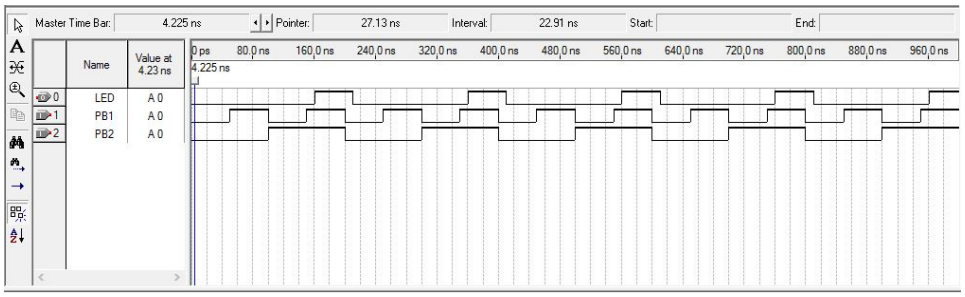

Figure 3.1: Part 1: Simulation Waveform

Part 3: Timing Analysis, Floorplan Editor, Symbol Creation, and Functional Simulation Tools
Using pages 38-41 as a reference, a timing analysis was performed to determine the maximum delay in the ORing gate circuit on the Cyclone II chip. Then, the floorplan editor was used to figure out the location of the ORing gate circuit on the Cyclone II chip. The Chip Planner under the Tools menu was used. Fan-in and fan-out tools were used to see the signal path delays between the LE and I/O circuits on the periphery of the Cyclone II chip. Then, a symbol block was generated from the VHDL code. And a functional simulation of the ORing gate circuit was performed.

Part 4: Additional Exercises
The chip editor was used to move the logic cell used in the ORing gate design to another location inside the Cyclone II chip. It was moved several columns away from the pushbutton and LED pins. The timing analyzer was run and compared to the original results. Finally, the logic circuit designed in the prelab was used to turn on the LED when both pushbuttons were pressed. The circuit was compiled, simulated, downloaded, and tested.

3.4 Calculations, Data, and Graphs

Refer to Figures 3.1-3.7.

3.5 Discussion of Results

Using Altera Quartus II timing analysis the time delay from PB1 to the LED was 10.560ns and 10.417ns from PB2 to the LED. These delays are virtually nonexistent since they are so small, us humans cannot even perceive the delay. Moving the logic cell placement from the default to another location changed the delays to 12.212ns for PB1 to the LED and 12.561ns for PB2 to the LED. Even with the larger delay times, this magnitude of delay is still unperceivable to us.

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.all;
3
4  entity orgate is
5      port
6      (
7          -- Input ports
8          PB1, PB2    : in  STD_LOGIC;
9
10         -- Output ports
11         LED : out STD_LOGIC
12     );
13 end orgate;
14
15 architecture a of orgate is
16 begin
17 LED <= NOT( NOT PB1 OR NOT PB2 );
18 end a;
19
20
```

Figure 3.2: Part 2: VHDL Code

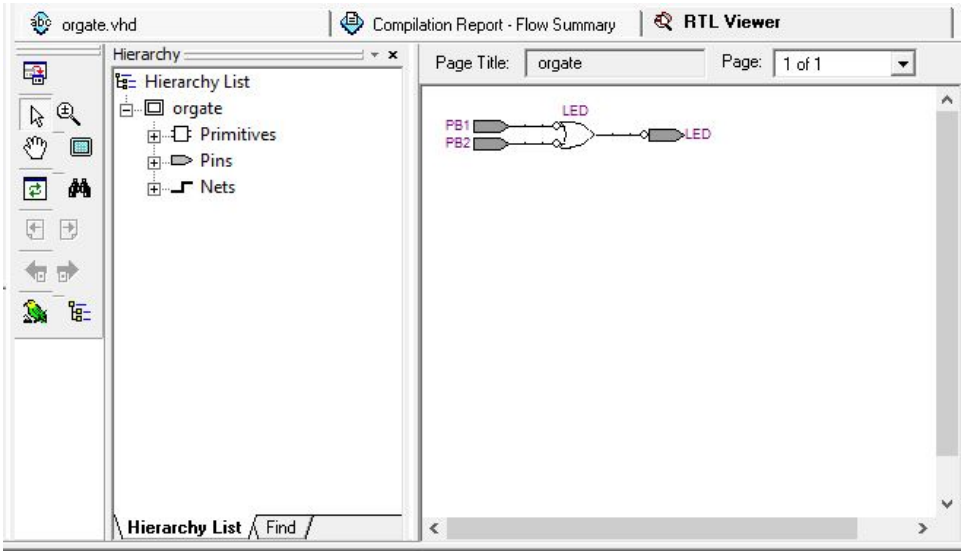

Figure 3.3: Part 2: Diagram From Code

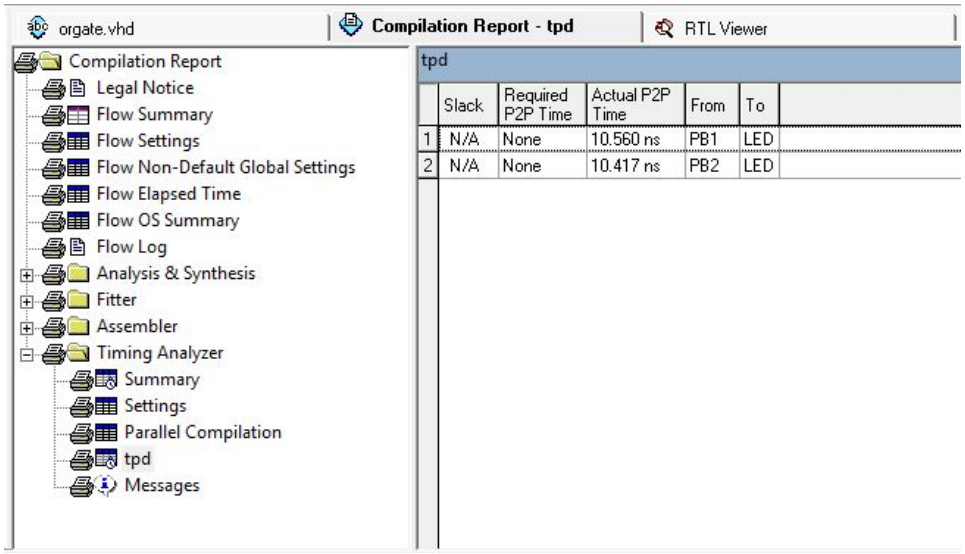

Figure 3.4: Part 3: Timing Analysis

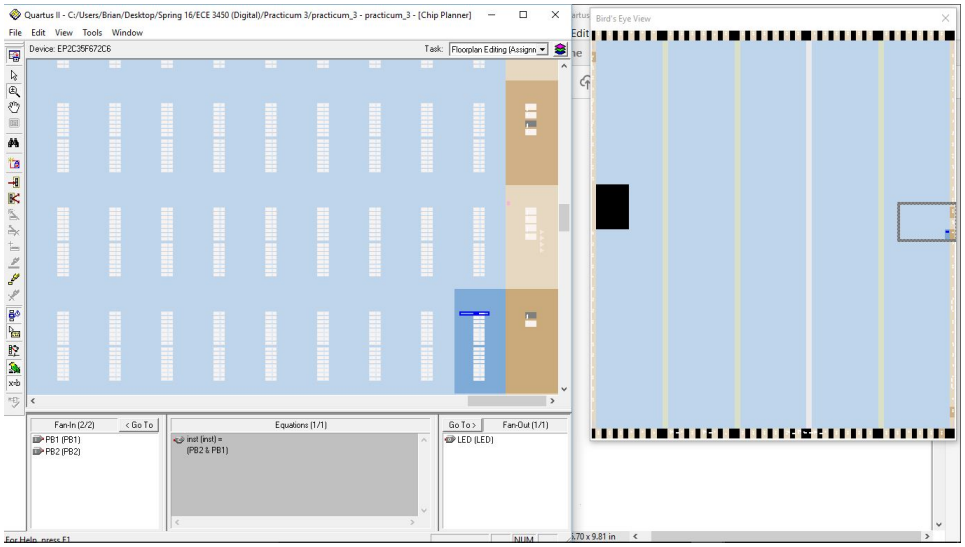

Figure 3.5: Part 3: Floorplan

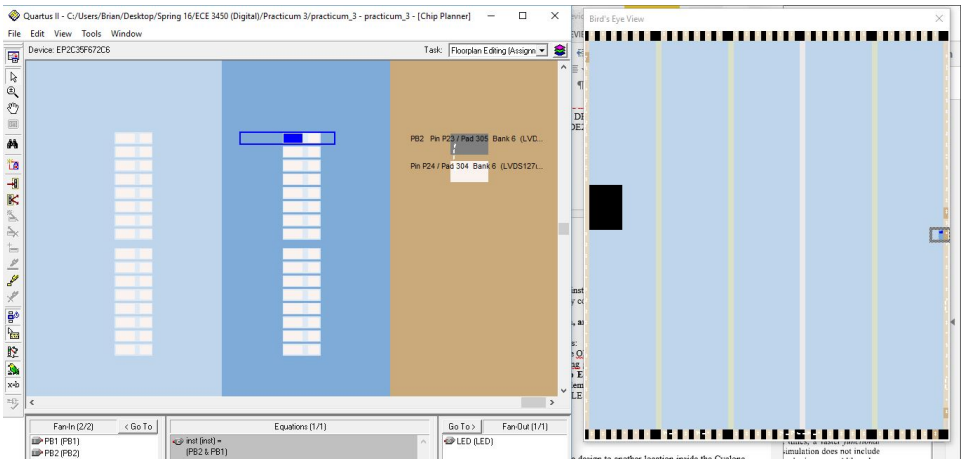

Figure 3.6: Part 3: Closer View of Floorplan

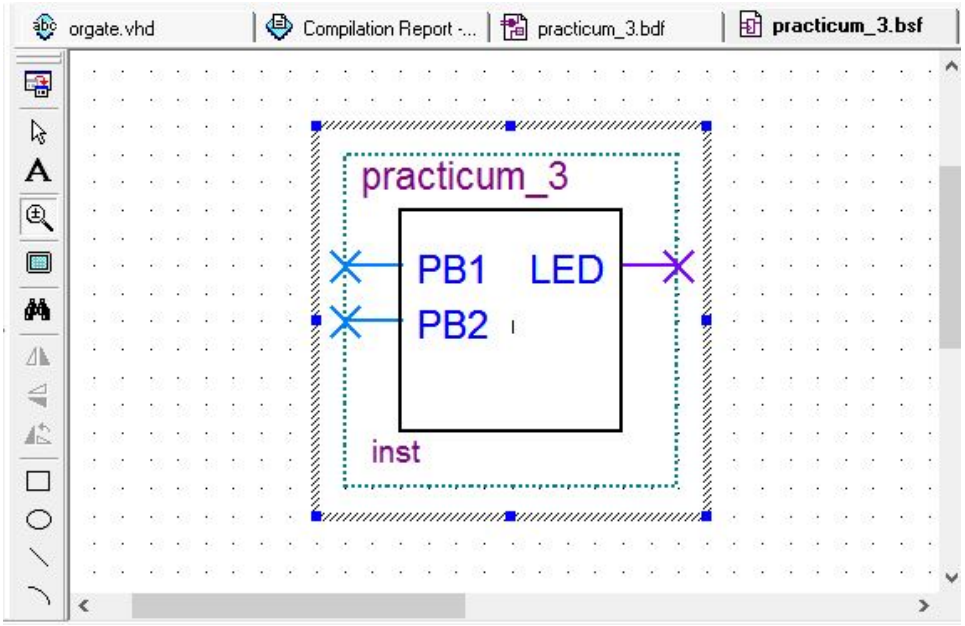

Figure 3.7: Part 3: Block Diagram

3.6 Conclusions

This practicum served as the introduction to a powerful digital CAD tool, Altera Quartus II. Each section of the lab was completed and documented successfully. By successfully modeling an ORing gate (acting as an AND gate, specifically) and observing the timing behavior of the circuit, we can understand how logic placement on a chip can be very important.

3.7 Prelab

Table 3.1: Prelab: Truth Table for an "ORing gate" with active-low inputs connected to pushbuttons and an active-low output driving an LED

PB1	PB2	OUT	LED (on or off)
0	0	0	ON
0	1	0	ON
1	0	0	ON
1	1	1	OFF

4

Practicum 4: VHDL Coding and Simulation Assignments

4.1 Overview

In this practicum, VHDL (VHSIC Hardware Description Language) code was written in Altera Quartus II. This code was then used to generate a timing diagram and TPD analyses. VHDL is a powerful tool used to describe and analyze digital systems such as field programmable gate arrays (FPGAs) and integrated circuits.

4.2 Purpose

The purpose of this practicum is to gain a better understanding of coding in VHDL and performing timing diagram analyses. In addition, by observing the TPDs of two different chips, we can observe how signals propagate through these chips. Depending on the application, one chip may be preferable over another.

4.3 Procedure

The following problems were performed using Brown and Vranesic's textbook *Fundamentals of Digital Logic with VHDL Design* 3^{rd} *Edition*.

1. Problem 2.52 on page 74. A functional simulation in Altera Quartus II was used to verify.
2. Problem 3.51 a and b on page 164. Quartus II was used to observe the timing simulation. A critical path (the signal path with the longest propagation delay) was found in the circuit. For **Part a**, the Max 7000S CPLD (EPM7128SLC84-7) was used. For **Part b**, the Cyclone II FPGA (EP2C35F672C6) was used.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  ENTITY problem1 IS
4      PORT (  x1, x2, x3, x4  : IN    BIT ;
5                  f1, f2      : OUT   BIT ) ;
6  END problem1 ;
7
8  ARCHITECTURE LogicFunc OF problem1 IS
9  BEGIN
10     f1 <= ((x1 AND x3) OR (NOT x1 AND NOT x3)) OR ((x2 AND x4) OR (NOT x2 AND NOT x4));
11     f2 <=  (x1 AND x2 AND NOT x3 AND NOT x4) OR (NOT x1 AND NOT x2 AND x3 AND x4) OR
12     (x1 AND NOT x2 AND NOT x3 AND x4) OR (NOT x1 AND x2 AND x3 AND NOT x4);
13 END LogicFunc ;
14
```

Figure 4.1: Laboratory Part 1 (Problem 2.52) Code

3. Problem 4.45 on page 245. A functional simulation in Quartus II was performed to verify.

4. Problem 5.16b on page 313. A functional simulation in Quartus II was performed to verify.

5. Problem 7.10 on page 478. A functional simulation in Quartus II was performed to verify.

The following problems were individual assignments performed outside of class:

1. Problem 2.51 on page 74. A functional simulation in Quartus II was performed to verify.

2. Problem 3.52 a and b on page 164. Quartus II was used to observe the timing simulation. A critical path (the signal path with the longest propagation delay) was found in the circuit. For **Part a**, the Max 7000S CPLD (EPM7128SLC84-7) was used. For **Part b**, the Cyclone II FPGA (EP2C35F672C6) was used.

3. Problem 4.47 on page 245. A functional simulation in Quartus II was performed to verify.

4. Problem 5.16a on page 313. A functional simulation in Quartus II was performed to verify.

5. Problem 7.11 on page 478. A functional simulation in Quartus II was performed to verify.

4.4 Calculations, Data, and Graphs

Refer to figures.

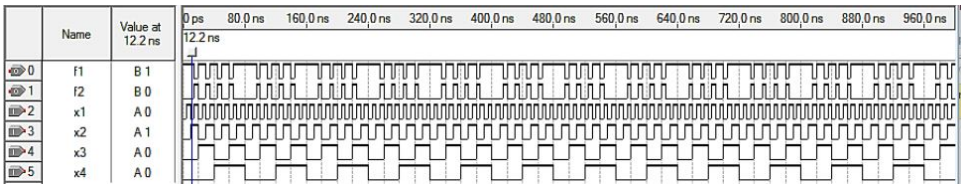

Figure 4.2: Laboratory Part 1 (Problem 2.52) Timing Diagram

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  ENTITY problem2 IS
4      PORT (  x1, x2, x3, x4  : IN    BIT ;
5              f       : OUT   BIT ) ;
6  END problem2 ;
7
8  ARCHITECTURE LogicFunc OF problem2 IS
9  BEGIN
10     f <= (x2 AND NOT x3 AND NOT x4) OR (NOT x1 AND x2 AND x4) OR (NOT x1 AND x2 AND x3) OR (x1 AND x2 AND x3);
11 END LogicFunc ;
12
```

Figure 4.3: Laboratory Part 2 (Problem 3.51) Code

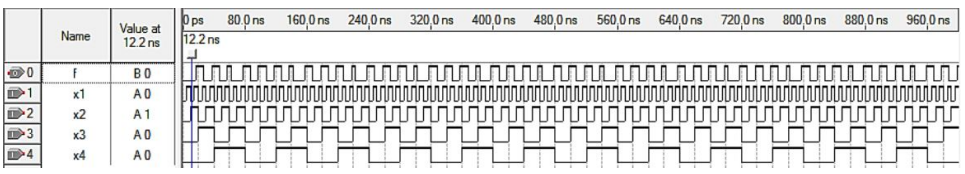

Figure 4.4: Laboratory Part 2 (Problem 3.51) Timing Diagram

tpd

	Slack	Required P2P Time	Actual P2P Time	From	To
1	N/A	None	7.500 ns	x1	f
2	N/A	None	7.500 ns	x2	f
3	N/A	None	7.500 ns	x3	f
4	N/A	None	7.500 ns	x4	f

Figure 4.5: Laboratory Part 2 (Problem 3.51) TPD Using Max 7000S Chip

tpd

	Slack	Required P2P Time	Actual P2P Time	From	To
1	N/A	None	9.457 ns	x4	f
2	N/A	None	9.272 ns	x1	f
3	N/A	None	5.407 ns	x2	f
4	N/A	None	5.133 ns	x3	f

Figure 4.6: Laboratory Part 2 (Problem 3.51) TPD Using Cyclone II Chip

4.5 Discussion of Results

Laboratory

1. Problem 1: f1 and f2 appear to be the complement of each other.

2. Problem 2: The MAX 7000S device exhibited a propagation delay of 7.5 ns whereas the Cyclone II FPGA had a range of propagation delays. The Cyclone II had the smallest propagation delay (5.133 ns), but it also had the largest (9.457 ns).

3. Problem 3: Problem 3 involved a series of logic gates.

4. Problem 4: Here, we observe 'Cout,' the ORing of 3 AND gates.

5. Problem 5: In this problem we have to define a 'Behavior' structure and a process with a reset and clock.

Homework

1. Problem 1: Observing the VHDL code, f1 and f2 look like very different logic functions. However, when we run the simulation, we see that f1 and f2 exhibit the same behavior. Using DeMorgan's Laws, we can prove again that the two logic functions are the same.

2. Problem 2: The MAX 7000S device exhibited a propagation delay of 7.5 ns whereas the Cyclone II FPGA had a range of propagation delays. The Cyclone II had the smallest propagation delay (5.318 ns), but it also had the largest (10.325 ns).

3. Problem 3: Problem 3 involved a series of NOR gates to produce a single output 'f.'

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   ENTITY problem3 IS
4       PORT (  x1, x2, x3, x4, x5  : IN    BIT ;
5                       f       : OUT   BIT ) ;
6   END problem3 ;
7
8   ARCHITECTURE LogicFunc OF problem3 IS
9       signal g: BIT;
10      signal k: BIT;
11  BEGIN
12      g <= (x1 OR x2 OR x5);
13      k <= ((x3 AND NOT x4) OR (NOT x3 AND x4));
14      f <= ((g AND k) OR (NOT g AND NOT k));
15  END LogicFunc ;
16
```

Figure 4.7: Laboratory Part 3 (Problem 4.47) Code

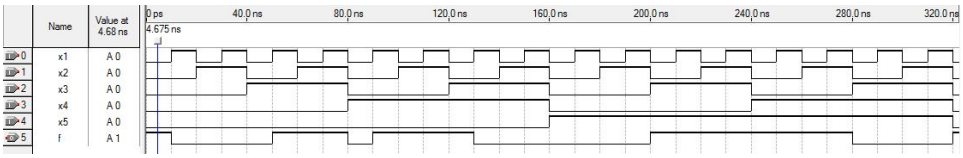

Figure 4.8: Laboratory Part 3 (Problem 4.47) Timing Diagram

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3
4   ENTITY Problem4 IS
5       PORT (Cin, y, m, q :IN STD_LOGIC;
6             s, Cout :OUT STD_LOGIC);
7   END Problem4;
8
9   ARCHITECTURE LogicFunc OF Problem4 IS
10  SIGNAL x :STD_LOGIC;
11  BEGIN
12      x <= (m AND q);
13      s <= x XOR y XOR Cin;
14      Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y);
15  END LogicFunc;
16
```

Figure 4.9: Laboratory Part 4 (Problem 5.16b) Code

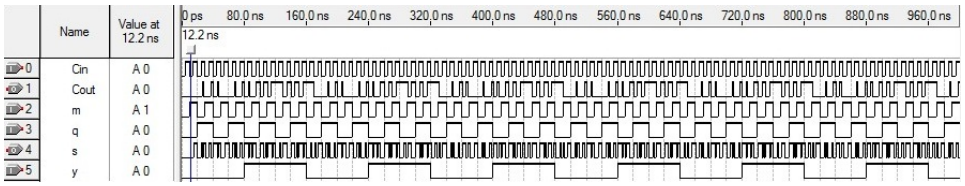

Figure 4.10: Laboratory Part 4 (Problem 5.16b) Timing Diagram

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3
4   ENTITY Problem6 IS
5       PORT (T, RESET, Clock :IN STD_LOGIC;
6             Q :OUT STD_LOGIC);
7   END Problem6;
8
9   ARCHITECTURE Behavior OF Problem6 IS
10  SIGNAL QN :STD_LOGIC;
11  BEGIN PROCESS (RESET, Clock)
12      BEGIN
13          IF RESET= '0' THEN
14              QN <= '0';
15          ELSIF Clock'EVENT AND Clock = '1' THEN
16              IF T='1' THEN
17              QN <= NOT QN;
18              ELSE
19              QN <= QN;
20              END IF;
21          END IF;
22          END PROCESS;
23          Q<=QN;
24  END Behavior;
25
```

Figure 4.11: Laboratory Part 5 (Problem 7.10) Code

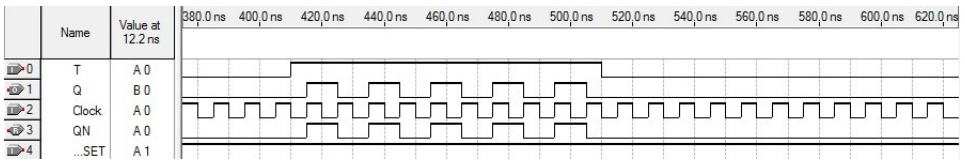

Figure 4.12: Laboratory Part 5 (Problem 7.10) Timing Diagram

```
1  --(1) Problem 2.51
2  library ieee;
3  use ieee.std_logic_1164.all;
4  ENTITY Problem1 IS
5      PORT (  x1, x2, x3, x4  : IN    BIT ;
6              f1, f2      : OUT   BIT ) ;
7  END Problem1;
8
9  ARCHITECTURE LogicFunc OF Problem1 IS
10 BEGIN
11     f1 <= (x1 AND NOT x3) OR (x2 AND NOT x3) OR (NOT x3 AND NOT x4) OR (x1 AND x2) OR (x1 AND NOT x4);
12     f2 <= (x1 OR NOT x3) AND (x1 OR x2 OR NOT x4) AND (x2 OR NOT x3 OR NOT x4);
13 END LogicFunc;
14
```

Figure 4.13: Homework Part 1 (Problem 2.51) Code

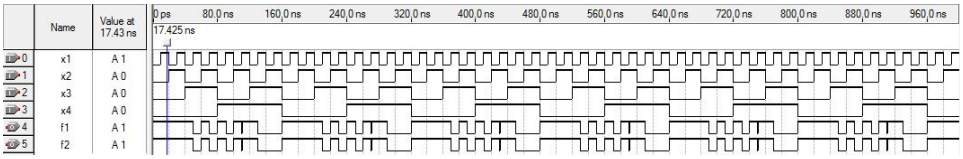

Figure 4.14: Homework Part 1 (Problem 2.51) Timing Diagram

```
1  --(2) Problem 3.52a,b
2   --Part a: EPM7128SLC84-7
3   --Part b: EP2C35F672C6
4   library ieee;
5   use ieee.std_logic_1164.all;
6  ENTITY Problem2 IS
7      PORT (  x1, x2, x3, x4  : IN    BIT ;
8              f       : OUT   BIT ) ;
9  END Problem2;
10
11 ARCHITECTURE LogicFunc OF Problem2 IS
12 BEGIN
13     f <= (x1 OR x2 OR NOT x4) AND (NOT x2 OR x3 OR NOT x4) AND (NOT x1 OR x3 OR NOT x4) AND (NOT x1 OR NOT x3 OR NOT x4);
14 END LogicFunc;
```

Figure 4.15: Homework Part 2 (Problem 3.52) Code

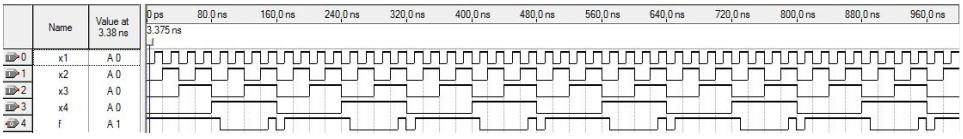

Figure 4.16: Homework Part 2 (Problem 3.52) Timing Diagram

tpd

	Slack	Required P2P Time	Actual P2P Time	From	To
1	N/A	None	7.500 ns	x1	f
2	N/A	None	7.500 ns	x2	f
3	N/A	None	7.500 ns	x3	f
4	N/A	None	7.500 ns	x4	f

Figure 4.17: Homework Part 2 (Problem 3.52) TPD Using Max 7000S Chip

tpd

	Slack	Required P2P Time	Actual P2P Time	From	To
1	N/A	None	10.325 ns	x4	f
2	N/A	None	9.471 ns	x1	f
3	N/A	None	5.586 ns	x2	f
4	N/A	None	5.318 ns	x3	f

Figure 4.18: Homework Part 2 (Problem 3.52) TPD Using Cyclone II Chip

```
1   --(3) Problem 4.47
2   --Refer to figure 4.28b (pg. 202)
3   library ieee;
4   use ieee.std_logic_1164.all;
5   ENTITY Problem3 IS
6       PORT (  x1, x2, x3, x4, x5, x6, x7  : IN    BIT ;
7                   f       : OUT   BIT ) ;
8   END Problem3;
9
10  ARCHITECTURE LogicFunc OF Problem3 IS
11  BEGIN
12      f <= (x1 NOR ((x2 NOR x2) NOR (x3 NOR x3))) NOR
13      (((x4 NOR x4) NOR (x5 NOR x6)) NOR x7);
14  END LogicFunc;
```

Figure 4.19: Homework Part 3 (Problem 4.47) Code

4. Problem 4: Here, we observe 'f', the output of the full adder in a segment of a 4x4 multiplier circuit and 'Cout,' the carry bit output of the full adder.

5. Problem 5: We notice that when 'Reset' has a logic state 'high,' output 'Q' exhibits JK flip-flop behavior as shown in lines 15-18 of the code. When 'Reset' has a logic state 'low,' 'Q' remains at logic state 'low.' However, when 'Reset' toggles from 'high' to 'low,' 'Q' has a logic state 'high' until 'J' toggles 'high.'

4.6 Conclusions

Each problem involved VHDL coding and a timing analysis. These steps were performed successfully. Problem 2 in both the laboratory and homework sections involved two TPD analyses using two different

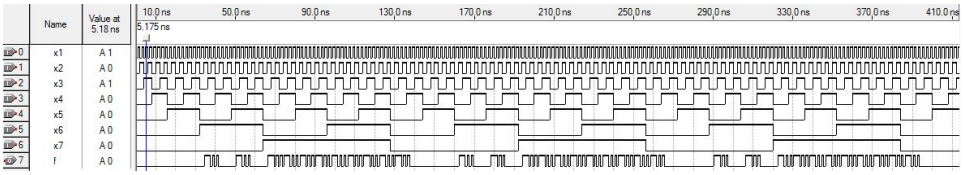

Figure 4.20: Homework Part 3 (Problem 4.47) Timing Diagram

```
1   --(4) Problem 5.16a
2   --Refer to Figure 5.32b (page 294)
3   library ieee;
4   use ieee.std_logic_1164.all;
5   ENTITY Problem4 IS
6       PORT (  mk0, mk1, q0, q1, Cin    : IN    BIT ;
7                       f,  Cout         : OUT   BIT ) ;
8   END Problem4;
9
10  ARCHITECTURE LogicFunc OF Problem4 IS
11      SIGNAL x: BIT;
12      SIGNAL y: BIT;
13  BEGIN
14      x <= q0 AND mk1;
15      y <= q1 AND mk0;
16      f <= Cin XOR x XOR y;
17      Cout <= (Cin AND x) OR (Cin AND y) OR (x AND y);
18  END LogicFunc;
```

Figure 4.21: Homework Part 4 (Problem 5.16a) Code

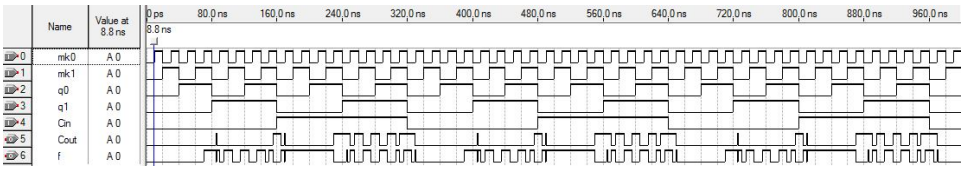

Figure 4.22: Homework Part 4 (Problem 5.16a) Timing Diagram

```
1  --(5) Problem 7.11
2  --JK flip-flop using behavioral code
3  library ieee;
4  use ieee.std_logic_1164.all;
5  ENTITY Problem5 IS
6      PORT (  J, K, Reset, Clock  : IN    BIT ;
7                    Q       : OUT   BIT ) ;
8  END Problem5;
9
10 ARCHITECTURE LogicFunc OF Problem5 IS
11     SIGNAL Qint: BIT;
12 BEGIN
13     PROCESS(Reset, Clock)
14     BEGIN
15         IF Reset = '0' THEN
16             Qint <= '0';
17         ELSIF Clock'EVENT AND Clock = '1' THEn
18             Qint <= (J AND NOT Qint) OR (NOT K AND Qint);
19         END IF;
20     END PROCESS;
21 Q <= Qint;
22 END LogicFunc;
```

Figure 4.23: Homework Part 5 (Problem 7.11) Code

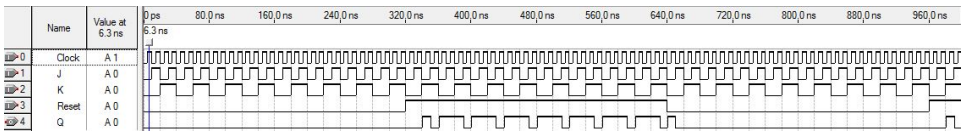

Figure 4.24: Homework Part 5 (Problem 7.11) Timing Diagram

chips: the MAX 7000S and the Cyclone II. From these simulations, we can observe how long signals take to propagate in each chip. Depending on the application, one chip may be preferable over the other. VHDL is a powerful, yet challenging, tool. It can be used to describe a logical circuit without building it. By observing the behavior of the circuit specified by VHDL code, we can make accurate predictions about how the physical circuit will behave.

5

Practicum 5: Capacitance Sensor Project

5.1 Overview

In this report, we will discuss two different implementations of a capacitance sensor: one using a Altera DE2 Development and Education Board in conjunction with Altera's Quartus II and the other using an Arduino.

To measure the capacitance of a given capacitor, we can measure the period of the output of a capacitor with an input clock signal. Because of the nature of a capacitor, the period is directly proportional to the capacitance. So, the higher the capacitance, the higher the period is. To measure the period, both implementations use a 555 timer (further detailed in Section 2: Design). Then, the job of the DE2 and Arduino is to parse the output of the 555 timer into something useful for us humans. Specifically, we have the DE2 board display the period on two of its seven segment LED displays. With the Arduino, we have the period displayed on the serial monitor in the Arduino application. If the period exceeds the range of the seven segment displays on the DE2 (> 99), we implement an offset feature to subtract powers of 2 from the output so that the display reads a value less than 99. To obtain the final period, we just need to add the offset value to the displayed value. This feature is implemented in the Arduino capacitance sensor in the form of a potentiometer.

In the sections that follow, we will run through the designs, simulations, and testing & verification of both the DE2 and Arduino implementations.

5.2 Design

In this section, we provide details on both the DE2-based design and the Arduino-based designs.

5.2.1 DE2

Our capacitance sensor had to meet a certain number of specifications. For instance, we needed to specify the maximum capacitance that our sensor can measure:

$$C_{max} = 4C_{min}$$

We chose $C_{min} = 0.1\mu F$ and specified a clock frequency f_{clk} of 10 KHz. We chose 10KHz because we did not want the clock to be too fast in the case of 100KHz or too slow in the case of 1 KHz. We decided that 10KHz would be a good middle-ground. In the calculations below, we solve for ΔT, the range of periods we can measure due to our ΔC range of capacitor values we can measure with our sensor. We also solve for resistor values R_A and R_B that we need in our 555 timer circuit.

The overall block diagram of our DE2 implementation of the capacitance sensor is as follows.

The 555 timer serves as the timer circuit that does the measuring of the capacitance. This block in the diagram is a physical circuit and will thus not be coded for later on. The divide by 2 circuit is a JK flipflop that transforms the output of the 555 timer circuit's period to twice that (from period T to 2T). The 50 MHz oscillator on the DE2 board feeds in to the CLK_DIV block, which gives us our operating frequency of 10KHz. That 10KHz clock signal is then fed into our TIMER COUNT block. This TIMER_COUNT block measures the change of period (ΔT) of oscillation due to the capacitor. The 555 timer The period and capacitance are in a direct relationship so any positive change in the capacitance will result in a positive change in the period. Likewise, a negative change in the capacitance will result in a negative change in the period of oscillation. The multiplexer (or adder) block, reads in binary values set by the switches and subtracts that value from the output of the TIMER_COUNT block. For instance, if the capacitance is so large the period exceeds 99 on the displays (let us say it is 135), we can subtract 64 to get 71. The offset switches are layed out in decreasing powers of two. Reading from left to right the switches subtract $2^8, 2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0$. Having two or more switches in the on position will add the powers and subtract them from the TIMER_COUNT output. The decoder is used to convert the binary from the output of the adder into the seven different segments in

Capacitance Sensor Project

$$f_{osc} = \frac{1}{T} = \frac{1}{0.69(R_A + 2R_B)C}$$

$$T = \underbrace{0.69(R_A + 2R_B)}_{R_{eff}} C$$

$$= R_{eff} C$$

$$\Delta C = C_{max} - C_{min}$$
$$= 3 C_{min}$$

$$T_{min} = R_{eff} C_{min}$$

$$T_{max} = R_{eff} C_{max}$$

$$\Delta T = T_{max} - T_{min} = R_{eff} \Delta C$$
$$= 99 T_{CLK}$$

$C_{min} = 0.1\,\mu F$

$\Delta C = 3 C_{min} = 3(0.1\,\mu F) = 0.3\,\mu F$

Determine possible values of ΔT

$$\Delta T = R_{eff} \Delta C$$
$$= 0.69(R_A + 2R_B)\Delta C = 99 T_{CLK}$$

$f_{CLK} = 10\,KHz$

$T_{CLK} = 10^{-4}\,s$

$\Delta T = 99(10^{-4}\,s) = \boxed{0.0099\,s}$

Clock Periods

$T_- = |10^{-4} - 0.0099| = 0.0098\,s$

$T_+ = |10^{-4} + 0.0099| = 0.01\,s$

Figure 5.1: Calculations for DE2 1/2

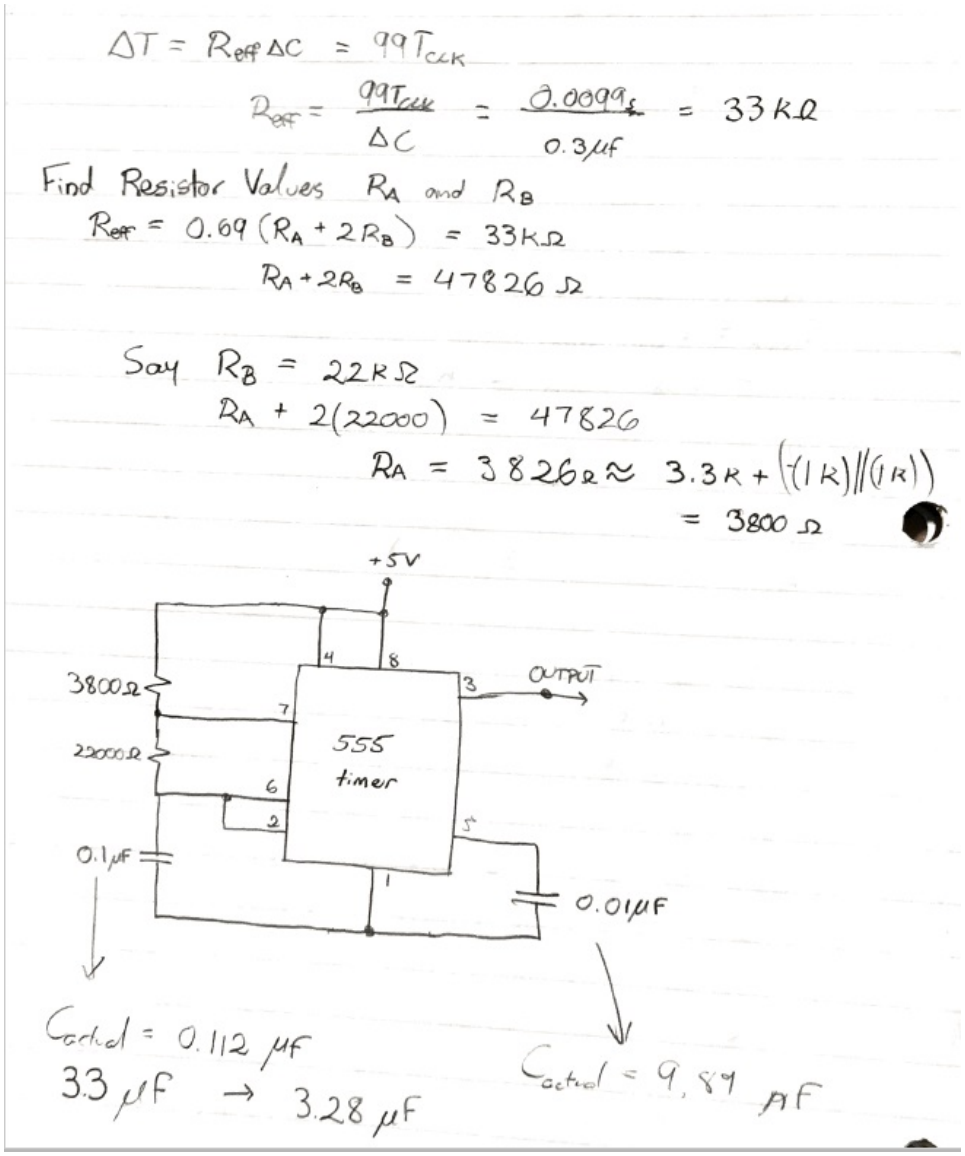

Figure 5.2: Calculations for DE2 2/2

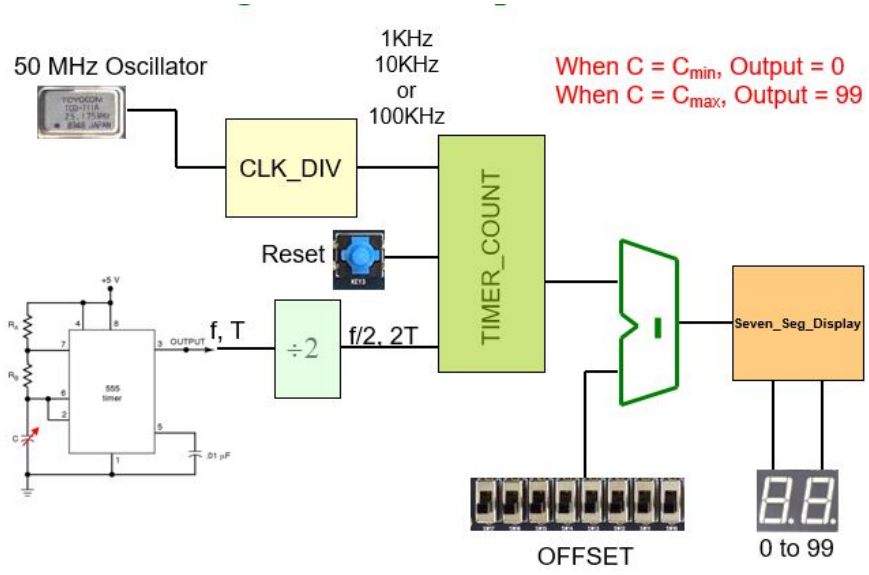

Figure 5.3: Block Diagram of the DE2 Capacitance Sensor

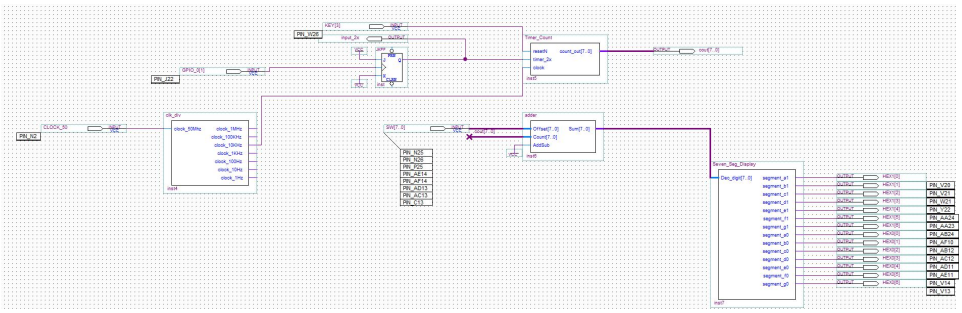

Figure 5.4: Block Diagram of the DE2 Capacitance Sensor in Altera Quartus II

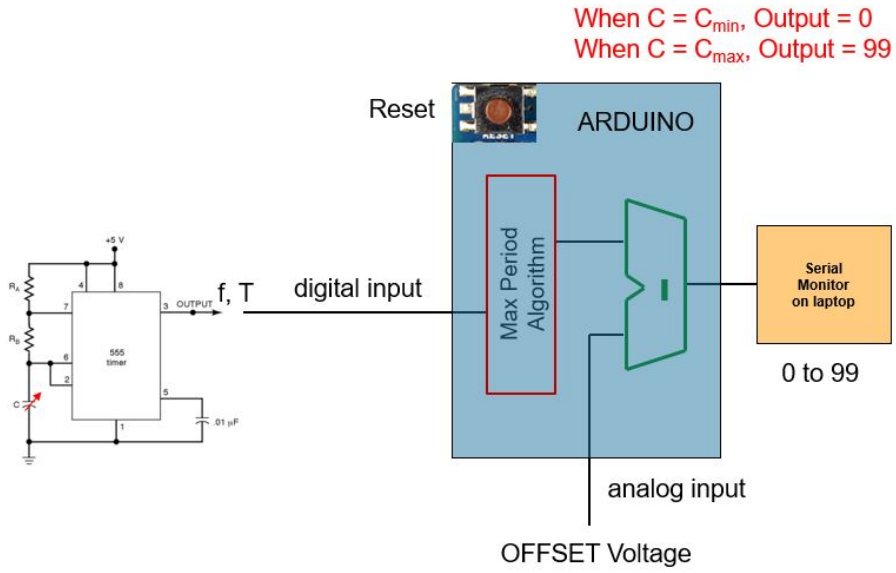

Figure 5.5: Arduino Block Diagram

the seven segment display. The reset button resets the values displayed on the seven segment displays (SEVEN_SEG_DISPLAY)

The CLK_DIV, TIMER_COUNT, adder, SEVEN_SEG_DISPLAY, and DEC 7SEG blocks were written in VHDL using Altera Quartus II. The code for each block is listed in the Appendix. The overall block diagram for the capacitance sensor is shown below.

5.2.2 Arduino

Using the same specifications from the DE2 implementation, we use the Arduino to replace the DE2 board. This requires all coding in the Arduino application since we keep the same 555 timer circuit. The only change we need to make is the offset voltage (as seen in figure 5). This offset voltage is the same concept as the switches in the DE2 implementation. We us a potentiometer to control the voltage reaching the Arduino. If the period reading is over 99 we set the offset voltage to a value such that we do not have overflow (in the range of 00 to 99). The reset button functionality is built in the Arduino. So, pressing the reset button restarts the program.

The Arduino sketch used for the capacitance sensor is listed in the Appendix. We set the timer_pin to pin 13 and the initial value of the offset to 0. The offset value is then read from analog pin A0. The

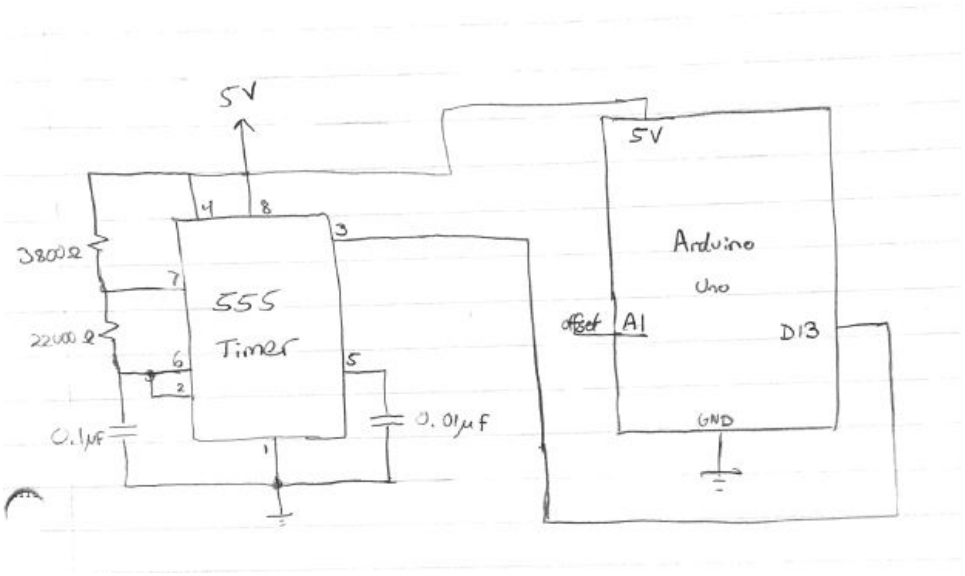

Figure 5.6: Arduino Pin Layout

duty cycle of the 555 timer is $\frac{2}{3}$ or approximately 0.67. We then assign the clock period in microseconds. Our clock frequency is 10KHz, so we set the period to 100 μs. Then, we initialize the count values to 0 (in the same fashion as in the DE2 TIMER_COUNT VHDL implementation). We define helper functions setup(), meas_period(), and output_value(). setup() defines the pin mode to use pin 13 as the input. meas_period() sets the pulse duration to the length of the pulse it reads from pin 13 (from the 555 timer circuit). Because the pulse has a duty cycle of 0.67, we can use this fact to calculate the period of the signal. Just like we did in the DE2 implementation, we only read out the maximum period the sensor measures. The output_value() function reads the offset value from the potentiometer and subtracts it from the count_final variable we define as the period of the signal in the meas_period() function. Then, in the main loop, we perform meas_period() and output_value() and print out the period measured in meas_period(), the offset, and the output value (with offset). These values are output to the serial monitor in the Arduino application. We can then observe the present values and compare them to past values the sensor reads.

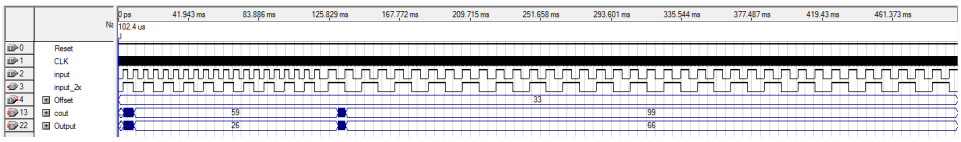

Figure 5.7: Vector waveform from simulation of sensor

5.3 Simulation

In this section, we will discuss the simulation results of the DE2 implementation of the sensor. The Arduino sensor did not have a simulation and thus will not be discussed here.

To test our DE2 capacitance sensor, we created a vector waveform file in Altera Quartus II. This waveform had four inputs (Reset, CLK, input, and Offset) and three outputs (input_2x, cout, and Output). The reset was set high so that all of our values would not be set to 0. Our clock CLK was set at 10KHz as described before. The input was a square pulse train that halved its period partway through the simulation. Our offset was set to 33 so that the maximum cout value would be 99. The output maximum value would then be $cout - offset = 99 - 33 = 66$. This is observed when the period of the input signal is halved. Before it is halved, however, the cout period is 59 and thus $cout - offset = 59 - 33 = 26$. This is seen from approximately 0ps to 130ms in the simulation. From these results, we can conclude that our capacitance sensor does what we want it to do and we can begin testing and verifying the design.

5.4 Testing and Verification

5.4.1 DE2

To test the sensor, we used various capacitors ranging from $C_{min} = 0.1\mu F$ to $0.43\mu F$. The oscilloscope displays in Figures 9-14 are the outputs from the 555 timer with the specified capacitor values. The table in Figure 15 summarizes the results obtained through the seven segment displays on the DE2 board (Output Values on DE2 Board column) as well as the measured periods from the oscilloscope (Period T column).

In the Output Values on DE2 Board column of Figure 5.14, we write the period as a sum of two values if it exceeds 99. The first values are the offsets subtracted using the switches and the second value is the reading on the seven segment displays after subtracting said value.

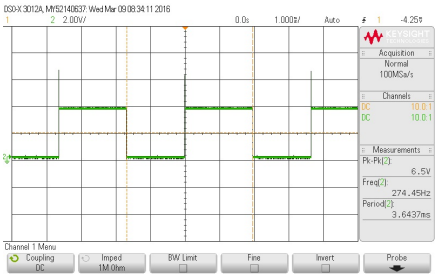

Figure 5.8: $0.1\mu F$ 555 timer output

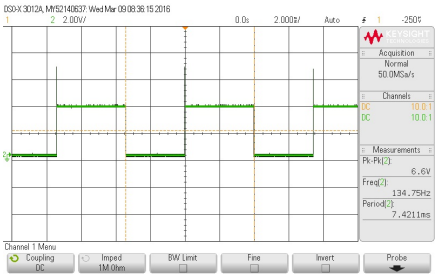

Figure 5.9: $0.22\mu F$ 555 timer output

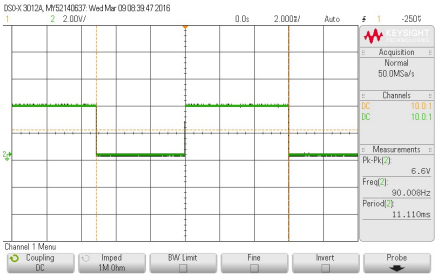

Figure 5.10: $0.32\mu F$ 555 timer output

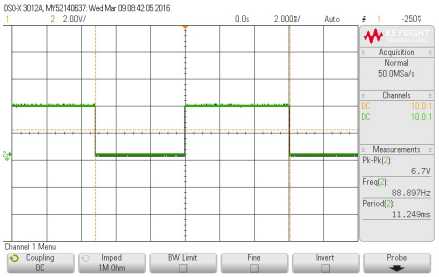

Figure 5.11: $0.33\mu F$ 555 timer output

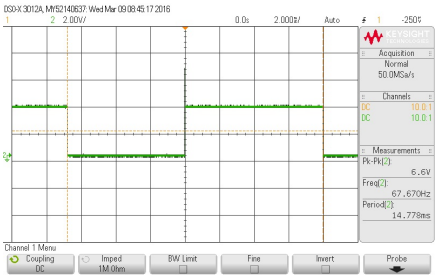

Figure 5.12: $0.41\mu F$ 555 timer output

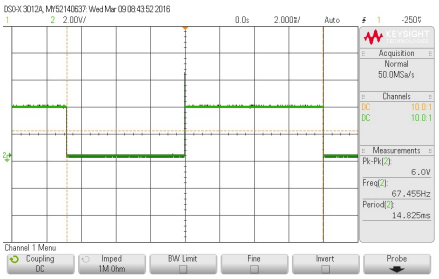

Figure 5.13: $0.43\mu F$ 555 timer output

Adding the two results in the overall period.

We used Altera Quartus II's Signal Tap feature to test our sensor. Signal Tap can be used to view the behavior and response of any signal within the Quartus design file. Figure 5.15 shows that when we use $C_{min} = 0.1\mu F$, our count_final value is 35. Figure 5.16 shows that when we use $C_{max} = 0.4\mu F$, our count_final value is 148. Hence, we

C value	Measured C value	Period T	Output Values on DE2 Board
0.1uF	0.103uF	3.6437ms	36
0.22uF	0.209uF	7.4211ms	64
0.1uF + 0.22uF	0.312uF	11.110ms	32 + 69 = 101
0.33uF	0.319uF	11.249ms	64 + 49 = 103
0.41uF	0.389uF	14.778ms	128 + 20 = 148
0.1uF + 0.33uF	0.422uF	14.825ms	128 + 21 = 149

Figure 5.14: Summary of our measurements with different capacitors

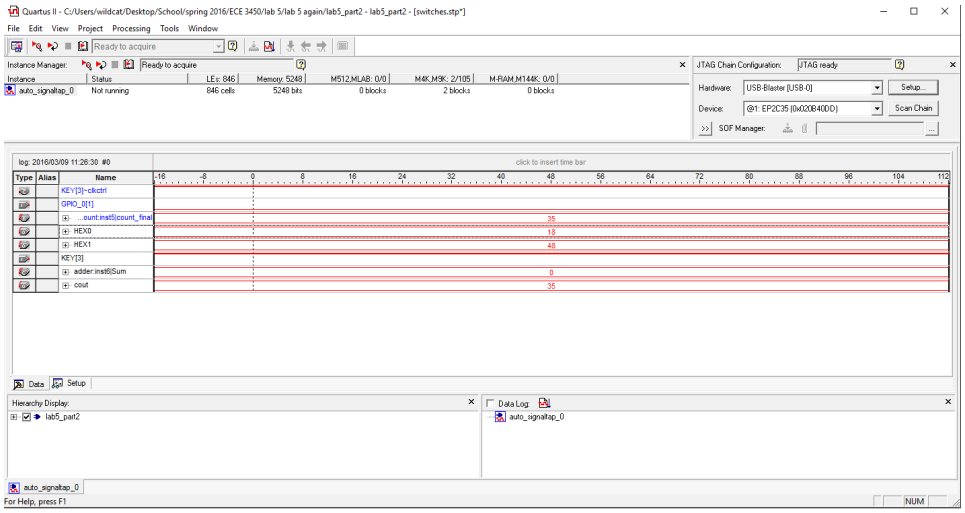

Figure 5.15: Altera Quartus II Signal Tap Using $C_{min} = 0.1\mu F$

need to use the offset switches to subtract from 148 to make the value less than 99.

We observe that the periods measured from the output of the 555 timer match fairly close to the readings we see on the DE2 board.

5.4.2 Arduino

To perform the testing and verification on the Arduino, we needed to measure the output waveforms of the 555 timer and observe the serial monitor outputs. We tested the same capacitor values as the DE2 (ranging from $0.1\mu Fto0.43\mu F$).

For capacitor values smaller than $0.41\mu F$ ($0.1\mu F$, $0.22\mu F$, $0.32\mu F$, and $0.33\mu F$) the count_final value of the period is smaller than 99 and there is no need for the offset. We do need the offset for capacitance values of $0.41\mu F$ and $0.43\mu F$ as evidenced in Figures 27 and 30 where we implement an offset of 64 to make the output less than 99. No-

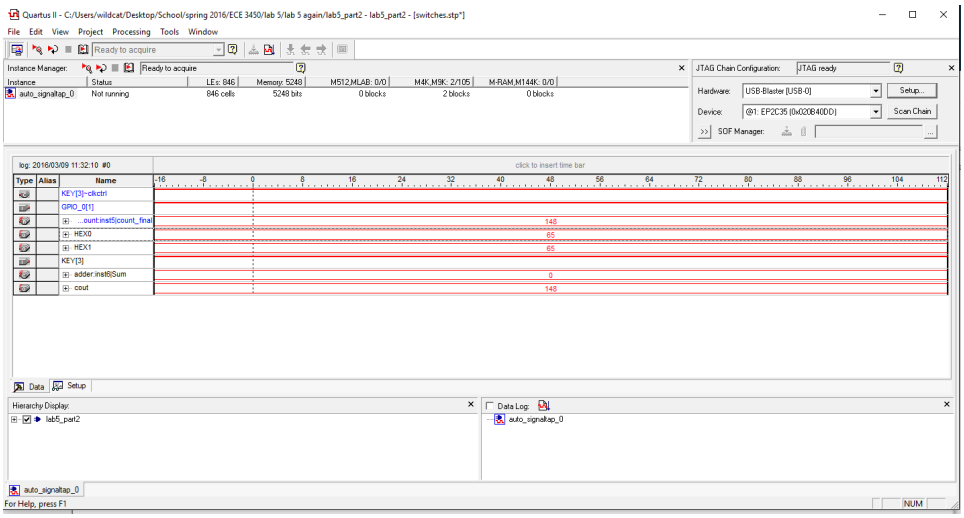

Figure 5.16: Altera Quartus II Signal Tap Using $C_{max} = 0.4\mu F$

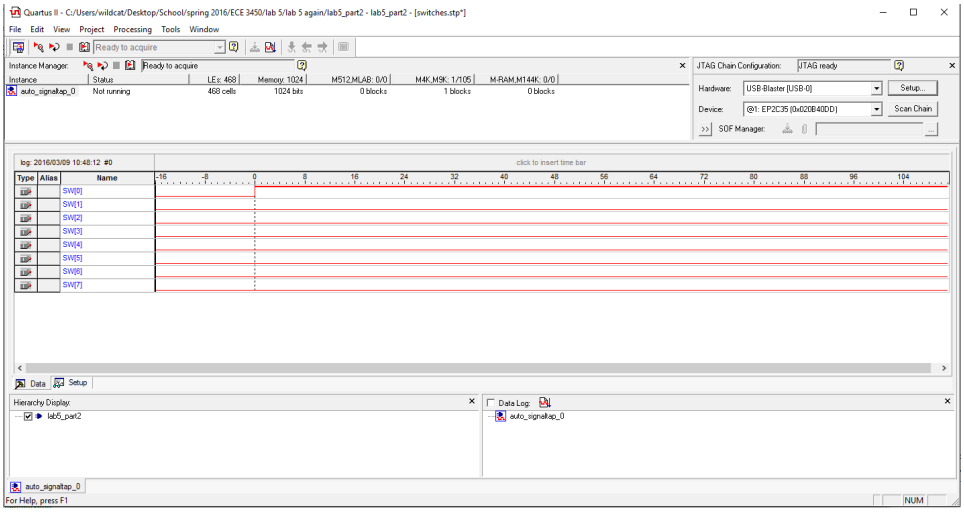

Figure 5.17: Altera Quartus II Signal Tap Observing Switches

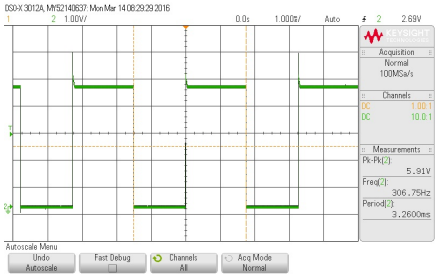

Figure 5.18: $0.1\mu F$ 555 timer output

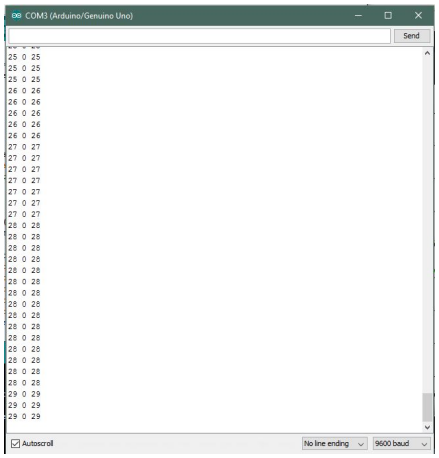

Figure 5.19: $0.1\mu F$ Serial Monitor

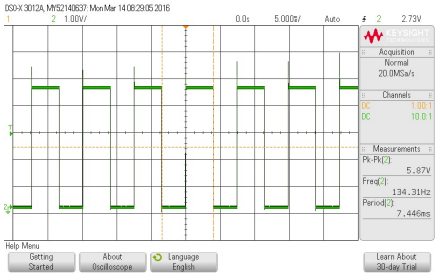

Figure 5.20: $0.22\mu F$ 555 timer output

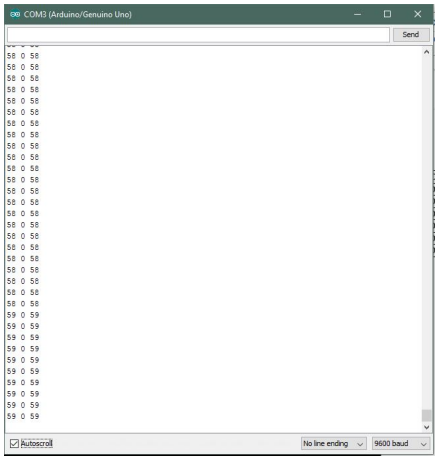

Figure 5.21: $0.22\mu F$ Serial Monitor

tice how the count_final value remains the same and only the output gets smaller as the offset increases. As stated earlier, this is because $output = count\_final - offset$.

In Figure 31, we demonstrate the maximum period functionality in our capacitance sketch. We switch a $0.22\mu F$ capacitor (with a period output of 59) to a $0.33\mu F$ capacitor (with a period output of 90) back to the $0.22\mu F$ capacitor. Notice how neither the count_final nor output values change back to 59. This is because we only store and print the maximum period in our code.

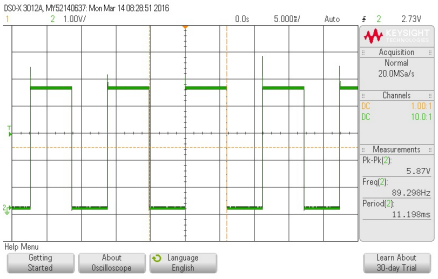

Figure 5.22: $0.32\mu F$ 555 timer output

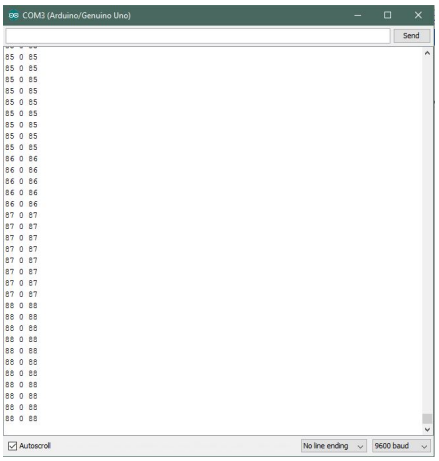

Figure 5.23: $0.32\mu F$ Serial Monitor

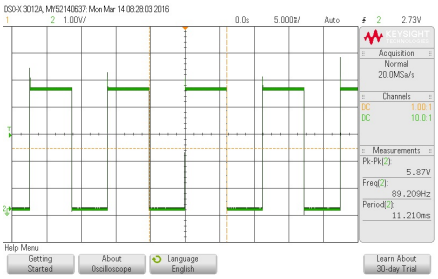

Figure 5.24: $0.33\mu F$ 555 timer output

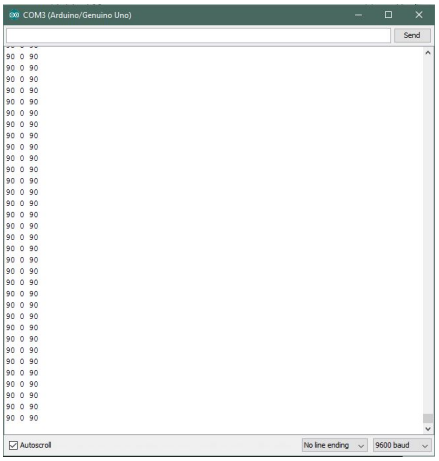

Figure 5.25: $0.33\mu F$ Serial Monitor

5.5 Conclusions

In terms of our results, our measured values on the DE2 board's seven segment display and Arduino's serial monitor came fairly close to the oscilloscope captures of the 555 timer output. Discrepancies could be due to the architecture of the DE2, FPGA, and Arduino and also the capacitor values were not exact.

While the Arduino capacitance sensor implementation is easier and quicker to implement than the DE2 implementation, the Arduino sen-

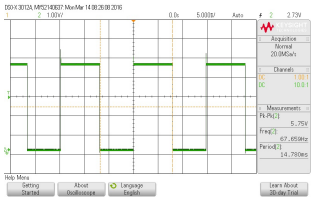

Figure 5.26: 0.41μF 555 timer output

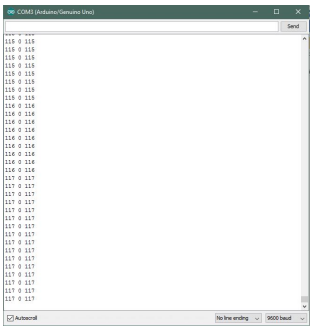

Figure 5.27: 0.41μF Serial Monitor

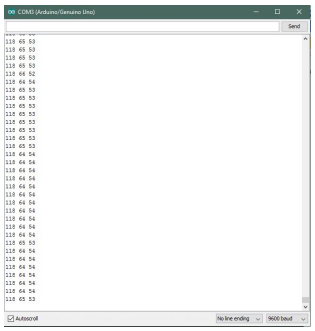

Figure 5.28: 0.41μF Serial Monitor With Offset

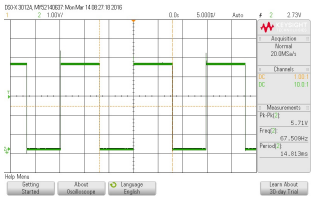

Figure 5.29: 0.43μF 555 timer output

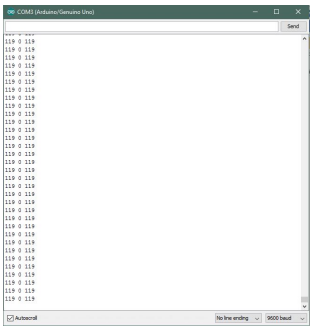

Figure 5.30: 0.43μF Serial Monitor

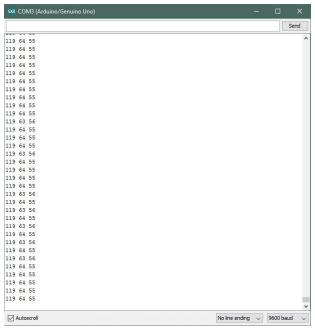

Figure 5.31: 0.43μF Serial Monitor With Offset

sor lacks some testing functionality that the DE2 has. For example, the DE2 has Altera Quartus II's Signal Tap and the vector waveforms to test the functionality of the system. In addition, with the block diagrams, the DE2 implementation is also more visual than the Arduino. The Arduino implementation solely consists of the script we wrote. There is no visual component in the Arduino work environment. This is another advantage of Altera Quartus II and DE2.

However, to quickly implement the sensor without some of the measurement features of the DE2, the Arduino is the optimal choice. The Arduino is easier to code since its language is a much higher level than the DE2's VHDL is. In addition, the DE2 is a hardware implementation while the Arduino is a software implementation using C-like code on fixed hardware. Furthermore, the nature of using two seven segment LED displays limits our output value to 99. Hence, we needed to implement an offset using the switches on the board. In the Arduino

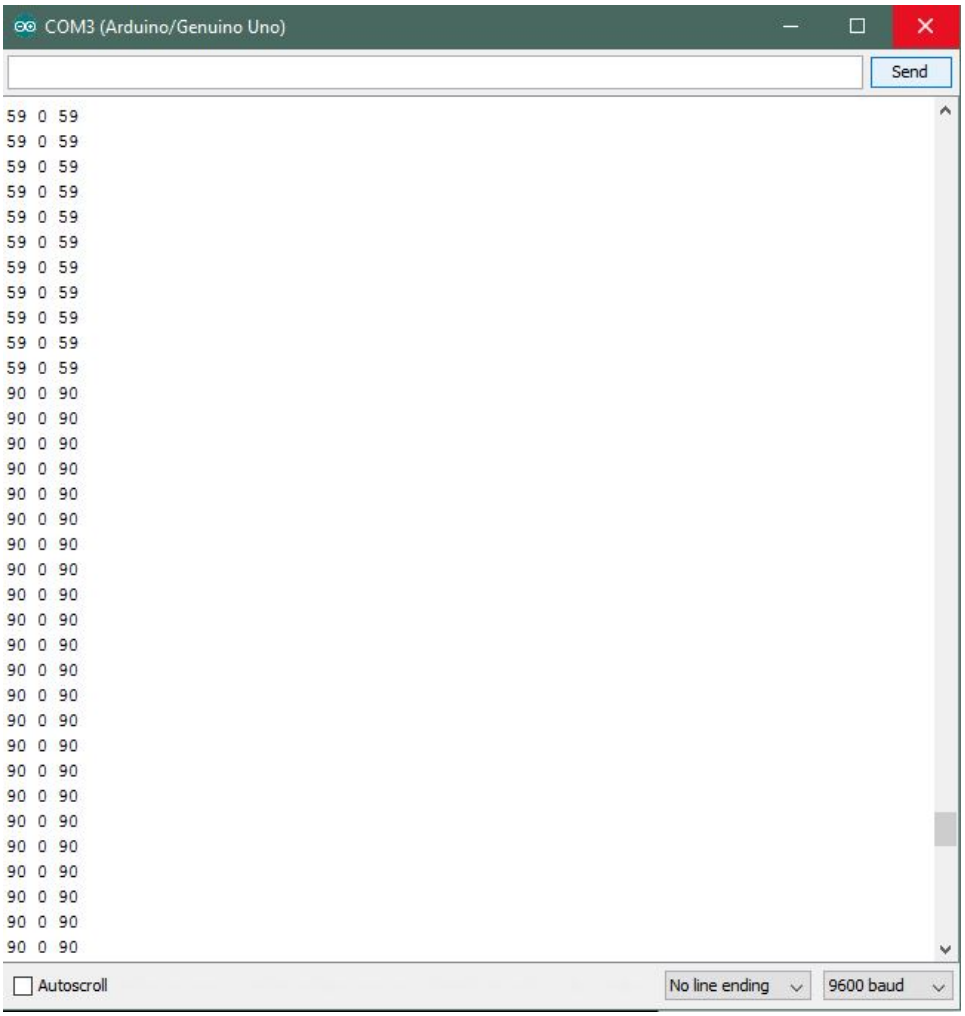

Figure 5.32: Demonstrating Max Period Feature in Code (Switching from $0.22\mu F$ to $0.33\mu F$ to $0.22\mu F$ Capacitors

capacitor sensor, we did not have this limitation, although we still created an offset feature using a potentiometer. The serial monitor can display output values larger than 99 as shown in Section 4: Testing and Verification.

Whether one implementation is better than another is tough to answer. In the end, it highly depends on the application of the sensor. We can only draw conclusions on whether the DE2 capacitance sensor or Arduino capacitance sensor is better when we know the application of the device.

6

Practicum 6: Memory Practicum

6.1 Overview

This practicum deals with the implementation of memory. Because memory is so ubiquitous, it is vital to understand how it works. Using FPGA technology, we will deploy and test the functionality of memory.

6.2 Purpose

Reading and writing are the two basic operations we can perform on memory. Thus, we will explore reading and writing from memory blocks using the Altera DE2 board and its FPGA technology.

6.3 Procedure

1. Part 1: Creating the memory block and testing its functionality
   (a) A new Altera Quartus II project was created to implement the memory module. The target chip was selected to be the Cyclone II EP2C35F672C6 (the FPGA chip on the Altera DE2 board). Create a new BDF file.
   (b) Using the MegaWizard Plug-In Manager, the desired LPM module is created. *RAM: 1-PORT LPM* was selected under the Memory Compiler category. The output file was set to generate VHDL code. The name of the output file was set to be *ramlpm.vhd*. On the next page, the memory size was set to be 32 eight-bit words. The type of RAM block was set to M4K. A single clock was used for the RAM's registers. On

the next page, the 'Q' OUTPUT PORT option was deselected under the WHICH PORTS SHOULD BE REGISTERED? section. The rest of the settings were chosen as the defaults. A symbol file of the *ramlpm.vhd* file was created and placed in the top-level BDF file. Figure 6.1 was used as a reference to select appropriate input and output signals for the BDF file.

(c) The BDF circuit was compiled and the Compilation Report was observed to ensure that Quartus II uses 256 bits in one of the M4K memory blocks in the RAM circuit.

(d) A simulation was performed to demonstrate the reading and writing capability of the memory block.

2. Part 2: Uploading the memory circuit to the Altera DE2 board and testing its functionality

(a) Another Altera Quartus II project was created. This was used to implement the circuit to the Altera DE2 board.

(b) Using another BDF file and the *ramlpm* module that was created in the previous section, use toggle switches SW_{7-0} to input a byte of date into the RAM location specified by a 5-bit address determined by toggle switches SW_{15-11}. Switch SW_{17} was used as the *Write* signal. Key KEY[0] was used as the *Clock* input. The value of the *Write* signal was displayed on LEDG[0]. The address value was displayed on HEX[7] and HEX[6]. The data input into the memory was displayed on HEX[5] and HEX[4] while the data read out of the memory was displayed on HEX[1] and HEX[0].

(c) The circuit was tested to ensure that all 32 memory locations can be loaded properly.

3. Part 4: Using SRAM pin names

(a) Yet another Altera Quartus II project was created. Then, a BDF file that included the ability to load the memory and read its contents. Using the same LED, switch, and 7-segment display assignments as in Part 2, we used the following SRAM pin names to interface the circuit to the IS61LV25616AL chip on the DE2 board:

(b) The circuit was compiled and uploaded to the FPGA chip on the Altera DE2 board.

(c) Then, the functionality of our design was tested by writing and reading values to many memory locations.

Table 6.1: SRAM Pin Assignments

SRAM Port Name	DE2 Pin Name
A_{17-0}	$SRAM\_ADDR_{17-0}$
I/O_{15-0}	$SRAM\_DQ_{15-0}$
$\overline{CE}$	$SRAM\_CE\_N$
$\overline{OE}$	$SRAM\_OE\_N$
$\overline{WE}$	$SRAM\_WE\_N$
$\overline{UB}$	$SRAM\_UB\_N$
$\overline{LB}$	$SRAM\_LB\_N$

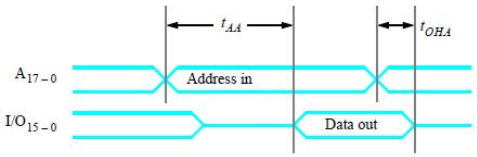

Figure 6.1: SRAM Read Cycle Timing Diagram

6.4 Calculations, Data, and Graphs

See figures for waveforms and block diagrams.

6.5 Discussion of Results

In part 1, we observed that the RAM circuit used 256 bits in one of the M4K blocks. Using Figure 6.1 and Figure 6.2 as a reference and referring to Figure 6.5, we see that when DataIn is set to AB and the Address is set to 10 DataOut reads AB on the next clock cycle. Thus, the RAM can be written to and be written from. Looking at Tables 6.2 and 6.3 for Parts 2 and 4, respectively, we see that the hex values can be stored and read successfully. For example in Table 6.2, the initial value read out of the memory was 00 (as we would expect from the write-only mode). We then write the value 0E to address location 06. After the clock is pressed, the value written out of the memory location is 0E. This indicates that we were successfully able to write

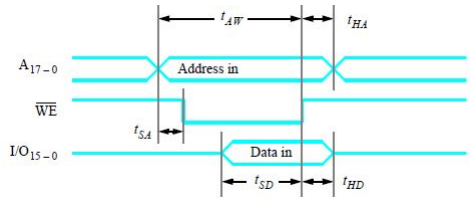

Figure 6.2: SRAM Write Cycle Timing Diagram

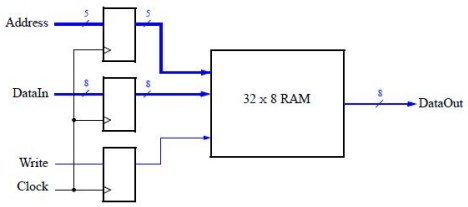

Figure 6.3: RAM Implementation

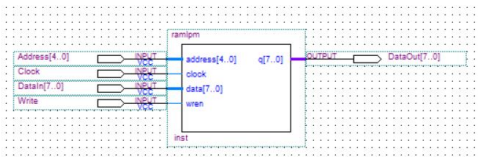

Figure 6.4: Part 1: Block Diagram

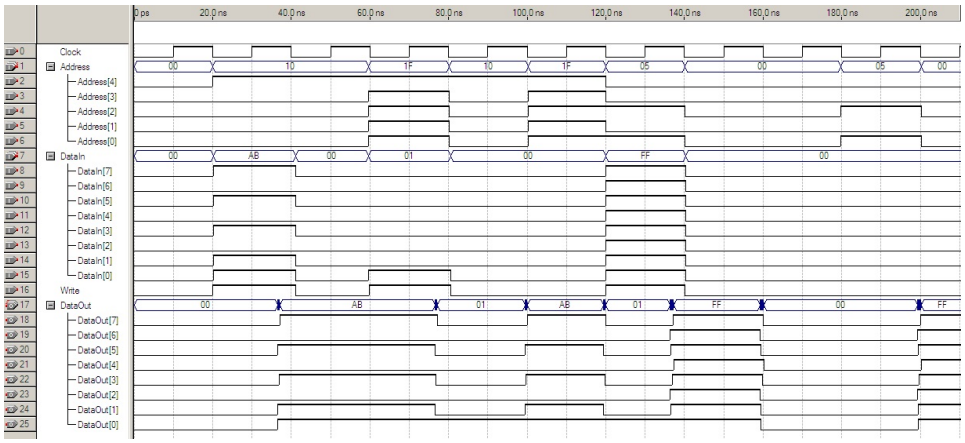

Figure 6.5: Part 1: Waveforms

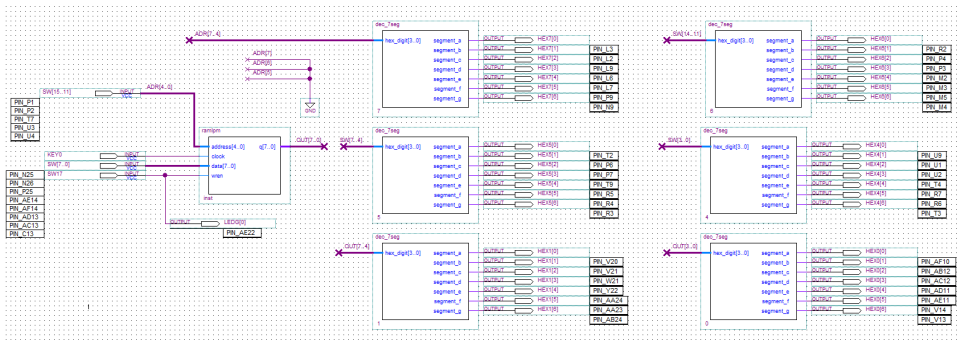

Figure 6.6: Part 2: Block Diagram

Table 6.2: Part 2: Reading/Writing from Memory

Read/Write?	Address Location	Value	Value Written Out of Memory	What is Memory Value After Clock is Pressed?
Write	06	0E	00	0E
Read	1C	F8	0E	00
Read	06	F8	00	0E
Write	1C	F8	00	F8
Read	0F	EA	F8	00
Write	0F	EA	00	EA

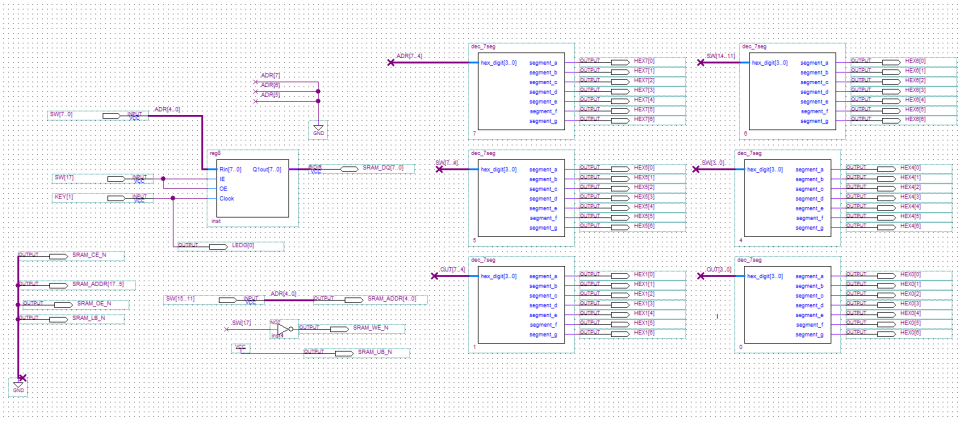

Figure 6.7: Part 4: Block Diagram

the value 0E to address location 06. Then, we switch to read-only mode and observe address location 1C. We set the value to F8 and press the clock to read out a value 00 (since there is no data in this address yet). We then switch back to the address location 06 and read a value of 0E after the clock is pressed. This demonstrates that we were able to read the previously written data in the address location 06. Since the value was set to F8 at the time of reading the data, this also demonstrates the read-only functionality of the memory. The same functionality is seen and demonstrated in Table 6.3 using the SRAM.

6.6 Conclusions

In this practicum, we have seen that the SRAM has a much larger capacity than the other two memory implementations used in Parts 1 and 2. Although, they still have the same functionality, the way each type is set up is different from one another. Using external memory is arguably more complicated than utilizing built-in memory blocks.

Table 6.3: Part 4: Reading/Writing from Memory

Read/Write?	Address Location	Value	Value Written Out of Memory	What is Memory Value After Clock is Pressed?
Write	0D	18	00	18
Write	03	42	18	42
Read	0D	00	18	18
Write	1D	3F	18	3F
Read	03	00	42	42
Write	12	7E	42	7E

7

Practicum 7: DAC and ADC Circuits

7.1 DAC and ADC Circuits

7.1.1 Overview

Digital to Analog Converters (DACs) are used to convert binary signals into analog voltages. Analog to Digital Converters (ADCs) are used to convert analog voltages into binary digital signals. These devices are ubiquitous in audio. For example, an MP3 player has to convert the digital bits of an MP3 into voltages that can be interpreted and played by headphones or speakers. DACs and ADCs are also used in video applications. The increasing presence of HDMI cables, which use digital signals, has called for the use of DACs and ADCs to be used in conjunction with them.

7.1.2 Purpose

The purpose of this section of the practicum is to introduce us to DAC and ADC chips. Through the procedures of this section of the practicum, the operations of the DAC0808 and the ADC0804 chips are shown. The DAC is used to convert digital signals into analog ones, while the ADC is used to convert analog signals into digital ones. Two circuits will be constructed (one for each chip). The DAC circuit will utilize the DAC0808 chip and a 741 op-amp so that we can input digital signals, and measure the output analog voltages. The ADC circuit will have analog voltages ranging from 0.0V to 5.0V in increments of 0.5V. We will measure the digital output in binary and compare this to the expected value.

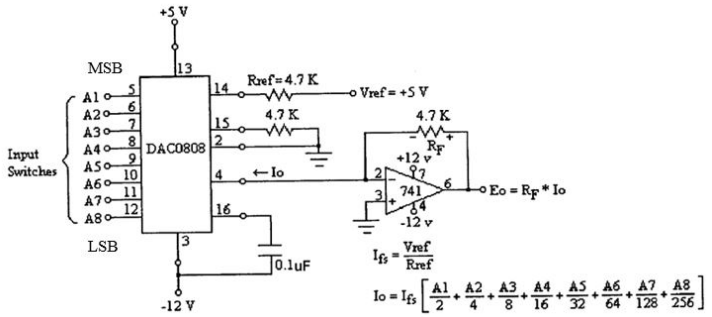

Figure 7.1: Part 1: DAC Circuit Diagram

7.1.3 Procedure

1. Part 1: Digital-to-Analog Converter Circuit with DC Inputs

   (a) First, we constructed the DAC circuit shown in Figure 7.1

   (b) We then applied the inputs by using wires to tie the inputs to 0 or 5V.

   (c) The output voltage values were then measured and recorded for each of the given input values. The offset error was determined and the measurements were compared with this error specification.

2. Part 2: Analog-to-Digital Converter Circuit

   (a) Using Figure 7.2 as a reference, the ADC circuit was constructed.

   (b) The clock frequency at pin 19 of the ADC0804 was measured using an oscilloscope.

   (c) One of the DC outputs of the power supply was used for the DC analog input voltage.

   (d) The push button was pressed to initialize the free-running mode of the ADC. Using a multimeter, the output voltages and values were recorded.

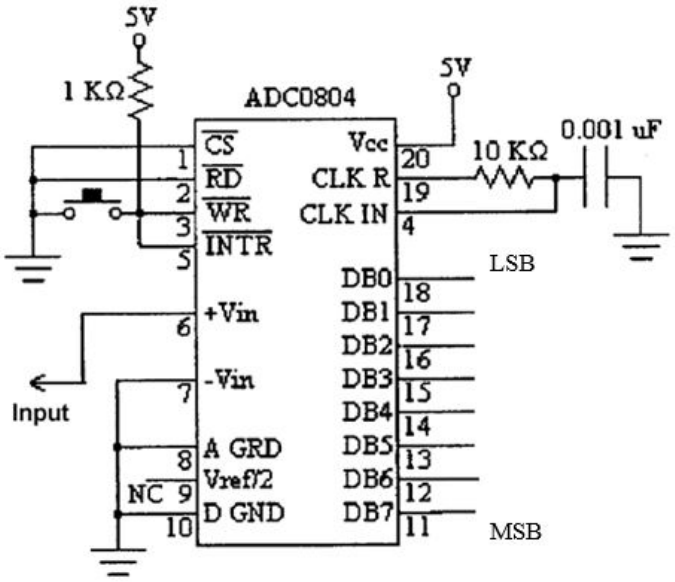

Figure 7.2: Part 2: ADC Circuit Diagram

7.1.4 Calculations, Data, and Graphs

Part 1
Example predicted DAC voltage for digital input 00001111,

$$\begin{aligned} E_0 &= V_{ref} \times (\frac{A_1}{2} + \frac{A_2}{4} + \frac{A_3}{8} + \frac{A_4}{16} + \frac{A_5}{32} + \frac{A_6}{64} + \frac{A_7}{128} + \frac{A_8}{256}) \\ &= 5V \times (\frac{0}{2} + \frac{0}{4} + \frac{0}{8} + \frac{0}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256}) \\ &= 0.2930V \end{aligned}$$

Example percent error calculation for digital input 00001111,

$$\begin{aligned} \text{Percent Error} &= \frac{\text{Predicted Output} - \text{Measured Output}}{\text{Predicted Output}} \times 100\% \\ &= \frac{0.2930 - 0.2900}{0.2930} \times 100\% \\ &= 1.02\% \end{aligned}$$

Part 2

Calculating V_{step},

$$\begin{aligned} V_{step} &= \frac{V_{FS}}{2^8 - 1} \\ &= \frac{5}{2^8 - 1} \\ &= 0.0196 = 19.61mV \end{aligned}$$

Calculating clock frequency f_{clock},

$$\begin{aligned} f_{clock} &= \frac{1}{1.1 \times R \times C} \\ &= \frac{1}{1.1 \times (10 \times 10^3\Omega) \times (0.001 \times 10^{-6}F)} \\ &= 9.0909 \times 10^4 Hz = 90.909kHz \end{aligned}$$

Calculating conversion time,

$$\text{Conversion Rate} = \frac{90.909kHz}{72} = 1262$$

$$\text{Conversion Time} = \frac{1}{\text{Conversion Rate}} = \frac{1}{1262} = 0.00079s$$

Example predicted ADC digital output for analog input 3V,

$$\begin{aligned} \text{Measured output} &= \frac{\text{Analog Input Voltage}}{V_{step}} \\ &= \frac{3}{0.0196} \\ &= 153.0612_{10} \end{aligned}$$

$$\text{Binary equivalent of measured output} = 10011001_2$$

Example percent error calculation for analog input 3V,

$$\begin{aligned} \text{Percent Error} &= \frac{\text{Predicted Output} - \text{Measured Output}}{\text{Predicted Output}} \times 100\% \\ &= \frac{3.0184 - 3}{3.0184} \times 100\% \\ &= 0.61\% \end{aligned}$$

Refer to Appendix for a complete set of calculations.

Table 7.1: Part 1: Predicted and Measured DAC Output Voltages

Input (MSB) → (LSB) A1 A2 A3 A4 A5 A6 A7 A8	Predicted Value of Op-Amp Voltage (E_0)	Measured Value of Op-Amp Voltage (E_0)	Percent Error
1 1 1 1 1 1 1 1	4.9805	4.967	0.27%
0 1 1 1 1 1 1 1	2.4805	2.47	0.42%
0 0 1 1 1 1 1 1	1.2305	1.222	0.69%
0 0 0 1 1 1 1 1	0.6055	0.60	0.91%
0 0 0 0 1 1 1 1	0.2930	0.29	1.02%
0 0 0 0 0 1 1 1	0.1367	0.135	1.24%
0 0 0 0 0 0 1 1	0.0586	0.0565	3.58%
0 0 0 0 0 0 0 1	0.0195	0.0185	5.13%
0 0 0 0 0 0 0 0	0	-0.0025	-

Table 7.2: Part 2: Predicted and Measured ADC Outputs

Input Voltage (V)	Predicted Output DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0	Measured Output DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0	Decimal Equivalent of Measured Binary Value	Multiply Decimal Equivalent by V_{step}	Percent Error
0	0000 0000	0000 0000	0	0	0
0.5	0001 1001	0001 1010	26	0.5096	1.88%
1.0	0011 0011	0011 0100	52	1.0192	1.88%
1.5	0100 1100	0100 1101	77	1.5092	0.61%
2.0	0110 0110	0110 0111	103	2.0188	0.93%
2.5	0111 1111	1000 0000	128	2.5088	0.35%
3.0	1001 1001	1001 1010	154	3.0184	0.61%
3.5	1011 0010	1011 0100	180	3.528	0.79%
4.0	1100 1100	1100 1100	204	3.9984	-0.04%
4.5	1110 0101	1110 0101	229	4.4884	-0.26%
5.0	1111 1111	1111 1111	255	4.998	-0.04%

7.1.5 Discussion of Results

Regarding Part 1, The DAC0808 converter's data sheet specifies that it has an accuracy of ±0.19%. As we can see from Table 7.1, our data have many other outliers. These errors may have arisen due to a faulty DAC0808 chip or the multimeter. It should be noted that for the 00000000 input, we expected an output voltage of 0V, however, we measured -0.0025V. Because we divide by zero in the percent error calculation, this value is undefined. Looking at the percent errors, we notice a trend of increasing percent errors as the digital input decreases. This is due to the fact that the output op-amp voltages become more closely spaced, thus introducing errors. For example, the voltage difference between digital inputs 11111111 and 01111111 is 2.5V, while the voltage difference between digital inputs 00000001 and 00000000 is 0.0195V. As these differences get smaller, the measured value may vary widely.

In terms of Part 2, we see a much smaller percent error overall for the ADC0804. It is worthwhile to mention that for input voltages 4.0V, 4.5V, and 5.0V, the percent errors are negative since the measured outputs in decimal were marginally smaller than what we input. The maximum magnitude of error was 1.88% for input voltages 0.5V and 1.0V, while the minimum percent error was 0% for the input voltage 0.0V. The DAC performed much better than the ADC converter.

7.1.6 Conclusions

The DAC was successfully used to convert digital inputs into an analog voltages, albeit it was somewhat inaccurate. The ADC was used to successfully convert analog voltages into digital outputs. Overall, the ADC converter was much more accurate than the DAC.

The laboratory procedures used here highlight the inevitability of error that arises through noise, human error, the offset error of the chips, and the laboratory instruments.

7.1.7 Prelab

To see the prelab for Part 1, please refer to Figure 7.1 for the schematic and Table 7.1 for the predicted output voltage values. For Part 2, Figure 7.2 is the schematic of the circuit and Table 7.2 contains the predicted output values for the ADC circuit. The calculations section contains the derivation of the frequency of oscillation of the internal Schmitt trigger oscillator, $90.909kHz$, the calculation of the conversion time and step voltage V_{step}.

7.2 Synthesized Source Project

7.2.1 Overview

The synthesized source we create in this project will be able to read in a signal and capture and store its data in SRAM on the the Altera DE2 board. Using the line-in and line-out interface in conjunction with the WM8731 Audio CODEC chip on the DE2 board, we can utilize the DACs and ADCs to easily convert analog signals to digital ones and digital signals to analog ones. Then, not only will it be able to read out the original data, but it will be able to manipulate its amplitude digitally. In other words, it will multiply or divide the original signal's amplitude by integer values using the DE2 board's switches. This can be observed by using Signal Tap II in Altera Quartus II. In general, synthesized sources are frequently used to reconstruct signals, modify the amplitude (as we do in this project), modify the frequency, or even the phase.

7.2.2 Purpose

The purpose of this practicum was to give us exposure to a practical application of SRAM used in conjunction with DAC and ADC converters. In this project, we record the 128 data points of a signal and store it in the SRAM on the Altera DE2 board. We then read this data and modify the amplitude by multiplying or dividing it by integer values using the switches on the DE2. This was done in three phases. The first part consisted of measuring the frequency of signals and ensuring that the data in was being recorded in the appropriate number of data bits and ensuring that the signals the device captures are easy to read. The second phase, we designed the synthesized source using CODEC and SRAM using the ALLPASS circuit from the first phase. In the third and final phase of the project, we tested, debugged, simulated, and verified the synthesized source by manipulating the data to produce a wave with larger and smaller amplitudes than the original source wave.

7.2.3 Procedure

1. Phase 1: Measurements with ADC2DAC Circuits

   (a) We measured the frequency response of the ADC2DAC circuit configured as an ALL PASS circuit at the following frequencies: 2kHz and 10kHz. Using a $1V_{pp}$ sine wave signal,

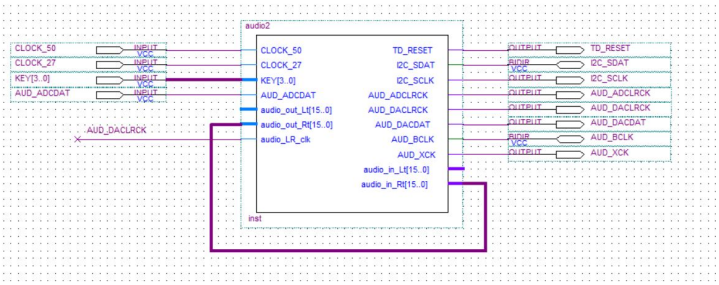

Figure 7.3: Phase 1: Allpass Diagram

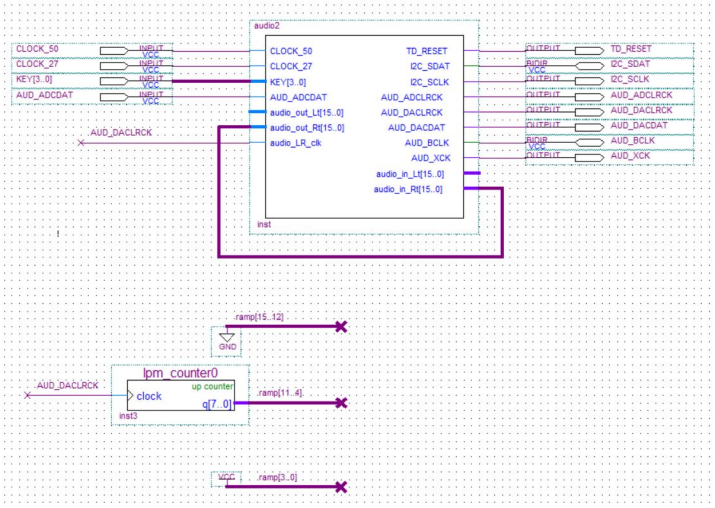

Figure 7.4: Phase 1: Allpass and Linear Ramp Generator

the input and output waveforms were captured using an oscilloscope

(b) Using a $1V_{pp}$, 1kHz square wave signal and Signal Tap II, the digital data of the waveform was captured using the ADC2DAC circuit configured as an ALL PASS circuit. This data was plotted in MATLAB.

(c) An input signal was generated with the following specifications: $1V_{pp}$, linear ramp signal with a frequency of 187.5 Hz fed to the right channel of the ADC. Using the ADC2DAC configured as both an ALL PASS circuit and a Linear Ramp Generator was used to capture and record both the left and right channels of the DAC circuit.

2. Phase 2: Design of a Synthesized Source using CODEC and SRAM

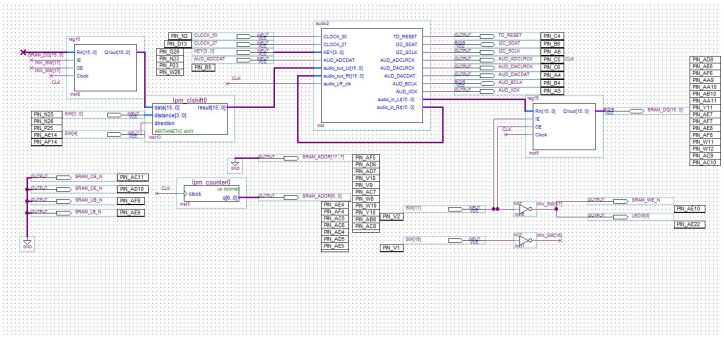

Figure 7.5: Phase 2: Block Diagram of our Design with CODEC and SRAM

(a) This phase of the project is set to design a 16-bit synthesized source that can display a function composed of 128 data points. The synthesized source should be able to display the function at the original frequency and amplitude and three other amplitudes, as per our specifications.

(b) The SRAM chip on the DE2 board was used to store and play back the 16-bit waveform data. The WM8731 Audio CODEC chip on the Altera DE2 board was utilized to support the line-in and line-out interfaces.

(c) The switches on the Altera DE2 board were used to control the amplitude modulation of the record/play-back circuit.

(d) A design proposal was made that consisted of a detailed block diagram and description of functionality.

3. Phase 3: Demo of a Synthesized Source using CODEC and SRAM

(a) With the completed design, we demonstrate that the synthesized source composed of 128 data points at the original frequency and amplitude and three other amplitudes can be generated.

(b) This functionality was recorded via an oscilloscope. In other words, the oscilloscope was used to capture the output waveforms at the original frequency and amplitude as well as three different amplitudes.

(c) Using Altera Quartus II's Signal Tap II, the activity in the circuit was captured during the record mode and for two particular play-back mode scenarios.

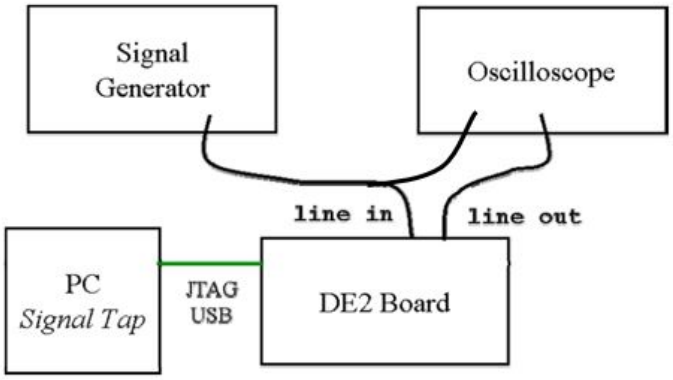

Figure 7.6: Phase 3: Measurement Setup

7.2.4 Calculations, Data, and Graphs

See Figures 7.6-7.20 for oscilloscope readings, the MATLAB plot, and the Signal Tap II readings from the synthesized source.

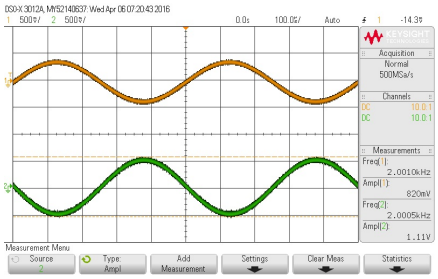

Figure 7.7: Phase 1: 2kHz Sine Wave (Green = Input; Yellow = Output)

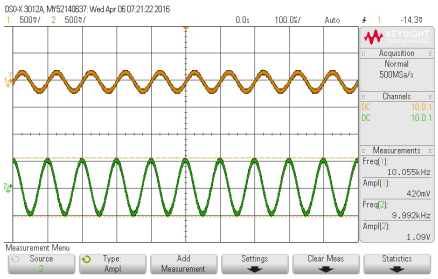

Figure 7.8: Phase 1: 10kHz Sine Wave (Green = Input; Yellow = Output)

7.2.5 Discussion of Results

As demonstrated through all three phases, we were successfully able to construct a synthesized source that can display a function composed of 128 data points. We were then able to modify the amplitude by multiplying or dividing the signal by an integer value using the switches on the Altera DE2 board. Specifically, we were able to read out the original signal from the synthesized source and multiply and divide its amplitude by 2 and 4. As we see from the Signal Tap II readings, when we divide the input signal by 2 or 4, the amplitude goes down (i.e. there are fewer bits representing the most significant ones). The

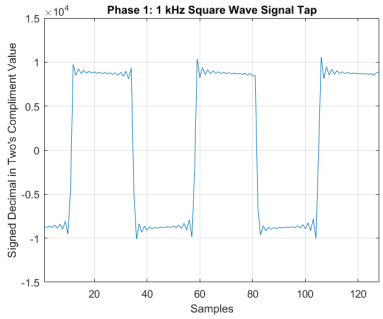

Figure 7.9: Phase 1: MATLAB Plot of Square Wave Using Data from Signal Tap II

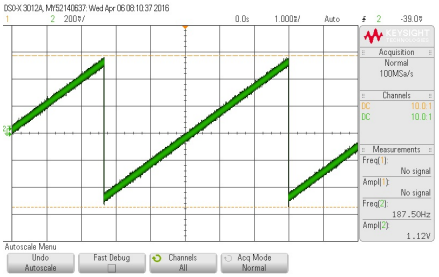

Figure 7.10: Phase 1: 187.5 Hz Ramp Input

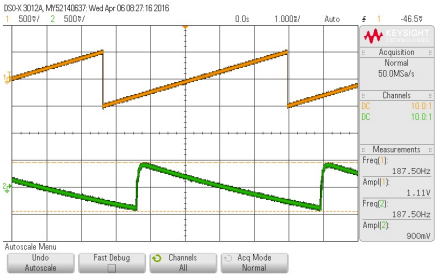

Figure 7.11: Phase 1: Ramp Input with Right Channel (Input = Yellow; Right Channel = Green)

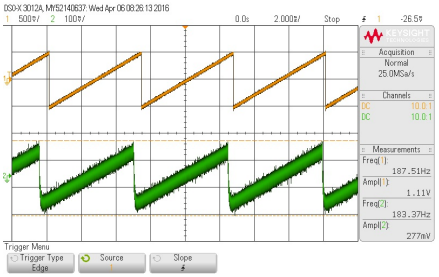

Figure 7.12: Phase 1: Ramp Input with Left Channel (Input = Yellow; Left Channel = Green)

converse is true for multiplying by 2 and 4. We observe that there are more bits being used in the more significant ones.

7.2.6 Conclusions

We have successfully been able to use the Altera DE2 board to record a signal, store its information, and read it out. Additionally, we have been able to modify its amplitude by multiplying or dividing the original signal's amplitude by integer amounts. All three phases of the project were completed successfully.

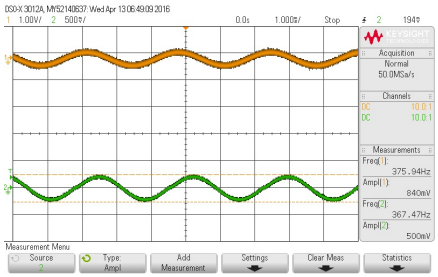

Figure 7.13: Phase 3: 500mV Input with No Amplitude Modulation on SRAM output (Input = Yellow; SRAM Left Channel = Green)

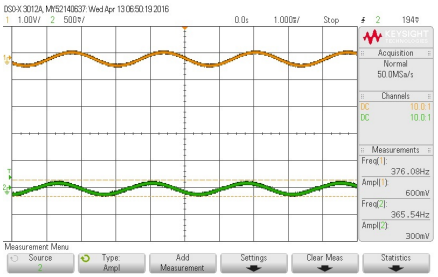

Figure 7.14: Phase 3: 500mV Input with Amplitude Divided by 2 on SRAM output (Input = Yellow; SRAM Left Channel = Green)

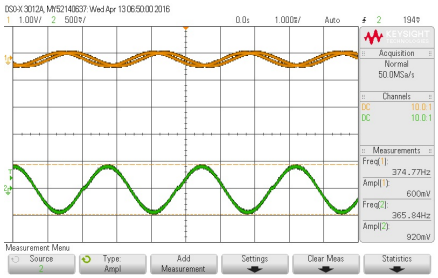

Figure 7.15: Phase 3: 500mV Input with Amplitude Multiplied by 2 on SRAM output (Input = Yellow; SRAM Left Channel = Green)

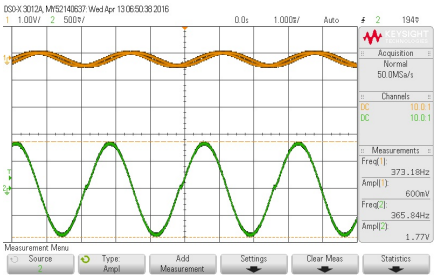

Figure 7.16: Phase 3: 500mV Input with Amplitude Multiplied by 4 on SRAM output (Input = Yellow; SRAM Left Channel = Green)

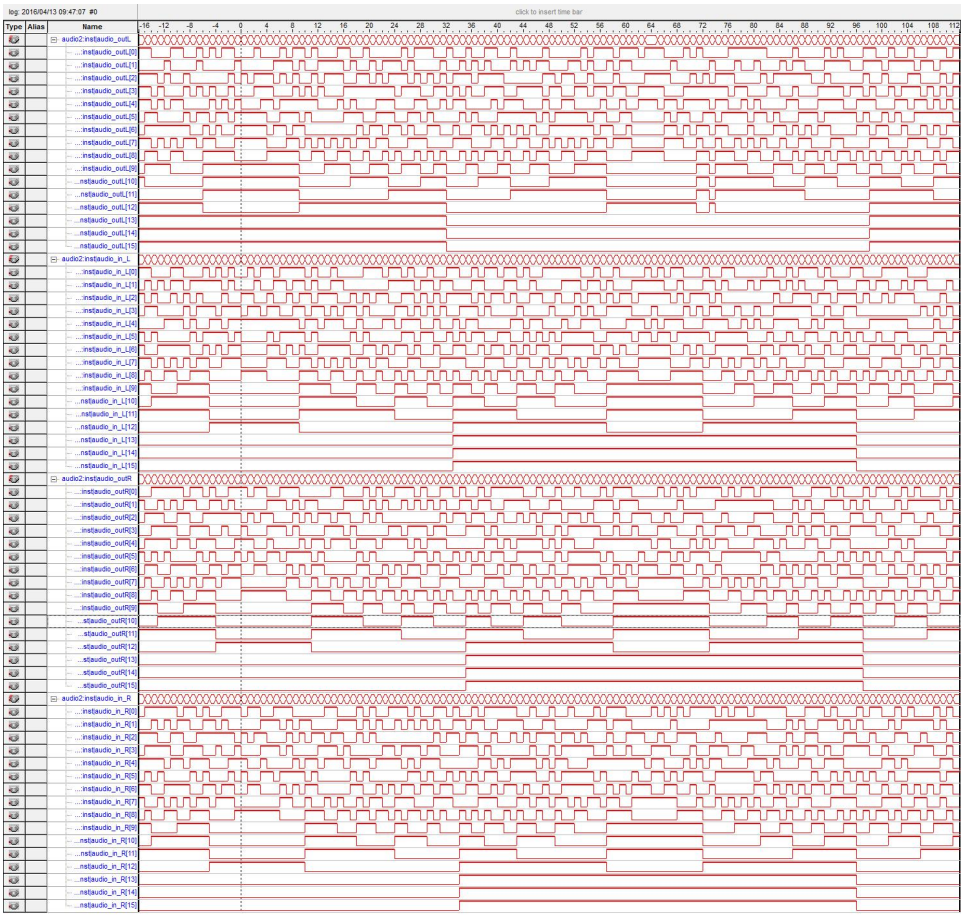

Figure 7.17: Phase 3: Signal Tap II: Reading 375 Hz & 500mV Amplitude Input Signal

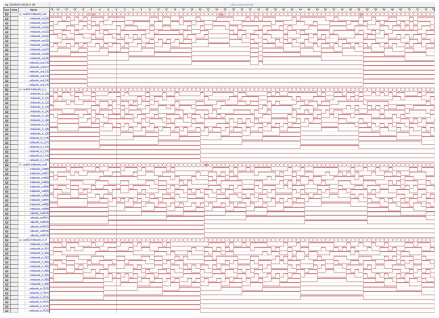

Figure 7.18: Phase 3: Signal Tap II: Dividing Input Signal by 4

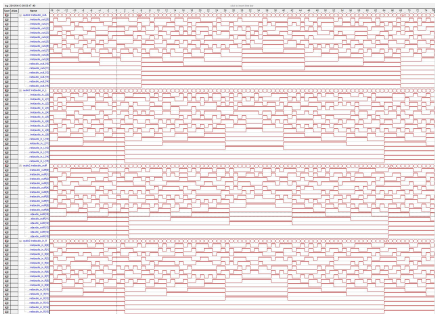

Figure 7.19: Phase 3: Signal Tap II: Dividing Input Signal by 2

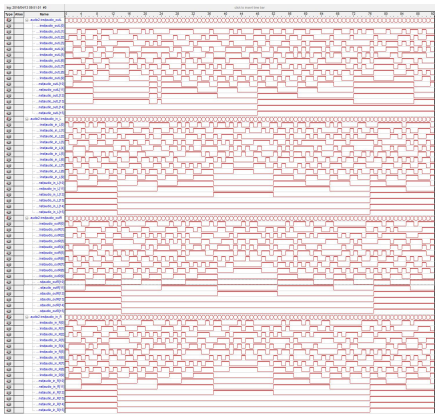

Figure 7.20: Phase 3: Signal Tap II: Multiplying Input Signal by 2

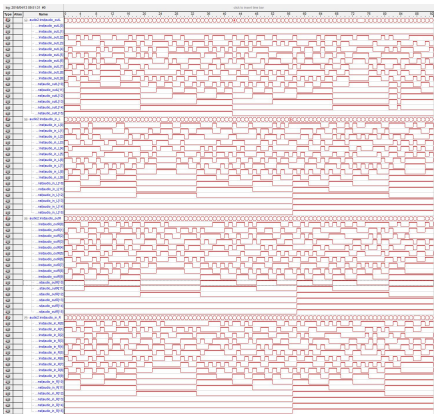

Figure 7.21: Phase 3: Signal Tap II: Multiplying Input Signal by 4

8

Practicum 8: Finite State Machine Project

8.1 Overview

A finite state machine (FSM) is an abstract representation of a system that consists of a finite set of states. They are used to design programs or sequential logic circuits. As the name finite state machine hints at, there exist only a finite number of states in a given finite state machine.

8.2 Purpose

The purpose of this practicum is to gain an understanding of how to implement a finite state machine in VHDL code. Finite state machines are widely used in designing hardware digital systems, software engineering, network protocols, and linguistics, to name a few.

8.3 Procedure

Prelab

1. A state table for the finite state machine shown in Figure 8.1 was made.

2. The VHDL code to implement the finite state machine was written using the toggle switch SW0 on the Altera DE2 board as an active-low synchronous reset input, switch SW1 as the w input, and the pushbutton KEY0 as the clock input which is applied manually. The green LED LEDG0 is used as the output z, and the state ouputs were assigned to the red LEDs LEDR8 to LEDR0.

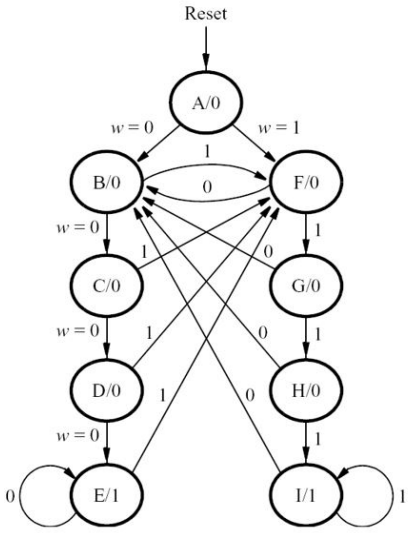

Figure 8.1: Finite State Machine State Diagram

In Lab

1. A new Altera Quartus II project was created with the Cyclone II EP2C35F672C6 as the target chip.

2. Next, we added the VHDL file that implements the finite state machine shown in Figure 8.1 to the project workspace. Using The pins on the FPGA were assigned to connect the switches and LEDs. From the VHDL code, we generated a state diagram and state table of the state machine.

3. The circuit's behavior was then simulated.

4. Then, the circuit was uploaded to the FPGA chip on the Cyclone II board. The functionality of the finite state machine circuit was tested by applying the input sequences and observing the output LEDs including LEDR8 to LEDR0 and the output LEDG0.

8.4 Calculations, Data, and Graphs

Refer to Figures 8.3 and 8.4 for the timing analysis and the Altera Quartus II-generated state diagram and state table.

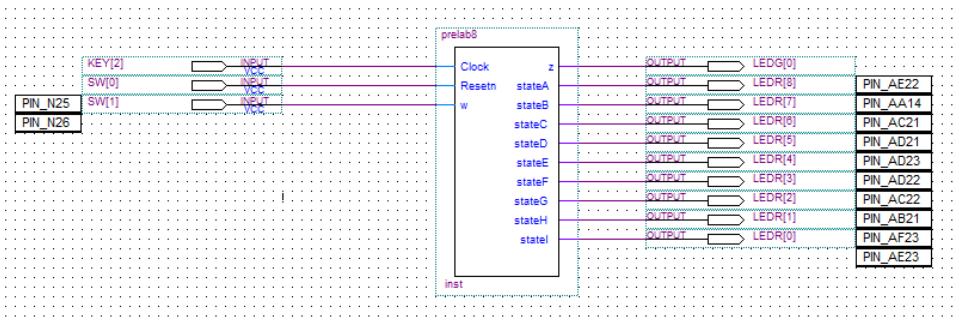

Figure 8.2: Altera Quartus II Block Diagram

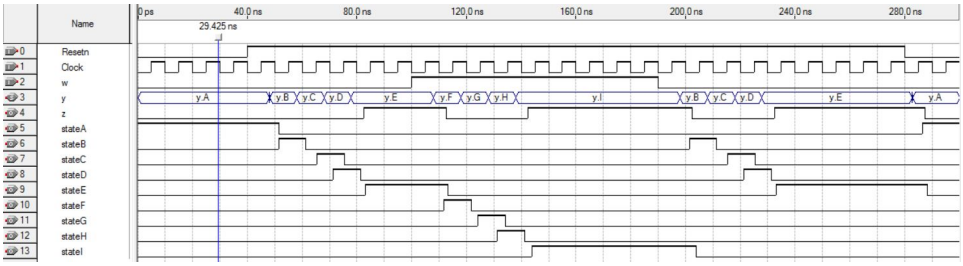

Figure 8.3: Timing Analysis

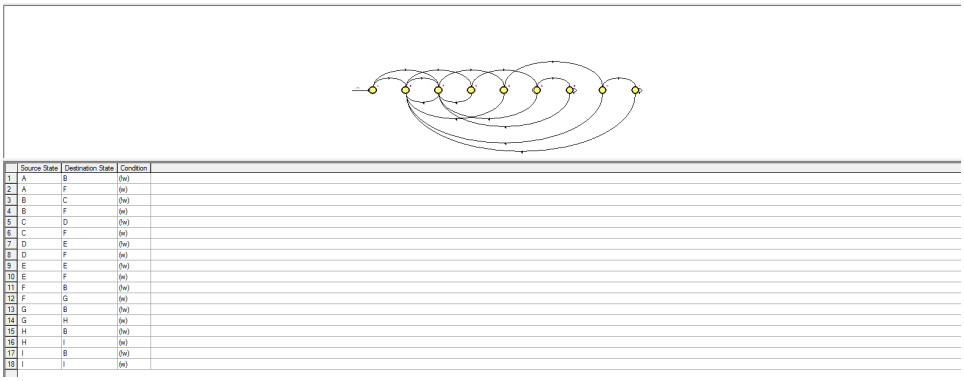

Figure 8.4: State Diagram and State Table

8.5 Discussion of Results

As we see in Figure 8.3, when Resetn is set to LOW the state remains at state A. When Resetn is set to HIGH and the clock begins its next cycle, the state switches from state A to state B since w = 0. As w remains set to 0, the state goes from state B to state C to state D to state E. Output z becomes 1 and the state remains at E while w is set to 0. Then, w is set equal to 1 and the state goes from state E to state F. Output z becomes 0 again. Then, as w remains 1, the state transitions from state F to state G to state H to state I. At state I the output z becomes 1. The current state remains at I and the output remains 1 until w changes from 1 to 0. At this point, the current state becomes B and the output 0. We cycle through the states B, C, D, and E like before and when reset becomes low we go back to state A. This behavior is what we would expect from the state diagram in Figure 8.1 and Figure 8.4.

After uploading the design to the FPGA chip on the Altera DE2 board, the input sequences were applied with the pushbutton as the clock. After cycling through the same sequence of states as described in the previous paragraph and timing analysis, we concluded that our finite state machine was functioning properly. The outputs for each state were displayed correcty on the green led LEDG0. In addition, the transition between states and thus proper states were displayed on the red LEDs on the DE2 board.

8.6 Conclusions

The finite state machine in Figure 8.1 was successfully designed, written in VHDL code, simulated, uploaded, and tested on the Altera DE2 board. The state diagram and state table that Altera Quartus II generated matches the state table we have constructed in the prelab.

8.7 Prelab

Refer to appendix section 10.2 for the VHDL code used to implement the finite state machine. The state table for the finite state machine is shown in Table 8.1.

Table 8.1: State Table

Present State	Next State		Output z
	w = 0	w = 1	
A	B	F	0
B	C	F	0
C	D	F	0
D	E	F	0
E	E	F	1
F	B	G	0
G	B	H	0
H	B	I	0
I	B	I	1

9

Practicum 9: Sonar Sensor Project

9.1 Overview

Using sonar technology, one can measure the position of an object. Sonar use sound waves that propagate to and from objects to detect how far away an object is with generally high precision. Once we bring time into the equation, we can determine the velocity of an object from the change in position.

9.2 Purpose

The purpose of this practicum is to design a sonar sensor circuit using the Altera DE2 board that can measure distances from 2-10 feet with 1% resolution. In addition to measuring the position of an object, the design must also measure the velocity as well.

9.3 Procedure

1. We begun by choosing the number of bits we wanted to use to store each data point. Specifically, we chose N = 16 bits. The range of the sonar sensor was set at 2 feet to 10 feet with 1% accuracy (1.2 inches).

2. Based on the number of bits used by the sonar sensor circuit, we determined the new frequency for the system to run at. This new frequency ensures that we collect enough data points. The calculations are shown in detail in the Calculations, Data, and Graphs section.

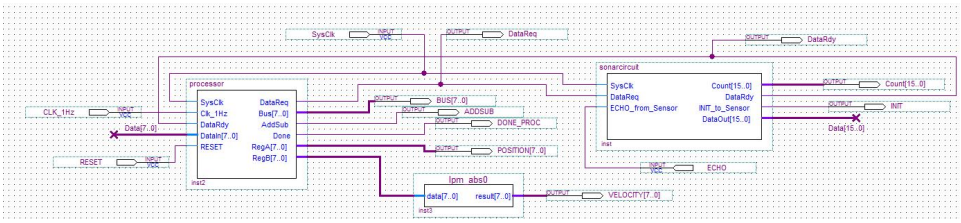

Figure 9.1: Sonar project 1 block diagram used for simulation only

3. The next step was to modify the existing Altera Quartus II simulation project to allow the correct number of bits. Refer to the Appendix to see how the VHDL code was modified. A new clock with the correct frequency was created. The VHDL code is shown in the Appendix as well.

4. Finally, the sonar sensor circuit was uploaded to the Altera DE2 board and position measurements were taken from two feet to 10 feet in increments of a foot. Our measurements are shown in Table 9.1. To ensure the precision of the device was what it was supposed to be, a measurement at 10 feet was taken. In hexadecimal, this value should have been 77, 78, or 79 (with 78 being the value with 0 error). In other words, the measurement at 10 feet away from the sensor should be ± 1 inch from the true value of 120 inches (or 78 in hexadecimal). This procedure ensures that our clock frequency and overall design maintains the 1% resolution parameter.

9.4 Calculations, Data, and Graphs

Calculations

$$v_s = 1.082 \text{ feet/ms} = 0.9 \text{ ms/foot}, \tag{9.1}$$

where v_s denotes the speed of sound.

Since we specified that our design will use 16 bits, our clock divides by

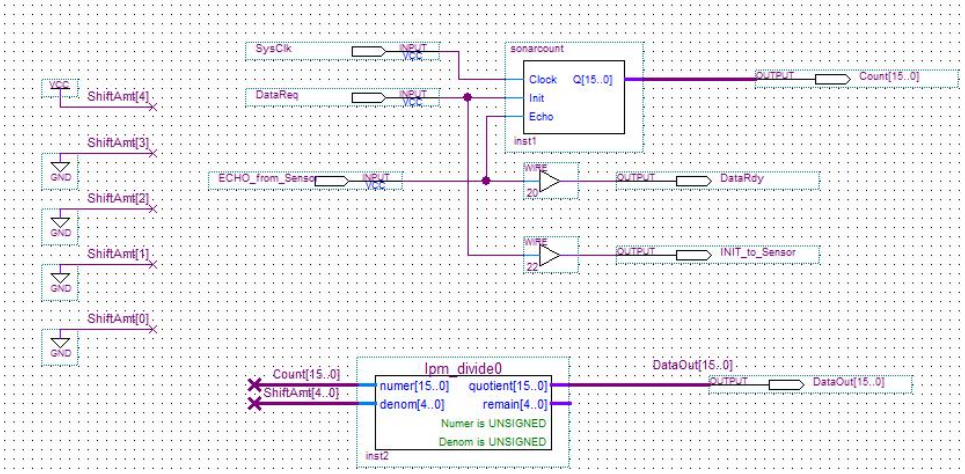

Figure 9.2: Sonar circuit block diagram used for both simulation and DE2 implementation

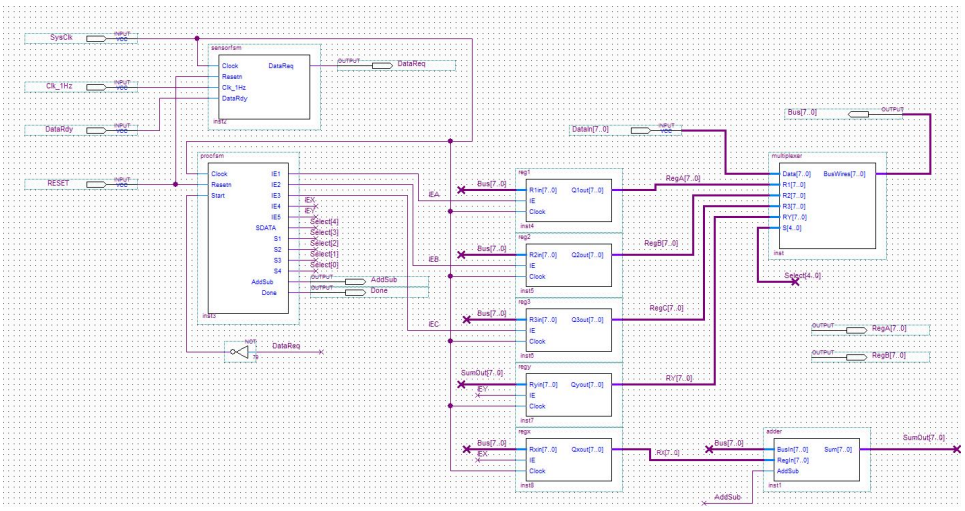

Figure 9.3: Processor block diagram used for both simulation and DE2 implementation

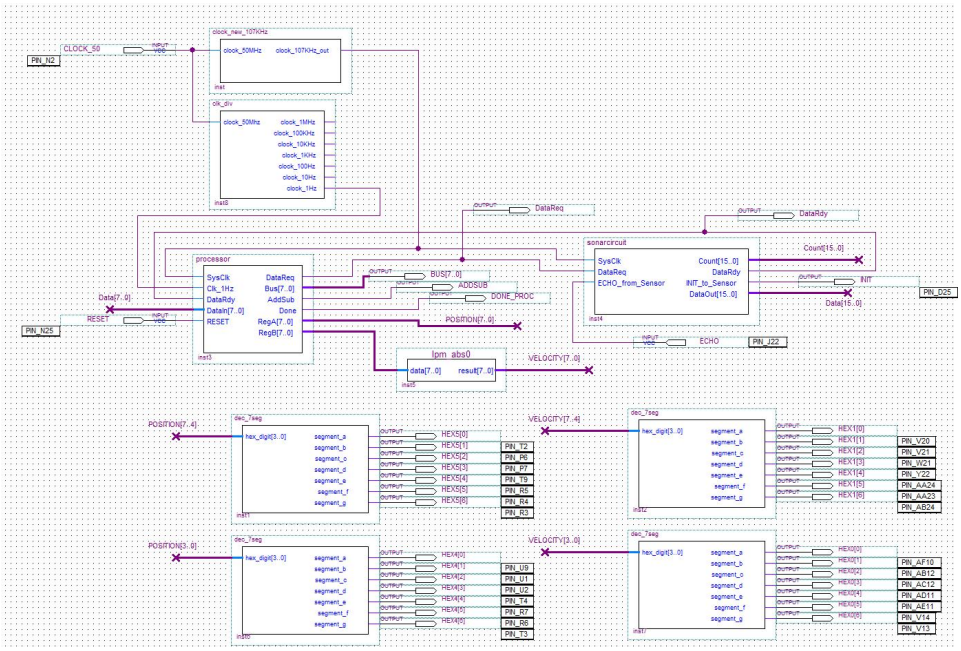

Figure 9.4: Sonar project 2 block diagram used for DE2 implementation

16. Hence,

$$
\begin{aligned}
Range &= \frac{Count}{N} \\
Count &= range \times N \\
Count &= 120 \text{ inches} \times 16 \text{ bits} \\
Count &= 1920
\end{aligned}
$$

And then we solve for the system clock frequency f_{sys}

$$
\begin{aligned}
\frac{0.9ms}{1ft} \times 2 \times 120in &= 18ms \times f_{sys} \\
18ms \times f_{sys} &= 1920 \\
f_{sys} &= 106.666 \text{ kHz}
\end{aligned}
$$

Data

See Figure 9.5 for the simulation waveform and Table 9.1 for the sonar sensor data.

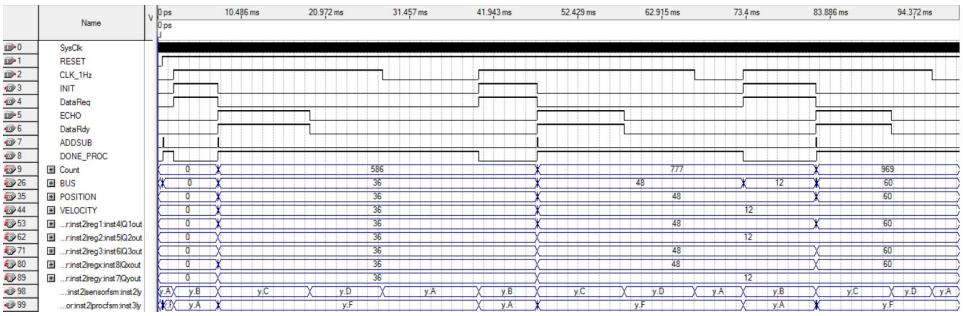

Figure 9.5: Simulation Waveform

Table 9.1: Sonar Sensor Data Read From DE2 Board

Distance From Sensor (feet)	Distance From Sensor (inches)	Hex Value (inches)	Decimal Value (inches)	Error Margin (inches)
2	24	17	23	-1
3	36	23	35	-1
4	48	2F	47	-1
5	60	3B	59	-1
6	72	47	71	-1
7	84	53	83	-1
8	96	5F	95	-1
9	108	6B	107	-1
10	120	78	120	0

9.5 Discussion of Results

From Table 9.1, we see that at 10 feet our sonar circuit was able to measure 10 feet with no error. However, our design underestimated each interval from 2-9 feet by one inch. This error may have arisen from placing the black board at the precise locations. However, because it was so consistent it is most likely due to the value we divide by in the textitclock_new_107KHz.vhd file. To make it more accurate, we would have to decrease this value by a margin since increasing it would make our measured distances even smaller. However, the margins of error we were observing are acceptable since they meet the $\pm$ 1 inch discrepancy requirement.

9.6 Conclusions

At a clock frequency of 106.666 kHz we were able to observe the 1% resolution and thus ± 1 inch accuracy while using 16 bits. Using a sonar proximity sensor, an Altera DE2 board, and Altera Quartus II, we have constructed a circuit that uses 16 bits to measure the distance and velocity of an object.

10

Appendix: Code and Calculations

10.1 Practicum 5

10.1.1 CLK_DIV_VHDL

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY clk_div IS

   PORT
   (
      clock_50Mhz      : IN  STD_LOGIC;
      clock_1MHz         : OUT STD_LOGIC;
      clock_100KHz       : OUT STD_LOGIC;
      clock_10KHz        : OUT STD_LOGIC;
      clock_1KHz         : OUT STD_LOGIC;
      clock_100Hz        : OUT STD_LOGIC;
      clock_10Hz         : OUT STD_LOGIC;
      clock_1Hz          : OUT STD_LOGIC);

END clk_div;

ARCHITECTURE a OF clk_div IS

   SIGNAL  count_1Mhz: STD_LOGIC_VECTOR(6 DOWNTO 0);
   SIGNAL  count_100Khz, count_10Khz, count_1Khz :
       STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL  count_100hz, count_10hz, count_1hz : STD_LOGIC_VECTOR(2
       DOWNTO 0);
```

```
   SIGNAL clock_1Mhz_int, clock_100Khz_int, clock_10Khz_int,
       clock_1Khz_int: STD_LOGIC;
   SIGNAL  clock_100hz_int, clock_10Hz_int, clock_1Hz_int :
       STD_LOGIC;
   SIGNAL clock_1Mhz_reg, clock_100Khz_reg, clock_10Khz_reg,
       clock_1Khz_reg: STD_LOGIC;
   SIGNAL  clock_100hz_reg, clock_10Hz_reg, clock_1Hz_reg :
       STD_LOGIC;
BEGIN
   PROCESS
   BEGIN
        clock_1Mhz <= clock_1Mhz_reg;
        clock_100Khz <= clock_100Khz_reg;
        clock_10Khz <= clock_10Khz_reg;
        clock_1Khz <= clock_1Khz_reg;
        clock_100hz <= clock_100hz_reg;
        clock_10hz <= clock_10hz_reg;
        clock_1hz <= clock_1hz_reg;
-- Divide by 50
      WAIT UNTIL clock_50Mhz'EVENT and clock_50Mhz = '1';
         IF count_1Mhz < 49 THEN
            count_1Mhz <= count_1Mhz + 1;
         ELSE
            count_1Mhz <= "0000000";
         END IF;
         IF count_1Mhz < 25 THEN
            clock_1Mhz_int <= '0';
         ELSE
            clock_1Mhz_int <= '1';
         END IF;

      -- Ripple clocks are used in this code to save prescalar
          hardware
      -- Sync all clock prescalar outputs back to master clock signal
         clock_1Mhz_reg <= clock_1Mhz_int;
         clock_100Khz_reg <= clock_100Khz_int;
         clock_10Khz_reg <= clock_10Khz_int;
         clock_1Khz_reg <= clock_1Khz_int;
         clock_100hz_reg <= clock_100hz_int;
         clock_10hz_reg <= clock_10hz_int;
         clock_1hz_reg <= clock_1hz_int;
   END PROCESS;

      -- Divide by 10
   PROCESS
   BEGIN
      WAIT UNTIL clock_1Mhz_reg'EVENT and clock_1Mhz_reg = '1';
         IF count_100Khz /= 4 THEN
            count_100Khz <= count_100Khz + 1;
```

```
      ELSE
         count_100khz <= "000";
         clock_100Khz_int <= NOT clock_100Khz_int;
      END IF;
END PROCESS;

   -- Divide by 10
PROCESS
BEGIN
   WAIT UNTIL clock_100Khz_reg'EVENT and clock_100Khz_reg = '1';
      IF count_10Khz /= 4 THEN
         count_10Khz <= count_10Khz + 1;
      ELSE
         count_10khz <= "000";
         clock_10Khz_int <= NOT clock_10Khz_int;
      END IF;
END PROCESS;

   -- Divide by 10
PROCESS
BEGIN
   WAIT UNTIL clock_10Khz_reg'EVENT and clock_10Khz_reg = '1';
      IF count_1Khz /= 4 THEN
         count_1Khz <= count_1Khz + 1;
      ELSE
         count_1khz <= "000";
         clock_1Khz_int <= NOT clock_1Khz_int;
      END IF;
END PROCESS;

   -- Divide by 10
PROCESS
BEGIN
   WAIT UNTIL clock_1Khz_reg'EVENT and clock_1Khz_reg = '1';
      IF count_100hz /= 4 THEN
         count_100hz <= count_100hz + 1;
      ELSE
         count_100hz <= "000";
         clock_100hz_int <= NOT clock_100hz_int;
      END IF;
END PROCESS;

   -- Divide by 10
PROCESS
BEGIN
   WAIT UNTIL clock_100hz_reg'EVENT and clock_100hz_reg = '1';
      IF count_10hz /= 4 THEN
         count_10hz <= count_10hz + 1;
      ELSE
```

```
            count_10hz <= "000";
            clock_10hz_int <= NOT clock_10hz_int;
         END IF;
   END PROCESS;

      -- Divide by 10
   PROCESS
   BEGIN
      WAIT UNTIL clock_10hz_reg'EVENT and clock_10hz_reg = '1';
         IF count_1hz /= 4 THEN
            count_1hz <= count_1hz + 1;
         ELSE
            count_1hz <= "000";
            clock_1hz_int <= NOT clock_1hz_int;
         END IF;
   END PROCESS;

END a;
```

10.1.2 TIMER_COUNT VHDL

If the reset button is pressed, we set count, count_temp, and count_final to 0. Otherwise, if there is a rising clock edge and timer_2x = 1, we increment count by 1 and set count_temp to count. Otherwise if timer_2x = 0, we set count to 0. If count_temp is smaller than count_final, we leave count_final alone and set it equal to itself. Otherwise, count_temp is larger than count_final and we set count_final to count_temp. This ensures that we only measure the maximum period. If we go from a small capacitor to a larger one and then a smaller one, the final reading of the display will read the period of the larger one even though the smaller capacitor is in the circuit. At the end of the cascading if statements, we set count_out to count_final.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY Timer_Count IS

   PORT(resetN                : IN  STD_LOGIC;
       timer_2x               : IN  STD_LOGIC;
       clock                  : IN   STD_LOGIC;
       count_out              : OUT    STD_LOGIC_VECTOR (7 DOWNTO
           0));
```

```
END Timer_Count;

ARCHITECTURE a OF Timer_Count IS
   SIGNAL count:      STD_LOGIC_VECTOR (7 DOWNTO 0);
   SIGNAL count_temp:    STD_LOGIC_VECTOR (7 DOWNTO 0);
   SIGNAL count_final:   STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN
   PROCESS(resetN, clock)
   BEGIN
      IF resetN = '0' then
         count <= "00000000";
         count_temp <= "00000000";
         count_final <= "00000000";
      ELSIF(clock'EVENT AND clock = '1') then
         IF timer_2x = '1' then
            count <= count + 1;
            count_temp <= count;
         ELSE
            count <= "00000000";
         END IF;

         IF count_temp < count_final then
            count_final <= count_final;
         ELSE
            count_final <= count_temp;
         END IF;
      END IF;
   END PROCESS;

   count_out <= count_final;
END a;
```

10.1.3 Adder VHDL

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


ENTITY adder IS
   PORT (  Offset     : IN    STD_LOGIC_VECTOR(7 DOWNTO 0) ;
         Count    : IN    STD_LOGIC_VECTOR(7 DOWNTO 0) ;
         AddSub     : IN    STD_LOGIC ;
         Sum        : OUT      STD_LOGIC_VECTOR(7 DOWNTO 0)) ;
END adder ;
```

```
ARCHITECTURE Behavior OF adder IS

BEGIN
   WITH AddSub SELECT
      Sum <=  Count + Offset WHEN '0',
            Count - Offset WHEN OTHERS;
   END Behavior ;
```

10.1.4 SEVEN_SEG_DISPLAY VHDL

Below is the code with numbers 10-89 omitted to conserve space.

```
library IEEE;
use  IEEE.STD_LOGIC_1164.all;
use  IEEE.STD_LOGIC_ARITH.all;
use  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY Seven_Seg_Display IS
   PORT(Dec_digit   : IN  STD_LOGIC_VECTOR(7 downto 0);
       segment_a1, segment_b1, segment_c1, segment_d1, segment_e1,
       segment_f1, segment_g1, segment_a0, segment_b0, segment_c0,
       segment_d0, segment_e0, segment_f0, segment_g0 : out
           std_logic);
END Seven_Seg_Display;

ARCHITECTURE a OF Seven_Seg_Display IS
   SIGNAL segment_data : STD_LOGIC_VECTOR(13 DOWNTO 0);
BEGIN
PROCESS (Dec_digit)
-- Decimal to Dual 7 Segment Decoder for LED Display (0 to 99)
BEGIN
CASE Dec_digit IS
-- Decimal 0 to 9
        WHEN "00000000" =>
            segment_data <= "11111101111110";
        WHEN "00000001" =>
            segment_data <= "11111100110000";
        WHEN "00000010" =>
            segment_data <= "11111101101101";
        WHEN "00000011" =>
            segment_data <= "11111101111001";
        WHEN "00000100" =>
            segment_data <= "11111100110011";
        WHEN "00000101" =>
            segment_data <= "11111101011011";
        WHEN "00000110" =>
            segment_data <= "11111101011111";
        WHEN "00000111" =>
```

```
            segment_data <= "11111101110000";
        WHEN "00001000" =>
            segment_data <= "11111101111111";
        WHEN "00001001" =>
            segment_data <= "11111101111011";
...
...
...
-- Decimal 90 to 99
        --90
        WHEN "01011010" =>
            segment_data <= "11110111111110";
        --91
        WHEN "01011011" =>
            segment_data <= "11110110110000";
        --92
        WHEN "01011100" =>
            segment_data <= "11110111101101";
        --93
        WHEN "01011101" =>
            segment_data <= "11110111111001";
      --94
        WHEN "01011110" =>
            segment_data <= "11110110110011";
        --95
        WHEN "01011111" =>
          segment_data <= "11110111011011";
      --96
        WHEN "01100000" =>
          segment_data <= "11110111011111";
          --97
        WHEN "01100001" =>
          segment_data <= "11110111011111";
      --98
        WHEN "01100010" =>
          segment_data <= "11110111111111";
      --99
        WHEN "01100011" =>
          segment_data <= "11110111111011";
-- If not between 0 and 99, then error code UU
       WHEN OTHERS =>
            segment_data <= "01111100111110";
END CASE;
END PROCESS;

-- extract segment data and LED driver is inverted
segment_a1 <= NOT segment_data(13);
segment_b1 <= NOT segment_data(12);
segment_c1 <= NOT segment_data(11);
```

```
segment_d1 <= NOT segment_data(10);
segment_e1 <= NOT segment_data(9);
segment_f1 <= NOT segment_data(8);
segment_g1 <= NOT segment_data(7);
segment_a0 <= NOT segment_data(6);
segment_b0 <= NOT segment_data(5);
segment_c0 <= NOT segment_data(4);
segment_d0 <= NOT segment_data(3);
segment_e0 <= NOT segment_data(2);
segment_f0 <= NOT segment_data(1);
segment_g0 <= NOT segment_data(0);

END a;
```

10.1.5 DEC_7SEG VHDL

Below is the code used to turn binary into a readable seven segment display output.

```
library IEEE;
use  IEEE.STD_LOGIC_1164.all;
use  IEEE.STD_LOGIC_ARITH.all;
use  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY dec_7seg IS
   PORT(hex_digit   : IN  STD_LOGIC_VECTOR(3 downto 0);
       segment_a, segment_b, segment_c, segment_d, segment_e,
       segment_f, segment_g : out std_logic);
END dec_7seg;

ARCHITECTURE a OF dec_7seg IS
   SIGNAL segment_data : STD_LOGIC_VECTOR(6 DOWNTO 0);
BEGIN
PROCESS (Hex_digit)
-- HEX to 7 Segment Decoder for LED Display
BEGIN
CASE Hex_digit IS
        WHEN "0000" =>
            segment_data <= "1111110";
        WHEN "0001" =>
            segment_data <= "0110000";
        WHEN "0010" =>
            segment_data <= "1101101";
        WHEN "0011" =>
            segment_data <= "1111001";
        WHEN "0100" =>
            segment_data <= "0110011";
        WHEN "0101" =>
```

```
            segment_data <= "1011011";
        WHEN "0110" =>
            segment_data <= "1011111";
        WHEN "0111" =>
            segment_data <= "1110000";
        WHEN "1000" =>
            segment_data <= "1111111";
        WHEN "1001" =>
            segment_data <= "1111011";
        WHEN "1010" =>
            segment_data <= "1110111";
        WHEN "1011" =>
            segment_data <= "0011111";
        WHEN "1100" =>
            segment_data <= "1001110";
        WHEN "1101" =>
            segment_data <= "0111101";
        WHEN "1110" =>
            segment_data <= "1001111";
        WHEN "1111" =>
            segment_data <= "1000111";
      WHEN OTHERS =>
            segment_data <= "0111110";
END CASE;
END PROCESS;

-- extract segment data and LED driver is inverted
segment_a <= NOT segment_data(6);
segment_b <= NOT segment_data(5);
segment_c <= NOT segment_data(4);
segment_d <= NOT segment_data(3);
segment_e <= NOT segment_data(2);
segment_f <= NOT segment_data(1);
segment_g <= NOT segment_data(0);

END a;
```

10.1.6 Arduino Sketch

```
// Capacitance sensor
// Tim McCormick & Brian Sakhuja
// 10kHz clock
// Cmin = 0.1uF
// Cmax = 0.4uF

// digtal input pin
int timer_pin = 13;
```

```
//digital offset
int offset = 0;

// digital offset value read on analog input pin A0
unsigned long pulse_duration = 0;

//  duty cycle 555 timer signal
float duty_cycle = 0.67;

//the clock period in microseconds that was used in DE2
    implementation
// 10kHz
unsigned long clock_period = 100;

// initialize count values
int count_temp = 0;
int count_final = 0;
int count_out = 0;

// initialize output
int output = 0;

void setup() {
  Serial.begin(9600);
  pinMode(timer_pin, INPUT);
}

void meas_period() {
  pulse_duration = pulseIn(timer_pin, HIGH);

  count_temp = pulse_duration / (duty_cycle * clock_period);
  if (count_temp < count_final) {
  count_final = count_final;
} else {
  count_final = count_temp;
}
}

void output_value() {
  offset = analogRead(A1);
  output = count_final - offset;
}

void loop() {
  meas_period();
  output_value();

  //print output once every 1 second
```

```
  Serial.print(count_final);
  Serial.print(' ');
  Serial.print(offset);
  Serial.print(' ');
  Serial.println(output);
  delay(1000);
}
```

10.2 Practicum 7

DAC and ADC Circuits: Complete Listing of Calculations for Prelab

Part 1

Calculating DAC output voltage for 11111111,

$$\begin{aligned} E_0 &= 5V \times (\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256}) \\ &= 4.9805V \end{aligned}$$

01111111,

$$\begin{aligned} E_0 &= 5V \times (\frac{0}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256}) \\ &= 2.4805V \end{aligned}$$

00111111,

$$\begin{aligned} E_0 &= 5V \times (\frac{0}{2} + \frac{0}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256}) \\ &= 1.2305V \end{aligned}$$

00011111,

$$\begin{aligned} E_0 &= 5V \times (\frac{0}{2} + \frac{0}{4} + \frac{0}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256}) \\ &= 0.6055V \end{aligned}$$

00001111,

$$\begin{aligned} E_0 &= 5V \times (\frac{0}{2} + \frac{0}{4} + \frac{0}{8} + \frac{0}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256}) \\ &= 0.2930V \end{aligned}$$

00000111,

$$\begin{aligned} E_0 &= 5V \times (\frac{0}{2} + \frac{0}{4} + \frac{0}{8} + \frac{0}{16} + \frac{0}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256}) \\ &= 0.1367V \end{aligned}$$

00000011,

$$\begin{aligned} E_0 &= 5V \times (\frac{0}{2} + \frac{0}{4} + \frac{0}{8} + \frac{0}{16} + \frac{0}{32} + \frac{0}{64} + \frac{1}{128} + \frac{1}{256}) \\ &= 0.0586V \end{aligned}$$

00000001,

$$\begin{aligned} E_0 &= 5V \times (\frac{0}{2} + \frac{0}{4} + \frac{0}{8} + \frac{0}{16} + \frac{0}{32} + \frac{0}{64} + \frac{0}{128} + \frac{1}{256}) \\ &= 0.0195V \end{aligned}$$

00000000,

$$\begin{aligned} E_0 &= 5V \times (\frac{0}{2} + \frac{0}{4} + \frac{0}{8} + \frac{0}{16} + \frac{0}{32} + \frac{0}{64} + \frac{0}{128} + \frac{0}{256}) \\ &= 0.0V \end{aligned}$$

Calculating the percent error, 11111111,

$$\begin{aligned} \text{Percent Error} &= \frac{\text{Predicted Output} - \text{Measured Output}}{\text{Predicted Output}} \times 100\% \\ &= \frac{4.9805 - 4.967}{4.9805} \times 100\% \\ &= 0.27\% \end{aligned}$$

01111111,

$$\begin{aligned} \text{Percent Error} &= \frac{2.4805 - 2.47}{2.4805} \times 100\% \\ &= 0.42\% \end{aligned}$$

00111111,

$$\begin{aligned} \text{Percent Error} &= \frac{1.2305 - 1.222}{1.2305} \times 100\% \\ &= 0.69\% \end{aligned}$$

00011111,

$$\begin{aligned} \text{Percent Error} &= \frac{0.6055 - 0.60}{0.0655} \times 100\% \\ &= 0.91\% \end{aligned}$$

00001111,

$$\begin{aligned} \text{Percent Error} &= \frac{0.2930 - 0.2900}{0.2930} \times 100\% \\ &= 1.02\% \end{aligned}$$

00000111,

$$\begin{aligned}\text{Percent Error} &= \frac{0.1367 - 0.135}{0.1367} \times 100\% \\ &= 1.24\%\end{aligned}$$

00000011,

$$\begin{aligned}\text{Percent Error} &= \frac{0.0586 - 0.0565}{0.0586} \times 100\% \\ &= 3.58\%\end{aligned}$$

00000001,

$$\begin{aligned}\text{Percent Error} &= \frac{0.0195 - 0.0185}{0.0195} \times 100\% \\ &= 5.13\end{aligned}$$

00000000,

$$\begin{aligned}\text{Percent Error} &= \frac{0 - 0.0025}{0} \times 100\% \\ &= -\end{aligned}$$

Part 2 For input: 0.0V,

$$\begin{aligned}\text{Measured output} &= \frac{0}{0.0196} \\ \text{Measured output} &= 0 \\ \text{Binary equivalent of measured output} &= 00000000\end{aligned}$$

0.5V,

$$\begin{aligned}\text{Measured output} &= \frac{0.5}{0.0196} \\ \text{Measured output} &= 25.5102 \\ \text{Binary equivalent of measured output} &= 00011001\end{aligned}$$

1.0V,

$$\begin{aligned}\text{Measured output} &= \frac{1.0}{0.0196} \\ \text{Measured output} &= 51.0204 \\ \text{Binary equivalent of measured output} &= 00110011\end{aligned}$$

1.5V,

$$\text{Measured output} = \frac{1.5}{0.0196}$$
$$\text{Measured output} = 76.5306$$
$$\text{Binary equivalent of measured output} = 01001100$$

2.0V,

$$\text{Measured output} = \frac{2.0}{0.0196}$$
$$\text{Measured output} = 102.0408$$
$$\text{Binary equivalent of measured output} = 01100110$$

2.5V,

$$\text{Measured output} = \frac{2.5}{0.0196}$$
$$\text{Measured output} = 127.5510$$
$$\text{Binary equivalent of measured output} = 01111111$$

3.0V,

$$\text{Measured output} = \frac{3.0}{0.0196}$$
$$\text{Measured output} = 153.0612$$
$$\text{Binary equivalent of measured output} = 10011001$$

3.5V,

$$\text{Measured output} = \frac{3.5}{0.0196}$$
$$\text{Measured output} = 178.5714$$
$$\text{Binary equivalent of measured output} = 10110010$$

4.0V,

$$\text{Measured output} = \frac{4.0}{0.0196}$$
$$\text{Measured output} = 204.0816$$
$$\text{Binary equivalent of measured output} = 11001100$$

4.5V,

$$\text{Measured output} = \frac{3.5}{0.0196}$$
$$\text{Measured output} = 178.5714$$
$$\text{Binary equivalent of measured output} = 10011001$$

5.0V,

$$\text{Measured output} = \frac{5.0}{0.0196}$$
$$\text{Measured output} = 255.1020$$
$$\text{Binary equivalent of measured output} = 11111111$$

Calculating the percent error, 0.0V,

$$\begin{aligned}\text{Percent Error} &= \frac{\text{Predicted Output} - \text{Measured Output}}{\text{Predicted Output}} \times 100\% \\ &= \frac{0.0 - 0.0}{0.0} \times 100\% \\ &= 0.0\%\end{aligned}$$

0.5V,

$$\begin{aligned}\text{Percent Error} &= \frac{0.5096 - 0.5}{0.5096} \times 100\% \\ &= 1.88\%\end{aligned}$$

1.0V,

$$\begin{aligned}\text{Percent Error} &= \frac{1.0192 - 1.0}{1.0192} \times 100\% \\ &= 1.88\%\end{aligned}$$

1.5V,

$$\begin{aligned}\text{Percent Error} &= \frac{1.5092 - 1.5}{1.5092} \times 100\% \\ &= 0.61\%\end{aligned}$$

2.0V,

$$\begin{aligned}\text{Percent Error} &= \frac{2.0188 - 2.0}{2.0188} \times 100\% \\ &= 0.93\%\end{aligned}$$

2.5V,

$$\begin{aligned}\text{Percent Error} &= \frac{2.5088 - 2.5}{2.5088} \times 100\% \\ &= 0.35\%\end{aligned}$$

3.0V,

$$\text{Percent Error} = \frac{3.0184 - 3}{3.0184} \times 100\% = 0.61\%$$

3.5V,

$$\text{Percent Error} = \frac{3.528 - 3.5}{3.528} \times 100\% = 0.79\%$$

4.0V,

$$\text{Percent Error} = \frac{3.9984 - 4.0}{3.9984} \times 100\% = -0.04\%$$

4.5V,

$$\text{Percent Error} = \frac{4.4884 - 4.5}{4.4884} \times 100\% = -0.26\%$$

5.0V,

$$\text{Percent Error} = \frac{4.998 - 5.0}{4.998} \times 100\% = -0.04\%$$

10.3 Practicum 8

VHDL Code for Prelab. Implements the finite state machine.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY prelab8 IS
   PORT ( Clock, Resetn, w : IN   STD_LOGIC ;
          z, stateA, stateB, stateC, stateD, stateE, stateF, stateG,
              stateH, stateI        : OUT  STd_LOGIC );
END prelab8 ;

ARCHITECTURE Behavior OF prelab8 IS
        TYPE State_type IS (A,B,C,D,E,F,G,H,I);
        SIGNAL y : State_type ;
```

```
BEGIN
   PROCESS ( Resetn, Clock )
   BEGIN
      IF Resetn = '0' THEN
         y <= A;
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         CASE y IS
            WHEN A =>
               IF w='0' THEN
                  y <= B;
               ELSE
                  y <= F;
               END IF;
            WHEN B =>
               IF w='0' THEN
                  y <= C;
               ELSE
                  y <= F;
               END IF;
            WHEN C =>
               IF w='0' THEN
                  y <= D;
               ELSE
                  y <= F;
               END IF;
            WHEN D =>
               IF w='0' THEN
                  y <= E;
               ELSE
                  y <= F;
               END IF;
            WHEN E =>
               IF w='0' THEN
                  y <= E;
               ELSE
                  y <= F;
               END IF;
            WHEN F =>
               IF w='0' THEN
                  y <= B;
               ELSE
                  y <= G;
               END IF;
            WHEN G =>
               IF w='0' THEN
                  y <= B;
               ELSE
                  y <= H;
               END IF;
```

```
            WHEN H =>
               IF w='0' THEN
                  y <= B;
               ELSE
                  y <= I;
               END IF;
            WHEN I =>
               IF w='0' THEN
                  y <= B;
               ELSE
                  y <= I;
               END IF;
            END CASE;
         END IF;
      END PROCESS;
      z <= '1' WHEN y = E OR y = I ELSE '0';
      stateA <= '1' WHEN y = A ELSE '0';
      stateB <= '1' WHEN y = B ELSE '0';
      stateC <= '1' WHEN y = C ELSE '0';
      stateD <= '1' WHEN y = D ELSE '0';
      stateE <= '1' WHEN y = E ELSE '0';
      stateF <= '1' WHEN y = F ELSE '0';
      stateG <= '1' WHEN y = G ELSE '0';
      stateH <= '1' WHEN y = H ELSE '0';
      stateI <= '1' WHEN y = I ELSE '0';

   END Behavior;
```

10.4 Practicum 9

10.4.1 clock_new_107KHz.vhd

The number of bits needed to store the count_107KHz logic vector signal depends on the number we divide by. In this case, we divide by 455, which can be represented by 9 bits. Hence, we store the signal in 9 bits.

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY clock_new_107KHz IS

   PORT
   (
      clock_50MHz      : IN  STD_LOGIC;
```

```
      clock_107KHz_out    : OUT   STD_LOGIC);

END clock_new_107KHz;

ARCHITECTURE a OF clock_new_107KHz IS
   SIGNAL   count_107KHz: STD_LOGIC_VECTOR(8 DOWNTO 0);
   SIGNAL   clock_107KHz_int: STD_LOGIC;
BEGIN
   PROCESS
   BEGIN
   -- Divide by 455 to get 107KHz clock
      WAIT UNTIL clock_50MHz'EVENT and clock_50MHz = '1';
         IF count_107KHz < 455 THEN
            count_107KHz <= count_107KHz + 1;
         ELSE
            count_107KHz <= "000000000";
         END IF;
      IF count_107KHz < 234 THEN
         clock_107KHz_int <= '0';
      ELSE
         clock_107KHz_int <= '1';
      END IF;
      clock_107KHz_out <= clock_107KHz_int;
   END PROCESS;
END a;
```

10.4.2 sonarcount.vhd

Original code was modified to allow 16 bits of data.

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

ENTITY sonarcount IS
   PORT (  Clock, Init, Echo   : IN   STD_LOGIC ;
         Q             : OUT    STD_LOGIC_VECTOR (15 DOWNTO 0)) ;
END sonarcount ;

ARCHITECTURE Behavior OF sonarcount IS
   SIGNAL Count     : STD_LOGIC_VECTOR (15 DOWNTO 0) ;
   SIGNAL FinalCount   : STD_LOGIC_VECTOR (15 DOWNTO 0) ;

BEGIN
   PROCESS (Init, Clock)
   BEGIN
      IF Init = '0' THEN
         Count <= "0000000000000000";
```

```
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         IF Echo = '0' THEN
            Count <= Count + 1;
         ELSE
            FinalCount <= Count;
         END IF;
      END IF;
   END PROCESS;
   Q <= FinalCount;
END Behavior;
```

10.4.3 adder.vhd

Original code was modified to allow 8 bits instead of 4.

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


ENTITY adder IS
   PORT (  BusIn      : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
         RegIn    : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
         AddSub     : IN    STD_LOGIC ;
         Sum        : OUT     STD_LOGIC_VECTOR(7 DOWNTO 0));
END adder;

ARCHITECTURE Behavior OF adder IS

BEGIN
   WITH AddSub SELECT
      Sum <=  RegIn + BusIn WHEN '0',
            RegIn - BusIn WHEN OTHERS;
   END Behavior;
```

10.4.4 Register blocks (reg1.vhd, reg2.vhd, reg3.vhd, regx.vhd, regy.vhd)

For all the register blocks, the number of bits used was changed from 4 to 8. For example, *reg1.vhd* is shown below.

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

ENTITY reg1 IS
   PORT (  R1in       : IN  STD_LOGIC_VECTOR(7 DOWNTO 0) ;
         IE, Clock    : IN  STD_LOGIC ;
```

```
        Q1out         : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END reg1 ;

ARCHITECTURE Behavior OF reg1 IS
BEGIN
   PROCESS
   BEGIN
      WAIT UNTIL Clock'EVENT AND Clock = '1' ;
      IF IE = '1' THEN
         Q1out <= R1in ;
      END IF ;
   END PROCESS ;
END Behavior ;
```