ELEC 470 – FPGA

Ultrasonic Distance Motor

Prof. Hector Daniel Rico-Aniles

By Jack Urso and Nimesh Patel

December 10th, 2021

Contents

List of Tables

List of Figures

Abstract

The ultrasonic distance senor was designed in VHDL and implemented on FPGA board. The stepper Motor is working on the principal of PWM and the speed of it depends on how fast it is receiving pulses. The ultrasonic sensor works on the echo and trigger signals which helps to find nearby distance. The whole project uses several counters, comparators, FSM, RAM, Serial Port. It will be tested on a MAX 10 lite Terasic Board that features 10M50DAF48C7G FPGA chip. It uses certain portion of the output pins. (The exact number of the counters, comparators, FSM, RAM, etc will be included in the final draft.)

1. Introduction

For our final project of the semester, we decided to come up with an Ultrasonic Distance Motor. Basically, it's an ultrasonic sensor that is placed on top of a stepper motor that rotates 360 degrees in one direction repeatedly. The sensor is capable of measuring the distance up to 200 centimeters, which means it has a detection area of 126000 cm^2 (area of circle). As you can guess, our circuit could have many practical uses. For example, let's say our circuit was able to detect outside the circumference of your car. If another car was about to hit your car, our circuit could be able to warn you in time and prevent a collision. Or, let's say you wanted to protect an item in your home. You could place our circuit near your precious item, and it could detect if anyone gets close to it.

Along with the ultrasonic sensor and the stepper motor, there are two other components, the random-access memory (RAM) and serial adapter. The RAM is responsible for saving the centimeter measurement readings, but it can only save up to 256 values. Once it reaches it memory storage, the address pointer goes back to zero and begins rewriting new measurement readings. It works like any of memory device, and it has similar features, such as read and write. Although we included the RAM in our final project, it doesn't have any other practical uses except for the saving the values. The serial adapter is responsible for displaying the centimeter measurement readings onto a serial monitor. To do this, we had to use an external device called SH-U09C, which is just a UART USB to TTL adapter. This device is capable of sending 9600 bits for second, and it only receive 8 bits during a total cycle.

To implement all of these components, we used HSIC Hardware Description Language (VHDL). As mentioned in the name, this language is a descriptive language that is used to model the behavior and structure of a digital system. To actually created and compile our VHDL files, we used a program called Quartus Prime 20.1. After this process, we had to simulate the files to determine whether the signals were operating correctly. We used a simulation program called ModelSim. After a long process of checking the simulation, we uploaded the files onto a device called a Field-Programmable Gate Array (FPGA). The FPGA is able to build the circuit from the given files and performed what we desired our circuit to do.

Before we actually began building the circuit, we had to come up with some specifications to abide by. For instance, the ultrasonic sensor has to read up to 2 meters, equal to 200 centimeters, in distance. It can read over 200 centimeters, but it has to at least read 200 centimeters. With the stepper motor, it must be able to rotate 360 degrees back and forth. And, the resolution of the stepper motor will be 10 degrees. One of our original specifications was that the sensor and the stepper motor would worked together by using start and stop signals; however, due to time constraints, we were not able to satisfy this condition. Instead of using these signals, we just synchronized the total clock cycles together for both of the components. Therefore, we would know the measurement reading at each direction (or angle) of the stepper motor.

2. HCSR04

The HCSR04 ultrasonic sensor is a fundamental component to our design because it tells us how far an object is away from the sensor. There are three parts to the HCSR04 component, sensor, distance, and calculator. The sensor part creates the trigger signal, while the other two parts interprets the echo signal, which is generated after the trigger. The distance part is responsible for displaying the centimeter measurements on the three seven-segment displays. And, the calculator part is responsible for converting the echo signal to a centimeter measurement, which is stored in a binary vector. The block diagram for the HCSR04 is below.

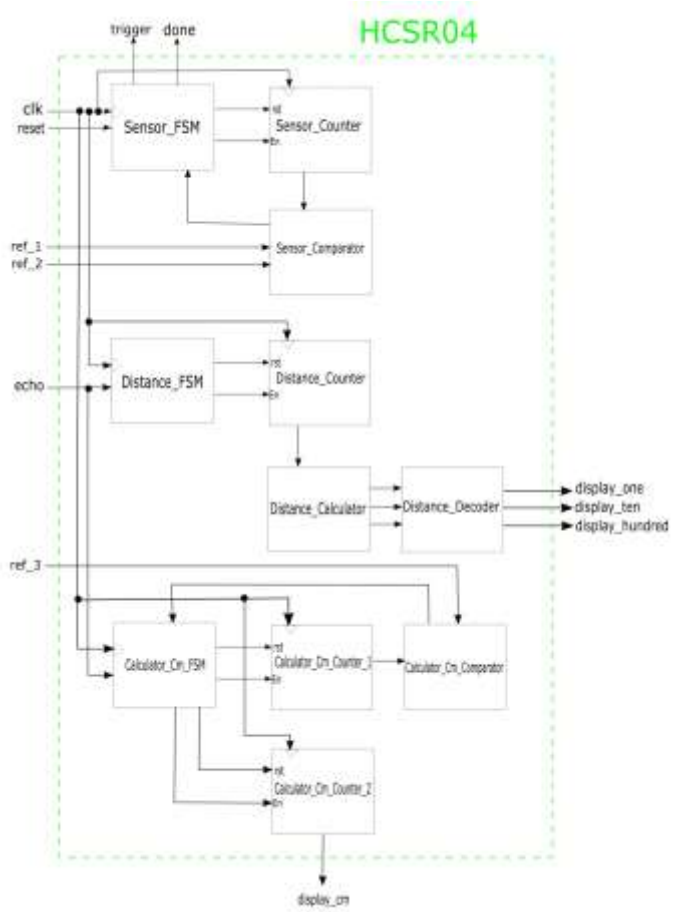

Figure 1: HCSR04 block diagram

2.1. Sensor Part

2.1.1. Sensor_Counter

The Sensor_Counter works exactly like any regular counter, and there's nothing special about it. There are three inputs, the reset, enable and clock, and only one output, cnt. For each clock cycle, the counter increments cnt by 1 if the enable bit in on, or cnt remains constant if the enable bit is off. And,

for every clock cycle, the cnt goes back to zero if the reset bit is on, or cnt remains constant if the reset bit is off.

2.1.2. Sensor_Comparator

The Sensor_Comparator is similar to most comparators, but this comparator differs a little. For my design, this comparator has two reference values, one to generate the 10-microsecond trigger signal, and the other to generate the total clock signal, which is 60 milliseconds. In addition to the reference values, there is one other input, the cnt from the Sensor_Counter. There is only one output value, the comp_out. Once the cnt is equal to one of the reference values, it the comp_out turns on and it sent to the Sensor_FSM; and, if cnt is not equal to the reference values, the comp_out remains off.

2.1.3. Sensor_FSM

The Sensor_FSM is the brain behind generating the trigger signal and the total clock cycle. It controls the outputs for the Sensor_Counter, enable and reset, and two other outputs, done and enable_trigger. The done signal tells the RAM component to start working, and it is only on for one clock cycle at the end of the total clock cycle. The enable_trigger output is the probably the most important signal in the circuit because it tells the HCSR04 device to generate the echo signal. There are four different states, and they perform the following operations, resting, generate trigger signal, continue to generate the total clock cycle, and enable done signal while resetting the counter. There is a state table and diagram below.

State	done	enable_trigger	enable_counter	reset_counter
S0	0	0	0	0
S1	0	1	1	0
S2	0	0	1	0
S3	1	0	0	1

Table 1: Sensor_FSM state machine output values

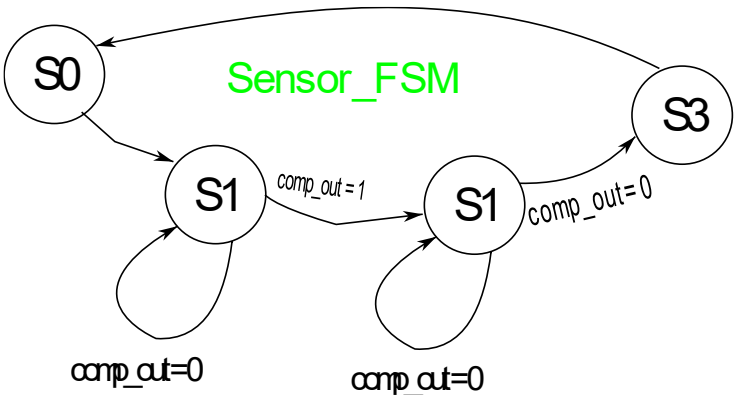

Figure 2: Sensor_FSM state machine diagram

2.2. Distance Part

2.2.1. Distance_Counter

The Distance_Counter works exactly the same as the previously mentioned counter above, Sensor_Counter. Nothing is different besides the name of the file.

2.2.2. Distance_Calculator

The Distance_Calculator is a custom design component that is used for taking the cnt from the Distance_Counter and converting it to a decimal representation (in binary). Since the max distance is 200 centimeters, there are three output values, one for each digit. Thus, in the code, there is a list of conditional statements that assigns the outputs, distance_one, distance_ten, and distance_hundred, to their proper decimal representation. Using the formula below will give you the centimeter measurement based on cnt.

$$\text{cm Measurement} = (\text{cnt}*0.02) / 58$$

The Distance_Calculator only displays the measurements to its tenth position, so the only possible values will be 10, 20, 30, …, and 200. For example, let's say the cnt value is 43500, which is equivalent to 15 cm based on the equation, then the measurement will be 20 because it got caught in the conditional statement between 10 and 20 centimeters. There are 20 conditional statements for catching the cnt values between 10 and 200.

2.2.3. Distance_Decoder

The Distance_Decoder simply converts the decimal representation outputs from the Distance_Calculator to their seven-segment representation. Thus, there are three inputs and three outputs. By using the table below, you can convert between the two different representations.

	Seven Segments						
Value	**A**	**B**	**C**	**D**	**E**	**F**	**G**
0000	0	0	0	0	0	0	1
0001	1	0	0	1	1	1	1
0010	0	0	1	0	0	1	0
0011	0	0	0	0	1	1	0
0100	1	0	0	1	1	0	0
0101	0	1	0	0	1	0	0
0110	0	1	0	0	0	0	0
0111	0	0	0	1	1	1	1
1000	0	0	0	0	0	0	0
1001	0	0	0	0	1	0	0
Others	1	1	1	1	1	1	1

Table 2: Binary to seven-segment output values

2.2.4. Distance_FSM

The Distance_FSM, like the Sensor_FSM, is the brain behind interpreting the echo signal. There are three different states, resting, counting, reset counter, and enable counter. When the echo signal is on, it switches from the resting state to resetting the counter. Immediately, it moves to the enable counter state and starts counting the number of cycles the echo signal is on for. When the echo signal is finally turned off, it moves back to the resting state. Below is a state table and diagram.

State	enable_counter	reset_counter
S0	0	0
S1	0	1
S2	1	0

Table 3: Distance_FSM state machine output values

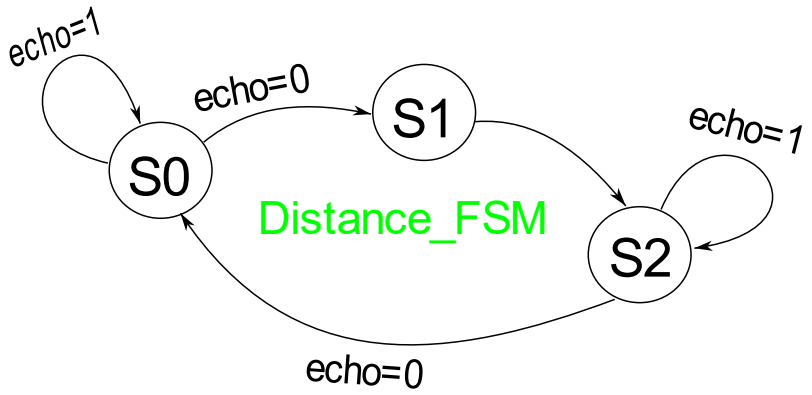

Figure 3: Distance_FSM state machine diagram

2.3. Calculator Part

2.3.1. Calculator_Cm_Counter_One

The Calculator_Cm_Counter_One works exactly the same as the previously mentioned counter above, Sensor_Counter. Nothing is different besides the name of the file. However, you might be wondering why there are two counters. This counter is for counting up to 2900, which is equal to 1 centimeter. Thus, when it finally reaches the value, the other counter, Calculator_Cm_Counter, is incremented by 1 and the reset is enabled on this counter.

2.3.2. Calculator_Cm_Comparator

The Calculator_Cm_Comparator was alluded to above, but it basically, tells the Calculator_Cm_FSM when cnt has reached 2900, which is the reference value. Thus, the only difference between this comparator and Sensor_Comparator is that this comparator has only one reference value.

2.3.3. Calculator_Cm_Counter_Two

The Calculator_Cm_Counter_Two works exactly the same as the previously mentioned counter above, Sensor_Counter. Nothing is different besides the name of the file. As already mentioned, this counter keeps track of the number of centimeters.

2.3.4. Calculator_Cm_FSM

The Calculator_Cm_FSM works very similar to the Distance_FSM because it uses the same signal, echo, to determine the measurement reading. There are 4 different states, resting, resetting counters, counting up to 1 centimeter, and incrementing the centimeter cnt. The echo signal works the exact same as it does in Distance_FSM. The only difference is that the echo signal essentially acts like a global reset, so when it’s turned off, it immediately goes back to

state zero. That's why there's no arrow going back to state zero. Below is a state table and diagram.

State	enable_counter_1	enable_counter_2	reset_counter_1	reset_counter_2
S0	0	0	0	0
S1	0	0	1	1
S2	1	0	0	0
S3	0	1	1	0

Table 4: Calculator_Cm_FSM state machine output values

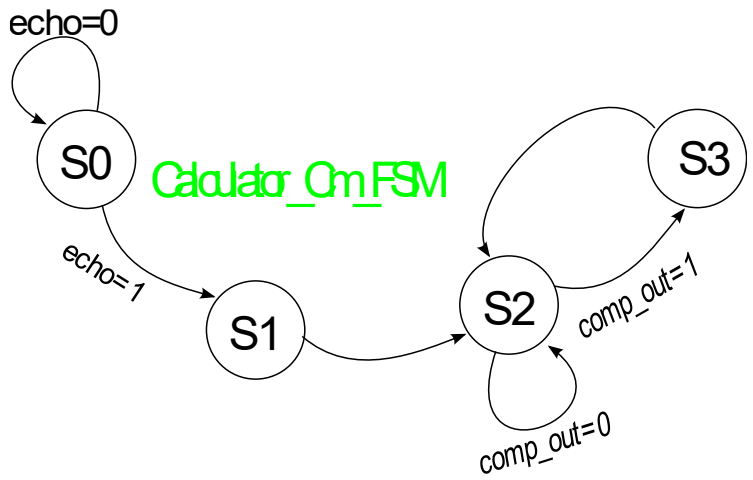

Figure 4: Calculator_Cm_FSM state machine diagram

3. Sensor_RAM

The Sensor_RAM primary responsibility in the circuit is to save the centimeter measurements. In order to do this, it uses an array of vectors, each containing a centimeter reading. Whenever the done_in signal is on, it tells the RAM_FSM to write the centimeter measurement value, data_in, to memory given an address that comes from the RAM_Counter. The Sensor_RAM circuit is intuitive to understand, and it was actually very easy to implement. The block diagram for Sensor_RAM is below.

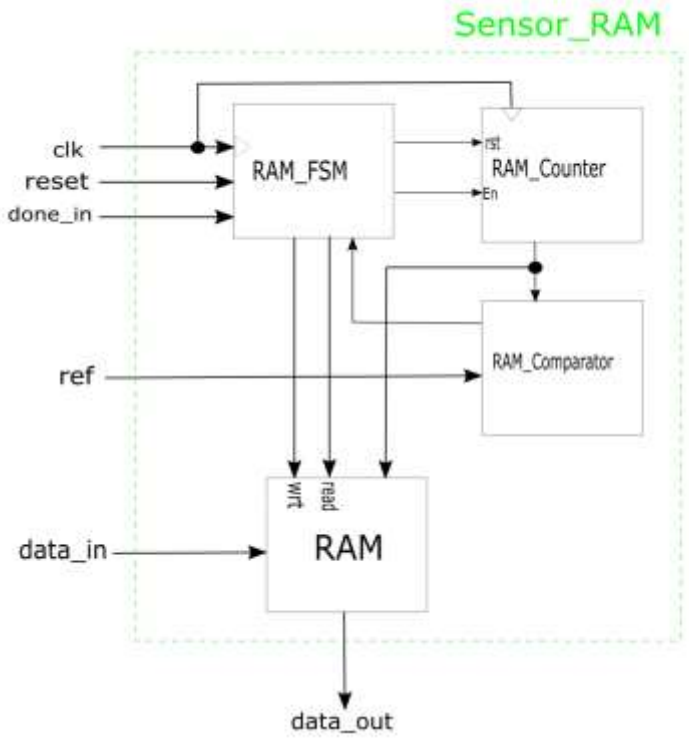

Figure 5: Sensor_RAM block diagram

3.1. RAM_Counter

The RAM_Counter works exactly the same as the previously mentioned counter above, Sensor_Counter. Nothing is different besides the name of the file.

3.2. RAM_Comparator

The RAM_Comparator works the same as the previously mentioned comparator above, Calculator_Cm_Counter_One, besides for one difference, the reference value is equal to 255. This specific reference value will make sense later.

3.3. RAM_FSM

Being the brains of the Sensor_RAM, the RAM_FSM is responsible for reading and writing to memory with the proper address value, cnt. It controls the following outputs, write, read, enable_counter, and reset_counter. There are six states to RAM_FSM, resting, write, read, increment counter, checking whether memory is filled, and resetting the counter. The only way for the RAM_FSM to get out of state zero is when the signal sensor_done is on. Remember the Sensor_FSM is responsible for creating this signal. Once the memory is filled, the cnt goes back to zero and it just rewrites new measurements into memory. Below is a state table and diagram.

State	write	read	enable_counter	reset_counter
S0	0	0	0	0
S1	1	0	0	0
S2	0	1	0	0
S3	0	0	1	0
S4	0	0	0	0
S5	0	0	0	1

Table 5: RAM_FSM state machine output values

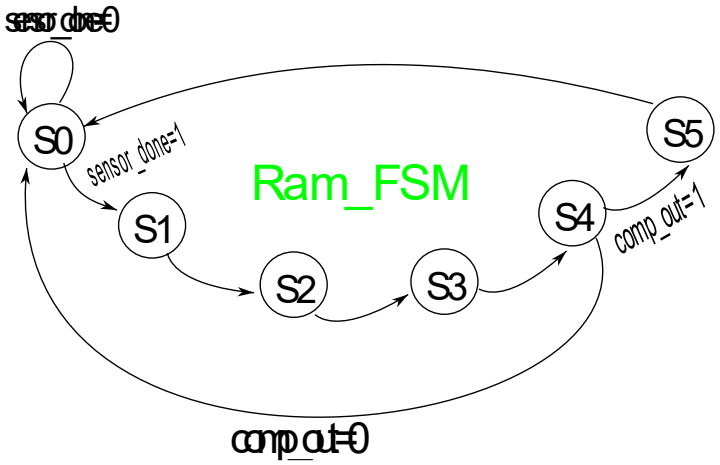

Figure 6: RAM_FSM state machine diagram

3.4. RAM

The RAM is the basic building block to the Sensor_RAM because it holds an array of 256 binary vectors, each containing the centimeter measurement value. And, working with the RAM_FSM, it writes and reads from memory. There are four inputs, data_in, address_in, write_in, and read_in. If write_in is on and read_in is off, then the data_in value is written to address address_in in memory; and, if write_in is off and read_in is on, then the measurement at address address_in is written to the only output signal, data_out. These two operations were able to be done using conditional statements. Although our circuit doesn't actually retrieve values from memory, it was not necessary to include the output, data_out; however, I felt like it made more sense to include it.

4. Sensor_Serial

The Sensor_Serial is responsible for converting the Calculator_Cm_Counter_Two 9-bit output signal, data_in, to a 1-bit signal, tx_port_out, and displaying the measurement reading onto a computer screen. To do this, we used UART USB to TTL adapter called

SH-U09C or FT232RL. Originally, we thought we were going to use the done_in signal to activate the Serial_FSM to start sending data values into the serial adapter; however, this ended up not working in implementation. Instead of using the data_in signal, we turned on the sensor_done signal for one clock cycle around every 126 milliseconds. The sensor_done signal is inside the Serial_FSM. To do this, we had to use the Serial_Counter and Serial_Comparator components again. The reference value for this Serial_Comparator was 6291456, which is equivalent 125.82912 milliseconds. Since these two components only generate the done_in signal, they will not be explained below. Essentially, the Sensor_Serial, uses communication protocols to transmit sensor_done out of the circuit. Since our circuit only transmits data, we weren't required to use any of the receiving ports. The block diagram for Sensor_Serial is below. The extra Serial_Counter and Serial_Comparator in the down-right corner of the block diagram are responsible for generating the sensor_done signal, and they should be ignored while looking at the specific components.

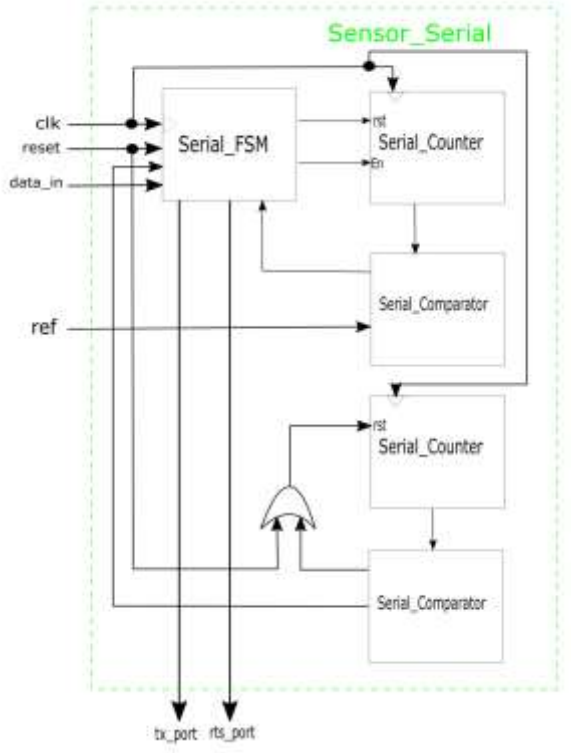

Figure 7: Sensor_Serial block diagram

4.1. Serial_Counter

The Serial_Counter works exactly the same as the previously mentioned counter above, Sensor_Counter. Nothing is different besides the name of the file.

4.2. Serial_Comparator

The Serial_Comparator works the same as the previously mentioned comparator above, RAM_Comparator, besides for one difference, the reference value is equal to 5209. The reference value was obtained by doing the following math below.

Baud Rate = 9600 bits per second
Clk Speed = 50 MHz = 2.0E-8
So, 1 bit (# cycles or ref) = ((1/9600)/2.0E-8) = 5208.33 = "5209 cycles"

4.3. Serial_FSM

The Serial_FSM was the most complicated state machine for the entire project because it requires 19 different states and four output controls, tx_port, rts_port, enable_counter, and reset_counter. The tx_port is the transmitter, so it is responsible for sending the data values. The rts_port is the request to send, so it tells the serial adapter to start reading the data. Thus, these two outputs work together in transferring the data. By default, at its resting state, the tx_port is on; however, once you want to transfer data, the tx_port must be off for 5209 cycles. After this, you reset the counter. Then, you set tx_port to data_in(0) and rts_port to on for 5209 cycles, and then reset the counter at the next state. You continue this process until you get to data_in(7), where you go back to state zero. Since the serial adapter is only capable of passing 8 data values, we weren't able to pass the last data value, data_in(8); however, this does not matter since the sensor only has to be able to read up to 200 centimeters. Below is a state table and diagram.

State	tx_port	rts_port	enable_counter	reset_counter
S0	1	0	0	0
S1	0	0	1	0
S2	0	0	0	1
S3	data_in(0)	1	1	0
S4	0	0	0	1
S5	data_in(1)	1	1	0
S6	0	0	0	1
S7	data_in(2)	1	1	0
S8	0	0	0	1
S9	data_in(3)	1	1	0
S10	0	0	0	1
S11	data_in(4)	1	1	0
S12	0	0	0	1
S13	data_in(5)	1	1	0
S14	0	0	0	1
S15	data_in(6)	1	1	0
S16	0	0	0	1
S17	data_in(7)	1	1	0
S18	0	0	0	1

Table 6: Serial_FSM state machine output values

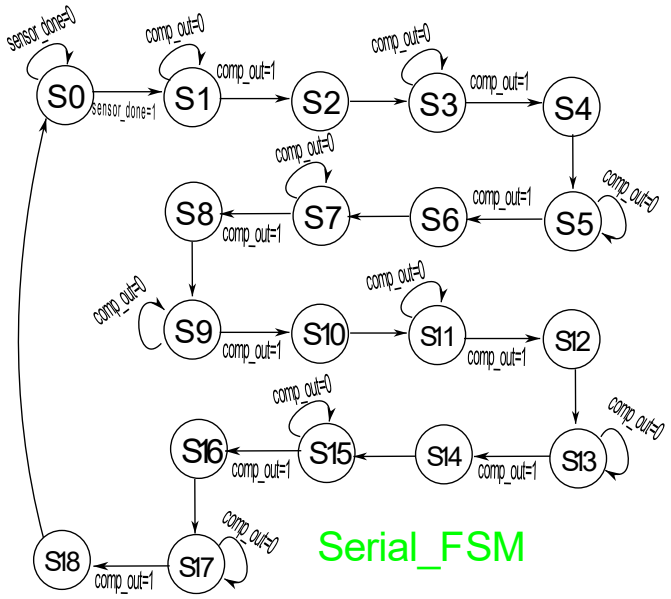

Figure 8: Serial_FSM state machine diagram

5. Stepper_Motor

The Stepper Motor is responsible to control the direction of the sensor. The main purpose of the Motor is to fix the sensor on the top of it and rotate it 360 degrees to measure the distance of the nearby objects within the range of the sensor. The motor used for this is a bipolar motor with 4 phases. The 4 phases are used to run smooth transition from one state to another. There are total of 5 states which are required to generate one step. The step generation is a continuous process to make full rotation.

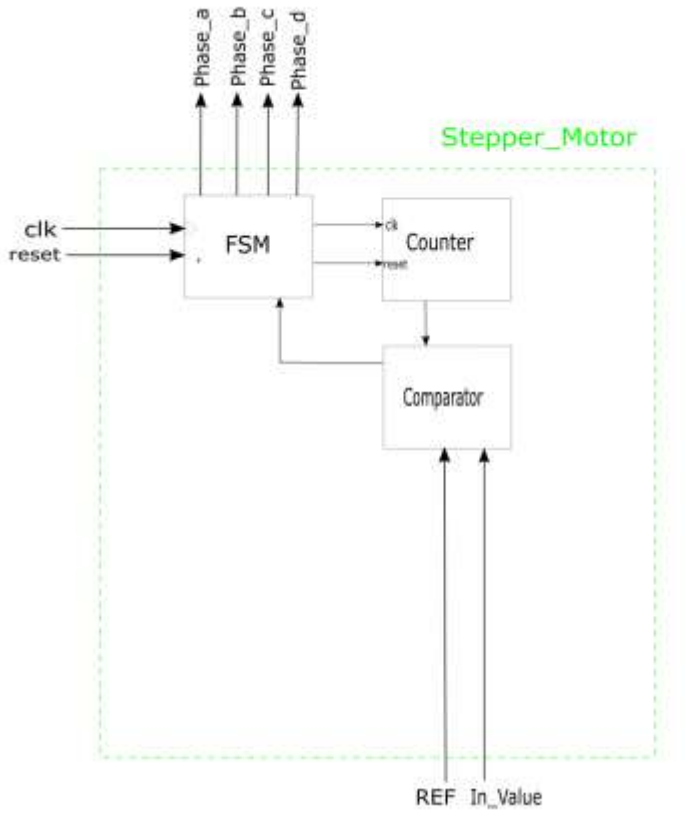

Figure 9: Stepper Motor block diagram

5.1. Counter

The Counter works exactly like any regular counter, and there's nothing special about it. There are three inputs, the reset, enable and clock, and only one output, cnt. For each clock cycle, the counter increments cnt by 1 if the enable bit in on, or cnt remains constant if the enable bit is off. And, for every clock cycle, the cnt goes back to zero if the reset bit is on, or cnt remains constant if the reset bit is off.

5.2. Comparator

The Comparator is like most comparators, but this comparator differs a little. For our design, this comparator has two reference values, one to generate the 50 million cycles and the other to pass the total clock signal. In addition to the reference values, there is one other input, the cnt from the Counter. There is only one output value, the comp_out. Once the cnt is equal to one of the reference values, it the comp_out turns on and it sent to the FSM; and, if cnt is not equal to the reference values, the comp_out remains off.

5.3. OneStep_FSM

The FSM is the brain behind generating the trigger signal and the total clock cycle. It controls the outputs for the Counter, enable and reset, and direction, comp out and generates the output in terms of phase. The direction signal tells the other components to start working, and it is only on for one clock cycle at the end of the total clock cycle. The comp_out input is the probably the most important signal in the circuit because it tells the motor device to generate the enable signal. There are five different states, and they perform the following operations, resting, generate comp_out continue to generate the total clock cycle, and enable another signal while resetting the counter. There is a state table and diagram below.

State	Phase 1	Phase 2	Phase 3	Phase 4
S0	0	0	0	0
S1	1	0	0	0
S2	0	1	0	0
S3	0	0	1	0
S4	0	0	0	1

Table 7: FSM state machine output values

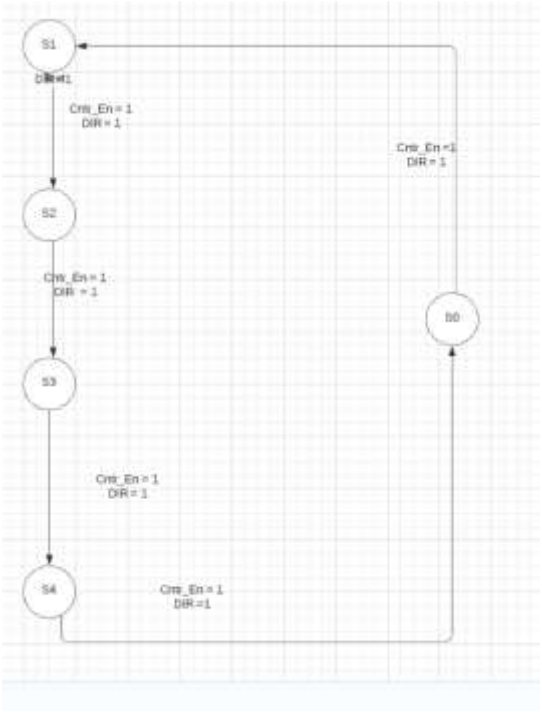

Figure 10: FSM state machine diagram

6. Circuit_Components

After completing the all the necessary components, we instantiated all of them together into Circuit_Components. Due to time constraints, we were unable to create a finite state machine that would communicate between the sensor and the motor. Instead, we are going to synchronous our total clock cycles, so we would know the measurement reading at a certain direction of the motor. Below is the block diagram for Circuit_Components.

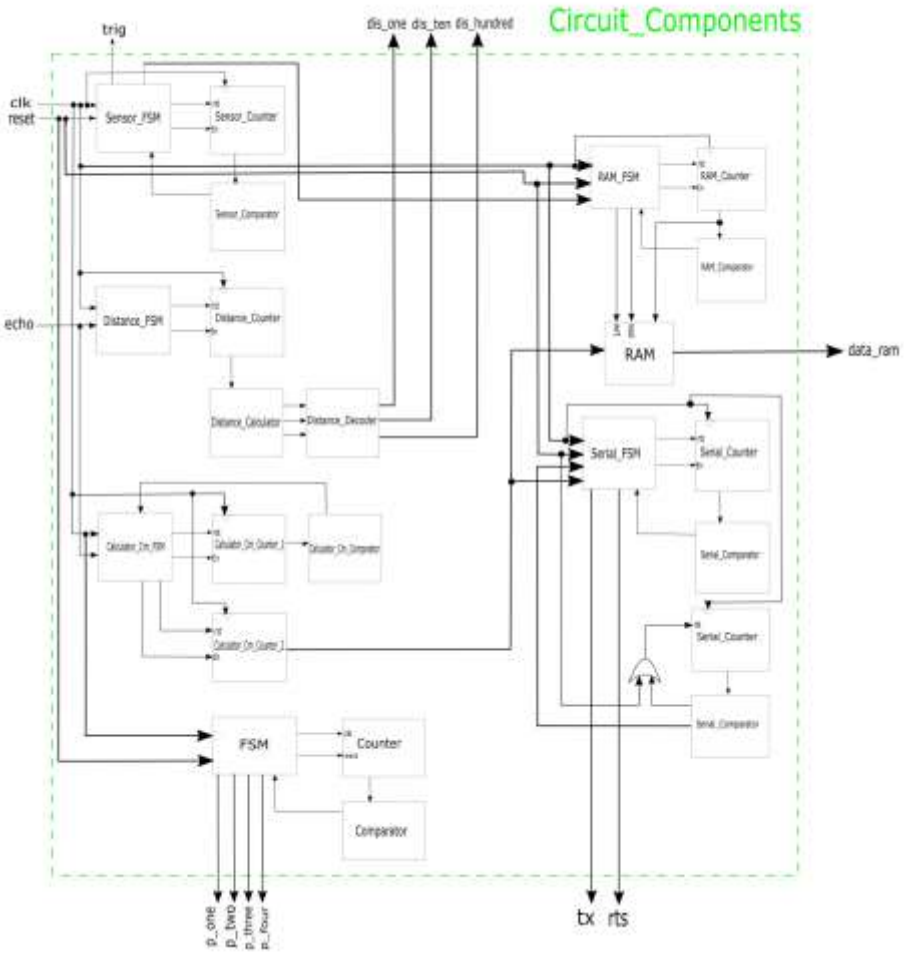

Figure 11: Circuit_Components block diagram

7. Simulations

For the simulation part, we used the software Modelsim Intel FPGA Starter Edition 2020 to simulate every component. By looking at the signals within Modelsim, you can tell whether your component is working properly. To make it easier to understand, we will be covering four different simulations, HCSR04, Sensor_RAM, Sensor_Serial, and

Stepper_Motor. Since Circuit_Components only instantiates all the components together, it will not be included in this part. For the actual testbench file, we ran the simulation for 120 milliseconds (two total clock cycles for the HCSR04) and left the echo signal on for 3,190,000 nanoseconds, which is equivalent to 55 centimeters. The math explaining this can be found below.

Echo Time (in microseconds) = (cm Measurement * 58)
Echo Time (in nanoseconds) = Echo Time (in microseconds) * 10^3

7.1. HCSR04 Simulation

Instead of showing the states for the three finite state machines, we are going to focus on the input and output values that are generated from those machines. However, all the finite state machines are working properly. In order to understand whether this component is working correctly, we must look at the output signals, trigger, done, display_cm, display_hundred, display_ten and display_one, and the only input signal, echo. So, in the figure below, you can see all of these signals.

Figure 12: HCSR04 first timing diagram

As you can see, the trigger, echo, and done signals are operating correctly. The trigger runs for 10 microseconds, echo runs for 3,190 microseconds, and the done signal is on for one cycle towards the end of the total cycle. And, for the display signals, those are close to operating correctly. The only problem is with display_cm. Since the counters are starting at zero, it fails to increment the Calculator_Cm_Counter_Two during the last cycle; therefore, it will always be one off from the actual value, 55. From looking at Table 2 above, you can tell that display_hundred, display_ten, and display_one are working properly. Next, let's take a closer look at display_cm, and examine why is incrementing as it should except for the last cycle. To do this, we must pay attention to one input signal, comparator_out, and two output signals, count_cycles and display_cm. Looking at the figure below, you can see all of these signals.

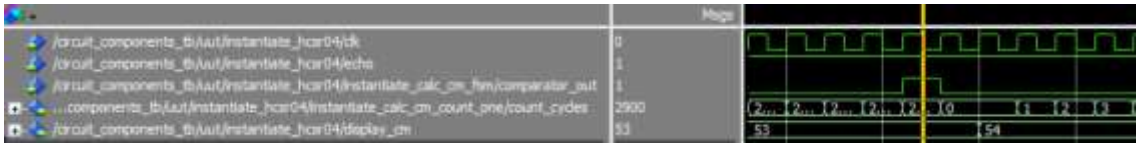

Figure 13: HCSR04 second timing diagram

Whenever count_cycles signal reaches the reference value 2900 (equivalent to 1 centimeter), the comparator_out signal bit turns on; and, when this happens, it tells the Calculator_Cm_Counter_Two to increment display_cm by 1. Therefore, from this figure above, you can easily see these signals working properly.

7.2. Sensor_RAM Simulation

With this simulation, we are going to go over the finite state machine signals and the memory list. First, let's start with the finite state machine signals. There are four output signals, wrt, rd, en_count, and reset_counter, and two input signals, done_in and comp_out. So, in the figure below, you can see all of these signals along with additional signals for clarity.

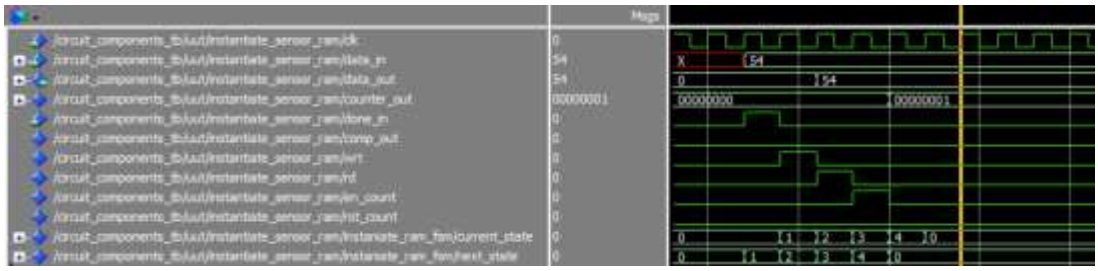

Figure 14: Sensor_RAM timing diagram

Based on Table 5 and Figure 6 above, the signals are working properly. When done_in bit is on, it goes from state 0 to state 1; and, when the comp_out bit is off, it goes from state 4 back to state 0. Along with the input signals, the output signals are also working correctly based on Table 5. Now, since the finite state machine looks good, let's move on to the memory list. There are a few important signals to keep in mind, one input signal, data_in, and two output signals, data_out and counter_out. All of these signals can be found in the figure above. The counter_out signal basically represents the address value within memory, while the other two signals are just the measurement readings. In order to check the memory, we must look at the "Memory List" tab on Modelsim. The figure below is a picture of the memory list.

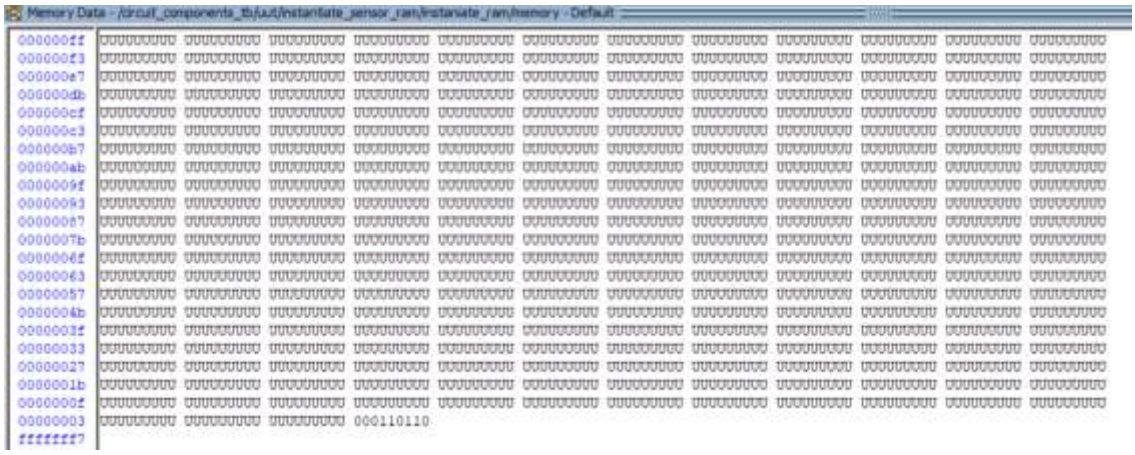

Figure 15: Sensor_RAM memory list

As you can see, there is a binary value at the first index in memory. And, as you might have guessed, this binary value is equivalent to 54 in decimal representation. At state 1, the data_in value is written to memory at address counter_out. Also, the data_out signal is just the value that is retrieved from memory at address counter_out, which happens at state 2. Therefore, based on the two figures above, we know that Sensor_RAM is working properly.

7.3. Sensor_Serial Simulation

To explain this simulation, we must go over the finite state machine signals. There are four output signals, tx_port_out, rts_port_out, en_count, and rst_count, and two input signals, data_in and done_in. The most important signal to pay attention to is the tx_port_out, which is responsible for sending the measurement reading into the serial adapter. So, in the figure below, you can see all of these signals along with additional signals for clarity.

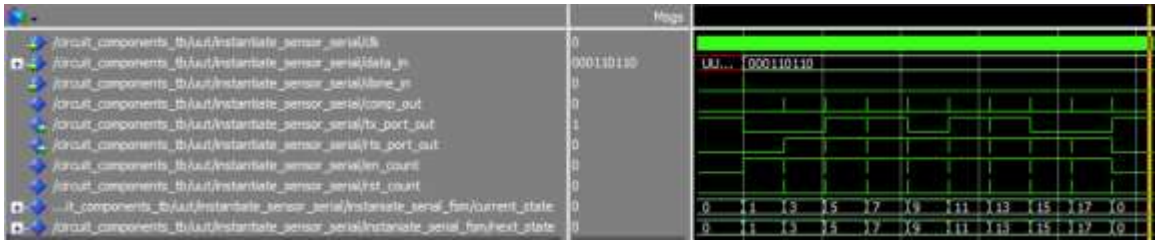

Figure 16: Sensor_Serial timing diagram

When the done_in signal bit is on, the finite state machines moves from state 0 to state 1, which begins the transmitting of data bits. This can easily be seen in the figure above. From state 3 to state 17 (odd states), every bit except for the last bit in data_in is transmitted using the tx_port_out signal. And, whenever a data bit is being transmitted, the rts_port_out signal bit is on, which is also shown in the figure above. Let's look at a specific signal, tx_port_out. At each odd state between state 3 and state 4, the timing diagram displays the following: low, high, high, low, high, high, low, and low. Now, let's rewrite this in binary, and then, reverse the binary value. After you do this, you should get the binary value "00110110." Since this is equivalent to the data_in binary value except for the last bit, we know that the data has been properly transmitted. And, from Table 6 and Figure 8, we know that the finite state machine is working correctly. Therefore, from what we can tell, Sensor_Serial components seems to be working as expected.

7.4. Stepper_Motor Simulation

The simulation of the motor contains the clock, reset, Phase a, phase b, phase c, phase d. The output depends on the current state and next state as they are always varying

according to the conditions. As the simulation below shows, how at every rising edge. As soon as phase a is set to zero another phase b is getting enabled. This goes on and on in loop. The state transition depends on how quickly the user wants to jump from one state to another. But since we want it to move from one state to another and set back to state zero. We are using count_rst_Fsm to do that. When the count_rst_Fsm is 1 the states get reset and start all over again.

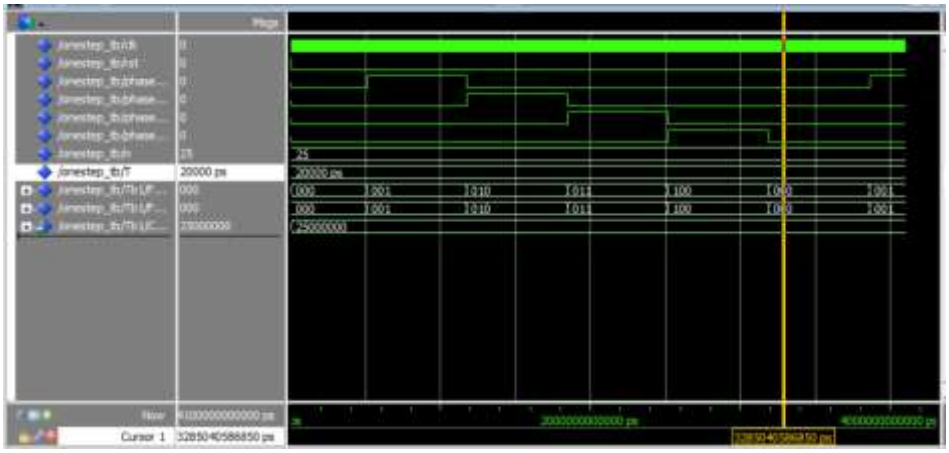

Figure 17: Stepper_Motor timing diagram

8. External Circuit

To implement our circuit onto the FPGA, we must go over the external or physical circuit. Since the ultrasonic sensor and stepper motor run on 5 volts, we needed a DC power supply to power the two components. The serial adapter didn't need any external power because it is connected to a computer that powers it. However, since we are switching between 5 volts and 3.3 volts (maximum voltage of FPGA), we needed to use two transistors and a voltage divider. The transistors are responsible for converting 3.3 volts to 5 volts. The transistors are just AND gates, so they are easy to understand. The voltage divider is exclusively for the echo signal, and it converts 5 volts to 3.3 volts. We were able to come up with the two resistor values based on the following process below.

- We know,
  V_{out} = 3.3 volts
  Vin = 5.0 volts
- Let's estimate,
  R1 = 1030 Ω
  R2 = 2000 Ω
- Voltage divider formula,
  $V_{out} = V_{in}$ [R2/(R1+R2)]
- Let's check,

3.3003 volts = (5 volts) [2000 Ω / (1030 Ω + 2000 Ω)]

As you can see, 3.3003 volts is close enough to 3.3 volts; therefore, we know that these resistor values will satisfy. To physically connect the wires to the FPGA, we had to use the GPIO pins. This is how we pass signals in and out of the circuit. The GPIO pin at index 11 acts as the ground while the others are responsible for the signal values. From index 0 to index 7, you have the following signal in order: trigger, echo, tx, rts, and the four coils. The figure below is a block diagram of the external circuit.

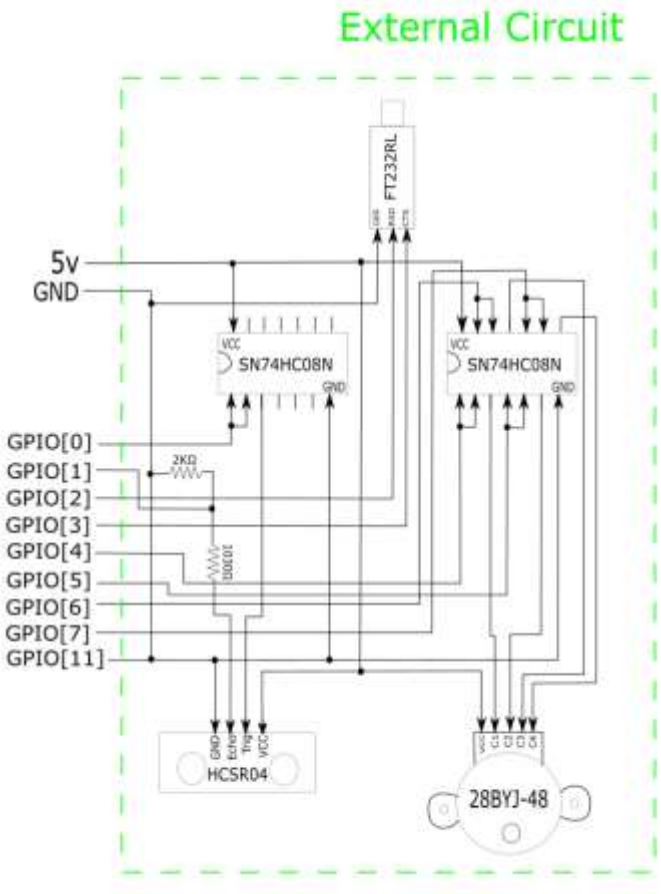

Figure 18: External Circuit block diagram

9. Results

The design was compiled for the FPGA of the family MAX 10 and device 10M50DAF48484C7G that is included in the Terasic development board MAX 10 Lite. The maximum clock that can be used is 250 MHz. The table below summarizes the resources utilized of the FPGA, and the table below that one is the RTL diagram generated by Quartus tool.

Resources	Utilized	Available
Total Logic Elements	2,289	41,910
Total Registers	192	-
Total Pins	40	499
Total Memory bits	0	5,662,720
Embedded Multipliers 9-bit Elements	0	288
Total PLLs	0	15
UFM Block	0	1
ADC Blocks	0	2

Table 8: Utilized resources of the FPGA

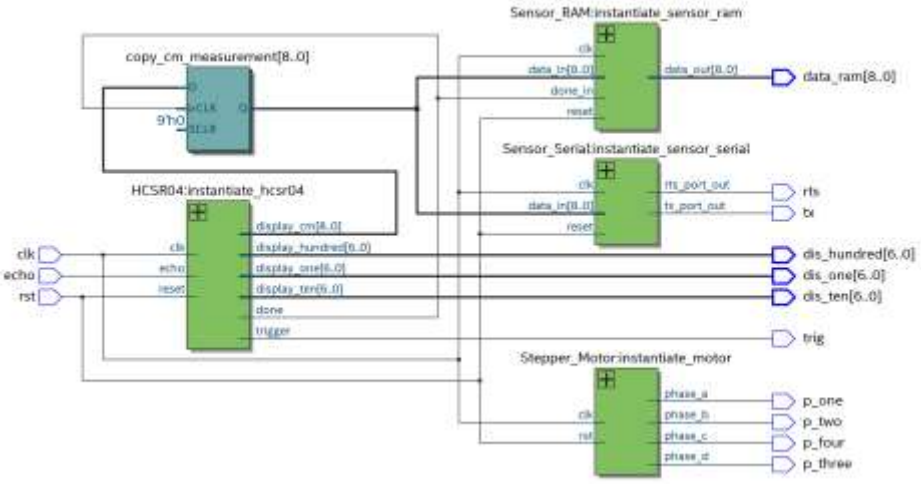

Figure 19: RTL view

Now, let's go over the results we got from our implementation. Let's start off with the ultrasonic sensor. We had a difficult time converting between 3.3 volts and 5 volts; however, after some time, we were able to figure it out. The figure below is a screenshot of the digital oscilloscope looking at the trigger signal.

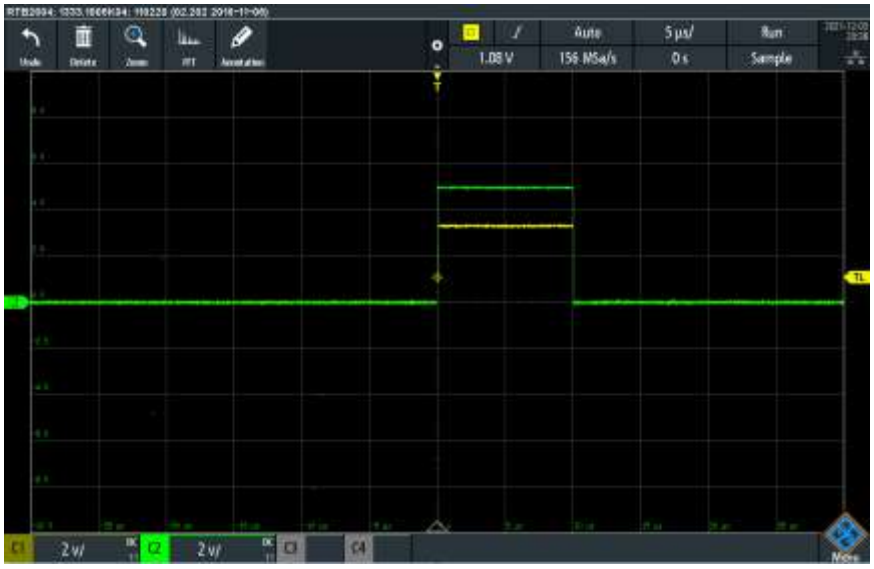

Figure 20: Ultrasonic sensor digital oscilloscope screenshot

As shown in the figure, the trigger signal goes from 3.3 volts to 5 volts without any problems. And, by looking at the bottom of the picture, the trigger signal is on for 10 microseconds, which is the time we expected it to be on. We didn't take any picture of the echo signal, but it was also working correctly. Next, let's go over the serial adapter. We are skipping the RAM since it was basically all software, which means it would have been impossible to test with a digital oscilloscope. We ran into a few problems with the serial adapter, one be the timing issue. However, after we changed the code, it was successfully able to transmit data onto the serial monitor. Originally, we used the Arduino Serial Monitor, but it was displaying unreadable values. Instead, we used a program called Serial Port Monitor, and it was successfully able to do the job. To test the serial adapter, we passed the number 55 as a binary data value. There two figures below, the first being a screenshot of the digital oscilloscope and the second being a screenshot of the Serial Port Monitor.

Figure 21: Serial adapter digital oscilloscope screenshot

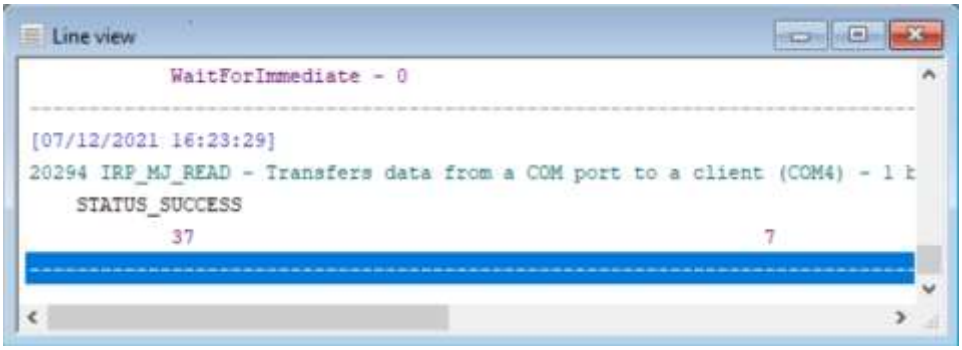

Figure 22: Serial monitor screenshot

It's a little hard to see but transmit signal (yellow) looks similar to the data value of 55 or "110111" in binary. The transmit signal is also off for 5209 cycles before sending any data, which tells the serial adapter to start reading in the data bits. Secondly, the rts signal (green) is working as expending. When data bits are being sent to through the serial adapter, the rts signal must be on. In the other figure, the serial monitor printed the hexadecimal value 37; however, in decimal representation, this value is equivalent to 55. Therefore, we know that the serial adapter is working correctly in implementation.

10. Timeline and Work Distribution

Before we began, we created a Gantt chart to distribute all the work over the next upcoming weeks. From week 8 to week 15, we had to finish the following tasks, ultrasonic sensor, stepper motor, combine components, and final presentation and report. Jack was responsible for the ultrasonic sensor, while Nimesh was responsible for the stepper motor. After we completed these tasks, we would both work on combining the

two components and building the other smaller components, such as the RAM and serial adapter. After completing this task, we would move on to the writing up the final report and presentation. The Gantt chart below shows everything, including tasks, people responsible for tasks, dates, and etc.

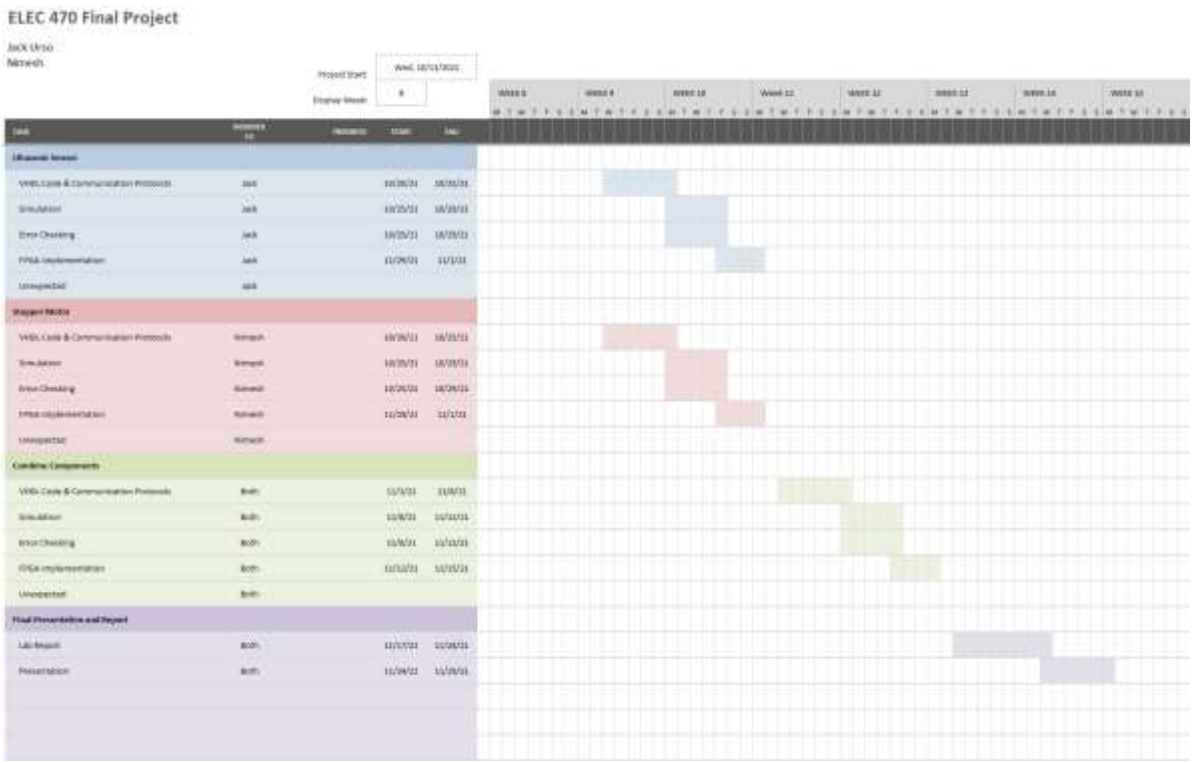

Figure 23: Gantt chart

Despite our well-structured Gantt, we had a little trouble keeping up with the due dates. Some of the tasks were moved to Week 16, such as the combining the components and the final presentation and report. Nimesh fell behind with the stepper motor task, and Jack ended up being responsible for the combine components task. For the final report, we both worked independently on the components we built ourselves; therefore, Jack did most of the report since he built the majority of the circuit. Nonetheless, we were successfully able to get everything done on time, but a lot of it was done during the last week, especially the final report and presentation.

11. Conclusions

Our circuit was designed using VHDL and implemented in an FPGA board. The design is capable of working at 250 MHz clock speed. It was tested and validated on an MAX 10 Lite Terasic board with the less than 5.5% of logic elements and 8% of output pins utilized.

Ultimately, we were able to get a somewhat functional circuit. The stepper motor was spinning, but it was doing it at a slow speed. The ultrasonic sensor was working, but it was fidgety. Both of these problems were likely due to timing issues since we didn't catch them in simulation. And, due to time constraints, were unable to fix these problems before everything had to be turned in. These were the only issues that we discovered while implementing all the components. However, despite these two small problems, our circuit did function correctly but not in the way see expected.

Lastly, after completing the final project, we both took away some invaluable skills, such as problem solving, analyzing simulations, circuit design and implementation. These skills are what employers are looking for in engineers, especially electrical and computer engineers. In addition, the final project taught us about teamwork and working together to solve a problem. Despite the overwhelming stress and anxiety, the final project has made us better engineers and that's what matters.