Design and implementation (in VHDL) of a SPI DAC interface and VGA controller to run on the Nexys 3 board

Project 1: for ECE 574

OCTOBER 9, 2014

WORCESTER POLYTECHNIC INSTITUTE
Engineer: Richard Walker

Table of Contents

Table of Figures

1 Introduction:

1.1 Project Description

This project involved the design of a number of interfaces to multiple peripheral devices. It drives a digital-to-analog converter (DAC) using a serial peripheral interface (SPI), and it also drives a video graphics array (VGA) display monitor. This project also utilized the Xilinx Core Generator and other existing IP to complete the final design.

1.2 Project Requirements

The project was divided into four major parts as defined below:

- Preliminary Part: Four-digit seven-segment display
- Part 1: Creating VGA Display using an existing VGA controller
- Part 2: Controlling a moving block on the VGA Display
- Part 3: Using a DAC to generate a sine wave

1.3 Outline

This report will discuss the requirements, system design, and implementation details, and test results of the each of the aforementioned sections in detail. Followed by a conclusion that discusses the problems faced while completing this project, in addition to the solutions to these problems. Further suggestions for improvement will also be discussed.

2 Four-Digit Seven-Segment Display:

This section of the project required the modification of a seven-segment display module that was created in a tutorial prior to this project. The original display had the capability to display a single digit on the seven-segment display. The modified module was required to be able to display the digits “0000” to “FFFF” on the four seven-segment displays.

2.1 System Requirements

1. The module should have a 16-bit input bus
2. The module should display the digits between “0000” to “FFFF”
3. The module should be a stand-alone entity

2.2 System Design

The design for this module is shown in the diagram below.

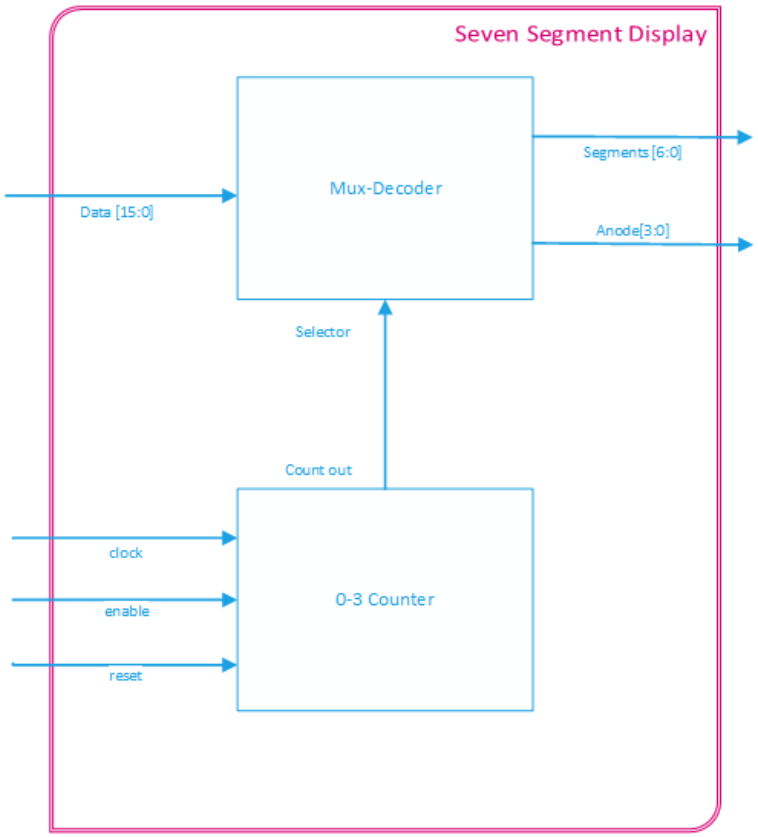

Figure 1: Seven Segment Display Module System Design

2.3 Implementation Details

The module accepts a 16-bit line as an input which is sent to a decoder and multiplexer component. The decoder splits the 16-bit signal into four 4-bit sub-signals. Each of the 4-bit sub-signals is then decoded in order to represent a hexadecimal digit to be displayed on one of the seven-segment anodes, as shown in the code sample below.

```
-- Splits the 16-bit bus into 4 4-bit parts.
-- Each part will serve at the number representation for an anode
-- AN3 AN2 AN1 AN0

      alias data_3 : std_logic_vector(3 downto 0) is data (15 downto 12);
      alias data_2 : std_logic_vector(3 downto 0) is data (11 downto 8 );
      alias data_1 : std_logic_vector(3 downto 0) is data ( 7 downto 4 );
      alias data_0 : std_logic_vector(3 downto 0) is data ( 3 downto 0 );
```

Code Sample 1: Seven Segment input splitter

```
-- Decodes the data provided by the mux
      process(data_to_seg)
      begin
            case data_to_seg is
                  when "0000" => segLED <= zero;
                  when "0001" => segLED <= one;
                  when "0010" => segLED <= two;
                  when "0011" => segLED <= three;
                  when "0100" => segLED <= four;
                  when "0101" => segLED <= five;
                  when "0110" => segLED <= six;
                  when "0111" => segLED <= seven;
                  when "1000" => segLED <= eight;
                  when "1001" => segLED <= nine;
                  when "1010" => segLED <= a;
                  when "1011" => segLED <= b;
                  when "1100" => segLED <= c;
                  when "1101" => segLED <= d;
                  when "1110" => segLED <= e;
                  when "1111" => segLED <= f;
                  when others => segLED <= off;
            end case;
      end process;
```

Code Sample 2: Seven Segment Decoder

Additionally, a 2-bit counter was connected to the select signal in order to cycle through the four anodes in the multiplexer component. Depending on the value on the select line, the corresponding anode would be enabled, and its respective digit would be displayed. The sample code for this functionality is shown below.

```
-- Selects the appropiate input data bus and
-- turns anode corresponding anode on
      process(mux_in, data_3, data_2, data_1)
      begin
            case mux_in is
                  when "11" =>
                        data_to_seg <= data_3;
                        anode <= "0111";
                  when "10" =>
                        data_to_seg <= data_2;
                        anode <= "1011";
                  when "01" =>
                        data_to_seg <= data_1;
                        anode <= "1101";
                  when others =>
                        data_to_seg <= data_0;
```

```
                    anode <= "1110";
        end case;
    end process;
```

Code Sample 3: Seven Segment Anode Mux

2.4 Test Results

A test bench was created to test the functionality of the module. The following waveforms verify that the module performed as expected.

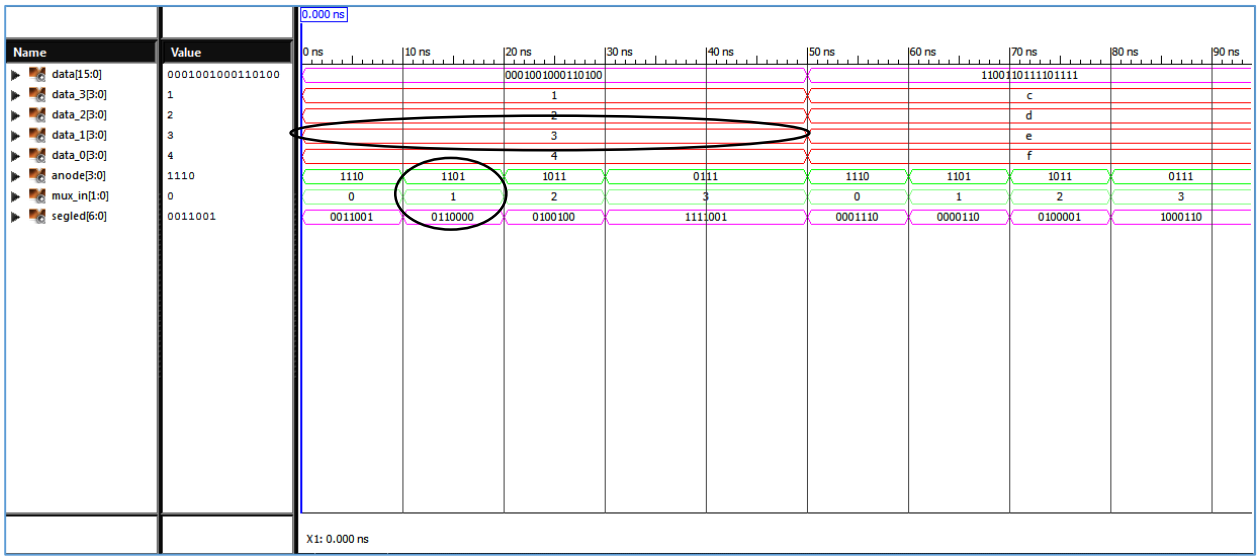

Figure 2: Seven Segment Test Bench Results

As shown in the diagram above the `segled` bus sets the bits low for the LEDs that correspond with the current `anode` and `data` signals. As an example, when the `mux_in` is 1 (in decimal), `anode[1]` is enabled and the digit on `data_1` (which is 3 in hexadecimal) is represented on the `segled` bus by driving all the cathodes low except for cathode ‘e’ and cathode ‘f’.

3 Creating VGA Display using an existing VGA controller:

This section of the project required the design of a module that could create a VGA display by using an existing VGA controller provided by Digilent. The display was required to be able to change colors and images depending on specific inputs from the user. Additionally, a digital clock manager (DCM) was required to be created to drive the VGA controller.

3.1 System Requirements

VGA Display Module

1. The module should use the VGA controller provided by Digilent
2. The display should have a resolution of 640 x 480 pixels
3. The module should have inputs to display the following patterns
   3.1. Completely blue display
   3.2. Completely blue display
   3.3. A black screen with a white rectangle 10 pixels in from the outside edges (one pixel wide)

DCM

1. The DCM should produce a 25MHz clock signal
2. The 25MHz clock signal should drive the VGA controller

3.2 System Design

The top-level design for this section is shown in the diagram below.

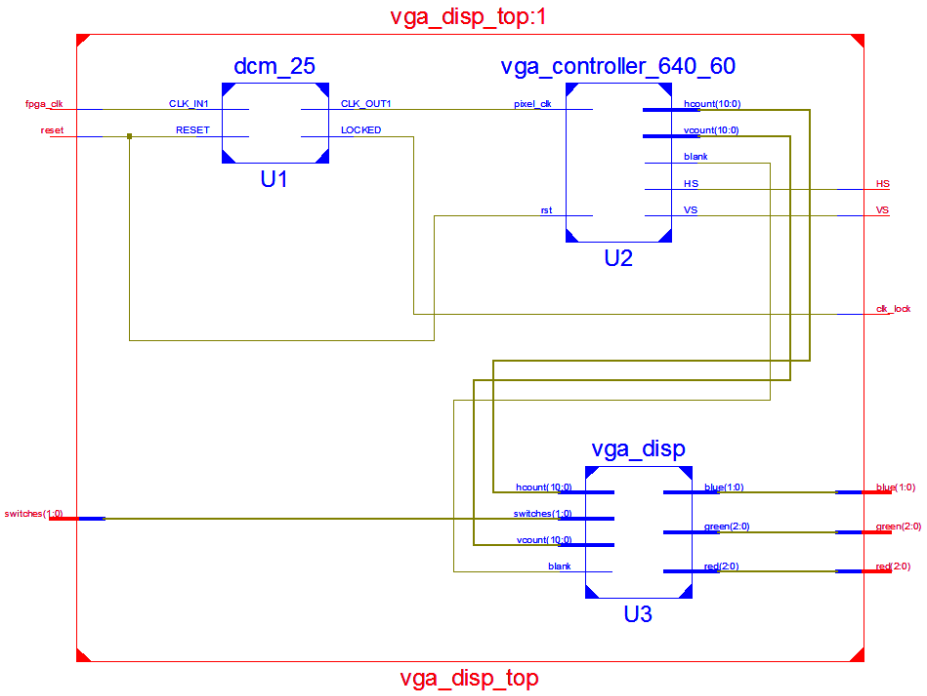

Figure 3: VGA Display Top-Level Design

The DCM converts the clock signal provided by the FGPA board into a 25 MHz signal that is used by the VGA controller. The VGA controller provides the horizontal sync (HS) and vertical sync (VS) outputs to an external monitor, the blank, horizontal count, and vertical count outputs are used as inputs to the VGA display module. The VGA display module outputs the corresponding red, green, and blue color signals to an external monitor based on the horizontal and vertical count inputs, and switch inputs provided.

3.3 Implementation Details

The VGA display was the only module in this section that had to be designed, as mentioned before. The module was implemented as a purely combinational design. The `switch` signal has only 4 states, three of the states were used to display the colors specified in the requirements, and the unused state displayed a black screen. It is also important to note that a monitor will not display an image when the blank state is on. The code sample below shows this implementation using a `case` statement.

```
-- Changes the screen to be either blue or green
-- depending on the switch input
      process (switches, blank, hcount, vcount)
      begin
            if blank ='0' then
                  case switches is
                        -- Make the screen blue
                        when BLUE_C => BLUE;
                        -- Make the screen green
                        when GREEN_C => GREEN;
                        -- Draw a white rectangle on the screen
                        when RECT_C => -- Omitted for clarity
                        -- if not blue or green, or a rectangle,
                        -- make the screen black.
                        when others => BLACK;
                  end case; -- End the switch-screen case
      end process;
```

Code Sample 4: VGA Display Case Statement

To implement the white rectangle display, the boarders of the rectangle had to be determined, additional logic was needed to accomplish this. The following flowchart illustrates how the boarders were formed.

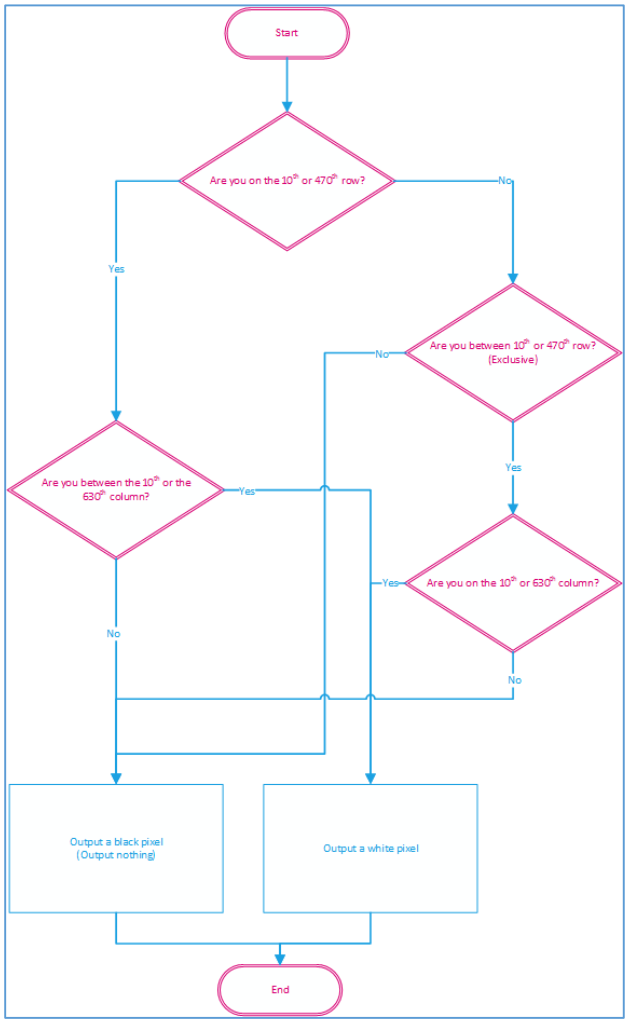

Figure 4: White Rectangle Decision Chart

The code used to implement this is shown below.

```
-- Check if vcount is on 9th row or the 470th row --
      if (vcount = Y1) OR (vcount = Y2) then
      -- Draw a white horizontal line starting from the 10px
            if (hcount >= X1) AND (hcount <= X2) then <=
                  px <= WHITE;
            else
                  px <= BLACK;
            end if;

      -- If vcount_sig is between the 10th row and the 470th row
      elsif (vcount >= Y1) AND (vcount < Y2) then
      -- Draw white pixel only on 9th coloumn and 630th column --
            if (hcount = X1) OR (hcount = X2) then
                  px <= WHITE;
            else
```

```
                px <= BLACK;
        end if;

-- It's not on the perimeter of the rectangle, set it to black
else
        px <= BLACK;
end if;       -- END DRAW RECTANGLE IF
```

Code Sample 5: White Rectangle Drawer

3.4 Test Results

The VGA display was tested on the FPGA board and as shown in the pictures below, the display changed depending on the input given.

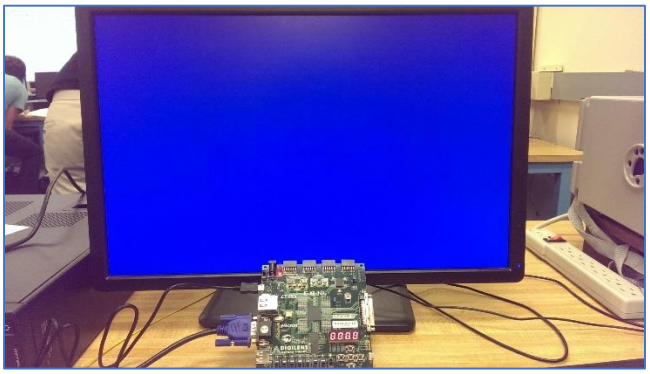

Figure 5: Blue VGA Display

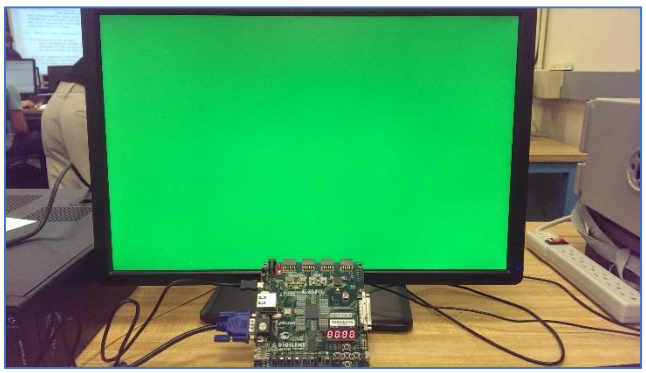

Figure 6: Green VGA Display

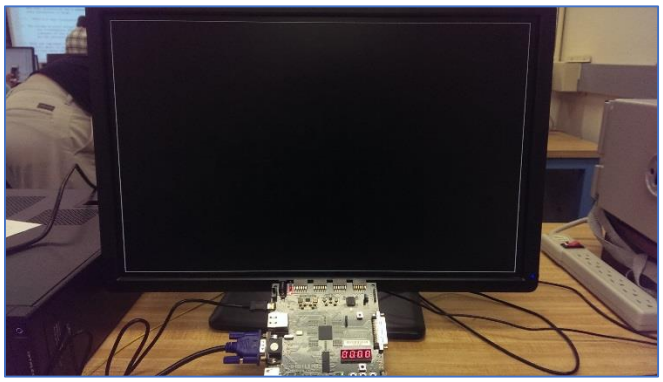

Figure 7: White Rectangle VGA Display

4 Controlling a moving block on the VGA Display:

This section of the project required a movable block to be displayed on a monitor. The 640 x 480 display was divided into 400 segments, each 32 pixels wide by 24 pixels high. A yellow block was displayed in an initial segment, and the block was able to move between positions depending on button presses provided by a user.

4.1 System Requirements

1. A yellow block should be displayed at a starting position determined by the value of the slider switches
   1.1. Three switches should be used for the x-position
   1.2. Three switches should be used for the y-position
2. Four separate pushbuttons should move the block either up, down, left, right at a time when pressed
3. The yellow block should not be able to move further if it hits the outside of the display
4. The current position of the block should be displayed on the seven segment display
   4.1. Two digits should be used for the x-position
   4.2. Two digits should be used for the y-position

4.2 System Design

The top-level design for this section is shown in the diagram below.

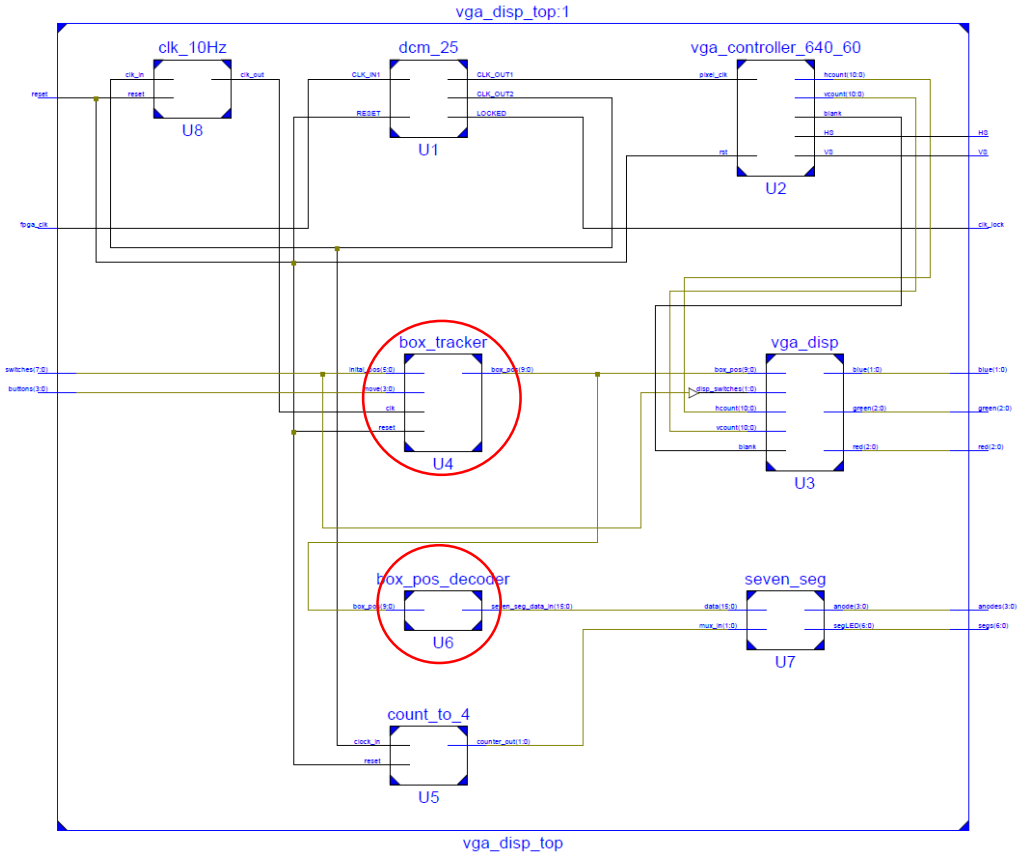

Figure 8: Moveable Block Top-Level Design

To accomplish the functionality for this section of the project, the `box_tracker` module (U4) was created to interface with a modified version of the `vga_disp` module (U3) that was discussed in Section 3, by providing the current position of the yellow block. Additionally, the `box_pos_decoder` module (U6) was created as a module that decoded the box's current position and converted it into a 16-bit input for the seven-segment display module which was discussed in Section 2.

4.3 Implementation Details

As mentioned above, `vga_disp` module was modified. The switch inputs that determined which screen would be displayed on the monitor was given additional functionality. The unused state (the black screen) became background for the yellow block.

A 10-bit input was that contained the block's current x and y positions was also added to the module. The positions, which are coordinates between 0-19, were multiplied by the block's height and width in order to determine the area of the block, as shown in the code sample below.

```
-- Sets the position of the box
      process (box_pos)
      begin
            -- x-coordinate * box width
            x_pos <= box_pos(9 downto 5) * BK_X;
            -- y-coordinate * box height
            y_pos <= box_pos(4 downto 0) * BK_Y;
      end process;
```

Code Sample 6: Determining the block's area

Lastly, formally unused state now checked the vertical count then and horizontal count to determine where the block should be displayed. The implementation is shown in the sample code below.

```
-- if the vcount and hcount are more than the starting position and
-- less than (the start position + the height & width) of the box
-- then that's the area that needs to be filled.
if ( (vcount >= y_pos) AND (vcount < (y_pos + BK_Y)) AND
     (hcount >= x_pos) AND (hcount < (x_pos + BK_X)) ) then
      color <= YELLOW_PX;
else
      color <= BLACK_PX;
end if;
```

Code Sample 7: Displaying the block's position on the screen

4.3.1 Box Tracker Module

The box tracker module contained two major components: a state machine that determined which direction the yellow block should be moved, and two bi-directional counters that stored the current x and y positions of the block (as a coordinate between 0-19).

The state machine that was used is represented by the diagram below.

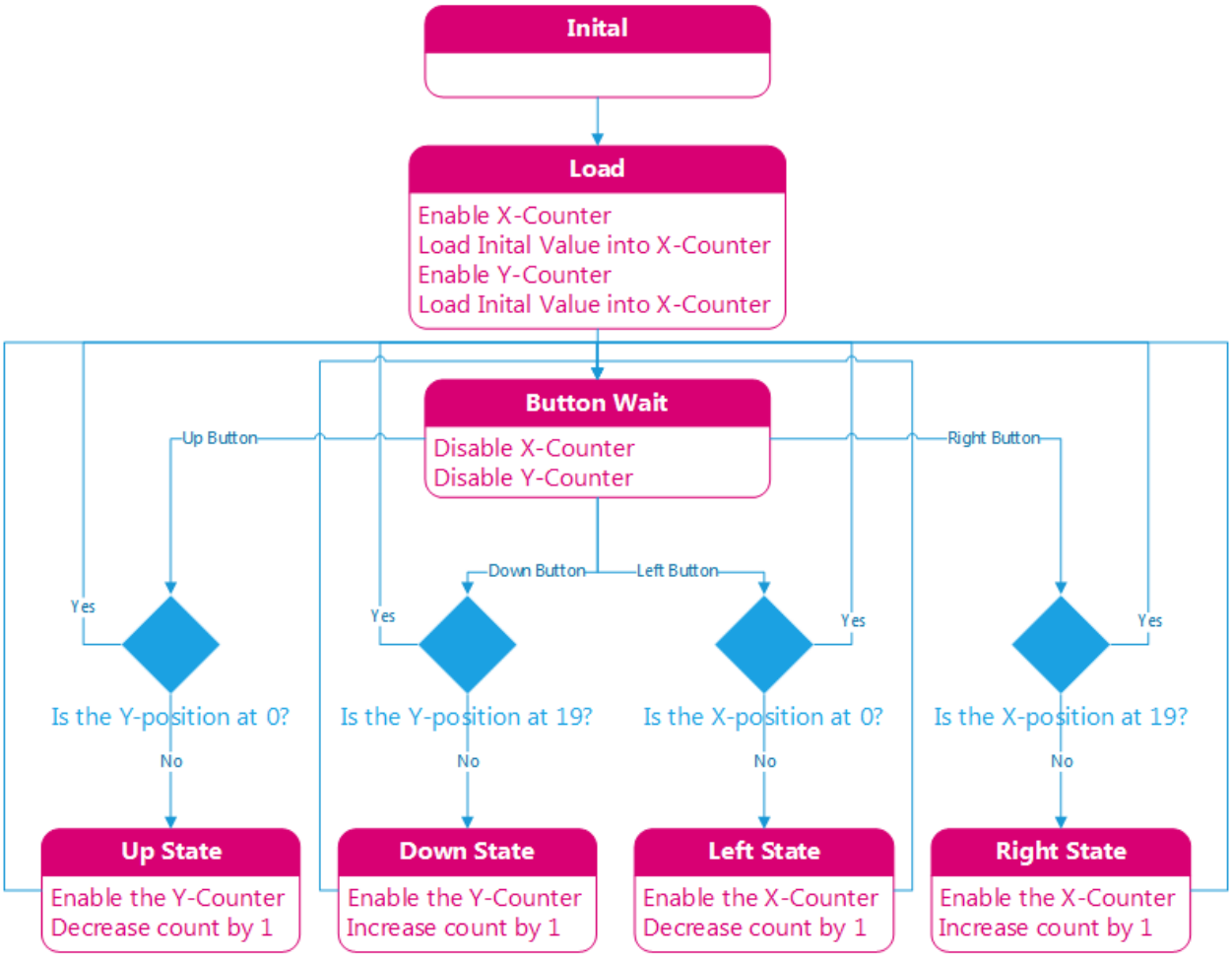

Figure 9: Box Tracker State Machine

The bi-directional counters were implemented as two separate modules. Depending on which state the state machine was currently in determined which directions the counter would count in. The sample code for the counters is shown below.

```
process(clk, reset, srt_pos, en, pre)
begin
	-- if reset, set the counter at the initial value
	if reset = '1' then
		count <= 0;
	-- Statements for clk edge
	elsif rising_edge(clk) then
		-- preset high, load the inital value
		if (en = '1' and pre = '1') then
			count <= conv_integer(srt_pos);
		-- counting up
		elsif (en = '1' and pre = '0' and dir = '1') then
			if count = 19 then
				count <= 0;
			else
				count <= count + 1;
```

```
                end if;
            -- counting down
            elsif (en = '1' and pre = '0' and dir = '0') then
                if count = 0 then
                    count <= 19;
                else
                    count <= count - 1;
                end if;
            end if;
        end if;
end process;
```

Code Sample 8: Bi-directional Counter

4.3.2 Box Position Decoder

The box position decoder consists of three concurrent processes: one that splits the input data into two 5-bit segments, another which decodes the block's x -position, and the last process decodes the y-position. The x and y decoder processes first determines if their respective positions are greater than 9, if this is true, then it sets the digits that correspond to the fourth and second anode as '1', else it sets them as '0', as shown in the code below.

```
-- if x_position less than 10, then set digit_1 to display 0
if (x_pos < "01010") then
    digit_3 <= "0000";
-- else it's more than 9, so set digit_1 to display 1
else
    digit_3 <= "0001";
end if;

-- if y_position less than 10, then set digit_1 to display 0
if (y_pos < "01010") then
    digit_1 <= "0000";
-- else it's more than 9, so set digit_1 to display 1
    else
    digit_1 <= "0001";
end if;
```

Code Sample 9: Determines the digit for the second and fourth anodes

The x and y decoder then determines the digit to be shown on the first and third anode as shown in the code below.

```
-- decode to determine what digit_0 should be
case x_pos is
    -- 0 to 9
    when "00000" => digit_2 <= "0000";
    when "00001" => digit_2 <= "0001";
    when "00010" => digit_2 <= "0010";
    when "00011" => digit_2 <= "0011";
    when "00100" => digit_2 <= "0100";
    when "00101" => digit_2 <= "0101";
    when "00110" => digit_2 <= "0110";
    when "00111" => digit_2 <= "0111";
    when "01000" => digit_2 <= "1000";
    when "01001" => digit_2 <= "1001";
```

```
        -- 10 to 19
        when "01010" => digit_2 <= "0000";
        when "01011" => digit_2 <= "0001";
        when "01100" => digit_2 <= "0010";
        when "01101" => digit_2 <= "0011";
        when "01110" => digit_2 <= "0100";
        when "01111" => digit_2 <= "0101";
        when "10000" => digit_2 <= "0110";
        when "10001" => digit_2 <= "0111";
        when "10010" => digit_2 <= "1000";
        when others  => digit_2 <= "1001";
end case;
```

Code Sample 10: Determines which digit to be displayed on the 1st and 3rd anodes

Finally, the decoder combines the four 4-bit digits into a 16-bit digit that is used at the input to the 7-segment decoder, as shown below.

```
-- Combines the digit data into a 16-bit bus
seven_seg_data_in <= digit_3 & digit_2 & digit_1 & digit_0;
```

Code Sample 11: Digits Combination

4.4 Test Results

4.4.1 Box Tracker Module

A test bench was created to verify the functionality of the box tracker module. The waveform below illustrate that the box tracker performed as specified.

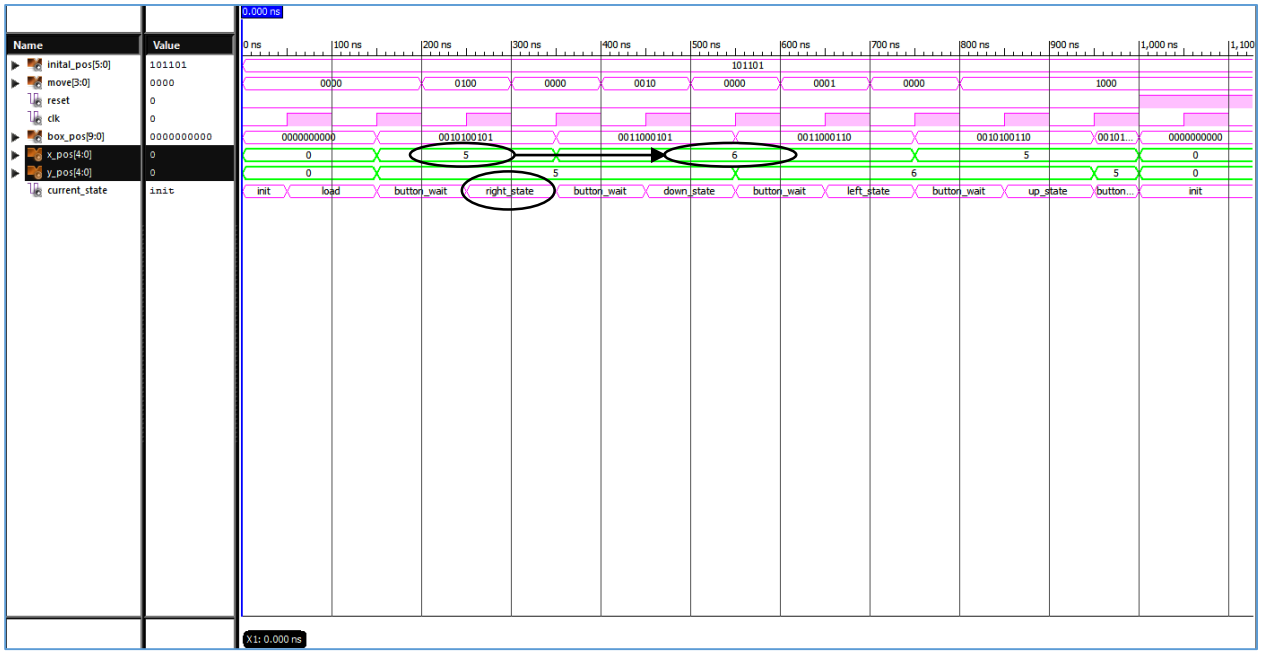

Figure 10: Box Tracker Waveform

As seen in the diagram above, the x and/or y position of the box changed respective to the current state of the box tracker.

Additionally, the boundary conditions were tested and verified as shown in the waveform below.

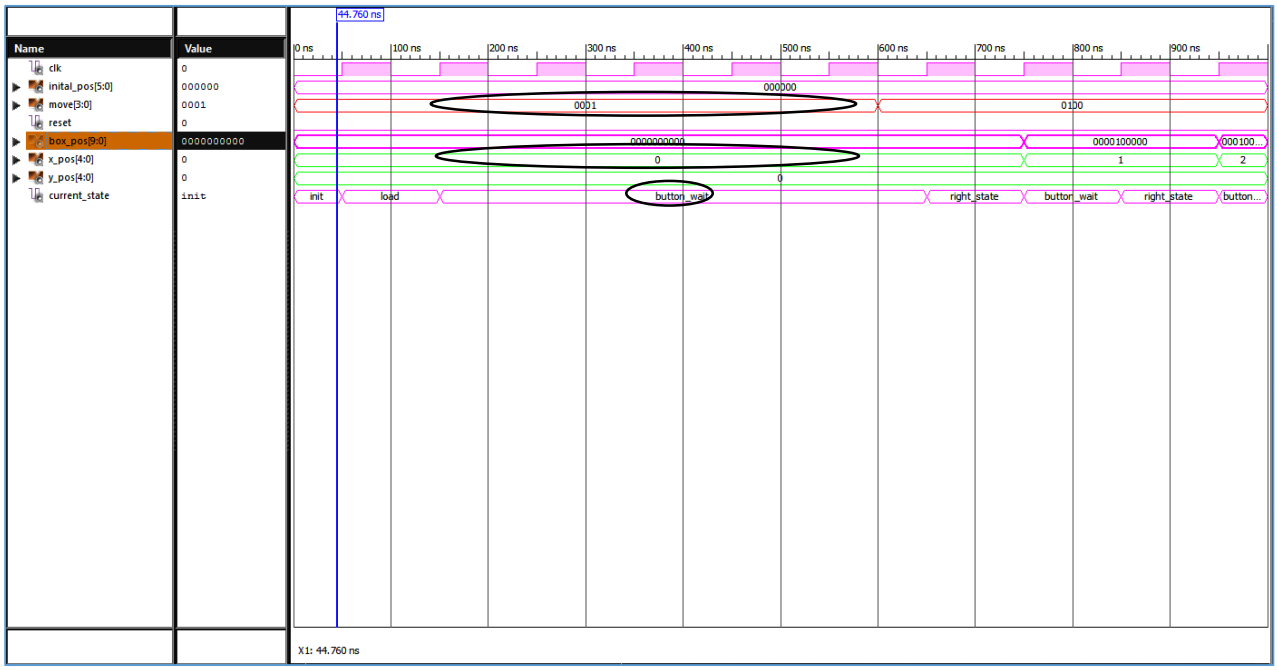

Figure 11: Testing Box Tracker Boundary Conditions

In the diagram above, the move button has a value of “0001” which represents left pushbutton being pressed. The button is held for six cycles, but the current x-position is at 0, the box should not be able to move left any further, hence the state machine stays in the button wait state.

4.4.2 Box Position Decoder

A test bench was also created to verify that the box position decoder functioned correctly, as shown in the waveform below.

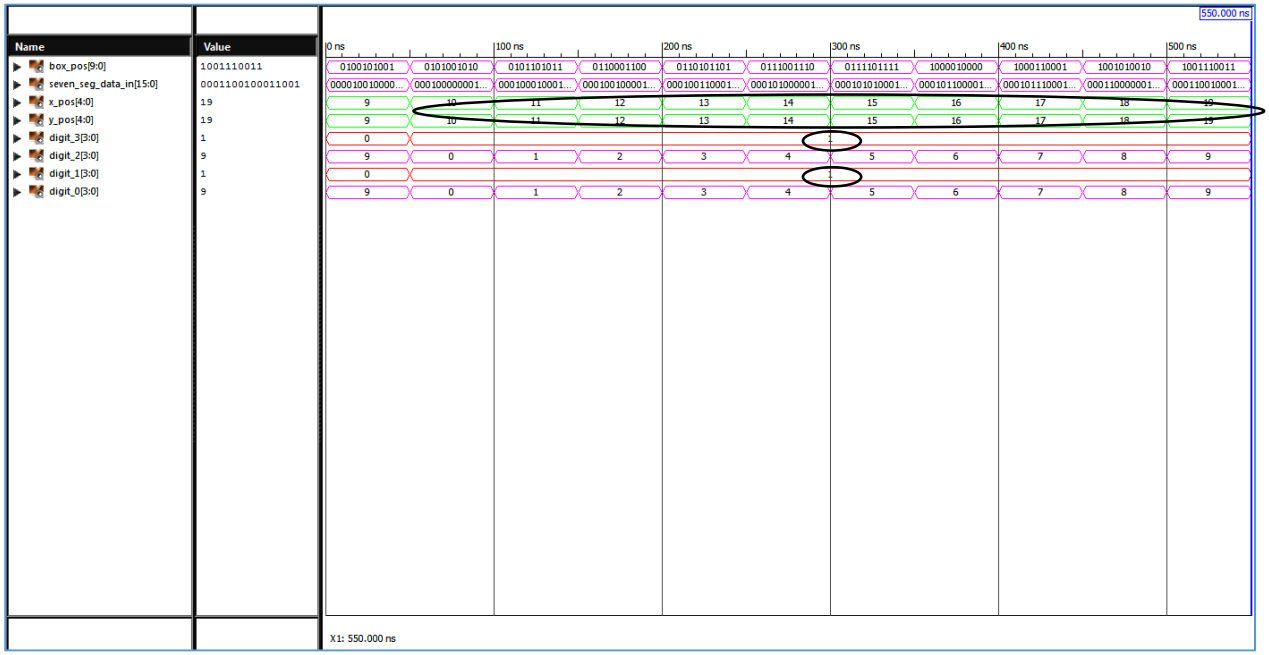

Figure 12: Testing the Box Position Decoder

As seen in the figure above, when the x or y position is greater than 9, `digit_3` and `digit_1` are both equal to 1, while the `digit_2` and `digit_0` contain the correct value of the block's position. Lastly, it is also shown that `seven_seg_data_in` correctly combines `digit_3`, `digit_2`, `digit_1`, and `digit_0` in order to provide a 16-bit output for the seven segment display decoder.

4.4.3 Combined Result

The figure below shows the combined implementation of the box tracker and box position on the FPGA board.

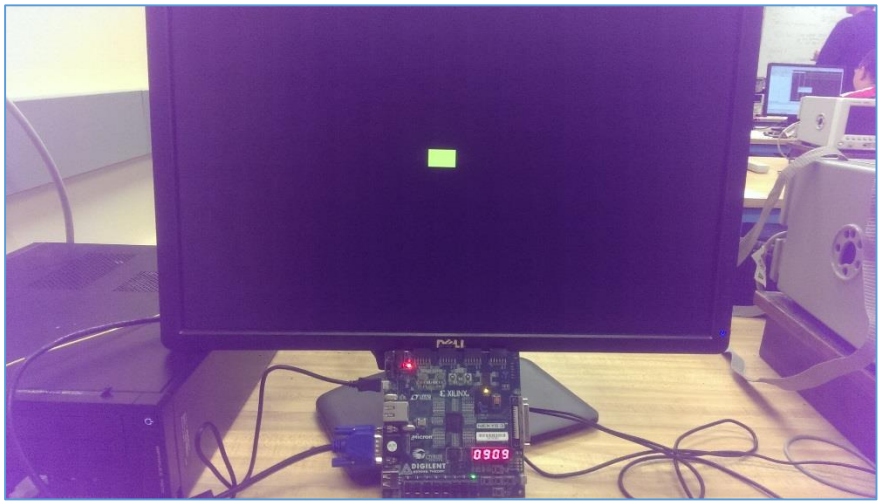

Figure 13: Box Tracker with 7 Segment display

5 Using a DAC to generate a Sine Wave:

This section of the project required the creation of a 6.25 kHz sine wave using a DAC. First, it was required to use the DAC to create a constant (DC) voltage, then modify the design in order to replicate the sine wave. The DAC that was used to create the voltage waveforms was the AD703 Serial Input, Dual 8-bit DAC provided to us for this project.

5.1 System Requirements

1. The DAC to be used should be the PMOD DA1 Module that was provided
2. The DAC interface should be implemented with a state machine and shift register
3. The DAC should use 16 constant values in the range (0 to 255) to generate a 6.25 kHz sine wave
   3.1. The values should be loaded into the DAC at a rate of 100 kHz

5.2 System Design

The top-level design for this section is shown in the diagram below.

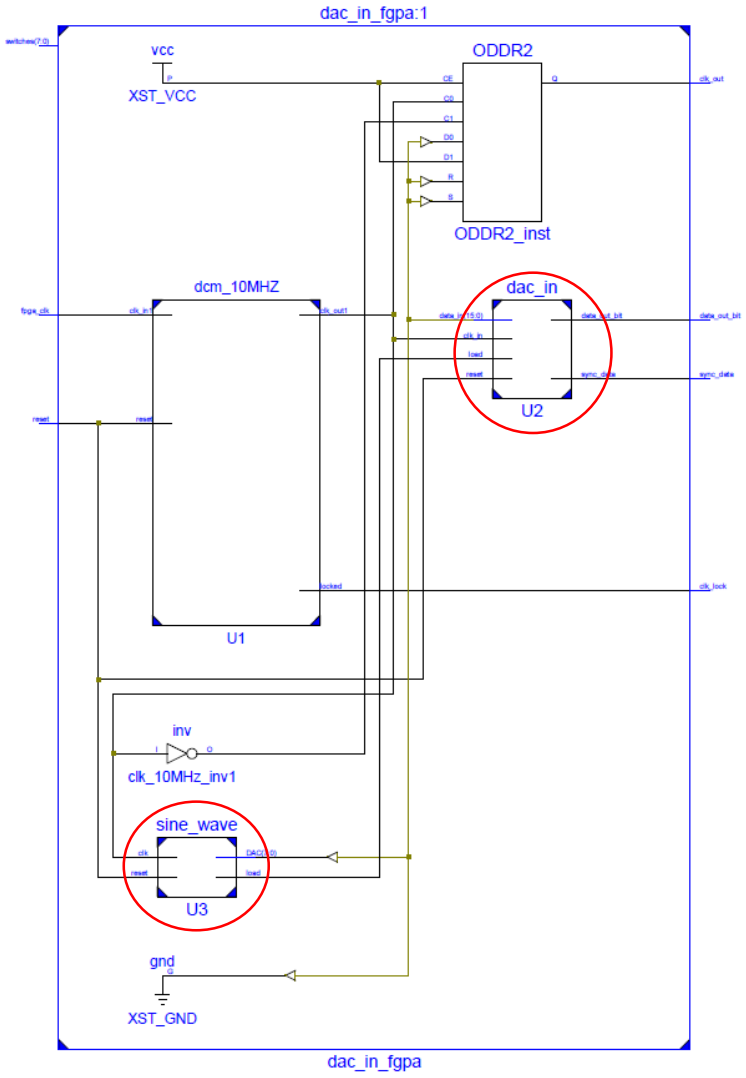

Figure 14: Sine Wave with DAC Top-Level Design

The creation of a sine wave was separated into two main modules, a sine wave creation module, and a DAC interface module. The sine wave module provided the DAC interface with the sixteen 8-bit values needed to produce a sine wave. It also has a load output that signals to the DAC when the 8-bit data value should be loaded. The DAC interface provides the sync signal and the data bits that are needed for the AD703 DAC.

The DAC interface consisted of a state machine, a 16-bit shift register, and a 0-15 counter as shown in the diagram below.

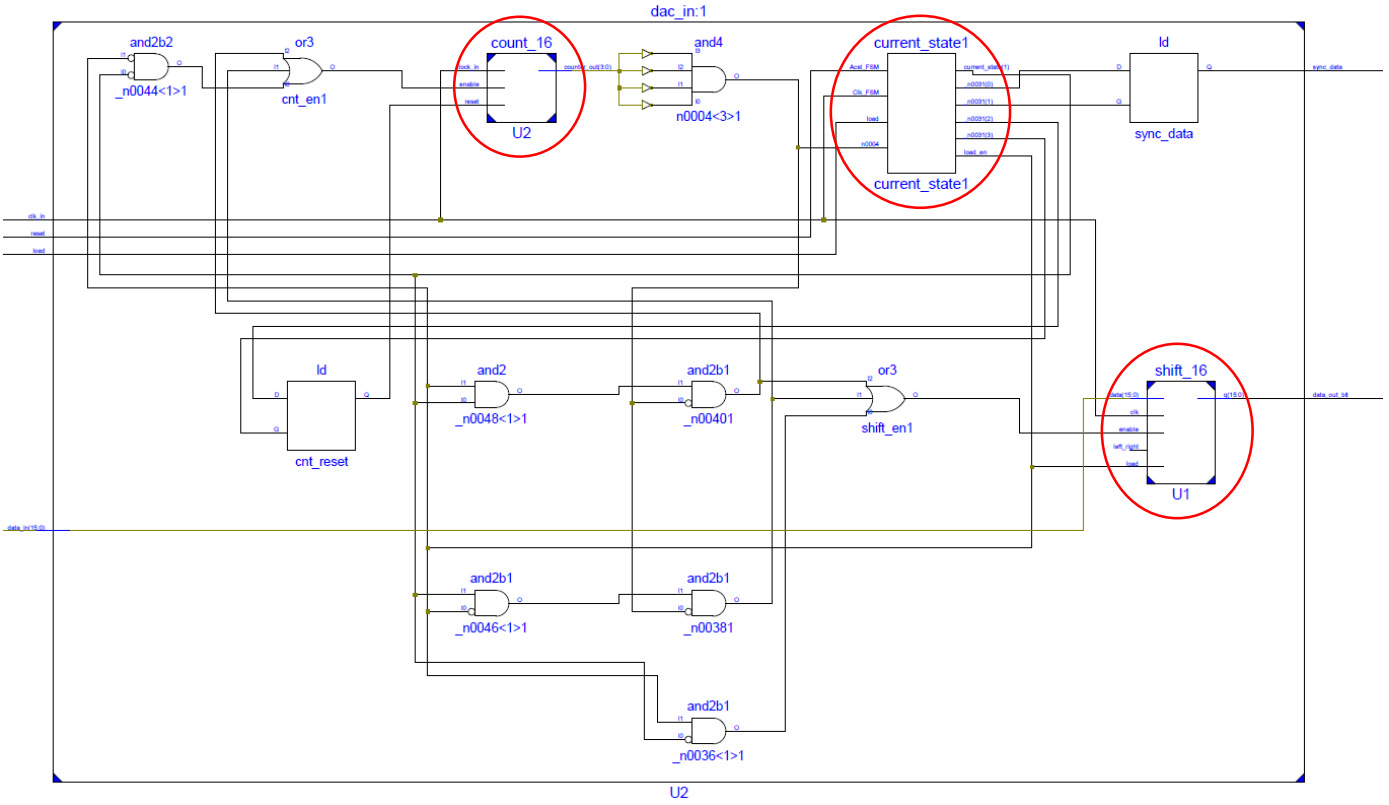

Figure 15: DAC Interface System Design

Lastly, the sine wave generator interface consisted of a 0-99 counter, a 0-15 counter and, a 16-to-1 multiplexer as shown in the diagram below.

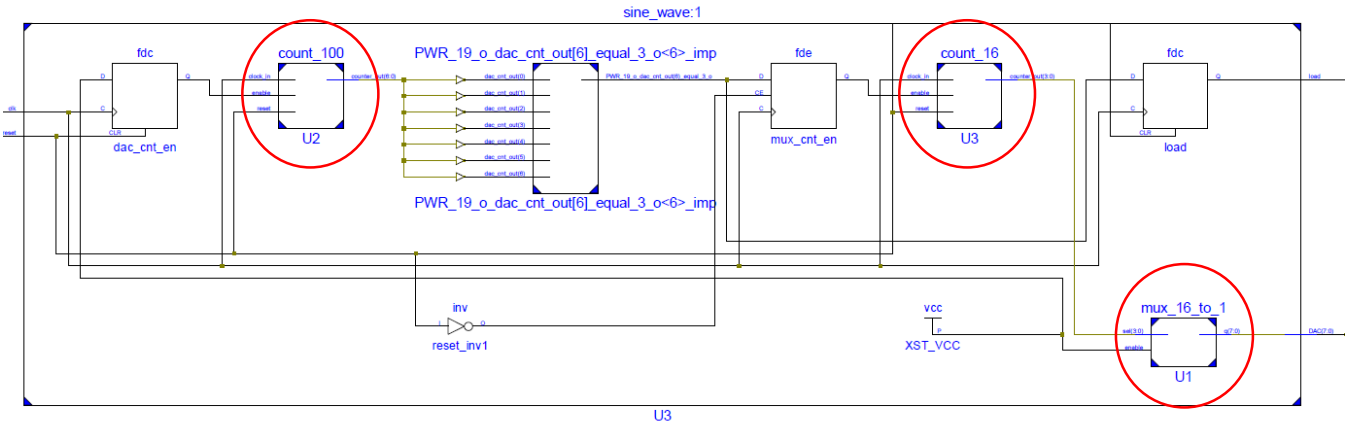

Figure 16: Sine Wave Generator System Design

5.3 Implementation Details

In order to implement a sine wave voltage, the design was broken down into two separate modules. One module was responsible for creating a voltage depending on its input data, and the other module provided the values needed to create the sine wave.

5.3.1 DAC Interface

The DAC interface was implemented by using a state machine in conjunction with a shift register and counter. The figure below illustrates the operation of the module.

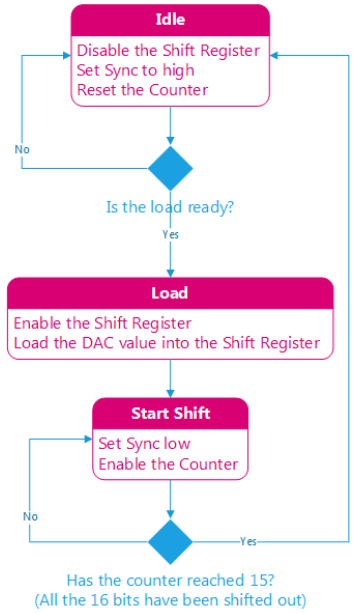

Figure 17: DAC interface State Machine

5.3.2 Sine Wave Generator

The Sine Wave generator was implemented by using a 16-to-1 multiplexer in conjunction with two counters. The multiplexer was responsible for holding the 16 constant values needed to produce a sine wave, and it outputted the next value needed for the DAC interface depending on its select signal which was driven by a 0-15 counter.

The main process of the Sine Wave generator included a 0-99 counter that determined when 100 cycles of the 10 MHz (100 kHz) has occurred. After 100 cycles, the new value for the sine was provided for the DAC, as demonstrated by the code sample below.

```
-- if the count reaches 99 then activate the mux counter
-- and load the signal
if (dac_cnt_out = "1100011") then

      -- enable the mux counter
      mux_cnt_en <= '1';

      -- Load a value to the DAC
      load <= '1';

-- else the DAC counter did not reach 99
else

      -- disable the mux counter
      mux_cnt_en <= '0';

      -- don't load a value to the DAC
      load <= '0';

end if;
```

Code Sample 12: Sine Wave Generator Main Process

5.4 Test Results

The DAC interface module and the Sine Wave generator module were tested as separate units. After confirming the correct operation of each unit, the units were combined and final tests for performed to verify the intended functionality.

5.4.1 DAC Interface

The DAC interface was tested to produce constant values initially. The correct operation of the Sync and D1 signals had to be confirmed. The following test bench verifies the functionality of the module.

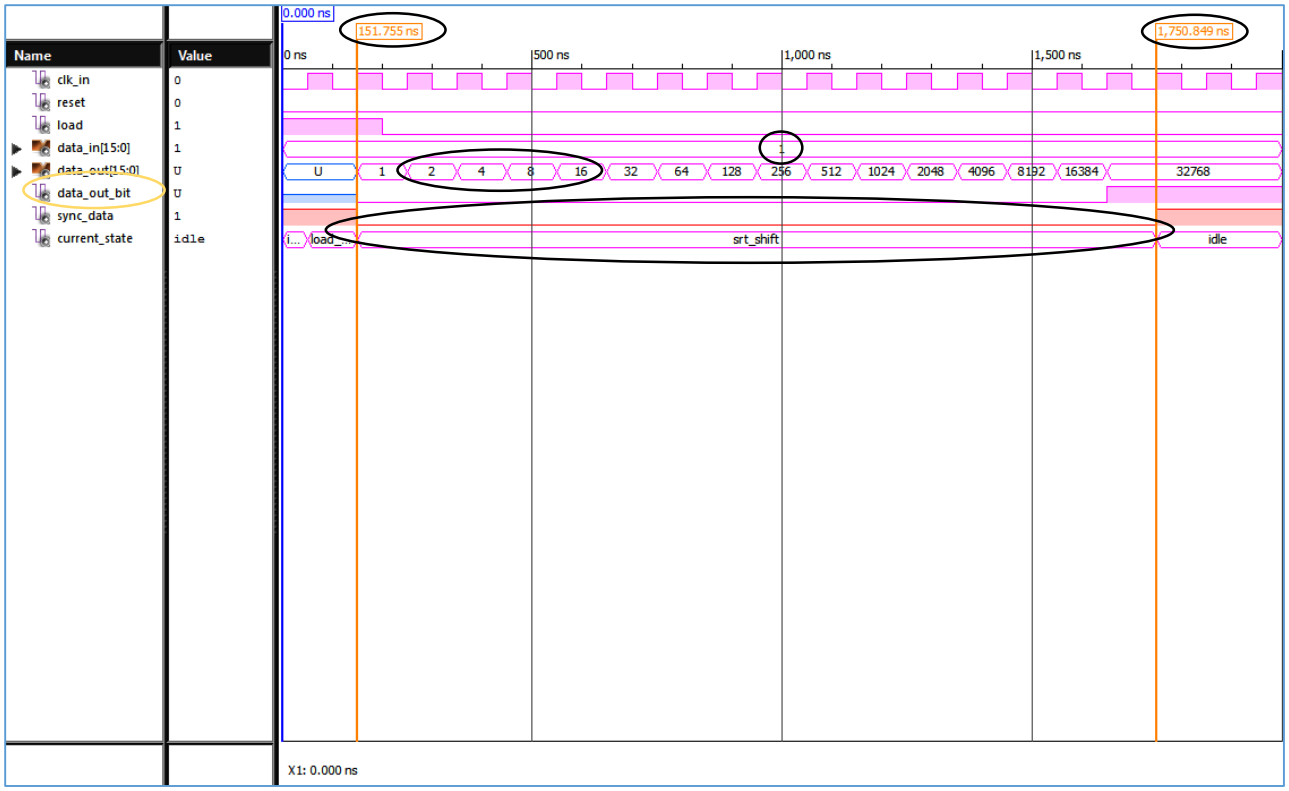

Figure 18: DAC interface showing proper functionality

The `data_in` was set as 1 for testing, because a shift left represents a multiplication by 2 in decimal. This made it easier to verify that the shift register was working correctly. As shown above, when the DAC interface is in the shifting state, the shift register is enabled and the sync is set low for 1600 ns (16 clock cycles). After leaving the shift state, the sync is set back to high, and the shift register stops shifting.

The DAC interface was also tested on the FGPA to verify it could provide a constant voltage value, the following oscilloscope waveforms and figures verify this operation.

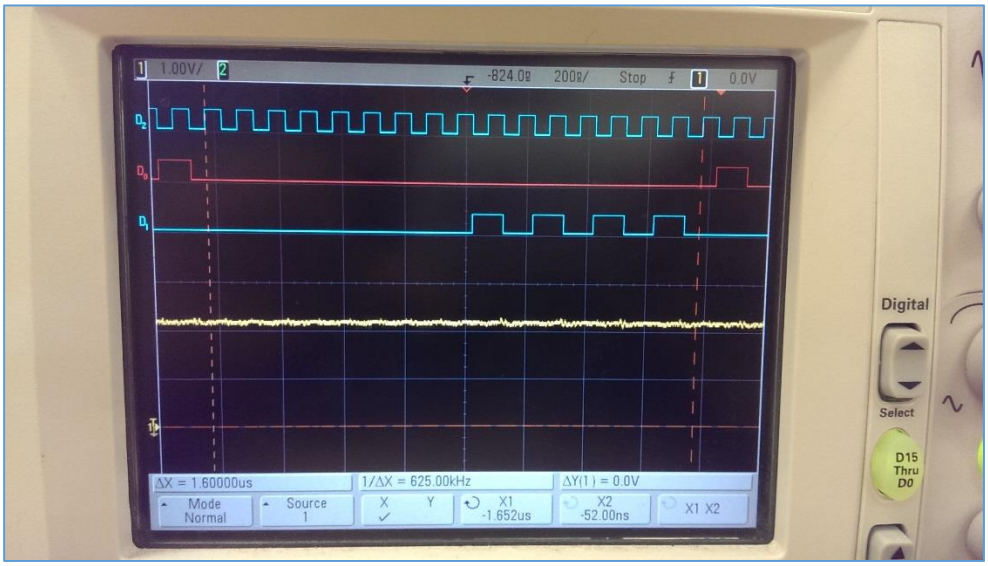

Figure 19: DAC Verification on the Oscilloscope

As shown in the figure above, the 16 bits are transferred to the DAC within 1.6 µs (16 clock cycles). The first 8 bits are the control bits and they were set to be all '0's, the last 8 bits were "10101010" which represented 170 in decimal, which corresponds to a 2.19 V voltage.

5.4.2 Sine Wave Generator

A test bench was created to verify that the Sine Wave generator produced the correct values. The results of the test bench is shown in the figure below.

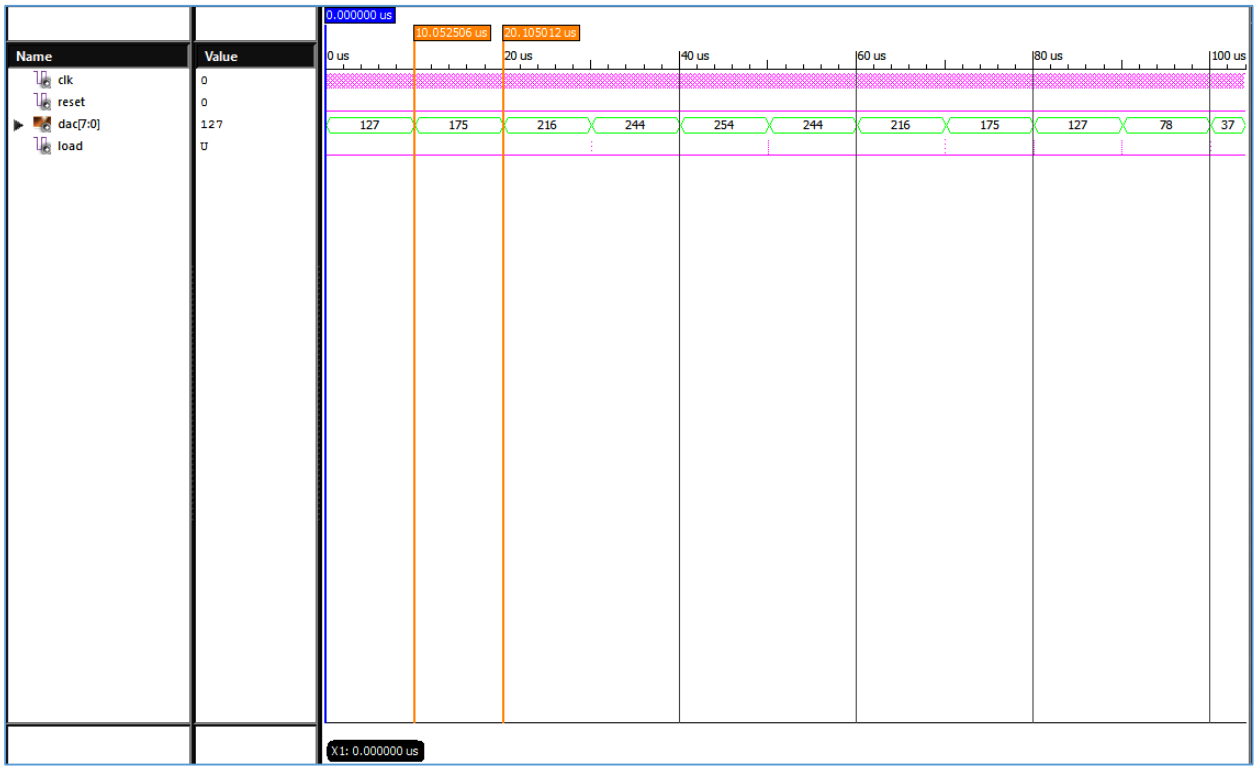

Figure 20: Sine Wave Test Bench

The markers in the figure show that a new value is produced by the sine wave generator every 10 µs (100 kHz). Please note that the values for the sine wave shown are the decimal equivalents.

5.4.3 Combined Result

The combined result of interfacing the DAC interface module with the Sine Wave Generator is shown in the figure below.

Figure 21: 6.25 kHz Sine Wave generated by the DAC

As shown by the markers, the frequency of the sine wave is indeed 6.25 kHz as specified.

6 Conclusion:

6.1 Overall combination

All the parts of the project were combined into a single module and it was demonstrated to the teaching assistant. A system diagram of the combined project is shown below.

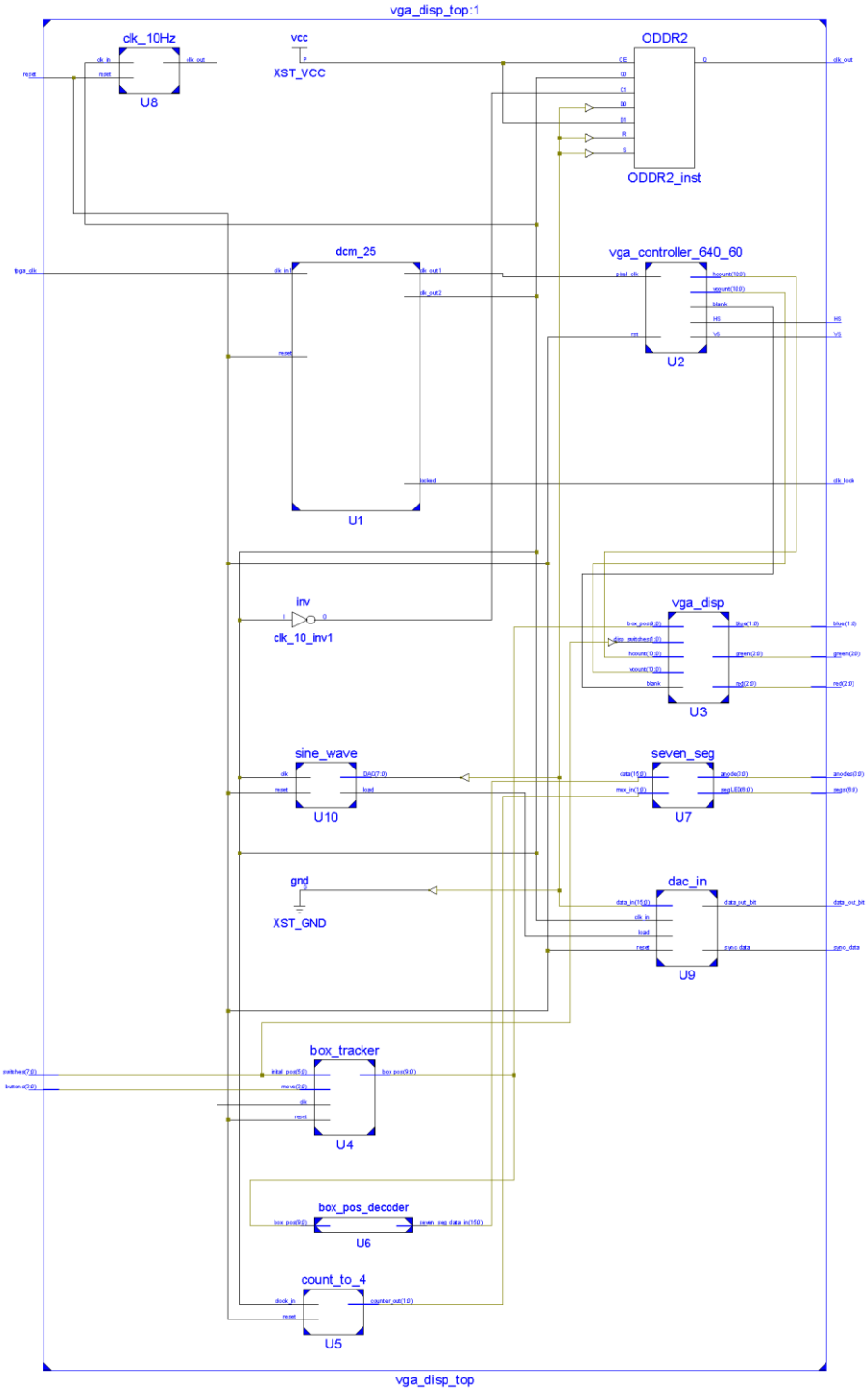

Figure 22: Overall Project System Diagram

6.2 Problems and solutions faced during implementation

There were two main problems that were faced while completing this project:

- Debouncing the buttons for the box tracker module
- Configuring the DAC correctly

The box tracker module was reading the push buttons input too quickly at first, this resulted in the yellow block moving multiple spaces during one press of the button. To solve this problem, I created a slower clock (10 Hz) that was used as the clock for the module.

Initially when I made the DAC interface module, I was under the impression that the DAC loaded the data bits before the control bits, this resulted in the DAC interface module providing the data to the DAC in reverse order. After careful inspection on the oscilloscope, I saw this error, and realized that I needed to use a shift-left register instead of a shift-right register.

6.3 Lessons learned from the project

I learned the numerous ways that a FPGA can interact with various peripheral devices. I also learned how a VGA display functions, and how to use a VGA controller to drive a display. Additionally, I learned more about VHDL, and some of the intricacies of the language.

6.4 Suggestions and Improvements

Thankfully, I was able to meet all the requirements set by this project, however there are additional functionalities that could be added in the future in order to make the project a bit more interactive. A feature that would be interesting to add to the project would be the ability to change all the frequency of the sine wave depending on the position of the box on the screen. Also, it would be curious to be able to attach a speaker so one is able to hear the produced sine wave.

7 Design Summary/Reports:

7.1 FPGA Resource Usage

The table below shows the resource usage of the final project.

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# RAMs                                                 : 5
 16x7-bit single-port Read Only RAM                    : 1
 16x8-bit single-port Read Only RAM                    : 1
 32x4-bit single-port Read Only RAM                    : 2
 4x4-bit single-port Read Only RAM                     : 1
# Multipliers                                          : 1
 5x5-bit multiplier                                    : 1
# Adders/Subtractors                                   : 13
 11-bit adder                                          : 2
 16-bit adder                                          : 2
 2-bit adder                                           : 1
 32-bit adder                                          : 1
 4-bit adder                                           : 2
 5-bit addsub                                          : 2
 7-bit adder                                           : 1
 9-bit adder                                           : 2
# Registers                                            : 20
 1-bit register                                        : 8
 11-bit register                                       : 2
 16-bit register                                       : 1
 2-bit register                                        : 1
 32-bit register                                       : 1
 4-bit register                                        : 2
 5-bit register                                        : 2
 7-bit register                                        : 1
 9-bit register                                        : 2
# Latches                                              : 7
 1-bit latch                                           : 7
# Comparators                                          : 16
 11-bit comparator greater                             : 5
 11-bit comparator lessequal                           : 5
 16-bit comparator greater                             : 2
 16-bit comparator lessequal                           : 2
 5-bit comparator greater                              : 2
# Multiplexers                                         : 29
 1-bit 4-to-1 multiplexer                              : 4
 16-bit 2-to-1 multiplexer                             : 1
 2-bit 2-to-1 multiplexer                              : 1
 4-bit 2-to-1 multiplexer                              : 2
 5-bit 2-to-1 multiplexer                              : 14
 7-bit 2-to-1 multiplexer                              : 1
 8-bit 2-to-1 multiplexer                              : 3
 8-bit 4-to-1 multiplexer                              : 1
 9-bit 2-to-1 multiplexer                              : 2
# FSMs                                                 : 2
```

Table 1: Resource Usage

7.2 Warnings

The table below shows the list of warning produced.

	Summary	New
WARNING	Xst:737 - Found 1-bit latch for signal <load_ctrl>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing	
WARNING	Xst:737 - Found 1-bit latch for signal <enable_x_count>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing	
WARNING	Xst:737 - Found 1-bit latch for signal <enable_y_count>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing	
WARNING	Xst:737 - Found 1-bit latch for signal <x_dir>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing	
WARNING	Xst:737 - Found 1-bit latch for signal <y_dir>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing	
WARNING	Xst:737 - Found 1-bit latch for signal <sync_data>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing	
WARNING	Xst:737 - Found 1-bit latch for signal <cnt_reset>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing	
INFO	Xst:1767 - HDL ADVISOR - Resource sharing has identified that some arithmetic operations in this design can share the same physical resources for reduced device utilization. For improved clock frequency you may try to disable resource sharing.	
INFO	Xst:3218 - HDL ADVISOR - The RAM <Mram_digit_2> will be implemented on LUTs either because you have described an asynchronous read or because of currently unsupported block RAM features. If you have described an asynchronous read, making it synchronous would allow you to take advantage of available block RAM resources, for optimized device usage and improved timings. Please refer to your documentation for coding	
INFO	Xst:3218 - HDL ADVISOR - The RAM <Mram_digit_0> will be implemented on LUTs either because you have described an asynchronous read or because of currently unsupported block RAM features. If you have described an asynchronous read, making it synchronous would allow you to take advantage of available block RAM resources, for optimized device usage and improved timings. Please refer to your documentation for coding	
INFO	Xst:3231 - The small RAM <Mram_q> will be implemented on LUTs in order to maximize performance and save block RAM resources. If you want to force its implementation on block, use option/constraint ram_style.	
INFO	Xst:3218 - HDL ADVISOR - The RAM <Mram_anode> will be implemented on LUTs either because you have described an asynchronous read or because of currently unsupported block RAM features. If you have described an asynchronous read, making it synchronous would allow you to take advantage of available block RAM resources, for optimized device usage and improved timings. Please refer to your documentation for coding	
INFO	Xst:3218 - HDL ADVISOR - The RAM <Mram_segLED> will be implemented on LUTs either because you have described an asynchronous read or because of currently unsupported block RAM features. If you have described an asynchronous read, making it synchronous would allow you to take advantage of available block RAM resources, for optimized device usage and improved timings. Please refer to your documentation for coding	
INFO	Xst:1901 - Instance U1/pll_base_inst in unit U1/pll_base_inst of type PLL_BASE has been replaced by PLL_ADV	
INFO	Xst:2169 - HDL ADVISOR - Some clock signals were not automatically buffered by XST with BUFG/BUFR resources. Please use the buffer_type constraint in order to insert these buffers to the clock signals to help prevent skew problems.	

Table 2: Table of Warnings

I noticed that latches were produced in the final design, these latches were the result of a not specifying the status for all the control signals in each state of in my state machines for the box tracker module, and the DAC interface module. This was caused by an oversight on my part, and due to a time constraint, I was not able to rectify these warnings.

8 Extra Content:

While trying develop the sine wave generator, I ended up making a triangle wave by accident. The source code for my triangle wave generator is shown below.

```
-- Constants that hold the DAC codes
      constant D_OFF : std_logic_vector(7 downto 0) := "00000000"; -- 0
      constant D0    : std_logic_vector(7 downto 0) := "00100000"; -- 32
      constant D1       : std_logic_vector(7 downto 0) := "01000000"; -- 64
      constant D2       : std_logic_vector(7 downto 0) := "01100000"; -- 96
      constant D3       : std_logic_vector(7 downto 0) := "10000000"; -- 128
      constant D4       : std_logic_vector(7 downto 0) := "10100000"; -- 160
      constant D5       : std_logic_vector(7 downto 0) := "11000000"; -- 192
      constant D6       : std_logic_vector(7 downto 0) := "11100000"; -- 224
      constant D7       : std_logic_vector(7 downto 0) := "11111111"; -- 255
```

Code Sample 13: Triangle Wave Generator

Instead of using 16 values for the multiplexer, I calculated 8 values which were incorrect for producing a sine wave, after testing the module, it was shown that the values produced a triangle wave, as shown in the figure below.

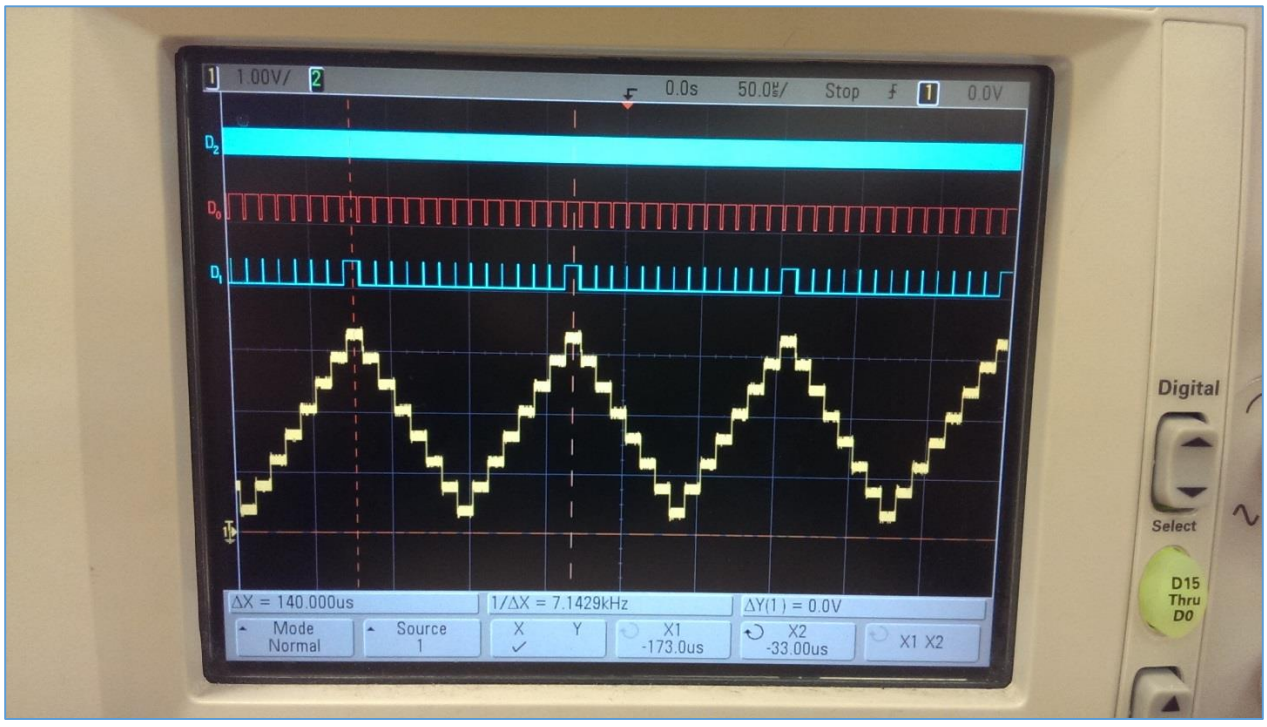

Figure 23: Triangle Wave generator

APPENDIX