Lab 3: Moving Block, Sine Wave Generation, and Simulation

Yil Verdeja, ECE Box 349
Yeggi Lee, ECE Box 191
Lab Section A01
September 28, 2018

TABLE OF CONTENTS

1 Introduction **2**

2 Design Description **2**

2.1 Moving Block 4

2.1.1 Creating the Block 4

2.1.2 Positioning 5

2.1.3 Debouncing 6

2.1.4 Extra Credit 7

2.2 DAC - Waveform Generation 7

2.2.1 SPI Creation 7

2.2.2 Sine Wave Logic 8

2.2.3 Output 9

2.2.4 Extra Credit 11

2.3 Simulation and Testing 13

3 FPGA Resource Usage and Warnings **14**

3.1 Flip Flops (FF) 14

3.2 Synthesis Warnings 16

4 Conclusion **16**

5 Appendix **18**

1 Introduction

The purpose of this lab was to further understand the VGA controller in order to generate a moving green block and to learn how to use a DAC module to generate a sine wave. The VGA monitor was programmed to display a green block that can be controlled via 4 push-buttons. The position of the block would then be displayed in the seven segment display. For the second part of the lab, a DAC SPI interface with a shift register and state machine was implemented which updated the DAC every 100 KHz with new 16-bit data values. Through this, a sine wave with a frequency of 6.25KHz was created using 16 constant values per cycle.

From this initial implementation, by switching on certain switch configurations, different modes were implemented. The first mode directly affected the VGA display by changing the color depending on the position of the block, and allowing the block to travel infinitely between the screen. The second and third mode outputted a triangle and sawtooth waveform respectively to the DAC.

2 Design Description

The figure below shows the block diagram of the *controller* module. Using the MMCM, the controller reduces the internal FPGA clock to 25 MHz and 10 MHz. Additionally, it instantiates lower-level modules that will produce the output described in the *Introduction*.

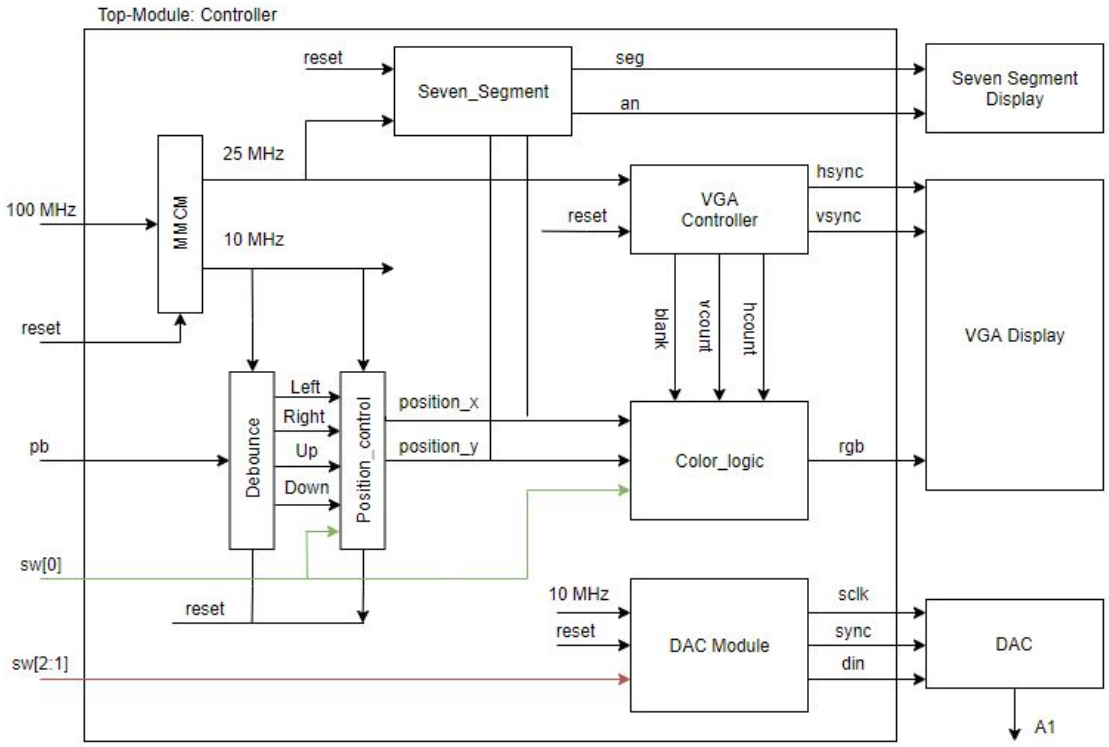

Figure 1. Top-Module Block Diagram

Debounce, Position Control, VGA and Color Logic, and Seven Segment Display

To create a 32 by 32 pixel moving block on the VGA display, the four push buttons to control the movement of the block have to be debounced. If it weren't, the block would instantly travel to the end of the screen from a single press of a button. Using a state machine, the *debounce* module outputs a clean button press.When the button is pressed, the output button goes high for one 10 MHz clock cycle every 100 milliseconds. By doing this, a quick button press is not registered as multiple button presses.

The *position_control* module is in charge of updating the X and Y position of the block as the push buttons (left, right, up and down) on the FPGA board are pressed. It is also in charge of limiting the block to the minimum and maximum boundaries of the VGA display. However, by activating a switch the position of the block becomes capable of wrapping around the screen. This means that if the block were to go past the maximum limit, it would appear at the minimum limit, and vice versa. This module was able to implement this logic through the use of a binary encoded state machine.

The *seven_segment* module uses all four seven-segment displays on the Basys 3 Board. Since only one seven segment display can be lit at a time, the four anodes that control each display are toggled at a frequency greater than 60 Hz. By increasing the frequency of each anode to 100 Hz, the switching between each anode is undetectable to most humans as it produces a steady light to an individual.[1] Since each anode uses a 100 Hz, the *seven_segment* module slows down the 25 MHz high speed clock to a 400 Hz input clock. The seven-segment displays the current X and Y position of the block in the format: XXYY. The X position ranges between 0 to 19, while the Y position ranges between 0 to 14.

The VGA logic is created through a *color_logic* module and a provided *VGA Controller* module. The *VGA controller* contains the logic to generate the synchronization signals, horizontal and vertical pixel counters and video disable signal for the 640x480@60Hz resolution. On the other hand, the *color_logic* module uses the horizontal and vertical pixel counters, as well as the current X and Y positions of the moving block. Knowing the position of the block, if the pixel count is between the general range shown below, then it should be colored green:

$$position*32 < pixel\ count < position*32 + 32$$

Note: hcount will use the current X position, while vcount will use the current Y position in determining the range.

[1] https://skeptics.stackexchange.com/questions/3348/can-the-human-eye-distinguish-frame-rates-above-60-hz

Using the same switch that provides the wrapping functionality of the block on the VGA display, the switch is also used to change the color of the block depending on its current position.

DAC SPI Interface

The *DAC* module acts as an SPI interface and is in charge of creating both the 10 MHz serial clock and the 100 kHz synchronous clock for the digital to analog converter. Through the use of a shift register, this module also outputs 16 single-bits of data when the synchronous clock is low for 16 cycles. By providing the digital information to create a sine, triangle or sawtooth waveform, the output of the DAC is somewhat clean waveform depending on the switch configuration provided.

2.1 Moving Block

For this lab, it was assumed that the 640x480 VGA monitor was divided into blocks of 32 pixels high x 32 pixels wide. This would create a 20x15 block grid onto the display as shown in the figure below. Therefore, the top left corner is in the x, y position (0, 0) while the bottom right corner is (19, 14).

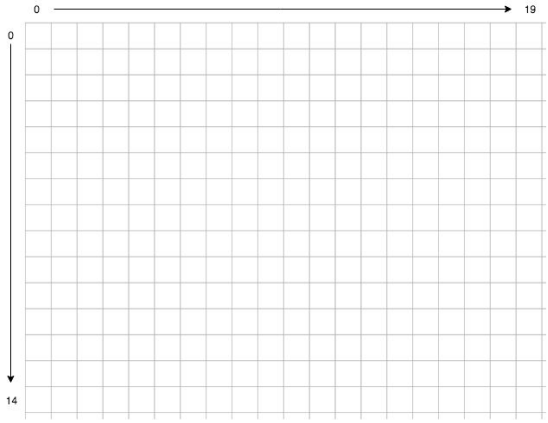

Figure 2. 20x15 Block Grid on VGA Monitor

2.1.1 Creating the Block

The main purpose of the *color_logic* module was to create a 32x32 pixel green block. In order to do this, boundaries for the block were established using *x_lines* and *y_lines*. In the example below, the boundaries of the horizontal lines would be 64 and 96. On the other hand the boundaries of the vertical lines would be 32 and 64 pixels. All the pixels within those boundaries would then become green.

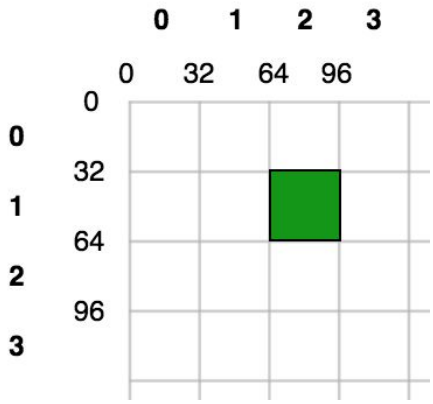

Figure 3. Color_logic Example

The x_lines and y_lines create the boundaries of the green block. Therefore, regardless of position, the block will always be 32 pixels wide x high.

2.1.2 Positioning

Next, in order to make the block move using the buttons, the *position_control* module was created. First, the module checks to make sure the reset button was not pressed. If pressed, then the block returns to its original position, (0,0). Otherwise, the block will remain in its current position.

Table 1. Position Rules of the Block

Button Pressed	Requirements	Next Position
UP	(current_y > MIN_Y) && up	position_y = current_y - 1'b1
DOWN	(current_y < MAX_Y) && down	position_y = current_y + 1'b1
RIGHT	(current_x < MAX_X) && right	position_x = current_x + 1'b1
LEFT	(current_x > MIN_X) && left	position_x = current_x - 1'b1

For the block to move anywhere, the attempted move must be within the boundaries of the VGA monitor. The table above shows the requirements that must be fulfilled before the block moves. For example, the block may only move up when its position is greater than the minimum value, 0, and when the UP button is pressed. To move right, the current horizontal position must be less than 19 and so on. Assuming that the green position was in position (4, 3) and the DOWN button was pressed, the current position would change into (4, 4). Once all the calculations are done, the current position is then displayed in the seven seg display.

2.1.3 Debouncing

When a button is pressed, two metal parts connect with each other. Though it may seem that the contact is made instantly, the components of the button mechanically bounce as they settle into their new position. This causes the circuit to be opened and closed several times which is known as bouncing. As the hardware usually works faster than the bouncing, the hardware thinks the button is pressed multiple times resulting in the waveform below. [2]

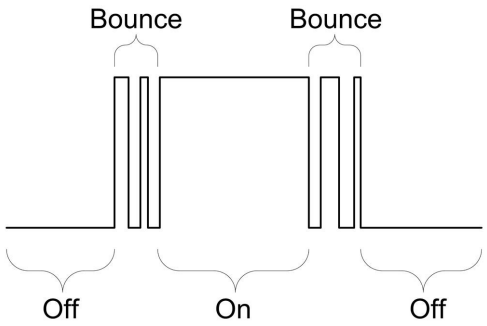

Figure 4. Bouncing Waveform when Button is Pressed[3]

Therefore to prevent this issue, debouncing is done to ensure that only one signal will pass for a single opening or closing of a button. In this case, through a state machine, the button will be deactivated for a specified length of time after the first contact is made.

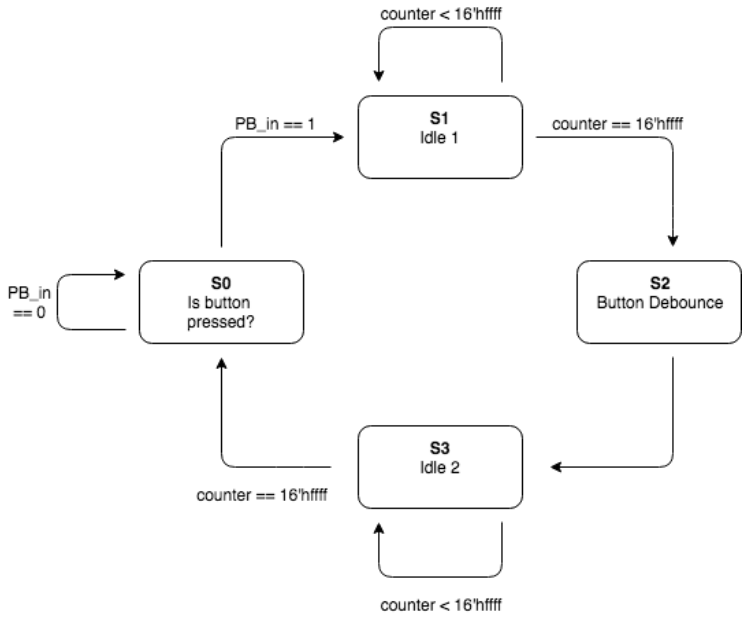

Figure 5. Debouncer State Machine

[2] https://www.pololu.com/docs/0J16/4
[3] http://vhdlguru.blogspot.com/2017/09/pushbutton-debounce-circuit-in-vhdl.html

The states for the *debouncer* state machine are:

- State 0 (S0) - Button Press
    - The initial state of the debouncer checks for a button press. If the button is pressed, the debouncer transitions into S1. Otherwise, it stays in S0.
- State 1 (S1) - Idle 1
    - S1 waits for a 15 Hz cycle and then moves on to the next state. Otherwise, the debouncer remains in this state.
- State 2 (S2) - Button Debounce
    - S2 waits for 10 MHz clock cycle and then moves on to the next state.
- State 3 (S3) - Idle 2
    - Similarly to S1, S3 also waits for 15 Hz cycle and then moves on to S0. Otherwise, the debouncer remains in this state.

2.1.4 Extra Credit

An additional feature was added in this part of the lab. If the switch is enabled, then the block will be able to move past the boundaries of the VGA monitor. Additionally, the block change colour depending on its position in the monitor. Depending on the x and y position, the block will change red and blue colours respectively. Otherwise, if the switch is turned off, then the block returns to green and is unable to move past the boundaries.

2.2 DAC - Waveform Generation

2.2.1 SPI Creation

When creating the SPI for the DAC, everything needs to be used on the negative edge of the serial clock since the DAC does its conversions using the positive edge of the clock. To avoid conflict, it's recommended to use the negative clock edge as a trigger for the sequential logic.

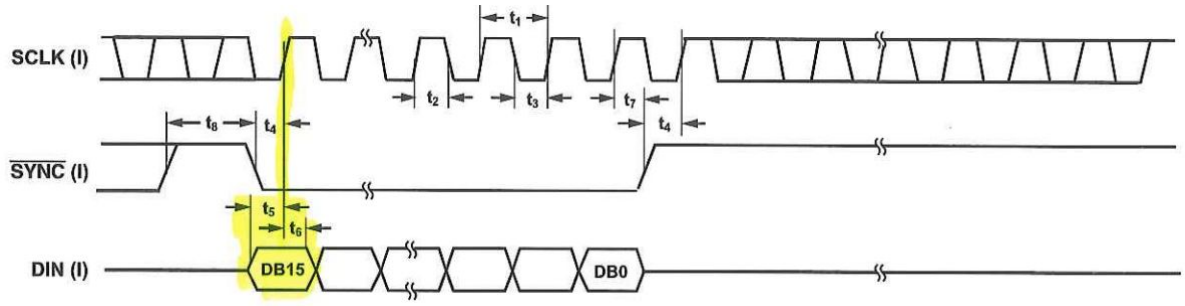

Figure 6. Timing Diagram for Continuous 16-Bit

This DAC module acts as an SPI Interface that creates the sync (synchronous clock) for the DAC (16 SCLK cycles at active low) at 100 kHz and outputs a single bit of data to the DAC. In order to do this, a 100 kHz clock that is at active low for 16 SCLK cycles was created as a sync

counter. Then, a state machine was implemented that changes states depending on the value of the sync counter as shown in the block diagram below. The output of the data is the most important bit of the data being read.

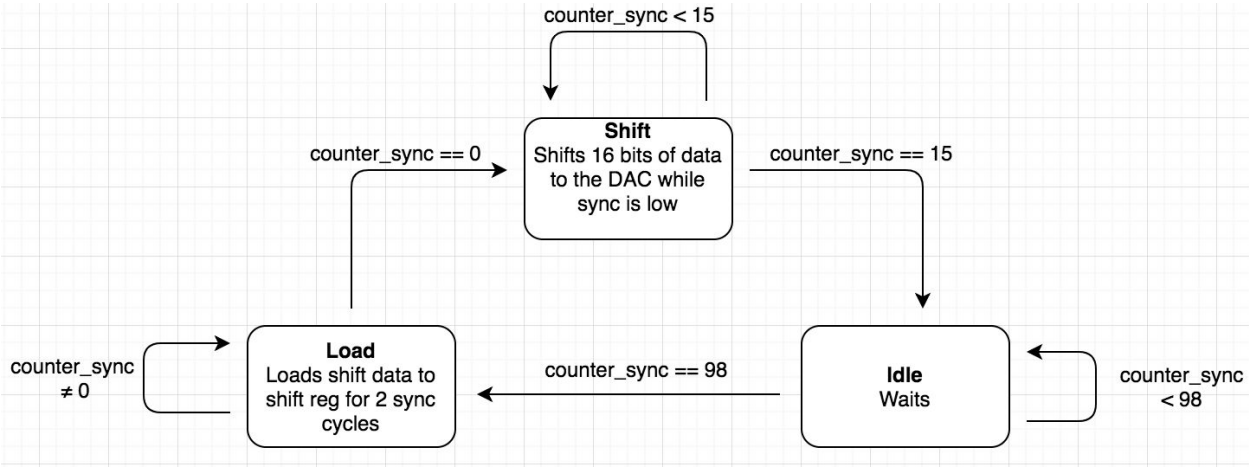

Figure 7. DAC State Machine

The states are as follows:

- Load
    - The initial state of the DAC checks the value of the sync counter. If the sync counter is 0, then the DAC transitions into the shift state. Otherwise, it stays in the load state.
    - In this state, data is loaded into the shift register for two sync cycles.
- Shift
    - The shift state shifts the 16-bit data received to the left while sync is low
    - If the sync counter reaches 15, it moves to the next state; otherwise, it remains in the shift state
- Idle
    - The idle state remains in this state until the sync counter reaches the value of 98. Then it moves back to the load state.

2.2.2 Sine Wave Logic

In order to generate the sine wave waveform with a frequency of 6.25 kHz, sixteen constant values per cycle were calculated using the formula below.

$$v(t) = 1.65 \cdot sin(2\pi ft) + 1.65$$

For the following equation f is the frequency, t is time which is denoted by the current step and the specified clock cycle. Since the frequency is 6250 kHz, then the period of the sine wave

should be 160 µs. Since each sine wave should be composed of 16 voltage steps, then each step should take a sixteenth of the period which is 10 µs.

As shown in the table below, to determine the digital value, the resolution was found to be 77.27 bits/voltage. This was determined knowing that the data output had 255 bits and the DAC ranged from a value of 0V to 3.3V.

Table 2. Constant Values for Sine Wave

Step	Step Voltage (V)	Digital Value	Round Up	Binary
0.00	1.65	127.50	128.00	10000000
1.00	2.28	176.29	177.00	10110001
2.00	2.82	217.66	218.00	11011010
3.00	3.17	245.29	246.00	10000000
4.00	3.30	255.00	255.00	11111111
5.00	3.17	245.29	246.00	11110110
6.00	2.82	217.66	218.00	11011010
7.00	2.28	176.29	177.00	10110001
8.00	1.65	127.50	128.00	10000000
9.00	1.02	78.71	79.00	1001111
10.00	0.48	37.34	38.00	100110
11.00	0.13	9.71	10.00	1010
12.00	0.00	0.00	0.00	0
13.00	0.13	9.71	10.00	1010
14.00	0.48	37.34	38.00	100110
15.00	1.02	78.71	79.00	1001111

2.2.3 Output

To verify the functionality of the DAC SPI, the board was connected to an oscilloscope. On the top two oscilloscope images below, CH2 is sclk while CH1 is for sync. Clearly, from the information provided by the oscilloscope, CH2 has a 10 MHz clock, while CH1 has a 100 kHz clock. Using the cursor tool, the sync is low for 1.6 µs, which is equivalent to 16 sclk cycles.

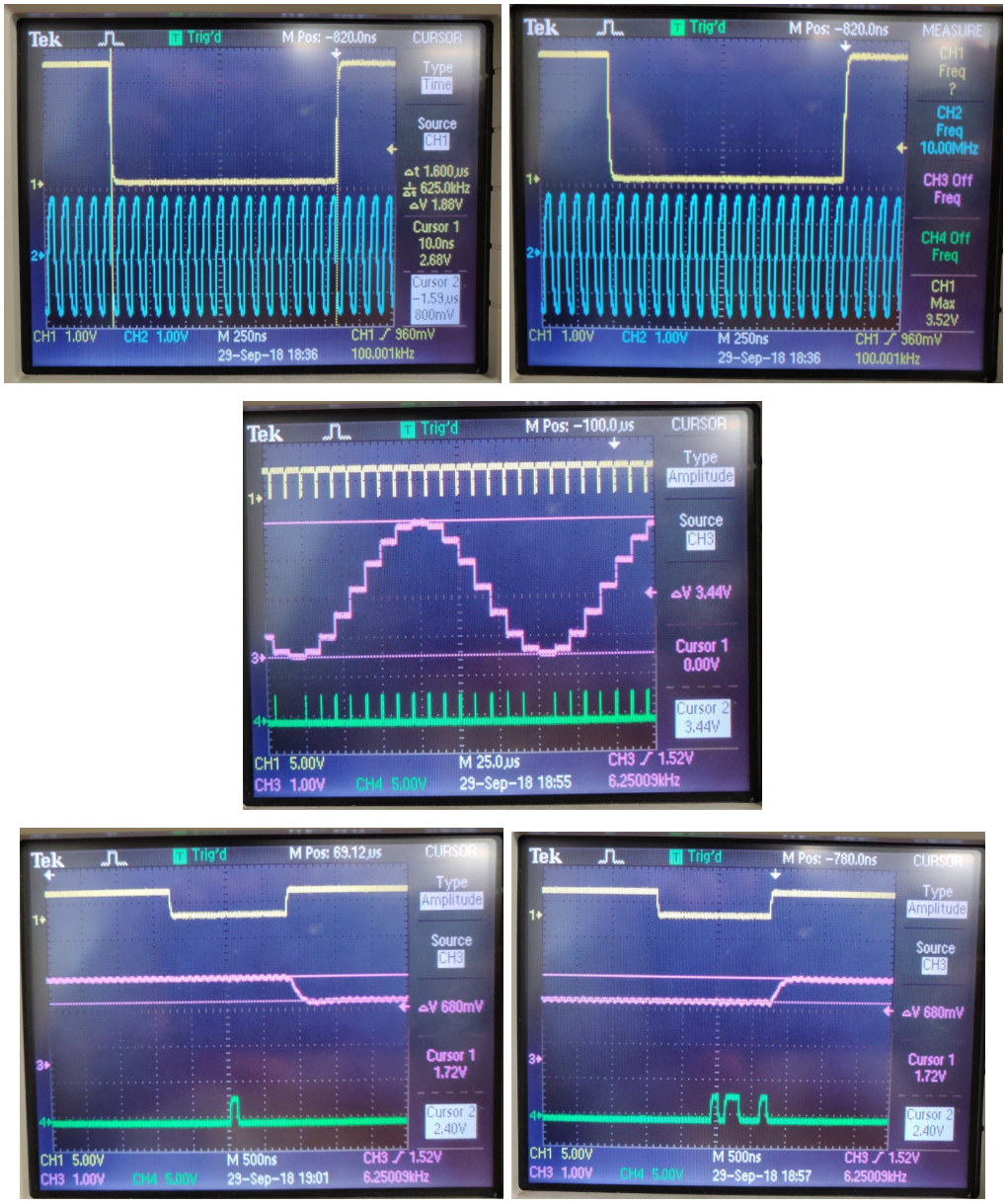

Figure 8. Oscilloscope Verifications of DAC

The image in the middle shows sync as CH1, the sine wave data on din at CH4, and the sine wave output on CH3. As expected, there are 16 steps to the sinusoidal wave and it operates at a frequency of 6.25 kHz as specified. The bottom two images verify the functionality of the oscilloscope by looking at the digital input shifted into DAC. On the right, as soon as a binary value of 8'b10000000 is synced, the output is dropped to 1.72 V. On the left, as soon as a binary value of 8'b10110001 is synced, the output increases to 2.40 V. Due to the assumption that the DAC would provide a maximum of 3.3V, the small discrepancy of the step voltage was expected.

2.2.4 Extra Credit

Using switches, the type of wave generated can be switched from a sine wave to either a triangle wave or a sawtooth wave. Considering that both these waves have a linear rising or falling edge, the potential difference between each step should be equal. The triangle wave is similar to the sinusoidal wave there are 8 unique steps from 0V to 3.3V. On the other hand, for the sawtooth wave, once it reaches 3.3V it drops to 0V. Therefore each voltage step on this waveform is about 0.206V.

The tables below show binary values determined for the triangle wave and sawtooth wave.

Table 3. Constant values for Triangle wave

Step	Voltage reading (V)	Digital Val	Binary
0	0.41	32	100000
1	0.83	64	1000000
2	1.24	96	1100000
3	1.66	128	10000000
4	2.06	159	10011111
5	2.47	191	10111111
6	2.89	223	11011111
7	3.30	255	11111111
8	2.89	223	11011111
9	2.47	191	10111111
10	2.06	159	10011111
11	1.66	128	10000000
12	1.24	96	1100000
13	0.83	64	1000000
14	0.41	32	100000
15	0.00	0	0

Table 4. Constant values for a sawtooth wave

Step	Voltage reading (V)	Digital Val	Binary
0	0.00	0	0
1	0.22	17	10001
2	0.44	34	100010
3	0.66	51	110011

4	0.88	68	1000100
5	1.10	85	1010101
6	1.32	102	1100110
7	1.54	119	1110111
8	1.76	136	10001000
9	1.98	153	10011001
10	2.20	170	10101010
11	2.42	187	10111011
12	2.64	204	11001100
13	2.86	221	11011101
14	3.08	238	11101110
15	3.30	255	11111111

The images below show the triangle wave and sawtooth wave output in different switch configurations.

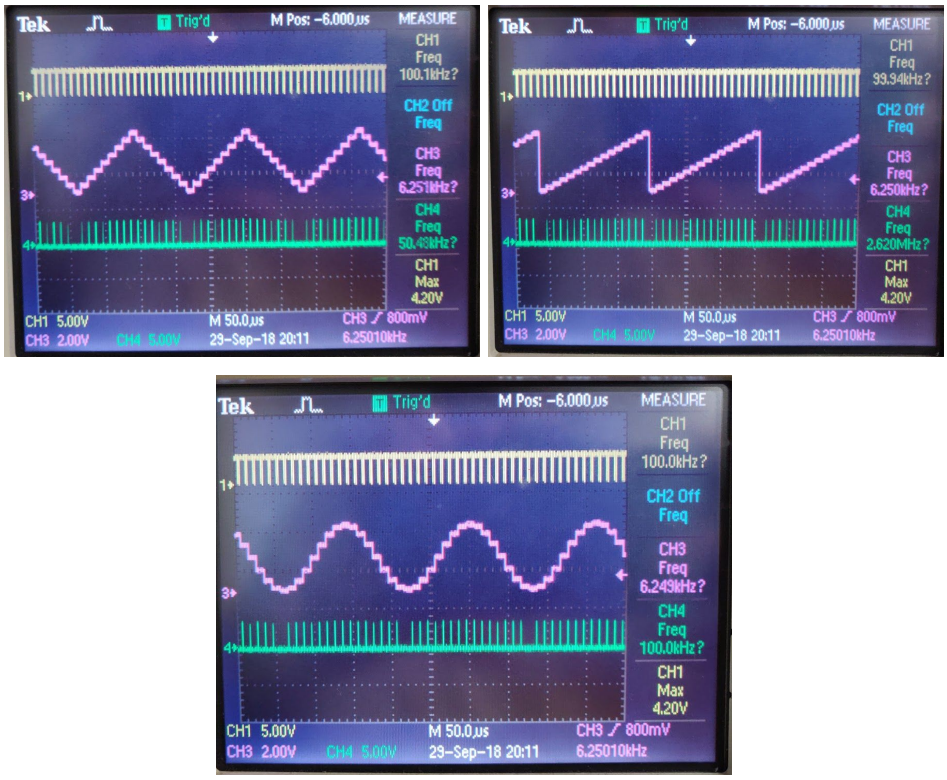

Figure 9. Triangle, Sawtooth and Sinusoidal waveforms at 6.25kHz using 16 steps

2.3 Simulation and Testing

A test bench was created to show how the sine wave data was transferred to the DAC. Creating the test bench was simple as the only input that had to be changed was the internal FPGA clock of 100 MHz with a 50% duty cycle. The only stimulus to the test bench was making reset high for 100 ns. Doing this would make sure that every part of the circuit was reset correctly.

Although the scope is very large and it's hard to see the detailed values, the sole purpose of the figure below is to show the 16 steps generated by the sine wave generator inside the *DAC* module.

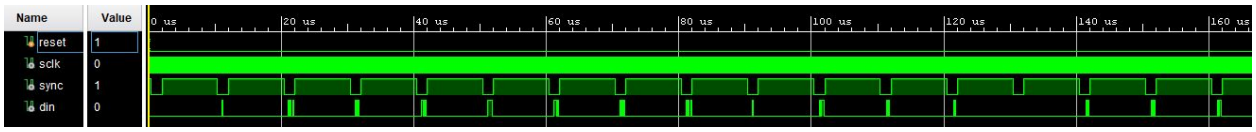

Figure 10. 16 Steps Generated by DAC Sine Wave

Nonetheless, the two figures below provide a better understanding of a single SPI cycle. As shown, the sclk has 5 cycles for every 500 ns. This proves that every cycle is as long as 100 ns which is equivalent to a 10 MHz clock as expected. Second, it's very clear that the synchronous clock goes low for 16 sclk cycles. From the picture above, the synchronous clock has a frequency of 100 kHz since it has a period of 10 us.

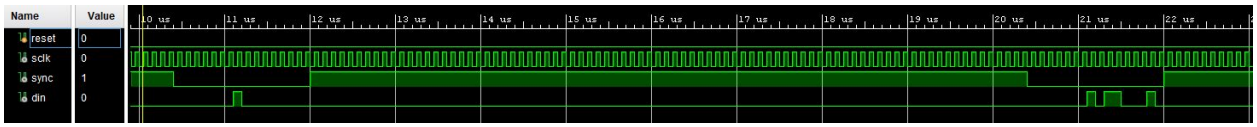

Figure 11. SPI Cycle

Figure 12. SPI Cycle Zoomed In

Lastly, to provide a better understanding of the sine wave data output, the figure above shows the second step of the sine wave which should have a binary value of 10110001. An interesting thing to note is that even though the DAC and the top-level controller are outputting the correct readings to sclk, sync and din on the oscilloscope, it seems to be shifted by 1 more than it's supposed to when simulating it on the test bench.

3 FPGA Resource Usage and Warnings

This section gives an explanation to the FPGA resource usage and the warning messages received.

3.1 Flip Flops (FF)

As shown in the figures below, the number of flip-flops used after implementation were 179. By looking at all the modules, this value can be determined.

Since *color_logic*, *sinewave_gen*, *trianglewave_gen*, and *sawtoothwave_gen* only use combinational logic, they do not generate any flip flops.

Number of Flip Flops in the ***debounce*** module:

Variable	FF
counter	20
current_state	2
?	2
Total	**24**

Note: Looking at the *debounce* module, it is unclear how there are two extra flip flops. Also, since the debounce module is instantiated four different times for four different push buttons, the total amount of flip flops amounts to 24*4 = **96 flip flops**.

Number of Flip Flops in the ***DAC*** module:

Variable	FF
current_state	2
count_16	4
data_reading	16
counter_sync	7
sync	1
Total	**30**

Number of Flip Flops in the ***position_control*** module:

Variable	FF
current_x	5
current_y	4
Total	**9**

Number of Flip Flops in the ***seven_segment*** module:

Variable	FF

count	3
clk_sevenseg instantiation	16
Total	**19**

Number of Flip Flops in the ***clk_sevenseg*** module:

Variable	FF
counter	16
Total	**16**

Number of Flip Flops in the ***vga_controller*** module:

Variable	FF
hcounter	11
vcounter	11
HS	1
VS	1
blank	1
Total	**25**

Our original analysis would have had a total flip flop value of 171, but due to the fact that the *debounce* module had a two extra flip flops, the total number of flip flops increased by 8.

Resource	Utilization	Available	Utilization %
LUT	230	20800	1.11
FF	179	41600	0.43
IO	37	106	34.91
BUFG	3	32	9.38
MMCM	1	5	20.00

Figure 13. Post-Implementation FPGA Resource Usage

Name	Slice LUTs (20800)	Slice Registers (41600)	Bonded IOB (106)	BUFGCTRL (32)	MMCME2_ADV (5)
controller	230	179	37	3	1
cl (color_logic)	4	0	0	0	0
d1 (debounce)	21	24	0	0	0
d2 (debounce_0)	21	24	0	0	0
d3 (debounce_1)	21	24	0	0	0
d4 (debounce_2)	21	24	0	0	0
dac (DAC)	38	30	0	0	0
mmcm (clk_wiz_0)	0	0	0	3	1
inst (clk_wiz_0_clk_...	0	0	0	3	1
pos (position_control)	38	9	0	0	0
sevenseg (seven_seg...	24	19	0	0	0
clkss (clk_sevenseg)	13	16	0	0	0
vgac (vga_controller_6...	46	25	0	0	0

Figure 14. Flip Flop usage per module after synthesis

3.2 Synthesis Warnings

The program did not generate any problematic synthesis warning. According to a xilinx employee[4] this type of warning can be safely ignored.

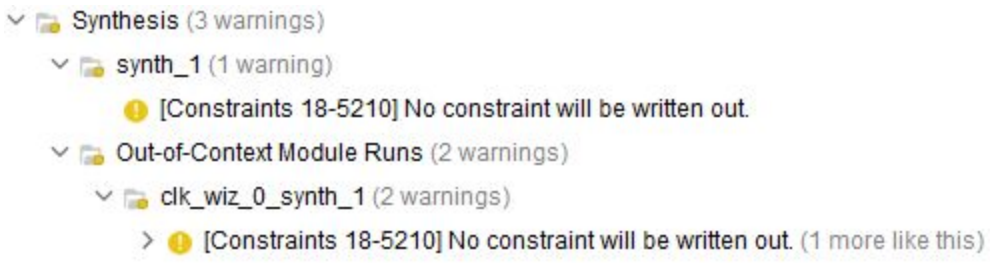

Figure 15. Synthesis Warnings

4 Conclusion

An issue we had during the implementation phase was correctly switching between states. Initially, our debounce state machine could not reach S2 and S3. After visualizing the solution on paper and running by our code, we realized that both our *current_state* and *next_state* were 1-bit wide instead of 2-bits wide. This was essentially keeping the debouncer in S0 and S1.

[4] https://forums.xilinx.com/t5/Synthesis/Meaning-of-synthesis-warning-Constraints-18-5210-No-constraint/m-p/881019?device-view=desktop

A source of error that hindered our progress, was a trimming waveform. Initially, the DAC was verified using switches, but since it was shifted to the left by one extra bit (thus doubling it), the output would have always gone from 0V to 3.3V. After taking a closer look at the oscilloscope data, and individually checking each voltage step, we realized the error and were able to solve it with a quick change in the shift-register logic.

Through this lab, we were able to see how to create a state machine for different implementations such as the shift register and the debouncer. Furthermore, we also figured out how to make a moving block through the use of state machines and learned why debouncing is important. We also learned how to properly use the DAC and create different waveforms using it.

Possible improvements in the code would be to reduce the number of states in the *debouncer* from four to three. The first state (S1) and third state (S3) are very similar and could be condensed into one state. However, the logic for the state would be much more complex. Perhaps in the future when we become more familiar with state machines, this could be done.

An extension could be to create a game of Snake on the VGA display. However, this would involve creating randomly generated blocks and a slowly increasing a snake body which is not something we have learned yet . Similar to the *position_control* module, the boundaries would be set. We would like to further explore how to create a game.

5 Appendix

Block produced through the VGA monitor and Basys3 board