# Lab 3: DAC Waveform Generation Digital System Designs with FPGAs

ECE 3829

Nicholas Lanotte & Matthew DiPlacido

October 2nd, 2017

# Table Of Contents

## Introduction

Digital circuitry is an integral part of almost every electrical application today, however, analog signals still need to be utilized for certain tasks in digital systems. One way of bridging digital and analog signals is through the use of a digital to analog converter that accepts digital signals and a clock signal as an input and outputs an analog signal corresponding to the digital input signal. Another aspect of utilizing analog and digital signals together, is using analog circuitry as an input for digital circuits. A simple push button is a common user input that allows for direct manipulation by the user. However, this component cannot be used in a high speed digital application without special implementation due to “bouncing” of the voltage as the mechanical elements of the switch come into and out of contact. This lab experiment will demonstrate the implementation a Digital to Analog Converter (DAC) using a Field Programmable Gate Array (FPGA), the implementation of push buttons in a high speed digital circuit and extend the VGA display connectivity explored in Lab 2.

## Problem

The first part of this experiment was to implement four pushbuttons as directional keys to move a block on a VGA 640x480 Display. A common problem with push buttons in a high speed digital circuit is that when pressed, the lever that closes the circuit bounces off of the metal connection at the output of the button. This will close and open the circuit multiple times before settling to either a high or low symbol. This bouncing may only last for nanoseconds at a time but in high speed applications, each bounce is detected as a button press, which can result in undesired behavior of the logic circuit. In order to complete part one of this lab a button debouncing circuit needs to be designed and implemented with the VGA controller module from Lab 2 and a VGA color controller module to display a block in a 10x10 array that can be moved by the buttons.

The second part of this lab was to construct a circuit using the PmodDAC and a state machine to produce a 6.25KHz sinewave output. The DAC should update the value on the output at a rate of 100KHz meaning 16 steps will make the output waveform. Communication between the DAC and the FPGA is dictated by a 10Mz clock signal, a sync signal and data bits that are accepted by the DAC when the sync signal is low. The sync signal must go low on the negative edge of the input clock and then on the subsequent 16 positive clock edges the DAC will read in the 16 bits of data and then after the 16th positive clock edge, on the following negative clock edge, the sync pin goes high indicating an end of data transmission.

The final part of this implementation on the FPGA was to create a simulation source to test and examine the transfer of data between the FPGA and the DAC. Additionally, bonus features could be added to show an in depth understanding of this lab.

## Overview of Design Approach

In order to implement the system described above, many verilog modules need to be created to simplify the top module, make debugging simpler, and allow each module to be used in future applications.

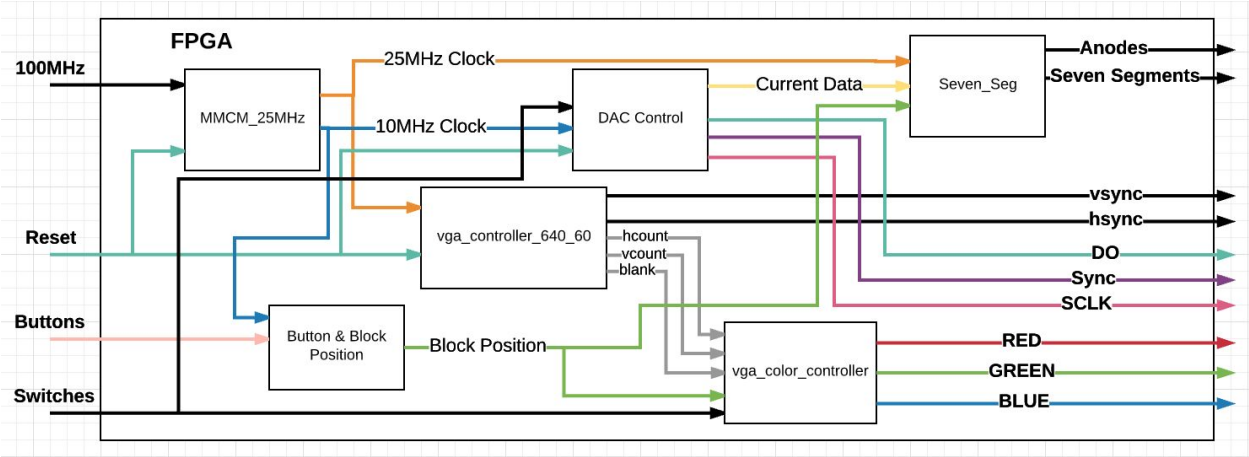



**Figure 1: Design Approach Block Diagram**

Figure 1 shows the 6 main modules instantiated by the top module. The seven segment display module and the vga_controller_640_60 module were reused from lab 2. The first new module was the Button & Block Position module (titled Button_Position in the code), to drive the 4 directional push buttons and determine the position of the block to be displayed on the screen. This module needs to debounce the buttons so that only one button press is detected when the button is pressed and store the position of the block in flip flops. To debounce the buttons, the module is run at 100Hz so that the buttons are not polled frequently enough to detect the bouncing as it only lasts for nano seconds. The x and y positions were stored in 4 bit registers, which is outputted to the seven segment display and the vga_color_controller. The buttons are arranged on the Basys3 board similar to directional keys on a keyboard or a direction pad on a video game console so each button moved the block one position per button pressed in their corresponding position. Each button had a flag that is set high on the negative edge of the clock after the button is pressed and set low on the negative edge of the clock when the button is released. This prevents the block from being moved more than one position when a button is held down for extended periods of time. Lastly, the block cannot be moved outside of the 10 by 10 array on screen so there is some additional logic that checks to see if the block would be moved into an area on screen. If the block would be moved off screen the position data will not be changed otherwise the position data will be modified and fed into the seven segment display and vga_color_controller.

The vga_color_controller module drives the display through the VGA port of the FPGA. This needed to display a 64 pixel long, by 48 pixel tall, yellow block, initially in the top left corner of the screen. This corresponds to position (0,0) from the block position module. This module

uses the vertical and horizontal pixel count signals from the vga_controller_640_60 along with the position data to determine when to print a yellow or black pixel on the screen. The x and y positions are brought through some mathematical operations that determine the upper and lower bounds of the pixels that the yellow block is to be printed in. When the horizontal and vertical counts are within the window determined by the block position, the color module outputs a yellow color signal to the VGA display, otherwise it outputs a black color signal. This method allows for the module description to be short and simple. This logic is only executed when the “blank” signal from the vga_controller_640_60 is low, indicating that the current pixel is on screen. The color signals are made up of three 4-bit busses, one for each subpixel (red, blue or green), that determine the intensity of each subpixel. For a black color, each bus outputs a low signal, turning off each of the subpixels and for yellow, the green and red subpixels are turned on to full intensity, while blue is kept off. This finished the requirements of part 1 of this lab assignment.

To complete the second part of the lab, the module Dac_Control was created that holds the state machine and spi interface that communicates with the Pmod Dac to produce the desired sine wave at 6.25KHz. The sine wave was created using 16 constant values so there was one state in the state machine that corresponds to each of these values. Each state immediately leads to the next state on the next clock cycle, as shown in the sub diagram of Figure 2.

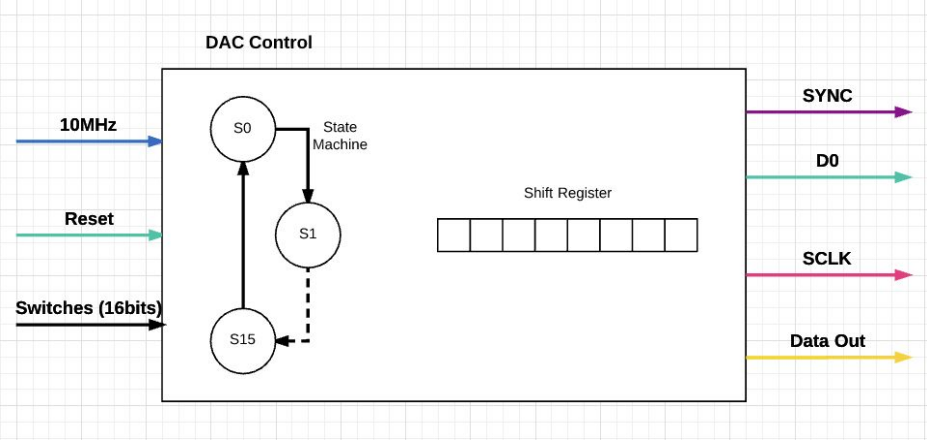



**Figure 2: DAC Control State Machine Diagram**

A 10MHz clock is inputted to the DAC_Control module that runs the SPI communication. This is too fast to run the state machine so this clock was brought down to 100KHz. The state machine determines the data to be sent to the DAC. When the state-machine enters a new state, its corresponding DAC value is loaded into the SPI shift register. The chip select for the DAC is then brought low and the shift register is shifted at 10MHz to transfer the data to the DAC. Upon completion the chip select is brought High so that the DAC outputs the new voltage. A 16-bit value is transferred to the DAC, with the 8 least significant bits containing the voltage information. The 8 most significant bits contain instructions for the DAC, which in this case corresponds to all 0s.

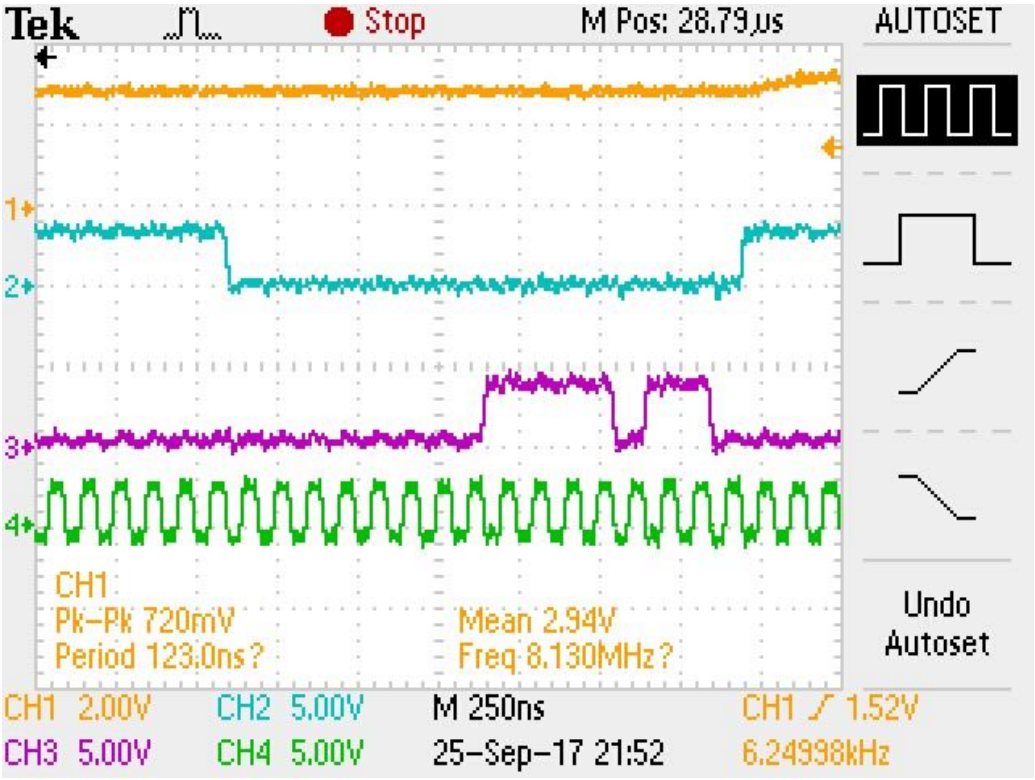



**Figure 3: SPI Communication Waveforms**

In Figure 3, the SPI communication waveforms are measured by the oscilloscope. The Yellow waveform is the output voltage from the DAC, the Blue waveform is the chip select, the purple waveform is the data transferred to the DAC and the green line is the 10MHz clock signal. The DAC reads the data on the positive edge so the DAC_Control module loads the data on the previous negative edge of the clock. The Waveform shows that the DAC reads eight 0s before reading 1111_0110. The DAC produces a voltage of about 3.3V when transferred 1111_1111b (255d) and 0V when the transferred 0000_0000 (0d). In Figure 3 a decimal value of 246 is sent and the DAC produces an output of about 3V, so the SPI module and DAC seem to be in sync and operating correctly.

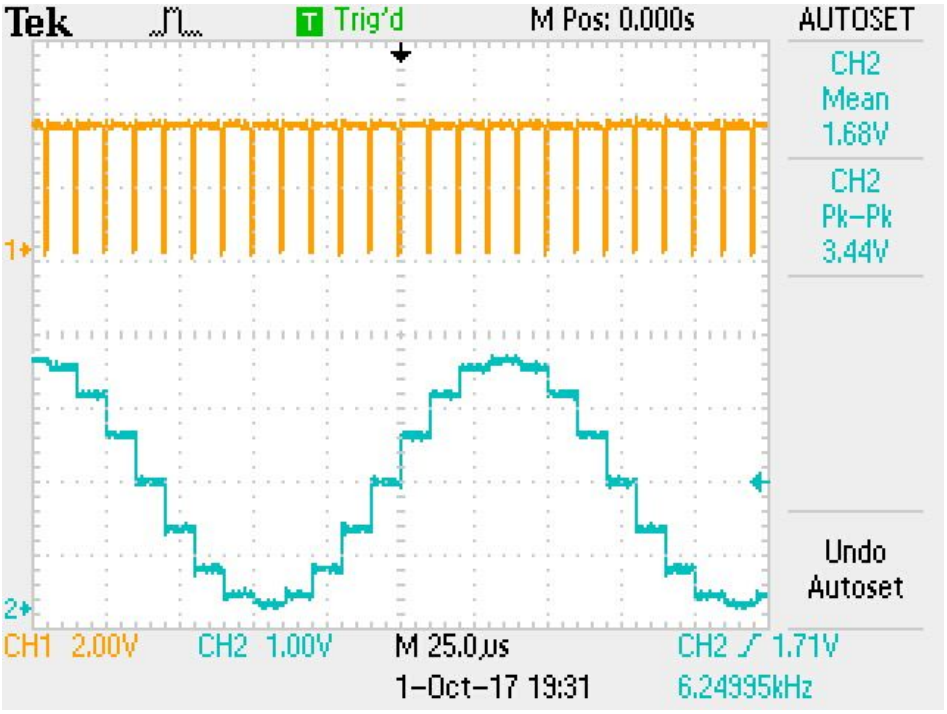



**Figure 4: Produced Sine Wave**

Figure 4 shows the sine wave produced by the sine wave. The frequency is about 6.25KHz as desired frequency. The amplitude of the sine wave is a little less than 4V, which is greater than expected but it appears that a voltage greater than 3.3V is being outputted by the FPGA.

The last new module for this lab was the MMCM which has a 10MHz and 25MHz output clock signal. The 10MHz clock is used to drive the SPI interface and is the clock that is brought down to 100KHz for the state machine in the DAC_control module. The 25MHz clock is used in the VGA_Controller_640_60 module and in the button_position module to drive the VGA display.

Two additional features were added to this design to display mastery of the design concepts studied in class and explored by this lab. In addition to the sine wave output, a square wave, triangle wave and sawtooth wave can also be generated at 6.25KHz, making this design a function generator. The waveform is selected by the slider switches on the FPGA board. These output waveforms are shown in Figures 5 through 7. Additionally, 16 constant voltages can be outputted using other slider switches and the value being sent to the DAC is displayed on the seven segment display next to the x and y position of the block diagram. This is implemented by containing the state-machine described above inside a case statement. The case statement determines which case to switch to based on the slider switches. Each case then contains a 16 state state-machine corresponding to an output waveform. Lastly, the color of the block and background on the VGA display can be modified using the 6 leftmost switches on the FPGA board. This allows the block and background to be set to 1 of eight different colors, ranging from black to white.

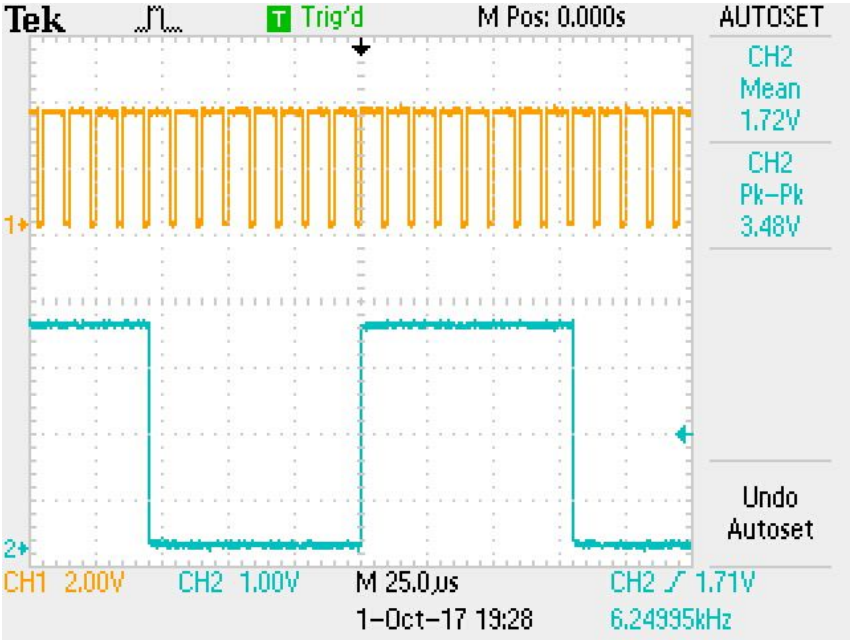



**Figure 5: Produced Square Wave**

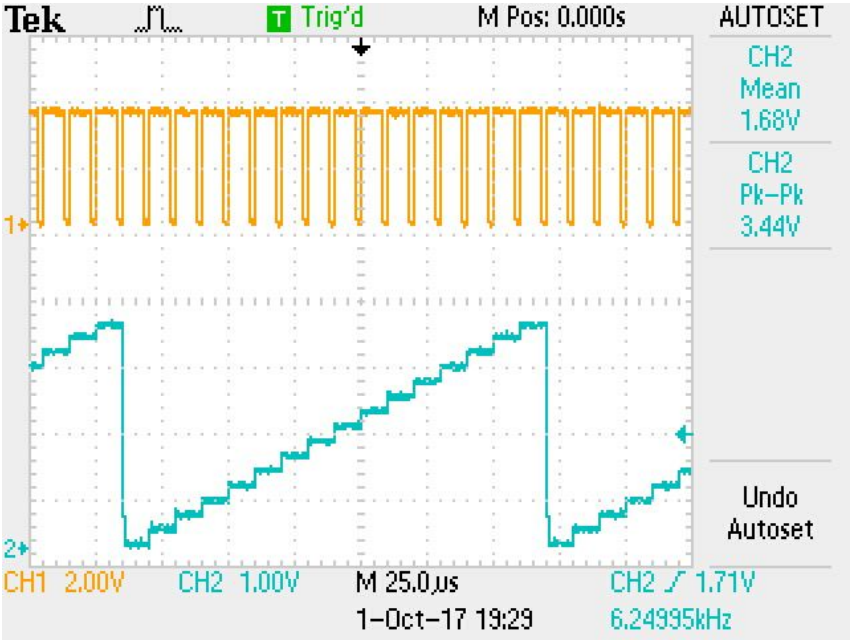



**Figure 6: Produced Sawtooth Wave**

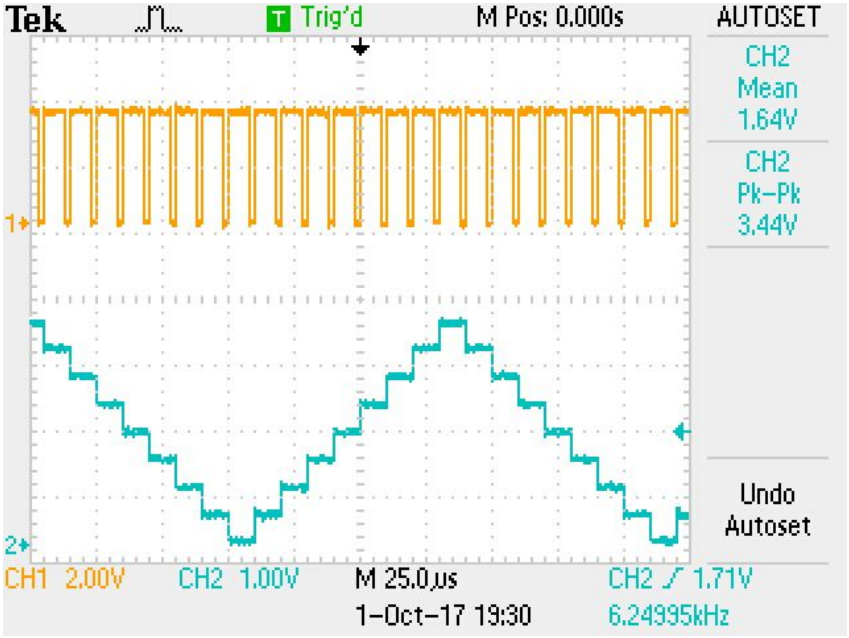



**Figure 7: Produced Triangle Wave**

## Final Solution

In order to implement this design on the the Basys 3 FPGA 107 flip-flops were used, with 183 look up tables and 50 I/O necessary for the final circuit after synthesis. The button_position module consumed the most resources on the FPGA at 35 FF's. Table 1 illustrates the breakdown of flip flops for the button_position module. More than half of the flip flops were used by the counter necessary to create the slower clock to debounce the buttons. The buttons were debounced at a frequency of 100Hz (a period of 10ms) and with the 25MHz clock as the input clock to the module a counter must count up to 249,999 and once it reaches that value the 100Hz flag will go high for 1 clock cycle for the 25MHz clock. In order to count that high 18 bits are necessary because the largest number that can be held in that many bits is 262,143. The remaining flip flops are used by flags that debounce the buttons (two flags per button) and by the registers that store the position of the block (which is based off of the button presses).

Table 1: FF Breakdown for button_position

| Register | Flip Flops Necessary |
|---|---|
| 100Hz Flag | 1 |
| Counter for Flag | 18 |
| Button Flags | 8 |
| Block Position | 8 |
| **Total:** | **35** |

The module that controls the transfer of data to the DAC used 30 FF's after synthesis. The majority of the FF's usd by this module were consumed by the 16-bit shift register that transfers the data to the D0 pin on the breakout board. Two of the FF's used by this module were consumed by pins on the DAC specifically the SYNC pin and the D0 pin. SYNC is driven by the 10MHz clock however SYNC can only go low to begin the DAC's data transfer at a frequency of 100KHz. This 100KHz flag consumes a FF and the counter necessary to control that flag uses an additional 7 FF's. Since the clock driving the DAC is 10MHz a counter that counts from 0 to 99 is necessary to divide the 100MHz clock by 100 to produce a 100KHz clock. Lastly 4 FF's were used to implement the state machine which has 16 states. The breakdown of FF's and associated registers for the DAC control module is shown in table 2.

Table 2: FF Breakdown for DAC_control

| Register | Flip Flops Necessary |
|---|---|
| SYNC | 1 |
| D0 | 1 |
| 100KHz flag | 1 |
| Counter for flag | 7 |
| State Machine | 4 |
| Shift Register | 16 |
| **Total:** | **30** |

The seven segment module requires a clock divider module to drive the anodes at a frequency that the human eye cannot see however it cannot be too fast or else the anodes exceed their switching frequency. This module, called clk_div1k, divides an input clock (in this case the 25MHz MMCM clock) by the 1000 to produce a slower clock at a frequency of 2.5KHz to drive the anodes. In order to divide a clock signal by 1000 a counter and a clock flag are need and will both consume FF's. This module counts to 500 and then inverts the output clock register, creating a signal that alternates every 500 input clock cycles and therefore has a period 1000 times that of the input clock. Counting to 500 requires 9 bits and therefore 9 FF's and the output register requires one FF to implement on the FPGA.

Table 3: FF Breakdown for button_position

| Register | Flip Flops Necessary |
|---|---|
| Clock out | 1 |
| Counter for clock out | 10 |
| **Total:** | **11** |

The seven segment module uses the clock module described above to drive the anodes and also drives the seven segments on each anode. Since there are seven segments to turn on and off individually 7 FF's are required to drive each anode. The anodes register requires 4 flip flops because there are 4 anodes however each one needs its own flip flop effectively creating a one-hot implementation on the FPGA. The overall usage of FF's by the seven segment module are shown in table 4.

Table 4: FF Breakdown for seven_seg

| Register | Flip Flops Necessary |
|---|---|
| Data For Segments | 4 |
| Anode to Display on | 4 |
| **Total:** | **8** |

Lastly the vga controller module provided by Xylinx used up the remaining twenty three FF's as shown in table 5 below. Twenty two flip flops were used by the vertical and horizontal counters and 1 was used by the blank signal.

Table 5: FF Breakdown for VGA controller

| Register | Flip Flops Necessary |
|---|---|
| vcount and hcount | 22 |
| blank | 1 |
| **Total:** | **23** |

The sum of all of these flip flops totals 107 as determined by the Vivado. The resulting circuit uses less than 1% of the available flip flops and look up tables and 47% of the available I/0 on the device.

Simulating this design showed the same operation that was seen on the scope in the lab but in a much faster manner. Debugging the circuit using the test bench file produce the desired output and the same output that was seen in lab. Figure 8 below shows the output of the test bench file and the desired output. This specific implementation also shows the current value that is going to be sent to the seven segment display before it is sent to the DAC. Figure 8 shows

this in data_out bus. The value that is going to be sent is sent across that bus to the seven segment module so the value can be displayed and then once transmission of that data begins the dat_out value is updated to the next value that is going to be sent.

Figure 8 shows one complete cycle of the sine wave output on the FPGA side of the DAC in the form of bits being transmitted over the corresponding pins. The first transmission starts at half of the maximum value as a sine wave does and then swings positive and then negative and then returning at its starting value. The first circle on the left in figure 8 shows the start of one cycle with the state machine in state S0, and the second shows the start of a new cycle, starting back in state S0 after passing through all 16 states, and thus the creating a sinewave output.

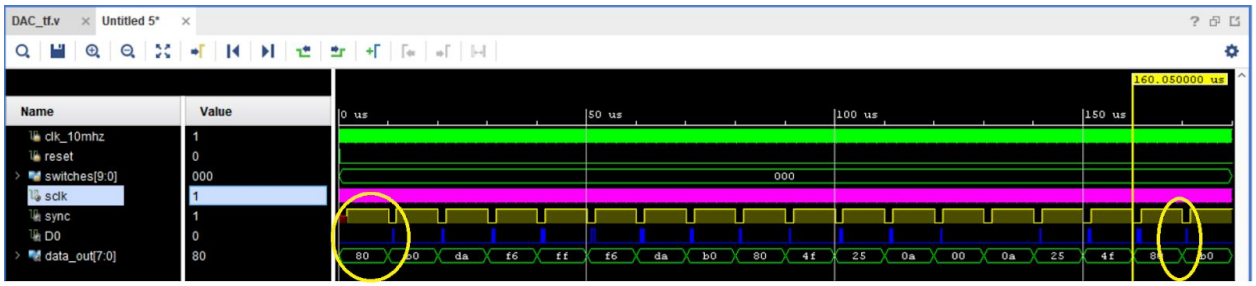



**Figure 8: Test Fixture Output**

A zoomed in view of one transmission is shown in Figure 9 below. The data that is sent to the DAC is sent over the pin D0 using the SCLK and the SYNC pins and the corresponding waveforms are shown below. The data transmission begins with the data that is going to be sent getting loaded into the shift register followed by the SYNC signal goes low on a negative edge of the clock signal (indicated by the leftmost circle). The data that is going to be sent is shown by the data_out register that updates to the next value once the shift register has been loaded. Then on every subsequent negative edge of the clock the shift register shifts and sets the data on D0 so that the DAC can read that information on the following rising edge. After 8 bits of zero (shown by the middle circle) the 8 bits of data begin to send. As before they are loaded on the negative edge of the clock and then on the following positive edge of the clock the DAC reads the bit on the D0 pin. After all 16 bits of data have been read, meaning 16 positive clock edges have passed, the SYNC pin goes high on the following negative edge of the clock.

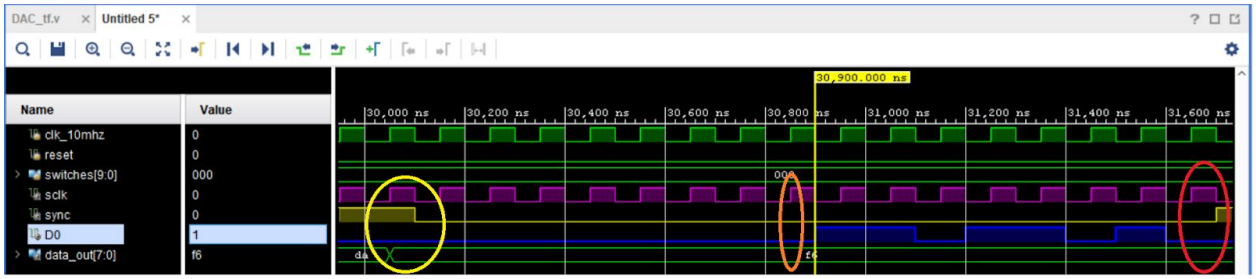



**Figure 9: Test Fixture Single Transmission**

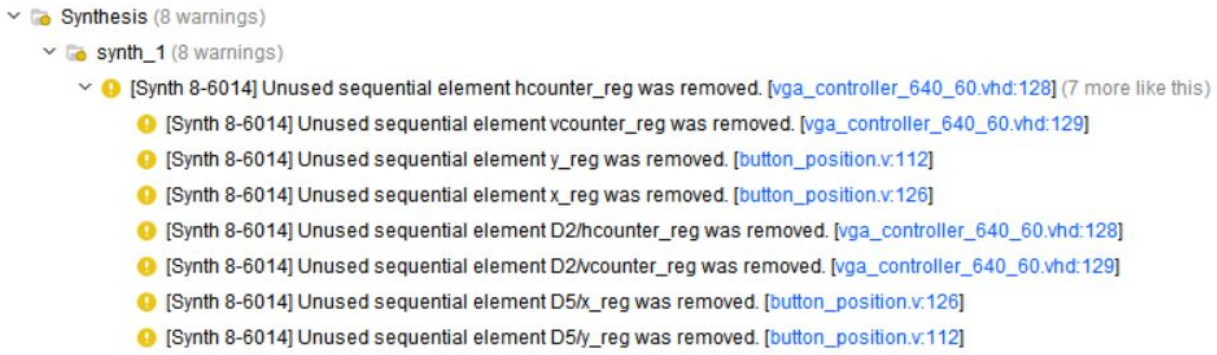



**Figure 10: Warning Messages**

Figure 10 shows the warning messages from the lab implementation. They are saying that the vertical and horizontal counters as well as the x and y position registers are unused. There is a bug in the version of verilog used (2017.2) that sometimes displays this error message when a register is not removed. It appears that these warnings are a result of this bug because these registers must be used for the design to function properly. The horizontal and vertical counter registers are counters that store the current position of the pixel being written to. This cannot be done with combinational logic because these counters are driven by a clock. The same is true for the x and y position registers. These registers store the current x and y position of the block using a clock to control when their values are modified. The block will hold its position indefinitely if no buttons are pressed so this cannot be combinational logic and thus require these registers to function properly. This leaves the implementation having no other errors.

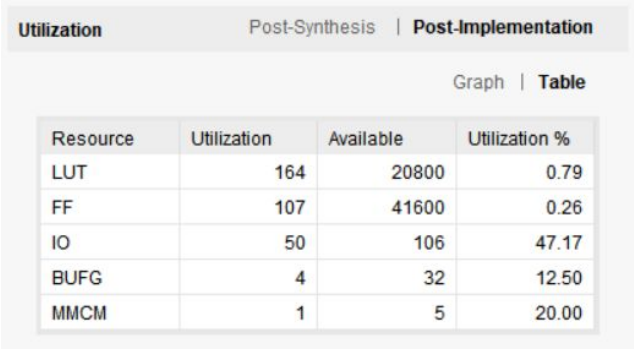
## Conclusions

In this lab a Basys3 board was used to implement a VGA display controller to display a yellow block that can be moved with 4 push buttons, and establish communication with a DAC using the SPI protocol to produce a sine wave. Additionally a sawtooth, triangle wave, square wave and 16 constant DC voltages can be selected using the sliders switches. The colors of the block and background of the VGA display can also be set to 1 of each colors. Multiple modules were created to complete specific tasks, make the debugging process easier and simplify the top level module. The main problems with this lab were establishing proper communication between the DAC and FPGA. Data needed to be shifted and loaded on the negative edge of the SCLK so that the data could be read by the DAC on the positive edge. This was not understood initially, so the DAC was not in sync and this read the data from the shift register incorrectly. At first, setting up the flags for the buttons was producing a lot of errors due to their being 3 flags per button but a later revision simplified this done to one flag per button, making the logic run smoothly. The main lessons learned were how to properly sync an SPI interface that requires actions to be taken on both edges of the clock and when to implement combinational logic or sequential logic to drive buttons. Overall the lab did not take much time as most of the problems were solved quickly. This project could be improved further by simplifying the logic driving the

function generator so that only on state machine is needed for all of the waveforms. Rather than have a unique state machine for each waveform, it would be better to have the switches point to a table of values that correspond to each waveform and have one state machine reference the appropriate one.

## Appendices

Appendix 1: Resource utilization chart



**Utilization** Post-Synthesis | **Post-Implementation**

Graph | **Table**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 164 | 20800 | 0.79 |
| FF | 107 | 41600 | 0.26 |
| IO | 50 | 106 | 47.17 |
| BUFG | 4 | 32 | 12.50 |
| MMCM | 1 | 5 | 20.00 |

Appendix 2: Test Fixture Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: WPI
// Engineer: Matthew DiPlacido Nick Ianotte
// Create Date: 09/30/2017 02:14:29 PM
// Design Name: Lab 3
// Module Name: DAC_tf
// Project Name: Lab 3
// Target Devices: Basys 3 development board
// Tool Versions:
// Description: Lab 3 test bench
// Dependencies:
// Revision:
// Revision 0.01 - File Created
//////////////////////////////////////////////////////////////////////////////////

module DAC_tf(
    );

    //inputs
    reg clk_10mhz, reset;
    reg [9:0] switches;
```

```
    //outputs
    wire sclk, sync, D0;
    wire [7:0] data_out;

    //setup 10MHZ clk
    always
        begin
        clk_10mhz = 0;
        #50
        clk_10mhz = 1;
        #50;
        end

    dac_control uut(
        .clk_10M(clk_10mhz),
        .reset(reset),
        .switches(switches[9:0]),
        .sync(sync),
        .D0(D0),
        .sclk(sclk),
        .data(data_out)
        );


    initial begin
        reset = 1'b1;
        #200;
        //make sure reset is not true
        //set switches for Sinewave output
        reset = 1'b0;
        switches = 10'b00_0000_0000;
        #160000;
    end
endmodule
```