Function Generator and Oscilloscope using the Artix-7 FPGA

CSCE-436 Advanced Embedded Systems - University of Nebraska

Brandon S. Ramos

Contents

1 Milestone 1: Video Synchronization **3**
1.1 Gate Check 1 . . . 4
1.2 Gate Check 2 . . . 5
1.3 Required Functionality . . . 5
1.4 A-level functionality . . . 6
1.5 Hardware schematic . . . 8
1.6 Debugging . . . 9
1.7 Testing methodology or results . . . 10

2 Milestone 2: Data Acquisition, Storage and Display **11**
2.1 Gate Check 1 . . . 13
2.2 Gate Check 2 . . . 14
2.3 Required Functionality . . . 14
2.4 B-Level Functionality . . . 15
2.5 A-Level Functionality . . . 15
2.6 Hardware schematic . . . 16
2.7 Debugging . . . 17
2.8 Testing methodology or results . . . 17

3 Milestone 3: Software control of a datapath **18**
3.1 Gate Check 1 . . . 19
3.2 Gate Check 2 . . . 19
3.3 Required Functionality . . . 19
3.4 B-level Functionality . . . 19
3.5 A-level Functionality . . . 19
3.6 Hardware schematic . . . 20
3.7 Debugging . . . 21
3.8 Testing methodology or results . . . 22

4 Milestone 4: Function Generation **23**
4.1 Gate Check 1 . . . 23
4.2 Gate Check 2 . . . 24
4.3 Required Functionality . . . 25
4.4 B-level Functionality . . . 25
4.5 A-level Functionality . . . 25
4.6 Bonus Functionality . . . 25
4.6.1 Hardware schematic . . . 27
4.7 Debugging . . . 28
4.8 Testing methodology or results . . . 29

1 Milestone 1: Video Synchronization

In this Milestone, we write a VGA controller in VHDL and implement it on the FPGA development board. Using a VGA-to-HDMI module, we can automatically format the output for the HDMI output port on your development board. This VGA controller will be tasked to generate the display portion of an oscilloscope. The scope face consists of a white grid, used to measure the signals, two trigger markers, and the waveforms. In this Milestone, the waveforms will be artificially generated by the code, but in later Milestones, the waveforms will be generated by incoming audio waveforms.

To start off with the design, we will add some sources to the project that will convert the VGA signal into HDMI. Built around this module, we will connect the other modules together to design the scopeface. The design of this Milestone is broken down into separate modules, some of which are sourced and others that will need to be created. The interconnection of the modules is illustrated in the following schematic. When a signal name appears just inside a box, that should correspond to the name of that signal in the entity description. Please note there are a few omissions.

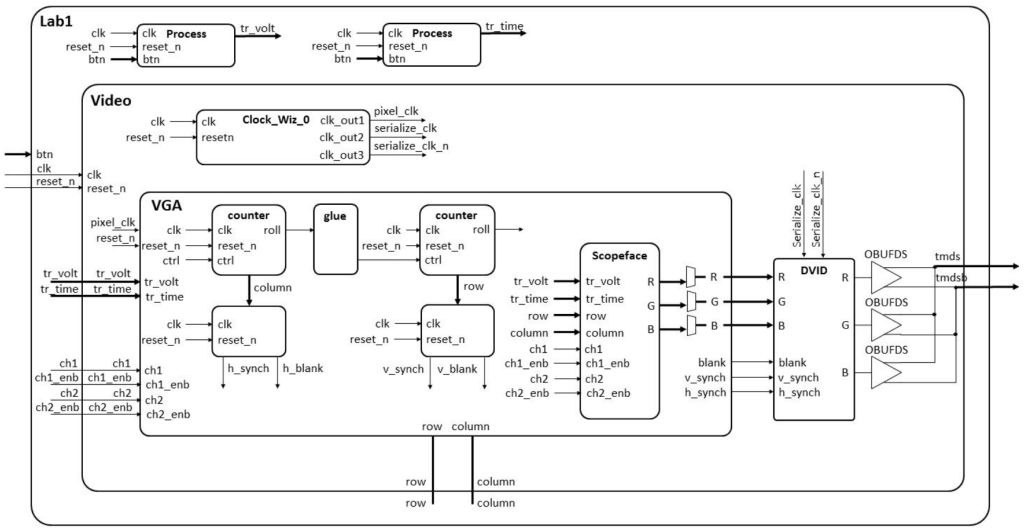

Figure 1: Architecture of Milestone 1

Before designing the VGA controller, to get a video set up on any monitor screen, we need to have some background information on what the standard protocol is. Back when CRT TVs were abundant, there were sync and blanking protocols so that the electron beam could have time to get ready for the next scan line. We have kept this protocol ever since then. Shown below are the sync times and the index of the pixel where we will need to prepare blanks and ready times for VGA.

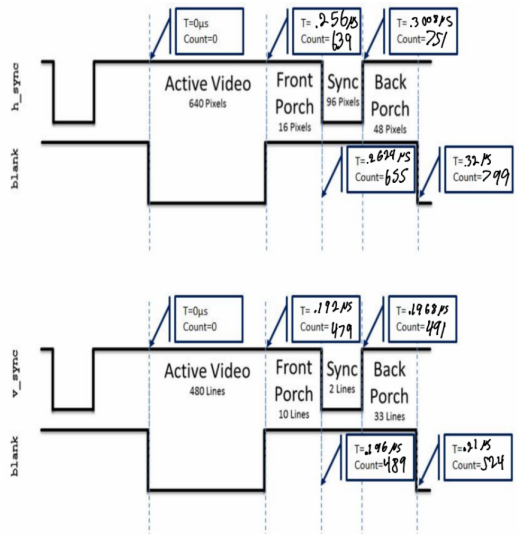

Figure 2: Sync and Blanking Times

Using the image above to proceed, we can now create a rough draft of what the scope face will look like. Shown in the image below will be the vertical and horizontal lines, hatch marks and trigger marks. The trigger marks will be only 9 pixels but will be the channels to trigger volt and time.

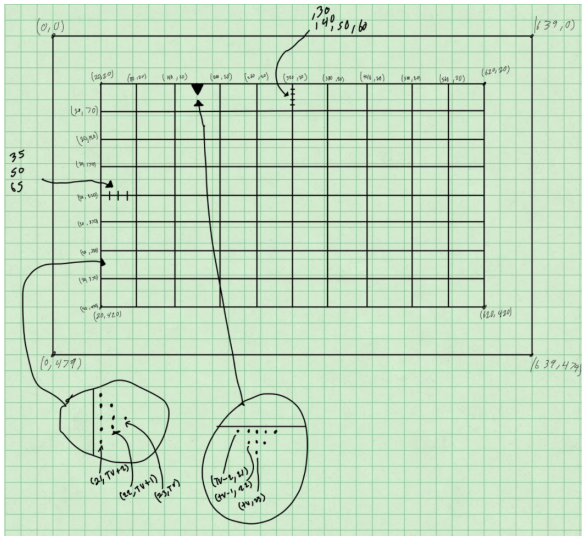

Figure 3: Rough depiction of Scope Face

1.1 Gate Check 1

By the first Gate Check, we finished setting up the VGA counters to generate the proper rows and columns on the waveform. This can be shown with waveform screenshots from the VGA testbench showing the h count rolling over causing the v count to increment. Please note that I will add only one image of the Column Rollover and the other waveforms can be seen on my GitHub to save printer inc.

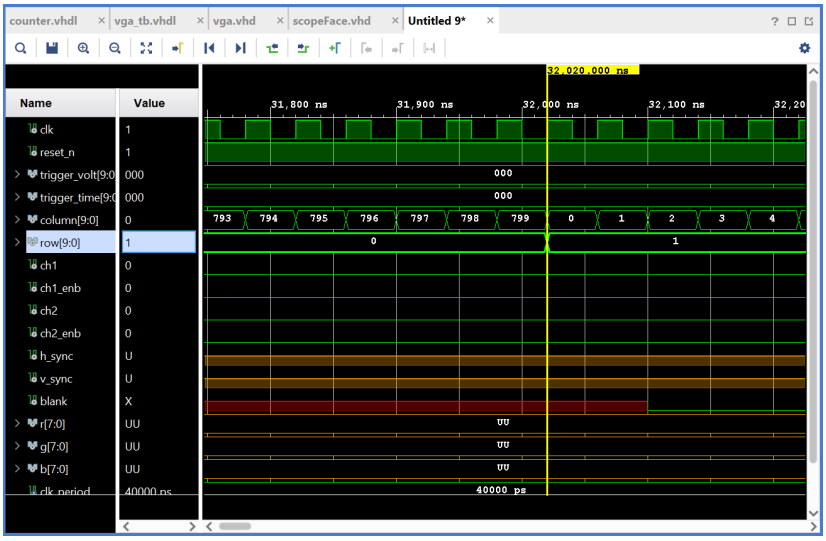

Figure 4: Column Rollover

1.2 Gate Check 2

By Gate Check 2 we have setup the appropriate v_synch, v_blank, h_synch, and h_blank signals on the waveform and created the scopeFace module to draw at least one line on the display. Again, these images can be found on my GitHub. I also created the scopeFace module to draw at least one line on the display. By the time this was achieved, I had already created the code in the scopeface module telling the FPGA what pixels to color in for the grid, hatch marks and trigger/trigger lines.

1.3 Required Functionality

For Required Functionality, the code generates the white oscilloscope grid pattern shown in the picture below. Also, there will be two channel traces. The channel 1 trace (yellow) along a diagonal where (row = column). The channel 2 trace (green) should be drawn along a diagonal where (row = 440-column). This test code is placed in the Lab1 entity. The code below can be seen in the Lab1 entity on the top level, from there it goes to the scopeFace module and gets printed out to the screen.

```
1 ch1_wave <= '1' when row = column else '0';
2 ch2_wave <= '1' when (row = 440-column) else '0';
```

Listing 1: Testing channel 1 and 2 input signals

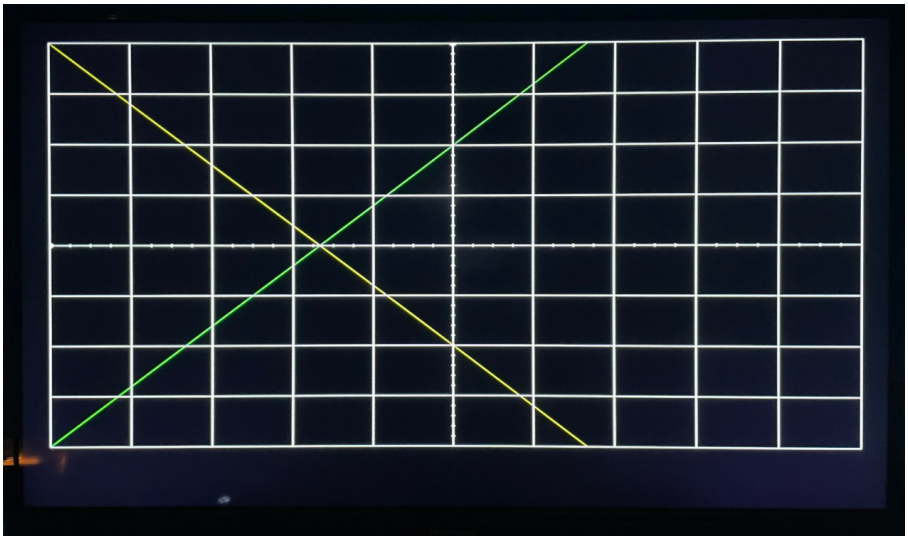

Figure 5: Complete Grid Pattern

1.4 A-level functionality

For A-level functionality, in addition to drawing the display, the display also updates when one of the buttons is pressed according to the list below. Pressing the upper directional button (BTNU) once should move the Trigger Level Marker up. Pressing the lower directional button (BTND) once should move the Trigger Level Marker down. Pressing the left directional button (BTNL) once should move the Trigger Time Marker left. Pressing the right directional button (BTNR) once should move the Trigger Time Marker right. In order to achieve this level of functionality, we need to implement the Process blocks in the Lab1 entity in the schematic. The code for this is shown below. For A-level functionality, you can find the video demo showing button-to-trigger movement on my GitHub or YouTube Channel.

```
1  ----------------------------------------------------------------------------
2  -- the variable button_activity will contain a '1' in any position which
3  -- has been pressed or released.  The buttons are all nominally 0
4  -- and equal to 1 when pressed.
5          -- button_activity <= (old_button xor btn) and btn;
6  ----------------------------------------------------------------------------
7  -- The buttons are all nominally 0 and equal to 1 when pressed.
8  --       btn(3) = '1'                        Right
9  --                btn(1) = '1'                          Left
10 --                btn(2) = '1'                          Down
11 --                btn(0) = '1'                          Up
12 --                btn(4) = '1'                          Center
13 ----------------------------------------------------------------------------
14 process(clk)
15 begin
16         if (rising_edge(clk)) then
17                 button_activity <= (old_button xor btn) and btn;
18                 --Reset trigger
19                 if (button_activity(4) = '1') then
20                         trigger_time <= to_unsigned(320,10);
21                         trigger_volt <= to_unsigned(220,10);
22                         button_activity <= (others => '0');
23                         old_button <= (others => '0');
24             --Move trigger right
25                 elsif (button_activity(3) = '1') then
26                       if(trigger_time+10 <= 620) then
27                           trigger_time <= trigger_time + 10;
```

```
28                      end if;
29
30              --Move trigger left
31                  elsif (button_activity(1) = '1') then
32                      if(trigger_time - 10 >= 20) then
33                              trigger_time <= trigger_time - 10;
34                          end if;
35
36              --Move trigger up
37                  elsif (button_activity(0) = '1') then
38                      if(trigger_volt - 10 >= 20) then
39                              trigger_volt <= trigger_volt - 10;
40                          end if;
41
42                  --Move trigger down
43              elsif (button_activity(2) = '1') then
44                if(trigger_volt + 10 <= 420) then
45                          trigger_volt <= trigger_volt + 10;
46                end if;
47                  end if;
48
49                  old_button <= btn;
50          end if;
51  end process;
```

Listing 2: Button functionality

Starting off with a module to count the rows and columns. This is done with an entity that is used twice where we cascade them together to get a rollover. One is a mod 525 counter and the other is a mod 800 counter.

```
1   ctrl
2   0           hold
3   1           Q+1 mod 5
```

Listing 3: Counter Control

```
1   process(clk)
2   begin
3           --generic counting module
4           if (rising_edge(clk)) then
5                   if (reset = '0') then
6                           processQ <= (others => '0');
7                           rollSynch <= '0';
8                   elsif ((processQ < countLimit) and (ctrl = '1')) then
9                           processQ <= processQ + 1;
10                          rollSynch <= '0';
11                  elsif ((processQ = countLimit) and (ctrl = '1')) then
12                          processQ <= (others => '0');
13                          rollSynch <= '1';
14                          rollCombo <= '0';
15                  end if;
16
17                  --prepare rollover
18                  if (processQ = countLimit -1) then
19                      rollCombo <= '1';
20                  end if;
21          end if;
22  end process;
```

Listing 4: Counter Module

The main idea of the first Milestone is to get the counter incrementing the display column and rows. This by default will give us the individual pixels on the screen where we can tell the display what pixels need to be white, yellow or green. Depending on the row and column index, we print out the different colors on the screen.

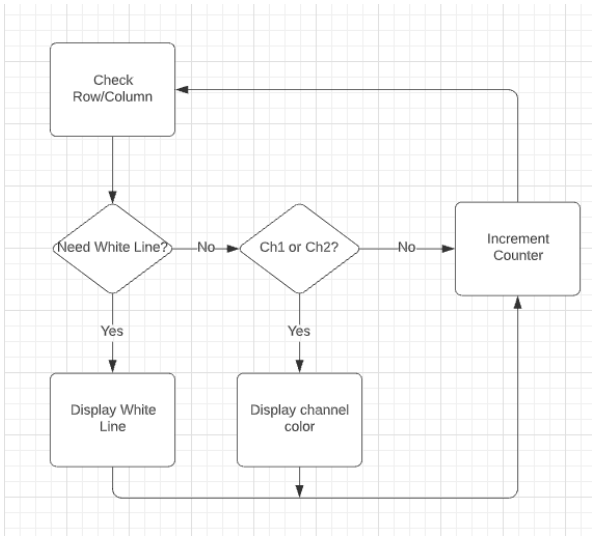

Figure 6: Flow Chart of displaying pixels to screen

1.5 Hardware schematic

The VGA Module: The main task is to build the VGA component for Milestone 1. This component sweeps across the display from left to right, and then return to the left side of the next lower row. The VGA interface determines the color of each pixel on this journey with the help of the scopeFace component.

```
1  entity vga is
2          Port(    clk: in  STD_LOGIC;
3                   reset_n : in  STD_LOGIC;
4                   h_sync : out  STD_LOGIC;
5                   v_sync : out  STD_LOGIC;
6                   blank : out  STD_LOGIC;
7                   r: out STD_LOGIC_VECTOR(7 downto 0);
8                   g: out STD_LOGIC_VECTOR(7 downto 0);
9                   b: out STD_LOGIC_VECTOR(7 downto 0);
10                  trigger_time: in unsigned(9 downto 0);
11                  trigger_volt: in unsigned (9 downto 0);
12                  row: out unsigned(9 downto 0);
13                  column: out unsigned(9 downto 0);
14                  ch1: in std_logic;
15                  ch1_enb: in std_logic;
16                  ch2: in std_logic;
17                  ch2_enb: in std_logic);
18 end vga;
```

Listing 5: VGA Module

Behavior: The VGA component contains a pair of cascaded counters which generate the row and column values of the current pixel being displayed. The row and column values are used to generate the blank, h_synch and v_synch signals according to the Figures above. The scopeFace component (more on this below), takes the row and column values (along with some other information) and generates the R,G,B color of that pixel. The three muxes on the output of the R,G,B output of the scopeFace component output the scopeFace R,G,B values for row,column values within the 640x480 displayable region, or 0's for values outside this region.

The scopeFace Module: Inside the VGA module sits an instance of the scopeFace entity. This entity only contains combinational logic. When given a row,column pair, its responsible

for generating the R,G,B value of that pixel.

```
entity scopeFace is
        Port (  row : in  unsigned(9 downto 0);
            column : in  unsigned(9 downto 0);
                        trigger_volt: in unsigned (9 downto 0);
                        trigger_time: in unsigned (9 downto 0);
            r : out  std_logic_vector(7 downto 0);
            g : out  std_logic_vector(7 downto 0);
            b : out  std_logic_vector(7 downto 0);
                        ch1: in std_logic;
                        ch1_enb: in std_logic;
                        ch2: in std_logic;
                        ch2_enb: in std_logic);
end scopeFace;
```

Listing 6: scopeFace Module

Behavior: The scopeFace component takes in the current row,column coordinates of the display and generates the R,G,B values at that screen coordinate. For example, if row,column = 20,20 then the R,G,B output should be 0xFF,0xFF,0xFF (white) because the upper left corner of the O'scope grid display is white. Note, you can get the RGB values for common colors at various online websites.

The Digilent board will have a lot of connections required to make this work. The image below shows where to make these connections to get Milestone 1 to work. The main connections are power, HDMI to the screen and micro USB to program the device.

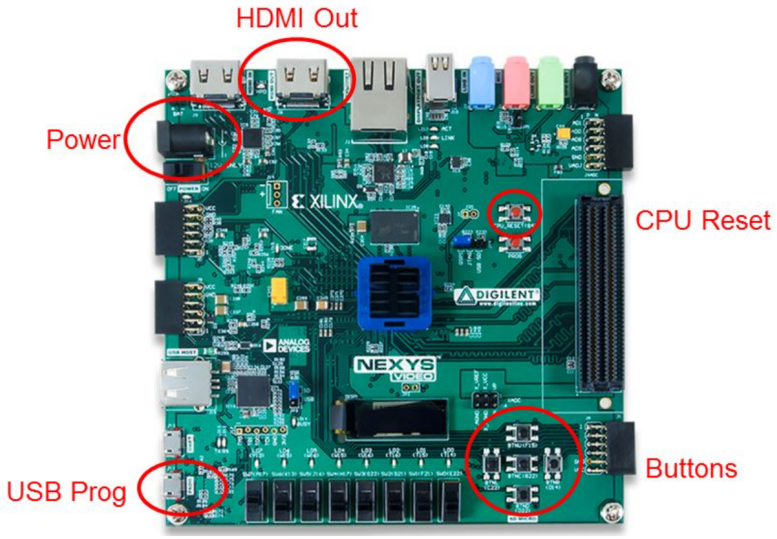

Figure 7: Image of ports to hook up for testing

1.6 Debugging

There were many issues along the way that had to be dealt with. The first case of debugging I went through was when I was creating the mod "xxx" counter where there was a generic integer value passed on and would count up to that specified number. The problem arises when I cascaded two generic counters together. I needed one to rollover and count up the next counter. Instead what happened is that it would either count up too early or too late. This was due to the roll signal that was having a problem with timing. So when the counter was

up to the limit minus one, I would set a rollcombo signal that would get ready to roll the next counter. This was an easy fix and would be used from then on.

The next problem was when I was setting the blanking signals. the issue was the timing as well with length of blanking, and sync. This was fixed by changing the different values of when the timing was for blanks. I found that it would typically be off by one since I usually had to only add an "=" sign to my comparison of "¡".

My big problem that confused me the most was the initial values of the trigger marks. My intension was to have the trigger marks set up in the center line of their respective vertical and horizontal lines were, instead they stayed at the initial marks at zero. My first goal was to go to the top level and change the initial value of the marks but that kept them at zero still. So at this point I knew they had to be reset before they could be printed. So when I created the buttons, I also created a reset signal that put them at their marks. This was a fix and no problems persisted after.

My last problem was the debouncing of the buttons. To fix this I had to check if the buttons had pull-down or pull-up resistors and find out how to reset. In lab, we were given a button_activity variable that basically debounced the buttons for us. It XOR'd old_button and button_activity together to see if anything was changed. This software debouncing technique is not fullproof, but will work for the most part.

1.7 Testing methodology or results

Some testing to get the scopeface to work can take some time, especially when you build a bit file. Instead, simulating using a testbench can take less time and show you crucial information such as bus transfer and logic information. The first testing method that was used was in the beginning with gate check 1. To check if the counter was going up correctly, we simulated and checked out the waveform that was produced with a clock. On the other hand when working with trying to get the line drawn on the screen. First thing I did was to make sure my RGB signal is being outputted properly. Thankfully the simulation is quicker and you are able to see if the "G" signal is being outputted as x"FF" for a green screen. The last testing method was with the buttons. This is where we would move the trigger marks and change the x and y axis for the triggers. To test this I would check again the simulation to see if the trigger values change +- 10 units.

During the project, what I have noticed that was critical to getting the VGA sync done was to make sure the timing of the signals were on time. Now as long as you were in the give or take one pixel from your blanking target, you didn't have any problem. Noticing the give in the signal processing, there were students having trouble displaying a line on their screen and I can almost guarantee that it came down to timing of the sync, blank and display signals. Otherwise it could be a bigger problem such as a faulty board.

Something to take away from this Milestone would be that the protocol for displaying pixels may be the same, but the history on where sync and blanking came from is an interesting concept to learn. Given that, creating the code had to be exact. Using this information that I was given for this, I will be able to add on to this with the soon to be Milestone 2.

2 Milestone 2: Data Acquisition, Storage and Display

In this Milestone, we integrate the video display controller developed in Milestone 1 with the Audio Codec on the Nexys Video board to build a basic 2-channel oscilloscope. When complete, Milestone 2 should generate an output to the scope face display.

Architecture of Milestone 2 is broken down into a bunch of separate modules, shown in the block diagram below. Some of the components in the block diagram will be built off of work done previously. It is important to note that some of the components and signals associated with this diagram will not be needed for Milestone 2, but have been included because they will be needed in the final design.

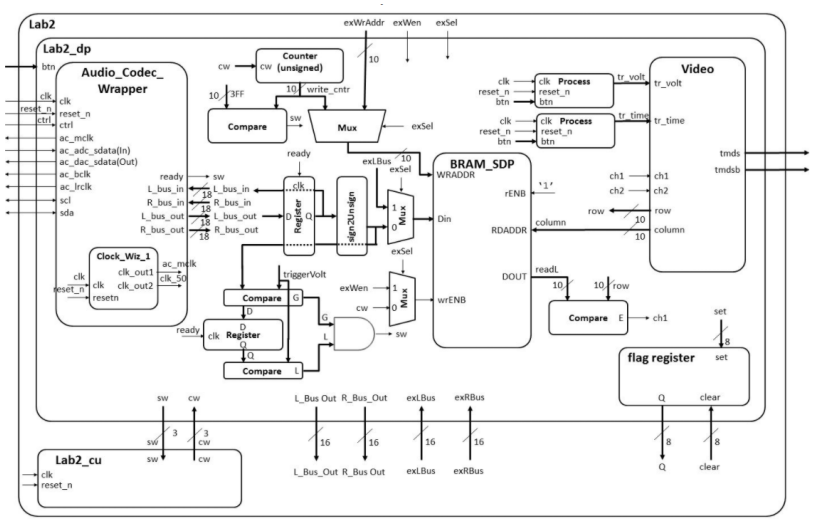

Figure 8: Architecture of Milestone 2

Consider the data in the diagram as flowing from left to Right. There will be an input signal to the Nexys board via a 3.5mm cable connected to the blue line-in jack. This signal then passes through an Analog Devices ADAU1761 SigmaDSP Audio Codec. The ADAU1761 samples the audio input at 48kHz into separate 18-bit 2's complement left and right channels. The ADAU1761 then transfers this data to our Artix 7 chip over a serial bus through the 7 signals on the left side of the Milestone 2 component. More information about the audio codec on page 27 of the Nexys Video manual, there will be a link to the Manual in the Documentation.

The serial protocol coming from the audio codec is quite complex, so given will be the Audio Codec Wrapper component as an interface to extract the incoming signal (Audio Codec Wrapper in the Figure above). Whenever new converted data is ready from the Audio Codec, the ready signal will go high for a single clock cycle. The circuit will then do two things with the incoming L_bus_out and R_bus_out signals: First, it will loop both of these signals back into the Audio Codec so that we can verify (by listening on the green line-out jack) that the Audio Codec hardware and firmware are operating correctly. This is accomplished using the VHDL code below.

```
1 -- Audio Code Loopback Process:
2  process (clk)
3         begin
4         if (rising_edge(clk)) then
5         if reset_n = '0' then
6                 L_bus_in <= (others => '0');
```

```
7                 R_bus_in <= (others => '0');
8             elsif(ready = '1') then
9                 L_bus_in <= L_bus_out;
10                R_bus_in <= R_bus_out;
11            end if;
12          end if;
13  end process;
```

Listing 7: VHDL code for audio loopback

Second, the circuit will need to send the L_bus_out and R_bus_out signals in an unsigned format to be stored in the block ram (BRAM). To do this, we will need to convert the 2's complement values to unsigned. Performing this conversion is technically easy, so let's do some examples. Consider the table below.

Input Value		Ouput Value	
2's complement	2's value	unsigned	unsigned value
100...000	-131072	000...000	0
111...111	-1	011...111	131071
000...000	0	100...000	131072
000...001	1	100...001	131073
011...111	131071	111...111	262143

Figure 9: I/O values for audio bus

The values in the left 2 columns represent the 2's complement number coming out of the Audio Codec, in this case L_bus_out, while the right 2 columns represent the output of the box labeled "sign2unsign" in the block diagram. What we are essentially doing is shifting the unsigned values up by 131072 (the value which will make all 18-bit numbers positive) so that they all fall into a positive range.

Here are a couple more pieces of info to help understand the block diagram above. Consider the mux with its input going to the WRADDR input of the "BRAM_SDP" block in the block diagram. This mux circuitry attached to the write address of the BRAM will be used in Milestone 3, allowing the microBlaze processor to take over write duties for the RAM (as opposed to an external signal from the Audio Codec). Independent of the write circuitry, the read circuit pulls data from the RAM, and draws the waveform. Unlike for the write circuitry, the read circuitry requires no FSM control.

In later Milestones, we will be integrating most of the components from this Milestone with the MicroBlaze processor (a processor we program onto our FPGA). In order to make this smooth, we will need a way to transfer information between the two systems a technique similar to a 2-line handshake. To make this possible, you will need to build a component called a flag register. The behavior of the flag register is shown in the table below.

reset_n	clk	set	clear	Q+
0	X	X	X	0
1	0,1,falling	X	X	Q
1	rising	0	0	Q
1	rising	1	0	1
1	rising	0	1	0
1	rising	1	1	X

Figure 10: Control for Flag Register

The flag register will interface our Milestone 2 component with a MicroBlaze as follows: The component will produce some data, put it on a data line to the MicroBlaze, and then set one of the bits of the flag register. Then, the MicroBlaze will, at some point, look at the flag register bit. When it sees that the 'set' bit is 1, the MicroBlaze will grab the data from the register and clear the set bit. These are just like the flag bits on the MSP 430. Use the following entity declaration for the flag register:

```
1 entity flagRegister is
2         Generic (N: integer := 8);
3         Port(    clk: in  STD_LOGIC;
4                          reset_n : in  STD_LOGIC;
5                          set, clear: in std_logic_vector(N-1 downto 0);
6                          Q: out std_logic_vector(N-1 downto 0));
7 end flagRegister
```

Listing 8: Flag Register VHDL

The set lines should be connected to the signals below. For the time being, you can leave the Q outputs to open.

-ready, -v_synch, -write_cnt compare output

We need to map the ports of BRAM to include it in the datapath. The component is declared in the UNIMACRO library - look at that library to figure out how to port map BRAM.

```
1 library UNIMACRO;                    -- This contains links to the Xilinx block RAM
2 use UNIMACRO.vcomponents.all;
```

Listing 9: Header Library

Generating Audio Waveforms: Since we need to use a 3.5mm jack to input signals to the Nexys board, a phone's audio output works quite well. However, make sure you get an app where you can control both the left and right audio channels individually.

2.1 Gate Check 1

By Gate Check 1 we must have dropped in the Video, VGA, Scopeface, dvid, and tdms files from the previous Milestone into this project in order to test if the Scopeface works when we implemented the Audio Code Wrapper. We will also have to re-implement the Milestone 1 Clocking Wizard in the Milestone 2 project. Doing this will eliminate a lot of errors from un-driven output signals.

Next, we will need to have implement another Clocking Wizard and the Audio Codec Wrapper inside the Datapath entity to get the Audio Codec to begin functioning. Once we

fully implement the Audio Codec Wrapper, we will drop in the Loopback process and make connections to loopback the serial ADC input back out to the DAC output (i.e. send the signal back into the Codec). Once we implement the design on the board, we can verify functionality by applying an audio signal to the audio line in jack (blue) and listening to it on the audio line out jack (Green) using a standard oscilloscope. Additionally the Scopeface and Button inputs from Milestone 1 should be functional as well.

From the GitHub for this project, there will be a VidGrid video of the functionality. This includes the scopeface and the audio codec loopback process that can be heard through a 3.5mm audio jack output.

2.2 Gate Check 2

By Gate Check 2 we have implemented and connected the left channel BRAM and BRAM Address Counter to write Audio Codec data to BRAM. Once implemented, we can verify the BRAM works by using the given datapath testbench and watching the BRAM write address increment and data be written/read from the BRAM. Once this is working, we must implement Video entity to take the left channel output from BRAM and send it to the Channel 1 waveform to be displayed when the readL value equals the row value. Once implemented, this functionality can be verified first with the given datapath testbench to verify the channel 1 values are being updated properly when readL equals the row value. Additionally, we may try to implement this on the hardware and verify that your scopeface is still present and some values are being displayed for Channel 1 (at this point the waveform will be scrolling across the display or may be scaled wrong).

In the GitHub, you can be seen the BRAM waveform that shows the unsigned 18 bit value going in and out. Below this on the readme.md there is a second image of the scopeface where the scrolling of ch1 across the grid. This is due to not setting the trigger and Finite State Machine States.

2.3 Required Functionality

Get a single channel of the oscilloscope to display with reliable triggering that holds the waveform at a single point on the left edge of the display. A 220Hz waveform should display something similar to what is shown in the screenshot at the top of this page. Additionally, there must be the following done: Use a package file to contain all component declarations. Use separate datapath and control unit. Datapath must use processes which are similar to our basic building block (counter, register, mux, etc.). Not one massive process in the datapath. Testbench for the flagRegister, control unit and datapath unit showing data (different value than what is given in the testbench) coming out of the audio codec and being converted from signed to unsigned and then to std_logic_vector to go into your BRAM. Include calculations to back up what the waveform shows. For Bonus Points: Testbench for the datapath unit showing that same data coming out of the BRAM. Make sure to show the read address and the data values coming out. This will require to set the control words on the testbench. Additionally, you will have to drive the pixel_clock on the Video Module. Once we get the datapath testbench running, we will notice that DCM module doesn't put out a clock in the Video Module.

The first figure below can be shown as the functionality where the Channel 1 wave is being triggered off of the trigger volt on the right side of the scope. In the documentation there will be a VidGrid video of the functionality. The next figure is the Flag Register Testbench where we set, clear and output the registers contents out. This is using a single line of code where

our registers contents are assigned "(Register OR (set AND (NOT clear)))" to the flag. Next is the Datapath Testbench, this will show a waveform of different values that go through BRAM. The specific signals to look at are the counter, BRAM in and out (which will be changed to unsigned and subtracted by 292) and control/set words. I will add the scope face and state machine below in consideration of printing images.

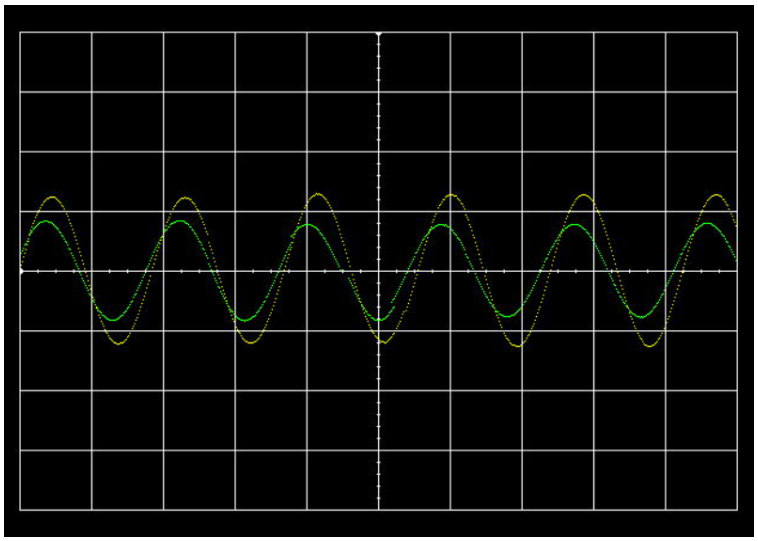

Figure 11: Ch1 triggering off of trigger volt

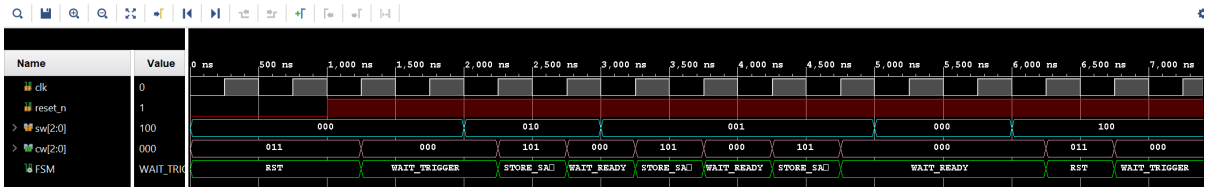

Figure 12: Finite State Machine Testbench

2.4 B-Level Functionality

For B-Level Functionality we ensure that our design will meet all the requirements of required functionality as well as add a second channel (in green). Be able to move the cursors on the screen.

2.5 A-Level Functionality

For A-Level Functionality we ensure the design will meet all the requirements of B-level functionality and use the trigger voltage marker to establish the actual trigger voltage used to capture the waveform. As the trigger is moved up and down, we should see the point at which the waveform intersects the left side of the screen change.

The control unit or Finite State Machine (FSM) has selected states that have to transition from one to the other defined by the FSM designer. The FSM has 4 states where we start at Reset from CPU reset. The control unit is determined by a 3 bit set word (sw) that transitions

the states. The control word (cw) is the output of the FSM and sw that are sent to the datapath as shown in the architecture.

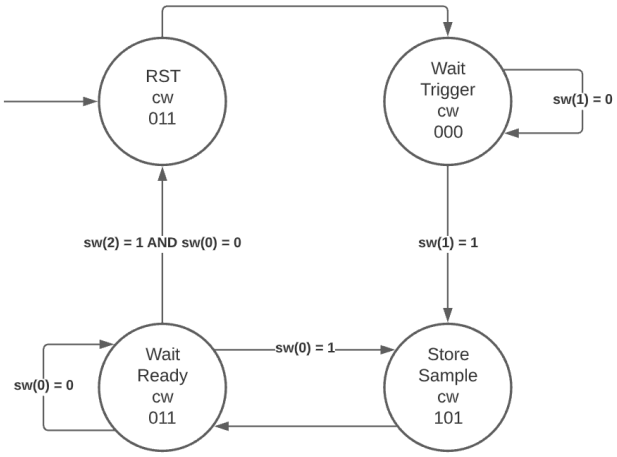

Figure 13: Finite State Machine

Below is the control word and set word table that determines the states and output of the FSM. Each word is 3 bits long and can be set individually. However, the cw two lower bytes determines the counter and the higher bit controls write enable. The set word has its 3 bits controlled individually where they are also checked according to its respective state that it is in at the time.

```
1  ------------------------------------------------------------------------------
2  The cw
3  00                          hold
4  01                          count up
5  10                          unused
6  11                          synch reset
7  There is a cw(2) bit that is set 1 when BRAM write enable, else 0
8  ------------------------------------------------------------------------------
9
10 ------------------------------------------------------------------------------
11 The sw
12 001     ready from audio codec
13 010     2 compare
14 100     reaches 1023
15 ------------------------------------------------------------------------------
```

Listing 10: Control and Set Word tables

2.6 Hardware schematic

The Nexys Board will be used again to draw out waveforms on the scopeface. The Audio I/O is used for signal in for the scopeface and the output is if we want to listen to the waveform generated. And of course we have the CPU Reset in case we need to start back at the beginning. The setup used was a 3.5mm audio jack from a computer to the board for signal processing. The HDMI out from the board was an input to a HDMI to USB capture card that is sent into a computer and VLC Media Player to either screen record or take screenshots. Milestone 2 was programmed and generated by the same computer using a USB.

2.7 Debugging

Debugging Gate Check 1 was simple but implementing it for the first time was different. All we had to do was to take Milestone 1 and to move it over to Milestone 2 where we put it inside the datapath with its original functionality. Gate Check 2 may have been the hardest part. Debugging was difficult and the concept of the BRAM connections was a lot to go over. The BRAM itself was easy to set up the signals and get mux to select the counter. But if I can not see what the problem is or even know what the next step is, it is very difficult to debug. This is where found myself with TA Jacob Fox and talking to Professor Falkinburg about my issues. I found out that the conversion was not difficult to change the scale and shift the value up the scope. When it came to printing out the ch1 wave to the screen I had to comment out all of the codded process I had changed cw to "101" so that write enable and counting was being done. This produced the scrolling seen earlier but I still couldnt figure out why my FSM didn't work. Eventually I found out my reset if statement was checking for an active high (not active low) reset and was never going into the FSM states. The last issue was in the triggervolt section. This is where we compared the previous bus value to the current and checked for a rising edge essentially. I had found out that my unsidned value was not created correctly and that I needed to subtract -292 from the value. This, and some tweaking eventually got the trigger mark to work. Before It would attempt to find a trigger but would reset before it came close.

2.8 Testing methodology or results

Testing in Milestone 2 was done by either making a testbench for a certain entity or to generate the bitstream and figure out what was not working. The generation of the bitstream was not as fast as simply making a testbench and running the simulation, but I would look around my code while the bitstream was being generated. I had created a testbench for both the control unit and flag register but not the top level entity. The entity was tested on the board using the top level design. Other testing such as checking the mux values and unsigned values were done in the datapath testbench where we could confirm that the in/out signals were the same.

From 18 bits coming in from the audio codec to the scopeface. We had to ask ourselves how to get it to fit on the screen. With essentially 18 bits we took the top 10 bits of ths signal and changed it to unsigned and added 2 to the 17th. This gave us a positive integer that was from 0 to 1024. From there the signal was not centered on the x axis so by decreasing the value by -292 we had the signal centered.

The other question we had to answer before making the FSM was what would our cw and sw be? I went with the standard shown in the datapath testbench and created the sw as it is in the flowchart shown earlier.

Capability: The horizontal axis represents time. There are 10 major divisions on the display; how long does each major division represent? Since there are in total 800x535 pixels (420,000) we know that number divided by 60 FPS gives us 7000p/s and by the division line being 600 pixels wide, 600p/7000p/s = 85.7ms divided by 10 divisions givs us 8.57ms/division. Each major time division is split into 4 minor division, how long does each minor division represent? Each hatch mark in those 10 divisions on the time axis represents 2.14ms/hatch mark. Generate a sine wave that can be fully captured on your display. record its height in major and minor vertical divisions. Measure this same audio output using the break out audio cable or connect to the tip or ring and ground to the sleeve ground. Record the peak-to-peak voltage. Compute the number of volts in each major and minor vertical division. Starting at address 0, how long does it take to fill the entire memory with audio samples (coming in at 48kHz)? It would take

1024/48KHz = 21.3ms to fill up the BRAM. How long does it take to completely draw the display once? Roughly 1/60 = 16.7ms. The question is likely relevant to Milestone 3 - how long is the vsynch signal held low? Since the screen is essentially 800 pixels wide, it would be 800x2 = 1,600pixels/7000pixels/s = 229ms.

3 Milestone 3: Software control of a datapath

In Milestone 3, we integrate the video display controller developed in Milestone 2 with the MicroBlaze processor built using the fabric of the Artix-7 FPGA. Using our knowledge about the Vivado and SDK tool chains, now it's time to put that knowledge to the test by building a software controlled datapath. Milestone 2 revealed some shortcomings of our oscilloscope that this Milesstone intends on correcting. Specifically, we will add:

- A horizontal trigger point
- The ability to enable and disable which channels are being displayed
- The ability to trigger off of channel 2
- The ability to change the slope direction of the trigger The following figure shows required functionality - the program should allow the user to change the position of the triggerVolt and triggerTime indicators with the result that the waveform should be drawn so that the periodic waveform is increasing through that voltage at that time.

In order to accomplish this, we will need to make some minor changes to our top level component, create a new piece of IP, and then program that IP using the MicroBlaze, as described in the block diagram below. We will walk through these steps below.

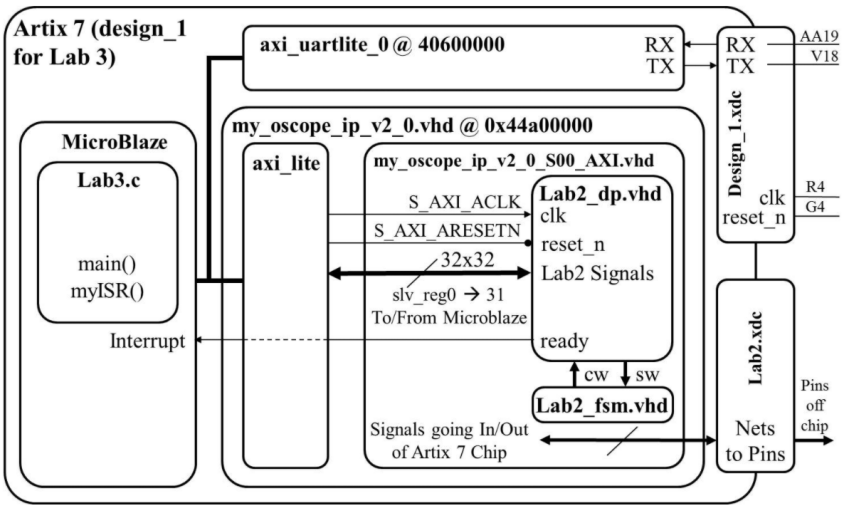

Figure 14: Architecture of Milestone 3 with Milestone 2 Integrated IP

Note: In the program, we will provide a user menu (through the terminal), allowing the user to adjust the trigger voltage and trigger time. Therefore, we may want to check if the user has hit the key on the keyboard without having to actually read the key. For these cases, the following command will prove useful. Note that "uartRegAddr" is a constant, the address of the uart.

```
1 'XUartLite_IsReceiveEmpty(uartRegAddr);'
```

Listing 11: UART Register Address

3.1 Gate Check 1

By Gate Check 1 we are able to have all of the Milestone 2 functionality implemented with the Microblaze. That is, we need to export Milestone 2 design into the SDK and be able to achieve the same functionality as we did in Milestone 2. Video of functionality is shown in Documentation on GitHub or my YouTube channel.

3.2 Gate Check 2

By Gate Check 2 we are able to send USART commands using Tera Term (or another terminal emulator) to the FPGA to adjust the trigger on the screen. The trigger on the screen should properly react to moving the trigger either up or down.

3.3 Required Functionality

In order to achieve required functionality, we needed to properly trigger the oscilloscope on channel 1 using a positive edge trigger. Control of this process is to be performed using the MicroBlaze. The main tasks of the MicroBlaze will include:

- Move audio samples into a pair of circular buffers. These circular buffers will be maintained in the address space of the MicroBlaze. That is, we should have two big arrays defined in your program. Use polling of the ready bit of the flag register.
- Examine the samples, looking for a trigger event.
- Fill the remaining sample slots in memory.
- Move the appropriate buffer values into the display memory of the oscilloscope (top level) component.
- Provide a user menu (through the terminal) allowing the user to adjust the trigger voltage and trigger time.

3.4 B-level Functionality

- Achieve required functionality.
- Use the ready bit of the flag register to trigger an interrupt. The ISR should store the samples (left and right), look for a triggering event, and signal when the stored samples should be transferred to the BRAM in the oscilloscope component.

3.5 A-level Functionality

- Achieve B-level functionality.
- Ability to enable and disable channels to display
- Ability to trigger off channel 2

- Ability to change the slope direction of the trigger.

Using one bit from a vector to trigger an interrupt: In order to achieve A functionality, this design requires the programmer to use a single bit of Q (the std_logic_vector output from the flag register) as the interrupt signal. This may require the programmer to extract the one bit Q as a separate signal to connect to the MicroBlaze in the block design.

All the memory mapped hardware registers will have their names setup as #define's with a name ending in "Reg". Any register with bit fields will have the bit index setup as #define's with a name ending in "Bit". The flagQ and flagClear registers need to be at the same address. Below is a flowchart of reading in data from the Left and Right bus using the ready signal. Note that the flowchart is referring to an algorithm where the Interrupt Service Routine (ISR) is not being used. In this case the algorithm uses polling to grab data where it waits in essentially a while loop waiting for the ready signal to go high. For later functionality this is discarded and a ISR is used instead. This way we can enter the ISR when the data is ready instead of slowing down the software further waiting for the ready signal.

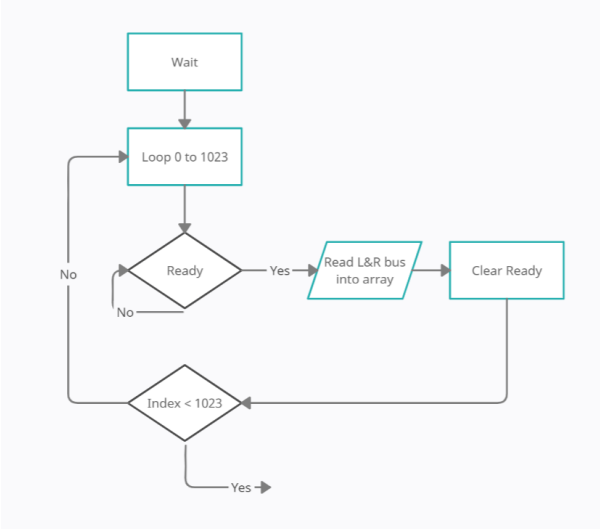

Figure 15: Functionality flowchart showing algorithm without ISR (Interrupt Service Routine)

A good portion of the functionality i.e. Rising/Falling Edge trigger, Ch1/Ch2 trigger or the top level internal functionality can all be changed by using a u8 data type and changing the values to their respective "Boolean" values. For example, to trigger off of channel 1 we would need to enable a variable. From the variable we look at an if statement and send it off to the slave register to be read in the datapath.

3.6 Hardware schematic

With the exception of the following Engineering Change Orders (ECO) in the table below, the hardware developed in Milestone 2 will be unchanged. For the following ECO, please refer to the high-level architecture in Milestone 2.

The Nexys Board will be used again in the Milestone to draw out waveforms on the scopeface. Below are the connections used in the Milestone such as the general power, HDMI Out and USB Prog. Others like the buttons are disabled in Milestone 3 by the ECO. The Audio I/O

is used for signal in for the scopeface and the output is if we want to listen to the waveform generated.

The setup used in this Milestone was a 3.5mm audio jack from a computer to the board for signal processing. The HDMI out from the board was an input to a HDMI to USB capture card that is sent into a computer and VLC Media Player to either screen record or take screenshots.

Name:	Trigger Voltage, Trigger Time
Scope:	lab2_dp and lab2
Type:	Change to the entity descriptions.
Details:	• Inside the lab2_dp component, remove the logic driving the triggerVolt and triggerTime signals into the video component. • Remove the buttons signal from the lab2 and lab2_dp entities. • Remove the buttons signal from the xdc file. • Add the triggerVolt and triggerTime signals to the lab2 and lab2_dp entity descriptions. • Drive the triggerTime and triggerVolt inputs on the video component with the corresponding signals on the lab2_dp entity.

Figure 16: Engineering Change Orders (ECO) Table

The first step will be to create a component for the top level component in a Vivado repository. This will require the programmer to think about what signals are routed to the MicroBlaze and what signals are going outside the Artix 7 chip. The following table should help.

Name:	Trigger Voltage, Trigger Time
Scope:	lab2_dp and lab2
Type:	Change to the entity descriptions.
Details:	• Inside the lab2_dp component, remove the logic driving the triggerVolt and triggerTime signals into the video component. • Remove the buttons signal from the lab2 and lab2_dp entities. • Remove the buttons signal from the xdc file. • Add the triggerVolt and triggerTime signals to the lab2 and lab2_dp entity descriptions. • Drive the triggerTime and triggerVolt inputs on the video component with the corresponding signals on the lab2_dp entity.

Figure 17: Milestone 2 signals to port out to software and outside design

3.7 Debugging

Gate Check 1 came with its own set of problems. Gate Check 1 used slave registers to push and pull data from the MicroBlaze to the datapath of the top level module. The issue that I had was deciding what needed to go on the read side of the slave register. Changing the slave registers was the heap of the debugging process, eventually using the lower 32 registers.

Gate Check 2 required the functionality of the slave registers to read data from the hardware. The problem that I was having with this gate check was the use of initial values to the read side of the registers. The debugging process included deleting the wrapper files in Vivado that were in the SDK and then rebuilding them and exporting it back to the SDK. Once I rebuilt the software and compiled, rumnning the software allowed for trigger manipulation with the "wasd" keys

Later in the Milestone before implementing the ISR, it was found that organized scattering of the waveform could be seen below in the picture. This is known to happen if the software is slow enough that it essentially skips the array index. Once the ISR is added to the software, this was seen to show better results and less scattering. A good idea was to use printf's whenever checking if the ISR was being used or if values were as expected. At one point the ISR was not loading and this caused the Left and Right bus arrays to be all zeros. Following the wire it was shown that there was an active low reset that was causing the flag register to be zero all the time. As soon as this was reversed, the ISR was working as expected.

Towards the end of the Milestone looking at the waveform. It was apparent that the software was not printing through all of the values in the bus array. On the Scopeface there was scattering pixels of a sine wave.

Though this O'Scope is not bulletproof, it was interesting to see that even with a perfect array of values of the sine wave, it was not printing every index. In the software, I used a for loop to make sure every time the index 'i' was being sent to the slave register. This included both the address and value to BRAM. After building this I was able to get my waveform to look as it did in previous Milestones.

The last problem I encountered occurred during the process of getting A-level functionality. It seemed as if the Right bus was being printed out exactly as the Left bus was. Using Ctrl+F to find the instances where I used the Right ram. For the longest time looking at all the code in the software. Professor Falkinburg and I both were looking in the hardware to see if there was a possibility that the values were not correctly connected to the external L/R bus. Following the wires it seemed that the connections were in the right place, but when in the datapath there was one little mistake. VHDL is not a case sensitive language, there was one letter that was lowercased that I changed to make sure it couldn't cause any problems. While in the file for the datapath. I saw this line of code:

```
1 'Rbus_out <= STD_LOGIC_VECTOR(L_bus_unsigned(17 downto 2));'
```

Listing 12: Audio Codec Right Bus

This was the Rbus_out that sent the data from the Audio Codec to the external bus. The code was checking the Left unsigned bus and putting it into the Right channel, essentially duplicating the waveform. A simple L to R change allowed the right to properly be displayed. A good part of debugging in this Milestone was using printf. The printf is what showed what was being put into registers and onto the slave register.

A noteworthy comment is that while you may trigger off channel 1 or 2, you must print the other channel on the screen. Otherwise you will get a line for channel 2 across the screen while printing channel 1

3.8 Testing methodology or results

The nice thing about building the bitstream and exporting it to the SDK, long waits for compilation is not an issue anymore. Just compile and run to change the triggerVolt and triggerTIme. Using the Lbus and Rbus we can fill in a buffer for the functionality checkpoint to bring in new data for a waveform to trigger with both trigger marks. Print statements helped out with checking values going inside and out of the bus. If a value was being incorrectly displayed, you could follow the value to see where it is being manipulated in the wrong way. Same with commenting out the code and checking if the bare bones work. This meant reading in the data and putting it right back into the slave register to get a wave output. From there I would uncomment code to find the broken code.

One question that I brought up quickly was that did we need the button functionality? They were allowed to stay on as long as they did not cause any trouble with Milestone 3 but were eventually removed. This was for the sake of time, resources and complexity. They were more trouble than needed, although they were only needed for Milestone 2.

Another question that was easier to answer was where to enable the interrupt after disabling it. Due to how Vivado ran, disabling interrupts inside the ISR was not an option. Instead, there was a function that did the data manipulation to get the trigger intersection and the two options for placement for the enable ISR was either before or after the function. It would be

in real time if the enable interrupt was set before the data manipulation but would be easier to enable it after the function whereas the array would be the previous state. This meant that the L/R bus values would not be the most recent values contained withing the ISR. But this was okay since the software was not time sensitive for its application so to speak.

4 Milestone 4: Function Generation

The goal of this Milestone is to generate an audio waveform with a high degree of accuracy in both its period and frequency.

We are to use Direct Digital Synthesis to reproduce your audio waveform. We may choose any waveform so long as its not Piecewise Linear. A few interesting examples would be sinusoids, the sine function, exponentially damped sinusoids, or a waveform from a musical instrument (guitar, piano, or clarinet). It is our responsibility to get the samples for this waveform. Suggesting either deriving the waveform using a program like Python, using a spreadsheet, or digitizing the samples using Milestone 3. Once we have the data, hardwired it into BRAM using "init" statements.

While we have the flexibility to design the waveform generator as we see fit, your system must meet the following requirements:

- Use an update rate of 48kHz
- At 440Hz, the LUT should be incremented by about 1 index.
- Be able to make between a 1Hz and 0.25Hz change in frequency.
- Be able to generate a full amplitude waveform.

We will have to generate the block diagram for this design. The design must be segregated into a datapath and control unit. Your design must show the blocks in the datapath, the states in the FSM, the control word, and the status word joining the datapath and control unit.

4.1 Gate Check 1

At Gate Check 1, we have a completed hardware diagram drawn in paint or another image editing software that is readable and can be printed onto an 8.5x11 sheet of paper. This diagram must contain the following:

- A border defining the top-level entity. Borders for each of the components instantiated within the top-level entity.
- All components must be named in the upper left corner.
- All signals entering and exiting components must have their port name defined just inside the border.
- All signals outside the components must have their width defined as well as be labeled with their names.

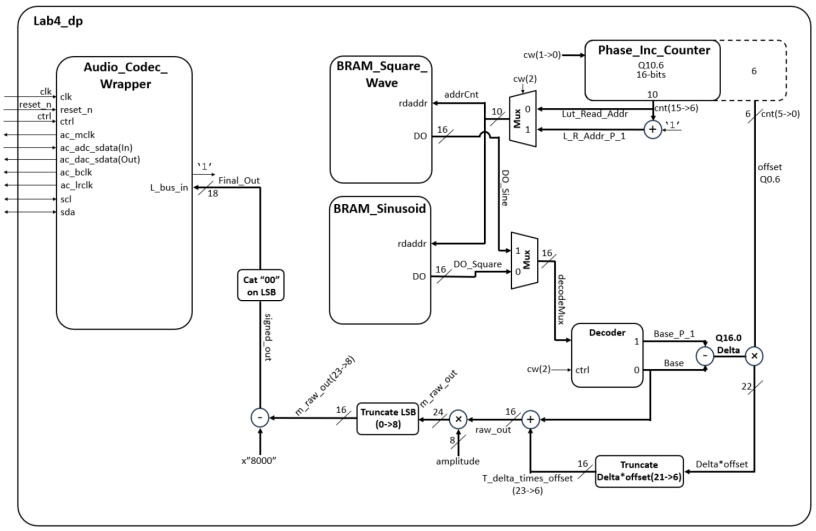

Figure 18: Diagram of Milestone 4

4.2 Gate Check 2

At Gate Check 2, we have a working testbench. When simulating the design, have the testbench supply a mock ready signal in place of the ready signal generated the Audio_Codec_Wrapper (when put in a testbench, the Audio_Codec_Wrapper is not able to generate a ready signal without a lot of extra work).

When complete, expect the timing diagram to look like the image below and contain at least:

- clk
- reset
- ready (simulated using CSA statements in testbench)
- FSM state
- BRAM address
- Phase increment
- BRAM data out
- Amplitude coefficient
- Multiplied data out
- Slide switches
- Button values The simulation needs to simulate a button press (and release) to change the phase increment. After that is done, we need to show that the BRAM address is being incremented by your new phase increment value.

Milestone 4 waveform will be in my GitHub to conserve printer ink for this report.

4.3 Required Functionality

Use the slide switches and push buttons to manipulate the phase angle and the amplitude of the waveform as follows:

- Pressing the left button should decrease the frequency of the waveform by the amount set on the slide switches.
- Pressing the right button should increase the frequency of the waveform by the amount set on the slide switches. The waveform should be played back through the Audio Codec interface. Remembering to wait for the ready signal.

4.4 B-level Functionality

- Pressing the up button should increase the amplitude of the waveform by the amount set on the slide switches.
- Pressing the down button should decrease the amplitude of the waveform by the amount set on the slide switches.
- Pressing the center button should toggle between 2-different waveforms.

4.5 A-level Functionality

Use the microBlaze to capture a keyboard input to manipulate the amplitude and frequency. The user will enter in an integer frequency and you are to produce a waveform with that frequency.

4.6 Bonus Functionality

Since Required functionality doesn't specifically require interpolation based on the two values in the LUT. If we modify the state machine to interpolate the values using the following equation:

```
1 -- Linearly_Interpolated_Value = Base + Offset*Delta;
```

Listing 13: Linear Interpolation Equation

When it came to the portion where we had to create the 1024 values in the sine wave and upload them to the BRAM, this was a time consuming process. Instead of doing it by typing or using excel, I decided to write a script in Java to speed up the process. Below is a code snippet of what does the concatenation for the user. This could have been done in TCL or Python just as easily.

```
1  public class main {
2          public static void main(String[] args) {
3                  String[] array = new String[16];
4                  int count = 0;
5                  try{
6                          File file = new File("squareWave.txt");
7                          Scanner fileScan = new Scanner(file);
8                          for(int every16 = 0; every16 < 64; every16++) {
9                                  for (int index = 0; index < 16; index++) {
10                                         array[15 - index] = fileScan.nextLine();
11                                 }
12                                 System.out.print(String.format("INIT_%02X => x\"",count)
                                       );
13                                 count ++;
```

```
14
15                                  for (int i = 0; i < 16; i++) {
16                                          System.out.print(array[i]);
17                                  }
18                                  if(every16 < 63){
19                                          System.out.println("\",");
20                                  }else{
21                                          System.out.println("\"");
22                                  }
23                          }
24                  }catch(Exception e){
25                          System.out.println("File not found\n");
26                  }
27          }
28  }
```

Listing 14: Java Script

In the control unit, there is only one finite state machine (FSM) that controls the datapath. In this FSM there are four main states, three which we will look at.In Wait_Ready we are waiting for the ready signal coming from the audio codec. From there we go to our first address by increasing the address using the phase increment. Once we store the base in a register we choose the base + 1 to look for the next LUT value. Below can be shown the states.

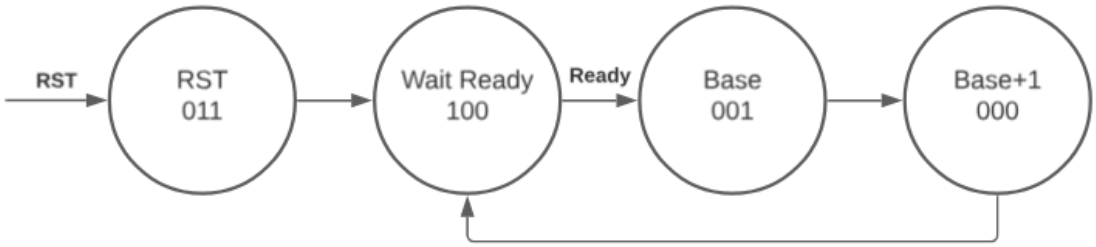

Figure 19: Finite State Machine States

In another diagram shown, we have a flowchart of the states. This is where we can change the counter so that we can have different frequencies.

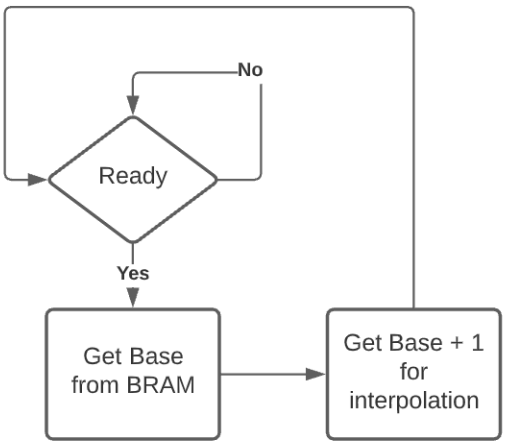

Figure 20: Counter Inc/Dec from FSM

4.6.1 Hardware schematic

For the connections from the Artix-7 Board, we will be using some from previous Milestones but others will not be used. Connecting the board to power and programming it via USB are the two main connections. We will now be using the switches to change the phase and amplitude as a Q2.6 value. The buttons will do this phase shift increase/decrease (left and right buttons) and amplitude increase/decrease (up and down buttons) for the waveform. We will not be using the audio in or the HDMI out to be displayed. The only source coming out from the audio codec is the left bus out to an audio source.

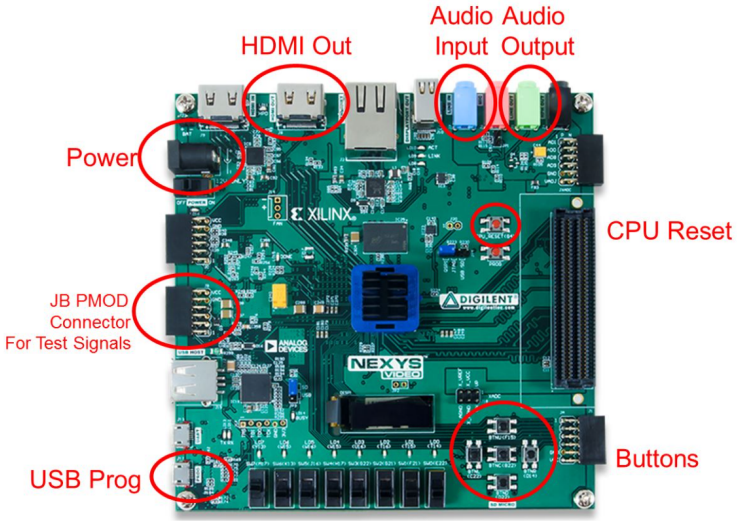

Figure 21: Connections to and from the Nexys Board

4.7 Debugging

Getting behind the concepts of Milestone 4 was the difficult part in a sense. Gate check 2 can be seen to have some troublesome getting it up and running. The main difficulty came from determining what Gate Check 1 had inside doing the math. To solve the issues shown in Gate Check 2 were to go through the math on paper (or a calculator) to make sure the values displayed were correct. This is how I got most of the signals to be displayed in the waveform. Coming to the bottom signals shown in the waveform in Gate Check 2 were the product, offsets and deltas used for interpolation.

As Projessor Falkinburg says, follow the wires. This helped when I was having problems at the end of the Milestone. I noticed that as soon as I tried to change the phase inc and amplitude of the signal, nothing audible could be heard to dictate a change. Of couse making sure the Q2.6 value was set to anything other than zero for the phase and 1 for the amplitude. I also recently added the other BRAM so that the audio codec could switch from using one waveform to another by a select signal to a mux. Testing this there was an audible difference in the two signals, this could only mean that the switches were not connected but the buttons were. Eventually following the wires to the top level vhd file there seems to have a signal for the switches coming in but not digging deeper into the datapath file where it is needed the most. After connection of the switches, I was able to change phase and amplitude as designed.

After Milestone 4 was complete, there was still one last problem I faced. My sine wave was spiking at the peak amplitudes of the waveform, this happened both for my sine and square wave. I had to go double check my signals to make sure it wasn't happening in the math portion of the datapath and when I was working with the signals it hit me. When working with the DDS spread sheet given in a previous lecture, I remember when I was trying to create my own square wave for the lab that there was something to look out for. If you were adding a bunch of sine functions together you had to make sure the value could fit inside a 4 digit hex value, otherwise you'd get an overflow. So I decided to divide all of my real sine values to get then all under the value of 1. From there I uploaded to the board and as shown below is my results.

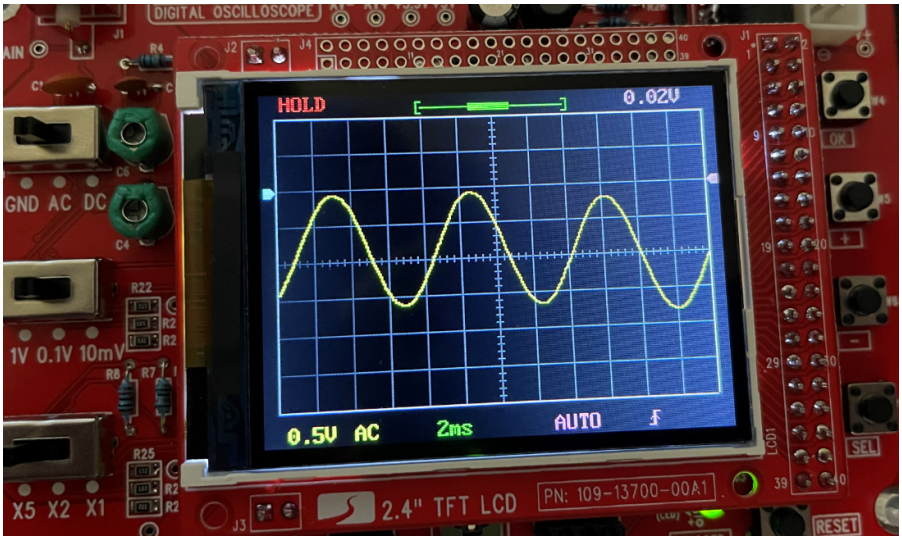

Figure 22: Sine wave without spikes

Also, this was happening to my square wave which I had to be especially careful with the values. They would sometimes still spike even with values below 1. So making sure the values

of the Real Sine were in range less than ".98" I uploaded them to the board and got the image below.

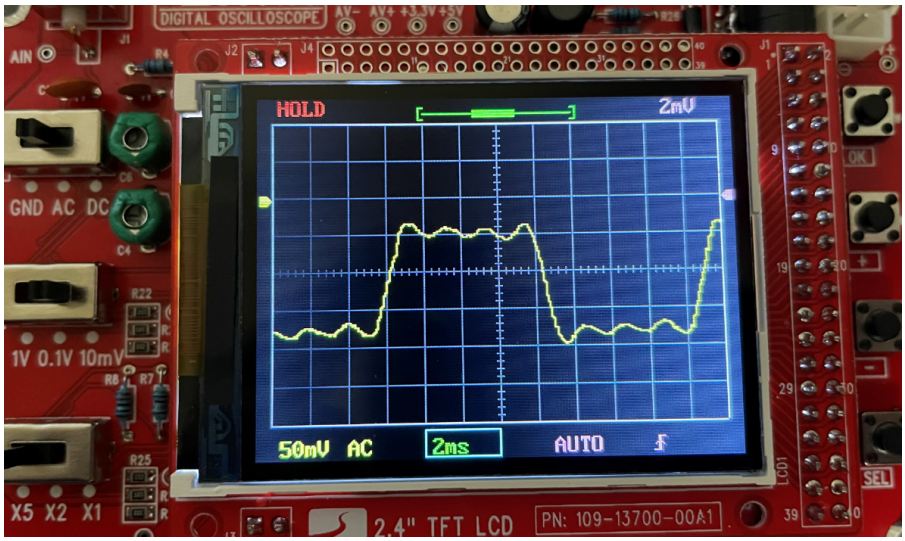

Figure 23: Square wave without spikes

4.8 Testing methodology or results

Using the testbench at the beginning of the Milestone helped out to check signal processing. The testbench was also a required for the gate checks so this made students use the waveform instead of building the bitstream and wasting valuable time. Using this waveform, I double checked the values going into BRAM and the data coming out. I knew for an initial counter value of zero. I should get 0x8000 for the first LUT of the sine wave. From there I followed the signals throughout the math and made sure to use unsigned and signed values where needed. During the multiplication of signals, there has to be a wider bus to transfer 22 wide signal if the inputs are a Q0.6 and a Q16.0. One question I had was how to use interpolation to get a better sine wave. As shown in Gate Check 1 where we draw out the diagram for the Milestone, we noticed that there is a 16-bit counter where the upper 10 bits are for addressing BRAM. The lower 6-bits of the counter are for the interpolated value later. Once we get the LUT value coming out of BRAM into the D0 signal into a decoder, we can find the current LUT value and the next LUT value. This becomes our delta signal and will be multiplied by the offset which was that lower 6-bits from the counter. This is how the interpolated value goes through the hardware and eventually goes through to the left bus out from the audio codec.