ECE 385

Fall 2022

A Motion Controlled Synthesizer

AKA: The Puppeteer Theremin

Final Project

Matthew Esses

TA Tianhao Yu

12/13/2022

The Story and Inspiration

Originally, I wanted to make an elevator model (external) and control system on an FPGA but I was advised against this due to low possible difficulty. The next idea obviously was a video game. I looked into some of them, but one that interested me a lot was Duck Hunt. I felt that using a keyboard, mouse would be a bit too uninspired though. So I looked into motion controls. I then realized the potential that motion controls would have on FPGA, and by the time we reached lab 7 and I had a strong disdain for working with VGA. I talked to some of the UAs about their projects, and a few of them completed ones on audio and received rather high difficulty ratings.

The first step was to find a source for motion data, via an accelerometer. My first thought was a video game controller with a direct interface with the FPGA. I tried making my own app to send motion control data from my phone to my computer, but I do not know Kotlin or Java. I tried looking for documentation on the communication protocols with Wii Remotes, and I realized this would cost much more than I would be willing to pay and that unfortunately all/ most Nintendo communication protocols are protected trade secrets.

Coincidentally, I had also begun porting the save data from my Nintendo Switch and emulating on my laptop using the Yuzu emulator during this time. One game I ported was “The Legend of Zelda: Breath of The Wild”. The game itself rendered beautifully with my AMD Radeon Graphics Card, yet there were parts of the game that I couldn’t play because they required motion controls. Thankfully, others in the emulation community had that figured out already. They had created applications such as Motion Source to be able to create the illusion of motion controls via a LAN server connection from an Android Phone. They had also created a program called PadTest that was able to validate whether or not the connections were accurate that read unit vector acceleration and gyroscope.

This is where the gears in my head began turning. If I found some way to either create a virtual keyboard for the FPGA, then maybe there could be some way that I could use a python application to read the app data. The first thing I did to figure out how to do that was a brilliant Google search for “python code to read text on screen.” This led me to the following stackoverflow post: https://stackoverflow.com/questions/68305486/how-to-read-text-directly-from-screen-using-python

This was exactly what I needed to read screen data using Tesseract-OCR. I kept running it, and I began to run into inconsolable issues with reading the vanilla UTF-8 font text from the PadTest application. I then tried to learn how to train my own model using the following tutorial: https://towardsdatascience.com/simple-ocr-with-tesseract-a4341e4564b6
This still had issues, so I just used regex to filter bad reads. I then easily converted the vector directions into ASCII characters and finished that portion of the project.

The next step was finding a way to move the data from Python onto the FPGA, so I made a post on the ECE 385 discord explaining what my issue was, and the UA Ian pointed me in the direction of the jtag_uart library in Python. After an hour with him, I was successfully able to send my motion control data to my FPGA. I then spent the next night figuring out how to do something other than light up hex displays with my data, which I decided was to do lab 6.2, but with motion controls. It only took me so long because I was taking in the Hex Driver outputs instead of the Hex Driver inputs, and it was truncating the bits, and in addition, trying to redo lab 6.2 from the beginning with the Platform Designer multiple times. From there, I had motion

controls fully implemented on FPGA. However, then my grandmother passed away over thanksgiving break, and I was unable to do anything at all regarding the project for around a week and a half, and then I had to complete my ECE 470 final project all by myself because I was the only one in my group who understood Linux OS. However, this gave me time to realize what I had to do.

With only one week left before the final project demo, the next step was three all-nighters where I attempted to build the I2C module where I was able to work with the audio input from the SGTL5000 microphone input and output it through headphones. I brought in my platform designer from lab 6.2 and modified it, it worked so I worked on bringing in an I2S module to play ROM sound data, and everything broke. I tried a different platform designer module. It was still broken. I tried working with lab 7 and modifying it from there. It was even more broken. I did this around 20 more times, and it was still broken. Until the next night, a UA Zayd told me “Find a 6.2 platform that works and go from there.” I used the save of the lab 6.2 platform from the night I was trying to get the ball to move, and it finally worked again. I learned from my mistake of not having prior saves while working, and made a copy project, verified it still worked, and moved on from there.

I had finally caught up with all the other groups that were doing audio, almost where nobody had any idea how to get audio to actually work from RAM. I had my I2S, a simple bit shift register verified by a TA as well as a group who was modifying input audio created in Audacity, and it should have worked. The issue I found was shown to me by one of the first people to get the RAM and I2S to work, a classmate named Sergei. The issue was that everyone was both instantiating RAM incorrectly, as well as using the wrong pin to input into the I2C. I made the promise to myself, I would go to sleep once it worked or once I figured out I2S.

The next day, I was talking to a friend and UA Phil and he told me more about audio sampling and how to get the correct sample size by dividing the 44.1KHz sound clock by the given note frequency. By doing this, the audio codec outputs a sound very close to the goal pitch, even with extremely small sample lengths. I then instantiated and piece tested each note as an individual RAM block to save on time, which may have taken days while there were only three days before the final demo) as well as memory, which ended up only using 12% of total memory and 12% of the on chip memory. Long before this, I also decided that I am bad at instruments and that I really cannot get anyone else to reliably play this, so for my own wellbeing as well as the well being of everyone’s ears in a 10 kilometer radius, I decided to only implement the key of C, which is good enough to play songs from “The Legend of Zelda: Ocarina of Time”.

Next steps were boring, and I just needed to write out a small finite state machine for deciding which audio to put into I2S. I decided to directly map them to the Hex display inputs, and it worked without any issues.

I then tried to make the video game snake as well, but ran out of time before my demo. I tried scrapping the idea before the demo but I was bitten by my hubris. In the end after around a week and a half’s worth of sleepless nights, I finished creating my motion controlled synth/ puppeteer theremin and I took part in the demo showcase playing Zelda’s Lullaby. It was an extremely fun, stressful, and surreal experience inventing an instrument and being the only person in the world to have the logistical capability or eccentricity to play it.

Introduction

Within my final project I created a completely new digitally synthesized instrument controlled via motion controls called a “Puppeteer Theremin” which receives accelerometer input from two phones and thereby 3 dimensions. Through this project I learned more about AI, computer vision, audio processing, the Windows Kernel, USB spoofing, and organology.

To convert the notes to a readable format, I modified a Librosa script given to me by the TA Pramod to convert .wav files produced by Audacity to create a .HEX file to load to the RAM of the FPGA.

I designed my top level design in System Verilog around the materials from lab 6.2, instantiating ROM modules modified from the ones found in Rishi’s helper tools, an I2S module, an I2C generated from platform designer, the SGTL5000 test programs provided by the course, and two finite state machine modules, one for selecting notes, one for displaying them.

The other part of my project was through a Python program that received input from two different Android phones from an application called Motion Source via reading the screen of a third program on the host computer called PadTest by using Tesseract-OCR to read the program’s text and controlling the JTAG connection to the FPGA.

The SGTL5000 Audio Codec

According to the datasheet:

> “The SGTL5000 is a Low Power Stereo Codec with Headphone Amp from Freescale, and is designed to provide a complete audio solution for products needing LINEIN, MIC_IN, LINEOUT, headphone-out, and digital I/O. Deriving its architecture from best in class, Freescale integrated products that are currently on the market. The SGTL5000 is able to achieve ultra low power with very high performance and functionality, all in one of the smallest footprints available. Target markets include media players, navigation devices, smart phones, tablets, medical equipment, exercise equipment, consumer audio equipment, etc. Features such as capless headphone design and an internal PLL help lower overall system cost.”

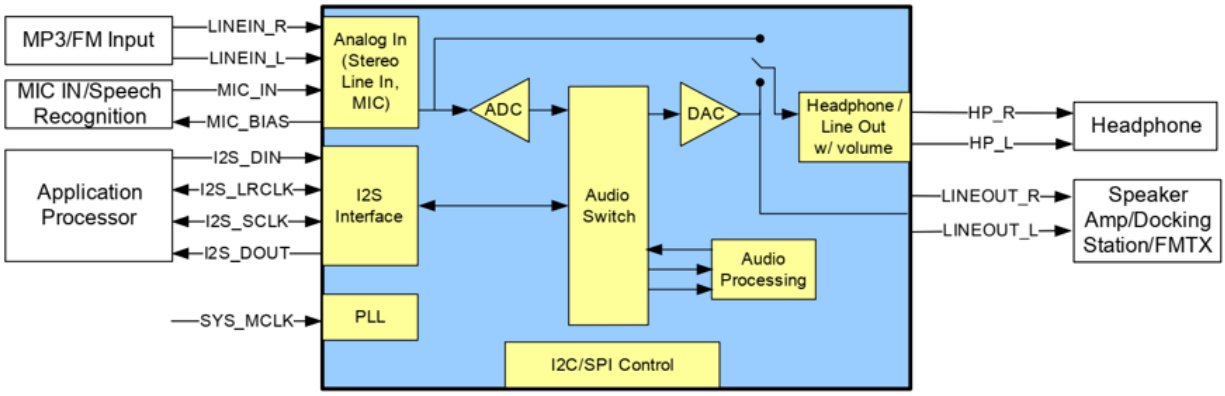


Figure 1: SGTL5000 Block Diagram

To Elaborate my usage, I created a 32-bit I2S interface and loaded in the Audio Switch to some of the Arduino pins. I used a C program and a platform designer I2C to run the I2C interfacing. With the given configurations, audio is output into the Headphone Jack.What I changed was that the input pin to the Audio switch was instead the I2S_DOUT which is discussed in further detail in the “Quartus Files” section.

Tesseract-OCR

Tesseract-OCR is an open source OCR (Optical Character Recognition) tool originally developed by Hewlett-Packard and now by Google that is able to “read” text using computer vision and trained AI. It is the same OCR used in Google Translate’s character recognition functions. Essentially, it works by comparing given letters in pictures or PDFs to characters it has been trained to “read” and compares them for the closest match in its dataset. For example in the picture below, the program may have a very difficult time matching the “a” and “r” to their proper letters, but it will alway correctly match the “e” at the end, so it may need additional training.

.

My usage for Tesseract-OCR to read text from the PadTest Program discussed in the next section. Personally, I had issues with reading the “+” and “ Ø ” so I tried to train the AI, but ultimately resorted to a much simpler Regular Expression solution, as discussed in the “Story and Inspiration” section and “Python Files Overview”

Motion Source and PadTest

Disambiguation:

Given the legally dubious nature of emulation, documentation on the Motion Source and PadTest programs are extremely limited. All that is known about them is that Motion Source, written in the kotlin language, takes in accelerometer data from multiple sensors on the phone and continuously uploads the data onto a LAN server while spoofing its device specifications to be a Dualshock 4 controller. PadTest is a program that can read data from the LAN servers and take in the information provided and assumes the given controller is a Dualshock 4, and appears to be primarily written in C based on the structure of the .dll files included.

The links to the tutorials with them are provided in the link below:
[UDP Pad motion data provider setup (sshnuke.net)]

Note Mapping

As mentioned in the “Story and Inspiration” section, I chose to only implement the key of C due to time constraints and ease of use. I only have 8 vector directions for motion input plus an octave, plus I needed some way to figure out how to pause between notes, so I designed the Puppeteer Theremin as follows. I chose to have the octaves 4 and 5 to have different mappings to make the middle notes easier to play while the lower and higher notes are rarely played.

For the Octave selection I did as follows

Fig 2: Octave Selection

For the fourth octave (G3 top left, G4 Top Center)

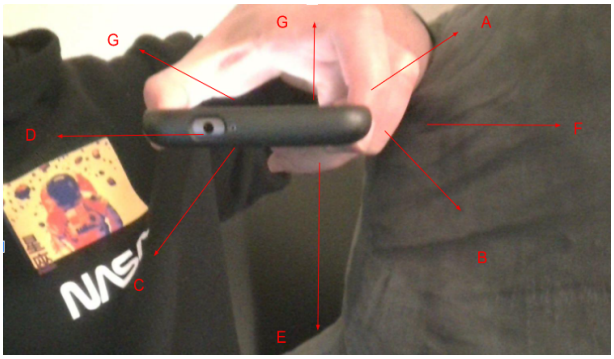


Fig 3: Octave 4 Note Selection

For the fifth octave (A5 top left, A4 centered)

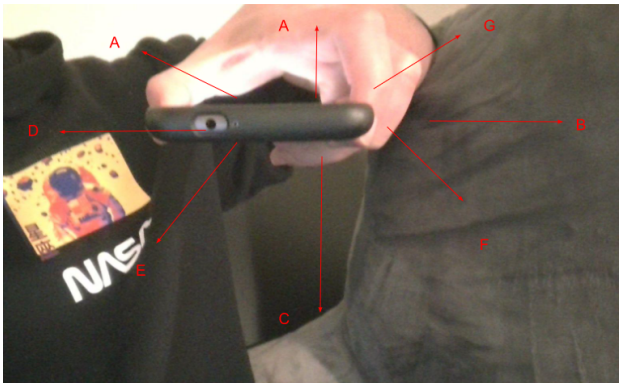


Figure 4: Octave 5 note selection

Python files overview

windowactivation.py

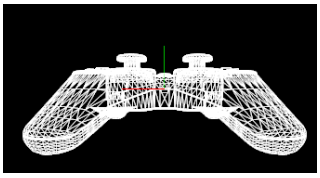
```python
import os
import subprocess
import time
import win32gui
def activate_windows(): #activate the necessary windows to open
    current_dir =r'C:\Users\matte\Downloads\PadTest_1011' #path to padtest
    subprocess.Popen(os.path.join(current_dir,"PadTest.exe")) #open padtest from path
    time.sleep(1)
    handle0=win32gui.FindWindow(0, 'DSU Controller Test') #find the opened padtest program
    win32gui.SetWindowText(handle0, 'lol') #rename it lol
    time.sleep(1)
    current_dir =r'C:\Users\matte\Downloads\PadTest_1011' #path to padtest
    subprocess.Popen(os.path.join(current_dir,"PadTest.exe")) #open the second padtest program
activate_windows() #activate the windows, I wanted to move to top level of python but got lazy
```

Description: First, import libraries, then set path to the padtest program, open the PadPest program from path, wait, rename the window, wait, open a new PadTest instance. End
Purpose: Open windows with the necessary names to run properly.

ece385finaldemo.py

```python
import time
import mss
import numpy
import pytesseract
import re
import win32gui, win32con
import os
import jtag_uart
```

```python
if (__name__ == "__main__"): #runnit
    main() #runnitallllllll
    print("success") #artifact of debugging
```



```python
handle1=win32gui.FindWindow(0, 'DSU Controller Test')#set handle for first instance
handle2=win32gui.FindWindow(0, 'lol') #set handle for second instance

pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-OCR\tesseract.exe' #path for tesseract
custom="c- tessedit_char_whitelist= 0123456789-+" #custom configs
mon = {'top': 313, 'left': 58, 'width': 192, 'height': 18} #location on screen of accelerometer data

time.sleep(5)#give 5 seconds after running to initialize everything, this is an artifact from testing
```

Description: Contains functions cleanchars, direction, bitreturn, takenums, minimize, fullscreenselect, write, and main. Uses libraries mss, pytesseract, jtag_uart, numpy, regex, os, win32gui, win32con, and time.
Purpose: Send keycodes via JTAG from motion controls using functions shown below

Function: cleanchars

```
def cleanchars(text):
    print(text)#print the badly read text, next do filtering
    ext=text.replace('8.', '0.')
    ext=ext.replace('@','0')
    ext=ext.replace('2.','0.')
    ext=ext.replace("20.","0.")
    ext=ext.replace('40', '+0')
    ext=ext.replace('41.','+1.')
    ext=ext.replace('%','00')
    ext=ext.replace('-.','-0.')
    ext=ext.replace('+.','+0.')
    ext=ext.replace('4.','+0.')
    ext=ext.replace('. ', '.')
    return ext#return the filtered text
```

Description: Use the replace attribute of strings to filter data that is read wrong (8 for 0, 4 for plus etc.)
Purpose: Make sure the program is receiving the correct data from Tesseract

Function: takenums

```
def takenums(text):
    p=re.compile(r"[+,\-]\d\.\d\d\d")#take stuff in signed decimal format
    v=p.findall(text)
    return v
```

Description: Take in text, return only signed numbers to 3 sig figs
Purpose: Only return numbers from the given text

Function: bitreturn

```
def bitreturn(num): #return a bit code for the inputs
    if (num==0):
        return str("00")
    elif (num==1):
        return str("01")
    else:
        return str("10")
```

Description: Returns a 2 bit number to encode given an input
Purpose: To keep track of states, I forgot why I used strings alongside binary.

Function: Write

```
def write(ju, writearray): #write a bit array through jtag to the fpga
    writedata = bytes() #instantiate the state data
    index = 0 #instantiate the index
    while (index < len(writearray)):
        writedata = bytes() #instantiate the byte data
        for i in range(0, 16 * 2**10): #big number of bits for if you are writing literally everything to memory on FPGA
            writedata += bytes([writearray[index]])  #write the bits in the index
            index += 1 #index up
            if (index == len(writearray)):#end if the index is done
                break
        ju.write(writedata)#write the data through jtag uart
```

Description: Given a string, send each ascii code for the string in decimal format through jtag
Purpose: Send any and all information to the FPGA

Function: keyreturn

```
def keyreturn(text): #takes in a bit combination, returns an ascii from a keycode
    keydic={"0000":"s","0100":"d","1000":"a","0001":"w","0101":"e","1001":"q","0010":"x","0110":"c","1010":"z"}
    print(keydic[text]) #for debugging
    return keydic[text] #returns keycode
```

Description: Given 4 bit state, return a keycode
Purpose: Return keycodes so it is in a format that can go to the FPGA predictably and easily

Function: direction

```
def direction(lis):#takes in numbers from accelerometer from gravity, finds direction to move
    direction=[]
    print(lis)
    try: #to test for index errors
        lisnew=[float(lis[0]), float(lis[1])]
        if (lisnew[0]<-0.4):
            direction.append(2)
        elif (lisnew[0]>0.4):
            direction.append(1)
        else:
            direction.append(0)
        if (lisnew[1]<-0.4):
                direction.append(2)
        elif (lisnew[1]>0.4):
                direction.append(1)
        else:
                direction.append(0)
        x=str(bitreturn(direction[0])+bitreturn(direction[1]))#list of bits combined into a string
        return keyreturn(x)
    except:
        return("bad read")
```

Description: First test if you have two directions read via the other parts, if not, return bad read
Purpose: To make sure that there are no misreads sent over JTAG to make the sound sound terrible and make operation completely unpredictable.

Functions: minimize and fullscreen select

```
def minimize():
        Minimize = win32gui.GetForegroundWindow()#grabs window in foreground
        win32gui.ShowWindow(Minimize, win32con.SW_MINIMIZE)#minimizes it

def fullscreenselect(handle): #select a window, handle being the encoded name for a window
    minimize() #runs minimize function
    try:
        win32gui.ShowWindow(handle, win32con.SW_MAXIMIZE) #mazimize needed window
    except:
        1
    try:
        win32gui.SetForegroundWindow(handle) #try to set it as top level
    except:
        try:
            win32gui.BringWindowToTop(handle) #if it fails, try it again but in two parts
            win32gui.SetForegroundWindow(handle)
        except:
            print(handle) #if it fails again, print it
    return handle #return the handle you just used
```

Description: first minimize the foreground window to open up the thread, try to open the other tab, if it fails, try again, if it fails, try again but in two steps, if it fails yet another time bring it to top. Windows Kernel sometimes needs a few tries to get it right.
Purpose: To make sure the correct window is always on top. The Windows kernel needs a few tries to get it right.

Function: main

```
def main():
    ju = jtag_uart.intel_jtag_uart(instance_nr=0) #grab first jtag connection instance
    with mss.mss() as sct: #defininging taking a screenshot as sct
        while True: #infinite loop
            fullscreenselect(handle1)#select window one then sleep for timing reasons
            time.sleep(0.1)#could be shorter, I put this time so that I could give the threads and ram on my computer a break
            im = numpy.asarray(sct.grab(mon))#take first screenshot

            text = pytesseract.image_to_string(im, lang="eng", config=custom)#convert image to text
            fullscreenselect(handle2)# do it again
            time.sleep(0.1) #could be shorter, I put this time so that I could give the threads and ram on my computer a break
            im1 = numpy.asarray(sct.grab(mon)) #take second screenshot
            text1 = pytesseract.image_to_string(im1, lang="eng", config=custom)#convert image to text

            a=[ord(c) for c in ((direction(takenums(cleanchars(text))))+(direction(takenums(cleanchars(text1)))))]
            #okay this has a lot going on, first, filter characters for both and take their
            # numbers for X and Y directions turn the directions into a character put the keycodes
            # into a tuple, for each character in the tuple, put the ascii code into a string called a

            if (a != [98, 97, 100, 32, 114, 101, 97, 100, 98, 97, 100, 32, 114, 101, 97, 100]): #if not a bad read
                write(ju, a) #send it to the fpga
                print(text)#artifact of debugging
                print(text1)#artifact of debugging

            else:
                print(a) #artifact of debugging
                print("bad read") #tell me what is wrong

            for i in a:
                    print(f"{i:02x} ", end="") #print the ascii codes
            print("")#newline for aesthetic
```

Description: Top level of the program to run, full algorithm better shown in the Block Diagrams and Flow-Chart section in figure X.
Purpose: Sends the phone accelerometer data to FPGA.

pramodhelpertools.py

```
import librosa
import numpy
import os

x, Fs = librosa.load(r"C:\Users\matte\Documents\Audacity\440a.wav", sr = 44100)#load .wav file
with open("440a.hex", "w") as f: #as the file to be written
    sine_int = numpy.zeros(101).astype(int) #blank array
    for i in range(101): #
        sine_int[i] += int(x[i] * 128) #adjust size
        if sine_int[i] < 0: #if not 0
            sine_int[i] = ~sine_int[i] + 1 #index up
        f.write("{:02x}\n".format(sine_int[i])) #hexcode return
#it is literally just this repeated 18 or so times
```

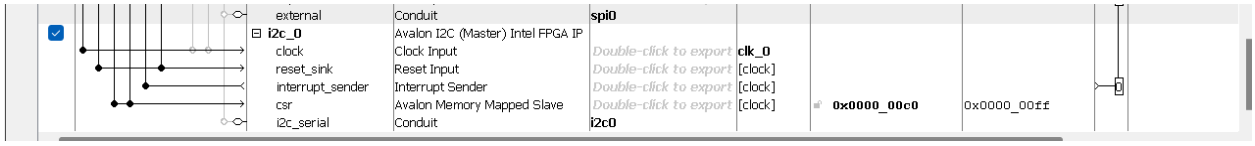
Description: Takes in .wav file and makes it into a hex via iterating over a loop
Purpose: creating wavetables to be loaded onto memory

New Quartus Files

Platform Designer

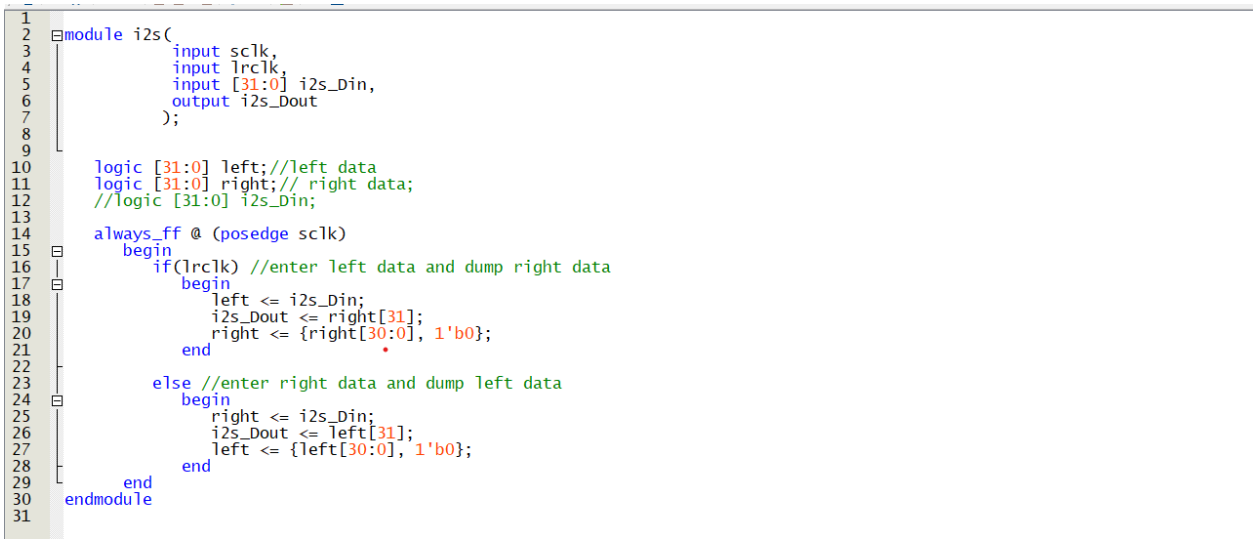
Description: The platform designer is completely unmodified other than instantiating an I2C module.
Purpose: Make it so I do not have to write all the code for memory instantiation as well as the nios ii protocols.

I2C_0 Block



Inputs: 50 MHz clock and system reset
Outputs: NIOS ii interrupts and csr
Exports: i2c_serial =i2c0
Description: Instantiate an I2C module
Purpose: Interfaces with the SGTL5000 chip and the rest of the FPGA in a way that is functional.

i2s.sv



```
module i2s(
           input sclk,
           input lrclk,
           input [31:0] i2s_Din,
           output i2s_Dout
          );

    logic [31:0] left;//left data
    logic [31:0] right;// right data;
    //logic [31:0] i2s_Din;

    always_ff @ (posedge sclk)
       begin
          if(lrclk) //enter left data and dump right data
             begin
                left <= i2s_Din;
                i2s_Dout <= right[31];
                right <= {right[30:0], 1'b0};
             end

          else //enter right data and dump left data
             begin
                right <= i2s_Din;
                i2s_Dout <= left[31];
                left <= {left[30:0], 1'b0};
             end
       end
endmodule

```

Important ports: input sclk, input lrclk, input i2s_din, input, i2s_dout
Description: Shifts bits once every sclk cycle it just does this two times, once for right, once for left, has a small defect/feature where the first cycle into the left channel outputs all zeros regardless of i2s_din and i2s_din is backwards. This has no impact due to the near inaudible frequency of these cycles as well as how they are loading in the same thing. All this produces is a very very slight shift in the left and right audios.
Purpose: Output audio data into the SGTL5000

RAM parser blocks (a440.sv etc)

Important ports: input Clk, inout wire [7:0] data_Out
Description: Register for parsing RAM, sets data output from the registers
Purpose: Storing audio hex files for usage

Notelight

Important Ports: input Clk,[7:0] keycod, [7:0]keycod1, output [7:0] noteled
Description: An FSM that outputs an 8 bit integer given two keycodes
Purpose: FSM module for sending the correct note to the Hexdrivers

choosesound1.sv

Important ports: Input Clk, sixteen Input [31:0] data line inputs from ram blocks, Output [31:0]daterout
Description: An FSM that outputs an 32 bit integer given two keycodes
Purpose: FSM module for sending the correct note to i2s

Shiftreg.sv (written by UA Ian)

Important Ports: input [8:0] Din, output [24:0] dout
Description and purpose: Simple shift register that shifts in 8 bit on clock

pulse.sv (written by UA Ian)

Important Ports: input clk, in, output out
Description and purpose: Determines whether or not the JTAG connection is making the shift register take in data or export it (if exporting then if does not shift)

hex_driver.sv (written by UA Ian)

Important Ports: input [3:0] in , input dp, [3:0], output [7:0] out
Description and Purpose: Loads data into the hex displays by a simple case block

Top Level: motionsynth.sv

Ports are the same as Lab 6.2 top level
Description: Top level, instantiates and has many shift registers for all the ram data instantiations as well as shifting clock data
Purpose: Makes this entire project possible

All other programs are given via ECE 385 from Lab 6.2 releases. I chose not to include my snake.sv because due to its bugs, it adds nothing to the project.

Nios ii Eclipse Software:

Disambiguation:

All C code (SGTL5000_test.C) was provided by the course and wholly unmodified. The documentation on the software and use is provided via the lecture 24 slides and recording as well as the information on the final project page. The only difference between intended operation and my usage, was that I ended connection with eclipse once the .elf file was flashed to the memory of the FPGA. I do not hold ownership over it, nor have I modified it.

Block Diagrams and Flow-Charts

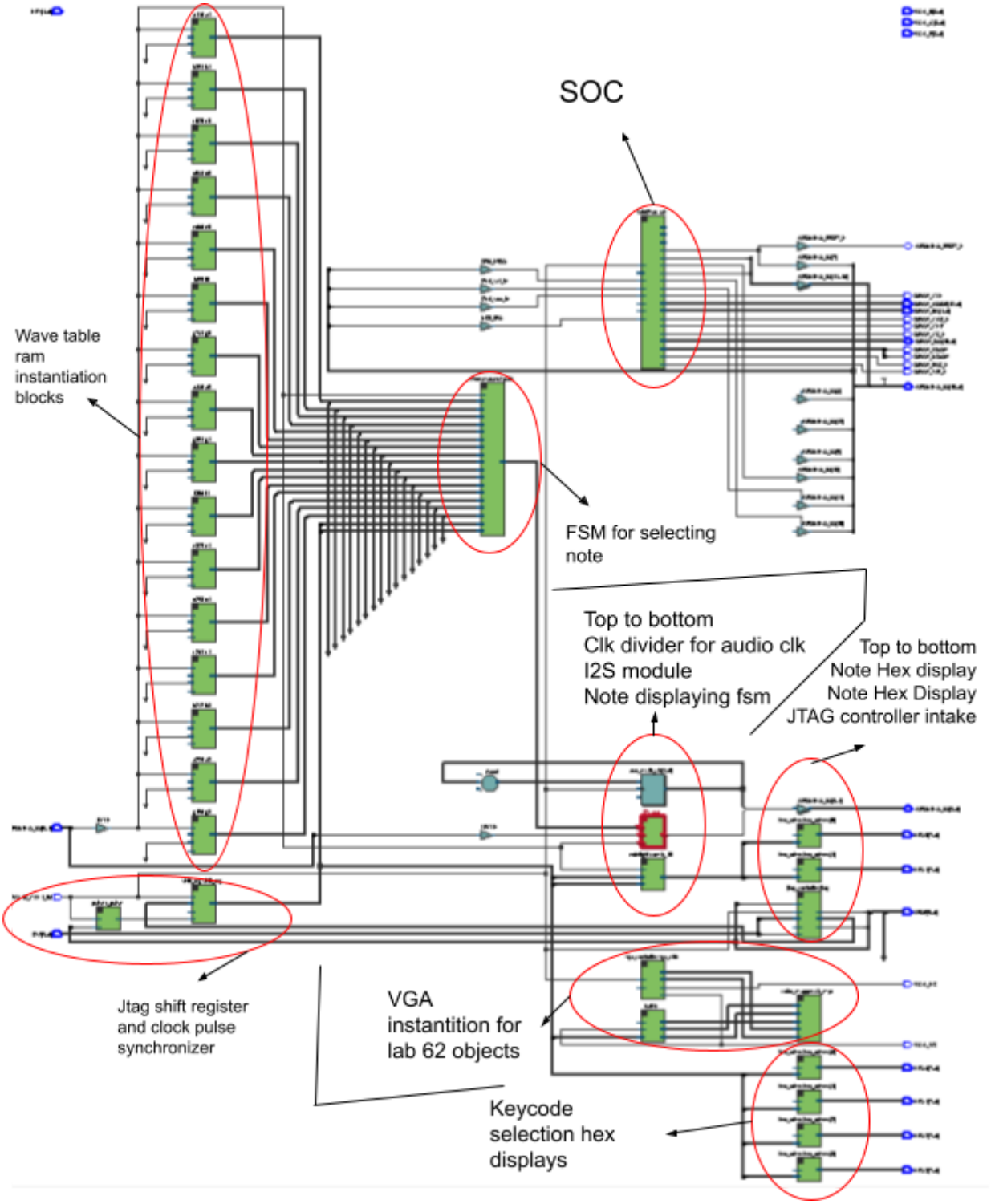


Figure X: Labeled RTL Diagram

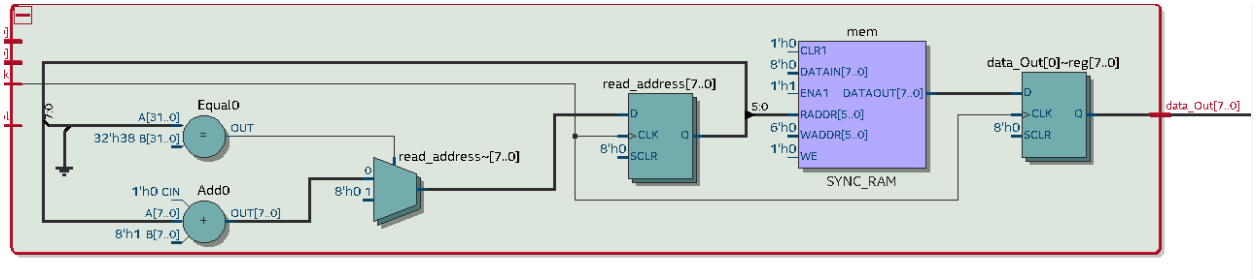


Figure X: Wave table memory reading block

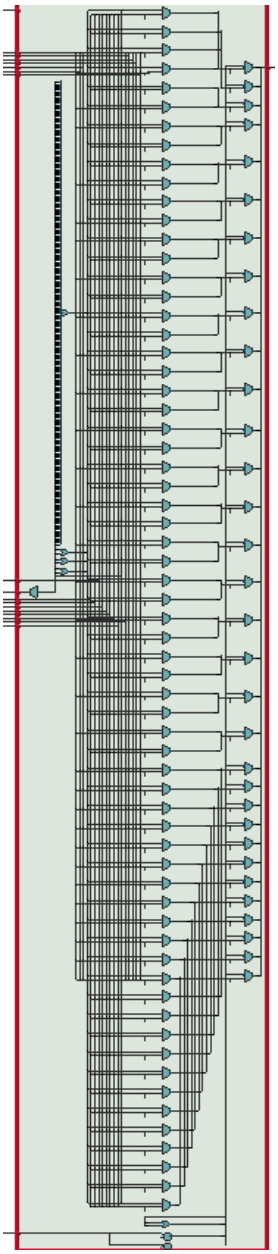


Figure X: choosenote1 block diagram

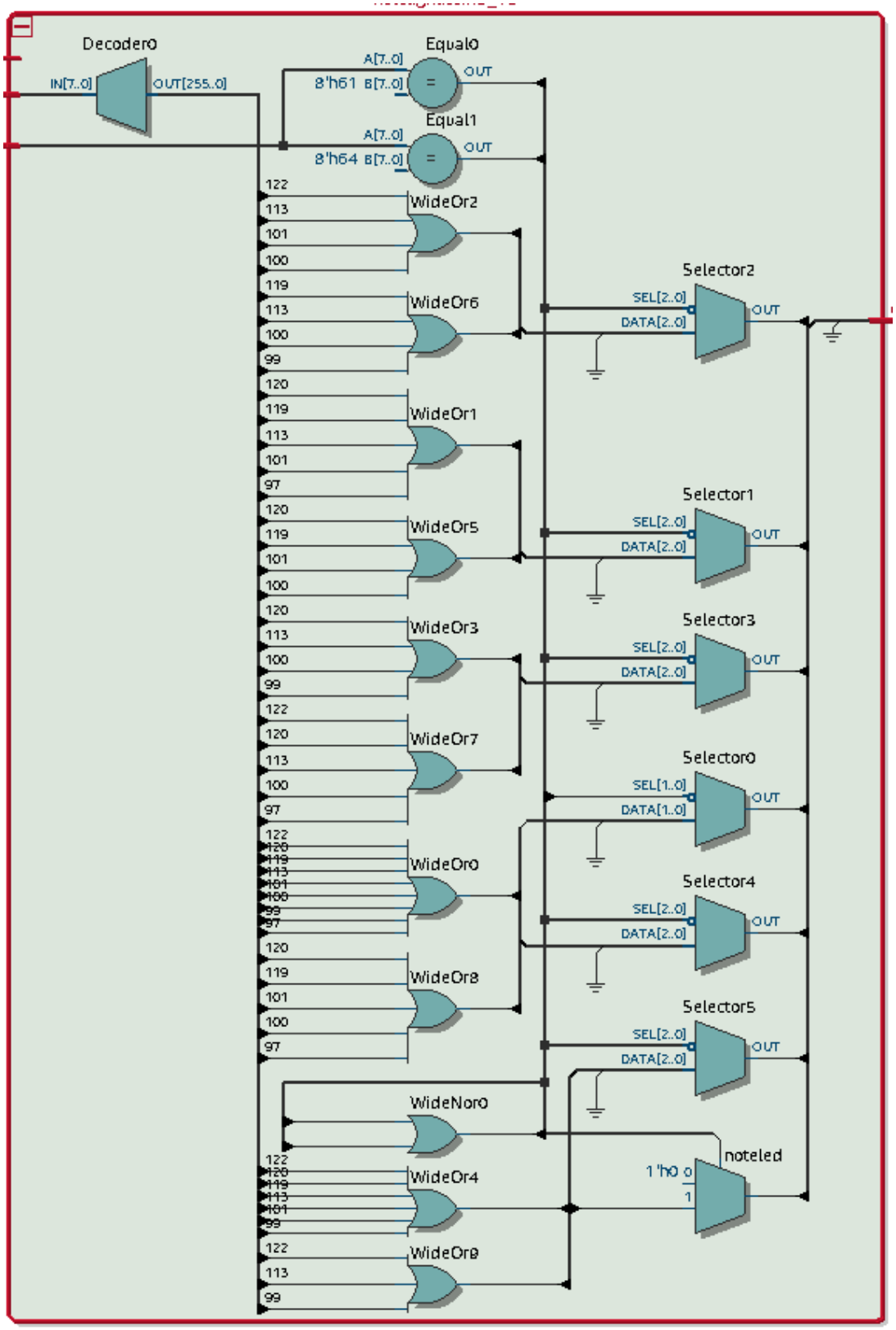


Figure X: Note Light Block Diagram

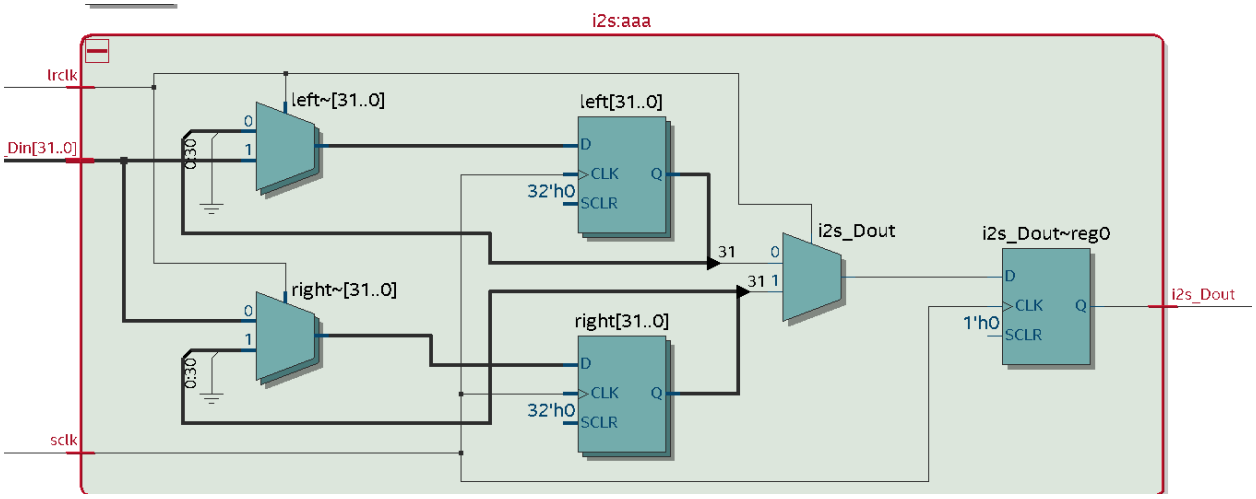


Figure X: I2S Block Diagram

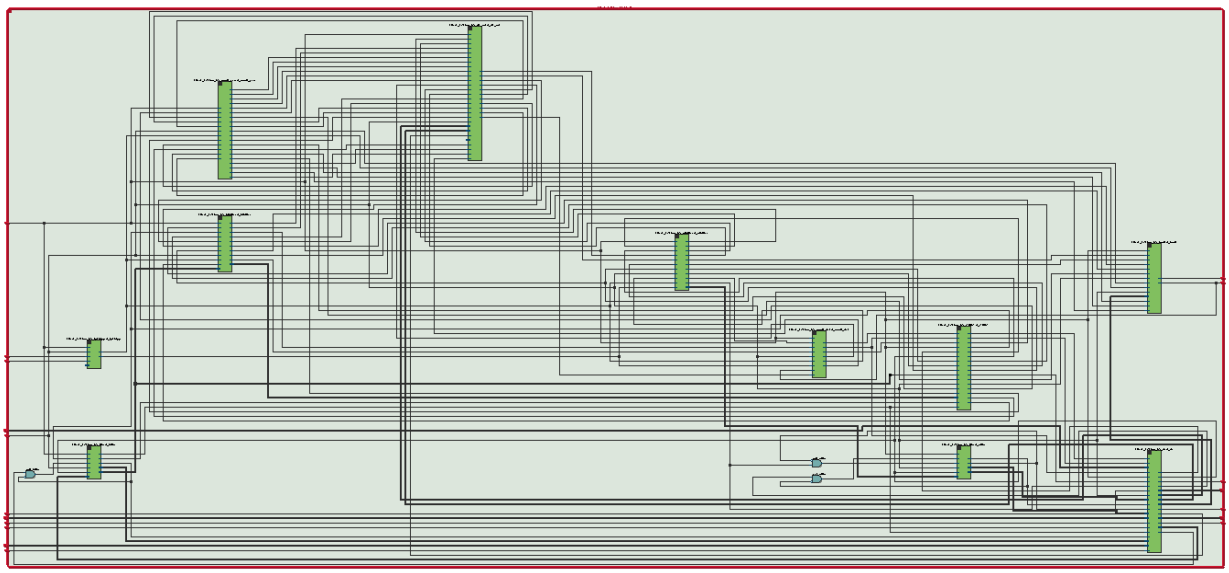


Figure X: I2C Block Diagram

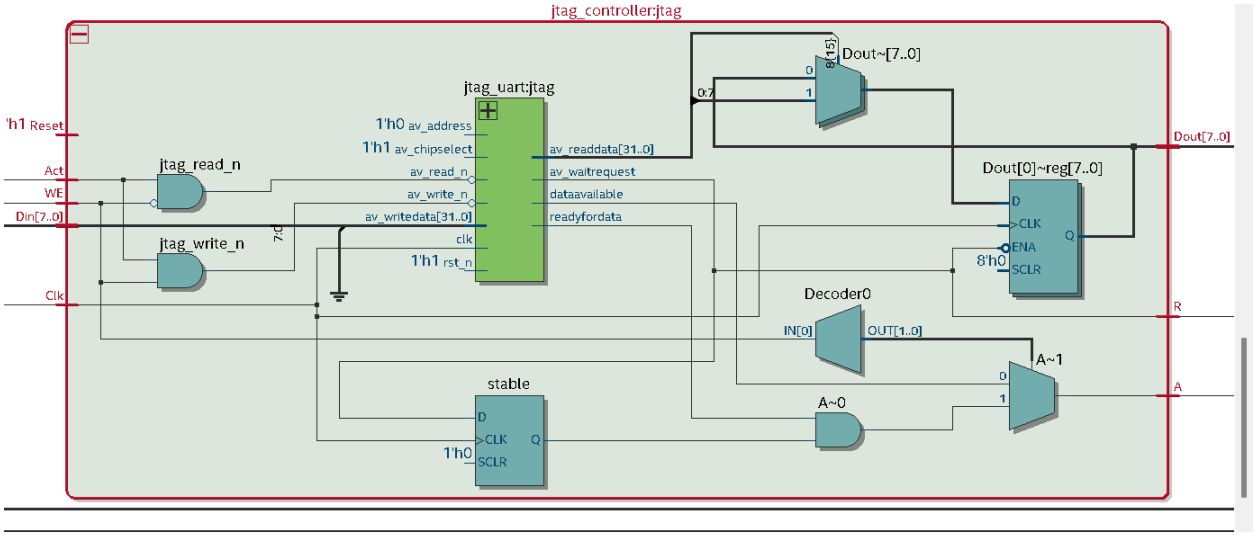


Figure X: JTAG controller block diagram

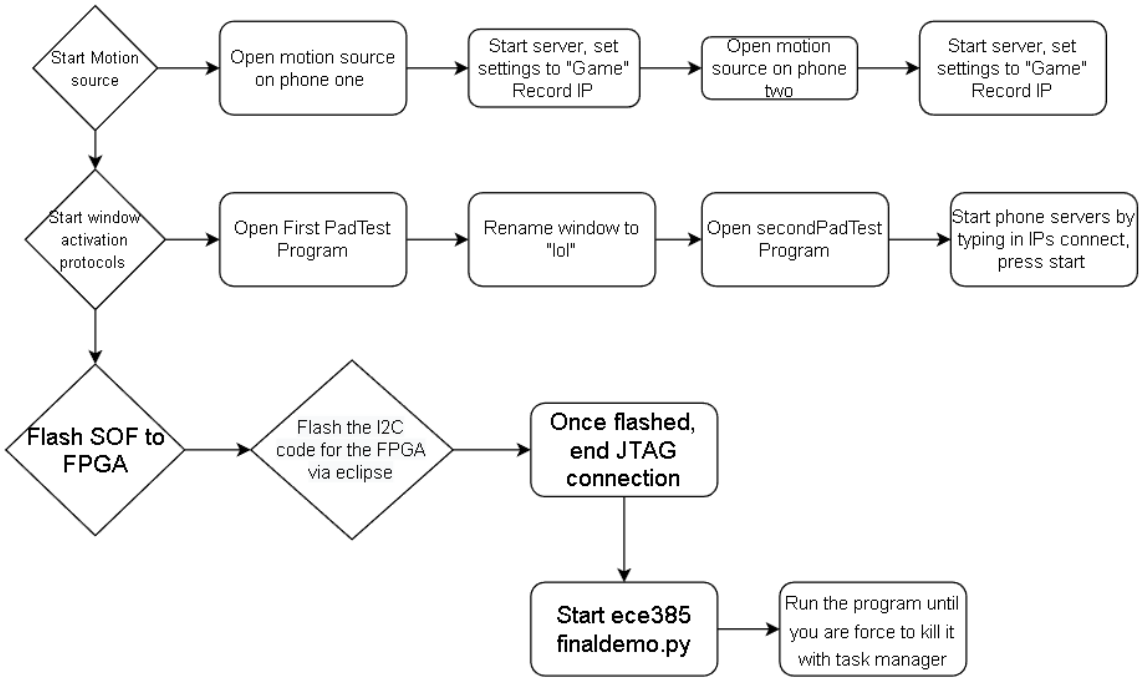


Figure X: FlowChart Block Diagram for Operation

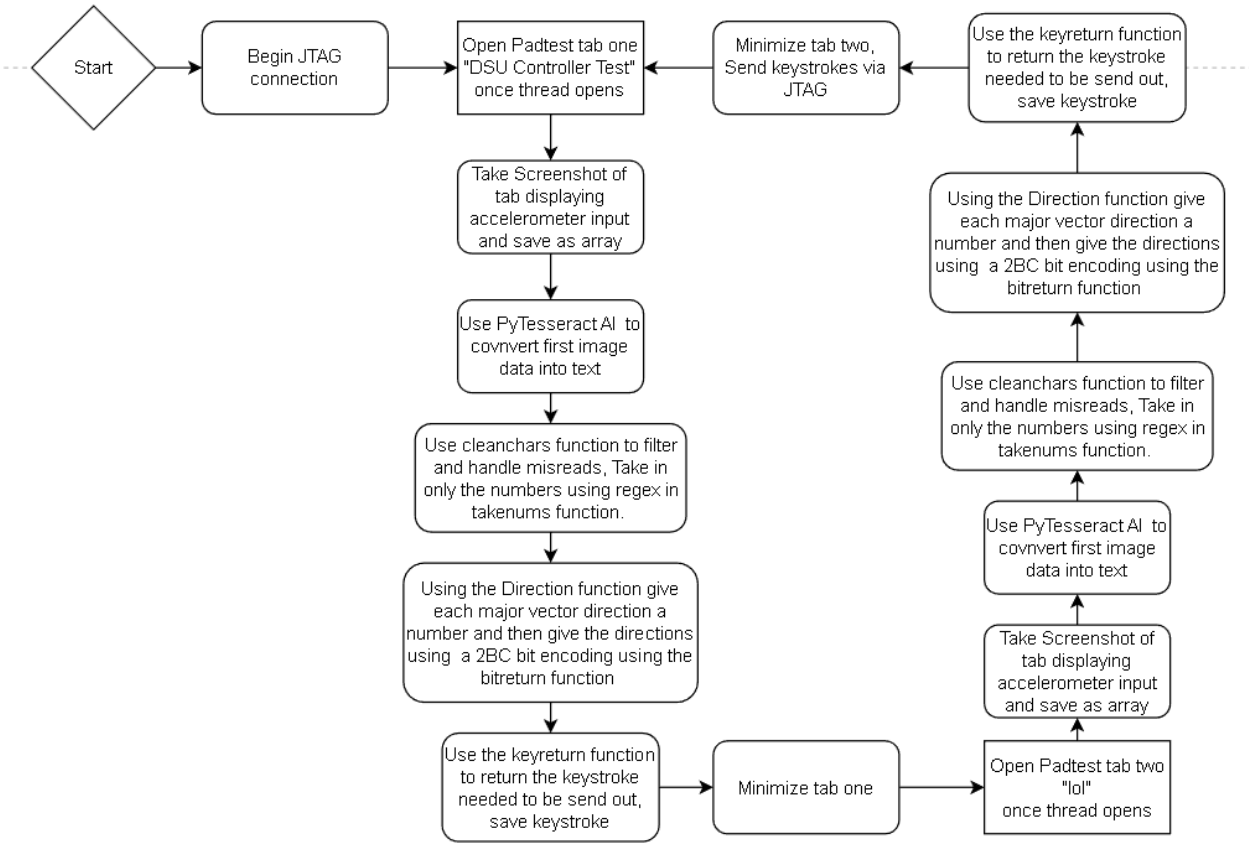


Figure x: Flowchart for main() in ece385

Design Resources and Statistics

Resource	Utilization
LUT	5806
DSP	10
Memory (BRAM)	193,536
Flip-Flop	3402
Frequency	76 MHz
Static Power	96 mW
Dynamic Power	66 mW
Total Power	242 mW

Figure X: Final Project Statistics

Conclusion

With all of these parts put together, I was able to interconnect all of them in order to create a motion controlled synthesizer. The most difficult part honestly was not being given a very good explanation of why the audio data had to be in a given format, because the professor more or less ran out of time and rushed the explanation of loading RAM data into the I2S as well as rushing the explanation. This made working with audio much harder than video game graphics because of the more or less complete lack of explanation of the work that the student had to do.

Credit to the much code in this final project goes to ECE department for the lab 62 code as well as the SGTL5000 code, the anonymous creators of Motion Source and PadTest, some posts I found on stackoverflow for inspiration, the entire team responsible for Tesseract-OCR, and the other python libraries I used, Rishi for Rishi's helper tools. Pramod and Ian the CAs who created small helper tools for wave table synthesis, and python JTAG connection respectively. Special thanks for help with debugging goes out to the TA's Phil and Zayd, could not have done it without them. All the love and emotional support goes out to Pramod.