California State University, Fresno
Lyles College of Engineering
Electrical and Computer Engineering Department

TECHNICAL REPORT

Assignment: Final Report

Course Title: ECE 178 (Embedded Systems)

Semester: Spring, 2021

Date Submitted: May 13th, 2021

Student ID #:

Prepared By: Jacob Sales

INSTRUCTOR SECTION

Comments:_____
_____
_____
_____
_____
_____
_____
_____

Final Grade: Jacob Sales: _____

TABLE OF CONTENTS

Section **Page**

1. OBJECTIVE STATEMENT

The purpose of this project is to create a keyboard synthesizer that is fully functional with the FPGA board. The keyboard here will be defined as an instrument that will generate sound for the purpose of creating music. The keyboard should be able to work at multiple octaves as a normal keyboard should. This means that we should have about 84 playable notes at varying frequencies. This would not be as playable as a normal keyboard, but it should be functional enough that if keyboard hardware were connected, then the FPGA could act as a synthesizer.

The project should use Nios II BSP to implement programming logic that is required for the synthesiser. We will be using a soft-core embedded system as our base design for our system. We want to make sure that the implementation of this project is within the scope of knowledge of the experimenters. This solution should also include a hardware interface for keyboard inputs. For example, the interface should allow the user to play 12 different keys. Each key will be based on a different tone or frequency. We would also like to be able to play at least 3 different notes at the same time.

2. HARDWARE REQUIREMENTS

- Personal Computer than supports Intel FPGA MonitorProgram 16.1 or greater
- DE2-115 FPGA Education Development Board
- Bread board
- Multi Colored Jumper Wires
- 1 kΩ Resistors x 10
- 680 Ω Resistors x 10
- 1.5 kΩ Resistors x 4
- 8 red LEDs
- 12 QTEATAK Tactile Switches, 2 pin, 6 x 6mm, SPST
- 6 x 0.01 uF MYLAR Capacitor, 100V, 10%
- 5 x 0.1 uF MYLAR Capacitor, 100V, 10%
- 2 x 1 uF MYLAR Capacitor, 100V, 10%

3. SOFTWARE REQUIREMENTS

- Quartus Prime 16.1 or greater
- Nios II 16.1 SBT for Eclipse
- Notepad

4. BACKGROUND

The work covered in the project covers a broad range of knowledge and techniques used to develop the FPGA and associated user interface. We assume that the experimenter has basic knowledge of the C programming language. We also assume that they understand basic circuit design or at least up to the point of RC circuit design. This project is an FPGA design project based in Nios II, and we will assume that the experimenter has basic design knowledge of Nios II and DE2-115 FPGA Education Development Board.

Before any experimenter begins to analyze or replicate this project, it is essential that they know the following concepts. The first concept that is needed to understand this project is pulse-code modulation. This will be used to create data for a DAC or Audio Codec. The next needed component would be the knowledge of the WM8731 Audio codec integrated in the DE2-115 board. This knowledge will be necessary to make any comprehensive sound on the board. They should also understand the setup and use of PIOs. This understanding will also help with the understanding of the operation of the GPIO/expansion header on the DE2-115 board. Apart from this, they should understand the basic setup and operation of interrupts in Nios II. Finally they should have a basic understanding of debouncing and ways to implement it for button presses.

4.1. Pulse-code Modulation(PCM)

Pulse-code Modulation is the method of converting an analog signal to a digital signal[1]. An analog signal will be defined as a continuous time signal and a digital will be considered as a discrete time signal. Here we use the idea that an analog signal can be represented by a sampled digital version of the signal. How well this signal can be represented is based on the rate at which this signal is sampled. This is known as the sampling rate or sample rate[1]. It is a common term used in audio, such as the sample rate of digital audio being 44,100 Hz or 48 kHz. This is often determined by the Nyquist frequency which is needed to retrieve all Fourier components of a waveform[2]. Since the range of musical frequencies reaches about 20 kHz, 44.1 kHz or 48 kHz are often used as sampling frequencies.

Samples taken of an analog signal at the specific sample rate can be set on a graph based on time and the amplitudes of the sampled signal as seen in Figure 4.1.1. A sample rate based on the Nyquist sample frequency will reveal a very similar signal to the original analog signal. The higher the sample rate goes, the more and more similar the signal becomes. Yet in terms of this project, the sampling rate can be set to one of 2 values; 44.1 kHz or 48 kHz. This is because the audio codec used is based on reading data in PCM or MP3 format. This is important to understand because the project needs to be able to generate a signal that represents a sound wave. To do this, the program must generate PCM data that will be used by the audio code to control voltage going to headphones or speakers. The only way of doing this is by understanding the

rate at which the DAC will convert the digital signal to analogue. This way we can recreate any signal at any given frequency.

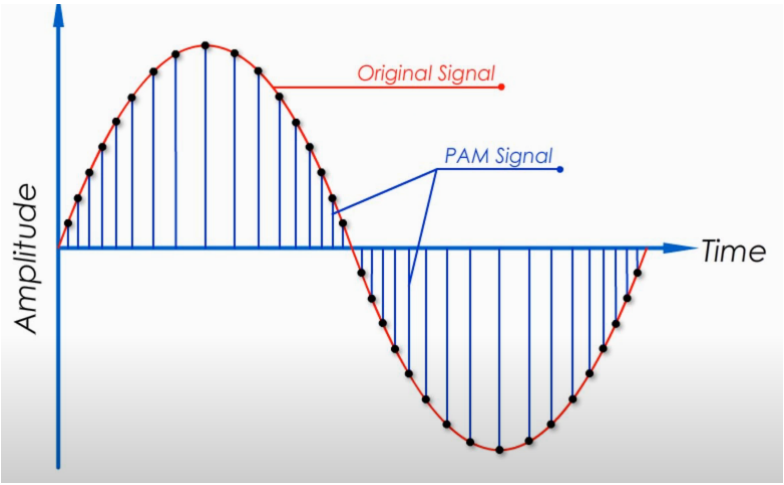

Figure 4.1.1: Image showing a sampled signal over the original analog signal[1]

Another important concept is bit depth. Normally bit depth is the number of bits that can be used to represent amplitude in each sample[1]. For an ADC and DAC, this normally means that data is unsigned. This works by defining the lowest amplitude point as 0 and the highest amplitude point as the maximum unsigned binary value for the bit depth. For this project, bit depth plays a slightly different role since data produced by the audio codec is represented as 2's complement, signed data. Here the lowest point will be based on the 2's complement signed data lowest binary value based on the bit depth. The highest will also be represented on the signed data's highest binary value based on the bit depth. By understanding bit depth and sample rate, we can create PCM data that can be inserted into the audio codec.

4.2. WM8731 Audio Codec

The WM8731 Audio Codec is described as a Portable Internet Audio CODEC with Headphone Driver and Programmable Sample Rates[3]. This audio codec is used to drive the line out for headphones on the DE2-115 board. There is much to be said about the audio codec which can be found in [3], but here we will focus on the sample rates and bit depth of the codec. The audio codec is said to use 24 bit ADCs and DACs. This should mean that the bit depth is also 24 bits. However, audio data length can be configured to be 16 bits, 20 bits, 24 bits, or 32 bits. Data length and bit depth should be distinguished from each other. Data length should be thought of as the format of the data that will be extracted or input into the audio code. The bit depth is the number of bits used by the ADC/DAC to create the amplitudes of the signal. If the ADC is programmed to use 16 or 20 bits, then it will strip the LSBs of the 24 bits of data[3]. Yet, if the ADC is programmed to use 32 bit length, then it will stuff the LSBs with 0s[3]. The exact opposite happens for the DAC. When the DAC is programmed to use 16 or 20 bits, it will stuff

zeros in the LSBs. If it is programmed to use 32 bits, then it will strip away the LSBs before using the data.

This is extremely important to understand. Since the program will be generating the PCM data, not only will it have to create logical data based on sample rate and bit depth, but it must also format that data correctly using the proper data length and data format. Data can also be formatted using the Audio codec. For this project, we decided to use the left justified format for our data. This means that since we also chose to use 32 bit data length to represent our data, our MSB of the 24 bit data will be the MSB of the 32 bit data length. The rest of the unused LSBs will be stuffed with zeros.

4.3. PIOs and GPIO/expansion header

I/O devices come as interrupt capable and are Qsys ready. This means that a designer does not have to define them in Qsys beforehand. They have already been defined for them. To use the I/O devices, they each need to have a defined PIO. PIOs have up to 32 I/O lines and contain 4 different registers: Data Register, DDR, Interrupt Mask Register, and Edge-Capture Register. These registers and their positions are defined in Table 4.3.1 The data register will contain the data from the I/O device. The data register can be read or written depending if the peripheral is an input or output device. For example, we could write to the green LED data register to change the display, we would read from the keys to gain the current data of the peripheral. The DDR is the data direction register. This register will configure if certain pins are input or output. This specific register will define if the GPIO pins are either an input or output. An input for the GPIO is defined as a 0 and an output for the GPIO is defined as a 1. The interrupt mask register will determine if the input port can generate an interrupt or not. The edge capture register will define if the interrupt will be captured at the rising or falling edge of the clock.

Table 4.3.1: Organization of PIO registers and their respective addresses.

Register Name	Address
Data Register	Base
DDR	Base + 4
Interrupt Mask Register	Base + 8
Edge-Capture Register	Base + C

Though some peripherals will not use the data direction register, the GPIO will use all the PIO registers. The data register will behave differently based on the DDR register. For example, if the GPIO pin in the DDR register is set to 1, then it is set to an output. This means that the

corresponding pin in the data register will control if the GPIO will output a logic high or low signal at that pin. If the GPIO pin in the DDR register is set to 0, then the pin is set to an input. Then when the data register is read it will show if the pin is receiving a logic high or logic low signal. Simply put, the data direction register will determine if the data from the data register is readable or writable. The IRQ mask register will be used for their interrupts based on inputs to the GPIO pin. Since the GPIO is defined as a PIO in Qsys, it's option to generate an interrupt based on the falling and rising edge of the signal is the same as other peripherals. Edge Capture also remains the same for the GPIO.

4.4. Interrupts

HAL registers an interrupt by using the alt_interupt_register function. This function will complete the entire process that was described for interrupts in assembly. This function will have three inputs. The first input is the devices' IRQ number. It needs to know the IRQ number to check control register 3 or the ienable register. This will make sure that the IRQ is activated so the CPU will accept this specific interrupt. Then it will need a context to be passed into the ISR. This content can be thought of as the saved context that was saved when handling interrupts in assembly. This is also data that needs to be used to figure out which line causes an interrupt in a PIO. In the case of a PIO, the context that is passed is the edge capture register. The last input should be the ISR function. This will act as a pointer to that ISR function that will service the interrupt. The ISR created will only have a pointer as an input that will contain the context passed into the alt_irq_register. After the ISR services the function, it will always clear the edge capture register and read from the edge capture register. It reads from the edge capture register before exit to make sure it doesn't leave too quickly.

4.5. Debouncing Switches

Switch debouncing is important so that switches do not make any residual noise than happens when a switch is pressed. The reason why bouncing needs to be eliminated is because it could cause a system to register a single press as multiple presses in quick succession. There are 2 methods to debounce a switch: the hardware method and the software method. For this project, we will use and discuss the hardware method. To understand how to stop a circuit from bouncing, we must understand why the switch bounces in the first place. The switch bounce is a cause of the contacts of a switch repeatedly rebounding when force is applied. No matter how much force is applied or how small the switch is, an equal and opposite force will be applied when the contacts connect. This means that when the contacts connect, they will rebound again and again until the connection between the contacts remains stable.

The goal of a hardware debouncer is to filter any changes in the switch signal[5]. An RC circuit diagram of a hardware debouncer can be found in Figure 4.5.1. This uses a capacitor to stop any major changes in the switch signal. When the is not close, it acts as an open circuit. This means

that current will flow through the diode since it has nearly no resistance. In this state. The capacitor will gain charge until the voltage level ideally reaches Vcc. At that point the capacitor will act as an open circuit and Vb will remain at a constant voltage. When the switch closes it will act as a short circuit. Without the capacitor, this switch will bounce and cause a change. But with the capacitor, the switch will bounce and the capacitor will take time to charge. This will make sure that changes from the bouncing will not be quick and can be smoothed out because the capacitor takes time to charge. In the project implementation, a Schmitt trigger will not be used due to constraints in resources. In Figure 4.5.1, the schmitt trigger makes sure that the transition between logic high and low is more smooth. For the purpose of this project, the omission of the schmitt trigger will not affect the results significantly.

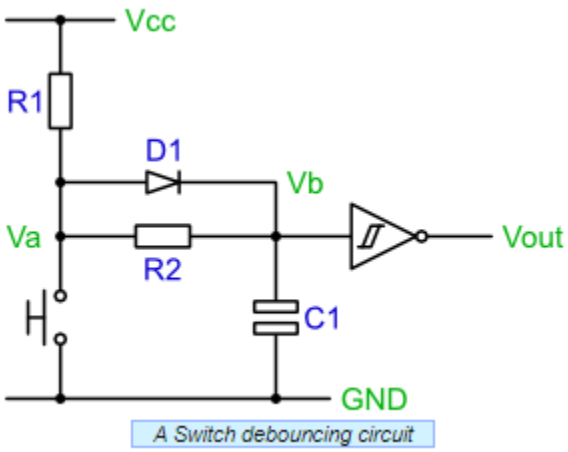

Figure 4.5.1: Circuit diagram of a switch debouncing circuit[5].

5. PROJECT OVERVIEW

1. This project starts in Qsys for a soft-core embedded system. Here we set up the Audio and Video Config Core, the Audio Core, Audio Clock for DE-series Boards, and GPIO expansion. We also made sure that all of our connections for our soft-core system were correct.
2. After generation of Verilog HDL the assignment will move into Quartus where pin assignments will be implemented and the module will be instantiated. Once this is done, the assignment will be compiled.
3. After this we needed to create a new Nios II SBT project and connect the to the DE2-115 board
4. We then began to work on the full software implementation of our synthesizer. We wrote code based on aspects mentioned in various manuals for the audio core. We also developed our own equation to create a saw wave that will be discussed in Section 8: Analysis of Results.
5. We then developed our external circuit to connect to the GPIO/extension header for the DE2-115. These were arranged to mimic keys on a synth keyboard.

6. PROJECT PROCEDURE

6.1. Setting up Audio Cores in Qsys and checking soft-core system

To access Qsys we clicked the Tools tab at the top of the Quartus project that contained the softcore System. We then selected Qsys. We first added Audio and Video Config Core to our softcore system. To do this we searched Audio in the search bar of the IP Catalog and selected Audio and Video Config from the University Program section. When presented with options for the core, we selected On-Board Peripherals, DE2-115, and Auto Initialize Devices. We set the rest of the audio settings as shown in Figure 6.1.1. We chose a bit length of 32 bits since we wanted our software to be able to send integer values into the Audio Core. 48 kHz was chosen to accommodate audio data. We then connected the core according to Figure 6.1.4.

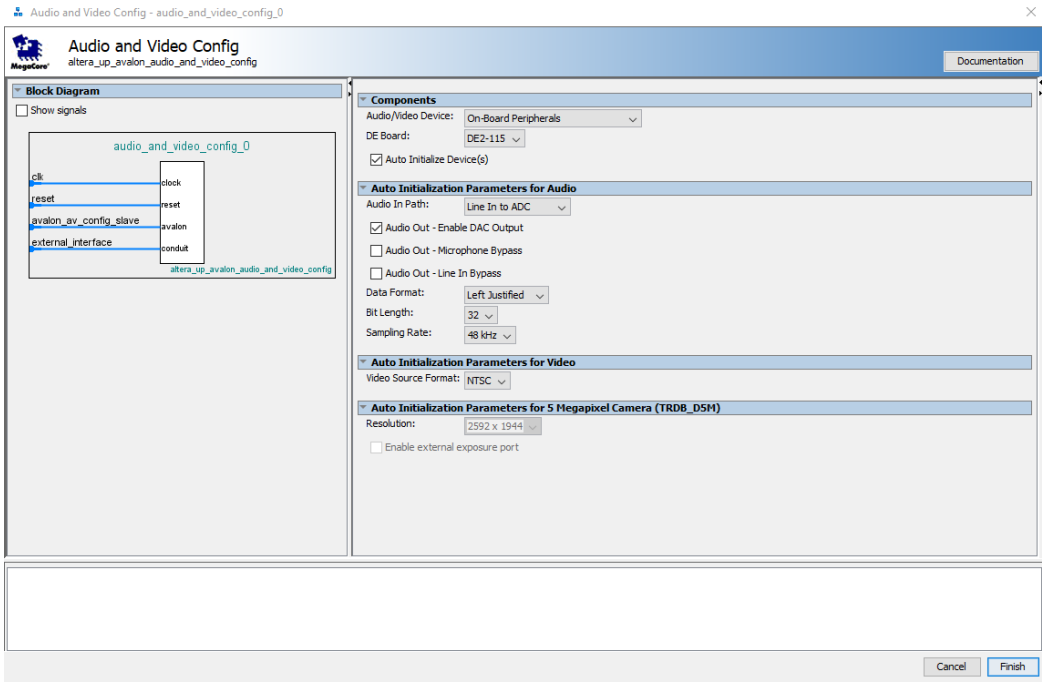

Figure 6.1.1: Image showing the settings selected for the Audio Video Config.

We then added the Audio Core by searching Audio in the search bar of the IP Catalog and selected Audio from the University Program section. For the settings, we selected the base setting given to us in Figure 6.1.5. Then we added the Audio Clock for DE-series Boards by searching Audio in the search bar of the IP Catalog and selected Audio Clock for DE-series Boards from the University Program section. For the settings, we selected the base setting given to us in Figure 6.1.6. These settings were chosen based on the datasheet for the WM8731 audio codec. We then connected the cores as shown in Figure 6.1.4.

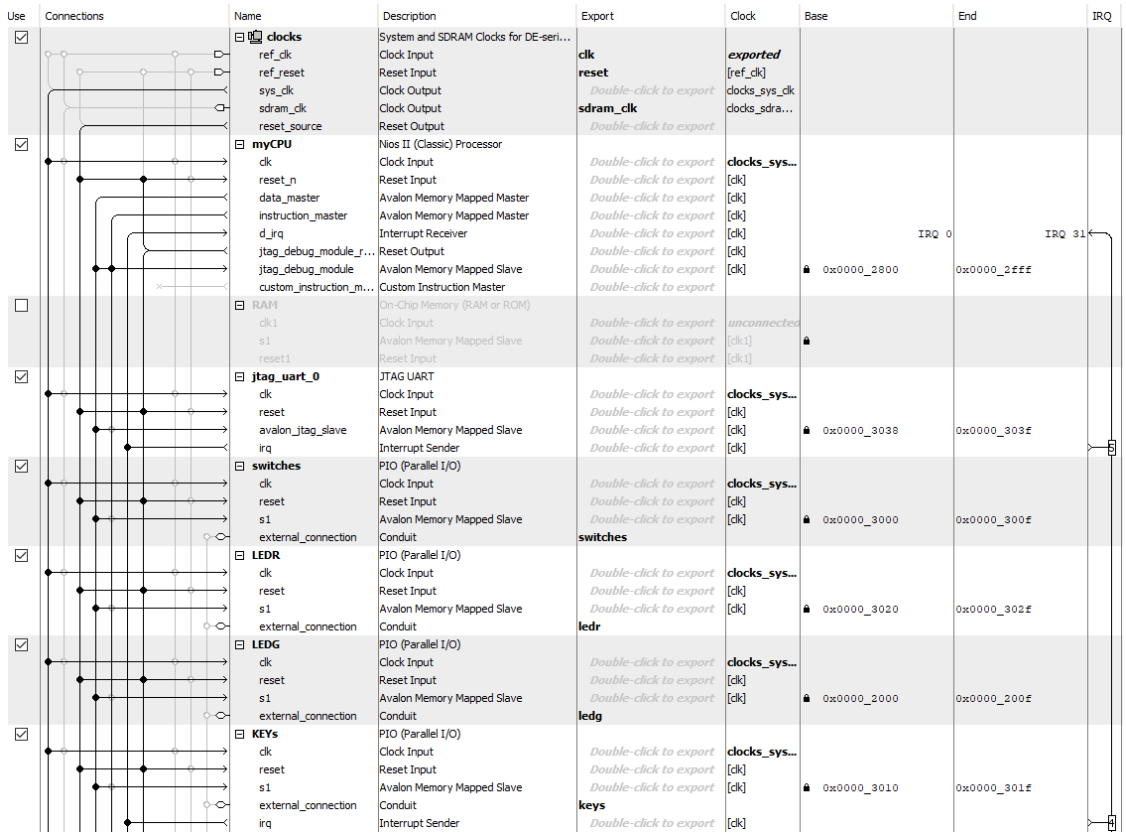

Figure 6.1.2: Image connections for the soft-core system.

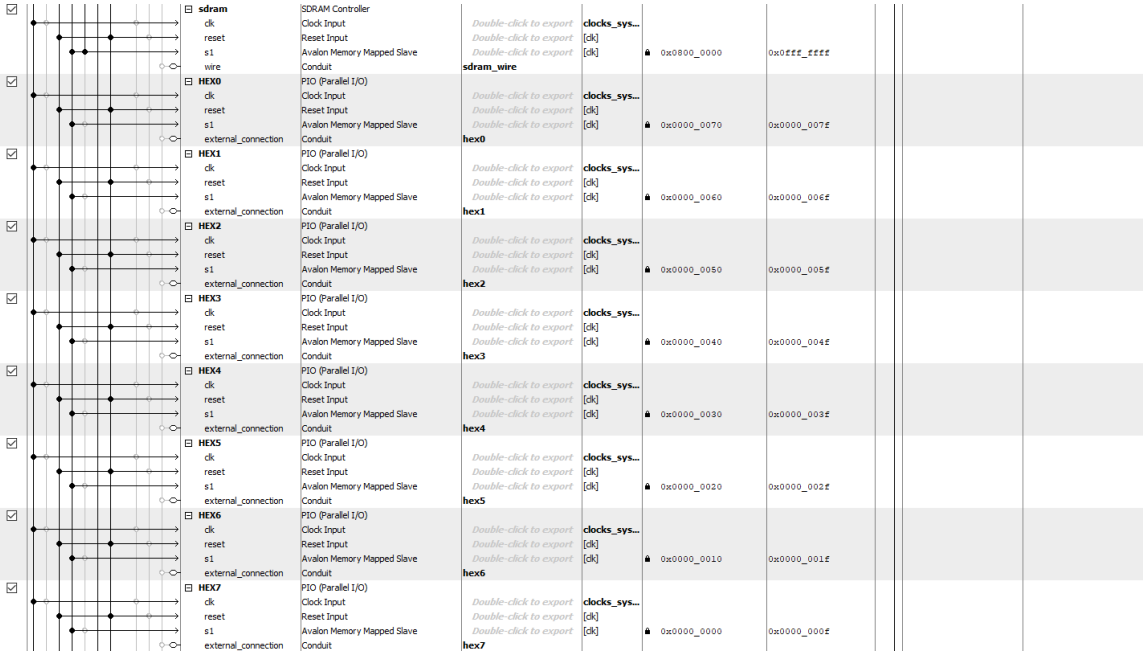

Figure 6.1.3: Image connections for the soft-core system.

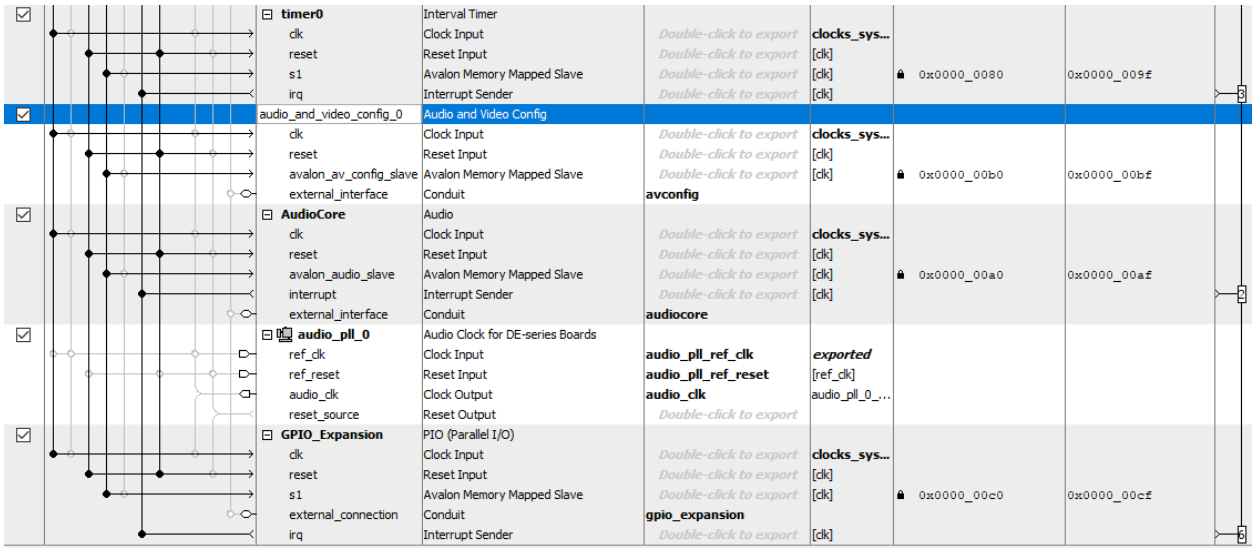

Figure 6.1.4: Image connections for the soft-core system, the Audio Cores, and GPIO.

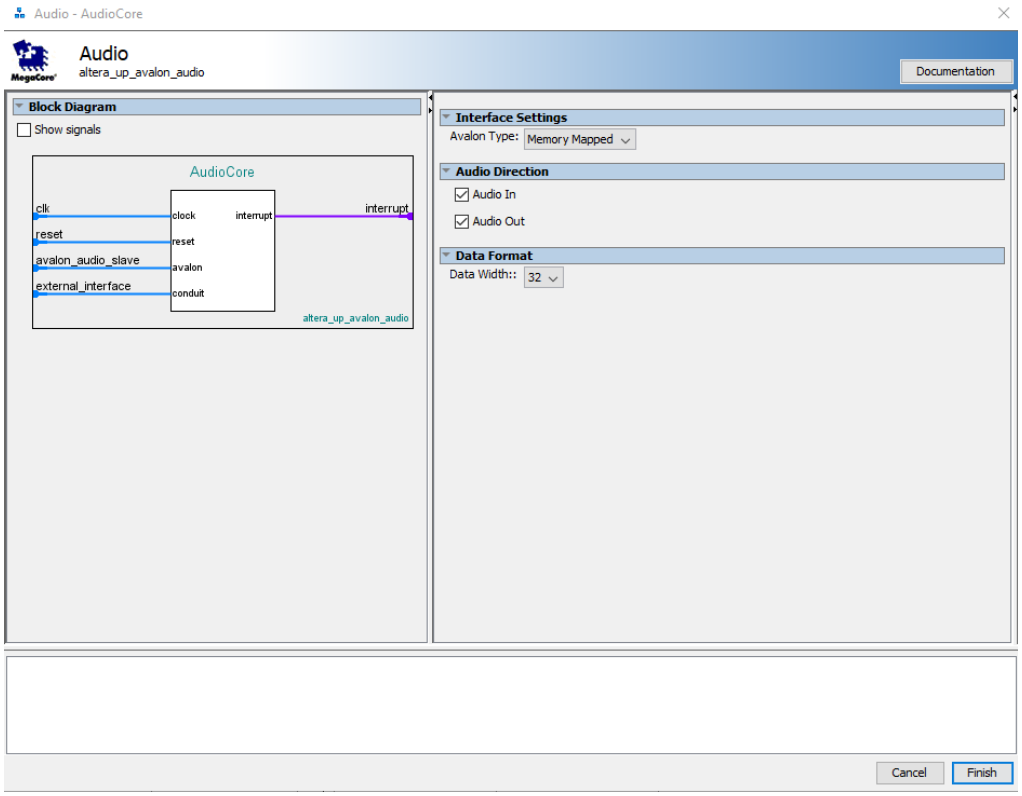

Figure 6.1.5: Image showing the settings selected for the Audio Core.

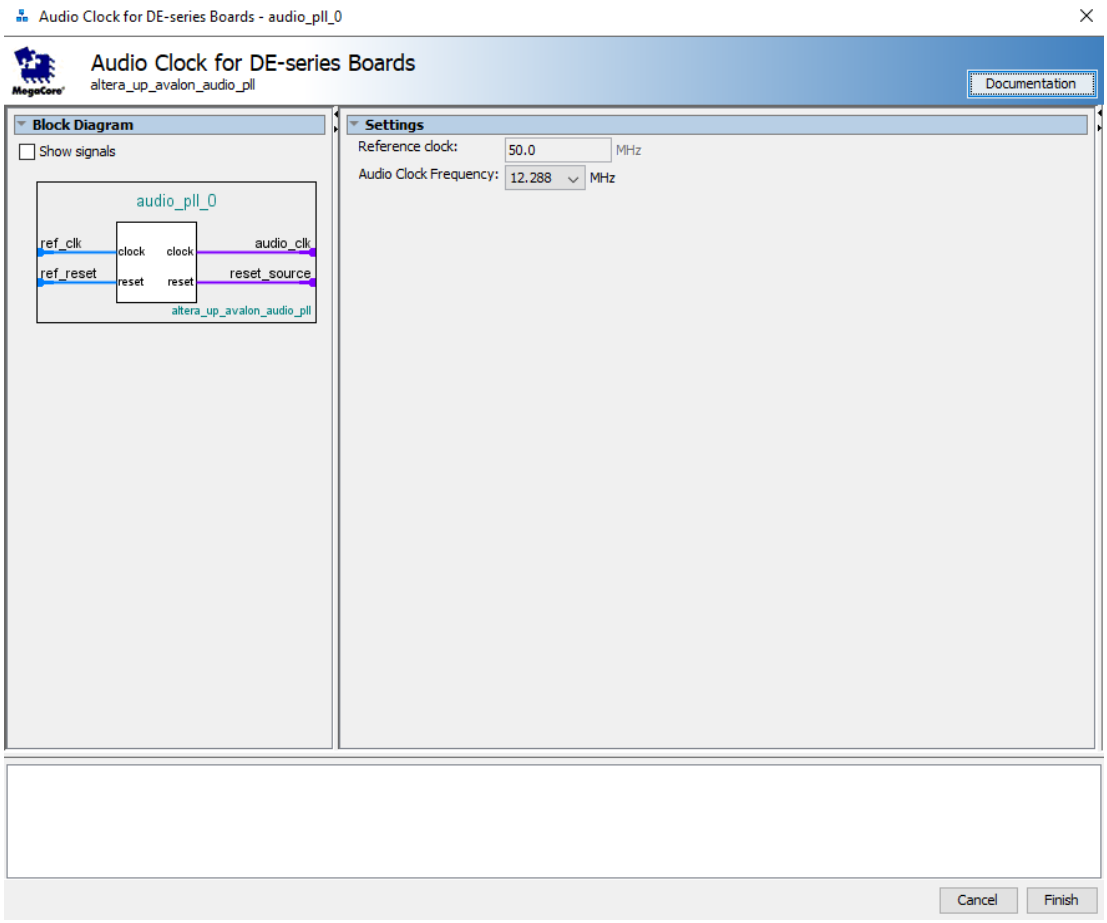

Figure 6.1.6: Image showing the settings selected for the Audio Clock for DE-series Boards.

After setting up the Audio Cores, we then set up the GPIO PIO. We added a normal PIO from Qsys since the GPIO is treated as a peripheral. We then filled out the settings as seen in Figure 6.1.7. We chose 32 bit width to make sure that the data register was at its maximum capacity. We also made sure to choose Bidirectional so that we can have pins on the GPIO be treated as inputs or outputs. We use the edge capture register to make sure to capture either edge. This was to make sure that the buttons would be checked whenever there was a change in the signal. We also use the GPIO to have the ability to throw interrupts. This helped us in the software implementation of the key presses. Once we completed the setting, we then connected the core based on Figure 6.1.6.We then selected the Generate tab at the top and selected Generate HDL. We saved our program in our project folder and once the generation was finished we continued onto the design instantiation.

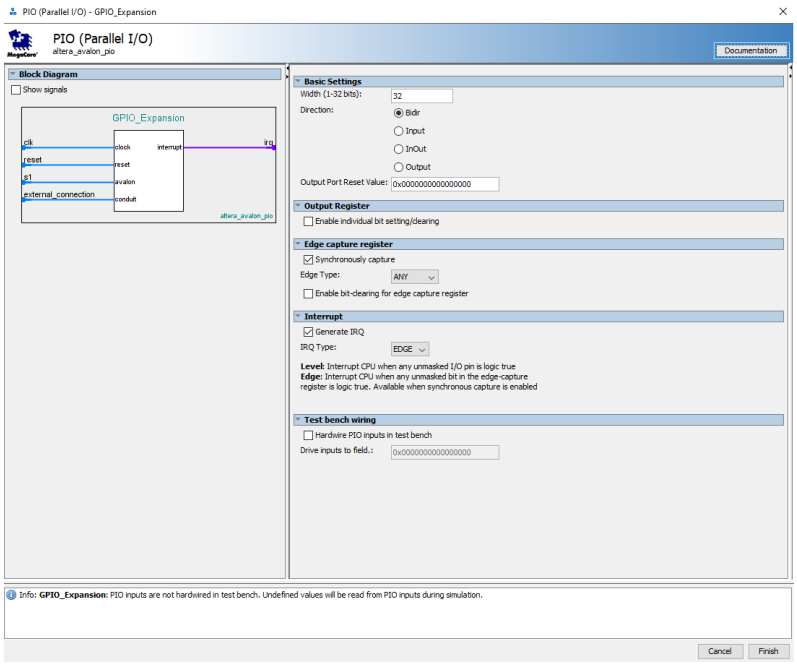

Figure 6.1.7: Image showing the settings selected for the GPIO PIO.

6.2. Instantiation, Pin Assignment, and Compilation

When we were finished generating our HDL, we need to create an instantiation of our new module in a .v file. To create a v-file we selected File > New at the top of Quatus. We then selected Verilog HDL File under Design Files. We copied and pasted the code in Appendix A under “Instantiation Code”. We made sure to save the file with the name of the project. To add this file to the project we selected the Project tab at the top of Quartus and selected Add/Remove Files in Project. Then we searched for the saved file in our directory and added it. We also added our .qip file which was found in *name of system*>synthesis. We selected, applied and then went back to the Quartus project. Here we started our initial compile in Quartus.

After Quartus stopped it’s compilation process we accessed the pin assignment. To do this we used a provided .qsf file which already contains all the pin assignments. The contents were given to us by the instructor. The file itself will not be included in this documentation, but a download to the file can be found in the reference section[6]. The file was then opened in Notepad and the entire file was copied and pasted into the .qsf file located in the project root directory. After copying and pasting, the file was saved.

6.3. Nios II SBT Project Creation and Board Connection

We began our project by first creating our project workspace. When we started the Nios II SBT application we were prompted to choose a directory for our workspace. We chose the directory we wanted and selected OK as shown in Figure 6.3.1. We then selected File on the top left of the

new window, hovered over New, and selected Nios II Application and BSP from Template as shown in Figure 6.3.2. In the new window, we selected our .sopcinfo file that was generated by Qsus and made sure that it was the correct CPU name. Then we gave the project name AudioKeyboard and made sure Blank Project was selected as a project template. The full selected settings are shown in figure 6.3.3. We then selected Finish.

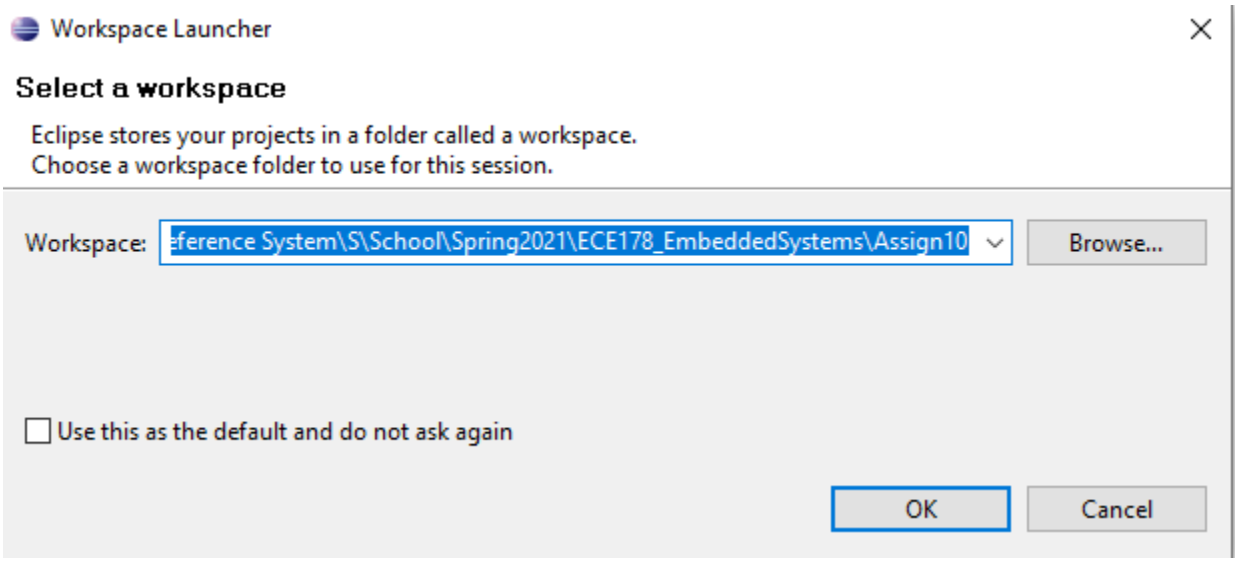

Figure 6.3.1: Figure showing a popup that occurs when opening Nios II SBT.

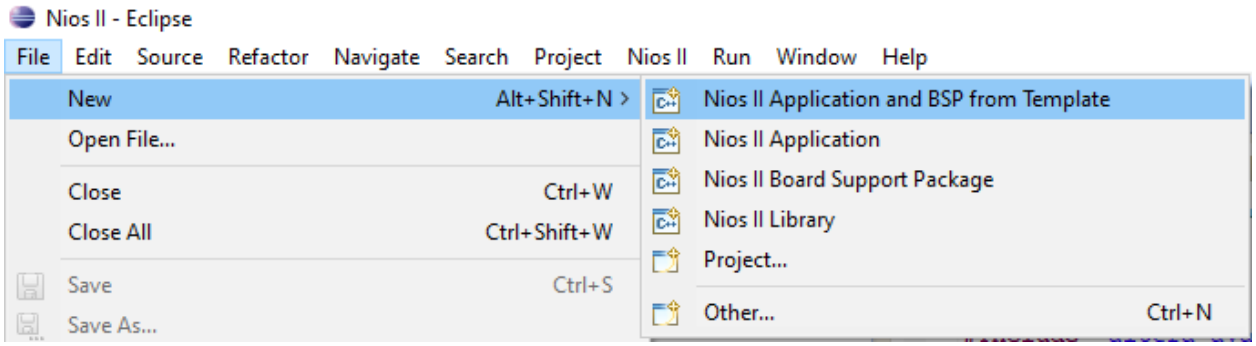

Figure 6.3.2: Figure showing how to create a new application with a template.

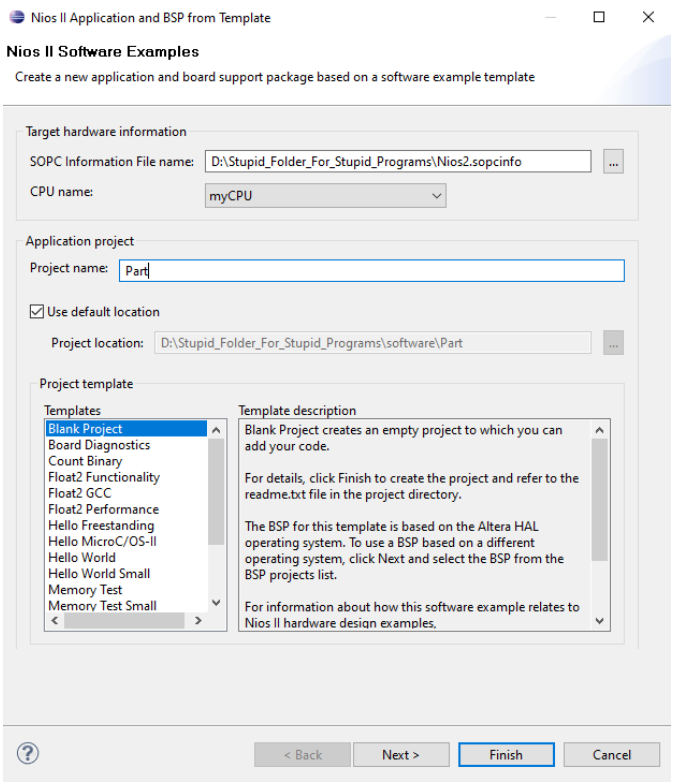

Figure 6.3.3: Window showing the needed settings for Nios II Application and BSP from Template.

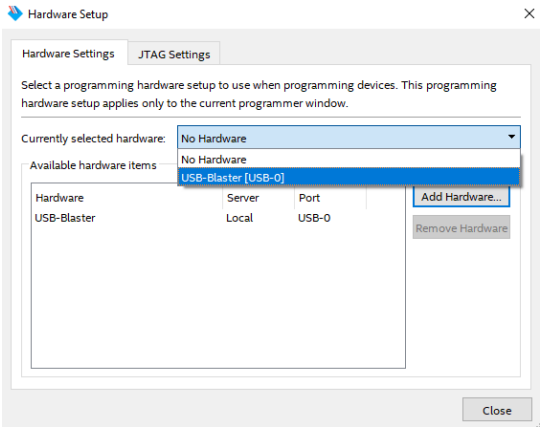

Figure 6.3.4: Hardware Setup menu with correct selected settings.

We then needed to connect our board via the USB blaster. We selected Nios II at the top and selected Quartus Prime Programmer. A new window appeared, and we then selected Add File on the side. We then navigated to where our .sof file was located and added it to the project. Then we needed to add our hardware. We selected Hardware Setup near the top left of the Quartus Prime Programmer window. Once selected we were greeted with another window. In the

Hardware Setup window, we clicked on the dropdown menu and selected USB-Blaster[USB-0] as shown in Figure 6.3.4. Once this was done we selected close. After this we selected Start to begin loading our program to the board. Once the download was completed it should look like Figure 6.3.5. After completion, we did not close out of the programmer. We left the programmer open in the background before we went back to Nios II SBT.

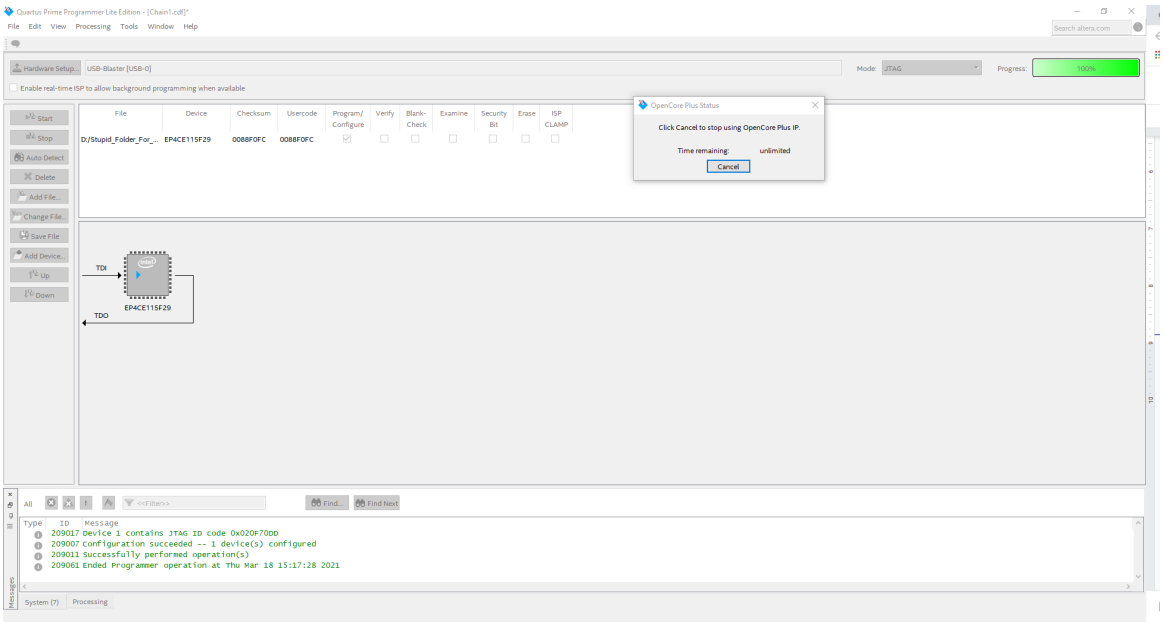

Figure 6.3.5: Figure showing the result of the completion of the program download.

6.4. Software Implementation of Solution

When starting the program implementation of our solution, we needed to make sure we included the needed libraries for our program. These libraries can be found in Appendix A under AudioKeyboard Code. Initially we created our needed globals that are used for testing and implementation. We created 4 global arrays called contextArray, sinArray, notes, and last_on. ContextArray was used to hold any contexts that were needed for interrupts. SinArray was used to keep any data about the generated waveform used by any of the functions. Notes array was used to keep the needed periods for the waveforms. Finally last_on was used to keep tack if the note was turned on or off when the audio interrupt is handled. We also defined our audio struct that is used to define our audio system. This audio struct will be used to fill in Audio FIFOs for reading and writing data. We also define a sampling frequency FS to be 48000.

We then moved onto working on initializing the interrupts needed for this project. We first had to make pointers to the contextArray, sin_array, and last_on arrays. We had originally set up the IRQ mask for the keys to make sure that interrupt functionality was working on the board. We also made sure to clear the edge capture register for the keys. We set this up so we could read from this register to delay the exit of ISRs. We then used the alt_up_audio_open_dev to open our

audio device and initialize the struct. These audio functions were created by the Nios II BSP and provided as a driver for the audio core. We also set up some logic that indicated if the device could be opened or not. We then moved onto setting the GPIO by setting the DDR to 0. This means that all the GPIO pins were set to input. We also set the IRQ mask register to 0xFFF by using IOWR. This made sure to set the first 12 pin of the GPIO to be able to set off interrupts based on the edges of the input signal. After setting this up, we made sure to clear the edge capture register of the GPIO.

For the Audio core, we needed to reset the FIFOs just in case there was any lingering data. To do this we used the alt_up_audio_reset_audio_core function. We then disabled the audio write and read interrupts to make sure the audio does not throw any interrupts when the initialization is complete. We then had to register audio interrupts with HAL by using alt_irq_register with the arguments of the IRQ number, context pointer for the interrupt, and name of the ISR function. How we registered the interrupts using alt_irq_register can be found in the Appendix under AudioKeyboard Code in the pio_init function. To delay the exit of the function, we cleared the edge capture of the keys and read the edge capture register.

We then worked on the ISR for the audio interrupts called handle_audio_interrupts. We first made sure to make a pointer to the sinArray global. We used this array to contain any changes to the wave function. We then initialized a few variables from the array. We created a variable called ‘n’ that would hold the first index of the array. This would hold the discrete step in time that the signal was at. We initialized the next variable called ‘f’ that will hold the second index of the array. This will hold the period that will be used in the equation used to develop a saw function. The next variable defined was signal which would hold the signal data that is inserted into the FIFO for the Audio Core. Then we create a logic block that would check the write FIFO to figure how much space was left to insert any data for the signal. This value was put into a variable called ‘a’. We then create another array that was used as a container for generated signal data to sit in before being sent to the FIFO. After setting up some initial variables, we started the main portion of the function by polling the switches. When polling the switches data register, we will shift the period ‘f’ to the right depending on the value of the switches.

The creation of the data for the signal was set up into a for loop that created as many data points as needed by the write FIFO. Each loop, we would update ‘n’ and set the signal to the absolute value of the modulus of the period. Then we would take that same signal and multiply it by the value inside test. The value inside the test was determined by the quotient of 16777216 and ‘f’, with 16777216 being the maximum peak to peak amplitude for the saw wave. We then make sure to increase the n value stored in the sinArray to keep track of where we are in the signal. We would then check if we are greater than or equal to the value of the limit, which is set to 10 times their period of the period value ‘f’. If this was true, then we would set the n value store in sinArray to 0. Then we would shift the signal 8 times to the left and store it into the x array. Then

we would calle the audio write FIFO function for the left and right channels and point the buffer to the x array that held our singal data.

We then moved on to create our GPIO handle interrupts ISR. Here we would use the last_on array to check if a button was turned on or off depending on the last time the array was checked. We first checked the edge capture array of the GPIO and initialized a variable called check. Then we set up a for loop that will cycle 12 times. On each loop, check will be set to 1 and shifted to the left based on the idex of the loop. We then check if the edgcapture was set to 1 based on the index of the loop. This is essentially checking if the signal has changed based on the key inputs. If the signal did change, then a logic block would determine if the button was turned on or off the last time it was serviced. If it was turned on, then the program will clear the red LEDs, disable the write interrupt for the audio, and reset the ‘n’ value stored in the sinArray. If the key was turned off the last time the key was serviced, then the program will turn on the red LED corespoining with the value of check. It will also turn on the write interrupts and set the period of the signal based on the notes array. Before the function leaves, the program will read the edge capture register to delay the exit.

Inside the main function, the program remains only to initialize the arrays and interrupts. The first thing the program does is clear the red LEDs. Then it creates a pointer to sinArray and sets the first index to 0. After, it will set the next index some period based on the notes array. Finally it will call the pio-init function to initialize the interrupts and move into an infinite while loop. We would normally try to use the program from here, but we first needed to create our user interface using circuit implementation.

6.5. User interface circuit implementation

When we created the user interface circuit, we started by creating a single implementation of the circuit described in Figure 4.5.1. However we did not have access to any schmitt triggers, or capacitor resources were limited, and our number of diodes were limited to 8 LEDs. So when we set up the first 8 circuits, we used the same circuit design but omitted the schmitt trigger. Then for the last 4 circuits we omitted the LEDs and schmitt trigger. A picture of our design can be found in Figure 6.5.1. We set up 12 separate circuits with each one corresponding to a separate note. We spaced the keys to represent the notes on a piano and the notes range chromatically from F to E.

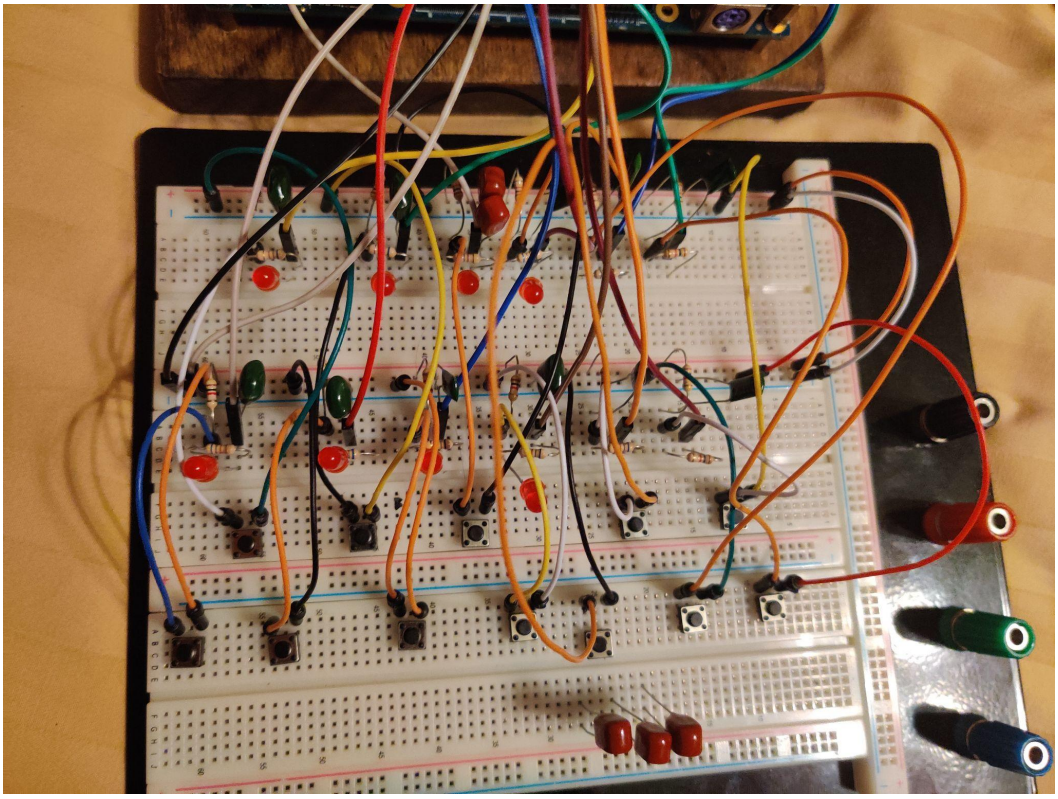

Figure 6.3.5: Image showing the completed circuit for the user interface.

7. PROOF OF COMPILATION

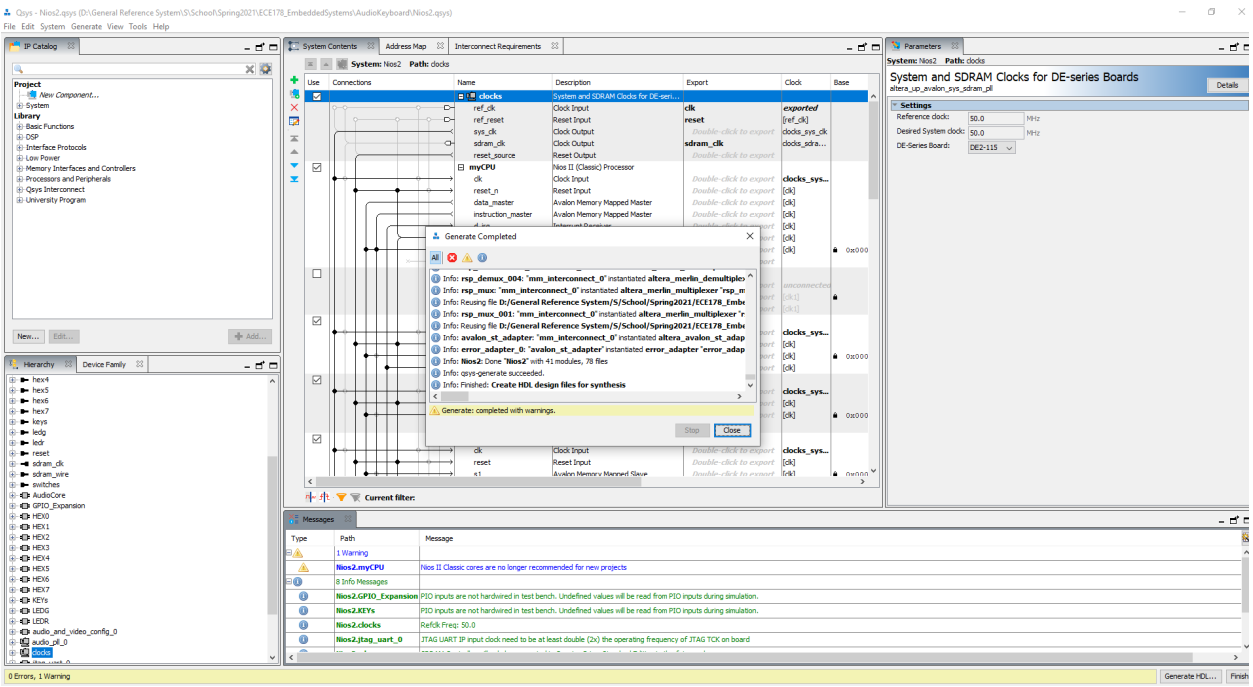

Figure 7.1: Proof of successfully generated HDL in Qsys.

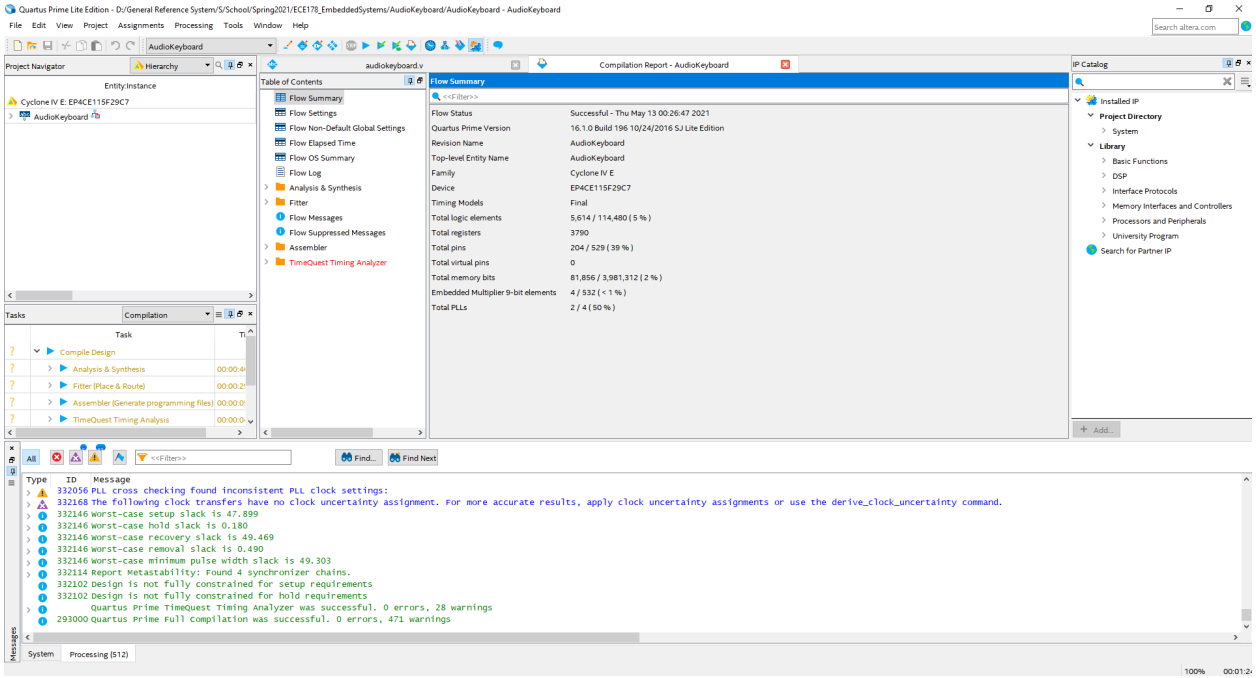

Figure 7.2: Proof of successfully compiled module instantiation.

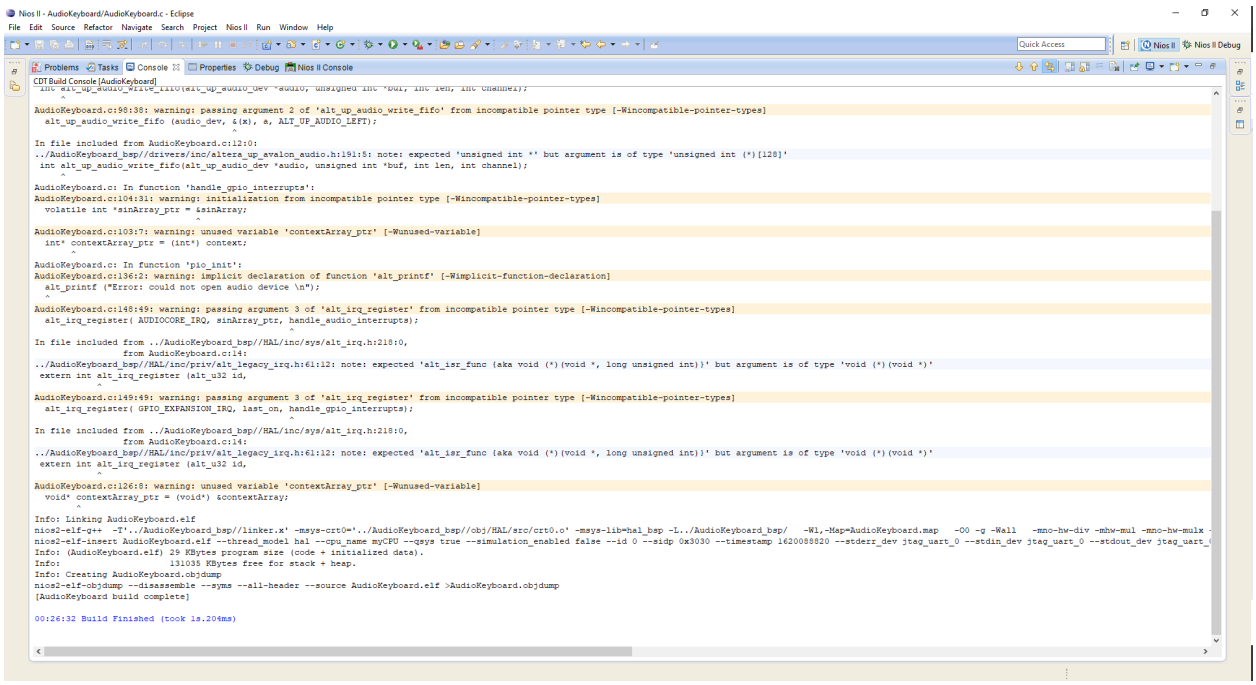

Figure 7.3: Image of Successful Compilation of Program

8. ANALYSIS OF RESULTS

PCM Data Solution:

To generate the correct generation of PCM data we needed to be able to generate the correct data to mimic a saw signal. We know that we would like to be able to control the period or frequency of the equation. This is because we would like to change the signal based on the input frequency. We also know that the maximum data value would be 0x7FFFFF. This is because 2's complement data ranges from -2^{n-1} to $2^{n-1}-1$. Thus the peak to peak amplitude should be 0xFFFFFF since the bit depth is 24 bits. We also knew that the sample rate was preset to 48 kHz. So when making this equation we first considered a continuous time signal. We came up with the following equation:

$$signal = abs(n \,\%\, f) * (16777216/f) - 8388607 \qquad \textbf{Equation: 8.1}$$

Where n represents time and f represents the period of the frequency.

This equation creates a saw that ranges from -8388607 to 8388606 and has a period of f. This is because the absolute value will produce a number ranging from 0 to f. Then this number will be divided by f which should create a fractional value from 0 to 1. Then that value will be multiplied 16777216 which will then be shifted down by 8388607. When n increases, then the value will increase through the peak to peak range it will then drop back to the lowest value and restart the incrementing process at the period of f.

The period f is predefined in the notes array. We calculated these values using the sample rate of the audio codec. Since we know that the sample frequency is set to 48 kHz, we can use the calculated period to find how many of those periods will fit into any needed period. For example, for a needed period of 440 Hz we would use 1/440 to find the period of the desired signal. Then we would find the period of 48 kHz by 1/48000. We would then take the period of the desired and dividing it by the period of the sample rate which would result in 48000/440. Thus the period of the discrete signal would be rounded to 109 sample units.

This equation can be seen implemented in hanfle_audio_interrupts. Inside the for loop the signal implements tha absolute value using math.h with a modulus operation. Then it would be multiplied with the result of $16777216/f$ which is defined outside the for loop to create efficient code. Then the result is shifted to create signed data. Octaves are defined by dividing the period by 2. This is implemented by simply shifting the data to the right depending on the octave. All periods are defined in notes array for each tone.

Result of Debouncing:

When creating the circuit, we wanted to make sure that the circuit would not be hazardous in any way. We also wanted to make sure parts would not be damaged in any way. When considering this, we needed to check out capacitor ratings first. We knew Mylar capacitors are normally

bidirectional, which was denoted by the lack of markings for the positive and negative terminals. Thus, we knew that we could insert the capacitors into the circuit without the fear of them exploding. These specific capacitors were also rated for 100V. So we were fairly confident that these capacitors could handle our circuit. We also wanted to make sure that the LED didn't burn out. We knew that our LED could only handle up to 20 mA of current. So we needed to do some basic RC analysis to figure out our range of resistance values needed for the circuit. We know that at the initial state of the circuit, we can assume that the capacitor will ideally act as a short circuit. This means that Current will run from Vcc through a pullup resistor and diode with a drop off voltage of 1.8V. This means that voltage across the resistor would be 3.2 V if Vcc is set to 5 V. When using Ohm's law, we can then get the resistance range of R>=160 Ω. This worked out since all of our resistors were all above this calculated value.

When testing the circuit of the debouncing effectiveness, we can see that the circuit works. Yet, sometimes the switches will bounce and set off incorrect data. We can see this happening when we press a button and the button press inverts operation. This means that when the button is pressed, it will no longer play sound. Instead when the button is depressed, it will create the tone associated with the key. This was an interesting result since we set up the circuit correctly. We can only think of a few possible problems. The first possible problem could be the lack of schmitt triggers. This could be the key component used to make sure that the transition of logic values is clean enough to reduce the probability of bouncing. The next could lay in the cheap switches used. No matter what we would do to the circuit, if the switches are cheap enough to create massive switching then there will definitely be problems with the circuit. More quality switches could be invested towards fixing this problem.

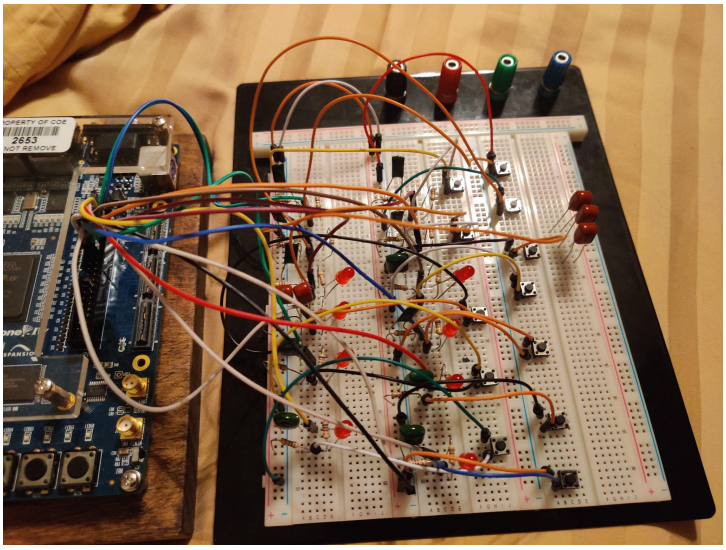

Figure 8.1: Figure showing completed user interface with debouncing circuit.

Functionality of the Implementation:
When starting the program, no sound is played from the headphones. Only until one of the buttons are pressed on the user interface do we receive any sound. In the fashion of a keyboard, we can play melodies based on the octave the board is set to. If none of the switches are flipped up, the default octave is set to the first octave. If the switches are set to 1 then the notes on the user interface are set to the second octave. If the switches are set to 3 then the notes are set to the third octave and so on. If a button is pressed and the switches are changed, then the octave of the current not being played will change.

To test the quality of a note, a musician must use a drone or a tuner to check the correctness of their pitch. A musician will use a drone to compare the quality of their pitch. They can either be flat, sharp, or right on pitch. When a drone is enabled, the true frequency of the tone will ring out of a device and the musician will use their ears to tune their instrument. A tuner on the other hand will tell the musician how sharp or flat the musician is by a display on the device. These are normally indicated by cents sharp or flat. Since it would be difficult to qualitatively describe the quality of the project's pitch, we decided to use a tuner to quantitatively measure the pitch produced by the project. In Figure 8.2 we are pressing the F key in the second octave. We tested the quality using an online tuner from theonlinemetronome.com. We simply placed the sound source next to a microphone and received the results seen in Figure 8.3.

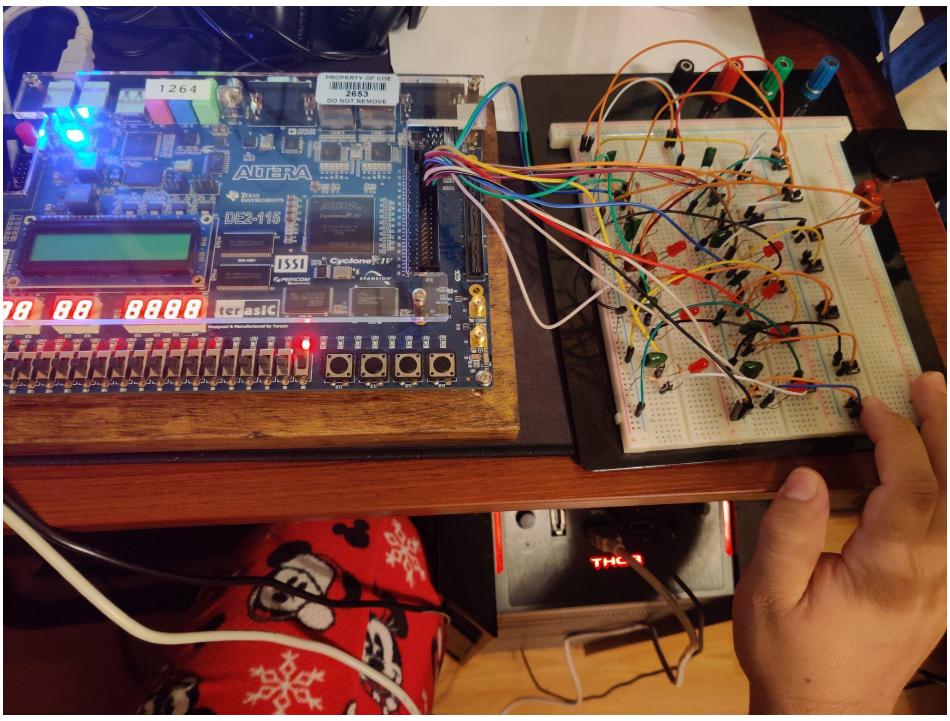

Figure 8.2: Image showing the user pressing

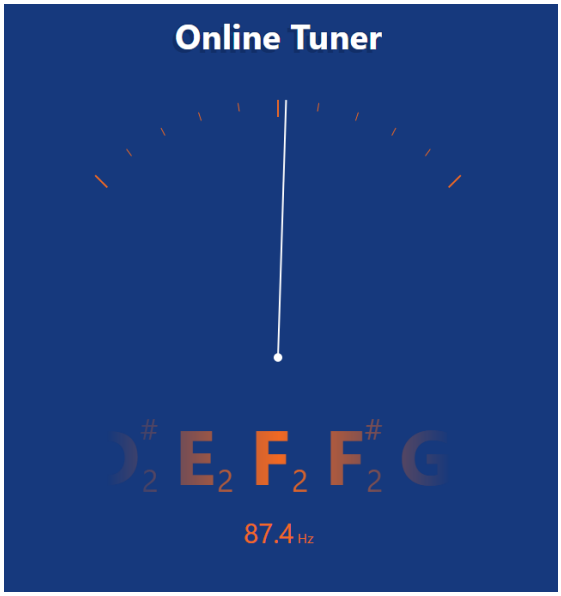

Figure 8.3: Image showing the results of the tuner when pressing the F key at the second octave.

The results from Figure 8.3 show that the quality of the tone is actually relatively close to being on perfect pitch. These same results remain consistent as we move up the octaves. However, once we reach the 5th octave the pitch becomes increasingly sharp as shown in Figure 8.4. The increase of dissonance in the pitch could be due to data becoming increasingly smaller. This may be due to the fact that we shift the period to the right when increasing the octave.

Figure 8.4: Image showing the results of the tuner at the 5th octave when pressing the F key.

Shortcomings of the Project:
The overall project had completed its main goal of producing sound and creating a playable keyboard. However, the project did have plenty of setbacks and shortcomings due to many reasons. Originally the design was to include the ability to play 3 different notes at the same time. This project was also supposed to be able to control the volume of the sound. Though looking at the implementation now, it may seem simple to add these features. Yet, they were not added due to one main resource. This resource was time. The project ran too long despite the fact that hours upon hours were spent researching and setting up this project. To get the project to its current state, it took weeks to research and test to even get a working premade demo. There were days where 12 hours or more were spent to implement a simple wave using PCM data. This was because the original project was also supposed to use sine waves and not saw waves.

Though time was the main reason why the project fell short of initial expectations, the lack of knowledge also became a key factor on why the project was not able to meet the original standards. We had found many implementations that practically created our same project. We even found one that accomplished what we set out to do from another college. However, they were a 2 man crew and they had a different method of implementation. They created their solution based on verilog code formed in Quartus. We had no idea that this could work, nor did we have any experience in using verilog to implement logic for an FPGA. Since the course we had was based on the Nios II and Intel FPGA monitor program, we could not in good confidence use any of their solutions for help. Instead we set out to create a Nios based solution which had little to no references to use. We had to spend hours upon hours researching manuals, specs of help from other similar implementations, and our own tinkering to get any adequate progress on the project.

9. CONCLUSION

Although the project did not meet the full expectations of the original design, we were able to create a working synthesizer with a working user interface that replicated piano keys. The tones generated were able to be relatively in tune and could be played up to 7 octaves. Overall, we believe that this was a success in it's own right. There were a lot of learning experiences presented in this project. Although we did learn how to use the Audio Codec, PCM, debouncing circuits, and GPIO, we felt we learned more about project development than anything.

This project has humbled us as designers and students. We originally underestimated the difficulty of this challenge which led us to be overconfident in our ability to promise more than we could handle. We can see that we have a weakness in our ambition to achieve things that are beyond our current ability and resources. Our stubbornness to change the scope of our project had also led to inefficient time use in the project since we used a lot of our time brute forcing our way to progress. Although dedication and hardwork are qualities to be valued, they are not to be confused with stubborness and unhealthy work habits. We often walk a tight line between a healthy balance of productivity and an obsession to work. As long as we can take note of this weakness and acknowledge it, we can grow further as future engineers and as a better person.

10. REFERENCES

[1] PCM - Analog to digital conversion. (2018, November 16). [Video]. YouTube. https://www.youtube.com/watch?v=HlGJ6xxbz8s&feature=youtu.be

[2] "Nyquist Frequency," from Wolfram MathWorld. [Online]. Available: https://mathworld.wolfram.com/NyquistFrequency.html. [Accessed: 11-May-2021].

[3] "WM8731 / WM8731L ." Wolfson Microelectronics, Edinburgh, Feb-2005.

[4] R. Raeisi, "HW5." Reza Raeisi, Fresno, 2021. File given in ECE 178 Fresno State Canvas

[5] A. Greensted, "The Lab Book Pages," Sitewide RSS, 17-Jun-2010. [Online]. Available: http://www.labbookpages.co.uk/electronics/debounce.html. [Accessed: 12-May-2021].

[6] R. Raeisi, "DE2_115." Reza Raeisi, Fresno, 2021. File used for pin assignments. https://fresnostate.instructure.com/courses/39085/files/5060768/download?download_frd=1

APPENDIX A

Instantionation Code:

```
module
AudioKeyboard(LEDR,LEDG,CLOCK_50,CLOCK2_50,RESET,KEY,SW,HEX0,HEX1,HEX2,
HEX3,HEX4,HEX5,HEX6,HEX7,
GPIO,
AUD_XCK,
AUD_ADCLRCK,AUD_ADCDAT,AUD_DACLRCK,AUD_BCLK,AUD_DACDAT,
I2C_SCLK, I2C_SDAT,
DRAM_CLK,DRAM_CKE,DRAM_ADDR,DRAM_BA,DRAM_CS_N,DRAM_CAS_N,DRA
M_RAS_N,DRAM_WE_N,DRAM_DQ,DRAM_DQM);

        input CLOCK_50, CLOCK2_50;
        input RESET;
        input [3:0] KEY;
        input [17:0] SW;
        output [17:0] LEDR;
        output [7:0] LEDG;
        output [12:0] DRAM_ADDR;
        output [1:0] DRAM_BA;
        output DRAM_CAS_N,DRAM_RAS_N,DRAM_CLK;
        output DRAM_CKE,DRAM_CS_N,DRAM_WE_N;
        output [3:0] DRAM_DQM;
        inout [31:0] DRAM_DQ;
        output [6:0] HEX0;
        output [6:0] HEX1;
        output [6:0] HEX2;
        output [6:0] HEX3;
        output [6:0] HEX4;
        output [6:0] HEX5;
        output [6:0] HEX6;
        output [6:0] HEX7;
        input AUD_ADCLRCK, AUD_ADCDAT, AUD_DACLRCK, AUD_BCLK;
        output AUD_DACDAT, AUD_XCK;
        inout I2C_SDAT;
        output I2C_SCLK;
        inout [31:0] GPIO;
        wire [31:0] HEX0_3;
        wire [31:0] HEX4_7;
```

```
	assign HEX0_3 = {1'b0,HEX3,1'b0,HEX2,1'b0,HEX4,1'b0,HEX0};
	assign HEX4_7 = {1'b0,HEX7,1'b0,HEX6,1'b0,HEX5,1'b0,HEX4};
	Nios2 mySystem(
		.audio_clk_clk(AUD_XCK),
		.audio_pll_ref_clk_clk(CLOCK2_50),
		.audio_pll_ref_reset_reset(RESET),
		.audiocore_ADCDAT(AUD_ADCDAT),
		.audiocore_ADCLRCK(AUD_ADCLRCK),
		.audiocore_BCLK(AUD_BCLK),
		.audiocore_DACDAT(AUD_DACDAT),
		.audiocore_DACLRCK(AUD_DACLRCK),
		.avconfig_SDAT(I2C_SDAT),
		.avconfig_SCLK(I2C_SCLK),
		.clk_clk(CLOCK_50),
		.gpio_expansion_export(GPIO),
		.keys_export(KEY),
		.ledr_export(LEDR),
		.ledg_export(LEDG),
		.reset_reset(RESET),
		.switches_export(SW),
		.hex0_export(HEX0),
		.hex1_export(HEX1),
		.hex2_export(HEX2),
		.hex3_export(HEX3),
		.hex4_export(HEX4),
		.hex5_export(HEX5),
		.hex6_export(HEX6),
		.hex7_export(HEX7),
		.sdram_clk_clk(DRAM_CLK),
		.sdram_wire_addr(DRAM_ADDR),
		.sdram_wire_ba(DRAM_BA),
		.sdram_wire_cas_n(DRAM_CAS_N),
		.sdram_wire_cke(DRAM_CKE),
		.sdram_wire_cs_n(DRAM_CS_N),
		.sdram_wire_dq(DRAM_DQ),
		.sdram_wire_dqm(DRAM_DQM),
		.sdram_wire_ras_n(DRAM_RAS_N),
		.sdram_wire_we_n(DRAM_WE_N));
endmodule
```

AudioKeyboard Code:

```
/*
 * AudioKeyboard.c
 *
 *  Created on: May 3, 2021
 *      Author: jacob
 */

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
#include "altera_up_avalon_audio.h"
#include "altera_avalon_pio_regs.h"
#include <sys/alt_irq.h>

static void handle_audio_interrupts(void* context);
static void handle_gpio_interrupts(void* context);
static void pio_init();

volatile int contextArray[4];
int sinArray[3];
static int notes[12] = {1099,1038,980,925,873,824,777,734,693,654,617,582};
static int last_on[12] = {0,0,0,0,0,0,0,0,0,0,0,0};

/* used for audio record/playback */
alt_up_audio_dev *audio_dev;
unsigned int l_buf;
unsigned int r_buf;
const double FS = 48000;

int main(void)
{       /*Clear LEDs*/
        IOWR_ALTERA_AVALON_PIO_DATA(LEDR_BASE,0);
        /*Set up global arrays*/
        //sin generation
        volatile int *sinArray_ptr = (volatile int*) sinArray;
        *sinArray_ptr = 0; //space for n value
        *(sinArray_ptr + 1) = notes[3]; //set base frequency to 440hz
        *(sinArray_ptr + 2) = 0;
```

```
	/*Initialize Interrupts*/
	pio_init();
	while(1)
	{

	}
}

static void handle_audio_interrupts(void* context){
	int *sinArray_ptr = (int*) sinArray;
	int n = *sinArray_ptr;
	int f = *(sinArray_ptr+1);
	int signal = 0;
	unsigned int a = alt_up_audio_write_fifo_space(audio_dev,
ALT_UP_AUDIO_RIGHT)/2;
	if (a < (alt_up_audio_write_fifo_space(audio_dev, ALT_UP_AUDIO_LEFT))){
		a = alt_up_audio_write_fifo_space(audio_dev, ALT_UP_AUDIO_LEFT);
	}
	unsigned int x[128];
	/*
	if(alt_up_audio_read_interrupt_pending(audio_dev)){
		// read audio buffer
		alt_up_audio_read_fifo (audio_dev, &(r_buf), 1, ALT_UP_AUDIO_RIGHT);
		alt_up_audio_read_fifo (audio_dev, &(l_buf), 1, ALT_UP_AUDIO_LEFT);
	}
	*/
	f = f >> IORD_ALTERA_AVALON_PIO_DATA(SWITCHES_BASE);
	int test = 16777216/f;
	int limit = 10*f;
	for(unsigned int i = 0; i < a; i++) {
		n = *sinArray_ptr;
		signal =  (abs(n % f));
		signal = signal *test-8388607;
		*sinArray_ptr = n + 1;
		if(*sinArray_ptr>=limit)*sinArray_ptr = 0;
		/*
		if(signal >= 8388606){
			*(sinArray_ptr+2) = 1;
			*sinArray_ptr = n - 1;
			signal = 8388606;
```

```
			}
			else if (signal <= -8388607){
				*(sinArray_ptr+2) = 0;
				*sinArray_ptr = n + 1;
				signal = -8388607;
			}
			else if (*(sinArray_ptr+2)) *sinArray_ptr = n - 1;
			else *sinArray_ptr = n + 1;
			*/
			x[i] = signal << 8;
		}
		/*
		if(alt_up_audio_write_interrupt_pending(audio_dev)){
			// read audio buffer
			alt_up_audio_write_fifo (audio_dev, &(r_buf), 1, ALT_UP_AUDIO_RIGHT);
			alt_up_audio_write_fifo (audio_dev, &(l_buf), 1, ALT_UP_AUDIO_LEFT);
		}
		*/
		alt_up_audio_write_fifo (audio_dev, &(x), a, ALT_UP_AUDIO_RIGHT);
		alt_up_audio_write_fifo (audio_dev, &(x), a, ALT_UP_AUDIO_LEFT);

}

static void handle_gpio_interrupts(void* context){
		int* contextArray_ptr = (int*) context;
		volatile int *sinArray_ptr = &sinArray;
		int edge = IORD_ALTERA_AVALON_PIO_EDGE_CAP(GPIO_EXPANSION_BASE);
		int check = 0;
		for(int i = 0; i<12;i++){
			check = 1 << i;
			if(edge & check){
				if(last_on[i]){
					IOWR_ALTERA_AVALON_PIO_DATA(LEDR_BASE, 0);
					alt_up_audio_disable_write_interrupt(audio_dev);
					*(sinArray_ptr+1) = 0;
				}else{
					IOWR_ALTERA_AVALON_PIO_DATA(LEDR_BASE,
(IORD_ALTERA_AVALON_PIO_DATA(LEDR_BASE)^check));
					alt_up_audio_enable_write_interrupt(audio_dev);
					*(sinArray_ptr+1) = notes[i];
```

```
                        }
                        last_on[i] = ~last_on[i];
                }
        }
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(GPIO_EXPANSION_BASE,0);
}

static void pio_init(){
        void* contextArray_ptr = (void*) &contextArray;
        void* sinArray_ptr = (void*) &sinArray;
        void* last_on = (void*) &last_on;
        /*Enable all 4 button interrupts*/
        IOWR_ALTERA_AVALON_PIO_IRQ_MASK(KEYS_BASE, 0xE);
        /* Reset the edge capture register. */
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0x0);
        // open the Audio port
        audio_dev = alt_up_audio_open_dev ("/dev/AudioCore");
        if (audio_dev == NULL)
        alt_printf ("Error: could not open audio device \n");
        else
        alt_printf ("Opened audio device \n");
        /* Set up GPIO expansion header*/
        IOWR_ALTERA_AVALON_PIO_DIRECTION(GPIO_EXPANSION_BASE,0);//set
pins to input

IOWR_ALTERA_AVALON_PIO_IRQ_MASK(GPIO_EXPANSION_BASE,0xFFF);//set 12
pins for interrupts
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(GPIO_EXPANSION_BASE,0);//clear
edge capture
        /*disable read and write audio interrupts*/
        alt_up_audio_reset_audio_core(audio_dev);//clear fifo
        alt_up_audio_disable_write_interrupt(audio_dev);
        alt_up_audio_disable_read_interrupt(audio_dev);
        /* Register the ISRs */
        alt_irq_register( AUDIOCORE_IRQ, sinArray_ptr, handle_audio_interrupts);
        alt_irq_register( GPIO_EXPANSION_IRQ, last_on, handle_gpio_interrupts);
        /*Clear Keys Edgecapture*/
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0);
        IORD_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE);
        /*Enable write interrupt for audio*/
```

```
        //alt_up_audio_enable_write_interrupt(audio_dev);
        //alt_up_audio_enable_read_interrupt(audio_dev);
        return;
}
```