Name: Sudharshan (A0152230Y) (Wednesday session)

EE2020 Module Project: Synthesia

This report outlines the feature-set and implementation of my EE2020 final project. In this project, we were required to implement an audio synthesizer with a certain compulsory feature set. My final product, Synthesia, provides a whole range of features along with the ability of control over a custom coded app. In the following sections, I will be outlining the full feature set of my project and explaining a few of the implementation methods.

Features

The core of my implementation can be divided into 2 main subunits. The first subunit covers real-time audio manipulation which uses the attached MIC unit as an audio source. In contrast, in the second subunit, the FPGA itself is the audio source and serves the purpose of an audio synthesizer. We will first look at the feature set of the first subunit.

- Audio pass through Mode (Audio from MIC is set to DAC)
- Fixed length delay (Audio from MIC is sent to DAC with a one second delay)
- Pitch shifter (Pitch shifts input audio up or down)

The second subunit allows for the user of the product to generate wave patterns of piano note frequencies (ie C,D,E,F,G). It also allows for several configuration options which can be said to be "features". These configuration options are shown below.

- Octave shift (Shifts octave of notes up or down)
- Note select (Allows user to select which key is pressed)

The last feature we will be covering is the serial receiver feature. This feature allows the user to configure the outputs of the synthesizer through a serial interface with a computer. This serial interface is utilized by my application to convert the users input to a serialized instruction format before sending it to the FPGA to set its output. The features of that serial interface are outlined below.

- Serial decoder module (Decodes UART signals through the onboard USB-UART converter)
- Serial instruction decoder (Decodes obtained 1 byte input for configuration)

Lastly, a unique feature of this product is that it comes with an application that can interface with this serial communication mechanism. The features of the application are outlined below.

- Built with Electron for cross platform support
- Built with HTML, CSS and Javascript
- Based on Node.js and npm modules
- Uses serialport.js for interface support

Feature table

Many of the input triggers below rely on the use of the serial instruction. However, it is hard to capture a comprehensive understanding of the instruction format through the table below. Therefore, please refer to the instruction format in the later section on the instruction decoder.

No.	Feature	Input trigger	Description
1	Disable all outputs	0b00 input through serial_in [7:6]	Disables all audio output sources
2	Audio pass-through	0b01 input through serial_in [7:6]	Output set to input from MIC_In
3	Delay Mode	0b100 input through serial_in [7:5]	Enables the 1 second delayed pass-through mode
4	Pitch Mode	0b101 input through serial_in [7:5]	Enables the pitch shift mode. Further params required below
5	Pitch Mode (Filter Enable)	0b1 input through serial_in [4]	Enables the output FIR filter when pitch mode is enabled
6	Pitch Mode (Set delta factor)	Input through serial_in [3:0]	Accepts a 4 bit increment value in 2.2 fixed point format.
7	Instrument Mode	0b11 input through serial_in [7:6]	Enables the instrument mode. Further params required below.
8	Instrument Mode (Octave value)	Input through serial_in [5:3]	Sets the octave shift of the instrument output. MSB indicates shift up/shift down.
9	Instrument Mode (Note value)	Input through serial_in [2:0]	Sets the output note value for the instrument

Now that we have gone over all the features of the product, we will now look into implementation of the key features that I have included.

Implementation of Pitch Mode

As seen in the feature table, the pitch mode feature takes in a few parameters, namely the **filter enable** and **delta factor** parameters. In this section, we will look at the pitch mode implementation and how these factors come into play.

Pitch buffer system

The core of the pitch shift mode is the pitch buffer. The pitch buffer is a FIFO 1024 bit long buffer system with a read and write pointer. The pitch shifting mechanism works as follows.

- The data passed into the buffer is stored in the FIFO buffer at the write pointer location which is then incremented by 1.
- When there is a read operation, data is read out from the read pointer before the read pointer is incremented by the **delta_factor** parameter.
- Given that the read pointer is in a fixed point 10.2 representation and that the **delta_factor** parameter is in a 2.2 fixed point representation it is possible to increment the read pointer by a value range of (3, 0.25). This is what enables both upwards and downwards pitch shifting.

Through the use of fixed point representations, the pitch buffer is able to increase and decrease the sampling frequency of the input signal. This cases the pitch shift effect that is observed.

Double buffer system

After the implementation of the pitch buffer before, it became obvious that there was a lot of high frequency noise present especially when pitch shifting up which cases oversampling. To solve this, 2 pitch buffers were used where each pitch buffer was offset by half to each other. Then, the average of the 2 buffers was treated as the output signal.

FIR filter

Despite the double buffer system outlined above, high frequency noise was still present in the output signal. To further decrease the high frequency noise in the output signal, a Fixed Impulse Response (FIR) filter was used to create a low pass filter with a cutoff frequency of 3500Hz. This was implemented through the use of the existing IP FIR filter blocks within Vivado.[1] The filter coefficients were calculated based on an online calculator tool. [2] The **filter enable** parameter enables and disables the output filter as the user desires. However, no extensive testing was done to confirm the effectiveness of the filter.

Together, the FIR filter, and the 2 pitch buffers form the basis of the pitch mode feature.

<u>Implementation of the Instrument Mode</u>

From the feature table, it can also be seen that the instrument mode takes 2 parameters, **octave value** and **note value**. In this section, we will look at the implementation of the instrument mode on the product.

Sine Lookup Table (LUT)

As I was designing the instrument mode, I had decided that I wanted my product to output proper sine waves instead of square waves. As such, I needed a way to generate a sine wave from an FPGA. The simplest implementation I could think of was to use a 4096 bit deep lookup table populated with 11-bit wide values which approximated the sine function. Using Xilinx's Distributed Memory function, I initialized 4096 bit ROM and populated it with generated sine values.[3]

Generating different notes with the LUT
Now, we need to use the same LUT to generate different pitch notes. This can be accomplished with a similar approach to that seen above with the pitch shifter. In this implementation, 12.20 fixed point representation was used for both the read pointer and the increment values. Because we know the number of values in ROM and the clock frequency, it is trivial to calculate the increment values for each note value. These values can then be stored in registers. The **note value** parameter is then used to determine which increment value should be used for a particular note.

Octave shift
The octave shift feature is achieved through the use of a shift operator. By shifting the increment values to the left, we are able to move up one octave. By shifting the increment values to the right, we can move down one octave. So the MSB of **octave value** sets whether it is a shift left or shift right, and the remaining 2 bits set the amount of shift.

Implementation of serial decoder

The hardest part of the project to design was the serial decoder. The serial decoder comes along with many problems such as crossing clock domains and requires things like synchronizers to avoid problems. In addition, multiple ticks needed to be generated to drive the sampling at the correct baud rate and ensure that the start bit of the serial protocol is detected. The general outline of the logic flow is outlined below.

1) Oversample the data line at 8 times the baudrate (115200*8)
2) Use D flip flops to synchronize the data to the clock
3) Filter the data input and check for start bit
4) After start bit is detected, use state machine to read the data values at another tick frequency which is exactly the baud rate. (115200)
5) Put the read data in the shift register and then assert the data ready after all 8 bits are shifted in

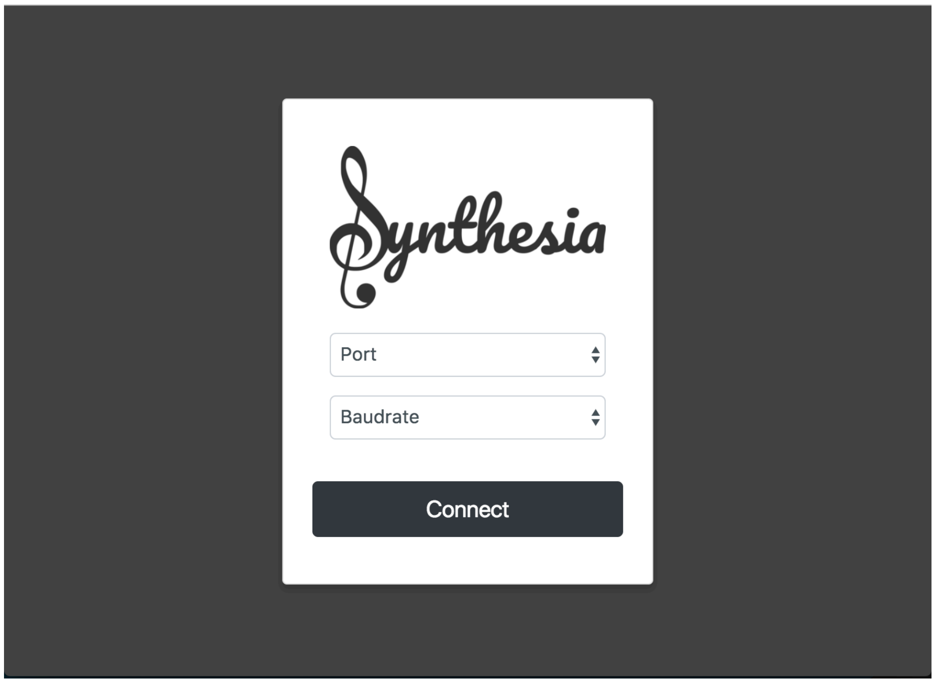
Instruction Decoder

After the byte is read in using the serial decoder, the 8-bit instruction needs to be decoded and set to the corresponding modules. As mentioned in the feature table, each feature requires a specific operational code (opcode) to trigger and has parameters such as filter enable, delta value, octave value and note value. We need to slice this data out of the instruction and assign it to the corresponding buses. This process is done by the instruction decoder according to the instruction format outlined below.

Instruction Type	7	6	5	4	3	2	1	0
Disable	0	0	X	X	X	X	X	X
Passthrough	0	1	X	X	X	X	X	X
Voice Delay	1	0	0	X	X	X	X	X
Voice Pitch	1	0	1	filt_en	delta_val[3:0]			
Instrument Mode	1	1	oct_val [2:0]			note_sel [2:0]		

Synthesia App

Lastly, let's look at the application that I programmed to act as a controller for the project. The project is coded in Electron using HTML, CSS and Javascript. Electron is the ideal framework to use for this project as it is crossplatform and can be coded using familiar web languages. Specifically, Node.js and npm modules such as serialport.js were used. Additionally, I had conformed to ES6 (EcmaScript 6) conventions while coding this application. A screenshot of the application is attached below.



Feedback section

I feel that this entire project has been an absolute blast and I had a lot of fun designing this synthesizer. The part I loved the most about this project is probably figuring out what features I can add to the system and the worst part is realizing that I don't have enough time to implement all the features I wanted to. For example, I wanted to add volume control as a feature along with automatic normalization of sound. I think that the project assignment is pretty good as it is and the only improvement that I would like would be to have even more of an open ended question.

References

[1] - https://www.xilinx.com/products/intellectual-property/fir_compiler.html
[2] - http://t-filter.engineerjs.com
[3] - https://www.xilinx.com/products/intellectual-property/dist_mem_gen.html