**Jason Arkin, Ian Feeney, Sebastian Rodriguez, Spencer Williams, Sebastian Witkowski**

**EECS 392: VLSI / FPGA System Design Projects - JS3Inth**

**Instructor: David C. Zaretsky, PhD**

**06/12/2015**

Table of Contents

[**Executive Summary** 3](#_Toc421875726)

[**Body** 4](#_Toc421875727)

[*Introduction* 4](#_Toc421875728)

[*Broader Considerations* 4](#_Toc421875729)

[*Design Constraints and Requirements* 4](#_Toc421875730)

[*Design Description* 6](#_Toc421875731)

[*Design Optimization* 8](#_Toc421875732)

[*Testing / Simulation* 9](#_Toc421875733)

[*Implementation / Synthesis* 9](#_Toc421875734)

[*Conclusion* 10](#_Toc421875735)

[**References** 11](#_Toc421875736)

[**Appendices** 12](#_Toc421875737)

# **Executive Summary**

For our undergraduate capstone project, we decided to undertake audio wave modulation using hardware through a Field Programmable Gate Array, specifically the Cyclone IV mounted on the Altera DE2 educational board. The project we proposed and then later decided to make was a 16-key synthesizer with multiple synth sound modulations, a graphical user interface which would be output through a VGA port, and multi-state volume control. We decided that the user should also be able to cycle through several octaves worth of notes and if desired, to have the option to mute the sound output temporarily on command. We were able to successfully have the project fully completed on time except for our proposed synth modulations which ended up not producing the desired effect in practice.

The benefits of implementing a synthesizer through hardware description languages such as VHDL and Verilog are that one is given total control on how signals are to be processed, allowing the programmer to alter the audio and video output as much as desired in order to achieve a desired timbre of sound and neatness of a GUI. However, the major limitation of using hardware to complete this project was that one very akin to using IA32 assembly when a high-level language could be used instead – even though you are given total control, it is much more difficult and time consuming to do something which would be typically done in software completely in hardware.

# **Body**

## *Introduction*

Our goal for this design project was to build JS3Inth, a 16-key synthesizer with multiple synth modulations and multi-state volume control implemented in real-time. It will be implemented on the Cyclone IV FPGA, and each switch, from left to right, will be mapped to a particular frequency corresponding to notes on a traditional keyboard, expanding through an octave and a half. We plan to implement JS3Inth by creating VHDL and Verilog entities that will take the inputs of which keys are turned on, get the corresponding notes from a ROM, add up the notes, amplify the result based on the current volume level, output this resultant sound to the speakers, and display information about the current state on the monitor.

## *Broader Considerations*

While there may not be any real use of our project outside of the lab as synthesizers are widely available in the market, we have learned many valuable things throughout the completion of this project which we would otherwise never have come across. One such example is learning exactly how to process audio via hardware and also how to go about setting up communications between various chips on board using the I2C standard protocol and also for those of us who had no training in FPGA implementation and for those of us that did, the mastery we have achieved in using an FPGA is an extremely valuable skill set which could have only been obtained through completing a project such as this one.

## *Design Constraints and Requirements*

The user should be able to cycle through several ranges of frequencies which represent octaves in a traditional keyboard. We will also allow the user to switch between multiple forms of sound modulation through a button input that will cycle through various programmed sound textures. We will also include a mute button that silences/un-silences the system. We will be also implementing a graphical user interface (GUI) which will be outputted through a video-graphics array (VGA), showing what keys are turned on, what volume level is being used, whether mute is on or off, and which synth modulation is being used. The only components/peripherals will be the Cyclone IV FPGA, the monitor, and the speakers.

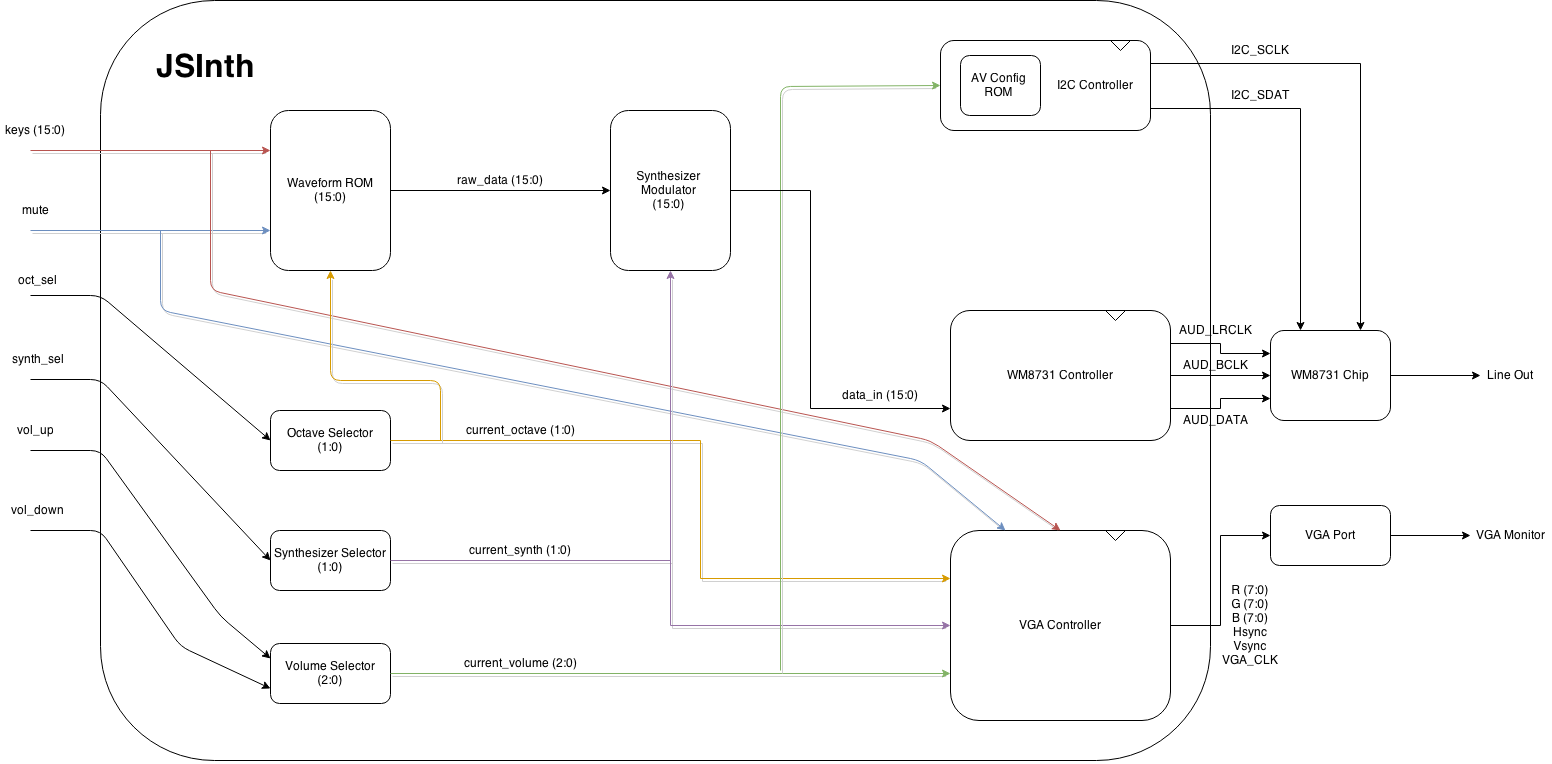


Figure A. Top-level Block Diagram

## *Design Description*

All inputs and outputs are sent through FIFO buffers to preserve the integrity of the signals. Initially, we have 16 switch inputs which are connected to a read-only memory (ROM). Each switch input will correspond to a point in memory which will then output a 16-bit sample to a frequency adder. Additionally, we will be multiplying the output of the frequency adder by a certain amount determined by the state of the octave state machine.

A finite state machine is implemented to determine the current synthesizer being used. Once a certain synth has been decided, the correct function inputs are pulled from the ROM and put into the synth modulation block, which combines with the frequency adder outputs to create a modified audio wave.

The volume is controlled through a finite state machine that cycles through 5 levels, these levels determine the maximum amplitude of the output signal. The output signal of the synth modulator is multiplied to increase the amplitude until it hits the correct volume. This signal is sent to the WM8731 controller to be serialized.

The I2C controller controls the setup of the WM8731 chip using the standard I2C protocol. The I2C protocol works on a master-slave system using 2 data lines: Serial Clock and Serial Data. First, the master (In this case, the DE2-115 FPGA) pulls Serial Data low while Serial Clock remains high to initialize a transfer. The master then sends a signal of 8 bits on the data pin while cycling the clock. Then, the master clock cycles once and waits for acknowledgement of low from the slave, which will attempt to pull the Serial Data line low. After this acknowledgement bit, the master system will continue to shift in 8 bits on the Serial Data line and waiting for an acknowledgement bit from the slave system. Finally, the master will end a transmission by pulling Serial Data from low to high while the Serial Clock is high. In the case of the WM8731, we send a total of 3 bytes. The first is the address byte, which is standard across all I2C systems. The second byte is the register address byte to tell the WM8731 which register to get ready to fill. The final byte tells the WM8731 what to fill the register with. In our case, the address byte for our chip is 0x34. Below is a table of the registers of the WM8731 and how we initialize them. The major things are that we set the sampling frequency to 48 kHz, initialize the DAC, and set the input sample length to 16 bits. For more information please refer to the WM8731 datasheet. See Appendix A for the I2C data passed into the WM8731.

The WM8731 controller prepares the data to be sent to the WM8731 on-board chip. It takes the master clock and divides it into 3 separate clocks: the audio master clock (rated at 12.5 megahertz (MHz)), the left-right clock, and the bit clock. The sampling frequency, set at 48 kilohertz (kHz), is determined by our left-right clock, or the amount of times a sample is processed by the audio chip per second. We constructed a parallel-to-serial converter to send the data one bit at a time to the audio chip. The audio chip takes the data, left-right clock and bit clock, and passes it through its Digital-to-Analog Converter (DAC), which connects to the Line Out component in the board, giving us sound.

All inputs are also passed through our VGA module. Our VGA controller contains rendering logic which draws colors and shapes on screen through a scanline, in which the synchronization module keeps track of the vertical and horizontal sync. The rendering logic checks every pixel the scanline is currently rendering, and draws a color correspondingly. Our controller reads the inputs and displays a simple user interface to help the user keep track of any of the keys, volumes, and octaves running in the system.

In order to produce a wave in ROM, we determine the wanted frequency and we extract the period in milliseconds. Due to the sampling rate, we sample 2 bytes of data every 30.82 microseconds. We create each angle in the wave by adding the cell’s value by 360, dividing by the number of total samples. Then we calculate the sine of the angle and we multiply by 32767 (28). Converting these values to hexadecimal gives us the final value to be placed in our look-up table. See Appendix A for an example of values for the tone E5 with a frequency of 660 Hz.

The two synth modules we decided to try and implement in hardware were two effects commonly used in all sorts of sound processing, the reverb and flanger effects. The way we theorized the reverb could be implemented was through sending the 16 bit unsigned audio vector through a FIFO of a set size, then after the samples get pushed out of the FIFO one by one, they get added to the source signal corresponding to the current clock number. The way we theorized to implement the flanger to was though the usage of 2 2-to-1 multiplexers, one of which would be controlled through a finite state machine and would select in between the source signal and the output of the mux and the other through a bit which would allow either the processed signal or the source signal through to the audio codec on board.

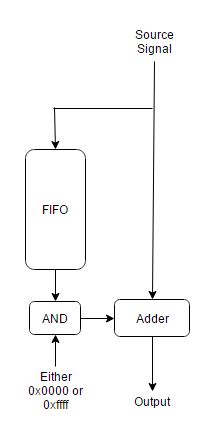
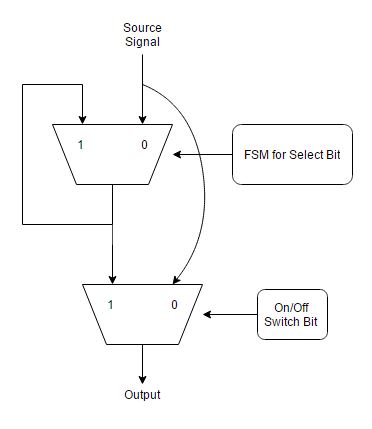


Figure B. Flanger Block Diagram Figure C. Reverb Block Diagram

## *Design Optimization*

After implementing the ROM and sample adder, we attempted to combine the waves by adding every available sample and dividing it by the number of waves available. We erroneously concluded that the easiest way to add the waves was to cycle over the amount of samples that the wave with the lowest frequency had. Cycling over these samples gave us a sound that was fundamentally the wave with the lowest frequency, drowning away the higher frequency tones. In order to produce a cleaner sound, we replaced the cycling with streaming architecture that would allow every wave to be available to the sample adder at any point in time. After implementing it, the quality of the sound increased; we were able to differentiate between multiple tones being played simultaneously.

Fortunately, this revealed a serious flaw in our calculations that needed to be fixed in order to ensure the correct sound was passed. Adding the samples in a streaming fashion revealed that certain samples were overflowing, thus distorting the wave. This distortion created sharp edges in the wave that were noticeable enough to change the frequency, ultimately changing the tone we were aiming for. Revising our sample generator, we had added an extra value of 0x0001, making values such as 0x7FFF go to 0x8000, and since the wave samples are treated as signed values, we shifted samples from the negative domain to the positive domain, resulting in the sharp increase of certain samples. Once this was rectified, the wave returned to being fully sinusoidal.

## *Testing / Simulation*

Modules in our design were tested in ModelSim to ensure functionality. The I2C module could not be simulated in ModelSim due to the need of the acknowledgement from the slave system (in this case the WM8731 chip). The finite state machines were tested by cycling through their states and ensuring they are encoded correctly and changing their outputs as appropriate. The VGA module was simulated to test the correct color output at a certain point in screen, but checking every single color is a tedious challenge, and many of the values remain the same because we are outputting blocks of color, so instead we synthesized the module into the board to ensure the rendering logic is working correctly.

We simulated the WM8731 controller to ensure the module is outputting every signal wired to the WM8731 chip correctly. We simulated the bit clock to make sure the serializer worked correctly, the left-right clock to ensure the sampling rate met the correct constraint, and most importantly, we simulated our 16-bit data input to determine the correct wave is being sent correctly.

The biggest hurdle we encountered during development of the prototype of the JS3Inth was figuring out how to get a digital audio signal out of the Altera DE2-115 successfully. This proved to be much more tedious than what we were originally anticipating since we were not sure on where to send our standard 16 bit audio signal in order to have it recognized and decoded by the on-board WM8731 audio codec properly. After scouring the DE2 user manual on how to set up the signal in the necessary serialized format, we eventually had to spend several days studying the datasheet for the Wolfson codec to learn how to communicate digitally with the chip as to set it up to receive and correctly understand the signal being sent to it through the I2C protocol.

## *Implementation / Synthesis*

All of our VHDL code was synthesized using Quartus 14.1 (64-bit) from Altera. We took advantage of the available block RAMs in the Cyclone IV FPGA (435 of them) to synthesize block ROMs and store our wave samples in them. The top-level entity was synthetized using structural VHDL to instantiate all components, while the output towards the WM8731 was done with a single line of dataflow VHDL.

Most of the lower entity components use processes to maintain operation through each clock cycle. The total number of logic elements synthesized was 6,782 out of the maximum of 114,480, a 6 percent use. Registers synthesized for this purpose amounted to 531.

The interesting portion is that synthesizing 41 ROMs only took 2,288 memory bits out of the available 3,981,312, less than 1 percent usage. We believe the Cyclone IV FPGA has its own way to pool values such that repeated values do not take extra space. Ultimately, we used 65 out of the 529 pins in the Altera DE2-115 board, which is reasonable for an application that uses most of the switches, VGA and the WM8731 pins.

## *Conclusion*

The design fully met our requirements. From the start we aimed to create a sound generator in an FPGA using multiple block ROMs, modulate these samples using VHDL, and pass the samples through a Digital-to-Analog Converter (DAC) to produce our final sound, along with a display module through VGA that would allow a user to receive immediate feedback from the system.

For further development, we could attempt to synthesize more modulators to allow the user for more variety between sounds. The tone ROMs can hold more natural sounding waves instead of purely sinusoidal waves. Ultimately, to incorporate user input, the Altera DE2-115 board has a Line In pin that allows the user to use the Analog-to-Digital Converter (DAC) in the WM8731 to convert a natural wave into a 16-bit output to the FPGA. This converted wave can be modulated instead of the waves stored in ROM.

# **References**

# **Appendices**

|  |  |
| --- | --- |
| **Register** | **Data** |
| 00 | 1A |
| 02 | 1A |
| 04 | 7B |
| 06 | 7B |
| 08 | F8 |
| 0A | 06 |
| 0C | 00 |
| 0E | 01 |
| 10 | 02 |
| 12 | 01 |

