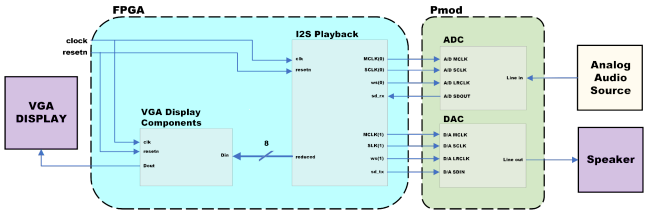
FPGA Music Visualizer

Benjamin Rojewski, Nicholas Spanos, Justin Janulewicz & Andrew Waite
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: brojewski@oakland.edu, nspanos@oakland.edu, jjjanulewicz@oakland.edu, agwaite@oakland.edu

***Abstract* — The implementation of a music intensity visualizer on the Artix-A7-100T board as a demonstration of the knowledge gained in this course.**

I. Introduction

The device we intend to make is an audio visualizer. What it will do is take the incoming music from the Pmod port, represented as a voltage, and convert it to digital data using an ADC and then display this data onto a VGA display. The inspiration behind this project is that we all really like music, and sometimes like to look at visualizers for them. The audio visualizer will be able to be shown on a display while the music is playing through a speaker or headphones. This project consists of VHDL code, data capture using an I2S2 Pmod component, data processing with registers, decoders, mux, and finally a display control with VGA drivers.



II. Methodology

A. Data Capture

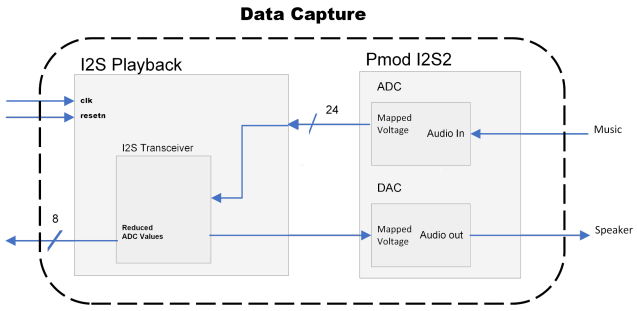
To start, our project uses a pmod to take in an audio input from an analog signal which is then fed into a ADC to transfer it into a 24-bit digital signal using the I2S protocol that is used in our project. Because the max voltage from the Nexys a7-100T pmod ports are 3.3V, and the ADC converts to a 24 bit signal, the step size is about 197 nV(1) which is much smaller than this project requires.

$$Equation\ 1:\ Step\ Size\ =\ \frac{3.3\ Volts}{2^{24}} \simeq\ 197\ nV$$

Because this step size is smaller than we needed, we decided to truncate this 24-bit signal down to a total of 8 bits before passing it to the rest of the data processing circuit. An 8 bit signal from a 3.3V source gives us a much larger step size of about 13mV.(2)

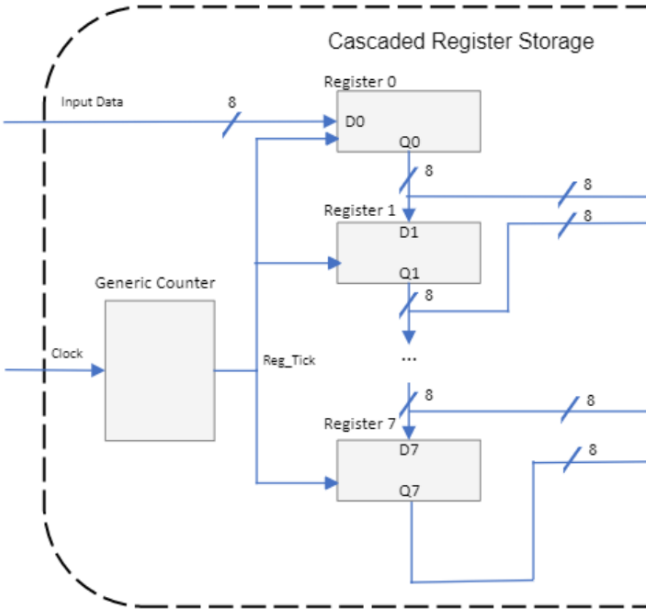
$$Equation\ 2:\ Step\ Size\ =\ \frac{3.3\ Volts}{2^{8}} \simeq\ 13mV$$

After we cut the 24-bit signal down to 8 bits, we decided to cut down the number of steps we displayed to be even lower. We changed the number of steps to a mere 8 by selecting only the most significant 1 in the 8-bit signal and making every bit lower than it a '1' as well. For example, if the 8-bit signal was "00100111" then we only collected the 8 bit value of "00111111".

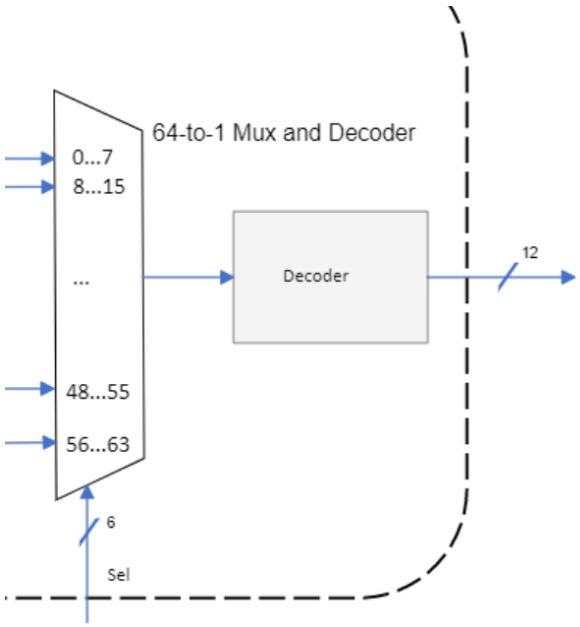


B. Data Processing

After receiving the data from the I2S2 Pmod, we were able to process it within the board and set it up so that it can be displayed through the VGA. To do so, the data processing section incorporates four main components:a series of 8 cascaded registers, a counter to control when registers shift the data down the stack of registers, one 64-to-1 mux used to select what data is output, and a decoder to sign extend the output data bit. First the data is passed from the Pmod input into the first register. There are a total of 8 registers, one for each column to display in the VGA output.

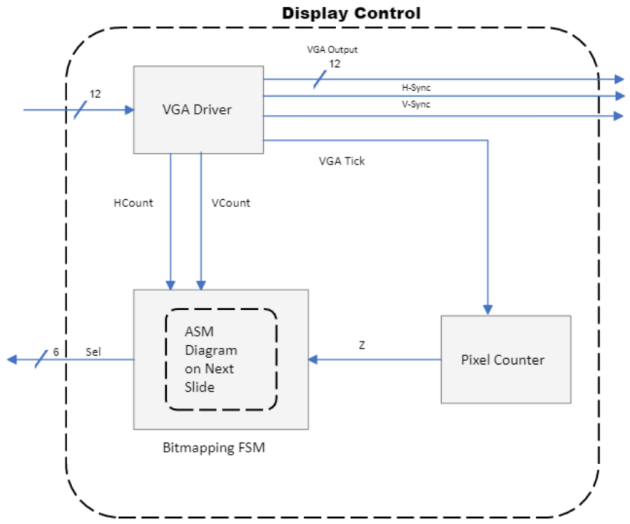


Implemented into the top file of the Data Processing portion of the music visualizer is a counter, with an output signal called “REGE_TICK”. This signal controls when the data is passed from one register into the next. This counter counts from an arbitrary value that we thought looked good, as the first priority of a music visualizer is to present the music signal in an attractive way. From the cascading register, the data signals are attached to a 64-to-1 MUX.

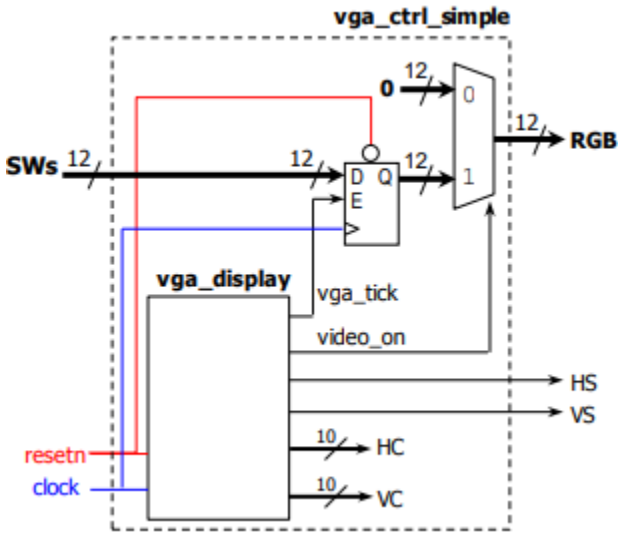


This, in conjunction with the select signal given by the BITMAP_FSM, is so that we are able to choose specifically which part of the bitmap stored in the registers needs to be shown at which location and moment for the VGA display. To facilitate this, we attached a decoder to the end of the mux that functions to sign extend the bit given to it into 12 bits. This is necessary as we have initialized the simple VGA controller to use an input of 12 bits for RGB control.

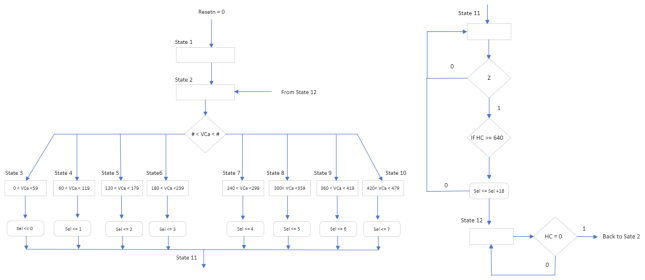
C. Display Control



The heart of the VGA signal output is a simple VGA controller provided by Prof. Llamocca. Its original design was to change the entire screen's color by using 12 switches as the input. With this, in combination with an FSM to generate different necessary signals like VS and HS, it is able to output 12-bit RGB values to the screen with control as to where pixels are drawn. We went ahead and modified this to simply take the decoded (sign-extended) output of the datapath, and draw the result on the screen.



In order to control this, we need an FSM to dictate when and where the registers are accessed. A critical part of the Display control is the bitmap_FSM. This FSM controls which register is selected at any given time, and which specific value is selected from this register.



This is important, as in order to draw and extend the signals into the 8x8 grid of the screen, we need precise control of what is being displayed. The FSM functions by first separating the screen into 8 equal rows, 60 pixels tall each. When it does this, it assigns an initial offset between 0 and 7 to the select signal which uses the mux to output from any value in any register. It then outputs this value for 80 pixels, using an external counter tied to VGA_TICK which determines the length of each square in the grid. After 80 pixels the counter flips signal Z high for a clock cycle, which then triggers the FSM to select the next signal to draw. In order to select from the next register, we add an offset of 18 to the select line every time the counter is finished drawing 80 pixels (. Using this, we were able to create a grid of 8x8 60 by 80-pixel squares on a VGA screen, with each being able to be created from a bitmap created in a series of cascaded registers. The reason for the 18 offset is because we subjectively chose REGE_TICK to have priority over timing, in order to create a more eye-catching visualizer. Because of this, the speed that the register data is presented was chosen to look appealing, then the offset was tuned until it got the best-looking output, which happened to be 18.

III. Experimental Setup

The experimental setup for this project was simple and to the point; we incorporated both hardware and software solutions to create this project. The main items used were: the Artix A7-100T, Vivado VHDL, Youtube, I2S Pmod Audio Headers, AUX connector, VGA connector, speaker, monitor, and an audio generating device (phone, Laptop, etc).

The hardware was connected as follows: the Phone or laptop was used to generate the audio input. This input was transferred from the device to the board via an aux connector. In order to receive the data, the Pmod audio header was placed onto the board where it functioned as a bridge between the hardware and software portion of our project. Once the audio was routed through the connector into the board, we utilized Vivado VHDL to manipulate the data into a format usable to display. After doing so, we sent the audio to a display connected via VGA while simultaneously sending it back through the Pmod header to a speaker. This meant our viewers could both see and hear the music while it was being played.

The VHDL code used can be split up into two main sections: code created and code borrowed from other sources. The portions borrowed are the Pmod I2S2 code [1] and a portion of the simple VGA driver [2]. The former was used because it was created specifically for the I2S Pmod Header and the latter was used to establish a starting point for the VGA driver used in the project. The I2S code was modified slightly to fit our requirements and so was the VGA code, as mentioned in the previous sections. The rest of the code was created from scratch to fit the project requirements and once pieced together, Vivados simulation software was used to debug the code and correct errors. Once the kinks were worked out, the program was uploaded to the board and then connected to the VGA port on the monitor via VGA cable. Once connected, the VGA window will pop up and the program is ready to begin.

IV. Results

Once the project is set up in accordance with the experimental setup, the user must start the music on the input device and then adjust the volume - as the volume increases, the bars grow vertically and the inverse happens when lowering the volume. This behavior is anticipated and what we had expected to create when setting out. This does not mean we didn't have any issues while creating our project, however.

There were a handful of issues we encountered. The first, and arguably most important, had to do with the I2S playback code we used in the data capture portion of the project. Here, we encountered two things: first, the code was created for a different board, and second, an internal clock wiz IP used in the code had to be changed. The former was a trivial fix, but the latter took a bit more problem-solving to figure out. Essentially, the IP was outdated (the code is from 2011) and this was causing issues as it was not compatible with our version of Vivado. Once we figured this out and were able to update it, the I2S code ran perfectly.

The next issues were inside our data processing and display portions. The 64-to-1 bus mux used is passed a select line from the bit mapping FSM. during testing, we had originally set the select line to have an offset of 8, but

this did not yield proper results as it instead was grabbing data that did not resemble what we had expected. Once we saw this, we tried out different integer offsets and eventually found 18 to work flawlessly. The registers connected to the mux also had an issue associated with them, specifically their clock frequency. They were originally hooked up directly to the 100MHz clock in the board, but this caused the outputs to be sent at inappropriate times and caused the data to be a scrambled mess. To fix this, we used a generic gen_pulse component and set the clock to match up with the one being used by the FSM.

Once we had fixed those problems, there was one interesting thing we found when playing certain tones through the board. For example, if we played a 500 Hz frequency tone, the device did not necessarily display the intensity of the sound. Instead, it seemed to display the physical wave itself, almost as if the device were functioning as an oscilloscope. What we think caused this was the sampling rate of our Pmod Audio Headers - for some reason, it seemed to sync up and produce the periodic signal we witnessed.

Conclusions

Overall, this project was extremely satisfying. To see our project come together and look almost exactly how we planned it was amazing and we learned some valuable lessons while we were at it. Firstly, we realized the importance of choosing a realistic scope for our project. Originally we wanted to have the ability to show both the loudness of the song(Voltage) and the frequency of the sound depending on the position of a switch. We quickly learned that with the amount of time and resources we had would not be capable of producing a circuit to determine the frequency of a song. Had we chosen to implement frequency toggling into our project, we most likely would not have been able to finish on time.

Additionally, we learned how to rely on our groupmates to get the best results out of our project. All of us realized that none of us would be able to get this project done on our own so we made sure to use many different communication methods to figure out who was doing what part of the project. We also solidified the idea that it is extremely important to make sure you are communicating all the way through each step of a project, especially since we basically made two different parts of the project that we needed to make sure would seamlessly interact with each other. It is important to make sure each group member knows about each and every change that is made to the plan and understands how it affects their individual parts.

REFERENCES

[1]
S. Scott_1767, “I2s Pmod Quick Start (VHDL),” *DigiKey*, 26-Mar-2021. [Online]. Available: https://forum.digikey.com/t/i2s-pmod-quick-start-vhdl/13065. [Accessed: 15-Mar-2022].

[2]
D. Llamooca, “VGA Cotroller,” *SECS Oakland University* .[Online].Available: http://www.secs.oakland.edu/~llamocca/Tutorials/VHDLFPGA/Vivado/Unit_7/VGA_control.pdf. [Accessed: 10-Apr-2022].