Final Project Report

Mitchell Gu & Ryan Berg
December 9, 2015

6.111 Introductory Digital Systems Laboratory
Massachusetts Institute of Technology

Contents

1. Introduction

The Guitar Hero game genre, pioneered at the MIT Media Lab, consists of a series of games where players play along to rock songs with a custom controller designed to imitate an actual guitar. In the game, players press up to six buttons that resemble different frets along with a flip switch that represents the guitar string at the times indicated on the screen to play the song and earn points. While the game is effective for inspiring many players to learn about guitar, we think it is lacking in realism, song flexibility, and education value. Its six controller buttons are a far cry from an actual guitar fretboard which has consequences for the game's song flexibility. The full spectrum of actual chords in a song become two arbitrary buttons played at once, and more involved passages must translate into some combination of the same six buttons. To the user, this is disappointing because they know they are enacting an imitation of how songs are actually played.

Guitar Hero: Fast Fourier Edition is a new take on the genre that enables users to play with an actual electric guitar as the controller. Furthermore, the song instructions are displayed on the screen as scrolling tablature, a format universally available and familiar to players when learning guitar songs. The tablature is both intuitive to read and allows for direct translation of skills learned in-game to the real world.

Instead of expecting some combination of button presses on a traditional controller, the Fast Fourier Edition performs a fast fourier transform of the analog signal from a guitars pickups to determine what notes are currently active. From the FFT results, the game logic awards the player points depending on their pitch and timing accuracy compared to what was supposed to be played. A graphical display of the scrolling tablature and supporting graphics will all be output to a VGA monitor. These features will allow users to play their favorite songs and learn applicable guitar skills along the way.

2. Technical Overview

At a high level, project is functionally divided into two halves: audio processing and game processing (Figure 2.1). Each half is implemented on its own Nexys 4 board, and a two-wire serialization interface is used to communicate between them.

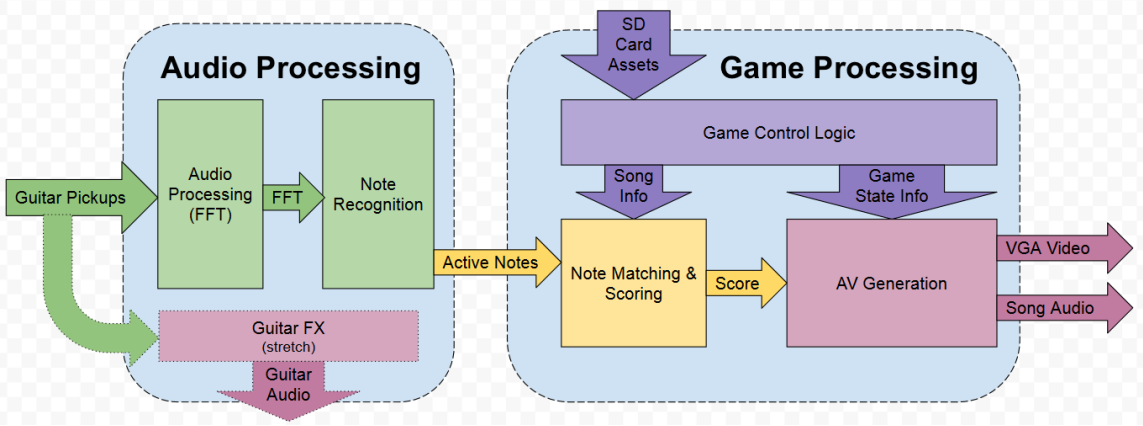


Figure 2.1: The high-level organization of the project

2.1 Audio Processing

The audio processing half is able to take input from a guitar's pickups, perform a fast fourier transform on the digitized guitar samples, and identify currently active notes by computing a correlation index for each of 37 notes in the guitar's range.

Firstly, the analog signal from a guitar's pickups is digitized using the Nexys 4's built-in ADC. These digitized samples are then fed through an oversampling module to decrease the sampling frequency before being streamed into a Fast Fourier Transform module that transforms the signal into the frequency domain.

Once in the frequency domain, it becomes significantly easier to differentiate different notes from each other based on the position of their spectral peaks. Our note detection method is based on a data-driven, correlational approach that calculates correlation indexes between a given frequency spectrum and the reference frequency spectra of each possible note, played correctly. From these correlation indexes, the system determines which notes are active by first running the indices through a low-pass filter, then imposing adjustable hysteresis thresholds on them.

Once new active notes are calculated, system finally serializes the active note data and transmits it over two wires to the game processing Nexys to be matched to actual song notes.

2.2 Game Processing

The game portion takes the active note information from the audio processing portion, and compares it to a song that it knows about to determine how accurately a player is playing the song. It displays what the player should be playing on a VGA monitor, and it also displays a score of how well the player has done so far. It is divided into three major sections, as follows:

The matching and scoring block of the game Nexys receives a serial-encoded list of the active notes currently being played from the audio Nexys and compares them to what notes should be playing currently, as given by the metadata of the current song loaded from the SD card. The scoring block scores the player's performance appropriately and forwards the score and matched note data to the video output.

The AV Generation block takes information about the current score, as well as the state of the game (playing, paused, song over). It combines graphical assets with this information and the song audio to produce a VGA output and an accompanying audio signal.

The Game control logic block is the backbone of the Nexys 4. It loads assets from an SD card, maintains game state, and handles timing of the scoring module.

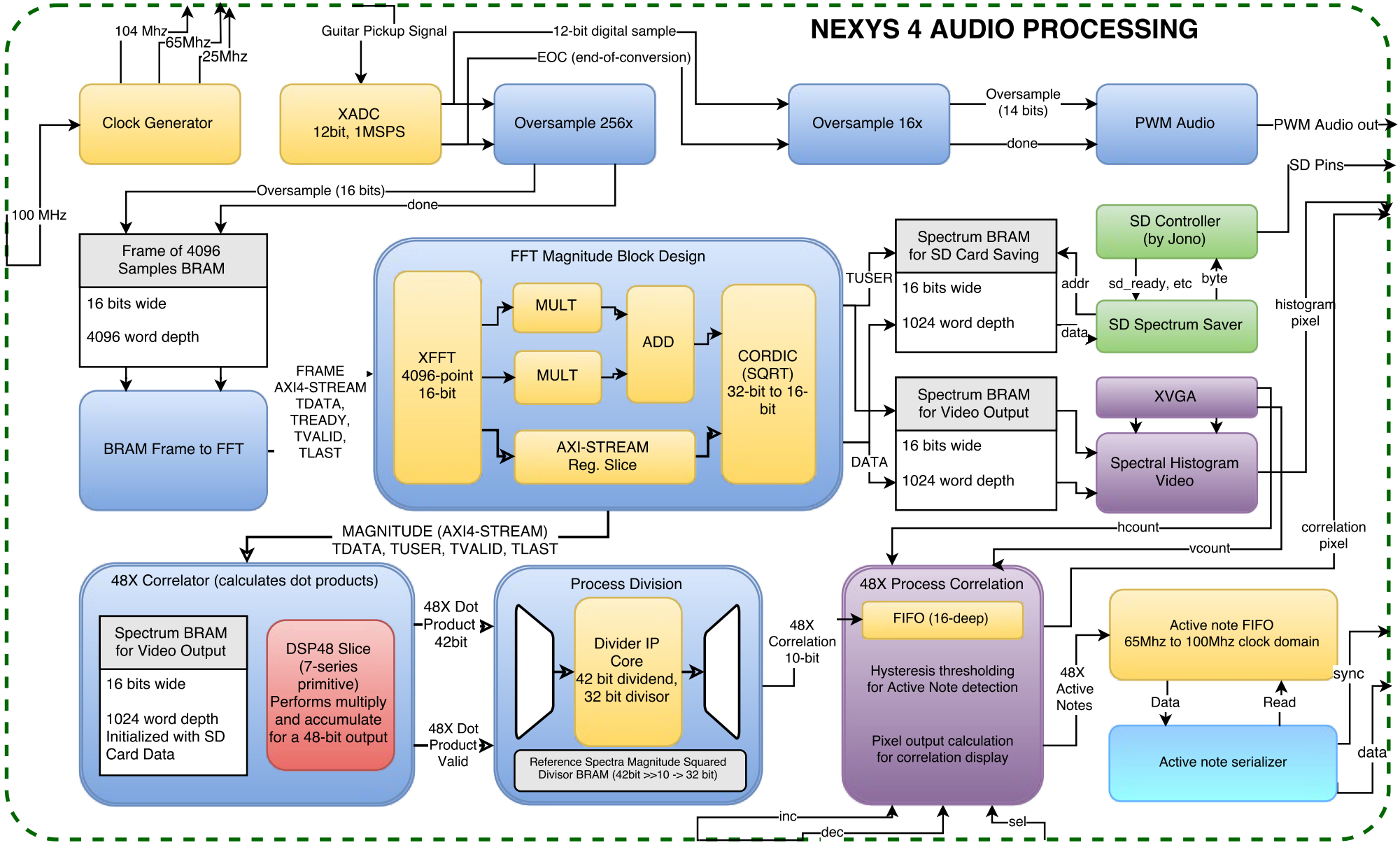
3. Audio Processing Implementation - Mitchell

3.1 Overall system organization

Since the project's initial proposal, there has been a significant overhaul of the audio processing approach. The most significant of these changes was switching the FPGA platform from the 6.111 Labkit that was initially chosen to the Nexys 4. This platform change enabled a lot of improvements to the subsequent modules. The signal chain now includes significant oversampling of the guitar's input signal, an updated and reconfigured XFFT IP core from Xilinx, and a series of newly developed modules that together facilitate the new data-driven, correlation based note recognition.

An updated block diagram for the audio processing system is included on the next page. The sections that follow in this chapter will describe in detail how each part of the design was developed and implemented.

Figure 1: Updated block diagram for the audio processing system



3.2 Switching from the Labkit to the Nexys 4

In our initial proposal, we selected the 6.111 Labkit rather than the Nexys 4 board for audio processing for several reasons. Firstly, the 6.111 Labkit has an integrated AC'97 audio codec that is well suited to digitizing a guitar pickup signal at 18 bit resolution and a 48khz sample rate. Secondly, we already had experience using the AC'97 in Lab 5 of the class, where we recorded, filtered, and down-sampled audio from a microphone input. Finally, we could base our Fourier Transform implementation on the provided Fast Fourier Transform demo running on the Labkit.

However, upon starting work with the Labkit, I increasingly found the decade-old development tools cumbersome. For example, the support for Virtex II FPGA used on the Labkit was abandoned in Xilinx's FFT module generator on ISE (the development environment) 10.1. In order to reconfigure FFT modules for the Labkit, I needed to create them with ISE 8.1, migrate them over to ISE 10.1, and struggle to get the module recognized in my existing project. Not only was the process inconsistent and time-intensive, but I figured if I was going to learn FPGA development for future projects, I wanted to learn with the most up-to-date technology and tools.

The more I looked into the newer Vivado development environment and Nexys 4 platform, the more I realized that the advantages we had seen in the Labkit had more powerful parallels on the newer platform. For audio input, the Nexys 4 had a built-in 12-bit XADC capable of 1 million samples per second. Xilinx's newer Vivado software also offered tight integration of pre-built IP (modules) with the Nexys 4 hardware, including the ADC. The Nexys 4's plentiful digital signal processing (DSP) slices and faster processing speed gave me more freedom in how I implemented the note recognition. Finally, the Vivado software made it easy to string IP modules together, reconfigure them, and synthesize them out-of-context to cut down on compile times.

3.3 Sampling the Guitar input

3.3.1 XADC input and biasing circuit

The integrated ADC on the Xilinx Artix-7 chip (XADC) features two channels, 12 bits of resolution, and a maximum sample rate of 1 million samples per second. To simplify the process of instantiating the XADC primitive and reduce errors, I used Vivado's built in XADC IP core to configure the XADC properly. For the sample rate to reach 1 MSPS, the XADC had to be clocked at 104 Mhz, not the 100MHz clock supplied from the Nexys 4 board. This 104MHz clock was easy to configure from a 100MHz clock input using the Clocking Wizard IP Core, and would be the main clock used for the majority of audio processing. Later on in the project, the same core would be used to generate a 65Mhz clock for the video output and a 25Mhz clock for writing to the SD Card.

On the analog side, the XADC accepts a differential analog signal in the range of 0V to 1V. The pickup signal from a guitar coincidentally has a peak-to-peak voltage of around 1V maximum, so no preamplification was necessary. However, a small biasing circuit was necessary to center the pickup input at around 0.5 volts. This was easily done by using a voltage divider to create a 0.5V line and a coupling capacitor to center the pickup's AC voltage at that 0.5V bias (Figure 3.1) The necessary supply voltages and XADC input pins could all be found on the Nexys 4's XDAC PMOD 12 pin header.

With a 104MHz input clock and a 1MSPS sample rate, the XADC takes 104 clock cycles to complete one analog-to-digital conversion. When a conversion is complete, the XADC asserts its EOC output and makes the sample data available on one of its Dynamic Reconfiguration Port (DRP) registers. The digitized samples were easily read by assigning the DRP address input to point to that register and reading the register data whenever EOC was asserted.

3.3.2 Oversampling the guitar signal

One area of improvement we decided to pursue was adjusting the FFT parameters to best target the guitar's frequency range with improved resolution. The 16,384-point FFT demonstration shown in class had a bandwidth of 24kHz at 3 Hz per

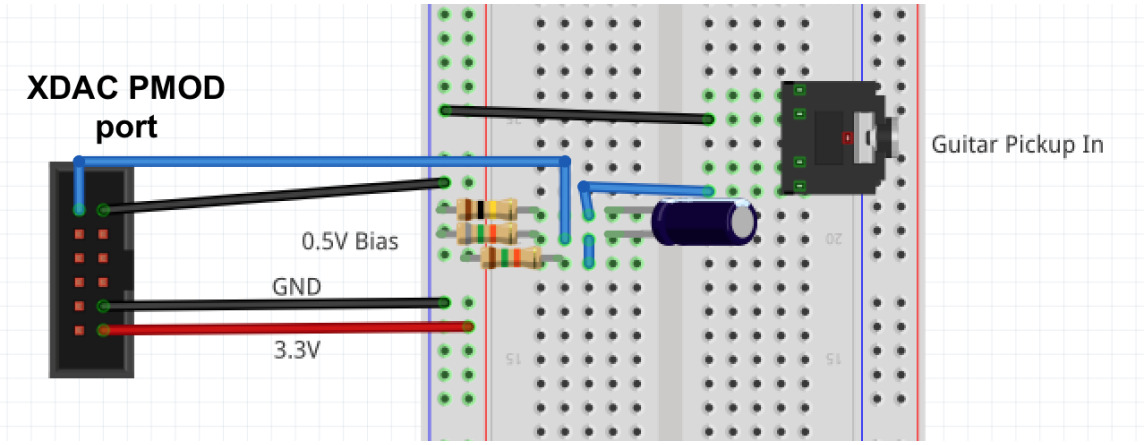


Figure 3.1: The 0.5V pickup biasing circuit

point, whereas the guitar has a (fundamental) frequency range of 65Hz to 1000Hz. In the lower frequency ranges, a half-step difference in frequency is as little as 5 Hz. Thus if we used the FFT as shown in class, only a small range of the output frequencies would be useful for identifying notes and the 3Hz resolution would be inadequate for differentiating between low-frequency guitar notes. For these reasons, I decided to lower the bandwidth of the FFT, which would allow the output to fit the guitar's frequencies better and yield higher frequency resolution.

Due to the principle of Nyquist rates, the sample rate must be decreased in order to decrease the bandwidth of a Fourier transform. At the XADC sample rate of 1MSPS, the bandwidth of an FFT would be 500 KHz, far higher than necessary. I first considered lowering the bandwidth by simply low-pass filtering and then downsampling the XADC samples by a certain ratio, as done in Lab 5A of the course. However, I felt unsatisfied with discarding the majority of the samples and instead looked into oversampling the XADC samples.

In oversampling, the signal is sampled at several times the desired sample rate and averaged across the extra samples to arrive at the oversampled sample. Because of the extra information that contributes to sample, the resolution and signal to noise ratio of the signal can be improved at the expense of sample rate.

I decided to oversample the 1MSPS XADC input at a 256x rate, which increases the SNR by $\sqrt{256} = 16$. Consequently, the samples gain $\log_2 16 = 4$ extra bits of precision (16 bits total) and the sample rate decreases to around 4KHz. At this sample rate, the FFT bandwidth becomes 2KHz, of which I would use the

lower half (0-1KHz) for note recognition. I also settled on an FFT size of 4096 points in order for the output of the FFT to have a 1Hz per bucket resolution, three times greater than that of the demo. This higher resolution would be useful in differentiating low-frequency notes played later on.

Functionally, the oversampling module takes a 12-bits of sample data and the EOC output of the XADC and outputs a 16-bit bus of oversampled data and a done line, which it asserts once every 256 XADC conversions, or once every $256 \times 104 = 26624$ clock cycles.

3.4 The Fast Fourier Transform

3.4.1 Sending samples to the FFT core

Compared to Xilinx's XFFT v3.2 module used in the FFT demo, the latest XFFT 9.0 IP core offered by Xilinx has a lot of changes, all described in detail in its product guide PDF. The most notable of these changes is the shift from native inputs and outputs to the AXI-Stream standard interface. The XFFT IP contains three of these interfaces: One slave interface for sending frames of sample data, one slave interface for re-configuring the core dynamically, and one master interface that outputs the FFT output. At the core of the AXI-Stream interface is a handshake signaling process, in which the master module asserts its `TVALID` line when its data bus has valid data, and the slave module asserts its `TREADY` line when it is able to accept data on the data bus. Because of this two-way handshake signaling, both the master and the slave understand that the data will advance to the next if and only if both `TVALID` and `TREADY` were asserted in a given clock cycle (Figure 3.2).

To send a new sample to the XFFT module, one has to send it the entire frame of samples for every sample at the desired sample rate. Since I chose an FFT size of 4096, at least 4096 of the 26,624 clock cycles in between new oversampled samples would have to be used to send the FFT core a new frame. Additionally, I would need to store the last 4096 16-bit samples of the frame across two 36Kbit BRAM primitives. By this reasoning, the FFT data streaming logic took the form shown in Figure 3.3.

When a new sample is output from the oversampling module, it is written into

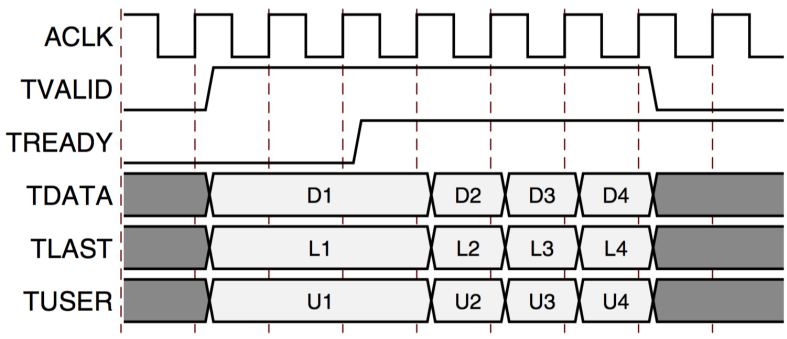


Figure 3.2: A timing diagram of the AXI-Stream handshaking process [1]

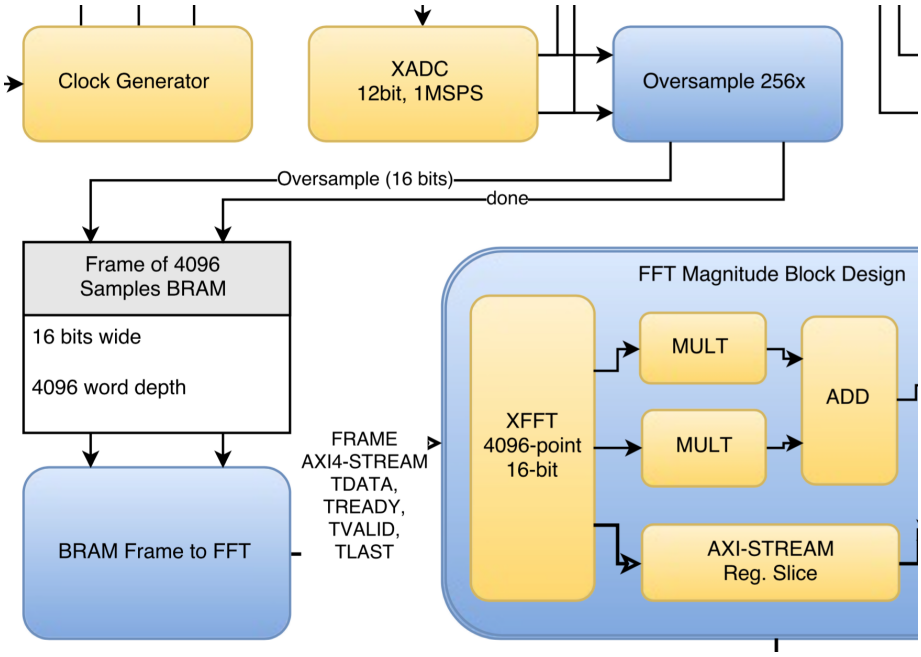


Figure 3.3: The data pathway for sending oversampled samples to the XFFT core

the HEAD address of the frame storage BRAM, and in the following clock cycle, HEAD is incremented (essentially implementing a circular buffer). The done signal from the oversampling module also indicates to the BRAM to FFT module that it can begin sending a new frame to the XFFT IP core via the AXI-Stream protocol. To send a new frame, a state machine in the BRAM to FFT module negotiates the

AXI-Stream protocol by outputting sample data from the frame BRAM (starting at HEAD) and incrementing the address to that BRAM whenever the handshake signaling has succeeded (typically every clock cycle). After 4095 successful data transfers, it also asserts the TLAST wire to indicate that the last sample in the frame is being sent. If the XFFT core ever thinks the TLAST wire wasn't asserted on time (there is a misalignment in frame positions), it asserts a last_missing wire which the BRAM to FFT module uses to realign its frame with the XFFT core.

3.4.2 Customizing an FFT block design

The output (and input) of the XFFT core has both real and imaginary components. In sending samples to the core, only the real component is used, but the output of the core will have both real and imaginary components. For the purposes of note recognition, it is desirable to work with just the magnitude of the FFT output, so a method was needed to convert from real and imaginary components to a magnitude, given by:

$$\sqrt{\text{real component}^2 + \text{imaginary component}^2}$$

My initial instinct was to compute the magnitude was to write all the necessary verilog modules, but this would have proved difficult because the square root module would need to be fully pipelined to be able to handle the fully pipelined output from the XFFT core.

Instead, I discovered how to use Vivado's block design functionality and learned to appreciate the beauty of the AXI-Stream interface standard. In Vivado's Block Design tool, users can use a graphical interface to connect both Xilinx-provided and user-created IP cores into one abstracted block design that can then be instantiated like any other IP core or module (Figure 3.4).

In my FFT Magnitude block design, I was able to connect an XFFT IP core's AXI output to a pair of multipliers that took both the real and imaginary parts and squared them in parallel. The outputs of these multipliers were then added and then fed into Xilinx's CORDIC IP core, which among many functions can perform a square root operation that is fully pipelined. The beauty of the AXI standard is that the XFFT and CORDIC IP's both accepted AXI-Stream interfaces, meaning they could be linked in series without the need of any supporting logic. Since I configured the squaring multipliers to have a 1 clock cycle latency, it was

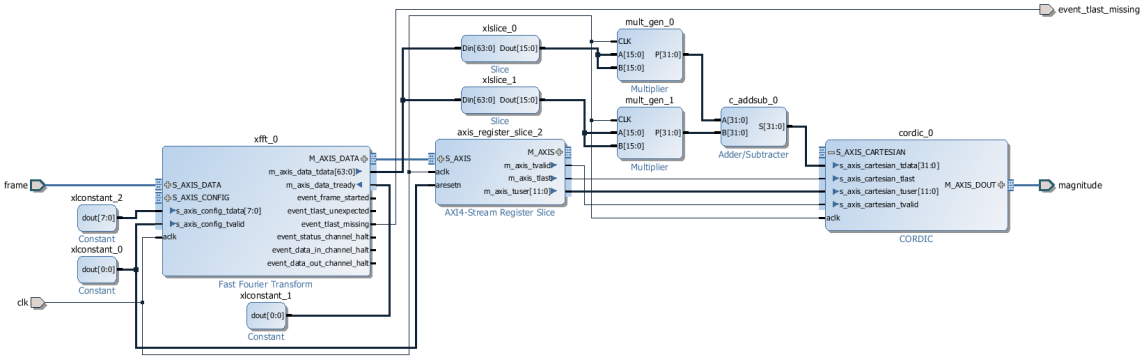


Figure 3.4: Vivado's block design interface

necessary to place a pipeline stage on the AXI interface between the XFFT and the CORDIC. However, this was as simple as inserting an AXI-Stream register slice IP in between the modules, which handles all handshake signaling as necessary. In this way, I was able to make a simplified FFT magnitude block design that had exactly the inputs and outputs I wanted without worrying about the implementation details contained within (Figure 3.5).

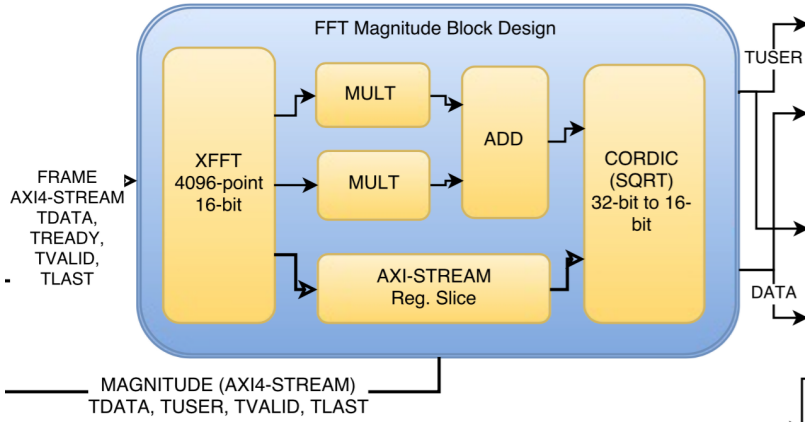


Figure 3.5: A schematic representation of the FFT Magnitude block design

3.4.3 XFFT output storage

In a similar fashion to how samples are stored in a frame BRAM to be sent to the FFT block, the FFT frequency spectrum output is stored in a spectrum BRAM for reading as a histogram on a VGA monitor or saving into an SD card. Conveniently, the AXI-STREAM output of the FFT block includes a TUSER bus in addition to the `TDATA` bus, which stores the frequency index that corresponds to the current data in the `TDATA` bus. This `TUSER` bus was easy to adapt into an address for separate BRAMS that store the spectral data for the histogram video output and SD card saving modules. The XFFT module prefers to give its output magnitudes in "reverse bit order," which increments its indices starting at the most significant bits rather than the least significant bits. However, using the TUSER index as an address into a BRAM guarantees that the data will make it into the right slot regardless of order.

For note recognition, I was only interested in the lower 1024 frequency bins corresponding to frequencies less than 1kHz and `TUSER` values less than 1024 (out of a maximum of 4096). Therefore, the write enable port of each BRAM was only asserted when both the `TVALID` AXI line was high and the `TUSER` value was less than 1024.

I used one BRAM for histogram video generation and one BRAM for SD Card because both the video module and the SD card saving module run in different clock domains (65Mhz and 25Mhz respectively). Because the two-port BRAM primitives allow separate clock domains on each port, they also allow data transfer between two clock domains without incurring timing violations. In order to ensure stability in multiple clock domain situations, the BRAMS were configured in "Write First" mode, per Xilinx recommendation. That way, when a collision does occur and a word is being read and written at the same time, the write will happen first in order to prevent data uncertainty.

3.4.4 Displaying the spectrum on VGA video output

Once the FFT data was stored in a BRAM, the histogram video module uses the read port of the BRAM to output pixel values that display a spectral histogram. At any particular horizontal position (hcount), the spectral histogram module only outputs a white pixel if the current vertical position is less than the BRAM data value at the address of the horizontal position. This works out to one column of

pixels per address (1Hz bucket) across the target resolution of 1024x768 pixels. In order for the 16-bit (0 to 65k) FFT magnitude values to fit on the vertical space of the monitor, the magnitude was right shifted 7 bits so that the maximum histogram height possible was 512 pixels. An example histogram output for when the guitar's high E string (≈330Hz) was played is shown in Figure 3.6.

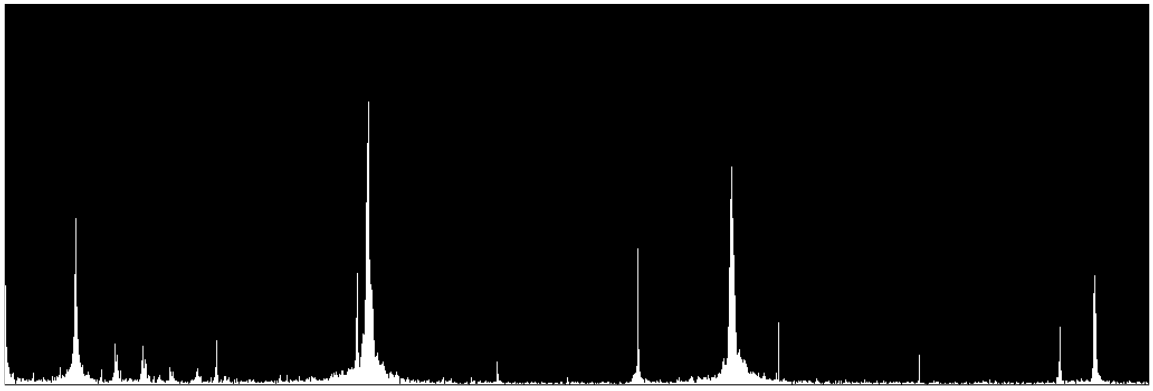


Figure 3.6: An example spectral histogram when a guitar's high E string is played

Upon first seeing what the spectra looked like, I was pleasantly surprised, as I expected them to be significantly more noisy. Compared to the input from a common electret microphone, the peaks in the pickup spectra were very clear and stable. The biggest anomaly in the spectrum was the consistent peak at 60Hz, which I assume was caused by mains hum, but since all the guitar frequencies were above 72 Hz, I figured I could easily crop out everything below 72 Hz.

3.5 Note Recognition

3.5.1 Choosing a note recognition strategy

With the spectra successfully processed and displayed, I needed to settle on a method for note recognition. The most straightforward approach that I considered first was to implement a peak finding algorithm to detect the frequencies at which peaks in the spectrum were centered. However, I felt that this approach would be very susceptible to anomalies such as harmonics or spurious peaks that were still present in the spectrum. In addition, such an algorithm would likely be expensive

with respect to clock cycles because one would have to traverse the spectrum in order.

Another approach I considered was looking only at the frequency bins at which each of the 37 notes under consideration were centered, and evaluate whether there presently is a peak there. I felt this was a slightly better method, but still would be very susceptible to harmonics and hard to calibrate.

In the end, I settled on an approach inspired by a 6.02 lab I had just worked through. The core of the approach is to calculate a correlation index between the current spectrum and several reference spectra that correspond to each note when played properly. The correlation index would be calculated, as inspired by 6.02, as the dot product of the current spectrum a reference spectrum, divided by the product of each vector's magnitude.

$$\text{correlation} = \cos\theta = \frac{\text{current spectrum} \cdot \text{reference spectrum}}{|\text{current spectrum}| \cdot |\text{reference spectrum}|}$$

Geometrically, if each spectrum is represented as a 1024-dimensional vector, this correlation index is identical to the cosine of the angle between the vectors. When the vectors are parallel (the peaks in the spectra coincide well), the angle approaches zero, and the cosine (correlation index) approaches When the vectors are very divergent (the peaks are in different places), the angle between vectors is large, and the cosine approaches zero.

I chose this approach firstly because I liked how it was data-driven in that it used its own output as a standard of comparison with the current output. I also liked how the approach could fully capitalize on the flexible and highly parallel capabilities of an FPGA to calculate something that would be expensive for a common processor to do. In order for the method to work for 37 notes, the FPGA would have to calculate 37 dot products in parallel, and rapidly enough to keep up with the FFT output sample rate. WIth its 220 DSP slices with multiply-and-accumulate functionality and 135 block rams, this would be an easy task for the Nexys 4. Finally, the method would be easily extensible for additional notes or even chords, because adding another correlation computation is as simple as adding another DSP slice and BRAM to store the reference spectrum.

Later on in the project, I decided to modify the correlation index formula to divide by the reference magnitude squared, rather than the product of each vector's magnitude. This was necessary because when the guitar was idle, the current spectrum vector would often be at a close angle to many reference spectra despite

being orders of magnitude smaller in size because nothing was being played. By dividing by the square of the reference magnitude instead, the new correlation index effectively scales the cosine of the angle by the ratio between the current magnitude and the reference magnitude, which necessitates that the current spectrum be roughly as "loud" as it is when a note is actually played.

$$\begin{aligned}\text{new correlation} &= \frac{\text{current spectrum} \cdot \text{reference spectrum}}{|\text{current spectrum}| \cdot |\text{reference spectrum}|} \cdot \frac{|\text{current spectrum}|}{|\text{reference spectrum}|} \\ &= \frac{\text{current spectrum} \cdot \text{reference spectrum}}{|\text{reference spectrum}|^2}\end{aligned}$$

To obtain the reference spectra, I chose the approach of playing each possible note correctly four times and saving their spectra to an SD card. Then the spectra would be averaged across all four trials to arrive at a reference spectrum that serves as initialization data for a BRAM on the FPGA chip. Then, as the FFT magnitude block design streams its magnitudes and indices to a correlator module, the module could look up the corresponding magnitudes at the same addresses each reference spectrum BRAM and perform the dot-product in real-time. The dot product computation would be completed only a few clock cycles after the FFT had finished streaming its frame of magnitudes.

3.5.2 Saving spectra onto an SD card

The first task to tackle was saving the current spectrum to sectors on an SD card. Thankfully, the SD card controller provided by Jono, a previous student, was a big time-saver. After importing his controller module, all I needed to do was send the appropriate bytes and control signals to the SD controller.

The SD card must be written in 512 byte sectors at a time. The saved spectra are 1024 words of 16 bits (2 bytes), or 2048 bytes total. Thus each saved spectrum must occupy four sectors on the SD card. The saver module handles this by advancing through one quarter of the spectrum BRAM for SD card data at a time and signaling to the SD controller module to switch to the next sector before advancing again. While the saver module is saving the BRAM data, it also outputs an active bit that forces the write enable line to the spectral BRAM low to ensure that no write/read collisions happen while it is saving.

Other inputs to the SD card saving module include a start input that is connected to the debounced center button, a memory slot selection bus where a user can select up to 64 different memory slots to record into, and a reset input resets the SD controller module and the saver's internal states. This reset line is often useful for resetting the controller when the SD card wasn't inserted initially.

3.5.3 Developing python scripts to initialize memory with saved spectra

Due to the difficulty of implementing filesystem navigation on an FPGA, the SD card is written in raw binary format as if it were a regular SDRAM. To read the data on a normal computer, I initially used tools such as wxHexEditor or Hex Fiend to copy and paste each set of four sectors for processing. However, with 40 notes and 8 chords that I wanted to record at 4 trials each, I wanted a quicker, more automated method for parsing the data. I eventually wrote a python script that opened the SD card volume, read the sectors in sets of four as requested by the user, averaged them, plotted a spectral histogram to confirm that the data was correct, and translated the data into a .MIF file that could be used to initialize a BRAM. The original plan for initializing BRAMS with the reference spectra was to generate .coe (Xilinx coefficent format) files, which are accepted by the Block Memory Generator IP for initializing. However, with 48 different .coe files, I would have to generate 48 different IP cores, one for each note. To keep the HDL sources modular and neat, I ideally wanted a solution whereby one correlator module could be used for all 48 notes, but parametrized to read from different memory files. With a bit of online research, I discovered a different syntax for memory initialization that used inferred BRAMs and a function called `$readmemb(<file>)`:

```
parameter REF_MIF = "../mif/00.mif";
// Inferred bram
reg [15:0] ref_mag [0:1023];
initial $readmemb(REF_MIF, ref_mag);
```

Because of the register array's size and the fact that it is read synchronously across a clock edge, Vivado will always infer that ref_mag should be implemented as a 16-bit wide by 1024 word deep BRAM. The addition of the readmemb command initializes the BRAM with the contents (in binary) of the REF_MIF file, with

one word per line. Because REF_MIF is a parameter, it can be configured independently for each instance upon instantiation. The syntax can be paired with a generator loop in the following concise syntax:

```
defparam correlator_gen[0].c.REF_MIF = "../mif/00.mif";
defparam correlator_gen[1].c.REF_MIF = "../mif/01.mif";
defparam correlator_gen[2].c.REF_MIF = "../mif/02.mif";
// etc...

wire [41:0] dot_product [0:47];
wire [47:0] dot_product_valid;
wire [81:0] debug [0:47];
genvar a;
generate for(a=0; a<48; a=a+1)
    begin : correlator_gen
        correlator c (
            .clk(clk_104mhz),
            .magnitude_tdata(magnitude_tdata[15:0]),
            .magnitude_tlast(magnitude_tlast),
            .magnitude_tvalid(magnitude_tvalid),
            .magnitude_tuser(magnitude_tuser),
            .dot_product(dot_product[a]),
            .dot_product_valid(dot_product_valid[a]),
            .debug(debug[a])
            );
    end
endgenerate
```

The use of input and output wires in a size 48 unpacked array dimension makes indexing into a certain instance's input and output packed buses easy.

3.5.4 The correlator (dot product) modules

The purpose of each of the 48 correlator modules (see Figure 3.7) is to compute the dot product between the current streaming FFT output and the reference spectrum it has been instantiated with. To achieve this, each correlator module has a BRAM

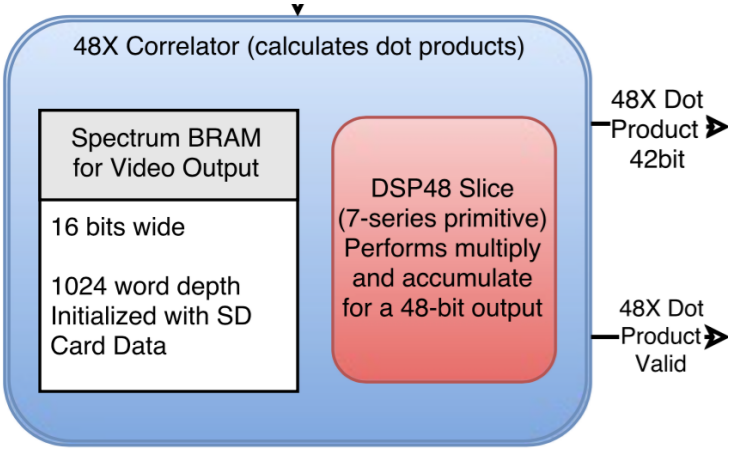


Figure 3.7: A schematic representation of the correlator (dot product) module

instantiated as previously described, as well as a DSP48 primitive slice. The DSP48 primitive offered in the Artix-7 FPGA is well-suited for this dot-product application because it can perform fully pipelined multiplies of two 16 bit inputs and accumulate (successively add) them to a maximum 48-bit accumulator (Figure 3.8). When one 16 bit input comes from the current spectrum and the other comes from the reference BRAM, this operation is exactly a dot product, whose accumulate output becomes valid at the end of each FFT output frame.

To integrate the DSP slice with the reference block ram and streaming FFT output, the correlator module uses the FFT output's `TUSER` address to retrieve the corresponding reference magnitude after one clock cycle, then send the current magnitude and that reference magnitude into the DSP. The DSP slice has a total latency of 4 clock cycles (4 pipeline stages), so four cycles after the correlator module sends the last pair of magnitudes to the module, the correct dot product will appear on the output. After the output has been read, the DSP is reset to setup the dot product for the next frame. The one caveat with this process is that the module must only consider `TUSER` addresses less than 1024, which occurs every four clock cycles (due to the reverse bit ordering of the FFT output). Therefore the module toggles the clock enable of the DSP slice according to this condition in order to only correlate the desired bandwidth.

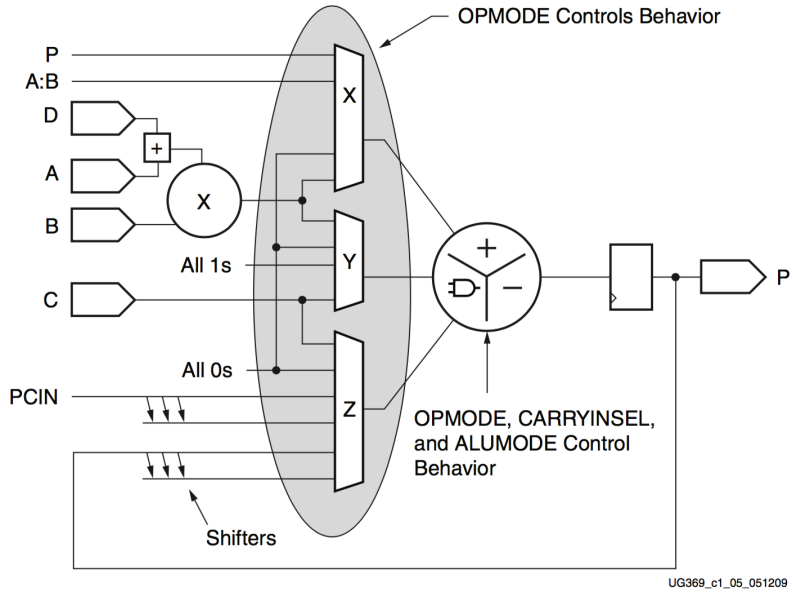


Figure 3.8: A simplified view of the possible operations in a DSP48 primitive[2]

3.5.5 Serializing the divide operation

The next step in finding correlation indexes after a dot product has been obtained for every note is to divide that dot product by the squared magnitude of the reference spectrum. These divisors can be easily precomputed on the computer when exporting MIF files from the SD card data. The divide operation requires some more consideration, however, because dividing by numbers that are not powers of 2 is non-trivial in FPGAs. To avoid getting into the details of division, I configured a Divider IP core that Xilinx provides. However, when implemented, the divider is complex enough that it is infeasible to instantiate 48 of them for each note or chord like the DSP primitives. Fortunately, neither is it necessary to have a divider for each note. While the dot product is best done in parallel while the FFT data is streaming in (it would take too long to do serially), the divider only needs to do 48 divisions total: one for each note's dot product and divisor. Since it is fully pipelined, the divisions can be done in series in a total time of (48 + division latency) clock cycles, where the division latency is 46 (Figure 3.9.

A process division container module handles this parallel-to-serial, division, and back to parallel process with a simple state machine that cycles through all 48

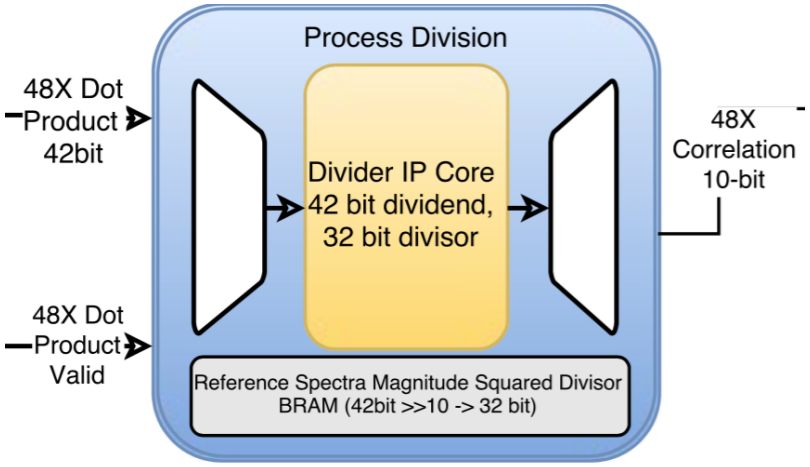


Figure 3.9: A schematic view of the process division module's operation

dot products and divisors, sending them into the divider core in series. The correlator modules that provide the dot product also assert a valid signal to indicate that the process divider module can begin division. Similarly, the process divider module asserts its own valid signal when all of its divisions have completed.

One annoyance with actually implementing this process division module was Vivado's refusal to allow unpacked bus arrays as input and output ports. To my knowledge, it appears that this functionality is specific to SystemVerilog, and while Vivado has support for many SystemVerilog features, it still does not allow unpacked array ports.

For example, I would have liked to have written the dot product inputs into the procesd division module as follows: `input [41:0] dot_product [0:47]`. Because this wasn't allowed, I chose to define a flattened, packed bus as input `[47*48-1:0] dot_product_f` and use 48 assign statements to map the unpacked array into the flattened bus, as follows:

```
assign dot_product_f[0*42 +: 42] = dot_product[0];
assign dot_product_f[1*42 +: 42] = dot_product[1];
assign dot_product_f[2*42 +: 42] = dot_product[2];
assign dot_product_f[3*42 +: 42] = dot_product[3];
assign dot_product_f[4*42 +: 42] = dot_product[4];
// ...etc
```

There is likely a more elegant solution to this problem (perhaps with a 2D packed array) but due to time limitations, I decided to stick with what worked and continue developing other modules.

All in all, the entire correlation calculation process reaches completion less than 100 cycles after each FFT frame has finished outputting is frame of spectral data. Out of the 26,624 clock cycles between 4kHz samples, less than $4096 + 100 \approx 4,200$ of them are needed for correlation processing.

3.5.6 Processing and displaying correlation values

To debug and calibrate these correlation values, displaying them in real time on a VGA monitor was critical. The process correlation module fulfills this task by outputting a horizontal bar chart of correlation values through its pixel output and also imposing hysteresis thresholds on the correlation to determine if a note is active or not (Figure 3.10).

One challenge for the module is the fact that correlation values are output in the 104MHz clock domain, while video must be output in the 65Mhz clock domain. To enable timing violation free correlation data transfer between these domains, I chose to use a distributed (in FPGA fabric) FIFO (first-in-first-out) IP core to load and release correlation values between the domains. Like a queue, each of the 48 FIFOs is written to whenever the divider has finished calculating the correlation indices. Then when each process correlation module sees that the FIFO is not empty, it reads the available correlation on the read port and signals to the FIFO that it has read the data (thus popping that correlation out of the queue). Since each FIFO is only 16 words deep and implemented in FPGA fabric, I figured this approach was less wasteful than using another BRAM. In ret-

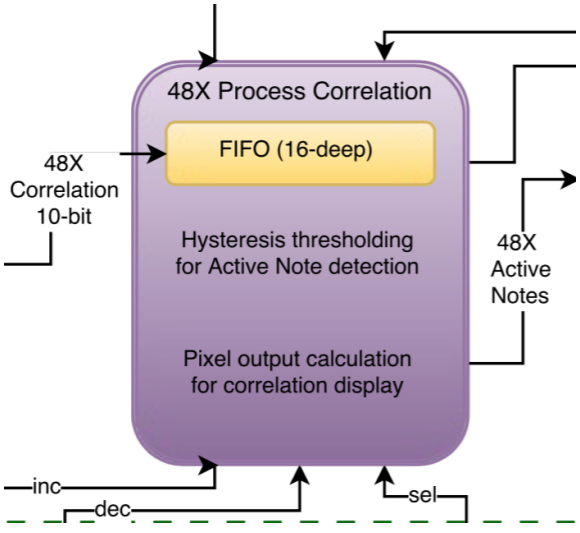


Figure 3.10: A schematic view of the process correlation module

rospect, I realize that using a single 48-word BRAM is not bad at all, and rolling my own synchronize module would likely be most efficient. Nevertheless, I enjoyed the opportunity to learn how FIFOs work.

Once the latest correlation value has been retrieved from the FIFO, the process correlation module performs an integratino step using an exponential moving average (also inspired by 6.02), which is a form of simple low-pass filtering that takes the following form:

$$\text{filtered correlation} = \alpha(\text{latest correlation}) + (1-\alpha)(\text{last filtered correlation})$$

In practice, I settled with an α value of $\frac{1}{32}$, but even with the filter, I felt there was a lot of room for improvement. (See Ch. 5. Review and Lessons Learned)

Each process correlation module also determines whether its note is active or not according to its hysteresis rules. If its filtered correlation value is greater than the inactive-to-active threshold, the note becomes active. Then, to become inactive, the correlation must drop to below the lower active-to-inactive threshold. This helps to alleviate the rapid fluctuations in value that the correlations experience.

To calibrate these hysteresis thresholds, the module offers increase, decrease, and threshold selection inputs, which allowed simple calibration of thresholds in using the buttons on the Nexys 4 board. This feature was very useful for us to calibrate each note's ideal thresholds experimentally and rapidly. Once we arrived at the best thresholds, we could read them off the hex display and override their initial values using parameters as necessary.

Finally, each process correlation module outputs its correlation value in video format by taking in hcount and vcount values from an XVGA module and outputting the appropriate pixel value at that position. This process is very similar to the spectral histogram video module, but with the addition of displaying upper and lower hysteresis bounds in video.

3.6 Serializing active notes

The last task for the audio processing design is to pass on the active note data to the game logic Nexys for scoring. Once again, there was an inter-clock domain

challenge, as the active note data is in the 65Mhz domain, but the serialized data should be in the 100Mhz clock domain, as that is the frequency at which the game Nexys scores note data. Once again, I used a distributed FIFO to transfer the 48 bit wide active note bus across domains. Once in the 100Mhz domain, a serializer module uses an internal counter to output the active notes serially over a data line, while asserting the sync line at the end of every data packet. Each packet was divided into 64 time segments that are each 8,192 clock cycles wide. The first 48 of these segments are the only meaningful ones, and the serial position of them in the packet represents the index of an active note. Such a slow rate was necessary because over the approximately 70cm length of wire connecting the Nexys 4's, high frequency pulses tend to deterioriate significantly, resulting in overflow of a pulse into neighboring time segments.

3.7 Testing the audio processing modules

In the process of testing several modules in the audio processing component, the simulation features integrated into the Vivado Design Suite were often helpful. For example, when testing the SD card saving module's state machine, the simulator's waveforms helped me realize that I had misinterpreted the form that one of Jono's SD Card Controller outputs. The ready_for_next_byte signal was actually high for eight clock cycles at a time rather than 1 clock cycle per byte as I had presumed. Figure 3.11 shows a simulation of the SD saving module in Vivado.

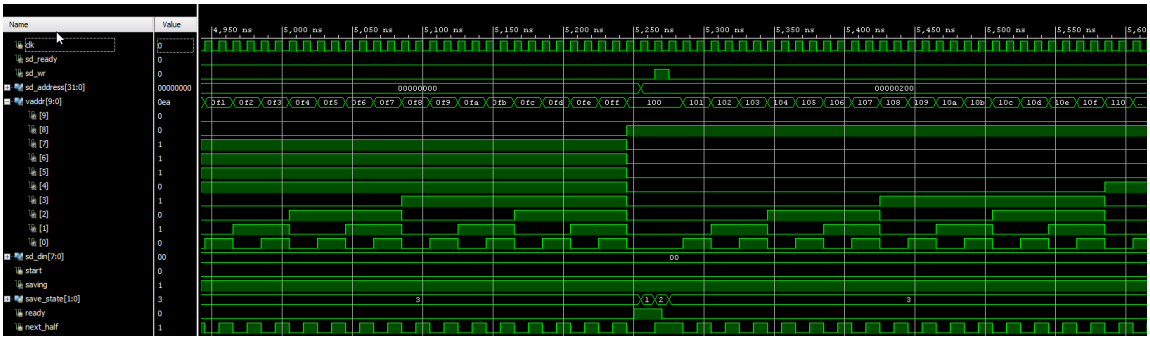


Figure 3.11: A logic waveform from a simulation of the SD saving module

Even more useful than the simulator tools was the Integrated Logic Analyzer feature available as a synthesizable IP core in Vivado. Essentially, the IP core im-

plemented its own mini logic analyzer for a user configurable set of probes within the FPGA fabric itself. Then, in the Vivado hardware manager, one can configure custom triggers for capturing logic frames of varying widths of all the probe lines. This was a lot easier than assigning a small number of probes to the PMOD ports and then fiddling with actual logic analyzer probes to capture the right frame. In fact, it was often easier to simply drop in an ILA core where a module was not working properly than to write a test bench module and simulate the design, which took significant time and was not guaranteed to be 100% accurate. The ILA was not without a cost, however; the core itself added significant time to the implementation of a design and also occupied a lot of space within the FPGA fabric.

4. Game Processing Implementation

4.1 Overview of overarching design decisions - Ryan

While it was unclear for a while whether the audio side should utilize the Nexys4 or the 6.111 labkit, it was clear from the beginning that the Nexys4 would be the correct choice for the game side. It's 12 bit VGA and PWM audio were suitable for the game's AV needs, and the built in hardware for interfacing with an SD card was necessary. Additionally, the amount of memory available was essential for storing large graphics assets and portions of song data.

From the beginning of the design stage, I had a desire to draw a large number of abstractions around portions of the game, to make each abstraction level very simple and easy to understand (Figure 4.1). I eventually realized that this was actually a complete pain, once new interconnections needed to be added and testing needed to be done. I suppose one of the most important things I got out of this project was the understanding that Verilog is hardware, not software, and I need to forget most of the things I know about writing software in order to be effective in writing HDL for FPGAs.

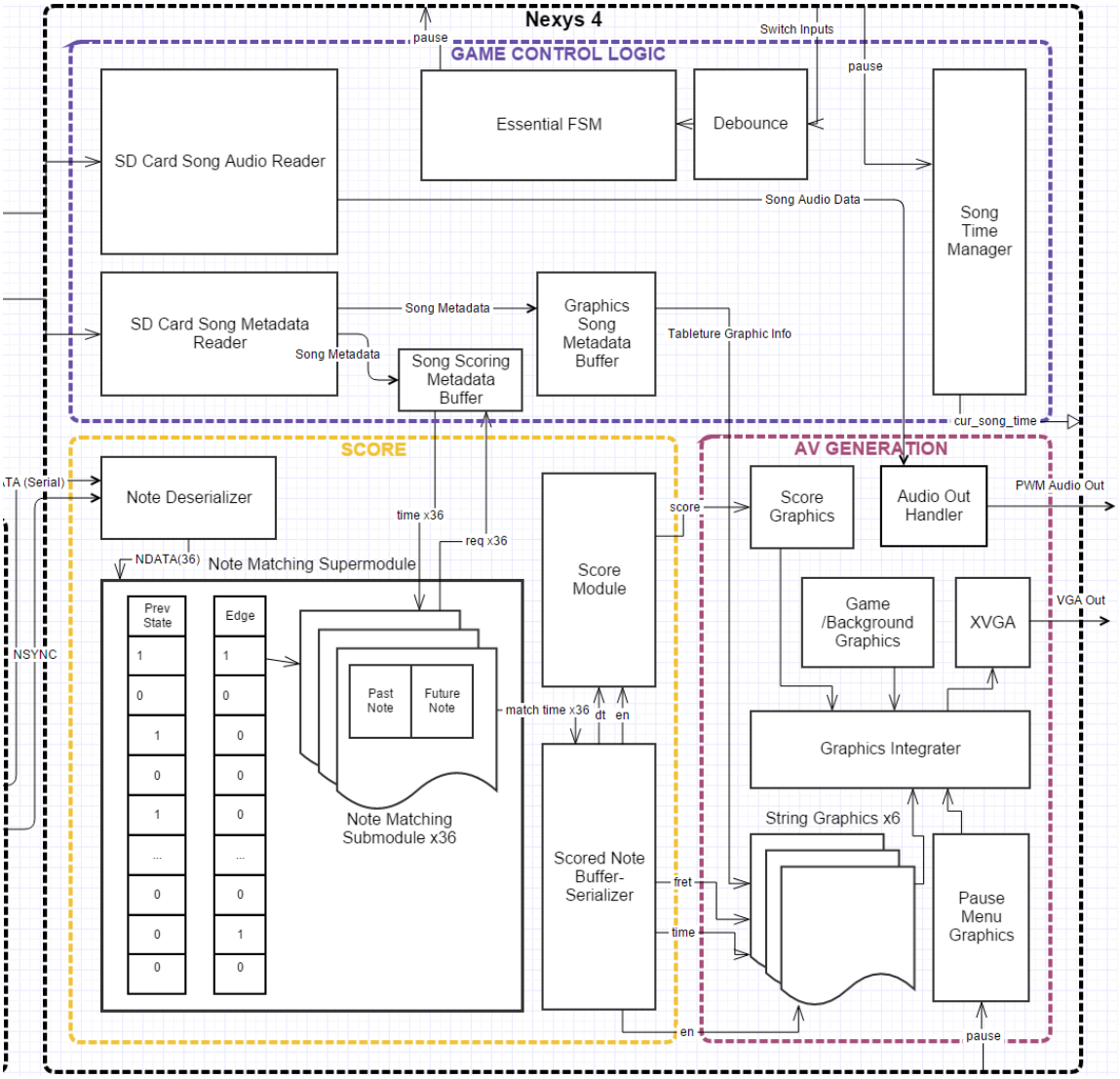


Figure 4.1: A diagram showing the design/organization of the game project

4.2 Game control logic - Ryan

4.2.1 Song time

The Control Logic block maintained the global song_time value, which was a 16-bit value that incremented every 10ms (1,000,000 clock cycles at 100mhz). This allowed for a song time time of up to 11 minutes, which was decided to be enough, and a note resolution of 10ms, which is also a very reasonable resolution, since timing differences of less than 1/100th of a second are essentially indistinguishable to humans. The designation for song_time to be 16-bits also made the storage in the SD card exactly 2 bytes per note time, which is convenient. This value stopped incrementing when the game was paused, and was set back to 0 when a reset signal was applied.

4.2.2 Simple FSM

Also in the Control Logic block was a simple FSM which took debounced inputs from the Nexys' buttons and switches and paused or reset the game accordingly. It originally also made sure the game did not progress until the metadata and audio was loaded, but this was dropped with the SD card functionality, because there was no longer anything to load. More on the SD complications below.

4.3 Metadata loading - Ryan

4.3.1 Overview of Metadata

In the context of this project, metadata refers to the information about a song that corresponds to the guitar part that needs to be played. A single "piece" of metadata contains a pitch index (out of the 37 pitches we were detecting), a string (out of 6) and fret index (out of 17), which are necessary because a single pitch can be played in several string/fret combinations on a guitar (we needed to pick one to render on the screen–playing any one of them was indistinguishable to us, and

also to anyone listening to the guitar, but there is usually one that is most convenient), and finally a 16-bit time value that represented what time in the song that should ideally be played (Figure 4.2).

e	f	#	g	#	a	#	b	C	#	D	#	E
B	c	#	d	#	e	f	#	g	#	a	#	b
G	#	A	#	B	c	#	d	#	e	f	#	g
D	#	E	F	#	G	#	A	#	B	c	#	d
A	#	B	C	#	D	#	E	F	#	G	#	A
E	F	#	G	#	A	#	B	C	#	D	#	E
0	1	2	3	4	5	6	7	8	9	10	11	12

Figure 4.2: A diagram showing the various places a single pitch can be played on a fretboard. Equivalent pitches have the same letter and color.

As far as storing metadata on the SD card was concerned, 4 bytes/word were needed (16 bits for time, 6 for pitch, 3 for string, 5 for fret, and 2 extra to make the total divisible by 8). The extra 2 bits could be used to signify whether or not the word was the end of the song (2'b00 for continue reading, and 2'b11 for End Of File). Additionally, 4 was a good number of bytes, due to it dividing evenly into 512 bytes, which was how much data the SD card controller we were using would read at a time.

The scoring block needed to know what note was being played, and when, so that it could compare that to the data it was receiving from the audio Nexys and score it appropriately. Meanwhile, the AV block needed to know when a note would be played, and the fret/string combination for that note, so that it could render a correctly numbered sprite on the right string, and such that when song_time was the time when the note should be played, the sprite would be passing directly over the "Play Now!" bar on the screen.

4.3.2 Utilization of BRAMs

A very serious design problem was the issue of getting metadata from a single source to be utilizable/sortable by several different components of the game sys-

tem. The scoring system needed the metadata to be in time-order after being separated by note, while the graphical interface needed the metadata to be separated by string before being sorted in time-order. A lot of time was spent figuring out how to have multiple different access schemes into this data table such that all of this information was available within a clock cycle or two for the modules that needed it. Eventually, in an effort to have a working and demonstrable product for the checkoff, I gave up on this pursuit and hardcoded the metadata for a simple song (Mary Had a Little Lamb–which was also desirable for gently testing the note-detection functionality due to it only having 4 different pitches, none of which were played simultaneously) into a few BRAMs that were contained within the modules that needed the data.

In retrospect, it is very unfortunate that this was only done as a last resort, because after I did it I realized that the memory allowance of a Nexys was far more than I had previously thought. Rather than have a single data storage with a very convoluted interface for getting the results I wanted, I actually had the BRAM allowance available on the device to just have a copy of all relevant data in each module that needed it. That just seemed like an approach that was so inefficient that I didn't figure the hardware allowance would be enough for it. Because I was unable to create a reliable interface into a singular table, I had to throw it out for the sake of the project, and with it the SD card functionality. However, with this distributed memory scheme, implementing the SD card functionality would've been relatively simple, with each word being broken up into it's relevant pieces and then written to the particular metadata buffer that needed it. My biggest regret in this project was probably not realizing this sooner, and just keeping my approach simple, because I had to throw out a lot of project potential with it when I finally compromised and went with the simple distributed approach, and no longer had the time to piece it all together again.

4.4 Scoring - Ryan

4.4.1 Parallel matching (37x)

Actual matching of notes was handled by 37 individual pitch-matching modules. Each module stored the most recent time that had already passed for which it's pitch should have been played, and the most recent time for which the should

pitch be played which hadn't occurred yet.

The parent module stored a 1-cycle history of note-states was saved, such that a match would only be triggered on the rising edge of a note.

When a match for this pitch occurred, this module received a trigger and then compared the current song time to the stored times for the past and future note, and returned whichever one was closer to the current time. Each of the 37 modules had a 16-bit time bus plus an enable line to the buffer-serializer, and whichever time was chosen was sent along to the buffer-serializer, along with an enable trigger.

4.4.2 Parallel metadata request

Each pitch module also had a 16-bit time bus plus a request' and available' line to the metadata controller. If the future note was written invalid in the module (either by being matched, or by becoming older than the current song_time and being shifted into the past note slot) then the module would set the request line high. Eventually the metadata controller would set the available line high, at which point the new future time was available on the 16-bit bus, and the module could set its request line back to low and write in the new note.

As I mentioned above, with a distributed BRAM module the pitch module would've had an entire BRAM for metadata, and would not need to interface with a controller except at the loading of a song, during which time the SD controller would write into the pitch module's BRAM all notes that matched it's pitch, as well as their time. Then the pitch module would pull metadata from it's internal memory as needed instead of interfacing with an external controller.

4.4.3 Serializing parallel-matching results

The buffer-serializer module was a critical piece of the scoring functionality, taking information from a myriad of pitch module busses, and outputting a single bus into the score module, and a few into the AV string modules (these rendered the note-sprites on the screen–more on them later). It played a key part in coordinating the efforts of many small, disconnected pieces of the project.

This module worked by checking the match triggers from the pitch modules. If

any were high, it would take the highest pitch that was active, compare it's match time to the current time, and calculate the absolute value of the difference, which it sent off to the score module as an indication of how accurately the note was played. With this pitch, it also sent a trigger to the graphics module corresponding to each string where that note could be played, along with the fret value on that string where it could've been played. It also sent the note's correct time to the strings, such that each string had all the information it needed to match the note matched to one of its sprites (more on how these modules worked later).

The decision to place priority on the highest pitch that matched was arbitrary, but motivated by the need to simplify the problem at the cost of being unable to match two notes in the same clock cycle. Without making this compromise, we would've either needed to have far more interfacing with the neighboring modules, or have some queueing of note-matches. This was determined to be not worth the complication, as the probability of two note-matches occurring on the same 100mhz clock cycle are negligibly small.

4.4.4 Score Tabulation

The scoring module graded on a roughly exponential scale. If a note was played within 100ms of it's intended time, the score would be incremented by 100. Similarly, accuracy within 250ms would be awarded 50 points, within 500ms would be awarded 25 points, and within 1s would be awarded a measly 10 points. Anything beyond that would not receive any points.

4.4.5 Testing scoring

Throughout the implementation process, the scoring portion received the most thorough testing, due to its relative non-reliance on other pieces of the game. Test benches were written for each component and simulated in Vivado Design Suite. Later, when the rest of the game project was in a more put-together state, an Internal Logic Analyzer (from Vivado's IP catalog) was synthesized with the project, and made sure that the behavior of the scoring was still correct when actually implemented.

Additionally, since the note recognition was not in a usable state for the majority of the testing and integration stage, an AI guitar player' was implemented,

which simply proceeded through the 37 pitches and set each one high for a single clock cycle, such that it would never miss a note. While this AI scored terribly (playing each note as early as physically possible, and only earning 10 points each time), it made testing of the scoring functionality (and later, the graphics as well) very straightforward.

4.5 Graphics generation - Mitchell

4.5.1 Compressing a high quality background image to fit in BRAM

For the high-quality game graphics, we wanted to create an environment akin to the original Guitar Hero games, where a crowd in the background helps foster the impression that the player is entertaining an audience. We also wanted to display graphics that were a notch above typical FPGA graphics that are limited to few colors and low resolution. I decided that the most efficient way to do this was to develop one large background image rather than isolate the game title, guitar strings, background, and crowd into separate sprites (Figure 4.3. This would lower complexity on the FPGA side and save time, since outside of note sprites, the game is fairly static.

Without involving the DDR3 memory and the world of memory controllers and frame buffers, the main challenge with displaying such a large image is storage space within the FPGA. If one were to store the entire 1024x768 image at 12-bit color, one would need roughly 260 36Kbit BRAM36 primitives, far greater than the 135 available on the Artix 7 100T. However, with the background image we wanted to display, there as a high degree of regularity, or predictability in the pixel data. For example, more than half of the background image is a solid, continuous background color. This lower entropy in the image was an easy opportunity to employ some compression strategies to reduce the number of BRAMs required.

In 6.02, I learned a variety of compression techniques, some more efficient than others. While using a Hamming code for the pixel data or LZW compression would have yielded higher compression ratios, I chose a much simpler to implement compression techinique: pixel run length. Essentially, at a certain address, the BRAM's corresponding word contains the pixel color value, but also a fixed-length value that indicates how many of the following pixels are identical. Because the background image largely consists of continuous runs of the same

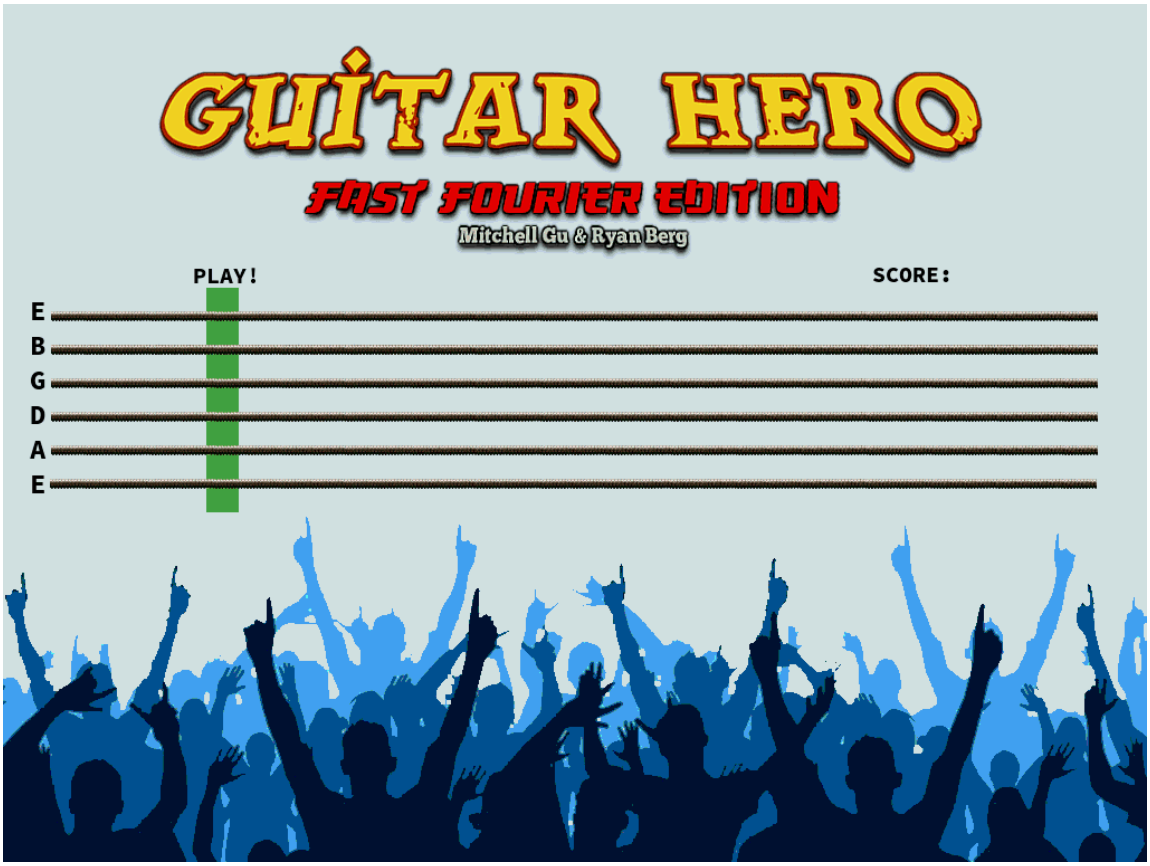


Figure 4.3: The desired background image for the game

color, this approach can yield an effective compression ratio.

Another observation is that the color palette of the background image could be reduced to only a few colors without compromising the output quality to an observable degree. With full 12-bit RGB color, there are a total $2^{16} = 4096$ colors. However, by reducing the color palette of our image in GIMP to 62 colors, nearly identical quality could be achieved with much less information storage.

In order to support the reduced color palette, the image has to be stored in indexed color format rather than 12-bit RGB format. The difference is that each pixel has an associated 6-bit index into a shared color palette table, rather than a 12-bit RGB value. That index is then used to look up the actual 12-bit color from the color palette table, at the expense of one extra clock cycle of latency.

To implement both run length compression and indexed color, I used a combination of the GIMP photo editing tool and a Python script. GIMP was used to adjust the image to my liking and reduce the color palette to 62 colors that best represented the image. Then, with the help of the PIL python image processing library, I iterated through every pixel in the image and created .coe files for two tables: a run table and a color lookup table. With some iteration, I found that the best fixed bit length to encode run length was 2 bits, which covers run lengths from 1 to 4 repeated pixels. If a run was more than four pixels, a new run would have to be started for each set of four identical pixels.

An entire word in the run table was thus 8 bits total: the first two bits would encode the run length, and the last six bits would index into the color lookup table. In the end, the image was reduced to around 2Mbits of data, a significant improvement compared to 9Mbits had it been uncompressed. When initialized into BRAM primitives, the run table takes up 62 BRAMs, less than half of the 135 total, which left plenty for the rest of the game graphics and metadata. The color lookup table was relatively small (64 words deep), so it was implemented as distributed memory instead.

4.5.2 Generating Note Sprites

Compared to configuring the background images, the note sprites were relatively simple to configure. Each sprite was 32x32 in resolution, so they could each fit easily uncompressed in a 13 bit wide by 1024 word deep BRAM18 primitive. (The rationale for having a 13th bit is detailed in the following graphics rendering section). Each sprite was again created in GIMP, with 18 versions with the numbers 0-17 on them corresponding to all possible fret positions. Because there were many of them, I again took the approach of instantiating the sprite brams using the `$readmemh` function, Python-generated MIF files, and inferred BRAMs. This time, instead of spectrum data, the MIF files would contain raw RGB values:

4.6 Graphics Rendering - Ryan

4.6.1 Integrating graphics and the alpha bit approach

The entire graphical interface could be split into a set of layers, with a certain priority such that if two layers wrote a pixel to the same location, the pixel from the layer with higher priority would be rendered. Whether or not a layer had written a pixel was determined by adding a 13th alpha' bit (onto the 12 bit RGB value) to the front of the signal, which was 1 if the pixel was to be rendered, and 0 if the layer was transparent at that location. More complicated alpha integration was unnecessary, because there was no reason to support partial opacity. This set of layers went as follows:

- **Pause Menu**
  This layer had the highest priority because if the menu were ever being rendered it would need to cover all of the game pieces underneath it. For preliminary testing this menu was a simple grey rectangle which rendered if the game was paused. As checkoff approached, there wasn't much time to make a prettier pause menu, and so this was eventually replaced by a filter that inverted every pixel on the entire screen to photonegative if the game were paused. This made it more clear that the game was paused, and left everything visible while the game was paused, which was a nice plus.

- **Score**
  This module would display the decimal score in a nice font on the upper-right side of the screen. This was cut due to time, as implementing another set of sprites and converting from binary/hex to decimal is a nontrivial matter. The score is instead displayed in hex (along with the current song time) on the segmented display on the Nexys board.

- **Strings 6 through 1**
  Each string module handled the rendering of the fret sprites onto the string (which was actually in the background layer), and moved the sprites such that they passed play at the proper time, timed-out, and also changed colors upon being matched. More on that later.

- **Background**
  This layer rendered the background image, and had no alpha bit because it

was the lowest layer, and needed to provide an RGB value for every pixel location.

4.6.2 Parallel sprite table access

It was arbitrarily decided that strings should render a maximum of 5 fret sprites each, which should be enough for most songs. In retrospect, making that number higher, like 10-15, might've been preferable, but it worked just fine for our purposes. This made the maximum number of fret sprites 30. The problem introduced by this was that there were 18 sprites to draw from BRAMs (for frets 0-17), so there could only reasonably be a single set of sprite lookup tables. This meant that a coordination scheme needed to be used to ensure that all 30 could cooperatively look into the table.

This problem was solved by only giving a single sprite handler control over the address into the memory for the read at a single time. This was determined by whether or not a sprite were present at any particular pixel, which required the easily-enforceable invariant that no two sprites would overlap. If a particular clock cycle fell into a sprites domain, it could decide the memory address, and all other sprites would return a string of 0's for their address. Then, all of the sprite handlers' address bits were bitwise OR'd, which produced the final address.

Another problem resulting from having a single coordinated memory access among all sprite handlers was that sprite handlers weren't assigned the same fret values, and as such needed to look at different tables. This problem was solved by having a single read of the memory return the pixel values for all of the sprites (for a particular pixel location on the sprite). All of the sprite handlers were privy to this single output, and indexed into it according to their fret value.

One final problem with sprite lookup, unrelated to the parallel access issues, was the fact that BRAMs take a clock cycle to look up the necessary data once the address is asserted. This means that there would need to be an additional clock cycle between calculating the address and receiving the data that was requested. Instead of pipelining to resolve this, I simply clocked the BRAMs at 130 mhz, so that the additional clock cycle occurred during the low portion of the 65 mhz clock that the rest of the graphics ran on.

4.6.3 Matching sprites

When a match occurred, all of the string modules received the time of the match, as well as the fret that the note would've been on their string (if applicable–each string cannot play every pitch). The matching occurred by checking all of the currently rendered sprites for a sprite that had a time and fret that matched the information received. If there were one, it's state was set to matched' and the sprite would be inverted to provide a visual indication that the note was scored.

5. Review and Lessons Learned

5.1 Mitchell

Overall, I'm happy with how the project turned out. I learned a lot about using interesting features in modern FPGA tools, and also made many connections across different subjects such as Fourier transforms and employing compression techniques in hardware. I was pleasantly surprised with the amount of overlap between the class and subjects such as 6.004 (computational structures) and 6.02 (digital communcation systems). A large part of my goals in taking this class were to learn digital hardware design (which is hard to learn by oneself with just the internet) and experiment with working in the frequency domain. I felt this project was a great way to achieve both of those goals and also gain a broad knowledge about how previously magical, high frequency digital devices work at their fundamental level. After doing this project, I feel like I'm in a better position to understand how digital boards such as oscilloscopes, computer motherboards, or smartphone boards operate, which is something I have always been curious about.

If I were to continue work on the project, some things that definitely need adjustment are the FFT frame size, correlation formula, and correlation low-pass filtering.

The issue with the FFT frame size is that with 4096 samples per frame and a 4KHz sample rate, a given stimulus's frequency content stays in the frame for approximately 1 second. This means that the system does not decay spectral peaks quickly enough, which greatly hinders the ability to play the same note many times in succession. WIth hysteresis on the correlation values, it is likely that the second time one plays the same note, it will still be active from the previous time

it was played. I experimented briefly with some approaches for shortening the decay time such as only passing samples for a subset of the frame to the FFT or passing a repeated subset of the samples several times in one frame. However, in both cases the output spectrum contained undesirable components due to the nature of the technique. To arrive at a solution, I would likely need to take a deeper look at how what effects the sample frame has on fourier transforms.

The correlation formula is also still far from perfect because when it is scaled by the ratio of the current spectral magnitude to the reference spectral magnitude, the current spectral magnitude drops out of the expression. This is a problem because notes played on the lower strings of the guitar are significantly higher energy and contain more harmonic content in the frame. Therefore they have very large magnitudes, which results in large dot products regardless of how similar the vectors are, which results in nearly every note becoming active. I think the optimal correlation expression would be some combination of the original cosine of the vector expression and a penalty for very low spectral magnitudes so that the current spectral magnitude term stays in the expression.

Finally, the use of an exponential weighted moving average filter for low-pass filtering the correlation indices has experimentally been not ideal and still results in rapid fluctuations that the hysteresis even has trouble compensating for. Given more time, I would experiment with an FIR filter as used in Lab 5 of the class to provide a more ideal low-pass filter.

5.2 Ryan

My biggest regret/learning experience resulted with not sticking to the mantra of "Keep it Simple, Stupid" in regards to the storage and distribution of metadata. After the fact, distributed storage of metadata is a very appealing concept and would've made reading/handling data from the SD card very, very straightforward.

I also learned a lot about the usefulness of IPs (procedurally generated modules, essentially) and the Vivado IP catalog. I wish I had come into this project knowing how useful they are, because reinventing the wheel in regards to generic Verilog functionality is a largely wasteful endeavor. The Block Memory Wizard, Clock Wizard, and the Internal Logic Analyzer all made my life a lot easier over the course of implementation, and perhaps if I'd known how IPs worked from the

start, I would've made more design decisions around the idea and had an easier time.

As a general philosophy regarding HDLs and FPGAs, I feel like I learned to stop treating Verilog like software. Abstractions are only good if they neatly compartmentalize functionality with a minimal amount of outside influence, and keeping the number of abstractions in a project down is extremely handy. Otherwise they become a serious pain to maintain over the implementation of a project.

Overall, I'm happy with results of this project. I'm sad that not all of it went according to plan, and there are still bugs present, but I think that is in the nature of picking an ambitious project and learning a lot on the go. I feel like this project gave me a much better feel for how large-scale Verilog development works, and how to better get what I want out of an FPGA in a timely fashion, much more so than I feel the labs had. The working product, though not as glamorous as imagined, is very close to the goals we set for this project, which in hindsight is quite a satisfying accomplishment.

6. Conclusion

Guitar Hero Fast Fourier Edition has proved to be a challenging, yet rewarding endeavour for us to learn digital hardware design and extend our shared interest in playing guitar. The project came close to achieving all its main goals, such as guitar note recognition, serialization and deserialization between Nexys boards, note matching and scoring, and advanced game graphics. Due to the complexity of the project, we did not have nearly the amount of time we would have liked for integration and testing of all the parts, but we were very happy with how much integration we did achieve by project checkoff time. If given more time, we feel that we could bring the project to the level of an entertaining, polished product featuring more accurate note recognition, proper note sprite matching for popping played notes off the display, loading of song audio and metadata from and SD card, and display of the score on the VGA display. Despite these shortcomings, the concepts we've learned throughout the project's challenges have been very rewarding and inspire us to explore the FPGA world further. We are also excited by the prospect of similar note recognition technologies using actual instruments spreading throughout the Guitar Hero and interactive music game genre.

References

[1] Fast Fourier Transform v9.0 Product Guide. (2015, Sep. 30). *Xilinx* [Online]. Available:
http://www.xilinx.com/support/documentation/ip_documentation/xfft/v9_0/pg109-xfft.pdf

[2] 7 Series DSP48E1 Slice User Guide. (2014, Nov. 10). *Xilinx* [Online]. Available:
http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

Appendices

A. Source files

All source files, IP configurations, block designs, memory initialization files, and bitstreams for both the audio and game Vivado projects in addition to suppoorting python scripts are available at Github.