Daniel Mendelsohn
6.111 Final Project Report
December 11, 2013

A plug-in piano game for FPGA

Introduction

I built a plug-in game for an electric keyboard. It is essentially a Guitar Hero variant, but unlike Guitar Hero, it uses an actual working instrument as an input device rather than a faux-instrument game controller.

When learning to play a new instrument, it's important for novices to become deeply engaged in the music from an early stage. For experienced players, such engagement comes naturally with the satisfaction of creating music. Novices, however, must work through a significant period of struggle before realizing any impressive results. It is natural to feel somewhat discouraged. Many people fail to get over the learning "hump". Many parents who pay for their children's music lessons have experienced the challenge of motivating a new learner through this difficult period.

In many fields of education, games help drive the learning process and motivate students. This is particularly effective with regards to children. The massive proliferation of educational kids' games have demonstrated the efficacy of gamification in teaching math, reading, language, typing and myriad other subjects. See Figure 1 for an example of this kind of graphical user interface. With my 6.111 final project, I saw a new opportunity to gamify piano education.

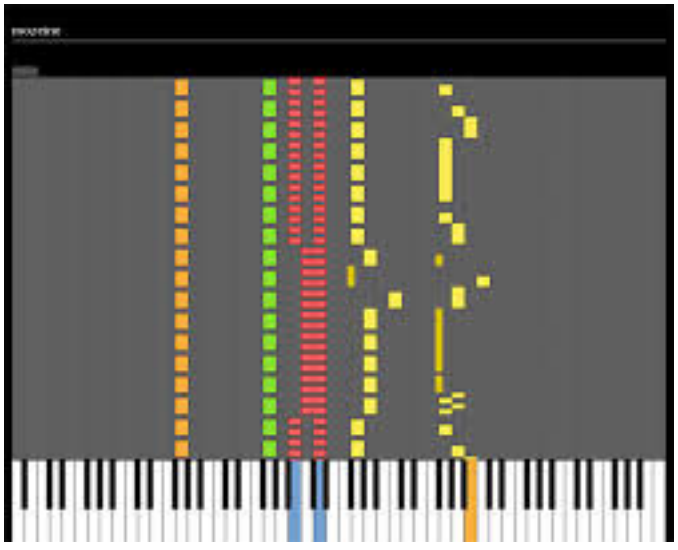

Figure 1. This is a screenshot of a different piano hero game. Notice that notes scroll from top to bottom, and a stationary keyboard on the bottom gives the user feedback.

The user interface is modeled after other popular instrument-based video games, such as Rock Band and the aforementioned Guitar Hero. In the game, notes flow vertically down the screen from top to bottom. There is a stationary keyboard

graphic along the bottom edge of the screen. When notes reach the keyboard graphic, the user is supposed to play the note. The user must press the key at the correct time and must also release the key at the correct time. If the user succeeds at this task, he or she is rewarded with a score bonus. If the user misses a note, presses the wrong key, or releases a note early or late, that results in a small penalty. The keyboard graphic shows the user which keys he/she is pressing and there is also an on screen indication of success/failure with each note. This general format allows for a wide range of difficulty settings; selecting a song with more notes or a higher tempo increases the challenge. As a stretch goal, I sought to build a mode of operation in which the system could learn new songs that are played by the user.

I believe that implementing this product in hardware rather than software is a good choice, because it fits with my vision for the consumer experience. I imagine a small, plug-in device into the MIDI port of an electric keyboard. A VGA port will allow for connection to a display. Thus, a user could set up the system with little hassle. They do not need their computer nearby in order to use this product. Furthermore, as children are very prone to damaging things, parents who buy this hardware product do not have to risk damaging their computers. With a software product, a computer connected to the keyboard would be necessary, and this is undesirable.

Technical Overview

As a major design principle, I wanted the interface between my system and the devices upon which it depends to be as simple as possible. The user simply plugs in the MIDI output of the keyboard into the FPGA, and connects the VGA output of the FPGA to a monitor. All game data, such as images and note sequences, is stored in memory on the FPGA. The MIDI output is a good choice for the interface to the FPGA due to its simple, compact, and low traffic protocol. Similarly, I chose to output display information via VGA because of its simplicity and familiarity (and because we already have a module to do this thanks to Lab 3). It is important to note that the scope of this project, at least initially, was limited to just one octave. In many respects, I managed to create a design that works for a keyboard as large as 61 keys (5 octaves).

My design is comprised of four major blocks, as seen in Figure 2, namely a MIDI interface block, a control block, an action interpretation block, and a display block. Here, I will very briefly describe each block, but they will be described in greater detail later on.

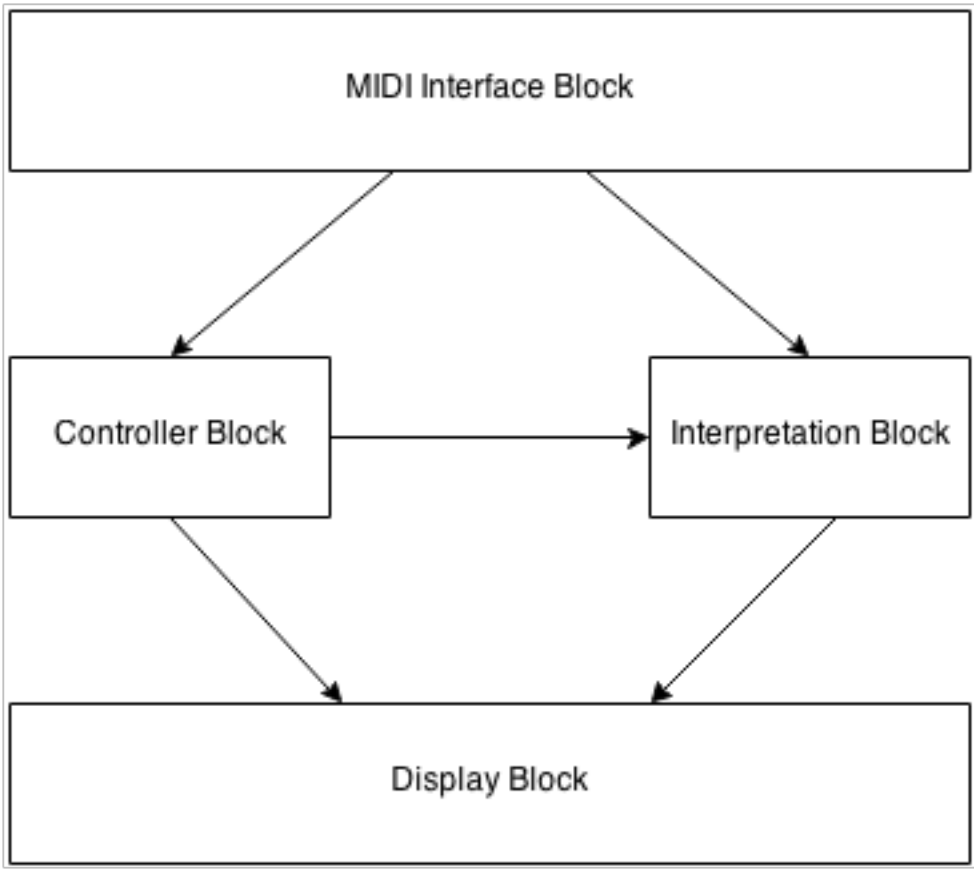

Figure 2. This is a high-level block diagram of my system. Raw input comes into the MIDI interface block, which feeds information to the interpretation and control blocks. The control block controls the timing and data, and sends relevant data and timing signals to the interpretation and display blocks. The display block shows the user graphically what is happening on a VGA display.

The MIDI interface block receives input from the piano keyboard and interprets it, relaying that data in a more concise, usable format. Specifically, it must de-serialize MIDI input according to the MIDI standard, and report its findings as output.

The control block handles the mode of operation of the game. For example, it has the power to pause and resume the game. It can also enable the "write" mode, in which a user plays a song of their choice and the system records their actions, and allows for them to be played back later as a newly created "level" in the game. Furthermore, the control block contains all the note data, and distributes it to other blocks as necessary.

The action interpretation block compares the "truly correct" notes from the internal logic to the notes that the user is actually playing. It is responsible for keeping track of which keys are currently pressed, and which of those pressed keys are currently scoring. It must categorize all key presses as "correct" or "incorrect", and all un-presses as "correct", "early", or "late". Furthermore, it must recognize

cases where a note should have been played but was not. It outputs information about which keys are pressed, which keys are scoring, and any events such as "incorrect press" that are occurring.

The display block gives the user a visual representation of the state of the game, and gives feedback on the user's actions. There are two main types of display sprites. First, there are static keys at the bottom of the screen, which are shaped and colored like real piano keys. These keys light up when the user is pressing the corresponding key on the physical keyboard, and they light up different colors depending on whether or not that key is being pressed correctly or incorrectly. The other main sprite is the note, which starts at the top of the screen and flows downward. Information about which notes are coming soon is received from the control block. The notes are also color coded to indicate user successes and failures. The output is VGA, and I accomplish this in the same way as we created a VGA signal in Lab 3.

The MIDI Interface Block

The purpose of this block is to convert the raw electrical signal from the piano keyboard into a convenient and easily used digital format. This requires an effective and accurate interpreter for a MIDI-compatible signal.

Some background on MIDI is helpful here. It is a serialized data protocol that is specific to the domain of music. It sends bits at a rate of 31,250 baud (bits per second). All bits are sent as part of a MIDI *word*. A MIDI word consists of a low start bit, followed by eight bits of data, followed by a high stop bit. Notice that MIDI calls for a steady-state high signal when no messages are being sent, so a low start bit and a high stop bit makes sense. A MIDI *message* contains up to three MIDI *words*, and is the basic unit of specification in the MIDI protocol. The first word in a message corresponds to a specific command; this command also tells the receiver how many words are in this message (1, 2, or 3). Any later words in a message are parameters for the aforementioned command. Table 1 below shows the list of MIDI commands words. Notice that the first four bits alone of the command word specify the command; the last four bits specify a *channel*, a concept into which we will not delve.

MIDI commands

Command	Meaning	#Parameters	Param1	Param2
0x**8**0	Note Off	2	key	velocity
0x**9**0	Note On	2	key	velocity
0x**A**0	Aftertouch	2	key	touch
0x**B**0	Continuous controller	2	controller#	value
0x**C**0	Patch change	2	instrument#	value
0x**D**0	Channel Pressure	1	pressure	
0x**E**0	Pitch bend	2	lsb(7bits)	msb(7bits)
0x**F**0	(non-musical commands)	0		

Table 1. This is a detailed look at MIDI commands and their parameters. In the context of our system, we only look at "Note On" and "Note Off" commands; any other commands are for more complicated musical structures and are ignored.

It was no trivial task to get the electrical signal from the keyboard into the FPGA. First of all, I had to hack together a custom converter for an ordinary MIDI cable. I bought a MIDI cable, and cut it in half. I stripped the exposed half of the cable, which contains four separately insulated wires surrounded by copper strands that carry a ground signal. One by one, I stripped soldered those interior wires to the 12-gauge wire required to interface with the FPGA proto-board. I also connected the ground copper to a 12-gauge wire. After plugging these wires into the proto-board, I found the two relevant ones; the voltage between these two wires represents the signal. I ran this signal through an optical isolator, which buffers the FPGA from any crazy electrical signals that might exist in the real world, and ensures that only 0V or 5V voltages go into the FPGA itself. I plugged the output of the optical isolator into one of the user ports of the FPGA; the rest of the story is internal, and a summary can be seen in Figure 3 below.

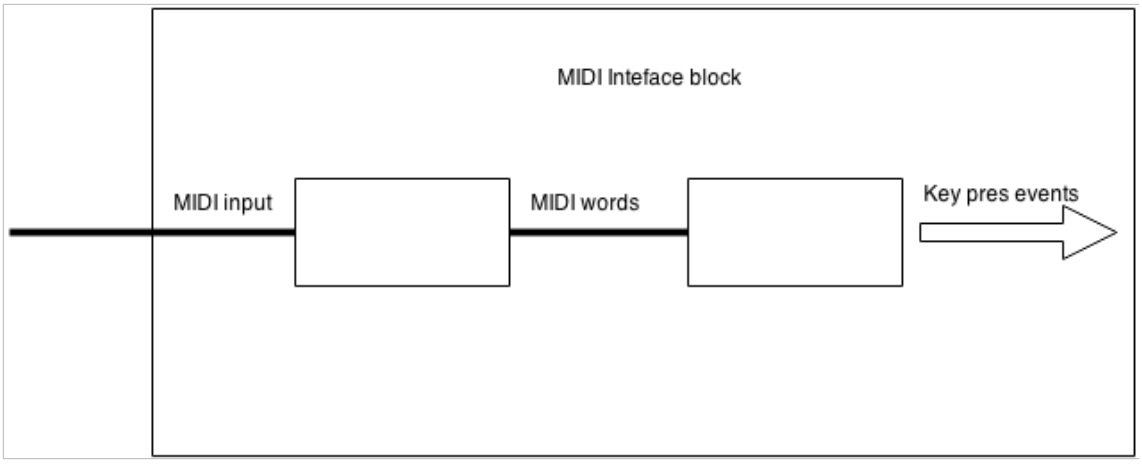

Figure 3. This is the MIDI interface block. It converts raw MIDI input into sensible commands by parsing the input into MIDI words and then parsing those words into MIDI messages. The key press event out is a function of that MIDI message. Only "Note On" and "Note Off" commands are handled.

I had to de-serialize the electrical signal into MIDI words by building a custom asynchronous receiver (Verilog in uart.v). This was a relatively simple matter of sampling the incoming signal, waiting for a start bit, and then reading the eight data bits. Once all eight bits are read, this module outputs them on a data bus and also raises a "data ready" signal for one clock cycle.

The MIDI words coming out of the previous module are fed into a parser module for MIDI. This parser accumulates the words into MIDI messages, and outputs data as a function of the content of that message. It ignores any messages with commands other than "Note On" or "Note Off"; for our purposes, we do not need more complicated musical commands.

After implementing the aforementioned modules, I realized that my particular Casiotone electric keyboard does not perfectly adhere to the MIDI protocol. First of all, it sends words with the least-significant bit first, as opposed to the most significant bit. Furthermore, for "Note On" and "Note Off" commands, in only sends a two-word message rather than three. Confusingly, the command word is NOT the first word to be transmitted; the "key" parameter comes first followed by the command. After adjusting my parser module to account for these quirks (see casiotone_parser.v), I had a working MIDI interface block!

In terms of results, this block worked quite effectively! I can almost always able to successfully interpret the relevant MIDI messages coming from the piano keyboard. In essence, I can tell when a key is pressed or un-pressed, and identify which key corresponds to that action. One unfortunate mistake I made is not oversampling the input. I should have been sampling the 31,250 baud signal at 240k samples per second or more; but I sampled it at exactly 31,250 samples per second. Yes, this is a rookie mistake of a relative EE-novice (I'm a converted Course VI-3). In any case, this caused the system to fail about 2% of the time. This was unfortunate because it resulted in missed key presses. If the "Note Off" command was missed, the system thought that the user was still holding the key. The fix for this problem is quite simple, but I ran out of time before my check off.

Overall, however, I am pleased with the results with regards to this block, and I met my checklist goals. Specifically, my checklist goal was to be able to reliable read "Note On" and "Note Off" commands for a single octave. I actually surpassed this goal since I can determine key presses and un-presses for the entire 61-key keyboard!

Control Block

The purpose of the control block is to run the game, manage the data, and send information to the other modules. Specifically, when the game is in write mode, it processes input from the user and saves records the sequence of key presses and un-presses in memory. When the game is changed to write mode, it reads that data from memory. Specifically, it pushes note data to the interpretation block and to the display block, which will be described later. A complete module diagram of this block can be seen in Figure 4.

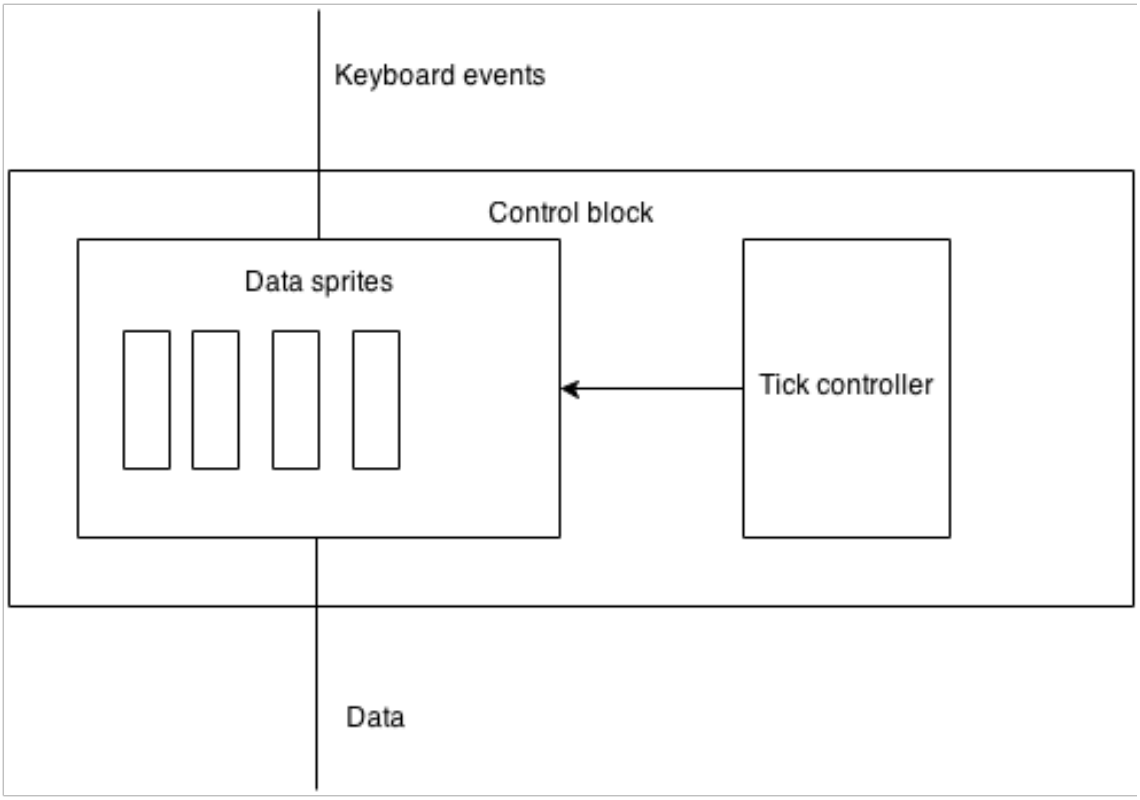

Figure 4. The control block takes in key press info from the MIDI interpretation module so that it can record user key presses in "write mode". It also contains a tick controller that dictates the timing of the entire system. In read mode, data is read from memory and fed to the relevant external blocks.

A key concept is that the control block enforces the timing of "ticks", the most atomic unit of time in the game logic. Think of a tick as a unit of time. The tick controller (tick_controller.v) outputs a high signal for exactly one clock cycle per tick. In the current implementation of the game, a tick is 10 milliseconds. Almost every subsequent module that we will discuss uses this signal. It allows us to specify time scales in a much more sane way than specifying the number of clock cycles, and it provides the hardware plenty of time to process input between ticks. We define the start time and duration of notes in terms of ticks, not absolute time. This also allows us to tweak the tick size, and thus speed or slow the tempo of a song. Furthermore, the tick controller can be toggled off and on (run/stop mode). When it is paused, it outputs a constant low on the "tick" output. When the ticks stop, so too do any modules that wait for ticks to proceed. In this way, ticks are a crucial mechanism in the pause functionality of the game. The user can control this run/stop functionality with a switch on the FPGA.

With this notion of ticks, we are more able to understand the read/write functionality of this block. In the fully polished implementation, I planned to allow the user to "record" a number of songs, and select one to play back. In the actual implementation, given the time constraints, I had to limit the system to holding just one song at a time. The user first enables write mode by flipping a switch. When

the user is ready to record some note data, he puts the game in run mode with another switch. After playing a song for some amount of time (the system can handle up to 40 minutes of play, which I felt was more than enough), the user turns write mode off. At this point, the user enables read mode, and the song he/she just played is displayed on the screen; the notes flow down the screen in sequence and the user can "play back" the song he/she just input. Once recorded, a song can be played back in read mode an unlimited number of times. It is important to note that turning on write mode always clears any note data, leaving the memory empty.

I will now detail how, exactly, I implemented the reading and writing functionality. The core concept is that I use a dedicated sub-module for **each** of the 61 keys. That is, each of these sub-modules records and plays back data for a single pitch; thus we can think of a song as 61 separate tracks, each of which is controlled by a sub-module. This will be a recurring theme in this system; I will colloquially refer to these dedicated sub-modules as "sprites", just as we refer to dedicated graphical objects as "sprites".

I will now detail how the data sprites work. Each sprite stores its notes sequentially as a 2D vector. The vector is 36 bits wide, which is the number of bits required to specify a single note. The first 6 bits indicate the pitch of the note, a number between 0 and 60 inclusive. The next 12 bits indicate the duration of the note, in ticks. Assuming a 10ms tick, this lets us have durations of up to 40 seconds, which is way more than enough. The next 18 bits indicate the start time of the note, in ticks. This allows us for songs that are 40 minutes long, assuming a 10ms tick size. Each sprite stores a vector of these 36 bit notes. In the current design, I make that vector 128 notes long; implying that I think no one note will be pressed more than 128 times. This parameter can be increased with ease.

In both read and write mode, the sprite keeps track of the total tick count so far. In write mode, when the sprite sees that the key it corresponds to has been pressed, it notes the tick count at the time of the press. It then counts ticks until a un-press, and stores that number as well. Combining this information, it is able to determine what 36 bit note should be written to memory. It then increments an index, so the next note will be written into the next slot of memory. In total, this system enforces that the notes are non-overlapping of a single key, which makes sense, and that the notes are stored chronologically in memory. This becomes very valuable in the read stage.

In read mode, the data sprite also keeps a tally of ticks. In also keeps an index, which starts at zero. Think of this index as a "tape head" reading memory. At each tick, the sprite checks if the note in memory at that index has a start time that matches the current tick count. If there is no match, nothing happens. If there is a match, the data sprite outputs the note data at the current index in memory and increments the index. Recall that when I say "memory", I mean this *simulation* of RAM I've implemented using a 2D bit array.

One more important implementation note about data sprites: we need to be performing parallel readings at two different tick counts. That is, the interpretation block, which determines if the user's actions are correct or incorrect, requires that data be provided about 100ms before the note's "true" start time. The display block, however, needs much more advance notice. In order to facilitate these two different

timing needs, I gave the data sprites **two** reading indices (think of dual tape heads), as well as information about how far ahead it should be reading. This way, it performs the exact same reading paradigm as described above, but at two different "tick" times *ahead* of the tick count, so as to inform two different blocks at a time that the block needs. This occurs in parallel, so the implementation was pretty straightforward in Verilog.

In order to instantiate and use all of these sprites, I learned how to use Verilog "generate" blocks, which are like for loops that don't have to be inside and always block. I was very easily able to instantiate and connect dozens of nearly identical sprites. I also built an output buffer for the sprites. Using the buffer, I was able to tell which sprites had information to send. I used a simple 61-to-6-bit priority encoder to pick a sprite that needed to be serviced, so to speak. My overall data_controller.v module then propagated that data, and continued to look for more sprite output. This was my approach to handling a situation in which two sprites give output at the same time (a quite likely event).

This module worked really well for reading. In fact, it seemed to work flawlessly. I, however, ran out of time to debug my implementation of the writing mode, which was problematic for gameplay since the writing mode was the main way to get data into the system. In order to test out the reading functionality, and other system functionality, I hard-coded in some note data, but it was extremely tedious, so I could not really do "stress-testing" on the controller block to see how it would handle lots and lots of data.

Interpretation Block

The purpose of this block is to compare the "true" note data that is being read from memory to the actions of the user. It also identifies which keys are currently pressed, and which of those are currently scoring. This block also contains the scoring module, which tabulates score. See Figure 5 below for a more detailed module diagram!

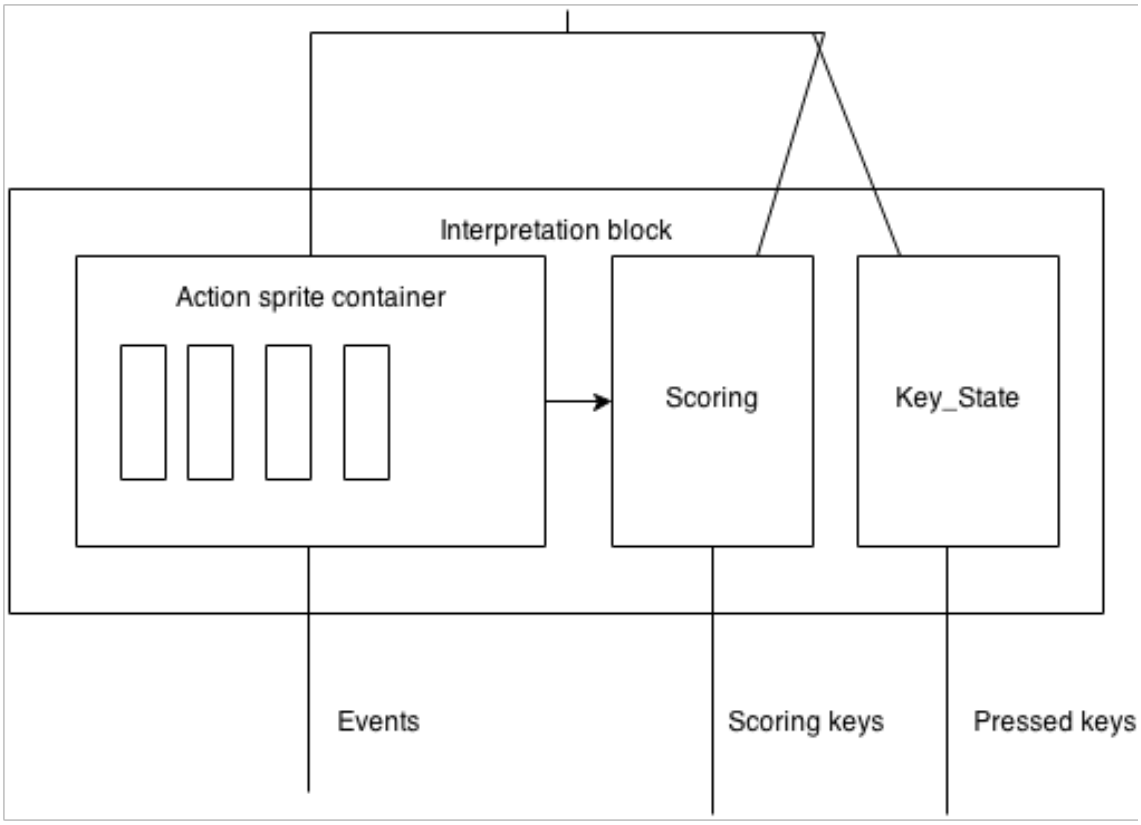

Figure 5. This is the interpretation block. The inputs are key press information from the MIDI interface block and internal data info from the control block. This block combines those two pieces of information to determine which "events", such as GOOD PRESS have occurred. It also determines which keys are currently pressed, and which of those are scoring.

In terms of "events", my system recognizes the following:

GOOD PRESS – The correct key was pressed at the correct time (within a specified tolerance)
BAD PRESS – An incorrect key was pressed
NO PRESS – The user missed a note
GOOD UNPRESS – A scoring key was released at the appropriate time
EARLY UNPRESS – A scoring key was released too early
LATE UNPRESS – A scoring note ended and the user kept pressing the key

Again, I used "sprites" to help out! I had a sprite for each key; thus I maintained my 61-track paradigm. A container module (action_interpreter.v) passes information into 61 sprites (which are instantiated using a generate block). Each sprite gets key press information from the user, and note information from the control block. It performs some basic logic to determine if and when any of the six events occurs, and indicates when one does.

It also determines if the key to which it corresponds is "scoring". We say a key is *scoring* when the user has pressed the key at the correct time to activate this note, the note is still occurring, and the user has not un-pressed the keys. Note that once a user un-presses a key, they can no longer continue gaining points for that note (this is identical functionality to Guitar Hero and Rock Band). Furthermore, if a user is too late in pressing the key at the beginning of a note, they can't just start in the middle of a note and begin racking up score. In fact, any key press that is outside the tolerance limit from the beginning of a note is considered a BAD PRESS. By default, my system gives a 100ms tolerance for pressing a key, and a 200ms tolerance for un-pressing. These parameters can be easily changed via a small alternation to the Verilog source code. Note that these tolerances are defined in terms of ticks, so by pausing the tick controller, we can pause this module's actions.

A module called key_state takes in MIDI events from the MIDI block, and simply keeps track of a bit vector representing which keys are pressed or not. This information is very useful later on, namely in the display block.

Lastly, this module contains a scoring module (score.v). Each good event results in a score bonus (GOOD PRESS and GOOD UNPRESS). The rest result in a score reduction. Furthermore, the user is rewarded points as they hold down a scoring key. I implemented this by counting the number of scoring keys on each tick and adding.

Thus, in summary, this block outputs "events" such as GOOD PRESS, and a 61 bit vector representing which keys are being pressed, and a 61 bit vector representing which keys are scoring.

In terms of performance, I got this module working perfectly in ModelSim, but I unfortunately used some Verilog constructs not supported on the Virtex 2. Thus, my design synthesized correctly, but could not be implemented. I made fixes in the code, but I did not have time to debug these fixes before I had to do my check off demonstration. I am confident that with a little more time, this module would have worked to specification (as defined by my project checklist).

Display Block

The purpose of this block is to create a useful VGA output that informs the user about upcoming notes, and gives the user feedback on any actions performed. See FIGURE, below, for an illustration. Specifically, the system displays upcoming notes, which flow from top to bottom of the screen. At the bottom, a mock-up of a keyboard gives the user feedback. When the user presses a key, that key is highlighted on the screen.

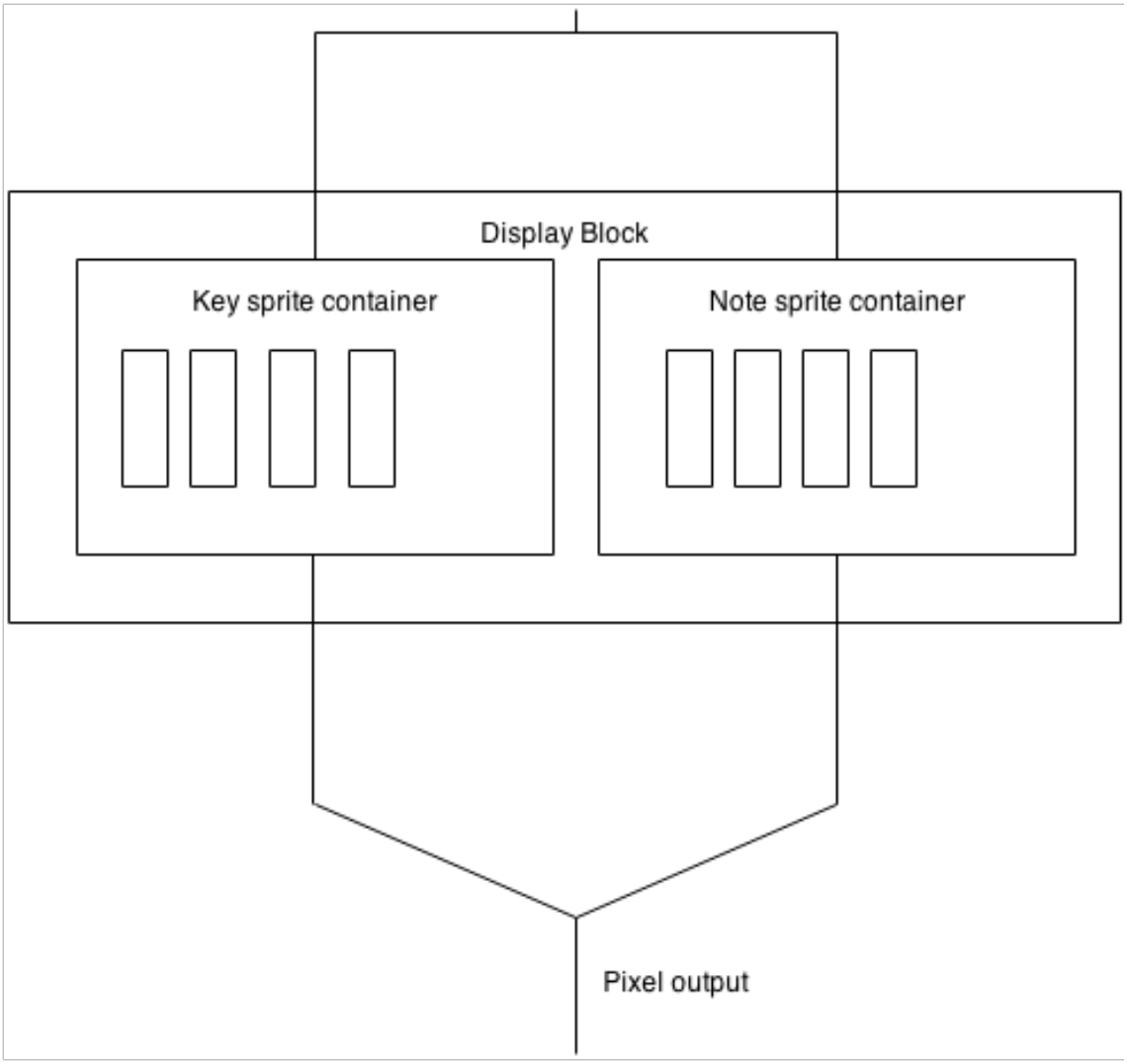

Figure 6. This is the display block. Note info comes from the control block and action info comes from the interpretation block. Both of these sources of information are integrated into a user-friendly VGA output, featuring flowing notes and a keyboard along the bottom of the screen.

Here, I employ a more conventional use of sprites. The key sprites are relatively straightforward since they are statically positioned. A single instance of a key sprite is always associated with a specific key on the keyboard. I model the keys as having one of four different types (BLACK, L, BACKWARD L, T), based on the physical shape of the key. The system renders the sprite by picking which pixels are within the boundaries of the key, and then it picks a color depending on whether the key is currently pressed or not. To decide which pixels are in bounds, it ensures that the pixel was within a bounding rectangle of the key, and NOT within one of the "cutouts" rectangles of the key. Of course, it only looks for a right cutout for "L" keys, a left cutout for "BACKWARD L" keys, both for "T" keys, and none for "BLACK" keys. It combines all the sprites using a big bitwise-OR statement. I also

implemented fancy functionality for color-coding the keys based on pitch (i.e. A#, Eb, etc.) and it worked. In the demonstrated version, I selected all keys to be the default color for the sake of simplicity.

The note sprites were more complicated, since I did not want to do a single sprite for each key. This is because there might (and often are) many notes upcoming for a certain key. I also didn't want to have each key have a fixed number of sprites, because that seems very wasteful in terms of space (61 keys time 16 sprites each is a lot). So the note sprites are key-agnostic. The notes are positioned based on inputs to the sprite. Note data coming in from the control block comes in and tell the sprites (3 seconds ahead of time) which notes are coming up. A note sprite manager picks a particular sprite to render this note, and sends it the data. The note sprite receives the note to be displayed, and starts displaying it on the screen. At each tick, it moves the position of the note down the screen. It also changes the color of the note when it is missed. It can tell a note is missed if it's corresponding key is not marked as "scoring", even though the start time of the note has passed. This is fairly simple for the note sprite to figure out. Just as with the key sprite, a container module combines the outputs from all the note sprites and outputs the appropriate pixels.

One interesting problem concerns the clock. The VGA display module (xvga.v) uses a 65Mhz clock. Throughout my system, however, I use a 27Mhz clock. I expected this to have serious problems with the rendering, but it wasn't that bad. In reality, some pixels have "off-by-one" errors, since my display module misses about half of the updates to the "horizontal" coordinate that is currently being rendered on screen. These off-by-one errors are not very noticeable, and I'm pleased with how the graphics look on screen.

In terms of performance, this module worked, but I would have liked to have done a lot more with it. First of all, color-coding would have added significantly to the user experience, I had built all of the infrastructure for it. Once again, the time constraint of the project proved challenging. There were also some problems with the display with *some* keys rendering incorrectly. I also wish I could have added more helpful graphics, such as vertical guidelines so it is easy to visually tell which key corresponds to a falling note. I unfortunately never got a picture of my project (which I deeply regret), and it is important for me to point out that the picture at the beginning of this paper is a stock photo, not a photo of my system.

Conclusion

Overall, I'm satisfied with my project, but I truly wish I had more time to work on it. The implementation took far longer than I expected. During the process, in any case, I had a lot of fun building and designing the system. This high-level design process is something that is particularly rewarding in 6.111.

I managed to implement most of my checklist goals. I successfully read input from the MIDI port of an electric keyboard, and built an appropriate display (with moving notes) on the VGA screen. This, truly, was the essence of the project. It is unfortunate that I was not able to connect all the pieces to create a great gameplay experience, but I am confident I could have done so. Specifically, a write mode (which was a stretch goal) in which users could input their own song would have

been fantastic. Furthermore, I wish I had made the display graphics a little prettier. Even an extra day would have probably been sufficient to fill these gaps.

The project was more ambitious than I expected. From the get-go, I focused heavily on engineering it well, and making it extensible. For example, even though I only called for a single octave in my checklist, I designed everything to function for the entire range of my electric keyboard. Unfortunately, my progress in the beginning was a little too slow, and the time rush at the end was not quite enough to finish everything. Next time I do a project like this, I will attempt to combine the competing priorities of building a quick minimum viable product, while still keeping the architecture organized and sound. Working alone was certainly a difficulty, since I did not have a partner to keep my ambitions in check.

I feel that I have learned an immense amount of Verilog and hardware design as a result of this project. It may be the most edifying project I've ever done. Several things I learned stand out. First, I learned how to use some really useful Verilog constructs like generate loops. Second, I got used to thinking about computation in a fundamentally parallel way by using sprites. This is not trivial at all for a former Course VI-3, and this paradigm shift is, by far, my greatest takeaway from the project. I also learned a lot about building a big system; my architecture features very deeply nested Verilog, and this really helped keep each module manageable in terms of complexity.

In the future, I'd love to come back to this project to finish it up and make it a truly playable system. I would like to implement multiple songs held in memory at once, a prettier display, and a system free of the current bugs. I've really loved working with FPGA, and I think my system gave me a great cross-section of its capabilities and limitations.