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. INSERT FIGURE HERE. With my 6.111 final project, I saw a new opportunity to gamify piano education.

IMAGE FROM A PIANO HERO GAME

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.

HIGH LEVEL BLOCK DIAGRAM HERE

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.

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