Keytar Hero

Bobby Barnett, Katy Kahla, James Kress, and Josh Tate

***Abstract*—This paper talks about the implementation of a Keytar game on a DE2 FPGA that was influenced by Guitar Hero. It implements a VGA, breadboard, keyboard, as well as a DE2 in order to play the game. The overall result shows falling keys on a frame in the VGA with the song playing along. It also allows the user to play along with the keyboard and gives an overall score at the end of the game.**

I. Introduction

THIS project implemented a version of Guitar Hero that used the DE2 for the processor and score keeping, a keyboard to represent a guitar, a breadboard to create sound, and a VGA monitor to show the game play. In the game of Guitar Hero, players match notes that scroll on-screen to colored fret buttons on the controller, strumming the controller in time to the music in order to score points.

While playing the game, an extended guitar neck is shown vertically on the screen (the frets horizontal), often called the "note highway", and as the song progresses, colored markers or "gems" indicating notes travel down the screen in time with the music (Figure 1); the note colors and positions match those of the five fret keys on the guitar controller (Figure 2). Once the note(s) reach the bottom, the player must play the indicated note(s) by holding down the correct fret button(s) and hitting the strumming bar in order to score points with a window of time in which the player can hit the note and receive points. Notes can be a single note, or composed of two to five notes that make a chord. There are four difficulty levels ranging from Easy to Expert which is determined by the number of notes and how many fret keys are being used.

Figure 1: Example of Guitar Hero game board

Figure 2: Example of the Guitar Hero controller

In order to implement Guitar Hero we needed similar graphics, we needed the graphics to move, we needed a controller to play the notes, and we needed a way to play the sound for the game. With this in mind we created graphics similar to the existing game play which included easy and hard modes and is discussed further in the Graphics section. We used a key board with keys F1 to F5 representing the five fret keys and F12 as the strumming bar (Figure 3) which is discussed further in the Keyboard section. For fun, we also decided to implement a cheat code that awards the user 5000 extra points at the beginning of the game. The current cheat code is “arlen” though it could be changed to anything the user wanted.

Figure 3: Example of the keyboard guitar controller

Since this program has many features, we included a user guide which can be seen in Figure 4.

II. VGA

The VGA was mainly coded by Bobby, Katy and Josh.

We first created the vga_regs.h file to get access to the VGA registers addresses from the system.h file. Then we took an image and converted it to a pixel array using the Matlab program provided and attempted to map it to the vga screen. We were able to do this by mapping each element of the resulting picture array to a 640x480 2D array and then storing it to the VGA→pixel register. We tried changing the colors on the image a couple of times, but couldn't get it to work and moved on. Josh created images for us to use as the main game screen (Figure 5), while Bobby converted them to arrays and implemented them into the new splash.h file. The original plan incorporated round colored notes, falling down the screen one row at a time. To test this, Bobby made a graphics array that represented one row of notes (Figure 6).

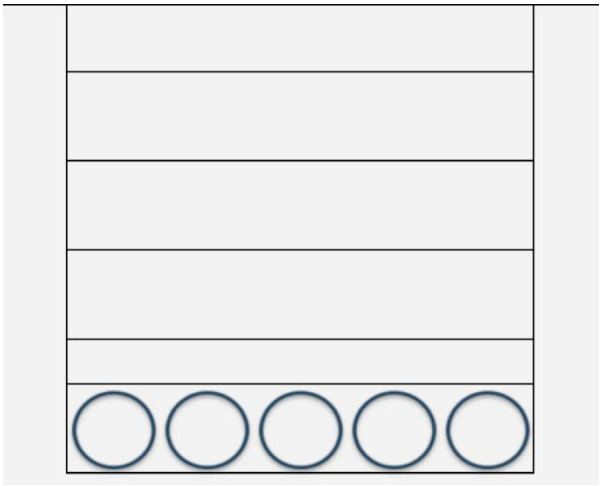

Figure 5: Example of original frame

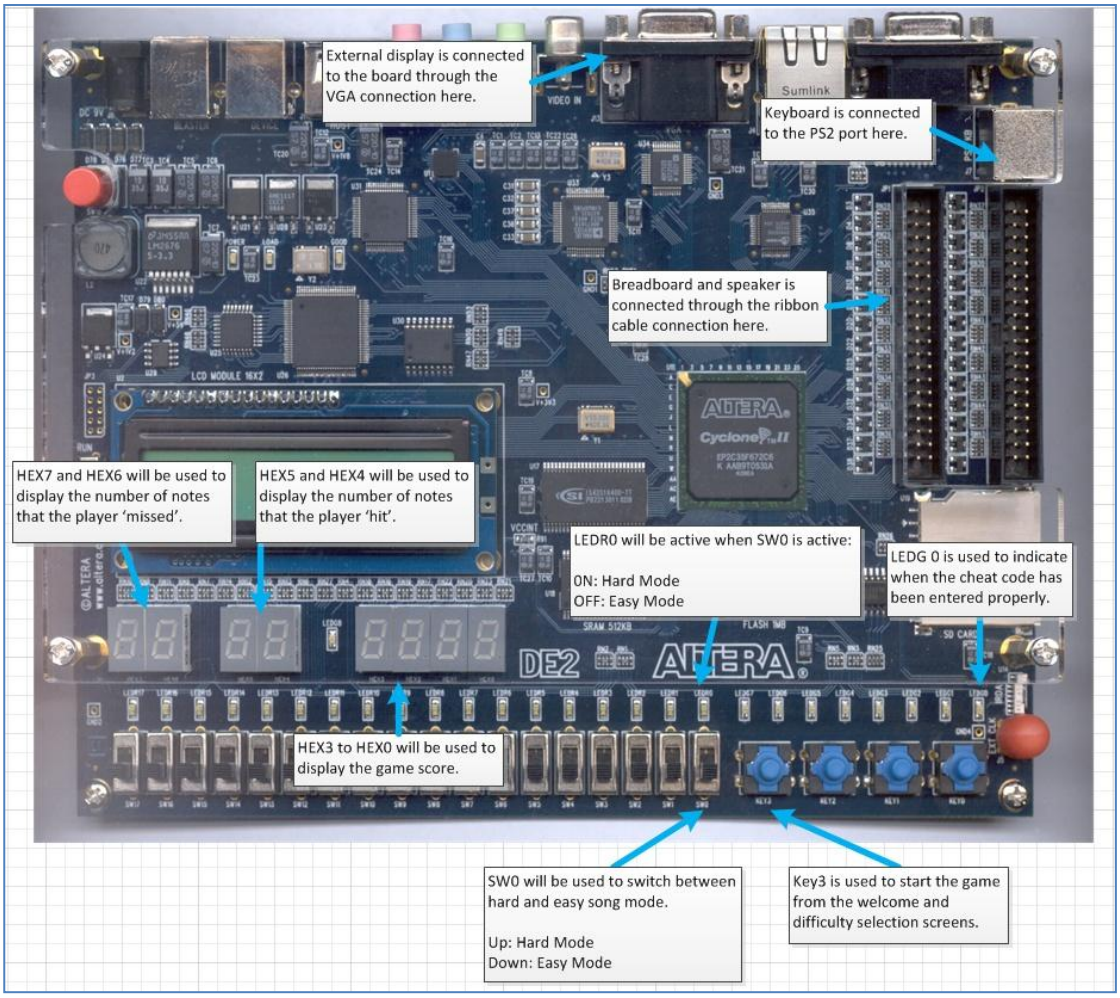

Figure 4: Overall design of DE2 showing how all the feature work

Figure 6: Example of original note

The idea was to display the row of notes (Figure 5) inside the lines of the splash screen and have it move down the screen. The method to do this was "and"ing the main game graphic with the new overlay of notes. This is where we encountered "tearing" of the image. This was where the image would be slanted and broken up across the screen but correctly placed on the screen. Since it wasn't clear that this behavior could be expected or why it was happening, Bobby recreated the picture arrays and rechecked all the assignment statements, taking great care to make sure all the bits were lining up. After determining that the bits were lining up correctly, Bobby worked to attempt to "correct" the shift by manipulating the arrays by trying to shift bits either to the right or left checking the alignment of the image. None of the solutions worked although for this we did end up writing some code to "move" the row of notes down the screen which was later incorporated into the final version of the lab.

We then, at Josh's suggestion, temporarily fixed the issue by making the row image a full 640 pixels wide (by 85 pixels high). This gave us the round notes we were looking for and allowed us to move on to color. We saw from this however, that it would take a great deal of time not only to load the images, but to paint the images each time we wished to move a row down the screen.

Once we had a base frame and a way of moving the notes down the screen, we began working on coloring each not to a different color. However, we made little progress with this. One problem we ran into was the fact that our first processor didn't have colors enabled on it. Katy and Bobby spent about 12 man hours between reading through data sheets, manuals, system.h file and experimenting with changes to the color registers. From this we were able to learn how to turn colors on and off and how the cursor was working.

When we received the new processor, we were able to begin experimenting using colors for the VGA. We also spent some time manipulating register values as the data sheet for the color-supporting processor update was incorrect (the bit positions were reversed). After we fixed this problem, we helped other people get theirs working to save them the experimentation time caused by bad documentation. We spent a couple of hours trying to determine how to get multiple colors to display at once. We assumed since it was a VGA display that we could expect to get at least sixteen colors. We finally realized, about the same time we were told, that it was only possible to display two colors at once though it was possible to "layer" colors to give the illusion of more than two colors.

Since we wanted our Keytar Hero to resemble Guitar Hero with the color scheme and we were not able to have more than two colors, we decided to redo our graphics that would instead show which key to press on the note instead of a color scheme. From this, we also decided that the note graphics needed to be square so we could calculate the note alignment better. We also needed to make the notes smaller to increase the speed of repainting the screen. Josh created these images and used the same technique of "and"ing as before to move the graphics correctly down the screen.

During this iteration, we paid very close attention to the size of the array and the precise location we needed the notes to land. Bobby also added another check to make sure the index of the picture arrays didn't go out of bounds. These calculations were done using Josh's Excel talents. He filled each cell with an array index and specified the height and width. This gave us a zoom-able view of our graphics as well as the precise location of all the corners and edges that we needed to prevent tearing. After this was done, we moved on to song creation and created the note patterns for our two songs, Twinkle, Twinkle Little Star (easy mode) and Binary Solo (expert mode).

In our final implementation of Keytar Hero, we were using solely black and white coloring and turned on optimization to make the notes fall faster down the screen. We created a frame that would always be displayed and we would paint the notes and have them move down the screen. This is discussed in greater detail in the Graphics portion of the report.

III. Graphics

This part of the project was mainly coded and designed by Josh with assistance from Bobby and Katy.

The initial graphic design mock-ups took place in MS Excel. We used excel to generate a simple graphic for each "button" and the frame of the game. After running the pictobit.m file supplied to us we were able to also copy the bits back into excel and this gave us a very good view of how the pixels were going to be laid out on the screen. We were able to use this view to assist in programing our VGA code on were we wanted to start drawing certain objects on the screen and also generating our loop offset while playing our song. We started with the 2-d version of the graphics to make things simple and to make sure our VGA code was going to work. So with our graphics we determined the best place to paint each note on the screen and the offset throughout the song to represent the falling note.

Early on in the development of graphics when we were still trying to determine the best way to draw on the VGA we thought that it might be easiest to just have a very big array representing the song as bits on the screen. Instead of having just a 640x480 array for the VGA we thought we might be able to import a 640x6000 (for example) array. This array would be a complete mock-up of the song. When we started the song we would see the first 640x480 bits and then basically repaint the song 1-20 "sliding bits" at a time. The width would always be the same and we would basically be slicing through one huge array until the end of the song. We quickly realized that the size of the image would be too big and that speed would be a huge concern as well. If we wanted to proceed with this route the program would run very slow because we would have to have a "frame" image of 640x480 that we would have to "and" with the song array. This would cause slow looping for re-painting throughout the song as well. We abandoned that idea and stuck to the frame image and a separate image for notes F1 – F5. When we implemented the drawing in this manner it went much

smoother. Figure 7 shows the frame and Figure 8 shows the notes that were used.

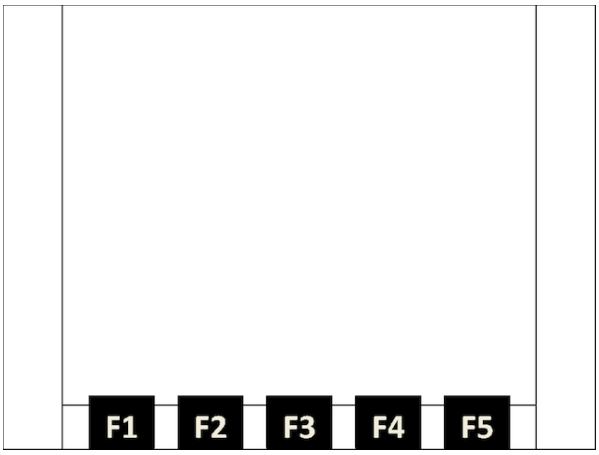

Figure 7: Example of the frame used for Keytar Hero

Figure 8: Example of the notes used

These are the 6 basic images used that make up the main game play. We then shifted our focus to writing complete songs, the timing of the songs and how we going to make the program work with only these 6 images. By the time we were comfortable with the program timing and song play we ran out of time in implementing a 3D view of the game. Although we know this could be done by generating different sized notes, and because it doesn't paint as smooth and fast as a 2 GHz processor could the images jump down the screen instead of scrolling smoothly. If this wasn't finals week and we had another good chunk of time we could have implemented a 3D version of the game, making it look like the notes are coming closer to the player instead of just falling down the screen. A screen shot of a section of Twinkle Twinkle Little Star is shown below. The 3D version was given as a bail out option in our original write-up of the program. Unfortunately it was only dropped due to lack of time, and not the inability to implement the idea. Figure X shows what the 3D version would have looked like.

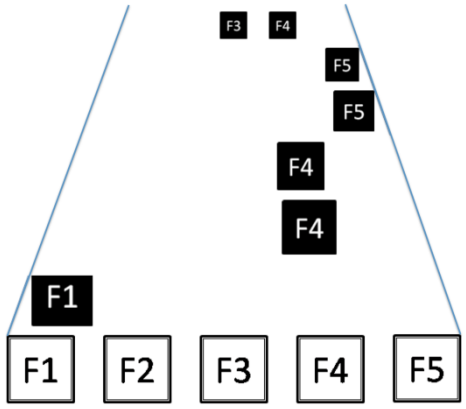

Figure 9: Example of 3D graphics that we were not able to implement

A. *Song Writing and Playing*

In order to keep graphics at a minimum we wanted to write a song using our 5 basic notes and being able to call a combination of the notes for the songs. To do this we made a switch statement for each combination of notes and a default case of 0 notes to paint. We will get into painting and timing a little bit later. To write a song we use a total of four arrays. We could have cut this down to three arrays if we had known the keyboard key values before setting up the switch statement and if we were re-writing this would be fixed. At this point we stuck with the following four arrays:

- Notes: This array uses the switch statement to determine the combination of notes to paint to the screen
- Delay: This array gives the row value of where to start painting each note in the song. Basically we start the song on the 9th row of the VGA and consider this "time 0" of the song. Each note after that gets a delay from time 0.
- Keys: This array tells the program what combinations of key presses on the keyboard are expected for each note of the song. This array is needed to determine how well the player did throughout the game.
- Sounds: This array tells the program what each note of the song sounds like.

With the combination of these four arrays we can iterate through each and know exactly what sound to play, what note to display, where to paint it on the VGA and if the player gets points for the note. The difficult part in playing the song was making the note stick around from the time you first see it on the screen to the time you need to play it. We run the array using the index we are currently pointing to minus eight. The minus eight note is the note we are currently trying to play and get points for, the other indexes represent the seven other rows of notes painted to the VGA.

IV. KEYBOARD

Another major component of the Keytar Hero project was the integration of the keyboard. This required creating several new classes, as well as being able to communicate over the DE2's built in PS2 connection. This part of the project was

mainly coded by James with occasional help from Katy and Josh.

The first thing that needed to be figured out about the keyboard was how to address the PS2 port. To start, we created a new C class and header, and put in the usual method of addressing a device. By creating a struct of the required registers in the header file and then addressing the correct address space, which was found in the system.h file. There was one register that needed to be checked and used for the PS2 connection, the control register. This register had three sections, an eight byte code section, an eight byte space, and a sixteen byte ravail section. With this created, we were able to create the ISR to handle the keyboard interrupts. Our design concept on this, was to have the ISR do all of the necessary calculations and checks to identify the key press that has taken place. This did mandate that the ISR be larger than normal, but it simplified the checks and overhead that went into detecting key presses. If we had not done it this way, we would have had to of created a ring buffer, or other storage device to store all of the key press events, and identify them at a later point. This would have been less efficient, and would have mandated further coding.

Once a key press is detected and deciphered in the ISR, the press event values are available in the main class. One design problem that we had with this though, was the fact that the keyboard actually generates three key press events for every key that is pressed. The first number is the key down code, next is the key press code, and finally, the key up code. The first number was used to identify the key that was pressed, and to turn on the correct key in the structure of keys that we created. The structure of keys is a struct of all of the keys that we are monitoring for the program. The second number that we generate is the key code from our ISR, and lastly is the key press terminator, which indicates that all keys have been released.

After a key press event, and after the key has been identified in the ISR, program control returns to the main method, where they key press is retrieved, and checked against the key that was supposed to have been hit at that time. To make the keyboard more like a guitar, in addition to the correct key needing to be pressed, the F12 button also must be hit at the exact same time. This method allowed us to generate a unique key code for every set of notes in the game. These unique codes were stored in an array so that when the key press is retrieved it can be checked against these values to determine if the correct key had been hit in the correct sequence.

In the event that the correct key was hit in sequence with the VGA, then the score will be incremented on the HEX3 to HEX0 display (Fig. X). Each time that a correct note is hit, the score is incremented by 100 pints, and the ‘hit count, which is displayed on HEX7 and HEX6 (Fig. X) will be incremented by one. If it was a miss, then the ‘miss count’, which is displayed on HEX5 and HEX4 (Fig. X), will be incremented

V. Sound

The starting point for sound generation was the original PWM that we had already designed in previous labs. We started with this implementation for sound, and used it for most of the project as we designed the interface for the keyboard, worked on the VGA, and got all of the timing protocols down for the keyhits. After we had all of this created and functioning we decided to start working on sine waves. In the end, we had created a class that produced pseudo sine waves, but we didn’t implement it because we were unable to clean them up any more in software, and did not have the hardware or hardware knowledge available to clean them up.

The first thing that we did as we began to create a sine wave was create a new class. We created the wave generation class to attempt sine waves in software. This class had an array of ‘sine’ numbers that was used to vary the height of the wave, by adjusting the duty cycle of the PWM. Under optimal settings, this worked very well, and produced waves on the monitor that looked similar to multiple square waves overlapping. We later found out that this really dirty wave was actually what the software pseudo sine waves were supposed to look like. However, once we found this out, we were unable to find the necessary hardware or expertise to clean up the sine waves in hardware. Unfortunately, that means the waves were not implemented in the final iteration of the project, but the capability in the software remains. Although it hasn’t been tested, and may, in fact, sound horrible even when using the correct hardware, the logic behind our software implementation was sound (no pun intended :) and should produce a rough estimation of a sine wave.

VI. Conclusion

Overall, this project was a success. We were able to implement not only the keyboard, the seven segment scoring system, sound, and VGA functionality, but also integrate each component into the larger project with great success. The hardest part of the entire project was in getting the scoring system to read and identify that a correct key had been pressed while the VGA was painting. After we figured out how to incorporate the scoring loops during painting however, we were able to make the easy mode of the game easier and the expert mode of the game very hard. In all, the game was a lot of fun to design and it was nice to end up with an actual playable game when we were done. Everyone seemed to respond positively to it and a few people provided valuable comments in assessing the game’s playability. After the excitement generated by playing version 1.3.4.33.2 of our keytar video game, we decided to incorporate a different score algorithm (in version 1.3.4.34.3) that will increment the score based on how close you get the hitting the keys at the perfect time. This should all culminate in the release of version 9.0.0.0.0 sometime around Christmas 2019.