

Guitar Hero: Nursery Rhyme Edition
Emily Hwang, Judy Ho

Abstract

Guitar Hero: Nursery Rhyme Edition is a guitar simulation game with a computer keyboard acting as a substitute for the guitar. The objective of the game is to play a nursery rhyme song in tempo with the rhythm by hitting the notes correctly through holding the respective keys and pressing the Shift key as an act of strumming. The game system consists of four major components, the keyboard interface, display, game logic, and audio. The game logic manages the overall functionality of the game by connecting the three other main components.

Table of Contents

List of Figures.....	2
1 Overview.....	3
2 Description.....	4
2.1 Keyboard Interface.....	5
2.1.1 Keyboard Module.....	5
2.1.2 PS2 Module.....	5
2.2 Music Memory.....	6
2.2.1 Music ROM.....	6
2.2.2 Song Selection Module.....	6
2.3 Display.....	6
2.3.1 SVGA Module.....	6
2.3.2 Display FSM Module.....	6
2.3.2.1 Character String Display Module.....	8
2.3.2.2 Binary to String Converter Module.....	8
2.3.2.3 Start Menu Module.....	8
2.3.2.4 Playing Display Module.....	9
2.3.2.4.1 Blob Module.....	10
2.3.2.4.2 Streak Module.....	11
2.3.2.4.3 Blob Outline Module.....	11
2.3.2.5 Song Over Module.....	12
2.4 Game Logic.....	12
2.4.1 Game Logic Module.....	12
2.4.1.1 Divider Module.....	13
2.4.1.2 Timer Module.....	13
2.5 Audio.....	14
2.5.1 Final Audio Module.....	14
2.5.1.1 Direct Digital Synthesizer.....	14
2.5.2 AC97 Module.....	14
2.5.3 AC97 Commands Module.....	14
3 Testing and Debugging.....	15
4 Conclusion.....	17
References.....	19

List of Figures

Figure 1. Overall Block Diagram	4
Figure 2. FSM and Look-up Tables within the Keyboard States.	5
Figure 3. Finite State Machine for the Display	7
Figure 4. Block Diagram for Display to Create Pixels	7
Figure 5. Start Menu Display with Highlighted Selector	8
Figure 6. Playing Display of Scrolling Notes with Various Lengths	10
Figure 7. Display of Keyboard Fret Buttons Being Pressed	11
Figure 8. Song Over Display	12
Figure 9. Data flow to and from AC97 and AC97 Commands modules.	14

1. Overview

This project implements a version of the popular game Guitar Hero. The project is composed of visual, audio, and interactive parts. The goal of the game is to be able to “play” the notes of a song chosen from a given list by pressing the appropriate buttons on the keyboard to match the ones on the screen.

On the PS2 keyboard, keys 1-8 are used for the fret buttons that represent the notes and the Shift key substitutes for the strumming of the strings of the guitar. This allows the player to hold the keyboard like a guitar. Shift must be pressed for the note to be interpreted as “played.”

Using a direct digital synthesizer, eight different frequencies are created to fulfill an octave of notes and a buzzer frequency is created to indicate incorrect notes played by the user. These frequencies are stored in one ROM and are only outputted as necessary. Simple nursery rhyme songs Mary had a Little Lamb, Twinkle, Twinkle Little Star, Row Your Boat, and Chopsticks are created from music sheet using the respective notes. These songs are stored in a ROM since there is no need to write into memory. Each address in the ROM consists of a 12-bit word. The 4 MSBs represent the duration of the note while the other 8 bits determine the notes to be played. The song is heard from the audio as the user plays the correct notes.

The visual display includes the start menu, the playing display, and the end display. The start menu contains all the songs available and allows the user to select and start a song. The playing display is the interactive game display that includes scrolling notes of the selected song, a matching zone, an indicator for the keys being pressed, and an updated score that keeps track of the points depending on the player’s accuracy. In the playing display, at most 6 notes of the song are displayed at once and scroll to the bottom of the screen as they should be played. The display also represents rhythm with longer rectangles to show notes of longer duration. The note should be played with the keyboard when the note reaches the matching zone at the bottom of the screen. An additional feature informs the user visually of whether the note is played and for how long it is being played. The game also includes different levels of difficulty, which involve increasing the speed of the songs and a smaller detection window to hit the note accurately.

The Keyboard Interface, Music Memory, Display, Game Logic, and Audio are the main underlying components. Figure 1 displays the block diagram that integrates the underlying modules.

Keyboard Interface

A PS2 keyboard that simulates the guitar controller is interfaced with the labkit. The Keyboard interface outputs the keys pressed as a nine-bit piece of data to represent 1-8 and the shift key. The MSB represents the shift key.

Music Memory

- *Music ROM:* The Music ROM will contain all the different song selections that we want to include for the user. It outputs the encoding for the notes to the Display module and it will take an address input from the Display to select the correct notes within the memory each time the address increments.
- *Song Selection*
This module allows the user to decide what song he would like to play. The switches, switch[1:0] are used to indicate which song has been selected, and it will be referenced to an assigned song within the module that would indicate the appropriate start and end addresses for the song. The switches are the input and the start and end addresses are the outputs to the Display.

Display

- *SVGA*: The SVGA generates the necessary signals for video display. These signals include hcount, vcount, hsync, vsync, vclock, and blank, which are sent to the Display.
- *Playing Display*: This module displays the scrolling notes to be played. These notes are received from the Music Rom. The Display is given the inputs of the start address and end address from the Song Selection module. The current note received will be newly displayed at the top of the screen while the previous notes will continuously scroll downwards until it reaches the bottom of the screen.

Game Logic

This module determines the note to be outputted to the audio module. The output note is based on the input of the keys being pressed and the expected note that would be currently in the designated area of the Display. The Game Logic also keeps track of the total score.

Audio

- *Audio*: This module only has one input coming from the Game Logic. This input is the note encoding that is mapped to the appropriate frequencies. This frequency is then outputted to the AC97 module.
- *AC97*: This module transmits the data received from the Audio module in order to output it to a speaker that is plugged into the labkit.

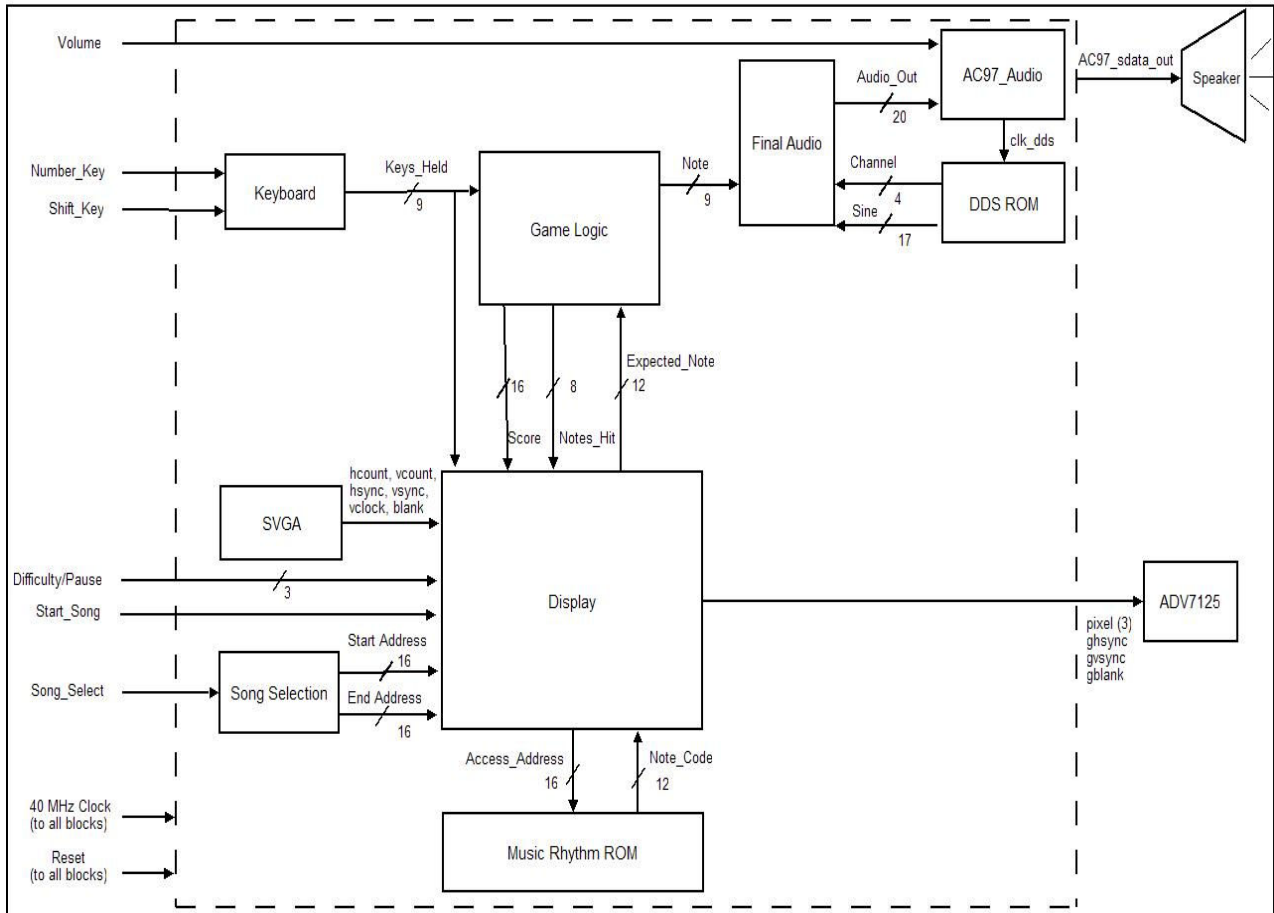


Figure 1. Overall Block Diagram

2. Description

The Guitar Hero: Nursery Rhyme Edition system is made of five major subsystems. These systems include the Keyboard Interface, Music Memory, Display, Game Logic, and Audio.

2.1 KEYBOARD INTERFACE (by Judy Ho)

2.1.1 PS2 Module (C. Terman and I. Chuang) [3]

When keys on a computer keyboard are pressed, certain codes are sent to the central processing unit to determine which keys were actually pressed. Each key has a unique make code, while every key has the same break code to indicate release. When a key is pressed, a make code is sent to indicate the pressed state. When a key is released, the break code is sent, followed by the make code for the key. This allows the system to know not only that a key was released, but which specific key was released. This module was created to read the codes coming in from a PS2 keyboard. The data coming in is put into a FIFO, as it is received.

2.1.2 Keyboard Module (JH)

The keyboard module allows the user interact with the rest of the game through the keyboard interface. A small, two-state FSM is created in this module to decide whether keys are being pressed or released. Each of these states contains one look-up table that sets the state of each key as on or off, outputting the keys as a nine-bit piece of data to represent 1-8 and the shift key. The MSB represents the shift key.

The FSM uses the data that is being pulled off the FIFO from the PS2 module to help determine the state of each key. The two states of the overall FSM are `READY_ALL` and `READY_RELEASE`. In the `READY_ALL` state, if the make code for a key is detected and found in the look-up table, that key is turned on. If the break code, `F0`, is pulled off the FIFO, then the overall FSM moves to the `READY_RELEASE` state. The look-up table within that state will compare the next piece of data from the FIFO with its contents and turn off the appropriate key. The FSM moves back to the `READY_ALL` state whether or not it received a relevant make code. The lookup table is shown in Figure 2.

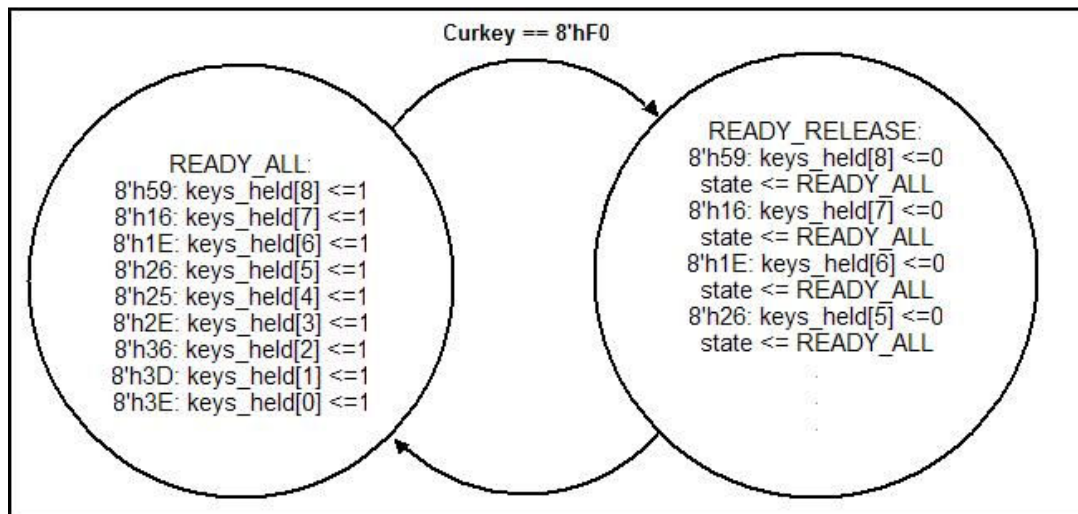


Figure 2. FSM and Look-up Tables within the Keyboard States.

2.2 MUSIC MEMORY (by Judy Ho)

2.2.1 Music ROM

A COE file is first created to be loaded into the ROM. The COE file consists of all the notes that need to be played on each beat. Each piece of data is 12 bits, with the first 4 bits representing the duration of the notes, and the last 8 bits representing which notes are supposed to be played. If the bit is high, then the note needs to be played. A ROM is then generated using the Xilinx tools with this preloaded COE file. The data within the ROM can be read out from the data port while an address selection is inputted through another port.

2.2.2 Song Selection Module

The song selection module allows the user to choose which song he wants to play from the ROM. It also keeps track of the start and end addresses of song since they are all stored in one ROM. This module takes in switches one and zero as inputs to choose one of the four songs in the ROM. Upon selection of a song, its start and end addresses are sent to the display module to begin displaying the corresponding notes from the ROM.

2.3 DISPLAY (by Emily Hwang)

2.3.1 SVGA Module

To create a video image for the game, an SVGA was created for an 800x600 pixel resolution at 60Hz refresh rate, which requires a 40MHz pixel clock. This allows 25ns for computation between each frame. Based off the XVGA, the SVGA creates an hcount, vcount, vsync, hsync, and blank signals using the appropriate active video, front porch, sync pulse, and back pulse for the horizontal pixels and vertical lines. Hcount counts pixels in a horizontal scan line while vcount counts scan lines in a frame. Hsync indicates the end of each horizontal scan line while vsync indicates the end of the frame. Blank also indicates when a pixel value will be displayed or when it will be off the screen.

2.3.2 Display FSM Module

The Display module is an FSM itself, which switches between the Start Menu, Playing Display, and the Song Over Display. At power on and reset, the Start Menu is displayed. This is where the user must select their song with switch[1:0] before pressing the Enter button to start the song.

The FSM switches to the Playing Display when the Enter button, Start Song, is asserted. This contains the actual interactive game with scrolling notes. The Playing Display contains a Continue_Song signal that is asserted low when the incrementing address reaches the end address specified by the Song Selection. Since the Continue_Song is asserted low when the last note is retrieved and placed at the top of the screen, there still requires time for the note to scroll down to the bottom because the song is still in play. Therefore, the Display FSM uses a timer that starts when the Continue_Song is asserted low when the song is over. The Display FSM changes to the Song Over state when the timer expires. A second timer is then started when transferring into the Song Over state. In the Song Over state, the congratulations banner is shown and the final score is displayed. When the second timer expires, the state transfer back to the Start_Menu. The user is also allowed to press the Start Song button again in the Playing Display

or the Song_Over display if they wish to restart the song. This finite state machine is shown in Figure 3.

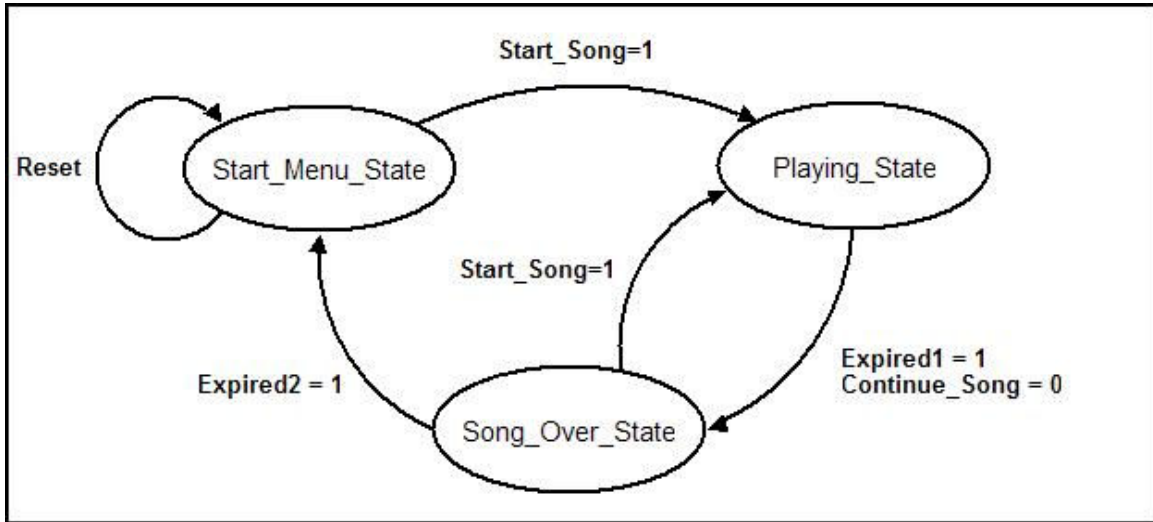


Figure 3. Finite State Machine for the Display

The Start Menu, Playing Display, and Song Over modules create pixels for each stage while the Binary to String creates the score pixels necessary for the Playing Display and Song Over modules. The Display FSM decides which pixels to display depending on its state to create the current output pixel, as shown in Figure 4.

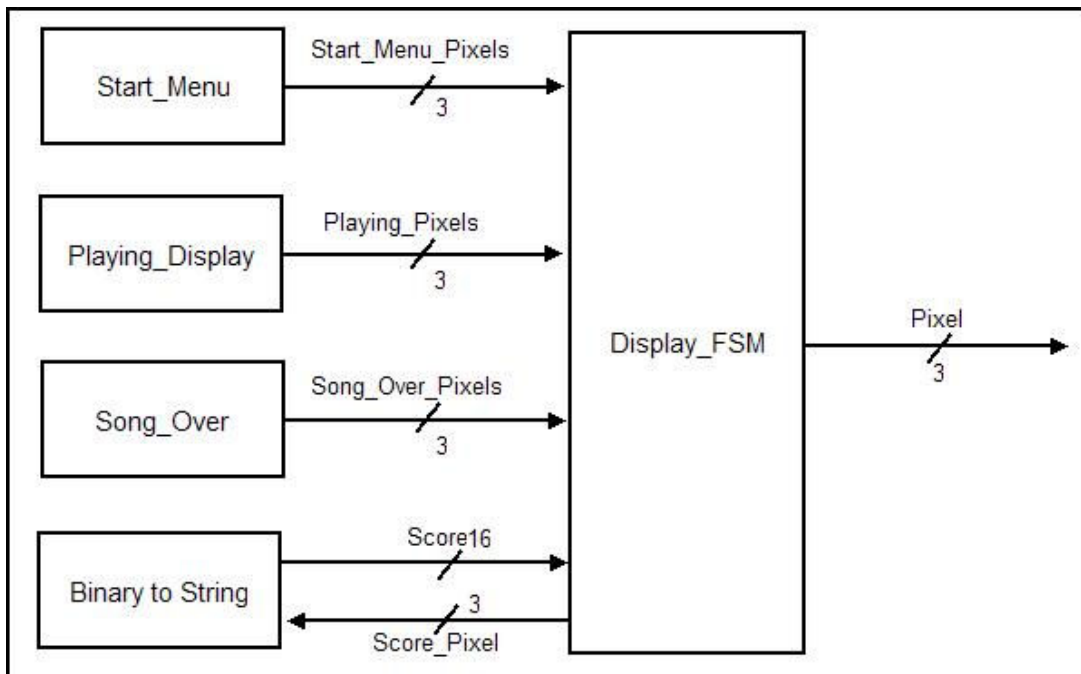


Figure 4. Block Diagram for Display to Create Pixels

2.3.2.1 Character String Display Module (C. Terman and I. Chuang) [3]

The Character String Display Module uses a font ROM to display strings. ASCII encoded character strings are displayed in a video window at a specified x,y pixel location. Each character is 8x12, but the pixels are doubled horizontally and vertically in order to magnify the fonts by 2. The Character String Display Module is used by the Start Menu module, Song Over Module, and the Binary to String Converter Module.

2.3.2.2 Binary to String Converter Module

The score display required calculating the number string for each decimal place from a binary number. Therefore, the Binary to String Converter (hex_to_decimal.v) was created to take in a score and display the string in decimal format. The Binary to String Converter is similar to an odometer and uses counters for each place. If the counter of the lower bit increases from 9 to 0, then the next bit increases by 1. The counter increments the amount of times specified by the score. Each counter is then mapped to a number string depending on whether it is a number from 0 to 9. Therefore, each digit of the score is its own Character String Display.

2.3.2.3 Start Menu Module

The Start_Menu primarily uses the Character String Display Module borrowed from the course website. For each of the eight lines of text, a Character String Display is instantiated. Therefore, however, there are eight font ROMs that are created, which could be improved in the future. The lines of text inform the user of the name of the game, the songs to select, and instructions to play the song by pressing enter.

The user must select the songs using the switch[1:0]. Changing the switch will also change the highlight bar, which is a blue colored blob that changes its y position depending on the selected song chosen through switch[1:0]. Figure 5 displays the strings of the Start Menu and the blue highlight bar that switches positions based on the song selected.

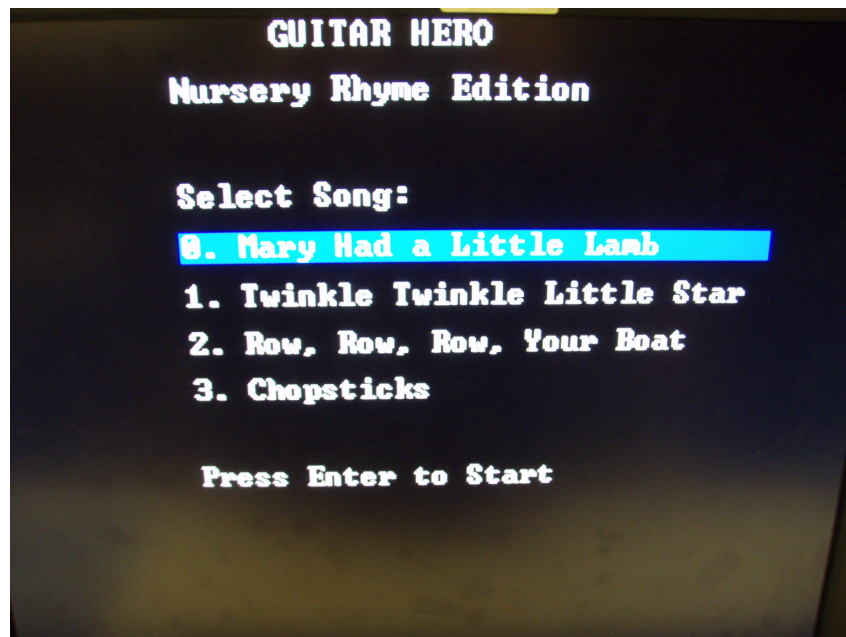


Figure 5. Start Menu Display with Highlighted Selector

2.3.2.4 Playing Display Module

The Playing Display module displays the scrolling notes to be played. It at most displays 6 notes. These notes are received from the Music ROM. The current note received is newly displayed at the top of the screen while the previous notes continuously scroll downwards until it reaches the bottom of the screen. Therefore, the Playing Display increments its address from the start address to the end address specified by Song Selection to retrieve the appropriate notes.

Once the user presses Start_Song, the start address and end address specified by Song Selection are kept on the internal signals address_init and address_final. This is done so that if the selection, switch[1:0], are changed in the middle of the song, the change of the start and end address from Song Selection does not affect the current address in the song being played. The address is incremented when a new note is entered at the top so that the next time the note must be changed, the next address retrieves the subsequent note from the Music ROM. This is done until the address reaches the end address.

The Playing Display contains its own FSM, which specifies which register to enter the new note. Originally there were 6 registers, but with the additional of rhythm, 9 registers were needed in order to continually display a note up to a whole note. Therefore, there are 9 y positions that specify each of the 9 registers that hold information about the last 9 notes of the song being played. These y positions cycle from top to bottom so that when a new register is needed, the register that contains the oldest note played is reused to hold the latest note of the song. Nine states are used in order to enter the new note into the correct register. For example, when register8 is filled, register7 will always be filled in next. A pulse signal Change_Note specifies when the next note should be entered into the new available register.

The signal that determines when the note must be changed is the Change_Note signal. This signal depends on the y position of the latest note and the state, which specifies which register the to-be-new note should enter. For example, if we are in state3, the newest note was entered into register4, which is specified by y4. Therefore, we ask if y4 is greater or equal to the standard y position of the 2nd row. This allows for the Change_Note signal to depend on the position and therefore speed of the scrolling notes. It then allows implementation of difficulty of the song, which increases the speed of the notes and shortens the time to accurately play the note. Difficulty is specified by switch[5:3].

The game's scoring is implemented so that the more accurate the user is, the more points will be accumulated. The maximum points obtained from a single note occurs when the squares align perfectly. Otherwise, the more the square strays away from the outline, the less points received. One could still receive at least one point if the very edge is aligned with the outline. The Playing Display module therefore outputs a score_worth signal that informs the Game Logic how many points the score is worth at a given moment. This score_worth depends on the y position of the blob that is crossing the matching zone. Since the bottom of the blob, y+height, could cross the y position of the matching zone outline, this is also taken into consideration. The area of overlap is calculated to create the score_worth so that perfectly matching would require the blob area to entirely overlap the outlined blob. The difficulty is also taken into consideration for the score_worth where higher difficulty will allow for more points to be accumulated.

In the actual display, the scrolling notes use the Blob module and the matching zone uses a Blob Outline module to outline where the user must hit the note when it scrolls through. Figure 6 displays a screenshot of the Playing Display with scrolling notes of various lengths.

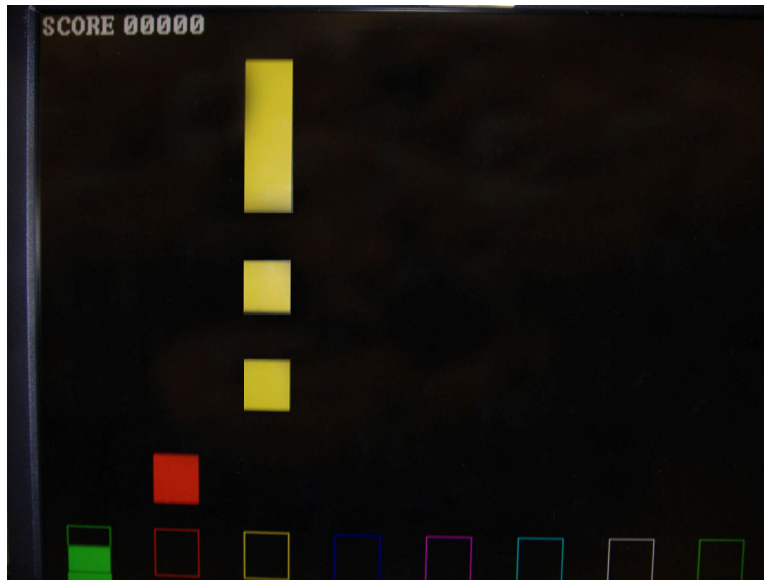


Figure 6. Playing Display of Scrolling Notes with Various Lengths

2.3.2.4.1 Blob Module (C. Terman) [3]

With eight total notes to represent the eight frets, each row must contain eight blobs, and with 9 possible rows, there are 72 total blobs. The Blob module was modified to include whether the color is displayed or to not display the color by displaying the blob as black. Therefore, the color is on when the bits of the note are high, which specify that the note should be played. This allows for multiple notes to be displayed as well since each blob of a row is independent of the other blobs. However, each blob of a row depend on the same y position that increments to allow for the scrolling effect.

The display of what key is being pressed helps the user know what fret their finger is located and assists in playing the notes correctly. Again, eight more blobs are created, but with a smaller width and height to fit into the matching zone since the user will keep their eye in this zone. The color is turned on based on the `keys_held` signal from the keyboard interface. The visual display can be seen in Figure 7 where the visual indication can be seen in the matching zone because some keys are being pressed.

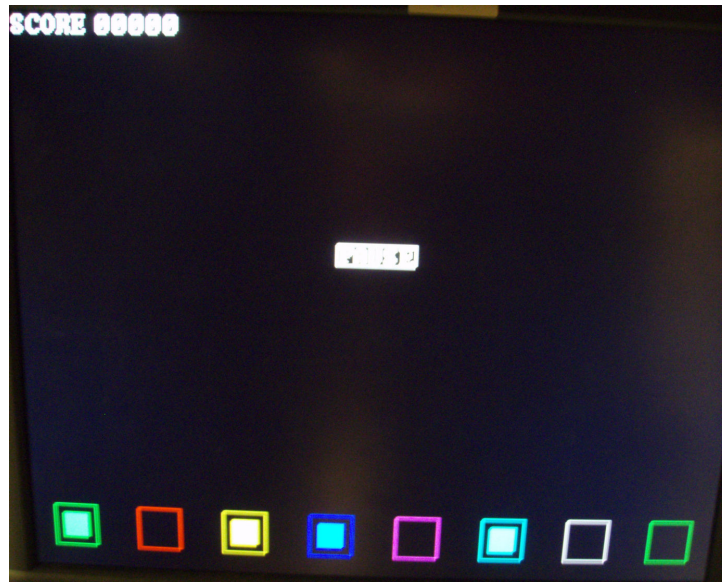


Figure 7. Display of Keyboard Fret Buttons Being Pressed

2.3.2.4.2 Streak Module

The addition of rhythm required a visual indication that a note was longer than one of the quarter notes. Therefore, the Streak Module was designed to add length to a note. The streak is created similar to the blob creation. Each blob needs its own streak, and the streak height is used as an input parameter. The length/height depends on the four most significant bits of the note from the Music ROM that were added for rhythmic purposes. Since 1 represents a quarter note, 2 a half note, and so on, we do not want any length if there is a quarter note. So the height = $(\text{note_type}-1)*100$. The x and y position in the streak represents the lower left corner instead of the upper left corner of the rectangle. This allows for the same x and y positions of the blob to be used to specify the length. A streak can be seen in Figure 6 with the longer length note.

2.3.2.4.3 Blob Outline Module

The Blob Outline module takes the x and y position to make a square where the x and y position represent the locations of the top left corner of the square. As stated in the name, only the outline of the square has the color of the pixel specified by the color input.

The Blob Outline Module is used for the matching zone area to indicate when to play the notes and to indicate when the notes were hit and therefore being played. Eight outlines were needed for the eight different notes of the matching zone while sixteen outlines were necessary for showing that the note was played.

The visual indication of correctly hitting the note is shown by two outlines surrounding the matching zone outline of the corresponding notes. The color is either black or the corresponding column color depending on the notes_hit signal, which is essentially the note sent to the audio. The notes_hit signal contains eight bits, each bit representing the eight positions, so if the bit is high, the note is being played. The user must also hold the fret buttons (number keys) in order to allow for the note to play its appropriate length, as it would work with a guitar. The user does not have to strum the whole time, but the number keys must continue to match in order to continually play the note, especially when the notes are more than a quarter note long.

2.3.2.5 Song Over Module

The Song Over Module congratulates the user and displays the final score. A congratulations banner uses the Character String Display module again. The x position is changed to demonstrate a scrolling effect. The x position decrements so that the congratulations scrolls from right to left. The x position is reinitialized to the x dimension, in the case of the resolution of 800x600, pixel 799, when the Start_Song or Reset is asserted, meaning that the user will play another song before the Song_Over display is shown again. The x position decrements by three pixels until it reaches a specified x position, which stops the movement of the banner in the middle. The score is also constantly displayed in the same manner as in the Playing Display. Figure 8 displays the Song Over Display of the game.

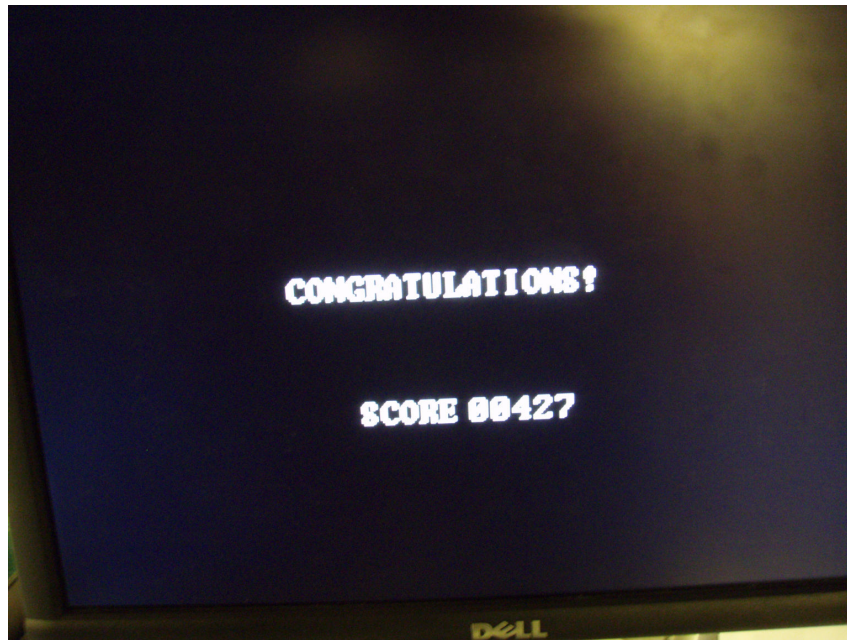


Figure 8. Song Over Display

2.4 GAME LOGIC (by Emily Hwang)

2.4.1 Game Logic Module

The Game Logic acts as a segue between the Display module and the Audio module. This module determines the note entered into audio module for a lookup table and also acts as a scorekeeper. The Timer and Divider modules are also necessary in assisting an accurate behavior of the Game Logic.

The Game Logic determines the note to be sent to the audio module by comparing the expected note to be played to the note being played. The output note is based on the input of what keys the user is pressing and the note that is expected to be in the designated matching area of the display. The user must press the shift button to enter the notes of the pressed frets. Therefore, the Game Logic module only computes the note to send and the score and starts the timer for the length of the note at the moment the shift button is pressed.

The Game Logic constantly outputs the “off” sound until the user presses the shift key, in which the output will either be the correct expected note, or the buzzer note. All eight buttons are taken in consideration of whether the user pressed the correct keys. The keys being held must

match with the expected note to output the corresponding note, otherwise, the buzzer note is sounded. It is also included that the user must hold down the fret buttons in order for the note to continue playing if it is a long note to simulate playing a guitar.

The Game Logic contains a timer in order to send the correct note or buzzer note for the amount of time the note is specified to be. Currently, there exist quarter notes, half notes, three-quarter notes, and whole notes. The Game Logic sets the value to count down from depending on the type of note stated above, which is included in the information of the `expected_note`, and the difficulty of the game, which establishes the speed of the scrolling notes. Therefore, in easy mode, $\text{difficulty} = 1$, the quarter notes are longer than the quarter notes at $\text{difficulty} = 3$, for example. The quarter, half, three-quarter, and whole notes are also relative to each other in length so that, for example, the half note is 2 times as long as the quarter note in the same difficulty.

The Game Logic acts as a scorekeeper by incrementing the total score by the points determined by the Playing Display module through `score_worth`. At the moment that the user presses the shift key to assert the desire to play the note, the score is incremented depending on the accuracy of the user's playing skills. This score is sent back to the Display module to be displayed during the game play and also at the end of the song.

2.4.1.1 Divider Module

The Divider was created to convert the 40 MHz clock into a Hz enable signal, which sends an enable signal at about a quarter of a second. To create a one Hz enable second using a 40Mhz clock, the divider would count to 40,000,000 to send a high signal, but a shorter length was necessary for the quarter notes, so a quarter of a second pulse was created by counting to 10,000,000. This enable is used for the Timer module to countdown its timer value every quarter-second until the timer value reaches zero. To increase accuracy of the Timer, the internal counter is restarted so that when `Start_Timer` is asserted, the Hz enable signal will enable exactly a quarter-second after the assertion.

2.4.1.2 Timer Module

The Timer counts down the number of seconds, initially starting at the value specified by the Game Logic module when the signal `Start_Timer` is asserted. When the timer is started, the expired signal is low. The counter counts down every quarter-second when the Hz enable is asserted. When the counter reaches zero, the Expired signal is asserted high. The Expired signal remains high until the time is started again. The timer is also used by the Display FSM to change states after a certain amount of time.

2.5 AUDIO (by Judy Ho)

2.5.1 Final Audio Module

The Final Audio module instantiates the Direct Digital Synthesizer, and selects the appropriate tones to output. There is a look-up table in the in this module that selects the samples on the different channels and drives them to separate registers. The sample coming from the DDS is 17 bits, but is sign-extended during a pipelining process of the data. If a key is pressed, then the appropriate samples are selected to be outputted. If a key isn't pressed, then zeroes are selected. In the end, the samples for all 8 tones are added together and outputted to the ac97 module. The 8 tones had to be added in a pipelined manner in order for the addition to be completed in one cycle.

2.5.1.1 Direct Digital Synthesizer

The DDS is generated using the Xilinx tools available. This is a ROM that already contains all the data samples necessary to generate a tone. It operates on a 48 kHz clock, and outputs one continuous stream of data samples on 9 different channels, one for each of the 8 tones in the octave as well as the buzzer tone. The 48 kHz clock is split equally between the 9 channels, so no output frequency can exceed 5.3 kHz.

2.5.2 AC97 Module (I. Chuang) [1]

The AC97 module takes in the data samples from the Final Audio module and reconstructs a sine wave based on those samples. This is then outputted to the speakers.

2.5.3 AC97 Commands Module (I. Chuang) [1]

The AC97 Commands module generates the command addresses necessary for the AC97 module to operate. It also controls the volume/amplitude of the sine wave being outputted. The data flow between AC97 and AC97 Commands is shown in Figure 9.

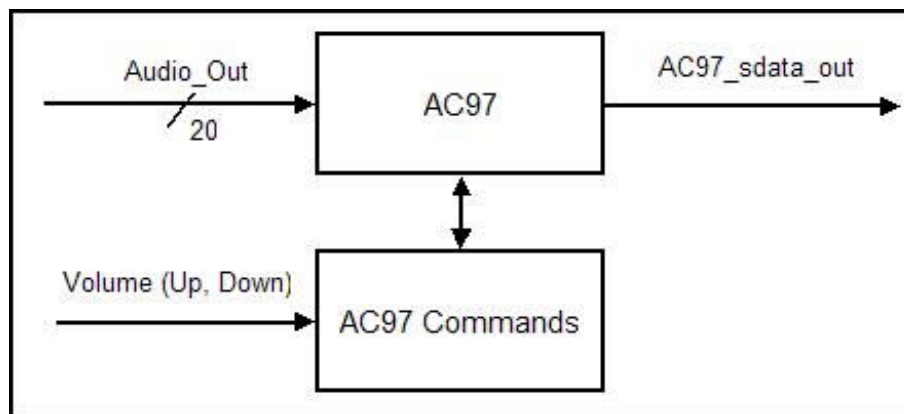


Figure 9. Data flow to and from AC97 and AC97 Commands modules.

3. Testing and Debugging

Keyboard (JH)

The method for testing the keyboard module was simple in a way. The state of the 8 keys was outputted to the led lights. One key would correspond to one led light, and then the lights were observed as the keys were pressed. If the key was detected, then the led light gets turned off. If this did not happen, then it was a matter of making sure the data was being pulled off the FIFO properly.

Music Memory (JH)

In order to make sure that the data was properly loaded into the ROM, a 1Hz enable was used to make a counter. The counter was used as an input for the address. The data was displayed on the LEDs. Therefore, as the counter incremented, the next piece of data would be pulled out of the ROM and displayed on the LEDs every second.

The song selection module was tested by using switches one and zero for the user input as planned, and displaying the start and end addresses to the hex display. For a while, only the end address, and after careful inspection, it became clear that the number of bits for the zeroes added to the hex display input were not specified. Once that was cleared up, the start and end addresses showed up properly.

SVGA (EH)

In order to create a 800x600 resolution for 60Hz refresh, a 40MHz pixel clock was necessary. Therefore, an SVGA was made for the 800x600 resolution and tested with the Pong display from one of the previous labs that had shown to work at 1024x768 resolution. The Pong display at 800x600 resolution verified that both the 40MHz pixel clock and the SVGA were running correctly.

Display (EH)

After the SVGA worked correctly, the actual display could be tested. Looking at the display itself could only test each component and/or subsystem of the Display module.

Originally, the Playing Display was the Display module since it is the actual game itself and is the most important aspect of the display. Simply tested, the outlines of the matching zone were tested to verify the colors in each column and the outline itself was correct. An important necessary feature was the scrolling of the notes, which was first tested without a ROM and without using addresses to retrieve new notes. Notes specified by the switches were used as the input notes. Each note was tested to verify the corresponding placement of eight different horizontal positions on the screen. When this occurred smoothly, the ROM was later used to verify the address process of retrieving notes. After scrolling, other features could be added such as increased speeds for increased difficulty and multiple notes using a song with multiple notes. Other visual aids included the display of visual blobs in the matching zone to indicate which keys the user is pressing and also the display of outlines around the matching outlines to indicate that the note is being hit

After the Playing Display was created, a Start Menu and a Song Over Display were created. These used the Character Display String module and a font ROM previously created by C. Terman and I. Chuang. The Start Menu displays lines of text that include the title, songs, and

instructions on how to play. The Song Over Display displays a “Congratulations” string that scrolls at the end of the song and also a score display.

The score display required the binary/hex score to be displayed in decimal values. Therefore, each digit of the decimal needed to be computed and mapped onto a number string. The Binary to String module was also tested through visual testing and knowledge of the score through the hex display on the labkit.

The problems that occurred was that the pixels sometimes had glitches and subsequently, some colors were not in the correct locations. However, the only testing I did was change whether the signals were OR’ed or XOR’ed together. With the correct combination of OR’ing the similar pixels together but XOR’ing the different pixels such as outline pixels vs. blob pixels, the display soon became absent of glitches.

Game Logic (EH)

The Game Logic was simply tested individually, but later tested while integrated in the system. Initially, without a ROM, one “dummy” note was used as the constant expected note. Without the keyboard, the labkit switches were used to denote the number keys pressed by the user and one of the buttons acted as the shift key. The output note that would be sent to the audio was displayed depending on the user inputs, which were verified to be correct in sending the buzzer note in incorrect matches, the expected note in correct matches, and the “off” note in the remaining situations.

When the ROM and keyboard were working, the Game Logic could be tested more fully. Without the audio, the LED lights were used to represent the new note created by the module. The light would light up if it was being sent. Since there are eight notes, the most significant bit was displayed in the hex display. This bit represented the buzzer note. Therefore, it was verified that pressing incorrect notes sent the buzzer note code to the audio.

Divider and Timer (EH)

The Divider and Timer were created in lab 3. The Divider was originally tested by creating a blinking LED light that would blink every second to verify a one Hz enable. The Timer was originally tested with the hex display. The hex displayed whether the system was in countdown or not along with the initial countdown value. The blinking LED light was also displayed to help count the number of seconds that the countdown should count. One of the buttons was used to start the timer. Different time values were set as initial countdown values. The status of the expired signal verified that the Timer module timed the correct time delay.

Audio (JH)

The tones had to be tweaked by just listening to it after compilation. The keys outputted from the keyboard module were fed into the audio module. This way, as the keys were being pressed, the corresponding note would be outputted to speakers. At first, 9 different tones were created by instantiating 9 different ROMs that took in 9 different phase increments. The sign extension on each of these frequency outputs had to be done correctly.

Initially, there was a lot of static because the data was outputted directly from the ROM to the AC97. The outputs had to be pipelined through a series of registers in order to make sure the data was stable by the time it would actually be outputted. In order to get multiple notes to output at the same time, all outputs from the ROMs had to be added together. With this method, only some of the notes would output correctly, while others would be static. The suspicion was

that the addition was not getting done quickly enough to be outputted correctly at the end of the clock cycle. To make sure that this was the problem, one of the notes that was static, was outputted alone to the AC97 without any addition. When this output resulted just fine, it was evident that the problem was in the addition. As a result, the addition was then pipelined so that it would be done in time to be outputted.

After this, a single ROM with 9 different channels was generated to save memory. Some of the debugging for this ROM was similar to the other ones, such as making sure that the sign extension was done correctly, and that the data was pipelined out of the ROM. The number of bits for the output from the DDS also had to be changed from 8 bits to 20 bits to make sure it was audible.

4. Conclusion

The overall project was completely functional in the end through good design and organized time. The design of the transfer of signals was important in determining the simplicity of each module. The block diagram was updated several times before becoming the simple implementation that integrated the project so well.

At first, the idea of the keyboard seemed very difficult because it wouldn't be easy to determine whether or not a key was actually pressed in conjunction with the shift key. If it were implemented that way, it would have required a more complicated FSM. However, after reviewing this idea, the implementation was changed to have 9 bits represent whether each of the nine keys were pressed or not and to have the Game Logic module do the actual check to see if the shift key had been pressed in its computations.

Another similar change that improved the design was the transfer of the note codes between the Music ROM and Display, the Display and Game Logic, and between the Game Logic and Audio. Originally the eight notes and the buzzer note would be represented as a 4-bit code to represent each tone. However, generalizing the signals proved to be easier and allowed for additional features such as adding multiple notes. The 4-bit code was changed to an 8-bit code, each bit representing each of the 8 frequencies. The Music ROM did not need to carry buzzer tones since it only contains the correct notes of each song. When rhythm was later implemented, the Music ROM also carried this information by adding 4 more bits to represent whether the notes were quarter, half, three-quarter, or whole notes. The Display also only cared about which notes were enabled, so it could easily look at each of the eight bits for each of its eight positions of the corresponding notes. The same note code was then sent to the Game Logic where the rhythm is used for its timer of how long it should send the note to the Audio. The transfer of notes between the Game Logic and Audio is a 9-bit signal to include the eight frequencies in each of the eight bits and also an additional bit for the buzzer frequency. Otherwise, the Game Logic would send a 9-bit code with all zeros to represent no frequencies being played. This representation of the notes allowed for simple communication between the main modules.

Improvement in design occurred by decreasing the amount of ROMs instantiated. A new direct digital synthesizer ROM with 9 channels was created so that each of the nine frequencies were on the separate channels instead of creating 9 ROMs with one channel each. If time permitted, an enhancement in the design of the Character String Display module would be to create one ROM instead of instantiating multiple ROMs for each string. The implementation in the project however, used multiple ROMs. The number of ROMs, however, was decreased in the score display. Since both the Playing Display module and the Song Over Display module display

the score and use the Binary to String Converter for the score where each place in the decimal number is its own string and therefore own ROM, it was important that the Binary to String Converter was only instantiated once. Therefore, the Display FSM instantiated the score display and chose to display it in the Playing Display and Song Over Display when necessary while using a multiplexer to determine the position of the score. This action lowered the number of ROMs by six.

Between the Game Logic and Display, the Playing Display took the longest time because many features could be added. Of these additional features, displaying the binary score into a decimal string was difficult. The design could be improved to be efficient in decreasing the amount of ROMs used, which would be a good idea to focus on when using strings in the final project.

Something that would have made the project go a lot faster was being aware of the sign extensions and remember that numbers outputted to the AC97 are signed numbers. The bit extensions caused other problems when old signals increased in bits from additional features such as adding the rhythm. If not all signals are updated, the new bits will not go through the entire system and bugs will occur. This took time especially when the code was compiled, which takes more time, to find that the new feature was not functional. After finding that there was a bug, it was unfortunate to find that some signals were not updated in extending the number of bits, which is a simple change. Another simple problem that would've made the project go faster is being aware of how the clocks are made through multiplication and division. I was unaware that the power of the labkit must be turned off to reset the clock when the 65MHz from lab 5 was tested and the 40MHz was created for the smaller resolution. When reprogramming the FPGA with the new clock, there would be errors. These errors were simply fixed by resetting the labkit, but this fact was not known and time was spent in trying to create the 800x600 pixel resolution when it was correct from the start.

The overall project was completely functional in the end, but there were some ways that it could have been improved if there had been more time. More features could have been added, such as a recording mode. This would allow the player to record his own songs and have them play back. It can also become another one of the challenge songs for the game. Another aspect that could have possibly been added was being able to play multiple notes at a time, but also have them be of different durations.

References

- [1] C. Terman, “6.111 Lab #3” Available HTTP:
<http://web.mit.edu/6.111/www/f2007/index.html>
- [2] C. Terman, “G.111 Lab#4” Available HTTP:
<http://web.mit.edu/6.111/www/f2007/index.html>
- [3] C. Terman, “Sample Code for Labkit” Available HTTP:
<http://web.mit.edu/6.111/www/f2005/index.html>