MIT Dance Dance Revolution

Submitted by Sharmeen Browarek and Anna Ayuso
Massachusetts Institute of Technology
6.111 Introductory Digital Systems Laboratory
Professor Anantha Chandrakasan
May 18th, 2006

The goal of this final project was to create a version of the popular video game, Dance Dance Revolution. This project was chosen because digital design concepts could be effectively applied to tie together the different components that the game requires. The objective of the game is to sync footsteps on a footpad with a pattern of arrows displayed on a screen that move to the beat of a song. The problem that the project posed was to implement and integrate audio, video, game logic, and a footpad to produce a working game. For this particular game, an invisible footpad was created by using a grid of infrared sensors to determine the location of the user's feet. The approach to implementation was to create a central control unit that communicated with the audio, video, and footpad components. This control system was responsible for scoring and controlling the flow of the game. The modular breakdown of the problem resulted in a successfully functioning game.

Table of Contents

List of Figures:

Introduction

One of the more popular pastimes and largest growing markets of the 21st century is the evolution of video games. Producers are constantly fighting to develop the best 3-dimensional graphics, or come up with the most original ideas. A particularly creative and extremely successful idea was Dance Dance Revolution, which took Japanese arcades by storm in 1998. Quickly spreading across the world, DDR continues to evolve and maintain its position as an entertaining arcade game. DDR requires foot-eye coordination and a knack for music and dancing. To play the game, a person selects a song and difficulty level. Once the song starts, arrows appear at the bottom of the screen and start moving up to the top. The goal of the player is to step on the corresponding arrow with their foot, at the same time the moving arrow goes over a static arrow at the top of the screen. The arrows converge with the beat of the music, and their sequences become more challenging over time. The player is receives a score for his or her performance, which reflects the accuracy of his or her footwork.

The goal of our project was to create a DDR game with a slight twist. The physical footpad was replaced with a grid created with infrared sensors that determined the location of the user's feet. A player of this new version of DDR has the option to choose between two modes of the game: using buttons or using the infrared sensors. The diagram below explains how a user would interact with the game on the FPGA labkit.

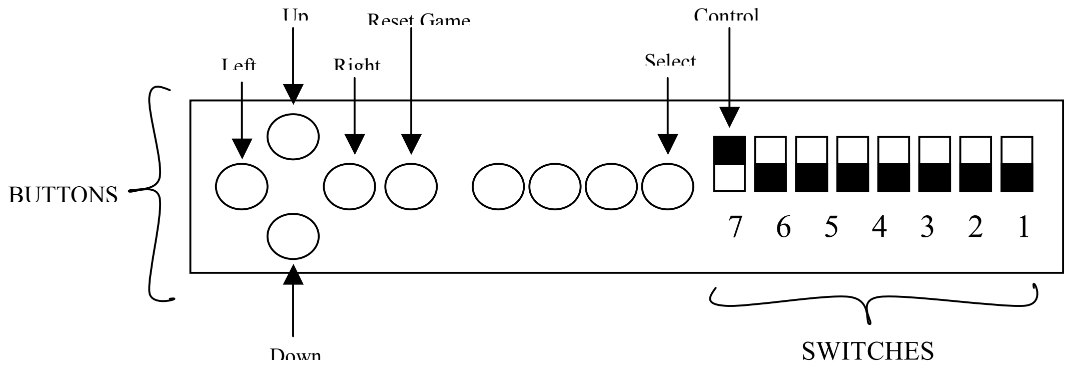


Figure 1. Labkit Interface

If the control mode switch is set to high as shown above, the game will be in sensor mode, and a player will need to use footwork to match the arrows on the screen. If the control mode switch is low, then the player pushes the up, down, right, left buttons instead.

This document will go through the design and implementation of this project in detail. A thorough review of the debugging process is also included, along with suggestions for future improvements.

Implementation

System Overview:

The project was implemented by breaking it down into four major components; audio, video, the control unit, and the sensor input component. Sharmeen Browarek handled the video and sensor input components. Anna Ayuso handled the control unit and the audio component.

The control unit is the central hub of the system and it is responsible for controlling the flow of the game. The video component is responsible for the visual interface that is needed to play the game. The audio component is responsible for playing music during the game. The sensor input component handles the processing of user inputs taken from the infrared sensor grid that represents the footpad to determine the location of the user's feet. All modules are clocked off of a pixel clock. State transitions in FSM's of the system depend on a frame_sync signal that pulses every time the screen is refreshed. Figure 2 shows a high-level block diagram that illustrates how the four components communicate with one another. Detailed descriptions of the four major components and how they relate to the functioning game are provided in this section.

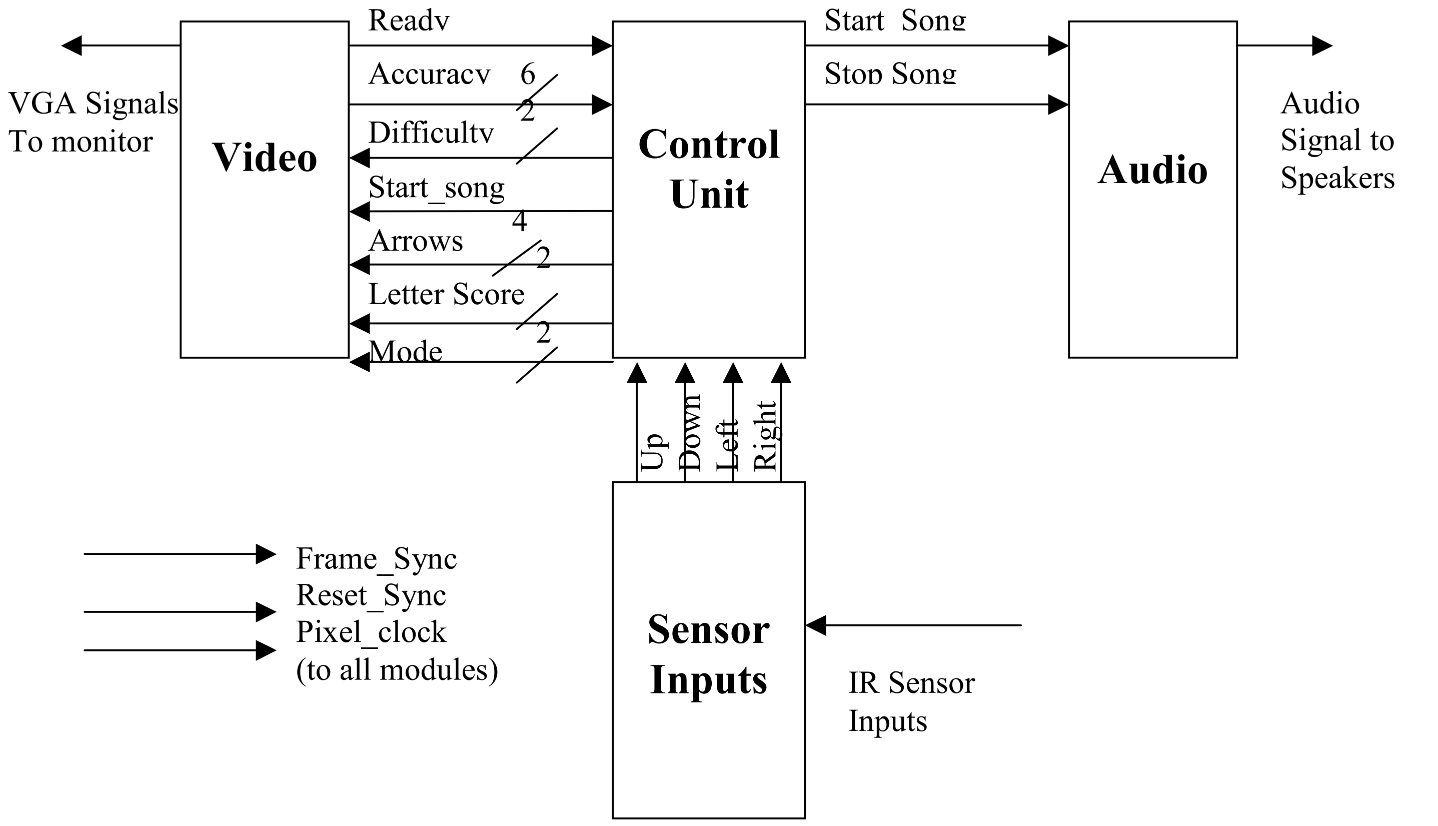


Figure 2. High Level Block Diagram

The Control Unit:

The Control Unit of the system is mainly responsible for three tasks; processing user inputs, calculating the user's score, and controlling the mode of the game. This is accomplished using a major-minor FSM setup. The control unit consists of four modules; the Menu FSM, Game FSM, Score Keeper, and Report modules. Score Keeper and Report constantly compute the score of the game while the control unit transitions between the other two modules like an FSM. Figure 4 illustrates the major-minor FSM setup of the control unit.

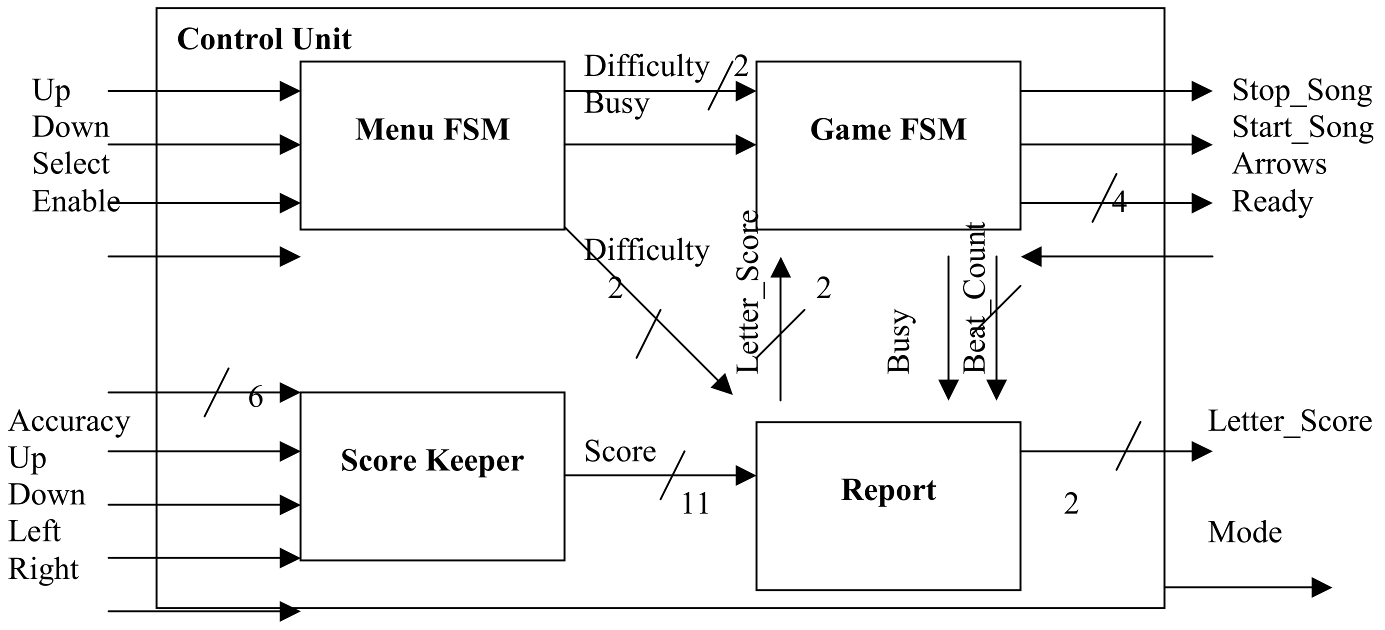


Figure 3. Control Unit Block Diagram

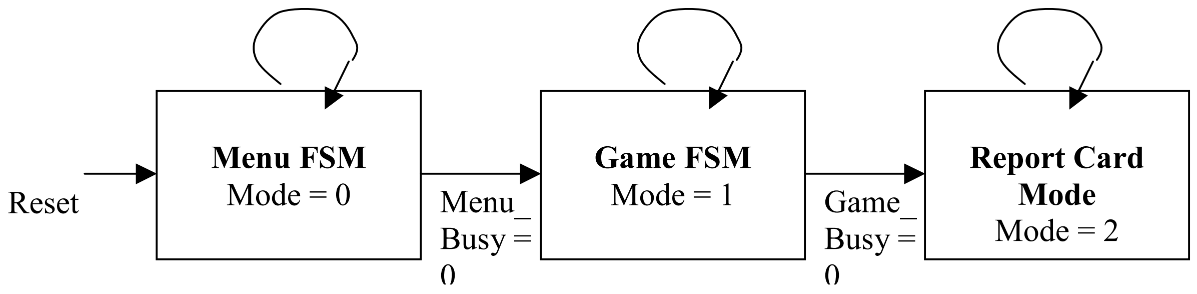


Figure 4. Control Unit Major FSM Diagram

The game has three modes; menu mode, game mode, and report card mode. The game always progresses in this particular order. In order to handle the menu and game modes, minor FSM's were used. In menu mode, the minor FSM transitions between states that output a difficulty. The Menu FSM module takes up, down, and select from the user as inputs. By pressing up and down, the user is able to toggle through the difficulty options. By pressing select, the difficulty is set to the one chosen by the user and the busy signal for this FSM is set low to indicate that the major FSM should now progress to game mode. An important thing to note about the Menu

FSM is that the states do not transition on frame_sync, but rather on an enable signal, which pulses every half of a second. This allows for a slower transition through the difficulty levels. Figure 5 illustrates the Menu Minor FSM. The next mode of the game will now be discussed.

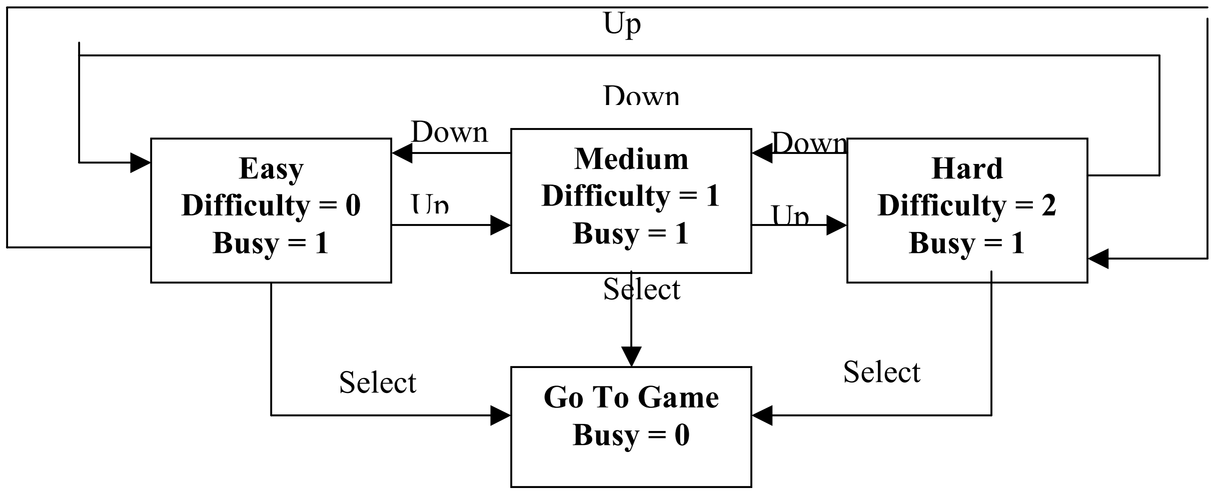

Up
Down
Down
Down
Up
Up
Easy Difficulty = 0 Busy = 1
Medium Difficulty = 1 Busy = 1
Hard Difficulty = 2 Busy = 1
Select
Select
Select
Go To Game Busy = 0

Figure 5. Menu Minor FSM

The game transitions to Game Mode after the user has selected his or her difficulty level. This mode also uses an FSM to function. Figure 6 illustrates the Game FSM. It begins in an initialize state. A song is loaded based on the difficulty the Menu FSM inputs to this module. Songs are represented as arrays of 4 bit numbers. Each element of the array corresponds to the beat of a song. The 4 bit numbers correspond to the arrows that will be displayed on the screen. There are 4 bits for four arrows; up, down, left, and right. A 1 corresponds to an illuminated arrow while a 0 corresponds to the absence of an arrow. In addition to loading a song, the initialize state also sets a start song signal high that tells the audio component to begin playing the song and the video component to begin moving arrows. Once initialization has completed, the FSM moves to the move arrow state.

The move arrow state checks for several things. First, it verifies if the song has ended by determining how far into the array of arrows it is. If it has counted down to the bottom of the array, then the FSM transitions to the go to report state. If the song is still going, the Game FSM awaits a ready signal from the video component. This signal pulses when the video component is ready to start a new set of arrows from the bottom of the screen. The ready signal is necessary to ensure that the timing is synced between the video component and the control unit. The move arrows state waits for this signal, and then checks to see if the letter score is a C or better. If these conditions are met, then the next row of arrows is sent to the video component and the FSM remains in this state. Letter scores come from the Report module. This signal will be described in greater detail momentarily. If the song ends or the score becomes to low, then Game FSM transitions from the move arrows state to the go to report state. The report state sends a signal to stop the music and it also tells the control unit major FSM to transition to mode 2, which is Report Card Mode, through a busy signal.

Scoring is a very important function of the game. The Report and Score Keeper modules both handle this functionality in the control unit. The Score Keeper module continually calculates the score throughout the game. It takes the up, down, right, and left inputs from the user as well as an accuracy signal from the video component.

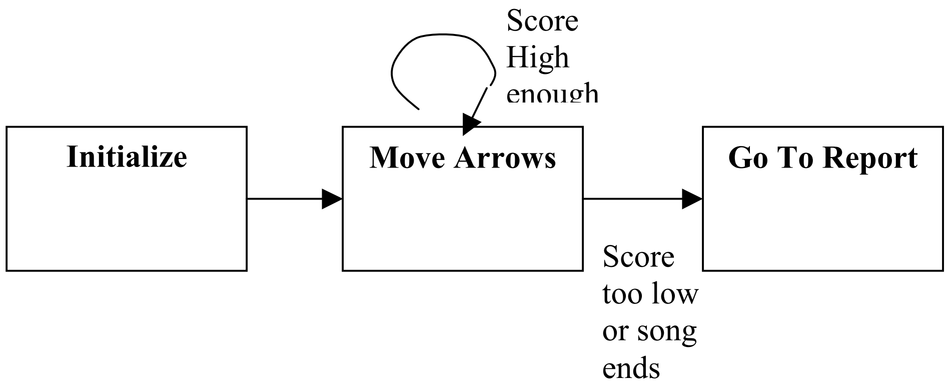


Figure 6. Game Minor FSM

Then, points are assigned based on a match between the user's footwork, which is represented by the up, down, left, and right signals, and the accuracy signal. This system of scoring allows the game to give the user feedback on his or her accuracy. The raw score then needs to be converted into something that can be used by the system. This is where the Report module comes in.

The Report module takes the score, the difficulty, and beat count as inputs and it produces a Letter Score signal as an output. The beat count is a signal, which indicates how many arrows of the song have been sent to the video component thus far. Based on the difficulty, certain ranges of scores are mapped to letter grades; A, B, C, and F. These ranges account for the current beat count, so they are dynamically being updated as the game progresses. As the difficulty level increases, the ranges are made more restrictive so that the cutoffs for good scores are made higher. The Letter Score is used in two ways. First, it is displayed in the game mode in the form of a bar. Different colors of the bar indicate the current letter score. The bar dynamically changes throughout game play. This allows the user to gage their performance. Second, the letter score is displayed in the final screen of the game that corresponds to the report card mode (mode 2).

The Video Component:

The Video Component of the Lab has many modules within it that interact with each other. These modules will be explained in the following paragraphs, and the block diagram in Figure 7 displays their connections.

A. VGA Module
The VGA, which stands for Video Gate Array, is in charge of displaying pixels to the video screen. It uses a counter to calculate 640 pixels horizontally across the screen, followed by a blank period where the cathode ray tube (CRT) situates itself back to the beginning of the screen. This process continues until all 480 rows have been drawn, at which point the vertical blank occurs, during which the CRT resets back to the upper left-hand corner of

the screen. The VGA module takes in the pixel clock signal and outputs the horizontal and vertical location of each pixel as it is displayed. These location signals are sent in the form of pixel_count and line_count to the Game Display Module, which assign the correct color values of each pixel.

B. Game Display Modules

The Game Display module tells the CRT each pixel's color assignment as the tube moves across the video screen. The color is set through a 24-bit combination of red, green, and blue values. The DDR game has three different modes displayed by the screen: the menu, the in-game display, and the report screen at the end. A functioning game with a straightforward and intuitive user interface requires a variety of smaller internal modules running simultaneously.

The first of these necessities is a font. Verilog code for a video display of character strings written by I. Chuang and C. Terman is available online. The package contains a Font ROM (read-only memory) and a module, which communicates with the ROM to output values and display letters

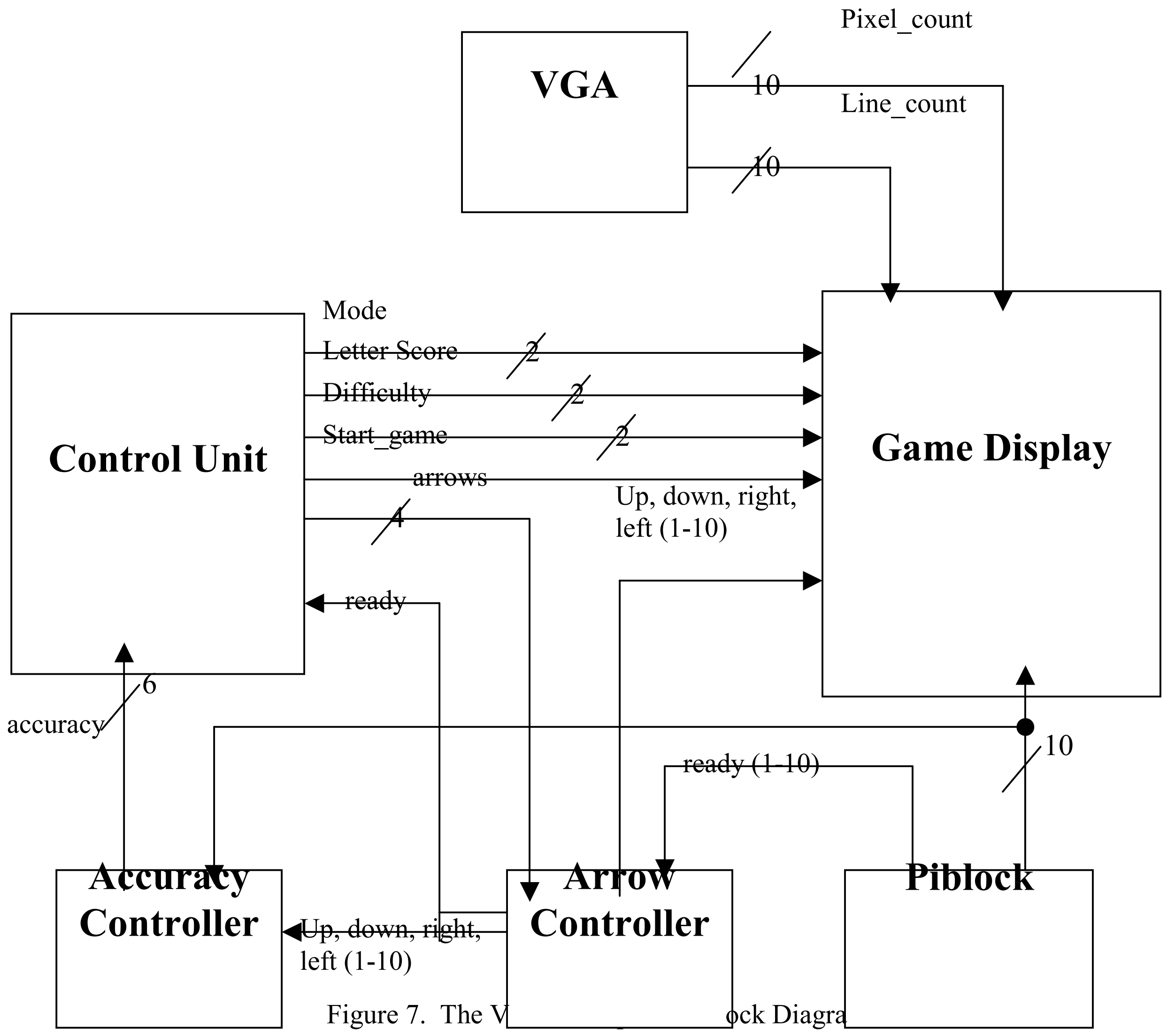


Figure 7. The V ck Diagra

on the video screen. The character string module is instantiated multiple times in the Game Display module to simulate a menu screen and a report screen at the end of the game.

Another aspect of the display is the group of image modules. The in-game screen has a dancing beaver on the right hand side. This is simulated through four beaver pictures repeated in a continuous loop. Each of the images is saved on the block ROM as an independent module. The Image Rectangle module assigns the location of the pictures on the video screen. Then the Image Loop module uses a counter to display each image for half a second and then moves on to the next image in the sequence. These two modules are instantiated in the Image module in order to communicate with each other, but maintain a simple, one-module interface with the game display.

Next, a Rectangle module is written within the Game Display. It creates an arbitrary rectangle of a specified size, at a given location, with an assigned color value. Simply put, the module takes in a coordinate and makes it the upper-left hand corner of a rectangle. Then, based on the input width, height, and color, all the pixels within the rectangle show up as their designated color, while the rest of the screen remains black. This module is used to create a border for the game, to designate, which difficulty level a user has chosen, and to display an accuracy bar during the game so the user can gauge how well they are doing.

The Piblock Display module, a more complicated version of the Rectangle, is also needed for this game. Piblock Display uses the same concept, but instead of a solid rectangle, it creates four pi's in a row, facing four directions: left, down, up right. One block of pi's remains static and is instantiated at the top of the screen in blue. Ten other blocks are fed a y-direction value from the Piblock module and a sequence of arrows from the Arrow Controller, thus creating the effect that pi's are moving up the screen.

All of these internal modules are instantiated with the logic, and their outputs are combined to create the three modes of the game.

C. Piblock Module

The Piblock module controls the y-direction values of the ten dynamic piblocks. It starts the movement of the blocks when a user selects a difficulty level and the game switches to in-game mode. Piblock receives the pixel clock and the frame sync signal. The frame sync signal is set to high when the pixel and line count are both at the bottom right corner of the screen. The overall pixel clock for the game was specifically chosen to have a 72 Hz frame refresh rate because the song used in the game has a speed of 144 bpm (beats per minute). Piblock moves the pi's up the screen at a rate of two pixels per frame refresh. Therefore, the movement of the pi's and when they go over the static piblock at the top of the screen is synchronized with the beats of the song. This module also outputs a ready signal for each piblock, which goes high when the piblock is reset to the bottom of the screen. The ready signal is sent to the Arrow Controller Module.

D. Arrow Controller Module

The Arrow Controller is in charge of telling the piblocks which pi's are displayed during the song. The values of the pi's are stored in the control unit in the form of arrays. The Arrow Controller takes in the ten ready signals (one for each dynamic piblock) from the piblock module, and in turn sends the control unit a single ready signal. For each ready, the control unit returns a four-bit arrow signal. The Arrow Controller interprets this signal, and assigns the new pi values to the correct piblock.

E. Accuracy Controller Module

The Accuracy Controller is the module that interacts with the Control Unit to help with the scoring algorithm of the game. It receives the y-direction value and which of the four arrows are lit for each piblock. The accuracy controller sets a number based on how close the dynamic pi's are to the static piblock at the top of the screen. Figure 8 illustrates the acceptable ranges for different accuracies.

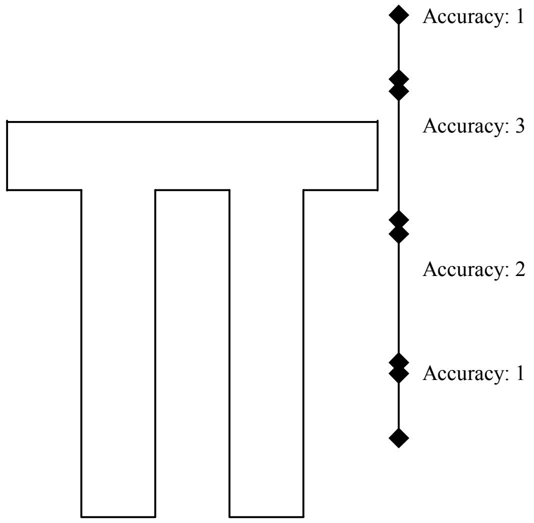


Figure 8. Accuracy Ranges

Based on these ranges, the module sends the Control Unit a 6-bit signal. The signal comprises of two bits for the accuracy range and four bits to denote which arrows are currently lit for the piblock. The Control Unit uses these values to decide the accuracy the player's footwork and output a score. The Accuracy Controller also takes in a control_mode signal. This is because the game can either be played using either the IR sensors or the buttons on the labkit. There is a delay before the data from the IR sensors is received, so the accuracy range is shifted accordingly to ensure the player's skill is correctly interpreted.

The Audio Component:

The Audio Component is responsible for playing the music when the game is in game mode. The chosen song for the project is Mr. Roboto by Styx. Figure 9 illustrates the block diagram for the audio component. For this project, code for a sound effects generator was modified to play music instead of sound effects. The original code was written by Eric Fellheimer and Nathan Ickes and can be found online.

The Audio Component consists of 5 modules. The AC97 and the AC97 Commands modules are both responsible for setting and configuring the frames that are sent to the AC97 codec. The AC97 codec processes the frames and sends sound output to speakers. The Audio Component has a simple Song FSM module which progresses through three states: Idle, Start, and Play. During the Play state, the module feeds PCM song data from the MrRoboto module to AC97, which will ultimately send the song data to the speakers. The MrRoboto module is responsible for running through memory to retrieve each music sample at the appropriate time to send to the Song FSM. It does this by counting to the length of samples in the song and retrieving the data at each count.

The song was stored in memory by using Matlab and java code to handle signal processing. A 47-second clip of the song was downsampled by a factor of 10. This was accomplished by averaging every 10 samples together and holding the average for 10 counts when actually playing the song. This allowed for a much smaller coefficients file to be loaded into the FPGA ROM, however, the downsampling also resulted in lower music quality. Linear interpolation was applied to smooth out the song signal, but unfortunately, it produced a crackling noise that was unavoidable, so the design decision was made to continue to use the downsampled song despite the lower sound quality. Due to the memory constraints, the song was stored on one FPGA while the rest of the project was stored on another one. The FPGA served as a memory device with an address input and a data output going back and forth between the two FPGA's.

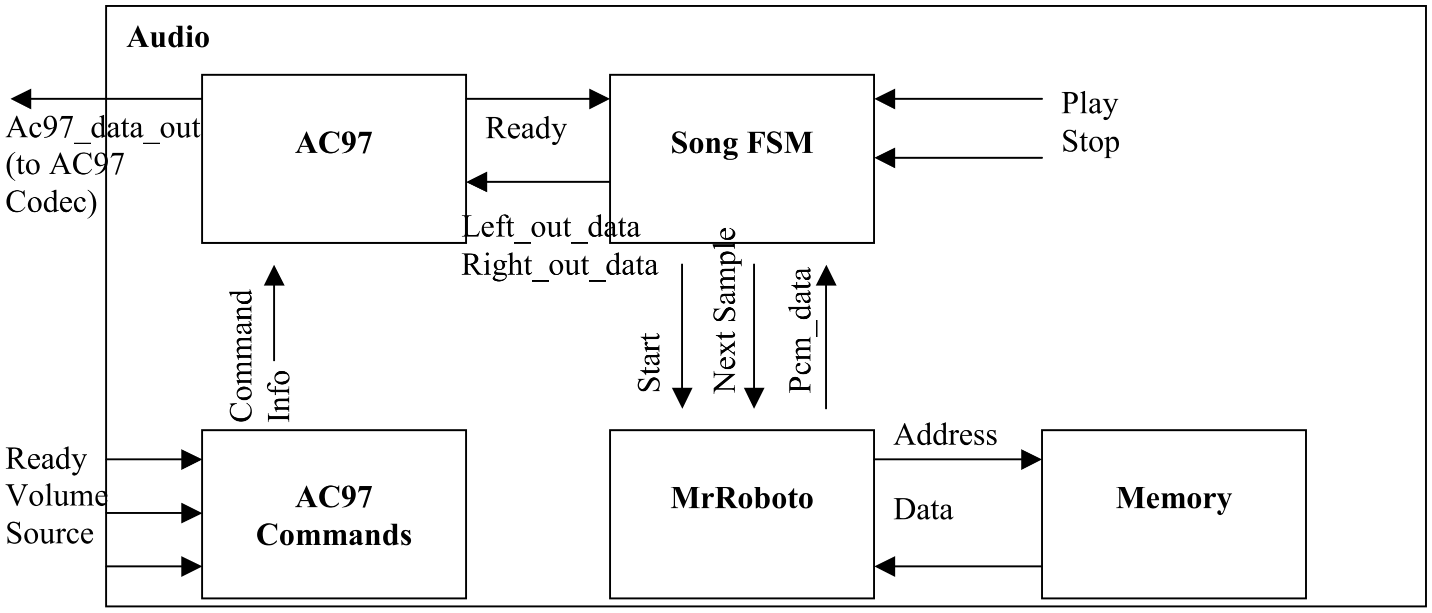


Figure 9. Audio Component Block Diagram

The song was able to be stored in memory by using Matlab and java code to handle signal processing. A 47-second clip of the song was downsampled by a factor of 10. This was accomplished by averaging every 10 samples together and holding the average for 10 counts when actually playing the song. This allowed for a much smaller coefficients file to be loaded into the FPGA rom, however, the downsampling resulted in lower music quality. Linear interpolation was applied to smooth out the song signal, however, it resulted in a crackling noise that was unable to be avoided, so the design decision was made to stick with the downsampled song despite the lower quality of sound. Due to the memory constraints, the song was stored on one FPGA while the rest of the project was stored on another one. The FPGA served as a memory device with an address input and a data output going back and forth between the two FPGA's.

IR Sensors Component:

The infrared sensors go through a series of debouncers and high pass filters to remove noise and output a steady signal. This process is described by the sensor's block diagram in Figure 10 below:

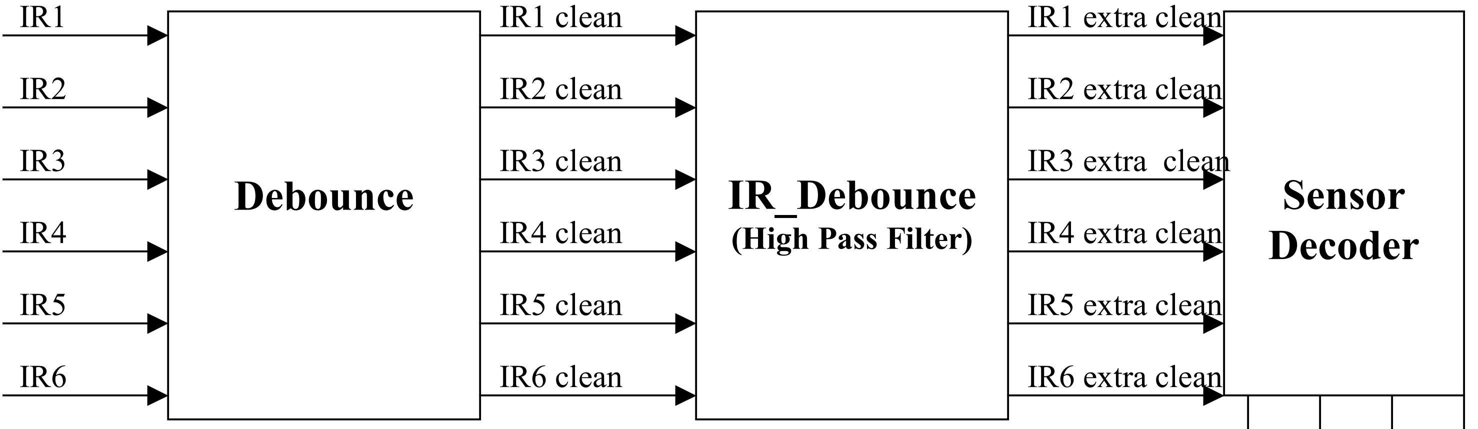
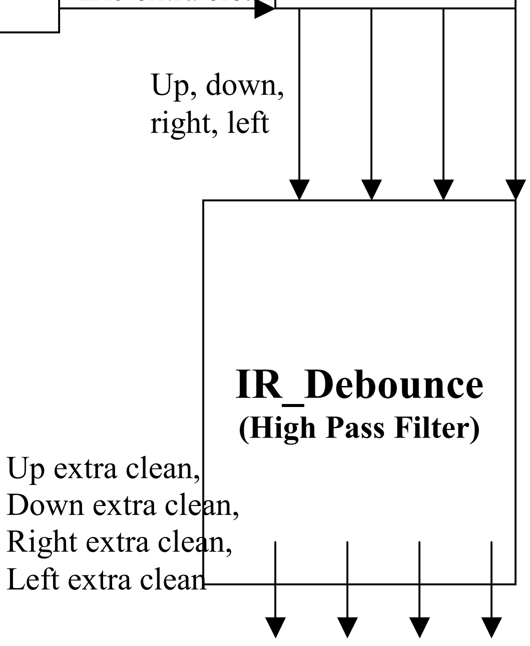


Figure 10. The Sensor Inputs Block Diagram



Decoding Module

This module is in charge of decoding the outputs from the infrared sensors. The sensors are sent through the debounce module, which synchronizes the signals and filters out noise. However, instead of sending steady values of one or zero, the sensors send a value of one when blocked, and many pulses when not blocked. The IRDebounce module helps to remove the noise of the pulses. This module acts as a high pass filter that only outputs a high when the sensors send a constant value of one (when they are blocked by the user's feet), not when they end a series of high pulses. The 'extra clean' outputs of the sensors are sent to the Decoding module, which in turn decides when the user's feet are in the up, down, right, or left position.Figure ? in the appendix shows how the six infrared sensors are used to create the grid for a player's footwork. These logic outputs are again sent through the IRDebounce to clean out the noise even further. The extra precaution is taken to avoid faulty inputs from the sensors as much as possible. The Decoding module also takes in the control_mode signal. Depending on whether the user wants to play in the sensor mode or the button mode, the Decoding module will either assign the final values of up, down, right, and left based on the sensor inputs or the button inputs.

Testing

Testing and Debugging the Control Unit:

Testing and Debugging the Control unit was accomplished by first testing each module individually. Test benches were written for each module to verify certain expected behaviors. For the Menu FSM, it was confirmed that states transitioned correctly given changing up and down inputs and finally a select input. It was also verified that the difficulty remained constant after a select was chosen. For the Game FSM, it was ascertained that the module kept producing arrows as long as the score was high enough. It was determined that this FSM would transition to the go to report state when the score became too low. For the Score Keeper module, it was asserted that the score incremented with the up, down , left, and right signals kept constant and a changing accuracy signal. It was confirmed that the user only received the points for the instant that he or she matched the accuracy signal. For the Report module, it was verified that the letter score that was outputted was correct for range that the score should have been in.

A test bench was then written that integrated all four modules together. It was determined that the major FSM of the control unit transitioned to the different modes appropriately. First, it began in mode 0, and when the user selected a difficulty, the major FSM transitioned to mode 1. In this state, arrows were outputted as long as the score was high enough. When the score became too low, the FSM transitioned to mode 2, the report mode.

Testing and Debugging the Video Component:

Testing of the video component of this game was mainly done through compiling and examining the output on the screen. Unfortunately, creating testbenches and using the logic analyzer were not as useful because the display incremented on half second intervals. Furthermore, the only way to see if the display was truly sending the correct values, was to physically load it to the screen and analyze the output. The method of compiling was not too time consuming because each part of the display screen was added one at a time. First, the border was created. While the project compiled, the piblock sections were added to the display. Then, the border was tested and fixed before the new project compiled. This pattern continued as the various components – accuracy bar, font character strings, dynamic piblocks – were inserted in the display module. There were two main points where the evolution of the display took large steps. The first of these points was the integration of the display with the control unit, which added the switch between the three modes in the game. How this was tested is explained in the integration testing portion. The next of these points was adding the image of the dancing beaver to the display. In order to create this smoothly, first the modules were written to instantiate a single picture. Once the code for a single picture worked, the other three were added in using a counter which looped between them. On the first try, the horizontal and vertical lengths of the picture had been reversed, creating a glitchy output on the screen. However, it was clear that this was the problem, and after it was fixed, the dancing beaver showed up on the screen.

Testing and Debugging the Audio Component:

Testing and Debugging the Audio Component was accomplished by ultimately listening to the results. First, the sound effects generator was tested to see how it would work. Second, one of the sound effects modules was altered to play a song instead. Third, a verilog test bench was written to check that the memory had been loaded properly with the coe file for the Mr. Roboto clip. Fourth, the project was compiled and synthesized to verify that the sound effects code would play a song instead of a sound effect. Once this milestone was completed, the next step was to improve the quality of the song.

The quality of the song is directly related to how much memory is available. Different coe files were created with the clip downsampled by factors of 6, 8, 10, and 12. It was determined that given the ROM available from the FPGA, the coe file with the lowest factor of downsampling that the FPGA would load was 10. Steps were taken to perform a linear interpolation on the signal that would smooth out the song. This was successful in making the song more clear, however, for some unknown reason, it added a crackling noise that made the quality even worse. It was decided to stick with the original song that was downsampled by a factor of 10.

Testing and Debugging the Sensors Component:

The infrared sensors were initially tested using an oscilloscope. They were hooked up to voltage and ground sources and then tested on the oscilloscope screen to see if they output highs when obstructed and lows when not. At this point it was discovered that when not obstructed, the sensors did not output steady lows, but in fact a series of pulses. In order to see how the FPGA would interface with the pulses, the sensors were next hooked up to the labkit, and their outputs were assigned to LED's. At this point it was clear that the sensors needed to be fed through a filter, which would remove the continuous pulses, and only output a high if the sensors were indeed obstructed. A filter module was written using logic which combined the thought processes behind debouncer and divider modules. In order to test the sensors without wasting time, a separate project was created on Xilinx. The values to the filter module were altered until the LED's finally displayed sensor outputs which filtered out almost all of the pulses.

Integration Testing:

Integration testing was carried out by first confirming that each of the four components worked individually. The next step was to link the Video and Control Unit together to verify that arrows would begin to ascend the screen in game mode. This did not occur immediately and it took many hours of debugging. Testbenches were created to integrate the two modules to better see what was going on. Also, the logic analyzers were used to look at certain control signals. The problem turned out to be that the Game FSM in the control unit expected the start game signal to be a constant high value when it was in fact a pulse. This resulted in the Game FSM never transitioning and only one arrow ascending screen. The bug was discovered by tying certain signals to high values and also verifying signal behavior with the LED's.

After it was determined that the video component and the control unit were communicating, scoring was tested by wiring the LED's to the accuracy signals and the up, down, left, and right signals. When it was verified that there were matches between the two, the LED's were then wired to the score from the control unit. The game was played and it was ascertained that the score was being incremented with successful matches of arrows and buttons. It was also confirmed that more points were given for more precise matches.

The next step in the integration process was adding in the audio. An attempt to load the memory on the FPGA failed because it said the project was at maximum capacity in terms of resources. At this point, it was determined that the music would be loaded on a separate FPGA that would serve as a memory unit. The two FPGA's were then connected with address and data signals running between the two. This setup worked the first time it was compiled and run. Since the signals were not bi-directional and there was no dependence on the clocking of the two systems, this was a relatively simple modification to make.

The final step of the integration process was to add the sensors. The sensors picked up much noise so extra processing had to be added to filter out the noise. This increased the delay between the time a user stepped on an arrow, and when it was recognized by the system. To fix this problem, the scoring was made more lenient to account for the delay. This was tested by wiring the accuracy signal and the user inputs from the footpad to LED's. The scoring leniency was tweaked until more of the accuracy signals matched up with the user selecting arrows with the delay included.

The final step of the integration testing was to find any way to break the project. One bug was found in the Menu FSM of the control unit. When select was pressed during game play, the arrows changed location on the screen. This was because the difficulty was not being held constant from the Menu FSM. The user was still transitioning between difficulty states when they were pressing the up and down buttons in game mode. The bug was fixed by adding another state to the Menu FSM that held the difficulty constant when a selection was made.

Conclusion

In conclusion, we would like to emphasize that our overall experience with this final project has been extremely positive. The core objectives of this project were to come up with an idea, design it, and implement it in a timely manner. The key to designing a successful project was creating a modular design, which was easy to test and debug and would fit together seamlessly. In order to do this, we separated the project into four major components: video, audio, sensors, and the control unit. These blocks were broken down further into multiple modules, each with their own functionality. Such an approach made it easier to debug as we came across different problems. Along with a solid initial design, we also made sure to use a variety of digital design concepts, from synchronizing our modules off of a single clock to using major and minor FSM's. We also interfaced with a variety of external components: playing an audio song, loading bitmap files, displaying a font, and using infrared sensors as an input. The combination of these allowed us to learn a lot more about concepts we hadn't previously covered in class.

In retrospect, we understand how important it was to integrate our individual portions of the project ahead of time. This allowed us to focus on all the minor details when the project deadline was around the corner. The one thing that was most frustrating was dealing with memory on the FPGA labkit. Even though we carefully calculated ahead of time how many blocks of ROM would be used by our song and pictures, we didn't take into account the logic gates created by the labkit for multiplication, and other such commands, in our code. Because of this, we had to make a last minute decision to use two labkits for the project: one acted as the master and controlled the entire game, the other was used to save the audio component on the memory. Another lesson we learned was that external interfaces always require extra processing. For example, the infrared sensors needed a more complicated debouncer to filter out the noise. Plus, both the audio and the images needed to be converted to .coe files and processed before they could be loaded to the labkit.

Finally, if we were to do the project again, our most important focus would be to find a way to have better song quality, perhaps by using the ZBT SRAM, or the flash ROM device. Also, instead of using the font ROM, it would be more aesthetically pleasing to either create a better font or to load images with nicer fonts to the display screen. Regardless, the most important part of our project was the modular break down, which allowed for ease of debugging and integrating the various components of the game. Creating an in-depth design ahead of time, and sticking to it, made the rest of the process run smoothly. The final result was successful project completed on time.