Reynaldo Farias Zorrilla
Noah Boorstin

Microprocessors Project Final Report

December 9, 2019

Abstract

This project creates a small video game system where the user controls a paddle to bounce pellets across the screen with the twist that multiple pellets will be present on the screen at once, requiring careful attention and timing from the user. This game was based on the flash game “Pel”. Control input is read by the ATSAM from a replica NES controller. Video output is then sent over SPI to the FPGA which generates an 800 x 600 pixel VGA display with the help of three 3-bit DACs. Sound output is sent to a small chip over UART , which can select .mp3 files from an SD card and play them.

Introduction

Motivation

The motivation for this project was to create a functioning videogame. Furthermore, we wanted the video game to be fun today; we did not want to make a replica of a game that has historical interest but is now relatively obsolete, such as Pong. To that end we chose to replicate the relatively obscure flash game “Pel”, which is both fun today (one author has spent many dozens of hours playing it), and simple enough to be implemented in our limited hardware and development time.

In this game, colorful pellets, henceforth known as “pels”, spawn from the top left of the screen and need to bounce three times to make it across the screen. The height of each bounce is randomly generated so the time between bounces varies. The user controls a paddle 1/3 of the screen wide and has to ensure the pels bounce instead of falling. As the game progresses, more and more pels will be on screen simultaneously, requiring concentration from the user to make sure all the pels bounce and none fall.

Overview and Block Diagram

The new hardware for this project was very much dictated by the project goal and the limitations of ATSAM and FPGA processing power, as well as development time. We wanted hardware that was simple to interface with but also familiar to those who play videogames. To that end we used a replica NES controller to get input from the player. As seen in the block diagram in Figure 1, that input is read by the ATSAM, which then sends display data to the FPGA over SPI and audio data over UART. Because of the relatively slow speed of the ATSAM and lack of memory on the FPGA, the ATSAM does not output a full pixel array; it only provides locations of the pels and paddle, the current score, and signals for “game over”, “start”, or “pause” to be displayed on the screen; the FPGA then generates the full resolution video signal on the fly. After generating the video signal, the FPGA sends 3-bit RGB data to the three 3-bit DACs made out of resistor ladders, and outputs the HSync and VSync control signals directly to the VGA cable. The audio is generated by an MP3 player module that reads MP3 (or

WAV) files from a microSD card. The generated audio is outputted to a simple amplifier stage and then sent to a full range speaker.

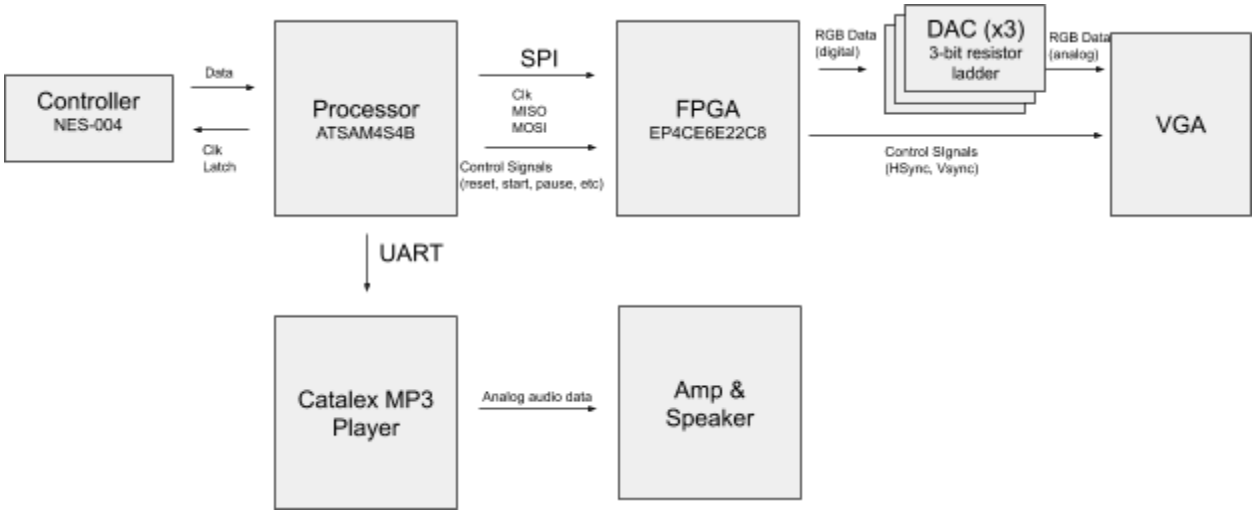

Figure 1. Overall system block diagram

New Hardware

NES Controller

Human input for this project was gathered through a replica NES controller. The schematic for this controller is shown in Figure 2. The main chip in the NES controller is a Texas Instruments 4021 8-bit shift register. This has two modes of operation: asynchronous mode, where data from the push buttons is “jammed into the 8-stage register via the parallel input lines”, and synchronous mode, where data is shifted out one bit at a time, on the rising edge of the clock cycle[1].

We are interfacing with the controller through our nes_controller_read() function, which first sets the latch to low to switch the controller into synchronous mode, then sends 8 clock pulses and shifts the data it reads out into a shift register, then sets the latch to high to return the controller to asynchronous mode for it to collect input from the buttons again.

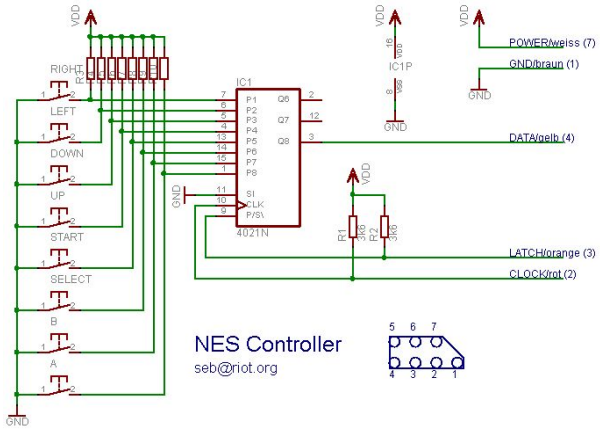

Figure 2. NES controller schematic

Reynaldo Farias Zorrilla
Noah Boorstin

VGA Display

The VGA Display is the primary output for this project. We are driving the display at a resolution of 800 x 600 pixels which requires a 40MHz pixel clock[2]. Conveniently, 40MHz is the same speed the oscillator on the MicroPs board runs at, so no clock dividers or PLLs were required.

The two digital signals required, HSync and VSync, were generated by the FPGA and output directly to the VGA display. Data for the red, green, and blue channels was converted to analog signals by three separate 3-bit DACs constructed from resistors on our breadboard. The schematic for the DAC is shown in Figure 3.

With our initial approach we noticed a problem known as “ghosting”, where the colors for pixels would bleed into the adjacent pixels on the right. We were not able to discover the root cause of this. The problem is probably caused by mismatched impedances, however what caused the mismatch remains unknown.

Despite not knowing the exact cause of this problem, we were somehow able to mostly eliminate it by adding a diode between the resistor ladder and VGA display. When we noticed one diode helped with the problem, we added another diode in series with the first diode which almost completely eliminated the problem. However, having two diodes in series cause a significant voltage drop (around 1.4 V) and so to compensate we increased the input voltage to the DACs from 3.3 V to 5 V by using a 74HC04 Hex inverter, since it can accept the 3.3 V signal from the FPGA but will output a 5 V signal[3].

Catalex MP3 Player

Game sounds are produced using an MP3 player module. The MP3 player module is controlled over UART using simple hexadecimal commands. These commands allow for song selection (based on the alphabetical order) of songs, volume control, and play/pause control. The MP3 player gets audio data from a microSD card, where the game sounds are stored in WAV format. The audio data gets converted to analog within the MP3 player module and is outputted through an audio jack[4]. Although the module provides both a left and right channel, we are only using one channel because we only have one speaker. The MP3 player module is interfaced to the amplifier using an AUX cable that is cut on one end.

Reynaldo Farias Zorrilla
Noah Boorstin

Breadboard Schematic

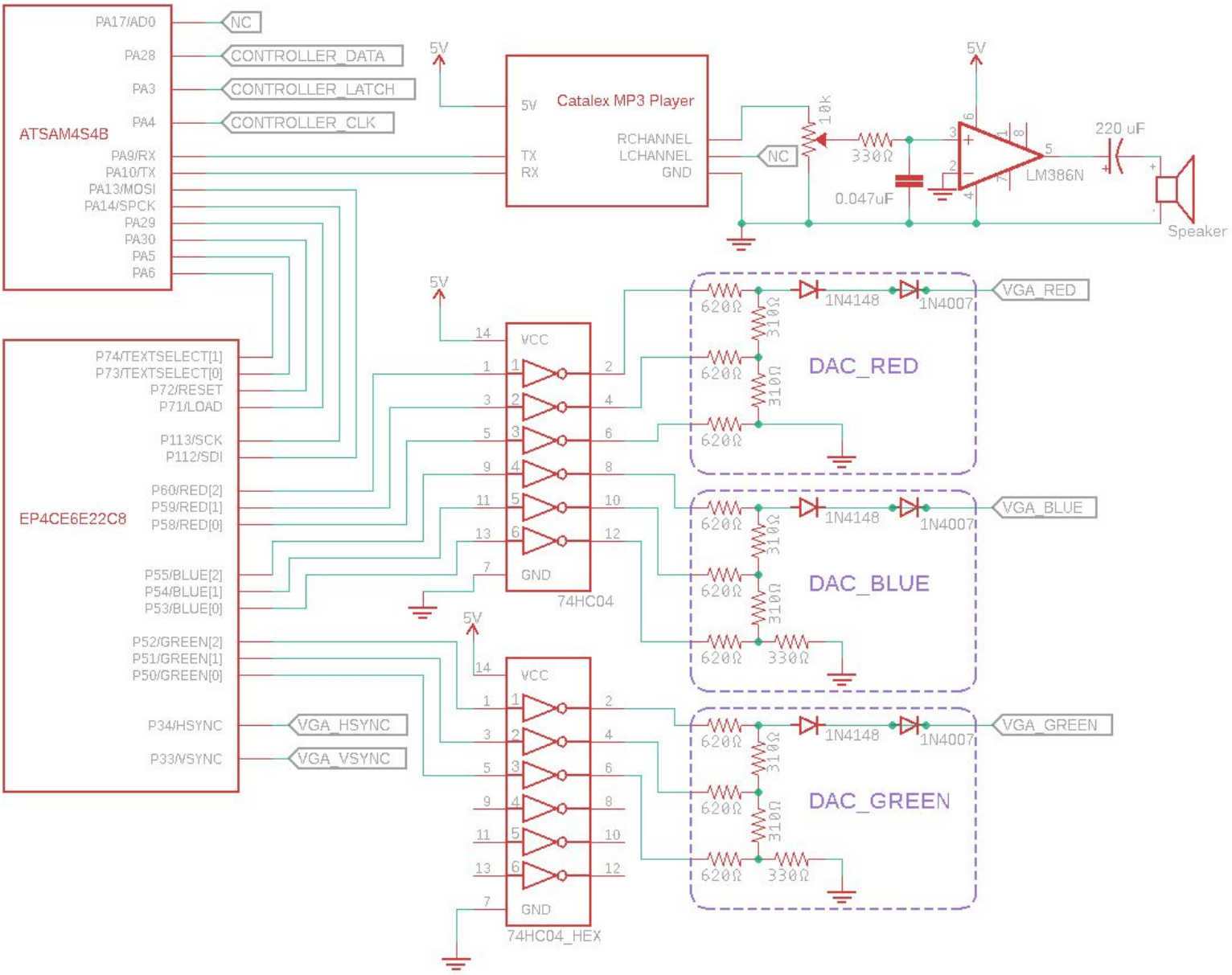

Figure 3. Schematic of breadboard (and some internal) connections.

Reynaldo Farias Zorrilla
Noah Boorstin

Microcontroller Design

The functions of the ATSAM microcontroller are to run the main game logic, send game updates to the FPGA, and control an external music player module. The ATSAM takes inputs from the NES controller and a single I/O pin, and outputs data through SPI, UART, and four I/O pins. The I/O input is setup as analog input, and is used for seeding the random number generator for the game. One of the output pins is used to reset the FPGA, and is toggled everytime the ATSAM is reset. The rest of the inputs and outputs of the microcontroller will be explained later.

The higher level game control is somewhat simple. Inside of the main() function, there is an outer loop that controls game state(start screen, game over, gameplay), and inside the outer loop there are three independent loops that run three different game states. Switching between game states is controlled using software switches (ints that are set to either a 1 or 0) and break statements, and the FPGA is notified of the status change using two I/O pins (gamestate pins). If the game is currently in the start screen the gamestate pins are set to 2'b10. For Game Over and Gameplay, the pins are set to 2'b11 and 2'b00 respectively.

The most important inner loop is the gameplay loop. This loop runs the multiple game logic blocks that are required to run the game, communicates with the controller to take in user input, sends game updates to the FPGA through SPI, and sends audio updates to the external music player model through UART. The main game logic blocks are summarized as follows, in order of appearance in the loop: 1) update pel locations, 2) check for pel collisions with the paddle and the edges of the screen, 3) receive controller data, 4) check for pause requests, 5) check for sound toggle requests, 6) update the paddle location, 7) remove dead pels, 8) add new pels, 9) send SPI data to the FPGA, 10) check for game over and 11) delay before the next loop. Lets go over each game logic block in more detail.

The first game logic block is the block that updates the location of the pels. This is done every loop, which means that the speed of the pels is entirely controlled by the delay block (block 11). The location of the pels is updated using a parabola equation. Each loop iteration we increase the x coordinate of the existing pels by one and subsequently calculate the y locations using the following equation:

$$y = -((A/133^2) * (x - PelArea * 266)^2 - parabolaHeight)$$

The $PelArea$ variable corresponds to the current area of the screen that the pel is located in. The screen is divided into 4 areas, as shown in figure 4. This splits the pel paths into 2 "full" parabolas (2nd and 3rd area) and two "half" parabolas (1st and 4th area). The A in the equation corresponds to the maximum height of the parabola. Each pel has random A values for the 2nd and 3rd areas, but fixed A values for the 1st and 4th areas. The $parabolaHeight$ is a constant that corresponds to the number of pixels between the top of the paddle and the top of the screen. The numbers 133 and 266 are just a result of the fact that our current resolution is 800 pixels wide. For different resolutions these numbers would have to be adjusted.

The second game logic block does collision checking. The collisions that are checked are between pels and the paddle, pels and the bottom of the screen, and pels and the right edge of the screen. The code consists of a bunch of if-statements, and can be found in the appendix A.

The third game logic block receives data from the NES controller. The controller collects data into an 8-bit shift register, where each bit corresponds to the state of one of the eight buttons (high means the button is down, and low is the button is released) and so to read inputs from it, a clock signal with 8 pulses is applied and data is shifted into an 8-bit variable to represent the current state of the 8 buttons of the controller. To prevent infinite detections from only one press, we store the previous presses on a variable and mask the current presses with that.

The fourth and fifth game logic blocks are if-statements that check if the select button (sound on/off toggle) or the start button (pause button) have been pressed. If the select button has been pressed, a software switch is toggled that sets the audio either on or off. If the start button has been pressed, the game enters the fourth and final game state in which the ATSAM constantly outputs SPI data of the gamestate at the time of pause and checks the controller input

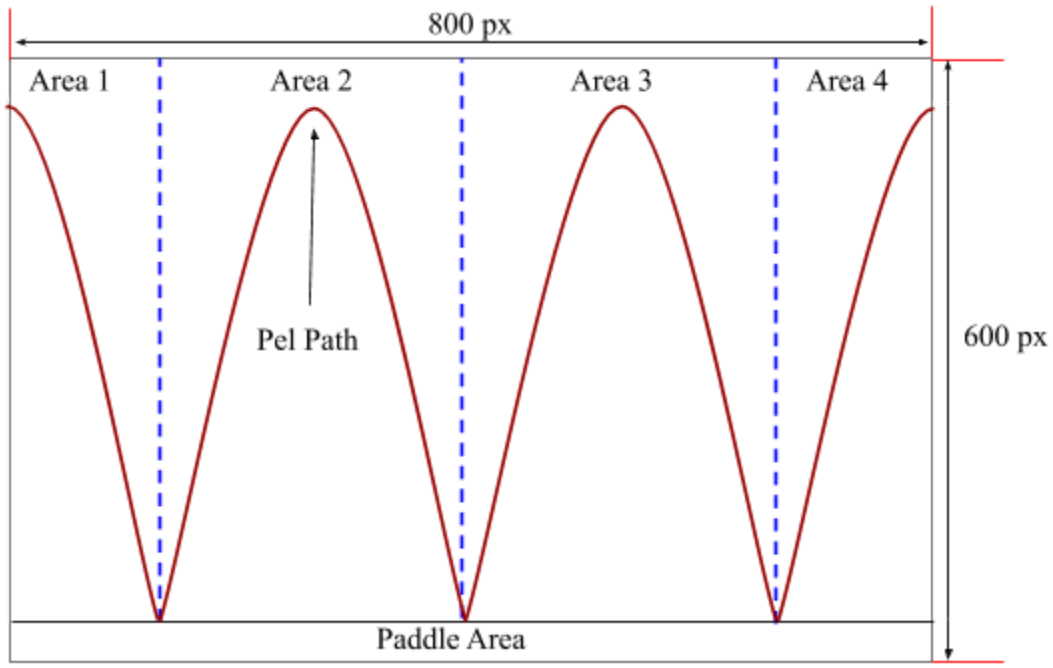

Figure 4. Helpful visualization of the area divisions and parabola paths. The area boundaries are represented by the dotted blue lines, and the parabola paths are represented by the solid red line

for start button presses. If the start button is pressed, the game unpauses and goes back to the gameplay state. The pause state is signaled to the FPGA by setting the gamestate pins to 2'b01.

The sixth logic game block is a simple if-else statement that changes the x-coordinate of the paddle depending on whether the left (A) or right (B) buttons are pressed. The paddle only has 3 locations it can be in (left, center and right), as those locations correspond to where the top of the paddle intersects with the pel parabola paths.

The seventh game logic block is a cleanup block that releases the pel objects that are dead. By dead we mean pels that have gone outside the screen boundaries, so any pels that leave the right screen or bottom screen boundaries are dead. This block is important as it opens up space for new pels in the static array that stores the existing pels.

On the topic of new pels, the eight game logic block is in charge of adding new pels. This block is quite important because it defines the game flow and game difficulty. Although there might be multiple ways of implementing the addition of new pels, we decided to go on a path that is based on waves. Much like some games send waves of enemies for users to battle independently, our game sends waves of pels for the user to juggle. The game is setup so that the first 6 waves send pels in a structured manner and waves above 6 send pels in a more randomized manner. Each wave sends an amount of pels corresponding to the wave number. For example, in wave one only one pel is sent out for the user to deal with. Once the pel from the first wave enters the fourth screen area, wave 2 is sent out, which means that two pels are sent out at a time. If the user survives to wave 15, 15 pels will be sent out at one time. Let's explain what "at one time" means in the case of our game. Previously we split the screen area in four areas -- for pel addition, areas 2 and 3 are further subdivided into two. This creates a total of six game areas. For waves 1 to 6, pels are released such that one pel exists in each game area at the same time. The exact time at which the pels are released relative to each other is randomized, however in order to avoid impossible games (in which a pel is released at the exact moment another pel is at the peak of their parabola path), a flag is set whenever a pel is located close to the peak of a parabola path that prevents the addition of new pels (until no pels exist close to a peak). For waves above 6, the amount of pels that exist in one game area at one time is randomized, however at least one pel must exist in each of the game areas at the same time. The amount of times each wave is repeated can be set in the code. A higher amount of repetitions makes the game easier, whilst a wave repetition of 1 makes the game very hard.

The ninth game logic block sends game information to the FPGA over SPI. For each pel that is currently in the game the ATSAM sends the (x, y, and color) as three 10-bit numbers. Although we are only using 9 bits for each color, to keep the transmission uniform we are wasting a bit. After sending all of the pels, the ATSAM then sends two 10 bits numbers corresponding to to the score and the lives. Finally, the ATSAM sends three 10-bit numbers to represent the paddle location and color, also as an (x, y, and color) tuple. We have no data to send from the FPGA to the ATSAM, so we are not using the MISO line. To make communication easier, we also have a signal from the ATSAM to the FPGA to signal the beginning and end of each SPI transfer. This signal is the fourth and final output pin that the ATSAM uses.

The tenth game logic block checks to see if the user has run out of lives. If the user has no more lives then the game goes out of the gameplay state and into the game over state. In the game over state, a "Game Over" text is displayed on the screen for three seconds and then the

game automatically exits the game over state and enters the start screen state. The game over state also resets game objects that must be reset, such as the pel array.

The final game logic block is just a delay. This delay is a hardcoded 8 milliseconds, as that amount of delay gives a smooth game flow.

In addition to the main code file, there are four header files that contain useful functions and constants. These header files are: musicPlayer.h, nes_controller.h, paddle.h, and pel.h. The musicPlayer.h file contains functions for initializing the Catalex MP3 player and for playing songs. The nes_controller.h file contains the functions for initializing the controller and for reading data from the controller. The paddle.h file defines a paddle structure and contains functions for initializing the paddle and sending paddle information over SPI. Similarly, the pel.h file defines a paddle structure and multiple constants related to the pels, and contained functions for adding pels, removing pels, and sending pels over SPI.

FPGA Design

The FPGA is responsible for taking graphics data from the ATSAM over SPI and sending it to the DACs and also for generating control signals for the VGA. The VGA controller section of the FPGA is very similar to the controller described in the MicroPs lecture except with different constants because we are running at a resolution of 800x600. The controller therefore has two counters, an X-coordinate and Y-coordinate counter. The X-coordinate counter increments every clock cycle and resets when it gets to the end of the line (screen width + front porch + hsync + back porch), and the Y-coordinate counter increments whenever the X-coordinate resets. The Y-coordinate correspondingly resets it gets to the end of the screen (screen height + front porch + vsync + back porch). The controller also emits hsync and vsync signals when in those areas, and emits a blank signal when not in the screen section. This resolution requires a clock speed of 40MHz, so we are using the oscillator on the MicroPs board and do not need a clock divider or PLL.

The FPGA has two data buffers for accepting data from the ATSAM; it reads SPI data into one buffer as a shift register while it renders data from the other buffer, and then swaps buffers at the start of the next frame after the transmission finishes. This setup allows SPI and the VGA controller to be run simultaneously on possibly different clocks without displaying any visual glitches that could be caused by rendering during the middle of a data transfer.

To decide what color each pixel is, the FPGA compares the pixel's coordinates against every Pel in the buffer and the paddle coordinates to see if any Pels are there. Since the background is black and no Pels should ever overlap we simply OR the color data of any Pels at the current coordinate together.

Rendering the 5 digits to display the score is also done in a manner similar to the way described in the MicroPs lecture, where monospaced bitmaps are loaded into memory, and then

accessed with X and Y coordinates. Because the screen is running at a higher resolution, higher resolution character bitmaps were created.

Text rendering is done differently. To look nicer than monospaced fonts, the entire block of text to be displayed, “Start”, “Paused”, and “Game Over” are encoded entirely as bitmaps and displayed accordingly, a sample of which is shown in Figure 5. Although it uses a lot of FPGA memory, this allows nice looking text without needing the complex logic for proper text rendering. The bitmaps were generated from black and white images using a python script that can be found in appendix C.

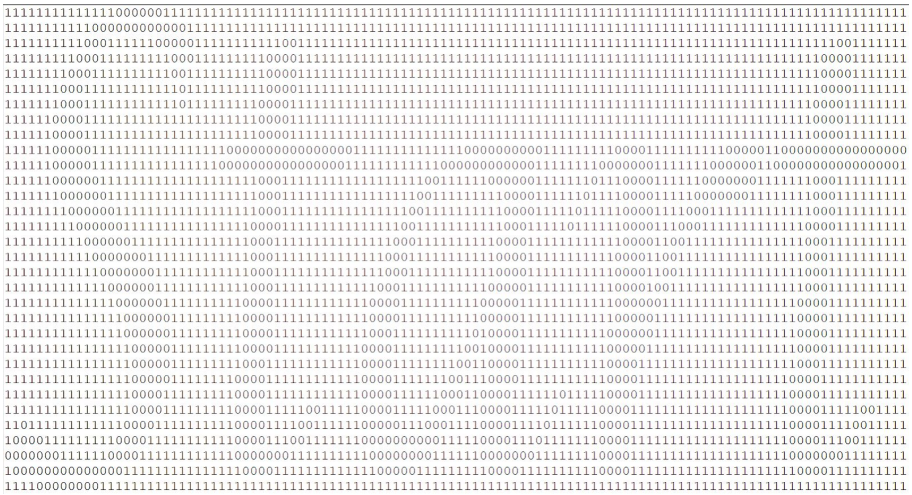

Figure 5. “Start” text display bitmap

Finally, the three dots for the score are displayed by comparing the current pixel coordinates with the current score and output accordingly. A block diagram of the overall FPGA design is shown in Figure 6.

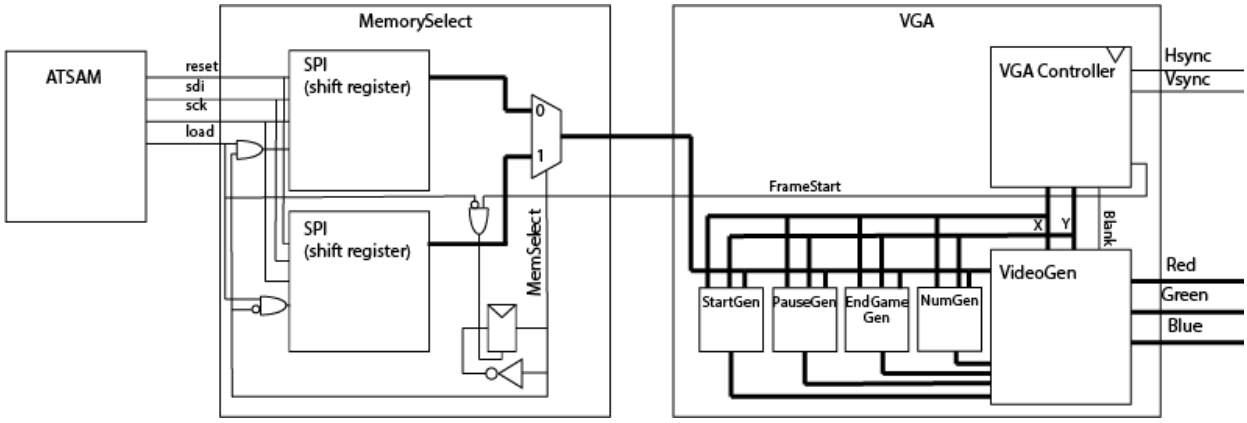

Figure 6. FPGA Block Diagram

Results

Given our hardware and development time constraints for this project our end results were satisfactory. We were able to interface with a standard video game controller, create reasonable-resolution VGA signal, produce sound, and create a cohesive video game that we consider fun.

Reynaldo Farias Zorrilla
Noah Boorstin

This project concluded with only one major difference from our project proposal, which was the exclusion of the ADV7125 video DAC in favor of a 3-bit resistor ladder. This tradeoff was made because of constraints on development time, and that the 3-bit resistor ladder performed well enough for this project. Initial testing with this chip produced an extremely noisy signal, as seen in Figure 7. Another problem noticed with this chip was ghosting was much worse than with the resistor ladder. Because our project did not need more than the three bits of data per color the resistor ladder allowed us and because it otherwise functioned better, the decision was made to use it instead of continuing to debug the chip. A likely cause for the difficulties faced with the ADV7125 was that we soldered it to a breakout board and attached it to our breadboard instead of making a custom PCB for it, which added much more capacitance than it was likely expecting.

Figure 7. Noisy ADV7125 DAC output

After replacing the ADV7125 with a 3-bit resistor ladder we were able to finish this project with only one issue we were not able to completely debug. That issue was “ghosting”, or the pixels’ colors bleeding into adjacent pixels, which was most likely caused by incorrect impedance matching between the 3-bit resistor ladder and the VGA transmission cable. An example of this problem is shown in Figure 8. As mentioned above, we were not able to completely fix ghosting, but adding two diodes between the DAC and VGA cable worked reasonably well.

A major limitation we faced in this project was that the FPGA did not have enough memory to hold a framebuffer (an image of the entire screen). This severely limited our display options because each frame had to be completely redrawn from scratch with no knowledge of what the previous frame held. This limitation was compounded by the VGA clock being the same speed as the main FPGA clock, so each pixel had to be computed in a single cycle. We therefore went with the not very scalable solution of having every pixel check against every pel

(and score, lives, and text pixel buffers), and ORing the result of every check together. Although this worked for our project, this method has a number of drawbacks. It uses an enormous

Figure 8. “Ghosting” of the blue line having a faint trail to the right

amount of FPGA area and scales extremely badly, so, for example, we would probably not be able to render 50 pels at once with this method. Furthermore, it relies on the background being black, but that was not a particular problem for our use case. Despite the obvious limitations of this approach, we chose it because not being able to use a framebuffer did not leave us many other options. There are a lot of graphical effects, such as small trails for the pels, we would have liked to explore if given the time and hardware, but that is beyond the scope of this project.

The only important software difficulty that we encountered was the logic for pel addition. Pel addition defines the game flow and whether the game is playable and fun. A game that adds pels such that the game is impossible is not fun, and a game that starts by having really low reaction margins between pel bounces is not fun either. A fun and playable game generally starts slow and gets harder progressively. This progression is hard to code, and we could not find an elegant solution to progression. The approach to progression that initially came to mind was to add more pels at faster rates as the game advances, but in our experience this did not make for a fun game. We analyzed the pel addition from the original game, and we theorized that they add pels using wave progression. As was already described in the microcontroller section above, we decided to approach the pel addition using waves. Each wave would add an additional pel, and the distribution of the pels would be randomized. We believe this method is a pretty good approach to progression, but our final version of wave control was not properly fine tuned. The difficulty progression was only based on adding a new pel, which is not very exciting. Additionally, even though the reaction time between pel bounces was supposed to be randomized, the game would repeatedly exhibit the same reaction time pattern for multiple waves (making the game boring to play). The randomization would also lead to really small reaction time margins for the beginning levels, which is not ideal. The implementation of the randomization would for sure have to be revised such that the reaction time margins are not small at the beginning of the level and such that they do not exhibit patterns. In addition, making

the reaction time margins large at the beginning of the game and decreasing them as the game goes on would make the game much more interesting and rewarding to play.

References

[1] CMOS 8-Stage Static Shift Register, https://www.ti.com/lit/ds/symlink/cd4021b-q1.pdf
[2] VGA Signal Timing, http://tinyvga.com/vga-timing
[3] 74HC04; 74HCT04 Hex Inverter, http://pages.hmc.edu/harris/class/e85/74HC04.pdf
[4] Catalex MP3 Player, http://geekmatic.in.ua/pdf/Catalex_MP3_board.pdf

Bill Of Materials

Part	**Source**	**Vendor Part #**	**Price**
VGA cable	Stockroom	N/A	N/A
NES controller (replica)	Amazon	NES-004	$9.79
Catalex MP3 Player	Rey	N/A	N/A
Speaker	Rey	N/A	N/A
VGA monitor	MicroPs Lab	N/A	N/A
MicroPs Board	MicroPs	uMudd Mark V.1	N/A

Total cost: $9.79

Reynaldo Farias Zorrilla
Noah Boorstin

Appendix A: C code

Project.c

```
//project.c
//Reynaldo Farias Zorrilla, Noah Boorstin
//This file contains the main game logic

////////////////////////////////////////////
// #includes
////////////////////////////////////////////
#include <stdio.h>
#include "SAM4S4B\SAM4S4B.h"
#include "nes_controller.h"
#include <stdlib.h>
#include "pel.h"
#include "paddle.h"
#include "musicPlayer.h"

////////////////////////////////////////////
// Constants
////////////////////////////////////////////

#define LOAD_PIN 29
#define RESET_PIN 30
#define SCREEN_HEIGHT 600
#define SCREEN_WIDTH 800
#define TEXT_PIN_1 5
#define TEXT_PIN_2 6
#define LEVELR 1

#define max(a, b) \
       ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a > _b ? _a : _b; })

#define min(a, b) \
       ({ __typeof__ (a) _a = (a); \
              __typeof__ (b) _b = (b); \
        _a > _b ? _b : _a; })

////////////////////////////////////////////
// Variables
////////////////////////////////////////////
short score = 0;
char prevController = 0;
char data = 0;
char validPresses = 0;
int test = 0;
int lives = 3;
int removePCount = 0;
int *removePelIndArr;
int error = 0;
int divI = 800 / 6;
int minParabolaHeight = 200;
int numArray[5];
int startScreen = 1;
int gamePlay = 0;
```

```
int gameOver = 0;
int parabolaHeight;
int *removePelIndC;
PEL *pelArray;
PADDLE_TYPE paddle;
long counter;
long rateCounter;
int modDiv;
int speedCounter = 0;
int level = 1;
int pelWave = 0;
int levelRep = LEVELR;
int noAddFlag = 0;
int noAddWindow = 30;
int pelReleaseArray[6] = {0};
int maxRand;
int scoreMod = 15;
int scoreCounter = 0;

////////////////////////////////////////////
// Function Prototypes
////////////////////////////////////////////
void scoreBreakDown(int score);
void sendScore(short score);
void collisionChecking(void);
void updatePaddleCoordinates(void);
void resetGameObjects(void);
void pause(void);
void addNewPel(void);
void populatePelAddArr(void);
int checkGameOver(void);
void updatePelLocs(void);

////////////////////////////////////////////
// Main
////////////////////////////////////////////
int main(void)
{

        //initializations
        samInit();
                  //SAM init
        uartInit(4, 260);
          //UART init
        paddle = paddleInit();
//paddle init
        initMusicPlayer();
          //music player init
        nes_controller_init();
//NES controller init
        led_init();
                  //LEDs init
        adcInit(ADC_MR_LOWRES_BITS_10);
//ADC init
        adcChannelInit(ADC_CH0, ADC_CGR_GAIN_X1, ADC_COR_OFFSET_OFF); //ADC channel 0 init
        spiInit(MCK_FREQ / 244000, 0, 1);                                                    //spi
init
```

```
	pioPinMode(LOAD_PIN, PIO_OUTPUT);
//load pin
	pioPinMode(RESET_PIN, PIO_OUTPUT);                                          //
reset pin
	int seed = (int)(adcRead(ADC_CH0) * 1000);
	srand(seed); //sets seed for randum number gen.

	//Allocations
	pelArray = malloc(MAXPELS * sizeof(PEL));   //pel array allocation
	removePelIndArr = malloc(10 * sizeof(int)); //allocate array for the removal of pels
	removePelIndC = removePelIndArr;                    //copy of removePelIndArr pointer
	if (pelArray == NULL | removePelIndArr == NULL)
	{ // malloc() was unable to allocate the memory, handle the error
		pioDigitalWrite(PIO_PA1, 1);
	}

	//Variable Initializations
	counter = 1;
	rateCounter = 0;
	modDiv = 600;
	parabolaHeight = paddle.y - PEL_SIZE;

	//output initialiations
	pioDigitalWrite(LOAD_PIN, 0);
	pioDigitalWrite(TEXT_PIN_1, 1);
	pioDigitalWrite(TEXT_PIN_2, 0);

	//resets FPGA
	pioDigitalWrite(RESET_PIN, 1);
	tcDelayMicroseconds(100);
	pioDigitalWrite(RESET_PIN, 0);
	tcDelayMillis(1500); //waits for fpga to start outputing

	//game state loop
	while (1)
	{

		while (startScreen)
		{
			//Read nes controller and find valid presses
			data = nes_controller_read();
			char data = nes_controller_read(); //read controller
			validPresses = (~prevController) & data;
			prevController = data;
			//checks for start press
			if (((validPresses >> 4) & 1))
			{
				gamePlay = 1;
				startScreen = 0;
				pioDigitalWrite(TEXT_PIN_1, 0);
				pioDigitalWrite(TEXT_PIN_2, 0);
				break;
			}
			//sends SPI data
			pioDigitalWrite(LOAD_PIN, 1);
			sendScore(score);
			spiSendRecieve10Bit(lives);
```

```
        sendPaddle(paddle);
        pioDigitalWrite(LOAD_PIN, 0);
} //startScreen loop

while (gamePlay)
{

        //resets adding flag
        noAddFlag = 0;

        //update pel locations
        updatePelLocs();

        //pel Addition Counter update
        if (counter)
        {
                counter--;
        }

        //checks for collisions
        collisionChecking();

        //Read nes controller and find valid presses
        data = nes_controller_read();
        char data = nes_controller_read(); //read controller
        validPresses = (~prevController) & data;
        prevController = data;

        //pauseCheck
        if (((validPresses >> 4) & 1))
        {
                pause();
                pioDigitalWrite(TEXT_PIN_1, 0);
                pioDigitalWrite(TEXT_PIN_2, 0);
        }

        //turn sound on and off
        if (((validPresses >> 5) & 1))
        {
                volumeToggle = ~volumeToggle;
        }

        //update paddle x-coordinate
        updatePaddleCoordinates();

        //remove dead pels
        if (removePCount)
        {
                removePel(pelArray, removePelIndArr, removePCount);
                removePelIndC = removePelIndArr;
                removePCount = 0;
        }

        //add new pel
        addNewPel();

        //send SPI-data
```

```
                        pioDigitalWrite(LOAD_PIN, 1);
                        sendPels(pelArray);                    //send pels
                        sendScore(score);                      //send score
                        spiSendRecieve10Bit(lives); //send lives number
                        sendPaddle(paddle);                    //send paddle
                        pioDigitalWrite(LOAD_PIN, 0);

                        //Game Over Check
                        if (checkGameOver())
                        {
                                break;
                        }

                        //update speed counter
                        speedCounter++;

                        //timing code
                        tcDelayMillis(8); //this delay just makes it look wonderful smooth

                } //gameplay loop

                while (gameOver)
                {
                        startScreen = 1;
                        gameOver = 0;
                        resetGameObjects();
                        break;
                } //gameoverloop

        } //game state loop

} //main

//updates pel locations
void updatePelLocs(void)
{
        for (int i = 0; i < pelCount; i++)
        {
                PEL *currentPel = &pelArray[i];
                currentPel->x++;
                int tempY = currentPel->y;
                //y-coordinate update
                currentPel->y = (int)(-1 * (((-1) * (((float)currentPel->maxHeight) /
((float)(divI * divI))) * ((float)((currentPel->x - currentPel->area * divI * 2) * (currentPel->x
- currentPel->area * divI * 2))) + (float)currentPel->maxHeight) - (float)parabolaHeight));
                //no pel addition check
                if (((currentPel->x <= 266 + noAddWindow) && (currentPel->x >= 266 - noAddWindow))
|| ((currentPel->x <= 532 + noAddWindow) && (currentPel->x >= 532 - noAddWindow)))
                {
                        noAddFlag = 1;
                }
                //score update logic
                if ((currentPel->y - tempY) != 0)
                {
                        scoreCounter++;
                }
                if (scoreCounter % scoreMod == 0)
```

```
                {
                        score++;
                }
        }
}

//checks for gameover
int checkGameOver(void)
{
        if (lives <= 0)
        {
                gamePlay = 0;
                gameOver = 1;
                pioDigitalWrite(TEXT_PIN_1, 1);
                pioDigitalWrite(TEXT_PIN_2, 1);

                int countDown = 3000 / 8;
                while (countDown)
                {
                        pioDigitalWrite(LOAD_PIN, 1);
                        sendPels(pelArray);
                        sendScore(score);
                        spiSendRecieve10Bit(lives);
                        sendPaddle(paddle);
                        pioDigitalWrite(LOAD_PIN, 0);
                        tcDelayMillis(8);
                        countDown--;
                }
                return 1;
        }
        else
                return 0;
}

//helper function for wave control. Decides the amount of pels per area
void populatePelAddArr(void)
{
        maxRand = level - 6;
        for (int i = 0; i < 6; i++)
        {
                int currentRand = max(1, rand() % (maxRand + 2));
                pelReleaseArray[i] = currentRand;
                maxRand -= currentRand - 1;
        }
}

//logic for adding new pels
void addNewPel(void)
{
        if ((counter == 0) && (!noAddFlag))
        {
                addPel(pelArray, parabolaHeight);
                if (level > 5)
                {
                        counter = (rand() % 105) + (133) * (0) + 10; //could be a discrete wave by
using pelWave instead of 0
                        pelReleaseArray[pelWave]--;
```

```
                        if (pelReleaseArray[pelWave] <= 0)
                        {
                                pelWave++;
                        }
                        if (pelWave == level)
                        {
                                levelRep--;
                                if (levelRep == 0)
                                {
                                        levelRep = LEVELR;
                                        if (level < 21)
                                        {
                                                level++;
                                        }
                                        populatePelAddArr();
                                }
                                pelWave = 0;
                        }
                }
                else
                {
                        counter = (rand() % 105) + (133) * ((6 - level)) + 10;
                        pelWave++;
                        if (pelWave == level)
                        {
                                levelRep--;
                                if (levelRep == 0)
                                {
                                        levelRep = LEVELR;
                                        level++;
                                        populatePelAddArr();
                                }
                                pelWave = 0;
                        }
                }
        }
}

//breaks down the score into digits
void scoreBreakDown(int score)
{
        char result[5];
        int num;

        sprintf(result, "%.5d", score);
        for (int i = 0; i < 5; i++)
        {
                sscanf(result + i, "%1d", &num);
                numArray[i] = num;
        }
}

//sends score over SPI
void sendScore(short score)
{
        scoreBreakDown(score);
        short first10 = (numArray[4] << 6 | (numArray[3] << 2) | numArray[2] >> 2) & 0x3FF;
```

```
        short second10 = (numArray[2] << 8 | (numArray[1] << 4) | numArray[0]) & 0x3FF;
        spiSendRecieve10Bit(first10);
        spiSendRecieve10Bit(second10);
}

//resets the game
void resetGameObjects(void)
{
        score = 0;
        pelCount = 0;
        memset(removePelIndArr, 0, 10 * sizeof(int));
        lives = 3;
        counter = 1;
        rateCounter = 0;
        removePCount = 0;
        modDiv = 600;
        speedCounter = 0;
        level = 1;
        pelWave = 0;
        pioDigitalWrite(TEXT_PIN_1, 1);
        pioDigitalWrite(TEXT_PIN_2, 0);
}

//function that runs during pause
void pause(void)
{
        pioDigitalWrite(TEXT_PIN_1, 0);
        pioDigitalWrite(TEXT_PIN_2, 1);
        while (1)
        {
                data = nes_controller_read();
                char data = nes_controller_read(); //read controller
                validPresses = (~prevController) & data;
                prevController = data;
                //checks for unpause
                if (((validPresses >> 4) & 1))
                {
                        break;
                }
                //send SPI-data
                pioDigitalWrite(LOAD_PIN, 1);
                sendPels(pelArray);
                sendScore(score);
                spiSendRecieve10Bit(lives);
                sendPaddle(paddle);
                pioDigitalWrite(LOAD_PIN, 0);
        }
}

//self explanatory function name
void collisionChecking(void)
{
        //check for collisions
        for (int i = 0; i < pelCount; i++)
        {

                PEL *currentPel = &pelArray[i];
```

```
		switch (currentPel->area)
		{
		case 0:
			//bottom screen collisions
			if (currentPel->y >= SCREEN_HEIGHT)
			{
				lives--;
				removePCount++;
				*removePelIndC = i;
				removePelIndC++;
				playSong(0x05);
			}
			//paddle collisions
			else if ((currentPel->y >= parabolaHeight) & (currentPel->area ==
paddle.area))
			{
				currentPel->area = 1;
				currentPel->y = paddle.y - (1 + PEL_SIZE);
				currentPel->maxHeight = max(rand() % parabolaHeight,
minParabolaHeight);
				playSong(0x01);
			}

			break;

		case 1:
			//bottom screen collisions
			if (currentPel->y >= SCREEN_HEIGHT)
			{
				lives--;
				removePCount++;
				*removePelIndC = i;
				removePelIndC++;
				playSong(0x05);
			}
			//paddle collisions
			else if ((currentPel->y >= parabolaHeight) & (currentPel->area ==
paddle.area))
			{
				currentPel->area = 2;
				currentPel->y = paddle.y - (1 + PEL_SIZE);
				currentPel->maxHeight = max(rand() % parabolaHeight,
minParabolaHeight);
				playSong(0x02);
			}
			break;

		case 2:
			//bottom screen collisions
			if (currentPel->y >= SCREEN_HEIGHT)
			{
				lives--;
				removePCount++;
				*removePelIndC = i;
				removePelIndC++;
				playSong(0x05);
```

```
                        }
                        //paddle collisions
                        else if ((currentPel->y >= parabolaHeight) & (currentPel->area ==
paddle.area))
                        {
                                currentPel->area = 3;
                                currentPel->y = paddle.y - (1 + PEL_SIZE);
                                currentPel->maxHeight = parabolaHeight - 50; //exit height
                                playSong(0x03);
                        }
                        break;
                case 3:
                        if (currentPel->x + PEL_WIDTH >= SCREEN_WIDTH)
                        {
                                removePCount++;
                                *removePelIndC = i;
                                removePelIndC++;
                        }
                        break;
                default:
                        //error handling?
                        break;
                }
        }
}

//another self explanatory function name
void updatePaddleCoordinates(void)
{
        if (((validPresses >> 1) & 1) || ((validPresses >> 6) & 1)) //move left
        {
                switch (paddle.x)
                {
                case 266:
                        paddle.area = 0;
                        paddle.x = 10;
                        break;
                case 532:
                        paddle.area = 1;
                        paddle.x = 266;
                        break;
                default:
                        paddle.x = 10;
                        break;
                }
                playSong(0x06);
        }
        else if (((validPresses >> 0) & 1) || ((validPresses >> 7) & 1))
        { //move right
                switch (paddle.x)
                {
                case 10:
                        paddle.area = 1;
                        paddle.x = 266;
                        break;
                case 266:
                        paddle.area = 2;
```

```
                paddle.x = 532;
                break;
        default:
                paddle.x = 532;
                break;
        }
        playSong(0x06);
    }
}
```

Reynaldo Farias Zorrilla
Noah Boorstin

Pel.h

```
//pel.h
//Reynaldo Farias Zorrilla, Noah Boorstin
//This file contains pel information and pel helper functions

#ifndef PEL_H
#define PEL_H

#include <stdint.h>
#include "SAM4S4B\SAM4S4B.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


//Bit field for the Pel_Obj
typedef struct {
               uint64_t maxHeight: 10;
               uint64_t area: 2;
    uint64_t color : 9;
    uint64_t y          : 10;
         uint64_t x            : 10;
} PEL;

//Pel count
short pelCount = 0;
//array of allowable colors
short colorArr[] = {0x1C0,0x38,0x7,0x3F,0x1FF,0x1F8,0x1C7};
//Pel dimensions
#define PEL_SIZE 10*2
#define PEL_WIDTH 6*2
//maximum allowable pels
#define MAXPELS 20

//sends pels over SPI
void sendPels(PEL* pelArray){
        for(int i = 0; i<pelCount;i++){
                spiSendRecieve10Bit(pelArray[i].color); //x
                spiSendRecieve10Bit((pelArray[i].x)); //y
                spiSendRecieve10Bit((pelArray[i].y)); //color
        }

}


//Adds a new pel
void addPel(PEL* pelArray, int initialHeight){
        PEL newPel;
        newPel.maxHeight = initialHeight-50;
        newPel.area = 0;
        newPel.color = colorArr[rand()%7];//random number between 0 and 6
        newPel.y = 50;
        newPel.x = 1;
        pelArray[pelCount] = newPel;
        pelCount++;
```

```
}

//removes a single from the pel array
void removePel(PEL* pelArray, int* pelIndArr, int pelIndCount){
        for(int i = 0; i<pelIndCount;i++){
                PEL *temp = (pelArray + pelIndArr[i]);
                PEL *temp1 = (pelArray + pelIndArr[i] +1);
                memmove((void*) temp, (const void*) temp1, (pelCount - pelIndArr[i]
-1)*sizeof(PEL));
                pelCount--;
        }
        memset((void*)pelIndArr, 0, pelIndCount*sizeof(int));
}


#endif //PEL_H
```

Reynaldo Farias Zorrilla
Noah Boorstin

Paddle.h

```
//paddle.h
//Reynaldo Farias Zorrilla, Noah Boorstin
//This file contains paddle information and paddle helper functions
#ifndef PADDLE_H
#define PADDLE_H

#include <stdint.h>
#include "SAM4S4B\SAM4S4B.h"

//paddle height
#define PADDLE_HEIGHT 8

typedef struct {
                uint32_t area: 2;
    uint32_t color : 9;
    uint32_t y          : 10;
          uint32_t x            : 10;
} PADDLE_TYPE;

//initializes paddle
PADDLE_TYPE paddleInit(void) {
                PADDLE_TYPE paddle;
                paddle.area = 0;
                paddle.x = 10; //starts it on the left
                paddle.y = 570; // constant paddle height
                paddle.color = 0b111111111; //idk what the paddle color should be
return paddle;
}

//sends paddle data over SPI
void sendPaddle(PADDLE_TYPE paddle){
                spiSendRecieve10Bit(paddle.color); //color
                spiSendRecieve10Bit(paddle.x); //x
                spiSendRecieve10Bit(paddle.y); //y
}

#endif //PADDLE_H
```

Reynaldo Farias Zorrilla
Noah Boorstin

Nes_controller.h

```
//nes_controller.h
//Reynaldo Farias Zorrilla, Noah Boorstin
//This file contains controller information and helper functions

#ifndef NES_CONTROLLER_H
#define NES_CONTROLLER_H

#include <stdint.h>
#include "SAM4S4B\SAM4S4B.h"

// Bit field struct for the controller
typedef struct {
    uint32_t right : 1;
    uint32_t left  : 1;
    uint32_t down  : 1;
    uint32_t up         : 1;
    uint32_t start : 1;
    uint32_t select: 1;
    uint32_t B          : 1;
          uint32_t A            : 1;
} NES_CONTROLLER;

//controller interface pins
#define CONTROLLER_CLK PIO_PA4
#define CONTROLLER_LATCH PIO_PA3
#define CONTROLLER_DATA PIO_PA28

//delay between shifts
#define CONTROLER_SPEED 10

//initializes 7 LEDs on uMudd board
void led_init(void) {
        pioPinMode(PIO_PA0, PIO_OUTPUT);
        pioPinMode(PIO_PA1, PIO_OUTPUT);
        pioPinMode(PIO_PA2, PIO_OUTPUT);
        pioPinMode(PIO_PA29, PIO_OUTPUT);
        pioPinMode(PIO_PA30, PIO_OUTPUT);
        pioPinMode(PIO_PA5, PIO_OUTPUT);
        pioPinMode(PIO_PA6, PIO_OUTPUT);
}

//initializes controller
void nes_controller_init(void) {
        pioInit();
        tcDelayInit();
        pioPinMode(CONTROLLER_CLK, PIO_OUTPUT);
        pioDigitalWrite(CONTROLLER_CLK, 0);
        pioPinMode(CONTROLLER_LATCH, PIO_OUTPUT);
        pioPinMode(CONTROLLER_DATA, PIO_INPUT);
}

//sets all LEDS to a value
void setLEDS(char LEDs) {
        pioDigitalWrite(PIO_PA0, 1 & (LEDs >> 0));
        pioDigitalWrite(PIO_PA1, 1 & (LEDs >> 1));
        pioDigitalWrite(PIO_PA2, 1 & (LEDs >> 2));
```

```
        pioDigitalWrite(PIO_PA29, 1 & (LEDs >> 3));
        pioDigitalWrite(PIO_PA30, 1 & (LEDs >> 4));
        pioDigitalWrite(PIO_PA5, 1 & (LEDs >> 5));
        pioDigitalWrite(PIO_PA6, 1 & (LEDs >> 6));
}


/* read from the controller
         - set latch low
         - apply clock, shift in data
         - set latch high
*/
char nes_controller_read(void) {
        //set latch low
        pioDigitalWrite(CONTROLLER_LATCH, 0);
        char result = 0;

        for(int i=0; i<8; i++) {
                result = (result << 1 ) | (!pioDigitalRead(CONTROLLER_DATA));
                pioDigitalWrite(CONTROLLER_CLK, 0);
                tcDelayMicroseconds(CONTROLER_SPEED);
                pioDigitalWrite(CONTROLLER_CLK, 1);
                tcDelayMicroseconds(CONTROLER_SPEED);
        }

        //set latch high
        pioDigitalWrite(CONTROLLER_LATCH, 1);
        //setLEDS(result);

        return  result;
}

#endif //NES_CONTROLLER_H
```

Reynaldo Farias Zorrilla
Noah Boorstin

MusicPlayer.h

```
//musicPlayer.h
//Reynaldo Farias Zorrilla, Noah Boorstin
//This file contains functions that help interface the Catalex MP3 player module

#ifndef MUSICPLAYER_H
#define MUSICPLAYER_H

#include <stdint.h>
#include "SAM4S4B\SAM4S4B.h"

//hexadecimal command constants
#define NEXT_SONG 0X01
#define PREV_SONG 0X02
#define CMD_PLAY_W_INDEX 0X03 //DATA IS REQUIRED (number of song)
#define VOLUME_UP_ONE 0X04
#define VOLUME_DOWN_ONE 0X05
#define CMD_SET_VOLUME 0X06//DATA IS REQUIRED (number of volume from 0 up to 30(0x1E))
#define SET_DAC 0X17
#define CMD_PLAY_WITHVOLUME 0X22 //data is needed  0x7E 06 22 00 xx yy EF;(xx volume)(yy number
of song)
#define CMD_SEL_DEV 0X09 //SELECT STORAGE DEVICE, DATA IS REQUIRED
#define SLEEP_MODE_START 0X0A
#define SLEEP_MODE_WAKEUP 0X0B
#define CMD_RESET 0X0C//CHIP RESET
#define CMD_PLAY 0X0D //RESUME PLAYBACK
#define CMD_PAUSE 0X0E //PLAYBACK IS PAUSED
#define CMD_PLAY_WITHFOLDER 0X0F//DATA IS NEEDED, 0x7E 06 0F 00 01 02 EF;(play the song with the
directory \01\002xxxxxx.mp3
#define STOP_PLAY 0X16
#define PLAY_FOLDER 0X17// data is needed 0x7E 06 17 00 01 XX EF;(play the 01 folder)(value xx we
dont care)
#define SET_CYCLEPLAY 0X19//data is needed 00 start; 01 close
#define SET_DAC 0X17//data is needed 00 start DAC OUTPUT;01 DAC no output

int volumeToggle = 0;

//sends command
void sendCommand(char data){
	char sendArr[] = {0x7E,0xFF,0x06,data,0x00,0x00,0x01,0xEF};
	for(int i=0;i<8;i++){
		uartTx(sendArr[i]);
	}
}

//sends command that requires additional data
void sendCommandAdditional(char data,char add1, char add2){
	char sendArr[] = {0x7E,0xFF,0x06,data,0x00,add1,add2,0xEF};
	for(int i=0;i<8;i++){
		uartTx(sendArr[i]);
	}

}

//initialize music player
void initMusicPlayer(){
	sendCommandAdditional(CMD_SEL_DEV,0x00,0x02);
```

```
}

//plays a song that is accessed by an index
void playSong(char index){
        if(volumeToggle){
                sendCommandAdditional(CMD_PLAY_WITHVOLUME,0x1E,index);
        }
}

#endif //MUSICPLAYER_H
```

Appendix B: Verilog

project.sv

```
// project.sv
// Author: Noah Boorstin
// nboorstin@hmc.edu
// 12/8/19
// This file contains the top-level project module
// and spi and buffer switch modules


`define MAX_PELS 20

module project(input  logic clk, reset,
       input  logic sck,
       input  logic sdi,
       input  logic load,
                  input  logic [1:0] textSelect,
                output logic [2:0] red, green, blue,
                output logic hsync, vsync);

       logic [127:0] key, plaintext, cyphertext;
     logic [1:0]  paddlePos;
     logic [`MAX_PELS-1:0][29:0] pelData;
     logic [`MAX_PELS-1:0][29:0] dataIn;
     logic [29:0] paddleIn, paddleData;
     logic        frameStart;
     logic [19:0] score;
     logic [9:0]  lives;

     vga vgamod(clk, reset, score, lives, paddleData, pelData, textSelect, red, green, blue,
hsync, vsync, frameStart);

     memorySelect mem(clk, reset, frameStart & (~load), load, sck, sdi, score, lives, paddleData,
pelData);
endmodule


module memorySelect(input  logic clk, reset, change, load,
                                        input  logic sck, sdi,
                                        output logic [19:0] score,
                                        output logic [9:0]  lives,
                                        output logic [29:0] paddlePosition,
                                        output logic [`MAX_PELS-1:0][29:0] pelData);

    logic memSelect;
    logic [`MAX_PELS-1:0][29:0] pel1;
    logic [`MAX_PELS-1:0][29:0] pel2;
    logic [29:0] paddle1, paddle2;
    logic [19:0] score1, score2;
    logic [9:0]  lives1, lives2;
    always_ff@(posedge clk, posedge reset)
        if(reset)
                memSelect <= 0;
        else if(change)
                        memSelect <= ~memSelect;

    spi spi1(sck, reset, clk, sdi, load & ~memSelect, pel1, paddle1, score1, lives1);
    spi spi2(sck, reset, clk, sdi, load & memSelect, pel2, paddle2, score2, lives2);

    assign paddlePosition = memSelect ? paddle1 : paddle2;
    assign pelData = memSelect ? pel1 : pel2;
    assign score = memSelect ? score1 : score2;
    assign lives = memSelect ? lives1 : lives2;
endmodule
```

```
module spi(input  logic sck, reset, clk,
               input  logic sdi, load,
               output logic [`MAX_PELS-1:0][29:0] pelData,
                            output logic [29:0] paddle,
                            output logic [19:0] score,
                            output logic [9:0] lives);
    logic lastLoad;
    logic clear;

    always_ff@(posedge clk)
        lastLoad <= load;

    assign clear = (load & ~lastLoad) | reset;

    always_ff @(posedge sck, posedge clear)
        if (clear) begin
                {score[19:0], lives[9:0], paddle[29:0]} <= 60'd0;
                pelData[0][29:0] <= 30'd0;
        end else    if (load) begin
                {score[19:0], lives[9:0], paddle[29:0]} <= {score[18:0], lives[9:0],
paddle[29:0], sdi};
                pelData[0][29:0] <= {pelData[0][28:0], score[19]};
        end
    genvar i;
    generate
        for(i=1; i<`MAX_PELS; i++) begin: forloop
                always_ff @(posedge sck, posedge clear)
                        if (clear)
                                pelData[i][29:0] <= 30'd0;
                        else if (load)
                                pelData[i][29:0] <= {pelData[i][28:0], pelData[i-1][29]};
        end
    endgenerate

endmodule
```

vga.sv

```
// vga.sv
// Author: Noah Boorstin
// nboorstin@hmc.edu
// 12/8/19
// This file contains the vga controller, videogen,
// and text rendering modules

`define MAX_PELS 20

// VGA constants
`define X_BACK_PORCH 88
`define X_FRONT_PORCH 40
`define X_DISP_WIDTH 800
`define X_SYNC 128
`define X_TOTAL (`X_BACK_PORCH + `X_FRONT_PORCH + `X_DISP_WIDTH + `X_SYNC)

`define Y_BACK_PORCH 23
`define Y_FRONT_PORCH 1
`define Y_DISP_WIDTH 600
`define Y_SYNC 4
`define Y_TOTAL (`Y_BACK_PORCH + `Y_FRONT_PORCH + `Y_DISP_WIDTH + `Y_SYNC)

// color constants
`define BACKGROUND 9'b000000000

// paddle display constants
`define PADDLE_WIDTH 266
`define PADDLE_HEIGHT 20

// pel display constants
```

Reynaldo Farias Zorrilla
Noah Boorstin

```
`define PEL_WIDTH 6*2
`define PEL_HEIGHT 10*2

// lives display constants
`define LIVES_START_X 394
`define LIVES_START_Y 10
`define LIVES_WIDTH 4
`define LIVES_HEIGHT 5
`define LIVES_SPACING 2
`define LIVES_COLOR 9'b111111111
`define MAX_LIVES 3

// score display constants
`define SCORE_START_X 640

// text display constants
`define ENDGAME_WIDTH 245
`define ENDGAME_START_X 278
`define ENDGAME_HEIGHT 40
`define ENDGAME_START_Y 200

`define PAUSED_WIDTH 168
`define PAUSED_START_X 316
`define PAUSED_HEIGHT 36
`define PAUSED_START_Y 200

`define START_WIDTH 114
`define START_START_X 343
`define START_HEIGHT 32
`define START_START_Y 200

// VGA interface module
module vga(input  logic clk, reset,
                  input  logic [19:0] score,
                  input  logic [9:0] lives,
                  input  logic [29:0] paddlePos,
                  input  logic [`MAX_PELS - 1:0][29:0] pelData,
                  input  logic [1:0] textSelect,
                  output logic [2:0] red, green, blue,
                  output logic hsync, vsync, frameStart);

    // video coordinates
    logic [10:0] x, y;
    logic blank;


    videogen vidgen(blank, x, y, score, lives, paddlePos, pelData, textSelect, red, green, blue);
    vgacontroller con2(clk, reset, hsync, vsync, blank, frameStart, x, y);

endmodule

module vgacontroller(input  logic clk, reset,
                                              output logic hsync, vsync, blank, frameStart,
                                              output logic [10:0] x, y);
    // VGA coordinate flops and incrementing
    always_ff @(posedge clk)
        if (reset) begin
               x <= 0;
               y <= 0;
        end
        else if (x > `X_TOTAL) begin
               x <= 0;
               if (y > `Y_TOTAL)
                      y <= 0;
               else
                      y <= y + 11'd1;
```

```
        end else
                x <= x + 11'd1;

    // sync and blank signals
    assign hsync = ~((x > `X_DISP_WIDTH + `X_FRONT_PORCH) &
                                    (x <= `X_DISP_WIDTH + `X_FRONT_PORCH + `X_SYNC));
    assign vsync = ~((y > `Y_DISP_WIDTH + `Y_FRONT_PORCH) &
                                    (y <= `Y_DISP_WIDTH + `Y_FRONT_PORCH + `Y_SYNC));
    assign blank = (x > `X_DISP_WIDTH) | (y > `Y_DISP_WIDTH);
    assign frameStart = (x == 0) & (y == 0);
endmodule

module videogen(input  logic blank,
                                input  logic [10:0] x, y,
                                input  logic [19:0] score,
                                input  logic [9:0] lives,
                                input  logic [29:0] paddlePos,
                                input  logic [`MAX_PELS - 1:0][29:0] pelData,
                                input  logic [1:0] textSelect,
                                output logic [2:0] red, green, blue);
    logic [8:0] pixel;

    // blanking logic
    assign green = ~pixel[2:0] & ~blank;
    assign red = ~pixel[5:3] & ~blank;
    assign blue = ~pixel[8:6] & ~blank;

    logic [`MAX_PELS - 1:0][8:0]pixBuf;
    logic [`MAX_LIVES - 1:0][8:0] livesBuf;
    logic [4:0][8:0] num, numBuf;
    logic [8:0] endgame, endgameBuf;
    logic [8:0] paused, pausedBuf;
    logic [8:0] start, startBuf;

    endgamegen endgamegen0(x[7:0] - `ENDGAME_START_X, y[5:0] - `ENDGAME_START_Y, endgame);
    assign endgameBuf = (x > `ENDGAME_START_X & x <= `ENDGAME_START_X + `ENDGAME_WIDTH &
                                                    y >= `ENDGAME_START_Y & y < `ENDGAME_START_Y
+ `ENDGAME_HEIGHT &
                                                    textSelect == 2'b11) ? endgame :
`BACKGROUND;

    pausedgen pausedgen0(x[7:0] - `PAUSED_START_X, y[5:0] - `PAUSED_START_Y, paused);
    assign pausedBuf = (x > `PAUSED_START_X & x <= `PAUSED_START_X + `PAUSED_WIDTH &
                                                    y >= `PAUSED_START_Y & y < `PAUSED_START_Y +
`PAUSED_HEIGHT &
                                                    textSelect == 2'b01) ? paused : `BACKGROUND;

    startgen startgen0(x[7:0] - `START_START_X, y[4:0] - `START_START_Y, start);
    assign startBuf = (x > `START_START_X & x <= `START_START_X + `START_WIDTH &
                                                    y >= `START_START_Y & y < `START_START_Y +
`START_HEIGHT &
                                                    textSelect == 2'b10) ? start : `BACKGROUND;
    genvar i;
        generate //pels
                for(i=0; i<`MAX_PELS; i++) begin: forloop
                                assign pixBuf[i] = (x > pelData[i][19:10] & x <= pelData[i][19:10]
+ `PEL_WIDTH &
                                                                                        y > pelData[i][9:0]
& y <= pelData[i][9:0] + `PEL_HEIGHT)
                                                                                        ? pelData[i][28:20]
: `BACKGROUND;
                end
        endgenerate

        generate //lives
                for(i=0; i<`MAX_LIVES; i++) begin: forloop2
```

```
                                assign livesBuf[i] = (x > `LIVES_START_X + i*(`LIVES_WIDTH +
`LIVES_SPACING) & x <= `LIVES_START_X + i*(`LIVES_WIDTH + `LIVES_SPACING) + `LIVES_WIDTH &
                                                                                y > `LIVES_START_Y &
y <= `LIVES_START_Y + `LIVES_HEIGHT & (lives > i) )
                                                                                 ? `LIVES_COLOR :
`BACKGROUND;
                end
        endgenerate

        generate //score
                for(i=0; i<5; i++) begin: forloop3
                        numgen numgen0(score[i*4+3:i*4], x[4:0] - `SCORE_START_X, y[4:0], num[i]);
                        assign numBuf[i] = (x >= `SCORE_START_X + 32 * i & x < `SCORE_START_X +
32*(i+1) &
                                                                        y >= 0      & y < 32) ?
num[i] : `BACKGROUND;
                end
        endgenerate

    always begin

        if (x > paddlePos[19:10] & x <= paddlePos[19:10] + `PADDLE_WIDTH &
                 y > paddlePos[9:0] & y <= paddlePos[9:0] + `PADDLE_HEIGHT)
                pixel = paddlePos[28:20];
        else
                pixel = pixBuf[0] | pixBuf[1] | pixBuf[2] | pixBuf[3] | pixBuf[4] |
                                pixBuf[5] | pixBuf[6] | pixBuf[7] | pixBuf[8] | pixBuf[9] |
                                pixBuf[10] | pixBuf[11] | pixBuf[12] | pixBuf[13] | pixBuf[14] |
                                pixBuf[15] | pixBuf[16] | pixBuf[17] | pixBuf[18] | pixBuf[19] |
                                livesBuf[0] | livesBuf[1] | livesBuf[2] |
                                numBuf[0] | numBuf[1] | numBuf[2] | numBuf[3] | numBuf[4] |
endgameBuf | pausedBuf | startBuf;
    end

endmodule

module numgen(input  logic [3:0] num,
                        input  logic [4:0] x, y,
                        output logic [8:0] pixel);
    logic [31:0] numrom[319:0];
    logic [31:0] line;
    initial
        $readmemb("numsBig.txt", numrom);
    assign line = {numrom[y+{num, 5'b00000}]};
    assign pixel = line[31-x] ? 9'b111000111 : `BACKGROUND;
endmodule

module endgamegen(input  logic [7:0] x,
                              input  logic [5:0] y,
                             output logic [8:0] pixel);
    logic [`ENDGAME_WIDTH - 1:0] endgameRom[`ENDGAME_HEIGHT - 1:0];

    logic [`ENDGAME_WIDTH - 1:0] endgameLine;

    initial
        $readmemb("endgame.txt", endgameRom);

    assign endgameLine = {endgameRom[y]};

    assign pixel = endgameLine[`ENDGAME_WIDTH - x] ? `BACKGROUND : 9'b111111000;
endmodule

module pausedgen(input  logic [7:0] x,
                              input  logic [5:0] y,
                             output logic [8:0] pixel);
    logic [`PAUSED_WIDTH - 1:0] pausedRom[`PAUSED_HEIGHT - 1:0];
```

```
    logic [`PAUSED_WIDTH - 1:0] pausedLine;

    initial
        $readmemb("paused.txt", pausedRom);

    assign pausedLine = {pausedRom[y]};

    assign pixel = pausedLine[`PAUSED_WIDTH - x] ? `BACKGROUND : 9'b111111000;
endmodule

module startgen(input  logic [7:0] x,
                               input  logic [4:0] y,
                              output logic [8:0] pixel);
    logic [`START_WIDTH - 1:0] startRom[`START_HEIGHT - 1:0];

    logic [`START_WIDTH - 1:0] startLine;

    initial
        $readmemb("start.txt", startRom);

    assign startLine = {startRom[y]};

    assign pixel = startLine[`START_WIDTH - x] ? `BACKGROUND : 9'b111111000;
endmodule
```

Reynaldo Farias Zorrilla
Noah Boorstin

Appendix C: Python script for creating bitmaps

convertText.py

```
from PIL import Image

for i in {"endgame", "paused", "start"}:
        with open(i+".txt", "w") as txt:
        im = Image.open(str(i)+".png")
        w, h = im.size
        pix = im.load()
        for j in range(h):
                for k in range(w):
                txt.write(str(int(1 - pix[k,j] / 255)))
                txt.write("\n")
```