Dance, Dance Revolution!

Final Project Report
December 13, 2019
E155

Stephanie Clifner and Elizabeth Hedenberg

Abstract

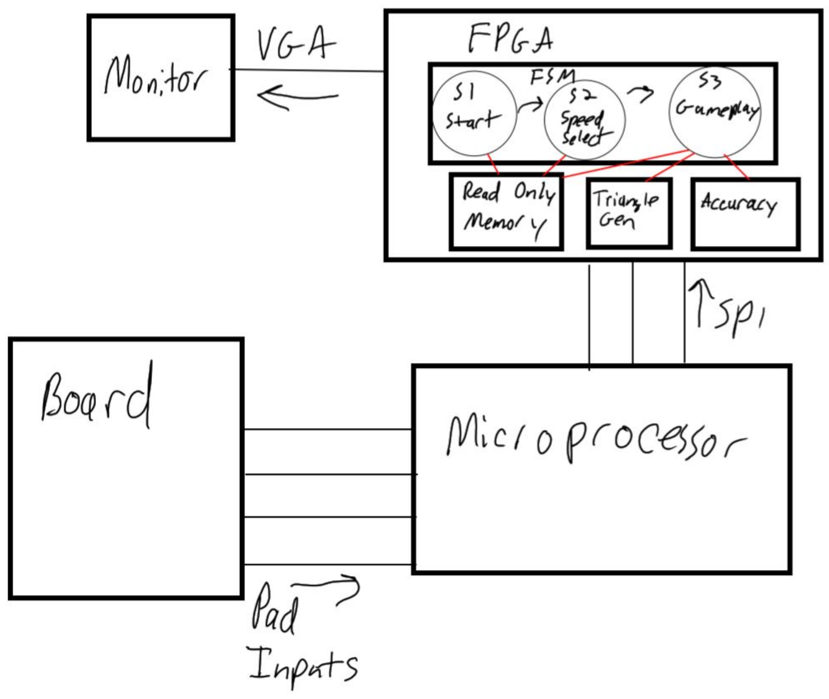
For our final project, we built a homebrew version of the arcade game Dance, Dance Revolution. We used the microcontroller on our provided MicroPs board to take in digital inputs from the DDR board we made, sent that data to our FPGA through SPI, and then used our FPGA do to the rest of the data processing and control a monitor display with VGA. Our user interface for the game includes a start screen, speed selection screen, and a live gameplay state that sends a sequence of triangles moving down the screen. The player's score increases whenever they correctly step on the board based on the lines of triangles. Our new hardware includes a monitor which we control through VGA and Velostat, the material we used to create a pressure sensitive board. Overall this project worked well, however, the Velostat proved trickier than expected since it's base resistance would vary, resulting in our board working on Demo Day and the night before checkoff, but only half working the day of checkoff.

Introduction

For our project, we decided to build a Dance, Dance Revolution (DDR) board and program a version of the game from scratch on the microcontroller and FPGA included on our microPs board. We chose this project because we found the material Velostat on Adafruit as we were searching for inspiration for our project and wanted to learn more about its capabilities and tendencies. We knew from our research that whenever pressure was applied, the Velostat increased in resistance. However, the resistance could change not just based on being stepped on perpendicularly to the sheet, but also due to the material flexing, or getting stretched. We knew going in that the material would be tough to work with, and that we'd need to make each pad as identical as possible, but we thought we could make it work. We found a few references online that gave us ideas on how to set up our board, and started planning our project around that.

While we had never worked with Velostat before, it was conceptually very easy to implement, so we wanted to include another piece of hardware that we had never worked with before to satisfy the requirement for new hardware. We chose using VGA to connect our FPGA to a monitor so we could create a display for our game. After going over the basics in class, and running into some issues with getting the screen to turn on, due to a misunderstanding of porches, we managed to first display scrolling triangles going down the screen, and worked our way up to state-based screens with the score displayed, speed options, and a scrolling changing sequence of triangles.

The block diagram (Appendix A1) shows a high level view of our final set up and what each component of our project does. This includes our Velostat pad inputs connected to comparator circuits that would output high whenever they were stepped on. This allowed us to use the microcontroller's digital input pins, debounce the data, and then transmit that data through SPI to the FPGA, where it was processed and used to display the game on the monitor through VGA.

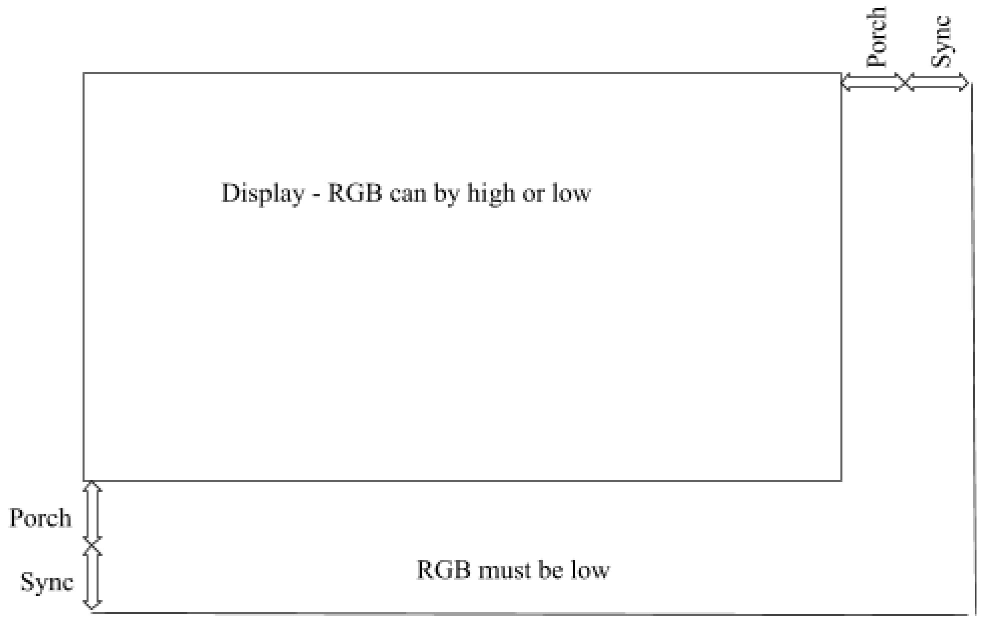


At the end of this project, our VGA output looked beautiful. Unfortunately, the Velostat proved trickier than expected, so only 2 of our 4 pads worked during checkoff, due to us not having time to adjust the resistors in our comparator circuit in accordance to how the resistance of the board changed based on it flexing over time due to a board that was not completely flat. However, during demo day, we did get all 4 pads working for over an hour of gameplay. As the demonstrations progressed, the resistance of our pads changed, as they became more compressed and took longer to return to their original state as more people tested our board and code.

New Hardware

We chose a monitor and Velostat as the new hardware we chose for this project. Velostat is a conductive plastic that increases in resistance when pressure is applied to it. When sandwiched between copper tape we can create a variable resistor that can be inserted into any circuit. Stepping on the sandwich raises the resistance of the Velostat, and thus the voltage drop across it increases when set up in a voltage divider. We had debated between using a comparator circuit to read the output digitally or ADC, but ultimately decided on the comparator circuit (Appendix A2.1) to get the mat up and running quickly without having to wade through more complicated code. In the end, this ended up being our downfall. One night, it was working smoothly: we could read the input from each pad consistently with almost no false positives or missing any steps. Yet the next day, we could only get two of the pads working, with many false positives and some of the steps never recording. The first problem we had was realizing how much our different pads would vary in resistance. Each pad had a difference resistance, since there weren't sized exactly the same, and the copper tape we stuck to both sides had air bubbles and didn't always line up with each other from one side to the other, decreasing the surface area we would be passing voltage through. However, we could account for this by using different resistors in each of our comparator circuits that corresponded with the base resistance of each Velostat mat. When initially researching Velostat, we found someone who had created DDR boards with Velostat, so we assumed it would be more consistent than ours actually was[2]. The main problem we had with our Velostat is that the board we attached them to was not flat (Appendix A2.2). When the Velostat is not flat, its resistance is higher, which means it has a higher voltage drop across it. When stepping on the curved board, it flattened the Velostat. So, while the resistance of the Velostat decreased when it flattened, it also increased because we were increasing the pressure on it with our body weight. With these two opposite forces acting in tandem, the output was not consistent. We attached the pads to plywood to keep them on a solid surface, yet the wood was curved. We remedied this by securing sturdier pieces of wood around the edges, which did help a lot, but it still bowed in the middle. After shifting the board around and it warping a little overnight, we could not get the consistent play we had been getting earlier.

The VGA display was more successful. Pins driven by FPGA logic controlled the LEDs on the monitor. We had digital control of RGB, giving us 8 colors. The screen gets drawn pixel by pixel, assigning each pixel with the current RGB value. We settled on a 60 Hz refresh rate for the screen, as the code was fairly simple and we wouldn't see any flickering. The main struggle in getting the screen to display anything was that at certain times, RGB must be 0. This happens at the porches and during hsync and vsync, a frame of "pixels" outside the screen that allow for the transition from one line of pixels to another.



Several sources online had said that most monitors would overwrite the signal during these times, driving everything needed low, however that was not the case for us. We tried with three different monitors (one of them being a different brand), but we could not get the screen to display anything. We did finally figure out that RGB must be low, we finally got our display working.

Schematics

Our schematic is comprised of 4 main parts: the DDR board electronics, the VGA cable and monitor, the FPGA, and the microcontroller as shown in Appendix A3. Our inputs from our DDR board go to to the microcontroller and get debounced, then sent through SPI to the FPGA where the signals get analyzed and control the shapes and characters to be displayed on the monitor through VGA.

Our DDR board used each Velostat pad as a variable resistor to set up a voltage divider that output a voltage that varied around 1.6 V, about half of our 3.3 V output on our microPs board, depending on how much pressure was applied. We used an operation amplifier from the MCP6004 chip to set up a comparator circuit with another voltage divider that we had set at exactly half of the 3.3 volt input using two 20 kOhm resistors. This comparator circuit output high when the voltage output from the pad's voltage divider breached the threshold of the constant voltage divider, so we could get a 3.3 V output (digital high) when the mat was stepped on and 0 V (digital low) otherwise. Our second resistors in the voltage dividers that included each pad had a variety of resistances, due to how the construction of the Velostat pads was not identical each time and each voltage divider needed a resistor rated just higher than the base resistiance of the mat, but lower than the resistance of the mat when it is stepped on. The output of each operational amplifier directly connected to the digital inputs of the microcontroller.

Our SPI between the microcontroller and the FPGA consisted of 4 wires: MOSI, SPCK, done, and load. We did not include chip select or MISO because we weren't sending data to any other chips or sending data back to the microcontroller.

To connect to VGA, we had 3 voltage dividers to bring down the voltage for the R, G, and B lines to values less than or equal to 0.7 V, but had hsync and vsync directly connected to the boards outputs of 3.3 V. We knew the inputs for R, G, and B had to be below 0.7 V so a voltage divider comprised of a 300 Ohm and a 65 Ohm resistor allowed us to bring 3.3 volts down to 0.58 V. While perhaps we could have gotten closer to 0.7, we still had clear color and crisp graphics. The outputs from this were connected to a VGA breakout board, which allowed us to plug the cable in securely and without fear of wires becoming unattached.

Microcontroller Design

The microcontroller controls our state machine and sends the mat's outputs to the FPGA, as shown in Appendix A4, along with how they should be interpreted by the FPGA: start, speed selection, live gameplay. When reading inputs during the live play phase, it also handles debouncing.

Upon start up, we send a string of zeroes through SPI to reset the game into state one. After that, we wait in state one until the player hits any of the pads. Once that input has been registered, we send through the SPI that we have moved to state two, speed selection. We wait in this stage until the player steps on the left or right arrow, which chooses slow or fast speed respectively, and then send this information through SPI along with moving it into state three. State three is the live play phase of the game. In this state, we continually read from our input pins until at least one of them is high (signaling a pad was stepped on). For our debouncer, we keep reading from the input pins for a brief period, which enables us to catch if a player hit two pads in quick succession since we do not expect humans to be able to hit them simultaneously. Then we send the data through SPI, wait 1200 milliseconds, and then send data through the SPI as if all inputs are low. This way even extremely short steps can be seen by the player and be properly interpreted as an input to our accuracy module on the FPGA.

The more complicated part of this code is formatting the data to send through the SPI. We have a function that returns 4'b0100 when the input to a specified pin is high, and 4'b0000 if the input is low. So when a pad is stepped on, it returns 4. This enabled us to simple add the state on top of it (2'b01, 2'b10, or 2'b11) without having any of the data overflow or get overwritten. When we needed to differentiate the pad outputs from each other (during speed selection and live play), we multiply the returned value (4 or 0) by multiples of 2 (i.e. 4*2 for the down arrow, 4*4 for the up arrow, etc.) and summed these. So if a player were to hit the **down** (4*2) and **left** (4*8) arrows during the live play state (3), the data sent over SPI would be **00101011**.

In our initial design, we had wanted to also create random play sequences and send those through SPI to store in RAM on the FPGA. However, due to time constraints and difficulty coding up the RAM on the FPGA, we decided to move the storing of play sequences to the FPGA only since we already had code to load in other bits through text files.

FPGA design

The FPGA as a whole is controlling the VGA display, allowing the user to see the effects of their inputs on the screen and interact with the game we created, as shown in Appendix A5. Our top level module, VGA, instantiated the modules "vga_pll", "vgacontroller", "videoGen", "vgaState", and "aes_spi". The module vga_pll was a wizard generated module that used a Phase Locked Loop (PLL) to create a 25.175 MHz VGA pixel clock. The "vgacontroller" module was a small module based on provided code [1] that created counters for the vertical and horizontal positions and specified when outputs should be forced to low when we entered the porches outside the legal display area. Originally, we had problems with this code, since we left our outputs high during the porches, which resulted in a solely black screen. We could tell from our oscilloscope that the expected voltages and waveforms were being provided to the r,g,b, hsync, and vsync VGA inputs, but could see nothing on the screen. While we ended up fixing the problem, it drove home how important the porches were.

The “videoGen” module was a large module that controlled everything on the screen. We instantiated our “accuracy” modules within this module, wrote a flip flop that allowed us to cycle through a list of dance moves for the user to complete and instantiated “seqgenrom” modules to pull those lists of tasks from rom. For debugging purposes, we created 9 triangles, one for each bit of our SPI so we could see the data we received through SPI in real time, and one triangle to show us when “load”, a signal that told us when bits were being transmitted, was high or low. This was very useful so we didn’t need to use the logic analyzer on the oscilloscope to see if our SPI was working correctly. Our VGA state module takes the last two bits of SPI data, and uses that to change the state of our FPGA. We get the start input data in state one (Appendix A2.3), speed settings in state two (Appendix A2.4), and live data and in state three, along with calculating accuracy and score. In state three, we generated moving triangles, using the “moveTriangle” and “moveBlocks” module. Only while in state three, triangles of the current line in the sequence would scroll down the screen through the accuracy box (Appendix A2.5). Upon reaching the bottom of the box, they would reset at the top of the screen, but which ones were on would change based on the next line in the sequence. The moving triangles and the ones that appeared when our input pads were pressed, used the “triangleGen” module, a module loosely based on “rectangleGen” from the provided code. The “moveBlocks” module controls the speed of the scrolling blocks. This speed is used in the “moveTriangle” module to update the location of the triangles created with “triangleGen”.

One of the other key elements in the top “VGA” module was “aes_spi”. Our code to transmit our inputs from the microcontroller to the FPGA closely resembled that from the aes lab, so we copied and edited it a large chunk of it. Accuracy was calculated using our “accuracy” module, which takes in the stomp inputs and compares them to the line of the sequences within the target rectangle. If they match exactly, the player gets a point, and this is turned into decimal numbers showing the player their score. This module also instantiated several instances of “chargenrom” a module that stored and pulled characters, including the numbers, from the ROM to display words in our “start”, “speed”, and “score” screens. Because we could easily avoid not using all of our FPGA to provide instructions to the user, we didn’t work on optimizing this, believing spending our time on other aspects of our project was more effective. Both our sequence of the falling triangles and the characters are stored in ROM, either “seqgenrom” which reads the play sequence from a text file and outputs the current line of triangles , or “chargenrom” which reads several lines from the ROM to display a specific character, indexing in based on the small alphabet we wrote.

Finally, our code wrapped up with our “accuracy” module. This module was very small and simple, but required a lot of thinking as we wanted to add one point to our score, but only if the correct pads were hit, and once hit, the player could score no more points until the next line of the sequence. When our game was restarted, points were reset to zero. Then, when the pads were hit while the triangles were within our scoring zone rectangle, we had a 4-bit “flag” that recorded the current line of the sequence so the player could not attempt to score more points on the same triangles. If the current stomp was the same as the sequence, we added one point. If any of the triangles were wrong, the player got no points. The score was automatically updated onto the monitor screen so the player could see if they got it right immediately. At the end of this all, the RBG outputs to the VGA were set to high if the current pixel of the display required that color (i.e. blue would be high if inside the blue, white, or teal triangles).

Results

On the final checkoff, our game was not working as consistently as we had hoped. The VGA graphics ran smoothly and looked clean, exactly how we wanted it to. The DDR board on the other hand, had some difficulties. The night before, it was working well and we were reading the inputs consistently. The next day, we were trying to swap out resistors so our comparator circuits worked with the Velostat, however since the stockroom would not let us in, we had limited access to resistors and limited time to test them. One of the pads worked great (consistent every time we stepped on it), another one decently (most of the time it registered the step), but the other two were all over the place, not displaying when stepped on sometimes and false positives other times. On Demo Day, the board worked perfectly for over an hour, only becoming inconsistent when almost one hundred points had been recorded and the Velostat had been compressed so many times in a short period of time it was not responding as it should be.

One of the more challenging parts of the project was interpreting SPI correctly. When we initially set up the SPI link, we could see that we were getting the data we expected, yet we were changing states unexpectedly. When we were accessing the data, we were doing so as the bits were shifted in one at a time, instead of only after all 8 had come through. Then we were accessing the data using the wrong clocks on our flip flops, which shifted our bits off by one (but our SPI debugged still displayed what we expected it to). Since our microcontroller was controlling the state through SPI, we had to make sure that we were reading the correct bits at the correct time. Once this was solved, it led us to another challenging problem: computing the score. We wanted the player to get a single point when they hit the pads to match what they saw on the screen within the accuracy rectangle. We needed to interpret the SPI data against the current sequence, and add only one point for each line of the sequence. We had initially set it up to only record the score when the input rose from zero to high (so any of the inputs were stepped on), however we soon realized that wouldn't work if the player hit two pads at different times, or hit the pad a little early but then correct (as it would stay high). So we modified our code to have a flag that would be set when the score updated for that sequence. This way it could only add one to the score for each part of the sequence. So we would check to see if the next part of the sequence was within the hitting zone, then update the score if the player stepped on the correct pads. We had to instantiate the accuracy module twice, since it had to be linked to each of the two sequences displayed on the screen, and we took the points output from these and added them together to get the player's total score.

We had initially wanted a speaker to play music in time with the game, however we quickly realized we probably wouldn't have time to set up a speaker after spending so long on trying to get the VGA display to turn on and abandoned the speaker part of the project. We had also planned on having the microcontroller store the play sequences, and randomize it for each play through, but since we had difficulties implementing RAM on the FPGA and realized that formatting and sending that data over SPI would probably use more of our time than we had, we moved to storing the sequence in the FPGA ROM with a text file.

References

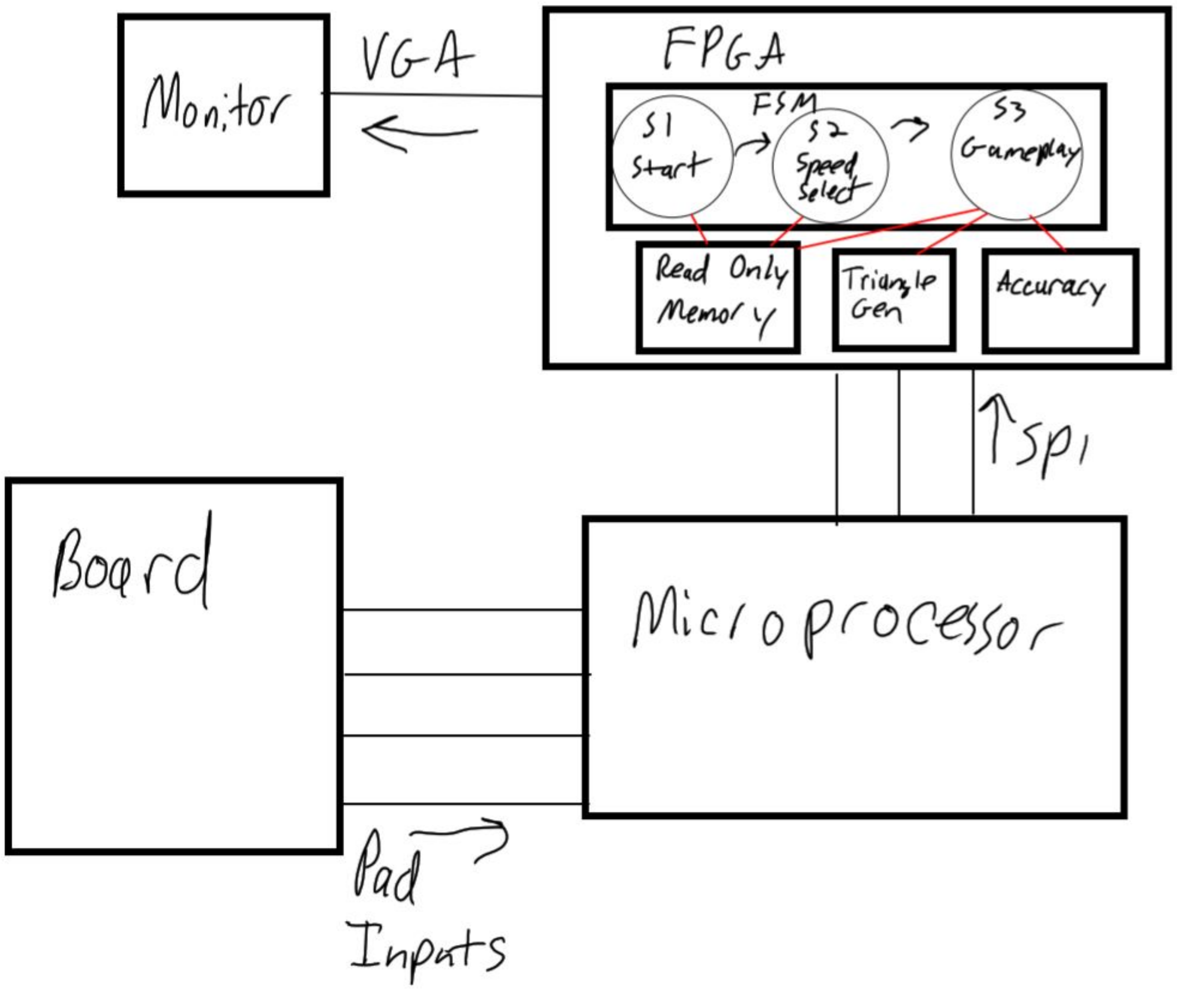
[1] S. Harris and D. Harris, *Digital Design and Computer Architecture: Arm edition*, Elsevier Science, 2015.
[2] Promit, "DanceForce V3 DIY dance pad for DDR" *Ventspace,* wordpress.com, April 2019.
https://ventspace.wordpress.com/2018/04/09/danceforce-v3-diy-dance-pad-controller/

Parts list

Part	**Source**	**Part Number**	**Cost**
Velostat (9 sheets)	**tinkersphere.com**	**TS1454**	**40.90**
Copper Tape (20 sheets)	**Amazon.com**	**B075TSX7F2**	**35.02**
VGA Cable & Breakout Board	**Borrowed**		
Monitor	**Borrowed**		
MCP6004 OpAmp	**HMC Stockroom**		
3ft by 3ft plywood	**HMC Machine Shop**		
Crafting supplies	**HMC Makerspace**		
		Reimbursement	**-50.00**
		Total	**25.92**

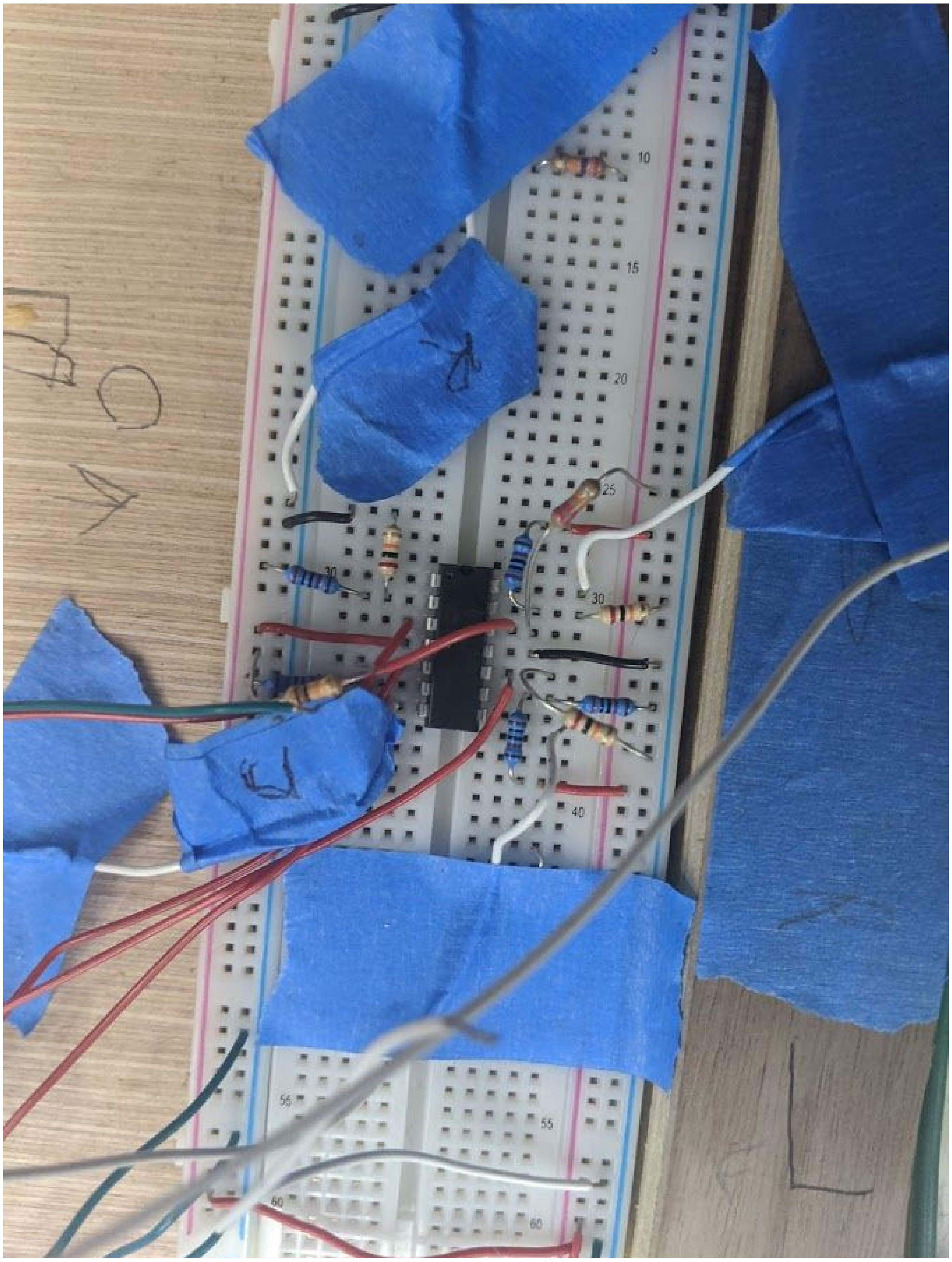
Appendices

A1: Block Diagram

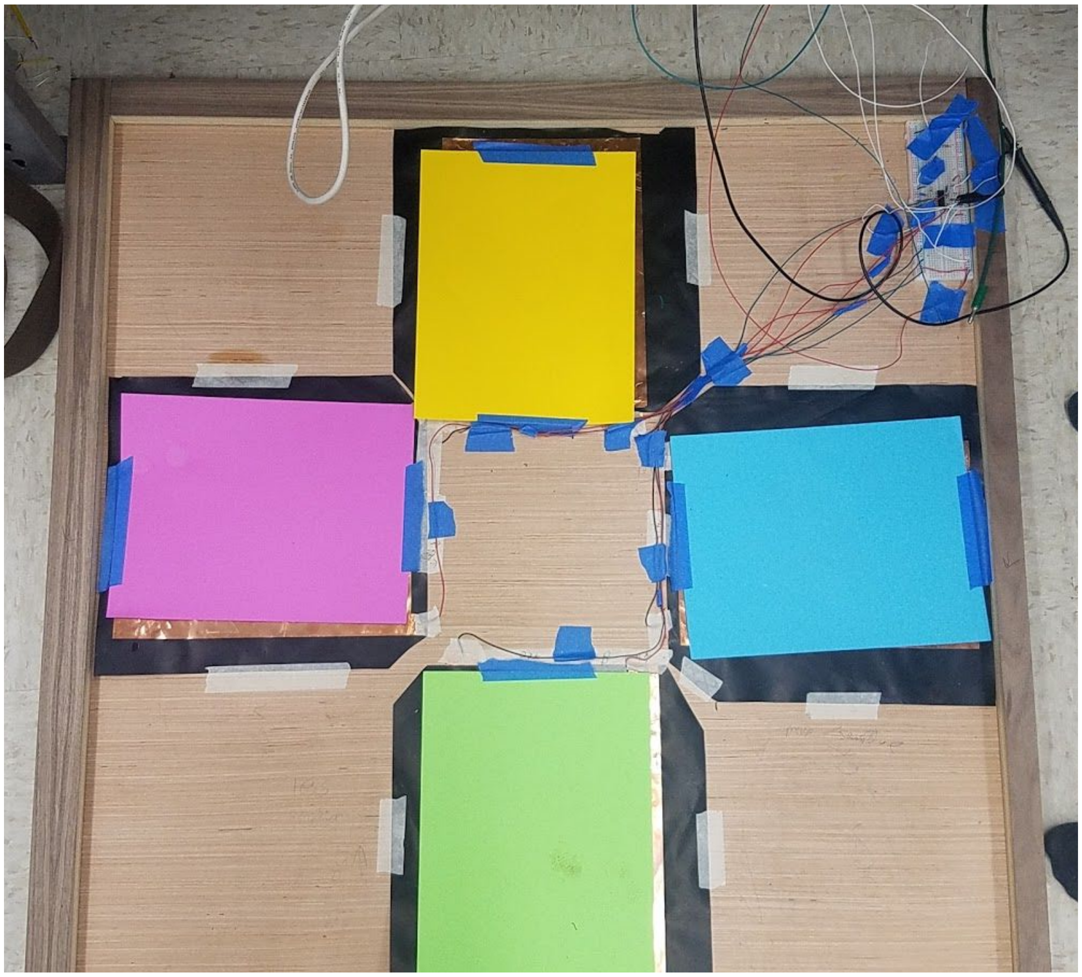


A2: Pictures

A2.1: 4 OpAmp comparator circuits

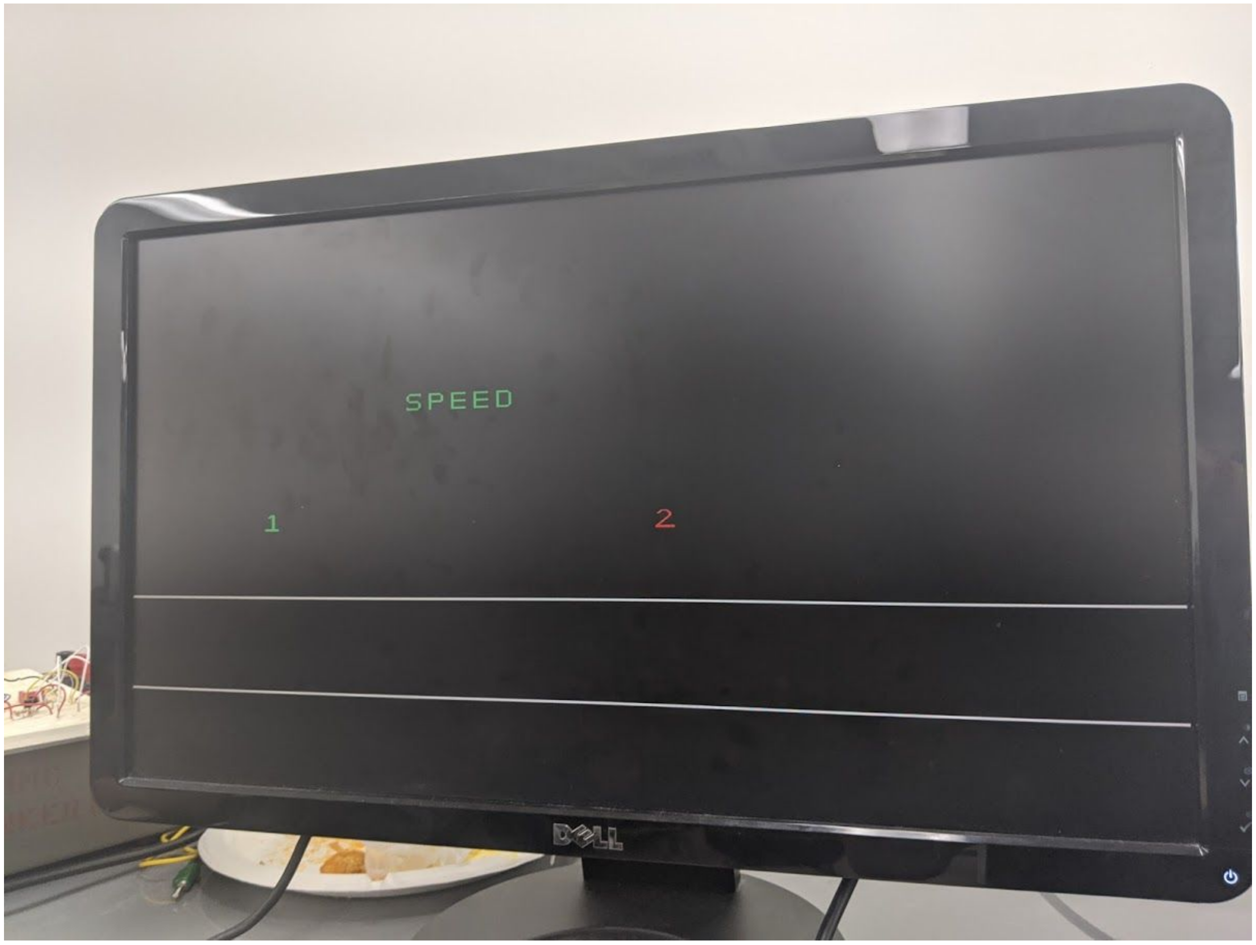


A2.2: DDR Board

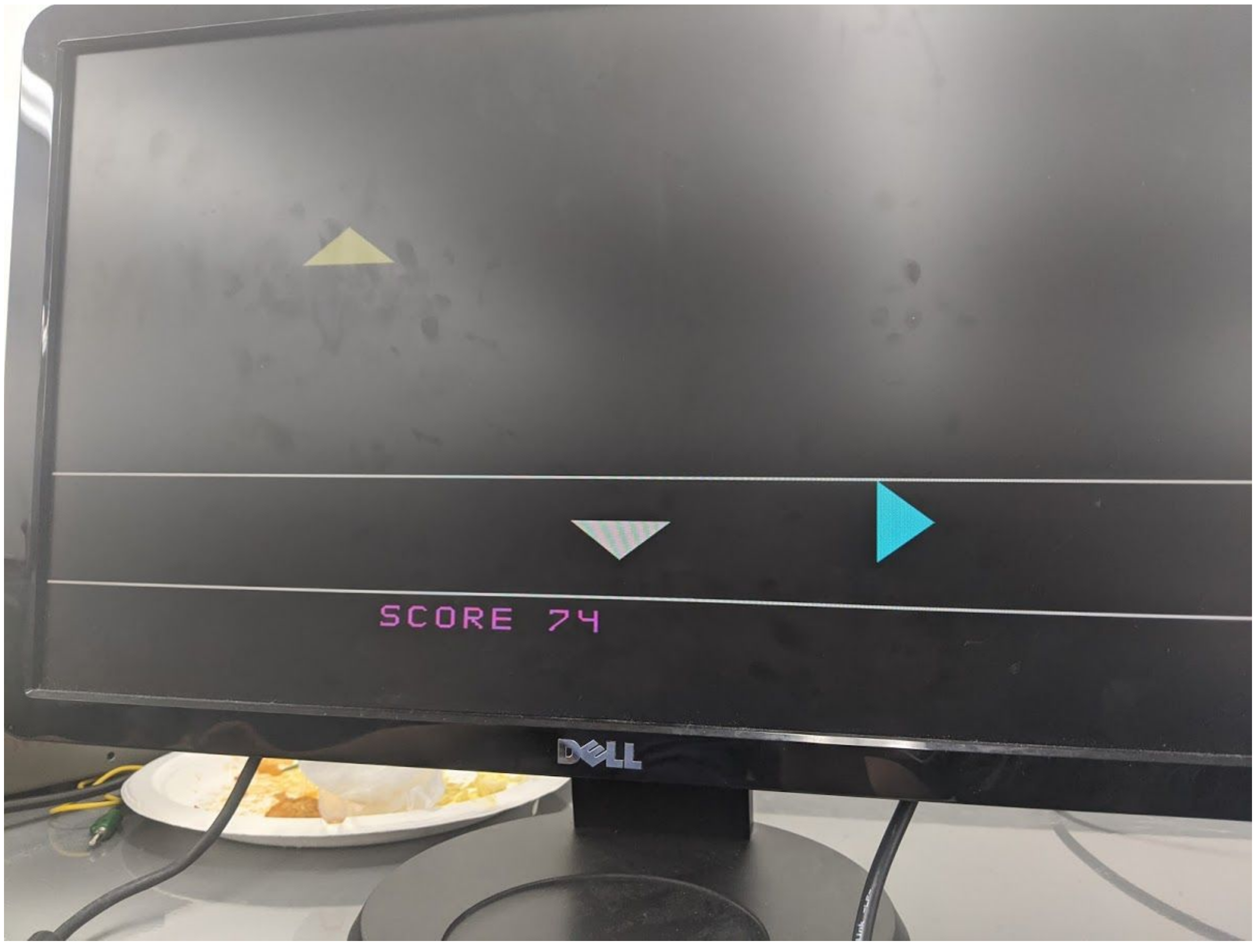


A2.3: Start Screen

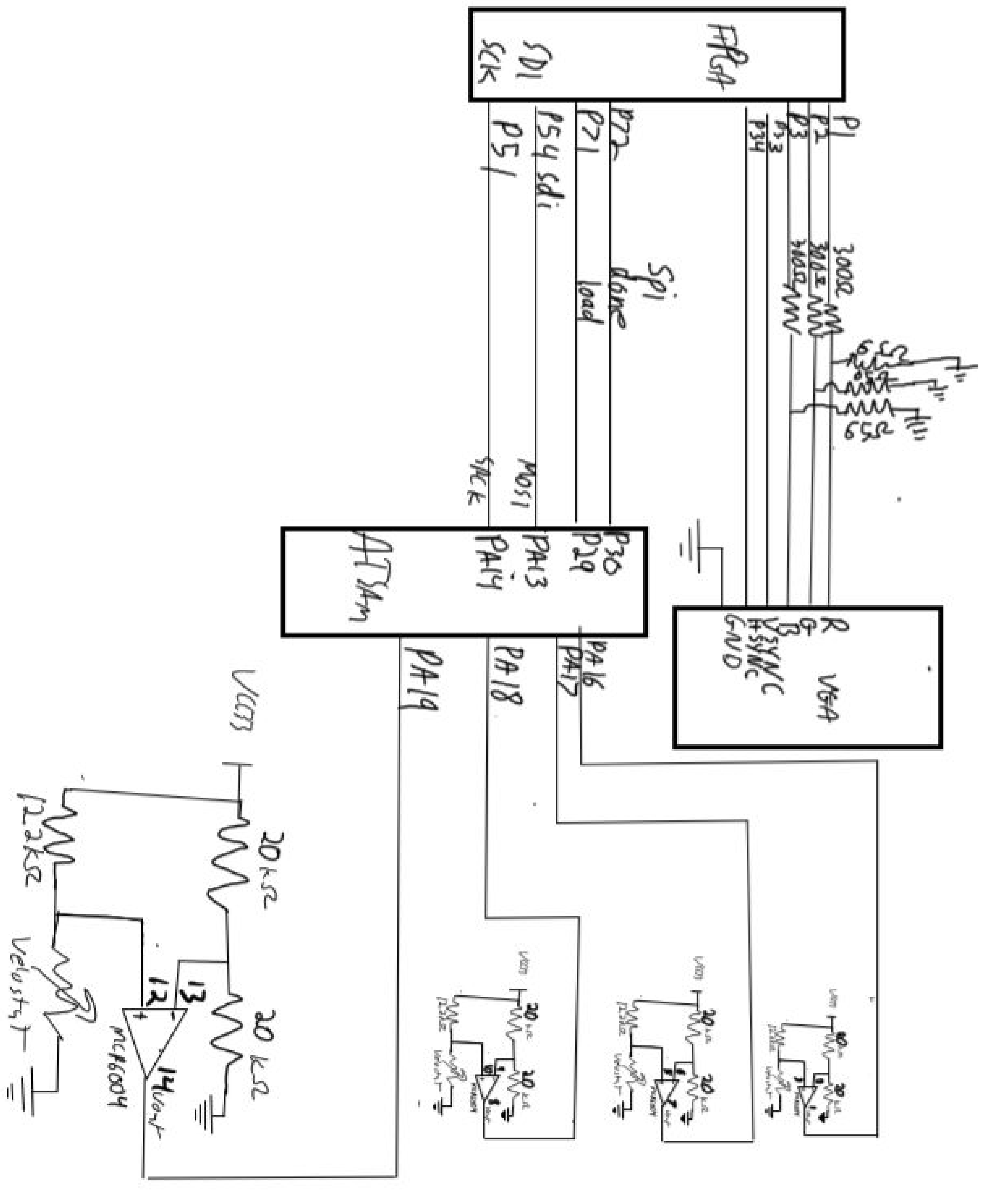
A2.4: Speed choice screen



A2.5: Display showing pressed button and 2 lines of sequence falling



A3:Schematic



A4: C code for microcontroller

```
// stomp.c
// Stephanie Clifner and Elizabeth Hedenberg
// sclifner@g.hmc.edu ehedenberg@g.hmc.edu
// 12/11/2019
// State machine, and sends inputs over SPI

////////////////////////////////////////////////
// #includes
////////////////////////////////////////////////

#include <stdio.h>
#include "SAM4S4B.h"

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

#define LOAD_PIN    29
#define DONE_PIN    30


# define STOMP_LEFT_PIN      16
# define STOMP_UP_PIN 17
# define STOMP_DOWN_PIN      18
# define STOMP_RIGHT_PIN 19



////////////////////////////////////////////////
// Function Prototypes
////////////////////////////////////////////////


int getStomp(int);
void getSpeed(void);
void showStomp(void);

////////////////////////////////////////////////
// Main
////////////////////////////////////////////////

int main(void) {

  samInit();
  pioInit();
  spiInit(MCK_FREQ/244000, 0, 1);
  // "clock divide" = master clock frequency / desired baud rate
  // the phase for the SPI clock is 1 and the polarity is 0

  // Load and done pins
  pioPinMode(LOAD_PIN, PIO_OUTPUT);
  pioPinMode(DONE_PIN, PIO_INPUT);


       // Stomp pins
       pioPinMode(STOMP_LEFT_PIN, PIO_INPUT);
```

```
        pioPinMode(STOMP_UP_PIN, PIO_INPUT);
        pioPinMode(STOMP_DOWN_PIN, PIO_INPUT);
        pioPinMode(STOMP_RIGHT_PIN, PIO_INPUT);

        pioDigitalWrite(LOAD_PIN, 1);
                spiSendReceive((char)0); // Hardcoded, we're sending 00000000 so spi has known
values at start (and functions as a reset)
        pioDigitalWrite(LOAD_PIN, 0);


        tcDelayInit();
        getSpeed();
        // delay a bit
        // did not use tc delay functions here because it was breaking the code
        for(int i = 0; i < 12800000; i++);

        while(1){
        showStomp();
        }

}

////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////

int getStomp(int pin) { // TODO: make stompUp
        if(pioDigitalRead(pin)) {
                if(pioDigitalRead(pin))
                        return 4; // so we can send {2'b10, state} over SPI (shift input over to
leave 2 bits for the state)
                else
                        return 0;
        }
        else
                return 0;
}

void getSpeed() {
        char speedSelected = 0;
        while(!speedSelected) {
                // left is slower speed (10), right is higher speed (11)
                speedSelected = (getStomp(STOMP_LEFT_PIN)*2) | (getStomp(STOMP_RIGHT_PIN)*2 +
getStomp(STOMP_RIGHT_PIN)); // Add all inputs from stomps (i.e. if stop up, speedSelected = 1
+0 +0 +0. If stomp down, then speedSelected = 0 + 2 + 0 + 0. ORing them together would
probably be okay
        }

                // send speed over SPI, with 2'b10 state at the end
                char speed = (char)((speedSelected - 1) + 2);
        pioDigitalWrite(LOAD_PIN, 1);
                spiSendReceive(speed); // in state one, send speed - TODO: hardcoded to 00
right now, change to input later. Hardcoded, we're sending 00001000
        pioDigitalWrite(LOAD_PIN, 0);
}
```

```
void showStomp() {
       int currentStomp = 0;
       int left = 0;
       int up = 0;
       int down = 0;
       int right = 0;
       while(!currentStomp) {
              // bit shifting so each pad corresponds to a unique bit
                     left = getStomp(STOMP_LEFT_PIN)*8;
                     up = getStomp(STOMP_UP_PIN) * 4;
                     down = getStomp(STOMP_DOWN_PIN)*2;
                     right = getStomp(STOMP_RIGHT_PIN);
                     currentStomp = left | up | down | right;
       }
       // once one goes high, wait a little bit to see if any other goes high
       for(int i = 0; i < 32000; i++) {
                     left = left | getStomp(STOMP_LEFT_PIN)*8;
                     up = up | getStomp(STOMP_UP_PIN) * 4;
                     down = down | getStomp(STOMP_DOWN_PIN)*2;
                     right = right | getStomp(STOMP_RIGHT_PIN);
                     currentStomp = left | up | down | right;
              }
       // send the data, with state 2'b11 at end
       char stomped = (char)(currentStomp + 3);
       pioDigitalWrite(LOAD_PIN, 1);
              spiSendReceive(stomped);
       pioDigitalWrite(LOAD_PIN, 0);

              // wait a little bit, then send low (artificial debouncing)
       tcDelayMillis(1200);
       pioDigitalWrite(LOAD_PIN, 1);
              spiSendReceive((char)3);
       pioDigitalWrite(LOAD_PIN, 0);

}
```

A5: FPGA System verilog code

```
// VGA.sv
// Stephanie Clifner and Elizabeth Hedenberg
// sclifner@g.hmc.edu ehedenberg@g.hmc.edu
// 12/11/2019
// FPGA hardware for DDR MicroPs final project, Fall 2019.
module VGA(input logic clk, sck, sdi, done, load, output logic r,g,b,hsync,vsync);
	logic [9:0] x, y;
	logic sdo, writeSeq, leftStep, rightStep, upStep, downStep;
	logic [1:0] state;
	logic [7:0] spiData, speed;
	logic vga_clk, blank;
	logic reset;

	assign reset = spiData == 8'b0;

	vga_pll vgapll(.inclk0(clk), .c0(vga_clk));
	vgacontroller vgac(vga_clk, hsync, vsync, blank, x, y);
	videoGen vg(clk, reset, x,y, hsync, vsync, blank, leftStep, rightStep, upStep,
downStep, load, speed, spiData, state, r,g,b);

	vgaState vstate(clk, reset, spiData, load, state);

	aes_spi spi(sck, sdi, sdo, done, spiData);

	always_ff @(posedge clk)
		if(state == 2'b00)
			begin
			speed[3:2] = 2'b01;
			end
		else if(state == 2'b10)
		begin
			speed = spiData[7:0];
		end
		else if (state == 2'b11)
			{leftStep, upStep, downStep, rightStep} = spiData[5:2];
endmodule

module videoGen (input logic clk, reset, input logic [9:0] x, y, input logic hsync, vsync,
blank,
			leftStep, rightStep, upStep, downStep, load,
			input logic[7:0] speed, test, // test is the spiData
			input logic [1:0] state,
			output logic r,g,b);
	logic pixel, pixel1, pixel2, pixel3, pixel4, pixelHollow, slowclk, pixelStart,
pixelSpeed;
	logic pixelLeftStep, pixelUpStep, pixelDownStep, pixelRightStep, pixelStep, spi;
	logic [3:0] nextStep1, pixelSeq1, nextStep2, pixelSeq2;
	logic [9:0] top1, top2;
	logic [5:0] points1, points2;

	//controls score
	accuracy stmpstmpbb1(clk, reset, load, top1, top1+50, 370, 450, {leftStep, upStep,
downStep, rightStep}, pixelSeq1, nextStep1, points1);
	accuracy stmpstmpbb2(clk, reset, load, top2, top2+50, 370, 450, {leftStep, upStep,
downStep, rightStep}, pixelSeq2, nextStep2, points2);

	// controls which step is currently displayed
	always_ff @(posedge slowclk)
		if(reset)
			begin
				nextStep1 = 0;
				nextStep2 = 1;
			end
		else if(top1 == 201) // when half way down the screen, update the other
sequence so it is now the next step in the sequence when it's loooped around to the top of the
screen
			nextStep2 = nextStep2 + 2;
		else if (top2 == 201)
			nextStep1 = nextStep1 + 2;

		//which arrows should be displayed for current step in sequence
	seqgenrom(nextStep1, pixelSeq1);
	seqgenrom(nextStep2, pixelSeq2);


	chargenrom Start(8'd73, x, y, 200, 200, pStart);
	chargenrom sTart(8'd74, x, y, 200, 215, psTart);
	chargenrom stArt(8'd65, x, y, 200, 230, pstArt);
	chargenrom staRt(8'd72, x, y, 200, 245, pstaRt);
	chargenrom starT(8'd74, x, y, 200, 260, pstarT);
	chargenrom Q(8'd86, x, y, 200, 275, pq0);
	// only on in state 1
	assign pixelStart = (pStart | psTart | pstArt |pstaRt | pstarT| pq0) & (state ==
2'b01);

	chargenrom sPeed(8'd71, x, y, 200, 215, pp0);
	chargenrom spEed(8'd68, x, y, 200, 230, pe0);
	chargenrom speEd(8'd68, x, y, 200, 245, pe1);
	chargenrom speeD(8'd67, x, y, 200, 260, pd0);
	//only on in state 2
	assign pixelSpeed = (pStart | pp0 | pe0 |pe1 | pd0) & (state == 2'b10);

	chargenrom Easy(8'd77, x, y, 300, 100, pEasy);
	chargenrom harD(8'd78, x, y, 300, 360, pharD);


	chargenrom S2(8'd73, x, y, 460, 200, ps2);
	chargenrom C0(8'd66, x, y, 460, 215, pc0);
	chargenrom O0(8'd70, x, y, 460, 230, po0);
	chargenrom R1(8'd72, x, y, 460, 245, pr1);
	chargenrom E2(8'd68, x, y, 460, 260, pe2);
	chargenrom twho(8'd76 + (points1 + points2)/10, x, y, 460, 290, digit1);
	chargenrom tree(8'd76 +(points1 + points2)%10, x, y, 460, 305, digit2);
	//only on in state 3
	assign pixelScore = (ps2 | pc0 | po0 |pr1 | pe2|digit1|digit2) & (state == 2'b11);

	// move the sequence down the screen
	moveblocks mb(vsync, speed[3:2], slowclk);

	//seq 1
	moveTriangle mt1(reset, slowclk, x, y, 150, 200, 2'b00, state, 1, pixel2, topp); // up
	moveTriangle mt2(reset, slowclk, x, y, 300, 350, 2'b01, state, 1, pixel3, top1); //
down
	moveTriangle mt3(reset, slowclk, x, y, 50, 100, 2'b10, state, 1, pixel1, toppp); //
left
	moveTriangle mt4(reset, slowclk, x, y, 450, 500, 2'b11, state, 1, pixel4, topppp); //
right
	//seq 2
	moveTriangle mt5(reset, slowclk, x, y, 150, 200, 2'b00, state, 0, pixel6, ttopp); // up
	moveTriangle mt6(reset, slowclk, x, y, 300, 350, 2'b01, state, 0, pixel7, top2); //
down
	moveTriangle mt7(reset, slowclk, x, y, 50, 100, 2'b10, state, 0, pixel5, tttoppp); //
left
	moveTriangle mt8(reset, slowclk, x, y, 450, 500, 2'b11, state, 0, pixel8, ttttopppp);
// right

	// triangles that are on when mat is stomped
	triangleGen tgLeft(x,y,50, 400, 450, 50, 100, 2'b10, pixelLeftStep); // input are x, y,
height, top, bottom, left, right
```

```
        triangleGen tgUp(x,y,50, 400, 450, 150, 200, 2'b00, pixelUpStep);
        triangleGen tgDown(x,y,50, 400, 450, 300, 350, 2'b01, pixelDownStep);
        triangleGen tgRight(x,y,50, 400, 450, 450, 500, 2'b11, pixelRightStep);

        //two lines of where arrows need to be within when hit to get a point
        hollowBox hb(x, y, 370, 450, pixelHollow);

        assign pixelStep = (leftStep & pixelLeftStep) | (upStep & pixelUpStep) | (downStep &
pixelDownStep) | (rightStep & pixelRightStep);


        // turn arrow on if it should be on for current step
        assign pixelLeftArr = (pixel1& pixelSeq1[3]) | (pixel5 & pixelSeq2[3]);
        assign pixelUpArr = (pixel2 & pixelSeq1[2]) | (pixel6 & pixelSeq2[2]);
        assign pixelDownArr = (pixel3 & pixelSeq1[1]) | (pixel7 & pixelSeq2[1]);
        assign pixelRightArr = (pixel4 & pixelSeq1[0]) | (pixel8 & pixelSeq2[0]);

        // which color each thing should be
        assign r = ((pharD & state ==2'b10) | pixelStart | pixelHollow | pixelStep | pixelUpArr
| pixelScore | pixelLeftArr) & ~blank;
        assign g = ((pEasy & state ==2'b10) | pixelSpeed  | pixelHollow | pixelStep |
pixelUpArr | pixelRightArr | pixelDownArr) & ~blank;
        assign b = (pixelHollow | pixelStep | pixelRightArr | pixelLeftArr|pixelScore) &
~blank;
endmodule

module vgacontroller(input logic vga_clk, output logic hsync, vsync, blank, output logic [9:0]
x,y);
        always_ff @(posedge vga_clk)
        begin
                x++;
                if(x==800)
                begin
                        x=0;
                        y++;
                end
                if(y==525)
                        y = 0;
                hsync = ~(x>=640+16 & x<640+16+96);
                vsync = ~(y>=480 +11 & y<480+11+2);
                blank = x>=640|y>=480;
        end

endmodule

module triangleGen(input logic [9:0] x, y, input logic [5:0] height,
        input logic [9:0] top, bottom, left, right, input logic [1:0] direction, output logic
pixel);

                logic [9:0] row;

                always_comb
                case (direction)
                2'b00: // up
                begin
                        if(y >= top & y <= bottom)
                                row = y - top + 1'b1;
                        else
                                row = 0;

                        pixel = (row != 0) & (x >= left + (bottom - top - row) & x <= right -
(bottom - top - row));
                end
                2'b01: // down
                begin
                        if(y >= top & y <= bottom)
                        row = y - top + 1'b1;
                    else
                        row = 0;

                    pixel = (row != 0) & (x >= left + (row-1) & x <= right - (row-1));
                end
                2'b10: // left
                begin
                    if(x >= left & x <= right)
                        row = x - left + 1'b1;
                    else
                        row = 0;
                    pixel = (row != 0) & (y >= top + (right - left - row) & y <= bottom -
(right - left - row));
                end
                2'b11: // right
                begin
                    if(x >= left  & x <= right)
                        row = x - left + 1'b1;
                    else
                        row = 0;
                    pixel = (row != 0) & (y >= top + (row-1) & y <= bottom - (row-1));
                end
                default:
                begin
                    row=0;
                    pixel=0;
                end
                endcase
endmodule

module moveblocks(input logic vsync, input logic [1:0] speed, output logic slowclk);
         logic [12:0] counter;

         always_ff @(posedge vsync)
         begin
                counter=counter+1;
         end
         assign slowclk=counter[speed];
 endmodule

module moveTriangle(input logic reset, input logic slowclk, input logic [9:0] x, y, input
logic [9:0] leftedge, rightedge, input logic [1:0] direction, state,  input logic odd,
                                                    output logic pixel, output logic [9:0]
top);
        logic [9:0] bottom;

        always_ff @(posedge slowclk)
                if (reset)

                                if (odd) // start triangles staggered
                                begin
                                        top=0;
                                        bottom=50;
                                end
                                else
                                begin
                                        top=200;
                                        bottom=250;
                                end
                else if (bottom==450) // if reaches the bottom of the screen, put back to the
top
                begin
                        top=0;
                        bottom=50;
                end
```

```
               else if (state == 2'b11) // only move the triangles if in state 3
               begin
                      top = top + 1;
                      bottom = bottom + 1;
               end
               // create the triangle at the current location
        triangleGen tg(x,y, 50, top, bottom, leftedge, rightedge, direction,  pixel);
endmodule

module hollowBox(input logic [9:0] x,y, top, bottom, output logic pixel);
// two horizontal lines across the screen
        assign pixel = (y == top | y == bottom) | (y == top + 1 | y == bottom + 1);
endmodule

module aes_spi(input  logic sck,
               input  logic sdi,
               output logic sdo,
               input  logic done,
               output logic [7:0] spiData);

    logic         wasdone;

    // assert load
    // apply 8 sclks to shift in speed, starting with speed[0]
    // then deassert load, wait until done
    always_ff @(posedge sck)
       begin
        if (!wasdone)  {spiData} = {spiData[6:0], sdi};
        else           {spiData} = {spiData, sdi};
       end

    always_ff @(negedge sck) begin
        wasdone = done;

    end
endmodule

module vgaState(input logic clk, reset, input logic [7:0] spiData, input logic load, output
logic [1:0] state);
        logic [1:0] nextState;

        always_ff @(posedge clk, posedge reset)
        if(reset)
               state <= 2'b01;
        else
               state <= nextState;
        // when s1, gets reset
        // when s2, getting speed data
        // when s3, getting live data
        always_comb
        case(state)
               2'b01:
               begin
                      if(spiData[1:0] == 2'b10)
                             nextState = 2'b10;
                      else
                             nextState = 2'b01;
               end
               2'b10:
               begin
                      if(spiData[1:0] == 2'b11 & ~load)
                             nextState = 2'b11;
                      else
                             nextState = 2'b10;
               end
               2'b11:
```

```
		begin
			if(spiData[1:0] == 2'b01 & ~load)
				nextState = 2'b01;
			else
				nextState = 2'b11;
		end
		default
		begin
			nextState = 2'b01;
		end
		endcase
endmodule

module chargenrom(input logic [7:0] ch, input logic [9:0] x, y, top, left,
			output logic pixel);
	logic [11:0] charrom[351:0]; // character generator ROM // TODO: make smaller. 176?
	logic [15:0] line; // a line read from the ROM
	// Initialize ROM with characters from text file
	initial
		$readmemb("charromdouble.txt", charrom);
	// Index into ROM to find line of character
	assign line = (y >= top  & y < top + 16) ? charrom[(y - top) + {ch-65, 4'b0000}] : 
16'b0;
	// is entry 0
	// Reverse order of bits
	assign pixel = (x >= left & x < left + 16) & line[3'd15-(x - left - 4)];
endmodule

module seqgenrom(input logic [3:0] current, output logic [3:0] pixel);
	logic [3:0] charrom[399:0]; // character generator ROM // TODO: make smaller. 176?
	// Initialize ROM with characters from text file
	initial
		$readmemb("seq.txt", charrom);

	assign pixel = charrom[current];
endmodule

module accuracy(input logic clk, reset, load,//assuming only one stomp
				input logic [9:0] movetriangletop, movetrianglebottom, linetop, 
linebottom,
				input logic [3:0] currentStomp, currentSeq, currentRow,
				output logic [5:0] points);
	logic yeahstomp;
	logic [2:0] accuracy;
	logic [3:0] tallied;

	assign yeahstomp = currentStomp[0]|currentStomp[1]|currentStomp[2]|currentStomp[3]; // 
high if any pad has been stomped

	always_ff@(posedge clk)
		if (reset)
			points = 0;
		else if ((tallied != currentRow) & (movetrianglebottom<linebottom) 
&(movetriangletop>linetop) & !load & yeahstomp) // if stomping at correct time
			begin
				tallied <= currentRow;
				// add 1 if stomped on the correct pads, otherwise add 0
				accuracy =currentStomp == currentSeq;
				points =points+accuracy;
			end
endmodule
```