Gravity Game

MicroPs Final Project Report
December 13, 2019
E155

Francisco Muñoz and Nico Naar

Abstract

Mobile games like Gravity Guy and Ninjump have fallen by the wayside, and this project prototypes a potential replacement consisting of two capacitive touch sensors, two circuits to interpret the touch signals, a microcontroller, an FPGA, and an 8x16 LED matrix display. The user taps the touch sensors to bounce a player LED back and forth across the display to dodge falling obstacles. The microcontroller detects if the sensors are touched, generates game data using a random selection from a set of obstacle groups, and speeds up the game as the player progresses. The level data is sent over SPI to the FPGA, which uses time-multiplexing to drive the 128 pixels on the display.

Introduction

This Gravity Game project aims to emulate popular mobile games like Gravity Guy and Ninjump by recreating their gameplay and offering a new way to experience the game. The game is played on an 8x16 LED dot matrix [1], with the player LED dodging falling obstacles using two capacitive touch sensors to control its position. The microcontroller, an Atmel ATSAM4S4B [2], reads the capacitive sensor data, generates the game data, and sends this information to the FPGA, a Cyclone IV EP4CE6E22C8N [3]. The FPGA drives the LED display using this information and sends back the player location to the microcontroller to determine if the player has collided with an obstacle. Figure 1, the system block diagram, depicts all the modules and their connections.

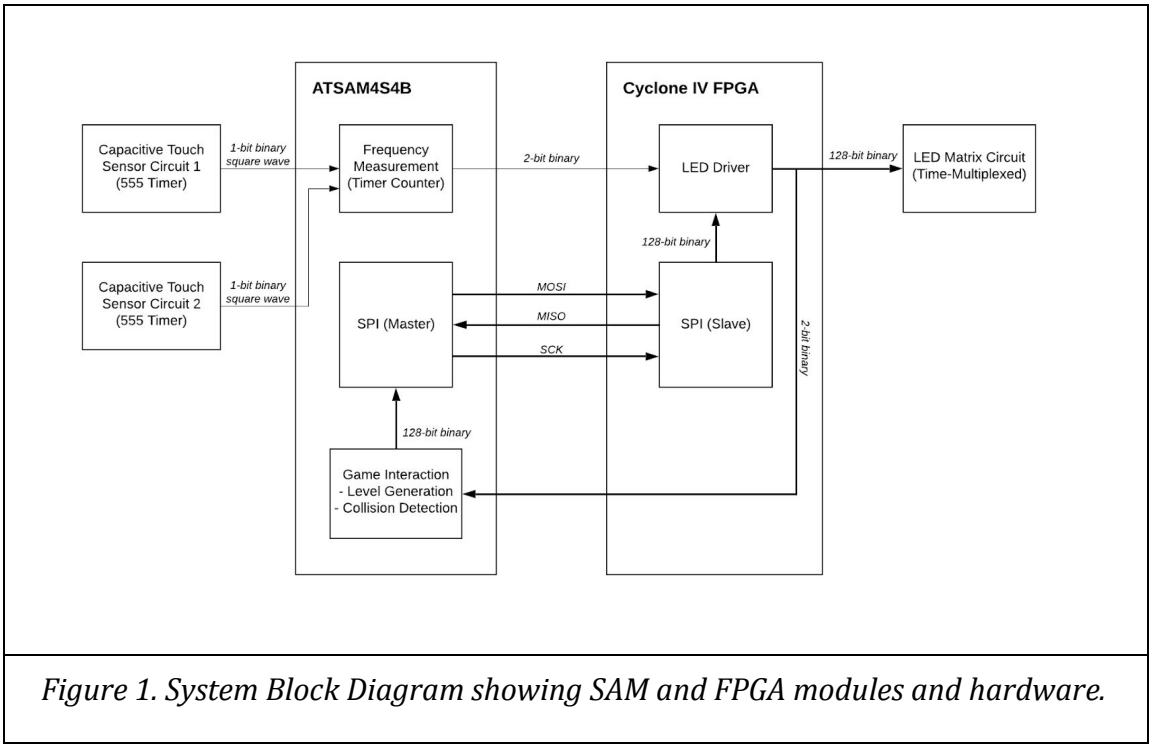


Figure 1. System Block Diagram showing SAM and FPGA modules and hardware.

New Hardware

This section documents the implementation of the capacitive touch sensors. The team used milled copper clad pads as the sensors and used a 555 timer for the sensing circuit, which outputs a square wave with a frequency that changes when the sensor is touched. The circuit is shown in Figure 2. The circuit design is from the LM555 datasheet [4]. The system implementation of the sensors is shown in Figure 3.

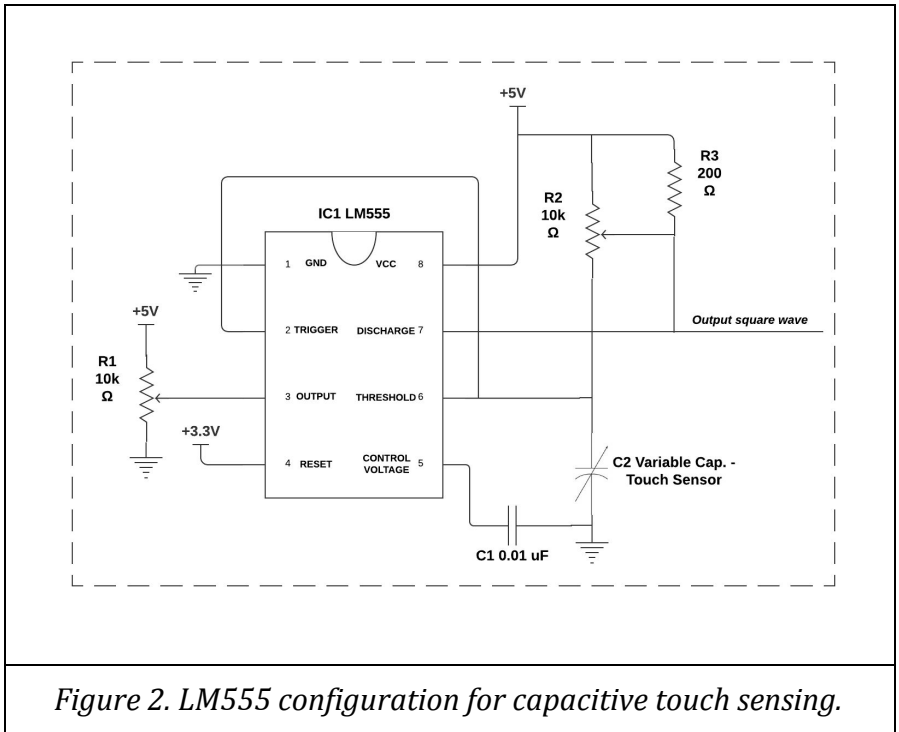


Figure 2. LM555 configuration for capacitive touch sensing.

Key features of the design shown above are C2, the capacitive touch sensor, and the potentiometers R1 and R2. R1 controls the duty cycle of the square wave, and R2 controls the time constant, and consequently the frequency. The team experimentally tuned the circuits to have approximately a 50% duty cycle and 500 kHz frequency when unpressed. Future teams should also note that moving the capacitive sensor farther from the circuit will result in more copper wire, which will change the frequency of the unpressed wave - any measuring systems should be tuned to accommodate this behavior.

Schematics

This section shows and discusses the circuitry used in the project.

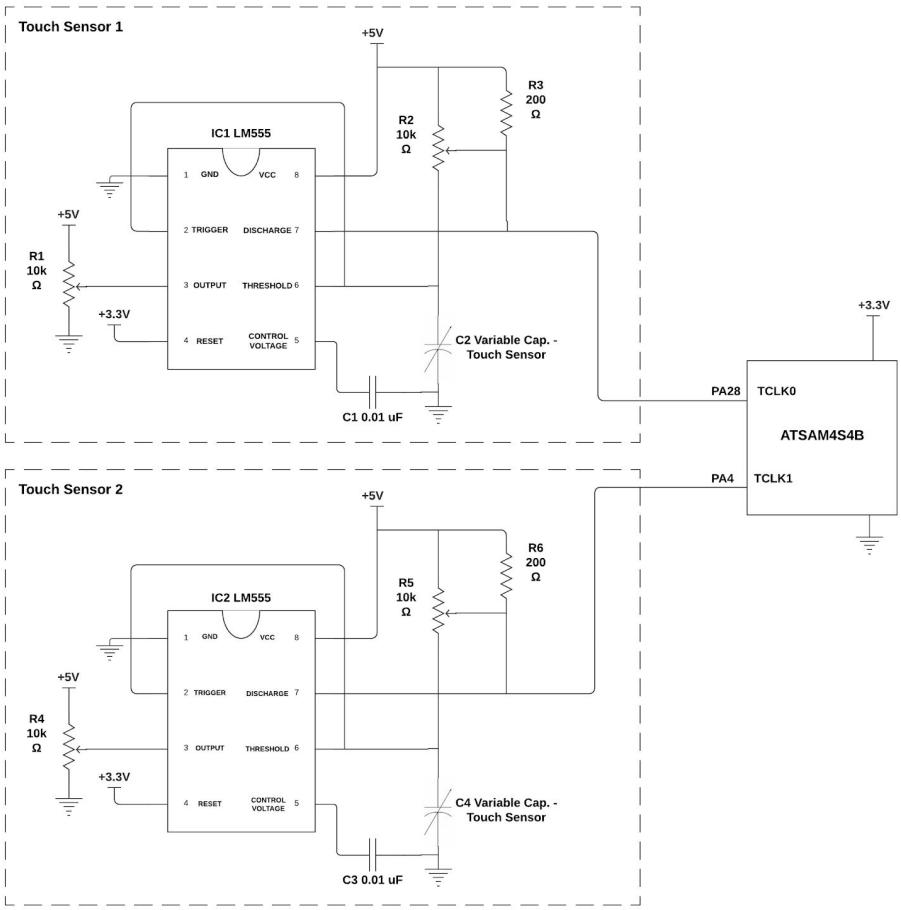


Figure 3. Capacitive touch sensing circuits and connectivity to the ATSAM.

Figure 3 shows the diagram for the capacitive touch sensors. The team built two sensors following the design discussed in the New Hardware section.

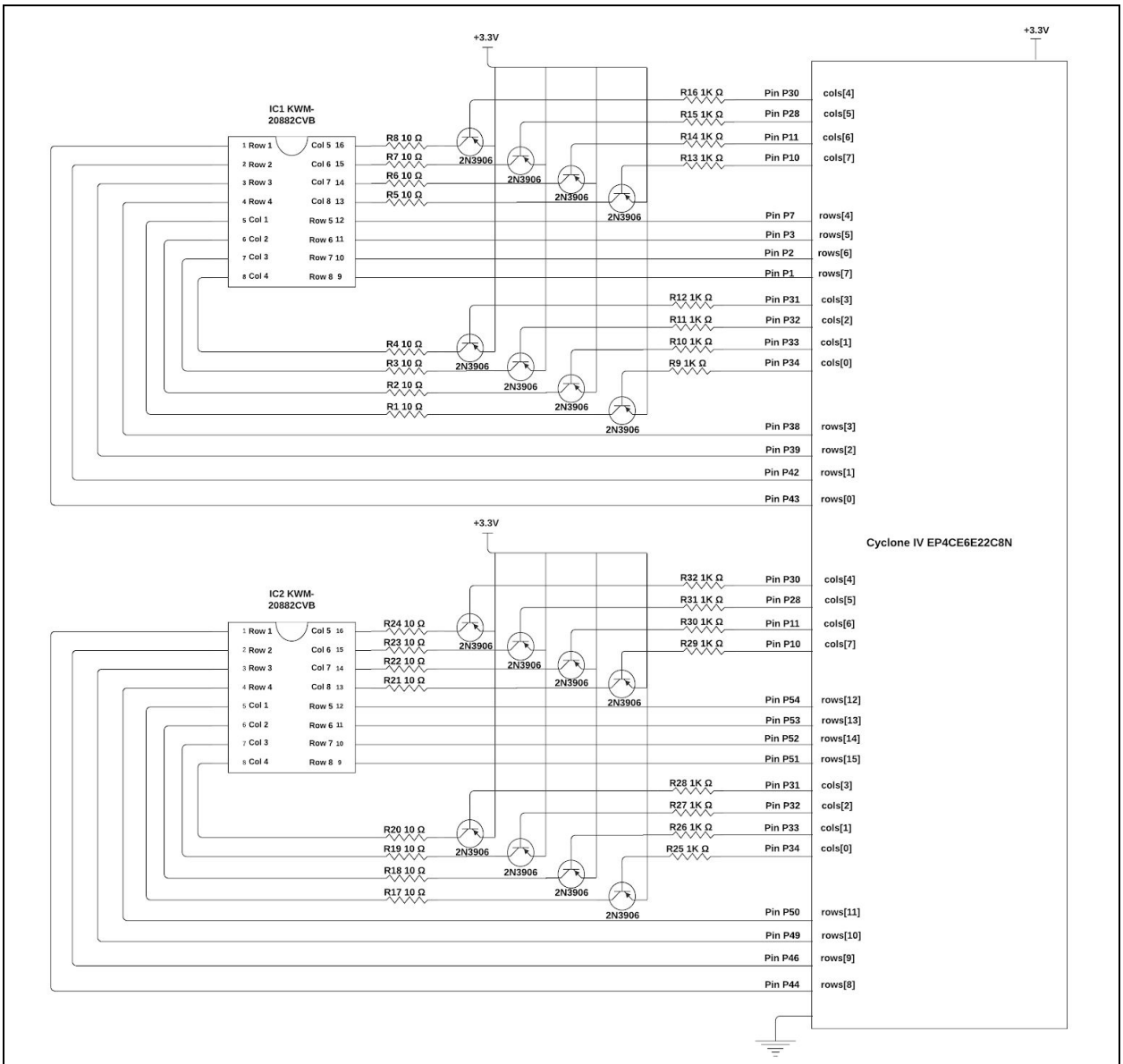


Figure 4. LED display driving circuits and connectivity to the FPGA.

Figure 4 shows the LED display driving circuits, which was set up to drive the 16-pin package according to the LED matrix datasheet [1], with the internal matrix circuitry shown in Figure 5. A key feature of this circuit is the use of PNP transistors [5] which gives a constant current to any illuminated LEDs, creating a consistent brightness across the screen. The resistor values connected to the base and collector were selected to drive the

transistor in saturation. Also note that the column-driving pins are shorted to each other—a design choice elaborated on below in the FPGA Design section.

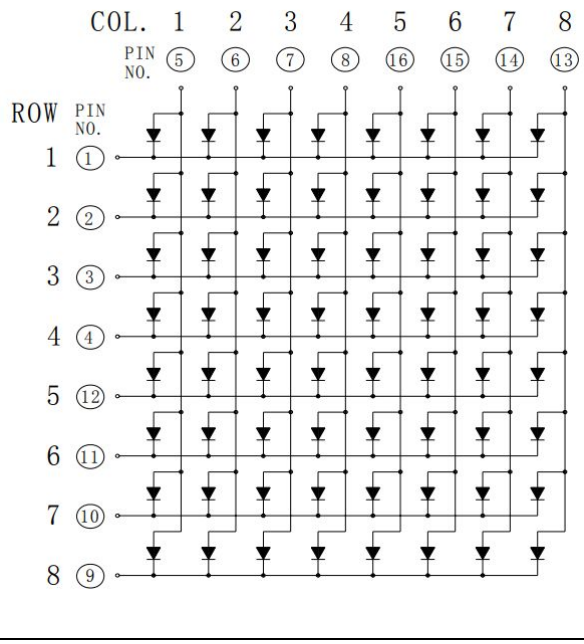


Figure 5. LED matrix internal schematic.

Figure 6 shows the circuitry between the ATSAM and FPGA, which consists of seven wires. Three of these wires are used to send the level data from the ATSAM to the FPGA over SPI—note that chip select is unused given that the FPGA is the only slave. Two wires carrying the *cap_1* and *cap_2* signals are used to transfer the signal from the touch sensors, with a pressed sensor giving a signal of 1 and an untouched sensor giving a signal of 0. Two wires carrying the signals *player_right* and *player_left* are used to are used to transmit the player location from the FPGA to the ATSAM to determine if a collision occurred.

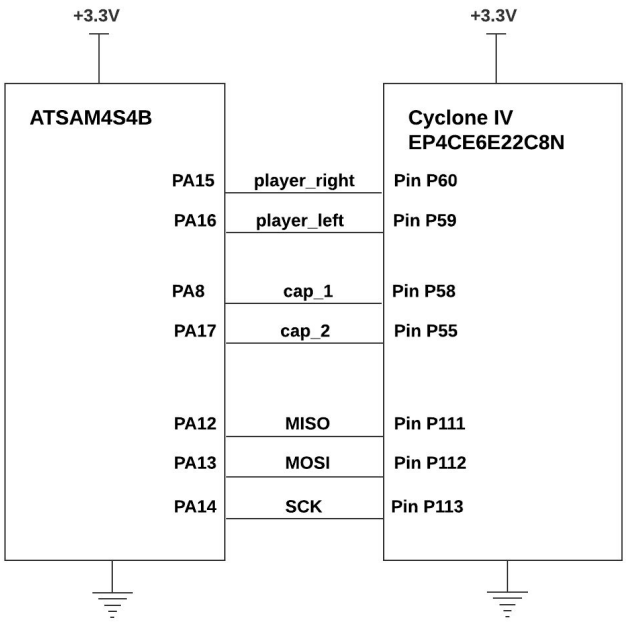


Figure 6. Connections between the ATSAM and FPGA.

Microcontroller Design

The ATSAM is tasked with measuring the capacitive touch inputs and handling the game interactions—level generation and collision detection, as well as sending the game data over SPI. The software routines implemented are:

Capacitive Touch Interpretation. The Timer Counter peripheral calculates the frequencies of the touch sensor circuits. This routine uses the square wave outputs from each touch sensor circuit to increment counters within the TC peripheral, then calculate their frequencies using a direct frequency counting method [6]. Comparing this value to an experimentally determined threshold, the routine determines if a sensor has been pressed, and sets outputs appropriately.

Level Shifting and Pseudo-Random Selection. The game data is stored as character arrays, which are sent over SPI on every time the level needs to be shifted down a row. Every 16 times the display is shifted down (after a full screen has passed), the routine takes a seed from the TC value to randomly select the next obstacle from a premade set, ensuring that each game instance is different and possible to win.

Difficulty Increasing and Level Incrementation. The game increases the difficulty over time by speeding up the refresh rate of the screen as the player progresses, giving the player a shorter window to react. After successfully passing a predetermined number of obstacles, the game increases the "Level" of the game, with a premade "Level Up" frame shown before the corresponding speed increase. If the player successfully beats Levels 0 - 3, they are shown a victory screen!

Game Data Transmission over SPI. Using the SPI framework from E155 Lab 7, this routine sends a full frame of level data, 128 bits, for the FPGA to display. Combined with the level-shifting routine, the obstacles move smoothly down the display one row at a time at the speed specified by the player's progress in the game.

Collision Detection and Game Over Screen. On every screen refresh, this routine determines if the player has collided with an obstacle by performing boolean algebra on the bottom row of the game data and the player's location. If the player and an obstacle are at the same location, then the routine sends a static image to the FPGA to be displayed until the player resets the game to try again.

FPGA Design

The FPGA intakes the game data over SPI and drives the LED display accordingly. It also determines the player location and sends this information to the ATSAM for collision detection.

Player Row Control. To incorporate the capacitive touch signals from the ATSAM, the FPGA has a series of flip-flops to intake the touch signal and change the player location appropriately. This allows for the player LED to stay on the same side if there is no touch signal, and to ignore overlapping presses (pressing the left while the right is already pressed).

Player Location Tracking. The ATSAM utilizes the player location to determine collisions. To communicate the location of the player, the FPGA communicates with the ATSAM whenever the player jumps from one side of the screen to the other. This is done in the form of the signals *player_right* and *player_left*, shown in Figure 6, which give a signal of 1 if the player is on the right or left side, respectively, and 0 if the player is on the other side.

Time-Multiplexed LED Driver. As seen in the LED matrix schematic shown in Figure 5, each row is driven by a single anode and each column is driven by a cathode. Thus to be able to independently control each LED, time-multiplexing is required. The FPGA therefore time-multiplexes using an FSM with has eight main states that it cycles through to drive each column, and sets the appropriate row values. The state transition diagram is shown below in Figure 7, and its outputs are shown in Table 1.

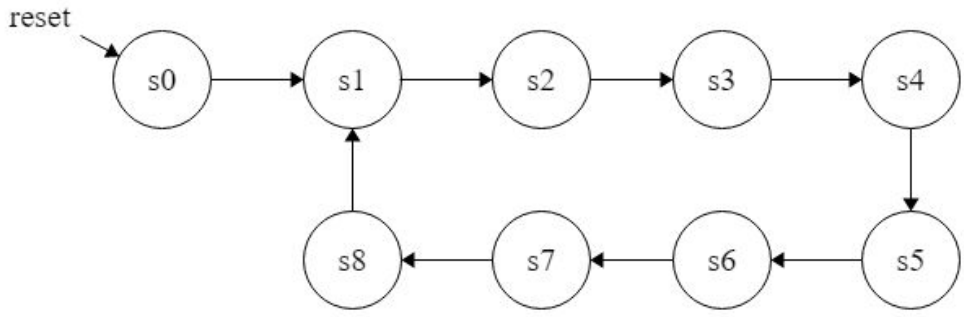


Figure 7. Column-multiplexing FSM state transition diagram.

Note that this FSM has a straightforward transition diagram, as the complexity in the design comes from the integration with SPI and the process of assigning values to the rows, shown below in Table 1.

	Outputs	
State	cols[7:0]	rows[15:0]
s0	[1111_1111]	[0000_0000_0000_0000]
s1	[0000_0001]	[~bits[63], ~bits[55], ~bits[47], ~bits[39], ~bits[31], ~bits[23], ~bits[15], ~bits[7], ~bits[127], ~bits[119], ~bits[111],~bits[103],~bits[95],~bits[87],~bits[79],~bits[71]]
s2	[0000_0010]	[~bits[62],~bits[54],~bits[46],~bits[38],~bits[30],~bits[22],~bits[14], ~bits[6], ~bits[126], ~bits[118],~bits[110],~bits[102],~bits[94],~bits[86],~bits[78],~bits[70]]
s3	[0000_0100]	[~bits[61],~bits[53],~bits[45],~bits[37],~bits[29],~bits[21],~bits[13], ~bits[5], ~bits[125], ~bits[117],~bits[109],~bits[101],~bits[93],~bits[85],~bits[77],~bits[69]]
s4	[0000_1000]	[~bits[60],~bits[52],~bits[44],~bits[36],~bits[28],~bits[20],~bits[12], ~bits[4],~bits[124], ~bits[116],~bits[108],~bits[100],~bits[92],~bits[84],~bits[76],~bits[68]]
s5	[0001_0000]	[~bits[59],~bits[51],~bits[43],~bits[35],~bits[27],~bits[19],~bits[11], ~bits[3],~bits[123], ~bits[115],~bits[107],~bits[99],~bits[91],~bits[83],~bits[75],~bits[67]]
s6	[0010_0000]	[~bits[58],~bits[50],~bits[42],~bits[34],~bits[26],~bits[18],~bits[10], ~bits[2], ~bits[122], ~bits[114],~bits[106],~bits[98],~bits[90],~bits[82],~bits[74],~bits[66]]
s7	[0100_0000]	[~bits[57],~bits[49],~bits[41],~bits[33],~bits[25],~bits[17],~bits[9], ~bits[1],~bits[121], ~bits[113],~bits[105],~bits[97],~bits[89],~bits[81],~bits[73],~bits[65]]
s8	[1000_0000]	[~bits[56],~bits[48],~bits[40],~bits[32],~bits[24],~bits[16],~bits[8], ~bits[0],~bits[120], ~bits[112],~bits[104],~bits[96],~bits[88],~bits[80],~bits[72],~bits[64]]

Table 1. Column-multiplexing FSM output table.

While the above outputs are admittedly an eyesore, the rows follow a straightforward pattern of ~bits[n+8], where “n” is an integer from 0-7, as the indexes wrap around after each 8 bits, representing a full row. The values in rows[7:0] and rows [15:8] are also

swapped from the expected indexing—this was implemented to address an issue where obstacles would update halfway down the screen, rather than at the top of the display.

Results

Overall, the project was successful because the team met all of our prescribed tasks and design goals. The team also implemented a better user interface for the system, creating a handheld system for the LED display and the capacitive sensors, shown in Appendix C. The riskiest part entering the project the capacitive touch sensors, as the team would be creating our own sensor which could be as involved as the team let it. However, our initial idea of an analog sensor changed from Prof. Harris' suggestion of a 555 timer circuit, which let us implement a familiar peripheral in the Timer Counter. This was the only deviation from our initial proposal, which proposed using an ADC and analog circuitry to measure capacitance. This sensor scheme made the capacitive sensors reliable and configurable, as the sensor signal was consistent in its behavior, and potentiometers allowed the team to tune the output frequency and duty cycle of the 555 timer.

The most challenging part of the design was the level generation system, as our implementation of SPI necessitated that the system use pointers to change the level data arrays, rather than simply changing which arrays were being sent, which would have been more straightforward. The team also encountered issues with the indexing, as at first the obstacles refreshed halfway down the screen rather than at the top of the display. The resulting fix can be seen in the bit swizzling in Table 1.

In the future, the team would likely change our implementation of SPI to avoid synchronicity issues. Rather than sending levels to the FPGA throughout the gameplay, the ATSAM could generate the entire game at once, send the level data over to the FPGA through a series of SPI transactions, and then cycle through the level data on the FPGA.

Additionally, there were several features that users at demo day recommended that would improve the overall design. They are presented in the following list:

- System Improvements:
    - *Reset button on handheld system.* The final product of our system was a handheld system, as shown in Appendix C. The team used the ATSAM reset button on the board to start a new game, which was effective but required prior knowledge. Having a clearly labeled reset button on the handheld system would improve the user experience.

- Functional Improvements:
    - *Score Counter*. Though the current game allowed for players to measure their progress by beating levels, player quickly became competitive and wanted to compare their exact progress to prior runs and other players. Adding an obstacle counter as a "score" and a display to show the current and max scores could thus add to the competition and enjoyment of the game.
    - *Scrolling Player*. One of the most common comments was that the player jumping (or rather, teleporting) faster than the level changes was awkwardly timed and difficult to get used to, causing some players to become frustrated (and at times, mildly addicted to the game). Having the player slide across the screen one LED at a time would add an additional factor of timing into the gameplay, and potentially make the gameplay more aligned with players' past experience.

References

[1] Luckylight 8×8 Hyper Red Dot Matrix LED Displays, http://pages.hmc.edu/harris/class/e85/ledmatrix.pdf
[2] Atmel ATSAM4S Family Datasheet, http://pages.hmc.edu/harris/class/e155/ATSAM4S_Family_Datasheet.pdf
- SPI Peripheral, pp. 686-718
- Timer Counter Peripheral, pp. 851-904

[3] Altera Cyclone IV Device Handbook, http://pages.hmc.edu/harris/class/e155/cyclone4-handbook.pdf
[4] Texas Instruments LM555 Timer, http://www.ti.com/lit/ds/symlink/lm555.pdf
[5] Small Signal PNP Transistor, http://pages.hmc.edu/harris/class/e155/2N3906.pdf
[6] Frequency Counter Methods, https://www.best-microcontroller-projects.com/article-frequency-counter.html

Parts List

Part	**Source**	**Vendor Part #**	**Price**
Microcontroller	E155	ATSAM4S4B	—
FPGA	E155	Cyclone IV EP4CE6E22C8N	—
8x8 LED Matrix	E85	KWM-20882CVB	—
PNP Transistor	E85	2N3906	—
Copper clad	Clinic Maker Space	—	—
Wood stock	Machine Shop (scrap wood)	—	—
555 Timer	Stockroom	LM555	—
10kΩ Potentiometer	Stockroom	—	—
39.4 in. Jumper Wires	EDGELEC (Amazon)	B07GD17ZF3	$21.88

Table 2. Parts used in the Gravity Game project.

Appendices

Appendix A: C Code (for ATSAM)

```
// final_project_fm_nn.c
// Francisco Munoz, Nico Naar
// fmunoz@hmc.edu, nnaar@hmc.edu 13 December 2019

/*
Gravity game implemented on 8x16 LED matrix,
       using two cap touch sensors to play game
*/

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

#include <stdio.h>
#include "SAM4S4B_pmc.h"
#include "SAM4S4B.h"
#include <stdlib.h> //for random function
#include <time.h>

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

//SPI
#define LOAD_PIN    29
#define DONE_PIN    30

// Touch Sensor 1 = TC0, CH1 (second sensor)
#define clock_id_1        5       // 5 = XC0 = TCLK0 = PA4
#define freq_threshold_1  500000  // touch sensitivity - > higher = more sensitive
#define TCLK0             PIO_PA4 // use TCLK0 for TC0CH1 direct clock -> output of
                                  // cap touch
#define goodpin_1         PIO_PA8 // reads high/low when touched/ not touched
#define badpin_1          PIO_PA9 // reads low/high when touched/ not touched
#define delay_dur         1000    // delay in us

// Touch Sensor 2 = TC0, CH2 (first sensor)
#define clock_id_2        6       // 6 = XC1 = TCLK1 = PA28
#define freq_threshold_2  500000  // touch sensitivity - > higher = more sensitive
```

```
#define TCLK1             PIO_PA28 // use TCLK1 for TC0CH2 direct clock -> output
                                   // of cap touch
#define goodpin_2         PIO_PA17 // reads high/low when touched/ not touched
#define badpin_2          PIO_PA18 // reads low/high when touched/ not touched

// Collision
#define coll_pin_right PIO_PA16 //PA16 on umudd board
#define coll_pin_left  PIO_PA15 //PA_15 on umudd board

//Interrupt
#define inter_loc_right PIO_PA10 //pin 20


//
// Frame Definitions
//
char frame_display[16] = {
       0x00, 0x00, 0x00, 0x00,
       0x00, 0x00, 0x00, 0x00,
       0x00, 0x00, 0x00, 0x00,
       0x00, 0x00, 0x00, 0x00
};
char intro_number_3[16] = {

       0x00,  0x00,  0x00,  0x00,
       0x00,  0x3C, 0x04, 0x04,
       0x3C, 0x04, 0x04, 0x3C,
       0x00,  0x00,  0x00,  0x00
};

char intro_number_2[16] = {

       0x00,  0x00,  0x00,  0x00,
       0x3C, 0x20, 0x20, 0x3C,
       0x04, 0x04, 0x3C, 0x00,
       0x00,  0x00,  0x00,  0x00
};

char intro_number_1[16] = {
       0x00,  0x00,  0x00,  0x00,
       0x00,  0x3C, 0x08, 0x08,
       0x08, 0x08, 0x28, 0x18,
       0x00,  0x00,  0x00,  0x00
};

char intro_number_0[16] ={
```

```
        0x00,  0x00,  0x00,  0x00,
        0x00,  0x3C, 0x24, 0x24,
        0x24, 0x24, 0x24, 0x3C,
        0x00,  0x00,  0x00,  0x00
};

char obs[16] = {
        0x80, 0xC0, 0xE0, 0xF0,
        0x00, 0x00, 0x00, 0x00,
        0x01, 0x03, 0x07, 0x0F,
        0x1F, 0x00, 0x00, 0x00,
};

// Obstacles
char body_level[192] = {

        // Triangles 1
        0x00, 0x80, 0xC0, 0xE0,
        0xF0,  0x00, 0x00, 0x00,
        0x00,  0x01, 0x03, 0x07,
        0x0F,  0x1F,  0x00, 0x00,

        // Triangles 2
        0x00, 0x00, 0x00, 0x80,
        0xC0, 0xE0, 0xF0, 0x00,
        0x00, 0x00, 0x00,   0x01,
        0x03, 0x07, 0x0F,   0x1F,

        // Triangles 3
        0x00, 0x00, 0x80, 0xC0,
        0xE0, 0xF0, 0x00, 0x00,
        0x00, 0x01, 0x03, 0x07,
        0x0F,  0x1F, 0x00, 0x00,

        // Flat 1
        0x00, 0x00, 0xF0, 0xF0,
        0xFC, 0xFC,  0x00, 0x00,
        0x00,  0x0F, 0x0F, 0x3F,
        0x3F,  0x00, 0x00, 0x00,

        // Flat 2
        0x00,  0xF0, 0xF0, 0xFC,
        0xFC,  0x00,  0x00, 0x00,
        0x00,  0x0F, 0x0F, 0x3F,
        0x3F,  0x00, 0x00, 0x00,

        // Flat 3
```

```
        0x00, 0x00, 0x00, 0xF0,
        0xF0, 0xFC, 0xFC,   0x00,
        0x00,  0x00, 0x0F, 0x0F,
        0x3F, 0x3F,  0x00, 0x00,

        // Cross 1
        0x00, 0x00, 0x90, 0xF8,
        0x90,  0x00, 0x00, 0x00,
        0x00, 0x00, 0x09, 0x1F,
        0x09, 0x00, 0x00, 0x00,

        // Cross 2
        0x00, 0x00, 0x00, 0x90, 0xF8,
        0x90,  0x00, 0x00, 0x00,
        0x00, 0x00, 0x09, 0x1F,
        0x09, 0x00, 0x00,

        // Cross 3
        0x00, 0x00,  0x90, 0xF8,
        0x90,  0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x09, 0x1F,
        0x09, 0x00, 0x00,


        // Longer Triangles 1
        0x00, 0x00, 0x80, 0xE0,
        0xF8, 0x00, 0x00, 0x00,
        0x01, 0x07, 0x1F, 0x00,
        0x00, 0x00, 0x00, 0x00,

        // Longer Triangles 2
        0x00, 0x00, 0x00, 0x80,
        0xE0,  0xF8, 0x00, 0x00,
        0x00, 0x00, 0x01, 0x07,
        0x1F, 0x00, 0x00, 0x00,

        // Longer Triangles 3
        0x00, 0x00, 0x00, 0x80,
        0xE0,  0xF8, 0x00, 0x00,
        0x00, 0x00, 0x01, 0x07,
        0x1F, 0x00, 0x00, 0x00,


};

// Game Over Screen
char game_over[16] = {
```

```
        0x00, 0x00, 0x3C, 0x24,
        0x2C,  0x20, 0x3C, 0x00,
        0x00,  0x3C, 0x24, 0x2C,
        0x20, 0x3C, 0x00, 0x00
};

// Level 0 Screen
char level_0[16] = {
        0x00, 0x00, 0x00, 0x00,
        0x00, 0x6E, 0x4A, 0x4A,
        0x4A, 0x4E, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00

};

// Level 1 Screen
char level_1[16] = {
        0x00, 0x00, 0x00, 0x00,
        0x00, 0x6E, 0x44, 0x44,
        0x4C, 0x44, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00

};

// Level 2 Screen
char level_2[16] = {
        0x00, 0x00, 0x00, 0x00,
        0x00, 0x6E, 0x48, 0x4E,
        0x42, 0x4E, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00

};

// Level 3 Screen
char level_3[16] = {
        0x00, 0x00, 0x00, 0x00,
        0x00, 0x6E, 0x42, 0x4E,
        0x42, 0x4E, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00

};

// GG Intro Screen
char gg_intro[16] = {
        0x00, 0x00, 0x3C, 0x24,
        0x2C,  0x20, 0x3C, 0x00,
        0x00,  0x3C, 0x24, 0x2C,
```

```
        0x20, 0x3C, 0x00, 0x00
};

// 'by' Intro Screen
char by_credits[16] = {
        0x00, 0x00, 0x3C, 0x24,
        0x3C,  0x20, 0x20, 0x00,
        0x00,  0x3C, 0x04, 0x04,
        0x3C, 0x24, 0x00, 0x00
};

// '&' Intro Screen
char and_credits[16] = {
        0x00, 0x00, 0x00, 0x00,
        0x00, 0x10, 0x38, 0x20,
        0x30, 0x20, 0x38, 0x10,
        0x00, 0x00, 0x00, 0x00
};

// 'fm' Intro Screen
char fm_credits[16] = {
        0x00, 0x00, 0x20, 0x20,
        0x3C,  0x20, 0x3C, 0x00,
        0x00,  0x24, 0x24, 0x24,
        0x3C, 0x24, 0x00, 0x00
};

// 'nn' Intro Screen
char nn_credits[16] = {
        0x00, 0x00, 0x24, 0x2C,
        0x3C,  0x34, 0x24, 0x00,
        0x00,  0x24, 0x2C, 0x3C,
        0x34, 0x24, 0x00, 0x00
};

// 'WP' Victory Screen!
char well_played[16] = {
        0x00, 0x00, 0x40, 0x40,
        0x7E,  0x42, 0x7E, 0x00,
        0x00,  0x66, 0x5A, 0x42,
        0x42, 0x42, 0x00, 0x00
};



////////////////////////////////////////////
// Function Prototypes
```

```
////////////////////////////////////////////////

void send_level(char*, char*, char*);
void refresh_level(int, char*, char*);

void send_level_test(char*, char*);
void move_level(char*);


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

int main(void) {

  char returned_SPI[16];
       int refresh = 0;
       int total_count = 0;
       int num_shifts = 0;
       int level_numb = 0;
       int speed = 24; // starting refresh rate

       int ran_num;

  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

       // TC Setup
       tcInit();
       tcDelayInit(); // set up TC0 as delay unit
       tcChannelInit(TC_CH1_ID,clock_id_1,0);
       tcChannelInit(TC_CH2_ID,clock_id_2,0);

       // I/O Setup
       // Touch Sensor 1
       pioPinMode(TCLK0,PIO_PERIPH_B);   // set XC0 pin for TC
       pioPinMode(goodpin_1,PIO_OUTPUT);  // set output 1 high
       pioPinMode(badpin_1,PIO_OUTPUT);         // set output 1 low

       // Touch Sensor 2
       pioPinMode(TCLK1,PIO_PERIPH_B);   // set XC1 pin for TC
       pioPinMode(goodpin_2,PIO_OUTPUT);  // set output 2 high
       pioPinMode(badpin_2,PIO_OUTPUT);         // set output 2 low
```

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

	//
	// Main Loop
	//
	while(1){

		// Display Powering
		if(refresh == speed)
		{
			// every time we want to refresh the screen, send the level to
the FPGA
			// over SPI and add the next frame

			// Start with title frame, follow with credit frames
			if(level_numb == 0){
				send_level (frame_display, gg_intro, returned_SPI);
				refresh_level(num_shifts, frame_display, gg_intro);
			}
			else if(level_numb == 1){
				send_level (frame_display, by_credits, returned_SPI);
				refresh_level(num_shifts, frame_display, by_credits);
			}
			else if(level_numb == 2){
				send_level (frame_display, fm_credits, returned_SPI);
				refresh_level(num_shifts, frame_display, fm_credits);
			}
			else if(level_numb == 3){
				send_level (frame_display, and_credits, returned_SPI);
				refresh_level(num_shifts, frame_display, and_credits);
			}
			else if(level_numb == 4){
				send_level (frame_display, nn_credits, returned_SPI);
				refresh_level(num_shifts, frame_display, nn_credits);
			}
			// After credits, display Level 0 frame
			else if(level_numb == 5){
				send_level (frame_display, level_0, returned_SPI);
				refresh_level(num_shifts, frame_display, level_0);
			}
			// After 10 levels, display Level 1 frame
			else if(level_numb == 15){
				send_level (frame_display, level_1, returned_SPI);
				refresh_level(num_shifts, frame_display, level_1);
			}
```

```
                    // After 20 levels, display Level 2 frame
                    else if(level_numb == 25){
                          send_level (frame_display, level_2, returned_SPI);
                          refresh_level(num_shifts, frame_display, level_2);
                    }
                    // After 30 levels, display Level 3 frame
                    else if(level_numb == 35){
                          send_level (frame_display, level_3, returned_SPI);
                          refresh_level(num_shifts, frame_display, level_3);
                    }
                    // After 38 levels, display victory frame!
                    else if(level_numb == 43){
                          send_level (well_played, well_played, returned_SPI);
                          while(1);
                    }
                    // Otherwise use randomly selected frame
                    else {
                          send_level( frame_display, obs, returned_SPI);
                          refresh_level(num_shifts, frame_display, obs);
                          total_count++;
                    }

                    refresh = 0;  // Reset display refresh count
                    num_shifts++; // Increment overall screen counter

                    // Difficulty Incrementation
                    if(total_count == 8)                          speed = 49; //
Level 0
                    else if (total_count == 145) speed = 32; // Level 1
                    else if (total_count == 290) speed = 24; // Level 2
                    else if (total_count == 430) speed = 19; // Level 3


                    // Obstacle selection and Level Incrementation

                    // if display has fully moved over previous frame, enter if
statement:
                    if(num_shifts == 16)
                          {
                          num_shifts = 0;
                          level_numb++;
                          int x_obs;

                          // If the player hasn't advanced a level, randomly select
the next obstacle
                          if(level_numb > 4) {
                                srand(TC0->TC_CH[1].TC_CV);
```

```
                                ran_num = rand()%12; //random number between 0 to
(moded number-1)
                                for(x_obs = 0; x_obs <16; x_obs++){
                                      obs[x_obs] = body_level[x_obs +
ran_num*16];
                                }     }
                          }
             }

                    // Collision Detection
                    int collision_right = pioDigitalRead(coll_pin_right) &
frame_display[0] & 1<<0x00; // Check player on right side
                    int collision_left  = pioDigitalRead(coll_pin_left)  &
frame_display[7] & 1<<0x00; // Check player on left side

                    if(collision_left|collision_right){
                          // Game over if player has collided on either side
                          while(1){
                                send_level(game_over, game_over, returned_SPI); //
reset to exit game over
                          }

                    }

             refresh++; // Add to refresh counter


             //
             //     Touch Sensor 1
             //
             tcResetChannel(TC_CH1_ID);
                    // reset to begin
             tcDelayMicroseconds(delay_dur);  // delay for 1000 us = 1 ms
             float counter_value_1 = TC0->TC_CH[1].TC_CV;  // read counter value
after 1 ms
             float time_dur_inv_1 = 1000000 / delay_dur; // scale by 10e6 to put in
1 / seconds
             float freq_1 = counter_value_1 * time_dur_inv_1; // signal frequency =
counter value * (1/time_dur)

                    // If pressed, set output high
                    if (freq_1 < freq_threshold_1) {
                          pioDigitalWrite(goodpin_1, 1);
                          pioDigitalWrite(badpin_1, 0);
                    }
                    // If not, set output low
                    else {
```

```
                        pioDigitalWrite(badpin_1, 1);
                        pioDigitalWrite(goodpin_1, 0);
                }

            //
            //    Touch Sensor 2
            //
            tcResetChannel(TC_CH2_ID);
                    // reset to begin
            tcDelayMicroseconds(delay_dur);  // delay for 1000 us = 1 ms
            float counter_value_2 = TC0->TC_CH[2].TC_CV; // read counter value
after 1 ms
            float time_dur_inv_2 = 1000000 / delay_dur; // scale by 10e6 to put in
1 / seconds
            float freq_2 = counter_value_2 * time_dur_inv_2; // signal frequency =
counter value * (1/time_dur)

                    // If pressed, set output high
                    if (freq_2 < freq_threshold_2) {
                        pioDigitalWrite(goodpin_2, 1);
                        pioDigitalWrite(badpin_2, 0);
                    }
                    // If not, set output low
                    else {
                        pioDigitalWrite(badpin_2, 1);
                        pioDigitalWrite(goodpin_2, 0);
                    }

        }
}

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


void send_level(char *first_level, char *second_level, char *cyphertext) {
  int i;

  pioDigitalWrite(LOAD_PIN, 1);

  for(i = 0; i < 16; i++) {
    spiSendReceive(second_level[i]);
  }

  for(i = 0; i < 16; i++) {
    spiSendReceive(first_level[i]);
```

```
  }

  pioDigitalWrite(LOAD_PIN, 0);

  while (!pioDigitalRead(DONE_PIN));

  for(i = 0; i < 16; i++) {
    cyphertext[i] = spiSendReceive(0);
  }
}


void refresh_level(int index, char *level, char *new_level) {
      // write function to move level down one by 1
      move_level(level);
      move_level(new_level);
      level[15] = new_level[15];
}


void move_level(char *level) {
      // write function to move level down one by 1

      int m;
      char hold;
      hold = level[0];
      for(m = 0; m<15; m++){
            level[m] = level[m+1];
      }
      level[15] = hold;
}
```

Appendix B: SystemVerilog Code for the Cyclone IV FPGA

```
// FPGA_final_project
// Francisco Munoz fmunoz@g.hmc.edu
// Nico Naar nnaar@g.hmc.edu
// 13 December 2019

module FPGA_final_project(
                    input  logic clk,
                    input      logic cap1,
                    input      logic cap2,
           input  logic sck,
           input  logic sdi,
           output logic sdo,
           input  logic load,
           output logic done,
                    output logic [15:0]rows,
                    output logic [7:0]cols,
                    output logic player_right,
                    output logic player_left);

    logic [127:0] level_1_1, level_1_2, return_txt;
        logic rst;

        //create slow clk
        logic [24:0]counter;
        logic slw_clk;
        logic [7:0]cols_inv;

        assign rst = 0;

        //counter
        always_ff @(posedge clk)
              counter <= counter + 25'b1;

       always_ff @(posedge clk)
              done <=~load & (counter[15:10]==0);

        assign slw_clk = counter[15];//[24];
        //matrix_multiplex mm(slw_clk, rows, cols, test_port);
        spi spi1(sck, sdi, sdo, done, level_1_1, level_1_2, return_txt);
        game g0(slw_clk, rst, level_1_2, return_txt, cap1, cap2, rows, cols_inv,
player_right);
        assign player_left = ~player_right;
```

```
        assign cols = ~cols_inv; //invert for transistors
endmodule

module game(
                    input logic clk,
                    input logic rst,
                    input logic [127:0]level_1_2,
                    input logic [127:0]return_txt,
                    input logic cap1,
                    input logic cap2,
                    output logic [15:0]rows,
                    output logic [7:0]cols,
                    output logic player_right);

      logic [15:0] rows_player;
      logic [15:0] rows_level;
      logic [7:0] cols_player;
      logic [7:0] cols_level;
      logic test_port;
      logic next_player_right;

            // shifting variables to hold player info
            logic [15:0] rows_player_old;
            logic [7:0] cols_player_old;


            // flop to shift over player info
            always_ff @(posedge clk)
                  begin
                        if(cap1|cap2)
                              begin
                                    rows_player_old <= rows_player;
                                    cols_player_old <= cols_player;
                              end
                        else
                              begin
                                    rows_player_old <= rows_player_old;
                                    cols_player_old <= cols_player_old;
                              end
                        player_right <= next_player_right;
                  end

      // Player Row Control
            always_comb
            begin
                  if(~cap1&~cap2)
                        begin
```

```
                        cols_player <= cols_player_old;
                        rows_player <= rows_player_old;
                        next_player_right <= player_right;
                    end
                else if (cap2&~cap1)
                    begin
                        cols_player <= 8'b1000_0000; //on == high
                        rows_player <= 16'b1111_1111_0111_1111; // on low
                        next_player_right<= 1;
                    end
                else if (~cap2&cap1)
                    begin
                        cols_player <= 8'b0000_0001; //1 = on
                        rows_player <= 16'b1111_1111_0111_1111; // 0 = on
                        next_player_right<= 0;
                    end
                else
                    begin
                        cols_player <= cols_player_old;
                        rows_player <= rows_player_old;
                        next_player_right <= player_right;
                    end
            end


    matrix_multiplex mm(clk, level_1_2, rows_level, cols_level, test_port);

    //flop to place the player dot
    always_ff @(posedge clk)
        if (cols_level == cols_player)
            begin
                rows <= rows_level&rows_player; //& because rows should
be low if either is low (to turn LED on
                cols <= cols_level;
            end
        else
            begin
                rows <= rows_level;
                cols <= cols_level;
            end

endmodule

module matrix_multiplex(input  logic clk,
                                                input   logic [127:0] bits,
                                                //input  logic reset,
                                                output logic [15:0] rows,
```

```
                                                output logic [7:0] cols,
                                                output logic test_port);


        logic l2p_load;
        assign test_port = clk;

        FSM_multiplex fsm01(clk,l2p_load, bits, rows, cols);

endmodule


module FSM_multiplex(input  logic clk,
                                                //input  logic reset,
                                                input  logic l2p_load,
                                                input  logic [127:0] bits,
                                                output logic [15:0] rows,
                                                output logic [7:0] cols);

        typedef enum logic [5:0] {s0,s1,s2,s3,s4,s5,s6,s7,s8} statetype;

        statetype state, ns;

        // state register
        always_ff@(posedge clk)
                state <= ns;

        always_comb
                case(state)
                        s0:
                                begin
                                        cols = 8'b0000_0001;
                                        rows = 16'b1111_1111_1111_0000;
                                        ns = s1;
                                end
                        s1:
                                begin
                                        cols = 8'b0000_0001;
                                        rows = { ~bits[63], ~bits[55], ~bits[47],
~bits[39], ~bits[31], ~bits[23], ~bits[15], ~bits[7], ~bits[127], ~bits[119],
~bits[111],~bits[103],~bits[95],~bits[87],~bits[79],~bits[71]};
                                        ns = s2;
                                end
                        s2:
                                begin
                                        cols = 8'b0000_0010;
                                        rows =
```

```
{~bits[62],~bits[54],~bits[46],~bits[38],~bits[30],~bits[22],~bits[14], ~bits[6],
~bits[126],
~bits[118],~bits[110],~bits[102],~bits[94],~bits[86],~bits[78],~bits[70]};
                                ns = s3;
                        end
                s3:
                        begin
                                cols = 8'b0000_0100;
                                rows =
{~bits[61],~bits[53],~bits[45],~bits[37],~bits[29],~bits[21],~bits[13], ~bits[5],
~bits[125],
~bits[117],~bits[109],~bits[101],~bits[93],~bits[85],~bits[77],~bits[69]};
                                ns = s4;
                        end
                s4:
                        begin
                                cols = 8'b0000_1000;
                                rows =
{~bits[60],~bits[52],~bits[44],~bits[36],~bits[28],~bits[20],~bits[12],
~bits[4],~bits[124],
~bits[116],~bits[108],~bits[100],~bits[92],~bits[84],~bits[76],~bits[68]};
                                ns = s5;
                        end
                s5:
                        begin
                                cols = 8'b0001_0000;
                                rows =
{~bits[59],~bits[51],~bits[43],~bits[35],~bits[27],~bits[19],~bits[11],
~bits[3],~bits[123],
~bits[115],~bits[107],~bits[99],~bits[91],~bits[83],~bits[75],~bits[67]};
                                ns = s6;
                        end
                s6:
                        begin
                                cols = 8'b00100000;
                                rows =
{~bits[58],~bits[50],~bits[42],~bits[34],~bits[26],~bits[18],~bits[10], ~bits[2],
~bits[122],
~bits[114],~bits[106],~bits[98],~bits[90],~bits[82],~bits[74],~bits[66]};
                                ns = s7;
                        end
                s7:
                        begin
                                cols = 8'b01000000;
                                rows =
{~bits[57],~bits[49],~bits[41],~bits[33],~bits[25],~bits[17],~bits[9],
~bits[1],~bits[121],
```

```
~bits[113],~bits[105],~bits[97],~bits[89],~bits[81],~bits[73],~bits[65]};
                                ns = s8;
                        end
                s8:
                        begin
                                cols = 8'b10000000;
                                rows =
{~bits[56],~bits[48],~bits[40],~bits[32],~bits[24],~bits[16],~bits[8],
~bits[0],~bits[120],
~bits[112],~bits[104],~bits[96],~bits[88],~bits[80],~bits[72],~bits[64]};
                                ns = s1;
                        end
                default:
                        begin
                                cols = 8'b11111111;
                                rows = 8'b00000000;
                                ns = s0;
                        end
        endcase

endmodule


////////////////////////////////////////////
// spi
//   SPI interface.  Shifts in level_1_1 and level_1_2
//      Can later configure to send out return_txt if needed
////////////////////////////////////////////

module spi(input  logic sck,
           input  logic sdi,
           output logic sdo,
           input  logic done,
           output logic [127:0] level_1_1, level_1_2,
           input  logic [127:0] return_txt);

    logic         sdodelayed, wasdone;
    logic [127:0] return_txtcaptured;

    // assert load
    // apply 256 sclks to shift in level_1_1 and level_1_2, starting with
level_1_2[0]
    // then deassert load, wait until done
    // then apply 128 sclks to shift out return_txt, starting with return_txt[0]
    always_ff @(posedge sck)
        if (!wasdone)  {return_txtcaptured, level_1_2, level_1_1} = {return_txt,
level_1_2[126:0], level_1_1, sdi};
```

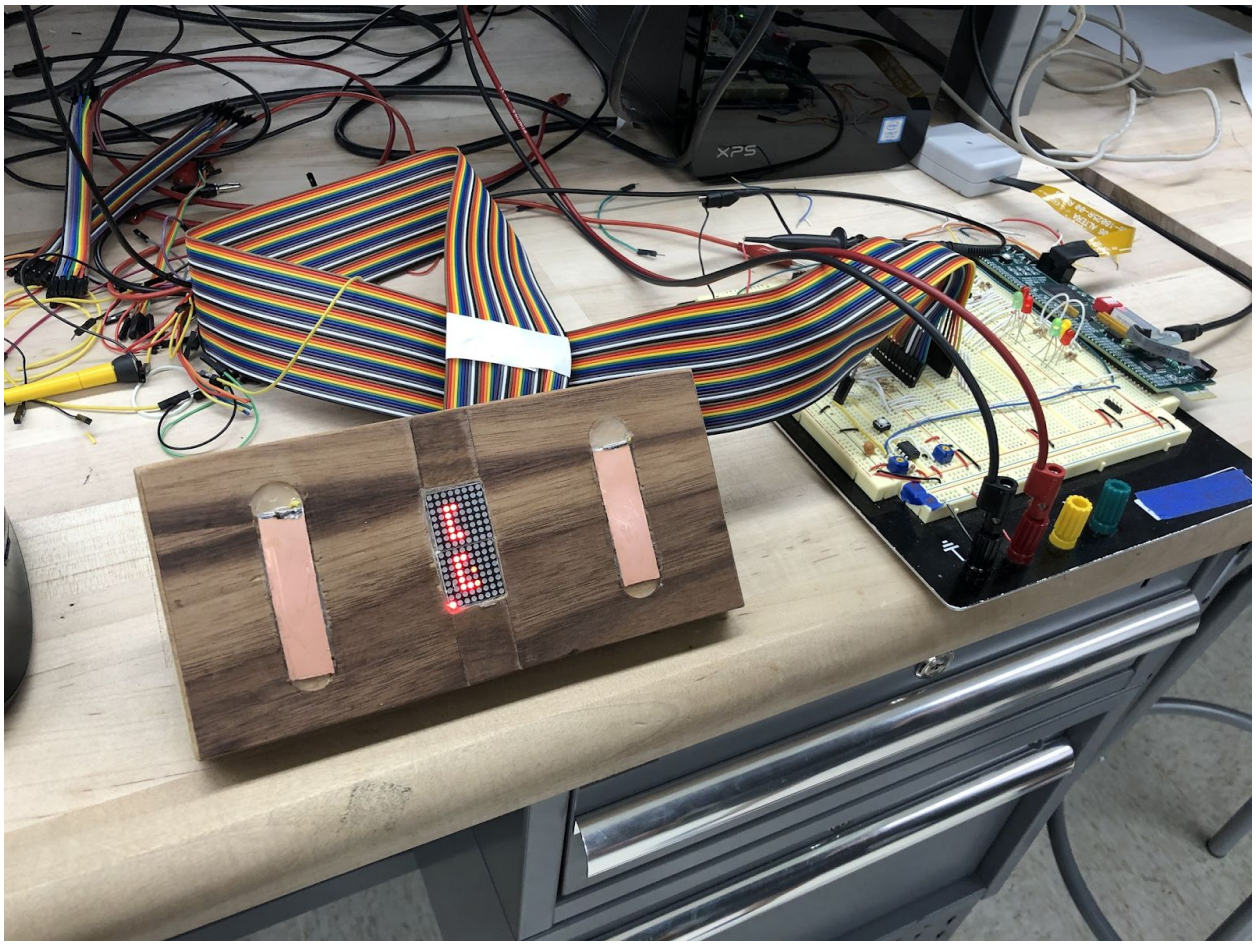
```
        else            {return_txtcaptured, level_1_2, level_1_1} =
{return_txtcaptured[126:0], level_1_2, level_1_1, sdi};

    // sdo should change on the negative edge of sck
    always_ff @(negedge sck) begin
        wasdone = done;
        sdodelayed = return_txtcaptured[126];
    end

    // when done is first asserted, shift out msb before clock edge
    assign sdo = (done & !wasdone) ? return_txt[127] : sdodelayed;
endmodule
```

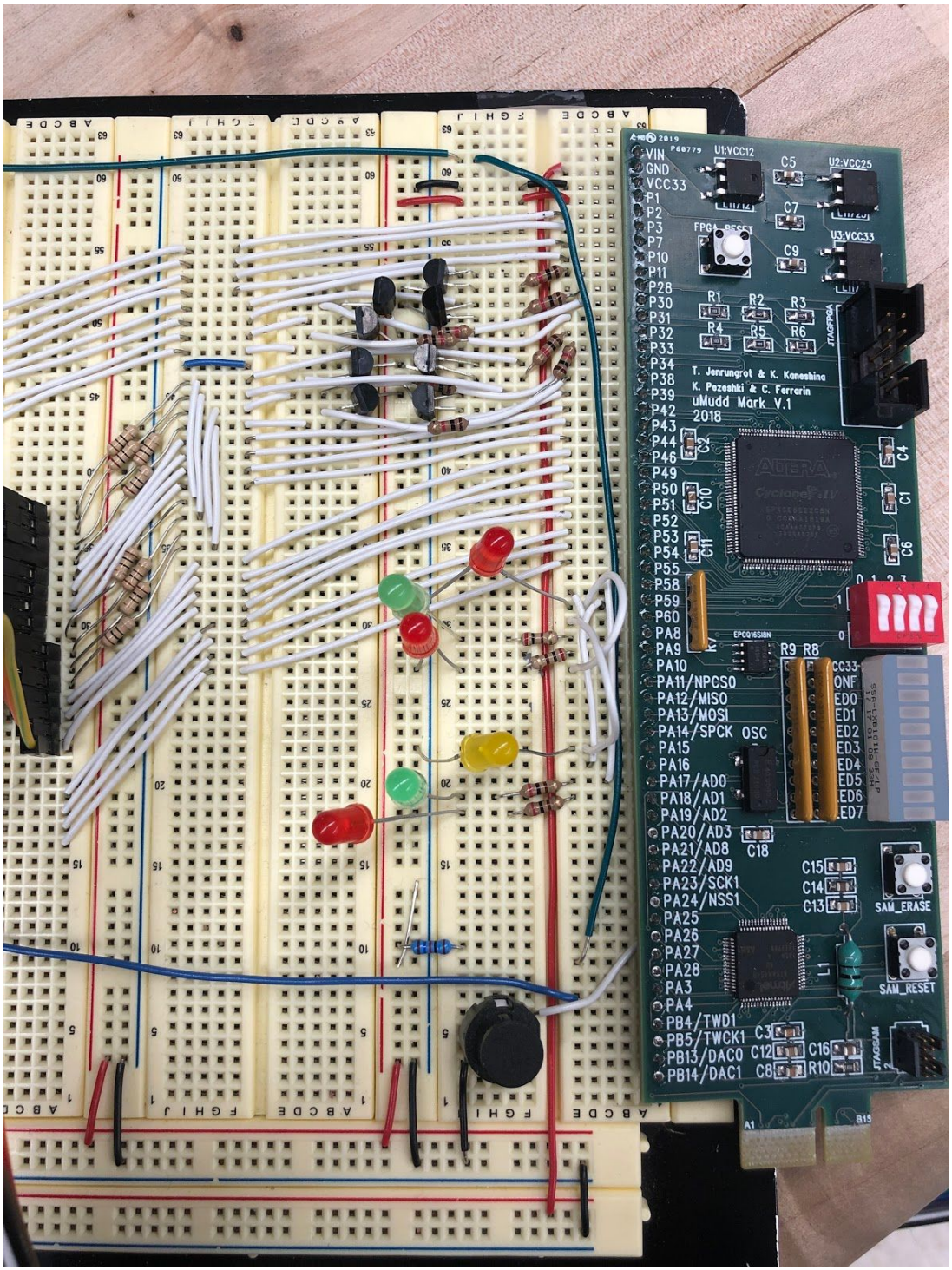
Appendix C: Pictures of final system

Complete System

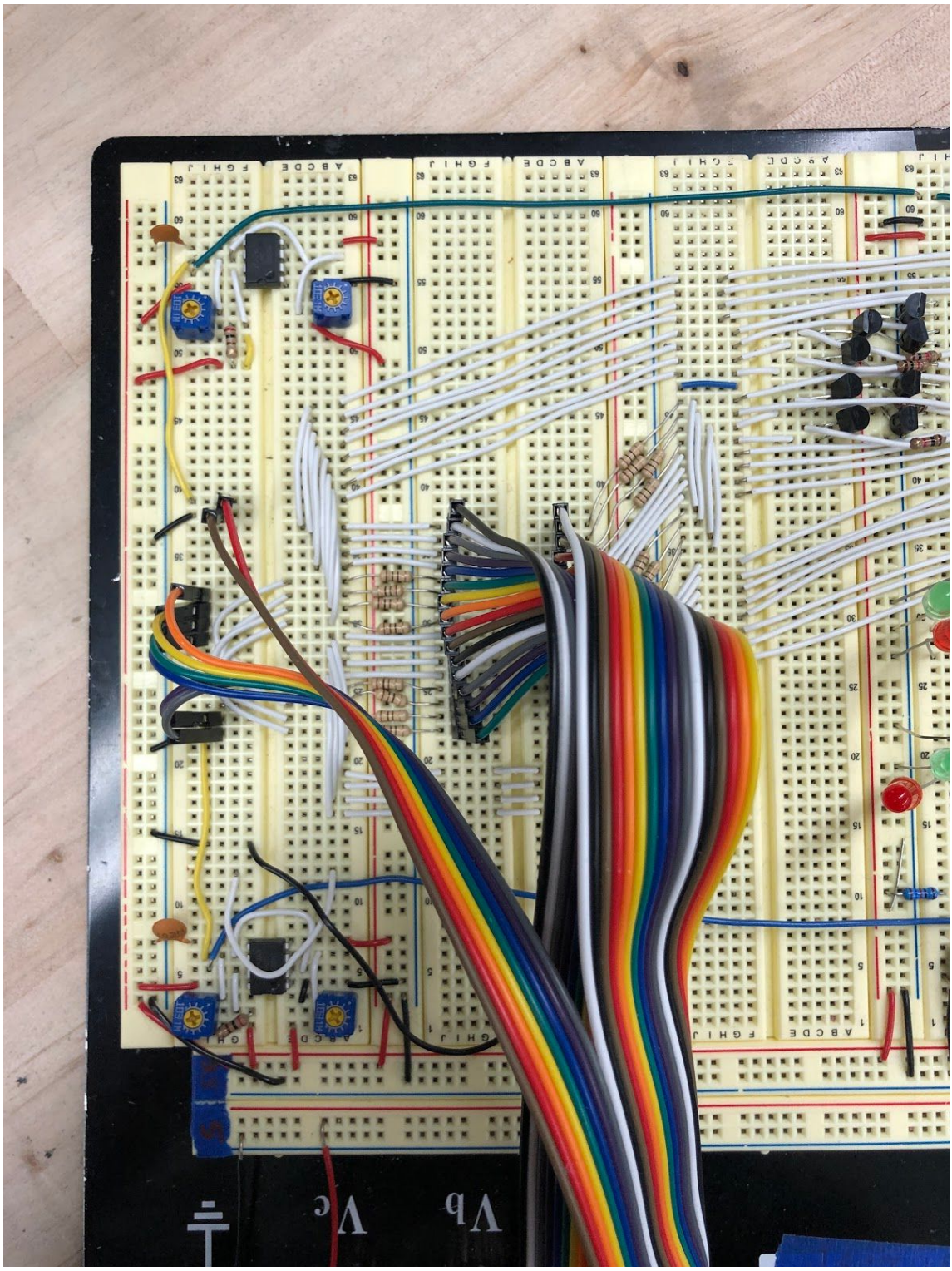


* Note that the camera shutter speed is fast enough that the image is not clear, but to human eyes this clearly reads “G G”

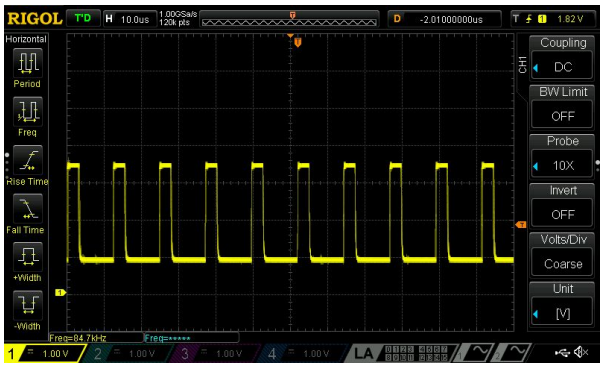
Breadboard 1/2



Breadboard 2/2



Appendix D: Capacitive Sensor Circuit Outputs

	
a. Unpressed 555 timer output, f = 454kHz, stable when unpressed for several seconds. Note that the frequency can vary between 400 to 600 kHz depending on the capacitance of each individual touch pad.	b. Pressed 555 timer output, f = 84.6kHz (and continued decreasing while held)