Final Project Report: Bead Maze with LED Matrix and Accelerometer

David Sobek and Jerry Liang
December 13th, 2019
E155 Microprocessor Systems: Design and Application

***Abstract* — The classic maze game (or sometimes referred to as the labyrinth game) is a small fun game where users rotate a small board to orient a bead through a maze. This game comes in other varieties as well, such as a control scheme where the player controls the orientation of the maze using two dials on the sides of the game. Our project aims to design a new version of this game, digitizing a maze and bead with an LED matrix and detecting the orientation with an accelerometer to control the bead – making the game more fun, flexible, colorful, and intriguing.**

Introduction

Our goal of this project is to recreate an old school maze puzzle toy digitally. There are many opportunities to enhance the classic maze game via a digital design, and it gives us the opportunity to learn how to interface with LED matrices (or displays in general) and design complex digital systems. The play style of the digital version of the game is very similar to the classic counterpart: Using a LIS3DH Triple-Axis Accelerometer, we detect the orientation of the board and use the readings from this device to simulate the particle physics of the bead. The brain of this operation is the ATSAM4S4B microcontroller we have been using for a good portion of the class. This microcontroller calculates the state of the bead and game as it reads from the accelerometer and updates the display by sending the new game state information to a Cyclone IV FPGA which dives the LED matrix. Displayed in the block diagram below (Figure 1) is the basic structure of the digital maze game.

Block Diagram

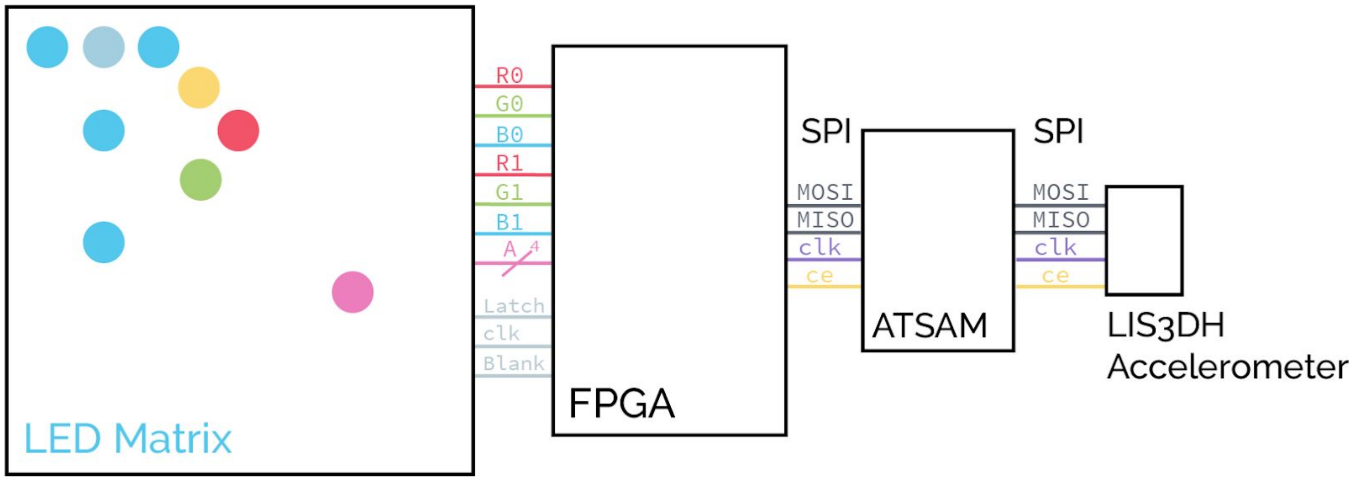

Figure 1. The block diagram of the digital maze game. Displayed is the FPGA driving the LED matrix over 13 buses and the ATSAM microcontroller communicating with both the FPGA and LIS3DH Accelerometer over SPI.

New Hardware

32x32 LED Matrix

Our first new hardware was the 32x32 LED matrix, which was particularly hard to work with because of the lack of documentation. There are no official datasheets for the LED matrix and the official Adafruit tutorial simply describes how to use their helper libraries and does not explain the operation of the LED matrix. Thankfully, we were able to find more in depth tutorials online, namely one from Glen Alkin[1], helping us understand the matrix pinout and operation basics, and one from Ray's Logic[2], which pointed to datasheets of shifters and drivers similar to those used in the Adafruit matrix.

How the panel works: The matrix is divided into upper 16 rows and lower 16 rows. Only two rows can be driven at the same time, namely rows `A` and `A+16`, where `A` is the address input `A[3:0]`. To configure a row, we shift data into the 3 shift registers for three bits of color: `R0,G0,B0` for row `A` and `R1,G1,B1` for row `A+16`. Then, we use the `blank` and `latch` signals to send the data from the shift registers to the LED drivers and then display them. This results in a 1/16 duty cycle row multiplexed display.

A *specific sequence* of shifting data and toggling the `blank` and `latch` signals is required to produce desired behavior in the LED matrix. Figuring out this sequence was by far the most difficult part of operating the LED matrix. Our working understand of the sequence is as follows:

1) While rows `A` and `A+16` are being displayed, shift the data for the next row into the registers through the `R0,G0,B0,R1,G1,B1,sclk,` and `A[3:0]` input pins. This is going to be 32 `sclk` cycles and make sure to change the RGB pin values on the neg clock edge between `sclk` posedges.
2) Assert `blank`. This will blank the display.
3) Assert and the de-assert `latch`.
4) Update `A[3:0]` to the address of the rows you want to display next.
5) De-assert `blank.`
6) Wait for a certain duration for the LED row to shine with the programmed pattern. Then repeat step 1).

We encountered many pitfalls in this process. First, we assumed that we could repeatedly flash the same address, as this would be necessary for binary coded modulation. However, when we did so, somehow the row address would show the desired pattern for one cycle and then go black or even not show anything at all. This was fixed when we flashed different addresses in successive sequences. Second, we assumed that we should update `A[3:0]` before toggling `latch` because we thought `latch` sends data from the shift registers to the LED drivers. We did so before step 1 so that we could also use the `A` variable to make assignment bits to the rgb pins. However, this resulted in "doubled" matrix patterns that only were fixed when we strictly followed the Glen Alkin tutorial and updated the `A` at step 4.

LIS3DH Triple-Axis Accelerometer

For this project, we also introduce a LIS3DH accelerometer. This accelerometer has a wide range of configuration options from 8-bit to 12-bit resolution, ±2g/±4g/±8g/±16g selectable scaling, and multiple data rate options and operates at 3.3 V. This device can be interfaced with over I^2C or SPI. We found that this is a much more established device and has much better documentation than the LED matrix. Adafruit provides a good tutorial on its setup[3] and the datasheet explains its operation well[4]. In the context of this project, we configured the accelerometer to read the acceleration in two axes with 12-bit readings at a data rate of 400 Hz. Setting up the accelerometer over SPI with this configuration involved writing to two of the device's control registers and then the accelerometer readings were read directly from the output registers on the board.

Schematic

Final Schematic

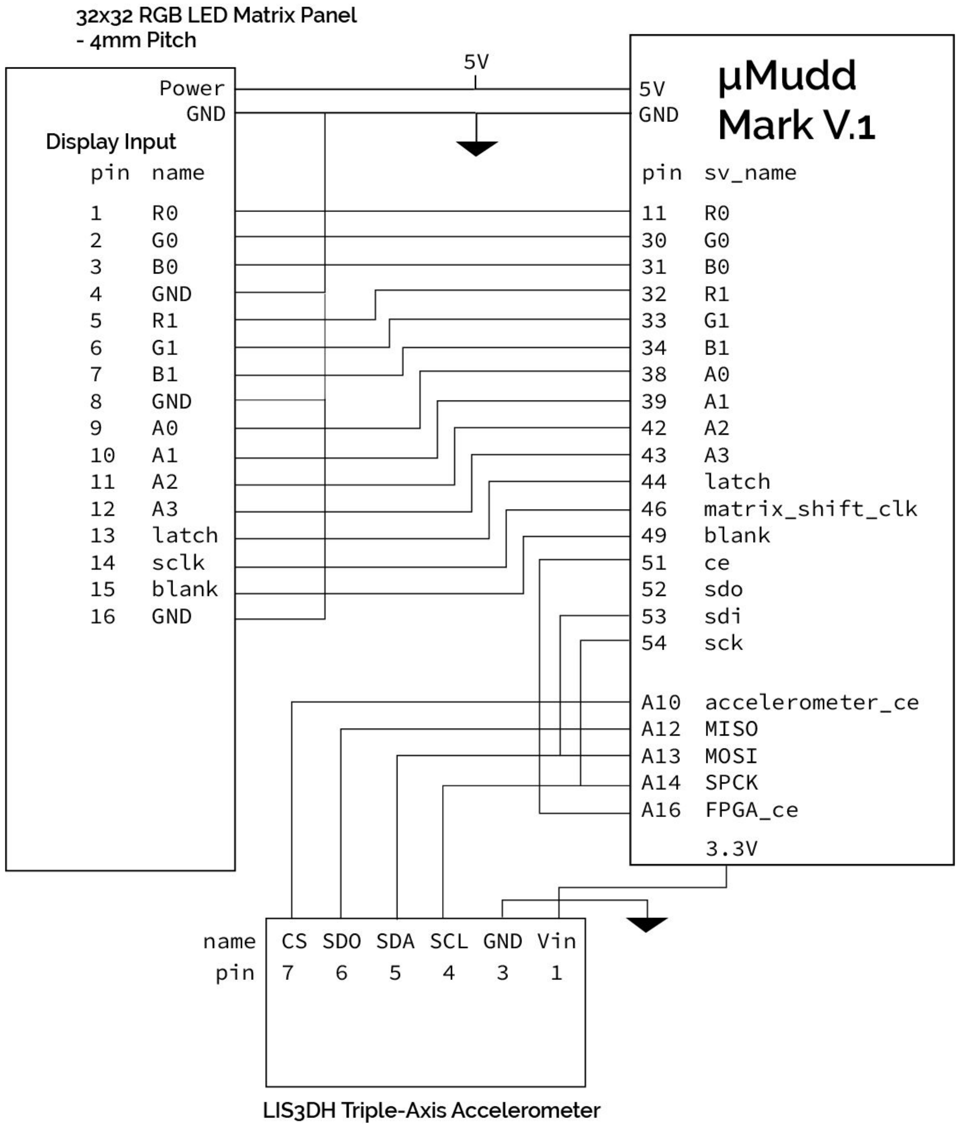

Figure 2. A schematic displaying the circuity between the components of the digital maze game.

FPGA System

The FPGA has three main functions in this project: driving the LED matrix, assigning colors to the different components of the game, and receiving and storing game state data from the ATSAM microcontroller over SPI.

Driving the LED Matrix

To drive the LED Matrix, we implement the input sequence described above using the FPGA. Our state logic consists of a 7-bit counter `col`, the 3-bitrow address and output `A`, and state variable `state`. To shift data into the matrix in a rate that it can handle, we slowed down the clock speed for the matrix display module to 1/128 of the global clock, i.e. about 156kHz. To ensure that the matrix receives stable inputs on the posedge of the slow clock, we update state values and outputs on the negedge. The counter `col` is incremented every cycle. The general FSM is shown below (Figure 3).

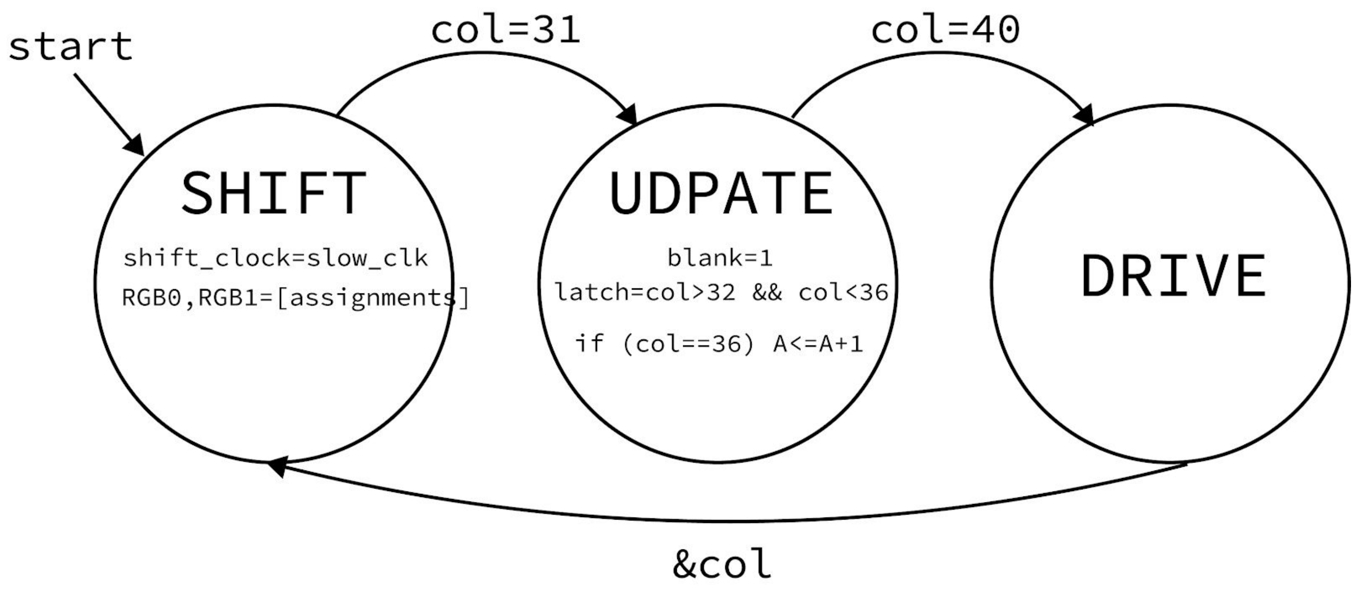

Figure 3. The finite state machine that updates the LED matrix's rows by shifting in RGB values and driving them

Note that after latch is de-asserted, while still in the `UPDATE` state, we update the `A` output to the row we want to display (`A+1`). In the `DRIVE` state, no outputs are asserted and we simply wait until the counter overflows to return to `SHIFT` and begin configuring the next address.

Color Assignment

We assign all wall pixels to blue. To do so, we read the two rows of the maze to drive from the memory module using the `nextA=A+1` and then specific cells in those rows with the `col` counter. Why `A+1`? Recall that we can only update the row address to the desired address after unlatching. Thus, when we are shifting in the next rows data, `A` is one less than the address we want to display next.

We assign the bead to green in the first 65 seconds of the game and red in the last 15 seconds. We do so by setting both the red and green bits high when the `nextA` and `col` corresponds to the bead position. However, for the green bit, there is an additional constraint that the countdown variable is greater than or equal to 3. The red bit requires that the countdown variable is less than 3. Both the bead position and the countdown variable are rows in memory, updatable by the microcontroller.

SPI Interface and Memory

The SPI interface on the FPGA side was implemented similarly to some of the other SPI slave devices we have used in this class such as the digital temperature sensor in lab 6 and much like the accelerometer used in this project where the master device reads and writes directly to the slave's memory. This was made simpler though because the FPGA's main purpose is to drive the LED matrix, therefore we did not have to implement reading from memory for the purpose of outputting the data over SPI. The protocol we implemented consisted of sending 5 bytes from the ATSAM to the FPGA. The first byte holds the address of the row in memory that will be written to (although our memory is small enough to be encoded in 6 bits), and the following 4 bytes make up a word of data that will replace the data in that row of memory. Implementing the SPI interface this way decreased the complexity of the FPGA circuitry and generalized updating any visual element or values important for controlling the display.

In our memory, three components of the game are stored. The first is the maze itself, taking up the first 32 rows in memory. Each wall of the maze is encoded as a single bit value determining if the wall exists in that position in the LED matrix. A row in memory represents the row on the board and the most significant bit in a row is the left most column on the matrix. Following these 32 words, the 32nd address is dedicated to the row and column value of the bead and the final word contains the countdown timer. We had originally intended for our memory to be made up of word aligned M9K units in the FPGA, however the final implementation of the memory model involved making asynchronous reads in order to read two maze rows at a time, the bead, and the game countdown timer. As it was pointed out to us, this implied that all of these values were stored in logic elements (LE) instead of M9K units. This was not a problem however because we still had plenty of LEs free on the FPGA and asynchronous reads saved us from adding more complexity to driving the LED matrix.

Microcontroller System

All game state logic and bead physics were implemented in the ATSAM microcontroller and then sent to the FPGA to be displayed. The position of the bead is also determined by accelerometer readings, so the microcontroller interfaces with the accelerometer to calculate the next game state.

Game Logic

The ATSAM waits 2 seconds after being turned on, during which it initializes the peripherals and accelerometer and also allows the user to admire the beautiful start screen. The µMudd board DIP switch values are read to store a hard or easy mode configuration, which will affect how the bead position is later updated. The ATSAM then takes an initial accelerometer reading to calibrate to an initial “flat” level and begins the game by sending the maze and the beginning bead position to the FPGA. Until the game ends, the ATSAM routinely samples the accelerometer and uses the results to update the bead position with bead physics scheme.

The total duration of the game is 80 seconds. After every sending the update bead position, the ATSAM also sends a countdown number to the FPGA. This is the number of seconds remaining in the game divided by 5, hence starting at 15 (=79.9s divided by 5 and then cast to int) and ending on 0. This number is used by the FPGA to alter the bead’s color from green to red when there are only 15 seconds left. Notice that a simple one bit flag would also suffice for this functionality. The reason we chose to send over a 4 bit counter is that we originally intended to use the counter to change the color of the maze walls using 4 bit binary coded modulation.

Finally, after the countdown is sent, the ASTAM checks if the bead position is at the exit position of the maze. If so, it sends the start screen to the FPGA and ends execution. If the game time expires before the user navigates out of the maze, the ATSAM writes a GAME OVER screen to the FPGA and ends execution. If the user wishes to play the game again, the game can be restarted using the ATSAM reset button on the µMudd board to start this game state logic over again.

Bead Physics

The ATSAM keeps track of the position and velocity of the bead. The accelerometer readings determine how the bead’s velocity vector changes with each sample. The velocity vector is filtered through collision logic, before it is used to update the bead’s position. We shall describe this process in greater detail below:

Acceleration: To start off a cycle of our bead physics, we first read 12 bits of acceleration data from the accelerometer on the x and y axis. This is received by the ATSAM in 2 bytes as a left-justified two's complement value. This is then casted to a signed short and shifted to remove the unused bits. From this value, velocity is then calculated in its x and y components, scaling to best simulate gravity, and multiplying by the duration elapsed between each accelerometer reading (a duration which we have set using a delay at the beginning of each loop).

Velocity: The ATSAM uses the new velocity to find the next position of the bead. To do so, it first zeros components of the velocity vector that point to a wall. Next, it selects a direction to move the bead in. This will be that of the non-zero velocity component. If there are two non-zero velocity components, it picks the direction with the larger velocity or acceleration, depending on whether the hard or easy mode is set, respectively. The other velocity component is zeroed.

Position: Finally, to update the bead position without allowing it to pass through walls, the ASTAM linearly scans the matrix between the old position and the potential new position (old pos + time passed * velocity) and sets the bead right before the first wall, if one exists. If no walls are detected in the scan, the bead is simply updated to the new position.

Spi Interfaces

As mentioned previously, the ATSAM communicates to both the FPGA and accelerometer over SPI. This was made fairly simple with the library given to us. These slave devices share common MISO, MOSI, and SPCK buses, however they are selected by different chip enable buses. It would have been interesting to put these on the same chip enable because the accelerometer is an active low device while the FPGA's SPI interface is implemented as an active high, however pins were not a limited resource two us, and so it was much simpler to put them on separate pins.

Results

Our project was successful. We were able to create a maze game that produces all the behavior we set out to create. It is also a portable handheld device, thanks to cardboard engineering and a battery pack. We received great feedback and we are really happy with how it turned out.

While we delivered on all our promises, we had hoped to vary the colors of the walls using binary coded modulation. However, to do so, we would need to flash the same address multiple times in a row, which we aren't sure is possible given our previous unsuccessful attempts despite producing expected waveforms. Thus, we believe that accomplishing this would require a leap in our understanding of how the matrix works. Also, binary coded modulation would require speeding up the matrix display module clock. The current clock frequency and counter width leads to a 76Hz matrix refresh rate. To do binary coded modulation, we need to dwarf the time shifting in data with the time driving the display (currently a 1:3 ratio). Thus, we need a much wider counter. To maintain the refresh rate with a wider counter, we need a faster clock. Unfortunately, initial attempts at increasing the matrix clock frequency resulted in severely corrupted display patterns. We believe that much more time and experimentation are necessary for this undertaking.

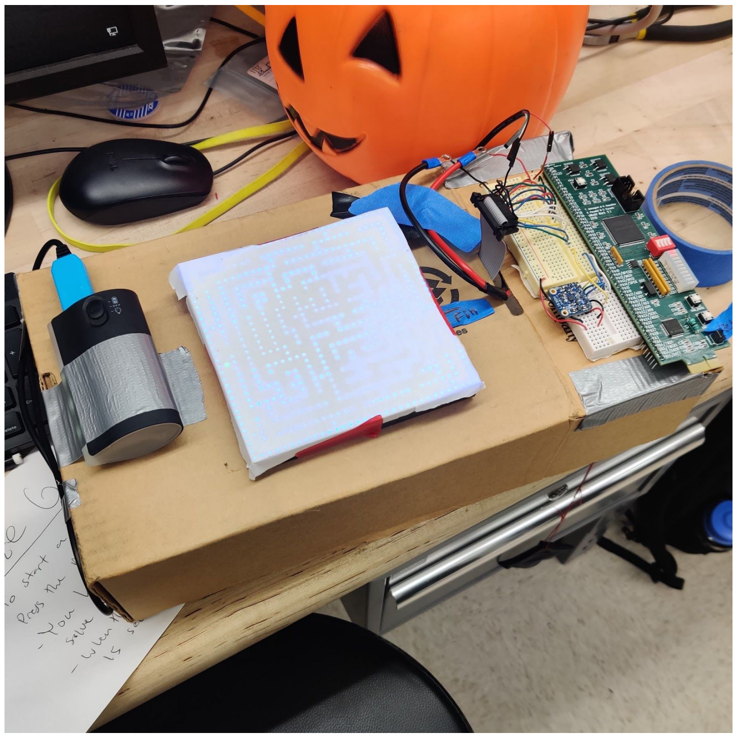

Figure 4. Our final project’s form factor: The components are assembled on a cardboard box and can be powered by a portable charger.

References

[1] Adkins, Glen. “RGB LED Panel Driver Tutorial.” RGB LED Panel Driver Tutorial, 2014, https://bikerglen.com/projects/lighting/led-panel-1up/.
[2] Ray's Logic. “Adafruit RGB LED matrix”, Adafruit RGB LED matrix, http://rayslogic.com/propeller/Programming/AdafruitRGB/AdafruitRGB.htm.
[3] Adafruit, “Adafruit LIS3DH Triple-Axis Accelerometer Breakout”, Adafruit LIS3DH Triple-Axis Accelerometer Breakout, 2015, https://learn.adafruit.com/adafruit-lis3dh-triple-axis-accelerometer-breakout.
[4] STMicroelectronics, “MEMS digital output motion sensor: ultra-low-power high performance 3-axes ‘nano’ accelerometer”, LIS3DH datasheet, Dec. 2016, https://www.st.com/resource/en/datasheet/lis3dh.pdf.

Part List

Part	**Source**	**Vendor Part #**	**Price**
32x32 RGB LED Matrix Panel - 4mm Pitch	Adafruit	PRODUCT ID: 607	$49.95
LIS3DH Triple-Axis Accelerometer	Adafruit	PRODUCT ID: 2809	$4.95

Appendix A: ATSAM C

final_project.c

```c
/* final_project.c
*
* dsobek@g.hmc.edu
* jyliang@g.hmc.edu
* 11/20/2019
*
* Implements a maze game's state using a 32 by 32 LED matrix and accelerometer. */

#include "SAM4S4B.h"
#include "maze.h"
#include "bead.h"
#include <stdint.h>

// Pin assignments
#define ACCELEROMETER_CE_PIN PIO_PA10
#define FPGA_CE_PIN PIO_PA16
#define MODE_SWITCH_PIN PIO_PB0

// Addresses on the LIS3DH accelerometer
#define LIS3DH_CTRL_REG0  0x1E
#define LIS3DH_CTRL_REG1  0x20
#define LIS3DH_CTRL_REG4  0x23
#define LIS3DH_OUT_X_L 0x28
#define LIS3DH_OUT_Y_L 0x2A
#define LIS3DH_OUT_Z_L 0x2C

#define COUNTDOWN_DURATION 80 // seconds

#define CALIBRATE 1 // Boolean to determine if we should calibrate at startup

// read a byte from the accelerometer
uint8_t accelerometerRead(uint8_t addr) {
   pioDigitalWrite(ACCELEROMETER_CE_PIN, PIO_LOW); // msb set high to specify a read
   uint16_t val = spiSendReceive16((0x80 | addr) << 8);
   pioDigitalWrite(ACCELEROMETER_CE_PIN, PIO_HIGH);
   return val;
}

// read two bytes from the accelerometer
uint16_t accelerometerReadTwoBytes(uint8_t addr) {
   pioDigitalWrite(ACCELEROMETER_CE_PIN, PIO_LOW);
   // msb set high to specify a read and 2nd msd to get multiple bytes
   // (incremented address)
   uint16_t val = spiSendReceive16((0xc0 | addr) << 8) & 0x00FF;
   val = val | (spiSendReceive(0x00) << 8);
   pioDigitalWrite(ACCELEROMETER_CE_PIN, PIO_HIGH);
   return val;
}

// write a byte to the accelerometer
void accelerometerWrite(uint8_t addr, uint8_t data) {
   pioDigitalWrite(ACCELEROMETER_CE_PIN, PIO_LOW);
   spiSendReceive16((addr << 8) | data);
   pioDigitalWrite(ACCELEROMETER_CE_PIN, PIO_HIGH);
}

void accelerometerInit(void) {
   accelerometerWrite(LIS3DH_CTRL_REG1, 0x73); // 400Hz
   accelerometerWrite(LIS3DH_CTRL_REG4, 0x08); // 12 bits resolution
   // uint8_t who_am_i = accelerometerRead(0x0F); // 0x33 expected
```

```c
}

void sendMaze(Maze *m) {
   for (int i = 0; i < 32; i++) {
       pioDigitalWrite(FPGA_CE_PIN, PIO_HIGH);
       spiSendReceive(i);
       spiSendReceive16((m->rows[i] >> 16) & 0x0000FFFF);
       spiSendReceive16(m->rows[i] & 0x0000FFFF);
       pioDigitalWrite(FPGA_CE_PIN, PIO_LOW);
   }
}

void sendBeadPosition() {
   // y is row, x is col
   int x = (int) bead.pos[0];
   int y = (int) bead.pos[1];
   uint16_t bead_pos = ((y & 0x1F) << 5) | ((x & 0x1F));
   pioDigitalWrite(FPGA_CE_PIN, PIO_HIGH);
   spiSendReceive(32);
   spiSendReceive16(0);
   spiSendReceive16(bead_pos);
   pioDigitalWrite(FPGA_CE_PIN, PIO_LOW);
}

void sendRemainingTime(int counter) {
   // y is row, x is col
   uint16_t count_down = ((counter) & 0xF);
   pioDigitalWrite(FPGA_CE_PIN, PIO_HIGH);
   spiSendReceive(33);
   spiSendReceive16(0);
   spiSendReceive16(count_down);
   pioDigitalWrite(FPGA_CE_PIN, PIO_LOW);
}

// update X, and Y accelerometer values
void getXY(int16_t *x_val, int16_t *y_val) {
   // Flipped because of the orientation of the accelerometer
   *x_val = ((int16_t) accelerometerReadTwoBytes(LIS3DH_OUT_Y_L)) >> 4;
   *y_val = ((int16_t) accelerometerReadTwoBytes(LIS3DH_OUT_X_L)) >> 4;
}

// Reads the state of the 0th dip switch and sets the game mode,
void setGameMode(void) {
   bead.mode = pioDigitalRead(MODE_SWITCH_PIN);
}

void game(void) {
   // initialize maze
   sendMaze(init_maze());
   init_bead(maze.begin_pos[0], maze.begin_pos[1]);
   sendBeadPosition();

   int16_t x, y;
   int remaining_time = COUNTDOWN_DURATION * 1000;
   int delta_t = 1000 / SAMPLE_FREQ;

   // Set to easy or hard mode and calibrate the accelerometer.
   setGameMode();
   if (CALIBRATE) {
       getXY(&x, &y);
       calibrate_bead(x, y);
   }

   while(remaining_time > 0) {
       // send remaining time to word in FPGA
```

```
        tcDelayMillis(delta_t);
        remaining_time -= delta_t;
        getXY(&x, &y);
        update_bead_velocity(x, y);
        update_bead_position(x, y);
        sendBeadPosition();
        sendRemainingTime(remaining_time / 1000 / 5); // transforms remaining time to 0 to 15
scale

        // if we reached final position
        if (is_final_pos((int) bead.pos[0], (int) bead.pos[1])) {
            sendMaze(start_screen());
            init_bead(maze.begin_pos[0], maze.begin_pos[1]);
            sendBeadPosition();
            return;
        }
    }
    // send game over screen
    sendMaze(game_over());
    init_bead(maze.begin_pos[0], maze.begin_pos[1]);
    sendBeadPosition();
}

int main(void) {
    // Initialize ATSAM peripherals
    samInit();
    pioInit();
    tcDelayInit();
    spiInit(MCK_FREQ/244000, 1, 0);
    // "clock divide" = master clock frequency / desired baud rate
    // the phase for the SPI clock is 1 and the polarity is 0

    // ce pin setup
    pioPinMode(ACCELEROMETER_CE_PIN, PIO_OUTPUT);
    pioDigitalWrite(ACCELEROMETER_CE_PIN, PIO_HIGH);
    pioPinMode(FPGA_CE_PIN, PIO_OUTPUT);
    pioDigitalWrite(FPGA_CE_PIN, PIO_LOW);

    // Dip switch setup
    pioPinMode(MODE_SWITCH_PIN, PIO_INPUT);

    // Hardcoded in the fpga but send when atsam is reset
    sendMaze(start_screen());
    init_bead(maze.begin_pos[0], maze.begin_pos[1]);
    sendBeadPosition();

    // Let the startup screen be shown for a couple seconds
    tcDelaySeconds(1);
    // Accelerometer Setup
    accelerometerInit();
    tcDelaySeconds(1);

    // Game logic
    game();

    return 0;
}
```

maze.h

```c
/* maze.h
*
* dsobek@g.hmc.edu
* jyliang@g.hmc.edu
* 11/20/2019
*
* Contains the maze, collision logic, and several other screens to
* display. */

#ifndef MAZE_H
#define MAZE_H

#include <stdint.h>

typedef struct {
   uint32_t rows[32];
   int8_t begin_pos[2];
   int8_t end_pos[2];
} Maze;

Maze maze;

Maze* init_maze(void) {
   maze.rows[0] = 0xFFFFFFFF;
   maze.rows[1] = 0x84000401;
   maze.rows[2] = 0xBFDFDFBD;
   maze.rows[3] = 0xA0081005;
   maze.rows[4] = 0x8FFBFFF5;
   maze.rows[5] = 0xA8100115;
   maze.rows[6] = 0xABF7DFD5;
   maze.rows[7] = 0xAA144047;
   maze.rows[8] = 0xAADDFFDD;
   maze.rows[9] = 0xAAD11451;
   maze.rows[10] = 0xA8945155;
   maze.rows[11] = 0xEAB55755;
   maze.rows[12] = 0xAAA74115;
   maze.rows[13] = 0xAAADEF55;
   maze.rows[14] = 0xAAA40255;
   maze.rows[15] = 0x8AB7EFF5;
   maze.rows[16] = 0xAA904115;
   maze.rows[17] = 0xABD77F54;
   maze.rows[18] = 0xA8440457;
   maze.rows[19] = 0xABF5FDD5;
   maze.rows[20] = 0xAA150551;
   maze.rows[21] = 0xA8D55D57;
   maze.rows[22] = 0xAE544145;
   maze.rows[23] = 0xA2DDFF75;
   maze.rows[24] = 0xBE544055;
   maze.rows[25] = 0x8BD5EFD1;
   maze.rows[26] = 0xA8040115;
   maze.rows[27] = 0xAFF7FF75;
   maze.rows[28] = 0xE0001015;
   maze.rows[29] = 0xAFFEFEFD;
   maze.rows[30] = 0x80100009;
   maze.rows[31] = 0xFFFFFFFF;

   maze.begin_pos[0] = 1;
   maze.begin_pos[1] = 17;
   maze.end_pos[0] = 31;
   maze.end_pos[1] = 17;

   return &maze;
}
```

```
// Returns 1 if the given coordinates is a wall or outside the maze.
int is_wall(int x, int y) {
   return x > 31 || x < 0 || y > 31 || y < 0 ||
          (((maze.rows[y] >> (31 - x)) & 1) == 1);
}

// Returns 1 if the given coordinates is in the winning position.
int is_final_pos(int x, int y) {
   return x == maze.end_pos[0] && y == maze.end_pos[1];
}

Maze* start_screen(void) {
   maze.rows[0] = 0x0;
   maze.rows[1] = 0x0;
   maze.rows[2] = 0x00186000;
   maze.rows[3] = 0x001ce000;
   maze.rows[4] = 0x0014a000;
   maze.rows[5] = 0x0017a000;
   maze.rows[6] = 0x00132000;
   maze.rows[7] = 0x00100000;
   maze.rows[8] = 0x00100000;
   maze.rows[9] = 0x0;
   maze.rows[10] = 0x0;
   maze.rows[11] = 0x18c47df0;
   maze.rows[12] = 0x1dce0500;
   maze.rows[13] = 0x175b0900;
   maze.rows[14] = 0x125111f0;
   maze.rows[15] = 0x105f2100;
   maze.rows[16] = 0x10514100;
   maze.rows[17] = 0x10517df0;
   maze.rows[18] = 0x0;
   maze.rows[19] = 0x0;
   maze.rows[20] = 0x0;
   maze.rows[21] = 0x0;
   maze.rows[22] = 0x0;
   maze.rows[23] = 0x02800670;
   maze.rows[24] = 0x02800540;
   maze.rows[25] = 0x02800570;
   maze.rows[26] = 0x02800510;
   maze.rows[27] = 0x0ee00670;
   maze.rows[28] = 0x0;
   maze.rows[29] = 0x0;
   maze.rows[30] = 0x0;
   maze.rows[31] = 0x0;

   maze.begin_pos[0] = 13;
   maze.begin_pos[1] = 14;

   return &maze;
}

Maze* game_over(void) {
   maze.rows[0] = 0x0;
   maze.rows[1] = 0x0;
   maze.rows[2] = 0x0;
   maze.rows[3] = 0x0;
   maze.rows[4] = 0x0;
   maze.rows[5] = 0x7E38447E;
   maze.rows[6] = 0xFE7CEEFE;
   maze.rows[7] = 0xC0EEEEC0;
   maze.rows[8] = 0xC0C6FEC0;
   maze.rows[9] = 0xDEC6FEFC;
   maze.rows[10] = 0xCEC6FEFC;
   maze.rows[11] = 0xC6FED6C0;
   maze.rows[12] = 0xC6EEC6C0;
   maze.rows[13] = 0xFEC6C6FE;
   maze.rows[14] = 0x7CC6C67E;
   maze.rows[15] = 0x0;
   maze.rows[16] = 0x0;
   maze.rows[17] = 0x7CC67E7C;
   maze.rows[18] = 0xFEC6FEFE;
   maze.rows[19] = 0xFEC6C0FE;
   maze.rows[20] = 0xC6C6C0C6;
   maze.rows[21] = 0xC6C6FCC6;
   maze.rows[22] = 0xC6C6FCC6;
   maze.rows[23] = 0xC6C6C0FC;
   maze.rows[24] = 0xC6EEC0F8;
   maze.rows[25] = 0xFE7CFEDC;
   maze.rows[26] = 0x7C387ECE;
   maze.rows[27] = 0x0;
   maze.rows[28] = 0x0;
   maze.rows[29] = 0x0;
   maze.rows[30] = 0x0;
   maze.rows[31] = 0x0;

   maze.begin_pos[0] = 11;
   maze.begin_pos[1] = 8;

   return &maze;
}

#endif // MAZE_H
```

bead.h

```
/* bead.h
*
* dsobek@g.hmc.edu
* jyliang@g.hmc.edu
* 11/20/2019
*
* Contains the bead that the user will control with the accelerometer. This
* file contains the logic for updating the bead position in the board and
* detecting collisions with the maze wall. */

#ifndef BEAD_H
#define BEAD_H

#include "maze.h"
#include <stdint.h>
#include <math.h>

#define GRAVITY 1 // m/s
#define SAMPLE_FREQ 10  // Frequency of accelerometer reads in Hz.
#define CELL_DISTANCE 0.004 // The distance between LEDs in the matrix in meters.
#define MAX_RAW_ACCEL 0x07FF  // The max value the the accelerometer will return.
                               // Dependent on bytes of resolution in accelerometer.

// Kinematic data for a bead
// pos: x and y position in LED matrix (in units of LEDs)
// vel: x and y velocity in LED matrix (in units of m/s)
// accel_base: raw accelerometer data for calibrating.
typedef struct {
   float pos[2];
   float vel[2];
   int16_t accel_base[2];
   int mode; // 0 is easy, 1 is hard.
} Bead;

Bead bead;

// Initializes the bead with the starting position.
void init_bead(int x, int y) {
   bead.pos[0] = (float) x;
   bead.pos[1] = (float) y;
   bead.vel[0] = 0;
   bead.vel[1] = 0;
   bead.accel_base[0] = 0;
   bead.accel_base[1] = 0;
   bead.mode = 0;
}

// Set the calibration data.
void calibrate_bead(int16_t x_accel, int16_t y_accel) {
   bead.accel_base[0] = x_accel;
   bead.accel_base[1] = y_accel;
}

/**
* Takes in acceleration sensor data, transforms it, and then
* updates the velocity vector of the the bead.
*/
void update_bead_velocity(int16_t x_accel, int16_t y_accel) {
   float accel_vec[2];
   float t = 1 / (float) SAMPLE_FREQ; // how long it has been since the last accelerometer
read.

   accel_vec[0] = (float) (bead.accel_base[0]-x_accel) * GRAVITY / MAX_RAW_ACCEL;
   accel_vec[1] = (float) (bead.accel_base[1]-y_accel) * GRAVITY / MAX_RAW_ACCEL;

   // Update velocity vector v = v0 + a * t
   bead.vel[0] = bead.vel[0] + (accel_vec[0] * t);
   bead.vel[1] = bead.vel[1] + (accel_vec[1] * t);
}

/**
* Takes in the new velocity vector of the bead. Sets the velocity of
* immobile directions of the bead to zero. Updates the position with the
* velocity of the free direction.
*
* If there are multiple free directions, we chose the dominant direction
* with the greater acceleration.
*/
void update_bead_position(int16_t x_accel, int16_t y_accel) {
   int old_x = (int) bead.pos[0];
   int old_y = (int) bead.pos[1];

   // zero_out velocity direction that can't move
   if (is_wall(old_x + (bead.vel[0] > 0 ? 1 : -1), old_y)) {
       bead.pos[0] = floor(bead.pos[0]);
       bead.vel[0] = 0;
   }
   if (is_wall(old_x, old_y + (bead.vel[1] > 0 ? 1 : -1))) {
       bead.pos[0] = floor(bead.pos[0]);
       bead.vel[1] = 0;
   }

   // choose direction for bead to move
   int direction = bead.vel[0] == 0; // 0 if vel[0] is non-zero, 1 if vel[0] is 0
   // if both directions are free, choose direction with more acceleration
   if (bead.vel[0] != 0 && bead.vel[1] != 0) {
       if (bead.mode) { // HARD mode
           direction = abs(bead.vel[0]) < abs(bead.vel[1]); // 0 if x is larger, 1 if y is
larger
       }
       else { // EASY mode
           direction = abs(bead.accel_base[0]-x_accel) < abs(bead.accel_base[1]-y_accel);
       }

       bead.vel[!direction] = 0;
   }
   if (bead.vel[0] == 0 && bead.vel[1] == 0) {
       return;
   }

   float t = 1.0/SAMPLE_FREQ;
   float old_pos = bead.pos[direction];
   float new_pos = bead.pos[direction] + (bead.vel[direction] * t) / CELL_DISTANCE;

   // loop to make bead stop before wall
   // assumes that we start at a non_wall position
   for (
           int i = old_pos;
           (bead.vel[direction] > 0 && i < new_pos) || (bead.vel[direction] < 0 && i >= (int)
new_pos);
           i += (bead.vel[direction] > 0 ? 1 : -1)
       ) {
       // if we reach a wall, then stop at previous position
       if (
           (direction == 0 && is_wall(i, (int) bead.pos[1])) ||
           (direction == 1 && is_wall((int) bead.pos[0], i))
       ) {
           bead.pos[direction] = (float) i + (old_pos < new_pos ? -1.0 : 1.0);
```

```
            bead.vel[direction] = 0;
            return;
        }
    }

    // if no wall is crossed, simply update position to new_pos
    bead.pos[direction] = new_pos;
}

#endif // BEAD_H
```

Appendix B: FPGA SystemVerilog

fpga.sv

```
module testbench();
   logic clk, reset;
   /** LED Matrix I/O **/
   logic latch, blank, R0, G0, B0, R1, G1, B1, A0, A1, A2, A3, matrix_shift_clk;

   fpga dut(clk, reset, latch, blank,
           R0, G0, B0, R1, G1, B1, A0, A1, A2, A3,
           matrix_shift_clk);

   // initialize test
   initial
       begin
       reset <= 1; # 22; reset <= 0;
       end
   // generate clock to sequence tests
   always
       begin
           clk <= 1; # 5; clk <= 0; # 5;
       end

endmodule

/////////////////////////////////////////////
// fpga
//   top level module for the fpga size of the maze game
//
/////////////////////////////////////////////

module fpga(input  logic clk, reset,
           /** SPI I/O **/
           input  logic ce,
           input  logic sck,
           input  logic sdi,
           output logic sdo,
           /** LED Matrix I/O **/
           output logic latch, blank,
           output logic R0, G0, B0, R1, G1, B1, // RGB Values
           output logic A0, A1, A2, A3,         // Row Address
           output logic matrix_shift_clk);            // Shift Register Clock

   // Busses for expanding display module output
   logic [2:0]        RGB0, RGB1; // Two pixel values
   logic [3:0]        A; // The Address

   // Busses for outputting memory to display module
   logic [31:0] row0, row1;
   logic [15:0] bead;

   // Busses for updating memory
   logic [31:0] data, wd;
   logic [5:0]  addr, wa;
   logic        we;
   logic [3:0]  count_down;

   // SPI I/O
   spi io(ce, sck, sdi, sdo, addr, data);

   // Update RAM with maze and update bead location
   update_mem_control mem_control(clk, reset, ce, addr, data, wa, wd, we);
   // Memory Module
   mem m(clk, reset,
         we, wa, wd,
         A + 1,
         row0, row1, bead,
         count_down);

   // Matrix Flashing Module
   matrix display(clk, reset,
                  row0, row1,
                  count_down,
                  bead,
                  latch, blank, RGB0, RGB1, A, matrix_shift_clk);

   // Output decomposition for wiring clarity
   assign {R0, G0, B0} = RGB0;
   assign {R1, G1, B1} = RGB1;
   assign {A3, A2, A1, A0} = A;

endmodule

/////////////////////////////////////////////
// matrix
//   module for flashing the led matrix.
//
//   Flash row A and A + 16 at the same time. First shift in data then
//   blank, latch, change address to desired address, unblank
//
//   Takes in the slow clock. Takes 128 cycles to shift data into 2
//   rows of the matrix and then display the rows. As there are 32 rows
//   to flash, this gives us a 1/16 duty cycle for the display.
//
//   Thus, the display is refreshed every 16 * 128 = 2048 slow clock
//   cycles, which is a frequency about 76Hz.
//
/////////////////////////////////////////////

module matrix(input  logic  clk, reset,
             input  logic  [31:0] row0, row1,
             input  logic  [3:0]  count_down,
             input  logic  [9:0]  bead,
             output logic latch,  blank,
             output logic  [2:0]  RGB0, RGB1,
             output logic  [3:0]  A,
             output logic  shift_clock);
   typedef enum logic[1:0] {SHIFT, UPDATE, DRIVE} statetype;
   statetype state, nextstate;

   logic [6:0] col, nextcol;
   logic slow_clock;
   logic [3:0] nextA;

   slow_clock sc(clk, reset, slow_clock);

   // always_ff @(negedge clk, posedge reset) begin
   always_ff @(negedge slow_clock, posedge reset) begin
       if (reset) begin
           state = SHIFT;
           col = 0;
           A = 15;
       end else begin
           state <= nextstate;
           col <= nextcol;
           if (col == 36) A <= nextA;
           else A <= A;
```

```
        end
    end

    // Next State logic
    always_comb
        case (state)
            SHIFT: if (col == 31) nextstate = UPDATE;
                   else nextstate = SHIFT;
            UPDATE: if (col == 40) nextstate = DRIVE;
                   else nextstate = UPDATE;
            DRIVE: if (&col) nextstate = SHIFT;
                   else nextstate = DRIVE;
            default: nextstate = SHIFT;
        endcase

    // always increment col / counter
    always_comb
        if (reset) nextcol = 1;
        else nextcol = col + 1;

    assign nextA = A + 1;

    assign RGB0 = {state == SHIFT && bead[9:5] == nextA && bead[4:0] == col && count_down < 3,
                   state == SHIFT && bead[9:5] == nextA && bead[4:0] == col && count_down >=
3,
                   state == SHIFT && row0[31 - col]};
    assign RGB1 = {state == SHIFT && bead[9:5] == nextA + 16 && bead[4:0] == col && count_down
< 3,
                   state == SHIFT && bead[9:5] == nextA + 16 && bead[4:0] == col && count_down
>= 3,,
                   state == SHIFT && row1[31 - col]};

    assign blank = (state == UPDATE);
    assign latch = (state == UPDATE && col > 32 && col < 36);
    // assign shift_clock = (state == SHIFT && clk);
    assign shift_clock = (state == SHIFT && slow_clock);

endmodule

/////////////////////////////////////////////
// slow_clock
//   Slows down the clock for the LED matrix display
//   logic. FPGA original frequency is 40MHz. This slows
//   it down by a factor of 128 to 156.25kHz.
//
/////////////////////////////////////////////
module slow_clock(input  logic clk, reset,
                  output logic slow_clock);
    logic [32-1:0] q, nextq;
    always_ff @(posedge clk)
        if (reset) q <= 0;
        else q <= nextq;
    assign nextq = q + 1;
    assign slow_clock = (q[7] == 1);
endmodule


/////////////////////////////////////////////
// mem
//   Memory module that stores the latest wall pixel
//   positions of the maze and the bead position
//   in the maze.
//
//   32 words for the maze state    (0x00 - 0x0F)
//   1 word for the bead state      (0x20)
//   unused                         (0x21 - 0xFF)
//
//////////////////////////////////////////////
module mem(
          input  logic        clk, reset,
          input  logic        we, // write enable
          input  logic [5:0]  wa, // write address
          input  logic [31:0] wd, // write data
          input  logic [5:0]  ad,
          output logic [31:0] row0,
          output logic [31:0] row1,
          output logic [9:0]  bead,
          output logic [3:0]  count_down);

   logic [31:0] MAZE[33:0]; // last address is the bead (only uses the 10 LSBs, 5 for row and
5 for col)

   initial
     $readmemb("startup_screen.dat", MAZE);

   // { unused bits, ROW, COLUMN } NOTE: both start from 0!
   // assign bead = {6'b000000, 5'b10001, 5'b00001}; // initialize bead at starting position

   assign row0 = MAZE[ad[3:0]]; // maze address space limited to 0x00-0x1F
   assign row1 = MAZE[ad[3:0] + 16];
   assign bead = MAZE[32][9:0];
   assign count_down = MAZE[33][3:0];

   always_ff @(posedge clk)
       if (we) MAZE[wa] <= wd;
endmodule



//////////////////////////////////////////////
// spi
//   SPI interface.  Copied from lab 7.
//   Only 40 bits are received at a time:
//      2 unused bits + 6 address bits + 32 bits for data
//
//////////////////////////////////////////////
module spi(input  logic ce,
          input  logic sck,
          input  logic sdi,
          output logic sdo,
          output logic [5:0] addr,
          output logic [31:0] data);

   // 38 bits in shift register: 6 for address, 32 for data
   always_ff @(posedge sck)
       if (ce)        {addr, data} = {addr[4:0], data, sdi};

   // fpga slave only reads/ram is written to (sdo not connected to anything on board)
   assign sdo = 1'b0;
endmodule

//////////////////////////////////////////////
// update_mem_control
//   Module for determining how memory is updated from spi
//   data. Pretty simple logic here for now. In previous versions
//   this was much more complicated.
//
//////////////////////////////////////////////
module update_mem_control(input  logic          clk,
                          input  logic          reset,
                          input  logic          ce,
                          input  logic [5:0]    addr,
                          input  logic [31:0]   data,
                          output logic [5:0]    wa,
                          output logic [31:0]   wd,
                          output logic          we);

   logic wasce; // used to find when ce first goes low

   always_ff @(posedge clk, posedge reset) begin
       if (reset) begin
           wasce <= 1'b0;
       end else begin
           wasce <= ce;
       end
   end

   assign we = (wasce & ~ce); // write when ce goes low
   assign wa = addr;
   assign wd = data;
endmodule
```

Appendix C: The Maze and Solution