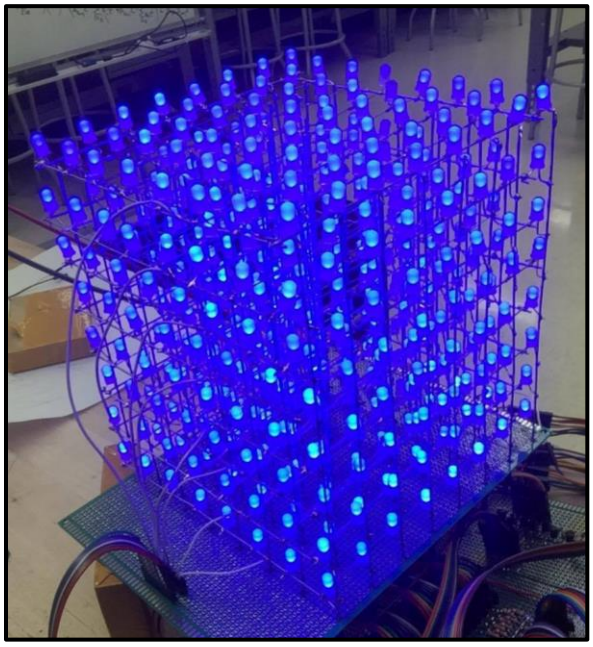
8x8x8 LED Matrix and 3D Snake

Final Project Report

December 8, 2017

E155

Sarp Misoglu



Abstract

There are many examples of three-dimensional LED matrices floating around on the internet. However most of them are designed to run pre-programmed animations and therefore it is difficult to display more complex programs, such as games, on them. The goal of this assignment was to build an 8x8x8 monochromatic LED matrix that could be easily controlled by the RaspberryPi via SPI connection. By having the Pi control the matrix as a display, it is possible to write interactive programs that accept inputs over SSH. This project includes the game Snake, played on a three-dimensional board, displayed on the matrix and controlled by keystrokes in console on the Pi.

Introduction

Three-dimensional LED matrices are not uncommon. But many of the designs simply aim to display simple animations and do not allow interactive programs such as games to be displayed on the matrix with ease.

This project aimed at designing a 3D LED cube that is easily controlled by the RaspberryPi. By doing so programs such as the game Snake can be played on a three-dimensional plane and displayed on the 3D matrix. Taking advantage of the Pi's operating system and using a simple API, any other program can be trivially displayed on the cube, creating room for future development.

The cube circuit is designed to be controlled with 24 control pins. Since the Pi does not have enough pins available for the job, the FPGA is tasked with controlling the cube. **Figure 1** shows a simplified view of the connections between components. The game lives inside the RaspberryPi and the Pi sends the FPGA a new batch of data every time the board changes. The FPGA stores a 512-bit array (one for each LED on the matrix) and continuously displays the current 512 bits until it receives a new batch via SPI.

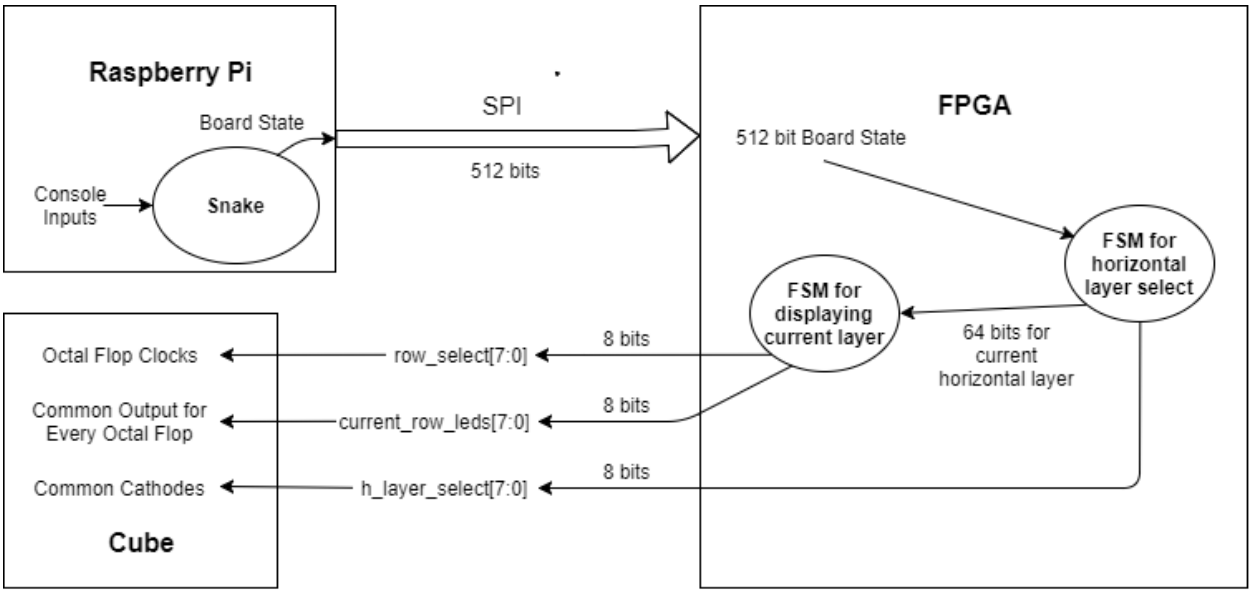


Figure 1. High-level block diagram of project components

Cube Design and Schematics

The cube is composed of 8 layers of 64 LEDs. All cathodes within a layer are connected and form 8 common cathodes in total. All anodes in a column are connected and form 64 common anodes in total. **Figure 2** shows the design of a sample layer. **Figure 3** shows layers and columns on the assembled cube.

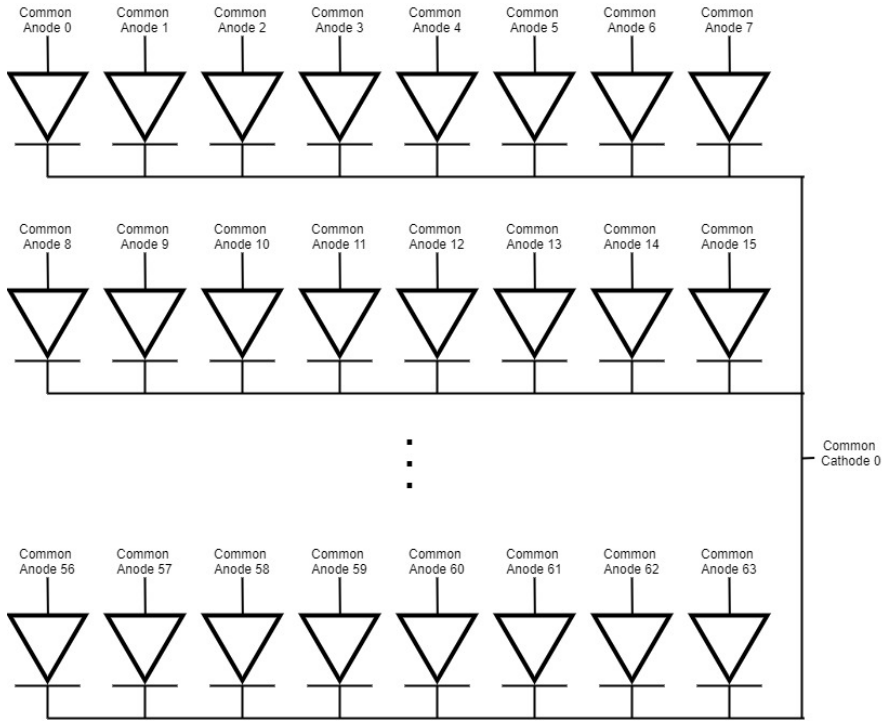


Figure 2. Layout for layer 0 of the cube. The layout is the same except for every layer however each layer is connected to a different common cathode.

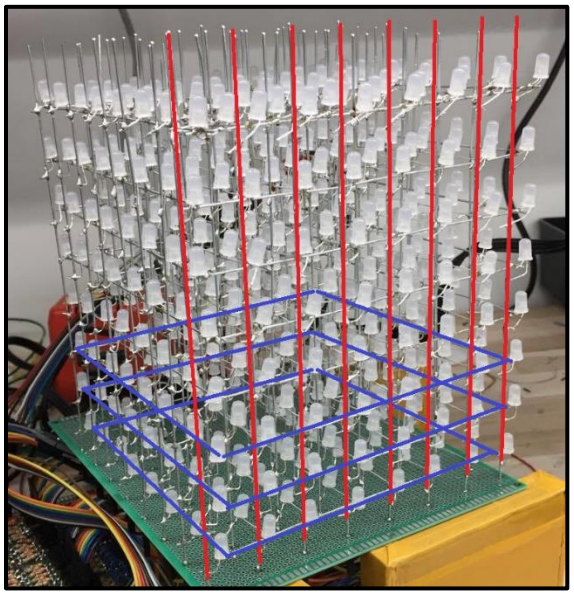


Figure 3. Blue squares outline horizontal layers that share a common cathode and red lines are columns of LEDs that share a common anode. Only 3 cathodes and 8 anodes are displayed for simplicity.

Each common cathode is driven by two NPN transistors in parallel as shown in **Figure 4**. Common anodes are driven by NPN transistors as well. However, since the FPGA does not have 64 pins, octal flops are used to control the anodes. **Figure 5** shows the circuit design of the octal flop chips. Every chip shares 8 inputs and have unique clock signals. Overall the necessary signals to control the cube are: 8 bits to control common cathodes, 8 bits for the inputs of octal flops, and another 8 bits for the clock signals of 8 octal flop chips.

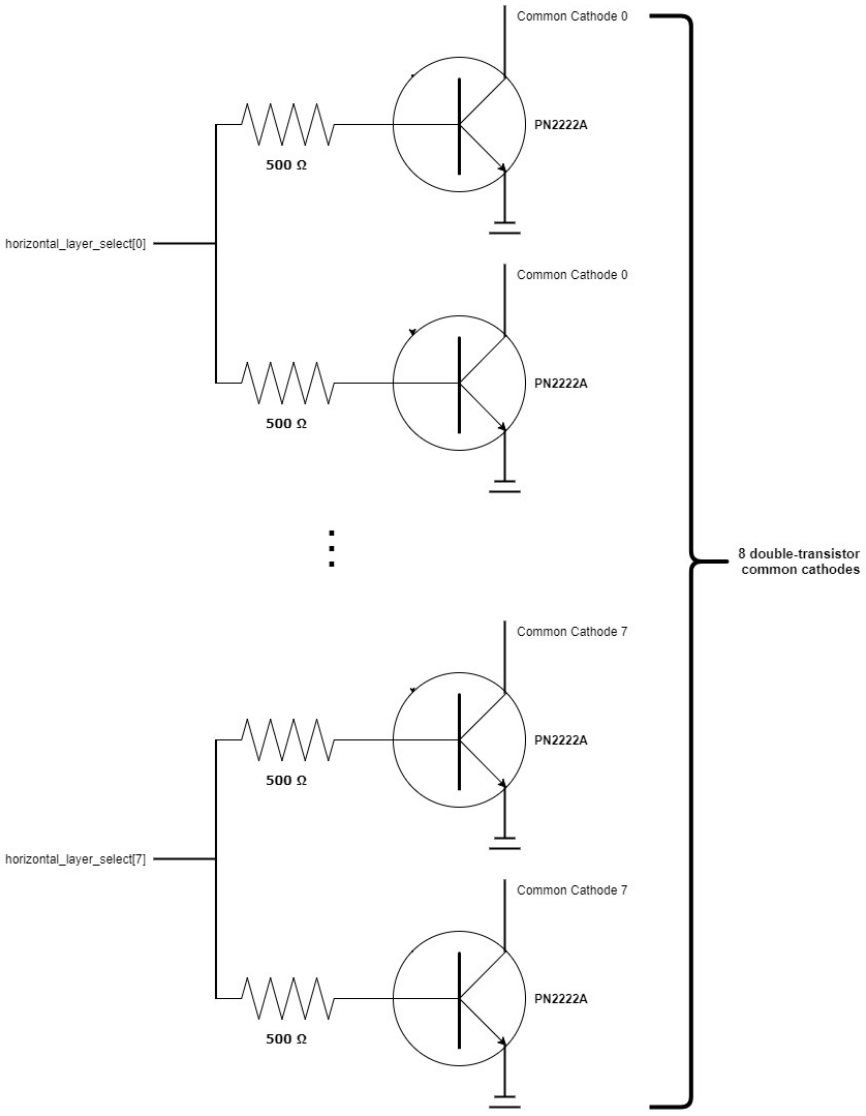


Figure 4. Common cathodes that drive one layer of the cube. Double transistors are used in order to prevent small NPN transistors from overheating.

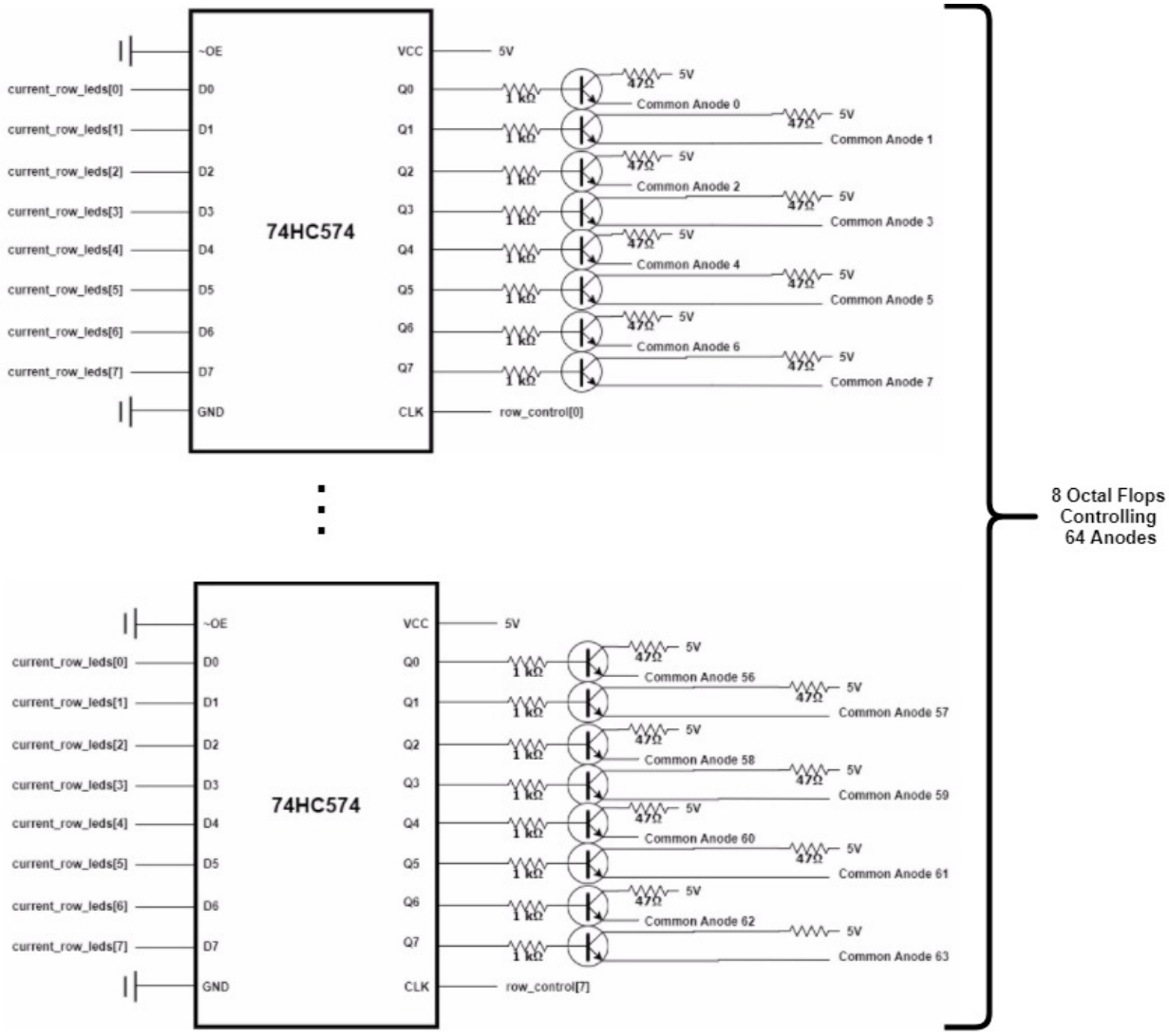


Figure 5. 8 octal flops drive 64 anodes. Common inputs are expected to change accordingly when a chip's clock is asserted.

FPGA Design and Schematics

Figure 6 shows all the connections of the FPGA. The FPGA contains an SPI module for communication with the Pi. When the load signal is asserted it shifts received bits into an array of 512 bits. When the load signal is deasserted, the captured array is copied into an array called the "board state".

The board state is a 512-bit array and each bit represents one LED in the matrix. Bits 0-63 are LEDs in the first layer of the cube, 64-127 the second layer and so on…

The top-level module uses a very simple finite state machine with a slow clock to multiplex between the 8 layers. At any given time the current layer's common cathode signal, horizontal_layer_select[x], is asserted. So at any given moment only a single layer's LEDs are on.

The FPGA contains another module called the h_layer_displayer. This module takes as input the 64 bits corresponding to the current layer that's cathode signal is asserted and contains another FSM to set each flop's outputs to the correct outputs for the row that the flop controls.

This FSM multiplexes between the clock signals of each clock, row_select[x], continuously. When the clock signal of a flop is asserted, the common inputs of the flops, current_row_leds[7:0] are set to the corresponding 8 bits from the array of 64 bits that represent the current layer's LEDs.

Figures 4 and **5** can be revisited to see a complete picture of how the 24 control bits from the FPGA { row_select[7:0], current_row_leds[7:0], h_layer_select[7:0] } control the cube.

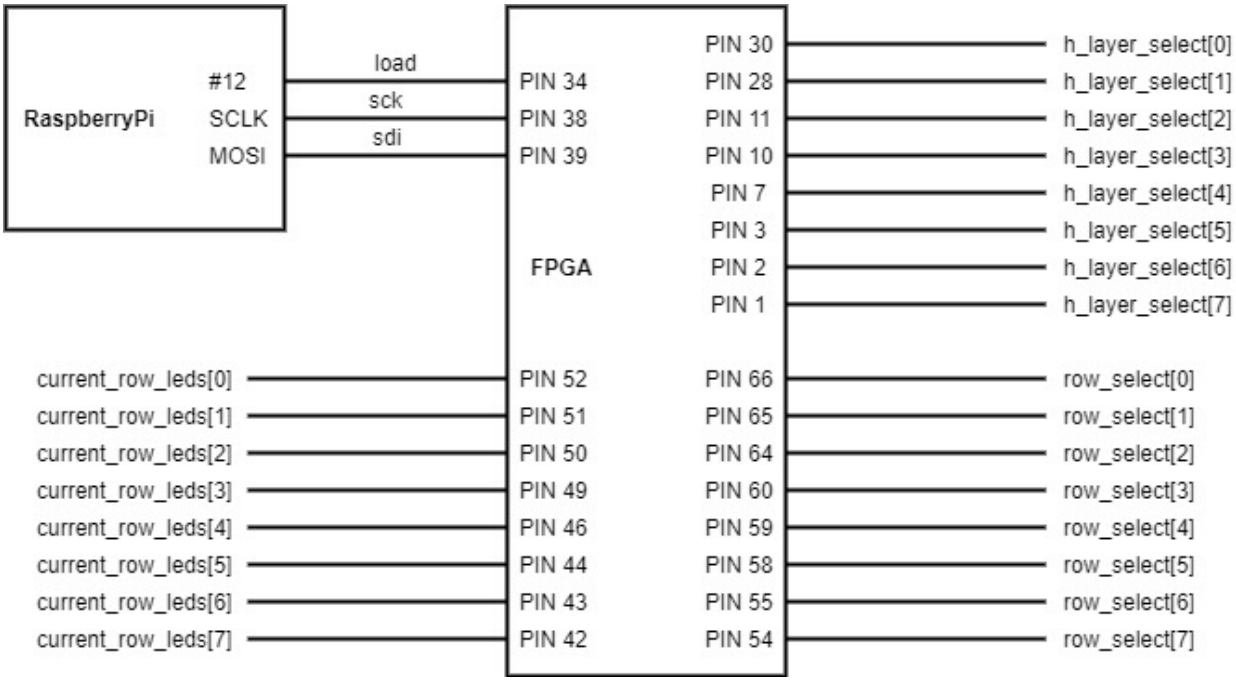


Figure 6. Schematics for all connections of the FPGA.

Microcontroller Design

The RaspberryPi is tasked with running the Snake game and sending the game's board state of 512 bits to the FPGA as the board state changes.

The software modules at work are:

Board Operations. A small C language API used to interface with the cube. Uses a character array of size 64 to represent the 512 bits. Contains functions like *addToBoard* which, given coordinates on the matrix in the form of (x,y,z), does bit-shifting and an OR operation to assert the corresponding bit on the character array. The function *clearBoard* resets the character array and makes every bit zero. The

function *drawBoard* takes a character array and sends it to the FPGA via SPI. It uses the EasyPIO.h library to use the pins available on the Pi.

User Inputs. The game is controlled by keyboard inputs read from console. However, to run the game while receiving inputs, two threads must be running. This module contains a pointer to a function that sets the Snake's direction on key press. This pointer is later given as an argument to a new thread. This module uses the Termios library to read from console continuously. It also has a method that sets the Snake's speed according to the user's input at the start of the game.

Snake List and Food. These modules control the objects on the board as the game is played. Snake List is a Linked List implementation. Every node of the list stores its own coordinates. There are enumerated values for each of the 6 directions on the three-dimensional plane and a method to insert a new head to the snake towards the direction the snake is currently traveling to. There is also a method to remove the tail of the snake. The *Food* module has logic to return if a given point contains food. The *makeMove* method inserts a new head, removes the tail if the previous head was not on food and checks if the snake by checking if the head is out of bounds or if the head is on a location that also contains another node that is not the head. If the snake died, the game is over.

Main. This module contains the *main* method for the game. It initializes a new thread for user inputs, sets up the game and starts the game loop. The game loop calls the *makeMove* method on the snake, waits for an amount of time decided by the snake speed the user selects at the start of the game, adds the snake and food to the board using the *addToBoard* function and then draws the board using the *drawBoard* function. If the player has died, the game loop ends and their score, which is the snake's length, is displayed. For delaying between turns, the EasyPIO library is used for its accurate *delayMicros* function.

Additional modules not within the scope of the proposal:

Rain. This is a rain animation for the cube. It creates points called "raindrops" randomly on the top of the cube and in each cycle, moves them down, creating a feeling of falling rain.

Squares. This module creates a data structure *Square* which is essentially 8 points for each vertex of a cube. It contains methods to draw lines between each vertex (edges) and add them to the board. This creates a hollow cube with only the edges drawn on the matrix. There is also functionality to "expand" the cube which pushes each vertex to the corners of the matrix. When the cube is expanded periodically and rapidly it creates the illusion of a cube starting at the center of the matrix and growing to fit the matrix perfectly.

Results

Overall the project ended up to be very successful but extremely time-consuming as well. The most challenging part of the design was to build the cube so that it would be firm and it would not lose its form. However using any nonconducting material or thick wiring would mean sacrificing visibility in the center of the cube which had to be avoided since the purpose of the project was to play Snake on the matrix. I ended up using galvanized steel wire that I straightened and the structure of the cube was quite strong.

The design of the cube has the matrix and the FPGA function as a single system that can be controlled by only 3 pins plus a VCC and GND pin. This, along with the API for interfacing with the cube has made using the cube extremely easy. Programs written for the cube can be quite high-level which means you can make use of the Pi's operating system to do anything you would like with the cube, from controlling it from your phone to visualizing any kind of data that can be represented on an 8x8x8 board.

References

[1] PN2222 Datasheet: https://www.onsemi.com/pub/Collateral/PN2222-D.PDF

[2] 74HC574 Datasheet: http://www.ti.com/lit/ds/symlink/sn74hc574.pdf

Parts List

Part	Price
600x White Diffues Blue LED	$6.87 for 100-pack
8x Major Brands 74HC574 Octal Flip Flop	$10.20 for 10-pack
80x PN2222 NPN Transistor	$10.80 for 100-pack

APPENDIX A: FPGA SYSTEM VERILOG CODE

```
//////////////////////////////////////////
// Top level module for LED cube
// Author: Sarp Misoglu
// Date: 11/20/2017
//////////////////////////////////////////
module led_cube(input logic clk,
                input  logic sck,
                input  logic sdi,
                input  logic load,
                output logic[7:0] leds,
                output logic[7:0] row_select,
                output logic[7:0] current_row_leds,
                output logic[7:0] h_layer_select);

        // get board state from Pi using SPI
        logic[511:0] board_state;
        cube_spi spi(clk,sck,sdi,load,board_state);

        // slow clock to choose horizontal layers
        logic display_clk;
        logic[31:0] counter;
        always_ff @(posedge clk)
                begin
                        if(counter == 200_00)
                                begin
                                        counter = 0;
                                        display_clk = ~display_clk;
                                end
                        else counter = counter + 1;
                end


        // state machine for 8 layers
        logic[7:0] next_layer;

        // next layer logic
        always_comb
                case(h_layer_select)
                        8'b0000_0001: next_layer <= 8'b0000_0010;
                        8'b0000_0010: next_layer <= 8'b0000_0100;
                        8'b0000_0100: next_layer <= 8'b0000_1000;
                        8'b0000_1000: next_layer <= 8'b0001_0000;
                        8'b0001_0000: next_layer <= 8'b0010_0000;
                        8'b0010_0000: next_layer <= 8'b0100_0000;
                        8'b0100_0000: next_layer <= 8'b1000_0000;
                        8'b1000_0000: next_layer <= 8'b0000_0001;
```

```
                default: next_layer <= 8'b0000_0001;
            endcase

        always_ff @(posedge display_clk)
            begin
                h_layer_select = next_layer;
            end
        // logic for current layer's data
        logic[63:0] current_h_layer;
        always_comb
            case(h_layer_select)
                8'b0000_0001: current_h_layer <= board_state[511:448];
                8'b0000_0010: current_h_layer <= board_state[447:384];
                8'b0000_0100: current_h_layer <= board_state[383:320];
                8'b0000_1000: current_h_layer <= board_state[319:256];
                8'b0001_0000: current_h_layer <= board_state[255:192];
                8'b0010_0000: current_h_layer <= board_state[191:128];
                8'b0100_0000: current_h_layer <= board_state[127:64];
                8'b1000_0000: current_h_layer <= board_state[63:0];
                default: current_h_layer <= 64'h0f0f_0f0f_0f0f_0f0f;
            endcase

        h_layer_displayer hld(clk, current_h_layer, current_row_leds, row_select);

endmodule

//////////////////////////////////////////
// Module to assign columns of the cube
// in a layer
//////////////////////////////////////////
module h_layer_displayer(input logic clk,
                input logic[63:0] current_h_layer,
                output logic [7:0] current_row_leds,
                output logic [7:0] row_select);

        // update each row one by one by choosing flops
        logic[7:0] next_row;
        always_comb
            case(row_select)
                8'b0000_0001: next_row <= 8'b0000_0010;
                8'b0000_0010: next_row <= 8'b0000_0100;
                8'b0000_0100: next_row <= 8'b0000_1000;
                8'b0000_1000: next_row <= 8'b0001_0000;
                8'b0001_0000: next_row <= 8'b0010_0000;
                8'b0010_0000: next_row <= 8'b0100_0000;
                8'b0100_0000: next_row <= 8'b1000_0000;
                8'b1000_0000: next_row <= 8'b0000_0001;
                default: next_row <= 8'b0000_0001;
```

```
		endcase

	// send correct 8 bits for the current row to the flops
	always_comb
		case(row_select)
			8'b0000_0001: current_row_leds <= current_h_layer[63:56];
			8'b0000_0010: current_row_leds <= current_h_layer[55:48];
			8'b0000_0100: current_row_leds <= current_h_layer[47:40];
			8'b0000_1000: current_row_leds <= current_h_layer[39:32];
			8'b0001_0000: current_row_leds <= current_h_layer[31:24];
			8'b0010_0000: current_row_leds <= current_h_layer[23:16];
			8'b0100_0000: current_row_leds <= current_h_layer[15:8];
			8'b1000_0000: current_row_leds <= current_h_layer[7:0];
			default: current_row_leds <= 8'b1010_1010;
		endcase

	always_ff @(posedge clk)
		begin
				row_select = next_row;
		end

endmodule

//////////////////////////////////////////
// Simple SPI module to retrieve
// current board state from Pi
//////////////////////////////////////////
module cube_spi(input logic clk,
		input  logic sck,
		input  logic sdi,
		input  logic load,
		output logic [511:0] board_state);

	logic [511:0] board_state_captured;

	 // capture serial input when load is asserted
	always_ff @(posedge sck)
		if(load) board_state_captured = {board_state_captured[510:0],sdi};

	always_ff @(posedge clk)
		if(!load) board_state = board_state_captured;

endmodule
```

APPENDIX B: C CODE FOR RASPBERRY PI

```
////////////////////////////////////////////////////////
// main.c
// Main method for snake game.
// Author: Sarp Misoglu
// 12/5/2017
////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "EasyPIO.h"
#include "function_declarations.h"
#include "board_operations.h"
#include "food.h"
#include "snake_list.h"
#include "getch.h"
#include "user_inputs.h"

void main(void) {
	initTermios(1);

	// set up SPI connection between Pi and FPGA
	initializeSPI();

	// start with a clean board
	char board[64];
	clearBoard(board);

	// initialize RNG for foods
	time_t t;
	srand((unsigned) time(&t));

	// turn all lights on until user selects speed
	everythingOn(board);
	drawBoard(board);
	delayMillis(2000);

	// user selects the speed (the delay between moves)
	int gameSpeed;
	speedSelect(&gameSpeed);

	// thread to handle user inputs during game
	pthread_t input_thread;
	pthread_create(&input_thread, NULL, getUserInputs, (void*) &snakeDirection);

	// clear the board
	clearBoard(board);
	drawBoard(board);

	// snake initialize animation
	initializeSnake(board, &gameSpeed, 4);

	// place food
	randomFood();

	// start the game
	playing = 1;
	// game loop
	while(playing) {
		delayMillis(gameSpeed);
		makeMove();
		clearBoard(board);
		addSnake(board);
		addFood(board);
		drawBoard(board);
	}

	// cancel the user input thread when game is over
```

```
        pthread_cancel(input_thread);
        printf("Game over.\n");
        printf("Your score was = %d\n", snakeLength());

        // free the memory to avoid memory leaks
        freeSnake();
}
```

```
//////////////////////////////////////////////////////////
// snake_list.h
// File containing data structures and methods for
// the snake.
//////////////////////////////////////////////////////////

struct SnakeNode {
	int x;
	int y;
	int z;
	struct SnakeNode *prev;
};

// snake directions
enum direction{POSX, NEGX, POSY, NEGY, POSZ, NEGZ};

// snake variables
struct SnakeNode *head = NULL;
int snakeDirection = POSX;
int ateFood = 0;
int playing = 0;

// prints the snake on the console
void printSnake(){
	struct SnakeNode* temp = head;
	printf("snake = ");
	while(temp != NULL){
		printf(" (%d, %d, %d)", temp->x, temp->y, temp->z);
		temp = temp->prev;
	}
	printf("\n");
}

// returns the number of nodes in the snake
int snakeLength(){
	int length = 0;
	struct SnakeNode* temp = head;
	while(temp != NULL){
		length++;
		temp = temp->prev;
	}
	return length;
}

// adds snake to the board
void addSnake(char* board){
	struct SnakeNode* temp = head;
	while(temp != NULL){
		addToBoard(board, temp->x, temp->y, temp->z);
		temp = temp->prev;
	}
}

////////////////////////// COORDINATE CHECKS ////////////////////////

// returns true if a given coordinates contains the snake
int containsSnake(int x, int y, int z){
	struct SnakeNode* temp = head;
	while(temp != NULL){
		if (x == temp->x && y == temp->y && z == temp->z) {
			return 1;
		}
		temp = temp->prev;
	}
	return 0;
}

// returns true if given coordinates contain snake excluding head
int containsSnakeBody(int x, int y, int z){
	if (head->prev == NULL) return 0;
```

```
	struct SnakeNode* temp = head->prev;
	while(temp != NULL){
		if (x == temp->x && y == temp->y && z == temp->z) {
			return 1;
		}
		temp = temp->prev;
	}
	return 0;
}

////////////////////////////// MOVEMENT ////////////////////////////

void insertHead(){
	if(head == NULL) return;

	struct SnakeNode* temp = (struct SnakeNode*)malloc(sizeof(struct SnakeNode));
	temp->x = head->x;
	temp->y = head->y;
	temp->z = head->z;

	switch(snakeDirection){
		case POSX :
			temp->x = head->x + 1;
			break;
		case NEGX :
			temp->x = head->x - 1;
			break;
		case POSY :
			temp->y = head->y + 1;
			break;
		case NEGY :
			temp->y = head->y - 1;
			break;
		case POSZ :
			temp->z = head->z + 1;
			break;
		case NEGZ :
			temp->z = head->z - 1;
			break;
	}

	temp->prev = head;
	head = temp;
}

void removeTail(){
	if(head == NULL) return;

	struct SnakeNode* temp = head;

	if(temp->prev == NULL) head = NULL;
	else {
		while(temp->prev->prev != NULL){
			temp = temp->prev;
		}
		free(temp->prev);
		temp->prev = NULL;
	}
}

void makeMove(){
	insertHead();
	if(ateFood == 0){
		removeTail();
	}
	// did the snake die?
	if(containsSnakeBody(head->x, head->y, head->z)
		|| outOfBounds(head->x, head->y, head->z)){
		playing = 0;
	}
```

```
	// did the snake eat food?
	if(containsFood(head->x, head->y, head->z)){
		ateFood = 1;
		randomFood();
	} else {
		ateFood = 0;
	}
	// printSnake();
}

void initializeSnake(char* board, int* gameSpeed, int size){
	snakeDirection = POSX;
	head = (struct SnakeNode*)malloc(sizeof(struct SnakeNode));
	head->z = 4;
	head->y = 3;
	head->x = 0;

	int i;
	for(i = 0; i < size-1; i++){
		addSnake(board);
		drawBoard(board);
		delayMillis(*gameSpeed);
		insertHead();
	}
	addSnake(board);
	drawBoard(board);
}

void freeSnake(){
	struct SnakeNode* temp = head;
	struct SnakeNode* temp2;

	while(temp != NULL){
		temp2 = temp->prev;
		free(temp);
		temp = temp2;
	}
}
```

```
//////////////////////////////////////////////////////////
// food.h
// File containing methods for the food in the snake game
//////////////////////////////////////////////////////////

int foodX;
int foodY;
int foodZ;

// puts food at a random point on the grid that does not
// contain snake
void randomFood(){
	foodX = rand() % 8;
	foodY = rand() % 8;
	foodZ = rand() % 8;
	if(containsSnake(foodX, foodY, foodZ)){
		randomFood();
	}
}

void addFood(char* board){
	addToBoard(board, foodX, foodY, foodZ);
}

int containsFood(int x, int y, int z){
	return (x == foodX && y == foodY && z == foodZ);
}
```

```
////////////////////////////////////////////////////////
// board_operations.h
// Simple API to interface with the FPGA driving the cube.
////////////////////////////////////////////////////////

#define LOAD_PIN 12

void initializeSPI(){
	pioInit();
	spiInit(244000, 0);
	pinMode(LOAD_PIN, OUTPUT);
}

// Send the board to the FPGA via SPI
void drawBoard(char *board){
	digitalWrite(LOAD_PIN, 1);
	int i = 0;
	for(i; i < 64; i++){
		spiSendReceive(board[i]);
	}
	digitalWrite(LOAD_PIN, 0);
}

void clearBoard(char* board){
	int i = 0;
	for(i; i < 64; i++){
		board[i] = 0x00;
	}
}

void addToBoard(char* board, int x, int y, int z){
	board[z*8 + y] |= (0x01 << x);
}

// returns true if the given coordinates are not in the grid
int outOfBounds(int x, int y, int z){
	return (x > 7 || x < 0 || y > 7 || y < 0 || z > 7 || z < 0);
}

////////////////// TESTING BELOW //////////////////

void testRow(char* board, int y){
	printf("testing row\n");
	int x = 0;
	for (x; x < 8; x++){
		delayMillis(500);
		clearBoard(board);
		addToBoard(board, x, y, 7);
		drawBoard(board);
	}
	delayMillis(500);
}

void everythingOn(char* board){
	int y = 0;
	int x = 0;
	int z = 0;
	for(z = 0; z < 8; z++){
		for(x = 0; x < 8; x++){
			for(y = 0; y < 8; y++){
				addToBoard(board, x, y, z);
			}
		}
	}
}

// a sample board state for testing
char a[64] = {
	0x11, 0xff, 0xff, 0xff,
	0x00, 0x12, 0x22, 0x00,
```

```
        0x21, 0xff, 0xff, 0xff,
        0xff, 0xff, 0xff, 0xff,

        0x31, 0xaa, 0xaa, 0xaa,
        0xaa, 0xaa, 0xaa, 0xaa,

        0x41, 0xaa, 0xaa, 0xaa,
        0xaa, 0xaa, 0xaa, 0xaa,

        0x51, 0xaa, 0xaa, 0xaa,
        0xaa, 0xaa, 0xaa, 0xaa,

        0x61, 0xaa, 0xaa, 0xaa,
        0xaa, 0xaa, 0xaa, 0xaa,

        0x71, 0xaa, 0xaa, 0xaa,
        0xae, 0xaa, 0xaa, 0xaa,

        0x81, 0x02, 0x04, 0x08,
        0x10, 0x20, 0x44, 0x84
};
```

```
//////////////////////////////////////////////////////
// user_inputs.h
// File containing methods to get input from user
// via Linux terminal
//////////////////////////////////////////////////////

// wait for user input from console to change snake direction
// pointer to the function to send it to new thread
void *getUserInputs(void *ptr){
	int *snake_dir;
	snake_dir = (int *)ptr;
	// printf("input thread started!\n");

	char c;
	while(1){
		c = getch();
		if(c == 'q') break;

		switch(c){
			case 'a' :
				// can't do a 180 turn!
				if(*snake_dir != NEGX)
					*snake_dir = POSX;
				break;
			case 'd' :
				if(*snake_dir != POSX)
					*snake_dir = NEGX;
				break;
			case 's' :
				if(*snake_dir != NEGY)
					*snake_dir = POSY;
				break;
			case 'w' :
				if(*snake_dir != POSY)
					*snake_dir = NEGY;
				break;
			case 'i' :
				if(*snake_dir != NEGZ)
					*snake_dir = POSZ;
				break;
			case 'k' :
				if(*snake_dir != POSZ)
					*snake_dir = NEGZ;
				break;
		}
	}
}

// user selects game speed or quits
void speedSelect(int *gameSpeed){
	int choosing = 1;
	char ch;
	printf("Select game speed: \n's' for snail, 'd' for dragon, 'i' for insane\n");
	while(choosing){
		ch = getch();
		switch(ch){
			case 's' :
				*gameSpeed = 1000;
				choosing = 0;
				printf("Snail it is! \n");
				break;
			case 'd' :
				*gameSpeed = 350;
				choosing = 0;
				printf("Dragon it is!\n");
				break;
			case 'i' :
				*gameSpeed = 120;
				choosing = 0;
				printf("Insane!!!\n");
```

```
                        break;
                default :
                        printf("That's not a valid selection!\n");
                        break;
                }
        }
}
```

```
////////////////////////////////////////////////////////
// square_main.c
// Displays a sequence of hollow cubes of changing sizes
////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "../EasyPIO.h"
#include "../board_operations.h"
#include "square.h"


void main(void) {
	initializeSPI();

	// start with a clean board
	char board[64];
	clearBoard(board);

	struct Cube cube;
	initializeCube(&cube);

	while(1){
		clearBoard(board);
		addCube(board, &cube);
		drawBoard(board);
		expandCube(&cube);
		if(cubeOutOfBounds(&cube)) initializeCube(&cube);
		delayMillis(80);
	}
}

////////////////////////////////////////////////////////
// square.h
// Methods for creating and modifying cubes
////////////////////////////////////////////////////////

struct Point{
	int x;
	int y;
	int z;
};

struct Cube{
	struct Point* points;
};

void initializeCube(struct Cube* cube){
	cube->points = malloc(8*sizeof(struct Point));

	cube->points[0].x = 4;
	cube->points[0].y = 4;
	cube->points[0].z = 4;

	cube->points[1].x = 3;
	cube->points[1].y = 4;
	cube->points[1].z = 4;

	cube->points[2].x = 4;
	cube->points[2].y = 3;
	cube->points[2].z = 4;

	cube->points[3].x = 3;
	cube->points[3].y = 3;
	cube->points[3].z = 4;

	cube->points[4].x = 4;
	cube->points[4].y = 4;
	cube->points[4].z = 3;
```

```
	cube->points[5].x = 3;
	cube->points[5].y = 4;
	cube->points[5].z = 3;

	cube->points[6].x = 4;
	cube->points[6].y = 3;
	cube->points[6].z = 3;

	cube->points[7].x = 3;
	cube->points[7].y = 3;
	cube->points[7].z = 3;
}

void expandCube(struct Cube* cube){
	cube->points[0].x = cube->points[0].x + 1;
	cube->points[0].y = cube->points[0].y + 1;
	cube->points[0].z = cube->points[0].z + 1;

	cube->points[1].x = cube->points[1].x - 1;
	cube->points[1].y = cube->points[1].y + 1;
	cube->points[1].z = cube->points[1].z + 1;

	cube->points[2].x = cube->points[2].x + 1;
	cube->points[2].y = cube->points[2].y - 1;
	cube->points[2].z = cube->points[2].z + 1;

	cube->points[3].x = cube->points[3].x - 1;
	cube->points[3].y = cube->points[3].y - 1;
	cube->points[3].z = cube->points[3].z + 1;

	cube->points[4].x = cube->points[4].x + 1;
	cube->points[4].y = cube->points[4].y + 1;
	cube->points[4].z = cube->points[4].z - 1;

	cube->points[5].x = cube->points[5].x - 1;
	cube->points[5].y = cube->points[5].y + 1;
	cube->points[5].z = cube->points[5].z - 1;

	cube->points[6].x = cube->points[6].x + 1;
	cube->points[6].y = cube->points[6].y - 1;
	cube->points[6].z = cube->points[6].z - 1;

	cube->points[7].x = cube->points[7].x - 1;
	cube->points[7].y = cube->points[7].y - 1;
	cube->points[7].z = cube->points[7].z - 1;
}

void addCube(char* board, struct Cube* cube){
	// connect adjacent vertices with lines
	// if two points have two attributes the same,
	// they are adjacent
	int i;
	int j;
	for(i = 0; i < 8; i++){
		for(j = 0; j < 8; j++){
			if(cube->points[i].x == cube->points[j].x
				&& cube->points[i].y == cube->points[j].y){
				int maxZ;
				int minZ;
				if(cube->points[i].z > cube->points[j].z){
					maxZ = cube->points[i].z;
					minZ = cube->points[j].z;
				} else {
					maxZ = cube->points[j].z;
					minZ = cube->points[i].z;
				}
				while(maxZ >= minZ){
					addToBoard(board, cube->points[i].x, cube->points[i].y, maxZ);
					maxZ--;
```

```
                                        }
                                }
                                if(cube->points[i].x == cube->points[j].x
                                        && cube->points[i].z == cube->points[j].z){
                                        int maxY;
                                        int minY;
                                        if(cube->points[i].y > cube->points[j].y){
                                                maxY = cube->points[i].y;
                                                minY = cube->points[j].y;
                                        } else {
                                                maxY = cube->points[j].y;
                                                minY = cube->points[i].y;
                                        }
                                        while(maxY >= minY){
                                                addToBoard(board, cube->points[i].x, maxY, cube->points[i].z);
                                                maxY--;
                                        }
                                }
                                if(cube->points[i].y == cube->points[j].y
                                        && cube->points[i].z == cube->points[j].z){
                                        int maxX;
                                        int minX;
                                        if(cube->points[i].x > cube->points[j].x){
                                                maxX = cube->points[i].x;
                                                minX = cube->points[j].x;
                                        } else {
                                                maxX = cube->points[j].x;
                                                minX = cube->points[i].x;
                                        }
                                        while(maxX >= minX){
                                                addToBoard(board, maxX, cube->points[i].y, cube->points[i].z);
                                                maxX--;
                                        }
                                }
                        }
                }
}

int cubeOutOfBounds(struct Cube* cube){
        return(cube->points[0].x > 7);
}
```

```
//////////////////////////////////////////////////////////
// rain_main.c
// Rain animation for the cube
//////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include "../EasyPIO.h"
#include "../board_operations.h"
#include "rain.h"
void main(void){
	initializeSPI();

	// initialize board
	char board[64];
	clearBoard(board);

	rainSetup();

	int i;
	while(1){
		moveDrops();
		for (i = 0; i < 2; i++){
			randomDrop();
		}
		clearBoard(board);
		addDrops(board);
		drawBoard(board);
		delayMillis(50);
	}
}




//////////////////////////////////////////////////////////
// rain.h
// Contains data structures for rain drops and animating
// drops
//////////////////////////////////////////////////////////

struct RainDrop{
	int dropX;
	int dropY;
	int dropZ;
	int on_board;
};

struct RainDrop* drops;

void rainSetup(){
	// initialize rng
	time_t t;
	srand((unsigned) time(&t));

	drops = malloc(64*sizeof(struct RainDrop));

	// int i = 0;
	// for(i = 0; i < 64; i++){
	//	drops[i].dropX = 0;
	//	drops[i].dropY = 0;
	//	drops[i].dropZ = 0;
	//	drops[i].on_board = 0;
	// }
}

void addDrops(char* board){
	int i;
	for(i = 0; i < 64; i++){
```

```
                        if (drops[i].on_board){
                                addToBoard(board, drops[i].dropX, drops[i].dropY, drops[i].dropZ);
                        }
                }
}

void moveDrops(){
        int i;
        for(i = 0; i < 64; i++){
                if (drops[i].on_board == 1){
                        if(drops[i].dropZ == 0) {
                                drops[i].on_board = 0;
                        } else{
                                drops[i].dropZ = drops[i].dropZ - 1;
                        }
                }
        }
}

void randomDrop(){
        int dropPos = rand() % 64;
        if(drops[dropPos].on_board == 0) {
                drops[dropPos].dropX = dropPos / 8;
                drops[dropPos].dropY = dropPos % 8;
                drops[dropPos].dropZ = 7;
                drops[dropPos].on_board = 1;
                // printf("(%d, %d, %d)\n", drops[dropPos].dropX, drops[dropPos].dropY, drops[dropPos].dropZ);
        }
}

void freeDrops(){
        free(drops);
}
```