

Tilting Maze Game: Final 6.111 Project Report

Matt Fishburn, fishburn@mit.edu
Hongyi Hu, hongyihu@mit.edu

December 13, 2005

Abstract

This document describes the design and implementation of an electronic tilting maze game in Verilog with an accelerometer-driven controller. The game renders a virtual tilting board with a maze of walls and holes. A player can tilt the controller to change the acceleration on a virtual ball. The goal of the game is to navigate the virtual ball around the maze, avoiding obstacles while guiding the ball to a target.

Contents

1 Overview	1
2 Design	1
2.1 Sensor Input	3
2.2 Game State FSM	3
2.2.1 Collision Detection	3
3 Module Specifications	4
3.1 Clock Dividers by Matt and Hongyi	4
3.2 Input Modules by Matt	4
3.2.1 User Interface Device by Matt	4
3.2.2 Debouncer from Lab Two	5
3.2.3 Synchronizer from Lab Two	5
3.3 Pause FSM by Matt	5
3.4 SPI Synchronizer by Matt	5
3.4.1 SPI FSMs	5
3.5 Physics Engine by Matt	6
3.5.1 Biasing	6
3.6 Game State by Hongyi	6
3.7 Collision Detection Unit by Hongyi	6
3.8 Collision Lookup Unit by Hongyi	6
3.9 Collision Detection Mask by Hongyi	7
3.10 Memory Interface by Hongyi	7
3.11 Memory by Matt	8
3.11.1 Level Generation	8
3.12 X VGA Unit from Lab 4	8
3.13 Draw Unit by Matt	8
3.13.1 Circle Draw Unit by Matt	8
3.13.2 Level Mask by Matt	8
4 Testing and Debugging	8
4.1 Frame Synchronization Bug	9
4.2 Collision Detection Timing Issue	9
4.3 Inferred Bit Flips	9
5 Conclusion	9
A Memory Field Codes	10
B Code	10
B.1 SPI_Synch.v	10
B.2 cd_mask.v	13
B.3 cd_mask_dummy.v	17
B.4 cd_mask_test.v	20
B.5 collision_detection.v	21
B.6 divider.v	23
B.7 drawBallBool.v	24
B.8 drawUnit.v	25
B.9 dynamicLevelMask.v	26
B.10 game_state.v	27
B.11 main.v	28

B.12	memory_interface.v	33
B.13	physics_module.v	33
B.14	word_read_fsm.v	35
B.15	word_write_FSM.v	37

List of Figures

1	Block Diagram	2
2	Collision Detection Diagram	3
3	SPI FSM	5
4	Collision Lookup FSM	7

1 Overview

Our project was inspired by Labyrinth, a children's game consisting of a tilting wooden maze board with holes and a metal ball. A Labyrinth game board has a configuration of obstacles such as walls and trap holes, along with a target hole. By appropriately tilting the board, the player controls how gravity affects the ball and can control the ball's motion. The objective of Labyrinth is to navigate the ball to the target hole without falling through any trap holes. We designed and implemented an electronic version of Labyrinth using the 6.111 labkit and a prototype accelerometer-driven user interface. Our version of Labyrinth consists of a virtual ball on a virtual tilting maze board. The user controls the ball's movement by interacting with the accelerometer driven interface.

The accelerometer-driven user interface device detects how the user wishes to tilt the virtual board, changing the effect of gravity on the virtual ball. The device consists of a two-axes accelerometer attached to a breadboard, complete with circuitry allowing the labkit to detect the current tilt of the breadboard. A state machine in the FPGA tracks the physical state of the game, updating the game state based on input from the user interface device. The game state consists of the ball's velocity and position along with miscellaneous information such as the current level. The 6.111 labkit renders the current state of the maze board, obstacles and ball via an XVGA display.

2 Design

The design of the tilting maze game can be divided into three major components: the sensor input, the game state FSM, and the display. The sensor input modules transform input information into useful data for the game state module. The game state module uses the input information to change the current state of the game and output display information to the display modules. The display modules render the graphics for the game on the screen. The sensor input modules consist of the SPI synchronizer, physics, and debouncer modules. The game state FSM modules consist of the game state, collision detection, collision detection lookup, collision detection mask, memory interface, and memory modules. The display modules consist of the level mask, the draw circle unit, the draw unit, and the XVGA modules.

The block diagram in Figure 1 shows all of the major modules and interconnections between these modules.

2.1 Sensor Input

The sensor input consists of accelerometers and ADCs outputting data to the labkit on the order of 200 Hz. We chose this sampling frequency because the signal to noise ratio would suffer if the inputs were clocked faster. However, if the inputs were clock more slowly, then the sensors would not update fast enough to make the game playable.

2.2 Game State FSM

The game state FSM contains the logic to drive the movement and behavior of objects in the game. Actions in the game occur in discrete time frames; each time frame represents a fraction of a second in real time. Each loop through the game state FSM modules runs through a number of computations and collision detection steps to update the state of the game for the next time frame.

The game state FSM updates the internal game state at 8192 Hz. This frequency was chosen because the ball will never move more than one pixel given one Earth g of acceleration during a clock cycle with the clock running at 8192 Hz. Running this update rate any faster will increase drift error from acceleration, but running this update rate slower would cause animation quality to suffer.

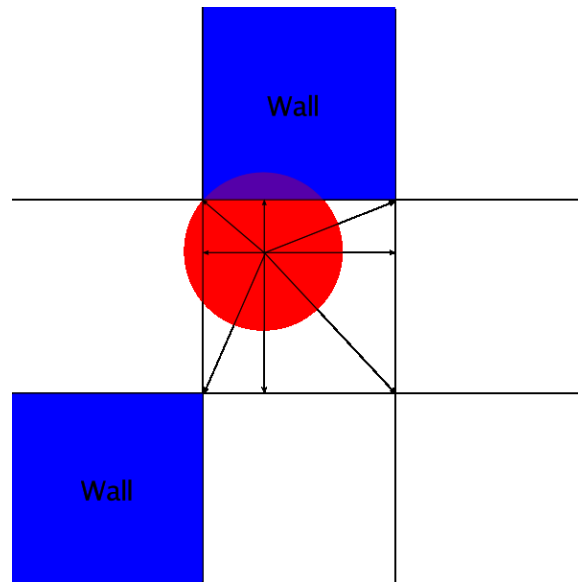


Figure 2: Collision detection diagram.

2.2.1 Collision Detection

The virtual game board is divided into discrete 16 pixel by 16 pixel blocks, so a level is 64 blocks wide and 48 blocks tall at 1024 by 768 resolution. Each block can be filled by a wall, a trap hole or a target hole.

Our collision detection scheme is performed on a block by block basis, as suggested by Professor Terman. On each collision detection cycle, we first determine the block that the center of the ball currently occupies. We then look up the contents of the eight blocks that immediately surround the center block as shown in Figure 2. If any part of the ball overlaps with a block that has a wall in it, then a collision has occurred with that block. We determine if the ball overlaps with one of the adjacent eight blocks by computing the distance from the center of the ball to the four corners and the sides of the center block and checking if the ball's radius is greater than any of these distances.

One advantage of this collision detection system is that it scales well with ball size. If the size of the ball needs to be changed, a one parameter can be changed rather than reimplementing an entire module.

3 Module Specifications

Our system design is composed of thirteen major modules:

- Clock dividers
- Accelerometer user interface device
- SPI synchronizer
- Physics engine
- Game state module
- Collision detection module
- Collision lookup module
- Collision detection mask
- Memory Interface
- Memory
- XVGA unit
- Draw unit
- Display level mask

3.1 Clock Dividers by Matt and Hongyi

Clock dividers are used to generate the serial clock and master clock signals used by the SPI synchronizer module, along with the update clock used by the game state FSM. The signals coming from a divider, such as the `sclk` signal, are not actual clock signals, and `posedge` or `negedge` should never be used to detect the rising or falling edge of these signals. If `posedge` or `negedge` are used to detect the edge of these clock signals, timing issues may result from the “clock” signal reaching one place on chip before another. If edge detection is required the signal or its inverse should be run through a level to pulse generator, as these signals are guaranteed to be synchronous with the input clock.

3.2 Input Modules by Matt

The inputs to the game consist of eight switches, three buttons and an accelerometer-based user interface device. The debouncer, synchronizer and SPI synchronizer modules interface to these components.

3.2.1 User Interface Device by Matt

The user interface device uses accelerometer sensors to detect the tilt of an interface device and transforms those motions into control signals for the game. The accelerometers output analog signals with a voltage proportional to static acceleration. These signals are fed into a set of AD7706 ADCs. These ADCs convert the analog signals into digital signals and communicate these digital signals with the labkit. The ADCs and labkit communicate using the SPI protocol. The ADCs output a 16-bit signal corresponding to the current orientation of the interface device. All SPI signals external to the FPGA are run through the synchronizer module.

3.2.2 Debouncer from Lab Two

The debouncer module debounces switch inputs. The switch inputs are routed to the physics unit, which uses the inputs to scale acceleration from gravity and acceleration from friction.

3.2.3 Synchronizer from Lab Two

The synchronizer module from lab two is used to synchronize button presses and external SPI signals. The reset button and pause button, for example, are routed through a synchronizer. Metastability issues may result if synchronizers are not used for external signals.

3.3 Pause FSM by Matt

A simple, two-state pause FSM takes a pause request and flips the pause state whenever a request is received. The pause signal is used within the game state FSM to pause gameplay.

3.4 SPI Synchronizer by Matt

The SPI synchronizer module acts as an interface for the external user components, sending the latest data from the UI device to the physics unit. The SPI synchronizer module uses the five-wire SPI protocol to communicate with the ADCs, and send the latest 16-bit word output from the ADC to the physics unit.

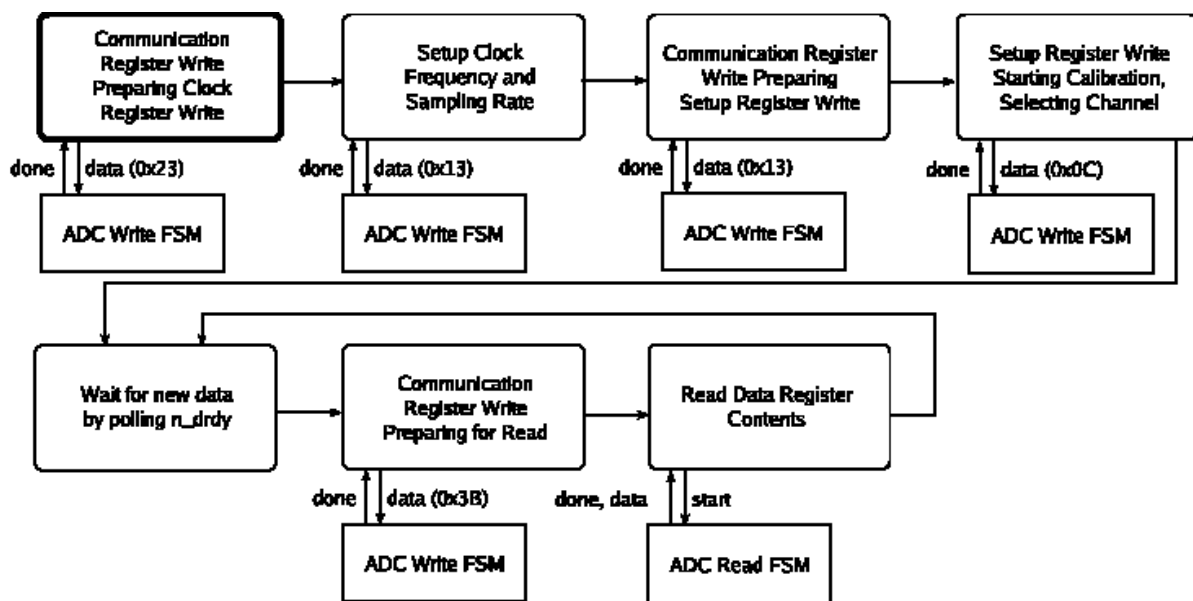


Figure 3: The SPI FSM. The FSM consists of an initialization routine followed by sampling.

3.4.1 SPI FSMs

The SPI Synchronizer module contains three finite state machines to communicate with an ADC: a main FSM, a minor write FSM and a minor read FSM. The main finite state machine oversees all communication with the ADC, controlling the minor FSMs. The main FSM writes initial values to the ADC to setup the ADC's operation, and then continually samples data from the ADC. The write FSM takes an eight bit word and controls the appropriate SPI signals to write the word to the ADC. The read FSM takes the data signals from the ADC and returns the latest 16-bit word output by the FSM.

When a read or write needs to occur, the main FSM signals the read or write FSM to begin communication with the ADC, and the minor FSM has control of all SPI signals until the minor FSM issues a done signal. After the done signal has been issued, the main FSM transitions to its next state and continues communication with the ADC. A read and write do not occur simultaneously in this design.

The data sheet of the ADC should contain appropriate information on the SPI protocol and timing constraints. All values used in this report are for the AD7706. A state diagram for the main FSM is included in Figure 3.

3.5 Physics Engine by Matt

The physics module computes the acceleration on the ball from gravity and friction, and then sends the x and y acceleration signals to the game state FSM. The x and y acceleration are 16-bit numbers corresponding to magnitude of acceleration in Earth g 's along with a sign bit corresponding to the direction of the acceleration. Acceleration from gravity is computed using the SPI synchronizer signals and a stored bias signal, while acceleration from friction is computed using the velocity from the game state FSM. The physics engine also takes four-bit, unsigned signals which scale the acceleration from gravity and acceleration from friction.

3.5.1 Biasing

The physics unit contains a simple FSM which waits for the x and y signals to settle and then stores a biased x and y acceleration signal. This biased signal corresponds to the default tilt of the user interface device, and the actual acceleration in either direction is computed from the difference between the current input signal and the bias.

3.6 Game State by Hongyi

The game state unit stores and updates the current state information of the game. It takes in x and y acceleration signals from the physics unit, x and y position, velocity, `inHole` and `targetReached` signals from the collision detection module, and the 1-bit `mask computed` signal from the level and collision detection masks. The unit outputs the ball's updated position, velocity and the new level number. If the mask has not been computed for the current level, gameplay is suspended until mask computation is complete.

The `inHole` and `targetReached` signals indicate whether a collision was detected for the ball and a trap hole or a target hole. If `inHole` is high, then the ball has fallen into a trap hole, so the position of the ball is reset to its initial position, and its velocity is reset to zero. If `targetReached` is high, then the ball has reached the target hole, so the current level number is incremented, and the ball's position and velocity are reset.

3.7 Collision Detection Unit by Hongyi

The wall collision unit takes the position and velocity signals for the ball from the game state unit. It sets up queries for the collision detection mask via the collision detection lookup module. The module outputs new position, velocity, `inHole` and `targetReached` signals for the ball based on the results of collision detection.

The ball has an x component and a y component of velocity. If the ball hits a wall along an axis, then its velocity component along that axis is set to 0. If the ball encounters a trap hole, then `inHole` is held high, and if the ball encounters the target hole, then `targetReached` is held high.

One advantage of this modularization is that if we want to implement different behavior for collisions, we can do so easily because this module would be the only piece of code we would have to change.

3.8 Collision Lookup Unit by Hongyi

The collision lookup unit takes the ball's position from the collision detection module and queries the collision detection mask to determine what collisions, if any, have occurred. The collision lookup unit contains a ten state FSM as shown in Figure 4.

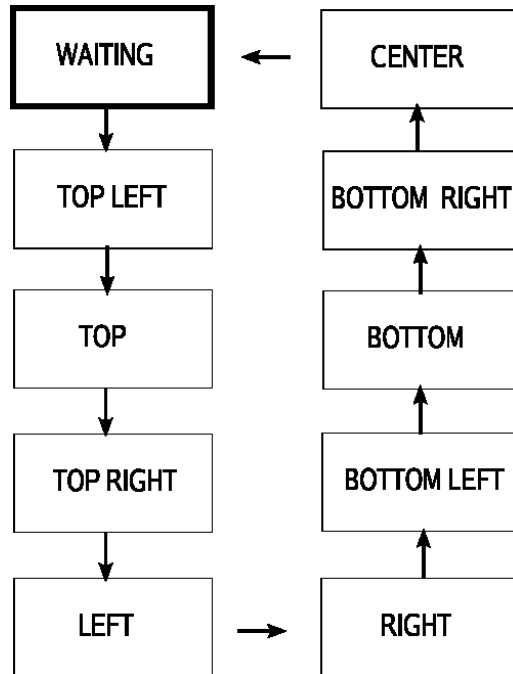


Figure 4: The Collision Lookup FSM. The FSM consists of a WAITING state followed by a series of memory lookup states.

Initially, the module is in the WAITING state, where it is waiting for new data from the collision detection unit. Once the collision detection unit has provided the ball’s updated position, the lookup unit loops through a series of nine states to query the collision detection mask for the contents of the block that holds the center of the ball and the eight blocks that surround that center block. At the end of this cycle, the module provides the results of the collision detection lookups and returns to the WAITING state.

3.9 Collision Detection Mask by Hongyi

The collision detection mask acts as a level cache for the main game memory. Both the collision detection logic and the display logic need to access the data in memory to perform their functions, so we implemented two masks, a collision detection mask and a level mask, to avoid interleaving memory lookups.

The collision detection mask stores the locations of all the walls and holes for one level. It contains a three-bit wide memory module with 3072 locations. Each location is associated with a specific block on the screen and contains a code indicating whether that block contains a wall, a trap hole, a target hole or nothing. The mask takes in the level number from the game state module so that when the level number changes, the mask holds `mask computed` low while it queries the game’s main memory to cache all the data for the next level. Once this process is complete, `mask computed` is held high.

3.10 Memory Interface by Hongyi

The memory interface takes access requests from the collision detection mask and the level mask during level transitions. The level mask has priority over the collision detection mask, so all of the level mask’s read requests are processed before the collision detection mask’s read requests.

3.11 Memory by Matt

The memory unit stores the locations of obstacles such as walls and holes for all levels in the game. The memory is composed of eight levels, with each level having 4096 fields. Each field is three bits, and represents a 16 by 16 block of pixels in a particular level. The three bit field corresponds to an object on the map - the codes for each map object can be found in Appendix A. In total, the memory is three bits wide with 2^{15} locations.

3.11.1 Level Generation

A python script generated a .coe file which contained level information to be loaded into memory. The input to the python script was a string of ASCII characters corresponding to walls, trap holes or target holes.

3.12 XVGA Unit from Lab 4

The XVGA unit is responsible for communication between the ADV7125 and the draw unit. The XVGA unit was identical to the unit from lab four, but relies on the draw unit instead of the pong_game unit.

3.13 Draw Unit by Matt

The draw unit is responsible for creating the pixel output the XVGA unit displays on the screen. The draw unit takes the x and y coordinates of the current ball position from the game state FSM and communicates with the current level mask to determine if any object is present at the current location. If a hole or ball exists at the current location, the draw unit communicates with the circle draw unit to determine what pixels are active to draw a circle. Based on whether any objects are present at the current location and when these objects should be drawn, the draw unit outputs what pixels should be colored and what colors these pixels should be.

3.13.1 Circle Draw Unit by Matt

The circle draw unit takes a the center and radius of a circle and returns whether or not the pixel currently being drawn is inside the circle. The circle draw unit takes at most one clock cycle to compute this information.

3.13.2 Level Mask by Matt

The level mask caches information about the topological layout of the current level. The level mask acts as a buffer for the memory to avoid draw unit latency and potential memory access conflicts with the FSM logic. Whenever the ball ends up in the target hole and moves onto a new level, the level mask communicates with the main memory to load the next level. Until the new level information is loaded from the main memory, the `mask_computed` signal is set low so that the game state FSM suspends gameplay. The level mask takes the current pixel being drawn from the draw unit and returns whether or not an object exists at that location, along with the object type if the object does exist.

4 Testing and Debugging

We generally tested our code by using individual test bench waveforms to test the functionality of a single module. After each module had been tested we would integrate our code on the labkit to test the functionality of our combined components. When a bug was encountered at this stage we used the logic analyzer to track down and fix the bug.

4.1 Frame Synchronization Bug

One of the more difficult bugs to track down was a frame synchronization bug with the ADCs. The ADCs use the `n_cs` signal to synchronize register writes and reads. It was incorrectly assumed that this signal's falling edge was required to start a write to the ADC, while in reality the ADC will continue to write bits to a register as long as the `n_cs` signal is low. The logic analyzer was of great use in tracking down this bug.

4.2 Collision Detection Timing Issue

Initially a simpler collision detection scheme was being used which checked only the four adjacent walls. A timing issue in the collision detection code caused the center of the ball to be checked instead of the left and right sides of the ball. This timing issue, coupled with a display bug causing the ball's center to be offset by a pixel, was difficult to track down due to the seemingly random nature of when collisions were being output. The timing bug was finally discovered when the waveforms for the collision detection unit were drawn with a closer look at the memory access latency.

4.3 Inferred Bit Flips

During the upgrade of the collision detection algorithm the ball would often stop seven or eight pixels from the wall. The error was caused by the synthesis tools behaving differently than expected. We compute a five-bit value by taking the bitwise complement of a four-bit value and adding one. What happened, however, was that the most significant bit of the five-bit value was always flipped to one, which caused an error in our collision detection calculations. We hypothesize that this was caused by Verilog casting the four-bit value to five-bit value before doing the bitwise complement. This bug was discovered after we ran the signals through the logic analyzer and saw some erroneous values.

5 Conclusion

In summary, we built an electronic tilting maze game based on the children's game Labyrinth. We used an accelerometer control device to feed input into the labkit via an SPI interface. The labkit maintained the state of the game and drove the movement and behavior of the game. The labkit also rendered the graphics using an X VGA interface. A Python script was used to generate levels for the game.

Our original design goals were to avoid complexity, to make our design scalable and to make our design easily mutable. Since we were able to add extensions easily to our project, we believe that we were fairly successful in meeting these objectives. For example, we implemented a collision detection scheme that was cleaner and far more efficient than our original solution. Due to the flexible modularization of our design, changes were required in only one module to upgrade the game from the old collision detection method to the new, superior one. We were also able to add pause functionality with few changes to existing code. Finally, because we reserved a number of unused memory codes, we could easily add new objects to the game. This can be accomplished with small changes to the collision detection module and the display logic.

We had hoped to eventually implement this design on a smaller FPGA, creating a unit that could simply be attached to an LCD, but after finishing the bulk of the project, we focused on implemented digital design changes and additions.

Acknowledgments

We would like to thank Professor Terman, Professor Chuang, and our TA Jae Lee for their expertise and help with our project. We would also like to thank Analog Devices for their generous donation of equipment for our project.

A Memory Field Codes

Code	Object
000	Nothing
001	Target Hole
010	Trap Hole
011	Wall
100 - 111	Reserved for Future Use

B Code

B.1 SPI_Synch.v

```
module SPI_sync(clock_65mhz, reset, mclk, sclk, n_cs, n_adc_reset,
               n_drdy, data_from_adc, data_to_adc, latest_word);
    input clock_65mhz;    //clock
    input reset;         //reset signal - resets ADCs and initiates correct settings
    input n_drdy;       //~data_ready - input whether new data available
    input data_from_adc; //input line for reading data from adc

    output n_cs;        //~chip select - frame sync signal for ADC
    output n_adc_reset; //~reset for adc
    input sclk;         //sclk - used to read data
    input mclk;         //master clock - used for ADC sampling
    output data_to_adc; //data that is going to the ADC
    output [15:0] latest_word;
    //output [4:0] debug;

    wire n_cs;
    wire data_from_adc;
    reg n_adc_reset;
    wire data_to_adc;
    wire mclk;
    wire sclk;

    wire data_to_adc_wwf;

    wire writing; //high when writing to adc
    wire reading; //high when reading to adc
    wire read_done; //when the read from adc is complete, goes high
    wire write_done; //when the write from adc is complete, goes high
    reg write_word; //signals start of writing
    wire read_word; //signals start of reading

    reg [5:0] wait_count; //simple count to wait

    wire [15:0] latest_word; //latest 16 bit word received from ADCs

    wire [7:0] data_to_write; //data to write to ADCs

    reg old_sclk; //old sclk value
    wire posedge_sclk = ~old_sclk & sclk; //high for a 65mhz clock cycle after sclk goes high
    wire negedge_sclk = old_sclk & ~sclk; //high for a 65mhz clock cycle after sclk goes low

    reg [4:0] state;
```

```

    reg [4:0] next_state; //used when entering a wait state
// may want to view page 32 of AD7706 data sheet for a state machine
// states --- 0 -> default state, reset adc
//           1 -> write to communications register value 0x20

//FSM works with two minor read/write fsms as described in lecture - when
//want to write a word to ADC, start writing, wait until done, and go to
//wait state
//
// the wait state isn't necessary, butthere are timing constraints involving
// when the write_FSM can start writing to the ADC (based on the clock), the wait state
// helps transition at the right time to ensure timing is met - these timing constraints
//

parameter state_DEFAULT = 0; //default state
parameter state_w0 = 1;      //wait state in-between writes
parameter state_c1 = 2;     //write to communication reg state 1 - setup to clock reg write
parameter state_c2 = 4;     //write to clock reg
parameter state_c3 = 6;     //write to communication reg state 2 - setup to setup reg write
parameter state_c4 = 8;     //write to setup reg

parameter state_EndSwitch = 9; //state noticing when new data available
parameter state_cREAD = 10;   //write to communication reg setting up reg
parameter state_StartSwitch = 11; //state entered when new data is loading
parameter state_READ = 12;    //state for reading data

parameter state_ERR = 13; //error state when setup communication doesn't seem to succeed

assign wait_done = (wait_count == 20); //when waiting is complete

parameter pWAIT = 7; //how long to wait in-between writes

assign data_to_write = (state == state_c1) ? 8'h23 : //communication reg setting up clock write
                       (state == state_c2) ? 8'h13 : //write to clock reg
(state == state_c3) ? 8'h13 : //communication reg setting up setup write
(state == state_c4) ? 8'h0C : //setup write
(state == state_cREAD) ? 8'h3B : 8'h5A;

assign n_cs = ~(writing || reading || (state == state_READ));
//note that n_cs must go low AS SOON AS WRITING CEASES
//i spent four days before i realized the ADC continues to clock bits in if you're
//trying to use n_cs as a synchronization signal to the ADCs
assign data_to_adc = (state == state_READ) ? 1 : data_to_adc_wwf;

always @ (posedge clock_65mhz) begin
    if (reset || ~n_adc_reset)
begin
state <= state_DEFAULT;
end
else
begin
case (state)
state_ERR: state <= (wait_count > pWAIT) ? state_DEFAULT : state_ERR;

```

```

//immediately transition to the waiting state to transition correctly for timing
state_DEFAULT: state <= state_w0;

        //wait until the posedge of sclk and the wait time is met, then move to next state
state_w0: state <= (wait_count > pWAIT) && posedge_sclk ? next_state : state;

//if drdy during a write, incorrectly setup ADC - transition to error state
// i tried debugging this issue for two days and couldn't find it - easier
// to add this additional state
//else if write done, move to wait state to setup next write
state_c1, state_c2, state_c3, state_c4: state <= ~n_drdy ? state_ERR :
        write_done ? state_w0 : state;

        //FSM stays in these four states to read forever

//if in the read state and the read is done, move to the state waiting for the
//adc to load new data
state_READ: state <= read_done ? state_StartSwitch : state;

//state to setup read from ADC
state_cREAD: state <= write_done ? state_w0 : state;

//state waiting for ADC to start moving new data
state_StartSwitch: state <= n_drdy ? state_EndSwitch : state;
//state waiting for ADC to end moving new data
state_EndSwitch: state <= ~n_drdy ? state_cREAD : state;
        default: state <= state_DEFAULT;
    endcase
    case (state)
        //set the next state before entering the wait state
//each communication state moves to next communication state
state_DEFAULT: next_state <= state_c1;
state_c1: next_state <= state_c2;
state_c2: next_state <= state_c3;
state_c3: next_state <= state_c4;
state_c4: next_state <= state_StartSwitch;
//setup of communication read moves to actually read state
state_cREAD: next_state <= state_READ;
        default: next_state <= next_state;
    endcase
    end
//old sclk value - used to detect sclk edges
old_sclk <= sclk;

//when to start writing a word
write_word <= ((state == state_c1) ||
        (state == state_c2) ||
(state == state_c3) ||
(state == state_c4) ||
(state == state_cREAD)) ? (posedge_sclk ? 1 : write_word) : 0;

        //wait_count increments when in the wait state or error state - else zero
wait_count <= (state == state_w0) || (state == state_ERR) ? wait_count + 1 : 0;

//reset the ADC in the error state, otherwise give the ADC the opposite of the global reset
n_adc_reset <= (state == state_ERR) ? 0 : negedge_sclk ? ~reset : n_adc_reset;

```

```

end

assign read_word = (state == state_READ); //when to start reading

word_read_fsm wrf(.clock_65mhz(clock_65mhz), .sclk(sclk), .reset(~n_adc_reset),
                 .read(read_word), .data_line(data_from_adc), .read_data(latest_word),
                 .done(read_done), .reading(reading));

word_write_FSM wwf(.clock_65mhz(clock_65mhz), .sclk(sclk), .reset(~n_adc_reset),
                 .write(write_word), .data_to_write(data_to_write), .data_line(data_to_adc_wwf),
                 .done(write_done), .writing(writing));

endmodule

```

B.2 cd_mask.v

```

module cd_mask(reset,clk_27,cd_new_data,level,x,y,vx_hat,vy_hat,data,
              center_collision_data,x_collision_data,y_collision_data,
              mask_new_data,cd_addr_x,cd_addr_y,cd_mask_computed);
input reset,clk_27; // clock and reset signals
input [2:0] level; // level number (from Game State module)
input signed [11:0] x,y; // x and y coords of ball
input vx_hat,vy_hat; // vx and vy sign bits (encodes direction of ball)
input [2:0] data; // data from memory module
input cd_new_data; // Is new data available from collision detection module?

// outputs to collision detection
output [2:0] center_collision_data; // collision data for center of ball
output [2:0] x_collision_data; // collision data for x-axis for ball
output [2:0] y_collision_data; // collision data for y-axis for ball
output mask_new_data; // inform the collision detection module that new data is available

reg [2:0] center_collision_data, x_collision_data, y_collision_data;

// outputs to memory interface
output [5:0] cd_addr_x, cd_addr_y; // x and y coords for addressing memory

// global
output cd_mask_computed; // Has this mask been computed for the current level?

reg [3:0] state;
reg [5:0] cd_addr_x, cd_addr_y;

// Parameters
parameter WAITING = 0;
parameter X_COL_P1 = 1;
parameter X_COL_P2 = 2;
parameter X_COL_P3 = 3;
parameter X_COL_P4 = 4;
parameter X_COL_P5 = 5;
parameter Y_COL_P1 = 6;
parameter Y_COL_P2 = 7;
parameter Y_COL_P3 = 8;
parameter Y_COL_P4 = 9;
parameter Y_COL_P5 = 10;
parameter C_COL = 11;

```



```

parameter LOADING = 12;

parameter BALL_R = 1;

assign cd_mask_computed = (state != LOADING);
assign mask_new_data = (state == WAITING);

// keep track of level number from previous cycle
reg [2:0] old_level;

reg [11:0] addr;
initial addr = 0;
reg [2:0] cache_data_in;
initial cache_data_in = 0;
wire [2:0] cache_data_out;
wire we;
assign we = (state == LOADING);

// instantiate the cache
cd_mask_memory cache(.addr(addr),.clk(clk_27),.din(cache_data_in),.dout(cache_data_out),.we(we));

// pixels we want to look at are:
// top/bottom: (x,y+BALL_R),(x,y-BALL_R)
// left: (x-BALL_R,y)
// right: (x+BALL_R,y)
// diagonal up left:
// diagonal right right:
// diagonal down left:
// diagonal down right:
wire [11:0] Y_MINUS_BALL_R, Y_PLUS_BALL_R, X_MINUS_BALL_R, X_PLUS_BALL_R;
wire [11:0] Y_MINUS_R_DIV2, Y_PLUS_R_DIV2, X_MINUS_R_DIV2, X_PLUS_R_DIV2;
assign Y_MINUS_BALL_R = y - BALL_R;
assign Y_PLUS_BALL_R = y + BALL_R;
assign X_MINUS_BALL_R = x - BALL_R;
assign X_PLUS_BALL_R = x + BALL_R;
assign Y_MINUS_R_DIV2 = y - (BALL_R>>1);
assign Y_PLUS_R_DIV2 = y + (BALL_R>>1);
assign X_MINUS_R_DIV2 = x - (BALL_R>>1);
assign X_PLUS_R_DIV2 = x + (BALL_R>>1);

wire [11:0] top_pixel,bottom_pixel,left_pixel,right_pixel,up_left_pixel,up_right_pixel,down_left_pixel,down_right_p
assign top_pixel = {x[9:4],Y_MINUS_BALL_R[9:4]};
assign bottom_pixel = {x[9:4],Y_PLUS_BALL_R[9:4]};
assign left_pixel = {X_MINUS_BALL_R[9:4],y[9:4]};
assign right_pixel = {X_PLUS_BALL_R[9:4],y[9:4]};
assign up_left_pixel = {X_MINUS_R_DIV2[9:4],Y_MINUS_R_DIV2[9:4]};
assign up_right_pixel = {X_PLUS_R_DIV2[9:4],Y_MINUS_R_DIV2[9:4]};
assign down_left_pixel = {X_MINUS_R_DIV2[9:4],Y_PLUS_R_DIV2[9:4]};
assign down_right_pixel = {X_PLUS_R_DIV2[9:4],Y_PLUS_R_DIV2[9:4]};

reg [11:0] pixel2,pixel3,pixel4,pixel5;
reg [2:0] pixel1_out,pixel2_out,pixel3_out,pixel4_out;

initial old_level = 0;
initial pixel1_out = 0;
initial pixel2_out = 0;

```

```

initial pixel3_out = 0;
initial pixel4_out = 0;
initial pixel2 = 0;
initial pixel3 = 0;
initial pixel4 = 0;
initial pixel5 = 0;
initial state = LOADING;
initial cd_addr_x = 0;
initial cd_addr_y = 0;
initial center_collision_data = 0;
initial x_collision_data = 0;
initial y_collision_data = 0;

always @(posedge clk_27) begin
old_level <= reset ? 0 : level;
if (reset) begin
pixel1_out <= 0;
pixel2_out <= 0;
pixel3_out <= 0;
pixel4_out <= 0;
pixel2 <= 0;
pixel3 <= 0;
pixel4 <= 0;
pixel5 <= 0;
state <= LOADING;
cd_addr_x <= 0;
cd_addr_y <= 0;
center_collision_data <= 0;
x_collision_data <= 0;
y_collision_data <= 0;
addr <= 0;
cache_data_in <= 0;
end
else begin
case (state)
WAITING: begin
if (cd_new_data) begin
state <= (old_level != level) ? LOADING : X_COL_P1;
pixel2 <= ~vx_hat ? up_right_pixel : up_left_pixel;
pixel3 <= ~vx_hat ? right_pixel : left_pixel;
pixel4 <= ~vx_hat ? down_right_pixel : down_left_pixel;
pixel5 <= bottom_pixel;
addr <= top_pixel;
end
end
X_COL_P1: begin
// get output from memory and send to collision detection
pixel1_out <= cache_data_out;

state <= (old_level != level) ? LOADING : X_COL_P2;
addr <= pixel2;

end
X_COL_P2: begin
// get output from memory and send to collision detection
pixel2_out <= cache_data_out;

```

```

state <= (old_level != level) ? LOADING : X_COL_P3;
addr <= pixel3;
end
X_COL_P3: begin
// get output from memory and send to collision detection
pixel3_out <= cache_data_out;

state <= (old_level != level) ? LOADING : X_COL_P4;
addr <= pixel4;
end
X_COL_P4: begin
// get output from memory and send to collision detection
pixel4_out <= cache_data_out;

state <= (old_level != level) ? LOADING : X_COL_P5;
addr <= pixel5;
end
X_COL_P5: begin
// get output from memory and send to collision detection

// determine output for x_collision_data
// check for outside border too
x_collision_data <= pixel1_out == 3 || pixel2_out == 3 || pixel3_out == 3 || pixel4_out == 3 || cache_data_out == 3

// setup for y axis collisions
state <= (old_level != level) ? LOADING : Y_COL_P1;
pixel2 <= ~vy_hat ? down_left_pixel : up_left_pixel;
pixel3 <= ~vy_hat ? bottom_pixel : top_pixel;
pixel4 <= ~vy_hat ? down_right_pixel : up_right_pixel;
pixel5 <= right_pixel;
addr <= left_pixel;
end
Y_COL_P1: begin
// get output from memory and send to collision detection
pixel1_out <= cache_data_out;

state <= (old_level != level) ? LOADING : Y_COL_P2;
addr <= pixel2;

end
Y_COL_P2: begin
// get output from memory and send to collision detection
pixel2_out <= cache_data_out;

state <= (old_level != level) ? LOADING : Y_COL_P3;
addr <= pixel3;

end
Y_COL_P3: begin
// get output from memory and send to collision detection
pixel3_out <= cache_data_out;

state <= (old_level != level) ? LOADING : Y_COL_P4;
addr <= pixel4;

```

```

end
Y_COL_P4: begin
// get output from memory and send to collision detection
pixel4_out <= cache_data_out;

state <= (old_level != level) ? LOADING : Y_COL_P5;
addr <= pixel5;

end
Y_COL_P5: begin
// get output from memory and send to collision detection

// determine output for x_collision_data
y_collision_data <= pixel1_out == 3 || pixel2_out == 3 || pixel3_out == 3 || pixel4_out == 3 || cache_data_out == 3;

state <= (old_level != level) ? LOADING : C_COL;
addr <= {x[9:4],y[9:4]};

end
C_COL: begin
// determine output for center_collision_data
center_collision_data <= cache_data_out;

state <= (old_level != level) ? LOADING : WAITING;

end
LOADING: begin
// get outputs from memory interface to write cache
addr <= {cd_addr_x,cd_addr_y};
cache_data_in <= data;

// supply memory interface with inputs
cd_addr_x <= (cd_addr_x == 2) ? 0 : (cd_addr_y == 2) ? cd_addr_x + 1 : cd_addr_x;
cd_addr_y <= (cd_addr_y == 2) ? 0 : cd_addr_y + 1;

// check if we are done
state <= (cd_addr_x == 1 && cd_addr_y == 2) ? WAITING : LOADING;

end
default: \${display("What state are we in?");
endcase
end // else
end // always @(posedge clk_27)

endmodule

```

B.3 cd_mask_dummy.v

```

module cd_mask_dummy(clk,reset,cd_new_data,x,y,vx_hat,vy_hat,data,
                    center_collision_data,x_collision_data,y_collision_data,
                    mask_new_data,cd_addr);
input clk, reset;
input signed [11:0] x,y; // x and y coords of ball
input vx_hat,vy_hat; // vx and vy sign bits (encodes direction of ball)
input [2:0] data; // data from the mask

```

```

input cd_new_data; // Is new data available from collision detection module?

// outputs to collision detection
output [2:0] center_collision_data; // collision data for center of ball
output [2:0] x_collision_data; // collision data for x-axis for ball
output [2:0] y_collision_data; // collision data for y-axis for ball
output mask_new_data; // inform the collision detection module that new data is available
reg mask_new_data;
reg [2:0] x_collision_data, y_collision_data, center_collision_data;

// outputs to memory interface
output [11:0] cd_addr; // x and y coords for addressing memory
wire [11:0] cd_addr;

// radius of ball
parameter BALL_R = 7;

// state machine's state
reg [3:0] state;

// Parameters
parameter WAITING = 0;
parameter TOP_LEFT = 1;
parameter TOP = 2;
parameter TOP_RIGHT = 3;
parameter LEFT = 4;
parameter RIGHT = 5;
parameter BOTTOM_LEFT = 6;
parameter BOTTOM = 7;
parameter BOTTOM_RIGHT = 8;
parameter CENTER = 9;

// util values
wire [4:0] DELTA_X, DELTA_Y, DELTA_X_COMP, DELTA_Y_COMP;
assign DELTA_X = x[3:0];
assign DELTA_Y = y[3:0];
assign DELTA_X_COMP = 16 - x[3:0];
assign DELTA_Y_COMP = 16 - y[3:0];

wire [5:0] Y_PLUS1, Y_MINUS1, X_PLUS1, X_MINUS1;
assign Y_PLUS1 = y[9:4] + 1;
assign Y_MINUS1 = y[9:4] - 1;
assign X_PLUS1 = x[9:4] + 1;
assign X_MINUS1 = x[9:4] - 1;

// blocks we want to look at are:
wire [11:0] top_block, bottom_block, left_block, right_block;
wire [11:0] top_left_block, top_right_block, bottom_left_block, bottom_right_block;
assign top_block = {Y_MINUS1,x[9:4]};
assign bottom_block = {Y_PLUS1,x[9:4]};
assign left_block = {y[9:4],X_MINUS1};
assign right_block = {y[9:4],X_PLUS1};
assign top_left_block = {Y_MINUS1,X_MINUS1};
assign top_right_block = {Y_MINUS1,X_PLUS1};
assign bottom_left_block = {Y_PLUS1,X_MINUS1};

```

```

assign bottom_right_block = {Y_PLUS1,X_PLUS1};

// collision results
reg top_col, bottom_col, left_col, right_col;
reg top_left_col, top_right_col, bottom_left_col, bottom_right_col;

initial mask_new_data = 0;
initial state = WAITING;

    assign cd_addr = (state == WAITING) ? top_left_block :
        (state == TOP_LEFT) ? top_block :
        (state == TOP) ? top_right_block :
        (state == TOP_RIGHT) ? left_block :
        (state == LEFT) ? right_block :
        (state == RIGHT) ? bottom_left_block :
        (state == BOTTOM_LEFT) ? bottom_block :
        (state == BOTTOM) ? bottom_right_block :
        {y[9:4],x[9:4]};

always @(posedge clk) begin
case(state)
WAITING: begin
mask_new_data <= 0;
center_collision_data <= mask_new_data ? center_collision_data : 0;
x_collision_data <= mask_new_data ? x_collision_data : 0;
y_collision_data <= mask_new_data ? y_collision_data : 0;
state <= cd_new_data ? TOP_LEFT : WAITING;
end
TOP_LEFT: begin
// get output from memory and send to collision detection
state <= reset ? WAITING : TOP;

// compute the outcome of the collision for this block
top_left_col <= ((data == 3) && (BALL_R*BALL_R >= DELTA_X * DELTA_X + DELTA_Y * DELTA_Y));
end
TOP: begin
// get output from memory and send to collision detection
state <= reset ? WAITING : TOP_RIGHT;

// compute the outcome of the collision for this block
top_col <= ((data == 3) && (BALL_R >= DELTA_Y)) || (y <= BALL_R);

end
TOP_RIGHT: begin
// get output from memory and send to collision detection
state <= reset ? WAITING : LEFT;

// compute the outcome of the collision for this block
top_right_col <= ((data == 3) && (BALL_R*BALL_R >= DELTA_X_COMP * DELTA_X_COMP + DELTA_Y * DELTA_Y));
end
LEFT: begin
// get output from memory and send to collision detection
state <= reset ? WAITING : RIGHT;

// compute the outcome of the collision for this block

```

```

left_col <= ((data == 3) && (BALL_R >= DELTA_X )) || (x <= BALL_R);
end
RIGHT: begin
// get output from memory and send to collision detection
state <= reset ? WAITING : BOTTOM_LEFT;

// compute the outcome of the collision for this block
right_col <= ((data == 3) && (BALL_R >= DELTA_X_COMP)) || (x + BALL_R >= 1024);
end
BOTTOM_LEFT: begin
// get output from memory and send to collision detection
state <= reset ? WAITING : BOTTOM;

// compute the outcome of the collision for this block
bottom_left_col <= ((data == 3) && (BALL_R*BALL_R >= DELTA_X * DELTA_X + DELTA_Y_COMP * DELTA_Y_COMP));

end
BOTTOM: begin
// get output from memory and send to collision detection
state <= reset ? WAITING : BOTTOM_RIGHT;

// compute the outcome of the collision for this block
bottom_col <= ((data == 3) && (BALL_R >= DELTA_Y_COMP)) || (y + BALL_R >= 768);

end
BOTTOM_RIGHT: begin
// get output from memory and send to collision detection
state <= reset ? WAITING : CENTER;

// compute the outcome of the collision for this block
bottom_right_col <= ((data == 3) && (BALL_R*BALL_R >= DELTA_X_COMP * DELTA_X_COMP + DELTA_Y_COMP * DELTA_Y_COMP));

end
CENTER: begin
// determine output for collision detection data signals
center_collision_data <= data;
x_collision_data <= vx_hat ? ((top_left_col || left_col || bottom_left_col) ? 3 : 0) :
((top_right_col || right_col || bottom_right_col) ? 3 : 0);
y_collision_data <= vy_hat ? ((top_left_col || top_col || top_right_col) ? 3 : 0) :
((bottom_left_col || bottom_col || bottom_right_col) ? 3 : 0);
state <= WAITING;

mask_new_data <= ~reset;

end
default: \${display("What state are we in?")};
endcase
end

endmodule

```

B.4 cd_mask_test.v

```

module cd_mask_test(reset,clk,level_from_fsm,addr,data,cd_mask_computed,
                    cdm_addr,data_from_mem);
input reset, clk;

```

```

input [2:0] level_from_fsm;
input [11:0] addr;
input [2:0] data_from_mem;
output [11:0] cdm_addr;
output [2:0] data;
output cd_mask_computed;

reg [11:0] count;
    reg [11:0] count_d1;
    reg [2:0] cur_level;

wire writing;

assign cdm_addr = count;
    assign writing = count_d1 != 3072;
    assign cd_mask_computed = ~writing;

always @(posedge clk) begin
cur_level <= reset ? 0 : level_from_fsm;
    count <= reset ? 0 : cur_level != level_from_fsm ? 0 : (count != 3072) ? count + 1 : count;
    count_d1 <= reset ? 0 : count;
end

ram_lm_ssx3 f( .addr( writing ? count_d1 : addr ), .clk(clk),
    .din(data_from_mem), .dout(data), .we( writing ) );

endmodule

```

B.5 collision_detection.v

```

module collision_detection(clk,reset,paused,in_x,in_y,in_vx,in_vy,
    cd_new_data,new_data,x_collision_data,
    y_collision_data,center_collision_data,
    mask_new_data,ball_x,ball_y,vx_hat,vy_hat,
    out_x,out_y,out_vx,out_vy,inHole,targetReached);
// This module is a 2 state machine:
// In the WAITING_NEW_DATA state, it has valid outputs to game_state and waits for
// new inputs from game_state on the next low to high transition of new_data.
// In the WAITING_CDETECTION state, it is querying the collision detection mask
// and handling collision detection computations.

// REMEMBER TO HANDLE COLLISIONS WITH BORDERS OF SCREEN!!!

// global input signals
input clk;
input reset;
input paused;

// inputs from game_state
input signed [31:0] in_x, in_y; // x,y coords of ball
input signed [31:0] in_vx, in_vy; // x,y velocity of ball
input new_data; // Is new data available from game_state?

// input from collision det mask

```



```

input [2:0] x_collision_data; // x collision data from the collision detection mask
input [2:0] y_collision_data; // y collision data from the collision detection mask
input [2:0] center_collision_data; // center collision data " "
input mask_new_data; // Is new data available from the mask?

// output to collision det mask
output signed [11:0] ball_x,ball_y; // ball's location
output vx_hat,vy_hat; // ball's velocity direction
output cd_new_data; // inform the mask that there is new data available
reg cd_new_data;

// output back to game_state and display logic
output signed [31:0] out_x,out_y;
output signed [31:0] out_vx,out_vy;
output inHole,targetReached;

reg signed [31:0] out_x,out_y;
reg signed [31:0] out_vx,out_vy;
reg inHole,targetReached;
reg [1:0] state;

parameter WAITING_NEW_DATA = 0;
parameter WAITING_CDETECTION = 1;
parameter RESET = 3;

assign vx_hat = (in_vx < 0);
assign vy_hat = (in_vy < 0);
assign ball_x = in_x[31:20];
assign ball_y = in_y[31:20];

parameter INITIAL_X = 32'b00000001100000000000000000000000;
parameter INITIAL_Y = 32'b00000001100000000000000000000000;

// initial state is WAITING
initial state = WAITING_NEW_DATA;
initial out_x = INITIAL_X;
initial out_y = INITIAL_Y;
initial out_vx = 0;
initial out_vy = 0;
initial inHole = 0;
initial targetReached = 0;
initial cd_new_data = 0;

//assign cd_new_data = reset ? 0 : (state == WAITING_CDETECTION);

always @(posedge clk) begin
case (state)
RESET: begin
inHole <= 0;
targetReached <= 0;
out_x <= INITIAL_X;
out_y <= INITIAL_Y;
out_vx <= 0;
out_vy <= 0;
state <= WAITING_NEW_DATA;
end
end

```

```

WAITING_NEW_DATA: begin
// check if new data is available from game_state
inHole <= reset ? 0 : inHole;
targetReached <= reset ? 0 : targetReached;
out_x <= reset ? INITIAL_X : out_x;
out_vx <= reset ? 0 : out_vx;

out_y <= reset ? INITIAL_Y : out_y;
out_vy <= reset ? 0 : out_vy;

cd_new_data <= reset ? 0 : new_data;

// change state
state <= reset ? RESET : (new_data ? WAITING_CDETECTION : WAITING_NEW_DATA);

end
WAITING_CDETECTION: begin
cd_new_data <= 0;
if (mask_new_data) begin
// handle inputs from collision det mask and set outputs to game_state
// first, deal with inHole and reachedTarget
inHole <= reset ? 0 : (center_collision_data == 2);
targetReached <= reset ? 0 : (center_collision_data == 1);

// second, deal with x axis
out_x <= reset ? INITIAL_X :
    paused ? in_x :
    (x_collision_data == 3) ? in_x : in_x+in_vx;
out_vx <= reset ? 0 :
    paused ? in_vx :
    (x_collision_data == 3) ? 0 : in_vx;

// finally, deal with y axis
out_y <= reset ? INITIAL_Y :
    paused ? in_y :
    (y_collision_data == 3) ? in_y : in_y+in_vy;
out_vy <= reset ? 0 :
    paused ? in_vy :
    (y_collision_data == 3) ? 0 : in_vy;

// change state
state <= reset ? RESET : WAITING_NEW_DATA;
end
end
default: \${display("What state are we in?");
endcase
end

endmodule

```

B.6 divider.v

```

// Clock Divider module
module divider(clk,reset,enable);
    input  clk,reset;
    output enable;

```

```

reg    [24:0] counter;

parameter TARGET = 5;

initial counter = 0;

always @(posedge clk) counter <= (reset || counter == TARGET) ? 0 : counter + 1;

// the enable is high when the counter counts up to target
assign enable = (counter == TARGET);
endmodule // divider

```

B.7 drawBallBool.v

```

module drawBallBool(hcount, vcount, b_x, b_y, b_r, ball_active);
    //takes in coordinates (hcount and vcount), along with circle information (radius b_r, centered
    //at (b_x, b_y), and returns whether or not the circle is active
    input [10:0] hcount;
    input [9:0] vcount;
    input [11:0] b_x;
    input [11:0] b_y;
    input [5:0] b_r;
    output ball_active;

    wire signed [11:0] b_x, b_y; //ball's coordinates
    wire [5:0] b_r;             //ball's radius
    wire signed [10:0] hcount;  //horiz. count
    wire [9:0] vcount;         //vert. count
    wire signed [10:0] ycount; //vert. count, adjusted for size and signed bit
    assign ycount = {1'b0, vcount};
    //wire [11:0] hMx, xMh, vMy, yMv, r_2;

    wire signed [11:0] d_x, d_y;
    wire signed [11:0] r_2;
    assign d_x = b_x - (hcount);
    assign d_y = b_y - ycount;
    assign r_2 = {6'b0, b_r} * {6'b0, b_r};

    assign ball_active = ((hcount < b_x - b_r ) || (hcount > b_x + b_r ) ||
                          (vcount < b_y - b_r ) || (vcount > b_y + b_r )) ? 0 :
    ((d_x * d_x) + (d_y * d_y)) < r_2;

    /*
    assign hMx = hcount - b_x;
    assign xMh = b_x - {1'b0, hcount};
    assign vMy = vcount - b_y;
    assign yMv = b_y - {2'b00, vcount};
    assign r_2 = {6'b000000, b_r} * {6'b000000, b_r};

    //assign ball_active = (((b_x - hcount)*(b_x - hcount)) < (b_r * b_r));
    wire ball_active;
    assign ball_active = ((hcount < b_x - b_r ) || (hcount > b_x + b_r ) ||
                          (vcount < b_y - b_r ) || (vcount > b_y + b_r )) ? 0 :
    (((xMh * xMh) + (yMv * yMv) < r_2) || ((hMx * hMx) + (yMv * yMv) < r_2) ||
    ((xMh * xMh) + (vMy * vMy) < r_2) || ((hMx * hMx) + (vMy * vMy) < r_2));
    //wire ball_active;

```

```

    //assign ball_active = ((hcount < b_x - b_r) || (hcount > b_x + b_r) ||
    //                      (vcount < b_y - b_r) || (vcount > b_y + b_r)) ? 0 : 1;
    */
endmodule

```

B.8 drawUnit.v

```

module drawUnit (vclock,reset,
    hcount,vcount,hsync,vsync,blank,
    ohsync,ovsync,oblank,ocolor, ball_x, ball_y, ball_r,
                mask_active, mask_color);
    input vclock; // 65MHz clock
    input reset; // 1 to initialize module

    input [10:0] hcount; // horizontal index of current pixel (0..1023)
    input [9:0] vcount; // vertical index of current pixel (0..767)
    input hsync; // XVGA horizontal sync signal (active low)
    input vsync; // XVGA vertical sync signal (active low)
    input blank; // XVGA blanking (1 means output black pixel)

    input mask_active;
    input [2:0] mask_color;

    input [11:0] ball_x;
    input [11:0] ball_y;
    input [5:0] ball_r;

    output ohsync; // game's horizontal sync
    output ovsync; // game's vertical sync
    output oblank; // game's blanking
    output [2:0] ocolor; // game's color pixel

    reg old_hsync, old_vsync, old_blank;

    //delay by a clock cycle
    assign ohsync = old_hsync;
    assign ovsync = old_vsync;
    assign oblank = old_blank;

    always @ (posedge vclock) begin
//latch old values
        old_hsync <= hsync;
        old_vsync <= vsync;
        old_blank <= blank;
    end

    /*
    assign ohsync = hsync;
    assign ovsync = vsync;
    assign oblank = blank;*/

    //parameters specifying display height, width
    //parameter dH = 768;
    //parameter dW = 1024;

```

```

//the pixel color to use
wire [2:0] ocolor;

wire mask_active;
wire [2:0] mask_color;

parameter ball_color = 3'b100;
wire ball_active;

drawBallBool fooBall(.hcount(hcount), .vcount(vcount), .b_x(ball_x),
                    .b_y(ball_y), .b_r(ball_r), .ball_active(ball_active));

assign ocolor = //(reset || ball_active) ? 3'b000 :
                (reset || ball_active) ? ball_color :
                mask_active ? mask_color : 3'b000;

endmodule

```

B.9 dynamicLevelMask.v

```

module dynamicLevelMask( clk, reset, hcount, vcount, level_from_fsm, data_from_mem,
                        mask_active, mask_color, mask_computed, lm_addr );
    input [10:0] hcount; // horizontal index of current pixel (0..1023)
    input [9:0] vcount; // vertical index of current pixel (0..767)
    input clk;
    input reset;
    input [2:0] level_from_fsm;
    input [2:0] data_from_mem;
    output mask_active;
    output [2:0] mask_color;
    output mask_computed;
    output [11:0] lm_addr;

    wire [11:0] lm_addr;
    wire [2:0] mask_color;
    wire [2:0] obj_at_loc;
    wire writing;
    wire mask_computed;

    wire circ_active;

    reg [11:0] count;
    reg [11:0] count_d1;
    reg [2:0] cur_level;

    parameter pWALL = 3'd3;
    parameter pTARGET = 3'd1;
    parameter pHOLE = 3'd2;
    parameter pHOLE_R = 6'd7;

    assign lm_addr = count;
    assign writing = count_d1 != 3072;
    assign mask_computed = ~writing;

    assign mask_color = obj_at_loc;

```

```

    drawBallBool dbb( .hcount(hcount), .vcount(vcount),
        .b_x({2'b00, hcount[9:4], 4'b1000}), .b_y({2'b00, vcount[9:4], 4'b1000}), .b_r(pHOLE_R),
        .ball_active(circ_active) );

    assign mask_active = (obj_at_loc == pWALL) ? 1 :
        (obj_at_loc == pTARGET) ? circ_active :
        (obj_at_loc == pHOLE) ? circ_active : 0;

    always @ (posedge clk) begin
        cur_level <= reset ? 0 : level_from_fsm;
    count <= reset ? 0 : cur_level != level_from_fsm ? 0 : (count != 3072) ? count + 1 : count;
    count_d1 <= count;
    end

    ram_lm_ssx3 f( .addr( writing ? count_d1 : {vcount[9:4], hcount[9:4]} ), .clk(clk),
        .din(data_from_mem), .dout(obj_at_loc), .we( writing ) );
endmodule

```

B.10 game_state.v

```

module game_state(reset, clk_27, sample_rate, x_acc, x_acc_hat,
    y_acc, y_acc_hat, mask_computed, new_x, new_y, new_vx,
    new_vy, inHole, targetReached, x, y, vx, vy, level, new_data);
// global input signals
input reset; //
input clk_27; // clock signal running at 27 MHz
input sample_rate; // speed that we run the game at

// physics input signals
input [15:0] x_acc, y_acc; // x and y acceleration data from physics module
// fractional
input x_acc_hat, y_acc_hat; // direction bits for x and y acceleration
// 0 pos, 1 neg

// mask input signals
input mask_computed; // has the mask for the current level been computed?

// collision input detection signals
input signed [31:0] new_x, new_y; // x and y position of ball from last time step
input signed [31:0] new_vx, new_vy; // x and y velocity of ball (micropixels/time_step)
input inHole; // Did the ball fall into a hole in the last time step?
input targetReached; // Did the ball reach the target in the last time step?

// outputs to collision detection unit
output signed [31:0] x, y; // current x and y position of ball
output signed [31:0] vx, vy; // new x and y velocity of ball (micropixels/time_step)
output [2:0] level; // current level number
output new_data; // signal Collision Detection that new data is available

// stored data
reg [2:0] level; // current level number
reg signed [31:0] x, y;
reg signed [31:0] vx, vy;
initial level = 0;

parameter INITIAL_X = 32'b00000010000000000000000000000000;

```

```

parameter INITIAL_Y = 32'b00100000000000000000000000000000;

initial x = INITIAL_X;
initial y = INITIAL_Y;
initial vx = 0;
initial vy = 0;

// conversion factor
// 9.8 m/s**2 * 102400000 micropixels/.0285 m * 1s**2/(8192 time_frame)^2 = 5246.9 micropixels/time_frame**2
parameter K = 5247;

// wires for computing the values of x, y, vx, and vy
// we run this logic at 8192 Hz, so shift right by 13
wire signed [31:0] computed_vx, computed_vy;
assign computed_vx = x_acc_hat ? new_vx - ((x_acc*K)>>13) : new_vx + ((x_acc*K)>>13);
assign computed_vy = y_acc_hat ? new_vy - ((y_acc*K)>>13) : new_vy + ((y_acc*K)>>13);
assign new_data = (mask_computed & sample_rate);

always @(posedge clk_27) begin
    // check if the mask has been computed
    if (mask_computed & sample_rate) begin
        level <= reset ? 0 : (targetReached ? level+1 : level);
        // figure out reset or starting positions for ball
        x <= (reset || inHole || targetReached) ? INITIAL_X : new_x;
        y <= (reset || inHole || targetReached) ? INITIAL_Y : new_y;
        vx <= (reset || inHole || targetReached) ? 0 : computed_vx;
        vy <= (reset || inHole || targetReached) ? 0 : computed_vy;
        end
    else begin
        level <= reset ? 0 : level;
        x <= reset ? INITIAL_X : x;
        y <= reset ? INITIAL_Y : y;
        vx <= reset ? 0 : vx;
        vy <= reset ? 0 : vy;
        end
end
endmodule

```

B.11 main.v

Irrelevant portions cut.

```

// feed XVGA signals to draw Unit
wire [2:0] pixel;
wire phsync,pvsync,pblank;

// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0] vcount;
wire hsync,vsync,blank;

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;

```

```

DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFV vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

// use FPGA's digital clock manager to produce a
// 2.4576MHz clock (actually 2.4545MHz)
//wire clock_245mhz_unbuf,clock_245mhz;
//DCM vclk3(.CLKIN(clock_27mhz),.CLKFX(clock_245mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk3 is 22
// synthesis attribute CLKFX_MULTIPLY of vclk3 is 2
// synthesis attribute CLK_FEEDBACK of vclk3 is NONE
// synthesis attribute CLKIN_PERIOD of vclk3 is 37
//BUFV vclk4(.O(clock_245mhz),.I(clock_245mhz_unbuf));

// power-on reset generation
wire power_on_reset; // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
.A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
// button 0 is adc reset
wire reset,user_reset, adc_reset;
debounce dbenter(power_on_reset, clock_65mhz, ~button_enter, user_reset);
debounce db0(power_on_reset, clock_65mhz, ~button0, adc_reset);
assign reset = user_reset | power_on_reset;

wire pause_request;
debounce db1(power_on_reset, clock_65mhz, ~button1, pause_request);

reg old_pause_request;
reg paused = 0;
wire pause_lpg = ~old_pause_request && pause_request;
always @ (posedge clock_65mhz) begin
    old_pause_request <= pause_request;
    paused <= reset ? 1 : pause_lpg ? ~paused : paused;
end

wire [7:0] sw;
debounce dbs[7:0]( {8{reset}} , {8{clock_65mhz}}, switch[7:0], sw[7:0]);

reg [2:0] rgb;
reg b,hs,vs;
always @(posedge clock_65mhz) begin
    /*
    if (switch[1:0] == 2'b01) begin
// 1 pixel outline of visible area (white)
hs <= hsync;
vs <= vsync;
b <= blank;
rgb <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 7 : 0;
    end else if (switch[1:0] == 2'b10) begin
// color bars

```



```

hs <= hsync;
vs <= vsync;
b <= blank;
rgb <= hcount[8:6];
    end else begin
*/
    // default: tilting maze game
hs <= phsync;
vs <= pvsync;
b <= pblank;
rgb <= pixel;
    /*end*/
end

wire mask_active;
wire [2:0] mask_color;

xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

wire [5:0] ball_r;
assign ball_r = 7;

wire signed [31:0] new_x, new_y;

drawUnit pg(clock_65mhz,reset,
hcount,vcount,hsync,vsync,blank,
phsync,pvsync,pblank,pixel,new_x[31:20], new_y[31:20], ball_r, mask_active, mask_color);

wire [2:0] level_from_fsm;
wire [2:0] data_from_mem;
wire mask_computed;
wire [11:0] lm_addr;
dynamicLevelMask slm( .clk(clock_65mhz), .reset(reset), .hcount(hcount), .vcount(vcount),
    .level_from_fsm(level_from_fsm), .data_from_mem(data_from_mem),
.mask_active(mask_active), .mask_color(mask_color), .mask_computed(mask_computed),
.lm_addr(lm_addr));

// VGA Output. In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.
assign vga_out_red = {8{rgb[2]}};
assign vga_out_green = {8{rgb[1]}};
assign vga_out_blue = {8{rgb[0]}};
assign vga_out_sync_b = 1'b1; // not used
assign vga_out_blank_b = ~b;
assign vga_out_pixel_clock = ~clock_65mhz;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;

wire sclk;
wire mclk;
wire n_cs;
wire n_adc_reset;
wire d_in;
wire n_drdy;
wire d_out;
wire [15:0] latest_word;

```

```

synchronize sn_drdy(.clk(clock_65mhz),.in(user1[2]),.out(n_drdy));
//assign n_drdy = user1[2];
synchronize sn_d_out(.clk(clock_65mhz),.in(user1[1]),.out(d_out));
//assign d_out = user1[1];

wire t_n_cs;
wire t_n_adc_reset;
wire t_d_in;
wire t_n_drdy;
wire t_d_out;
wire [15:0] t_latest_word;
synchronize st_n_drdy(.clk(clock_65mhz),.in(user3[2]),.out(t_n_drdy));
synchronize st_n_dout(.clk(clock_65mhz),.in(user3[1]),.out(t_d_out));
//assign t_n_drdy = user3[2];
//assign t_d_out = user3[1];

wire [4:0] debug;

// wires are:
//7: debug
//6: sclk
//5: mclk
//4: ~cs (should hard-wire low)
//3: ~reset
//2: ~drdy
//1: d_out (ADC produces this signal)
//0: d_in (ADC consumes this signal)

assign user1 = {24'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ, 1'b1, sclk, mclk, n_cs, n_adc_reset, 2'bZZ, d_in};
assign user3 = {24'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ, 1'b1, sclk, mclk, t_n_cs, t_n_adc_reset, 2'bZZ, t_d_in};

wire x_sign;
wire y_sign;

wire [15:0] x_bias;

// clock divider
wire clock_8192hz;
divider clk_divider(.clk(clock_65mhz), .reset(reset), .enable(clock_8192hz));
defparam clk_divider.TARGET = 7935;

// FSM logic. We need to run it at 8192kHz
wire [15:0] x_acc,y_acc;
//assign y_acc = 16'b0000000000000000;
//assign x_acc = {11'b000000000000,sw[1:0],3'b000};
wire x_acc_hat, y_acc_hat;
//assign x_acc_hat = sw[2];
//assign y_acc_hat = 0;

wire new_data;
wire signed [31:0] new_vx, new_vy;
wire signed [31:0] x, y, vx, vy;
wire cd_mask_computed;

game_state gs(.reset(reset), .clk_27(clock_65mhz), .sample_rate(clock_8192hz),

```

```

    .x_acc(x_acc), .y_acc(y_acc), .mask_computed(mask_computed && cd_mask_computed),
    .new_x(new_x), .new_y(new_y), .new_vx(new_vx), .new_vy(new_vy),
    .inHole(inHole), .targetReached(targetReached), .x(x), .y(y),
    // .inHole(inHole), .targetReached(1'b0), .x(x), .y(y),
    .vx(vx), .vy(vy), .level(level_from_fsm), .new_data(new_data),
    .x_acc_hat(x_acc_hat), .y_acc_hat(y_acc_hat));

    wire [2:0] cd_data, center_collision_data, x_collision_data, y_collision_data;
    wire mask_new_data, cd_new_data;
    wire [11:0] cd_addr, cdm_addr;
    wire signed [11:0] ball_x, ball_y;
    wire vx_hat, vy_hat;
    // wire inHole, targetReached;
    wire [11:0] addr;
    wire [3:0] state;

    collision_detection cd(.reset(reset), .clk(clock_65mhz), .paused(paused), .in_x(x), .in_y(y),
        .in_vx(vx), .in_vy(vy), .new_data(new_data),
        .center_collision_data(center_collision_data),
        .x_collision_data(x_collision_data),
        .y_collision_data(y_collision_data),
        .ball_x(ball_x), .ball_y(ball_y), .vx_hat(vx_hat), .vy_hat(vy_hat),
        .cd_new_data(cd_new_data), .out_x(new_x), .out_y(new_y),
        .out_vx(new_vx), .out_vy(new_vy), .inHole(inHole), .targetReached(targetReached),
        .mask_new_data(mask_new_data));

    cd_mask_dummy mask(.reset(reset), .clk(clock_65mhz), .x(ball_x), .y(ball_y), .cd_new_data(cd_new_data),
        .center_collision_data(center_collision_data),
        .x_collision_data(x_collision_data),
        .y_collision_data(y_collision_data),
        .data(cd_data), .mask_new_data(mask_new_data), .cd_addr(cd_addr),
        .vx_hat(vx_hat), .vy_hat(vy_hat));

    cd_mask_test mask_test(.reset(reset), .clk(clock_65mhz), .level_from_fsm(level_from_fsm),
        .addr(cd_addr), .data(cd_data), .cd_mask_computed(cd_mask_computed),
        .cdm_addr(cdm_addr), .data_from_mem(data_from_mem));

    memory_interface foobarmem(.clk_65(clock_65mhz), .reset(reset), .mask_computed(mask_computed),
        .level(level_from_fsm), .lm_addr(lm_addr), .cdm_addr(cdm_addr),
        .data(data_from_mem));

    physics_module pmod(.clock(clock_65mhz), .reset(reset | adc_reset), .scale_acc(sw[7:4]), .scale_f(sw[3:0]),
        .x_in(latest_word), .x_v(vx), .x_sign(x_acc_hat), .x_out(x_acc),
        .y_in(t_latest_word), .y_v(vy), .y_sign(y_acc_hat), .y_out(y_acc), .y_bias(x_bias) );

    flip_divider onefd(.clk_in(clock_65mhz), .clk_out(mclk), .height(20'd32), .reset( 1'b0 ) );
    flip_divider fourfd(.clk_in(clock_65mhz), .clk_out(sclk), .height(20'd8), .reset( 1'b0 ) );//|| n_cs );

    SPI_sync s_sync(.clock_65mhz(clock_65mhz), .reset(reset | adc_reset), .mclk(mclk), .sclk(sclk), .n_cs(n_cs),
        .n_adc_reset(n_adc_reset),
        .n_drdy(n_drdy), .data_from_adc(d_out), .data_to_adc(d_in),
        .latest_word(latest_word));

    SPI_sync t_s_sync(.clock_65mhz(clock_65mhz), .reset(reset | adc_reset), .mclk(mclk), .sclk(sclk), .n_cs(t_n_cs),

```

```

        .n_adc_reset(t_n_adc_reset),
        .n_drdy(t_n_drdy), .data_from_adc(t_d_out), .data_to_adc(t_d_in),
        .latest_word(t_latest_word));

// leds
//assign led = ~{vx_hat,cd_mask_computed,mask_computed,power_on_reset,user_reset,reset,switch[1:0]};
//assign led = ~{sw,power_on_reset,user_reset,reset,switch[1:0]};

// logic analyzer
//assign analyzer1_clock = clock_65mhz;
//assign analyzer3_data[3:0] = x_collision_data;
//assign analyzer4_data[3:0] = y_collision_data;

// hex digits display module
wire [63:0] display_input;
assign display_input = {ball_x,ball_y,36'h0,state};
//assign display_input = 64'hDEADBEEFDEADFACE;
display_16hex hex(reset,clock_27mhz,display_input,disp_blank, disp_clock, disp_rs, disp_ce_b,disp_reset_b, disp_

//assign led = ~{3'b0, n_cs, n_drdy, user_reset, power_on_reset, reset};
//assign led = ~latest_word[15:8];
assign led = ~{paused, pause_request, y_acc[5:0]};

assign analyzer1_clock = clock_65mhz;
assign analyzer1_data = x_bias;//{8'b0, d_in, d_out, n_drdy, n_adc_reset, n_cs, mclk, sclk};
assign analyzer2_data = {user1[7:0], 5'b0, power_on_reset, reset, clock_27mhz};
assign analyzer3_data = t_latest_word;
assign analyzer4_data = y_acc;
//assign analyzer3_data = {8'b0, 1'b0, d_in, d_out, n_drdy, n_adc_reset, n_cs, mclk, sclk};

endmodule

```

B.12 memory_interface.v

```

module memory_interface(clk_65,reset,mask_computed,level,lm_addr,cdm_addr,data);
input reset,clk_65,mask_computed; // clock, reset, mask_computed (from level mask)
input [2:0] level; // current level number
input [11:0] lm_addr; // x and y from Level Mask
input [11:0] cdm_addr; // x and y from Collision Detection Mask

output [2:0] data; // data for supplied x and y

wire [14:0] addr;
assign addr = mask_computed ? {level,cdm_addr} : {level,lm_addr};

// instantiate memory
mem_main_8lv1x3 f( .addr( addr ), .clk(clk_65),
                  .din(3'b000), .dout(data), .we( 1'b0 ) );

endmodule

```

B.13 physics_module.v

```

module physics_module( clock, reset, scale_acc, scale_f,
                      x_in, x_v, x_sign, x_out, y_in, y_v, y_sign,

```

```

        y_out, y_bias );
input clock;
input reset;
input [3:0] scale_acc, scale_f;

input [15:0] x_in;
input [31:0] x_v;
input [15:0] y_in;
input [31:0] y_v;

output [15:0] x_out;
output [15:0] y_out;
output x_sign;
output y_sign;

output [15:0] y_bias;          //debugging output that goes to logic analyzer but is unused

wire [3:0] scale_acc, scale_f;
wire signed [31:0] x_v, y_v;    //signed velocity wires for friction
wire [15:0] x_in, y_in;        //latest x and y words from ADCs
wire [15:0] x_out, y_out;      //output magnitude
wire x_sign, y_sign;          //sign of magnitude

reg [15:0] old_x_in, old_y_in; //old x and y ins - used for biasing accelerometers

reg [15:0] x_bias, y_bias;     //bias for signals

reg [2:0] state;              //state - only used during startup/reset to bias
parameter state_RESET = 7;
parameter state_BIAS = 2;
parameter state_READ = 1;

reg signed [15:0] x_acc_mf, y_acc_mf; //x and y acc. minus friction

//signal from friction
wire signed [31:0] x_acc_ff, y_acc_ff, x_acc, y_acc;

//assign x_acc_ff_tmp = ((x_v > 0) ? x_v : -x_v);
//assign y_acc_ff_tmp = ((y_v > 0) ? y_v : -y_v);
//acc from friction is applied in the opposite direction of the velocity sign

//parameter frictionC = 4; //amount to multiply friction coming into the unit
parameter shiftC = 14; //bits to shift out of the acceleration
parameter acc_scale_const = 8;

assign x_acc_ff = (x_v * scale_f);
assign y_acc_ff = (y_v * scale_f);

assign x_acc = x_acc_mf * acc_scale_const * scale_acc + x_acc_ff; // + x_acc_ff;
assign y_acc = y_acc_mf * acc_scale_const * scale_acc + y_acc_ff; // + y_acc_ff;

assign x_sign = (x_acc > 0);
assign y_sign = (y_acc > 0); // (y_in < y_bias);

assign x_out = (x_sign ? x_acc : - x_acc) >> shiftC;

```

```

assign y_out = (y_sign ? y_acc : - y_acc) >> shiftC;

//
// Takes the latest data and converts said data to acceleration
// deals with reset/calibration by watching
//
// will eventually pass through a LPF and HPF to obtain spikes in data, too

always @ (posedge clock) begin
if (reset)
begin
state <= state_RESET;
old_x_in <= x_in;
old_y_in <= y_in;
x_bias <= 0;
y_bias <= 0;
x_acc_mf <= 0;
y_acc_mf <= 0;
end
else if ((old_x_in != x_in) && (old_y_in != y_in)) begin
case (state)
state_RESET: state <= state - 1;
7,6,5,4,3,2: state <= state - 1;
state_READ: state <= state_READ;
default: state <= state_RESET;
endcase

old_x_in <= x_in;
old_y_in <= y_in;
x_bias <= (state == state_BIAS) ? x_in : x_bias; //if in the biasing state, store the bias
y_bias <= (state == state_BIAS) ? y_in : y_bias;

//compute signals without friction (purely from ADCs)
x_acc_mf <= ((x_in[15:8] == x_bias[15:8]) || (state != state_READ)) ? 0 :
(x_in - x_bias);

y_acc_mf <= ((y_in[15:8] == y_bias[15:8]) || (state != state_READ)) ? 0 :
(y_bias - y_in);
end
end

endmodule

```

B.14 word_read_fsm.v

```

module word_read_fsm(clock_65mhz, sclk, reset, read, data_line,
read_data, done, reading);
input clock_65mhz; //main clock
input sclk; //serial clock
input reset; //module reset
input read; //read signal
input data_line; //data line

output [15:0] read_data; //data (latched) to write
output done; //high when done
output reading;

```

```

//output state;
//output data;

reg [4:0] state;          //current state
reg [15:0] read_data;    //last read word - held so doesn't glitch while reading
reg [15:0] data;        //currently read in data - changes every clock cycle when reading
wire done;              //when reading is done
reg old_sclk;

wire sclk;
wire posedge_sclk = ~old_sclk & sclk;
//wire negedge_sclk = old_sclk & ~sclk;
wire reading;
//wire data_line;

/*
Current state of the word reader. Note that states 0 to 15 are guaranteed to occur
when reading that bit.
state: 18 - finished, output done until read lowers
      * 17 - waiting to start reading, default value
      16 - pseudo state before first sclk negedge
      15 - reading bit 15
      14 - reading bit 14
      ...
      0 - reading bit zero

state starts at 16, when a write signal is received, FSM latches data_to_write and moves to state 15
the appropriate bit is moved onto the data_line on the next negedge of sclk
the state diagram transitions from 15->14, 14->13, etc. on positive edges of sclk
*/
parameter state_DONE = 18;
parameter state_WAITING = 17;
parameter state_PSEUDO = 16;

assign reading = (state < state_WAITING);

always @ (posedge clock_65mhz) begin
    old_sclk <= sclk;
if (reset)
    begin
        state <= state_WAITING;
data <= 0;
    end
else
    begin
        case (state)
            state_DONE: state <= ~read ? state_WAITING : state;
            state_WAITING: state <= read ? state_PSEUDO : state; // ? state_w1 : state;
state_PSEUDO: state <= 15;          //delay a state to deal with posedge_sclk issues
                                //wait state negates this issue for write FSM

                                //read in appropriate values
                                1,2,3,4,5,6,7,8,9,10,11,12,13,14,15: state <= posedge_sclk ? state - 1 : state;
//transition to done state from last value
                                0: state <= posedge_sclk ? state_DONE : state;
                                default: state <= state_WAITING;
        endcase
    end
end

```

```

        endcase
        //data is synchronous to deal with inferred latch warnings from xilinx
        data <= ((state < state_PSEUDO) && posedge_sclk) ? (data << 1) + data_line : data;
    end
    read_data <= reset ? 0 : (state == state_DONE) ? data : read_data;
end

    assign done = (state == state_DONE);

endmodule

```

B.15 word_write_FSM.v

```

module word_write_FSM(clock_65mhz, sclk, reset, write,
                    data_to_write, data_line, done, writing);
    input clock_65mhz;          //main clock
    input sclk;                 //serial clock
    input reset;                //module reset
    input write;                //write signal
    input [7:0] data_to_write;  //data (latched) to write

    output data_line;          //data line
    output done;               //high when done
    output writing;
    //output state;

    reg [3:0] state;
    reg [7:0] data;
    reg old_sclk;
    wire done;
    wire posedge_sclk = ~old_sclk & sclk;
    wire negedge_sclk = old_sclk & ~sclk;
    reg data_line;
    wire writing;

    /*
    Current state of the word writer.
    state: 10 - finished, output done for a clock cycle
           * 9 - waiting to start writing, default value
           8 - pseudo state before first sclk negedge
           7 - writing bit seven
           6 - writing bit six
           ...
           1 - writing bit one
           0 - writing bit zero

           state starts at 16, when a write signal is received, FSM latches data_to_write and moves to state 15
           the appropriate bit is moved onto the data_line on the next negedge of sclk
           the state diagram transitions from 15->14, 14->13, etc. on positive edges of sclk
    */
    parameter state_DONE = 11;
    parameter state_DELAY = 10;
    parameter state_WAITING = 9;
    parameter state_PSEUDO = 8;

```



```

assign done = (state == state_DONE);
assign writing = (state < state_WAITING);

always @ (posedge clock_65mhz) begin
    old_sclk <= sclk;
if (reset)
    begin
        state <= state_WAITING;
data <= 0;
    end
else
    begin
        case (state)
            state_DONE: state <= ~write ? state_DELAY : state;
            state_DELAY: state <= state_WAITING;// ? state_c1 : state;
            state_WAITING: state <= write ? state_PSEUDO : state;// ? state_w1 : state;
            1,2,3,4,5,6,7,8: state <= negedge_sclk ? state - 1 : state;
            0: state <= posedge_sclk ? state_DONE : state;
            default: state <= state_WAITING;
        endcase
        //latch data when starting to write
        data <= ((state == state_WAITING) && write) ? data_to_write : data;
    end
//data_line is synchronous as feedback not in combinational blocks
data_line <= (state < state_PSEUDO) ? data[state[2:0]] : data_line;
end

endmodule

```