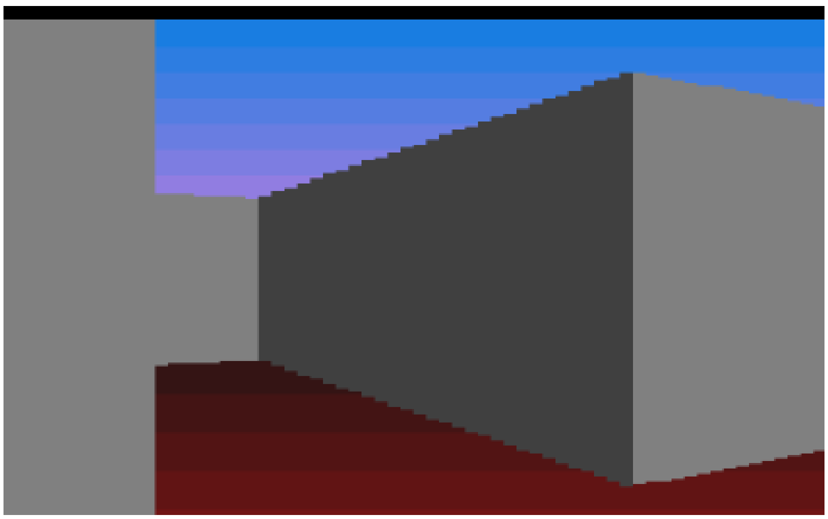
Labyrinth

Get in the Maze



Laplie Anderson and Mihalis Papalampros

6.111 – Introductory Digital Systems Lab

December 13, 2006

Abstract

Labyrinth is a 3D maze game. The goal of the game is to reach the center of the map without getting lost. To complicate things, players start the game in a random location on a randomly generated map. The player aided by only what they see in front of them, and a minimap that shows their general location on the map.

Labyrinth was programmed using Verilog and the Xilinx tool-chain. Labyrinth runs on the 6.111 Labkit displaying the game at 1024x768 resolution.

Table of Contents

List of Figures

1. Overview

Labyrinth is a 3D maze game designed in Verilog. Players are presented with a 3D view of their location in the maze and must use this information along with a sparse minimap to navigate. The goal of the game is to reach the center of the map. There is at least one path to the center and the player must find that path. To complicate things, the starting location of the player and the map are both randomly generated at the start of each game. As a result, the replayability of the game is increased dramatically.

1.1 Initialization

When the game begins, it is in its initialization state. In this state, the game is randomly generating maps and then checking to see if the generated map is a valid map. When a valid map is found, the game begins. You can generate a new random map by pressing the start button at any time.

1.2 User Interface

The game is controlled using the buttons and switches on the Labkit. Using the forward and back buttons, the player steps forward or steps backward in the general direction that they are looking. Using the left and right buttons, the player turns left or turns right. Due to time constraints, the player does not move in the exact direction that they are looking. The current implementation only distinguishes between eight distinct directions of movement though the player is able to see a full 360 degrees.

In addition, there is a start and a reset button. The reset button puts the game into the initialization state and tells all modules to stop what they are doing. The start button is very similar to the reset button in that the game begins again. The difference is that the start button only generates a new map without initiating a global reset. The player location changes to a start position, and the map changes, but other modules are not told to reinitialize. There is not a noticeable difference to the player between start and reset, but there is a difference for diagnostic purposes.

The minimap is controlled by two switches. Switch0 toggles whether or not the minimap is displayed on the screen. Switch1 toggles ‘cheat mode.’ In cheat mode, the location of the walls is displayed on the minimap. With this extra information, the game in instantly winnable. Therefore, in an actual game, the use of ‘cheat mode’ is strongly discouraged.

1.3 Winning

The game is over when the player reaches the center of the map. The player is free to continue exploring the map or pressing 'start' to restart the game.

2. Module Description and Implementation

Labyrinth is divided into two major parts: the map generation and game logic, and the graphics display system. Each part has various modules responsible for a specific tasks described later in this document. The interconnections between the modules are shown in Figure 1.

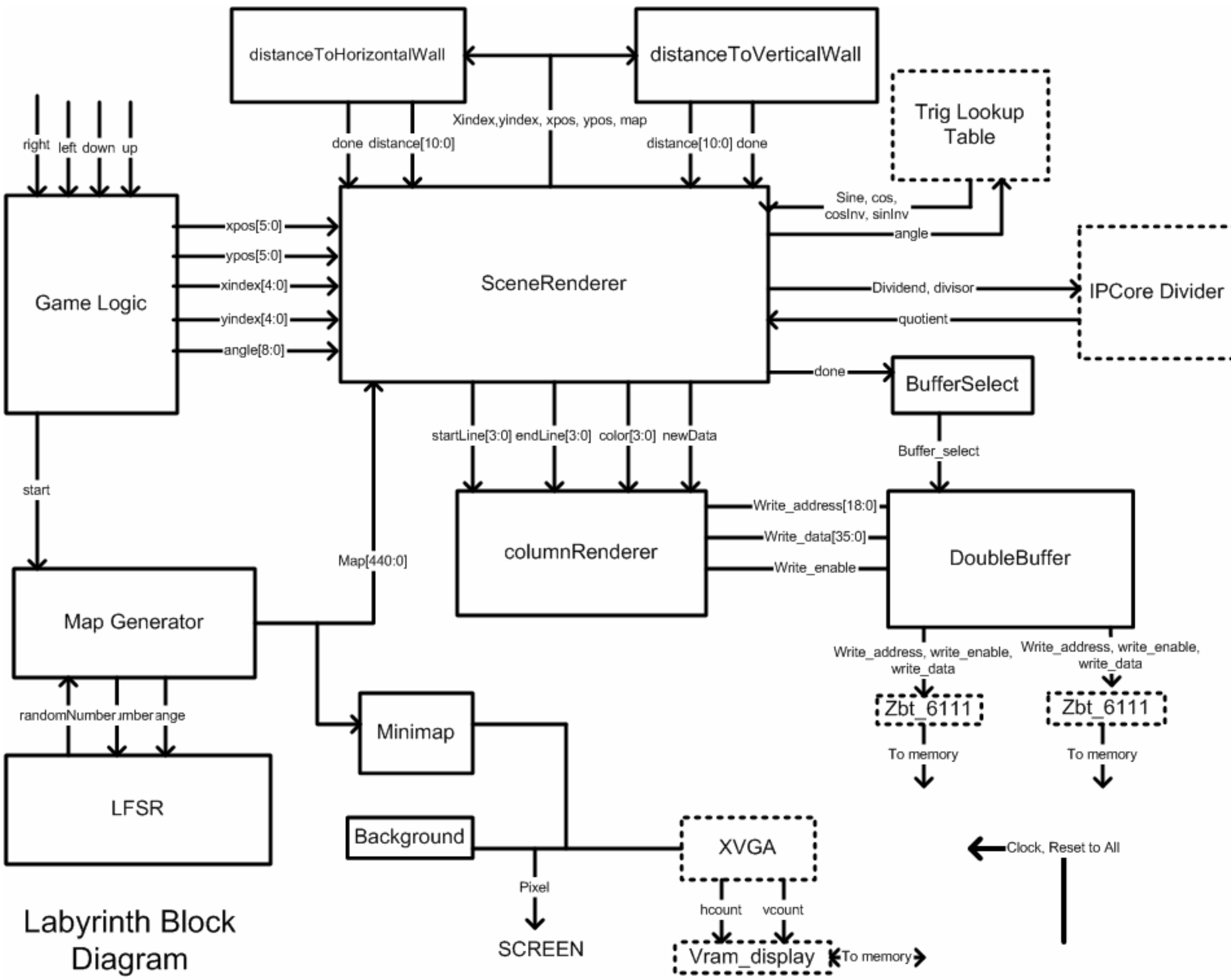


Figure 1: Block diagram showing the connections and interfaces to the major modules of Labyrinth. The modules with the dotted outline are modules either provided by the 6.111 staff or generated modules from IPCore. Each directed arrow is an information flow from one major module to another. The labels for the arrows are the Verilog names for the given wires.

2.1.1 Map Generator

The Map Generator creates a random 21x21 map for the labyrinth each time start is pressed. It has as inputs the clock, the start signal, and reset. The outputs are the map, which consists of 441 bits, and the done signal. Due to the fact that we cannot have two dimensional arrays in Verilog, the representation of the labyrinth map must be done through a one dimensional array of binary digits, with one corresponding to a wall and zero to empty space. That means that the first twenty one digits correspond to the first row, the next twenty one bits to the second and so on. In the very beginning, namely right after the compilation of the whole project, reset should be pressed in order to have the appropriate initializations of the registers. Then each time start is pressed setup goes high and done is set to low.

With that condition, that is with setup being high and done being low, the main core of the code inside the always statement starts. We first set the reset signals of the lfsrs to be low again since the reset is done once for every lfsr. That happens because an lfsr should be reset once in order to produce its pseudorandom sequence of numbers.

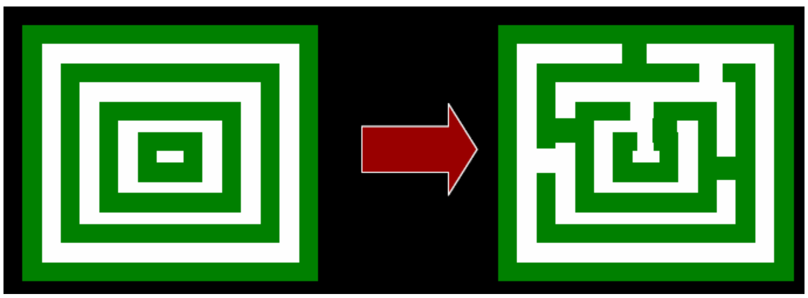


Figure 2. Map Generation technique. The goal is to go from the initialization picture on the left to the maze on the right.

The first step of the always block is the initialization. The initialization is actually creating concentric squares of empty spaces and walls, that is all zeros or ones, alternately. We could do that with three nested loops, one for k, the number of the square, one for j, the number of the row we are working on, and one for i, the number of the column. The best way to do that is through a finite state machine with four states. The first state deals with the k loop, that is increases the k right after j reaches 21, the second state deals with the j loop , and the third state deals with the i loop, where the assignments take place. We check if the pair (i,j) is on the k square and if it is we assign map[21*j+i], which corresponds to map[i,j] in two dimensions, to be 0 for even numbered squares and 1 for odd numbered squares. When k reaches 11 we are done with the initialization so we set the loopstate to be 3, in which state we set the scene for the next major step of the always statement, where we randomly add and remove walls from the concentric squares. That is we set initialization to be low, k to be zero, start1 and start2, the signals for the lfsrs to be high.

The second part deals with each box almost separately, grouping some cases together. For k being zero/nine we add/erase one wall, for k being one/two we erase/add two walls, for k being from three to eight, the remaining values of k, we add/remove four walls from the corresponding squares. Each time we wait until all lfsrs are done. Whenever an lfsr is done we set its start to be low in order to prevent this lfsr from continuing changing the random number. When all of the lfsrs are done we decide which edge the change will take place from the value of the odd numbered random numbers. Then we do the appropriate assignment for map[21*j+i], substituting j or i with the value k or 20-k, where each of the four combinations (i<=k, j<=k, i<=20-k, j<=20-k) corresponds to one of the four edges. The other index will take the value of an even numbered random number. After the assignments we increment k and we also start the lfsrs we need for the next value of k. When k reaches eleven we set the scene for the last major part, the checking part.

The checking uses two kinds of directions. The major direction is clockwise or counterclockwise and the minor is if we go to the right, left, down, or up. Hence, there are eight different cases dealt separately. The method consists of the following idea. When we enter a certain square we update the i1,j1 indexes which tell the spot where we entered the last square. Then we go clockwise or counterclockwise to find an opening. If we reach a dead end, we act according to the value of last_chance. If it is high, it means we have checked the other direction so there is no opening to the next square and hence no path to the center. If it is low, we set last_chance to be high and go back to the (i1,j1) spot taking the other major direction. Of course, we have to take the opposite minor direction. Eventually if we have k to be eight and we are at the point where we enter the next empty-spaced square we are done, since that is the center of the maze.

```
000000000000000000000
011111111111111111110
010000000000000000010
010111111111111111010
010100000000000001010
010001111111111101010
010101000000000101010
010100011111110101010
010101010000010101010
010111000111010101010
010101010001010101010
111101111111010101010
010101010000010101010
010101011111110101010
```

```
000101000000000101010
010101111111111101010
010100000000000001010
010111111111111111010
010000000000000000010
011111111111111111110
000000000000000000000
```

Figure 7: Internal Representation of Map. The internal bit representation of the map aligned so that it can be viewed.

2.1.2 Linear shift register

The lfsr has as inputs the clock, the reset and the start signals and also the range of the random number, which is given through the maximum and the minimum the number can be. It outputs the number and the done signal. The LFSR method, which can be found in many sources, takes an arbitrary initial state of the five digits, not all zeros (in our case we just chose the state corresponding to thirteen). Then it just shifts the digits to the right except for the most significant bit, which is the exclusive or of the pivot elements, which for the case of five digits numbers are the first and the fourth digits. In our case we also had to check if the number is within the range and to include states. The second case is when we actually try to produce a new random number. The first state is when we have found it and we wait until the start signal is again high. That modification had to be made due to the structure of the second part of the map generation module, where any of the lfsrs has to stop working as soon as it has found the random number and wait until all the others are done.

2.1.3 Game Logic

The Game Logic module is rather simple. If we press turn left or right we just increase or decrease the angle. The initial angle zero corresponds to facing east. Now if we press up we have to decide in which of the eight adjacent boxes to go depending on where we face. The directions are eight and correspond to angles 0,2*256,4*256, 6*256,8*256,10*256,12*256,14*256,16*256. Therefore, we choose the closest direction to the angle so we make our decision depending on which of the intervals [15*256,1*256], [1*256,3*256], [3*256,5*256], [5*256,7*256] [7*256,9*256] [9*256,11*256] [11*256,13*256], [13*256,15*256] we are.

At each of the eight directions we check if we are at the corresponding edges, both of the small boxes (the ones of the 64*64 lattices) and the boxes of the map. If we are at an edge of a small box we have to check if there is a wall in the position we want to move to. In the simple case where we are inside a box we just need to upgrade the xpos, ypos indexes, which are indexes for the 64x64 lattices.

2.2 Video System (Laplie Anderson)

The goal of the video system is to present players with a 3D view of the world. The picture is generated through an implementation of the ray-casting algorithm. The raycasting algorithm consists of a few simple steps outlined in figure 3.

```
For every angle in view:
     Trace line from player's eye to wall
     Scale wall based on this distance
     Draw column to screen
```

Figure 3. The ray tracing algorithm.

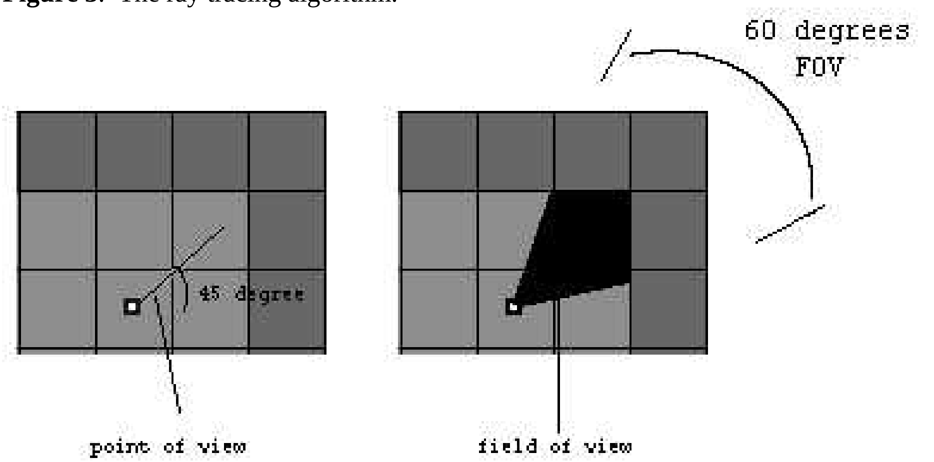


Figure 4. The visible field of a player. This also highlights the block of rays that have to be traced to properly display an image. The gradiency of the angle between subsequent rays is up to the implementer.

There are a few caveats to using recasting, most notably is that all walls have to be perpendicular to all floors. While this is not a major concern on a system like this, if we wanted to add more complexity in that regard, it would not be possible with the current algorithm.

While raycasting might be easy to implement on a software system, using the FPGA presents many difficulties involving dealing with fixed point numbers, generating sine, cosine, and their multiplicative inverses, and lastly dividing arbitrary numbers. The modules composing the video subsystem help perform these tasks.

2.2.1 MiniMap

The MiniMap is a small representation of the map in the top left of the screen. It is composed of 3 layered rectangle sprites which comprise the minimap border, the minimap background, and the minimap player icon. Using algebra, the minimap updates the position of the player icon to properly show where the player is in relation to everything else.

In cheat mode, the minimap modules uses more complicated algebra to figure out what a particular vcount and hcount correspond to in terms of the map as a whole. The module then looks up that index in the map, and if that index contains a wall, it shows an alternate color.

All of the various possible pixels are layered in a priority queue. The backmost layer outputs a pixel to the layer on top, if that layer does not have a pixel to output, it outputs the pixel that it received, otherwise it outputs its own pixel. In this way, the module is able to efficiently output the correct pixels for the whole module without and glitching occurring.

2.2.2 SceneRenderer

SceneRenderer is the main module of the video system. It is responsible for gathering and dispersing data throughout the other modules in the video system. It is controlled by an FSM that designates which task it is currently on. (See Figure 5).

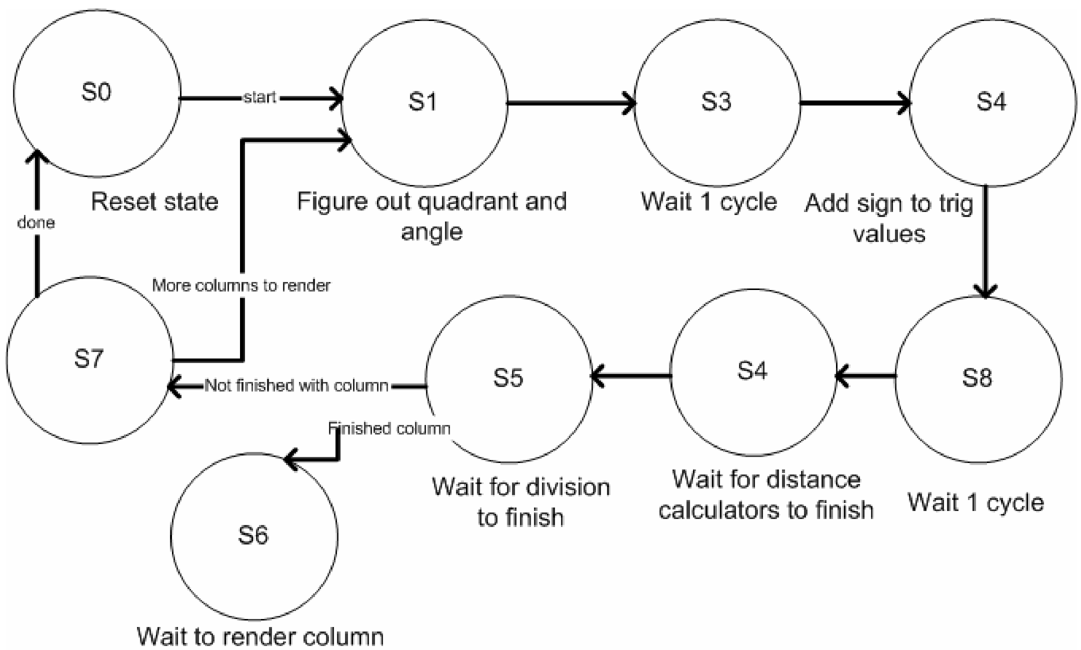


Figure 5. The FSM of SceneRenderer. Almost all of the states go sequentially to the next state. Most of the states are waiting states: waiting for other modules to complete.

The FSM does not have much logic to control which state is the next since in most cases, there is only one possibility. To simplify the transitions further, many of the states are simply the SceneRenderer waiting for another module to finish.

SceneRenderer first figures out the angle to trace by multiplying the column on the screen to be traced. This multiplication factor was found by dividing the view angle into 1024 separate columns and using that to determine the offset.

The next step is retrieving the trigonometric data. This task is done by waiting a cycle. The lookup table outputs the correct value with one-cycle latency so a one-clock delay is all that's needed.

Afterwards, the trigonometric data is fed into the distance calculators. SceneRenderer idles in S4 until both distance calculators complete their task and return with values. The closer of the two distances is given to the divider as the denominator.

After the divider finishes, SceneRenderer checks to see if all four lines of a column have been traced. If all four lines have been traced, then the column is drawn. If not, SceneRenderer continues with the next lines of the column

When all columns are drawn, SceneRenderer outputs a done signal.

2.2.3 Trigonometric Lookup Table

The lookup table was needed in order to get the sine and cosine values as well as 1/sine and 1/cosine. These values were needed for the distance calculators. Because of the cyclic nature of the sine and cosine, only 90 degrees were stored in the lookup table to save space. This turned out to be more effort than it was worth because there were no end to problems caused by sine and cosine having the wrong sign or simply being incorrect.

The .coe file for the lookup table was created with a 20 line Java program which simply wrote out the sign and cosine of all the angles from 0 to 90, converted them to binary then added the commas where appropriate.

2.2.4 DistanceToHorizontalWall/DistanceToVerticalWall

The two distance calculators are responsible for figuring out where the players' line of sight hits a wall. One of the restrictions in designing the map was that everything was on a grid system. Because of this restriction, finding where the line of sight hits the wall is just a matter of calculating a few offsets and then iterating through the gridlines of the map.

For its operation, the distance calculators have four states. In the first state, the distance calculators are idle and waiting to receive data. They know there is new data when the newData signal transitions high. Afterward, the distance calculators initialize all the registers and calculate the initial offsets.

Third, the distance calculators cycle through the gridlines using trigonometry and semi-complex equations to calculate where the next gridline is. When the distance calculators hit a wall they return the distance to that wall, and if they never hit a wall, they return MAX_DISTANCE.

The two distance calculators are mirrors of each other differing only in terms of whether sine or cosine is used and rotations between using the x or y coordinate.

2.2.5 DividerWrapper

DividerWrapper is a wrapper for the IPCoregen pipelined divider. For my purposes, I did not need to use the pipelining at all. The time between successive divides was much greater than the latency. To hide the pipelining from SceneRenderer, this module takes the inputs to the Coregen module, and then latches the answer on the correct cycle. It then asserts a done signal so that modules waiting for the result know that the result is ready without having to count cycles themselves. In situations where pipelining is desired, use of this module would be a was of ~20 cycles/divide.

2.2.6 ColumnRenderer

ColumnRenderer is responsible for draw a single vertical column to the buffer. Each column consists of four distinct lines. Since there are 1024 total lines, and 4 lines make a column, there are 256 separate columns that can be drawn. For each line, a starting point and an ending point are supplied. ColumnRenderer cycles through all the addresses in the column and either draws the color if its within the bounds of the column or draws black if it is not.

ColumnRenderer starts when the newData is asserted. When ColumnRenderer is finished, it asserts the done signal.

2.2.7 DoubleBuffer

The DoubleBuffer controls which ZBT is read from and which ZBT is written. In essences it is just a big multiplexer between the ZBT inputs and outputs, and the read and write addresses and data.

Inside DoubleBuffer also contains a vram_display module for reading from the ZBT and writing to the screen. The reason this module is necessary is that the ZBT has a latency of three clock cycles. The internal module deals with pipelining issues so that they are not a problem when outputting the pixel to the display.

The main complication in building the DoubleBuffer is that the system runs on multiple clocks. The slow clock controls most of the logic while the fast clock displays graphics to the screen. Since the DoubleBuffer is the only module that uses both clocks, it is the only one that has to deal with both clocks at the same time. The clocks are simply multiplexed like the rest of the ZBT signals with the slow clock going to the writeZBT while the fast clock goes to the readZBT.

2.2.8 BufferSelect

This module manages the flipping of the back buffer and the front buffer. When it receives a done signal from SceneRenderer, it changes which buffer is set to be read and which buffer is set to be written to.

On the next screen refresh, the most current buffer is read instead of the previous one.

3. Testing

The modules were primary tested using ModelSim on the computer rather than directly. ModelSim was immediately able to tell if the problem was with logic, rather than delays due to routing on the labkit and also had a much quicker turnaround time that a full compile. Thirdly, some modules that are functional blocks of other modules cannot be tested directly on the labkit and must be instead tested separately.

One of the difficult issues with ModelSim was the fact that the IPCore modules were not initially available for simulation. In order to get ModelSim to correctly simulate the behavior, extra libraries had to be compiled using obscure ModelSim codes.

3.1 MiniMap/Background

These modules, unlike almost all the others were tested only by displaying them. Because of their simple nature, it was very easy to spot problems and come up with solutions to them.

3.2 DividerWrapper

The main issue with the DividerWrapper module was figuring out the latency of the IPCore module. Though the datasheet specified a latency based on how many bits wide the divisor and dividend were, it was not clear whether or not the result was ready on that cycle or the cycle after. Using ModelSim, it was easy to determine the lowest possible delay before the result of the division was ready.

3.5 DistanceToHorizontalWall/DistanceToVerticalWall

These modules were tested first by insuring that they worked for a few representative angles and representative locations on a demo map, then checking how the scene was rendered with everything connected. For the first part of testing, all the values of sine and cosine were created by hand using a calculator. Using pen and paper, the actual distances were calculated, then these results were compared to those in shown in ModelSim.

This process was extremely tedious as a result of the way fixed bit arithmetic works. The answers in ModelSim were all multiplied by 256 so a lot of converting back and forth from binary and grungy multiplication resulted especially when signed numbers were involved.

3.6 DoubleBuffer

This module was tested directly on the labkit. Using a switch to change the active and inactive buffer, I drew various patterns to the unseen buffer and then switched it. While the back buffer was being drawn, I made certain that nothing changed in what was being displayed. After trying many combinations of writing and switching, I decided that the DoubleBuffer did in fact work.

3.7 ColumnRenderer

Like the DoubleBuffer module, the ColumnRenderer module was tested in a live setting. Using the switches to select the size and position, I was able to see the effect of drawing columns at various locations on the screen. This testing also further tested the DoubleBuffer since columns were always drawn to the inactive buffer. Since there are only 256 total columns, it was relatively easy to exhaustively test almost every column possibility.

3.8 SceneRenderer

The bulk of the problems seen in this module were a result of problems in other modules specifically the distance calculators returning the incorrect value because they were faulty or they received incorrect trigonometric data. Since it was difficult to trace problems directly to this module, testing it usually required tracking 30+ signals at once. (See Figure 6).

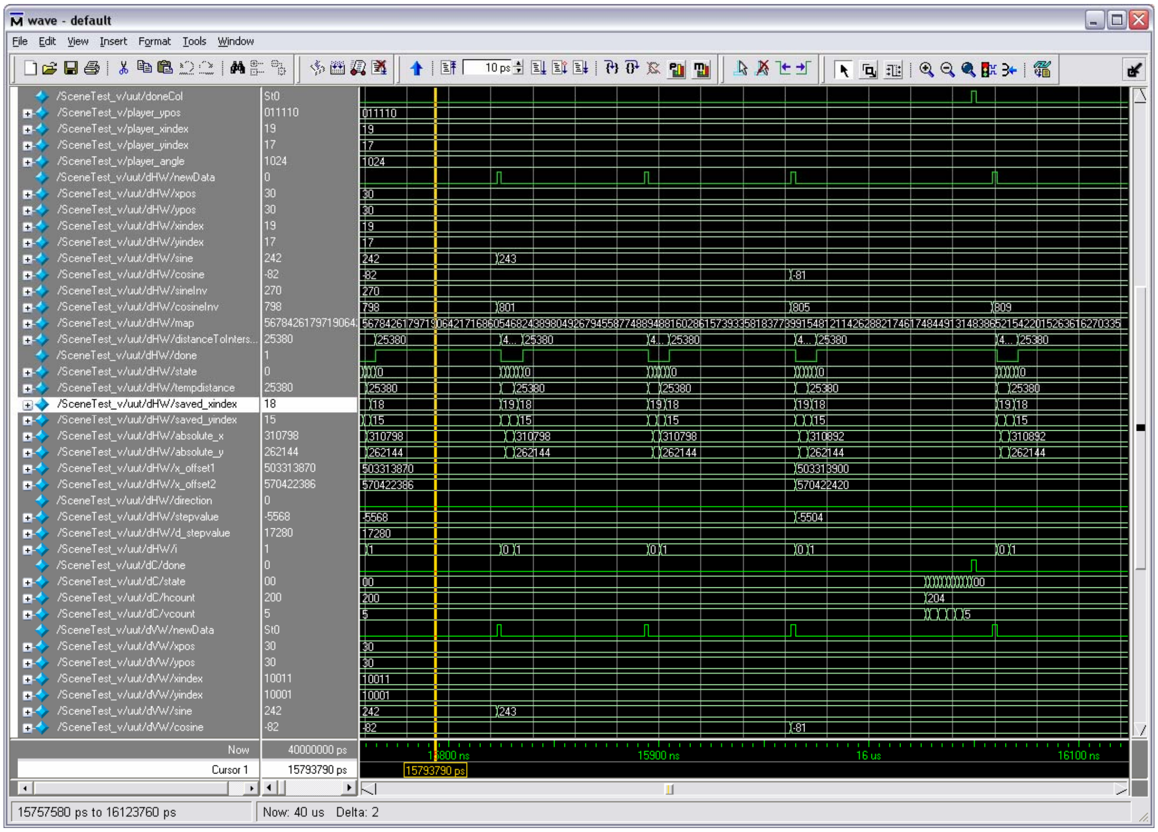


Figure 6. A common screen seen when debugging SceneRenderer

Even with complete track of all the signals, it still too very long to track down any errors. Some of the modules were modified to make testing in the manner easier. For example, ColumnRenderer was modified so that in only drew the first 5 rows of a column so that the test bench would not have the show the full 768.

The ModelSim testing and "live" testing shared almost an equal amount of time towards the end of the project because nearing the completion of the project, it took an extraordinary amount of time to compile the Verilog and generate a bitfile.

By looking at states, start, and done signals, the effort required to debug in ModelSim was reduced. In live testing, it was always very easy to see what was wrong, but it was not always as easy to see why. Errors such as having gaps in the walls could have any number of causes from a bug in the distance calculator to dividing by zero.

4.1 Conclusion

This project was a good glimpse into the amount of time and effort it takes when designing a digital system on a large scale. Even when the logic is completely accurate, the world is not digital so there are problems caused by routing delays and problems caused not meeting timing constraints. Though in the end the project worked, there was still room for improvement in a few areas. More bits of precision could have been used for distance calculations and as a result, it would have meant less jagged edges.

Overall the project was a success and we were very satisfied with the result in the end.

Appendix: Verilog Code

```
/**
  Controls the mode the doublebuffer is in and
  thereby controls buffer the SceneRender writes to as well as the
  buffer that is displayed

  Laplie Anderson
*/
module BufferSelector(clock, reset, buffer_select, scene_done);

	input reset;
	input clock;
	input scene_done;

	output buffer_select;

	reg buffer_select = 0;
	always @ (posedge clock) begin
		if (reset) begin
			buffer_select <= 0;
		end
		else if (scene_done)
			buffer_select <= ~buffer_select;
	end


endmodule

////////////////////////////////////////////////////////////////////////////
///
//Displays the sky and floor
//
//Laplie Anderson
module background(hcount, vcount, pixel);
   input [10:0] hcount;
   input [9:0] vcount;

   output [7:0] pixel;

//	wire [9:0] bottom;

	//assign bottom = vcount - 512;
	//assign pixel = vcount < 254 ? {3'b111 - vcount[7:5], 5'b11111} :
{{2{bottom[8]}}, bottom[7:5], {3{bottom[8]}}};
	assign pixel = vcount < 383 ? {2'b00, 3'b000, 3'b000} : {2'b10, 3'b101,
3'b101};

endmodule

/*  This module draws a vertical column to the video buffer.  The "column
number" is where horizontally
    on the screen the column should be drawn.  The start and end position
specify where vertically the
       column should be colored.  Everywhere between the start and end lines
are colored the input color,
       everywhere else is colored black.

       Laplie Anderson
```

```
*/
module columnRenderer(clock, reset, newData,
                                          columnNumber,
                                          startLine0, endLine0, startLine1,
endLine1,
                                          startLine2, endLine2, startLine3,
endLine3,
                                          color0, color1, color2, color3,
done,
                                          write_address, write_data,
write_enable);

       input clock, reset;
       input newData;         //this is raised to high when there is a new
column to be written
       input [7:0] columnNumber;  //index of the 4-pixel wide column. can be
from 0-255
       input [7:0] color0, color1, color2, color3;                 //color of
the column to be written
       input [9:0] startLine0, endLine0, startLine1, endLine1, startLine2,
endLine2, startLine3, endLine3; //vertical start and end positions for all 4
columns rendered


       output write_enable;   //    memory control signals
       output [18:0] write_address;
       output [35:0] write_data;

       output done; // raised high when done working

      reg [1:0] state =0;
      reg [10:0] hcount;
      reg [9:0] vcount;
      reg [9:0] intStartLine0, intEndLine0, intStartLine1, intEndLine1,
intStartLine2, intEndLine2, intStartLine3, intEndLine3;
      reg [7:0] columnColor0, columnColor1, columnColor2, columnColor3;
      reg done, write_enable;

      reg [18:0] write_address = 0;
      reg [35:0] write_data = 0;

      always @ (posedge clock) begin
            if (reset) state <= 0;
            else begin
                  case (state)
                  0: begin      //idle state
                        write_enable <= 0;
                        done <= 0;
                        if (newData) begin  //if there is newData, save the
inputs and go to next state
                               state <= 1;

                               hcount <= columnNumber << 2; //there are only
256 columns so we multiply by 4
                               intStartLine0 <= startLine0;
                               intEndLine0 <= endLine0;
                               intStartLine1 <= startLine1;
                               intEndLine1 <= endLine1;
                               intStartLine2 <= startLine2;
                               intEndLine2 <= endLine2;
```

```
                                intStartLine3 <= startLine3;
                                intEndLine3 <= endLine3;
                                columnColor0 <= color0;
                                columnColor1 <= color1;
                                columnColor2 <= color2;
                                columnColor3 <= color3;
                                vcount <= 0;
                         end
                  end
                  1: begin  //writing state
                         done <= 0;
                         write_address <= {1'b0, vcount, hcount[9:2]};
                         write_enable <= 1;
                         //write 4 pixels of the column color if between start
and endpoints
                         write_data <=

                         ((vcount >= intStartLine0 && vcount <= intEndLine0)
      ? columnColor0 << 24 : 0) +
                         ((vcount >= intStartLine1 && vcount <= intEndLine1)
      ? columnColor1 << 16 : 0) +
                         ((vcount >= intStartLine2 && vcount <= intEndLine2)
      ? columnColor2 << 8 : 0) +
                         ((vcount >= intStartLine3 && vcount <= intEndLine3)
      ? columnColor3 << 0 : 0);

                         vcount <= vcount + 1;
                         state <= 2;
                  end
                  2: begin //check if finished state.
                         if (vcount >= 768) state <= 3;
                         else state <= 1;

                         write_enable <= 0;
                  end
                  3: begin //finished state
                         write_enable <= 0;
                         write_data <= 0;
                         write_address <= 0;
                         done <= 1;
                         state <= 0;
                  end
                  endcase
            end
      end

endmodule

/*
      Calculates the distance until a horizontal face of a wall is hit
      after the signal "newData" is asserted high.

      The inputs sine and cosine are of the form 1QN:  1.8
      the least 8 significant bits are below the decimal place.  To convert
from a decimal
      to the correct sine and cosine, multiply by 256 then convert the
integer part to binary
```

```
	The inputs sineInv and cosineInv should always be positive with the 
lowest 8 bits being below
	the decimal place as sine and cosine.  These values represent 1/sine 
and 1/cos respectively

	When finished, this module asserts done to be high.  The last 
calculated value is
	distanceToIntersection.  The lowest 8 bits of this value represent the 
fractional part of the answer.

	distanceToVerticalWall mirrors this module.

	Laplie Anderson
*/

module distanceToHorizontalWall(clock, reset, newData,
							xpos, ypos, xindex, 
yindex,
							sine, cosine,
							sineInv, cosineInv,
							map,
distanceToIntersection, done);

localparam TILE_SIZE = 64 * 256;  //the 256 here is because the first 8 bits 
are below the decimal point
localparam MAX_DISTANCE = 1910 *256;  //the max distance possible on a 21x21 
is slightly less than this

input clock;
input reset;
input newData;
input signed [6:0] xpos; //signed for multiplication only.  Should be 6-bit 
values
input signed [6:0] ypos;
input [4:0] xindex;
input [4:0] yindex;

input signed [9:0] sine;
input signed [9:0] cosine;

// abs(1/sine) and abs(1/cosine)
// These are signed so that the multiplication works, but they should always 
be positive
input signed [17:0] sineInv;
input signed [17:0] cosineInv;

input [440:0] map;

output [18:0] distanceToIntersection;
output done;

reg done = 0;
reg [1:0] state = 0;

reg [31:0] distanceToIntersection;
reg [31:0] tempdistance;

reg [4:0] saved_xindex = 1;
reg [4:0] saved_yindex = 1;
reg [5:0] saved_xpos = 1;
```

```
reg [5:0] saved_ypos = 1;


reg signed [19:0] absolute_x;  //signed to make add/subtract much easier
without checks
reg signed [19:0] absolute_y;

reg signed [31:0] x_offset1;  //lots of "unused" bits so multiplication works
reg signed [31:0] x_offset2;

reg direction;// 0 for up, 1 for down

reg signed [19:0] stepvalue;  //the distance "x" is stepped each iteration
reg [31:0] d_stepvalue;       //the change in the total distance each
iteration

reg [4:0] i = 0;              //counts iterations

always @ (posedge clock) begin
      if (reset) begin
            state <= 0;
            done <= 0;
      end
      else begin
            case (state)
                  0: begin //idle
                        if (newData) begin
                              direction <= sine < 0;

                              //save some stuff for later
                              saved_xindex <= xindex;
                              saved_yindex <= yindex;
                              saved_xpos <= xpos;
                              saved_ypos <= ypos;

                              distanceToIntersection <= MAX_DISTANCE;

                              d_stepvalue <= (TILE_SIZE * sineInv) >> 8;
                              stepvalue <= (TILE_SIZE >> 8) * ((sineInv *
cosine) >> 8);

                              //distance to where your gaze hits the first
horizontal line
                              tempdistance <= (sine < 0) ? ((TILE_SIZE >> 8)
- ypos) * sineInv : ypos * sineInv;

                              //tempdistance is the hypotenuse of the
triangle
                              //the offsets are just tempdistance/tan(theta)
                              //we must calculate the offsets here because
Xilinx ISE chokes if you try to do everything in one step
                              x_offset1 <= ((cosine * sineInv) >> 8) * ypos;
                              x_offset2 <= ((cosine * sineInv) >> 8) *
((TILE_SIZE >> 8) - ypos);

                              if (sine == 0) begin // if the angle is 0 or
180, you'll never hit
                                    state <= 0;            // this is an
optimization since the loop will still end eventually
```

```
                                    done <= 1;              // if it never
hits
                              end
                              else begin
                                    done <= 0;
                                    state <= 1;
                                    i <= 0;
                              end
                        end
                  end
                  1: begin  //set the initial absolute_x and y


                        //x_offset must be done elsewhere for XST to work
(Xilinx bug)
                        absolute_x <= (saved_xpos << 8) +
saved_xindex*TILE_SIZE + (direction ? x_offset2 : x_offset1);
                        absolute_y <= saved_yindex*TILE_SIZE;
                        state <= 2;
                  end
                  2: begin     //this state corrects the indexes
                        saved_xindex <= absolute_x / TILE_SIZE;
                        saved_yindex <= (absolute_y / TILE_SIZE) + (direction
? 1 : -1);
                        state <= 3;
                  end
                  3: begin //this state increments to the next intersection
                        //if outside the bounds of the map, you'll never hit
a wall
                        if (saved_xindex < 21 && saved_yindex < 21
                            && absolute_x >= 0 && absolute_y >= 0
                               && absolute_x < 21*TILE_SIZE && absolute_y <
21*TILE_SIZE
                               && i < 21)  begin
                              if ( map[saved_yindex*21 + saved_xindex] == 1)
begin
                                          distanceToIntersection <=
(tempdistance > MAX_DISTANCE) ? MAX_DISTANCE : tempdistance;
                                          state <= 0;
                                          done <= 1;
                              end
                              else begin
                                    absolute_x <= absolute_x + stepvalue;
                                    absolute_y <= absolute_y +
TILE_SIZE*(direction ? 1 : -1);

                                    //check for overflow
                                    if (d_stepvalue > MAX_DISTANCE -
tempdistance) begin
                                          distanceToIntersection <=
MAX_DISTANCE;

                                          state <= 0;
                                          done <= 1;
                                    end
                                    else begin
                                          tempdistance <= tempdistance +
d_stepvalue;

                                          i <= i + 1;
                                          state <= 2;
                                    end
```

```
                              end
                        end
                        else begin //hit a wall, or can never hit a wall
                              state <= 0;
                              done <= 1;
                        end
                  end
            endcase
      end
end

endmodule

/**
      This module is a mirror of distanceToHorizontalWall but instead looks
for a vertical
      intersection.  See distanceToHorizontalWall for more detailed
documentation.  Most changes
      are sine <-> cosine, sineInv <-> cosineInv, xpos <-> ypos, and a few
minor sign changes

      Laplie Anderson
**/
module distanceToVerticalWall(clock, reset, newData,
                                                      xpos, ypos, xindex,
yindex,
                                                      sine, cosine,
                                                      sineInv, cosineInv,
                                                      map,
distanceToIntersection, done);

localparam TILE_SIZE = 64 * 256;  //the 256 here is because the first 8 bits
are below the decimal point
localparam MAX_DISTANCE = 1910 *256;  //the max distance possible on a 21x21
is slightly less than this

input clock;
input reset;
input newData;
input signed [6:0] xpos; //signed for multiplication only.  Should be 6-bit
values
input signed [6:0] ypos;
input [4:0] xindex;
input [4:0] yindex;

input signed [9:0] sine;
input signed [9:0] cosine;

// abs(1/sine) and abs(1/cosine)
// These are signed so that the multiplication works, but they should always
be positive
input signed [17:0] sineInv;
input signed [17:0] cosineInv;

input [440:0] map;

output [18:0] distanceToIntersection;
output done;
```

```
reg done = 0;
reg [1:0] state = 0;

reg [31:0] distanceToIntersection;
reg [31:0] tempdistance;

reg [4:0] saved_xindex;
reg [4:0] saved_yindex;
reg [5:0] saved_xpos;
reg [5:0] saved_ypos;


reg signed [19:0] absolute_x;  //signed to make add/subtract much easier
without checks
reg signed [19:0] absolute_y;

reg signed [31:0] y_offset1;  //lots of "unused" bits so multiplication works
reg signed [31:0] y_offset2;

reg direction;// 1 for right, 0 for left

reg signed [19:0] stepvalue;  //the distance "y" is stepped each iteration
reg [31:0] d_stepvalue;       //the change in the total distance each
iteration

reg [4:0] i = 0;              //counts iterations

wire [9:0] mapLoc;
assign mapLoc = saved_yindex*21 + saved_xindex;
assign mapHit = map[saved_yindex*21 + saved_xindex];
always @ (posedge clock) begin
      if (reset) begin
            state <= 0;
            done <= 0;
      end
      else begin
            case (state)
                  0: begin //idle
                        if (newData) begin
                              direction <= cosine > 0;

                              //save some stuff for later
                              saved_xindex <= xindex;
                              saved_yindex <= yindex;
                              saved_xpos <= xpos;
                              saved_ypos <= ypos;

                              distanceToIntersection <= MAX_DISTANCE;

                              d_stepvalue <= (TILE_SIZE * cosineInv) >> 8;
                              stepvalue <= (TILE_SIZE >> 8) * ((cosineInv *
sine) >> 8);

                              //distance to where your gaze hits the first
horizontal line
                              tempdistance <= (cosine > 0) ? ((TILE_SIZE >>
8) - xpos) * cosineInv : xpos * cosineInv;

                              //tempdistance is the hypotenuse of the
triangle
```

```
                              //we must calculate the offsets here because
Xilinx ISE chokes if you try to do everything in one step
                              y_offset1 <= ((sine * cosineInv) >> 8) * xpos;
                              y_offset2 <= ((sine * cosineInv) >> 8) *
((TILE_SIZE >> 8) - xpos);

                              if (cosine ==     0) begin // if the angle is
90 or 270, you'll never hit
                                    state <= 0;             // this is an
optimization since the loop will still end eventually
                                    done <= 1;              // if it never
hits
                              end
                              else begin
                                    done <= 0;
                                    state <= 1;
                                    i <= 0;
                              end
                        end
                  end
                  1: begin  //set the initial absolute_x and y


                        //x_offset must be done elsewhere for XST to work
(Xilinx bug)
                        absolute_y <= (saved_ypos << 8) +
saved_yindex*TILE_SIZE - (direction ? y_offset2 : y_offset1);
                        absolute_x <= saved_xindex*TILE_SIZE;
                        state <= 2;
                  end
                  2: begin     //this state corrects the indexes, the indexes
are the map positions to be checked next
                        saved_yindex <= absolute_y / TILE_SIZE;
                        saved_xindex <= (absolute_x / TILE_SIZE) + (direction
? 1 : -1);
                        state <= 3;
                  end
                  3: begin //this state increments to the next intersection
                        //if outside the bounds of the map, you'll never hit
a wall
                        if (saved_xindex < 21 && saved_yindex < 21
                               && absolute_x >= 0 && absolute_y >= 0
                               && absolute_x < 21*TILE_SIZE && absolute_y <
21*TILE_SIZE
                               && i < 21)  begin
                              if ( map[saved_yindex*21 + saved_xindex] == 1)
begin
                                          distanceToIntersection <=
(tempdistance > MAX_DISTANCE) ? MAX_DISTANCE : tempdistance;
                                          state <= 0;
                                          done <= 1;
                              end
                              else begin
                                    absolute_y <= absolute_y - stepvalue;
                                    absolute_x <= absolute_x +
TILE_SIZE*(direction ? 1 : -1);

                                    //check for overflow
                                    if (d_stepvalue > MAX_DISTANCE -
tempdistance) begin
```

```
                                                distanceToIntersection <=
MAX_DISTANCE;
                                                state <= 0;
                                                done <= 1;
                                          end
                                          else begin
                                                tempdistance <= tempdistance +
d_stepvalue;
                                                i <= i + 1;
                                                state <= 2;
                                          end
                                    end
                              end
                              else begin //hit a wall, or can never hit a wall
                                    state <= 0;
                                    done <= 1;
                              end
                        end
                  endcase
      end
end

endmodule

////////////////////////////////////////////////////////////////////////////
///
/*
  A wrapper for the IPCore pipelined divider.  The wrapper is not pipelined
but takes
  care of grabbing the result on the correct cycle.

  To start, assert newData to high.

  The result of the last division is outputed in quotient when done is high.

      IMPLEMENTATION NOTE:  The IPCore module returns q = dividend/divisor +
r where
      r is ALWAYS positive.  This means 20/3 returns 6 even though 7 is
closer to the real result

  Laplie Anderson
*/
module divider_wrapper(clock, reset, divisor, dividend, quotient, newData,
done);
      input clock, reset, newData;

      input [18:0] divisor, dividend;

      output [18:0] quotient;
      output done;

      localparam latency = 21;      //latency of IPCore divider

      reg done = 1;
      reg state = 0;

      reg [4:0] delay;
      reg [18:0] saved_divisor, saved_dividend, quotient;

      //instantiation of the pipedlined IPCore module
```

```
	wire [18:0] quot_wires, remd_wires;
	piped_divider myDivide(.clk(clock), .remd(remd_wires), .rfd(rfd_wire),
								.dividend(saved_dividend),
.divisor(saved_divisor), .quot(quot_wires) );
	always @ (posedge clock) begin
		if (reset)  begin
			done <= 0;
		end
		else begin

		case (state)
			0:begin     //we're done so we idle until new data
				done <= 0;
				if (newData) begin
					saved_divisor <= divisor;
					saved_dividend <= dividend;
					state <= 1;
					delay <= 0;
				end
			end
			1: begin //we're waiting for IPCore to return
				if (delay == latency) begin
					done <= 1;
					state <= 0;
					quotient <= quot_wires;
				end
				else delay <= delay + 1;
			end
			endcase
		end
	end


endmodule

/*
  A double buffer.

  One buffer is displayed while the other buffer is written.  The buffer read
is selected by "buffer_select", the buffer
  not selected is always the one written to when write_enable is high.

  Laplie Anderson
*/
module DoubleBuffer(clock_screen, clock, reset, read_hcount, read_vcount,
pixel, buffer_select,
					write_address, write_data, write_enable,

						ram0_clk, ram0_we_b, ram0_address,
ram0_data, ram0_cen_b,
						ram1_clk, ram1_we_b, ram1_address,
ram1_data, ram1_cen_b);

	input clock_screen, clock, reset, buffer_select, write_enable;
	input [10:0] read_hcount;
	input [9:0] read_vcount;
	input [18:0] write_address;
	input [35:0] write_data;
```

```
   output    ram0_clk, ram1_clk;    // physical line to ram clock
   output    ram0_we_b, ram1_we_b;  // physical line to ram we_b
   output [18:0]  ram0_address, ram1_address;   // physical line to ram
address
   inout  [35:0]  ram0_data, ram1_data;   // physical line to ram data
   output    ram0_cen_b, ram1_cen_b;      // physical line to ram clock
enable

      output [7:0] pixel;
      reg [7:0] pixel;

      wire [35:0] zbt0_read_data, zbt1_read_data;//, ram0_data, ram1_data,
write_data;
      wire [18:0] vram_addr;//, ram0_address, ram1_address, write_address;


      wire [7:0] vr_pixel;

      wire [35:0] vram_read_data;
      wire [18:0] zbt0_addr, zbt1_addr;

      //wire ram0_clk, ram0_we_b, ram0_cen_b, ram1_clk, ram1_we_b,
ram1_cen_b;
      //wire zbt0_we, zbt1_we;
      //display
   vram_display vd0(reset,clock_screen, read_hcount - 1, read_vcount,
vr_pixel, vram_addr, vram_read_data);


      //zbt buffers
      zbt_6111 zbt0(zbt0_clock, 1'b1, zbt0_we, zbt0_addr,
               write_data, zbt0_read_data,
               ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

   zbt_6111 zbt1(zbt1_clock, 1'b1, zbt1_we, zbt1_addr,
               write_data, zbt1_read_data,
               ram1_clk, ram1_we_b, ram1_address, ram1_data, ram1_cen_b);


      assign zbt0_we = buffer_select && write_enable;
      assign zbt1_we = !buffer_select && write_enable;

      assign zbt0_addr = buffer_select ? write_address : vram_addr;
      assign zbt1_addr = !buffer_select ? write_address : vram_addr;

      assign vram_read_data = buffer_select ? zbt1_read_data :
zbt0_read_data;

      assign zbt0_clock = buffer_select ? clock : clock_screen;
      assign zbt1_clock = buffer_select ? clock_screen : clock;

      always @ (posedge clock_screen) begin
            pixel <= vr_pixel;
      end
endmodule

/**
      Displays a small minimap of the player on the screen.
      When cheat is high, the walls in the map are also displayed
```

```
      Laplie Andersone
**/
module minimap(clock, reset,
                              player_xpos, player_ypos,
                              player_xindex, player_yindex, map,
                              hcount, vcount,
                              pixel, cheat_mode);

localparam SCREEN_WIDTH = 1024; //resolution of the screen
localparam SCREEN_HEIGHT = 768;

localparam BORDER = 3;
localparam WIDTH = 168 + 2*BORDER; //Size of the map
localparam HEIGHT = 168 + 2*BORDER;

localparam PLAYER_WIDTH = 8; //size of the player icon
localparam PLAYER_HEIGHT = 8;

localparam [10:0] X_LOCATION = SCREEN_WIDTH - WIDTH;
localparam [9:0] Y_LOCATION = 0;

parameter backgroundcolor = 8'b00000111;
parameter playercolor = 8'b11111000;
parameter wallcolor = 8'b00111000;
parameter bordercolor = 8'b11111111;

input clock, reset;
input cheat_mode;  //if cheatmode is set to 1, walls of the maze are
displayed

input [5:0] player_xpos, player_ypos; //player position in the square

input [4:0] player_xindex, player_yindex;//player position on the map

input [440:0] map; // the map

input [10:0] hcount;    //xvga signals
input [9:0]  vcount;

output [7:0] pixel; // the outputted color

wire [7:0] border_out, background_out, player_out;


//border is simply another square under the background
rectangle bordersquare(X_LOCATION, Y_LOCATION, hcount, vcount, 8'b0,
border_out);
defparam bordersquare.COLOR = bordercolor;
defparam bordersquare.WIDTH = WIDTH;
defparam bordersquare.HEIGHT = HEIGHT;

rectangle background(X_LOCATION + BORDER, Y_LOCATION + BORDER, hcount,
vcount, border_out, background_out);
defparam background.COLOR = backgroundcolor;
defparam background.WIDTH = WIDTH - (2 * BORDER);
defparam background.HEIGHT = HEIGHT - (2 * BORDER);

//the player icon
reg [10:0] player_icon_x;
reg [9:0] player_icon_y;
```

```
rectangle player(player_icon_x, player_icon_y, hcount, vcount,
background_out, player_out);
defparam player.COLOR = playercolor;
defparam player.WIDTH = PLAYER_WIDTH;
defparam player.HEIGHT = PLAYER_HEIGHT;

reg [8:0] map_grid;
reg map_grid_valid;

reg [7:0] pixel;

//the "+1"s and "-1"s  are here because the output pixel isnt updated till a
cycle later
always @ (posedge clock) begin
            player_icon_x <= X_LOCATION + BORDER + 8*player_xindex;
      player_icon_y <= Y_LOCATION + BORDER + 8*player_yindex;


            map_grid <= (hcount + 1 - X_LOCATION - BORDER) / 8 + (vcount -
Y_LOCATION - BORDER)/8 * 21;  //convert hcount/vcount to where in the map you
are in the minimap
            map_grid_valid <= hcount >= X_LOCATION + BORDER - 1 && hcount <
X_LOCATION + WIDTH - BORDER - 1
                     && vcount >= Y_LOCATION + BORDER && vcount < Y_LOCATION
+ HEIGHT - BORDER;  //determines if the conversion was valid (no overflow and
within the map area)
            pixel <= (cheat_mode && map_grid_valid && map[map_grid]) ?
wallcolor : player_out;
end
endmodule

/**
      Renders a single 3d frame and asserts Done when it is finished

      Laplie Anderson
**/
module SceneRenderer(clock, reset,
                                          player_xpos, player_ypos,
                                          player_xindex, player_yindex,
player_angle,
                                          map,
                                          ram_address, ram_data, ram_enable,
                                          start, done );
      input clock;
      input reset;

      input [5:0] player_xpos; //player position in the square
      input [5:0] player_ypos;

      input [4:0] player_xindex;//  player position on the map
      input [4:0] player_yindex;

      input [11:0] player_angle; //angle the player is looking at

      input [440:0] map; //21x21 map

      input start;

      output done;
```

```
	//column renderer signals to ram
	output ram_enable;
	output [18:0] ram_address;
	output [35:0] ram_data;

	parameter vertical_wall_color = 8'b01000100;//8'b11010101;
	parameter horizontal_wall_color = 8'b0100111;//8'b00101010;


	//  # of columns/2tan(30) ..rightshifted for fixed point math
	// rightshifted again to maximize division accuracy
	localparam distanceToProjectionPlane = 19'd524287; //<< 8 << 2;
	localparam wall_height = 64 << 8;  //height of the walls
	localparam screen_height = 768;
	localparam screen_center = screen_height/2;  //center of monitor

	reg done;
	reg [4:0] state = 0;
	reg [11:0] workingAngle, startAngle;
	reg [2:0] quadrant;
	//reg [16:0] offset;

	reg [5:0] xpos;
	reg [5:0] ypos;
	reg [4:0] xindex;
	reg [4:0] yindex;

	reg [18:0] CalcedDistance;


	//instantiate lookup table
	reg signed [9:0] CalcedSine, CalcedCosine, fishBowlCosine;
	reg signed [17:0] CalcedSineInv, CalcedCosInv;

	wire [51:0] trig_out;
	//we only use the first 10 bits cause we only keep 90 degrees in the
lookup table
	trig_lookup_table tLT(.addr(workingAngle[9:0]), .clk(clock),
.dout(trig_out));

	//instantiate distanceCalculators
	reg newDistance;
	wire [18:0] distanceHW;
	wire [18:0] distanceVW;

	distanceToHorizontalWall dHW(.clock(clock), .reset(reset),
.newData(newDistance),

.xpos({1'b0,xpos}), .ypos({1'b0,ypos}), .xindex(xindex), .yindex(yindex),

.sine(CalcedSine), .cosine(CalcedCosine),

.sineInv(CalcedSineInv), .cosineInv(CalcedCosInv),
								.map(map),
.distanceToIntersection(distanceHW), .done(doneHW));
	distanceToVerticalWall dVW(.clock(clock), .reset(reset),
.newData(newDistance),

.xpos({1'b0,xpos}), .ypos({1'b0,ypos}), .xindex(xindex), .yindex(yindex),
```

```
.sine(CalcedSine), .cosine(CalcedCosine),

.sineInv(CalcedSineInv), .cosineInv(CalcedCosInv),
                                                                .map(map),
.distanceToIntersection(distanceVW), .done(doneVW));

      //instantiate divider
      reg newDiv;
      wire [18:0] quotient;
      divider_wrapper dWrap(.clock(clock), .reset(reset),
                                                   .divisor(CalcedDistance),
.dividend(distanceToProjectionPlane), .quotient(quotient),
                                                   .newData(newDiv),
.done(doneDiv));


      //instantiate columnDrawer
      reg newCol;
      reg [7:0] column_number, wallColor0, wallColor1, wallColor2,
wallColor3;
      reg [9:0] cStartLine0, cEndLine0, cStartLine1, cEndLine1, cStartLine2,
cEndLine2, cStartLine3, cEndLine3;
      reg [1:0] cIndex;

      wire ram_enable;
      wire [18:0] ram_address;
      wire [35:0] ram_data;
      columnRenderer dC(.clock(clock), .reset(reset),
                                          .color0(wallColor0),
.color1(wallColor1),
                                          .color2(wallColor2),
.color3(wallColor3),
                                          .startLine0(cStartLine0),
.endLine0(cEndLine0),
                                          .startLine1(cStartLine1),
.endLine1(cEndLine1),
                                          .startLine2(cStartLine2),
.endLine2(cEndLine2),
                                          .startLine3(cStartLine3),
.endLine3(cEndLine3),

                                          .columnNumber(column_number),
                                          .newData(newCol), .done(doneCol),
                                          .write_address(ram_address),
.write_data(ram_data), .write_enable(ram_enable));

      always @ (posedge clock) begin
            if (reset) begin
                  done <= 0;
                  state <= 0;
                  newDistance <= 0;
                  newCol <= 0;
                  newDiv <= 0;
                  column_number <= 0;
            end
            else begin
                  case (state)
                  0: begin
                        if(start) begin
```

```
                              //save the current attribs    (assume map never
changes)
                              //start at the current angle approx. +30
degrees (to the left)
                              startAngle <= player_angle + 340;
                              column_number <= 0;
                              cIndex <= 0;
                              done <= 0;
                              xpos <= player_xpos;
                              ypos <= player_ypos;
                              xindex <= player_xindex;
                              yindex <= player_yindex;

                              cStartLine0 <= 0;
                              cEndLine0 <= 0;
                              cStartLine1 <= 0;
                              cEndLine1 <= 0;
                              cStartLine2 <= 0;
                              cEndLine2 <= 0;
                              cStartLine3 <= 0;
                              cEndLine3 <= 0;

                              wallColor0 <= 0;
                              wallColor1 <= 0;
                              wallColor2 <= 0;
                              wallColor3 <= 0;


                              state <= 1;
                        end
                  end
                  1: begin
                        //here is where i compensate for only having 90
degrees in the table
                        //and special case the some points
                        if ((startAngle - ((column_number*640 + cIndex*160)
>> 8)) == 3072) begin
                              quadrant <= 4;
                        end
                        else if ((startAngle - ((column_number*640 +
cIndex*160) >> 8)) == 2048)  begin
                              quadrant <= 5;
                        end
                        else if ((startAngle - ((column_number*640 +
cIndex*160) >> 8)) == 1024) begin
                              quadrant <= 6;
                        end
                        else if ((startAngle - ((column_number*640 +
cIndex*160) >> 8)) < 1024) begin
                              workingAngle <= startAngle -
((column_number*640 + cIndex*160) >> 8);
                              quadrant <= 0;
                        end
                        else if ((startAngle - ((column_number*640 +
cIndex*160) >> 8)) < 2048) begin
                              workingAngle <= 2048 - (startAngle -
((column_number*640 +  cIndex*160) >> 8));
                              quadrant <= 1;
                        end
```

```
                        else if ((startAngle - ((column_number*640 +
cIndex*160) >> 8)) < 3072) begin
                              workingAngle <= startAngle -
(((column_number*640 + cIndex*160) >> 8)) - 2048;
                              quadrant <= 2;
                        end
                        else begin
                              workingAngle <= 4096 - (startAngle -
((column_number*640 + cIndex*160) >> 8));
                              quadrant <= 3;
                        end
                        state <= 2;
                  end
                  2: begin
                        state <= 3; //wait a cycle for the trig table
                  end
                  3: begin
                        //get the trig values and add the correct sign
                        case (quadrant)// first quadrant
                              0: begin
                                    CalcedSine <= {1'b0, trig_out[8:0]};
                                    CalcedCosine <= {1'b0, trig_out[17:9]};
                              end
                              1:begin     //second quadrant
                                    CalcedSine <= {1'b0,trig_out[8:0]};
                                    CalcedCosine <= -{1'b0,trig_out[17:9]};
                              end
                              2:begin      //3rd quadrant
                                    CalcedSine <= -{1'b0,trig_out[8:0]};
                                    CalcedCosine <= -{1'b0,trig_out[17:9]};
                              end
                              3: begin    //4th quadrant
                                    CalcedSine <= -{1'b0,trig_out[8:0]};
                                    CalcedCosine <= {1'b0,trig_out[17:9]};
                              end
                              4: begin  //1024
                                    CalcedSine <= -256;
                                    CalcedCosine <= 0;
                                    CalcedCosInv <= 131071;
                                    CalcedSineInv <= 256;
                              end
                              5: begin     //2048
                                    CalcedSine <= 0;
                                    CalcedCosine <= -256;
                                    CalcedCosInv <= 256;
                                    CalcedSineInv <= 131071;
                              end
                              6: begin     //3072
                                    CalcedSine <= 256;
                                    CalcedCosine <= 0;
                                    CalcedCosInv <= 131071;
                                    CalcedSineInv <= 256;
                              end
                        endcase
                        if (quadrant < 4) begin
                              CalcedSineInv <= {1'b0, trig_out[34:18]};
                              CalcedCosInv <= {1'b0, trig_out[51:35]};
                        end

                        //start up the distance calculators
```

```
                        newDistance <= 1;
                        state <= 8;

                  end
                  8: begin //wait a cycle
                        newDistance <= 0;
                     state <= 4;
                  end
                  4: begin

                              //wait for the distance calculators to say that
they're finished
                              //then go to next state
                              if (doneHW & doneVW) begin

                                    //drop the lowest 4 bits to maximize
division accuracy
                                    CalcedDistance <= (((distanceHW <
distanceVW) ? distanceHW : distanceVW) >> 4);

                                    newDiv <= 1; //calculate
distanceToProjectionPlane/distanceToWall
                                    state <= 5;
                              end


                        end
                  5:begin
                        newDiv <= 0;
                        //the complexity is so only the register for the
current index is changed
                        //this could have been done with a case statement or
using an array
                        if (doneDiv) begin
                              wallColor0 <= (cIndex == 0) ? ((distanceHW <
distanceVW) ? horizontal_wall_color : vertical_wall_color) : wallColor0;
                              wallColor1 <= (cIndex == 1) ? ((distanceHW <
distanceVW) ? horizontal_wall_color : vertical_wall_color) : wallColor1;
                              wallColor2 <= (cIndex == 2) ? ((distanceHW <
distanceVW) ? horizontal_wall_color : vertical_wall_color) : wallColor2;
                              wallColor3 <= (cIndex == 3) ? ((distanceHW <
distanceVW) ? horizontal_wall_color : vertical_wall_color) : wallColor3;



                              //this logic  changes the correct startline and
endline

                              if ((quotient << 1) < screen_height) begin
                                    cStartLine0 <= (cIndex == 0) ?
(screen_center - quotient) : cStartLine0;
                                    cEndLine0 <= (cIndex == 0) ?
(screen_center + quotient) : cEndLine0;

                                    cStartLine1 <= (cIndex == 1) ?
(screen_center - quotient) : cStartLine1;
                                    cEndLine1 <= (cIndex == 1) ?
(screen_center + quotient) : cEndLine1;
```

```
                                    cStartLine2 <= (cIndex == 2) ?
(screen_center - quotient) : cStartLine2;
                                    cEndLine2 <= (cIndex == 2) ?
(screen_center + quotient) : cEndLine2;

                                    cStartLine3 <= (cIndex == 3) ?
(screen_center - quotient) : cStartLine3;
                                    cEndLine3 <= (cIndex == 3) ?
(screen_center + quotient) : cEndLine3;
                              end
                              else begin
                                    cEndLine0 <= (cIndex == 0) ?
(screen_height - 1) : cEndLine0;
                                    cEndLine1 <= (cIndex == 1) ?
(screen_height - 1) : cEndLine1;
                                    cEndLine2 <= (cIndex == 2) ?
(screen_height - 1) : cEndLine2;
                                    cEndLine3 <= (cIndex == 3) ?
(screen_height - 1) : cEndLine3;
                              end
                              if (cIndex == 3) begin
                                    newCol <= 1; // draw the column;
                                    state <= 6;
                              end
                              else state <= 7;
                        end
                  end
                  6: begin
                        newCol <= 0;
                        //wait for the column to be finished drawn
                        if (doneCol) begin
                              state <= 7;
                        end
                  end
                  7: begin
                         if (cIndex == 3) begin
                                    cStartLine0 <= 0;
                                    cEndLine0 <= 0;
                                    cStartLine1 <= 0;
                                    cEndLine1 <= 0;
                                    cStartLine2 <= 0;
                                    cEndLine2 <= 0;
                                    cStartLine3 <= 0;
                                    cEndLine3 <= 0;

                                    wallColor0 <= 0;
                                    wallColor1 <= 0;
                                    wallColor2 <= 0;
                                    wallColor3 <= 0;

                                    cIndex <= 0;

                                    if (column_number == 255) begin
//rendered all 60 degrees
                                          state <= 0;
                                          done <= 1;
                                    end else begin
                               //approx. 60 degrees is a change of 640 in
workingAngle
```

```
                                //since there are 256 columns to render, the
difference between each column is 2.5
                                //when we right shift by 8 (for fractional
decimals)

                                           column_number <= column_number + 1;
                                           state <= 1;
                                     end
                        end
                        else begin
                              state <= 1;
                              cIndex <= cIndex + 1;
                        end
                  end
                  endcase
            end
      end

endmodule

/////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.

module vram_display(reset,clk,hcount,vcount,vr_pixel,
                vram_addr,vram_read_data);

   input reset, clk;
   input [10:0] hcount;
   input [9:0]    vcount;
   output [7:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

   wire [18:0]     vram_addr = {1'b0, vcount, hcount[9:2]};

   wire [1:0]      hc4 = hcount[1:0];
   reg [7:0]       vr_pixel;
   reg [35:0]      vr_data_latched;
   reg [35:0]      last_vr_data;

   always @(posedge clk)
     last_vr_data <= (hc4==2'd3) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (hc4==2'd1) ? vram_read_data : vr_data_latched;

   always @(* )          // each 36-bit word from RAM is decoded to 4 bytes
     case (hc4)
       2'd3: vr_pixel = last_vr_data[7:0];
       2'd2: vr_pixel = last_vr_data[7+8:0+8];
       2'd1: vr_pixel = last_vr_data[7+16:0+16];
       2'd0: vr_pixel = last_vr_data[7+24:0+24];
     endcase
```

```
endmodule // vram_display


///////////////////////////////////////////////////////////////////////////
///
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
///////////////////////////////////////////////////////////////////////////
///
module xvga(vclock,hcount,vcount,hsync,vsync,blank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output   vsync;
   output   hsync;
   output   blank;

   reg        hsync,vsync,hblank,vblank,blank;
   reg [10:0]      hcount;    // pixel number on current line
   reg [9:0] vcount;     // line number

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 1023);
   assign    hsyncon = (hcount == 1047);
   assign    hsyncoff = (hcount == 1183);
   assign    hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 767);
   assign    vsyncon = hreset & (vcount == 776);
   assign    vsyncoff = hreset & (vcount == 782);
   assign    vreset = hreset & (vcount == 805);

   // sync and blanking
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule

//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
```

```
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//

/////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not
available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.

module zbt_6111(clk, cen, we, addr, write_data, read_data,
              ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

   input clk;                 // system clock
   input cen;                 // clock enable for gating ZBT cycles
   input we;                  // write enable (active HIGH)
   input [18:0] addr;         // memory address
   input [35:0] write_data;   // data to write
   output [35:0] read_data;   // data read from memory
   output    ram_clk;   // physical line to ram clock
   output    ram_we_b;  // physical line to ram we_b
   output [18:0] ram_address; // physical line to ram address
   inout [35:0]  ram_data;    // physical line to ram data
   output    ram_cen_b; // physical line to ram clock enable

   // clock enable (should be synchronous and one cycle high at a time)
   wire      ram_cen_b = ~cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;

   always @(posedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;

   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

   // wire to ZBT RAM signals

   assign      ram_we_b = ~we;
   assign      ram_clk = ~clk;     // RAM is not happy with our data hold
                                   // times if its clk edges equal FPGA's
                                   // so we clock it on the falling edges
                                   // and thus let data stabilize longer
```

```
   assign      ram_address = addr;

   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
   assign      read_data = ram_data;

endmodule // zbt_6111


/*Makes a rectangle on the screen
//draws itself if its at x, y with the provided Height and width */
module rectangle(x, y, hcount, vcount, inpixel, pixel);
   parameter WIDTH = 64;      // default width: 64 pixels
   parameter HEIGHT = 64;     // default height: 64 pixels
   parameter COLOR = 8'b11111111;  // default color: white

   input [10:0] x,hcount;
   input [9:0] y,vcount;
      input [7:0] inpixel;

   output [7:0] pixel;

   reg [7:0] pixel;
   always @ (x or y or hcount or vcount or inpixel) begin
      if ((hcount >= x && hcount < (x+WIDTH)) &&
       (vcount >= y && vcount < (y+HEIGHT)))
            pixel = COLOR;
      else pixel = inpixel;
   end
endmodule

//////////////////////////////////////////////////////////////////////////////
///
// Company:
// Engineer:
//
// Create Date:    17:11:48 11/29/06
// Design Name:
// Module Name:    game_logic
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////
///
module game_logic(clk,reset,done,up,down,left,right, map,xindex,
yindex,xpos,ypos,angle);
      input clk,reset;
      input done; // from the map generation module
      input up,down, left,right;
      input [440:0] map;

      output [4:0] xindex, yindex;  //position1         0 to 20
      output [5:0]xpos,ypos; //position2                      0 to 63
```

```
	output [11:0] angle ;	//	0 to 4095

	reg [11:0] angle;
	reg [4:0] xindex, yindex;	//position1	0 to 20
	reg [5:0]xpos,ypos; //position2			0 to 63


	always @ (posedge clk )
		begin
		if (reset)
			begin
			xindex<=0;//position1<=0;
			yindex<=0;
			xpos<=0; //xpos<=0;
			ypos<=0; //ypos<=0;

			angle<=0;
			end

		if (up)
			begin
			if (angle<=256 || angle>=4096-256) //go east
				begin
				if (xpos<=62)	xpos<=xpos+1;

				else // if (xpos==63)
					begin
					if (xindex==20) xindex<=20; // cannot move
since we are at the right edge of the map
					else if (map[21*ypos+xpos+1]) // corresponds to
map[i+1,j]
					yindex<=yindex;
	//cannot move since  there is a wall to the right
					else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
						begin
						xindex<=xindex+1;
						xpos<=0;
						end
					end
				end
			else if (angle>256 && angle<=3*256) //go northeast
				begin
				if (xpos<=62 && ypos>=1 )
					begin
					xpos<=xpos+1;
					ypos<=ypos-1;
					end
				else if (xpos==63 && ypos>=1)
					begin
					if (xindex==20) xindex<=20; // cannot move
since we are at the right edge of the map
					else if (map[21*yindex+xindex+1]) //
corresponds to map[i+1,j]
					yindex<=yindex;
	//cannot move since there is a wall to the right
					else // there is no wall to the right // if
(ypos==0 && xindex>0 && ~map[21*(j-1)+i])
						begin
```

```
                                        xindex<=xindex+1;
                                        xpos<=0;
                                        ypos<=ypos-1;
                                        end
                                  end
                            else if (xpos<=62 && ypos==0)
                                  begin
                                  if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                                  else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                        yindex<=yindex;//cannot move since above
there is a wall above
                                  else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
                                        begin
                                        yindex<=yindex-1;
                                        ypos<=63;
                                        xpos<=xpos+1;
                                        end
                                  end
                            else //if    (xpos==63 && ypos==0)
                                  begin
                                  if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                                  else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                        yindex<=yindex;//cannot move since above
there is a wall above
                                  else if (xindex==20) xindex<=20; // cannot move
since we are at the right edge of the map
                                  else if (map[21*yindex+xindex+1]) //
corresponds to map[i+1,j]
                                  yindex<=yindex;
      //cannot move since there is a wall to the right
                                  else if (map[21*(yindex-1)+xindex+1]) //
corresponds to map[i+1,j-1]
                                  yindex<=yindex;
      //cannot move since there is a wall to the right and above
                                  else        // there is no wall to the right
and above
                                        begin
                                        xindex<=xindex+1;
                                        yindex<=yindex-1;
                                        xpos<=0;
                                        ypos<=63;
                                        end
                                  end
                            end
                      else if (angle>3*256 && angle<=5*256) //go north
                            begin
                            if (ypos>=1 ) ypos<=ypos-1;
                            else // if (ypos==0)
                                  begin
                                  if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                                  else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                        yindex<=yindex;//cannot move since above
there is a wall above
```

```
                              else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
                                    begin
                                    yindex<=yindex-1;
                                    ypos<=63;
                                    end
                              end
                        end
                  else if (angle>5*256 && angle<=7*256) //go northwest
                        begin
                        if (xpos>=1 && ypos>=1 )
                              begin
                              xpos<=xpos-1;
                              ypos<=ypos-1;
                              end
                        else if (xpos==0 && ypos>=1)
                              begin
                              if (xindex==0) xindex<=0; // cannot move since
we are at the left edge of the map
                              else if (map[21*yindex+xindex-1]) //
corresponds to map[i-1,j]
                              yindex<=yindex;
      //cannot move since there is a wall to the left
                              else // there is no wall to the left
                                    begin
                                    xindex<=xindex-1;
                                    xpos<=63;
                                    ypos<=ypos-1;
                                    end
                              end
                        else if (xpos>=1 && ypos==0)
                              begin
                              if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                              else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                    yindex<=yindex;//cannot move since above
there is a wall above
                              else
                                    begin
                                    yindex<=yindex-1;
                                    ypos<=63;
                                    xpos<=xpos-1;
                                    end
                              end
                        else //if    (xpos==0 && ypos==0)
                              begin
                              if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                              else if (xindex==0) xindex<=0; // cannot move
since we are at the left edge of the map
                              else if (map[21*(yindex-1)+xindex-1]) //
corresponds to map[i-1,j-1]
                                    yindex<=yindex;//cannot move since above
there is a wall above and left

                              else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                    yindex<=yindex;//cannot move since above
there is a wall above
```

```
                        else if (map[21*yindex+xindex-1]) //
corresponds to map[i-1,j]
                              yindex<=yindex;
      //cannot move since there is a wall to the right
                        else        // there is no wall to the left and
above
                              begin
                              xindex<=xindex-1;
                              yindex<=yindex-1;
                              xpos<=63;
                              ypos<=63;
                              end
                        end
                  end
            else if (angle>7*256 && angle<=9*256) //go west
                  begin
                  if (xpos>=1 ) xpos<=xpos-1;
                  else // if (xpos==0)
                        begin
                        if (xindex==0) xindex<=0;// cannot move since
we are at the left edge of the map
                        else if (map[21*yindex+xindex-1]) //
corresponds to map[i,j-1]
                        xindex<=xindex;               //cannot move
since above there is a wall to the left
                        else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
                              begin
                              xindex<=xindex-1;
                              xpos<=63;
                              end
                        end
                  end
            else if (angle>9*256 && angle<=11*256) //go south west
                  begin
                  if (xpos>=1 && ypos<=62 )
                        begin
                        xpos<=xpos-1;
                        ypos<=ypos+1;
                        end
                  else if (xpos==0 && ypos<=62)
                        begin
                        if (xindex==0) xindex<=0; // cannot move since
we are at the left edge of the map
                        else if (map[21*yindex+xindex-1]) //
corresponds to map[i-1,j]
                        yindex<=yindex;
      //cannot move since there is a wall to the left
                        else // there is no wall to the left
                              begin
                              xindex<=xindex-1;
                              xpos<=63;
                              ypos<=ypos+1;
                              end
                        end
                  else if (xpos>=1 && ypos==63)
                        begin
                        if (yindex==63) yindex<=63; // cannot move
since we are at the bottom of the map
```

```
                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below
                              else
                                    begin
                                    yindex<=yindex+1;
                                    ypos<=0;
                                    xpos<=xpos-1;
                                    end
                              end
                        else //if    (xpos==0 && ypos==63)
                              begin
                              if (yindex==63) yindex<=63; // cannot move
since we are at the bottom of the map
                              else if (xindex==0) xindex<=0; // cannot move
since we are at the left edge of the map
                              else if (map[21*(yindex+1)+xindex-1]) //
corresponds to map[i-1,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below and left

                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below
                              else if (map[21*yindex+xindex-1]) //
corresponds to map[i-1,j]
                                    yindex<=yindex;
      //cannot move since there is a wall to the left
                              else        // there is no wall to the left and
below
                                    begin
                                    xindex<=xindex-1;
                                    yindex<=yindex+1;
                                    xpos<=63;
                                    ypos<=0;
                                    end
                              end
                        end
                  else if (angle>11*256 && angle<=13*256) //go south
                        begin
                        if (ypos<=62 ) ypos<=ypos+1;
                        else // if (ypos==63)
                              begin
                              if (xindex==20) xindex<=20;// cannot move since
we are at the bottom of the map
                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    xindex<=xindex;//cannot move since above
there is a wall below
                              else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
                                    begin
                                    xindex<=xindex+1;
                                    ypos<=0;
                                    end
                              end
                        end
```

```
                  else //if (angle>13*256 && angle<=15*256) //go to hell, no
go southeast
                        begin
                        if (xpos<=62 && ypos<=62 )
                              begin
                              xpos<=xpos+1;
                              ypos<=ypos+1;
                              end
                        else if (xpos==63 && ypos<=62)
                              begin
                              if (xindex==20) xindex<=20; // cannot move
since we are at the right edge of the map
                              else if (map[21*yindex+xindex+1]) //
corresponds to map[i+1,j]
                              yindex<=yindex;
      //cannot move since there is a wall to the right
                              else // there is no wall to the right
                                    begin
                                    xindex<=xindex+1;
                                    xpos<=0;
                                    ypos<=ypos+1;
                                    end
                              end
                        else if (xpos<=62 && ypos==63)
                              begin
                              if (yindex==63) yindex<=63; // cannot move
since we are at the bottom of the map
                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below
                              else
                                    begin
                                    yindex<=yindex+1;
                                    ypos<=0;
                                    xpos<=xpos+1;
                                    end
                              end
                        else //if    (xpos==63 && ypos==63)
                              begin
                              if (yindex==63) yindex<=63; // cannot move
since we are at the bottom of the map
                              else if (xindex==63) xindex<=63; // cannot move
since we are at the right edge of the map
                              else if (map[21*(yindex+1)+xindex+1]) //
corresponds to map[i+1,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below and left

                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below
                              else if (map[21*yindex+xindex+1]) //
corresponds to map[i-1,j]
                                    yindex<=yindex;
      //cannot move since there is a wall to the left
                              else        // there is no wall to the left and
below
                                    begin
```

```
                                    xindex<=xindex+1;
                                    yindex<=yindex+1;
                                    xpos<=0;
                                    ypos<=0;
                                    end
                              end
                        end

                  end

                  else if (down)
                  begin
                  if (angle<=256 || angle>=4096-256) //go west

                        begin
                        if (xpos>=1 ) xpos<=xpos-1;
                        else // if (xpos==0)
                              begin
                              if (xindex==0) xindex<=0;// cannot move since
we are at the left edge of the map
                              else if (map[21*yindex+xindex-1]) //
corresponds to map[i,j-1]
                              xindex<=xindex;               //cannot move
since above there is a wall to the left
                              else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
                                    begin
                                    xindex<=xindex-1;
                                    xpos<=63;
                                    end
                              end
                        end

                  else if (angle>256 && angle<=3*256) //go southwest

                        begin
                        if (xpos>=1 && ypos<=62 )
                              begin
                              xpos<=xpos-1;
                              ypos<=ypos+1;
                              end
                        else if (xpos==0 && ypos<=62)
                              begin
                              if (xindex==0) xindex<=0; // cannot move since
we are at the left edge of the map
                              else if (map[21*yindex+xindex-1]) //
corresponds to map[i-1,j]
                              yindex<=yindex;
      //cannot move since there is a wall to the left
                              else // there is no wall to the left
                                    begin
                                    xindex<=xindex-1;
                                    xpos<=63;
                                    ypos<=ypos+1;
                                    end
                              end
                        else if (xpos>=1 && ypos==63)
                              begin
                              if (yindex==63) yindex<=63; // cannot move
since we are at the bottom of the map
```

```
                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below
                              else
                                    begin
                                    yindex<=yindex+1;
                                    ypos<=0;
                                    xpos<=xpos-1;
                                    end
                              end
                        else //if    (xpos==0 && ypos==63)
                              begin
                              if (yindex==63) yindex<=63; // cannot move
since we are at the bottom of the map
                              else if (xindex==0) xindex<=0; // cannot move
since we are at the left edge of the map
                              else if (map[21*(yindex+1)+xindex-1]) //
corresponds to map[i-1,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below and left

                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below
                              else if (map[21*yindex+xindex-1]) //
corresponds to map[i-1,j]
                                    yindex<=yindex;
      //cannot move since there is a wall to the left
                              else        // there is no wall to the left and
below
                                    begin
                                    xindex<=xindex-1;
                                    yindex<=yindex+1;
                                    xpos<=63;
                                    ypos<=0;
                                    end
                              end
                        end

                  else if (angle>3*256 && angle<=5*256) //go south

                        begin
                        if (ypos>=1 ) ypos<=ypos-1;
                        else // if (ypos==0)
                              begin
                              if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                              else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                    yindex<=yindex;//cannot move since above
there is a wall above
                              else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
                                    begin
                                    yindex<=yindex-1;
                                    ypos<=63;
                                    end
                              end
```

```
                        end


                  else if (angle>5*256 && angle<=7*256) //go southeast

                        begin
                        if (xpos<=62 && ypos<=62 )
                              begin
                              xpos<=xpos+1;
                              ypos<=ypos+1;
                              end
                        else if (xpos==63 && ypos<=62)
                              begin
                              if (xindex==20) xindex<=20; // cannot move
since we are at the right edge of the map
                              else if (map[21*yindex+xindex+1]) //
corresponds to map[i+1,j]
                              yindex<=yindex;
      //cannot move since there is a wall to the right
                              else // there is no wall to the right
                                    begin
                                    xindex<=xindex+1;
                                    xpos<=0;
                                    ypos<=ypos+1;
                                    end
                              end
                        else if (xpos<=62 && ypos==63)
                              begin
                              if (yindex==63) yindex<=63; // cannot move
since we are at the bottom of the map
                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below
                              else
                                    begin
                                    yindex<=yindex+1;
                                    ypos<=0;
                                    xpos<=xpos+1;
                                    end
                              end
                        else //if    (xpos==63 && ypos==63)
                              begin
                              if (yindex==63) yindex<=63; // cannot move
since we are at the bottom of the map
                              else if (xindex==63) xindex<=63; // cannot move
since we are at the right edge of the map
                              else if (map[21*(yindex+1)+xindex+1]) //
corresponds to map[i+1,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below and left

                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    yindex<=yindex;//cannot move since above
there is a wall below
                              else if (map[21*yindex+xindex+1]) //
corresponds to map[i-1,j]
                                    yindex<=yindex;
      //cannot move since there is a wall to the left
```

```
                              else        // there is no wall to the left and
below
                                    begin
                                    xindex<=xindex+1;
                                    yindex<=yindex+1;
                                    xpos<=0;
                                    ypos<=0;
                                    end
                              end
                        end

                  else if (angle>7*256 && angle<=9*256) //go east

                        begin
                        if (xpos<=62)     xpos<=xpos+1;

                        else // if (xpos==63)
                              begin
                              if (xindex==20) xindex<=20; // cannot move
since we are at the right edge of the map
                              else if (map[21*ypos+xpos+1]) // corresponds to
map[i+1,j]
                              yindex<=yindex;
      //cannot move since  there is a wall to the right
                              else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
                                    begin
                                    xindex<=xindex+1;
                                    xpos<=0;
                                    end
                              end
                        end

                  else if (angle>9*256 && angle<=11*256) //go northeast

                        begin
                        if (xpos<=62 && ypos>=1 )
                              begin
                              xpos<=xpos+1;
                              ypos<=ypos-1;
                              end
                        else if (xpos==63 && ypos>=1)
                              begin
                              if (xindex==20) xindex<=20; // cannot move
since we are at the right edge of the map
                              else if (map[21*yindex+xindex+1]) //
corresponds to map[i+1,j]
                              yindex<=yindex;
      //cannot move since there is a wall to the right
                              else // there is no wall to the right // if
(ypos==0 && xindex>0 && ~map[21*(j-1)+i])
                                    begin
                                    xindex<=xindex+1;
                                    xpos<=0;
                                    ypos<=ypos-1;
                                    end
                              end
                        else if (xpos<=62 && ypos==0)
                              begin
```

```
                              if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                              else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                    yindex<=yindex;//cannot move since above
there is a wall above
                              else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
                                    begin
                                    yindex<=yindex-1;
                                    ypos<=63;
                                    xpos<=xpos+1;
                                    end
                              end
                        else //if    (xpos==63 && ypos==0)
                              begin
                              if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                              else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                    yindex<=yindex;//cannot move since above
there is a wall above
                              else if (xindex==20) xindex<=20; // cannot move
since we are at the right edge of the map
                              else if (map[21*yindex+xindex+1]) //
corresponds to map[i+1,j]
                              yindex<=yindex;
      //cannot move since there is a wall to the right
                              else if (map[21*(yindex-1)+xindex+1]) //
corresponds to map[i+1,j-1]
                              yindex<=yindex;
      //cannot move since there is a wall to the right and above
                              else        // there is no wall to the right
and above
                                    begin
                                    xindex<=xindex+1;
                                    yindex<=yindex-1;
                                    xpos<=0;
                                    ypos<=63;
                                    end
                              end
                        end


                  else if (angle>11*256 && angle<=13*256) //go north

                        begin
                        if (ypos<=62 ) ypos<=ypos+1;
                        else // if (ypos==63)
                              begin
                              if (xindex==20) xindex<=20;// cannot move since
we are at the bottom of the map
                              else if (map[21*(yindex+1)+xindex]) //
corresponds to map[i,j+1]
                                    xindex<=xindex;//cannot move since above
there is a wall below
                              else // if (ypos==0 && xindex>0 && ~map[21*(j-
1)+i])
                                    begin
                                    xindex<=xindex+1;
```

```
                                    ypos<=0;
                                    end
                              end
                        end

                  else //if (angle>13*256 && angle<=15*256) //go to hell, no
go northwest

                        begin
                        if (xpos>=1 && ypos>=1 )
                              begin
                              xpos<=xpos-1;
                              ypos<=ypos-1;
                              end
                        else if (xpos==0 && ypos>=1)
                              begin
                              if (xindex==0) xindex<=0; // cannot move since
we are at the left edge of the map
                              else if (map[21*yindex+xindex-1]) //
corresponds to map[i-1,j]
                              yindex<=yindex;
      //cannot move since there is a wall to the left
                              else // there is no wall to the left
                                    begin
                                    xindex<=xindex-1;
                                    xpos<=63;
                                    ypos<=ypos-1;
                                    end
                              end
                        else if (xpos>=1 && ypos==0)
                              begin
                              if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                              else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                    yindex<=yindex;//cannot move since above
there is a wall above
                              else
                                    begin
                                    yindex<=yindex-1;
                                    ypos<=63;
                                    xpos<=xpos-1;
                                    end
                              end
                        else //if    (xpos==0 && ypos==0)
                              begin
                              if (yindex==0) yindex<=0; // cannot move since
we are at the top of the map
                              else if (xindex==0) xindex<=0; // cannot move
since we are at the left edge of the map
                              else if (map[21*(yindex-1)+xindex-1]) //
corresponds to map[i-1,j-1]
                                    yindex<=yindex;//cannot move since above
there is a wall above and left

                              else if (map[21*(yindex-1)+xindex]) //
corresponds to map[i,j-1]
                                    yindex<=yindex;//cannot move since above
there is a wall above
```

```
                              else if (map[21*yindex+xindex-1]) //
corresponds to map[i-1,j]
                                    yindex<=yindex;
      //cannot move since there is a wall to the right
                              else        // there is no wall to the left and
above
                                    begin
                                    xindex<=xindex-1;
                                    yindex<=yindex-1;
                                    xpos<=63;
                                    ypos<=63;
                                    end
                              end
                        end


                  end




            else if (left)
                  begin
                  if (angle<=4095)  angle <= angle + 1;
                  else angle<=0;
                  end

            else if (right)
                  begin
                  if (angle>=1) angle <= angle - 1;
                  else angle<=4095;
                  end

            end   // ending the always




endmodule

`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////////////////
///
// Company:
// Engineer:              Mihalis Papalampros
//
// Create Date:    20:48:12 11/15/06
// Design Name:
// Module Name:    lfsr
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
```

```
////////////////////////////////////////////////////////////////////////////
///
module lfsr(clk,reset,start,min,max,random,done );           //linear
feedback shift register
      input clk,reset, start;
      input [4:0] min,max; // min and max can be only up to 20
      output [4:0] random;
      output done;


      reg a,b,c,d,e; // digits of the random number
      reg done;
      assign random={a,b,c,d,e};

      reg state = 0;
      always @ (posedge clk)
            begin

            if (reset)
                  begin
                  a<=0;
                  b<=1;
                  c<=1;
                  d<=0;
                  e<=1;
                  done<=0;
                  state <= 0;
                  end

            else
                  begin
                  case (state)
                        0:begin
                              if (start) begin
                                    state <= 1;
                                    done<=0;
                                    end
                              end
                        1: begin

                              a<=a ^d;
                              b<=a;
                              c<=b;
                              d<=c;
                              e<=d;
                              if (min<={a^d,a,b,c,d} && {a^d,a,b,c,d}<=max)
begin
                                          done<=1;
                                          state <= 0;
                              end
                              else done <= 0;
                        end


                  endcase

                  end
```

```
            end

      endmodule
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////
///
// Company:
// Engineer:             Mihalis Papalampros
//
// Create Date:    21:00:16 11/15/06
// Design Name:
// Module Name:    map_generation
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////
///


module map_generation (clk,reset,start, done,map);
      input clk,start,reset;
      output [440:0] map;
      output done;


      reg done, setup;
      reg [440:0] map;

      reg [1:0] loopstate; // ranging 0 to 3
      reg [4:0] i,j,i1,j1, i2,j2,i3,j3; // indeces ranging from 0 to 20

      reg [4:0] k; // index holding the rectangle number

      wire[4:0] random1, random3, random5, random7,random9, random11,
random13;
      // random number corresponding to the part of the rectangle we change
0 to 3

      wire[4:0] random2, random4, random6, random8,random10, random12,
random14;
      // random number corresponding to the i or j index of the rectangle we
change

      wire done1, done2, done3, done4, done5, done6,done7, done8,
done9,done10, done11, done12,done13, done14;
      wire donemod1,donemod2,donemod3,donemod4;

      reg start1, start2, start3, start4,start5, start6, start7,
start8,start9, start10, start11, start12;
      reg start13, start14,startmod1,startmod2,startmod3,startmod4;
```

```
      reg reset1, reset2,
reset3,reset4,reset5,reset6,reset7,reset8,reset9,reset10,reset11,reset12;
      reg reset13,reset14;


      reg initialization; // a flag that tells us if we are working on the
initialization
      reg checking,CW, last_chance;

      reg [1:0] direction, last_direction; //during checking we need to know
if we go up, down, right or left

      wire [13:0] rem1,rem2,rem3;

            lfsr rand1(clk,reset1,start1,0,3,random1,done1);
            lfsr rand2(clk, reset2,start2,k+1,19-k,random2,done2);
            lfsr rand3(clk,reset3,start3,0,3,random3,done3);
            lfsr rand4(clk,reset4,start4,k+1,19-k,random4,done4);

            lfsr rand6(clk,reset6,start6,k+1,19-k,random6,done6);

            lfsr rand7(clk,reset7,start7,0,3,random7,done7);
            lfsr rand8(clk,reset8,start8,k+1,19-k,random8,done8);

            lfsr rand9 (clk,reset9,start9,0,3,random9,done9);
            lfsr rand10(clk,reset10,start10,k+1,19-k,random10,done10);

            lfsr rand12(clk,reset12,start12,k+1,19-k,random12,done12);

            lfsr rand14(clk,reset14,start14,k+1,19-k,random14,done14);




      always @ (posedge clk)
            begin
            if (reset)
                  begin
                  i<=0;
                  j<=0;
                  k<=0;
                  counter<=0;
                  initialization<=1;
                  checking<=0;
                  done<=0;
                  loopstate<=2;
                  reset1<=1;
                  reset2<=1;
                  reset3<=1;
                  reset4<=1;
                  reset6<=1;
                  reset7<=1;
                  reset8<=1;
                  reset9<=1;
                  reset10<=1;
                  reset12<=1;
                  reset14<=1;
                  end
            if (start)
                  begin
```

```
            setup<=1;
            done<=0;
            end
      if (setup && ~done)
      begin

            reset1<=0;
            reset2<=0;
            reset3<=0;
            reset4<=0;
            reset6<=0;
            reset7<=0;
            reset8<=0;
            reset9<=0;
            reset10<=0;
            reset12<=0;
            reset14<=0;

      if (initialization)
      begin
            // FSM equivalent to the nested loops
            // for k=0 to 10
            //   for j=0 to 20
            //          for i=0 to 20
            //                      begin
                              //create walls or empty spaces
            //                      end
       case (loopstate)
            0: begin
                  if (k == 11) loopstate <= 3;
                  else begin
                        j <= 0;
                        loopstate <= 1;
                        k <= k+1;

                  end
            end
            1: begin
                  if (j == 21) loopstate <= 0;
                  else begin
                        i <= 0;
                        loopstate <= 2;
                        j <= j+1;
                  end
            end
            2: begin
                  if (i == 21) loopstate <= 1;
                  else
                        begin
                        if ((i==k && k<=j && j<=20-k) || (j==k && k<=i
&& i<=20-k)
                        ||(i==20-k && k<=j && j<=20-k) || (j==20-k &&
k<=i && i<=20-k))
                              begin
                              if (k%2 == 0) map[21*j+i]<=0;
                              else map[21*j+i]<=1;
                              end
                              // map[21*j+i] corresponds to map[i,j]
                              // and is initially 0, ie empty space for
the even numbered boxes
```

```
                                    // and 1, ie wall for the odd ones.

                                    i <= i + 1;

                              end
                        end

                  3: begin
                        initialization<=0;
                        k<=0;
                        start1<=1;
                        start2<=1;

                        end

            endcase


        end   //ending the initialization

        // adding/removing walls from the concentric squares
        else if (~initialization && ~checking)
            begin
            if (k==0 )//|| k==9)
                  begin
                  //create one wall in #0 rectangle
                  //erase one wall in #9 rectangle

                  if (done1) start1<=0;
                  if (done2) start2<=0;
                  if (done1 && done2)
                  begin


                  if (random1==0)               // left edge of the rectangle
                        begin
                        i<=k;
                        j<=random2;
                        map[21*random2+k]<= (k==0) ? 1:0; //corresponding to
map[i,j]

                        end
                  else if (random1==1)    // top edge
                        begin
                        j<=k;
                        i<=random2;
                        map[21*k+random2]<= (k==0) ? 1:0; //corresponding to
map[i,j]

                        end
                  else if (random1==2)     // right edge
                        begin
                        j<=random2;
                        i<=20-k;
                        map[21*random2+20-k]<= (k==0) ? 1:0; //corresponding
to map[i,j]

                        end
                  else                                        // bottom edge
                        begin
```

```
                        i<=random2;
                        j<=20-k;
                        map[21*(20-k)+random2]<= (k==0) ? 1:0;
//corresponding to map[i,j]

                        end

                  k<=k+1;
                  start1<=1;
                  start3<=1;
                  start4<=1;
                  start6<=1;

                  end
                  end
            else if (k==1 || k==2)
                  begin
                  //erase two walls in #1 rectangle
                  // create two walls in #2 rectangle


                  if (done1) start1<=0;
                  if (done3) start3<=0;
                  if (done4) start4<=0;
                  if (done6) start6<=0;

                  if (done1 && done3 && done4 && done6)
                  begin


                  if (random1==0)                 //top edge of the rectangle
                        begin
                        i<=k;
                        j<=random4;
                        map[21*random4+k]<= (k==1) ? 0:1; //corresponding to
map[i,j]

                        end
                  else if (random1==1)      // left edge
                        begin
                        j<=k;
                        i<=random4;
                        map[21*k+random4]<= (k==1) ? 0:1; //corresponding to
map[i,j]

                        end
                  else if (random1==2)      // bottom edge
                        begin
                        j<=random4;
                        i<=20-k;
                        map[21*random4+20-k]<= (k==1) ? 0:1; //corresponding
to map[i,j]

                        end
                  else                                        // right edge
                        begin
                        i<=random4;
                        j<=20-k;
                        map[21*(20-k)+random4]<= (k==1) ? 0:1;
//corresponding to map[i,j]
```

```
                        end

                  if (random3==0)                     // top edge of the
rectangle
                        begin
                        i1<=k;
                        j1<=random6;
                        map[ 21 * random6 + k]<= (k==1) ? 0:1;
//corresponding to map[i1,j1]

                        end
                  else if (random3==1)     // left edge
                        begin
                        j1<=k;
                        i1<=random6;
                        map[ 21 * k +random6]<= (k==1) ? 0:1; //corresponding
to map[i1,j1]

                        end
                  else if (random3==2)      // bottom edge
                        begin
                        j1<=random6;
                        i1<=20-k;
                        map[ 21 * random6 + 20-k]<= (k==1) ? 0:1;
//corresponding to map[i1,j1]

                        end
                  else                                            // right
edge
                        begin
                        i1<=random6;
                        j1<=20-k;
                        map[ 21 * (20-k) + random6]<= (k==1) ? 0:1;
//corresponding to map[i1,j1]

                        end

//                map[ 21*j + i ]<= (k==1) ? 0 :1; //corresponding to
map[i,j]
//                map[ 21 * j1 + i1]<= (k==1) ? 0:1; //corresponding to
map[i1,j1]
                  // erase, ie put 0, if k=1
                  // create walls, ie put 1, if k=2
                  k<=k+1;
                  start1<=1;
                  start3<=1;
                  start4<=1;
                  start6<=1;

                  start7<=1;
                  start8<=1;
                  start9<=1;
                  start10<=1;
                  start12<=1;
                  start14<=1;


                  end
                  end
```

```
            else if (k>=3 && k<=8)
                  begin
                  //erase four walls in rectangles #4, #6,#8
                  // create four walls in rectangles #3, #5, #7


               if (done1) start1<=0;
                  if (done3) start3<=0;
                  if (done7) start7<=0;
                  if (done8) start8<=0;
                  if (done9) start9<=0;
                  if (done10) start10<=0;
                  if (done12) start12<=0;
                  if (done14) start14<=0;

                  if (done1 && done3 && done7 && done8 && done9 && done10 &&
done12 && done14)
                  begin

                  if (random1==0)                 //top edge of the rectangle
                        begin
                        i<=k;
                        j<=random8;
                        map[ 21 * random8 + k]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else if (random1==1)      // left edge
                        begin
                        j<=k;
                        i<=random8;
                        map[ 21 * k + random8]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else if (random1==2)      // bottom edge
                        begin
                        j<=random8;
                        i<=20-k;
                        map[ 21 * random8 + 20-k]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else                                        // right edge
                        begin
                        i<=random8;
                        j<=20-k;
                        map[ 21 * (20-k) + random8]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end

                  if (random3==0)                     // top edge of the
rectangle
                        begin
                        i1<=k;
                        j1<=random10;
                        map[ 21 * random10 + k]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]
```

```
                        end
                  else if (random3==1)     // left edge
                        begin
                        j1<=k;
                        i1<=random10;
                        map[ 21 * k + random10]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else if (random3==2)      // bottom edge
                        begin
                        j1<=random10;
                        i1<=20-k;
                        map[ 21 * random10 + 20-k]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else                                            // right
edge
                        begin
                        i1<=random10;
                        j1<=20-k;
                        map[ 21 * (20-k)+random10]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end

                  if (random7==0)                 //top edge of the rectangle
                        begin
                        i2<=k;
                        j2<=random12;
                        map[ 21 * random12 + k]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else if (random7==1)      // left edge
                        begin
                        j2<=k;
                        i2<=random12;
                        map[ 21 * k + random12]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else if (random7==2)      // bottom edge
                        begin
                        j2<=random12;
                        i2<=20-k;
                        map[ 21 * random12 + 20-k]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else                                        // right edge
                        begin
                        i2<=random12;
                        j2<=20-k;
                        map[ 21 *(20-k)+ random12 ]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
```

```
                  if (random9==0)                     // top edge of the
rectangle
                        begin
                        i3<=k;
                        j3<=random14;
                        map[ 21 * random14 + k]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else if (random9==1)     // left edge
                        begin
                        j3<=k;
                        i3<=random14;
                        map[ 21 * k+random14 ]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else if (random9==2)      // bottom edge
                        begin
                        j3<=random14;
                        i3<=20-k;
                        map[ 21 * random14 + 20-k]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end
                  else                                            // right
edge
                        begin
                        i3<=random14;
                        j3<=20-k;
                        map[ 21 * (20-k) + random14]<=((k%2)==0 ) ? 1:0;
//corresponding to map[i3,j3]

                        end

      //          map[ 21*j + i ]<=((k%2)==0 ) ? 1: 0; //corresponding to
map[i,j]
      //          map[ 21 * j1 + i1]<=((k%2)==0 ) ? 1:0; //corresponding to
map[i1,j1]
      //          map[ 21*j2 + i2 ]<=((k%2)==0 ) ? 1:0; //corresponding to
map[i2,j2]
      //          map[ 21 * j3 + i3]<=((k%2)==0 ) ? 1:0; //corresponding to
map[i3,j3]
                  k<=k+1;
                  start1<=1;
                  start3<=1;
                  start7<=1;
                  start8<=1;
                  start9<=1;
                  start10<=1;
                  start12<=1;
                  start14<=1;

                  start2<=1;

                  end
                  end

            else if (k==9 )
                  begin
```

```
			//erase one wall in #9 rectangle

/*			lfsr rand1(clk,reset1,start1,0,3,random1,done1);*/
			if (done1) start1<=0;

			if (done1 )
			begin


			if (random1==0)			// left edge of the rectangle
				begin
				i<=k;
				j<=10; //random2 can only be 10
				map[21*10+k]<= (k==0) ? 1:0; //corresponding to
map[i,j]

				end
			else if (random1==1)	// top edge
				begin
				j<=k;
				i<=10;
				map[21*k+10]<= (k==0) ? 1:0; //corresponding to
map[i,j]

				end
			else if (random1==2)	 // right edge
				begin
				j<=10;
				i<=20-k;
				map[21*10+20-k]<= (k==0) ? 1:0; //corresponding to
map[i,j]

				end
			else						// bottom edge
				begin
				i<=10;
				j<=20-k;
				map[21*(20-k)+10]<= (k==0) ? 1:0; //corresponding to
map[i,j]

				end

			k<=k+1;
			start1<=1;
			start3<=1;
			start4<=1;
			start6<=1;

			end
			end



		else //if (k>=10)
			begin
			checking<=1;
			i<=0;
			j<=0;
			i1<=0;
```

```
                    j1<=0;
                    k<=0;
                    CW<=1;
                    direction<=0; // to the right
                    last_direction<=0;
                    last_chance<=0;

                    end
              end     //ending the else of ~initialization && ~checking

              // check if the map has a path to the center
              else if (checking)
                    begin
                    // now i,j  and i1,j1 have different functionality
                    // i,j is the current place
                    // i1,j1 is the last place we started

                    if (CW)
                          begin
                          if (direction==0)  // go right
                                begin
                                if ((i==k && j==k) || (map[21*(j+1)+i]&& i<=19-
k && ~map[21*j+i+1])) i<=i+1;
                                // if we are at the beginning of the box or
                                // if there is a wall down and we have not
reached the right edge
                                // and we can move to the right

                                else if ((map[21*(j+1)+i]&& i==20-k) ||
(map[21*(j+1)+i] && i<=19-k && map[21*j+i+1] ))

                                // if there is a wall down and we are at the
edge of the k box
                                // or if there is a wall down and to the right
                                // both cases are DEAD ENDS
                                      begin
                                      if (last_chance) //do procedure again
                                            begin
                                            initialization<=1;
                                            checking<=0;
                                            k<=0;
                                            i<=0;
                                            j<=0;
                                            loopstate<=2;
                                            end
                                      else
                                            begin
                                            i<=i1;
                                            j<=j1;
                                            CW<=0;
                                            if (last_direction==0)
direction<=2;
                                            else if (last_direction==1)
direction<=3;
                                            else if (last_direction==2)
direction<=0;
                                            else /*if (last_direction==3)*/
direction<=1;

                                            last_chance<=1;
```

```
                                        // we start again from the point we
entered the k box
                                        // but now having CCW direction and
also taking the opposite
                                        //direction of that of CW
                                        end
                                  end
                            else if (~map[21*(j+1)+i]&& i<=19-k &&
~map[21*(j+2)+i])
                            // if there is no wall down and we are not at
the edge
                            // and if there is no wall two boxes down
                                  begin
                                  j<=j+2;
                                  if (k==8)  done<=1;             // we
reached the center
                                  else k<=k+2; // we moved two boxes closer
to the center
                                  // we skip the box consisting of all
walls except for the opening

                                  // we update the information about the
point we entered the box
                                  i1<=i;
                                  j1<=j+2;
                                  last_direction<=0;
                                  last_chance<=0;


                                  end

                            else if (~map[21*(j+1)+i]&& i<=19-k &&
map[21*(j+2)+i] && map[21*j+i+1])
                            // if there is no wall down and we are not at
the edge
                            // and if there is a wall two boxes down, and a
wall to the right
                            // all these make it a DEAD END!!!
                                  begin
                                  if (last_chance) //do procedure again
                                        begin
                                        initialization<=1;
                                        checking<=0;
                                        k<=0;
                                        i<=0;
                                        j<=0;
                                        loopstate<=2;

                                        end
                                  else
                                        begin
                                        i<=i1;
                                        j<=j1;
                                        CW<=0;
                                        if (last_direction==0)
direction<=2;
                                        else if (last_direction==1)
direction<=3;
                                        else if (last_direction==2)
direction<=0;
```

```
                                                else /*if (last_direction==3)*/
direction<=1;
                                                last_chance<=1;
                                                end
                                          end

                                    else if (~map[21*(j+1)+i]&& i<=19-k &&
map[21*(j+2)+i] && ~map[21*j+i+1])
                                    // if there is no wall down and we are not at
the edge
                                    // and if there is a wall two boxes down, and
no wall to the right

                                          i<=i+1; //we just move to the right!

                                    else //if (~map[21*(j+1)+i]&& i==20-k)// if
(~map[i+21] && i%21==20)
                                          begin
                                          direction<=1; // go down
                                          end
                                    end

                              if (direction==1) // down

                                    begin
                                    if ((i==20-k && j==k) ||(map[21*j+i-1]&& j<=19-
k && ~map[21*(j+1)+i])) j<=j+1;
                                    //if we are at the beginning of the box
                                    // if there is a wall on the left and we have
not reached the bottom edge

                                    else if ((map[21*j+i-1]&& j==20-k) ||
(map[21*j+i-1]&& j<=19-k && map[21*(j+1)+i]))
                                    //(map[i+21] && i%21==20)
                                    // if there is a wall on the left and we are at
the edge of the k box
                                          begin
                                          if (last_chance) //do procedure again
                                                begin
                                                initialization<=1;
                                                checking<=0;
                                                k<=0;
                                                i<=0;
                                                j<=0;
                                                loopstate<=2;
                                                end
                                          else
                                                begin
                                                i<=i1;
                                                j<=j1;
                                                CW<=0;
                                                if (last_direction==0)
direction<=2;
                                                else if (last_direction==1)
direction<=3;
                                                else if (last_direction==2)
direction<=0;
                                                else /*if (last_direction==3)*/
direction<=1;
                                                last_chance<=1;
```

```
                                    end
                              end


                        else if (~map[21*j+i-1]&& j<=19-k &&
~map[21*j+i-2])
                        // if there is no wall on the left and we are
not at the edge
                        // and there is no wall two boxes left
                              begin
                              i<=i-2;
                              if (k==8)  done<=1;             // we
reached the center
                              else k<=k+2; // we moved two boxes closer
to the center
                              // we skip the box consisting of all
walls except for the opening

                              i1<=i-2;
                              j1<=j;
                              last_chance<=0;
                              last_direction<=1;

                              end


                        else if (~map[21*j+i-1] && j<=19-k &&
map[21*j+i-2] && map[21*(j+1)+i])
                        // if there is no wall on the left and we are
not at the edge
                        // and if there is a wall two boxes left, and a
wall down
                        // all these make it a DEAD END!!!
                              begin
                              if (last_chance) //do procedure again
                                    begin
                                    initialization<=1;
                                    checking<=0;
                                    k<=0;
                                    i<=0;
                                    j<=0;
                                    loopstate<=2;
                                    end
                              else
                                    begin
                                    i<=i1;
                                    j<=j1;
                                    CW<=0;
                                    if (last_direction==0)
direction<=2;
                                    else if (last_direction==1)
direction<=3;
                                    else if (last_direction==2)
direction<=0;
                                    else /*if (last_direction==3)*/
direction<=1;
                                    last_chance<=1;
                                    end
                              end
```

```
					else if (~map[21*j+i-1] && j<=19-k &&
map[21*j+i-2] && ~map[21*(j+1)+i])
					// if there is no wall on the left and we are
not at the edge
					// and if there is a wall two boxes left, and
no wall down

						j<=j+1; //we just move down!

					else
						begin
						direction<=2; // go left
						end
					end

				if (direction==2) // left

					begin
					if ((i==20-k && j==20-k) || (map[21*(j-1)+i]&&
i>=k+1 && map[21*j+i-1])) i<=i-1;
					//if we are at the beginning of the box or

					// if there is a wall above and we have not
reached the left edge

					else if ((map[21*(j-1)+i]&& i==k) ||
(map[21*(j-1)+i]&& i>=k+1 && map[21*j+i-1]))
					// if there is a wall above and we are at the
edge of the k box
						begin
						if (last_chance) //do procedure again
							begin
							initialization<=1;
							checking<=0;
							k<=0;
							i<=0;
							j<=0;
							loopstate<=2;
							end
						else
							begin
							i<=i1;
							j<=j1;
							CW<=0;
							if (last_direction==0)
direction<=2;
							else if (last_direction==1)
direction<=3;
							else if (last_direction==2)
direction<=0;
							else /*if (last_direction==3)*/
direction<=1;
							last_chance<=1;
							end
						end
					else if (~map[21*(j-1)+i]&& i>=k+1 &&
~map[21*(j-2)+i])
					// if there is no wall above and we are not at
the edge
					// and there is no wall two boxes above
```

```
                                        begin
                                        j<=j-2;
                                        if (k==8)  done<=1;             // we
reached the center
                                        else k<=k+2; // we moved two boxes closer
to the center
                                        // we skip the box consisting of all
walls except for the opening

                                        i1<=i;
                                        j1<=j-2;
                                        last_direction<=2;
                                        last_chance<=0;

                                        end


                              else if (~map[21*(j-1)+i]&& i>=k+1 &&
map[21*(j-2)+i] && map[21*j+i-1])
                              // if there is no wall above and we are not at
the edge
                              // and if there is a wall two boxes above, and
a wall to the left
                              // all these make it a DEAD END!!!
                                        begin
                                        if (last_chance) //do procedure again
                                              begin
                                              initialization<=1;
                                              checking<=0;
                                              k<=0;
                                              i<=0;
                                              j<=0;
                                              loopstate<=2;
                                              end
                                        else
                                              begin
                                              i<=i1;
                                              j<=j1;
                                              CW<=0;
                                              if (last_direction==0)
direction<=2;
                                              else if (last_direction==1)
direction<=3;
                                              else if (last_direction==2)
direction<=0;
                                              else /*if (last_direction==3)*/
direction<=1;
                                              last_chance<=1;
                                              end
                                        end

                              else if (~map[21*(j-1)+i]&& i>=k+1 &&
map[21*(j-2)+i] && ~map[21*j+i-1])
                              // if there is no wall above and we are not at
the edge
                              // and if there is a wall two boxes above, and
no wall to the left

                                        i<=i-1; //we just move to the left!
```

```
                              else //if (~map[21*j+i-1]&& j==20-k)
                                    begin
                                    direction<=3; // go up
                                    end
                              end

                        if (direction==3) // up

                              begin
                              if ((i==k && j==20-k) || (map[21*j+i+1]&&
j>=k+1 && ~map[21*(j-1)+i])) j<=j-1;
                              //if we are at the beginning of the box or

                              // if there is a wall on the right and we have
not reached the top edge

                              else if ((map[21*j+i+1]&& j==k) ||
(map[21*j+i+1]&& j>=k+1 && map[21*(j-1)+i]))

                              // if there is a wall on the right and we are
at the edge of the k box
                                    begin
                                    if (last_chance) // do procedure again
                                          begin
                                          initialization<=1;
                                          checking<=0;
                                          k<=0;
                                          i<=0;
                                          j<=0;
                                          loopstate<=2;
                                          end
                                    else
                                          begin
                                          i<=i1;
                                          j<=j1;
                                          CW<=0;
                                          if (last_direction==0)
direction<=2;
                                          else if (last_direction==1)
direction<=3;
                                          else if (last_direction==2)
direction<=0;
                                          else /*if (last_direction==3)*/
direction<=1;
                                          last_chance<=1;
                                          end
                                    end
                              else if (~map[21*j+i+1]&& j>=k+1 &&
~map[21*j+i+2])
                              // if there is no wall to the right and we are
not at the edge
                              /// and there is no wall two boxes to the right
                                    begin
                                    i<=i+2;
                                    if (k==8)  done<=1;             // we
reached the center
                                    else k<=k+2; // we moved two boxes closer
to the center
                                    // we skip the box consisting of all
walls except for the opening
```

```
                                  i1<=i+2;
                                      j1<=j;
                                      last_direction<=3;
                                      last_chance<=0;

                                      end

                                else if (~map[21*j+i+1]&& j>=k+1 &&
~map[21*j+i+2] && map[21*(j-1)+i])

                                // if there is no wall to the right and we are
not at the edge
                                // and if there is a wall two boxes to the
right, and a wall above
                                // all these make it a DEAD END!!!
                                      begin
                                      if (last_chance) //do procedure again
                                            begin
                                            initialization<=1;
                                            checking<=0;
                                            k<=0;
                                            i<=0;
                                            j<=0;
                                            loopstate<=2;
                                            end
                                      else
                                            begin
                                            i<=i1;
                                            j<=j1;
                                            CW<=0;
                                            if (last_direction==0)
direction<=2;
                                            else if (last_direction==1)
direction<=3;
                                            else if (last_direction==2)
direction<=0;
                                            else /*if (last_direction==3)*/
direction<=1;
                                            last_chance<=1;
                                            end
                                      end

                                else if (~map[21*j+i+1]&& j>=k+1 &&
~map[21*j+i+2] && ~map[21*(j-1)+i])
                                // if there is no wall to the right and we are
not at the edge
                                // and if there is a wall two boxes to the
right, and no wall above

                                      j<=j-1; //we just move above!


                                else //if (~map[21*j+i+1]&& j==k)
                                      begin
                                      direction<=0; // go right
                                      end
                                end
                          end   //ending the if CW
```

```
                    else  // Counterclockwise direction

                          begin
                          if (direction==0)  // go right
                                begin
                                if ((i==k && j==20-k) ||(map[21*(j-1)+i]&&
i<=19-k && ~map[21*j+i+1])) i<=i+1;
                                //if we are at the beginning of the box or

                                // if there is a wall above and we have not
reached the right edge

                                else if ((map[21*(j-1)+i]&& i==20-k)
||(map[21*(j-1)+i]&& i<=19-k && map[21*j+i+1]))
                                // if there is a wall above and we are at the
edge of the k box
                                      begin
                                      if (last_chance) //do procedure again
                                            begin
                                            initialization<=1;
                                            checking<=0;
                                            k<=0;
                                            i<=0;
                                            j<=0;
                                            loopstate<=2;
                                            end
                                      else
                                            begin
                                            i<=i1;
                                            j<=j1;
                                            CW<=1;
                                            if (last_direction==0)
direction<=2;
                                            else if (last_direction==1)
direction<=3;
                                            else if (last_direction==2)
direction<=0;
                                            else /*if (last_direction==3)*/
direction<=1;
                                            last_chance<=1;
                                            end
                                      end
                                else if (~map[21*(j-1)+i] && i<=19-k &&
~map[21*(j-2)+i])
                                // if there is no wall above and we are not at
the edge
                                // and there is no wall two boxes above
                                      begin
                                      j<=j-2;
                                      if (k==8)  done<=1;            // we
reached the center
                                      else  k<=k+2; // we moved two boxes
closer to the center
                                      // we skip the box consisting of all
walls except for the opening

                                      i1<=i;
```

```
                                        j1<=j-2;
                                        last_direction<=0;

                                        end

                                else if (~map[21*(j-1)+i]&& i<=19-k &&
map[21*(j-2)+i] && map[21*j+i+1])

                                // if there is no wall above and we are not at
the edge
                                // and if there is a wall two boxes above, and
a wall to the right
                                // all these make it a DEAD END!!!
                                        begin
                                        if (last_chance) //do procedure again
                                                begin
                                                initialization<=1;
                                                checking<=0;
                                                k<=0;
                                                i<=0;
                                                j<=0;
                                                loopstate<=2;
                                                end
                                        else
                                                begin
                                                i<=i1;
                                                j<=j1;
                                                CW<=1;
                                                if (last_direction==0)
direction<=2;
                                                else if (last_direction==1)
direction<=3;
                                                else if (last_direction==2)
direction<=0;
                                                else /*if (last_direction==3)*/
direction<=1;
                                                last_chance<=1;
                                                end
                                        end

                                else if (~map[21*(j-1)+i]&& i<=19-k &&
map[21*(j-2)+i] && ~map[21*j+i+1])
                                // if there is no wall above and we are not at
the edge
                                // and if there is a wall two boxes above, and
no wall to the right

                                        i<=i+1; //we just move right!



                                else //if (~map[21*(j-1)+i]&& i==20-k)
                                        begin
                                        direction<=3; // go up
                                        end
                                end

                        if (direction==1) // down

                                begin
```

```
                                if ((i==k && j==k) || (map[21*j+i+1]&& j<=19-k
&& ~map[21*(j+1)+i])) j<=j+1;
                                //if we are at the beginning of the box or

                                // if there is a wall on the right and we have
not reached the bottom edge

                                else if ((map[21*j+i+1]&& j==20-k)
||(map[21*j+i+1]&& j<19-k && map[21*(j+1)+i]))
                                // if there is a wall on the right and we are
at the edge of the k box
                                      begin
                                      if (last_chance) //
      no path to the center do procedure again
                                            begin
                                            initialization<=1;
                                            checking<=0;
                                            k<=0;
                                            i<=0;
                                            j<=0;
                                            loopstate<=2;
                                            end
                                      else
                                            begin
                                            i<=i1;
                                            j<=j1;
                                            CW<=1;
                                            if (last_direction==0)
direction<=2;
                                            else if (last_direction==1)
direction<=3;
                                            else if (last_direction==2)
direction<=0;
                                            else /*if (last_direction==3)*/
direction<=1;
                                            last_chance<=1;
                                            end
                                      end
                                else if (~map[21*j+i+1]&& j<=19-k &&
~map[21*j+i+2])
                                // if there is no wall on the right and we are
not at the edge
                                // and there is no wall two boxes to the right
                                      begin
                                      i<=i+2;
                                      if (k==8 ) done<=1;
                                      else  k<=k+2; // we moved two boxes
closer to the center
                                      // we skip the box consisting of all
walls except for the opening

                                      i1<=i+2;
                                      j1<=j;
                                      last_direction<=1;

                                      end

                                else if (~map[21*j+i+1]&& j<=19-k &&
map[21*j+i+2] && map[21*(j+1)+i])
```

```
                              // if there is no wall to the right and we are
not at the edge
                              // and if there is a wall two boxes to the
right, and a wall down
                              // all these make it a DEAD END!!!
                                    begin
                                    if (last_chance) //do procedure again
                                          begin
                                          initialization<=1;
                                          checking<=0;
                                          k<=0;
                                          i<=0;
                                          j<=0;
                                          loopstate<=2;
                                          end
                                    else
                                          begin
                                          i<=i1;
                                          j<=j1;
                                          CW<=1;
                                          if (last_direction==0)
direction<=2;
                                          else if (last_direction==1)
direction<=3;
                                          else if (last_direction==2)
direction<=0;
                                          else /*if (last_direction==3)*/
direction<=1;
                                          last_chance<=1;
                                          end
                                    end

                              else if (~map[21*j+i+1]&& j<=19-k &&
map[21*j+i+2] && ~map[21*(j+1)+i])
                              // if there is no wall to the right and we are
not at the edge
                              // and if there is a wall two boxes to the
right, and no wall down

                                    j<=j+1; //we just move down!

                              else //if (~map[21*j+i+1]&& j==20-k)
                                    begin
                                    direction<=0; // go right
                                    end
                              end

                        if (direction==2) // left

                              begin
                              if ((i==20-k && j==k) || (map[21*(j+1)+i]&&
i>=k+1 && ~map[21*j+i-1])) i<=i-1;
                              //if we are at the beginning of the box or

                              // if there is a wall below and we have not
reached the left edge

                              else if ((map[21*(j+1)+i]&& i==k) ||
(map[21*(j+1)+i]&& i>=k+1 && map[21*j+i-1]))
                               //(map[i+21] && i%21==20)
```

```
                              // if there is a wall below and we are at the
edge of the k box
                                    begin
                                    if (last_chance)              //
            no path to the center do procedure again
                                          begin
                                          initialization<=1;
                                          checking<=0;
                                          k<=0;
                                          i<=0;
                                          j<=0;
                                          loopstate<=2;
                                          end
                                    else
                                          begin
                                          i<=i1;
                                          j<=j1;
                                          CW<=1;
                                          if (last_direction==0)
direction<=2;
                                          else if (last_direction==1)
direction<=3;
                                          else if (last_direction==2)
direction<=0;
                                          else /*if (last_direction==3)*/
direction<=1;
                                          last_chance<=1;
                                          end
                                    end
                              else if (~map[21*(j+1)+i]&& i>=k+1 &&
~map[21*(j+2)+i])
                              // if there is no wall down and we are not at
the edge
                              // and there is no wall two boxes down
                                    begin
                                    j<=j+2;
                                    if (k==8)  done<=1;           // we
reached the center
                                    else k<=k+2; // we moved two boxes closer
to the center
                                    // we skip the box consisting of all
walls except for the opening

                                    i1<=i;
                                    j1<=j+2;
                                    last_direction<=2;

                                    end

                              else if (~map[21*(j+1)+i]&& i>=k+1 &&
map[21*(j+2)+i] && map[21*(j+1)+i])

                              // if there is no wall below and we are not at
the edge
                              // and if there is a wall two boxes below, and
a wall to the left
                              // all these make it a DEAD END!!!
                                    begin
                                    if (last_chance) //do procedure again
                                          begin
```

```
                                                initialization<=1;
                                                checking<=0;
                                                k<=0;
                                                i<=0;
                                                j<=0;
                                                loopstate<=2;
                                                end
                                          else
                                                begin
                                                i<=i1;
                                                j<=j1;
                                                CW<=1;
                                                if (last_direction==0)
direction<=2;
                                                else if (last_direction==1)
direction<=3;
                                                else if (last_direction==2)
direction<=0;
                                                else /*if (last_direction==3)*/
direction<=1;
                                                last_chance<=1;
                                                end
                                          end

                                    else if (~map[21*(j+1)+i]&& i>=k+1 &&
map[21*(j+2)+i] && ~map[21*(j+1)+i])
                                    // if there is no wall below and we are not at
the edge
                                    // and if there is a wall two boxes below, and
no wall to the left

                                          i<=i-1; //we just move left!




                                    else //if (~map[21*j+i-1]&& j==20-k)
                                          begin
                                          direction<=1; // go down
                                          end
                                    end

                              if (direction==3) // up

                                    begin
                                    if ((i==20-k && j==20-k) || (map[21*j+i-1]&&
j>=k+1 && ~map[21*(j-1)+i])) j<=j-1;
                                    //if we are at the beginning of the box or

                                    // if there is a wall on the left and we have
not reached the top edge

                                    else if ((map[21*j+i-1]&& j==k) || (map[21*j+i-
1]&& j>=k+1 && map[21*(j-1)+i]))
                                     //(map[i+21] && i%21==20)
                                    // if there is a wall on the left and we are at
the edge of the k box
                                          begin
                                          if (last_chance) //do procedure again
                                                begin
```

```
                                        initialization<=1;
                                        checking<=0;
                                        k<=0;
                                        i<=0;
                                        j<=0;
                                        loopstate<=2;
                                        end
                                  else
                                        begin
                                        i<=i1;
                                        j<=j1;
                                        CW<=1;
                                        if (last_direction==0)
direction<=2;
                                        else if (last_direction==1)
direction<=3;
                                        else if (last_direction==2)
direction<=0;
                                        else /*if (last_direction==3)*/
direction<=1;
                                        last_chance<=1;
                                        end
                                  end
                            else if (~map[21*j+i-1]&& j>=k+1 &&
~map[21*j+i-2])
                            // if there is no wall to the left and we are
not at the edge
                            // and there is no wall two boxes to the left
                                  begin
                                  i<=i-2;
                                  if (k==8)  done<=1;             // we
reached the center
                                  else k<=k+2; // we moved two boxes closer
to the center
                                  // we skip the box consisting of all
walls except for the opening

                                  i1<=i-2;
                                  j1<=j;
                                  last_direction<=3;

                                  end

                            else if (~map[21*j+i-1]&& j>=k+1 &&
~map[21*j+i-2]&& map[21*(j-1)+i])

                            // if there is no wall to the left and we are
not at the edge
                            // and if there is a wall two boxes to the
left, and a wall above
                            // all these make it a DEAD END!!!
                                  begin
                                  if (last_chance) //do procedure again
                                        begin
                                        initialization<=1;
                                        checking<=0;
                                        k<=0;
                                        i<=0;
                                        j<=0;
                                        loopstate<=2;
```

```
                                                     end
                                               else
                                                     begin
                                                     i<=i1;
                                                     j<=j1;
                                                     CW<=1;
                                                     if (last_direction==0)
direction<=2;
                                                     else if (last_direction==1)
direction<=3;
                                                     else if (last_direction==2)
direction<=0;
                                                     else /*if (last_direction==3)*/
direction<=1;
                                                     last_chance<=1;
                                                     end
                                               end

                                         else if (~map[21*j+i-1]&& j>=k+1 &&
~map[21*j+i-2]&& ~map[21*(j-1)+i])
                                         // if there is no wall below and we are not at
the edge
                                         // and if there is a wall two boxes below, and
no wall above

                                               j<=j-1; //we just move up!


                                         else //if (~map[21*j+i+1]&& j==k)
                                               begin
                                               direction<=2; // go left
                                               end
                                         end

                                   end   // ending else (CCW direction)

                             end //ending checking



  end //(ending the if start)

  end // ending the always

  endmodule
```