Gim's Labyrinth

6.111 Fall 2019
Final Project Report

Gian Delfin, Vivian Huang, Luis Terrones-Verastegui

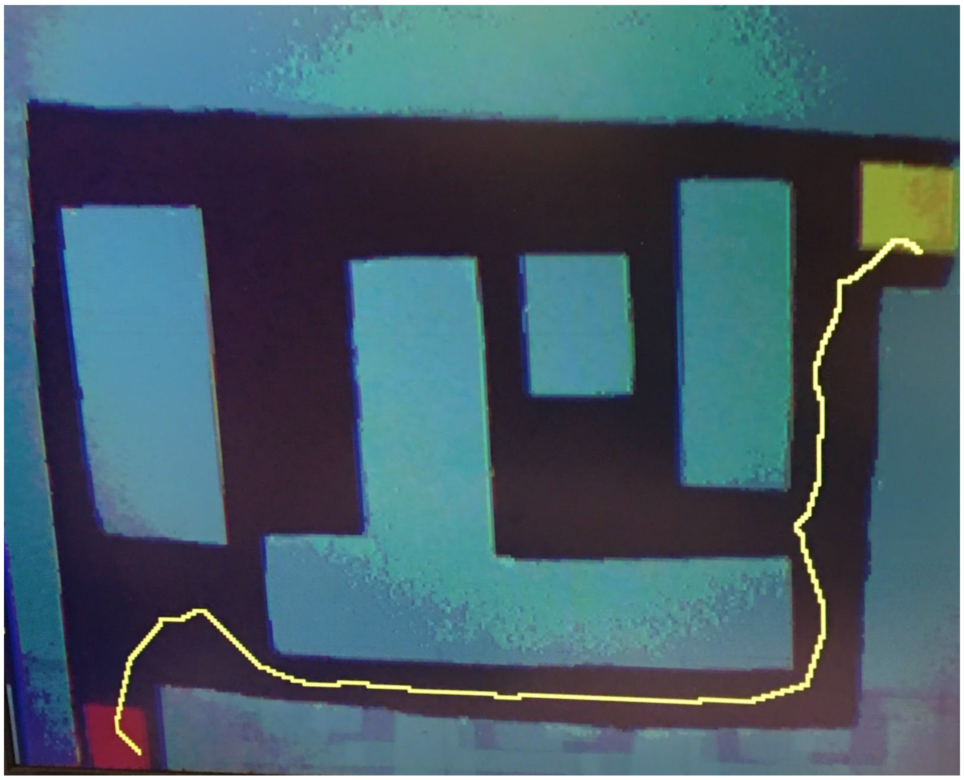


Table of Contents

I. Introduction

The problem of maze routing involves finding a shortest-path connection between a starting point and an endpoint on a grid structure of interconnected paths. Our team is intrigued by this topic because it carries many practical applications such as VLSI wire routing, intelligent traffic control and robot path planning. With mobile robots designed to carry out rescues, for example, their planned paths should be collision-free as well as the shortest solutions. However, the traditional method for path planning is based on trial and error—that is, the robot has no prior knowledge of the maze and tries each available path in the maze until it reaches the destination point. Such a method is time-consuming (especially for complex mazes), does not always give the shortest path, and could trap the robot in an infinite loop.

Hence, our approach to this topic is *Gim's Labyrinth*, a maze-solving system that quickly displays the shortest path through a physical, user-defined maze. The user can define a maze by laying out paper "walls" in front of the camera setup. From there, we capture an image of the maze and process it using the Nexys4 in order to determine the layout of the maze walls and paths. Next, we use the Nexys4 to run the processed image through a maze-solving algorithm in order to generate and display a shortest-path solution to the maze. We perform this entire process as quickly as possible using the Nexys 4 DDR FPGA board so that real-time maze-solving occurs as the user changes the layout of the maze.

II. Goals

<u>Baseline:</u>

- Send an image from the camera to the FPGA and display it on the monitor
- Implement an image processing pipeline to convert the RGB image to a binary array representing the maze
- Determine the correct path through the maze using a maze-solving algorithm

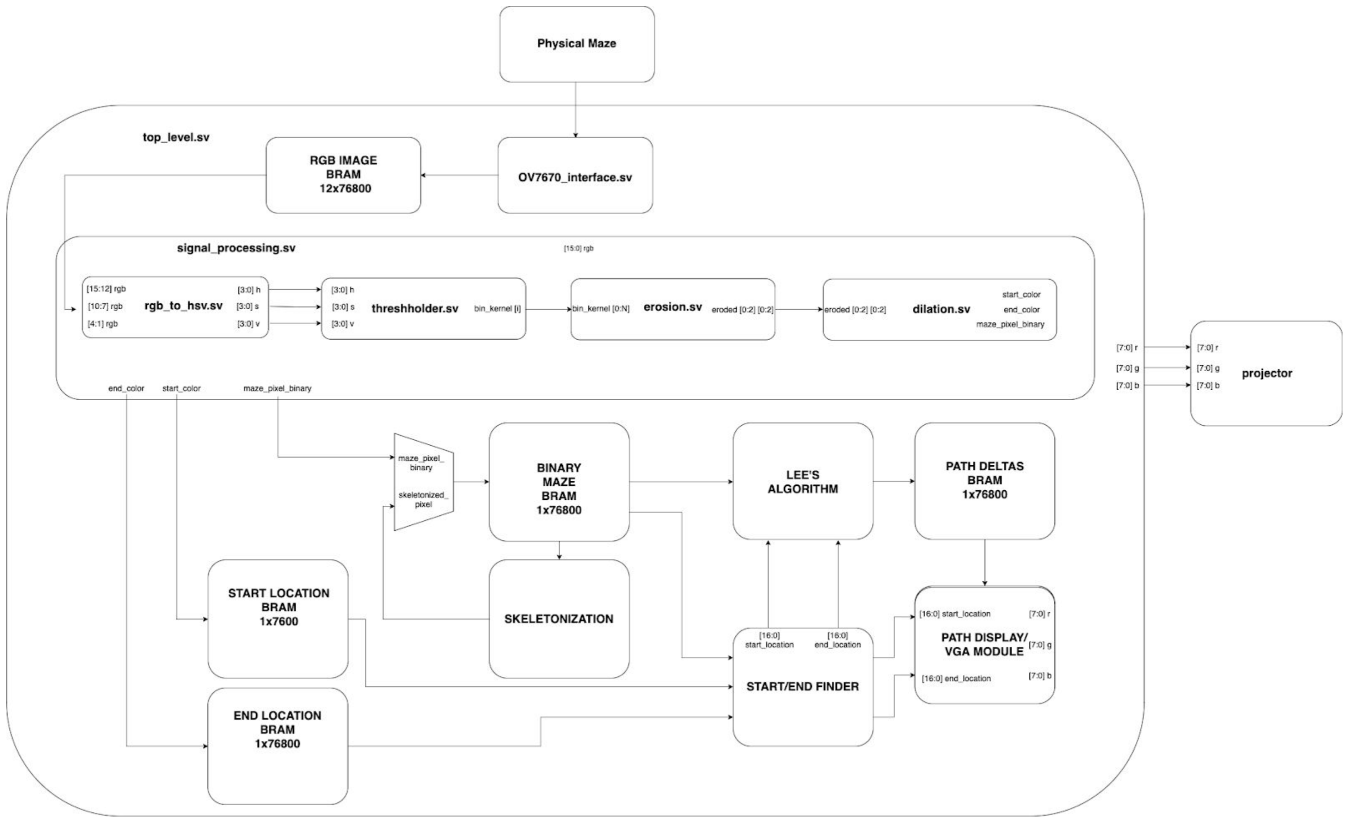
<u>Expected:</u>

- Display the path determined by the maze-solving algorithm on the maze

<u>Stretch:</u>

- Perform real-time maze-solving; the displayed shorted path changes as the user changes the physical walls of the maze

III. System Block Diagram

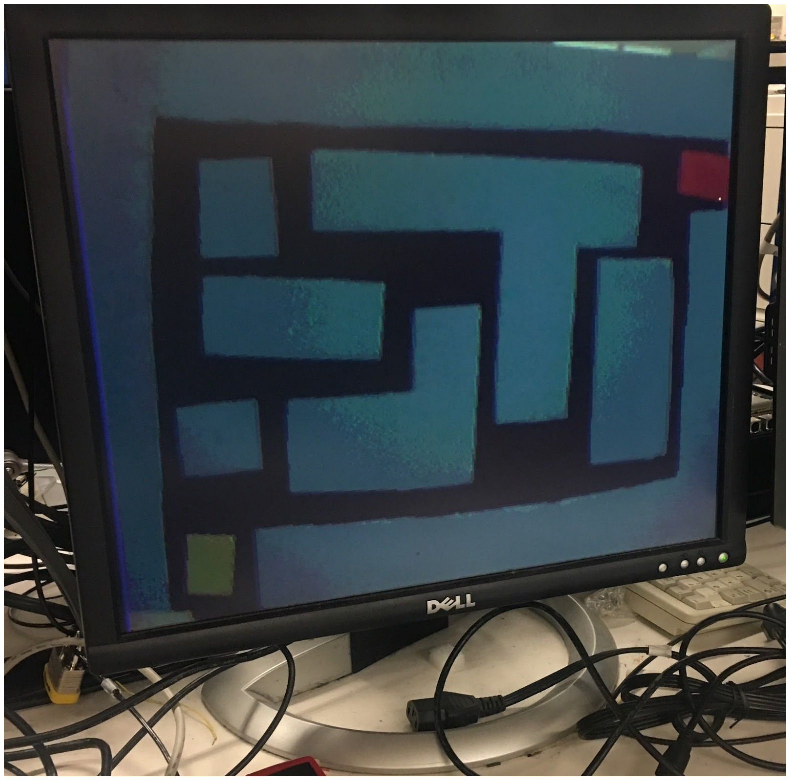


(*landscape view on last page of appendix)

IV. Subsystems

1. Camera Input (Vivian)

Module(s): top_level.sv



Displaying a 320x240 image of the maze on the monitor

The maze is constructed out of foam board, while the system configuration for taking images of this maze involves the OV7670 camera sensor and the ESP8266 microcontroller. The camera has 640x480 pixel resolution, but taking BRAM constraints into consideration, we kept it set at 320x240 pixels in QVGA mode which was specified by the microcontroller.

In top_level.sv, frame_done_out represents the camera's VSYNC signal, such that when it signals true, the camera signal starts to send out pixels starting with the left upper corner of the frame. The entire frame is then stored in a frame buffer, and we use that frame buffer to render the video on a display.

We approached implementing the camera input process in four steps. First, we tested the camera setup (using Joe Steinmeyer's camera starter code), to see if we could capture frames continuously in XVGA mode (1024x768), shown in the top left corner of the computer monitor. Second, we adapted the code for VGA mode (640x480) and used 2X magnification so that we could ultimately display the maze full-screen on the computer with the solved paths overlaid on top. Third, we wrote test code to freeze a snapshot of the maze on the monitor (instead of showing frames continuously) whenever we toggled sw[15], for the purpose of passing the snapshot into the image processing pipeline. Fourth, we integrated the camera code into the state machine in top_level.sv.

Luckily, we did not encounter major challenges while implementing this step of the pipeline, although it required a sizable chunk of the BRAM. The RGB image was stored with four bits each for red, green, and blue, consuming a total of 12x320x240 = 12x76800 bits.

2. RGB → HSV Conversion and Thresholding (Gian)

Module(s): signal_processing.sv, rgb_2_hsv.sv, thresholder.sv

The RGB to HSV conversion module takes in 8-bits for each of red, green and blue as inputs. The outputs include a 17-bit hue, 8-bit saturation and 8-bit value. They are all represented in fixed point notation, where the 17 bits for hue are partitioned into 9 bits for the integer and 8 bits for the fraction, and each of the 8 bits for saturation and value represent a fraction between 0 and 1. The module implements the equations below:

$$\begin{aligned} C_{\max} &= \max(R, G, B) \\ C_{\min} &= \min(R, G, B) \\ \Delta &= C_{\max} - C_{\min} \end{aligned}$$

$$\begin{aligned} H &= \begin{cases} 0 & \text{if } C_{\max} = 0 \\ \left(60 \times \frac{G-B}{\Delta} + 360\right) \bmod 360 & \text{if } R = C_{\max} \\ 60 \times \frac{B-R}{\Delta} + 120 & \text{if } G = C_{\max} \\ 60 \times \frac{R-G}{\Delta} + 240 & \text{if } B = C_{\max} \end{cases} \\ S &= \frac{\Delta}{C_{\max}} \\ V &= \frac{C_{\max}}{255} \end{aligned}$$

It was necessary to use divider modules to implement these equations. The divider modules would take 16 clock cycles to complete due to the dividend bit width for each being 16 bits wide. As such, the RGB to HSV module took around that many clock cycles to complete.

The binary_maze_filtering module does the RGB to HSV conversion, thresholding, erosion and dilation. It does so by reading a camera image pixel stored in the cam_image_buffer BRAM each clock cycle. However, we just mentioned that the RGB to HSV module contains dividers that take 16 clock cycles to complete. Because we wanted to quickly process the image at 1 pixel per clock cycle, we decided to instantiate enough rgb_2_hsv modules such that they could each process a pixel at the same time and finish in time to process the next pixel. We instantiated as many rgb_2_hsv modules as clock cycles it took to process one pixel.

The idea behind the thresholding is simple. We just set upper and lower bounds for the H, S, and V channels for each pixel and would output a 1 or a 0 depending on whether all the bounds were satisfied or not. In this way, we were able to create binary images representing paths, starting regions, and ending regions. Each of these binary images were represented as BRAM of size 1x76800.

3. Erosion (Luis)

Module(s): erosion.sv

We applied a K x K (here 5 x 5) kernel on the binary image to erode noise. We chose to ignore pixels where the kernel would be out of the image; in this case, we simply outputted a 0. For each pixel, we read a “window” of the image from BRAM. To do this, we read one pixel every clock cycle from BRAM to fill a (K -1) * IMG_W + K bit register. Using combinational logic, we were able to create the appropriate K x K window using this buffer to process one pixel every clock cycle. Since we are eroding, our output was the AND of all the pixels in the window; this output was fed directly to the dilation module. We used pixel_valid as the start signal for the dilation module. We stop eroding once we have processed every pixel in the image.

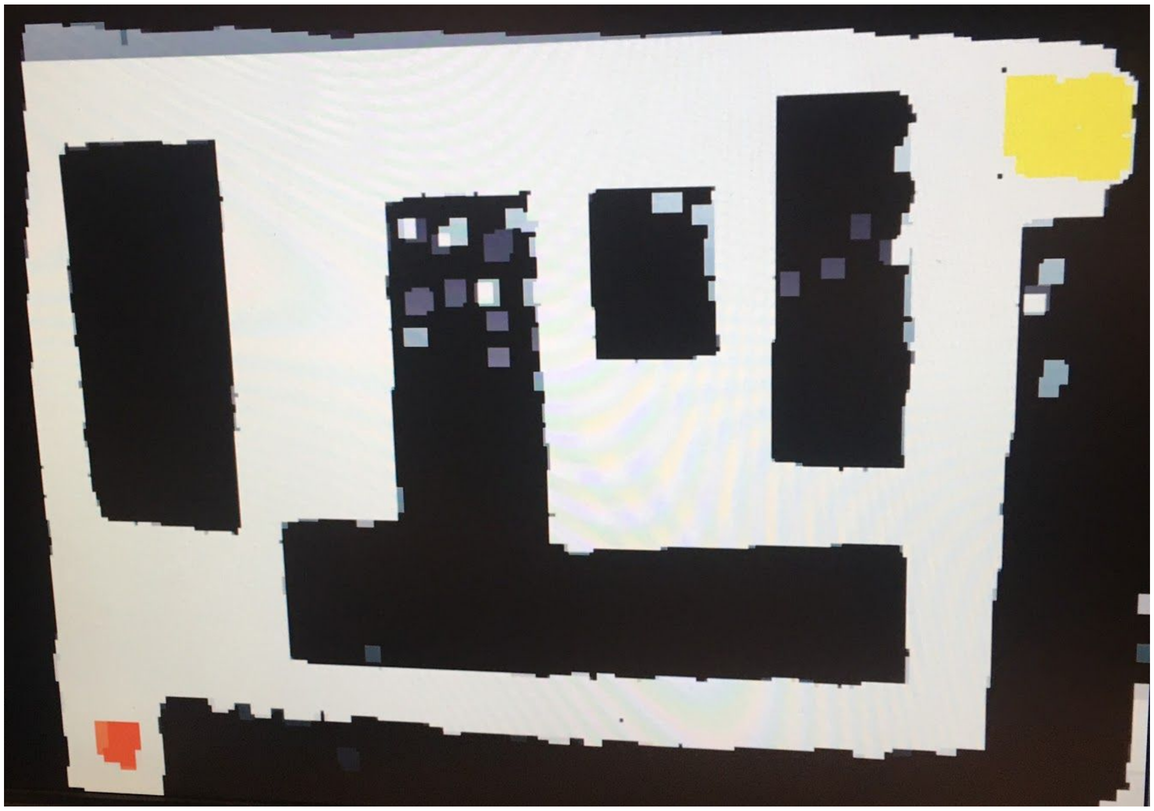
The most difficult part of erosion was finding a way to create the appropriate windows without having to do a series of reads from BRAM each time we wanted to process the next pixel. Initially, we considered having a K x K buffer that we could update by shifting in K new values from BRAM each time; however, this would result in around K + 2 clock cycles worth of delay each time we wanted to process a new pixel. Shifting in a new pixel from BRAM into a large buffer and creating the appropriate window using this buffer allowed us to process one pixel every clock cycle. Using this structure, each pixel we read from BRAM remains in the buffer from the first time it is needed until the last time it is needed, but it appears in the window only when it is relevant to the pixel we are processing. This optimization saved us numerous cycles of compute time during erosion and anywhere else where we need to create a window into an image stored in BRAM. Thus, this optimization was essential for solving our mazes in real-time.

4. Dilation (Gian)

Module(s): dilation.sv

This module was set up to apply a K x K window on every pixel. If the pixel had a value of 1, every pixel within the K x K window was also set to a 1. Otherwise, the pixels were left as they were. This was implemented through a similar technique as used in the erosion module, that was, using a (K -1) * IMG_W + K bit register. A register of this bit width was able to store all the pixels necessary to create a K x K window around the desired pixel. To obtain the window for the next pixel in the binary image, we simply shifted the contents of the register to the left, and inserted a new pixel at the 0th index. This was done every clock cycle. Unlike erosion, however, we had to modify all the pixels within the window, rather than just the center pixel. To achieve this, we added some combinational logic that created a mask ensuring that if the center pixel in the window was a 1, every pixel within the K x K window was set to a 1 as well. In this way, we were able to dilate one pixel per clock cycle.

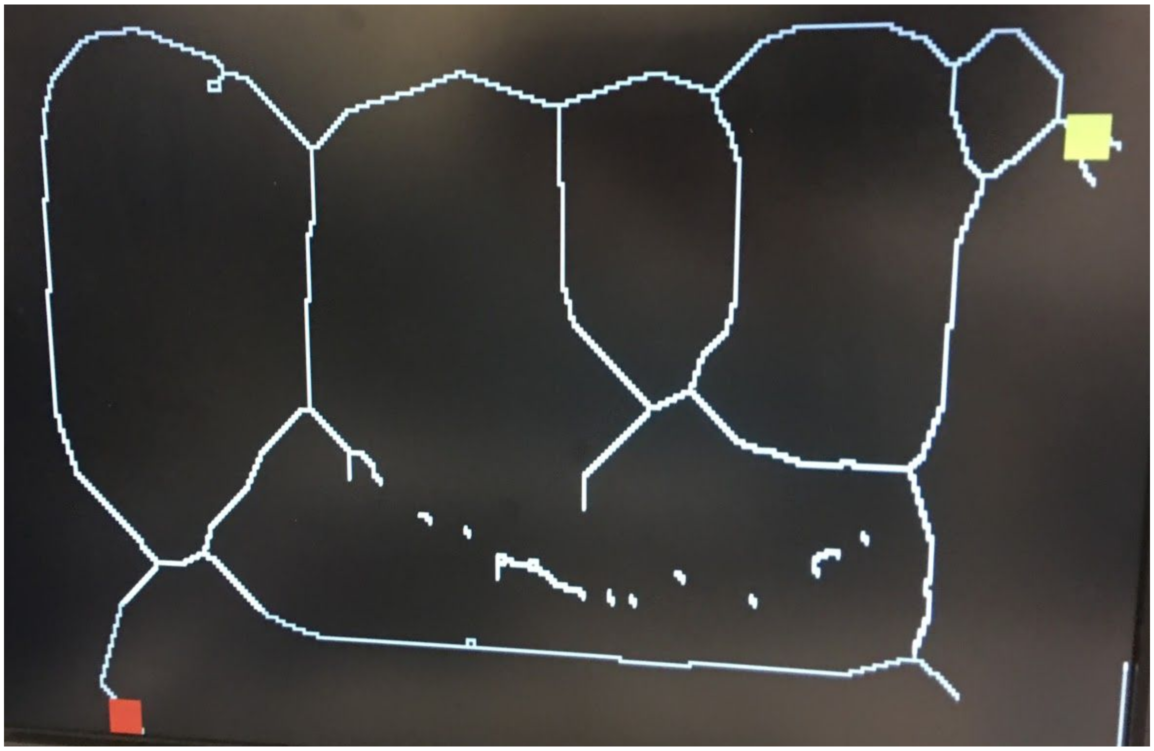
In our project, we only used 5 x 5 windows for our dilation kernels. They were used for smoothing out the binary images of the paths and of the start and end regions. Dilation was used to recover the remaining structures after erosion removed any noise. Below is an image showing 3 overlaid binary images. The black and white binary image represents the white paths (ones) and black walls (zeros). The yellow represents the start region and has a value of 1. The red is the end region and has a value of 1 as well. All three of these binary images were eroded and dilated with 5x5 kernels to remove any noise and were crucial for our system to work properly.



Maze after undergoing erosion and dilation; white represents the paths

5. Skeletonization (Luis & Vivian)

Module(s): skeletonizer.sv



Maze after undergoing skeletonization; white represents the paths

Skeletonization reduces a binary image to a skeletal remnant that largely preserves the extent and connectivity of the original region while throwing away most of the original foreground pixels. In our case, this process provides a compact yet effective representation of the maze paths.

We used eight 3x3 kernels that skeletonized until we had 1-pixel-wide possible paths without any discontinuities. In order to do this, we applied successive rounds of skeletonization until no further changes could be made. We applied the eight kernels to each pixel by first creating the appropriate window around it. We created this window exactly like we created windows during erosion and dilation—by using a buffer that we shifted every clock cycle and combinational logic. From here, we were able to apply the kernels in parallel and check if skeletonizing the pixel would cause a discontinuity. In order to check for discontinuity, we had to keep a second buffer and window that tracked our recent changes to the image during the current round of skeletonization. If the kernels showed that we were able to skeletonize the pixel and doing so would not have caused a discontinuity, we wrote the skeletonized pixel to BRAM and updated our skeletonized buffer; otherwise, we simply wrote the original pixel to BRAM and our skeletonized buffer. After we completed a round of skeletonization with no changes made to the image, we stopped. The fully skeletonized binary image was used as input to the maze solving module.

6. Maze Solving Algorithm (Gian)

Module(s): lees_algorithm.sv

This module implements Lee's Algorithm, which is based on the breadth-first search algorithm and finds an optimal solution to the maze if one exists. This module takes as inputs the skeletonized binary image of the maze, the start position, and the end position. As outputs, it writes to a 2-bit image where each pixel coordinate corresponds to a backpointer to the previous pixel along the shortest path from start to end. The backpointer values 00, 01, 10, and 11 represent movements movements from pixel (x, y) to pixels (x - 1, y), (x, y + 1), (x + 1, y) and (x, y - 1), respectively.

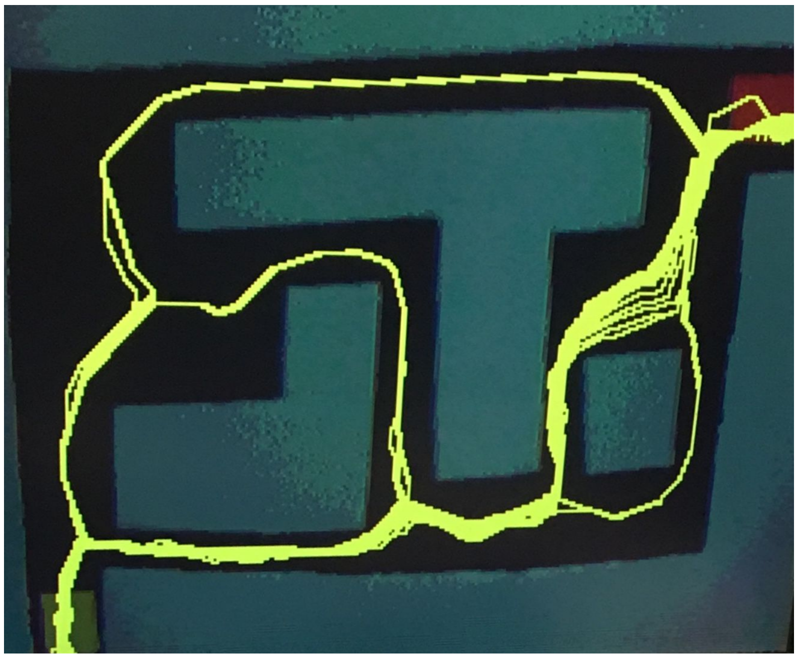
The module starts off in the IDLE state until the start signal is set HIGH, at which point we repeatedly transition back and forth between the FETCH_NEIGHBORS and the VALIDATE_NEIGHBORS states. In this algorithm, we have a queue that contains all the pixel coordinates at various depth levels from the starting position. Pixels coordinates that are closer to index 0 of the queue are closer to the start position than those at a higher index. During the FETCH_NEIGHBORS state, we take the pixel coordinate at queue[0] and store all 4 of its neighboring pixels. Then in the VALIDATE_NEIGHBORS state, we check for each of the 4 neighbors if it has been visited before and whether it lies along the skeletonized maze path. If the neighbor pixel coordinate has not been visited and has a value of 1 in the skeletonized binary image, we mark it as visited, record its backpointer, and store it in the queue to obtain its neighbors at a later time. If the neighbor pixel is the end position, we stop the algorithm, as we have found the minimum length path that solves the maze. Otherwise, we ignore the neighbor pixel.

Once we have found the end position, we mark each pixel coordinate as not visited. This is done in the CLEAR_VISITED_MAP state, and it just writes a 0 to each address of visited_map, which is a BRAM of size 1x76800. Each address of this BRAM corresponds to a pixel coordinate, and has a value of 1 if it has been visited and a value of 0 if it has not been visited. Clearing this BRAM is necessary to reset the visited state of each pixel coordinate and allow Lee's algorithm to be run again.

7. Path Display (Gian & Vivian)

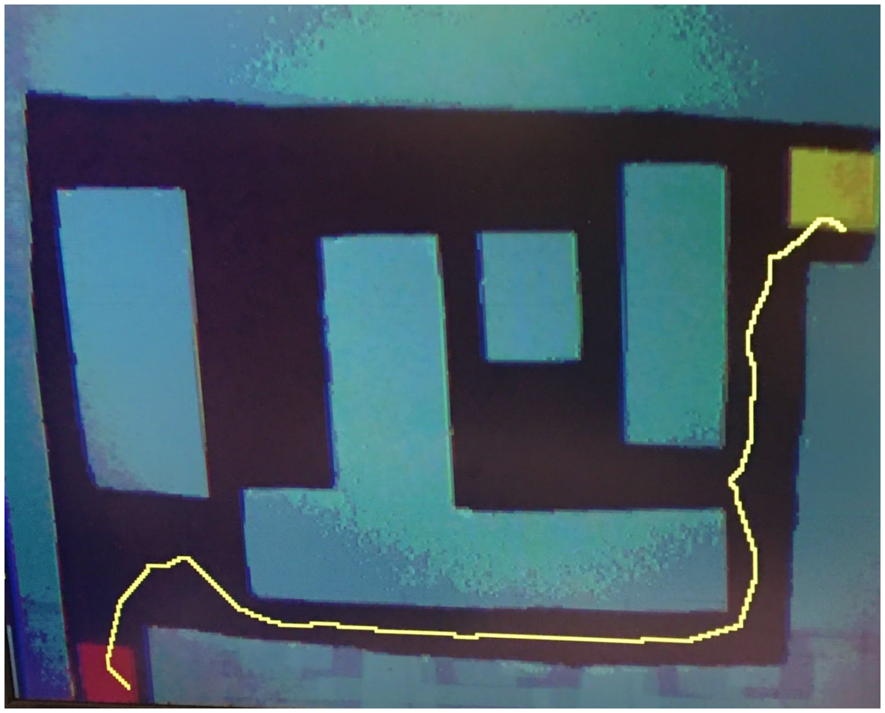
Module(s): backpointer_tracer.sv, top_level.sv, path_car_drawer.sv

Our backpointer tracer module takes in the output of Lee's algorithm, where each pixel location maps to a displacement that defines the next pixel along the shortest path to the start position. This module also takes in the end position, which is where we start tracing the path from. The output is a 2D image where a pixel address maps to a 1 if it is part of the shortest path, or a 0 otherwise.



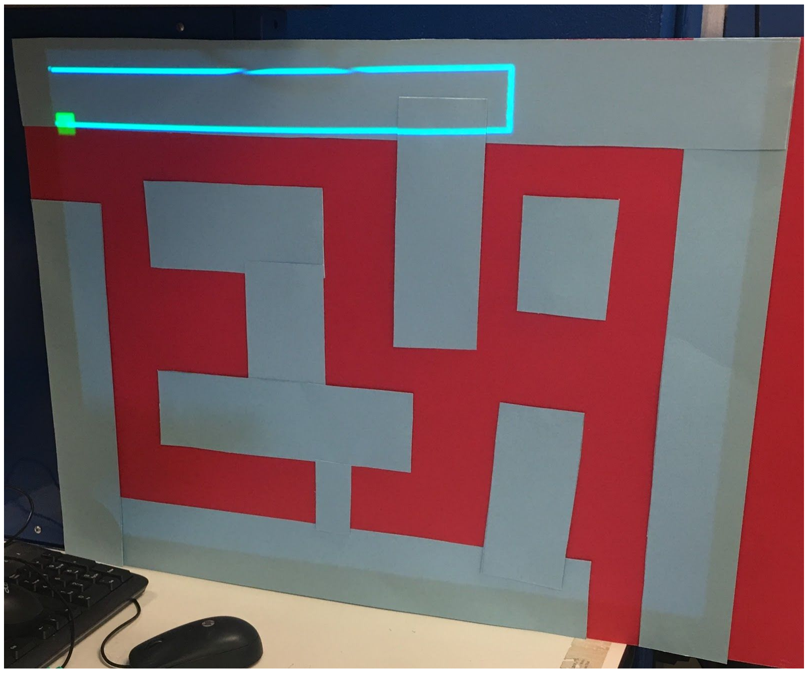
New solutions displayed after physically blocking already-displayed solutions

As seen by the multiple paths displayed on the monitor in the above figure, we initially stored all paths in the same BRAM of size 1x76800. We then added the CLEAR_PATH state to our state machine in top_level.sv to clear the BRAM for each new solution. The resulting display of only one solution at a time is shown in the following figure.



Displaying the shortest-path solution (yellow line) on the monitor

Given a couple of more days, we would have integrated the projection of the solved path as well as an image moving along the path onto the physical maze. To this end, we wrote a module (path_car_drawer.sv) which takes backpointers from bram0 to create a path and then stores the path in bram1 to display. For the moving object, we also used backpointer information from bram0 to move a green square along the path. Because the position updated at the negative edge of VSYNC to sync with the start of each frame, and because VGA display signals were generated at 60Hz with traversal running 2px at a time through a 640px display width, it took approximately five seconds for the square to move from the left side of the screen to the right—a speed we deemed satisfactory. However, fully integrating this feature would have involved changing the physical setup to align both the camera and the projector with the maze (perhaps introducing the need to keystone the projection), as well as accounting for thresholding changes due to the camera taking images as the projector illuminated the maze. Nevertheless, we were satisfied with the way the solution was clearly displayed on the computer monitor.



Testing path projection using a predefined path

V. Main Challenges

- Color thresholding
    - Some colors for maze walls or start and end regions were difficult to threshold for. This process required much trial and error with different colors, materials, and threshold ranges.
- Noise
    - Lighting in the room and reflectance of objects added noise to the images captured by the camera and made the image processing less robust. We tried to minimize this effect by using non-reflective maze materials and angling the music stand that the maze was propped up on slightly downward, which helped avoid glare from the ceiling lights.

VI. Lessons Learned & Advice

Gian:

- Make sure to allot enough time for debugging after integrating all the modules together. Most of the bugs we encountered resulted from having fully working and tested modules interacting with each other.
- Be realistic with what you can accomplish on an FPGA. Algorithms that seem easy to implement on a CPU are not always easy to implement on an FPGA.
- Make sure that you will have the resources on the FPGA to handle the project you are doing. This advice applies especially to image processing / computer graphics projects where it is necessary to store images.

Vivian:

- Maintain high-level organization and structure of the codebase. Although we were lucky to be mostly aware of each other's parameters and module purposes, for a larger project we would absolutely need to keep track of our modules in an organized document.
- Begin integrating the project pipeline early. It's good to approach a project by determining the modules and assigning them to team members, but be aware that integrating the code may introduce an unanticipated set of problems (e.g. different clock setups or inputs/outputs between modules). I learned that it saves panic to encounter these problems sooner than later.
- Looking back, we could have made our code more ready for change. Especially while we lingered as lab closed, waiting for "just one more bitstream" to compile, we recognized the importance of setting tunable values (e.g. color thresholds) on the FPGA board to save time.

Luis:

- It is important to have constant communication with your team. When we first started our project, we made an overall plan and divided our work up; however, we each had to make design changes as we encountered problems/complications we had not anticipated. These design changes resulted in slight changes that trickled down to later stages in our pipeline. In other words, we each ended up having to adapt our code in order for it to play well with code the other two had written.

VII. Future Work & Conclusion

Overall, we are proud to have successfully made *Gim's Labyrinth*. Furthermore, we have several ideas for extensions of this project. For example, we would find it interesting to gamify our system. To do this, we have already written a module for Djikstra's Shortest Path algorithm. Dijkstra's algorithm is a general version of breadth-first search where edge weights are no longer considered to be equal. Using Dijkstra's, we could implement a game where the user places prizes along maze paths to add more value to the corresponding edge and the solution is the path with the highest value. Another one of our ideas is to add a physical element traversing through the determined shortest path. This could involve a marble on a tiltable maze setup, or a four-wheeled robot traversing the maze on its own.

VIII. Acknowledgements

We would like to sincerely thank many people for their support. Professor Gim Hom and our project mentor, Mike Wang, guided us through project design and answered important questions about implementation details. We appreciate Joe Steinmeyer's help with the camera as well as his dedication to answering Piazza questions and keeping the lab open. Additionally, Diana Wofk was immensely helpful with debugging and explaining concepts to our group. Finally, we would like to express our gratitude to the entire 6.111 course staff for their dedication, positivity, and great sense of humor throughout the term. Thank you!

VIV. Appendix / Verilog Source

GitHub Repository Link:
https://github.com/TheEpicDolphin/gims-labyrinth/tree/master/gims_labyrinth

top_level.sv

```
module top_level(
   input clk_100mhz,
   input[15:0] sw,
   input btnc, btnu, btnl, btnr, btnd,
   input [7:0] ja,
   input [2:0] jb,
   output   jbclk,
   input [2:0] jd,
   output   jdclk,
   output[3:0] vga_r,
   output[3:0] vga_b,
   output[3:0] vga_g,
   output vga_hs,
   output vga_vs,
   output led16_b, led16_g, led16_r,
   output led17_b, led17_g, led17_r,
   output[15:0] led,
   output ca, cb, cc, cd, ce, cf, cg, dp,  // segments a-g, dp
   output[7:0] an    // Display location 0-7
   );
    logic clk_25mhz;
    // create 25mhz system clock, happens to match 640 x 480 VGA timing
    clk_wiz_0 clkdivider(.reset(0),.clk_in1(clk_100mhz),
.clk_out1(clk_25mhz));

    wire [9:0] hcount;    // pixel on current line
    wire [9:0] vcount;     // line number
    wire hsync, vsync, blank;
    wire [11:0] pixel;
    reg [11:0] rgb;
    vga vga1(.vclock_in(clk_25mhz),.hcount_out(hcount),.vcount_out(vcount),
          .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));
    // btnc button is user reset
    wire reset;
    wire reset_cam;
    wire up,down;
    debounce
db1(.reset_in(0),.clock_in(clk_25mhz),.noisy_in(btnc),.clean_out(reset));

    logic xclk;

    logic pclk_buff, pclk_in;
    debounce
db2(.reset_in(0),.clock_in(pclk_in),.noisy_in(btnc),.clean_out(reset_cam));
    logic vsync_buff, vsync_in;
    logic href_buff, href_in;
    logic[7:0] pixel_buff, pixel_in;

    logic [15:0] output_pixels;
    logic [15:0] old_output_pixels;
    logic [12:0] processed_pixels;
    logic valid_pixel;
    logic frame_done_out;

    logic [16:0] pixel_addr_in;
    logic [16:0] pixel_addr_out;

    assign xclk = clk_25mhz;    //changed for vga
    assign jbclk = xclk;
    assign jdclk = xclk;


    parameter IMG_W = 320;
    parameter IMG_H = 240;

    parameter IDLE = 3'b000;
    parameter CAPTURE_IMAGE = 3'b001;
    parameter BINARY_MAZE_FILTERING = 3'b010;
    parameter SKELETONIZING = 3'b011;

    parameter FIND_END_NODES = 3'b100;
    parameter CLEAR_PATH = 3'b101;
    parameter SOLVING = 3'b110;
    parameter TRACING_BACKPOINTERS = 3'b111;

    logic [2:0] state;
    logic [1:0] cam_state;
    wire [31:0] data;      //  instantiate 7-segment display; display (8)
4-bit hex
    wire [6:0] segments;
    assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];
    display_8hex display(.clk_in(clk_25mhz),.data_in(data),
.seg_out(segments), .strobe_out(an));
    //assign seg[6:0] = segments;
    assign  dp = 1'b1;  // turn off the period


    logic [16:0] end_pos;

    assign led = sw;                        // turn leds on

    logic did_lees_work;
    assign data = did_lees_work ? {14'b0, state, 14'b0, 2'b10} : {14'b0,
state, 14'b0, 2'b00};
    //assign data = {14'b0, cam_state, 13'b0, state};   // display 0123456
+ sw[3:0]
    //assign data = {15'b0, end_pos[16], end_pos[15:0]};

    assign led16_r = btnl;                  // left button -> red led
    assign led16_g = btnc;                  // center button -> green led
    assign led16_b = btnr;                  // right button -> blue led
    assign led17_r = btnl;
    assign led17_g = btnc;
    assign led17_b = btnr;

    logic [16:0] cam_pixel_wr_addr;
    logic [16:0] cam_pixel_r_addr;

    logic cam_pixel_we;
    logic [11:0] rgb_pixel;

    cam_image_buffer cam_img_buf(.clka(pclk_in),
                            .addra(cam_pixel_wr_addr),

.dina({output_pixels[15:12],output_pixels[10:7],output_pixels[4:1]}),
                                .wea(cam_pixel_we),
                                .clkb(clk_25mhz),
                                .addrb(cam_pixel_r_addr),
                                .doutb(rgb_pixel));

    logic [16:0] bin_pixel_wr_addr;
    logic bin_pixel_in;
    logic bin_pixel_we;
    logic [16:0] bin_pixel_r_addr;
    logic bin_pixel_out;

    binary_maze skel_maze(.clka(clk_25mhz),
                          .addra(bin_pixel_wr_addr),
                          .dina(bin_pixel_in),
                          .wea(bin_pixel_we),
                          .clkb(clk_25mhz),
                          .addrb(bin_pixel_r_addr),
                          .doutb(bin_pixel_out)
                          );

    logic [16:0] start_end_wr_addr;
    logic [16:0] start_end_r_addr;
    logic start_end_wea;
    logic start_color_wr;
    logic end_color_wr;
    logic start_color_r;
    logic end_color_r;



    logic bin_maze_filt_start;
    logic bin_maze_filt_done;
    logic [16:0] filt_pixel_wr_addr;
    logic filt_pixel;
    logic filt_pixel_we;
    logic [16:0] filt_pixel_r_addr;

    binary_maze start_color_map(.clka(clk_25mhz),
                        .addra(filt_pixel_wr_addr),
                        .dina(start_color_wr),
                        .wea(filt_pixel_we),
                        .clkb(clk_25mhz),
                        .addrb(start_end_r_addr),
                        .doutb(start_color_r)
                        );

    binary_maze end_color_map(.clka(clk_25mhz),
                          .addra(filt_pixel_wr_addr),
                          .dina(end_color_wr),
                          .wea(filt_pixel_we),
                          .clkb(clk_25mhz),
                          .addrb(start_end_r_addr),
                          .doutb(end_color_r)
                          );

    binary_maze_filtering #(.IMG_W(IMG_W),.IMG_H(IMG_H)) bin_maze_filt
        (
         .clk(clk_25mhz),
         .rst(reset),
         .start(bin_maze_filt_start),
         .rgb_pixel(rgb_pixel),
         .cam_pixel_r_addr(filt_pixel_r_addr),
         .done(bin_maze_filt_done),
         .pixel_wr_addr(filt_pixel_wr_addr),
         .pixel_wea(filt_pixel_we),
         .pixel_out(filt_pixel),

         .start_color(start_color_wr),
         .end_color(end_color_wr)
         );

    logic start_skeletonizer;
    logic skeletonizer_done;
    logic [16:0] skel_pixel_wr_addr;
    logic skel_pixel;
    logic skel_pixel_we;
    logic [16:0] skel_pixel_r_addr;
    skeletonizer
#(.IMG_WIDTH(IMG_W),.IMG_HEIGHT(IMG_H),.BRAM_READ_DELAY(2)) skel
    (
             .clk(clk_25mhz),
             .rst(reset),
             .start(start_skeletonizer),
             .pixel_in(bin_pixel_out),
            .pixel_r_addr(skel_pixel_r_addr),
            .pixel_wr_addr(skel_pixel_wr_addr),
            .pixel_we(skel_pixel_we),
            .pixel_out(skel_pixel),
            .done(skeletonizer_done)
            );

    logic start_end_node_finder;
    logic end_node_finder_done;
    logic [16:0] start_pos;
    //logic [16:0] end_pos;

    logic [16:0] start_pos_out;
    logic [16:0] end_pos_out;

    logic [16:0] maze_r_addr;

    end_node_finder #(.IMG_W(IMG_W),.IMG_H(IMG_H),.BRAM_READ_DELAY(2))
        (
            .clk(clk_25mhz),
            .start(start_end_node_finder),
            .rst(reset),
            .skel_pixel(bin_pixel_out), // added by viv
            .start_yellow_pixel(start_color_r), // added by viv
            .end_red_pixel(end_color_r), // added by viv
            .done(end_node_finder_done),
            .maze_pixel_addr(maze_r_addr),
            .start_pos(start_pos_out),
            .end_pos(end_pos_out)
        );

    logic start_lees_alg;
    logic path_found;
    logic lees_alg_done;
    logic [16:0] bp_wr_addr;
    logic [1:0] bp;
    logic bp_we;
    logic [1:0] bp_r;
    logic [16:0] bp_tracer_addr, bp_tracer_addr1, cp_bp_tracer_addr;
    logic [16:0] solver_pixel_r_addr;
    pixel_backpointers p_bp(.clka(clk_25mhz),
                            .addra(bp_wr_addr),
                            .dina(bp),
                            .wea(bp_we),
                            .clkb(clk_25mhz),
                            .addrb(bp_tracer_addr),
                            .doutb(bp_r));



    lees_algorithm #(.MAX_OUT_DEGREE(4),.BRAM_DELAY_CYCLES(2),
                     .IMG_W(IMG_W),.IMG_H(IMG_H)) maze_solver
                (
                 .clk(clk_25mhz),
                 .rst(reset),
                 .start(start_lees_alg),
                 .start_pos(start_pos),
                 .end_pos(end_pos),
                 .skel_pixel(bin_pixel_out),
                 .pixel_r_addr(solver_pixel_r_addr),
                 .pixel_wr_addr(bp_wr_addr),
                 .backpointer(bp),
                 .bp_we(bp_we),
                 .done(lees_alg_done),
                 .success(path_found)
                 );



   logic start_bp_tracer;
   logic bp_tracer_done;
   logic write_path, write_path1, write_path2, cp_write_path1,
cp_write_path2;
   backpointer_tracer #(.BRAM_DELAY_CYCLES(2),.IMG_W(IMG_W),.IMG_H(IMG_H))
bp_tracer
                (
                 .clk(clk_25mhz),
                 .rst(reset),
                 .start(start_bp_tracer),
                 .start_pos(start_pos),
                 .end_pos(end_pos),
                 .bp(bp_r),
                  .pixel_addr(bp_tracer_addr),
                  .write_path(write_path),
                  .done(bp_tracer_done)
                 );

    logic [16:0] path_r_addr;
    logic path_pixel;
    path_bram pb(.clka(clk_25mhz),
                .addra(bp_tracer_addr1),
                .dina(write_path1),
                .wea(write_path2),
                .clkb(clk_25mhz),
                .addrb(path_r_addr),
                .doutb(path_pixel));

    logic [16:0] fpga_read_addr;

    always_comb begin
        case(state)
            IDLE: begin
                cam_pixel_r_addr = fpga_read_addr;
                bin_pixel_r_addr = fpga_read_addr;
                start_end_r_addr = fpga_read_addr;
                path_r_addr = fpga_read_addr;
            end
            CAPTURE_IMAGE: begin
                cam_pixel_r_addr = fpga_read_addr;
                bin_pixel_r_addr = fpga_read_addr;
                start_end_r_addr = fpga_read_addr;
                path_r_addr = fpga_read_addr;
            end
            BINARY_MAZE_FILTERING: begin
                bin_pixel_wr_addr = filt_pixel_wr_addr;
                bin_pixel_in = filt_pixel;
                bin_pixel_we = filt_pixel_we;
                cam_pixel_r_addr = filt_pixel_r_addr;

                bin_pixel_r_addr = fpga_read_addr;
                path_r_addr = fpga_read_addr;
            end
            SKELETONIZING: begin
                bin_pixel_wr_addr = skel_pixel_wr_addr;
                bin_pixel_in = skel_pixel;
                bin_pixel_we = skel_pixel_we;
                bin_pixel_r_addr = skel_pixel_r_addr;
                cam_pixel_r_addr = fpga_read_addr;
                path_r_addr = fpga_read_addr;
            end
            FIND_END_NODES: begin
                bin_pixel_r_addr = maze_r_addr;
                start_end_r_addr = maze_r_addr;

                cam_pixel_r_addr = fpga_read_addr;
                path_r_addr = fpga_read_addr;
            end
            SOLVING: begin
                bin_pixel_r_addr = solver_pixel_r_addr;

                cam_pixel_r_addr = fpga_read_addr;
                path_r_addr = fpga_read_addr;
            end
            TRACING_BACKPOINTERS: begin
                cam_pixel_r_addr = fpga_read_addr;
                path_r_addr = fpga_read_addr;
            end

        endcase
    end

    assign write_path1 = (state==CLEAR_PATH)? cp_write_path1 : write_path;
    assign write_path2 = (state==CLEAR_PATH)? cp_write_path2 : write_path;
    assign bp_tracer_addr1 = (state==CLEAR_PATH)? cp_bp_tracer_addr :
bp_tracer_addr;


    logic frame_done;
    logic [3:0] delay;
    always_ff @(posedge clk_25mhz)begin
        if(reset)begin
            state <= IDLE;
        end
        else begin
            case(state)
                IDLE: begin
```

```
                    if(frame_done)begin
                        //Wait for camera to finish current incomplete
frame
                        state <= CAPTURE_IMAGE;
                        delay <= 0;
                    end
                end
                CAPTURE_IMAGE: begin
                    if(delay < 2)begin
                        delay <= delay + 1;
                    end
                    else begin
                        if(frame_done)begin
                            //camera image is now stored in bram
                            state <= BINARY_MAZE_FILTERING;
                            bin_maze_filt_start <= 1;
                        end
                    end
                end
                BINARY_MAZE_FILTERING: begin
                    bin_maze_filt_start <= 0;
                    if(bin_maze_filt_done)begin
                        if(sw[1:0] == 2'b01)begin
                            //If want to skip skeletonization
                            state <= IDLE;
                        end
                        else begin
                            state <= SKELETONIZING;
                            start_skeletonizer <= 1;
                        end
                    end
                end
                SKELETONIZING: begin
                    start_skeletonizer <= 0;
                    //get stuck in here for now
                    if(skeletonizer_done)begin
                        if(sw[1:0] == 2'b10)begin
                            //If want to skip FIND_END_NODES
                            state <= IDLE;
                        end
                        else begin
                            state <= FIND_END_NODES;
                            start_end_node_finder <= 1;
                        end
                    end
                end
                FIND_END_NODES: begin
                    start_end_node_finder <= 0;
                    if(end_node_finder_done)begin
                        //If want to skip Lee's algorithm
                        if(sw[1:0] == 2'b11)begin
                            state <= IDLE;
                        end
                        else begin
                            state <= CLEAR_PATH;
                            cp_write_path1 <= 0;
                            cp_write_path2 <= 1;
                            cp_bp_tracer_addr <= 0;
                            start_pos <= start_pos_out;
                            end_pos <= end_pos_out;
                        end
                    end

                end
                CLEAR_PATH: begin
                    if (cp_bp_tracer_addr < 76800) begin
                        cp_bp_tracer_addr <= cp_bp_tracer_addr + 1;
                        state <= CLEAR_PATH;
                    end else begin
                        start_lees_alg <= 1;
                        state <= SOLVING;
                    end
                end

                SOLVING: begin
                    start_lees_alg <= 0;
                    if(lees_alg_done)begin
                        did_lees_work <= path_found;
                        if(path_found)begin
                            //Path found, now we can trace the backpointers
and display the path
                            state <= TRACING_BACKPOINTERS;
                            start_bp_tracer <= 1;
                        end
                        else begin
                            //no path found, take another image with the
camera
                            state <= IDLE;
                        end

                    end
                end
                TRACING_BACKPOINTERS: begin
                    start_bp_tracer <= 0;
                    if(bp_tracer_done)begin
                        state <= IDLE;
                    end
                end
            endcase
        end
    end


    logic frame_done_buff;
    always_ff @(posedge clk_25mhz) begin
        if(reset)begin
            frame_done_buff <= 0;
            frame_done <= 0;
        end
        else begin
            //Comes from camera sensor
            pclk_buff <= jb[0];//WAS JB
            vsync_buff <= jb[1]; //WAS JB
            href_buff <= jb[2]; //WAS JB
            pixel_buff <= ja;

            pclk_in <= pclk_buff;
            vsync_in <= vsync_buff;
            href_in <= href_buff;
            pixel_in <= pixel_buff;

            frame_done_buff <= frame_done_out;
            frame_done <= frame_done_buff;
        end

    end
```

```
logic sync_state;
logic state_buff;
always_ff @(posedge pclk_in) begin
    if(reset_cam)begin
        sync_state <= IDLE;
        state_buff <= IDLE;
    end
    else begin
        state_buff <= state;
        sync_state <= state_buff;
        if(frame_done_out)begin
            cam_pixel_wr_addr <= 0;
        end
        else if(valid_pixel)begin
            cam_pixel_wr_addr <= cam_pixel_wr_addr + 1;
        end
    end

end

assign cam_pixel_we = valid_pixel && (sync_state == CAPTURE_IMAGE);

camera_read  my_camera(.p_clock_in(pclk_in),
                         .rst(reset),
                       .vsync_in(vsync_in),
                       .href_in(href_in),
                       .p_data_in(pixel_in),
                       .pixel_data_out(output_pixels),
                       .pixel_valid_out(valid_pixel),
                       .frame_done_out(frame_done_out),
                       .FSM_state(cam_state));


wire border = (hcount==0 | hcount==639 | vcount==0 | vcount==479 |
               hcount == 320 | vcount == 240);

parameter HALF_WIDTH = 5;
parameter HALF_HEIGHT = 5;
reg b,hs,vs;
always_ff @(posedge clk_25mhz) begin
  if(reset)begin
```

```
      end
      else begin
          hs <= hsync;
          vs <= vsync;
          b <= blank;
          if (sw[1:0] == 2'b01 || sw[1:0] == 2'b10) begin
             if(state == IDLE || state == CAPTURE_IMAGE)begin
                fpga_read_addr <= (hcount>>1)+ ((vcount>>1) * IMG_W);

                if(start_color_r)begin
                    //yellow
                    rgb <= 12'hFF0;
                end
                else if(end_color_r)begin
                    //red
                    rgb <= 12'hF30;
                end
                else begin
                    rgb <= bin_pixel_out ? 12'hFFF : 12'b0;
                end
             end
          end
          else if(sw[1:0] == 2'b11) begin
            //Display skeletonization with start and end blobs on top of
skeletonized path
             if(state == IDLE || state == CAPTURE_IMAGE)begin
                fpga_read_addr <= (hcount>>1)+ ((vcount>>1) * IMG_W);


                if (((hcount >> 1) >= (start_pos[16:8] - HALF_WIDTH) &&
(hcount >> 1) < (start_pos[16:8] + HALF_WIDTH)) &&
                    ((vcount >> 1) >= (start_pos[7:0] - HALF_HEIGHT) &&
(vcount >> 1) < (start_pos[7:0] + HALF_HEIGHT)))begin
                    //yellow
                    rgb <= 12'hFF0;
                end
                else if (((hcount >> 1) >= (end_pos[16:8] - HALF_WIDTH) &&
(hcount >> 1) < (end_pos[16:8] + HALF_WIDTH)) &&
                    ((vcount >> 1) >= (end_pos[7:0] - HALF_HEIGHT) &&
(vcount >> 1) < (end_pos[7:0] + HALF_HEIGHT)))begin
                    //red
                    rgb <= 12'hF30;
                end
                else begin
                    rgb <= bin_pixel_out ? 12'hFFF : 12'b0;
                end

                /*
                if (((hcount >> 1) == start_pos[16:8]) && ((vcount >> 1) ==
start_pos[7:0]))begin
                    //yellow
                    rgb <= 12'hFF0;
                end
                else if (((hcount >> 1) == end_pos[16:8]) && ((vcount >> 1)
== end_pos[7:0]))begin
                    //red
                    rgb <= 12'hF30;
                end
                else begin
                    rgb <= bin_pixel_out ? 12'hFFF : 12'b0;
                end
                */
              end

            end
            else begin
                //Display camera image with solved path drawn over it
                fpga_read_addr <= (hcount>>1)+ ((vcount>>1) * IMG_W);
                if(path_pixel)begin
                    rgb <= 12'hFF0;
                end
                else begin
                    rgb <= rgb_pixel;
                end
            end
        end

    end

    // the following lines are required for the Nexys4 VGA circuit - do not
change
    assign vga_r = ~b ? rgb[11:8]: 0;
    assign vga_g = ~b ? rgb[7:4] : 0;
    assign vga_b = ~b ? rgb[3:0] : 0;

    assign vga_hs = ~hs;
    assign vga_vs = ~vs;

endmodule

////////////////////////////////////////////////////////////////////////////
/////
//
// camera_read
//
////////////////////////////////////////////////////////////////////////////
/////

module camera_read(
      input  p_clock_in,
      input rst,
      input  vsync_in,
      input  href_in,
      input  [7:0] p_data_in,
      output logic [15:0] pixel_data_out,
      output logic pixel_valid_out,
      output logic frame_done_out,
      output logic [1:0] FSM_state
    );


      //logic [1:0] FSM_state = 0;
    logic pixel_half = 0;

      localparam WAIT_FRAME_START = 0;
      localparam ROW_CAPTURE = 1;


      always_ff@(posedge p_clock_in) begin
          if(rst)begin
             FSM_state <= WAIT_FRAME_START;
          end
          else begin
             case(FSM_state)
                    WAIT_FRAME_START: begin //wait for VSYNC
```

```
                       FSM_state <= (!vsync_in) ? ROW_CAPTURE :
WAIT_FRAME_START;
                       frame_done_out <= 0;
                       pixel_half <= 0;
                    end

                    ROW_CAPTURE: begin
                       FSM_state <= vsync_in ? WAIT_FRAME_START :
ROW_CAPTURE;
                       frame_done_out <= vsync_in ? 1 : 0;
                       pixel_valid_out <= (href_in && pixel_half) ? 1 : 0;
                       if (href_in) begin
                           pixel_half <= ~ pixel_half;
                           if (pixel_half) pixel_data_out[7:0] <=
p_data_in;
                           else pixel_data_out[15:8] <= p_data_in;
                       end
                    end
                endcase
          end

      end

endmodule

///////////////////////////////////////////////////////////////////////////
////
//
// Pushbutton Debounce Module (video version - 24 bits)
//
///////////////////////////////////////////////////////////////////////////
////

module debounce (input reset_in, clock_in, noisy_in,
                 output reg clean_out);

   reg [19:0] count;
   reg new_input;

//   always_ff @(posedge clock_in)
//     if (reset_in) begin new <= noisy_in; clean_out <= noisy_in; count <=
0; end
```

```
//     else if (noisy_in != new) begin new <= noisy_in; count <= 0; end
//     else if (count == 200000) clean_out <= new;
//     else count <= count+1;

   always_ff @(posedge clock_in)
     if (reset_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
        count <= 0; end
     else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0;
end
     else if (count == 200000) clean_out <= new_input;
     else count <= count+1;


endmodule

///////////////////////////////////////////////////////////////////////////
///////
// Engineer:   g.p.hom
//
// Create Date:    18:18:59 04/21/2013
// Module Name:    display_8hex
// Description:  Display 8 hex numbers on 7 segment display
//
///////////////////////////////////////////////////////////////////////////
///////

module display_8hex(
    input clk_in,                 // system clock
    input [31:0] data_in,         // 8 hex numbers, msb first
    output reg [6:0] seg_out,     // seven segment display output
    output reg [7:0] strobe_out   // digit strobe
    );

    localparam bits = 13;

    reg [bits:0] counter = 0;  // clear on power up

    wire [6:0] segments[15:0]; // 16 7 bit memorys
    assign segments[0]  = 7'b100_0000;  // inverted logic
    assign segments[1]  = 7'b111_1001;  // gfedcba
    assign segments[2]  = 7'b010_0100;
    assign segments[3]  = 7'b011_0000;
    assign segments[4]  = 7'b001_1001;
    assign segments[5]  = 7'b001_0010;
    assign segments[6]  = 7'b000_0010;
    assign segments[7]  = 7'b111_1000;
    assign segments[8]  = 7'b000_0000;
    assign segments[9]  = 7'b001_1000;
    assign segments[10] = 7'b000_1000;
    assign segments[11] = 7'b000_0011;
    assign segments[12] = 7'b010_0111;
    assign segments[13] = 7'b010_0001;
    assign segments[14] = 7'b000_0110;
    assign segments[15] = 7'b000_1110;

    always_ff @(posedge clk_in) begin
      // Here I am using a counter and select 3 bits which provides
      // a reasonable refresh rate starting the left most digit
      // and moving left.
      counter <= counter + 1;
      case (counter[bits:bits-2])
          3'b000: begin  // use the MSB 4 bits
                  seg_out <= segments[data_in[31:28]];
                  strobe_out <= 8'b0111_1111 ;
                 end

          3'b001: begin
                  seg_out <= segments[data_in[27:24]];
                  strobe_out <= 8'b1011_1111 ;
                 end

          3'b010: begin
                   seg_out <= segments[data_in[23:20]];
                   strobe_out <= 8'b1101_1111 ;
                  end
          3'b011: begin
                  seg_out <= segments[data_in[19:16]];
                  strobe_out <= 8'b1110_1111;
                 end
          3'b100: begin
                  seg_out <= segments[data_in[15:12]];
                  strobe_out <= 8'b1111_0111;
                 end

          3'b101: begin
                  seg_out <= segments[data_in[11:8]];
                  strobe_out <= 8'b1111_1011;
                 end

          3'b110: begin
                   seg_out <= segments[data_in[7:4]];
                   strobe_out <= 8'b1111_1101;
                  end
          3'b111: begin
                  seg_out <= segments[data_in[3:0]];
                  strobe_out <= 8'b1111_1110;
                 end

       endcase
      end

endmodule

///////////////////////////////////////////////////////////////////////////
///////
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// vga: Generate VGA display signals (640 x 480 @ 60Hz)
//
//                                ---- HORIZONTAL -----     ------VERTICAL
-----
//                                Active                    Active
//                      Freq      Video   FP  Sync   BP      Video   FP  Sync
BP
//   640x480, 60Hz     25.175    640     16    96   48       480    11    2
31
//   800x600, 60Hz     40.000    800     40   128   88       600     1    4
23
//   1024x768, 60Hz    65.000    1024    24   136  160       768     3    6
29
//   1280x1024, 60Hz  108.00     1280    48   112  248       768     1    3
38
```

```
//   1280x720p 60Hz   75.25     1280    72    80  216       720     3   5
30
//   1920x1040 60Hz   148.5     1920    88    44  148      1080     4   5
36
//
// change the clock frequency, front porches, sync's, and back porches to
create
// other screen resolutions
///////////////////////////////////////////////////////////////////////////
/////

module vga(input vclock_in,
            output reg [9:0] hcount_out,    // pixel number on current line
            output reg [9:0] vcount_out,     // line number
            output reg vsync_out, hsync_out,
            output reg blank_out);

   parameter DISPLAY_WIDTH  = 640;      // display width
   parameter DISPLAY_HEIGHT = 480;       // number of lines

   parameter  H_FP = 16;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 96;        // horizontal sync
   parameter  H_BP = 48;                // horizontal back porch

   parameter  V_FP = 11;                  // vertical front porch
   parameter  V_SYNC_PULSE = 2;          // vertical sync
   parameter  V_BP = 31;                 // vertical back porch

   // horizontal: 800 pixels total
   // display 640 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //655
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE -
1));  // 751
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE +
H_BP - 1));  //799

   // vertical: 524 lines total
   // display 480 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   //
479
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));
// 490
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP +
V_SYNC_PULSE - 1));  // 492
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP +
V_SYNC_PULSE + V_BP - 1)); // 523

   // 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_ff @(posedge vclock_in) begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

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

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

endmodule
```

backpointer_tracer.sv

```
module backpointer_tracer #(parameter BRAM_DELAY_CYCLES = 2, parameter
IMG_W = 320, parameter IMG_H = 240)(
    input clk,
    input rst,
    input start,
    input [16:0] start_pos,
    input [16:0] end_pos,
    input [1:0] bp,
    output logic [16:0] pixel_addr,
    output logic write_path,
    output done
    );

    parameter IDLE = 2'b00;
    parameter READING_BACKPOINTER = 2'b01;
    parameter NEXT_PIXEL = 2'b10;
    parameter DONE = 2'b11;
    logic [1:0] state;
    logic [2:0] cycles;
    logic [16:0] cur_pos;
    logic [16:0] next_pos;

    always_comb begin
        case(bp)
            2'b00: begin
                next_pos = {cur_pos[16:8] - 9'b1, cur_pos[7:0]};
            end
            2'b01: begin
                next_pos = {cur_pos[16:8], cur_pos[7:0] + 8'b1};
            end
            2'b10: begin
                next_pos = {cur_pos[16:8] + 9'b1, cur_pos[7:0]};
            end
            2'b11: begin
                next_pos = {cur_pos[16:8], cur_pos[7:0] - 8'b1};
            end
        endcase
    end

    integer maze_sol_f;

    assign done = state == DONE;
    always_ff @(posedge clk)begin
        if(rst)begin
            state <= IDLE;
        end
        else begin
            case(state)
                IDLE: begin
                    if(start)begin
                        state <= READING_BACKPOINTER;
                        write_path <= 1;
                        pixel_addr <= end_pos[7:0] * IMG_W + end_pos[16:8];
                        cur_pos <= end_pos;
                        cycles <= 0;
                    end
                end
                READING_BACKPOINTER: begin
                    write_path <= 0;
                    if(cycles == BRAM_DELAY_CYCLES - 1)begin
                        state <= NEXT_PIXEL;
                    end
                    else begin
                        cycles <= cycles + 1;
                    end
                end
                NEXT_PIXEL: begin
                    if(next_pos == start_pos)begin
                        state <= DONE;
                    end
                    else begin
                        cycles <= 0;
                        state <= READING_BACKPOINTER;
                    end
                    pixel_addr <= next_pos[7:0] * IMG_W + next_pos[16:8];
                    write_path <= 1;
                    cur_pos <= next_pos;
                end
                DONE: begin
```

```
                    write_path <= 0;
                    state <= IDLE;
                end
            endcase
        end
    end

endmodule
```

dijkstras_algorithm.sv

```
module dijkstras_algorithm #(parameter MAX_OUT_DEGREE = 4, parameter
MAX_NODES = 1024)
(
    input clk,
    input rst,
    input run,
    input n_start,

    //Set high when we can safely read bram output
    input bram_ready,
    //The minimum weight to get to node n
    input [15:0] min_weight_in,
    //Contains the edge weights and nodes of the neighbors of n
    input [63:0] adjacency,

    //Used to index into the bram rows
    output logic [9:0] n,

    //read is set high when reading from any of the brams
    output logic read,
    //below are used for writing backpointers and minimum weights
    output logic [9:0] backpointer,
    output logic [15:0] min_weight_out,
    //These are set high when writing to the appropriate bram
    output logic write_bp,
    output logic write_min_weight,
    output logic done,
    output logic [2:0] state
    );
    //Every iteration of the algorithm, we output the current node and the
minimum weight
    //to reach it. This minimum weight is written into the bram that stores
the minimum weight
    //to each node. At the same time, we also fetch the neighbors for that
node for the
    //next iteration

    parameter IDLE = 3'b000;
    parameter FETCH_NEIGHBORS = 3'b001;
    parameter CONSIDER_NEIGHBOR = 3'b010;
    parameter CHECK_IF_NEIGHBOR_IN_SPT = 3'b011;
    parameter WAIT_FOR_HEAP_READY = 3'b100;
    parameter ADD_TO_FRONTIER = 3'b101; //Checks if adjacents are already
included in SPT
    parameter SELECT_MIN_NODE = 3'b110;
    parameter DONE = 3'b111;
    //logic [2:0] state;

    reg spt_set [0:MAX_NODES - 1];
    reg [15:0] neighbors [0:MAX_OUT_DEGREE - 1];
    logic [2:0] i;

    logic frontier_ready;
    logic insert_into_frontier;
    logic retrieve_min;

    logic [35:0] k;
    logic [35:0] min_k;

    min_heap #(.MAX_NODES(MAX_NODES))frontier_heap(
        .clk(clk),
        .rst(rst),
        .insert(insert_into_frontier),
        .k(k),
        .retrieve_min(retrieve_min),
        .min_k(min_k),
        .ready(frontier_ready));

    logic [9:0] neighbor;
    logic [5:0] neighbor_weight;

    integer j;
    logic [9:0] parent;
    logic [15:0] parent_min_weight;
    assign done = state == DONE;

    always_ff @(posedge clk)begin
        if(rst)begin
            state <= IDLE;
            read <= 0;
            write_bp <= 0;
            write_min_weight <= 0;
            insert_into_frontier <= 0;
            retrieve_min <= 0;
        end
        else begin
            case(state)
                IDLE: begin
                    if(run)begin
                        i <= 3'b0;
                        state <= FETCH_NEIGHBORS;
                        //Initialize spt_set for dijkstra's algorithm
                        for (j = 0; j < MAX_NODES; j = j + 1) begin
                            spt_set[j] <= j == n_start;
                        end
                        n <= n_start;
                        parent <= n_start;
                        parent_min_weight <= 16'b0;
                        read <= 1;
                    end
                end
                FETCH_NEIGHBORS: begin
                    read <= 0;
                    write_bp <= 0;
                    write_min_weight <= 0;
                    if(bram_ready)begin
                        for (j = 0; j < MAX_OUT_DEGREE; j = j + 1) begin
                            neighbors[j] <= adjacency[(j << 4) +: 16];
                        end
                        state <= CONSIDER_NEIGHBOR;
                        i <= 3'b0;
                    end
                end
                CONSIDER_NEIGHBOR: begin
                    if(i == MAX_OUT_DEGREE)begin
                        retrieve_min <= 1;
                        state <= SELECT_MIN_NODE;
                    end
                    else begin
                        {neighbor, neighbor_weight} <= neighbors[i];
                        state <= CHECK_IF_NEIGHBOR_IN_SPT;
                    end
                end
                CHECK_IF_NEIGHBOR_IN_SPT: begin
                    if(spt_set[neighbor] == 0)begin
                        state <= WAIT_FOR_HEAP_READY;
                    end
                    else begin
                        state <= CONSIDER_NEIGHBOR;
                        i <= i + 1;
                    end
                end
                WAIT_FOR_HEAP_READY: begin
                    if(frontier_ready)begin
                        if(neighbor == 10'b0)begin
                            //We are finished inserting neighbors
                            retrieve_min <= 1;
                            state <= SELECT_MIN_NODE;
                        end
                        else begin
                            insert_into_frontier <= 1;
                            //Store parent so that we can have a
backpointer to the minimum weight path
                            k <= {neighbor_weight + parent_min_weight,
neighbor, parent};
                            i <= i + 1;
                            state <= ADD_TO_FRONTIER;
                        end
                    end
                end
                ADD_TO_FRONTIER: begin
                    insert_into_frontier <= 0;
                    if(frontier_ready)begin
                        state <= CONSIDER_NEIGHBOR;
                    end
                end
                SELECT_MIN_NODE: begin
                    retrieve_min <= 0;
                    {min_weight_out, n, backpointer} <= min_k;
                    //Store parent of new neighbors and the minimum weight
                    //of the path to this parent
                    parent <= min_k[19:10];
                    parent_min_weight <= min_k[25:20];
                    spt_set[n] <= 1;
                    read <= 1;
                    write_min_weight <= 1;
                    write_bp <= 1;
                    state <= FETCH_NEIGHBORS;
                end
                DONE: begin

                end
            endcase

        end
    end

endmodule
```

dilation.sv

```
module dilation #(K = 5, IMG_W=320, IMG_H=240)(
    input clk,
      input rst,
      input start,
      input pixel_in,

      output logic [16:0] pixel_addr,
    output logic pixel_valid,
      output logic processed_pixel,
      output logic done                                                //
flag for when erosion is done
    );

      logic [(K-1)*IMG_W + K - 1: 0] pixel_buffer;           // buffer of
relevant pixels
    logic [(K-1)*IMG_W + K - 1: 0] pixel_buffer_dilated;    // buffer of
relevant pixels after modification by dilation
    logic [(K-1)*IMG_W + K - 1: 0] temp;
      logic [17:0] pixel_counter;                                      // #
of pixels we have received since start
      logic [1:0] state;
      integer i;
      // used to build window
      integer j;
      // used to build window


      logic [8:0] oldest_pixel_x;                                 // x location of
the oldest pixel we have received
      logic [7:0] oldest_pixel_y;                                 // y location of
the oldest pixel we have received

      logic [(K-1)*IMG_W + K - 1: 0] dilation_mask;

      parameter IDLE = 2'b00;
      parameter DELAY = 2'b01;
      parameter DILATING = 2'b10;
      parameter DONE = 2'b11;
      assign pixel_addr = oldest_pixel_x + oldest_pixel_y * IMG_W;

      assign pixel_valid = (state == DILATING);
      assign done = (state == DONE);

      always_comb begin
            // this constructs the window from our pixel buffer
            dilation_mask = 0;
            for(i = 0; i < K; i = i + 1) begin
                  for(j = 0; j < K; j = j + 1) begin
                      dilation_mask[i*IMG_W+j] = pixel_buffer[((K-1)*IMG_W
+ K - 1) >> 1];
                      //dilation_mask[i*IMG_W+j] = pixel_buffer[2*IMG_W +
2];
                  end
            end

            temp = pixel_buffer_dilated | dilation_mask;
            processed_pixel = temp[(K-1)*IMG_W + K - 1];
      end



    `ifdef SIM
    integer bin_maze_dilated_f;
    `endif

    always_ff @(posedge clk)begin
        if(rst)begin
            state <= IDLE;
                  pixel_counter <= 18'b0;
                  pixel_buffer <= 0;
                  pixel_buffer_dilated <= 0;
        end
        else begin
                  pixel_buffer <= {pixel_buffer[(K-1)*IMG_W + K - 2:0],
pixel_in};
                  pixel_buffer_dilated <= {temp[(K-1)*IMG_W + K - 2:0],
pixel_in};
            case(state)
                IDLE: begin
                    if(start)begin
```

```
                        state <= DELAY;
                        pixel_counter <= 18'b1;
                    end
                end
                DELAY: begin
                    pixel_counter <= pixel_counter + 1;
                    if(pixel_counter >= ((K-1)*IMG_W + K - 1))begin
                        state <= DILATING;
                        oldest_pixel_x <= 9'b0;
                        oldest_pixel_y <= 8'b0;
                        `ifdef SIM
                        bin_maze_dilated_f =
$fopen("C:/Users/giand/Documents/MIT/Senior_Fall/6.111/gims-labyrinth/gims_
labyrinth/python_stuff/verilog_testing/bin_maze_dilated_img.txt","w");
                        `endif
                    end
                end
                DILATING: begin
                    `ifdef SIM
                    $fwrite(bin_maze_dilated_f,"%b\n",processed_pixel);
                    `endif

                    pixel_counter <= pixel_counter + 1;
                            if(oldest_pixel_x == (IMG_W - 1) &&
oldest_pixel_y == (IMG_H - 1))begin
                                state <= DONE;
                            end

                            if(oldest_pixel_x == (IMG_W - 1)) begin
                                  oldest_pixel_x <= 9'b0;
                                  oldest_pixel_y <= oldest_pixel_y + 1;
                            end
                            else begin
                                  oldest_pixel_x <= oldest_pixel_x + 1;
                          end
                end
                DONE: begin
                    `ifdef SIM
                    $fclose(bin_maze_dilated_f);
                    `endif
                    state <= IDLE;
                end
```

```
            endcase
        end
    end

endmodule
```

divider.sv

```
`timescale 1ns / 1ps
// The divider module divides one number by another. It
// produces a signal named "ready" when the quotient output
// is ready, and takes a signal named "start" to indicate
// the input dividend and divider is ready.
// sign -- 0 for unsigned, 1 for twos complement

// It uses a simple restoring divide algorithm.
// http://en.wikipedia.org/wiki/Division_(digital)#Restoring_division
//
// Author Logan Williams, updated 11/25/2018 gph

module divider #(parameter WIDTH = 8)
  (input clk, sign, start,
   input [WIDTH-1:0] dividend,
   input [WIDTH-1:0] divider,
   output reg [WIDTH-1:0] quotient,
   output [WIDTH-1:0] remainder,
   output ready);

   reg [WIDTH-1:0]  quotient_temp;
   reg [WIDTH*2-1:0] dividend_copy, divider_copy, diff;
   reg negative_output;

   assign remainder = (!negative_output) ?
             dividend_copy[WIDTH-1:0] : ~dividend_copy[WIDTH-1:0] + 1'b1;

   reg [5:0] bit_n = 0;
   reg del_ready = 1;
   assign ready = (bit_n==0) & ~del_ready;

   wire [WIDTH-2:0] zeros = 0;
   initial bit_n = 0;
   initial negative_output = 0;
   always @( posedge clk ) begin
      del_ready <= (bit_n==0);
      if( start ) begin

         bit_n = WIDTH;
```

```
          quotient = 0;
          quotient_temp = 0;
          dividend_copy = (!sign || !dividend[WIDTH-1]) ?
                          {1'b0,zeros,dividend} :
                          {1'b0,zeros,~dividend + 1'b1};
          divider_copy = (!sign || !divider[WIDTH-1]) ?
                    {1'b0,divider,zeros} :
                    {1'b0,~divider + 1'b1,zeros};

          negative_output = sign &&
                            ((divider[WIDTH-1] && !dividend[WIDTH-1])
                             ||(!divider[WIDTH-1] && dividend[WIDTH-1]));
        end
       else if ( bit_n > 0 ) begin
          diff = dividend_copy - divider_copy;
          quotient_temp = quotient_temp << 1;
          if( !diff[WIDTH*2-1] ) begin
             dividend_copy = diff;
             quotient_temp[0] = 1'd1;
          end
          quotient = (!negative_output) ?
                     quotient_temp :
                     ~quotient_temp + 1'b1;
          divider_copy = divider_copy >> 1;
          bit_n = bit_n - 1'b1;
       end
    end
endmodule
```

erosion.sv

```
module erosion #(K = 5, IMG_W=320, IMG_H=240)(
    input clk,
      input rst,
      input start,
      input pixel_in,
      output pixel_valid,
      output logic processed_pixel,
      output logic done       // flag for when erosion is done

    );

      logic [K*K-1:0] window; // window into original image for kernel
      logic [(K-1)*IMG_W + K - 1: 0] pixel_buffer;    // buffer of relevant
pixels
      logic [17:0] pixel_counter; // # of pixels received since start
      logic [1:0] state;
      logic in_bounds;            // kernel is in bounds
      integer i;
      // used to build window
      integer j;
      // used to build window

      logic [8:0] center_x; // x location of the center pixel of the window
      logic [7:0] center_y; // y location of the center pixel of the window
      integer half_k = K >> 1; // half width of kernel
      integer center_idx = (K*K) >> 1;    // index into window of center
pixel of the window

      parameter IDLE = 2'b00;
      parameter DELAY = 2'b01;
      parameter ERODING = 2'b10;
      parameter DONE = 2'b11;

      //assign pixel_valid = pixel_counter >= (((K-1)*IMG_W + K - 1) >> 1);
      assign pixel_valid = state == ERODING;
      always_comb begin
            done = (state == DONE);
            // this constructs the window from our pixel buffer
            for(i = 0; i < K; i = i + 1) begin
```

```
                    for(j = 0; j < K; j = j + 1) begin
                          window[i*K + j] = pixel_buffer[i*IMG_W+j];
                    end
              end
              in_bounds = center_x >= half_k && center_x <= (IMG_W - 1 -
half_k) && center_y >= half_k && center_y <= (IMG_H - 1 - half_k);

              processed_pixel = in_bounds ? &window : 1'b0;   // change to
window[center_idx] if we want to pass through
        end

    `ifdef SIM
    integer bin_maze_eroded_f;
    `endif

    always_ff @(posedge clk)begin
        if(rst)begin
            state <= IDLE;
                  pixel_counter <= 18'b0;
                  pixel_buffer <= 0;
        end
        else begin
                  pixel_buffer <= {pixel_buffer[(K-1)*IMG_W + K - 2:0],
pixel_in};
            case(state)
                IDLE: begin
                    if(start)begin
                        state <= DELAY;
                        pixel_counter <= 18'b1;
                    end
                end
                DELAY: begin
                    pixel_counter <= pixel_counter + 1;
                    if(pixel_counter >= (((K-1)*IMG_W + K - 1) >> 1))begin
                        state <= ERODING;
                        center_x <= 9'b0;
                        center_y <= 8'b0;
                        `ifdef SIM
                        bin_maze_eroded_f =
$fopen("C:/Users/giand/Documents/MIT/Senior_Fall/6.111/gims-labyrinth/gims_
labyrinth/python_stuff/verilog_testing/bin_maze_eroded_img.txt","w");
                        `endif
                    end
                end
                ERODING: begin
                    `ifdef SIM
                    $fwrite(bin_maze_eroded_f,"%b\n",processed_pixel);
                    `endif

                    pixel_counter <= pixel_counter + 1;
                              state <= (center_x == (IMG_W - 1) && center_y
== (IMG_H - 1)) ? DONE : ERODING;
                              if(center_x == (IMG_W - 1)) begin
                                    center_x <= 9'b0;
                                    center_y <= center_y + 1;
                              end else
                                    center_x <= center_x + 1;
                end
                DONE: begin
                    state <= IDLE;
                    `ifdef SIM
                    $fclose(bin_maze_eroded_f);
                    `endif
                end
            endcase
        end
    end

endmodule
```

graph_manager.sv

```
module graph_manager(
    input clk,
    input rst,
    input [9:0] n,
    input write_adj,
    input write_bp,
    input write_min_weight,
    input write_pos,
    input read,
    //writing
    input [63:0] adjacency_in,
    input [15:0] weight_in,
    input [9:0] backpointer_in,
    input [18:0] xy_in,
    //reading
    output [63:0] adjacency_out,
    output [15:0] weight_out,
    output [9:0] backpointer_out,
    output [18:0] xy_out,
    output ready
    );
    parameter DO_NOTHING = 3'b000;
    parameter SET_ADJACENCY = 3'b001;
    parameter SET_BACKPOINTER = 3'b010;
    parameter SET_MIN_WEIGHT = 3'b011;
    parameter SET_NODE_POS = 3'b100;
    parameter READ = 3'b101;

    parameter READY = 2'b00;
    parameter FIRST_CYCLE = 2'b01;
    parameter SECOND_CYCLE = 2'b10;
    logic [1:0] state;

    //All these brams have as many rows as the maximum number of nodes in
the graph

    //10 bits for adjacent node
    //6 bits for weight
    //repeat above 4 times for each of the four possible neighbors
    adjacency_list adj_list(.clka(clk),
                            .addra(n),
                            .dina(adjacency_in),
                            .douta(adjacency_out),
                            .wea(write_adj)
                            );

    //10 bits for x
    //9 bits for y
    node_list n_list(.clka(clk),
                     .addra(n),
                     .dina(xy_in),
                     .douta(xy_out),
                     .wea(write_pos)
                     );

    //10 bits for pointer to previous node for minimum weight path
    spt_backpointers spt_bp(.clka(clk),
                            .addra(n),
                            .dina(backpointer_in),
                            .douta(backpointer_out),
                            .wea(write_bp));

    //16 bits for minimum weight path
    spt_min_weights spt_min_w(.clka(clk),
                              .addra(n),
                              .dina(weight_in),
                              .douta(weight_out),
                              .wea(write_min_weight)
                              );

    assign ready = (state == SECOND_CYCLE);
    logic do_something;
    assign do_something = write_adj || write_bp || write_min_weight ||
write_pos || read;

    always_ff @(posedge clk)begin
        if(rst)begin
            state <= READY;
        end
        else begin
            case(state)
```

```
                READY: begin
                    if(do_something)begin
                        state <= FIRST_CYCLE;
                    end
                end
                FIRST_CYCLE: begin
                    state <= SECOND_CYCLE;
                end
                SECOND_CYCLE: begin
                    state <= READY;
                end
                default: begin
                    //do nothing
                end
            endcase
        end
    end

endmodule
```

lees_algorithm.sv

```
module lees_algorithm #(parameter MAX_OUT_DEGREE = 4, parameter
BRAM_DELAY_CYCLES = 2, parameter IMG_W = 320, parameter IMG_H = 240)
(
    input clk,
    input rst,
    input start,
    input [16:0] start_pos,
    input [16:0] end_pos,
    input skel_pixel,

    //Used to index into the maze
    output logic [16:0] pixel_r_addr,
    output logic [16:0] pixel_wr_addr,

    //below are used for writing backpointers
    output logic [1:0] backpointer,
    //These are set high when writing to the appropriate bram
    output logic bp_we,
    output logic done,
    output logic success,

    output logic [2:0] state
    );

    parameter QUEUE_SIZE = 64;

    parameter IDLE = 3'b000;
    parameter FETCH_NEIGHBORS = 3'b001;
    parameter VALIDATE_NEIGHBORS = 3'b010;
    parameter DELAY = 3'b011;
    parameter CLEAR_VISITED_MAP = 3'b100;
    parameter DONE = 3'b101;

    logic [5:0] q_idx;
    reg [16:0] queue[0:QUEUE_SIZE - 1];
    integer k;

    reg [16:0] neighbors[0:MAX_OUT_DEGREE - 1];
    reg [1:0] backpointers[0:MAX_OUT_DEGREE - 1];
    logic [2:0] i;
    logic [2:0] j;
    assign done = state == DONE;

    logic visited_we;
    logic visit_val;
    logic visited;
    visited_map visited_map(.clka(clk),
                            .addra(pixel_wr_addr),
                            .dina(visit_val),
                            .wea(visited_we),
                            .clkb(clk),
                            .addrb(pixel_r_addr),
                            .doutb(visited));


    logic neighbor_within_bounds;

    always_comb begin
        neighbor_within_bounds = neighbors[j][16:8] > 2 &&
neighbors[j][16:8] < IMG_W - 3 && neighbors[j][7:0] > 2 &&
neighbors[j][7:0] < IMG_H - 3;
    end

    integer bp_f;
    always_ff @(posedge clk)begin
        if(rst)begin
            state <= IDLE;
        end
        else begin
            case(state)
                IDLE: begin
                    if(start)begin
                        state <= FETCH_NEIGHBORS;
                        queue[0] <= start_pos;
                        q_idx <= 1;
                        visited_we <= 1;
                        visit_val <= 1;
                        pixel_wr_addr <= start_pos[7:0] * IMG_W +
start_pos[16:8];
                        success <= 0;
                    end
                end
                FETCH_NEIGHBORS: begin
                    visited_we <= 0;
                    if(q_idx == 0)begin
                        //End was not found
                        state <= DELAY;
                        success <= 0;
                        visited_we <= 1;
                    end
                    else begin
                        //right
                        neighbors[0] <= {queue[0][16:8] + 9'b1, 
queue[0][7:0]};

                        backpointers[0] <= 2'b00;
                        //up
                        neighbors[1] <= {queue[0][16:8], queue[0][7:0] - 
8'b1};

                        backpointers[1] <= 2'b01;
                        //left
                        neighbors[2] <= {queue[0][16:8] - 9'b1, 
queue[0][7:0]};

                        backpointers[2] <= 2'b10;
                        //down
                        neighbors[3] <= {queue[0][16:8], queue[0][7:0] + 
8'b1};

                        backpointers[3] <= 2'b11;

                        state <= VALIDATE_NEIGHBORS;

                        for (k = 0; k < QUEUE_SIZE - 1; k = k + 1) begin
                            queue[k] <= queue[k + 1];
                        end
                        q_idx <= q_idx - 1;

                        i <= 0;
                        j <= 0;
                    end

                end
                VALIDATE_NEIGHBORS: begin
                    if(j == MAX_OUT_DEGREE)begin
                        bp_we <= 0;
                        visited_we <= 0;
                        state <= FETCH_NEIGHBORS;
                    end
                    else if(i == j + BRAM_DELAY_CYCLES + 1)begin
                        j <= j + 1;
                        if(neighbors[j] == end_pos)begin
                            pixel_wr_addr <= neighbors[j][7:0] * IMG_W +
neighbors[j][16:8];
                            bp_we <= 1;
                            backpointer <= backpointers[j];

                            state <= DELAY;
                            success <= 1;
                            visited_we <= 1;
                        end
                        else if(skel_pixel == 1 && !visited &&
neighbor_within_bounds)begin
                            queue[q_idx] <= neighbors[j];
                            q_idx <= q_idx + 1;

                            pixel_wr_addr <= neighbors[j][7:0] * IMG_W +
neighbors[j][16:8];

                            bp_we <= 1;
                            backpointer <= backpointers[j];
                            visited_we <= 1;
                        end
                        else begin
                            bp_we <= 0;
                            visited_we <= 0;
                        end

                    end
                    if(i < MAX_OUT_DEGREE)begin
                        pixel_r_addr <= neighbors[i][7:0] * IMG_W +
neighbors[i][16:8];
                    end
                    i <= i + 1;
                end

                DELAY: begin
                    pixel_wr_addr <= 0;
                    bp_we <= 0;
                    visit_val <= 0;
                    state <= CLEAR_VISITED_MAP;
                end

                CLEAR_VISITED_MAP: begin
                    if(pixel_wr_addr == IMG_W * IMG_H)begin
                        state <= DONE;
                        visited_we <= 0;
                    end
                    else begin
                        pixel_wr_addr <= pixel_wr_addr + 1;
                    end
                end

                DONE: begin
                    state <= IDLE;
                    success <= 0;
                end

                default: begin
                    //Do nothing
                end

            endcase
        end
    end
endmodule
```

median_filt.sv

```
module median_filt #(parameter KERNEL_SIZE = 3, parameter N = 9)
    (
        input clk,
        input rst,
        input start,
        input [1:0] ar [0:N - 1],
        output logic [4:0] median,
        output ready,
        output logic [1:0] state
    );

    parameter READY = 2'b00;
    parameter COUNT_SORT = 2'b01;
    parameter FIND_MEDIAN = 2'b10;
    parameter DONE = 2'b11;

    //logic [1:0] state;

    logic [4:0] hashmap [0:3];
    logic [4:0] cumsum [0:3];
    logic [4:0] i;
    assign ready = state == READY;

    always_ff @(posedge clk)begin
        if(rst)begin
            i <= 0;
            state <= READY;
            hashmap <= {0, 0, 0, 0};
        end
        else begin
            case(state)
                READY: begin
                    if(start)begin
                        state <= COUNT_SORT;
                        i <= 0;
                        hashmap <= {0, 0, 0, 0};
                    end
                end
                COUNT_SORT: begin
```

```
                    if(i < N)begin
                        hashmap[ar[i]] <= hashmap[ar[i]] + 1;
                        i <= i + 1;
                    end
                    else begin
                        state <= FIND_MEDIAN;
                        i <= 1;
                        cumsum[0] <= hashmap[0];
                    end
                end
                FIND_MEDIAN: begin
                    if(cumsum[i - 1] > (N >> 1))begin
                        median <= i - 1;
                        state <= READY;
                    end
                    else begin
                        cumsum[i] <= cumsum[i - 1] + hashmap[i];
                        i <= i + 1;
                    end

                end
                default: begin
                    //do nothing
                end
            endcase
        end
     end
endmodule
```

min_heap.sv

```
module min_heap #(parameter MAX_HEAP_SIZE = 50,
                  parameter MAX_NODES)
    (
    input clk,
    input rst,
    input insert,
    input [35:0] k,
    input retrieve_min,
    output logic [35:0] min_k,
    output ready,
    output logic [1:0] state
    );
    parameter READY = 2'b00;
    parameter MIN_HEAPIFY_UP = 2'b01;
    parameter MIN_HEAPIFY_DOWN = 2'b10;
    //logic [1:0] state;

    logic [5:0] parent_i;
    logic [5:0] left_i;
    logic [5:0] right_i;
    logic [5:0] i;

    logic [5:0] heap_idx;

    assign ready = (state == READY);
    reg [35:0] heap [0:MAX_HEAP_SIZE - 1];
    reg [5:0] heap_size;
    reg [9:0] heap_idx_map [0:MAX_NODES - 1];
    integer j;

    always_comb begin
        parent_i = (i - 1) >> 1;
        left_i = (i << 1) + 1;
        right_i = (i << 1) + 2;

        heap_idx = heap_idx_map[k[19:10]];
    end

    assign min_k = heap[0];
```

```
always_ff @(posedge clk)begin
    if(rst)begin
        heap_size <= 0;
        for (j = 0; j < MAX_NODES; j = j + 1) begin
            heap_idx_map[j] <= MAX_HEAP_SIZE;
        end
        state <= READY;
    end
    else begin
        case(state)
            READY: begin
                if(insert)begin
                    if(heap_idx == MAX_HEAP_SIZE)begin
                        heap[heap_size] <= k;
                        //update heap index map
                        heap_idx_map[k[19:10]] <= heap_size;

                        heap_size <= heap_size + 1;
                        i <= heap_size;
                        state <= MIN_HEAPIFY_UP;
                    end
                    else begin
                        i <= heap_idx;
                        if(k < heap[heap_idx])begin
                            heap[heap_idx] <= k;
                            state <= MIN_HEAPIFY_DOWN;
                        end
                    end

                end
                if(retrieve_min)begin
                    if(heap_size == 1)begin
                        //min_k <= heap[0];
                        heap_size <= heap_size - 1;
                        //update heap index map
                        heap_idx_map[heap[0][19:10]] <= MAX_HEAP_SIZE;
                    end
                    else begin
                        //min_k <= heap[0];
                        heap[0] <= heap[heap_size - 1];
                        heap_size <= heap_size - 1;
```

```
                            state <= MIN_HEAPIFY_DOWN;
                            i <= 0;
                            //update heap index map
                            heap_idx_map[heap[0][19:10]] <= MAX_HEAP_SIZE;
                            heap_idx_map[heap[heap_size - 1][19:10]] <= 0;
                        end



                    end
                end

                MIN_HEAPIFY_UP: begin
                    if(i > 0 && heap[parent_i] > heap[i])begin
                        {heap[i], heap[parent_i]} <= {heap[parent_i],
heap[i]};
                        //update heap index map
                        heap_idx_map[heap[parent_i][19:10]] <= i;
                        heap_idx_map[heap[i][19:10]] <= parent_i;
                        i <= parent_i;
                    end
                    else begin
                        state <= READY;
                    end

                end

                MIN_HEAPIFY_DOWN: begin
                    if(left_i < heap_size && right_i < heap_size &&
heap[right_i] < heap[i] && heap[right_i] < heap[left_i])begin
                        {heap[i], heap[right_i]} <= {heap[right_i],
heap[i]};
                        //update heap index map
                        heap_idx_map[heap[right_i][19:10]] <= i;
                        heap_idx_map[heap[i][19:10]] <= right_i;
                        i <= right_i;
                    end
                    else if(left_i < heap_size && heap[left_i] <
heap[i])begin
                        {heap[i], heap[left_i]} <= {heap[left_i], heap[i]};
                        //update heap index map
                        heap_idx_map[heap[left_i][19:10]] <= i;
                        heap_idx_map[heap[i][19:10]] <= left_i;
                        i <= left_i;
                    end
                    else begin
                        state <= READY;
                    end
                end
                default: begin
                    //do nothing
                end
            endcase

        end
    end

endmodule
```

path_car_drawer.sv

```
module path_car_drawer(
   input clk_100mhz,
   input[15:0] sw,
   input btnc, btnu, btnl, btnr, btnd,
   output[3:0] vga_r,
   output[3:0] vga_b,
   output[3:0] vga_g,
   output vga_hs,
   output vga_vs,
   output led16_b, led16_g, led16_r,
   output led17_b, led17_g, led17_r,
   output[15:0] led,
   output ca, cb, cc, cd, ce, cf, cg, dp,  // segments a-g, dp
   output[7:0] an    // Display location 0-7
   );

    logic clk_25mhz;
    logic blank;

    // create 65mhz system clock, happens to match 1024 x 768 XVGA timing
    clk_wiz_vga clkdivider(.clk_in1(clk_100mhz), .clk_out1(clk_25mhz));

    logic [31:0] data;      //  instantiate 7-segment display; display (8)
4-bit hex
    logic [6:0] segments;
    assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];
    assign  dp = 1'b1;  // turn off the period

    assign led = sw;                        // turn leds on
    //assign data = {28'h0123456, sw[3:0]};   // display 0123456 + sw[3:0]
    assign led16_r = btnl;                  // left button -> red led
    assign led16_g = btnc;                  // center button -> green led
    assign led16_b = btnr;                  // right button -> blue led
    assign led17_r = btnl;
    assign led17_g = btnc;
    assign led17_b = btnr;

    logic [9:0] hcount;     // pixel on current line
    logic [9:0] vcount;     // line number
    logic hsync, vsync;
    logic [11:0] pixel, pixel_path;
    logic [11:0] rgb;
    vga vga1(.vclock_in(clk_25mhz),.hcount_out(hcount),.vcount_out(vcount),
          .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));

    // btnc button is user reset
    logic reset;
    debounce
db1(.reset_in(0),.clock_in(clk_25mhz),.noisy_in(btnc),.clean_out(reset));

    logic find_path, move_car;
    debounce db2(.reset_in(reset),.clock_in(clk_25mhz),.noisy_in(btnu),
            .clean_out(find_path));
    debounce db3(.reset_in(reset),.clock_in(clk_25mhz),.noisy_in(btnd),
            .clean_out(move_car));

    // BRAMs operating mode No_Change
    parameter BRAM_SIZE = 76800;
    logic my_wea0, my_wea1;
    logic [16:0] addr0, my_addr0, addr1;
    logic [3:0] data_to_bram0, data_from_bram0;
    logic data_to_bram1, data_from_bram1;
    logic BRAM0_initialized, BRAM1_initialized;

    blk_mem_gen_0 bram0(.clka(clk_25mhz), .wea(my_wea0), .addra(addr0),
                .dina(data_to_bram0), .douta(data_from_bram0));
    blk_mem_gen_1 bram1(.clka(clk_25mhz), .wea(my_wea1), .addra(addr1),
                .dina(data_to_bram1), .douta(data_from_bram1));

    logic phsync,pvsync,pblank;
    logic b,hs,vs;
    logic [9:0] hcount1, vcount1, hcount2, vcount2, hcount3, vcount3;
    logic hsync1, vsync1, blank1, hsync2, vsync2, blank2;

    logic [9:0] car_x, car_y;
    parameter car_x_init = 0;   // starting address provided by maze alg
    parameter car_y_init = 9;   // these two parameters are TO BE DELETED
    path pa(.vclock_in(clk_25mhz),.reset_in(reset),
.data_from_bram1(data_from_bram1),
                .hsync_in(hsync2),.vsync_in(vsync2),.blank_in(blank2),
.phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),
                .pixel_out(pixel_path));

    logic border = (hcount==0 | hcount==639 | vcount==0 | vcount==479 |
                   hcount == 320 | vcount == 240);

    logic addr0_inc, addr1_inc;
    always_ff @(posedge clk_25mhz) begin
        if (reset) begin  //initialize BRAM path
            BRAM0_initialized <= 0;
            my_wea0 <= 1;
            addr0 <= 0;
            BRAM1_initialized <= 0;
            my_wea1 <= 1;
            addr1 <= 0;
            addr0_inc <= 0;
            addr1_inc <= 0;
        end else if (!BRAM0_initialized) begin
            if (addr0 < BRAM_SIZE) begin
                if (!addr0_inc) begin
                    addr0_inc <= 1;
                    if ((addr0 >= 2880) && (addr0 < 3100))
                        data_to_bram0 <=  4'b0001;
                    else if (addr0==3100) data_to_bram0<=4'b0010;
                    else if ((addr0 == 3420) || (addr0 == 3740) || (addr0
== 4060) || (addr0 == 4380)
                        || (addr0 == 4700) || (addr0 == 5020) || (addr0 ==
5340) || (addr0 == 5660))
                        data_to_bram0 <=  4'b0010;
                    else if ((addr0 == 5980) || (addr0 == 6300) || (addr0
== 6620) || (addr0 == 6940)
                        || (addr0 == 7260) || (addr0 == 7580) || (addr0 ==
7900) || (addr0 == 8220))
                        data_to_bram0 <=  4'b0010;
                    else if ((addr0 == 8540) || (addr0 == 8860) || (addr0
== 9180) || (addr0 == 9500)
                        || (addr0 == 9820) || (addr0 == 10140) || (addr0 ==
10460) )
                        data_to_bram0 <=  4'b0010;
                    else if ((addr0 >= 10561) && (addr0 < 10781))
                        data_to_bram0 <=  4'b0100;
                    else  data_to_bram0 <=  4'b0;
                end else begin
                    addr0_inc <= 0;
                    addr0 <= addr0 + 1;
                end
            end else begin
                BRAM0_initialized <= 1;
                addr0 <= 0;
                my_wea0 <= 0;
            end

        // find path
        end else if (!BRAM1_initialized) begin
            if (addr1 < BRAM_SIZE) begin
                if (!addr1_inc) begin
                    data_to_bram1 <= (data_from_bram0 == 4'b0) ? 1'b0 :
1'b1;
                    addr1_inc <= 1;
                end else begin
                    addr1_inc <= 0;
                    addr0 <= addr0 + 1;
                    addr1 <= addr1 + 1;
                end
            end else begin
                BRAM1_initialized <= 1;
                addr0 <= 0;
                addr1 <= 0;
                my_wea1 <= 0;
            end
        //read from BRAM1 for solved path
        end else begin
            addr0 <= my_addr0;
            addr1 <= (hcount>>1)+ ((vcount>>1)*32'd320);
        end

        //1 step delay of addr1 calculation + data out from bram of another
step delay
        hcount3 <= hcount2;
        vcount3 <= vcount2;
        hsync2 <= hsync1;
        vsync2 <= vsync1;
        blank2 <= blank1;
        hcount2 <= hcount1;
        vcount2 <= vcount1;
        hsync2 <= hsync;
        vsync1 <= vsync;
        blank1 <= blank;
        hcount1 <= hcount;
        vcount1 <= vcount;

        if (sw[1:0] == 2'b01) begin
            // 1 pixel outline of visible area (white)
            hs <= hsync;
            vs <= vsync;
            b <= blank;
            rgb <= {12{border}};
        end else if (sw[1:0] == 2'b10) begin
            // color bars
            hs <= hsync;
            vs <= vsync;
            b <= blank;
            rgb <= {{4{hcount[7]}}, {4{hcount[6]}}, {4{hcount[5]}}};
        end else begin
            // default: path_car
            hs <= phsync;
            vs <= pvsync;
            b <= pblank;
            rgb <= pixel;
        end
    end

    parameter EAST  = 4'b0001;
    parameter SOUTH = 4'b0010;
    parameter WEST  = 4'b0100;
    parameter NORTH = 4'b1000;
    always_ff @(negedge vsync) begin
        if(!BRAM1_initialized) begin
            car_x <= car_x_init;    // car_x_init and car_y_init are from
maze algorithm
            car_y <= car_y_init;    // not from defined parameters (theh
two parameters to be deleted
        end else begin
            if (data_from_bram0 == EAST) car_x <= (addr0 % 320) +1;
            else if (data_from_bram0 == SOUTH) car_y <= (addr0 / 320)+1;
            else if (data_from_bram0 == WEST) car_x <= (addr0 % 320)-1;
            else if (data_from_bram0 == NORTH) car_y <= (addr0 / 320)-1;
        end
    end

    assign my_addr0 = car_x + (car_y*32'd320);

    parameter WIDTH = 4;
    parameter HEIGHT = 4;
    parameter COLOR = 12'h0F0;
    logic [9:0] car_x_min, car_y_min, car_x_max, car_y_max, x3, y3;
    logic [11:0] pixel_car;
    always_ff @(posedge clk_25mhz) begin
        if(!BRAM1_initialized) begin
            car_x_min <= 0;
            car_y_min <= 0;
            car_x_max <= 0;
            car_y_max <= 0;
            x3 <= (hcount2>>1);
            y3 <= (vcount2>>1);
        end else begin
            car_x_min <= (car_x < WIDTH)? 0 : (car_x-WIDTH);
            car_y_min <= (car_y < HEIGHT)? 0 : (car_y-HEIGHT);
            car_x_max <= car_x+WIDTH;
            car_y_max <= car_y+HEIGHT;

            x3 <= (hcount3>>1);
            y3 <= (vcount3>>1);
            if ((x3 >= car_x_min && x3 <= car_x_max) &&
                (y3 >= car_y_min && y3 <= car_y_max))
                pixel_car <= COLOR;
            else pixel_car <= 12'b0;
            end
    end
    assign pixel = pixel_path | pixel_car;

    // the following lines are required for the Nexys4 VGA circuit - do not
change
    assign vga_r = ~b ? rgb[11:8]: 0;
    assign vga_g = ~b ? rgb[7:4] : 0;
    assign vga_b = ~b ? rgb[3:0] : 0;

    assign vga_hs = ~hs;
    assign vga_vs = ~vs;

endmodule


///////////////////////////////////////////////////////////////////////////
/////
//
// path: draw path
//
///////////////////////////////////////////////////////////////////////////
/////
module path (
    input vclock_in,        // 25MHz clock
    input reset_in,         // 1 to initialize module
    input data_from_bram1,
    input hsync_in,         // VGA horizontal sync signal (active low)
    input vsync_in,         // VGA vertical sync signal (active low)
    input blank_in,         // VGA blanking (1 means output black pixel)
    output phsync_out,      // path_car's horizontal sync
    output pvsync_out,      // path_car's vertical sync
    output pblank_out,      // path_car's blanking
    output [11:0] pixel_out // path_car's pixel  // r=23:16, g=15:8, b=7:0
    );

    parameter WHITE = 12'hFFF;
    parameter GREEN = 12'h0F0;

    logic [11:0] pixel_car, pixel_path;

    assign pixel_path = data_from_bram1? WHITE : 12'b0;

    assign phsync_out = hsync_in;
    assign pvsync_out = vsync_in;
    assign pblank_out = blank_in;

    assign pixel_out = pixel_path;
endmodule
```

rgb_2_hsv.sv

```
module rgb_2_hsv(
    input clk,
    input rst,
    input start,
    input [7:0] r_in,
    input [7:0] g_in,
    input [7:0] b_in,
    output logic [16:0] h,   //Q9.8
    output logic [7:0] s,  //Q8
    output logic [7:0] v,  //Q8
    output ready,
    output logic [1:0] state
    );

    parameter DIVIDEND_WIDTH = 16;

    parameter IDLE = 2'b00;
    parameter DIV = 2'b01;
    parameter DONE = 2'b10;
    //logic [1:0] state;
    logic [7:0] r;
    logic [7:0] g;
    logic [7:0] b;

    logic div_start;
    logic div_ready;

    reg [7:0] min, max, delta;

    logic [15:0] h_dividend;
    logic [15:0] h_quotient;

    logic [15:0] s_dividend;
    logic [15:0] s_quotient;

    logic [15:0] v_dividend;
    logic [15:0] v_quotient;

    logic [31:0] h_prod;
```

```
assign ready = div_ready;

divider #(.WIDTH(DIVIDEND_WIDTH)) h_div1 (
.clk(clk),
.sign(0),
.start(div_start),
.dividend(h_dividend),
.divider({8'b0, delta}),
.quotient(h_quotient),
.ready(div_ready)
);

divider #(.WIDTH(DIVIDEND_WIDTH)) s_div1(
.clk(clk),
.sign(0),
.start(div_start),
.dividend({delta, 8'b0}),
.divider({8'b0, max}),
.quotient(s_quotient)
);

divider #(.WIDTH(DIVIDEND_WIDTH)) v_div1(
.clk(clk),
.sign(0),
.start(div_start),
.dividend({max, 8'b0}),
.divider(16'd255),
.quotient(v_quotient)
);

always_comb begin
    delta = max - min;
    h_prod = 16'h3c00 * h_quotient;

    if(delta == 0)begin
        h_dividend = 0;
        h = 0;
    end
    else if(r == max)begin
        if(g >= b)begin
```

```
                h_dividend = {g - b, 8'b0};
                h = h_prod[23:8];
            end
            else begin
                h_dividend = {b - g, 8'b0};
                h = 17'h16800 - h_prod[23:8];
            end
        end
        else if (g == max)begin
            if(b >= r)begin
                h_dividend = {b - r, 8'b0};
                h = 17'h07800 + h_prod[23:8];
            end
            else begin
                h_dividend = {r - b, 8'b0};
                h = 17'h07800 - h_prod[23:8];
            end
        end
        else if(b == max)begin
            if(r >= g)begin
                h_dividend = {r - g, 8'b0};
                h = 17'h0F000 + h_prod[23:8];
            end
            else begin
                h_dividend = {g - r, 8'b0};
                h = 17'h0F000 - h_prod[23:8];
            end
        end

        if(delta == max)begin
            s = 8'hff;
        end
        else if(max == 0)begin
            s = 8'h00;
        end
        else begin
            s = s_quotient;
        end

        if(max == 8'hff)begin
            v = 8'hff;
        end
        else begin
            v = v_quotient;
        end

    end


    always_ff @(posedge clk) begin
        if(rst)begin
            state <= IDLE;
        end
        else begin
            case(state)
                IDLE: begin
                    if(start)begin
                        if((r_in >= g_in) && (r_in>= b_in))begin
                            max <= r_in;
                        end
                        else if((g_in >= r_in) && (g_in >= b_in))begin
                            max <= g_in;
                        end
                        else begin
                            max <= b_in;
                        end

                        if((r_in <= g_in) && (r_in <= b_in))begin
                            min <= r_in;
                        end
                        else if((g_in <= r_in) && (g_in <= b_in))begin
                            min <= g_in;
                        end
                        else begin
                            min <= b_in;
                        end

                        r <= r_in;
                        g <= g_in;
                        b <= b_in;
                        div_start <= 1;
                        state <= DIV;
                    end
                end
                DIV: begin
                    div_start <= 0;
                    if(div_ready)begin
                        state <= IDLE;
                    end
                end
                default: begin
                    //Do nothing
                end
            endcase
        end
    end
endmodule
```

signal_processing.sv

```
module binary_maze_filtering #(parameter IMG_W = 320, parameter IMG_H =
240, parameter BRAM_READ_DELAY = 2)
    (
        input clk,
        input start,
        input rst,
        input [11:0] rgb_pixel,

        output logic [16:0] cam_pixel_r_addr,

        output done,
        //Used for writing filtered pixel values into bram
        output logic [16:0] pixel_wr_addr,
        output pixel_wea,
        output pixel_out,

        output logic start_color,
        output logic end_color
    );
    logic [7:0] r;
    logic [7:0] g;
    logic [7:0] b;
    assign r = {rgb_pixel[11:8], 4'b0};
    assign g = {rgb_pixel[7:4], 4'b0};
    assign b = {rgb_pixel[3:0], 4'b0};

    parameter IDLE = 2'b00;
    parameter READ_DELAY = 2'b01;
    parameter SMOOTHING = 2'b10;
    parameter DONE = 2'b11;
    logic [1:0] state;

    parameter RGB_2_HSV_CYCLES = 19;

    reg [32:0] hsv_buffer [0:RGB_2_HSV_CYCLES - 1];
    logic [4:0] rgb_2_hsv_sel;

    genvar i;
    generate
```

```
        for(i = 0; i < RGB_2_HSV_CYCLES; i++)begin
            rgb_2_hsv inst(
                .clk(clk),
                .rst(rst),
                .start(rgb_2_hsv_sel == i),
                .r_in(r),
                .g_in(g),
                .b_in(b),
                .h(hsv_buffer[i][32:16]), //Q9.8
                .s(hsv_buffer[i][15:8]),  //Q8
                .v(hsv_buffer[i][7:0])    //Q8
            );
        end
    endgenerate

    logic [16:0] h;
    logic [7:0] s;
    logic [7:0] v;
    assign h = hsv_buffer[rgb_2_hsv_sel][32:16];
    assign s = hsv_buffer[rgb_2_hsv_sel][15:8];
    assign v = hsv_buffer[rgb_2_hsv_sel][7:0];

    logic bin_maze_pixel;


    logic wall_color;
    thresholder #(.H_LOW(17'h0_A0_00),.H_HIGH(17'h1_04_00),
                  .S_LOW(8'b0000_0000),.S_HIGH(8'b1111_1111),
                  .V_LOW(8'b0010_0000),.V_HIGH(8'b1111_1111))
skel_thresh_wall
                    (
                       .h(h),   //Q9.8
                       .s(s),  //Q8
                       .v(v),  //Q8
                       .b(wall_color)
                    );
    assign bin_maze_pixel = 1 - wall_color;


    //This threshold for yellow (start)
    logic yellow;
    thresholder #(.H_LOW(17'h0_28_00),.H_HIGH(17'h0_5a_00),
                  .S_LOW(8'b1000_0000),.S_HIGH(8'b1111_1111),
                  .V_LOW(8'b0010_0000),.V_HIGH(8'b1111_1111))
skel_thresh_end
                (
                   .h(h),   //Q9.8
                   .s(s),  //Q8
                   .v(v),  //Q8
                   .b(yellow)
                );

    //This thresholds for red (end)
    logic red;
    logic red1;
    thresholder #(.H_LOW(17'h0_00_00),.H_HIGH(17'h0_14_00),
                      .S_LOW(8'b1100_0000),.S_HIGH(8'b1111_1111),
                      .V_LOW(8'b0010_0000),.V_HIGH(8'b1111_1111))
skel_thresh_red1
                (
                    .h(h),   //Q9.8
                    .s(s),  //Q8
                    .v(v),  //Q8
                    .b(red1)
                );
    logic red2;
    thresholder #(.H_LOW(17'h1_54_00),.H_HIGH(17'h1_FF_FF),
                  .S_LOW(8'b1100_0000),.S_HIGH(8'b1111_1111),
                  .V_LOW(8'b0010_0000),.V_HIGH(8'b1111_1111))
skel_thresh_red2
                (
                   .h(h),   //Q9.8
                   .s(s),  //Q8
                   .v(v),  //Q8
                   .b(red2)
                );
    assign red = red1 || red2;

    logic [23:0] cycles;

    logic eroded_pixel;
    logic dilated_pixel;

    logic eroded_yellow;
    logic dilated_yellow;

    logic eroded_red;
    logic dilated_red;

    assign pixel_out = dilated_pixel;
    assign start_color = dilated_yellow;
    assign end_color = dilated_red;

    logic start_erosion;
    logic start_dilation;
    logic dilation_done;

    assign start_erosion = cycles == RGB_2_HSV_CYCLES;

    erosion #(.K(5),.IMG_W(320),.IMG_H(240)) bin_erosion
                (
                .clk(clk),
                .rst(rst),
                .start(start_erosion),
                .pixel_in(bin_maze_pixel),
                .pixel_valid(start_dilation),
                .processed_pixel(eroded_pixel)
                );

    dilation #(.K(5),.IMG_W(320),.IMG_H(240)) bin_dilation
                    (
                    .clk(clk),
                    .rst(rst),
                    .start(start_dilation),
                    .pixel_in(eroded_pixel),
                    .pixel_addr(pixel_wr_addr),
                    .pixel_valid(pixel_wea),
                    .processed_pixel(dilated_pixel),
                    .done(dilation_done)
                    );

    //SMOOTH YELLOW
    erosion #(.K(5),.IMG_W(320),.IMG_H(240)) yellow_erosion
                (
                .clk(clk),
                .rst(rst),
                .start(start_erosion),
                .pixel_in(yellow),
                .processed_pixel(eroded_yellow)
                );

    dilation #(.K(5),.IMG_W(320),.IMG_H(240)) yellow_dilation
                    (
                    .clk(clk),
                    .rst(rst),
                    .start(start_dilation),
                    .pixel_in(eroded_yellow),
                    .processed_pixel(start_color)
                    );


    //SMOOTH RED
     erosion #(.K(5),.IMG_W(320),.IMG_H(240)) red_erosion
                 (
                 .clk(clk),
                 .rst(rst),
                 .start(start_erosion),
                 .pixel_in(red),
                 .processed_pixel(eroded_red)
                 );

     dilation #(.K(5),.IMG_W(320),.IMG_H(240)) red_dilation
                     (
                     .clk(clk),
                     .rst(rst),
                     .start(start_dilation),
                     .pixel_in(eroded_red),
                     .processed_pixel(end_color)
                     );

     assign done = state == DONE;
     //Debugging
    `ifdef SIM
    integer bin_maze_f;
    `endif
    always_ff @(posedge clk)begin
        if(rst)begin
            state <= IDLE;
        end
        else begin
            case(state)
                IDLE: begin
                    if(start)begin
                        state <= READ_DELAY;
                        cam_pixel_r_addr <= 0;
                    end
                end
                READ_DELAY: begin
                    //cam_pixel_r_addr reads from BRAM the value that we
will want BRAM_READ_DELAY cycles from now
                    cam_pixel_r_addr <= cam_pixel_r_addr + 1;
                    if(cam_pixel_r_addr == BRAM_READ_DELAY - 1)begin
                        state <= SMOOTHING;
                        rgb_2_hsv_sel <= 0;
                        cycles <= 0;
                        `ifdef SIM
                        bin_maze_f =
$fopen("C:/Users/giand/Documents/MIT/Senior_Fall/6.111/gims-labyrinth/gims_
labyrinth/python_stuff/verilog_testing/bin_maze_img.txt","w");
                        `endif
                    end
                end
                SMOOTHING: begin
                    cam_pixel_r_addr <= cam_pixel_r_addr + 1;
                    if(rgb_2_hsv_sel == RGB_2_HSV_CYCLES - 1)begin
                        rgb_2_hsv_sel <= 0;
                    end
                    else begin
                        rgb_2_hsv_sel <= rgb_2_hsv_sel + 1;
                    end

                    if(dilation_done)begin
                        state <= DONE;
                    end

                    `ifdef SIM
                    if(cycles == (RGB_2_HSV_CYCLES + IMG_W * IMG_H))begin
                    $fclose(bin_maze_f);
                    end
                    else if(cycles >= RGB_2_HSV_CYCLES)begin
                    $fwrite(bin_maze_f,"%b\n",bin_maze_pixel);
                    end
                    `endif

                    cycles <= cycles + 1;
                end
                DONE: begin
                    state <= IDLE;
                    cycles <= 0;
                end
            endcase
        end

    end

endmodule


module end_node_finder #(parameter IMG_W = 320, parameter IMG_H = 240,
parameter BRAM_READ_DELAY = 2)
    (
        input clk,
        input start,
        input rst,
        input [11:0] skel_pixel,
        input [11:0] start_yellow_pixel,
        input [11:0] end_red_pixel,

        output done,
        output logic [16:0] maze_pixel_addr,
        output logic [16:0] start_pos,
        output logic [16:0] end_pos
    );

    parameter IDLE = 2'b00;
    parameter READ_DELAY = 2'b01;
    parameter PROCESSING = 2'b10;
    parameter DONE = 2'b11;
    logic [1:0] state;

    logic [8:0] x;
    logic [7:0] y;
    assign done = state == DONE;

    always_ff @(posedge clk)begin
        if(rst)begin
            state <= IDLE;
        end
        else begin
            case(state)
                IDLE: begin
                    if(start)begin
                        state <= READ_DELAY;
                        maze_pixel_addr <= 0;
                    end
                end
                READ_DELAY: begin
                    //maze_pixel_addr reads from BRAM the value that we
will want BRAM_READ_DELAY cycles from now
                    maze_pixel_addr <= maze_pixel_addr + 1;
                    if(maze_pixel_addr == BRAM_READ_DELAY - 1)begin
                        state <= PROCESSING;
                        x <= 9'b0;
                        y <= 8'b0;
                    end
                end
                PROCESSING: begin
                    maze_pixel_addr <= maze_pixel_addr + 1;
                    if(x == IMG_W - 1)begin
                        x <= 9'b0;
                        y <= y + 1;
                    end
                    else begin
                        x <= x + 1;
                    end
                    // if skel_pixel && start_yellow pixel, set output
start_pos
                    if(skel_pixel && start_yellow_pixel)begin
                        start_pos <= {x, y};
                    end
                    // if skel_pixel && end_red_pixel, set output end_pos
                    else if(skel_pixel && end_red_pixel)begin
                        end_pos <= {x, y};
                    end

                    if(x == (IMG_W - 1) && y == (IMG_H - 1))begin
                        state <= DONE;
                    end
                end
                DONE: begin
                    state <= IDLE;
                end
            endcase
        end

    end

endmodule
```

skeleton_to_graph.sv

```
module skeleton_intersection_finder #(parameter IMG_W = 640, parameter
IMG_H = 480, parameter BRAM_READ_DELAY = 2)
    (
    input clk,
    input rst,
    input start,
    input pixel_r,

    output logic [18:0] window_end_i_read,

    output logic [18:0] window_center_i,
    output logic [2:0] pixel_out,
    output logic write_pixel,

    output logic write_node,
    output logic [9:0] node,
    output logic [18:0] node_xy,
    output done,
    output logic [1:0] state
    );

    parameter IDLE = 2'b00;
    parameter READ_DELAY = 2'b01;
    parameter FIND_INTERSECTIONS = 2'b10;
    parameter DONE = 2'b11;

    //This contains the 3x3 window that we will apply the kernel on.
    reg [((IMG_W << 1) + 3) - 1 : 0] skel_window;
    //logic [1:0] state;

    //This constructs the 3x3 window shown below
    // w_00 w_10 w_20
    // w_01 w_11 w_21
    // w_02 w_12 w_22

    logic w_11, w_01, w_10, w_21, w_12;
    assign w_11 = skel_window[IMG_W + 1];
    assign w_01 = skel_window[IMG_W + 2];
    assign w_10 = skel_window[(IMG_W << 1) + 1];
    assign w_21 = skel_window[IMG_W];
    assign w_12 = skel_window[1];

    //This marks the location on the image of pixel skel_window[0]
    logic [9:0] x_end;
    logic [8:0] y_end;

    //This marks the location on the image of the center of skel_window
    logic [9:0] x_c;
    logic [8:0] y_c;

    logic [18:0] window_end_i;
    always_comb begin
        window_end_i = IMG_W * y_end + x_end;
        //Gets the pixel value that we will want in BRAM_READ_DELAY cycles
        x_c = x_end - 1;
        y_c = y_end - 1;
        window_center_i = window_end_i - (IMG_W + 3);
    end

    //Below is the kernel being applied to the 3x3 window
    logic intersection;
    assign intersection = w_11 &&
                         ((w_01 && w_21 && w_12) ||
                         (w_10 && w_21 && w_12) ||
                         (w_10 && w_01 && w_12) ||
                         (w_10 && w_01 && w_21));

    logic [9:0] n;
    assign done = (state == DONE);

    always_ff @(posedge clk)begin
        if(rst)begin
            state <= IDLE;
        end
        else begin
            skel_window <= {skel_window[((IMG_W << 1) + 3) - 2:0],
pixel_r};
            case(state)
                IDLE: begin
```

```
                    if(start)begin
                        state <= READ_DELAY;
                        window_end_i_read <= 0;
                        //The zeroth node is reserved as a dummy node for
dijkstra
                        n <= 10'b1;
                    end
                end
                READ_DELAY: begin
                    //window_end_i_read reads from BRAM the value that we
will want BRAM_READ_DELAY cycles from now
                    window_end_i_read <= window_end_i_read + 1;
                    if(window_end_i_read == BRAM_READ_DELAY)begin
                        x_end <= 10'd00;
                        y_end <= 9'd00;
                        state <= FIND_INTERSECTIONS;
                    end
                end
                FIND_INTERSECTIONS: begin
                    window_end_i_read <= window_end_i_read + 1;
                    //Wait until window is completely filled before finding
intersections
                    //Avoid y <= 2, y >= IMG_H - 3, x <= 2 and x >= IMG_W -
3 to avoid artifacts around edges of image from skeletonization
                    if(window_end_i > ((IMG_W << 1) + 3) && y_c > 9'd2 &&
y_c < IMG_H - 3 && x_c > 10'd2 && x_c < IMG_W - 3)begin
                        if(intersection)begin
                            //If we have an intersection, we mark it with
the value 3'b010 in the bram
                            write_pixel <= 1;
                            pixel_out <= 3'b010;

                            node <= n;
                            node_xy <= {x_c, y_c};
                            n <= n + 1;
                        end
                        else begin
                            write_pixel <= 0;
                        end
                    end

                    //Increment pixel coordinates
                    if(x_end == IMG_W - 1)begin
                        x_end <= 10'b0;
                        y_end <= y_end + 1;
                    end
                    else begin
                        x_end <= x_end + 1;
                    end

                    //Window has reached the end. We can stop now
                    if(x_end == IMG_W - 1 && y_end == IMG_H - 1)begin
                        write_pixel <= 0;
                        state <= DONE;
                    end
                end
                DONE: begin
                    state <= IDLE;
                end
            endcase
        end
    end

endmodule
```

skeletonizer.sv

```
module skeletonizer #(IMG_WIDTH=320, IMG_HEIGHT=240, BRAM_READ_DELAY=2)(
    input clk,
    input rst,
    input start,
    input pixel_in,
      // read data from bram

    output logic [16:0] pixel_r_addr,
    output logic [16:0] pixel_wr_addr,
      output logic pixel_we,        // bram write enable
      output logic pixel_out,       // write data to bram

    output logic done
    );

    parameter IDLE = 2'b00;
    parameter INITIAL_READ_DELAY = 2'b01;
    parameter SKELETONIZING = 2'b10;
    parameter DONE = 2'b11;


    logic [2*IMG_WIDTH + 2:0] unmod_pixel_buffer, mod_pixel_buffer, temp;
      logic [8:0] unmod_window, mod_window;
      logic [7:0] disc_check_buffer1, disc_check_buffer2;   //used to check
for discontinuity
      logic [7:0] xored_disc_buffers;
      logic [3:0] num_transitions;  // number of transitions for
discontinuity check
      logic causes_discontinuity;

      logic [1:0] state;
      logic changes_made;
      // flag to see if changes were made while skeletonizing
      integer i;
      integer j;

      logic [8:0] center_x;                                            // x
```

```
location of the center pixel of the windows
      logic [7:0] center_y;                                        // y
location of the center pixel of the windows

      logic h0, h1, h2, h3, h4, h5, h6, h7, h0_7;
      logic skeletonized_pixel;
      integer skel_maze_f;

    always_comb begin
            //This constructs the windows from our pixel buffers
            for(i = 0; i < 3; i = i + 1) begin
                  for(j = 0; j < 3; j = j + 1) begin
                        unmod_window[3*i + j] =
unmod_pixel_buffer[i*IMG_WIDTH + j];
                        mod_window[3*i + j] = mod_pixel_buffer[i*IMG_WIDTH
+ j];
                  end
            end
            pixel_wr_addr = IMG_WIDTH*center_y + center_x;
            pixel_we = (state == SKELETONIZING);
            done = (state == DONE);

            // check for discontinuity
            disc_check_buffer1 = {mod_window[8], mod_window[7],
mod_window[6], mod_window[3], mod_window[0], mod_window[1], mod_window[2],
mod_window[5]};
            disc_check_buffer2 = {mod_window[7], mod_window[6],
mod_window[3], mod_window[0], mod_window[1], mod_window[2], mod_window[5],
mod_window[8]};
            xored_disc_buffers = disc_check_buffer1 ^ disc_check_buffer2;
            num_transitions = xored_disc_buffers[7] + xored_disc_buffers[6]
+ + xored_disc_buffers[5] + xored_disc_buffers[4] +
                                          xored_disc_buffers[3] +
xored_disc_buffers[2] + xored_disc_buffers[1] + xored_disc_buffers[0];
            // flag to see if skeletonizing would cause discontinuity
            causes_discontinuity = num_transitions > 2;


            // skeletonizing logic
            h0 = (unmod_window[8] == 0) & (unmod_window[5] == 0) &
(unmod_window[2] == 0) & (unmod_window[4] == 1) & (unmod_window[6] == 1)
            & (unmod_window[3] == 1) & (unmod_window[0] == 1);
```

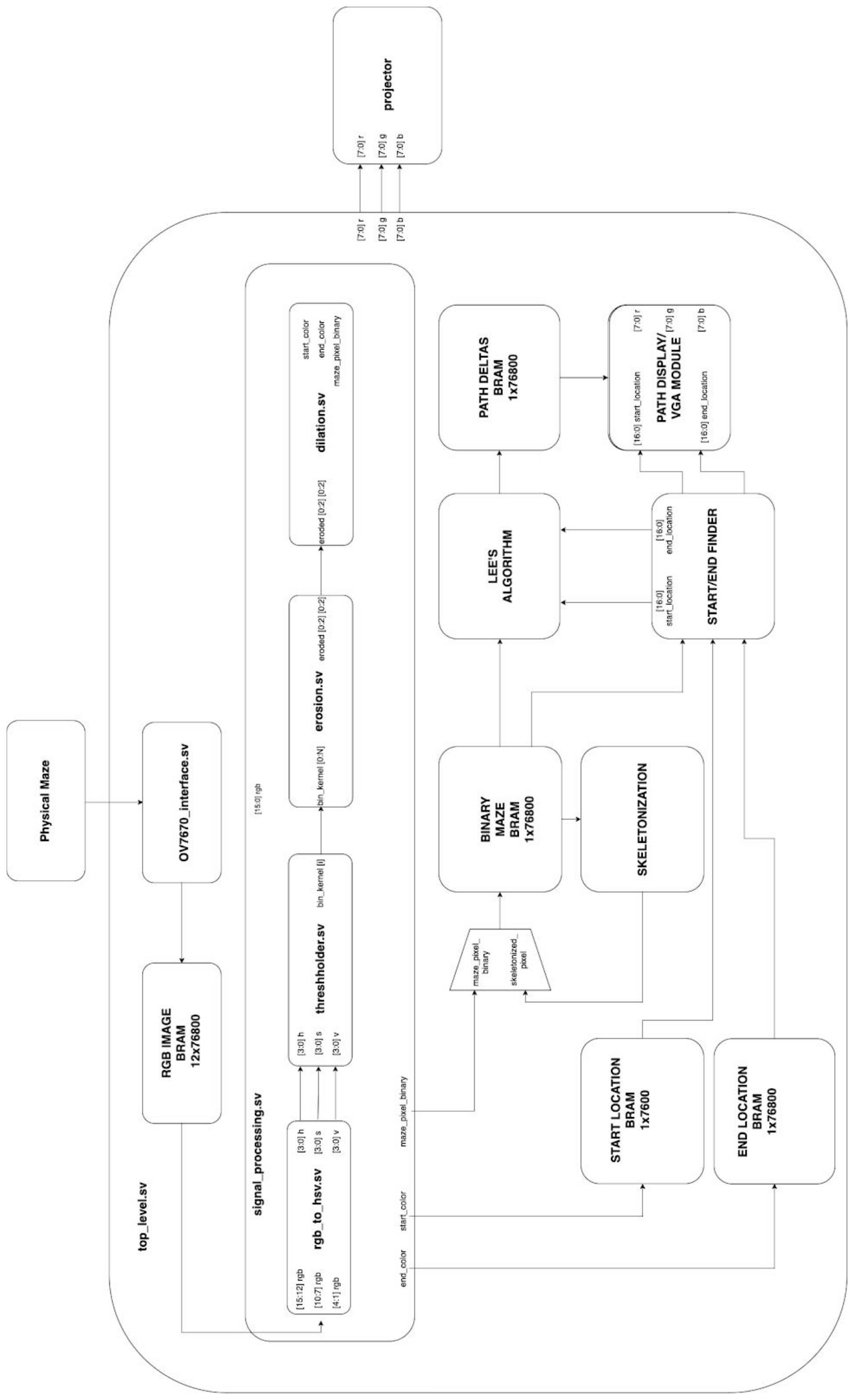
```
        h1 = (unmod_window[5] == 0) & (unmod_window[2] == 0) &
(unmod_window[1] == 0) & (unmod_window[7] == 1) & (unmod_window[4] == 1)
            & (unmod_window[3] == 1);
        h2 = (unmod_window[2] == 0) & (unmod_window[1] == 0) &
(unmod_window[0] == 0) & (unmod_window[8] == 1) & (unmod_window[7] == 1)
            & (unmod_window[4] == 1) & (unmod_window[6] == 1);
        h3 = (unmod_window[1] == 0) & (unmod_window[3] == 0) &
(unmod_window[0] == 0) & (unmod_window[5] == 1) & (unmod_window[7] == 1)
            & (unmod_window[4] == 1);
        h4 = (unmod_window[6] == 0) & (unmod_window[3] == 0) &
(unmod_window[0] == 0) & (unmod_window[8] == 1) & (unmod_window[5] == 1)
            & (unmod_window[2] == 1) & (unmod_window[4] == 1);
        h5 = (unmod_window[7] == 0) & (unmod_window[6] == 0) &
(unmod_window[3] == 0) & (unmod_window[5] == 1) & (unmod_window[4] == 1)
            & (unmod_window[1] == 1);
        h6 = (unmod_window[8] == 0) & (unmod_window[7] == 0) &
(unmod_window[6] == 0) & (unmod_window[2] == 1) & (unmod_window[4] == 1)
            & (unmod_window[1] == 1) & (unmod_window[0] == 1);
        h7 = (unmod_window[8] == 0) & (unmod_window[5] == 0) &
(unmod_window[7] == 0) & (unmod_window[4] == 1) & (unmod_window[1] == 1)
            & (unmod_window[3] == 1);

        h0_7 = ~(h0 | h1 | h2 | h3 | h4 | h5 | h6 | h7);

        skeletonized_pixel = unmod_window[4] & h0_7;
            pixel_out = causes_discontinuity ? unmod_window[4] :
skeletonized_pixel;

            // combinationally sets the center pixel of temp to be the
output pixel (will shift this into mod_pixel_buffer later)
            temp = {mod_pixel_buffer[2*IMG_WIDTH + 2: IMG_WIDTH + 2],
pixel_out, mod_pixel_buffer[IMG_WIDTH:0]};

    end

    always_ff @(posedge clk)begin
        if(rst)begin
            state <= IDLE;
                  unmod_pixel_buffer <= 0;
                  mod_pixel_buffer <= 0;
        end
        else begin
```

```
                  // always shift what we read from bram into
unmod_pixel_buffer
                  unmod_pixel_buffer <= {unmod_pixel_buffer[2*IMG_WIDTH +
1: 0], pixel_in};
                  // Update mod_pixel_buffer by shifting temp
            mod_pixel_buffer <=  {temp[2*IMG_WIDTH + 1:0], pixel_in};

            case(state)
                IDLE: begin
                    if(start)begin
                        state <= INITIAL_READ_DELAY;
                        pixel_r_addr <= 17'b0;
                                    changes_made <= 1'b0;

                    end
                end
                INITIAL_READ_DELAY: begin
                    // read_address reads from BRAM the value that we will
want BRAM_READ_DELAY cycles from now
                    pixel_r_addr <= pixel_r_addr + 1;
                              // wait until buffer is half full before
starting skeletonization
                    if(pixel_r_addr == ((BRAM_READ_DELAY - 1) + (IMG_WIDTH
+ 2)))begin
                        center_x <= 9'b0;
                        center_y <= 8'b0;
                        state <= SKELETONIZING;
                        skel_maze_f =
$fopen("C:/Users/giand/Documents/MIT/Senior_Fall/6.111/gims-labyrinth/gims_
labyrinth/python_stuff/verilog_testing/maze_skel.txt","w");
                    end
                end
                SKELETONIZING: begin
                    $fwrite(skel_maze_f,"%b\n",pixel_out);
                    //Increment center pixel coordinates
                    if(center_x == IMG_WIDTH - 1)begin
                        center_x <= 9'b0;
                        center_y <= center_y + 1;
                    end
                    else begin
                        center_x <= center_x + 1;
                    end
                    if(pixel_r_addr == IMG_WIDTH * IMG_HEIGHT - 1)begin
                        pixel_r_addr <= 17'b0;
                    end
                    else begin
                        pixel_r_addr <= pixel_r_addr + 1;
                    end

                    //Have processed all the pixels
                    if(center_x == IMG_WIDTH - 1 && center_y == IMG_HEIGHT
- 1)begin
                                    if(changes_made) begin
      // repeat if any changes were made
                                          center_x <= 9'b0;
                            center_y <= 8'b0;
                                          changes_made <= 1'b0;
                                          $fclose(skel_maze_f);
                                          skel_maze_f =
$fopen("C:/Users/giand/Documents/MIT/Senior_Fall/6.111/gims-labyrinth/gims_
labyrinth/python_stuff/verilog_testing/maze_skel.txt","w");
                                    end else begin
            // otherwise we are done
                                          state <= DONE;
                                          $fclose(skel_maze_f);
                                    end
                    end
                              else begin
                                    changes_made <= changes_made |
(pixel_out != unmod_window[4]);
                              end
                end
                DONE: begin
                    state <= IDLE;
                end
            endcase
        end
    end

endmodule
```

thresholder.sv

```
module thresholder #(parameter H_LOW, parameter H_HIGH,
                     parameter S_LOW, parameter S_HIGH,
                     parameter V_LOW, parameter V_HIGH)
    (
        input logic [16:0] h,   //Q9.8
        input logic [7:0] s,  //Q8
        input logic [7:0] v,  //Q8
        output b
    );

    assign b = (H_LOW <= h && h <= H_HIGH) && (S_LOW <= s && s <= S_HIGH)
&& (V_LOW <= v && v <= V_HIGH);
endmodule
```

System Block Diagram


projector
[7:0] r
[7:0] g
[7:0] b
Physical Maze
top_level.sv
OV7670_interface.sv
RGB IMAGE BRAM 12x76800
signal_processing.sv
[15:0] rgb
rgb_to_hsv.sv
[15:12] rgb
[10:7] rgb
[4:1] rgb
[3:0] h
[3:0] s
[3:0] v
threshholder.sv
bin_kernel [i]
erosion.sv
bin_kernel [0:N]
eroded [0:2] [0:2]
dilation.sv
start_color
end_color
maze_pixel_binary
maze_pixel_binary
start_color
end_color
maze_pixel_binary
skeletonized_pixel
BINARY MAZE BRAM 1x76800
SKELETONIZATION
LEE'S ALGORITHM
PATH DELTAS BRAM 1x76800
PATH DISPLAY/ VGA MODULE
[16:0] start_location
[16:0] end_location
START/END FINDER
START LOCATION BRAM 1x7600
END LOCATION BRAM 1x76800