Bacteria "Colonalyzer"

Yaw Anku

MIT Course 6.111: Introductory Digital Systems Laboratory

October 30, 2006

Abstract

This report describes the design and implementation of a digital raster graphic analyzer that detects the number of distinct blobs in a digital image. The name of the project is motivated by the possible application of the project to detect bacteria colonies on a petri dish sample. The design utilizes a sequential scan labeling algorithm to identify the pixels of the image that form one entity. The steps involved in this algorithm are filtering, initial scan labeling and second scan labeling of the 3200x240 pixel input image. The design uses a controller module that manages the transitions between the stages of the image processing implemented by other modules. A 640x320 VGA display is used to show an overlay detailing the pixel by pixel progression of the algorithm at user-variable speeds. Both image processing and the VGA display rely on the BRAM memory in the FPGA.

Table of Contents

1.0 Overview....4
  1.1 Theory of Blob Detection Algorithm....5
  1.2 User Interface....7
2.0 Description....8
  2.2 Binary Image Generator....13
  2.3 Image Labeler FSM....13
  2.4 Image Labeler....14
  2.5 Image Label Correcter....15
  2.6 Blob Counter....15
  2.6 VGA....15
  2.7 Image Pixel Pointer....16
4.0 Reflection....17
5.0 Conclusion....17
6.0 Reference....17
7.0 Acknowledgments....17
8.0 Appendices....18
  A Bacteria_Colonalyzer (Top level Module)....18
  B Controller....38
  C Binary Image Generator....41
  D Image Labeler FSM....43
  E Image Labeler....46
  F Image Label Correcter....53
  G Blob Counter....55
  E Image Pixel Pointer....58

List of Figures

Figure 1: System Process Flow..4
Figure 2: Test Image...5
Figure 3: Result Image..5
Figure 4: 6-connectedness Neighborhood Scheme...6
Figure 5: Status of image labels after first scan..6
Figure 6: Pseudo-code for left-skewed sequential labeling...7
Figure 7: User Interface...8
Figure 8: Overall System Block Diagram..9
Figure 9: Controller FSM...10
Figure 10: Image Labeler FSM..13

1.0 Overview

The Bacteria Colonalyzer is an implementation of a digital raster scan algorithm for detecting the number of distinct blobs in an image. The system takes an image as input and runs it through an ordered sequence of processing steps to detect the number of distinct blobs that satisfy a particular color filtering constraint (red or blue). The system was implemented using the 6.111 lab kit and the lab computer monitor. Figure 1 Shows the process flow of the system. First the blob type is detected based on whether the user specifies red or blue blobs to be detected. The result of this detection is then processed in the image analyses stage.

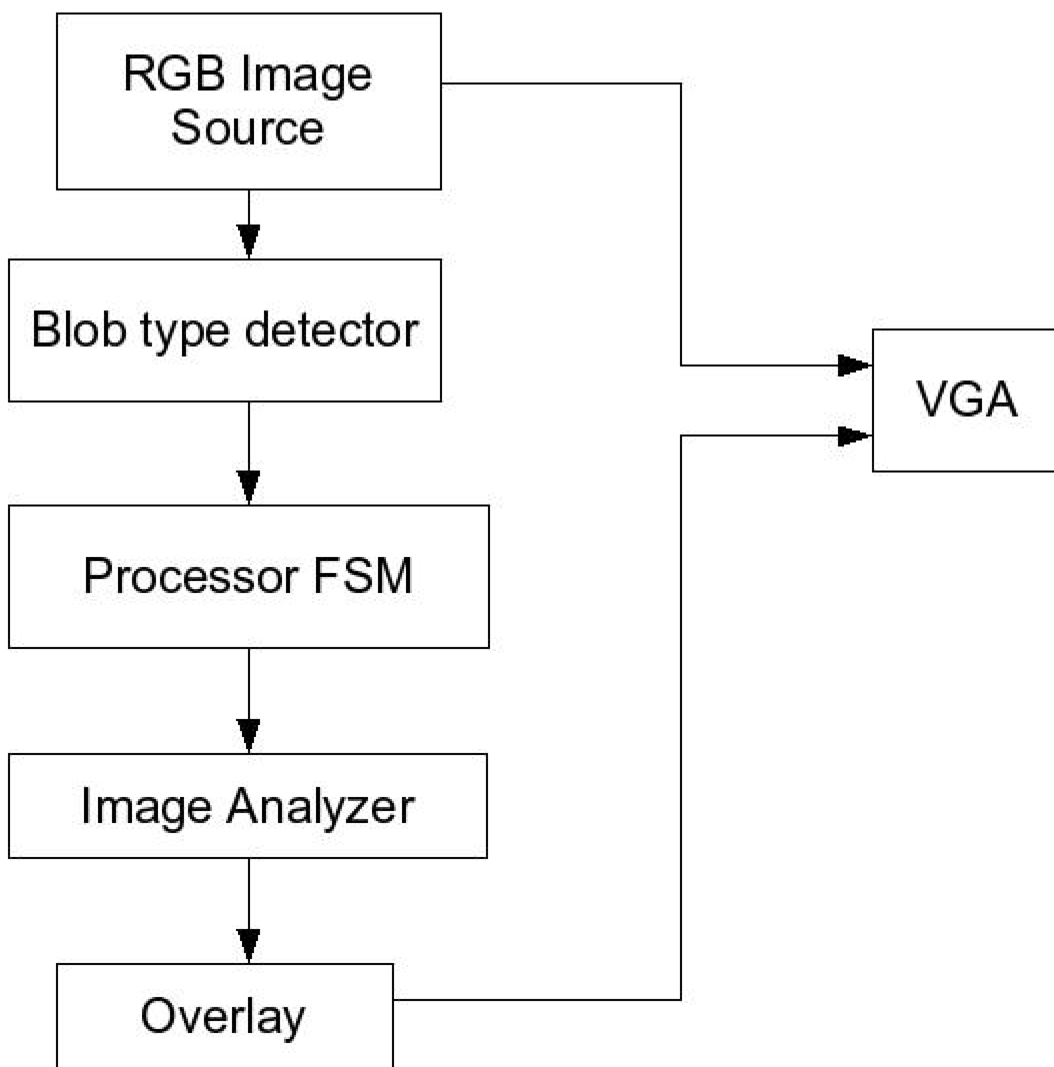

Figure 1: System Process Flow

For a example, the result of passing the test image in Figure 2 through the system would be the result image in Figure 3. The result image represents the detection of all the blobs in the original image irrespective of color. The system labels the blobs with distinct colors to signify that they belong to distinct blobs.

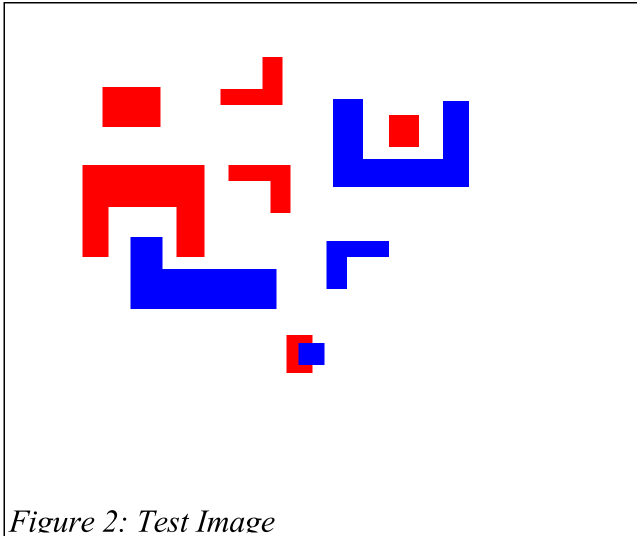

Figure 2: Test Image

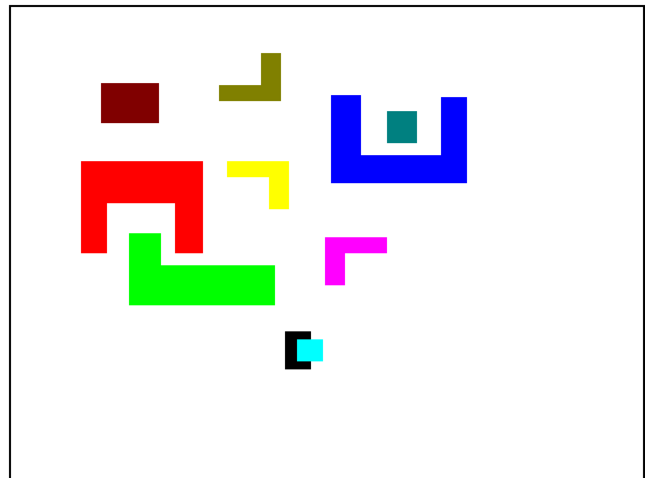

Figure 3: Result Image

1.1 Theory of Blob Detection Algorithm

The algorithm used to detect distinct blob is a sequential, left-skewed, 6-connectedness neighborhood scheme. This algorithm was chosen over several other existing algorithms because of is simplicity both in understanding and implementation. The process requires two scans. During the first scan, the image is scanned from left to right, row by row, from top to bottom. For every pixel encountered during the scan that passes the filtering test, the labels of neighbors to the left, north-west diagonal and above the current pixel are used to determine the label to be assigned to the current pixel location. Figure 4 is an example of a typical region in the image, for any particular pixel or cell, C we know that the cell to its left, L, the cell directly above, U, and the diagonal cell D have already been labeled. As such, we can safely say that cell is is likely to belong to the same blob as at least one of these blobs. If the diagonal pixel is labeled, the current cell is assigned the same pixel value. Otherwise, cell U and then cell L are considered in the same manner. If both cell L and U have the same pixel value, then C

obviously takes on that value. If however the labels for L and U are different, then we decide to assign C the value of U and specify in an equivalence table that L should have the same label as cell U. It would be wrong to make the correction during the first scan because each we do not have the power to determine what the label values are for pixels more than one cell away. This is precisely the reason why a second scan is needed. The second scan uses the values in the equivalence table to correct the wrongly labeled cells in the first scan. Using figure 5 as an example, a second scan through the image will identify the regions 1 and 2 as one region.

D	U	
L	C	

Figure 4: The 6-connectedness neighborhood scheme(By Robyn Owens - “Math Morphology”)

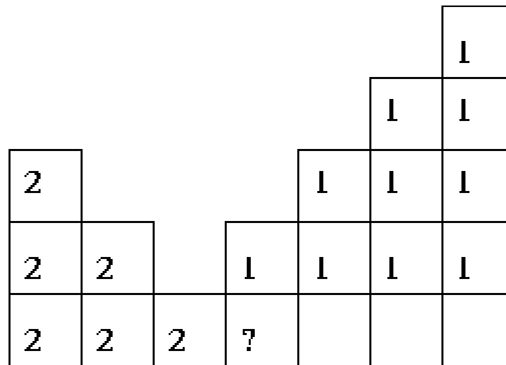

Figure 5: Status of image labels after first scan (Robyn Owens - "Math Morphology")

The special cases in the algorithm are when the pixel being considered is on the top or left edge of the image. L, D will not exist on the left edge just as U and D will not exist on the top edge. The system takes care of these discrepancies by assigning label values of zero for cells that do not exist. The

Image Labeler FSM module generates signals to handle these special cases. The pseudo-code for the first scan of the algorithm is show in figure 6.

```
if P = O
 do nothing
else if D labeled
     copy label to C

else if (not L labeled) and (not U labeled)
     increment label numbering and label C

else if L xor U labeled
     copy label to C

else if L and U labeled
     if L label = U label
          copy label to C
     else
          copy U label to C
          record equivalence of label L to U
```

Figure 6 : Pseudo-code for Left Skewed Sequential Labeling

1.2 User Interface

The user interface to the system consists of the push buttons, LEDs and switches of on the 6.111 lab kit as shown in Figure 2. The the image processing is initiated when the user hits the start button. The reset button resets the entire system. The left most switch (switch[7]) is used to switch display between the test image and the overlay of the labeling algorithm. Switches 6 and 5 allow the user to specify which blob colors to filter out. Led 7 lights when the start button is pressed. LED 6 lights when the labeling

process is done. LEDs 0 through 2 display which of the state the processor is in, whether idling, doing the initial scan, correcting the labels on the second scan, or counting the labels. When the processing completes, the number of the blobs filtered and detected is shown on the HEX display.

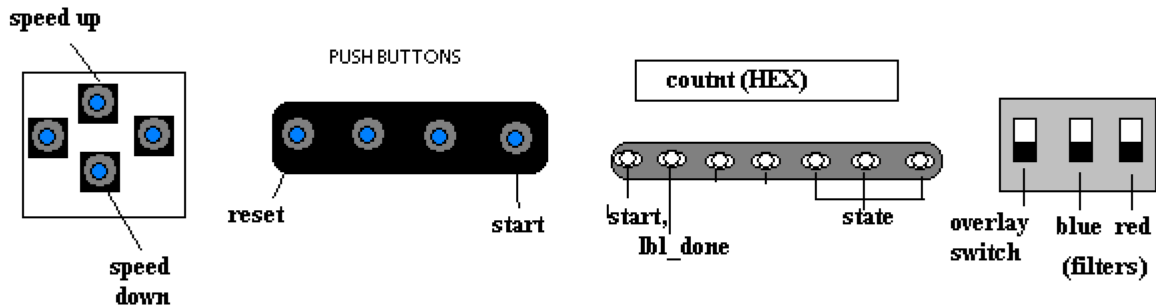

Figure 7: User Interface (6.111 Lab kit).

2.0 Description

The Image processor is the defining super-module of the system because it is the module that implements the core functionality of the system. The VGA module outputs the processing stages and the results of the Image processor. This section describes the submodules of the Image processor as well as those modules used for the display. Every substantially complex processing step in the blob detection process is abstracted into a separate module. The idea was to have a design that would allow easy scaling up of the overall functional complexity of the system. As a result the Image Processing task was divided amongst the Binary Generator Module, The Image Labeler FSM, the Image Labeler, the Image Label Corrector and the Blob Counter modules. The Image Pixel Pointer module and the VGA sync modules were used to generate pixel values for the 640x480 display. The design uses several dual port buffers to store the state of the image pixels and labels at different stages in the process. The reason for the use od dual port memory is to allow the VGA to constantly read from one port whiles the process writes through the other port. The overall system diagram is shown in Figure 7.

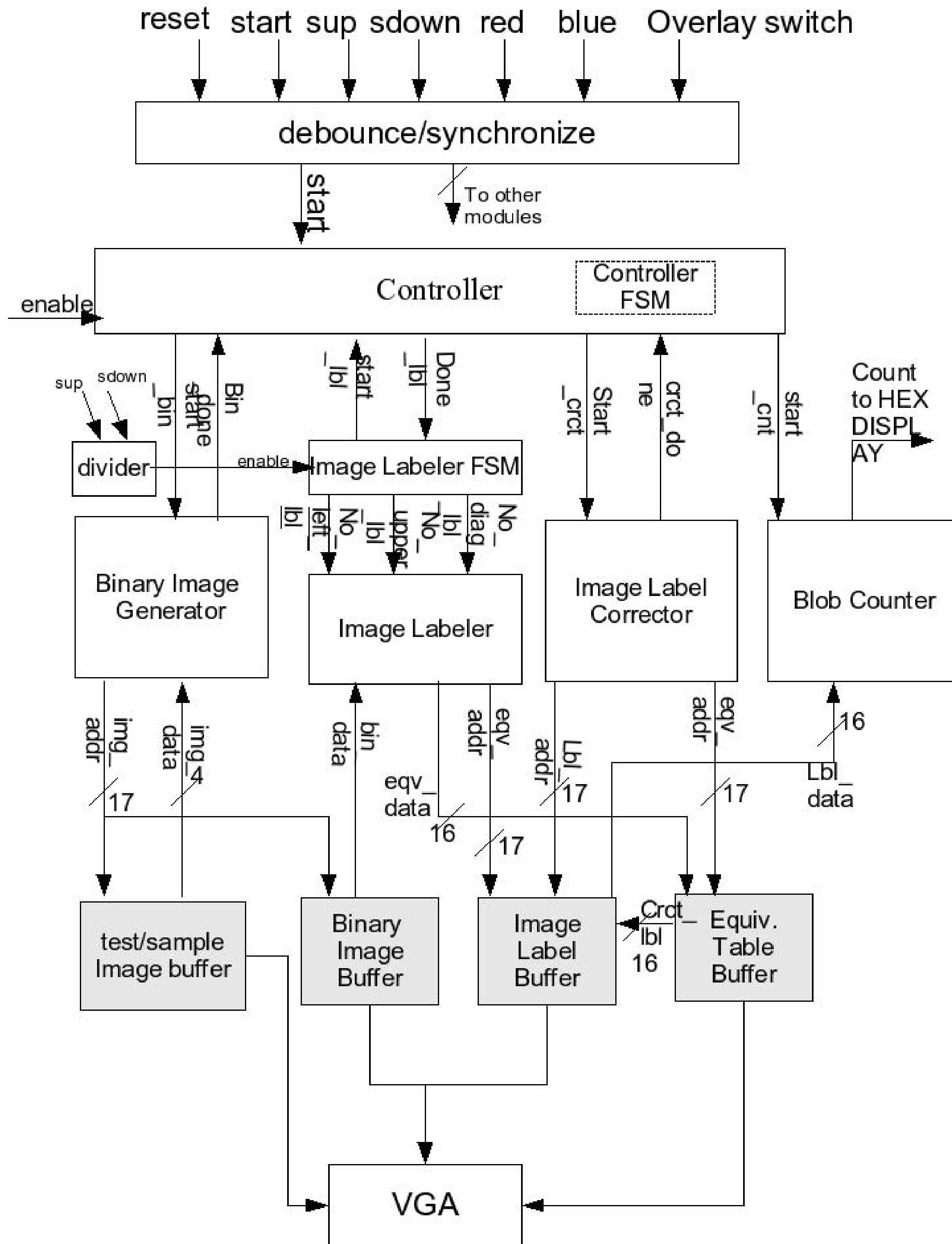

Figure 8: Overall System Block Diagram: Image processing occurs from left to right of the middler row of the diagram. The buffers which store the image pixels and label values are shaded. The system is designed to display the status of the image (the buffer contents) as the process progresses.

2.1 Controller

The controller, as is name implies, controls the different states and signals associated with the image processing. It takes in a , the binary image generator, the image labeler, the image label corrector, and the blob counter. These signals from the user and the other modules are used by the controller to determine the next state transition. The controller specifies the next state by sending a start signal to the next module in the process when it receives a done signal from the current processing module. For example it uses a start signal from the user to signal the binary image generator to start processing. When the binary image processor completes processing, its sends a done signal which the controller uses as an indicate to send a start signal to the labeler.

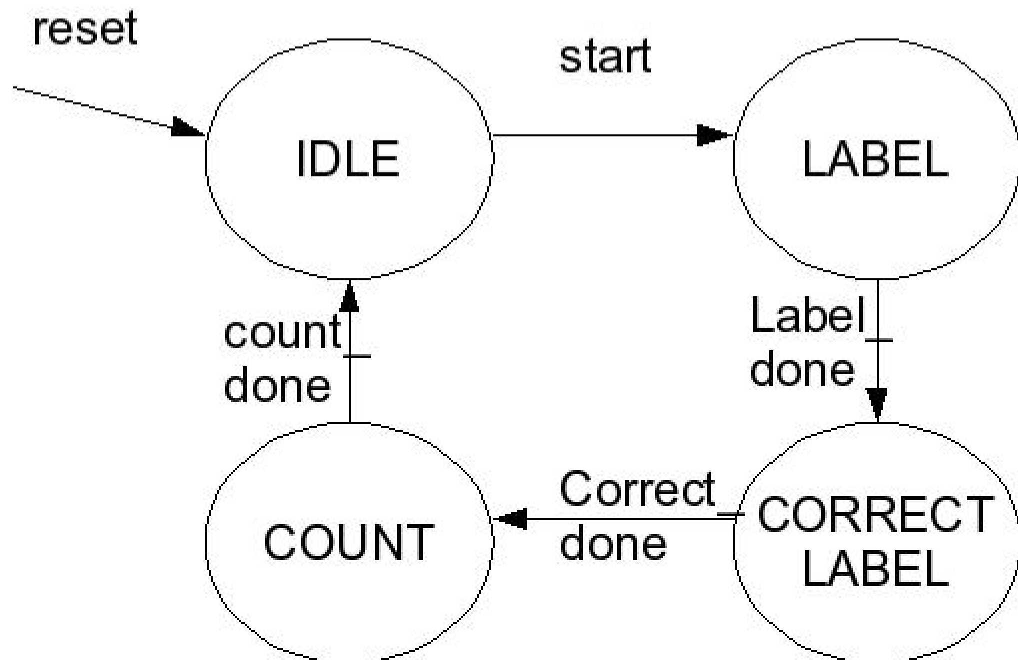

Figure 9: Controller FSM

2.2 Binary Image Generator

The Binary Image Generator implements the filtering stage of the image analyses process. It loads up the input image pixel by pixel from the test image buffer and filters out pixels that do not meet the filter threshold. The threshold is specified by user input through switches 6 and 0 of the FPGA. It writes a value of 1 for pixels that meet the threshold and a value of zero for pixels that do not. Given this simple function, it was very tempting to move this functionality directly into to Image Labeler module. However to ensure a modular and easily extensible design that will scale to filter on several user specifications, the module was not removed.

2.3 Image Labeler FSM

The purpose of the Labeler FSM is to determine the image labeling states of the Image Labeler module. The pixel positions are represented by the memory addresses of the Image buffer. The pixel value at each address location has to be considered for labeling. As the address of the buffer is incremented, the process encounters certain special states. These states are `START, ON_TOP_EDGE`, `ON_LEFT_EDGE`, and `MIDDLE_SCAN`. As the names imply, the top and left edges of the image present special cases for the image labeling algorithm. In the `START` state, the process is acting on the pixel in the top left corner of the image so that the top,left and north-west pixel neighbors cannot have non-zero labels. In the `ON_TOP_EDGE` state, only the left pixel neighbor of the current pixel under consideration can have a non-zero label value. In the `ON_LEFT_EDGE` state, neither the left nor north-west pixel labels can have non-zero label values. In the `MIDDLE_SCAN` state, all of the relevant neighbors of the current pixel may have non-zero values. The scan line buffer used in the Image labeler module uses the no_left_lbl signal of the Image Labeler FSM to reset its address value. Once in the `MIDDLE_SCAN` state, the states oscillate between the `ON_LEFT_EDGE` state and the `MIDDLE_SCAN` state until the the last address contents of the binary image buffer is considered and `label_done` asserts.

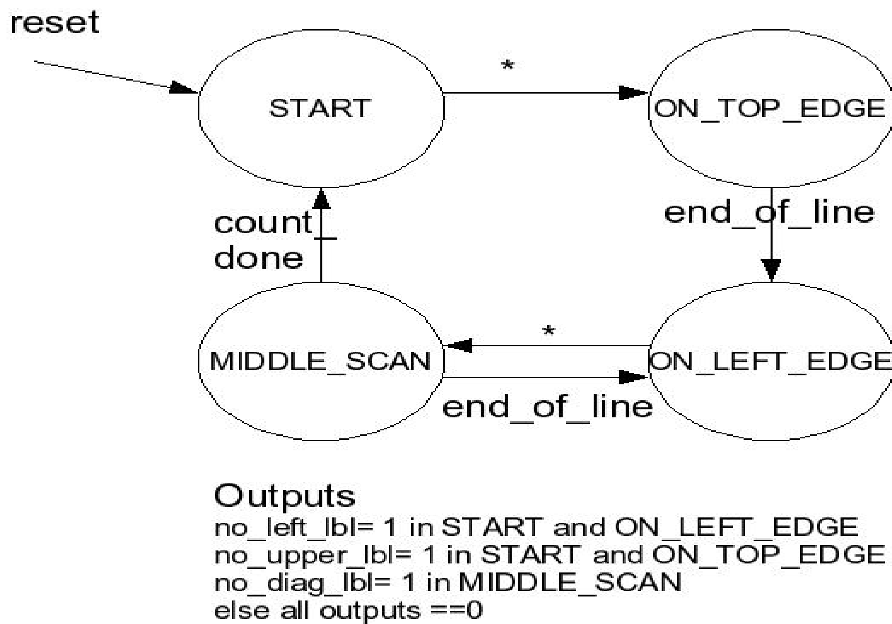

Figure 10: Image Labeler FSM

2.4 Image Labeler

The Labeler module the implements the logic for determining the label to be assigned the current pixel under consideration. It operates on the binary buffer generated by the Binary Image Generator module, using the signals and address locations generated by Image Labeler FSM module. It is used during the image analyses process to assign labels to those pixels in the 'binarized' image that have value 1. It performs the first scan over the image that is described in The image input to this module is a binary image generated by the Binary Image Generator module. On a start_lbl signal from the controller, the labeler, using no_left_lbl,no_top_lbl, and no_diag_lbl signals from the labeler FSM, determines the label to be assigned to the current pixel passed in from memory. The assigned label is recorded in the label table (a mapping of pixels address to pixel label) and/or an equivalence table if necessary. Both of these tables are implemented as memory buffers. In order to remember the location of the pixel to be corrected, the equivalent label is written to the same location in memory as the current pixel address value minus 1. The address of the pixel to be considered is generated by the image labeler FSM module. This implementation uses a scan line buffer to help keep track of the pixel labels. The left pixel label takes on

the old value of the current pixel (curr_pxl_lbl in the previous iteration) The diagonal pixel label takes on the old value of the upper pixel label (upper_pxl_lbl in the previous iteration). The upper pixel label is read from the address location of the scan line buffer which is the arithmetic modulo of the current address location with the width of the image. The curr_pxl_lbl, once computered is written to the scan line buffer at the address location equal to the current pixel address minus 2. The labeling process continues until the Image Labeler FSM module asserts a label_done signal, which is sent to the controller.

2.5 Image Label Correcter

The image label correcter cycles through the equivalence table buffer and and the label table buffer and collects the contents of both buffers. The process begins on a correct_label signal from the controller module. For every label value in the current address of the equivalence table that is non-zero, the value from the equivalence buffer is chosen as the correct pixel label. If the equivalence label value is zero, then the pixel value for the current address location retains its value. Once all the pixel locations have been considered, a correct_done signal is sent the controller.

2.6 Blob Counter

When the count_blob signal is received from the controller, the blob counter performs a very simple operation indeed. It cycles through the corrected label buffer and returns the largest label value so far. This functions is so simple that it was realized that the Image Correcter might well perform this function. Again on the old argument of modularity, it was decided to assign this task to a unique module. A count_done signal is sent to the controller so that the system returns to an idle state when the count is determined.

2.6 VGA

The 640x320 VGA was implemented based on the architecture used in the ROM-to-VGA display sample files from the Fall 2005 6.111 Website. In order to be able to display the processing of the image, dual port memories were used for all the buffers from which the VGA reads from. In order to allow reading from RAM, slight modifications were made to one of the files. The Image Pixel Pointer module was the result.

2.7 Image Pixel Pointer

The Image Pixel Pointer module is used to enable the display of a 320x240 image on a 640x480 VGA. It was coded by a modifying I. Chuang's vg_romdisp.v module from the 6.111 Fall 2005 handouts. This module generates the correct addressing required to position the image pixel on the right location of the screen. The value of the a pixel position is defined by the hcount and vcount signals from the VGA sync module (also from the Fall website). The VGA sync module generates the count and signals necessary for the 640x480 screen resolution using a 50MHz clock.

3.0 Testing and Debugging

Each of he modules was separately tested using either the Modelsim tool, the logic analyzer, the VGA display or their combination. It would have been impossible to test most of the modules with Modelsim except for the modular design of separating the memory buffers from the modules. With the exception of the Image Labeler module which uses an embedded scan line buffer. All the modules were successfully tested separately in Modelsim. The steps used in testing and debugging the most difficult modules is explained below

The VGA output was very crucial in testing the Image Labeler module because it displayed the exact step by step actions in the image labeler module. Unfortunately, though the VGA proved to work perfectly when reading data from the test image ROM, it seemed to fail to read from the buffers (RAMs) that displayed the on-going processing. The image display from the Ram was off and skewed and showed up in a different pattern on every reset. The immediate guess was that the must be a timing issue in another module; that the combinational delay was not meeting the requirements of the frequency of the video clock. But that guess was useless as the most complex computation in the system was an XOR that safely met timing constraints. For a long while, the bug was suspected to be with the Binary Image Generator not releasing the write_enable signals in time for the image pixel pointer modular to handle the addressing. After about a week of trying several suggestions with no results, I resorted to using dual port memories so the image pointer module could constantly read from the buffers. This seemed like the ultimate solution to the problem, but problem persisted in a new form. The display would display correctly for about two thirds of the screen and then skew to the right. It was obvious that this error could not be caused from reading from memory as the write and read ports were defined to be write only and read only. After careful scrutiny, I decided to set the write enable signals to a constant high for image

process modules, which solved the problem much to my surprise. Beyond that point, I was able to see the full processing of the image from start to finish and began attempting to perfect the result on the screen. Alas the break through for this bug occurred a few hours to the deadline and gave me just enough time to integrate the whole system for presentation.

4.0 Reflection

The experience of working on a solo project was filled with mixed feelings. I felt empowered by the fact that I had to understand the details of the entire system in order to make it work. Every hurdle that was crossed brought deep satisfaction to me. Looking back however, it is easy to envision how much more easily I would have identified some of the bugs I run into if I had at least one other per of eyes to look over my code. I remain more convinced of the opinion that two heads are better than one. I wish I had spent more time and effort trying to convince others to join my project(I was not going to give up this wonderful project). The results of the project would have been significantly more impressive if more than one person worked on it.

5.0 Conclusion

Overall, it was a useful experience to master all the aspects of the project and a wonderful one to see the project come very close to a completion. It was very interesting to see the little and yet important bugs that can take several days to master and fix. Seeing the algorithm work to the extent that I was able to implement is was a very rewarding experience. I plan to continue work on this project and make it into a true BACTERIA COLONALYZER.

6.0 Reference

"Mathematical Morphology", Robyn Owens 1997.

7.0 Acknowledgments

I am very highly indebted to all the members of staff who helped debug some of the most mysterious and subtle bugs. My special thanks go to Pvga_romdisp_sample.vrofessor Chris Terman for the encouragement he offered and to Jim Hom for being one of the very few to look through my code.

8.0 Appendices

A Bacteria_Colonalyzer (Top level Module)

```
//
// File:  Bacteria Colonalyzer
// Date:   Fall-2006
// Author: Yaw Anku  <anku@mit.edu>
//
// Top level module for integrating submodules namely Binary Image Generator,
// Controller, Image Labeler FSM, Image Labeler , Image Label Correcter,
// Blob Counter, Image to pixel pointer
 //
//  System Funtionality:
/*  An implementation of a digital raster graphic analyzer that detects the number
 of distinct blobs in a digital image. The name of the project is motivated by
 possible application of the project to detect bacteria colonies on a petri dish
 sample. The design utilizes a sequential scan labeling algorithm to identify the
 pixels of the image that form one entity. The steps involved in this algorithm are
 filtering, initial scan labeling and second scan labeling of the 3200x240 pixel
 input image. The design uses a controller module that manages the transitions
between
 the stages of the image processing implemented by other modules. A 640x320 VGA
display
  is used to show an overlay detailing the pixel by pixel progression of the
algorithm
 at user-variable speeds. Both image processing and the VGA display rely on the
BRAM
 memory in the FPGA. */

///////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
//
///////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock_65mhz, noisy, clean);
```

```
   input reset, clock_65mhz, noisy;
   output clean;//

   reg [19:0] count;
   reg new, clean;

   always @(posedge clock_65mhz)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;

endmodule

///////////////////////////////////////////////////////////////////////////////
// top level module

module bacteria_colonalyzer(
    beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock,

             vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
             vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
             vga_out_vsync,

             tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
             tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
             tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

             tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
             tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
             tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
             tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

             ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
             ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,
```

```
ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

clock_feedback_out, clock_feedback_in,

flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
flash_reset_b, flash_sts, flash_byte_b,

rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

mouse_clock, mouse_data, keyboard_clock, keyboard_data,

clock_27mhz, clock1, clock2,

disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
disp_reset_b, disp_data_in,

button0, button1, button2, button3, button_enter, button_right,
button_left, button_down, button_up,

switch,

led,

user1, user2, user3, user4,

daughtercard,

systemace_data, systemace_address, systemace_ce_b,
systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

analyzer1_data, analyzer1_clock,
analyzer2_data, analyzer2_clock,//
analyzer3_data, analyzer3_clock,
analyzer4_data, analyzer4_clock
     );
```

```
output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
     vga_out_hsync, vga_out_vsync;

output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
     tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
     tv_out_subcar_reset;

input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
     tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
     tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

inout  [35:0] ram1_data;
output [18:0] ram1_address;
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output [3:0] ram1_bwe_b;

input  clock_feedback_in;
output clock_feedback_out;

inout  [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;
```

```
   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   inout  mouse_clo////////////////////////////////////////////////////////////
// Project: Bacteria Colonalyzer (6.111 Final Project)
//    File: controller.v
//  Author: Yaw B. Anku
//    Date:
//
// Functional Description:
/*
  The controller controls the different states and signals
associated with the image processing. It takes in a , the
binary image generator, the image labeler, the image label
corrector, and the blob counter. These signals from the
user and the other modules are used by the controller to
determine the next state transition. The controller specifies
the next state by sending a start signal to the next module
in the process when it receives a done signal from the current
processing module. For example it uses a start signal from the
 user to signal the binary image generator to start processing.
When the binary image processor completes processing, its sends
a done signal which the controller uses as an indicate to send
 a start signal to the labeler
///////////////////////////////////////////////////////////
module   controller(
            //inputs
            clk,
            reset,
            enable,
            start,
            lbl_done,

            lbler_to_lbl_tbl,
            lbler_to_lbl_tbl_we,
            disp_addr,
            lbler_addr,
```

```
         ilc_to_lbl_tbl_we,
         ilc_to_lbl_tbl,

         correct_lbl_done,
         correcter_addr,
         count_done,
         bc_lbl_tbl_raddr,

         //outputs

         lbl_tbl_raddr,
         lbl_tbl_waddr,
         to_lbl_tbl,
         lbl_tbl_we,
         start_label,
         start_correcting,
         start_count,
         state
         );

//CONSTANTS
parameter DATA_WIDTH=4;
parameter ADDR_WIDTH = 17; //buffer address width

input clk; //global clk
input reset; //global reset
input start; //start user signal
input enable;
input lbl_done; //from image labeler
   input count_done;
input correct_lbl_done; //from image lbl correcter
   input [DATA_WIDTH-1:0] lbler_to_lbl_tbl,ilc_to_lbl_tbl;

   input [ADDR_WIDTH-1:0] lbler_addr,bc_lbl_tbl_raddr,
                          correcter_addr,disp_addr;
input lbler_to_lbl_tbl_we,ilc_to_lbl_tbl_we;
```

```
 output start_label; //to image labeler fsm
 output start_correcting; //to image lbl corrector
 output start_count; //to blob counter
    output [DATA_WIDTH-1:0]   to_lbl_tbl;
 // we signals to buffers
// output bin_buff_we;//
 output lbl_tbl_we;//,eqv_tbl_we;

 //buffer addresses
 output [ADDR_WIDTH-1:0] lbl_tbl_raddr,lbl_tbl_waddr; //bin_buff_addr

 //states: represent substages in processing of image
 parameter IDLE =0;

 parameter LABEL=2;
 parameter CORRECT_LABEL=3;
 parameter COUNT_LABEL = 4;

 output reg [2:0] state;
 //COMBINATIONAL LOGIC
 //OUTPUT Signals for various states
// assign bin_start = (state==BINARIZE);
 assign start_label = (state ==LABEL);
 assign start_correcting = (state == CORRECT_LABEL);
 assign start_count = (state == COUNT_LABEL);

 wire start_pulse,lbl_done_pulse,correct_lbl_done_pulse,
      count_lbl_done_pulse;
 level_to_pulse p0(clk, start,start_pulse);
 level_to_pulse p1(clk, lbl_done,lbl_done_pulse);
 level_to_pulse p2(clk, correct_lbl_done,correct_lbl_done_pulse);
 level_to_pulse p3(clk, count_lbl_done,count_lbl_done_pulse);

 assign lbl_tbl_raddr = (state==COUNT_LABEL) ?
                  bc_lbl_tbl_raddr:disp_addr;

 assign lbl_tbl_waddr = (state == LABEL) ?
```

```
                        lbler_addr:correcter_addr;

   assign lbl_tbl_we = (state == LABEL) ? lbler_to_lbl_tbl_we:
                ilc_to_lbl_tbl_we;

   assign to_lbl_tbl = (state==LABEL) ?
                lbler_to_lbl_tbl:
                ilc_to_lbl_tbl;




   //SEQUENTIAL LOGIC
   always @ (posedge clk)
     begin
      if(reset)
        state <= IDLE;
      else
        case (state)
          IDLE : state <= start_pulse ? LABEL :state;//BINARIZE : state;
         // BINARIZE: state <= bin_done ? LABEL : state;
          LABEL: state <= lbl_done_pulse ? CORRECT_LABEL : state;
          CORRECT_LABEL: state <= correct_lbl_done_pulse ? COUNT_LABEL : state;
          COUNT_LABEL: state <= count_lbl_done_pulse ? IDLE : state;
          default: state <= state;
        endcase // case(state)
     end // always @ (posedge clk)

endmodule // controllerck, mouse_data;
   input  keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
```

```
   output  disp_data_out;

   input  button0, button1, button2, button3, button_enter, button_right,
        button_left, button_down, button_up;
   input  [7:0] switch;
   output [7:0] led;

   inout [31:0] user1, user2, user3, user4;

   inout [43:0] daughtercard;

   inout  [15:0] systemace_data;
   output [6:0]  systemace_address;
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
           analyzer4_data;
   output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

   ////////////////////////////////////////////////////////////////////////////
   //
   // I/O Assignments
   //
   ////////////////////////////////////////////////////////////////////////////

   // Audio Input and Output
   assign beep= 1'b0;
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;

   // ac97_sdata_in is an input

   // VGA Output
//   assign vga_out_red = 10'h0;
//   assign vga_out_green = 10'h0;
```

```
//   assign vga_out_blue = 10'h0;
   assign vga_out_sync_b = 1'b1;
//   assign vga_out_blank_b = 1'b1;
//   assign vga_out_pixel_clock = 1'b0;
//   assign vga_out_hsync = 1'b0;
//   assign vga_out_vsync = 1'b0;

   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

   // Video Input
   assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b0;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b0;
   assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = 1'b0;
   assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_clk = 1'b0;
   assign ram0_cen_b = 1'b1;
   assign ram0_ce_b = 1'b1;
```

```
 assign ram0_oe_b = 1'b1;
 assign ram0_we_b = 1'b1;
 assign ram0_bwe_b = 4'hF;
 assign ram1_data = 36'hZ;
 assign ram1_address = 19'h0;
 assign ram1_adv_ld = 1'b0;
 assign ram1_clk = 1'b0;
 assign ram1_cen_b = 1'b1;
 assign ram1_ce_b = 1'b1;
 assign ram1_oe_b = 1'b1;
 assign ram1_we_b = 1'b1;
 assign ram1_bwe_b = 4'hF;
 assign clock_feedback_out = 1'b0;
 // clock_feedback_in is an input

 // Flash ROM
 assign flash_data = 16'hZ;
 assign flash_address = 24'h0;
 assign flash_ce_b = 1'b1;
 assign flash_oe_b = 1'b1;
 assign flash_we_b = 1'b1;
 assign flash_reset_b = 1'b0;
 assign flash_byte_b = 1'b1;
 // flash_sts is an input

 // RS-232 Interface
 assign rs232_txd = 1'b1;
 assign rs232_rts = 1'b1;
 // rs232_rxd and rs232_cts are inputs

 // PS/2 Ports
 // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

 // LED Displays
/* assign disp_blank = 1'b1;
 assign disp_clock = 1'b0;
 assign disp_rs = 1'b0;
```

```
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0; */
/*
 */
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
//lab3 assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
//assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs

// Logic Analyzer
//assign analyzer1_data = 16'h0;
//assign analyzer1_clock = 1'b1;
//assign analyzer2_data = 16'h0;
//assign analyzer2_clock = 1'b1;
assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
// assign analyzer4_data = 16'h0;
```

```
// assign analyzer4_clock = 1'b1;



// use FPGA's digital clock manager to produce a 50 Mhz clock from 27 Mhz
// actual frequency: 49.85 MHz
wire clock_50mhz_unbuf,clock_50MHz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_50mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 13
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_50MHz),.I(clock_50mhz_unbuf));

////////////////////////////////////////
// The ps2 mouse test: display 640x480 screen with mouse controlled cursor

wire   clk = clock_50MHz;

wire power_on_reset;
SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
            .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

wire user_reset;
debounce dbreset(1'b0,clk,~button_enter,user_reset);



wire show_res;
   wire [1:0] f;
 debounce dbf1(user_reset,clk,switch[6],f[1]);
    debounce dbf2(user_reset,clk,switch[5],f[0]);
debounce dbshow_res(user_reset,clk,switch[7],show_res);

wire reset = power_on_reset | user_reset;

// 640x480 VGA display
wire [7:0] pixel;
```

```
   wire       blank;
   wire       pix_clk;
   wire [9:0] hcount, vcount;
   assign     vga_out_red ={8{pixel[0]}}; // black and white display (for now)
   assign     vga_out_green = {8{pixel[1]}};
   assign     vga_out_blue = {8{pixel[2]}};
   assign     vga_out_blank_b = ~blank;
   assign     vga_out_pixel_clock = pix_clk;    // vga pixel clock

   vga_sync vga1(clk,vga_out_hsync,vga_out_vsync,hcount,vcount,pix_clk,blank);

   // vga image display
   wire [3:0]  rom_data;
      wire [1:0]  rom_data2;
   wire [7:0] rpix;
   wire [16:0] disp_addr;
   wire [3:0] from_lbl_tbl,test_img_dout,test_img_dout2, from_res_tbl;
      wire bin_disp_data;
   assign      rpix = show_res ? {4'b0,from_lbl_tbl} | {4'b0,from_res_tbl} :
test_img_dout2;///{4'b0,bin_disp_data}; /*show_res ?
from_lbl_tbl[7:0];//:*///{6'b0,rom_data2};

   img_pxl_pointer img_src_ptr(clk,hcount,vcount,pix_clk,disp_addr);

   // video output

   wire       fpixel = (hcount==0 | hcount==639 | vcount==0 | vcount==479);
   assign     pixel = blank ? 8'b0
                  : ( switch[0] ? {hcount[7:5],5'b0}
                    : rpix | (fpixel ? 8'hff : 8'h00) );

      //DIVIDER
            wire sup,sdown,enable,start;
   debounce dstart(user_reset,clk,~button0, start);

   debounce dup(user_reset, clk, ~button_up, sup);
   debounce ddown(user_reset, clk, ~button_down, s_down);
      //module divider(clk,sup,sdown,reset,one_hz_enable);
```

```
   divider div(clk,sup,sdown,user_reset,enable);

//WIRES
//signals
wire lbl_start,no_left_lbl,no_upper_lbl,
   no_diag_lbl,lbl_done,
   count_done,correct_lbls,correct_lbl_done,
   eqv_tbl_we,lbl_tbl_we,count_blobs,
   lbler_to_lbl_tbl_we,lbler_to_eqv_tbl_we,
   ilc_to_lbl_tbl_we;
//addresses
wire [16:0] curr_img_pxl_addr,eqv_tbl_waddr,
          eqv_tbl_raddr,
          test_img_raddr,lbler_fsm_addr,
          lbl_tbl_raddr,lbl_chnge_addr,
          lbler_to_eqv_tbl_waddr,
          ilc_to_lbl_tbl_waddr, lbl_tbl_waddr,
          bc_lbl_tbl_raddr;

//data
wire [3:0]  to_lbl_tbl,to_eqv_tbl, from_eqv_tbl,


                           from_match_tbl,
          lbl_data_in,
            lbler_to_lbl_tbl,lbler_to_eqv_tbl,
          lbler_data_in,from_eqv_tbl_to_ilc,
          from_lbl_tbl_to_ilc,ilc_to_lbl_tbl,
          from_lbl_tbl_to_bc;
    wire [3:0] to_res_tbl;

                wire res_tbl_we;
 wire [2:0] state;
//CONTROLLER
/*controller c(
         //inputs
         .clk(clk),
         .reset(user_reset),
```

```
        .enable(enable),
        .start(start),
        .lbl_done(lbl_done),

        .disp_addr(disp_addr),
        .lbler_to_lbl_tbl(lbler_to_lbl_tbl),
        .lbler_to_lbl_tbl_we(lbler_to_lbl_tbl_we),
        .lbler_addr(lbler_fsm_addr),
        .ilc_to_lbl_tbl_we(ilc_to_lbl_tbl_we),
        .ilc_to_lbl_tbl(ilc_to_lbl_tbl),
        .correct_lbl_done(correct_lbl_done),
        .count_done(count_done),
        .bc_lbl_tbl_raddr(bc_lbl_tbl_raddr),

        //outputs

        .lbl_tbl_raddr(lbl_tbl_raddr),
        .lbl_tbl_waddr(lbl_tbl_waddr),
        .to_lbl_tbl(to_lbl_tbl),
        .lbl_tbl_we(lbl_tbl_we),
        .start_label(lbl_start),
        .start_correcting(correct_lbls),
        .start_count(count_start),
        .state(state)
        );
         */

//BUFFERS
//TEST IMAGE FROM ROM
  test_img_rom test_img2(.clk(clk),
               .addr(disp_addr),
               .dout(test_img_dout2)
               );
test_img_rom test_img(.clk(clk),
               .addr(test_img_raddr),
               .dout(test_img_dout)
               );
```

```
            //wire lbler_done;
            //wire [16:0] lbl_buff_waddr,correcter_waddr;
            //assign lbl_buff_waddr = lbler_done ? correcter_waddr :
test_img_raddr;
   //LABEL TABLE
   dp76800x4 lbl_buff(.addra(disp_addr),//lbl_tbl_raddr),
                  .addrb(test_img_raddr),
                  .clka(clk),
                  .clkb(clk),
                  .dinb(to_lbl_tbl),
                  .douta(from_lbl_tbl),
                  .web(lbl_tbl_we));
   //EQUIVALENCT TABLE
   dp76800x4 eqv_buff(.addra(lbl_chnge_addr),
                  .addrb(eqv_tbl_waddr),
                  .clka(clk),
                  .clkb(clk),
                  .dinb(to_eqv_tbl),
                  .douta(from_eqv_tbl),
                  .web(eqv_tbl_we));

       //LBL_TBL MATCH
       wire [16:0] rand_addr;
   dp76800x4 match_t(.addra(rand_addr),//lbl_chnge_addr),
                  .addrb(lbl_chnge_addr),
                  .clka(clk),
                  .clkb(clk),
                  .dinb(to_lbl_tbl),
                  .douta(from_match_tbl),
                  .web(lbl_tbl_we));

             //RES_TBL

   dp76800x4 res_t(.addra(disp_addr),//lbl_chnge_addr),
                  .addrb(test_img_raddr),
                  .clka(clk),
                  .clkb(clk),
                  .dinb(to_res_tbl),
```

```
                  .douta(from_res_tbl),
                  .web(res_tbl_we));
/*image_labeler_fsm ilf(
                  //inputs
                  .clk(clk),
                  .reset(user),
                  .enable(enable),
                  .lbl_start(start),//lbl_start),

                  //outputs
                  .no_left_lbl(no_left_lbl),
                  .no_upper_lbl(no_upper_lbl),
                  .no_diag_lbl(no_diag_lbl),
                  .curr_pxl_addr(test_img_raddr),
                  .lbl_done(lbl_done)); */

    //wire lbler_we,correcter_we;
      // assign lbl_tbl_we =lbler_done ?  correcter_we : lbler_we ;
       //wire [3:0] correcter_to_lbl_tbl;
      // assign to_lbl_tble = lbler_done ? correcter_to_lbl_tbl: lbler_to_lbl_tbl;
      wire [3:0] filter_val,upper_pxl_lbl,left_pxl_lbl,diag_pxl_lbl;
      assign filter_val =  f[1] == f[0] ?  4'hF: f[1] ? 4'hC :4'h9 ;
   labeler labeler0(
             //inputs
             .reset(user_reset),
             .clk(clk),
                   .start(start),
             .enable(enable),
             .filter_val(filter_val),
          .curr_img_pxl(test_img_dout),
             .curr_pxl_addr(test_img_raddr),
             //.no_upper_lbl(no_upper_lbl),
             //.no_left_lbl(no_left_lbl),
             //.no_diag_lbl(no_diag_lbl),

             //outputs
             .curr_pxl_lbl(to_lbl_tbl),//lbler_to_lbl_tbl),
```

```
          .to_eqv_tbl(to_eqv_tbl),
          .lbl_tbl_we(lbl_tbl_we),
          .eqv_tbl_we(eqv_tbl_we),
          .eqv_tbl_waddr(eqv_tbl_waddr),
                .done(lbl_done),
                .upper_pxl_lbl(upper_pxl_lbl),
                .left_pxl_lbl(left_pxl_lbl),
                .diag_pxl_lbl(diag_pxl_lbl)
          );

/*reg crrct_trig;
   always @ (posedge enable)
     begin
     crrct_trig <= lbler_done ? crrct_trig +1;
     end*/

     //assign correct_lbls =    lbler_done;
     /*module img_lbl_corrector(clk,correct_lbls,enable,
                reset,from_eqv_tbl,from_match_tbl,to_res_tbl,
                lbl_chnge_addr,
                res_tbl_we,
                correct_lbl_done);*/
   assign correct_lbls = lbl_done;
   wire [3:0] blob_count;
img_lbl_corrector ilc(
                //inputs
                .clk(clk),
                .reset(user_reset),
                .enable(enable),
                .correct_lbls(correct_lbls),
                .from_eqv_tbl(from_eqv_tbl),
                .from_match_tbl(from_match_tbl),
                //outputs
                .blob_count(blob_count),
                .to_res_tbl(to_res_tbl),//ilc_to_lbl_tbl),
                .lbl_chnge_addr(lbl_chnge_waddr),
                .res_tbl_we(res_tbl_we),//ilc_to_lbl_tbl_we),
```

```
                  .correct_lbl_done(correct_lbl_done));
   assign count_blobs = correct_lbl_done;
  blob_counter bc(.clk(clk),
             .reset(user_reset),
             .count_blobs(count_blobs),
             .lbl_data_in(blob_count),

             //outputs
             .disp_blank(disp_blank),
             .disp_clock(disp_clock),
             .disp_rs(disp_rs),
             .disp_ce_b(disp_ce_b),
             .disp_reset_b(disp_reset_b),
             .disp_data_out(disp_data_out),
             .count_done(count_done),
             .lbl_tbl_raddr(bc_lbl_tbl_raddr)
             );



//debugging
   assign        led = ~
{start,lbl_done,lbl_done,correct_lbl_done,count_done,state};//,
   // start_lbl,lbl_done,lbl_tbl_we,pixel[2:0]};


   //my testing
   assign        analyzer1_clock = clk;//clock_27mhz;
   assign        analyzer2_clock = clk;//ock_27mhz;
   assign        analyzer4_clock = clk;//ock_27mhz;


   assign analyzer1_data = test_img_raddr[15:0];//
{rpix[7:0],we,2'b0,lbl_tbl_we,from_lbl_tbl[3:0]};//rom_data;//lbler_addr[15:0];


   assign analyzer4_data = 0;//to_ram;//{show_res,lbl_tbl_we,lbl_done,
                      //no_diag_lbl,no_upper_lbl,no_left_lbl,
                      //start_lbl,
                     //8'b0};// to_lbl_tbl[8:0]};
   assign analyzer2_data =
{upper_pxl_lbl,diag_pxl_lbl,to_lbl_tbl,left_pxl_lbl};//{3'b0,from_lbl_tbl,test_img_
dout,{4{bin_disp_data}}};//from_lbl_tbl;//to_lbl_tbl;
```

```
  //assign analyzer2_data = {8'b0,rpix}  ;
```

B Controller

```
//////////////////////////////////////////////////////////
// Project: Bacteria Colonalyzer (6.111 Final Project)
//    File: controller.v
//  Author: Yaw B. Anku
//    Date:
//
// Functional Description:
/*
  The controller controls the different states and signals
associated with the image processing. It takes in a , the
binary image generator, the image labeler, the image label
corrector, and the blob counter. These signals from the
user and the other modules are used by the controller to
determine the next state transition. The controller specifies
the next state by sending a start signal to the next module
in the process when it receives a done signal from the current
processing module. For example it uses a start signal from the
 user to signal the binary image generator to start processing.
When the binary image processor completes processing, its sends
a done signal which the controller uses as an indicate to send
 a start signal to the labeler
///////////////////////////////////////////////////////////
module   controller(
            //inputs
            clk,
            reset,
            enable,
            start,
            lbl_done,

            lbler_to_lbl_tbl,
            lbler_to_lbl_tbl_we,
            disp_addr,
            lbler_addr,
```

```
        ilc_to_lbl_tbl_we,
        ilc_to_lbl_tbl,

        correct_lbl_done,
        correcter_addr,
        count_done,
        bc_lbl_tbl_raddr,

        //outputs

        lbl_tbl_raddr,
        lbl_tbl_waddr,
        to_lbl_tbl,
        lbl_tbl_we,
        start_label,
        start_correcting,
        start_count,
        state
        );

//CONSTANTS
parameter DATA_WIDTH=4;
parameter ADDR_WIDTH = 17; //buffer address width

input clk; //global clk
input reset; //global reset
input start; //start user signal
input enable;
input lbl_done; //from image labeler
   input count_done;
input correct_lbl_done; //from image lbl correcter
   input [DATA_WIDTH-1:0] lbler_to_lbl_tbl,ilc_to_lbl_tbl;

   input [ADDR_WIDTH-1:0] lbler_addr,bc_lbl_tbl_raddr,
                          correcter_addr,disp_addr;
input lbler_to_lbl_tbl_we,ilc_to_lbl_tbl_we;
output start_label; //to image labeler fsm
```

```
 output start_correcting; //to image lbl corrector
 output start_count; //to blob counter
    output [DATA_WIDTH-1:0]   to_lbl_tbl;
 // we signals to buffers
// output bin_buff_we;//
 output lbl_tbl_we;//,eqv_tbl_we;

 //buffer addresses
 output [ADDR_WIDTH-1:0] lbl_tbl_raddr,lbl_tbl_waddr; //bin_buff_addr

 //states: represent substages in processing of image
 parameter IDLE =0;

 parameter LABEL=2;
 parameter CORRECT_LABEL=3;
 parameter COUNT_LABEL = 4;

 output reg [2:0] state;
 //COMBINATIONAL LOGIC
 //OUTPUT Signals for various states
// assign bin_start = (state==BINARIZE);
 assign start_label = (state ==LABEL);
 assign start_correcting = (state == CORRECT_LABEL);
 assign start_count = (state == COUNT_LABEL);

 wire start_pulse,lbl_done_pulse,correct_lbl_done_pulse,
      count_lbl_done_pulse;
 level_to_pulse p0(clk, start,start_pulse);
 level_to_pulse p1(clk, lbl_done,lbl_done_pulse);
 level_to_pulse p2(clk, correct_lbl_done,correct_lbl_done_pulse);
 level_to_pulse p3(clk, count_lbl_done,count_lbl_done_pulse);

 assign lbl_tbl_raddr = (state==COUNT_LABEL) ?
                  bc_lbl_tbl_raddr:disp_addr;

 assign lbl_tbl_waddr = (state == LABEL) ?
                       lbler_addr:correcter_addr;
```

```
   assign lbl_tbl_we = (state == LABEL) ? lbler_to_lbl_tbl_we:
                ilc_to_lbl_tbl_we;

   assign to_lbl_tbl = (state==LABEL) ?
                lbler_to_lbl_tbl:
                ilc_to_lbl_tbl;







   //SEQUENTIAL LOGIC
   always @ (posedge clk)
     begin
      if(reset)
        state <= IDLE;
      else
        case (state)
          IDLE : state <= start_pulse ? LABEL :state;//BINARIZE : state;
         // BINARIZE: state <= bin_done ? LABEL : state;
          LABEL: state <= lbl_done_pulse ? CORRECT_LABEL : state;
          CORRECT_LABEL: state <= correct_lbl_done_pulse ? COUNT_LABEL : state;
          COUNT_LABEL: state <= count_lbl_done_pulse ? IDLE : state;
          default: state <= state;
        endcase // case(state)
     end // always @ (posedge clk)

endmodule // controller
```

C Binary Image Generator

```
/* Yaw Anku
   Bacterial Colonalyzer project
   File : Binary Image generator
   The Binary Image Generator implements the filtering
stage of the image analyses process. It loads up the input
image pixel by pixel from the test image buffer and filters
```

```
 out pixels that do not meet the filter threshold. The
threshold is specified by user input through switches 6
and 0 of the FPGA. It writes a value of 1 for pixels that
meet the threshold and a value of zero for pixels that do not.
Given this simple function, it was very tempting to move this
functionality directly into to Image Labeler module.
However  to ensure a modular and easily extensible d
esign that will scale to filter on several user specifications,
the module was not removed.
 */
module binary_image_generator(reset,clk,bin_start,pxl_addr,
img_data_in,bin_data,enable,bin_done,bin_buff_we);
   parameter IMG_DATA_WIDTH = 4;
   parameter ADDR_WIDTH = 17;
   parameter NUM_PXLS=76800;

   input reset;
   input clk;
   input bin_start;
   input [IMG_DATA_WIDTH-1:0] img_data_in;
   input                enable;

   output reg [ADDR_WIDTH-1:0] pxl_addr;

   output bin_data;
   output bin_done;
   output reg bin_buff_we;
   assign bin_data =(img_data_in != 4'hF);//or use < 4'hF instead
   assign bin_done = pxl_addr == NUM_PXLS - 1;

   wire   start_bin_pls; //initial start

   level_to_pulse start_bin_pulse(clk,bin_start,start_bin_pls);

   always @ (posedge clk)
     begin
        if(reset || start_bin_pls)
```

```
        begin
           pxl_addr<=0;
        end
     else  if(~bin_done)
          if (enable)begin
       pxl_addr <= pxl_addr + 1;
       bin_buff_we <=1;
             end
             else
             bin_buff_we <=0;
     end // always @ (posedge clk)
endmodule // binary_image_generator
```

D Image Labeler FSM

```
////////////////////////////////////////////////////////////
// Project: Bacteria Colonalyzer (6.111 Final Project)
//    File:
//  Author: Yaw B. Anku
//    Date:
//
// Functional Description:
// The purpose of the Labeler FSM is to determine the image labeling states
// of the Image Labeler module. The pixel positions are represented by
// the memory addresses of the Image buffer. The pixel value at each address
// location has to be considered for labeling. As the address of the buffer
// is incremented, the process encounters certain special states.
// The top and left edges of the image present special cases for the image
// labeling algorithm. In the START state, the process is acting on the pixel
// in the top left corner of the image so that the top,left and north-west
// pixel neighbors cannot have non-zero labels. In the ON_TOP_EDGE state,
// only the left pixel neighbor of the current pixel under consideration can
// have a non-zero label value. In the ON_LEFT_EDGE state, neither the left
// nor north-west pixel labels can have non-zero label values.
// In the MIDDLE_SCAN state, all of the relevant neighbors of the current
// pixel may have non-zero values. A scan line buffer used in the Image
// labeler module uses the no_left_lbl signal of the Image Labeler FSM to
// reset its address value.
//
```

```
/////////////////////////////////////////////////////////////
module image_labeler_fsm(
                   //inputs
                   clk,
                   reset,
                   enable,
                   lbl_start,

                   //outputs
                   no_left_lbl,
                   no_upper_lbl,
                   no_diag_lbl,
                   curr_pxl_addr,
                   lbl_done,

                   state);

   //constansts
   parameter PXL_ADDR_WIDTH=17;
   parameter NUM_PXLS= 500;//76800;
   parameter IMG_WIDTH = 10;//320;

   //scanning states
   parameter START =0;  //1st pxl position => no pxls up or left.
   parameter ON_TOP_EDGE = 1; //=> no pxls up/above
   parameter ON_LEFT_EDGE = 2; //=> no pxls on left
   parameter MIDDLE_SCAN = 3; //includes when on last line

   input clk;              //global clk
   input reset;            //global reset
   input enable;         // 1 to increment addr,else 0
   input lbl_start;      //signal from controller to start labeling

   //address location of pxl under consideration
   output reg [PXL_ADDR_WIDTH-1:0] curr_pxl_addr;

   //outputs indicating the pixel positon.
```

```
//based on states specified above
output no_left_lbl;
output no_upper_lbl;
output no_diag_lbl;
output lbl_done;

output reg [2:0] state;
   reg [9:0] count;
//COMBINATIONAL LOGIC for outputs
wire end_of_line;      //indicates at right edge of image
assign end_of_line = (count == IMG_WIDTH - 1);

//assigning outputs depending on states
assign no_left_lbl = ((state == START) ||
                (state == ON_LEFT_EDGE));

assign no_upper_lbl = ((state == START) ||
                 (state == ON_TOP_EDGE));

assign no_diag_lbl = ((state == ON_LEFT_EDGE) ||
                (state == ON_TOP_EDGE));

assign lbl_done = (curr_pxl_addr >= NUM_PXLS-1);

wire   start_lbl_pulse; //high for one clk cycle after
                        //start_lbl is high
//generate start pulse
level_to_pulse pulse0(clk,lbl_start,start_lbl_pulse);


//SEQUENTIAL LOGIC
always @ (posedge clk)
  begin
   if(reset || start_lbl_pulse)
     begin
        state <= START;
        curr_pxl_addr <=0;
```

```
        end
     if(reset || start_lbl_pulse || end_of_line) count <=0    ;
        else if(~reset && ~lbl_done)
          begin
             //if(enable)
              // begin
                curr_pxl_addr <=  curr_pxl_addr + 1;
                count <= count +1;
                //the order of cycling is start -> on-top-edge ->
                //on_left_edge and then oscillation between
                //on_left_edge and middle_scan until all pxls
                //have been examined ie.curr_pxl_addr==NUM_PXLS-1
                case (state)
                  START : state <= ON_TOP_EDGE;
                  ON_TOP_EDGE : state <= end_of_line ?
                                   ON_LEFT_EDGE : state;
                  ON_LEFT_EDGE: state <= MIDDLE_SCAN;
                  MIDDLE_SCAN: state <= end_of_line ?
                                   ON_LEFT_EDGE : state;
                  default: state <= state;
                endcase // case(state)
               //end
          end
       end // always @ (posedge clk)

endmodule // image_labeler_fsm
```

E Image Labeler

```
////////////////////////////////////////////////////////////////////////////////
// Project: Bacteria Colonalyzer (6.111 Final Project)
//    File: image_labeler.v
//  Author: Yaw B. Anku
//    Date: November 28, 2006
//
//
/*
```

```
 * Functional Description:
 * This file is an implementation of the first scan of the
 * left-skewed 6-connectedness neighborhood scheme(a two
 * scan algorithm) for identifying blobs in an image. It
 * scans through the image using a typical raster scan,
 * row by row, top to bottom, left to right. The image input
 * to this module is a binary image from memory. For any
 * particular pixel/cell,C we know that the cell to its left,L,
 * the cell directly above, U, and the cell above L,D have
 * already been labeled.See the illustration below
 *     --- ---
 *    | D | U |
 *     --- --- ---
 *    | L | C |   | -->direction of scan
 *     --- --- ---
 *        |   |   |
 *         --- ---
 * We determine the label of the current
 * pixel based on the labels of the surrounding pixels
 * The algorithm(pseudocode) is as follows
 * (P is the pixel value of image).

if P = O
 do nothing
else if D labeled
     copy label to C

else if (not L labeled) and (not U labeled)
     increment label numbering and label C

else if L xor U labeled
     copy label to C

else if L and U labeled
     if L label = U label
          copy label to C
     else
```

```
          copy U label to C
          record equivalence of label L to U

 * The special cases in the algorithm are when the pixel
 * being considered is on the top or left edge of the image
 * That is L,U and D (if in top left corner) will have zero
 * label values. The Image Labeler FSM module generates signals
 * to handle these special cases.
 *
 * On a start_lbl signal from the controller, the labeler, using
 * no_left_lbl,no_top_lbl, and no_diag_lbl signals from the
 * labeler FSM, determines the label to be assigned to the current
 * binary pixel passed in from memory. The assigned label is
 * recorded in the label table (a mapping of pixels address to
 * pixel label) and/or an equivalence table if necessary. The
 * address of the pixel to be considered is generated by the
 * image laber FSM module.
 *
 * This implementation uses a scanline buffer to help keep
 * track of the pixel labels. See code comments
 *
 */
//
/////////////////////////////////////////////////////////////
module labeler(
             //inputs
             reset,
             clk,
             enable,
             curr_img_pxl,
             curr_img_pxl_addr,
             no_upper_lbl,
             no_left_lbl,
             no_diag_lbl,

             //outputs
             to_lbl_tbl,
```

```
          to_eqv_tbl,
          lbl_tbl_we,
          eqv_tbl_we,
          eqv_tbl_waddr
          );

//constants
parameter TBL_ADDR_WIDTH = 17;
parameter SCAN_LINE_ADDR_WIDTH = 9;
parameter NUM_IMG_PXLS = 76800;
parameter LBL_DATA_WIDTH = 4;
parameter IMAGE_WIDTH=320;

input reset; //global reset
input clk; //clk
input enable;
input start_lbl; //from controller 1 to start,else 0
input [3:0] curr_img_pxl; // from (binary) img buffer
input [TBL_ADDR_WIDTH-1:0] curr_img_pxl_addr; //pxl addr from fsm
input no_upper_lbl; //from FSM 1 => on top edge of img,else 0
input no_left_lbl; //from FSM 1=> on left edfe of img,else 0
input no_diag_lbl; //from FSM 1=> top left corner,else 0

//outputs to label table and Eqv table
output reg lbl_tbl_we;
output reg eqv_tbl_we;
output [LBL_DATA_WIDTH-1:0] to_lbl_tbl;
output reg [LBL_DATA-WIDTH-1:0] to_eqv_tbl;
output reg [TBL_ADDR_WIDTH-1:0]    eqv_tbl_waddr;

//scanline addr widths
wire [SCAN_LINE_ADDR_WIDTH-1:0] lbl_scanline_raddr; //
reg [SCAN_LINE_ADDR_WIDTH-1:0]  lbl_scanline_waddr;

reg [LBL_DATA_WIDTH-1:0] new_lbl; //next lbl to be assigned

//variables for current and surrounding pxl labels
```

```
reg [LBL_DATA_WIDTH-1:0] curr_pxl_lbl,upper_pxl_lbl,
                   diag_pxl_lbl,left_pxl_lbl;

   //module dpram320x4(addra,
   //addrb,
   //clka,
   //clkb,
   //dina,
   //doutb,
   //wea);
wire [LBL_DATA_WIDTH-1:0] lbl_scanline_rdata;
wire [LBL_DATA_WIDTH-1:0] lbl_scanline_wdata;

reg lbl_scanline_we;

//dual port ram write after read
dpram320x4
  lbl_scan_line(.addra(lbl_scanline_waddr),
            .addrb(lbl_scanline_raddr),
            .clka(clk),
            .clkb(clk),
            .dina(lbl_scanline_wdata),
            .doutb(lbl_scanline_rdata),
            .wea(lbl_scanline_we)
            );

//COMBINATIONAL LOGIC
assign to_lbl_tbl = curr_pxl_lbl;
assign lbl_scanline_wdata = no_left_lbl ? 0 : left_pxl_lbl;
assign more_pxls = (curr_img_pxl_addr < NUM_IMG_PXLS-1);
assign lbl_scanline_raddr = no_left_lbl ? 0 : curr_img_pxl_addr;


//SEQUENTIAL LOGIC
always @ (posedge clk)
  begin
   if(reset)
```

```
        begin
           new_lbl <=1;//labeling starts from 1
        end
      else
        begin
           if(more_pxls)
             begin
              if (enable)
                begin
                   lbl_tbl_we <=1;
                   lbl_scanline_we <=1;
                   lbl_scanline_waddr <=(curr_pxl_addr==IMAGE_WIDTH-1)? 0
:curr_pxl_addr-1;

                   //upper_pxl_lbl is the rdata from scanline
                   //if not on top edge of image
                   upper_pxl_lbl <= no_upper_lbl ?
                           0 : lbl_scanline_rdata;
                   //left_pxl_lbl what was curr_pxl_lbl
                   //if not on left edge of image
                   left_pxl_lbl <= no_left_lbl ?
                               0 : curr_pxl_lbl;
                   //diag_pxl_lbl==what was upper_pxl
                   //if not in top left corner of image
                   diag_pxl_lbl <= no_diag_lbl ?
                               0 :upper_pxl_lbl;

                   //the algorithm code
                   /*     --- ---
                  *    | D | U |
                  *     --- --- ---
                  *    | L | C |   | -->direction of scan
                  *     --- --- ---
                  *        |   |   |
                  *         --- ---
                  */
                   if(curr_img_pxl==4'hF)
                   //do nothing
```

```
curr_pxl_lbl<=0;

//NOTE AN label value==0 <=>  no label
//else if D labeled
//   copy label to C
else if (diag_pxl_lbl>0)
curr_pxl_lbl <=diag_pxl_lbl;

//else if (not L labeled) and (not U labeled)
//increment label numbering and label C
else if ((left_pxl_lbl==0) && (upper_pxl_lbl==0))
begin
   //note order does not matter since non-blocking
   curr_pxl_lbl <=new_lbl;
   new_lbl <= new_lbl + 1;
end
//else if L xor U labeled
//copy label to C
else if ((left_pxl_lbl>0)^(upper_pxl_lbl>0))
curr_pxl_lbl <= left_pxl_lbl | upper_pxl_lbl;

//else if L and U labeled
if ((left_pxl_lbl>0) && (upper_pxl_lbl>0))
begin
   //if L label = U label
   // copy label to A
   if(left_pxl_lbl==upper_pxl_lbl)
     curr_pxl_lbl<=upper_pxl_lbl;

   else
     //copy either U label to C
     //record equivalence of labels, L=U;
     begin
      curr_pxl_lbl<=upper_pxl_lbl;
      //eqv_tbl assigns upper_pxl_lbl to left_pxl_lbl
      eqv_tbl_we <=1;
      eqv_tbl_waddr <= curr_pxl_addr - 1; //pxl to left check
for error
```

```
                        to_eqv_tbl <= upper_pxl_lbl;
                       end // else: !if(left_pxl_lbl==upper_pxl_lbl)
                  end // if ((left_pxl_lbl>0) && (upper_pxl_lbl>0))
               end // if (enable)
             else
               begin
                  lbl_tbl_we <=0;
                  eqv_tbl_we <=0;
                  lbl_scanline_we <=0;
               end // else: !if(enable)
            end // if (more_pxls)
        end // else: !if(reset)
     end // always @ (posedge clk)
endmodule // image_processor
```

F Image Label Correcter

```
////////////////////////////////////////////////////////////////
// Project: Bacteria Colonalyzer (6.111 Final Project)
//    File:
//  Author: Yaw B. Anku
//    Date:
//
/*
 * Functional Description:
 * This module implements the second scanning step in the left-skewed
 * 6-connectedness neighborhood scheme. This second scan through the
 * image pixels corrects the labels of the pixels that were incorrectly
 * assigned on the first scan. This module fetches data from the Equiva-
 * lence Table/buffer. (For only addresses in the LABEL TABLE with non-zero
 * corresponding value in the Equivalence table will be corrected.
 * The module receives a correct_lbls signal from the controller module
 * to begin the corrections and sends a signal back to the controller after
 * the last label to be corrected is processed.
 */
//
//
////////////////////////////////////////////////////////////////
```

```
module img_lbl_corrector(clk,correct_lbls,enable,
                  reset,from_eqv_tbl,from_match_tbl,to_res_tbl,
                  lbl_chnge_addr,
                  res_tbl_we,blob_count,
                  correct_lbl_done);

   //CONSTANTS
   parameter ADDR_WIDTH = 17;
   parameter LBL_DATA_WIDTH = 4;
   parameter NUM_PXLS = 76800;

   //inputs
   input clk; //global clk
   input reset;//global reset
   input enable;//enable to help synchronize writes
   input correct_lbls; //from controller, active high to start
   input [LBL_DATA_WIDTH-1:0] from_eqv_tbl;//correction data
   input [LBL_DATA_WIDTH-1:0] from_match_tbl; //data to correct

   output reg [LBL_DATA_WIDTH-1:0] to_res_tbl;
   output reg res_tbl_we;//we 1 to correct,else 0

   //outputs
   output correct_lbl_done; //to label table/buffer
   output reg [ADDR_WIDTH-1:0] lbl_chnge_addr; //addr to be corrected

   //generate a start pulse
   level_to_pulse pulse0(clk,correct_lbls,start_pulse);

   //COMBINATIONAL LOGIC
   //signal 1 when last label has been corrected, else,0
   assign correct_lbl_done = (lbl_chnge_addr >= NUM_PXLS);
   //assign to_res_tbl = from_eqv_tbl;
   output reg [3:0] blob_count;
   //SEQUENTIAL LOGIC
   always @ (posedge clk)
```

```
     begin
      if(reset )
        begin
           lbl_chnge_addr <= 0;
             blob_count<=0;
        end
      else if (~reset && ~correct_lbl_done)
        begin
           if(enable)
             begin
              //only write if eqv data is > zero
              res_tbl_we <=1;//(from_eqv_tbl>0);
              lbl_chnge_addr <= lbl_chnge_addr + 1;
              to_res_tbl<= (from_eqv_tbl>0) ? from_eqv_tbl : from_match_tbl;
              blob_count<= (to_res_tbl > blob_count) ? to_res_tbl : blob_count;
             end

        end
     end // always @ (posedge clk)
endmodule // image_processor_scan2
```

G Blob Counter

```
//////////////////////////////////////////////////////////
// Project: Bacteria Colonalyzer (6.111 Final Project)
//    File:
//  Author: Yaw B. Anku
//    Date:
//
// Functional Description:
// This module calculates the total number distinct blobs
```

```
// by cycling through all the addresses of the corrected
// label table and returning the label with the largest
// numerical value. This numerical value is then displayed
// on the hex_display of the FPGA. The module awaits a
// count_blobs signal from the controller, upon which it
// begins cycling through the label table. It also outputs
// a signal count_done after cycling through the entire table
// ie. All the address locations.
//////////////////////////////////////////////////////////
module blob_counter(reset,clk,count_blobs,lbl_data_in,
            disp_blank, disp_clock, disp_rs, disp_ce_b,
            disp_reset_b, disp_data_out,count_done,lbl_tbl_raddr);

   //constants
   parameter MAX_LBL_TBL_RADDR =76800;
   parameter LBL_DATA_WIDTH = 4;
   parameter LBL_TBL_RADDR_WIDTH = 17;

   //inputs
   input reset;  //global reset
   input clk;    //global clk
   input [LBL_DATA_WIDTH-1:0] lbl_data_in; //from lable table/buffer

   input count_blobs;//signal from controller

   output reg [LBL_TBL_RADDR_WIDTH-1:0] lbl_tbl_raddr;//addr spec for lable table

   //outputs to hex display
   output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b,
        disp_reset_b;

   //signal to controller 1 when counting completes,else 0
   output count_done;

   //stores the count to be displayed on hex screen
   wire [63+20:0] count;
      assign count=lbl_data_in;
```

```
   wire      count_blobs_pulse; //1 for one clk cycle after start count asserts

   //generate start pulse
   level_to_pulse pulse0(clk,count_blobs,count_blobs_pulse);

   //generate hex display
   display_16hex hexdisp1(reset, clk, count[63+19:19],
            disp_blank, disp_clock, disp_rs, disp_ce_b,
            disp_reset_b, disp_data_out);

   assign count_done = count > 0;

   //SEQUENTIAL LOGIC
//   always @ (posedge clk)
//     begin
//    if(reset || count_blobs_pulse)
//      begin
//         count <=0;
//         lbl_tbl_raddr <=0;
//      end
//    else if (~reset && ~count_done)
//      begin
//         lbl_tbl_raddr <= lbl_tbl_raddr + 1;
//
//         //count only none-zero labels
//         if(lbl_data_in > count)
//           count <= {80'b0,lbl_data_in};
//      end // else: !if(reset)
//     end // always @ (posedge clk)
endmodule // blob_counter
```

E Image Pixel Pointer

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////

/

// Module Name:    img_pxl_pointer



// Description:

/  Modified version of vga_rom_disp by I. Chuang <ichuang@mit.edu>

//

// Example demonstrating display of image from ROM on 640x480 VGA

//

// This module reads data from ROM and creates the proper 8-bit pixel

// value for a pixel position defined by (hcount,vcount).  Note that

// there is a one cycle delay in reading from memory, so we may

// have a single pixel offset error here, to be fixed.  But the displayed

// image looks respectable nevertheless.

//

// To create the image ROM, use the Xilinx IP tools to generate a

// single port block memory, and load in an initial value file (*.coe)

// for your image.  This ROM should have width 8 (8-bit pixel output)

// and depth 76800 (320x240).

//
```

```
// The COE file may be generated from an 8-bit PGM format image file
// using pgm2coe.py, or using your own tool.  Read the Xilinx documentation
// for more about COE files.
//
////////////////////////////////////////////////////////////////////////////////
module img_pxl_pointer(clk,hcount,vcount,pix_clk,addr_reg);
   input clk;           // video clock
   input [9:0] hcount;  // current x,y location of pixel
   input [9:0] vcount;
   input       pix_clk; // pixel clock

   // the memory address is hcount/2 + vcount/2 * 320
   // (4 pixels per memory location, since image is 320x240, and
   // display is 640x480).

   reg [16:0]     raddr;
   always @(posedge clk)
     raddr <= (hcount==0 & vcount==0) ? 0
            : (hcount==0 & pix_clk & ~vcount[0]) ? raddr + 320 : raddr;

   wire [16:0]    addr = {8'b0,hcount[9:1]} + raddr[16:0];
      //wire [3:0] rom_pxl;
```

```
  output reg [16:0]     addr_reg;

      //reg [16:0] addr_med;

   always @(posedge clk) begin

        //addr_med <=addr;

        addr_reg <= addr;//_med;



        end

endmodule
```