Fingerprint Verification System

Cheryl Texin
Bashira Chowdhury

6.111 Final Project
Spring 2006

Appendix

Code Appendix:

Control FSM, Instantiate submodules, handle Minor FSMs: control

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
//control module
//function:
//    FSM that handles the minor FSM filtering operations, interfaces
with the labkit module.
//    Also instantiates subfunctions.  Handles the ROM/RAM bus lines,
to avoid
//    contention when switching between filtering operations and the
display.
//
//inputs:
//    clock
//    reset
//    enter:      button to start filtering
//    pixel: 10 bit VGA location
//    line: 9 bit VGA location
//    switch: used for displaying the images
//                        also sets the threshold for the edge filter
//
//outputs:
//    24 bit VGA color
//////////////////////////////////////////////////////////////////////////////////
module control(clk, reset, enter, pixel, line, switch, vga_color, led);

       input clk;
    input reset;
    input enter;
    input [9:0] pixel;  //VGA
    input [8:0] line;   //VGA
       input [7:0] switch;
    output [23:0] vga_color;
       output [7:0] led;

       //state parameters
       parameter RESET        = 0;        // reset mode
       parameter DISP         = 1;        // display image, wait for
input
       parameter EDGE         = 2;        // do edge filtering
       parameter WAIT_FILT    = 3;        // pause til filtering done
       parameter START_DIR    = 4;        // do direction filtering
       parameter WAIT_DIR     = 5;        // pause til direction done
       parameter RUN_MATCH    = 6;        // get match values
       parameter WAIT_MATCH= 7;           // hold til match is done

       // other params
       parameter TRUE   = 1;
       parameter FALSE  = 0;
       parameter M            = 256;
```

```
        parameter N            = 256;
        //for rom/rams (image)
        parameter IMG_PIXEL_BITS     = 16;
        parameter IMG_ADDR_BITS      = IMG_PIXEL_BITS - 3;
        parameter DIR_ADDR_BITS      = IMG_PIXEL_BITS -1;

        //state regs
        reg [5:0] state, next;

        // sobel filter
        reg en_FILT, start_filt;
        wire busy_filt;

        reg en_DIR, start_dir;
        wire busy_dir;

        wire enables;
        assign enables   = (en_FILT || en_DIR);  // if either enable is
TRUE, enables is TRUE

        //vals for match
        wire [3:0] nibble;
       wire [11:0] match0_UP, match0_DOWN;
       wire [11:0] match1_UP, match1_DOWN;
       wire [11:0] match2_UP, match2_DOWN;
       wire [11:0] match3_UP, match3_DOWN;
       wire busy_match;
       reg start_match;

/*************************************************************/
//signals for ram/rom accesses
       //ROM
       wire [IMG_ADDR_BITS-1:0] rom_addr;
       wire [7:0] rom_data;
       //EDGE rams
       wire edge_we;
       wire [IMG_ADDR_BITS-1:0] vedge_addr, hedge_addr;
       wire [7:0] vedge_din, hedge_din;
       wire [7:0] vedge_dout, hedge_dout;
       //DIRECTION rams
       wire dir_we;
       wire [DIR_ADDR_BITS-1:0] vdir_addr, hdir_addr;
       wire [7:0] vdir_din, hdir_din;
       wire [7:0] vdir_dout, hdir_dout;

//signals for display ram/rom accesses
       //ROM
       wire [IMG_ADDR_BITS-1:0] rom_addr_disp;
       wire [7:0] rom_data_disp;
       //EDGE rams
       wire [IMG_ADDR_BITS-1:0] vedge_addr_disp, hedge_addr_disp;
       //no din (not writing)
       wire [7:0] vedge_dout_disp, hedge_dout_disp;
       //DIRECTION rams
       wire [DIR_ADDR_BITS-1:0] vdir_addr_disp, hdir_addr_disp;
       //no din (not writing)
       wire [7:0] vdir_dout_disp, hdir_dout_disp;
```

```
//signals for switched color output
      wire [23:0] color_ROM;
      wire [23:0] color_vedge, color_hedge;
      wire [23:0] color_vdir, color_hdir;

//signals for direction filter ram/rom accesses
      //no ROM
      //EDGE rams
      wire [IMG_ADDR_BITS-1:0] edge_addr_dir;
      //no din (only reading from edge rams)
      wire [7:0] vedge_dout_dir, hedge_dout_dir;
      //DIRECTION rams
      wire dir_we_out;
      wire [DIR_ADDR_BITS-1:0] dir_addr_dir;
      wire [7:0] vdir_din_dir, hdir_din_dir;
      //no dout, not reading

//signals for edge filter ram/rom accesses
      //ROM
      wire [IMG_ADDR_BITS-1:0] rom_addr_edge;
      wire [7:0] rom_data_edge;
      //EDGE rams
      wire edge_we_out;
      wire [IMG_ADDR_BITS-1:0] edge_addr_edge;
      wire [7:0] vedge_din_edge, hedge_din_edge;
      // no dout, not reading
      //no need for DIRECTION rams

/*************************************************************/

       // sequential logic block
       always @ (posedge clk)
       begin
            if (!reset)
            begin
                  state <= RESET;
            end // reset
            else // no reset
            begin
                  state <= next;
            end // no reset
       end // sequential logic

       // comb logic block
       always @ (state, enter, busy_filt, busy_dir, busy_match)
       begin
            case (state)

                  RESET:
                  begin
                        start_dir   = FALSE;
                        en_DIR            = FALSE;
                        start_filt  = FALSE;
                        en_FILT     = FALSE;
                        next  = DISP;     // go to display mode
                        start_match = FALSE;
```

```
                    end // RESET

                    DISP:
                    begin
                          start_dir   = FALSE;
                          en_DIR            = FALSE;
                          start_filt  = FALSE;
                          en_FILT     = FALSE;
                          start_match = FALSE;
                          // if enter pressed, do edge filter
                          // otherwise continue doing display
                          next  = enter ? EDGE : DISP;
                    end // DISP

                    EDGE:
                    begin
                          start_dir   = FALSE;
                          en_DIR            = FALSE;
                          start_filt  = TRUE;
                          en_FILT     = TRUE;
                          start_match = FALSE;
                          next  = WAIT_FILT;
                    end // EDGE

                    WAIT_FILT:
                    begin
                          start_dir   = FALSE;
                          en_DIR            = FALSE;
                          start_filt  = FALSE;
                          en_FILT     = TRUE;
                          start_match = FALSE;
                          // do nothing while filtering
                          // do direction filtering when done
                          next  = busy_filt ? WAIT_FILT : START_DIR;
                    end // WAIT

                    START_DIR:
                    begin
                          start_dir   = TRUE;
                          en_DIR            = TRUE;
                          start_filt  = FALSE;
                          en_FILT     = FALSE;
                          start_match = FALSE;
                          next  = WAIT_DIR;
                    end

                    WAIT_DIR:
                    begin
                          start_dir   = FALSE;
                          en_DIR      = TRUE;
                          start_filt  = FALSE;
                          en_FILT     = FALSE;
                          start_match = FALSE;
                          // do nothing while filtering, calc match when
done
                          next  = busy_dir ? WAIT_DIR : RUN_MATCH;
                    end
```

```
                  RUN_MATCH:
                  begin
                        start_dir   = FALSE;
                        en_DIR            = FALSE;
                        start_filt  = FALSE;
                        en_FILT     = FALSE;
                        start_match = TRUE;
                        next  = WAIT_MATCH;
                  end

                  WAIT_MATCH:
                  begin
                        start_dir   = FALSE;
                        en_DIR            = FALSE;
                        start_filt  = FALSE;
                        en_FILT     = FALSE;
                        start_match = FALSE;
                        next  = busy_match ? WAIT_MATCH : DISP;
                  end

                  default
                  begin
                        start_dir   = FALSE;
                        en_DIR            = FALSE;
                        start_filt  = FALSE;
                        en_FILT     = FALSE;
                        start_match = FALSE;
                        next  = DISP;
                  end

            endcase
      end// comb logic block
/*****************************************************/
//ASSIGN INPUT WIRES for edge filter
// assign to output of mem block
assign rom_data_edge    = rom_data;

/*****************************************************/
        ////FILTER EDGE
        sobel do_sobel(.clk(clk), .reset(reset),
                              .enable(en_FILT), .start(start_filt),
.busy_wire(busy_filt), .switch(switch[1:0]),
                              .rom_data_wire(rom_data_edge),
.rom_addr_wire(rom_addr_edge),
                              .ram_addr_wire(edge_addr_edge), //output
                              .vert_din_wire(vedge_din_edge),
      //output -> RAM
                              .horiz_din_wire(hedge_din_edge),
                              .we_wire(edge_we_out)
//output
                              );
       defparam do_sobel.M    = M;
       defparam do_sobel.N    = N;
       defparam do_sobel.PIXEL_BITS = IMG_PIXEL_BITS;
/*****************************************/
//ASSIGN INPUT WIRES FOR direction filter
```

```
// assign to output of mem block;
assign vedge_dout_dir   = vedge_dout;
assign hedge_dout_dir   = hedge_dout;
/*****************************************/
       //FILTER DIRECTION
       dir_copy filt_dir(.clk(clk), .reset(reset), .enable(en_DIR),
.start(start_dir),
                        //read from EDGE rams
                        .read_addr_wire(edge_addr_dir),
                        .readv_dout_wire(vedge_dout_dir),
.readh_dout_wire(hedge_dout_dir),
                        // write to DIR rams
                        .write_addr_wire(dir_addr_dir),
                        .writev_din_wire(vdir_din_dir),
.writeh_din_wire(hdir_din_dir),
                        .we_wire(dir_we_out), .busy_wire(busy_dir)
                              );
      defparam filt_dir.PIXEL_BITS  = IMG_PIXEL_BITS;
      defparam filt_dir.M     = M;
      defparam filt_dir.N     = N;
/*****************************************************/
//ASSIGN INPUT WIRES for image displays
//assign to output of memory block
assign rom_data_disp    = rom_data;
assign vedge_dout_disp  = vedge_dout;
assign hedge_dout_disp  = hedge_dout;
assign vdir_dout_disp   = vdir_dout;
assign hdir_dout_disp   = hdir_dout;
/*****************************************************/
      wire[1:0] state_match;
      reg [7:0] led_reg;
      //assign led = ~{1'b1, 1'b1, state_match, busy_match, state};
      assign led  = led_reg;
      always @ (switch[7:4],
                              match0_UP, match0_DOWN,
                              match1_UP, match1_DOWN,
                              match2_UP, match2_DOWN,
                              match3_UP, match3_DOWN
                              )
      begin
            case (switch[7:4])
            0:
                  led_reg     = ~match0_UP;
            1:
                  led_reg     = ~match0_DOWN;
            2:
                  led_reg     = ~match1_UP;
            3:
                  led_reg     = ~match1_DOWN;
            4:
                  led_reg     = ~match2_UP;
            5:
                  led_reg     = ~match2_DOWN;
            6:
                  led_reg     = ~match3_UP;
            7:
                  led_reg     = ~match3_DOWN;
```

```
            default:
                  led_reg     = 8'hff;
                  endcase
      end//switch[7:4]
      //match FSM
      match make_match(.reset(reset),
                                          .clk(clk),
                                          .pixel(pixel), .line(line),
                                          .nibble(nibble),
.enable(start_match),
                                          .busy_wire(busy_match),
                                          .match0_UP_wire(match0_UP),
.match0_DOWN_wire(match0_DOWN),
                                          .match1_UP_wire(match1_UP),
.match1_DOWN_wire(match1_DOWN),
                                          .match2_UP_wire(match2_UP),
.match2_DOWN_wire(match2_DOWN),
                                          .match3_UP_wire(match3_UP),
.match3_DOWN_wire(match3_DOWN),
                                          .state_wire(state_match)
                                          );

      defparam make_match.M   = M;
      defparam make_match.N   = N;
      gen_imgdisp run_disp(
                  .clk(clk), .pixel(pixel), .line(line),
                  .rom_addr(rom_addr_disp), .rom_data(rom_data_disp),
.color_ROM(color_ROM),
                  .vedge_addr(vedge_addr_disp),
.vedge_dout(vedge_dout_disp), .color_vedge(color_vedge),
                  .hedge_addr(hedge_addr_disp),
.hedge_dout(hedge_dout_disp), .color_hedge(color_hedge),
                  .vdir_addr(vdir_addr_disp),
.vdir_dout(vdir_dout_disp), .color_vdir(color_vdir),
                  .hdir_addr(hdir_addr_disp),
.hdir_dout(hdir_dout_disp), .color_hdir(color_hdir),
                  .nibble(nibble)
                  );
      defparam run_disp.PIXEL_BITS  = IMG_PIXEL_BITS;
      defparam run_disp.M     = M;
      defparam run_disp.N     = N;

      //determine color
        colorOUT get_vga_color(
                              .switch(switch[3:0]),
                              .en_FILT(en_FILT),
                              .en_DIR(en_DIR),
                              .color_ROM(color_ROM),
                              .color_vert_edge(color_vedge),
.color_horiz_edge(color_hedge),
                              .color_vert_dir(color_vdir),
.color_horiz_dir(color_hdir),
                              .vga_color_out(vga_color)
                              );

/*****************************************************/
//ASSIGN INPUT WIRES for the memories
```

```
// ROM: default is display, otherwise from edge RAMs
assign rom_addr   = (!enables) ? rom_addr_disp : rom_addr_edge;

// EDGE RAMS: used in edge filter, direction filter, display
assign edge_we    = (en_FILT) ? edge_we_out : 1;      // hold high
unless running edge filter
assign vedge_addr = (en_FILT) ? edge_addr_edge  :
                                          (en_DIR) ? edge_addr_dir
      : vedge_addr_disp;
assign vedge_din  = (en_FILT) ? vedge_din_edge  : FALSE;    // only
writing if en_FILT
assign hedge_addr = (en_FILT) ? edge_addr_edge  :
                                          (en_DIR) ? edge_addr_dir
      : hedge_addr_disp;
assign hedge_din  = (en_FILT) ? hedge_din_edge  : FALSE;    // only
writing if en_FILT

//DIRECTION RAMS: used in direction filter, display
assign dir_we     = (en_DIR)  ? dir_we_out : 1; // hold high unless
running direction filter
assign vdir_addr  = (en_DIR) ? dir_addr_dir     : vdir_addr_disp;
assign vdir_din   = (en_DIR) ? vdir_din_dir     : FALSE;          //
only writing if en_DIR
assign hdir_addr  = (en_DIR) ? dir_addr_dir     : hdir_addr_disp;
assign hdir_din   = (en_DIR) ? hdir_din_dir     : FALSE;          //
only writing if en_DIR
/*************************************************************/
      generate_mems run_mems(
                  .clk(clk),
                  .rom_addr(rom_addr), .rom_data(rom_data),
                  .edge_we(edge_we),
                  .vedge_addr(vedge_addr), .vedge_din(vedge_din),
.vedge_dout(vedge_dout),
                  .hedge_addr(hedge_addr), .hedge_din(hedge_din),
.hedge_dout(hedge_dout),
                  .dir_we(dir_we),
                  .vdir_addr(vdir_addr), .vdir_din(vdir_din),
.vdir_dout(vdir_dout),
                  .hdir_addr(hdir_addr), .hdir_din(hdir_din),
.hdir_dout(hdir_dout)
                  );
      defparam run_mems.ADDR_BITS   = IMG_ADDR_BITS;

endmodule
```

VGA Color: colorOUT

```
//////////////////////////////////////////////////////////////
//colorOUT module
//function:
//    handles the color display to the VGA
//    based on the switch position, chooses the appropriate
//    24 bit color
//
//inputs:
//    switch: chooses which color to output
//                0: original image
//                1: vertical edges
//                2: horizontal edges
//                3: vertical directions
//                4: horizontal directions
//                default: GREEN (no ROM/RAM for that input)
//    en_FILT, en_DIR: when high, indicate that RAM addresses are being
used for a different
//                operation.  screen outputs all blue
//    color wires from ROM, 4 RAMS (2 each edge, direction filter)
//
//outputs:
//    24 bit color for the VGA
//////////////////////////////////////////////////////////////

module colorOUT(switch, en_FILT, en_DIR,
                              color_ROM, color_vert_edge,
color_horiz_edge,
                              color_vert_dir, color_horiz_dir,
vga_color_out
                              );
      input [3:0] switch;
      input en_FILT, en_DIR;
      input [23:0] color_ROM;
      input [23:0] color_vert_edge, color_horiz_edge;
      input [23:0] color_vert_dir, color_horiz_dir;
      output [23:0] vga_color_out;

      parameter BLUE          = 24'h0000ff;
      parameter GREEN   = 24'h00ff00;

      wire enables;
      assign enables    = (en_FILT || en_DIR);

      reg [23:0] vga_color;
      assign vga_color_out    = vga_color;

      always @ (switch, enables,
                              color_ROM, color_vert_edge,
color_horiz_edge,
                              color_vert_dir, color_horiz_dir
                        )
      begin
            case (switch)
```

```
                  0:
                  begin
                        vga_color   = enables ? BLUE : color_ROM;
      //blue if filtering
                  end // 0
                  1:
                  begin
                        vga_color   = enables ? BLUE : color_vert_edge;
      //blue if filtering
                  end
                  2:
                  begin
                        vga_color   = enables ? BLUE :
color_horiz_edge; // blue if filtering
                  end
                  3:
                  begin
                        vga_color   = enables ? BLUE  : color_vert_dir;
                  end
                  4:
                  begin
                        vga_color   = enables ? BLUE  :
color_horiz_dir;
                  end
                  default
                  begin
                        vga_color   = GREEN;    // invalid switch input
                  end
            endcase
      end // always @ switch

endmodule
```

Sobel Edge Filter: sobel

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////
//sobel module
//function:
//    performs a convolution with the sobel edge detection filters,
//    keeping only positive values above a variable threshold
//    splits the result into the horizontal and vertical components
//    filter addressing is as below:
//          | 0 | 1 | 2 |
//          | 3 |N/A| 4 |
//          | 5 | 6 | 7 |
//
//    values for the filter-
//
//    vertical filter:
//          | 1 | 0 | -1|
//          | 2 | 0 | -2|
//          | 1 | 0 | -1|
//
//    horizontal filter:
//          | 1 | 2 | 1 |
//          | 0 | 0 | 0 |
//          | -1| -2| -1|
//
//inputs:
//    clock
//    reset
//    enable: must be high to run code
//    start: indicates that the filtering should begin
//    rom_data:  value presented on original image data bus
//    switch (2 bits): sets the threshold for the filter
//
//outputs:
//    ROM address
//    RAM address: one line, used for both the horizontal and vertical
RAM simultaneously
//    din lines to the RAMs
//    we to the RAMs (one for both)
//    busy signal: to indicate that the FSM is still processing
//////////////////////////////////////////////////////////////////////////////////
////////
module sobel(clk, reset, enable, start,
                        rom_data_wire, rom_addr_wire,
                        ram_addr_wire,
                        vert_din_wire, horiz_din_wire,
                        we_wire, busy_wire, switch
                              );

      parameter PIXEL_BITS    = 12;
      parameter ADDR_BITS     = PIXEL_BITS - 3;

       input clk;
    input reset;
```

```
input enable;
input start;
   output [ADDR_BITS-1:0] rom_addr_wire;
   input [7:0] rom_data_wire;
output [ADDR_BITS-1:0] ram_addr_wire;
output [7:0] vert_din_wire, horiz_din_wire;
output we_wire;
output busy_wire;
   input [1:0] switch;

   reg [ADDR_BITS-1:0] ram_addr;
   reg [7:0] vert_ram_din, horiz_ram_din;
   assign ram_addr_wire   = ram_addr;
   assign vert_din_wire   = vert_ram_din;
   assign horiz_din_wire  = horiz_ram_din;

   // rom
   reg[ADDR_BITS-1:0] rom_addr;
   reg[7:0] rom_val;
   wire [ADDR_BITS-1:0] rom_addr_wire =     rom_addr;

   assign rom_addr_wire   = rom_addr;

  parameter TRUE    = 1;
  parameter FALSE   = 0;
  parameter M = 32;
  parameter N = 32;

  //state
  reg [3:0] state, next;
  parameter INIT                          = 0;
  parameter GET_ROM_ADDR        = 2;
  parameter READ                          = 3;
  parameter GET_RAM_ADDR        = 5;
  parameter WRITE                       = 8;
  parameter UPDATE_PIXEL        = 9;
  parameter UPDATE_ADDR         = 11;
  parameter READ_ADDR                   = 1;
  parameter UPDATE_FILT_VAL     = 4;
  parameter INC_FILT_BIT        = 10;
  parameter PROC_FILT                   = 6;
  parameter UPDATE_WRITE_DATA= 7;


  //pixel writing location
  reg [PIXEL_BITS-1:0] pixel;
  reg pixel_int;
  wire [ADDR_BITS-1:0]pixel_byte_wire;
  wire [2:0]pixel_bit_wire;

  //pixel reading location
  wire [PIXEL_BITS-1:0] pixel_read_wire;
  reg [PIXEL_BITS-1:0] pixel_read;
  wire[ADDR_BITS-1:0] read_byte_wire;
  wire[2:0] read_bit_wire;

  //filter
```

```
      reg [2:0] filt_bit;
      reg filt_bit_int;
      reg [7:0] filt_val;
      reg [2:0] vert_filt_pos, vert_filt_neg, horiz_filt_pos,
horiz_filt_neg;   // max value = 4
      reg vert_filt_result, horiz_filt_result;  // boolean result of
comparing pos and neg
      reg [7:0] vert_write_val, horiz_write_val;            // holds 8
filt_result bits
      reg [1:0] thresh;

      reg busy;
      assign busy_wire  = busy;
      reg we;
      assign we_wire    = we;

      // sequential logic block
       always @ (posedge clk)
       begin
            //in reset, set to init conds
            if (!reset || !enable)  // in reset, or disabled
            begin
                  state <=    INIT;
                  pixel <= 0;
                  filt_bit    <= 0;
            end // in reset

            else // no reset
            begin
                  state <= next;

                  pixel <= pixel_int ? (pixel + 1) : pixel;
                  filt_bit    <= (filt_bit_int) ? (filt_bit+1) :
filt_bit;
            end // no reset

       end // posedge clk

       always @ (state, start, thresh, switch,
                              pixel, pixel_byte_wire, pixel_bit_wire,
                              rom_data_wire, rom_val,
                              pixel_read_wire, read_byte_wire,
read_bit_wire,
                              filt_bit, filt_val,
                              vert_filt_result, vert_write_val,
                              horiz_filt_result, horiz_write_val
                              )
       begin
            case (state)

            INIT:
            begin
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;  // disable write to RAM
                  if (start)
                  begin
```

```
                        thresh      = switch;
                        busy  = TRUE;
                        next  = READ_ADDR;      // start getting data
                  end // enabled
                  else  // not enabled
                  begin
                        busy  = FALSE;
                        next = INIT;      // hold initial conditions
                  end // not enabled
            end // INIT

            READ_ADDR:
            begin
                  //set read location (based on write location,
filt_bit)
                  pixel_read  = pixel_read_wire;

                  // not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = GET_ROM_ADDR;
            end

            GET_ROM_ADDR:
            begin
                  //set ROM read address based on pixel_read byte
                  rom_addr    = read_byte_wire;

                  //not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = READ;
            end // GET_ROM_ADDR

            READ:
            begin
                  //read ROM
                  rom_val     = rom_data_wire;

                  //not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = UPDATE_FILT_VAL;


            end   //READ

            UPDATE_FILT_VAL:
            begin
                  //enter in correct filter position value at the bit
being read
```

```
                  // invert bit location (addressing -> for pixel, <-
for hex data)
                  filt_val[filt_bit]      = rom_val[7-read_bit_wire];

                  //not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  // start updating WRITE VALUE only if the last value
has been received
                  next  = (filt_bit == 7) ? GET_RAM_ADDR :
INC_FILT_BIT;
            end // UPDATE_FILT_VAL

            GET_RAM_ADDR:
            begin
                  // set RAM write address (based on pixel being
written)
                  ram_addr    = pixel_byte_wire;

                  //not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = PROC_FILT;
            end   //GET_RAM_ADDR

            PROC_FILT:
            begin
                  //process the filter -> determine if there is an edge
                  //vertical filter
                  vert_filt_pos     = filt_val[0] + 2*filt_val[3] +
filt_val[5];      //intermediate val
                  vert_filt_neg     = filt_val[2] + 2*filt_val[4] +
filt_val[7];      // intermediate val
                  vert_filt_result  = (

(vert_filt_pos > vert_filt_neg) &&

((vert_filt_pos - vert_filt_neg) > thresh)
                                                                  );    //
write a 1 if difference is positive & greater than thresh
                  //horizontal filter
                  horiz_filt_pos    = filt_val[0] + 2*filt_val[1] +
filt_val[2];      //intermediate val
                  horiz_filt_neg    = filt_val[5] + 2*filt_val[6] +
filt_val[7];      // intermediate val
                  horiz_filt_result = (

(horiz_filt_pos > horiz_filt_neg) &&

((horiz_filt_pos - horiz_filt_neg) > thresh)
                                                                  );    //
write a 1 if difference is positive & greater than thresh
```

```
                  //not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = UPDATE_WRITE_DATA;
            end

            UPDATE_WRITE_DATA:
            begin
                  //set value to be written in current bit location
                  //pixel bit inverted from hex addressing
                  vert_write_val[7-pixel_bit_wire]    =
vert_filt_result;
                  horiz_write_val[7-pixel_bit_wire]   =
horiz_filt_result;

                  //not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  // WRITE value to RAM iff all bits have been
determined,
                  //else just increment the write position
                  next  = (pixel_bit_wire == 7) ? WRITE : UPDATE_PIXEL;
            end // WRITE_DATA

            WRITE:
            begin
                  // set data to be written
                  //invert bits -> edges in black
                  vert_ram_din      = ~vert_write_val;
                  horiz_ram_din     = ~horiz_write_val;

                  //not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 0;                    // set WE active
                  busy  = TRUE;
                  next  = UPDATE_PIXEL;
            end // WRITE

            UPDATE_PIXEL:
            begin
                  //not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = TRUE;           //set pixel_int
                  we    = 1;
                  busy  = TRUE;
                  next  = (pixel    == M*N-1) ? INIT : INC_FILT_BIT;
      // done on last pixel
            end // UPDATE_PIXEL


            INC_FILT_BIT:
```

```
            begin
                  //not latched
                  filt_bit_int      = TRUE;     //update filt_bit
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = UPDATE_ADDR;
            end // INC FILT_BIT;

            UPDATE_ADDR: //pause for pixel to be updated
            begin

                  // not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = READ_ADDR;
            end // UPDATE_ADDR

            default:
            begin
                  //not latched
                  filt_bit_int      = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = INIT;
            end

            endcase
       end// comb logic

       byte_bit pixel_byte_bit(.pixel_number(pixel),

      .byte(pixel_byte_wire), .bit(pixel_bit_wire)
                                                  );
      defparam pixel_byte_bit.PIXEL_BITS  = PIXEL_BITS;

      byte_bit read_byte_bit(.pixel_number(pixel_read),

      .byte(read_byte_wire), .bit(read_bit_wire)
                                                  );
      defparam read_byte_bit.PIXEL_BITS   = PIXEL_BITS;

      read_loc UPDATE_READ_ADDR(.clk(clk),
                              .filt_loc(filt_bit), .pixel_write(pixel),
                              .read(pixel_read_wire)
                              );
      defparam UPDATE_READ_ADDR.M   = M;
      defparam UPDATE_READ_ADDR.N   = N;
      defparam UPDATE_READ_ADDR.PIXEL_BITS      = PIXEL_BITS;


endmodule
```

Debounce module: debounce

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce (reset, clock, noisy, clean);
   parameter DELAY = 270000;   // .01 sec with a 27Mhz clock
   input reset, clock, noisy;
   output clean;

   reg [18:0] count;
   reg new, clean;

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

endmodule
```

VGA delay: delay

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////
// Company:
// Engineer:
//
// Create Date:    11:30:30 03/13/06
// Design Name:
// Module Name:    delay
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////
module delay(reset, vsync, hsync, pixel_clock, vsync_delay,
hsync_delay);
    input reset;
       input vsync;
    input hsync;
    input pixel_clock;
    output vsync_delay;
    output hsync_delay;

       reg v1, v2;
       reg h1, h2;

       assign vsync_delay = v2;
       assign hsync_delay = h2;

       always @ (posedge pixel_clock)
       begin
            if (!reset)
            begin
                  v1 <= 1;
                  v2 <= 1;

                  h1 <= 1;
                  h2 <= 1;
            end // in reset
            else
            begin
                  v1 <= vsync;
                  v2 <= v1;

                  h1 <= hsync;
```

```
                h2 <= h1;
            end // else not in reset
        end // always

endmodule
```

Direction Filter, Get Filter Values: dir_copy

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////
// Company:
// Engineer:
//
// Create Date:    22:37:35 05/02/06
// Design Name:
// Module Name:    copy
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////
module dir_copy(clk, reset, enable, start,
                        read_addr_wire, readv_dout_wire,
readh_dout_wire,
                        write_addr_wire, writev_din_wire,
writeh_din_wire,
                        we_wire, busy_wire
                              );

      parameter PIXEL_BITS    = 12;
      parameter READ_ADDR_BITS      = PIXEL_BITS - 3; //use 1 bit per
pixel
      parameter WRITE_ADDR_BITS     = PIXEL_BITS - 1; // use 4 bits per
pixel

       input clk;
    input reset;
    input enable;
    input start;
       output [READ_ADDR_BITS-1:0] read_addr_wire;
       input [7:0] readv_dout_wire, readh_dout_wire;
    output [WRITE_ADDR_BITS-1:0] write_addr_wire;
    output [7:0] writev_din_wire, writeh_din_wire;
    output we_wire;
    output busy_wire;

      parameter TRUE    = 1;
      parameter FALSE   = 0;
      parameter M = 32;
      parameter N = 32;
      parameter FILT_MAX      = 24; // 25 filter locations
```

```
      //state
      reg [3:0] state, next;
      parameter INIT                          = 0;
      parameter SET_READ_ADDR       = 2;
      parameter READ                          = 3;
      parameter GET_WRITE_ADDR      = 5;
      parameter WRITE                     = 8;
      parameter UPDATE_PIXEL        = 9;
      parameter UPDATE_ADDR         = 11;
      parameter READ_ADDR                 = 1;
      parameter UPDATE_FILT_VAL     = 4;
      parameter INC_FILT_BIT        = 10;
      parameter PROC_FILT                 = 6;
      parameter UPDATE_WRITE_DATA= 7;

      reg busy;
      assign busy_wire  = busy;
      reg we;
      assign we_wire    = we;

      reg [PIXEL_BITS-1:0] pixel;
      reg pixel_int;
      reg [PIXEL_BITS-1:0] pixel_read;
      reg [READ_ADDR_BITS-1:0] read_addr;
      wire [READ_ADDR_BITS-1:0] read_byte_wire;
      wire [2:0] read_bit_wire;
      reg [7:0] readh_dout, readv_dout;
      reg [WRITE_ADDR_BITS-1:0] write_addr;
      wire [WRITE_ADDR_BITS-1:0] pixel_byte_wire;
      wire pixel_nibble_wire;
      reg [3:0] vert_filt_resultHI, vert_filt_resultLO;
      reg [3:0] horiz_filt_resultHI, horiz_filt_resultLO;
      reg [7:0] vert_write_val, horiz_write_val;
      reg [7:0] writev_din, writeh_din;
      reg [24:0] hval, vval;
      reg [4:0] val_loc;
      reg val_loc_int;
      wire [PIXEL_BITS-1:0] pixel_read_wire;

wire [24:0] vval_wire, hval_wire;
assign vval_wire  = vval;
assign hval_wire  = hval;
wire [3:0] vresult_wire, hresult_wire;

      assign write_addr_wire  = write_addr;
      assign writeh_din_wire  = writeh_din;
      assign writev_din_wire  = writev_din;
      assign read_addr_wire   = read_addr;

      // sequential logic block
       always @ (posedge clk)
       begin
            //in reset, set to init conds
            if (!reset || !enable)  // in reset, or disabled
            begin
                  state <=    INIT;
                  pixel <= 0;
```

```
                  val_loc     <= 0;
            end // in reset

            else // no reset
            begin
                  state <= next;
                  pixel <= pixel_int ? (pixel + 1) : pixel;
                  val_loc     <= !val_loc_int ? val_loc :
                                    ( (val_loc== 24) ? 0 : (val_loc +1)
);          //maxes out at 24
            end // no reset

       end // posedge clk

       always @ (
                              state, start,
                              pixel, read_byte_wire, readh_dout_wire,
readv_dout_wire,
                              readh_dout, read_bit_wire, readv_dout,
pixel_byte_wire, pixel_nibble_wire,
                              hval, vval, vert_filt_resultHI,
vert_filt_resultLO, horiz_filt_resultHI,
                              horiz_filt_resultLO, vert_write_val,
horiz_write_val,
                              val_loc, pixel_read_wire,
                              vresult_wire, hresult_wire
                              )
       begin
            case (state)

            INIT:
            begin
                  val_loc_int = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;  // disable write to RAM
                  if (start)
                  begin
                        busy  = TRUE;
                        next  = READ_ADDR;      // start getting data
                  end // enabled
                  else  // not enabled
                  begin
                        busy  = FALSE;
                        next = INIT;      // hold initial conditions
                  end // not enabled
            end // INIT

            READ_ADDR:
            begin
                  //set read location
                  pixel_read  = pixel_read_wire;

                  // not latched
                  val_loc_int = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
```

```
                next  = SET_READ_ADDR;
            end

            SET_READ_ADDR:
            begin
                //set ROM read address based on pixel_read byte
                read_addr   = read_byte_wire;

                //not latched
                val_loc_int = FALSE;
                pixel_int   = FALSE;
                we    = 1;
                busy  = TRUE;
                next  = READ;
            end // GET_ROM_ADDR

            READ:
            begin
                //read RAMs
                readh_dout  = readh_dout_wire;
                readv_dout  = readv_dout_wire;

                //not latched
                val_loc_int = FALSE;
                pixel_int   = FALSE;
                we    = 1;
                busy  = TRUE;
                next  = UPDATE_FILT_VAL;

            end   //READ

            UPDATE_FILT_VAL:
            begin
                //enter in correct filter position value at the bit
being read
                // invert bit location (addressing -> for pixel, <-
for hex data)
                hval[val_loc]     = ~readh_dout[7-read_bit_wire];
                vval[val_loc]     = ~readv_dout[7-read_bit_wire];

                //not latched
                val_loc_int = FALSE;
                pixel_int   = FALSE;
                we    = 1;
                busy  = TRUE;
                // start updating WRITE VALUE only if the last value
has been received
                next  = (val_loc == 24) ? GET_WRITE_ADDR :
INC_FILT_BIT;
            end // UPDATE_FILT_VAL

            GET_WRITE_ADDR:
            begin
                // set RAM write address (based on pixel being
written)
                write_addr  = pixel_byte_wire;
```

```
                  //not latched
                  val_loc_int = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = PROC_FILT;
            end   //GET_WRITE_ADDR

            PROC_FILT:
            begin
                  //process the filter -> determine direction
                  //do DIR_FILT
                  if (!pixel_nibble_wire)// getting 1st (left most,
MS)nibble
                  begin
                        vert_filt_resultHI      = vresult_wire;
                        horiz_filt_resultHI     = hresult_wire;
                  end
                  else // getting 2nd (rightmost, LS) nibble
                  begin
                        vert_filt_resultLO      = vresult_wire;
                        horiz_filt_resultLO     = hresult_wire;
                  end

                  //not latched
                  val_loc_int = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  next  = UPDATE_WRITE_DATA;
            end

            UPDATE_WRITE_DATA:
            begin
                  //set value to be written in current bit location
                  //2 results written per address
                  vert_write_val    = {vert_filt_resultHI,
vert_filt_resultLO};
                  horiz_write_val   = {horiz_filt_resultHI,
horiz_filt_resultLO};

                  //not latched
                  val_loc_int = FALSE;
                  pixel_int   = FALSE;
                  we    = 1;
                  busy  = TRUE;
                  // WRITE value to RAM iff all nibbles have been
determined,
                  //else just increment the write position
                  next  = (pixel_nibble_wire) ? WRITE : UPDATE_PIXEL;
            end // WRITE_DATA

            WRITE:
            begin
                  // set data to be written
                  writev_din  = vert_write_val;
                  writeh_din  = horiz_write_val;
```

```
            //not latched
            val_loc_int = FALSE;
            pixel_int   = FALSE;
            we    = 0;                    // set WE active
            busy  = TRUE;
            next  = UPDATE_PIXEL;
      end // WRITE

      UPDATE_PIXEL:
      begin
            //not latched
            val_loc_int = FALSE;
            pixel_int   = TRUE;           //set pixel_int
            we    = 1;
            busy  = TRUE;
            next  = (pixel    == M*N-1) ? INIT : INC_FILT_BIT;
// done on last pixel
      end // UPDATE_PIXEL

      INC_FILT_BIT:
      begin
            //not latched
            val_loc_int = TRUE;
            pixel_int   = FALSE;
            we    = 1;
            busy  = TRUE;
            next  = UPDATE_ADDR;
      end // INC FILT_BIT;

      UPDATE_ADDR: //pause for pixel to be updated
      begin

            // not latched
            val_loc_int = FALSE;
            pixel_int   = FALSE;
            we    = 1;
            busy  = TRUE;
            next  = READ_ADDR;
      end // UPDATE_ADDR

      default:
      begin
            //not latched
            val_loc_int = FALSE;
            pixel_int   = FALSE;
            we    = 1;
            busy  = TRUE;
            next  = INIT;

      end

      endcase
 end// comb logic

 byte_nibble pixel_byte_nibble(.pixel_number(pixel),
```

```
      .byte(pixel_byte_wire), .nibble(pixel_nibble_wire)
                                                      );
      defparam pixel_byte_nibble.PIXEL_BITS     = PIXEL_BITS;

      byte_bit read_byte_bit(.pixel_number(pixel_read),

      .byte(read_byte_wire), .bit(read_bit_wire)
                                                      );
      defparam read_byte_bit.PIXEL_BITS   = PIXEL_BITS;

      read_loc5x5 switch_read_addr(.clk(clk),
                           .filt_loc(val_loc), .pixel(pixel),
                           .read(pixel_read_wire)
                           );
      defparam switch_read_addr.M   = M;
      defparam switch_read_addr.N   = N;
      defparam switch_read_addr.PIXEL_BITS      = PIXEL_BITS;

      dir_filt do_dir(.vval(vval_wire), .hval(hval_wire),
                                  .vresult_wire(vresult_wire),
.hresult_wire(hresult_wire)
                                  );
      defparam do_dir.FILT_MAX      = FILT_MAX;

endmodule
```

Direction Filter, Choose Direction: dir_filt

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////
//dir_filt module
//function:
//    take the values from the 5x5 convolution and determine
//    the appropriate direction vector (current pixel is #12)
//    filter is 5x5 indexed as below:
//    | 0 | 1 | 2 | 3 | 4 |
//    | 5 | 6 | 7 | 8 | 9 |
//    | 10| 11| 12| 13| 14|
//    | 15| 16| 17| 18| 19|
//    | 20| 21| 22| 23| 24|
//
//    direction vectors are:
//          DIAG_DOWN : \
//          DIAG_UP      : /
//          HORIZONTAL: -
//          VERTICAL     : |
//          NO_DIR       : does not match a direction vector
//
//    The vertical and horizontal edge images can only have
//    vertical and horizontal directions, respectively.
//
//inputs:
//    vval: 25  pixels in the 5x5 array surrounding the vertical edge
//    hval: 25  pixels in the 5x5 array surrounding the horizontal edge
//
//outputs:
//    vresult_wire:     direction vector for the current pixel in the
vertical direction filter
//    hresult_wire:  direction vector for the current pixel in the
horizontal direction filter
//////////////////////////////////////////////////////////////////////////////////
////////
module dir_filt(vval, hval, vresult_wire, hresult_wire);

      parameter FILT_MAX      = 24;

      input [FILT_MAX:0] vval, hval;
      output [3:0] vresult_wire, hresult_wire;

      reg [3:0] vresult, hresult;
      reg [8:0] v_3x3, h_3x3;
      assign vresult_wire     = vresult;
      assign hresult_wire     = hresult;

      parameter NO_DIR        = 5;
      parameter DIAG_DOWN     = 1;
      parameter DIAG_UP       = 2;
      parameter HORIZONTAL    = 3;
      parameter VERTICAL      = 4;
```

```
      always @ (vval, hval)
      begin
            if (vval[0] && vval[6] && vval[12] && vval[18] &&
      vval[24])
                  begin
                        vresult     = DIAG_DOWN;
                  end
            else if (vval[20] && vval[16] && vval [12] && vval[8] &&
vval[4])
                  begin
                        vresult     = DIAG_UP;
                  end
            else if (vval[2] &&  vval[7] && vval[12] && vval[17] &&
vval[22])
                  begin
                        vresult     = VERTICAL;
                  end
            else vresult      = NO_DIR;

            if (hval[0] && hval[6] && hval[12] && hval[18] &&
      hval[24])
                  begin
                        hresult     = DIAG_DOWN;
                  end
            else if (hval[20] && hval[16] && hval [12] && hval[8] &&
hval[4])

                  begin
                        hresult     = DIAG_UP;
                  end
            else if (hval[10] && hval[11] && hval[12] && hval[13] &&
hval[14])
                  begin
                        hresult     = HORIZONTAL;
                  end
            else hresult      = NO_DIR;

      end //always

endmodule
```

Instantiate Image Display Functions: gen_imgdisp

```
////////////////////////////////////////////
//gen_imgdisp module
//function:
//    instantiates the calls to the image display modules
//
//inputs:
//    clock
//    pixel location
//    line location
//    dout lines for ROM, RAMS
//
//outputs:
//    address line for ROM, RAMS
//    24 bit color for ROM, RAMS
////////////////////////////////////////////////////////////////

module gen_imgdisp(
                  clk, pixel, line,
                  rom_addr, rom_data, color_ROM,
                  vedge_addr, vedge_dout, color_vedge,
                  hedge_addr, hedge_dout, color_hedge,
                  vdir_addr, vdir_dout, color_vdir,
                  hdir_addr, hdir_dout, color_hdir, nibble
                  );

      parameter PIXEL_BITS    = 16;
      parameter ADDR_BITS     = PIXEL_BITS-3;
      parameter DIR_ADDR_BITS = ADDR_BITS+2;
      parameter M = 256;
      parameter N = 256;


      input clk;
      input [9:0] pixel;
      input [8:0] line;
      //ROM
      output [ADDR_BITS-1:0] rom_addr;
      input [7:0] rom_data;
      output [23:0] color_ROM;
      //EDGE rams
      output [ADDR_BITS-1:0] vedge_addr, hedge_addr;
      input [7:0] vedge_dout, hedge_dout;
      output [23:0] color_vedge, color_hedge;
      //DIR rams
      output [DIR_ADDR_BITS-1:0] vdir_addr, hdir_addr;
      input [7:0] vdir_dout, hdir_dout;
      output [23:0] color_vdir, color_hdir;

      output [3:0] nibble;
      wire [3:0] nibble_hold;

      //256x256 fingerprint ROM
      imgdisp ROMdisp_finger256(.pixel_clk(clk),
```

```
                        .pixel(pixel), .line(line),
                        .color_out(color_ROM),
                        .addr_wire(rom_addr), .data(rom_data)
                        );
      defparam ROMdisp_finger256.M  = M;
      defparam ROMdisp_finger256.N  = N;
      defparam ROMdisp_finger256.IMG_PIXEL_BITS = PIXEL_BITS;

      //256x256 fingerprint edge displays
      //vert
      imgdisp finger256_vertRAM(.pixel_clk(clk),
                        .pixel(pixel), .line(line),
                        .color_out(color_vedge),              //output
                        .addr_wire(vedge_addr), // output
                        .data(vedge_dout)       //input
                        );
      defparam finger256_vertRAM.M  = M;
      defparam finger256_vertRAM.N  = N;
      defparam finger256_vertRAM.IMG_PIXEL_BITS = PIXEL_BITS;

      //horiz
      imgdisp finger256_horizRAM(.pixel_clk(clk),
                        .pixel(pixel), .line(line),
                        .color_out(color_hedge),              //output
                        .addr_wire(hedge_addr), // output
                        .data(hedge_dout)       //input
                        );
      defparam finger256_horizRAM.M = M;
      defparam finger256_horizRAM.N = N;
      defparam finger256_horizRAM.IMG_PIXEL_BITS      = PIXEL_BITS;

      imgdisp_dir vert_dir(.pixel_clk(clk), .pixel(pixel), .line(line),
                              .color_out(color_vdir),
                              .addr_wire(vdir_addr),
                              .data(vdir_dout),
                              .nibble(nibble)
                              );
      defparam vert_dir.IMG_PIXEL_BITS    = PIXEL_BITS;
      defparam vert_dir.M     = M;
      defparam vert_dir.N     = N;

      imgdisp_dir horiz_dir(.pixel_clk(clk), .pixel(pixel),
.line(line),
                              .color_out(color_hdir),
                              .addr_wire(hdir_addr),
                              .data(hdir_dout),
                              .nibble(nibble_hold)
                              );
      defparam horiz_dir.IMG_PIXEL_BITS   = PIXEL_BITS;
      defparam horiz_dir.M    = M;
      defparam horiz_dir.N    = N;

endmodule
```

Image Display for Sobel Edge Filter: imgdisp

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////
//imgdisp
//
//function:
//    reads an image off of a ram/rom
//    displays it in the center of the screen,
//    background set to MIT red.
//    The image is the ROM,
//    or one of the 2 EDGE RAMS.
//    Each bit corresponds to one pixel.
//
//inputs:
//    clock
//    pixel location
//    line location
//    data (for ram/rom)
//outputs:
//    24 bit color
//    address (for ram/rom)
//
//////////////////////////////////////////////////////////////////////////////////
////////
module imgdisp(pixel_clk, pixel, line, color_out,
                        addr_wire, data);
      //for rom/rams (image)
       parameter IMG_PIXEL_BITS     = 12;
       parameter IMG_ADDR_BITS      = IMG_PIXEL_BITS - 3;
   input pixel_clk;
   input [9:0] pixel; //VGA
   input [8:0] line;     //VGA
   output [23:0] color_out;
      output [IMG_ADDR_BITS-1:0] addr_wire;
      input [7:0] data;

    //color params
    parameter BLACK        = 24'h0;
    parameter WHITE        = 24'hffffff;
    parameter MIT_red    = 24'b0101_1111_0001_1111_0001_1111;

    //size params
    parameter M     = 32;    // 32xN bit image
    parameter N    = 32;    // Mx32 bit image
    parameter scale    = 1;        // zero order hold 10 pixels
    parameter scrn_h    = 480;    // screen is 480 pixels high
    parameter scrn_w    = 640;    // screen is 640 pixels wide

    parameter ROW_START    = scrn_h/2-M*scale/2;
    parameter ROW_END    = scrn_h/2+M*scale/2;
    parameter COL_START    = scrn_w/2-N*scale/2;
    parameter COL_END    = scrn_w/2+N*scale/2;
```

```
parameter TRUE    = 1;
parameter FALSE    = 0;

reg [IMG_ADDR_BITS-1:0] rom_addr    = 0;
reg [23:0] color;

reg line_disp, pixel_disp;    // booleans for display area

reg [7:0] rom_val;
wire [7:0] data;

reg [2:0] hex_convert    = 0;

   // address hack CLF
assign addr_wire = rom_addr + 7'd1;
reg nextRom;
//instantiate rom

//determine output color
always @ (posedge pixel_clk)
begin

   rom_val <= (nextRom) ? data : rom_val;

// check if row in display range
   if ((line == 0) && (pixel == 0))//reset
       begin
           hex_convert    <= 0;
           rom_addr    <= 0;
       end

       else if(line_disp && pixel_disp)    // within display area
       begin
           hex_convert    <= hex_convert +1;    // move to next bit
           rom_addr <= (hex_convert == 7) ? rom_addr + 1: rom_addr;
       end// in disp range
       else // not in disp range;
       begin
           hex_convert    <= hex_convert;
           rom_addr    <= rom_addr;
       end

end      // seq. logic

always @ (line or pixel)
begin
    if ((line >= ROW_START) && (line < ROW_END))
       line_disp    = TRUE;
   else line_disp    = FALSE;

    // check if column in display range
    if ((pixel >= COL_START) && (pixel < COL_END))
       pixel_disp    = TRUE;
   else pixel_disp     = FALSE;
end

always @(data, rom_addr, line_disp, pixel_disp,
```

```
                rom_val, hex_convert)
    begin
        if (line_disp && pixel_disp)
            begin
                nextRom = (hex_convert ==7) ? 1 : 0;
               color = (rom_val[7-hex_convert]) ? WHITE : BLACK;
          end
         else
            begin
               color    = MIT_red;     // border stands out from image
                nextRom = 0;
          end
    end // comb logic

    // assign color outputs
    assign color_out    = color;

endmodule
```

Image Display for Direction Filter: imgdisp_dir

```
////////////////////////////////////////////////////////////////////////
/////////
//imgdisp_dir module
//
//function:
//    reads an image off of a ram
//    displays it in the center of the screen,
//    background set to MIT red.
//    The image is one of the direction rams, which encodes
//    each pixel with a 4 bit color.
//
//inputs:
//    clock
//    pixel location
//    line location
//    data (for ram/rom)
//outputs:
//    24 bit color
//    address (for ram/rom)
//
////////////////////////////////////////////////////////////////////////
/////////

module imgdisp_dir(pixel_clk, pixel, line, color_out,
                        addr_wire, data, nibble);
      //for rom/rams (image)
       parameter IMG_PIXEL_BITS     = 12;
       parameter IMG_ADDR_BITS      = IMG_PIXEL_BITS - 1;   //need one
nibble each time

   input pixel_clk;
   input [9:0] pixel; //VGA
   input [8:0] line;     //VGA
   output [23:0] color_out;
      output [IMG_ADDR_BITS-1:0] addr_wire;
      input [7:0] data;
      output [3:0] nibble;

      parameter NO_DIR        = 5;
      parameter DIAG_DOWN     = 1;
      parameter DIAG_UP       = 2;
      parameter HORIZONTAL    = 3;
      parameter VERTICAL      = 4;

    //color params RGB
    parameter BLACK        = 24'h0;
    parameter WHITE        = 24'hffffff;
    parameter MIT_red         = 24'b0101_1111_0001_1111_0001_1111;
       parameter RED                    = 24'hff_00_00;
       parameter GREEN              = 24'h00_ff_00;
       parameter BLUE               = 24'h00_00_ff;

    //size params
```

```
    parameter M     = 32;    // 32xN bit image
    parameter N    = 32;    // Mx32 bit image
    parameter scale    = 1;        // zero order hold 10 pixels
    parameter scrn_h    = 480;    // screen is 480 pixels high
    parameter scrn_w    = 640;    // screen is 640 pixels wide

    parameter ROW_START    = scrn_h/2-M*scale/2;
    parameter ROW_END    = scrn_h/2+M*scale/2;
    parameter COL_START    = scrn_w/2-N*scale/2;
    parameter COL_END    = scrn_w/2+N*scale/2;

    parameter TRUE    = 1;
    parameter FALSE    = 0;

    reg [IMG_ADDR_BITS-1:0] rom_addr    = 0;
    reg [23:0] color;

    reg line_disp, pixel_disp;    // booleans for display area

    reg [7:0] rom_val;
       reg [3:0] nibble;
    wire [7:0] data;

    reg hex_convert    = 0;

       // address hack CLF
    assign addr_wire = rom_addr + 7'd1;
    reg nextRom;
    //instantiate rom

    //determine output color
    always @ (posedge pixel_clk)
    begin

       rom_val <= (nextRom) ? data : rom_val;

    // check if row in display range
       if ((line == 0) && (pixel == 0))//reset
           begin
               hex_convert    <= 0;
               rom_addr    <= 0;
           end
           else if(line_disp && pixel_disp)    // within display area
           begin
               hex_convert    <= hex_convert +1;    // move to next
nibble
               rom_addr <= (hex_convert == 1) ? rom_addr + 1: rom_addr;
           end// in disp range
           else // not in disp range;
           begin
               hex_convert    <= hex_convert;
               rom_addr    <= rom_addr;
           end

    end      // seq. logic

    always @ (line or pixel)
```

```
    begin
        if ((line >= ROW_START) && (line < ROW_END))
           line_disp    = TRUE;
       else line_disp    = FALSE;

        // check if column in display range
        if ((pixel >= COL_START) && (pixel < COL_END))
           pixel_disp    = TRUE;
       else pixel_disp     = FALSE;
    end

    always @(data, rom_addr, line_disp, pixel_disp,
                rom_val, nibble, hex_convert)
    begin
        if (line_disp && pixel_disp)
            begin
                nextRom = (hex_convert == 1) ? 1 : 0;
                if (hex_convert)    // on 2nd nibble
                             begin
                                  nibble      = rom_val[3:0];
                             end
                             else //on first nibble
                             begin
                                  nibble      = rom_val[7:4];
                             end
                            case (nibble)
                            NO_DIR:
                                  color = WHITE;
                            DIAG_DOWN:
                                  color = RED;
                            DIAG_UP:
                                  color = GREEN;
                            HORIZONTAL:
                                  color = BLUE;
                            VERTICAL:
                                  color = BLUE;
                            default:
                                  color = BLACK;
                            endcase
           end
         else
            begin
                            nibble      = 0;
               color    = MIT_red;     // border stands out from image
                nextRom = 0;
           end
    end // comb logic

    // assign color outputs
    assign color_out    = color;


endmodule
```

Labkit file (edited portion):

```
//////////////////////////////////////////////////////////////////////
////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module for Lab 4 (Spring
2006)
//
//
// Created: March 13, 2006
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////
////////

//////////////////////////////////////////////////////////////////////
/////
   //
   // Project Components
   //

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

   //
   // Generate a 31.5MHz pixel clock from clock_27mhz
   //

   wire pclk, pixel_clock;


   DCM pixel_clock_dcm (.CLKIN(clock_27mhz), .CLKFX(pclk));
   // synthesis attribute CLKFX_DIVIDE of pixel_clock_dcm is 6
   // synthesis attribute CLKFX_MULTIPLY of pixel_clock_dcm is 7
      // synthesis attribute CLKIN_PERIOD of pixel_clock_dcm is 37
   // synthesis attribute CLK_FEEDBACK of pixel_clock_dcm  is "NONE"
   BUFG pixel_clock_buf (.I(pclk), .O(pixel_clock));

   //
   // VGA output signals
   //

   // Inverting the clock to the DAC provides half a clock period for
signals
   // to propagate from the FPGA to the DAC.
   assign vga_out_pixel_clock = ~pixel_clock;

   // The composite sync signal is used to encode sync data in the
green
   // channel analog voltage for older monitors.  It does not need to
be
   // implemented for the monitors in the 6.111 lab, and can be left at
1'b1.
   assign vga_out_sync_b = 1'b1;
```

```
/*****************************************************/
      wire hsync;
      wire vsync;
      wire [9:0] pi_c;
      wire [8:0] li_c;
      wire hblank, vblank;
      wire hsync_delay, vsync_delay;

   assign vga_out_blank_b = (hblank & vblank);
   assign vga_out_hsync = hsync_delay;
   assign vga_out_vsync = vsync_delay;

      wire reset_all;
      wire reset_sync;
      assign reset_all  = button0;

      wire enter_noisy;
      wire enter_sync;
      assign enter_noisy      = ~button_enter;

      wire [23:0] color;
      assign vga_out_red      = color[23:16] ;
      assign vga_out_green    = color[15:8] ;
      assign vga_out_blue     = color[7:0] ;

      //debounce/synchronize button inputs
      debounce DEB_res(.reset(1'b0), .clock(pixel_clock),
                                          .noisy(reset_all),
.clean(reset_sync)
                                          );

      debounce DEB_enter(.reset(~reset_sync), .clock(pixel_clock),
                                          .noisy(enter_noisy),
.clean(enter_sync)
                                          );

      // display modules
      vga_pulse VGA_controller(.reset(reset_sync), .clock(pixel_clock),
                                                      .vsync(vsync),
.hsync(hsync),

      .pixel_count(pi_c), .line_count(li_c),
                                                      .hblank(hblank),
.vblank(vblank));

      delay Delay_sync(.reset(reset_sync), .vsync(vsync),
.hsync(hsync),
                              .pixel_clock(pixel_clock),
                              .vsync_delay(vsync_delay),
.hsync_delay(hsync_delay) );

      // control minor FSMs, instantiate memories, etc.
      control RUN_FSM(.clk(pixel_clock),
                                    .reset(reset_sync),
.enter(enter_sync),
                                    .pixel(pi_c), .line(li_c),
```

```
                                        .switch(switch[7:0]),
.vga_color(color), .led(led)
                                        );

endmodule
```

Match module: match

```
//////////////////////////////////////////////////////////////////
// Match module:
// function:
//    Divides the screen into quadrants, as below
//     | 0 | 1 |

//     | 2 | 3 |
//    sums the 45 degree direction vectors for each quadrant
//
// inputs:
//    reset
//    clk
//    enable
//    pixel and line counts
//    nibble: direction vector at current pixel location
//
// outputs:
//    busy signal
//    match sums for each quadrant and for + & - 45 degrees
//////////////////////////////////////////////////////////////////

module match(reset, clk, pixel, line, nibble, enable, busy_wire,

                        match0_UP_wire, match0_DOWN_wire,
                        match1_UP_wire, match1_DOWN_wire,
                        match2_UP_wire, match2_DOWN_wire,
                        match3_UP_wire, match3_DOWN_wire, state_wire);
      input reset;
      input clk;
      input [9:0] pixel;
      input [8:0] line;
      input [3:0] nibble;
      input enable;
      output busy_wire;
      output [11:0] match0_UP_wire, match0_DOWN_wire;
      output [11:0] match1_UP_wire, match1_DOWN_wire;
      output [11:0] match2_UP_wire, match2_DOWN_wire;
      output [11:0] match3_UP_wire, match3_DOWN_wire;
      output [1:0] state_wire;

      //size params
    parameter M     = 32;    // 32xN bit image
    parameter N    = 32;    // Mx32 bit image
    parameter scale    = 1;        // zero order hold 10 pixels

    parameter scrn_h    = 480;    // screen is 480 pixels high
    parameter scrn_w    = 640;    // screen is 640 pixels wide

    parameter ROW_START    = scrn_h/2-M*scale/2;
    parameter ROW_END    = scrn_h/2+M*scale/2;
    parameter COL_START    = scrn_w/2-N*scale/2;
    parameter COL_END    = scrn_w/2+N*scale/2;
```

```
   parameter DIAG_DOWN    = 1;
  parameter DIAG_UP       = 2;

parameter TRUE    = 1;
parameter FALSE    = 0;

   reg [1:0] match_loc;
  reg match0_int, match1_int, match2_int, match3_int;
  reg [11:0] match0_UP, match0_DOWN;
  reg [11:0] match1_UP, match1_DOWN;
  reg [11:0] match2_UP, match2_DOWN;
  reg [11:0] match3_UP, match3_DOWN;

  assign match0_UP_wire   = match0_UP;
  assign match1_UP_wire   = match1_UP;
  assign match2_UP_wire   = match2_UP;
  assign match3_UP_wire   = match3_UP;
  assign match0_DOWN_wire = match0_DOWN;
  assign match1_DOWN_wire = match1_DOWN;
  assign match2_DOWN_wire = match2_DOWN;
  assign match3_DOWN_wire = match3_DOWN;

  parameter INIT    = 0;
  parameter ENABLED = 1;
  parameter GO      = 2;
  parameter DONE    = 3;
  reg [1:0] state, next;
  assign state_wire = state;
  reg CLEAR, BUSY;

   always @ (pixel, line)
   begin
        if ((pixel < scrn_w/2) && (line < scrn_h/2))
              match_loc   = 0;
        else if ((pixel >= scrn_w/2) && (line < scrn_h/2))
              match_loc   = 1;
        else if ((pixel < scrn_w/2) && (line >= scrn_h/2))
              match_loc   = 2;
        else
              match_loc   = 3;
   end

   always @ (state, enable, line, pixel, match_loc)
   begin
        case (state)
        INIT:
        begin
              match0_int  = FALSE;
              match1_int  = FALSE;
              match2_int  = FALSE;
              match3_int  = FALSE;
              BUSY  = FALSE;
              CLEAR = FALSE;
              next  = enable ? ENABLED : INIT;
        end // INIT;

        ENABLED:            //pause until pixel, line = 0,0
```

```
            begin
                  match0_int  = FALSE;
                  match1_int  = FALSE;
                  match2_int  = FALSE;
                  match3_int  = FALSE;
                  BUSY  = TRUE;
                  CLEAR = TRUE;
                  next = ((line == 0) && (pixel == 0)) ? GO : ENABLED;
            end // ENABLED

            GO:
            begin
                  case (match_loc)
                  0:
                  begin
                        match0_int  = TRUE;
                        match1_int  = FALSE;
                        match2_int  = FALSE;
                        match3_int  = FALSE;
                  end //0
                  1:
                  begin
                        match0_int  = FALSE;
                        match1_int  = TRUE;
                        match2_int  = FALSE;
                        match3_int  = FALSE;
                  end //1
                  2:
                  begin
                        match0_int  = FALSE;
                        match1_int  = FALSE;
                        match2_int  = TRUE;
                        match3_int  = FALSE;
                  end //2
                  3:
                  begin
                        match0_int  = FALSE;
                        match1_int  = FALSE;
                        match2_int  = FALSE;
                        match3_int  = TRUE;
                  end// 3
                  default:
                  begin
                        match0_int  = FALSE;
                        match1_int  = FALSE;
                        match2_int  = FALSE;
                        match3_int  = FALSE;
                  end  //default
                  endcase
                  BUSY  = TRUE;
                  CLEAR = FALSE;
                  next  =     ((line == ROW_END) && (pixel == COL_END))
? DONE : GO;
            end //GO

            DONE:
            begin
```

```
                match0_int  = FALSE;
                match1_int  = FALSE;
                match2_int  = FALSE;
                match3_int  = FALSE;
                BUSY  = TRUE;
                CLEAR = FALSE;
                next  = INIT;
          end // DONE

          default:
          begin
                match0_int  = FALSE;
                match1_int  = FALSE;
                match2_int  = FALSE;
                match3_int  = FALSE;
                BUSY  = FALSE;
                CLEAR = FALSE;
                next  = INIT;
          end

          endcase // state
     end // state

     always @ (posedge clk)
     begin
          if (!reset)
          begin
                match0_UP   <= 0;
                match0_DOWN <= 0;
                match1_UP   <= 0;
                match1_DOWN <= 0;
                match2_UP   <= 0;
                match2_DOWN <= 0;
                match3_UP   <= 0;
                match3_DOWN <= 0;
                state <= INIT;
          end

          else
          begin
                if (CLEAR)
                begin
                      match0_UP   <= 0;
                      match0_DOWN <= 0;
                      match1_UP   <= 0;
                      match1_DOWN <= 0;
                      match2_UP   <= 0;
                      match2_DOWN <= 0;
                      match3_UP   <= 0;
                      match3_DOWN <= 0;
                end
                else
                begin
                      //check for DIAG_UP
                      match0_UP <= (!match0_int) ? match0_UP :
                                                   (nibble == DIAG_UP) ?
(match0_UP + 1) : match0_UP;
```

```
                        match1_UP <= (!match1_int) ? match1_UP :
                                                     (nibble == DIAG_UP) ?
(match1_UP + 1) : match1_UP;
                        match2_UP <= (!match2_int) ? match2_UP :
                                                     (nibble == DIAG_UP) ?
(match2_UP + 1) : match2_UP;
                        match3_UP <= (!match3_int) ? match3_UP :
                                                     (nibble == DIAG_UP) ?
(match3_UP + 1) : match3_UP;
                        //check for DIAG_DOWN
                        match0_DOWN <= (!match0_int) ? match0_DOWN :
                                                     (nibble == DIAG_DOWN) ?
(match0_DOWN + 1) : match0_DOWN;
                        match1_DOWN <= (!match1_int) ? match1_DOWN :
                                                     (nibble == DIAG_DOWN) ?
(match1_DOWN + 1) : match1_DOWN;
                        match2_DOWN <= (!match2_int) ? match2_DOWN :
                                                     (nibble == DIAG_DOWN) ?
(match2_DOWN + 1) : match2_DOWN;
                        match3_DOWN <= (!match3_int) ? match3_DOWN :
                                                     (nibble == DIAG_DOWN) ?
(match3_DOWN + 1) : match3_DOWN;
                    end
                    state <= next;
                end // no reset
            end// posedge clk

            assign busy_wire = BUSY;
endmodule
```

Instantiate memories: generate_mems

```
//////////////////////////////////////////////////////////
//generate_mems module
//function:
//    instantiates memories
//    passes control signals to the control FSM
//
//inputs/outputs:
//    all of the control signals for:
//          ROM with fingerprint image
//          2 RAMS for edge detection
//          2 RAMS for direction detection
//////////////////////////////////////////////////////////

module generate_mems(
                  clk,
                  rom_addr, rom_data,
                  edge_we,
                  vedge_addr, vedge_din, vedge_dout,
                  hedge_addr, hedge_din, hedge_dout,
                  dir_we,
                  vdir_addr, vdir_din, vdir_dout,
                  hdir_addr, hdir_din, hdir_dout
                  );

      parameter ADDR_BITS     = 13;
      parameter DIR_ADDR_BITS = ADDR_BITS+2;

      input clk;
      input [ADDR_BITS-1:0] rom_addr;
      output [7:0] rom_data;
      input edge_we;
      input [ADDR_BITS-1:0] vedge_addr, hedge_addr;
      input [7:0] vedge_din, hedge_din;
      output [7:0] vedge_dout, hedge_dout;
      input dir_we;
      input [DIR_ADDR_BITS-1:0] vdir_addr, hdir_addr;
      input [7:0] vdir_din, hdir_din;
      output [7:0] vdir_dout, hdir_dout;



      //256x256 binary fingerprint ROM
      finger256rom ROM_finger256(.addr(rom_addr),

      .clk(clk),

      .dout(rom_data)
                                                                  );

      //256x256 binary edge ram - vertical
      finger256ram vert_filt_finger256(
                                    .addr(vedge_addr),      // input
                                    .clk(clk),
```

```
                                    .din(vedge_din),        // input
                                    .dout(vedge_dout),      //output
                                    .we(edge_we)
      //input
                                    );

      //256x256 binary edge ram - horizontal
      finger256ram horiz_filt_finger256(
                                    .addr(hedge_addr),      // input
                                    .clk(clk),
                                    .din(hedge_din),        // input
                                    .dout(hedge_dout),      //output
                                    .we(edge_we)
      //input
                                    );

      //256x256 nibbles direction ram - vertical
      dir_ram vert256_dir(.clk(clk),
                                                .addr(vdir_addr),
      //input
                                                .din(vdir_din),
      //input
                                                .dout(vdir_dout),
//output
                                                .we(dir_we)
       //input
                                                );

      //256x256 nibbles direction ram - horizontal
      dir_ram horiz256_dir(.clk(clk),
                                                .addr(hdir_addr),
                                                .din(hdir_din),
                                                .dout(hdir_dout),
                                                .we(dir_we)
                                                );
endmodule
```

Filter Position for 3x3 and 5x5 Filters: read_loc, read_loc5x5

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////////
//read_loc module
//function:
//    takes in a write location and a position for the filter, and
//    outputs the next location to be read
//    filter is 3x3 indexed as below:
//    | 0 | 1 | 2 |
//    | 3 |N/A| 4 |
//    | 5 | 6 | 7 |
//
//inputs:
//    clk
//    filt_loc    = position of the filter as above
//    pixel_write = current pixel thats being filtered
//
//outputs:
//    read = pixel thats to be read
//////////////////////////////////////////////////////////////////////////////////
//////////

module read_loc(clk, filt_loc, pixel_write, read);
      parameter PIXEL_BITS    = 12;
      parameter ADDR_BITS     = PIXEL_BITS-3;

      input clk;
      input [2:0] filt_loc;
      input [PIXEL_BITS-1:0] pixel_write;
      output [PIXEL_BITS-1:0] read;

      parameter M = 32;
      parameter N = 32;

      reg [PIXEL_BITS-1:0] pixel_read;

      always @ (posedge clk)
      case(filt_loc)
            0:          //top left
            begin
                  pixel_read  = pixel_write-N-1;
            end

            1:          // top center
            begin
                  pixel_read  = pixel_write-N;
            end

            2:          // top right
            begin
                  pixel_read  = pixel_write-N+1;
            end
```

```
            3:          // middle left
            begin
                  pixel_read  = pixel_write-1;
            end

            4:          // middle right
            begin
                  pixel_read  = pixel_write+1;
            end

            5:          // bottom left
            begin
                  pixel_read  = pixel_write+N-1;
            end

            6:          // bottom center
            begin
                  pixel_read  = pixel_write+N;
            end

            7:          // bottom right
            begin
                  pixel_read  = pixel_write+N+1;
            end

            default:
            begin
                  pixel_read  = pixel_write;
            end

      endcase

      assign read = pixel_read;

endmodule

//////////////////////////////////////////////////////////////////////
////////
//read_loc5x5 module
//function:
//    takes in a write location and a position for the filter, and
//    outputs the next location to be read
//    filter is 5x5 indexed as below:
//    | 0 | 1 | 2 | 3 | 4 |
//    | 5 | 6 | 7 | 8 | 9 |
//    | 10| 11| 12| 13| 14|
//    | 15| 16| 17| 18| 19|
//    | 20| 21| 22| 23| 24|
//
//inputs:
//    clk
//    filt_loc    = position of the filter as above
//    pixel_write = current pixel thats being filtered (position 12)
//
//outputs:
//    read = pixel thats to be read
```

```
////////////////////////////////////////////////////////////////////
/////////

module read_loc5x5(clk, filt_loc, pixel, read);
      parameter PIXEL_BITS    = 12;
      parameter ADDR_BITS     = PIXEL_BITS-3;

      input clk;
      input [4:0] filt_loc;
      input [PIXEL_BITS-1:0] pixel;
      output [PIXEL_BITS-1:0] read;

      parameter M = 32;
      parameter N = 32;

      reg [PIXEL_BITS-1:0] pixel_read;

      //N shifts between rows (- for up, + for down)

      always @ (posedge clk)
      case(filt_loc)
            0:          //top left
            begin
                  pixel_read  = pixel-2*N-2;
            end

            1:          // top
            begin
                  pixel_read  = pixel-2*N-1;
            end

            2:          // top center
            begin
                  pixel_read  = pixel-2*N;
            end

            3:          // top
            begin
                  pixel_read  = pixel-2*N+1;
            end

            4:          // top right
            begin
                  pixel_read  = pixel-2*N+2;
            end

            5:          // left
            begin
                  pixel_read  = pixel-N-2;
            end

            6:          //
            begin
                  pixel_read  = pixel-N-1;
            end

            7:          // center
```

```
begin
      pixel_read  = pixel-N;
end

8:          //
begin
      pixel_read  = pixel-N+1;
end

9:          // right
begin
      pixel_read  = pixel-N+2;
end

10:         // middle left
begin
      pixel_read  = pixel-2;
end

11:         //
begin
      pixel_read  = pixel-1;
end

12:         //
begin
      pixel_read  = pixel;
end

13:         //
begin
      pixel_read  = pixel+1;
end

14:         //

begin
      pixel_read  = pixel+2;
end

15:         //
begin
      pixel_read  = pixel+N-2;
end

16:         //
begin
      pixel_read  = pixel+N-1;
end

17:         //
begin
      pixel_read  = pixel+N;
end

18:         //
begin
```

```
                pixel_read  = pixel+N+1;
            end

            19:         //
            begin
                pixel_read  = pixel+N+2;
            end

            20:         //
            begin
                pixel_read  = pixel+2*N-2;
            end

            21:         //
            begin
                pixel_read  = pixel+2*N-1;
            end

            22:         //
            begin
                pixel_read  = pixel+2*N;
            end

            23:         //
            begin
                pixel_read  = pixel+2*N+1;
            end

            24:         //
            begin
                pixel_read  = pixel+2*N+2;
            end

            default:
            begin
                pixel_read  = pixel;
            end

      endcase

      assign read = pixel_read;

endmodule
```

Addressing modules: byte_bit, byte_nibble

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////////
//byte_bit
//function:
//    takes in a 12 bit pixel location value
//    converts that location to a byte address
//    and a bit number inside that byte
//
//inputs:
//    pixel_number      = location of the pixel
//
//outputs:
//    byte = byte address
//    bit    = bit location w/in that byte
//////////////////////////////////////////////////////////////////////////////////
//////////

module byte_bit(pixel_number, byte, bit);
      parameter PIXEL_BITS    = 12;
      parameter ADDR_BITS     = PIXEL_BITS - 3;

      input [PIXEL_BITS-1:0] pixel_number;
      output [ADDR_BITS-1:0] byte;
      output [2:0] bit;

      assign byte = pixel_number/8;
      assign bit  = pixel_number[2:0];
endmodule


//////////////////////////////////////////////////////////////////////////////////
//////////
//byte_nibble
//function:
//    takes in a 12 bit pixel location value
//    converts that location to a byte address
//    and a nibble location inside that byte
//
//inputs:
//    pixel_number      = location of the pixel
//
//outputs:
//    byte = byte address
//    nibble       = nibble location w/in that byte (0 or 1)
//////////////////////////////////////////////////////////////////////////////////
//////////

module byte_nibble(pixel_number, byte, nibble);
      parameter PIXEL_BITS    = 12;
      parameter ADDR_BITS     = PIXEL_BITS -1;

      input [PIXEL_BITS-1:0] pixel_number;
```

```
      output [ADDR_BITS-1:0] byte;
      output nibble;          // either high or low

      assign byte = pixel_number/2;       // drop lowest bit
      assign nibble = pixel_number[0];
endmodule
```

VGA code: vga_pulse

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////
// Company:
// Engineer:
//
// Create Date:    16:04:01 03/12/06
// Design Name:
// Module Name:    vga_pulse
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////
module vga_pulse(reset, clock, vsync, hsync, pixel_count, line_count,
hblank, vblank);
    input reset;
    input clock;
    output vsync;
    output hsync;
    output [9:0] pixel_count; // max val = 640, 10 bits
    output [8:0] line_count;  // max val = 480, 9 bits
       output hblank, vblank;

       reg vsync;
       reg hsync;
       reg [9:0] pixel_count;
       reg [8:0] line_count;
       reg hblank, vblank;

       //reg h_blank;
       //reg v_blank;
       reg [9:0] pixel_clk_count;    // max = 640, 10 bits needed
       reg [8:0] hblank_count;      // max = 480, 9 bits needed

       //define constants

       parameter h_active_region    = 0;
       parameter h_front_porch      = 1;
       parameter h_syncing                 = 2;
       parameter h_back_porch       = 3;

       parameter v_active_region    = 0;
       parameter v_front_porch      = 1;
       parameter v_syncing                 = 2;
```

```
       parameter v_back_porch       = 3;

       // count # pixel_clk cycles
       parameter h_active     = 640;
       parameter h_front      = 16;
       parameter h_back       = 48;
       parameter h_pulse      = 96;

       //count # h_blank cycles
       parameter v_active     = 480;
       parameter v_front      = 11;
       parameter v_back       = 32;
       parameter v_pulse      = 2;

       reg [3:0] h_state;
       reg [3:0] h_next;

       reg [3:0] v_state;
       reg [3:0] v_next;

       reg h_int_reset;        // signal to indicate that the FSM needs
to reset pixel_clk_count
       reg v_int_reset;             // signal to indicate that the FSM
needs to reset hblank_count
       reg v_inc;                        // signal to increment the
count for vert

       always @ (posedge clock)
       begin
            if (!reset) //reset conditions
            begin
                  h_state                       <= h_active_region;
      // goto initial states
                  v_state                       <= v_active_region;
                  pixel_clk_count   <= 0;                  // reset
counters
                  hblank_count            <= 0;
            end // reset conditions

            else  //not in reset
            begin
                  h_state <= h_next;
                  v_state <= v_next;
                  if (h_int_reset)
                        pixel_clk_count         <= 0; // have been
signaled to reset my count
                  else
                        pixel_clk_count         <= pixel_clk_count + 1;
      // increment

                  if (v_int_reset)
                        hblank_count                  <= 0; // have
been signaled to reset count
                  else if (v_inc)
                        hblank_count                  <= hblank_count +
1;          // increment
                  else
```

```
                        hblank_count                  <= hblank_count;

            end // not in reset

      end // always @ reset

      // start HORIZONTAL component
      always @ (h_state, pixel_clk_count, hblank_count)
      begin
            hsync = 1;        // default
            h_int_reset = 0;  // default
            v_inc = 0;
            pixel_count = 0;
            hblank = 0; // default

            case (h_state)

                  h_active_region:  //state 0
                  begin
                        hblank = 1;
                        pixel_count = pixel_clk_count;      // track
pixel location
                        if (pixel_clk_count < (h_active - 1))     //
remain in active region (count til = h_active-1)
                              h_next = h_active_region;

                        else
                        begin
                              h_next = h_front_porch;
                              h_int_reset = 1;
     // reset count
                        end // else done w/ active region

                  end // state 0, h_active_region

                  h_front_porch:          // state 1
                  begin
                        if (pixel_clk_count < (h_front - 1))      //
remain in front porch
                              h_next = h_front_porch;
                        else                                  // goto
sync
                        begin
                              h_next = h_syncing;
                              h_int_reset = 1;              // reset
count
                        end // else goto sync

                  end // state 1, h_front_porch

                  h_syncing:        //state 2
                  begin
                        hsync = 0;
                        if (pixel_clk_count < (h_pulse - 1))      //
remain syncing
                              h_next = h_syncing;
                        else  // go to back porch
```

```
                        begin
                              h_next = h_back_porch;
                              h_int_reset = 1;              //reset
count
                        end // goto back porch

                  end //state 2, h_syncing

                  h_back_porch:           //state 3
                  begin
                        if (pixel_clk_count < (h_back - 1)) // remain
in back porch
                              h_next = h_back_porch;
                        else
                        begin
                              h_next = h_active_region;
                              h_int_reset = 1;        // reset count
                              v_inc = 1;        // increment for the
vert component
                        end // go to active region
                  end // back porch

            endcase
       end// always h_state

  // start VERTICAL component
       always @ (v_state, hblank_count)
       begin
            vsync = 1;        // default
            v_int_reset = 0;  // default
            line_count = 0;
            vblank = 0; // default
            case (v_state)

                  v_active_region:  //state 0
                  begin
                        vblank = 1;
                        if (hblank_count < v_active)  // remain in
active region (count til = v_active-1)
                        begin
                              v_next = v_active_region;
                              line_count = hblank_count;    // track
pixel location
                        end// if v is active

                        else
                        begin
                              v_next = v_front_porch;
                              v_int_reset = 1;
      // reset count
                        end // else done w/ active region

                  end // state 0, v_active_region

                  v_front_porch:          // state 1
                  begin
```

```
                        if (hblank_count < v_front)   // remain in
front porch
                              v_next = v_front_porch;
                        else                                  // goto
sync
                        begin
                              v_next = v_syncing;
                              v_int_reset = 1;              // reset
count
                        end // else goto sync

                  end // state 1, v_front_porch

                  v_syncing:        //state 2
                  begin
                        vsync = 0;
                        if (hblank_count < v_pulse)   // remain syncing
                              v_next = v_syncing;
                        else  // go to back porch
                        begin
                              v_next = v_back_porch;
                              v_int_reset = 1;              //reset
count
                        end // goto back porch

                  end //state 2, v_syncing

                  v_back_porch:           //state 3
                  begin
                        if (hblank_count < v_back)    // remain in back
porch
                              v_next = v_back_porch;
                        else
                        begin
                              v_next = v_active_region;
                              v_int_reset = 1;        // reset count
                        end // go to active region
                  end // back porch

            endcase
      end// always v_state

endmodule
```

Code for Image Acquisition

```
//////////////////////////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
  input vclock;
  output [10:0] hcount;
  output [9:0] vcount;
  output        vsync;
  output        hsync;
  output        blank;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

endmodule // vram_display
*/
```

```
//////////////////////////////////////////////////////////////////////
// parameterized delay line

/*module delayN(clk,in,out);
  input clk;
  input in;
  output out;

  parameter NDELAY = 3;

  reg [NDELAY-1:0] shiftreg;
  wire          out = shiftreg[NDELAY-1];

  always @(posedge clk)
    shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule*/



module zbt_6111_sample(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;

input  mouse_clock, mouse_data, 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

    // 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'b1;
    assign tv_in_fifo_clock = 1'b0;
    assign tv_in_iso = 1'b1;
    //assign tv_in_reset_b = 1'b0;
    assign tv_in_clock = clock_27mhz;//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

/* change lines below to enable ZBT RAM bank0 */
```

```
/*
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;        // clock enable
*/

/* enable RAM pins */

   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_bwe_b = 4'h0;

/**********/

   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;

  //////////////////////////////////////////////////////////////////////////
```

```
// Demonstration of ZBT RAM as video memory

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

    //
// Generate a 31.5MHz pixel clock from clock_27mhz
//

wire pclk, pixel_clock;
DCM pixel_clock_dcm (.CLKIN(clock_27mhz), .CLKFX(pclk));
// synthesis attribute CLKFX_DIVIDE of pixel_clock_dcm is 6
// synthesis attribute CLKFX_MULTIPLY of pixel_clock_dcm is 7
    // synthesis attribute CLKIN_PERIOD of pixel_clock_dcm is 37
// synthesis attribute CLK_FEEDBACK of pixel_clock_dcm  is "NONE"
BUFG pixel_clock_buf (.I(pclk), .O(pixel_clock));
    wire clk = clock_65mhz;


// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
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;

// ENTER button is user reset
wire reset,user_reset, on_off;
debounce db1(power_on_reset, clk, ~button_enter, user_reset);
    debounce db2(power_on_reset, clk, ~button0, on_off);
assign reset = user_reset | power_on_reset;

// display module for debugging

reg [63:0] dispdata;
display_16hex hexdisp1(reset, clk, dispdata,
                    disp_blank, disp_clock, disp_rs, disp_ce_b,
                    disp_reset_b, disp_data_out);

// generate basic XVGA video signals
```

```
 /*  wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);
*/
   // wire up to ZBT ram

   wire [35:0] vram_write_data;
   wire [35:0] vram_read_data;
   wire [18:0] vram_addr;
   wire        vram_we;

   zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
                 vram_write_data, vram_read_data,
                 ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

   // generate pixel value from reading ZBT memory
 /*  wire [7:0]         vr_pixel;
   wire [18:0] vram_addr1;*/

  /* vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
                   vram_addr1,vram_read_data);*/

   // ADV7185 NTSC decoder interface code
   // adv7185 initialization module
   adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
                       .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
                       .tv_in_i2c_clock(tv_in_i2c_clock),
                       .tv_in_i2c_data(tv_in_i2c_data));

   wire [29:0] ycrcb;        // video data (luminance, chrominance)
   wire [2:0] fvh;        // sync for field, vertical, horizontal
   wire       dv;         // data valid

   ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                       .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                       .ycrcb(ycrcb), .f(fvh[2]),
                       .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

   // code to write NTSC data to video memory

wire capture;
onoffhigh onoffhigh1(clk, reset, on_off, capture);


   wire [18:0] ntsc_addr;
```

```
  wire [35:0] ntsc_data;
  wire        ntsc_we;
  ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, ycrcb[29:22],
                   ntsc_addr, ntsc_data, ntsc_we, switch[6], capture);

  // code to write pattern to ZBT memory
  reg [31:0]        count;
  always @(posedge clk) count <= reset ? 0 : count + 1;

  wire [18:0]       vram_addr2 = count[0+18:0];
  wire [35:0]       vpat = ( switch[5] ? {4{count[3+3:3],4'b0}}
                           : {4{count[3+4:4],4'b0}} );

  // mux selecting read/write to memory based on which write-enable is chosen

wire [9:0] hcount;
wire [8:0] vcount;


  wire        sw_ntsc = ~switch[7];
  wire        my_we = sw_ntsc ? (hcount[1:0]==2'd2) : 1;
  wire [18:0]       write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
  wire [35:0]       write_data = sw_ntsc ? ntsc_data : vpat;
      wire [18:0] vram_addr1;

  assign        vram_addr = my_we ? write_addr : vram_addr1;
  assign        vram_we = my_we;
  assign        vram_write_data = write_data;

  // select output pixel data

 /*  reg [7:0]        pixel;
  wire        b,hs,vs;

  delayN dn1(clk,hsync,hs);       // delay by 3 cycles to sync with ZBT read
  delayN dn2(clk,vsync,vs);
  delayN dn3(clk,blank,b);

  always @(posedge clk)
    begin
      pixel <= switch[4] ? {hcount[8:6],5'b0} : vr_pixel;
    end*/

  // VGA Output.  In order to meet the setup and hold times of the
  // AD7125, we send it ~clock_65mhz.
        /*assign vga_out_red = pixel;
```

```
  assign vga_out_green = pixel;
  assign vga_out_blue = pixel;*/
 /* assign vga_out_sync_b = 1'b1;    // not used
  assign vga_out_pixel_clock = ~clock_65mhz;
  assign vga_out_blank_b = ~b;
  assign vga_out_hsync = hs;
  assign vga_out_vsync = vs;*/

//   assign led = ~{vram_addr[18:13],reset,switch[4]};
            assign led = 8'hFF;
/*   always @(posedge clk)
     dispdata <= {vram_read_data,9'b0,vram_addr};
    dispdata <= {ntsc_data,9'b0,ntsc_addr};*/

// for FIFO instantiation

wire wr_en;
wire [18:0] image_addr;
wire [7:0] image;
wire full, done;
wire [7:0] dout;
wire [7:0] image_data;
wire [15:0] image_addr_read;
wire empty_en;
wire [7:0] fifo_dout;

//FSM to write to FIFO RAM
writefifomemory write1(.clk(clk), .reset(reset),
.vram_read_data(vram_read_data[29:22]),
                                        .vram_addr(vram_addr), .capture(capture), .wr_en(wr
_en), .wr_clk(clk),
                                        .image(image), .image_addr(image_addr), .full(full), .
done(done));

//FSM to read from FIFO RAM, send output to write FSM for print RAM
readfifomemory read1(.clk(pixel_clock), .reset(reset), .dout(dout),
                                        .rd_en(rd_en), .rd_clk(rd_clk), .done(done), .image_da
ta(image_data),
                                        .image_addr_read(image_addr_read), .empty_en(empt
y_en));

fifo fifo_mem(.din(image),
                        .wr_en(wr_en),
                        .wr_clk(clk),
                        .rd_en(rd_en),
                        .rd_clk(pixel_clock),
```

```
                        .dout(fifo_dout),
                        .full(full),
                        .ainit(ainit),
                        .empty(empty)
                                        );

wire done_ram;
wire [7:0] image_print;
wire [15:0] image_address;
wire we;
//wire [7:0] dout;
// print written to block RAM


writetoprintRAM print1(.clk(pixel_clock), .reset(reset),
.image_data(image_data),
                                .image_addr(image_addr), .empty_en(empty_en), .don
e_ram(done_ram),
.image_print(image_print),
                                        .image_address(image_address), .we(we));
//check instantiation
print_ram print_ram1 (.din(image_print),        //should input image_print
                                .addr(image_address),
                                .we(we),
                                .dout(dout));

        //////////////////////////////////////////////////
        /* add CTEXIN */
        wire hsync;
        wire vsync;
        wire [9:0] pi_c;
        wire [8:0] li_c;
        wire hblank, vblank;
        wire hsync_delay, vsync_delay;


//added to allow writing to ZBT
        assign hcount = hblank;
        assign vcount = vblank;

   assign vga_out_blank_b = (hblank & vblank);
   assign vga_out_hsync = hsync_delay;
   assign vga_out_vsync = vsync_delay;

        wire reset_all;
        wire reset_sync;
```

```
	assign reset_all	= button0;

	wire enter_noisy;
	wire enter_sync;
	assign enter_noisy	= ~button_enter;

	wire [23:0] color;
	assign vga_out_red	= color[23:16] ;
	assign vga_out_green	= color[15:8] ;
	assign vga_out_blue	= color[7:0] ;
	assign vga_out_sync_b = 1'b1;
	assign vga_out_pixel_clock = ~pixel_clock;

	//debounce/synchronize button inputs
	debounce_ct DEB_res(.reset(1'b0), .clock(pixel_clock),
					.noisy(reset_all), .clean(reset_sync)
					);

	debounce_ct DEB_enter(.reset(~reset_sync), .clock(pixel_clock),
					.noisy(enter_noisy), .clean(enter_sync)
					);

	// display modules
	vga_pulse VGA_controller(.reset(reset_sync), .clock(pixel_clock),
						.vsync(vsync), .hsync(hsync),
						.pixel_count(pi_c), .line_count(li_c),
						.hblank(hblank), .vblank(vblank));

	delay Delay_sync(.reset(reset_sync), .vsync(vsync), .hsync(hsync),
				.pixel_clock(pixel_clock),
				.vsync_delay(vsync_delay), .hsync_delay(hsync_delay) );

	// control minor FSMs, instantiate memories, etc.
	control RUN_FSM(.clk(pixel_clock),
					.reset(reset_sync), .enter(enter_sync),
					.pixel(pi_c), .line(li_c),
					.switch(switch[3:0]), .vga_color(color)
					);
endmodule

//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
```

```
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.

//////////////////////////////////////////////////////////////////////////
// Prepare data and address values to fill ZBT memory with NTSC data

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw,
capture);

  input         clk;       // system clock
  input         vclk;       // video clock from camera
  input [2:0]         fvh;
  input         dv;
  input [7:0]         din;
  output [18:0] ntsc_addr;
  output [35:0] ntsc_data;
  output          ntsc_we;       // write enable for NTSC data
  input         sw;              // switch which determines mode (for debugging)
      input capture;

  parameter         COL_START = 10'd30;
  parameter         ROW_START = 10'd30;

  // here put the luminance data from the ntsc decoder into the ram
  // this is for 1024 x 768 XGA display

  reg [9:0]         col = 0;
  reg [9:0]         row = 0;
  reg [7:0]         vdata = 0;
  reg               vwe;
  reg               old_dv;
  reg               old_frame;       // frames are even / odd interlaced
  reg               even_odd;       // decode interlaced frame to this wire

  wire          frame = fvh[2];
  wire          frame_edge = frame & ~old_frame;

  always @ (posedge vclk) //LLC1 is reference
    begin
      old_dv <= dv;
      vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
      old_frame <= frame;
```

```
    even_odd = frame_edge ? ~even_odd : even_odd;

    if (!fvh[2])
      begin
        col <= fvh[0] ? COL_START :
              (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
        row <= fvh[1] ? ROW_START :
              (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
        vdata <= (dv && !fvh[2]) ? din : vdata;
      end
  end

// synchronize with system clock

reg [9:0] x[1:0],y[1:0];
reg [7:0] data[1:0];
reg       we[1:0];
reg             eo[1:0];

always @(posedge clk)
  begin
    {x[1],x[0]} <= {x[0],col};
    {y[1],y[0]} <= {y[0],row};
    {data[1],data[0]} <= {data[0],vdata};
    {we[1],we[0]} <= {we[0],vwe};
    {eo[1],eo[0]} <= {eo[0],even_odd};
  end

// edge detection on write enable signal

reg old_we;
wire we_edge = we[1] & ~old_we;
always @(posedge clk) old_we <= we[1];

// shift each set of four bytes into a large register for the ZBT

reg [31:0] mydata;
always @(posedge clk)
  if (we_edge)
    mydata <= { mydata[23:0], data[1] };

// compute address to store data in

wire [18:0] myaddr = {1'b0, y[1][8:0], eo[1], x[1][9:2]};

// alternate (256x192) image data and address
```

```
  wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};
  wire [18:0] myaddr2 = {1'b0, y[1][8:0], eo[1], x[1][7:0]};

  // update the output address and data only when four bytes ready

  reg [18:0] ntsc_addr;
  reg [35:0] ntsc_data;
  wire       ntsc_we = sw ? we_edge : (we_edge & (x[1][1:0]==2'b00));

  always @(posedge clk)
    begin
          if (ntsc_we && capture)
                ntsc_addr <= sw ? myaddr2 : myaddr;       // normal and expanded
modes
                ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};
    end
/*    if (ntsc_we && on_off) // ntsc is ready to write pixel and on_off
indicates streaming
    begin
          // tell zbt to write the value data at address a
                ntsc_addr <= sw ? myaddr2 : myaddr;       // normal and expanded
modes
                ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};
    end
        else // ntsc is ready to write pixel and on_off indicates store
        begin
          // tell zbt to hold value, no writes....
                ntsc_addr <= sw ? myaddr2 : myaddr;       // normal and expanded
modes
                ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};
        end
*/
endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Create Date:    15:36:07 05/15/06
// Module Name:    onoffhigh
// Project Name:   fingerprint7
// Description: keeps capture always high
//////////////////////////////////////////////////////////////////////////////////
module onoffhigh(clk, reset, on_off, capture);

input clk, reset, on_off;

output capture;
```

```
reg capture;

always @ (posedge clk)
	begin
		if (reset)
			capture <= 1;
		if (on_off)
			capture <= 0;
	end

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//reads from the FIFO
//////////////////////////////////////////////////////////////////////////////////
module readfifomemory(clk, reset, dout, rd_en, rd_clk, done, image_data,
image_addr_read, empty_en);

input clk, reset, done;
input [7:0] dout;


output rd_en, rd_clk, empty_en;
output [7:0] image_data;
output [15:0] image_addr_read;

parameter IDLE = 0;
parameter READ1 = 1;
parameter READ2 = 2;
parameter READ3 = 3;

parameter M 	= 256;
parameter N 		= 256;

reg [2:0] state, nextstate;
reg rd_enable;
reg [7:0] data;
reg [15:0] image_addr_count;
reg [7:0] image_data;
reg [15:0] image_addr;
reg rd_en;
reg empty_enable, empty_en;
```

```
always @ (posedge clk)
	begin
		if (reset)
		begin
			image_data <= 8'b00000000;
			image_addr <= 16'd0;
			rd_en <= 0;
			empty_en <= 0;
			state <= IDLE;
		end
		else
		begin
			image_data <= data;
			image_addr <= image_addr + image_addr_count;
			rd_en <= rd_enable;
			empty_en <= empty_enable;
			state <= nextstate;
		end
	end

always @ (done or image_addr)
	begin
		image_addr_count = 0;
		data = 8'd0;
		rd_enable = 0;
		case(state)
			IDLE: nextstate = done ? READ1: IDLE;
			READ1: begin
					nextstate = READ2;
					data = dout;
					image_addr_count = 1;
					rd_enable = 1;
				end
			READ2: begin
					nextstate = READ3;
					data = dout;
					image_addr_count = 1;
					rd_enable = 1;
				end
			READ3: begin
				if (image_addr == M*N-1)
					begin
					nextstate = IDLE;
					data = dout;
					image_addr_count = 1;
					rd_enable = 1;
```

```
                                    empty_enable = 1;
                                    end
                                else
                                    begin
                                    nextstate = READ1;
                                    data = dout;
                                    image_addr_count = 1;
                                    rd_enable = 1;
                                    end
                                end
                        endcase
                end

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//reads from ZBT memory and puts it into an asynchronous FIFO
//////////////////////////////////////////////////////////////////////////////////
module writefifomemory(clk, reset, vram_read_data, vram_addr, capture, wr_en,
wr_clk,
                                image, /*image_addr,*/ full, done);

input clk, reset;
input [7:0] vram_read_data;
input [15:0] vram_addr;
input capture, full;

output wr_en, wr_clk, done;
output [7:0] image;
//output [15:0] image_addr;

parameter IDLE = 0;
parameter WRITE1 = 1;
parameter WRITE2 = 2;
parameter WRITE3 = 3;

reg [1:0] state, nextstate;
reg wr_enable, done;
reg [7:0] pixel_four;
reg [2:0] image_count;
reg [7:0] image;
reg [15:0] image_addr;
reg wr_en;

always @ (posedge clk)
```

```
	begin
		if (reset)
		begin
			image <= 8'b00000000;
			image_addr <= 16'd0;
			wr_en <= 0;
			state <= IDLE;
		end
		else
		begin
			image <= pixel_four;
			image_addr <= vram_addr + image_count;
			wr_en <= wr_enable;
			state <= nextstate;
		end
	end

always @ (capture, state, vram_read_data, full)
	begin
		image_count = 16'd0;
		pixel_four =  8'd0;
		wr_enable = 0;
		done = 0;
		case(state)
			IDLE: nextstate = capture ? WRITE1: IDLE;
			WRITE1: begin
					nextstate = WRITE2;
					pixel_four = vram_read_data;
					image_count = 8'b0000_0100;
					wr_enable = 1;
				end
			WRITE2: begin
					pixel_four = vram_read_data;
					nextstate = WRITE3;
					wr_enable = 1;
				end
			WRITE3: begin
					if (full)
						begin
							pixel_four = vram_read_data;
							wr_enable = 1;
							done = 1;
							nextstate = IDLE;
						end
					else
						begin
```

```
                                        pixel_four = vram_read_data;
                                        wr_enable = 1;
                                        nextstate = WRITE1;
                                        end
                                end
                        endcase
                end
endmodule


`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//writes to the print RAM
//////////////////////////////////////////////////////////////////////////////////

module writetoprintram(clk, reset, image_data, image_addr, empty_en,
done_ram_wire, image_print_wire,
                                        image_address_wire, we_wire);

input clk, reset;
input [7:0] image_data;
input [15:0] image_addr;
input empty_en;

output done_ram_wire, we_wire;
output [7:0] image_print_wire;
output [15:0] image_address_wire;

parameter IDLE = 0;
parameter WRITE1 = 1;
parameter WRITE2 = 2;
parameter WRITE3 = 3;

reg [2:0] state, nextstate;
reg done_ram;
reg image_addr_enable;
reg [7:0] image_print;
reg [15:0] image_address;
reg we;
reg [7:0] image_print_temp;
reg write_enable;
assign done_ram_wire        = done_ram;
assign we_wire        = we;
assign image_print_wire        = image_print;
assign image_address_wire        = image_address;
always @ (posedge clk)
```

```
	begin
		if (reset)
		begin
			image_print <= 8'd0;
			image_address <= 16'd0;
			we <= 1;
			state <= IDLE;
		end
		else
		begin
			image_print <= image_print_temp;
			image_address <= image_addr + image_addr_enable;
			we <= write_enable;
			state <= nextstate;
		end
	end

always @ (empty_en, state, image_data)
	begin
		image_addr_enable = 0;
		done_ram = 0;
		image_print_temp = 8'd0;
		write_enable = 1;
		case(state)
			IDLE: begin
					if (empty_en)
						begin
							nextstate = IDLE;
							done_ram = 0;
						end
					else
						begin
							nextstate = WRITE1;
						end
				end
			WRITE1: begin
						nextstate = WRITE2;
						image_print_temp = image_data;
						image_addr_enable = 1;
						write_enable = 0;
					end
			WRITE2: begin
						nextstate = WRITE3;
						image_print_temp = image_data;
						image_addr_enable = 1;
						write_enable = 0;
```

```
                    end
            WRITE3: begin
                    nextstate = IDLE;
                    image_print_temp = image_data;
                    image_addr_enable = 1;
                    write_enable = 0;
                    end
            endcase
        end

endmodule
```