## A Labkit Listing

```
module labkit (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;
```

```
// 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
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;

// 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;

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;

// 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;

// Buttons, Switches, and Individual LEDs
assign led = 8'hFF;

// 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;

// Logic Analyzer
//assign analyzer1_data = 16'h0;
assign analyzer1_clock = 1'b1;
//assign analyzer2_data = 16'h0;
//assign analyzer2_clock = clock_64mhz;
//assign analyzer3_data = 16'h0;
```

```
   assign analyzer3_clock = 1'b1;
   //assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

   ////////////////////////////////////////////////////////////////////////
   // Fingerprint ID
   ////////////////////////////////////////////////////////////////////////

   // use FPGA's digital clock manager to produce a
   // 31.5 MHz pixel clock from clock_27mhz                // 15, 14
   wire clock_32mhz_unbuf,clock_32mhz;
   DCM pxl_clk_dcm (.CLKIN(clock_27mhz), .CLKFX(clock_32mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of pxl_clk_dcm is 15
   // synthesis attribute CLKFX_MULTIPLY of pxl_clk_dcm is 14
   // synthesis attribute CLK_FEEDBACK of pxl_clk_dcm  is "NONE"
// synthesis attribute CLKIN_PERIOD of pxl_clk_dcm  is 37
   BUFG pxl_clk_buf (.O(clock_32mhz), .I(clock_32mhz_unbuf));
   wire pxl_clk = clock_32mhz;

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

   // main 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;

   // button0 is reset
   wire reset,user_reset;
   debounce db1(user_reset, clk, ~button0, user_reset);
   assign reset = user_reset | power_on_reset;

   // generate basic VGA video signals
   // delay for memory delay shift
   wire hs_out, vs_out, bb_out;
   reg hs, hs1, hs2, hs3, hs4;
   reg vs, vs1, vs2, vs3, vs4;
   reg bb, bb1, bb2, bb3, bb4;
   always @ (posedge pxl_clk) begin
      hs <= hs4; /*hs1 <= hs2; hs2 <= hs3; hs3 <= hs4;*/ hs4 <= hs_out;
      vs <= vs4; /*vs1 <= vs2; vs2 <= vs3; vs3 <= vs4;*/ vs4 <= vs_out;
      bb <= bb4; /*bb1 <= bb2; bb2 <= bb3; bb3 <= bb4;*/ bb4 <= bb_out;
   end
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;
   assign vga_out_blank_b = bb;
   // 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 = ~pxl_clk;
   wire [9:0] h_cnt, v_cnt;
   wire hsync,vsync,blank;
   vga vga1(reset,hs_out,vs_out,vga_out_sync_b,bb_out,h_cnt,v_cnt,pxl_clk);

   // 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 [13:0] bram_addr;
   wire [7:0]  bram_data;
   wire        bram_we, frame_clk;
   ntsc_to_bram n2ram (tv_in_line_clock1,fvh,dv,ycrcb[29:22],
                 bram_addr, bram_data, bram_we, frame_clk,
                       video_clk, reset);


   // START is button_enter
```

```
   wire start, start_sync;
   debounce db2(power_on_reset, clk, ~button_enter, start_sync);
   pulse pls1(power_on_reset, clk, start_sync, start, 4'b0);

   //wire add_user_in;
   //debounce db3(power_on_reset, clk, switch[7], add_user_in);

   // USER ID is switch[2:0]
   // ADD_USER is switch[7]
   // controller
   wire [2:0] state;
   wire acquire;
   wire [2:0] user_id;
   wire add_user;
   wire process_start, process_busy,
        validate_start, validate_busy,
        validate_result;
   wire [2:0] match_id;
   controller control( .start(start),
                       .disp_state(state),
                       .add_user( switch[7] ),
                       .add_user_latched(add_user),
                       .user_id( switch[2:0] ),
                       .user_id_latched(user_id),
                       .acquire(acquire),
                       .process_start( process_start ),
                       .validate_start( validate_start ),
                       .validate_result( validate_result ),
                       .busy( process_busy | validate_busy ),
  .clock( clk ),
  .reset(reset) );


   // ~30 Hz frame clock
   //wire frame_clk;
   //divider mk_frame_clk(frame_clk, 4'd3, clk, reset);

   // display with video memory
wire proc_disp_cycle, proc_disp_we;
   wire [13:0] proc_disp_addr;
   wire [7:0] proc_disp_pxl;
   reg disp_cycle_source, disp_we;
   reg [13:0] disp_addr;
   reg [7:0] disp_pxl_in;
   always @ (posedge clk) begin
```

```
      disp_cycle_source <= acquire ? frame_clk : proc_disp_cycle;
      disp_addr <= acquire ? bram_addr : proc_disp_addr;
      disp_pxl_in <= acquire ? bram_data : proc_disp_pxl ;
      disp_we <= acquire ? bram_we : proc_disp_we;
   end
   DisplayField display(
/* Memory Access */
.cycle_source( disp_cycle_source ),
      .addr( disp_addr ),
      .pxl_in( disp_pxl_in ),
      .we( disp_we ),
/* Normal VGA Interface */
.pixel(h_cnt), .line(v_cnt),
      .red(vga_out_red), .green(vga_out_green), .blue(vga_out_blue),
      .clock( clk ), .reset( reset )
);

   // Original Image Storage
   reg scan_img_we;
   reg [7:0] scan_img_write_pxl;
   reg [13:0] scan_img_addr;
   always @ (posedge clk) begin
      scan_img_we <= bram_we & acquire;
      scan_img_write_pxl <= bram_data;
      scan_img_addr <= bram_addr;
   end
   wire scan_img_re;
   wire [7:0] scan_img_read_pxl;
   wire scan_img_rst_pxl;
   wire scan_img_en_seq;
   img_buffer scanned_img(
      .we( scan_img_we ), .write_pxl( scan_img_write_pxl ), .addr( scan_img_addr ),
      .re( scan_img_re ), .read_pxl( scan_img_read_pxl ),
      .reset_pxl_pos( scan_img_rst_pxl ), .enable_seq_access( scan_img_en_seq ),
.en(1'b1), .reset(reset), .clock(clk)
);

   // And we're off!
   wire we_row_coeff, fft_row_dv,
        we_col_coeff, fft_col_dv;
   wire [31:0] row_coeff, col_coeff;
   process_image process( process_start, process_busy,
      scan_img_rst_pxl, scan_img_re, scan_img_read_pxl, scan_img_en_seq,
      proc_disp_cycle, proc_disp_addr, proc_disp_pxl, proc_disp_we,
      we_row_coeff, fft_row_dv, row_coeff,
      we_col_coeff, fft_col_dv, col_coeff,
      reset, clk,
      sel_data_begin, sel_data_in, sel_row_data_out, sel_col_data_out, sel_row_we, sel_col_we )

   assign analyzer2_data = { 6'b0,
                             we_row_coeff,
                             fft_row_dv,
                             we_col_coeff,
                             fft_col_dv,
                             sel_row_we,
                             sel_col_we,
                             sel_data_begin,
                             sel_data_in,
                             sel_row_data_out,
                             sel_col_data_out};

   assign analyzer1_data = row_coeff[15:0];
   assign analyzer3_data = col_coeff[15:0];
   assign analyzer2_clock = clk;

   // Multiplexer for database storage
   wire [31:0] rowA_data, rowB_data, colA_data, colB_data;
   wire rowA_we,   rowB_we,   colA_we,   colB_we;
   wire [3:0] rowA_addr, colA_addr;
   wire [6:0] rowB_addr, colB_addr;
   fft_coeff_mux row_mux( .clock(clk),.reset(reset),
      // Selection Criterion
      .add_user(add_user), .user_id(user_id),
      // Input Signals
      .in_we( we_row_coeff ), .in_rst( fft_row_dv ), .in_data( row_coeff ),
      // Routed Signals (Output)
      .bufA_data( rowA_data ),.bufA_we( rowA_we ),.bufA_addr( rowA_addr ),
      .bufB_data( rowB_data ),.bufB_we( rowB_we ),.bufB_addr( rowB_addr ) );
   fft_coeff_mux col_mux( .clock(clk),.reset(reset),
      // Selection Criterion
      .add_user(add_user), .user_id(user_id),
      // Input Signals
      .in_we( we_col_coeff ), .in_rst( fft_col_dv ), .in_data( col_coeff ),
      // Routed Signals (Output)
      .bufA_data( colA_data ),.bufA_we( colA_we ),.bufA_addr( colA_addr ),
      .bufB_data( colB_data ),.bufB_we( colB_we ),.bufB_addr( colB_addr ) );


   assign analyzer4_data = {2'b0, process_busy, add_user,
                            rowA_we, rowB_we, colA_we, colB_we,
                            rowA_addr,
                            colA_addr };

   // THRESHOLD up is button_up
   // THRESHOLD down is button_down
   // THRESHOLD shift left is button_left
   // THRESHOLD shift right is button_right
   // VIEW THRESHOLD is switch[6]
   wire threshold_up, threshold_down, threshold_sl, threshold_sr,
        t_up_sync, t_down_sync, t_sl_sync, t_sr_sync;
   wire disp_threshold = switch[6];
   debounce db_tup(reset, clk, ~button_up, t_up_sync);
   pulse pls_tup(reset, clk, t_up_sync, threshold_up, 4'd5);
   debounce db_tdown(reset, clk, ~button_down, t_down_sync);
   pulse pls_tdown(reset, clk, t_down_sync, threshold_down, 4'd5);
   debounce db_tsl(reset, clk, ~button_left, t_sl_sync);
   pulse pls_tsl(reset, clk, t_sl_sync, threshold_sl, 4'd10);
   debounce db_tsr(reset, clk, ~button_right, t_sr_sync);
   pulse pls_tsr(reset, clk, t_sr_sync, threshold_sr, 4'd10);

   reg [63:0] threshold;
   always @ (posedge clk)
      if (reset) threshold <= 20000;
      else if (threshold_up) threshold <= threshold + 1;
      else if (threshold_down) threshold <= threshold - 1;
      else if (threshold_sl) threshold <= threshold + 1;
      else if (threshold_sr) threshold <= threshold - 1;
      else threshold <= threshold;


   // Validation Unit with Database
   validation validate(.clock(clk), .reset(reset),
      .compare(validate_start), .busy( validate_busy ),
      .match(validate_result), .img_num( match_id ),
      .din_row_A( rowA_data ), .din_row_B( rowB_data ),
      .we_row_A( rowA_we ), .we_row_B( rowB_we ),
      .row_addr_A( rowA_addr ), .row_addr_B( rowB_addr ),
      .din_col_A( colA_data ), .din_col_B( colB_data ),
      .we_col_A( colA_we ), .we_col_B( colB_we ),
      .col_addr_A( colA_addr ), .col_addr_B( colB_addr ),
      .threshold( threshold), .add_user( add_user & process_start ), .user_id( user_id ) );

   //assign validate_result = switch[6];
   //assign match_id = switch[5:3];
```

```
   // Alphanumeric Display
   // clock_27mhz, ascii, bits, dots
   wire [639:0] dots1, dots2;
   display_state led_matrix1(state,(state == 4)?match_id:user_id,clock_27mhz,dots1);
   display_threshold led_matrix2( threshold, clock_27mhz, dots2 );
   alphanumeric_displays led_matrix_driver(clock_27mhz, reset,
      disp_test, disp_blank, disp_clock, disp_rs, disp_ce_b,
      disp_reset_b, disp_data_out, disp_threshold ? dots2 : dots1);


endmodule
```

## B Controller Module

```
/************************************************************
 * CONTROLLER MODULE
 *
 * This is the main module of the system. It functions as
 * a major FSM.
 *
 * Its main functions are:
 *   - User interaction
 *   - Data flow
 *   - Sytem output
 *
 * MODES of OPERATION
 *
 *   - Idle (waiting for user stimulus)
 *
 ************************************************************/

module controller( // User Interface
                        start,
                        disp_state,
                        add_user,
                        add_user_latched,
                        acquire,
                        user_id,
                        user_id_latched,
                   // Processing
                        process_start,
                   // Validation
                        validate_start,
                        validate_result,
                   // System
```

```
                        busy,
clock,
reset );

   //////////////////////////////////
   // Interface Pins
   //////////////////////////////////
   // User Interaction
   input start;            // pulsed input
   input add_user;         // mode: 0 - validate, 1 - add user
   output add_user_latched; // output
   output [2:0] disp_state;// for display
   output acquire;         // enable movie mode during acquire
   input [2:0] user_id;     // input
   output [2:0] user_id_latched; // output
   // Process
   output process_start;   // start processing
   // Validation
   output validate_start;  // start validation
   input validate_result;  // 0 - no match, 1 - match found
   // System
   input busy;             // something is busy (input)
   input clock;
   input reset;

   ////////////////////////////////////
   // Registers
   ////////////////////////////////////
// REGISTERED OUTPUTS
reg [2:0] disp_state;
   reg acquire,
       process_start,
       validate_start;
   reg [2:0] user_id_latched;
   // INTERNAL REGS
reg [2:0] state, next, waitnext;
   reg add_user_latched;
   reg [31:0] disp_result_pause;

   ////////////////////////////////////
   // Parameters
   ////////////////////////////////////
   // STATE
parameter ACQUIRE = 0;     // IDLE
   parameter SYS_RESET = 1;
parameter WAIT = 2;
parameter PROCESS = 3;
   parameter VALIDATE = 4;
   parameter DISP_RESULT = 5;

   ////////////////////////////////////
   // Code
   ////////////////////////////////////
always @ (posedge clock) begin

      if (reset) state <= ACQUIRE;
      else state <= next;

      acquire           <= (next == ACQUIRE);
      process_start     <= (next == PROCESS);
      validate_start    <= (next == VALIDATE);
      add_user_latched  <= (next == ACQUIRE)     ? (add_user) : add_user_latched;
      disp_result_pause <= (next == ACQUIRE)     ? 0 :
                           (next == DISP_RESULT) ? disp_result_pause + 1 :
                            disp_result_pause;
      user_id_latched <=  (next == ACQUIRE) ? user_id : user_id_latched;
      case (state)
         ACQUIRE: if (~add_user) disp_state <= 0;  // 'identify      '
                  else disp_state <= 1;            // 'add user     #'
         SYS_RESET: disp_state <= 7;               // (blank)
         WAIT: disp_state <= 2;                    // 'processing    '
         PROCESS: disp_state <= 2;                 // 'processing    '
         VALIDATE: disp_state <= 2;                // 'processing    '
         DISP_RESULT:
            if (add_user_latched) disp_state <= 3;      // 'added user   #'
            else if (validate_result) disp_state <= 4;  // 'matched user #'
            else disp_state <= 5;                       // 'no match      '
      endcase
      case (state)
         WAIT: waitnext <= waitnext;
         PROCESS: waitnext <= (add_user ? DISP_RESULT : VALIDATE);
         VALIDATE: waitnext <= DISP_RESULT;
         default: waitnext <= waitnext;
      endcase

   end

   always @ (state or start or busy or disp_result_pause)
      case (state)
         ACQUIRE:
            if (start) next = PROCESS;
            else next = ACQUIRE;
         SYS_RESET:
            next = ACQUIRE;
         WAIT:
            if (busy) next = WAIT;
            else next = waitnext;
         PROCESS:
            next = WAIT;
         VALIDATE:
            next = WAIT;
         DISP_RESULT:
            if (disp_result_pause > 32'h4FFFFFF) next = ACQUIRE;
            else next = DISP_RESULT;
         default:
            next = ACQUIRE;
      endcase
endmodule
```

## C Image Capture

```
`timescale 1ns / 1ps

/****************************************************************************
 *  NTSC to Block RAM (128x128x8)
 *
 *
 *
 *
 *
 ****************************************************************************/

module ntsc_to_bram( vclk,
                     fvh,
                     dv,
                     din,

                     bram_addr,
                     bram_data,
                     bram_we,
                     frame_clk,

                     clk,
                     reset );
```

```
// System
input clk;
input reset;
// Video Input
input vclk; // video clock from camera
input [2:0] fvh;
input dv;
input [7:0] din;
// Block RAM Output
output [13:0] bram_addr;
output [7:0] bram_data;
output bram_we;
output frame_clk;


///////////////////////////////////////////////////////
// Create Intermediate Output Signals (Unsyncronized)
///////////////////////////////////////////////////////

parameter   COL_START = 0;
parameter   ROW_START = 0;

reg [9:0]   col = 0, row = 0;   // intermediate address
reg [7:0]   vdata = 0;          // data
reg vwe;                         // write enable
reg old_dv;
reg old_frame; // frames are even / odd interlaced
reg even_odd; // decode interlaced frame to this wire
reg frame_clk; // update display per frame clock

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

always @ (posedge vclk) 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;
   frame_clk <= frame_edge;
if (!fvh[2]) begin
      // if horizontial blank, reset col count
      // else if not a vertical blank and data is valid and col < 720, increment col
      // else col <= col
      col <= fvh[0] ? COL_START :
       (!fvh[1] && dv && (col < 720)) ? col + 1 :
              col;
   // if vertical blank, reset row count
      // else if horizontial blank and row < 525, increment row
      // else row <= row
      row <= fvh[1] ? ROW_START :
       (!fvh[2] && fvh[0] && (row < 525)) ? row + 1 :
              row;
   // if data valid and not new frame, update vdata
      // else vdata
      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];

 // buffer the signals
 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

 // we only write the data in the buffer
 // if the vwe was enabled, meaning the
 // data was valid.
 // edge detection on write enable signal
 reg old_we;
 wire we_edge = we[1] & ~old_we;
 always @(posedge clk) old_we <= we[1];

 parameter V_OFFSET = 24;
 wire [6:0] addr_row = (y[1][6:0]-V_OFFSET);
 wire [6:0] addr_col = x[1][7:1]-4;
 reg [13:0] bram_addr;
 reg [7:0]  bram_data;
 reg bram_we;
```

```
   always @(posedge clk) begin
      // sample every other pixel by dropping
      // LSB from y and x.
   bram_addr <= {addr_row, addr_col};
   bram_data <= data[1];
      bram_we = we_edge &
                (x[1] > 2) & (x[1] < 258) &
                (y[1] < 128+V_OFFSET) & (y[1] > V_OFFSET);
   end


endmodule
```

## D VGA Display

```
`timescale 1ns / 1ps

module DisplayField( /* INPUTS */ pixel, line,
                                  cycle_source,
                                  addr, pxl_in, we,
   /* OUTPUT */ red, green, blue,
                     /* SYSTEM */ clock, reset );

parameter PXL_WIDTH = 8-1;
   parameter NUM_PIXELS = 16384-1;
   parameter ADDR_WIDTH = 14-1;
   parameter ROWS = 128;
   parameter ROW_WIDTH = 7-1;

   input [9:0] pixel, line;
   input [ADDR_WIDTH:0] addr;
   input [PXL_WIDTH:0] pxl_in;
   input we;
   input cycle_source;
output [7:0] red, green, blue;
   input clock, reset;

// Registered Outputs
reg [7:0] red, green, blue;

   // Internal
wire [PXL_WIDTH:0] color;

// Screen Diameters
```

```
parameter WIDTH = 640;
parameter HEIGHT = 480;


   // Double Buffered Data Source
   double_buffer source (
.re( 1'b0 ),
.we( we ),
.read_addr( {line[ROW_WIDTH:0], pixel[ROW_WIDTH:0]} ),
.write_addr( addr ),
.write_pxl( pxl_in ),
.read_pxl( color ),
.en_seq_read( 1'b0 ),
.en_seq_write( 1'b0 ),
.reset_write_pos( 1'b0 ),
.reset_read_pos( 1'b0 ),
.swap( cycle_source ),
.en( 1'b1 ),
.reset(reset),
.clock(clock)
);
   defparam source.PXL_WIDTH = PXL_WIDTH;
   defparam source.NUM_PIXELS = NUM_PIXELS;
   defparam source.ADDR_WIDTH = ADDR_WIDTH;

always @ (pixel or line) begin
      if (pixel == 255 + 63 && line == 127 + 63)
         {red, green, blue} = {255, 0, 255};
      else if (255 < pixel && pixel < 383 &&
               127 < line && line < 255)
          if (color)
            {red, green, blue} = {color,color,color};
          else
            {red, green, blue} = {0, 0, 0};
      else
   {red, green, blue} = {0, 0, 0};
end


endmodule
```

## E LED Display

### E.1 Display State

```
`timescale 1ns / 1ps

module display_state( state,
                      userID,
                      clock,
                      dots );

   input [2:0] state;
   input [2:0] userID;
   input clock;
   output [639:0] dots;


   // state              display
   // --------------------------------
   // 0                  'Identify     '
   // 1                  'Add user    #'
   // 2                  'Processing   '
   // 3                  'added user  #'
   // 4                  'matched     #'
   // 5                  'no match     '
   // 6                  (not used)
   // 7                  '             ' (blank)

   wire [7:0] char15 = (state == 0) ? 8'h49 :         // I
                       (state == 1) ? 8'h41 :         //   A
                       (state == 2) ? 8'h50 :         //     P
                       (state == 3) ? 8'h41 :         //       A
                       (state == 4) ? 8'h4D :         //         M
                       (state == 5) ? 8'h4E :         //           N
                       (state == 6) ? 8'h00 :         //             _
                       8'h00;
   wire [7:0] char14 = (state == 0) ? 8'h64 :         // d
                       (state == 1) ? 8'h64 :         //   D
                       (state == 2) ? 8'h72 :         //     r
                       (state == 3) ? 8'h64 :         //       d
                       (state == 4) ? 8'h61 :         //         a
                       (state == 5) ? 8'h6F :         //           o
                       (state == 6) ? 8'h00 :         //             _
                       8'h00;
   wire [7:0] char13 = (state == 0) ? 8'h65 :         // e
                       (state == 1) ? 8'h64 :         //   D
```

```
                     (state == 2) ? 8'h6F :          //     o
                     (state == 3) ? 8'h64 :          //       d
                     (state == 4) ? 8'h74 :          //         t
                     (state == 5) ? 8'h00 :          //           _
                     (state == 6) ? 8'h00 :          //             _
                     8'h00;
wire [7:0] char12 = (state == 0) ? 8'h6E :          // n
                     (state == 1) ? 8'h00 :          //   _
                     (state == 2) ? 8'h63 :          //     c
                     (state == 3) ? 8'h65 :          //       e
                     (state == 4) ? 8'h63 :          //         c
                     (state == 5) ? 8'h4D :          //           M
                     (state == 6) ? 8'h00 :          //             _
                      8'h00;
wire [7:0] char11 = (state == 0) ? 8'h74 :          // t
                     (state == 1) ? 8'h55 :          //   U
                     (state == 2) ? 8'h65 :          //     e
                     (state == 3) ? 8'h64 :          //       d
                     (state == 4) ? 8'h68 :          //         h
                     (state == 5) ? 8'h61 :          //           a
                     (state == 6) ? 8'h00 :          //             _
                      8'h00;
wire [7:0] char10 = (state == 0) ? 8'h69 :          // i
                     (state == 1) ? 8'h73 :          //   S
                     (state == 2) ? 8'h73 :          //     s
                     (state == 3) ? 8'h00 :          //       _
                     (state == 4) ? 8'h65 :          //         e
                     (state == 5) ? 8'h74 :          //           t
                     (state == 6) ? 8'h00 :          //             _
                     8'h00;
wire [7:0] char09 = (state == 0) ? 8'h66 :          // f
                     (state == 1) ? 8'h65 :          //   e
                     (state == 2) ? 8'h73 :          //     s
                     (state == 3) ? 8'h55 :          //       U
                     (state == 4) ? 8'h64 :          //         d
                     (state == 5) ? 8'h63 :          //           c
                     (state == 6) ? 8'h00 :          //             _
                     8'h00;
wire [7:0] char08 = (state == 0) ? 8'h79 :          // y
                     (state == 1) ? 8'h72 :          //   R
                     (state == 2) ? 8'h69 :          //     i
                     (state == 3) ? 8'h73 :          //       s
                     (state == 4) ? 8'h00 :          //         _
                     (state == 5) ? 8'h68 :          //           h
                     (state == 6) ? 8'h00 :          //             _
```

```
                    8'h00;
wire [7:0] char07 = (state == 0) ? 8'h00 :          // _
                    (state == 1) ? 8'h00 :          //   _
                    (state == 2) ? 8'h6E :          //     n
                    (state == 3) ? 8'h65 :          //       e
                    (state == 4) ? 8'h00 :          //         _
                    (state == 5) ? 8'h00 :          //           _
                    (state == 6) ? 8'h00 :          //             _
                    8'h00;
wire [7:0] char06 = (state == 0) ? 8'h00 :          // _
                    (state == 1) ? 8'h00 :          //   _
                    (state == 2) ? 8'h67 :          //     g
                    (state == 3) ? 8'h72 :          //       r
                    (state == 4) ? 8'h00 :          //         _
                    (state == 5) ? 8'h00 :          //           _
                    (state == 6) ? 8'h00 :          //             _
                    8'h00;
wire [7:0] char05 = (state == 0) ? 8'h00 : 8'h00;  //
wire [7:0] char04 = (state == 0) ? 8'h00 : 8'h00;  //
wire [7:0] char03 = (state == 0) ? 8'h00 : 8'h00;  //
wire [7:0] char02 = (state == 0) ? 8'h00 : 8'h00;  //
wire [7:0] char01 = (state == 0) ? 8'h00 : 8'h00;  //
wire [7:0] char00 = (state == 1 ||
                     state == 3 ||
                     state == 4 )  ? {5'b0,userID} : 8'h00; // [user ID]


alpha_display af15(clock, 1'b1, char15, dots[639:600]);
alpha_display af14(clock, 1'b1, char14, dots[599:560]);
alpha_display af13(clock, 1'b1, char13, dots[559:520]);
alpha_display af12(clock, 1'b1, char12, dots[519:480]);
alpha_display af11(clock, 1'b1, char11, dots[479:440]);
alpha_display af10(clock, 1'b1, char10, dots[439:400]);
alpha_display af09(clock, 1'b1, char09, dots[399:360]);
alpha_display af08(clock, 1'b1, char08, dots[359:320]);
alpha_display af07(clock, 1'b1, char07, dots[319:280]);
alpha_display af06(clock, 1'b1, char06, dots[279:240]);
alpha_display af05(clock, 1'b1, char05, dots[239:200]);
alpha_display af04(clock, 1'b1, char04, dots[199:160]);
alpha_display af03(clock, 1'b1, char03, dots[159:120]);
alpha_display af02(clock, 1'b1, char02, dots[119:080]);
alpha_display af01(clock, 1'b1, char01, dots[079:040]);
alpha_display af00(clock,
                   ~(state == 1 || state == 3 || state == 4),
                   char00, dots[039:000]);
```

```
endmodule
```

### E.2 Display Threshold

```
`timescale 1ns / 1ps

module display_threshold( threshold,
                          clock,
                          dots );

   input [63:0] threshold;
   input clock;
   output [639:0] dots;

   alpha_display af15(clock, 1'b0, threshold[63:60], dots[639:600]);
   alpha_display af14(clock, 1'b0, threshold[59:56], dots[599:560]);
   alpha_display af13(clock, 1'b0, threshold[55:52], dots[559:520]);
   alpha_display af12(clock, 1'b0, threshold[51:48], dots[519:480]);
   alpha_display af11(clock, 1'b0, threshold[47:44], dots[479:440]);
   alpha_display af10(clock, 1'b0, threshold[43:40], dots[439:400]);
   alpha_display af09(clock, 1'b0, threshold[39:36], dots[399:360]);
   alpha_display af08(clock, 1'b0, threshold[35:32], dots[359:320]);
   alpha_display af07(clock, 1'b0, threshold[31:28], dots[319:280]);
   alpha_display af06(clock, 1'b0, threshold[27:24], dots[279:240]);
   alpha_display af05(clock, 1'b0, threshold[23:20], dots[239:200]);
   alpha_display af04(clock, 1'b0, threshold[19:16], dots[199:160]);
   alpha_display af03(clock, 1'b0, threshold[15:12], dots[159:120]);
   alpha_display af02(clock, 1'b0, threshold[11: 8], dots[119:080]);
   alpha_display af01(clock, 1'b0, threshold[ 7: 4], dots[079:040]);
   alpha_display af00(clock, 1'b0, threshold[ 3: 0], dots[039:000]);


endmodule
```

## F Memory Buffer Modules

### F.1 Image Buffer

```
`timescale 1ns / 1ps

/**************************************************
 * Image Buffer (128 x 128 pixels, 8 bit / pxl)
 * David Friend, friend@mit.edu
 *
 * Buffer stores 128 by 128 grayscale image in a
 * register array. Access is random.
 *
 * DELAYS
 *
 *  - Read delay:  2 (non-critical)
 *  - Write delay: 1 (non-critical)
 *      --    --    --    --
 *    _|  |__|  |__|  |__|
 *     0     1     2     3
 *
 *  A read at 0 will become valid at 2.
 *  Inputs at 1 and 2 will not affect the data
 *  becoming valid at 2.
 *
 *  A write at 0 will actually occur at 1. Once
 *  the data has latched at 0, it will be written,
 *  regardless of changes in input.
 *
 *  If you reset the pxl position, this
 *  takes a clock cycle!
 *
 ************************************************/

module img_buffer( /* Standard */
                   en,
                   we,
                   write_pxl,
                   read_pxl,
                   addr,
                   /* Sequencial Access */
                   re,
                   reset_pxl_pos,
                   enable_seq_access,
                   /* System */
                   reset,
                   clock );

   parameter PXL_WIDTH  = 8-1;
   parameter NUM_PIXELS = 16384-1;
   parameter ADDR_WIDTH = 14-1;

   //////////////////////////////////
   //  Inputs
   //////////////////////////////////
```

```
// Standard Interface
input en;
input we;
input [ADDR_WIDTH:0] addr;

// Sequencial Access Interface
input re;
input reset_pxl_pos;
input enable_seq_access;

// Data Lines
input [PXL_WIDTH:0] write_pxl;
output [PXL_WIDTH:0] read_pxl;

// System
input reset;
input clock;

//////////////////////////////////
// Registers
//////////////////////////////////
// internal storage
reg  [ADDR_WIDTH:0] pxl_pos;
reg  [14:0] buf_addr;
reg  [8:0] buf_din;
wire [8:0] buf_dout;
reg  buf_we;

//////////////////////////////////
// Buffer
//////////////////////////////////
buffer_128x128x8 img_buffer(
.addr( buf_addr[13:0] ),
.clk( clock ),
.din( buf_din[7:0] ),
.dout( buf_dout[7:0] ),
.en( en ),
.we( buf_we )
);

//////////////////////////////////
// Code
//////////////////////////////////

assign read_pxl = buf_dout[PXL_WIDTH:0];
```

```
   always @ (posedge clock)
      if (en) begin
         buf_addr[ADDR_WIDTH:0] <= (enable_seq_access ? pxl_pos : addr);
         buf_addr[14:ADDR_WIDTH+1] <= 14'b0;
         buf_din[PXL_WIDTH:0] <= write_pxl;
         buf_din[8:PXL_WIDTH+1] <= 8'b0;
         buf_we <= we;

         if (reset | reset_pxl_pos) pxl_pos <= 0;
         else if (enable_seq_access & (we | re))
            pxl_pos <= pxl_pos + 1;
         else pxl_pos <= pxl_pos;
      end

endmodule
```

## F.2 Double Buffer

```
`timescale 1ns / 1ps

/*********************************************
 *  Double Buffer Module
 *  David Friend, friend@mit.edu
 *
 *  For the proccessing of digital images,
 *  it is very convenient to use a double
 *  buffered design. We can read from the
 *  first buffer and write changes to the
 *  second. We can then switch the read and
 *  write buffers and perform the next
 *  filtering.
 *
 *  Sequencial access, employing two
 *  pointers, one for the read buffer
 *  and one for the write buffer.
 *
 *  DELAYS
 *
 *  - Read delay:  3 (critical)
 *  - Write delay: 2 (non-critical)
 *      __    __    __    __
 *    _|  |__|  |__|  |__|
 *     0     1     2     3
 *
 *  If a read_addr or re is valid at 0,
 *  read_pxl will be valid at 3. Swap
 *  should not be asserted between 0 and
 *  3, or the data will never become valid.
 *
 *  If a we is asserted at 0, the data
 *  is actually written at 2. Data are
 *  written to the buffer that is the
 *  write buffer at 0, regardless of input
 *  changes between 0 and 2. For instance, if
 *  swap is high at 1, the data will still
 *  be written, and then be available
 *  through a read (after the swap).
 *
 ********************************************/

module double_buffer( re,
                      we,
                      read_addr,
                      write_addr,
                      write_pxl,
                      read_pxl,
                      en_seq_read,
                      en_seq_write,
                      reset_write_pos,
                      reset_read_pos,
                      swap,
                      en,
                      reset,
                      clock );

   parameter PXL_WIDTH  = 8-1;
   parameter NUM_PIXELS = 16384-1;
   parameter ADDR_WIDTH = 14-1;

   /////////////////////////////
   // Pins
   /////////////////////////////

   // Standard Interface
   input en;
   input we;
   input [ADDR_WIDTH:0] read_addr;
   input [ADDR_WIDTH:0] write_addr;
   input swap;
```

```
// Sequencial Access
input re;
input en_seq_read;
input en_seq_write;
input reset_write_pos;
input reset_read_pos;

// Data Lines
input [PXL_WIDTH:0] write_pxl;
output [PXL_WIDTH:0] read_pxl;

// System
input reset;
input clock;

//////////////////////////////
// Registers
//////////////////////////////
// registered outputs
//reg [PXL_WIDTH:0] read_pxl;
// state
reg read_buffer;  // indicates which
                  // buffer (0 or 1)
                  // is the read buffer

reg buf0_we;
reg [PXL_WIDTH:0] buf0_write_pxl;
wire [PXL_WIDTH:0] buf0_read_pxl;
reg [ADDR_WIDTH:0] buf0_addr;
reg buf0_re, buf0_reset_pxl_pos,
    buf0_en_seq;

reg buf1_we;
reg [PXL_WIDTH:0] buf1_write_pxl;
wire [PXL_WIDTH:0] buf1_read_pxl;
reg [ADDR_WIDTH:0] buf1_addr;
reg buf1_re, buf1_reset_pxl_pos,
    buf1_en_seq;

//////////////////////////////
// Buffers
//////////////////////////////

img_buffer buf0(
 .en(en),
.we(buf0_we),
.write_pxl(buf0_write_pxl),
.read_pxl(buf0_read_pxl),
.addr(buf0_addr),
.re(buf0_re),
.reset_pxl_pos(buf0_reset_pxl_pos),
.enable_seq_access(buf0_en_seq),
.reset(reset),
.clock(clock)
   );

   defparam buf0.PXL_WIDTH = PXL_WIDTH;
   defparam buf0.NUM_PIXELS = NUM_PIXELS;
   defparam buf0.ADDR_WIDTH = ADDR_WIDTH;

   img_buffer buf1(
    .en(en),
.we(buf1_we),
.write_pxl(buf1_write_pxl),
.read_pxl(buf1_read_pxl),
.addr(buf1_addr),
.re(buf1_re),
.reset_pxl_pos(buf1_reset_pxl_pos),
.enable_seq_access(buf1_en_seq),
.reset(reset),
.clock(clock)
   );

   defparam buf1.PXL_WIDTH = PXL_WIDTH;
   defparam buf1.NUM_PIXELS = NUM_PIXELS;
   defparam buf1.ADDR_WIDTH = ADDR_WIDTH;

   ////////////////////////////
   // Code
   ////////////////////////////

   // read_pxl is only signal that isn't
   // buffered against a change in read_buffer
   // Don't swap the buffers while you're
   // waiting for read_pxl to become valid!
   assign read_pxl = ( read_buffer) ? buf1_read_pxl : buf0_read_pxl;

   always @ (posedge clock) begin

      if (reset) read_buffer <= 0;
      else if (swap) read_buffer <= ~read_buffer;

      buf1_we            <= ( read_buffer) ? 0              : we;
      buf1_write_pxl     <= ( read_buffer) ? 0              : write_pxl;
      buf1_re            <= ( read_buffer) ? re             : 0;
      buf1_addr          <= ( read_buffer) ? read_addr      : write_addr;
      buf1_en_seq        <= ( read_buffer) ? en_seq_read    : en_seq_write;
      buf1_reset_pxl_pos <= ( read_buffer) ? reset_read_pos : reset_write_pos;

      buf0_we            <= (!read_buffer) ? 0              : we;
      buf0_write_pxl     <= (!read_buffer) ? 0              : write_pxl;
      buf0_re            <= (!read_buffer) ? re             : 0;
      buf0_addr          <= (!read_buffer) ? read_addr      : write_addr;
      buf0_en_seq        <= (!read_buffer) ? en_seq_read    : en_seq_write;
      buf0_reset_pxl_pos <= (!read_buffer) ? reset_read_pos : reset_write_pos;

   end

endmodule
```

## F.3 Line Buffer

```
`timescale 1ns / 1ps

/**************************************************
 * Line Buffer (1x128 Pixels, 1 bit / pxl)
 * David Friend, friend@mit.edu
 *
 * Buffer stores 1 by 128 binary image in a
 * register array. Access is random.
 *
 * DELAYS
 *
 *  - Read delay:  2 (non-critical)
 *  - Write delay: 1 (non-critical)
 *      __    __    __    __
 *    _|  |__|  |__|  |__|
 *     0     1     2     3
 *
 *  A read at 0 will become valid at 2.
 *  Inputs at 1 and 2 will not affect the data
 *  becoming valid at 2.
 *
 *  A write at 0 will actually occur at 1. Once
 *  the data has latched at 0, it will be written,
 *  regardless of changes in input.
 *
 *  If you reset the pxl position, this
 *  takes a clock cycle!
 *
 ************************************************/

module line_buffer( /* Standard */
                    en,
                    we,
                    write_pxl,
                    read_pxl,
                    addr,
                    /* Sequencial Access */
                    re,
                    reset_pxl_pos,
                    enable_seq_access,
                    /* System */
                    reset,
                    clock );

   parameter PXL_WIDTH  = 1-1;
   parameter NUM_PIXELS = 128-1;
   parameter ADDR_WIDTH = 7-1;

   //////////////////////////////////
   //  Inputs
   //////////////////////////////////
   // Standard Interface
   input en;
   input we;
   input [ADDR_WIDTH:0] addr;

   // Sequencial Access Interface
   input re;
   input reset_pxl_pos;
   input enable_seq_access;

   // Data Lines
   input [PXL_WIDTH:0] write_pxl;
   output [PXL_WIDTH:0] read_pxl;

   // System
   input reset;
   input clock;
   //////////////////////////////////
   // Registers
   //////////////////////////////////
   // internal storage
   reg  [ADDR_WIDTH:0] pxl_pos;
   reg  [ADDR_WIDTH:0] buf_addr;
   reg  [PXL_WIDTH:0] buf_din;
   wire [PXL_WIDTH:0] buf_dout;
   reg  buf_we;

   //////////////////////////////////
   // Buffer
   //////////////////////////////////
   buffer_1x128x1 line_buf(
   .addr( buf_addr ),
   .clk( clock ),
   .din( buf_din ),
   .dout( buf_dout ),
   .en( en ),
   .we( buf_we )
   );

   //////////////////////////////////
   // Code
   //////////////////////////////////

   assign read_pxl = buf_dout;

   always @ (posedge clock)
      if (en) begin
         buf_addr <= (enable_seq_access ? pxl_pos : addr);
         buf_din <= write_pxl;
         buf_we <= we;
         if (reset | reset_pxl_pos) pxl_pos <= 0;
         else if (enable_seq_access & (we | re))
            pxl_pos <= pxl_pos + 1;
         else pxl_pos <= pxl_pos;
      end

endmodule
```

# G Processing

## G.1 Processing Controller

```
`timescale 1ns / 1ps

module process_image( /* Interface */
                      start,
                      busy,
                      /* Image Input */
                      scanned_img_reset_pxl_pos,
                      scanned_img_request_pxl,
                      scanned_img_pxl_in,
                      scanned_img_en_seq_access,
                      /* Display Output */
                      cycle_display,
                      disp_addr, disp_pxl_out,
                      disp_we,
                      /* Fourier Coefficients */
                      we_row_coeff, fft_row_dv, row_coeff,
                      we_col_coeff, fft_col_dv, col_coeff,
                      /* System */
                      reset,
                      clock,
                      sel_data_begin, sel_data_in, sel_row_data_out, sel_col_data_out,
                      sel_row_we, sel_col_we );

   // Debug:
   output sel_data_begin;
   output sel_data_in;
   output sel_row_data_out;
   output sel_col_data_out;
   output sel_col_we, sel_row_we;


   /* Process Image Interface */
   input start;
   output busy;

   /* Original Image Memory Interface */
   output scanned_img_reset_pxl_pos;
   output scanned_img_request_pxl;
   input [7:0] scanned_img_pxl_in;
   output scanned_img_en_seq_access;

   /* Output */
```

```
output cycle_display;
output [13:0] disp_addr;
output [7:0] disp_pxl_out;
output disp_we;
output we_row_coeff, fft_row_dv,
       we_col_coeff, fft_col_dv;
output [31:0] row_coeff, col_coeff;

/* System */
input reset;
input clock;

///////////////////
// Registers
///////////////////
// registered outputs
reg busy;
reg scanned_img_en_seq_access = 0;
// FSM
reg [2:0] state, next, waitnext;
// other
reg buf_swap;
reg ds_start, disp_start;
reg sel_start, fft_row_start, fft_col_start;

///////////////////
// Parameters
///////////////////
// FSM state identifiers
parameter IDLE = 0;
parameter SYS_RESET = 1;
parameter WAIT = 2;
parameter DOWNSAMPLE = 3;
parameter ROW_COL_SEL = 4;
parameter FOURIER = 5;
parameter UPDATE_DISP = 6;


///////////////////
// Modules
// (Processing Units)
//////////////////////////////////////////
// Double Buffer
//////////////////////////////////////////
wire buf_re, buf_we,
     buf_write_pxl, buf_read_pxl,
     buf_rst_write, buf_rst_read;

double_buffer buffers(
   .re( buf_re ),
   .we( buf_we ),
   .read_addr( 14'b0 ),
   .write_addr( 14'b0 ),
   .read_pxl( buf_read_pxl ),
   .write_pxl( buf_write_pxl ),
   .en_seq_read( 1'b1 ),
   .en_seq_write( 1'b1 ),
   .reset_write_pos( buf_rst_write ),
   .reset_read_pos( buf_rst_read ),
   .swap( buf_swap ),
   .en( 1'b1 ),
   .reset( reset ),
   .clock( clock )
);

defparam buffers.PXL_WIDTH = 0;
defparam buffers.NUM_PIXELS = 16383;
defparam buffers.ADDR_WIDTH = 13;


///////////////////////////////////////////
// Downsample Module
///////////////////////////////////////////

wire ds_busy;
wire ds_we, ds_write_pxl;
wire ds_rst_write;

downsample ds(
   .in_reset_pxl_pos( scanned_img_reset_pxl_pos ),
   .in_re( scanned_img_request_pxl ),
   .in_pxl( scanned_img_pxl_in ),
   .out_reset_pxl_pos( ds_rst_write ) ,
   .out_we( ds_we ),
   .out_pxl( ds_write_pxl ),
   .start( ds_start ),
   .busy( ds_busy ),
   .clock( clock ),
   .reset( reset )
);
```

```
///////////////////////////////////////////
// Update Display Module
///////////////////////////////////////////
wire disp_busy;
wire disp_re, disp_rst_read;

update_display disp(
   .start( disp_start ),
   .busy( disp_busy ),
   .re( disp_re  ),
   .we( disp_we ),
   .addr( disp_addr ),
   .reset_input( disp_rst_read ),
   .cycle_display( cycle_display ),
   .read_pxl( buf_read_pxl ),
   .write_pxl( disp_pxl_out ),
   .clock(clock),
   .reset(reset)
);

//////////////////////////////////////////
// Select Row Column
//////////////////////////////////////////

wire sel_busy;
wire sel_re, sel_rst_read;
wire row_we, row_write_pxl, row_read_pxl;
wire row_re, row_rst, row_en_seq;
wire col_we, col_write_pxl, col_read_pxl;
wire col_re, col_rst, col_en_seq;
wire sel_row_rst, sel_col_rst;

row_col_selector selector(.clock(clock), .reset(reset),
   .row_select(7'd64), .col_select(7'd64),  // input row and column numbers
   .start( sel_start ), .busy( sel_busy ),
   // INPUT from img buffer
   .data_in( buf_read_pxl ), .re( sel_re ), .reset_pxl_pos_data_in( sel_rst_read ),
   // OUTPUT to line buffers
   .reset_pxl_pos_row( sel_row_rst ), .we_row( row_we ), .data_out_row( row_write_pxl ),
   .reset_pxl_pos_col( sel_col_rst ), .we_col( col_we ), .data_out_col( col_write_pxl ) );

line_buffer row_buffer( .en( 1'b1 ), .reset(reset), .clock(clock),
   .we( row_we ), .write_pxl( row_write_pxl ), .addr( 7'b0 ),
   .read_pxl( row_read_pxl ), .re( row_re ), .reset_pxl_pos( row_rst ),
   .enable_seq_access( 1'b1 ));

line_buffer col_buffer( .en( 1'b1 ), .reset(reset), .clock(clock),
   .we( col_we ), .write_pxl( col_write_pxl ), .addr( 7'b0),
   .read_pxl( col_read_pxl ), .re( col_re ), .reset_pxl_pos( col_rst ),
   .enable_seq_access( 1'b1 ));

// debugging wires
assign sel_data_begin = sel_start;
assign sel_data_in = buf_read_pxl;
assign sel_row_data_out = row_read_pxl;
assign sel_col_data_out = col_read_pxl;
assign sel_row_we = row_re;
assign sel_col_we = col_re;

//////////////////////////////////////////
// FFT
//////////////////////////////////////////
wire fft_row_busy;
wire fft_col_busy;
// outputs
// we_row_coeff, rst_row_coeff, row_coeff,
// we_col_coeff, rst_col_coeff, col_coeff,

wire fft_row_rst, fft_row_re, fft_col_rst, fft_col_re;

fft fft_row(.clock(clock), .reset(reset),
   .start( fft_row_start ), .busy( fft_row_busy ),
   .reset_pxl_pos_in( fft_row_rst ), .re_in( fft_row_re ), .data_in( row_read_pxl ),
   .we_out( we_row_coeff ),
   .M_sum_sq( row_coeff ), .fft_dv( fft_row_dv) );
fft fft_col(.clock(clock), .reset(reset),
   .start( fft_col_start ), .busy( fft_col_busy ),
   .reset_pxl_pos_in( fft_col_rst ), .re_in( fft_col_re ), .data_in( col_read_pxl ),
   .we_out( we_col_coeff ),
   .M_sum_sq( col_coeff ), .fft_dv( fft_col_dv) );

assign row_rst = fft_row_rst | sel_row_rst;
assign row_re = fft_row_re ;
assign col_rst = fft_col_rst | sel_col_rst ;
assign col_re = fft_col_re ;


assign buf_re = disp_re | sel_re ;
assign buf_we = ds_we ;
assign buf_rst_write = ds_rst_write;
assign buf_rst_read = disp_rst_read | sel_rst_read;
assign buf_write_pxl = ds_write_pxl;

always @ (posedge clock) begin
   if (reset) state <= SYS_RESET;
   else state <= next;

   busy <= (next != IDLE);
   scanned_img_en_seq_access <= (next != IDLE);

   ds_start <= (next == DOWNSAMPLE);
   disp_start <= (next == UPDATE_DISP);
   sel_start <= (next == ROW_COL_SEL);
   fft_row_start <= (next == FOURIER);
   fft_col_start <= (next == FOURIER);

   // this is a bit cryptic, but it means
   // swap the buffers on the exit of Downsample
   buf_swap <= ( state == WAIT &&
                 next == waitnext &&
                 next == UPDATE_DISP );

   case (state)
      WAIT: waitnext <= waitnext;
      DOWNSAMPLE: waitnext <= UPDATE_DISP;
      UPDATE_DISP: waitnext <= ROW_COL_SEL;
      ROW_COL_SEL: waitnext <= FOURIER;
      FOURIER: waitnext <= IDLE;
      default: waitnext <= waitnext;
   endcase
end

wire internal_busy = ds_busy |
                     disp_busy |
                     sel_busy |
                     fft_row_busy |
                     fft_col_busy;

always @ (state or start or internal_busy )
   case (state)
      IDLE:
         if (start) next = DOWNSAMPLE;
         else if (next == DOWNSAMPLE);
         else next = IDLE;
```

```
        SYS_RESET:
           next = IDLE;
        WAIT:
           if ( internal_busy ) next = WAIT;
           else next = waitnext;
        DOWNSAMPLE:
           next = WAIT;
        UPDATE_DISP:
           next = WAIT;
        ROW_COL_SEL:
           next = WAIT;
        FOURIER:
           next = WAIT;
        default:
           next = IDLE;
     endcase

endmodule
```

## G.2 Downsample

```
`timescale 1ns / 1ps

/***************************************************
 *  Downsample Module
 *  David Friend, friend@mit.edu
 *
 *  Input: 128 x 128 image, 8-bits per pxl
 *  Output: 128 x 128 image, 1-bit per pxl
 *
 *  Computes the average pixel value of the 8-bit
 *  image and uses this as a threshold. If a pixel
 *  has a pixel value greater than the threshold,
 *  it is a '1'. Below or equal to the threshold
 *  is interpreted as a '0'.
 *
 *  There are three steps:
 *    1. Sum all 8-bit pixels
 *    2. Compute threshold (average of sum
 *       from step 1)
 *    3. Output downsampled image
 *  These steps correspond to the states of
 *
 * SOURCE: img_buffer, sequencial
 *
 *  Reading from img_buffer. Delay is 2
 *  clock cycles. Must have two initialization
 *  reads to account for delay. Also must
 *  reset buffer position before reading. This
 *  is a total of three initialization states.
 *
 * DESTINATION: double_buffer, sequencial
 *
 *  Writing to a double_buffer. Delay is two
 *  clock cycles, but it is not critical in the
 *  sense that nothing can preven the write
 *  from occuring once the data has been
 *  latched. Access is sequencial. The position
 *  must be reset before writing. Care must be
 *  taken to start the write at the appropriate
 *  time relative to the read.
 *
 ***************************************************/

module downsample( in_reset_pxl_pos,
                   in_re,
                   in_pxl,
                   out_reset_pxl_pos,
                   out_we,
                   out_pxl,
                   start,
                   busy,
                   clock,
                   reset );

   parameter PXL_WIDTH = 8-1;
   parameter ADDR_WIDTH = 14-1;
   parameter NUM_PIXELS = 16383-1;

   // Input Memory Interface
   output in_reset_pxl_pos;
   output in_re;
   input [PXL_WIDTH:0] in_pxl;

   // Output Memory Interface
   output out_reset_pxl_pos;
   output out_we;
output out_pxl;

   // Module Interface
   input start;
```

```
output busy;

// System
input clock;
input reset;

/////////////////////
// Parameters
/////////////////////
// states
parameter IDLE          = 0;
parameter SYS_RESET     = 1;

// read to accumulate
parameter RST_PXL       = 2;
parameter INITREAD1     = 3;
parameter INITREAD2     = 4;
parameter READSUM       = 5;

// calculate
parameter THRESHOLD     = 6;

// read to write
parameter RST_PXL_2     = 7;
parameter INITREAD1_2   = 8;
parameter INITREAD2_2   = 9;
parameter INITREAD3_2   = 10;  // to account for delay
parameter READWRITE     = 11;  // or latch_read_compare



/////////////////////
// Registers
/////////////////////
// registered outputs
reg in_reset_pxl_pos;
reg in_re;
reg out_reset_pxl_pos;
reg out_we;
reg out_pxl;
reg busy;
// internal storage
reg [21:0] sum;
reg [PXL_WIDTH:0] threshold;
reg [3:0] state, next;
reg [ADDR_WIDTH:0] pxl_cnt;

//////////////////////
// Code
//////////////////////

always @ (posedge clock) begin

   if (reset) state <= SYS_RESET;
   else state <= next;

   busy <= (next != IDLE);

   sum <= ((next == IDLE)    ? (22'b0) :
          ((next == READSUM) ? (sum + in_pxl) : (sum)));
   threshold <= ((next == IDLE) ? (127) :
                ((next == THRESHOLD) ? (sum >> (ADDR_WIDTH + 1)) : (threshold)));


   // increment pxl_cnt on exit from readwrite states
   if (next == IDLE  ||
       next == RST_PXL ||
       next == RST_PXL_2 ) pxl_cnt <= 0;
   else if ((next == READSUM && state == READSUM) ||
            (next == READWRITE && state == READWRITE))
      pxl_cnt <= pxl_cnt + 1;
   else
      pxl_cnt <= pxl_cnt;

   in_reset_pxl_pos <= (next == RST_PXL || next == RST_PXL_2);
   in_re <= (next == INITREAD1 ||
             next == INITREAD2 ||
             next == READSUM ||
             next == INITREAD1_2 ||
             next == INITREAD2_2 ||
             next == INITREAD3_2 ||
             next == READWRITE );

   out_reset_pxl_pos <= (next == RST_PXL_2);
   out_we <= (next == READWRITE);
   // delays out_pxl by one
   // from in_pxl becoming valid ---
   // hence three initial reads
   out_pxl <= (next == IDLE) ? 0 : (in_pxl > threshold);
   end

   always @ (state or start or pxl_cnt)
      case (state)
         IDLE:
            if (start) next = RST_PXL;
            else if (next == RST_PXL) next = RST_PXL;
            else next = IDLE;
         SYS_RESET:
            next = IDLE;
         RST_PXL:
            next = INITREAD1;
         INITREAD1:
            next = INITREAD2;
         INITREAD2:
            next = READSUM;
         READSUM:
            if (pxl_cnt == NUM_PIXELS) next = THRESHOLD;
            else next = READSUM;
         THRESHOLD:
            next = RST_PXL_2;
         RST_PXL_2:
            next = INITREAD1_2;
         INITREAD1_2:
            next = INITREAD2_2;
         INITREAD2_2:
            next = INITREAD3_2;
         INITREAD3_2:
            next = READWRITE;
         READWRITE:
            if (pxl_cnt == NUM_PIXELS) next = IDLE;
            else next = READWRITE;
         default:
            next = IDLE;
      endcase


endmodule
```

## G.3 Display Processed Unit

```
‘timescale 1ns / 1ps

/*********************************************
 * Update Display from Processed Image
 * David Friend, friend@mit.edu
 *
 * Reads data from the processed image buffer
 * and writes into the Display. Finally it
 * cycles the display buffers to put the img
 * on the screen.
 *
 * SOURCE: double_buffer, sequencial
 *
 * Reading from a duble_buffer. Delay is
 * 3 clock cycles. Must have three initialization
 * reads to account for delay. But must
 * reset buffer position before reading.
 * This a total of four initialization
 * states.
 *
 * DESTINATION: double_buffer, random
 *
 * Writing to a double_buffer. Delay is
 * 2 clock cycles, but is not critical,
 * in the sense that nothing can prevent
 * the write from occuring once it has
 * latched. Access is random; care must
 * be taken to ensure that data is written
 * to the proper location. Address should
 * be incremented on exit from the WRITE
 * state.
 *
 ************************************************/

module update_display( start, busy,
                       re, we, addr,
                       reset_input, cycle_display,
                       read_pxl, write_pxl,
                       clock, reset);

   parameter PXL_WIDTH = 8-1;
   parameter ADDR_WIDTH = 14-1;
   parameter NUM_PIXELS = 16384-1;

   // Module Interface
   input start;
   output busy;

   // Source Memory Interface
```

```
output re, reset_input;
input read_pxl;

// Display Memory Interface
output [ADDR_WIDTH:0] addr;
output we, cycle_display;
output [PXL_WIDTH:0] write_pxl;

// System
input clock;
input reset;

//////////////////////////////
// Parameters
//////////////////////////////
// state
parameter IDLE = 0;
parameter SYS_RESET = 1;
parameter RESET = 2;
parameter INITREAD1 = 3;
parameter INITREAD2 = 4;
parameter INITREAD3 = 5;
parameter READWRITE = 6;

//////////////////////////////
// Registers
//////////////////////////////
// registered outputs
reg re, reset_input;
reg we, cycle_display;
reg busy;
reg [ADDR_WIDTH:0] addr;
// internal
reg [2:0] state, next;

//////////////////////////////
// Code
//////////////////////////////

assign write_pxl = {read_pxl, read_pxl, read_pxl, read_pxl,
                    read_pxl, read_pxl, read_pxl, read_pxl};

always @ (posedge clock) begin
   if (reset) state <= SYS_RESET;
   else state <= next;
   busy <= (next != IDLE);
   reset_input <= (next == RESET);
   re <= (next == INITREAD1 |
          next == INITREAD2 |
          next == INITREAD3 |
          next == READWRITE);
   we <= (next == READWRITE);
   cycle_display <= (next == IDLE && state == READWRITE);

   // addr
   if (reset | next == RESET)
      addr <= 0;
   else if ( next == READWRITE &&
             state == READWRITE )
      addr <= addr + 1;
   else
      addr <= addr;
end


always @ (state or start or addr)
   case (state)
      IDLE:
         if (start) next = RESET;
         // if we don't include this condition:
         else if (next == RESET) next = RESET;
         // then this might be true on
         // negedge of start, and we
         // don't start at all
         else next = IDLE;
      SYS_RESET:
         next = IDLE;
      RESET:
         next = INITREAD1;
      INITREAD1:
         next = INITREAD2;
      INITREAD2:
         next = INITREAD3;
      INITREAD3:
         next = READWRITE;
      READWRITE:
         if (addr == NUM_PIXELS) next = IDLE;
         else next = READWRITE;
      default:
            next = IDLE;
      endcase

endmodule
```

## G.4 Row and Column Selector

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//    Kevin Amendt   kamendt@mit.edu
//    6.111 Final Project: Fingerprint Identification
//
//    Create Date:    10:03:14 05/15/06
//    Design Name:
//    Module Name:    my_row_your_cloumn
//    Project Name:
//    Target Device:
//    Tool versions:
//    Description:      This module will write a specific row and column of a 128x128
//                      Binary image.
//
//    Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module row_col_selector(clock, reset, row_select, col_select,
                          start, data_in, busy, re,
                          reset_pxl_pos_data_in, reset_pxl_pos_row,
                          we_row, data_out_row, reset_pxl_pos_col,
                          we_col, data_out_col
                          );

   //   Inputs
   input          clock, reset;
   input [6:0]    row_select, col_select;
   input          start;
   input          data_in;

   //    Outputs
   output         busy;
   output         re, reset_pxl_pos_data_in;
   output         reset_pxl_pos_row, we_row, data_out_row;
   output         reset_pxl_pos_col, we_col, data_out_col;
```

```
//    Registers
reg   [2:0]     state, next;
reg             busy;
reg             re, reset_pxl_pos_data_in;
reg             reset_pxl_pos_row, we_row;
reg             reset_pxl_pos_col, we_col;
reg   [14:0]    pxl_count;

//    Wires
wire  [6:0]     row_val = pxl_count[13:7];
wire  [6:0]     col_val = pxl_count[6:0];

//    FSM State Parameters
parameter       SYS_RESET = 0;
parameter       IDLE = 1;
parameter       RESET_PXL = 2;
parameter       WRITE_ROW_COL = 3;

//    Sequential Always Block
always @ (posedge clock)
   if(reset)   begin
      state <= SYS_RESET;
      pxl_count <=0;
   end
   else begin
      state <= next;
      reset_pxl_pos_data_in <= (next == RESET_PXL);
      reset_pxl_pos_row <= (next == RESET_PXL);
      reset_pxl_pos_col <= (next == RESET_PXL);
      we_row <= ((next == WRITE_ROW_COL) & (row_val == row_select));
      we_col <= ((next == WRITE_ROW_COL) & (col_val == col_select + 1));
      re <= (next == WRITE_ROW_COL);
      busy <= ~(next == IDLE);
      if((next == WRITE_ROW_COL) && (state == WRITE_ROW_COL))
         pxl_count <= pxl_count + 1;
      else if(next == RESET_PXL)
         pxl_count <= 0;
   end

//    Combinational Always Block
always @ (state or start or pxl_count)
   case(state)
      SYS_RESET:
         next = IDLE;
         IDLE:
            if(start)   next = RESET_PXL;
            else        next = IDLE;
         RESET_PXL:
            next = WRITE_ROW_COL;
         WRITE_ROW_COL:
            if(pxl_count == 16383) next = IDLE;
            else     next = WRITE_ROW_COL;
      endcase

   //    Combinational Output Assignments
   assign data_out_row = data_in;
   assign data_out_col = data_in;

endmodule
```

## G.5 Fourier Transform

### G.5.1 FFT

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//    Kevin Amendt   kamendt@mit.edu
//    6.111 Final Project: Fingerprint Identification
//
//
// Create Date:    15:57:20 05/14/06
//
// Module Name:    fft
//
//                This module is to interface with the FFT coregen instance.
//                It will read the values of the input out of memory, and
//                supply them to the FFT at the correct timing.  Streams them in
//                every clock cycle.
//
//                It will also square the real/img parts of the coefficients and
//                write them to memory
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module fft(clock, reset, data_in, start, sum_sq,
            reset_pxl_pos_in, re_in, we_out,
            enable_seq_access_out, reset_pxl_pos_out,
            busy,
            fft_xk_re, fft_xk_im
            ,fft_xn_index, fft_xk_index
            ,M_sum_sq
            ,fft_dv );

input          clock, reset;
input     data_in;
input          start;

output   [16:0] sum_sq;
output   [31:0] M_sum_sq;

output         reset_pxl_pos_in, re_in;
output         we_out, enable_seq_access_out, reset_pxl_pos_out;
output         busy;
output   [15:0]fft_xk_re, fft_xk_im;
output   [6:0] fft_xn_index, fft_xk_index;
output   fft_dv;

//    Instance of FFT
reg   [7:0] fft_xn_re;                    // X (time) in real             INPUT
wire  [7:0] fft_xn_im;                    // X (time) in imaginary        INPUT
wire        fft_start;                    // start FFT                    INPUT
wire        fft_fwd_inv;                  // Forward or Inverse FFT       INPUT
wire        fft_fwd_inv_we;               // Write Enable for fwd_inv     INPUT
wire        fft_sclr;                     // System Reset: sets all values to default INPUT
wire        fft_ce;                       // clock enable                 INPUT
wire        fft_clock;                    // clock, synchronous           INPUT
wire  [15:0]fft_xk_re;                    // X (freq) out real            OUTPUT
wire  [15:0]fft_xk_im;                    // X (freq) out imaginary       OUTPUT
wire  [6:0] fft_xn_index;                 // index number for X (time) in OUTPUT
wire  [6:0] fft_xk_index;                 // index number for X (freq) out OUTPUT
wire        fft_rfd;                      // Ready For Data                OUTPUT
wire        fft_busy;                     // FFT is busy                   OUTPUT
wire        fft_dv;                       // Data Valid                    OUTPUT
wire        fft_edone;                    // Early done, high one clock cycle before done goes
wire        fft_done;                     // FFT transform is done, streams out data   OUTPUT

fft_128x1 fft(
.xn_re(fft_xn_re),
.xn_im(fft_xn_im),
.start(fft_start),
.fwd_inv(fft_fwd_inv),
.fwd_inv_we(fft_fwd_inv_we),
.sclr(fft_sclr),
.ce(fft_ce),
.clk(fft_clock),
.xk_re(fft_xk_re),
.xk_im(fft_xk_im),
.xn_index(fft_xn_index),
.xk_index(fft_xk_index),
.rfd(fft_rfd),
.busy(fft_busy),
.dv(fft_dv),
.edone(fft_edone),
.done(fft_done)
   );

// FFT initializer
reg [1:0] initclock;
always @ (posedge clock)begin
initclock <= reset ? 0 : ((initclock < 3) ? initclock + 1 : initclock);
fft_xn_re[0] <= data_in;
fft_xn_re[7:1] <= 7'b0;
//fft_xn_re[7:0] <= data_in[7:0];
end

// FFT port assignments
assign fft_fwd_inv_we = (initclock == 2'b01);
assign fft_fwd_inv = 1;
assign fft_ce = 1;
assign fft_xn_im[7:0] = 8'b0;
assign fft_sclr = 0;
assign fft_clock = clock;





//    FFT_FSM to control the FFT instance

   fft_FSM  fft_FSM(
.clock(clock),
.reset(reset),
.start(start),
.reset_pxl_pos_in(reset_pxl_pos_in),
.re_in(re_in),
.we_out(we_out),
.enable_seq_access_out(enable_seq_access_out),
.reset_pxl_pos_out(reset_pxl_pos_out),
      .fft_dv(fft_dv),
      .fft_start(fft_start),
      .busy(busy)
);

//    Sum_Square to compute the square of the magnitude (sum of square of Img and Real parts)

   sum_square uut (
      .clock(clock),
      .reset(reset),
.re(fft_xk_re[15:0]),
.im(fft_xk_im[15:0]),
.sum_sq(sum_sq),
      .M_sum_sq(M_sum_sq)
);
endmodule
```

### G.5.2 FFT FSM

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//    Kevin Amendt   kamendt@mit.edu
//    6.111 Final Project: Fingerprint Identification
//
// Create Date:    19:23:55 05/14/06
// Design Name:
// Module Name:    fft_FSM
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module fft_FSM(clock, reset, start,
               reset_pxl_pos_in, re_in,
               we_out, enable_seq_access_out,
               reset_pxl_pos_out
               ,fft_dv,fft_start,
            busy

            );

//    Inputs
input          clock, reset, start;
input          fft_dv;

//    Outputs
output         reset_pxl_pos_in, re_in;
output         we_out, enable_seq_access_out, reset_pxl_pos_out;
output         fft_start;
output         busy;

//    Registers
reg   [2:0]    state, next;
reg   [7:0]    read_count;
reg   [7:0]    write_count;

reg            reset_pxl_pos_in, reset_pxl_pos_out, re_in, we_out, enable_seq_access_out, f
reg            busy;
//    State Parameters
parameter SYS_RESET = 0;
parameter IDLE = 1;
parameter STARTFFT = 2;
parameter READ_VALUES = 3;
parameter WAIT = 4;
parameter WRITE_COEF = 5;
parameter DISCARD_COEF = 6;

//    Sequential Always Block
always @ (posedge clock or posedge reset)
   if(reset)
      state <= SYS_RESET;
   else begin
      state <= next;
      reset_pxl_pos_in <= (next == STARTFFT);
      reset_pxl_pos_out <= (next == STARTFFT);
      re_in <= (next == READ_VALUES);
      we_out <= (next == WRITE_COEF);
      enable_seq_access_out <= (next == WRITE_COEF);
      fft_start <= ((next == STARTFFT)) ;
      busy <= ~(next == IDLE);
      if(next == IDLE) begin
         read_count <= 0;
```

```
            write_count <= 0;
         end
         else if(next == READ_VALUES)
            read_count <= read_count + 1;
         else if(next == WRITE_COEF || next == DISCARD_COEF)
            write_count <= write_count + 1;
      end


   //    Combinational Always Block
   always @ (state or start or fft_dv or read_count or write_count)
      case(state)
         SYS_RESET:
            next = IDLE;
         IDLE:
            if(start)   next = STARTFFT;
            else        next = IDLE;
         STARTFFT:
            next = READ_VALUES;
         READ_VALUES:
            if(read_count == 128)   next = WAIT;
            else                    next = READ_VALUES;
         WAIT:
            if(fft_dv)  next = WRITE_COEF;
            else        next = WAIT;
         WRITE_COEF:
            if(write_count == 16)   next = DISCARD_COEF;
            else                    next = WRITE_COEF;
         DISCARD_COEF:
            if(write_count == 200) next = IDLE;
            else  next = DISCARD_COEF;
      endcase

   //Combinational Output Assignment Block

endmodule
```

### G.5.3 Squared Magnitude of Coefficients

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//    Kevin Amendt   kamendt@mit.edu
//    6.111 Final Project: Fingerprint Identification
//
//    Create Date:    17:51:48 05/14/06
//    Module Name:    multiply_test
//
//    This module takes in the real and imaginary parts from the FFT unit
//    and sums the square of each part.
//
//    Registers are used to ensure that timing requirements are met in the fft module
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module sum_square(clock, reset, re, im, sum_sq, M_sum_sq);

   //    Inputs
   input             clock, reset;
   input    [15:0]    re, im;

   //    Outputs
   output   [16:0]    sum_sq;
   output   [31:0]    M_sum_sq;
   //    Registers
   reg [31:0] M_sum_sq;

   //    Wires -- Switch from 2's complement to pos unsigned int
   wire [7:0]     re_unsigned =  (re[7]) ? ({1'b0,~re[6:0]}+1) : ({1'b0, re[6:0]});
wire [7:0]     im_unsigned =  (im[7]) ? ({1'b0,~im[6:0]}+1) : ({1'b0, im[6:0]});

   //    Sequential Always Block
   always @ (posedge clock)
      if(reset)   M_sum_sq <= 0;
      else begin
         M_sum_sq <= re_unsigned*re_unsigned + im_unsigned*im_unsigned;
      end

   //    Assignments
   assign sum_sq[16:0] = M_sum_sq[16:0];        // discard upper 9 bits of M_sum_sq
                                              //  shouldn't ever break 8 bits

endmodule
```

## H Buffer Selector

```
`timescale 1ns / 1ps
```

```
/************************************************************
 * FFT Coefficient Reverse Multiplexer
 *
 * Determines destination database for fourier coefficients
 * based on the add_user input and the user_id input.
 *
 * Runs synchronously with FFT module. in_rst is the start
 * signal.
 *
 ************************************************************/

module fft_coeff_mux( // selector
                      add_user,
                      user_id,
                      // data in
                      in_we,
                      in_rst,
                      in_data,
                      // data out
                      bufA_data,
                      bufA_we,
                      bufA_addr,
                      bufB_data,
                      bufB_we,
                      bufB_addr,
                      // system
                      clock,
                      reset );


   ////////////////////////////////////////////////////
   // Data Pins
   ////////////////////////////////////////////////////
   // MUX Selector
   input add_user;
   input [2:0] user_id;

   // Data In
   input in_we;
   input in_rst;
   input [31:0] in_data;

   // Data Out
   output [31:0] bufA_data;
   output bufA_we;
   output [3:0] bufA_addr;
   output [31:0] bufB_data;
   output bufB_we;
   output [6:0] bufB_addr;

   // System
   input clock;
   input reset;


   ////////////////////////////////////////////////////
   // Parameters
   ////////////////////////////////////////////////////
   parameter IDLE = 0;
   parameter SYS_RESET = 1;
   parameter START = 2;
   parameter WRITE_COEFF = 3;
   parameter PAUSE = 4;


   ////////////////////////////////////////////////////
   // Registers
   ////////////////////////////////////////////////////
   reg [31:0] bufA_data;
   reg bufA_we;
   reg [3:0]  bufA_addr_delay, Mid_bufA_addr;
   reg [31:0] bufB_data;
   reg bufB_we;
   reg [6:0] bufB_addr_delay, Mid_bufB_addr;
   reg [2:0] state, next;
   reg [4:0] coeff_cnt;
   reg add_user_save;

   ////////////////////////////////////////////////////
   // Code
   ////////////////////////////////////////////////////

   //    Sequential Always block
   always @ (posedge clock or posedge reset)
      if (reset) begin
         state <= SYS_RESET;
         coeff_cnt <= 0;
      end
      else begin
```

```
      state <= next;
      bufA_data <= (coeff_cnt > 2) ? in_data : 0;
      bufB_data <= (coeff_cnt > 2 ) ? in_data : 0;
      Mid_bufA_addr <= bufA_addr_delay;
      Mid_bufB_addr <= bufB_addr_delay;
      if (next == START) begin
         add_user_save <= add_user;
         coeff_cnt <= 0;
         bufA_addr_delay <= 0;
         bufB_addr_delay <= user_id << 4;
      end
      if (next == WRITE_COEFF) begin
         if (add_user_save) begin
            bufB_we <= in_we;
            bufA_we <= 0;
         end
         else begin
            bufA_we <= in_we;
            bufB_we <= 0;
         end
         bufA_addr_delay <= bufA_addr_delay + 1;
         bufB_addr_delay <= bufB_addr_delay + 1;
      end
      else begin
         bufB_we <= 0;
         bufA_we <= 0;
      end
      if (state == WRITE_COEFF)
      coeff_cnt <= coeff_cnt + 1;
      else
      coeff_cnt <= 0;
end

//    Combinational Always Block
always @ (state or in_rst or coeff_cnt)
   case (state)
      IDLE: if (in_rst) next = START;
            else next = IDLE;
      SYS_RESET: next = IDLE;
      START: next =  /*PAUSE;
      PAUSE:   next = */WRITE_COEFF;
      WRITE_COEFF: if (coeff_cnt == 15 ) next = IDLE;
                   else next = WRITE_COEFF;
   endcase
   assign bufA_addr = (state == IDLE) ? (4'hz): (Mid_bufA_addr);
   assign bufB_addr = (state == IDLE) ? (7'hz): (Mid_bufB_addr);

endmodule
```

# I Validation Unit

## I.1 Validation

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//    6.111 Fingerprint ID
//    Kevin Amendt   kamendt@mit.edu
//
//  Top level instanstiation of memory and sq_errors and comparator
//  Processing block can write to the memory via the we & din inputs
//
////////////////////////////////////////////////////////////////////////////////

module validation(clock, reset, compare, match, din_row_A, din_row_B,
                     we_row_A, we_row_B, din_col_A, din_col_B, we_col_A,
                     we_col_B, row_addr_A, row_addr_B, col_addr_A, col_addr_B,
                     busy, img_num, threshold,
                     add_user,
                     user_id
                    );

   parameter ENABLED = 1;

   //    Inputs
   input          clock, reset, compare;
   input [31:0]    din_row_A, din_row_B, din_col_A, din_col_B;
   input          we_row_A, we_row_B, we_col_A, we_col_B;
   input [63:0]   threshold;
   input [2:0] user_id;
   input add_user;


   //    Outputs
   output         match, busy;
   output [3:0]   img_num;

   //    Inout
   inout   [3:0] row_addr_A, col_addr_A;
   inout   [6:0] row_addr_B, col_addr_B;

   //    Wires
   wire  [3:0]    row_addr_A, col_addr_A;
   wire  [6:0]    row_addr_B, col_addr_B;
   wire  [31:0]   row_in_A, row_in_B,col_in_A, col_in_B;
   wire           row_done,  row_start;
   wire  [15:0]   row_sq_error,col_sq_error;
   wire           col_done, col_start;
//  wire  [3:0]    img_num;

   //    Instances

   //    Instanstiation of memory elements

   buffer_1x16x32 row_mem_A(
    .addr(row_addr_A),
    .clk(clock),
    .din(din_row_A),
    .dout(row_in_A),
    .en(ENABLED),
    .we(we_row_A)
      );

   buffer_1x16x32 col_mem_A(
    .addr(col_addr_A),
    .clk(clock),
    .din(din_col_A),
    .dout(col_in_A),
    .en(ENABLED),
    .we(we_col_A)
      );

   buffer_8x16x32 row_mem_B(
    .addr(row_addr_B),
    .clk(clock),
    .din(din_row_B),
    .dout(row_in_B),
    .en(ENABLED),
    .we(we_row_B)
      );

   buffer_8x16x32 col_mem_B(
    .addr(col_addr_B),
    .clk(clock),
    .din(din_col_B),
    .dout(col_in_B),
    .en(ENABLED),
    .we(we_col_B)
      );

   //    Instantiation of validation logic
    comparator comparator (
    .clock(clock),
    .reset(reset),
         .threshold(threshold),
    .sq_error_row(row_sq_error),
    .sq_error_col(col_sq_error),
    .done_row(row_done),
    .done_col(col_done),
    .compare(compare),
    .match(match),
    .start_row(row_start),
    .start_col(col_start),
    .img_num(img_num),
         .add_user(add_user),
         .user_id(user_id),
         .busy(busy)
    );

      sq_error ROW_SQ_ERROR (
    .clock(clock),
    .reset(reset),
    .start(row_start),
    .Ain(row_in_A),
    .Bin(row_in_B),
         .img_num(img_num),
    .addr_A(row_addr_A),
    .addr_B(row_addr_B),
    .sq_error(row_sq_error),
    .done(row_done)
    );

      sq_error COL_SQ_ERROR (
    .clock(clock),
    .reset(reset),
    .start(col_start),
    .Ain(col_in_A),
    .Bin(col_in_B),
         .img_num(img_num),
    .addr_A(col_addr_A),
    .addr_B(col_addr_B),
    .sq_error(col_sq_error),
    .done(col_done)
    );

endmodule
```

## I.2 Comparator

```
‘timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//    6.111 Fingerprint ID
//    Kevin Amendt   kamendt@mit.edu
//
//    Create Date:    22:32:37 05/11/06
//    Design Name:   Comparator
//    Module Name:    comparator
//    Description:   takes in the Fourier Series Coefficients of two different
//                   signals and compares them by summing the error.  If this error
//                   is below a certain threshold, then it is considered a match.
//
//                   This module is the comparator.  It takes in the squared error
//                   from the row and columns, sums them, and compares to a threshold
//                   Also responsible for start and img_num commands to the sq_error modules
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module comparator(clock, reset,
                  sq_error_row, sq_error_col,
                  done_row, done_col,
                  compare, match,
                  start_row, start_col, img_num, busy, threshold,
                  add_user, user_id
                  );

   //    Parameters
   parameter error_width = 64-1;              //bit width of sq_error from sq_error module
   //parameter threshold   = 20;//64'd200000;         // Threshold value for comparison
```

```
//    Inputs
input                        clock, reset;
input    [error_width:0]     sq_error_row, sq_error_col;
input                        done_row, done_col, compare;
input    [63:0] threshold;

input add_user;
input [2:0] user_id;
//    Registered Inputs

//    Outputs
output                       match, start_row, start_col, busy;
output   [3:0]               img_num;

//    Regestered Outputs
reg                          match, start_row, start_col, busy;
reg      [3:0]               img_num;

//    Registers
reg      [3:0]               state, next;
reg      [error_width:0]     sq_error;
reg      [7:0]               entry_exists = 8'b0;

//    State Parameters
parameter   no_match_idle  = 0;
parameter   start_errors   = 1;
parameter   Wait           = 2;
parameter   sum            = 3;
parameter   test           = 4;
parameter   done_check     = 5;
parameter   match_idle     = 6;
parameter   reset_img_num  = 7;
parameter   inc_img_num    = 8;

//    Sequential Always Block
always @ (posedge clock or posedge reset)
   if(reset)   begin
      state <= no_match_idle;
      img_num <= 0;
      sq_error <= 0;
      entry_exists <= 8'b0;
   end
   else begin
      if (add_user) entry_exists[user_id] <= 1;
      state <= next;
```

```
         busy <= ~((next == match_idle) || (next == no_match_idle));
         if(next == sum)
            if(sq_error_row > sq_error_col) sq_error <= sq_error_row;
            else sq_error <= sq_error_col;
         else if(next == inc_img_num)  img_num <= img_num + 1;
         else if((next == no_match_idle) || (next == reset_img_num)) img_num <= 0;
         match <= (next == match_idle);
         start_row <= (next == start_errors);
         start_col <= (next == start_errors);
      end

   //    Combinational Always Block
   always @ (state or compare or done_row or done_col or match or sq_error or img_num )
      case(state)
         no_match_idle:
            if(compare) next = start_errors;
            else        next = no_match_idle;
         start_errors:
            next = Wait;
         Wait:
            if(done_row && done_col)   next = sum;
            else                       next = Wait;
         sum:
            next = test;
         test:
            if(sq_error < threshold &&
               entry_exists[img_num])  next = match_idle;
            else                       next = inc_img_num;
         inc_img_num:
            next = done_check;
         done_check:
            if(img_num < 8)   next = start_errors;
            else              next = no_match_idle;
         match_idle:
            if(compare)    next = reset_img_num;
            else           next = match_idle;
         reset_img_num:
            next = start_errors;
      endcase

endmodule
```

## I.3 Summed Squared Error

```
‘timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
```

```
//    6.111 Fingerprint ID
//    Kevin Amendt   kamendt@mit.edu
//
//
//    Create Date:    22:45:05 05/11/06
//    Design Name:    Validation
//    Module Name:    sq_error
//
//    Description:   takes in the Fourier Transform Coefficients of two different
//                   signals and compares them by summing the square of the error.
//                   This error is sent to a comparator to sum the col and row errors
//                   and compared that error with the threshold value.  Memory A is the
//                   scanned image, memory B is the database.  IMG_NUM refers to the image
//                   in memory B that is being fetched.  Should come from the comparator modul
//
//
//                      Default values:
//                      Calculating 16* FS coefficients
//       ‘              Can store 7 databased fingerprints
//                      8 bits per coefficient (max magnitude = 255)
//
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module sq_error(clock, reset, start, img_num, Ain, Bin, addr_A, addr_B, sq_error, done
               );

   //    Parameters
   parameter coef_width    = 32  -1;                           //# bits per coefficient
   parameter coef_number   = 16 -1;                           //# coefficients per image
   parameter error_width   = 64 -1;                              // width of error output da
   parameter addr_width    = 4 -1;                               //bitlength of coef_number

   //    Inputs
   input                      clock, reset;
   input                      start;
   input    [3:0]             img_num;
   input    [coef_width:0]    Ain, Bin;

   //    Registered Inputs
   reg      [coef_width:0]    mid_Ain, mid_Bin;
```

```
//    Outputs
output  [addr_width:0]     addr_A;
output  [6:0]              addr_B;
output  [error_width:0]    sq_error;
output                     done;

//    Registered Outputs
reg     [2:0]              state, next;
reg     [addr_width:0]     M_addr_A;
reg     [6:0]              M_addr_B;
reg     [error_width:0]    sq_error;
reg                        done;


//    State Names
parameter IDLE = 0;
parameter set_sq_error = 1;
parameter read0 = 2;
parameter read1 = 3;
parameter read2 = 4;
parameter error = 5;
parameter error2 = 6;
parameter finished = 7;

//    Sequential Always Block
always @ (posedge clock or posedge reset)
   if(reset) begin
      state <= IDLE;
      done <=1;
      sq_error <= 0;
      M_addr_A <= 0;
      M_addr_B <= 0;
   end
   else begin
      state <= next;
      if((next == error) && (M_addr_A == (coef_number)))  done <= 1;
      else if(start)   done <= 0;
      if(next == read2) begin
         mid_Ain <= Ain;
         mid_Bin <= Bin;
      end
      if(next == set_sq_error) begin
         sq_error <= 0;
         M_addr_A <= 0;
         M_addr_B <= img_num << 4;
```

```
        end
        if(next == IDLE)
           sq_error <= sq_error;
        if(next == error) begin
           if(Ain>Bin) sq_error <= sq_error + (mid_Ain-mid_Bin)*(mid_Ain-mid_Bin);
           else        sq_error <= sq_error + (mid_Bin-mid_Ain)*(mid_Bin-mid_Ain);
           M_addr_A <= M_addr_A + 1;
           M_addr_B <= M_addr_B + 1;
        end
     end

  //    Combinational Always Block
  always @ (state or start or done)
     case(state)
        IDLE:
           if(start) next = set_sq_error;
           else      next = IDLE;
        set_sq_error:
           next = read0;
        read0:
           next = read1;
        read1:
           next = read2;
        read2:
           next = error;
        error:
           if(~done) next = read0;
           else      next = finished;
        finished:
           next = IDLE;
     endcase

  //    Combinational output assignment
  assign addr_A = (state == IDLE) ? (4'hz):(M_addr_A);
  assign addr_B = (state == IDLE) ? (7'hz):(M_addr_B);

endmodule
```