Michael Huhs
6.111 Final Project Code

**//Write Module**

```
module write_module(clk, reset, master_in, data_enable, sram_addr, we, sram_data_out,
                    dct_data_out, RST, rdy_out, dct_data_in, cen, dimension, blocks,
start_address);
input clk, reset;

//Control interface
input [7:0] master_in;
input data_enable;
output reg [35:0] sram_data_out;
output reg [18:0] sram_addr;
output reg we;
output reg cen;


//DCT interface
output reg [7:0] dct_data_out;
output reg RST;
input rdy_out;
input [11:0] dct_data_in;
input [3:0] dimension;
input [13:0] blocks;
input [18:0] start_address;

reg [3:0] column_count;
reg [3:0] row_count;
reg [13:0] block_count;
//reg [3:0] size;
reg [2:0] state;

parameter idle = 0;
parameter wait_for_dct = 1;
parameter write_to_sram = 2;
parameter skip_write = 3;

always @ (posedge clk) begin
if (data_enable) dct_data_out <= master_in;
else dct_data_out <= 0;
end

always @ (posedge clk)

case (state)
```

```
idle: begin
        if (data_enable) begin
                state <= wait_for_dct;
                //dct_data_out <= master_in;
                RST <= 0;
                cen <= 0;
            end
            else begin
                sram_addr <= 0;
                sram_data_out <= 0;
                state <= idle;
                we <= 0;
                RST <= 1;
                column_count <= 0;
                row_count <= 0;
                block_count <= 0;
                //size <= dimension;
                cen <= 0;
                //dct_data_out <= 0;
            end
        end


wait_for_dct: begin
                        if (reset)
                        state <= idle;
                        else begin
                          if(!rdy_out) begin
                          state <= wait_for_dct;
                          sram_addr <= start_address;
                          //if (data_enable) dct_data_out <= master_in;
                          end
                          else begin
                          state <= write_to_sram;
                          //if (data_enable) dct_data_out <= master_in;
                          we <= 1;
                          cen <= 1;
                          sram_data_out <= {24'b0,dct_data_in};
                          column_count <= 1;
                          row_count <= 1;
                          block_count <= 1;
                          end
    end
                    end
```

```
write_to_sram: begin
                                        if (reset) begin
                                        state <= idle;
                                        RST <= 1;
                                        end
                                        else begin
                                                if(column_count < dimension) begin
                                                  //if (data_enable) dct_data_out <=
master_in;
                                                        sram_data_out <=
{24'b0,dct_data_in};
                                                        sram_addr <= sram_addr + 1;
                                                        column_count <= column_count + 1;
                                                        we <= 1;
                                                        cen <= 1;
                                                        state <= write_to_sram;
                                                end
                                                else begin
                                                        if (dimension == 8)begin
                                                        sram_data_out <=
{24'b0,dct_data_in};
                                                        sram_addr <= sram_addr + 1;
                                                        column_count <= column_count + 1;
                                                        we <= 1;
                                                        cen <= 1;
                                                        state <= write_to_sram;
                                                        end
                                                        else begin
                                                        state <= skip_write;
                                                        we <= 0;
                                                        cen <= 1;
                                                        //if (data_enable) dct_data_out <=
master_in;
                                                        column_count <= column_count + 1;
                                                        sram_addr <= sram_addr + 1;
                                                        sram_data_out <= 0;
                        //if ((row_count >= (size - 1)) && ((size - 1) > 0)) size <= (size - 1);
                                                        end
                                                end
                end
                                        end

skip_write: begin
                if (reset)
                                        state <= idle;
                                        else begin
```

```
                                        if(column_count < 8) begin
                                                column_count <= column_count + 1;
                                                state <= skip_write;
                                                cen <= 1;
                                                //if (data_enable) dct_data_out <=
master_in;
                                        end
                                        else begin
                                                if (row_count == 8) begin
                                                                                        if
(block_count >= blocks) begin
                                                                                        state <= idle;
                                                        we <= 0;
                                                                                        sram_addr <=
0;
                                            sram_data_out <= 0;
                                        RST <= 1;
                                            column_count <= 0;
                                            row_count <= 0;
                                            block_count <= 0;
                                            cen <= 0;
                                            end
                                                                                        else begin
                                                                                        row_count <=
1;
                                                                                        column_count
<= 1;
                                                                                        block_count
<= block_count + 1;
                                                                                        sram_data_out
<= {24'b0,dct_data_in};
                                                                                        we <= 1;
                                                                                        cen <= 1;
                                                                                        state <=
write_to_sram;
                                                                                        end
                                                        end
                                                else begin
                                                                if (row_count  >= dimension)
begin
                                                                                column_count <= 1;
                                                                                row_count <=
row_count + 1;
                                                                                we <= 0;
                                                                                cen <= 1;
                                                                                state <= skip_write;
```

```
                                                                    end
                                                                    else begin
                                                                column_count <= 1;
                                                                    row_count <= row_count +
1;

                                                                we <= 1;
                                                                    cen <= 1;
                                                                    sram_data_out <= {24'b0,
dct_data_in};

                                                                    state <= write_to_sram;
                                                                    end
                                                            end
                                                    end
                                            end
                                        end

default: begin
                                        state <= idle;
                                        we <= 0;
                                        sram_addr <= 0;
                                        sram_data_out <= 0;
                                        RST <= 1;
                                        column_count <= 0;
                                        row_count <= 0;
                                        block_count <= 0;
                                        cen <= 0;
                                        //dct_data_out <= 0;
end


endcase

endmodule
```

**//Read Module**

```
module read_module(clk, reset, read_frame, compression, blocks, start_address,
ram_address,
                sram_read_data, idct_data_out, idct_data_in, rdy, RST, master_out,
data_ready, done);
//image size is 480 x 712
//(480 x 712 / 8) = 5340 blocks
// 5340 * 64 = 341,760 pixels + 93 for latency = done goes high 341,853 clock cycles
from rdy high
input clk, reset, read_frame;
input [18:0] start_address;
input    [3:0] compression;
input [35:0] sram_read_data;
input [7:0] idct_data_in;
input [13:0] blocks;

output reg rdy, RST, done;
output reg [18:0] ram_address;
output reg [11:0] idct_data_out;
output reg [7:0] master_out;
output reg data_ready;

reg [3:0] read_column_count;
reg [3:0] read_row_count;
reg [13:0] read_block_count;

reg [2:0] state;
reg [18:0] count;
reg [3:0] wait_count;

parameter idle = 0;
parameter wait_for_sram =1;
parameter pass_to_idct = 2;
parameter pass_zeros = 3;
parameter idct_output = 4;

always @ (posedge clk) begin
        if (count > 93) begin
        master_out <= idct_data_in;
        data_ready <= 1;
        end
        else begin
        master_out <= 0;
        data_ready <= 0;
        end
end
```

```
always @ (posedge clk)

case (state)

idle: begin
        if (read_frame) begin
                state <= wait_for_sram;
                ram_address <= start_address;
                wait_count <= 2;
                RST <= 0;
        end
        else begin
                state <= idle;
                rdy <=0;
                RST <= 1;
                wait_count <= 0;
                ram_address <= 0;
          idct_data_out <= 0;
                read_column_count <=0;
                read_row_count <=0;
          read_block_count <=0;
                count <= 0;
                done <= 0;
        end
end

wait_for_sram: begin
        if (reset)
        state <= idle;
        else begin
                        if (wait_count > 0)begin
                                ram_address <= ram_address + 1;
                                wait_count <= wait_count - 1;
                        end
                        else begin
                                state <= pass_to_idct;
                                rdy <= 1;
                                count <= count + 1;
                                idct_data_out <= sram_read_data[11:0];
                                read_column_count <= 1;
                                read_row_count <= 1;
                                ram_address <= ram_address + 1;
                        end
   end
end
```

```
pass_to_idct: begin
        if (reset)
        state <= idle;
        else begin
          if (read_column_count == (compression - 1))begin
                  if (compression == 7) ram_address <= ram_address + 1;
                        state <= pass_zeros;
                        idct_data_out <= sram_read_data[11:0];
                        read_column_count <= read_column_count + 1;
                        wait_count <= (7 - compression);
                        count <= count + 1;
                end
                else if (read_column_count >= (compression - 2)) begin
                        state <= pass_to_idct;
                        idct_data_out <= sram_read_data[11:0];
                        read_column_count <= read_column_count + 1;
                        count <= count + 1;
                end
                else begin
                        state <= pass_to_idct;
                        idct_data_out <= sram_read_data[11:0];
                        read_column_count <= read_column_count + 1;
                        ram_address <= ram_address + 1;
                        count <= count + 1;
                end
        end
end

pass_zeros: begin
        if (reset)
        state <= idle;
        else begin
                if (read_column_count == 8) begin
                        if (read_row_count == 8) begin
                                if (read_block_count == (blocks - 1))begin
                                        state <= idct_output;
                                                count <= count + 1;
                                                RST <= 0;
                                                wait_count <= 0;
                                                ram_address <= 0;
                                        idct_data_out <= 0;
                                                read_column_count <=0;
                                                read_row_count <=0;
                                        read_block_count <=0;
                        end
```

```
                        else begin
                                read_column_count <= 1;
                                read_row_count <= 1;
                                read_block_count <= read_block_count + 1;
                                state <= pass_to_idct;
                                ram_address <= ram_address + 1;
                                idct_data_out <= sram_read_data[11:0];
                                count <= count + 1;
                        end
                end
                else if (read_row_count >= compression) begin
                        state <= pass_zeros;
                        idct_data_out <= 0;
                        read_row_count <= read_row_count + 1;
                        read_column_count <= 1;
                  count <= count + 1;
        end
                else begin
                        state <= pass_to_idct;
                        ram_address <= ram_address + 1;
                        idct_data_out <= sram_read_data[11:0];
                        read_row_count <= read_row_count + 1;
                        read_column_count <= 1;
                        count <= count + 1;
                end
            end
                else begin
                        state <= pass_zeros;
                        if (read_row_count == 8 && (read_column_count == 7 ||
read_column_count ==            6)) begin
                                idct_data_out <= 0;
                                read_column_count <= read_column_count + 1;
                                ram_address <= ram_address + 1;
                                count <= count + 1;
        end
                        else if (read_row_count >= compression) begin
                                idct_data_out <= 0;
                                read_column_count <= read_column_count + 1;
                                count <= count + 1;
                        end
                        else begin
                                if (wait_count <= 1) begin
                                        ram_address <= ram_address + 1;
                                        idct_data_out <= 0;
                                        read_column_count <= read_column_count + 1;
                                        count <= count + 1;
```

```
                                end
                                else begin
                                        wait_count <= wait_count - 1;
                                        idct_data_out <= 0;
                                        read_column_count <= read_column_count + 1;
                                        count <= count + 1;
                                end
                        end
                end
          end
end

idct_output: begin
if (count == 285)begin          //for picture size = 3 blocks
state <= idct_output;
rdy <= 0;
RST <= 1;
done <= 1;
end
else if (count == 286)
state <= idle;
else
count <= count + 1;
end


default: begin
     state <= idle;
                rdy <=0;
                RST <= 1;
                wait_count <= 0;
                ram_address <= 0;
          idct_data_out <= 0;
                read_column_count <=0;
                read_row_count <=0;
          read_block_count <=0;
                count <= 0;
                done <= 0;
end


endcase
endmodule
//unsigned to signed
module sign_y(clk, reset, RST, RST_delay, Y_in, Y_out);
input clk, reset, RST;
```

```
input [7:0] Y_in;
output reg [7:0] Y_out;
output reg RST_delay;


reg signed [8:0] sign_extend;
reg RST_reg;

always @ (posedge clk) begin
if (reset) begin
sign_extend <= 0;
Y_out <= 0;
RST_reg <= RST;
RST_delay <= RST_reg;
end
else begin
sign_extend <= {1'b0,Y_in};
Y_out <= sign_extend - 8'b10000000;
RST_reg <= RST;
RST_delay <= RST_reg;
end
end

endmodule
```

**//Signed to unsigned**

```
module signed_to_unsigned (clk, reset, Y_in, Y_out);

input clk, reset;
input signed [7:0] Y_in;
output reg [7:0] Y_out;


reg signed [8:0] ytemp;


always @ (posedge clk) begin
if (reset) begin
Y_out <= 0;
ytemp <= 0;
end
else begin
ytemp <= (Y_in + 8'b1000_0000);
Y_out <= ytemp[7:0];
end
end
```

```
endmodule
```

**//ZBT driver (from fall 2005 website)**

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


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

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

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

  reg [1:0]   we_delay;

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

  // create two-stage pipeline for write data

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

  // wire to ZBT RAM signals

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

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

endmodule
```

```
//integrated write module
module write_compression_test(clk, reset, dimension, blocks, start_address,
input_enable, x_in,
                        ram0_clk, ram0_we_b, ram0_cen_b, ram0_address, sram_read_data,
ram0_data, idct_2d);

input clk, reset;
input [13:0] blocks;
input [3:0] dimension;
input [18:0] start_address;
input input_enable;
input [7:0] x_in;

output ram0_clk;
output ram0_we_b;
output ram0_cen_b;
output [18:0] ram0_address;
output [35:0] sram_read_data;
output [35:0] ram0_data;
output [7:0] idct_2d;

wire [7:0] sign_uncompressed;
wire RST, RST_delay;
wire start;
wire [11:0] dct_2d;
wire rdy_sig;
wire [7:0] uncompressed;
wire [18:0] sram_addr;
wire we, cen;
wire [35:0] sram_data_out;
wire [35:0] sram_read_data;


sign_y unsign_to_sign (clk, reset, RST, RST_delay, uncompressed, sign_uncompressed);

dct forward (clk, RST_delay, sign_uncompressed, dct_2d, rdy_sig);

idct inverse (clk, ~cen, cen, sram_data_out[11:0], idct_2d);

write_module writeit (.clk(clk), .reset(reset), .master_in(x_in),
.data_enable(input_enable), .sram_addr(sram_addr), .we(we),
.sram_data_out(sram_data_out),
                .dct_data_out(uncompressed), .RST(RST), .rdy_out(rdy_sig),
.dct_data_in(dct_2d), .cen(cen), .dimension(dimension), .blocks(blocks),
.start_address(start_address));
```

```
zbt_6111 memory (.clk(clk), .cen(cen), .we(we), .addr(sram_addr),
.write_data(sram_data_out), .read_data(sram_read_data),
                            .ram_clk(ram0_clk), .ram_we_b(ram0_we_b),
.ram_address(ram0_address), .ram_data(ram0_data), .ram_cen_b(ram0_cen_b));

endmodule
```

```
//integrated read module
module read_compressed_test(clk, reset, read_button, ram_data, master_out,
data_out_ready, ram_address, ram_cen_b, ram_we_b, ram_clk);

input clk, reset, read_button;
input [35:0] ram_data;
output [7:0] master_out;
output data_out_ready, ram_cen_b, ram_we_b, ram_clk;
output [18:0] ram_address;

wire read_frame, read_reset, RST, rdy_in, we, cen;
wire [11:0] data_to_idct;
wire [7:0] data_from_idct;
wire [18:0] addr;
wire [35:0] sram_read_data;
wire [3:0] compression;
wire [13:0] blocks;
wire [18:0] start_address;
wire [35:0] write_data;

assign we = 0;
assign cen = 1;
assign write_data = 0;
assign compression = 7;
assign blocks = 3;
assign start_address = 1;

read_register readreg (clk, reset, read_button, read_frame, read_reset);

idct inverse (clk, RST, rdy_in, data_to_idct,data_from_idct);

read_module read_control (clk, reset, read_frame, compression, blocks, start_address,
addr,
                sram_read_data, data_to_idct, data_from_idct, rdy_in, RST, master_out,
data_out_ready, read_reset);

//zbt module has been changed so that ram_data is an input rather than an inout for test
benching purposes.
zbt_6111 memory (clk, cen, we, addr, write_data, sram_read_data,
                                ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

endmodule


//read register
module read_register(clk, reset, read_button, read_frame, read_reset);
```

```
input clk, reset, read_button, read_reset;
output reg read_frame;

always @ (posedge clk)

if (read_reset||reset) read_frame = 0;
else if (read_button||read_frame) read_frame =1;
else read_frame = 0;

endmodule
```

**//upper level instantiation**

```
wire reset, write_input_enable, read_frame;
wire DRST_delay, DRST, IRST;
wire dct_data_rdy, idct_input_rdy;
wire [7:0]sign_data_to_dct, data_to_dct;
wire [7:0]sign_data_from_idct, signed_master_out;
wire [11:0] data_from_dct, data_to_idct;
wire [3:0] dimension;
wire [13:0] blocks;
wire [18:0] start_address;
wire [18:0] read_sram_addr, write_sram_addr, sram_addr;
wire we, write_we;
wire cen, write_cen;
wire [35:0] sram_data_out, sram_read_data;
wire [7:0] master_out, master_in;
wire done, read_out_ready;

assign dimension = 4'b0011;  //{switch4, switch3, switch2, switch1};
assign blocks = 14'b0000_0000_0000_11;
assign start_address = 19'b0000_0000_0000_0000_001;
assign master_in = 8'b00000011;

debounce reset_debounce (~button_enter, clock_27mhz, ~button_enter, reset);
debounce read_debounce (~button3, clock_27mhz, ~button3, read_frame);
debounce write_debounce (~button2, clock_27mhz, ~button2, write_input_enable);

dct dct_1 ( .CLK(clock_27mhz), .RST(DRST_delay), .xin(sign_data_to_dct),
.dct_2d(data_from_dct), .rdy_out(dct_data_rdy));
idct idct_1 (.CLK(clock_27mhz), .RST(IRST), .dct_2d(data_to_idct),
.rdy_in(idct_input_rdy), .idct_2d(signed_data_from_idct));

signed_to_unsigned sign_to_unsign (.clk(clk_27mhz), .reset(reset),
.Y_in(signed_master_out), .Y_out(master_out));
```

```
sign_y unsign_to_sign (.clk(clock_27mhz), .reset(reset), .RST(DRST),
.RST_delay(DRST_delay), .Y_in(data_to_dct), .Y_out(sign_data_to_dct));
//////////////////////////////////////////////////////////////
assign sram_addr = read_frame? read_sram_addr: write_sram_addr;
assign we = read_frame? 0: write_we;


assign cen = read_frame? 1: write_cen;
//////////////////////////////////////////////////////////////////////////////////

read_module readit (.clk(clock_27mhz), .reset(reset), .read_frame(read_frame),
.compression(dimension), .blocks(blocks), .start_address(start_address),
                .ram_address(read_sram_addr), .sram_read_data(sram_read_data),
                 .idct_data_out(data_to_idct), .idct_data_in(signed_data_from_idct),
.rdy(idct_input_rdy), .RST(IRST), .master_out(signed_master_out),
.data_ready(read_out_ready), .done(done));

write_module writeit (.clk(clock_27mhz), .reset(reset), .master_in(master_in),
.data_enable(write_input_enable), .sram_addr(write_sram_addr), .we(we),
.sram_data_out(sram_data_out),
                 .dct_data_out(data_to_dct), .RST(DRST), .rdy_out(dct_data_rdy),
.dct_data_in(data_from_dct), .cen(cen), .dimension(dimension), .blocks(blocks),
.start_address(start_address));

zbt_6111 memory (.clk(clock_27mhz), .cen(cen), .we(we), .addr(sram_addr),
.write_data(sram_data_out), .read_data(sram_read_data),
                                .ram_clk(ram0_clk), .ram_we_b(ram0_we_b),
.ram_address(ram0_address), .ram_data(ram0_data), .ram_cen_b(ram0_cen_b));

assign analyzer2_clock = clock_27mhz;
assign analyzer2_data = {write_input_enable, clock_27mhz, we, dct_data_rdy,
sram_data_out[11:0]};
assign analyzer4_data = {data_to_dct, sram_addr[7:0]};
assign analyzer4_clock = clock_27mhz;

endmodule

//DCT module from Xilinx applications notes
module dct ( CLK, RST, xin,dct_2d,rdy_out);
output [11:0] dct_2d;
input CLK, RST;
input[7:0] xin; /* input */
output rdy_out;
wire[11:0] dct_2d;

/* constants */
```

```
reg[7:0] memory1a, memory2a, memory3a, memory4a;

/* 1D section */
/* The max value of a pixel after processing (to make their expected mean to zero)
is 127. If all the values in a row are 127, the max value of the product terms
would be (127*8)*(23170/256) and that of z_out_int would be (127*8)*23170/65536.
This value divided by 2raised to 16 is equivalent to ignoring the 16 lsb bits of the value */

reg[7:0] xa0_in, xa1_in, xa2_in, xa3_in, xa4_in, xa5_in, xa6_in, xa7_in;
reg[8:0] xa0_reg, xa1_reg, xa2_reg, xa3_reg, xa4_reg, xa5_reg, xa6_reg, xa7_reg;
reg[7:0] addsub1a_comp,addsub2a_comp,addsub3a_comp,addsub4a_comp;
reg[9:0] add_sub1a,add_sub2a,add_sub3a,add_sub4a;
reg save_sign1a, save_sign2a, save_sign3a, save_sign4a;
reg[18:0] p1a,p2a,p3a,p4a;
wire[35:0] p1a_all,p2a_all,p3a_all,p4a_all;
reg[1:0] i_wait;
reg toggleA;
reg[18:0] z_out_int1,z_out_int2;
reg[18:0] z_out_int;
wire[10:0] z_out_rnd;
wire[10:0] z_out;
integer indexi;

/* clks and counters */
reg[3:0] cntr12 ;
reg[3:0] cntr8;
reg[6:0] cntr79;
reg[6:0] wr_cntr,rd_cntr;
reg[6:0] cntr92;

/* memory section */
reg[10:0] data_out;
wire en_ram1,en_dct2d;
reg en_ram1reg,en_dct2d_reg;
reg[10:0] ram1_mem[63:0],ram2_mem[63:0]; // add the following to infer block RAM in
synlpicity
                                //    synthesis syn_ramstyle = "block_ram"  //shd be within /*..*/

/* 2D section */
wire[10:0] data_out_final;
reg[10:0] xb0_in, xb1_in, xb2_in, xb3_in, xb4_in, xb5_in, xb6_in, xb7_in;
reg[11:0] xb0_reg, xb1_reg, xb2_reg, xb3_reg, xb4_reg, xb5_reg, xb6_reg, xb7_reg;
reg[11:0] add_sub1b,add_sub2b,add_sub3b,add_sub4b;
reg[10:0] addsub1b_comp,addsub2b_comp,addsub3b_comp,addsub4b_comp;
reg save_sign1b, save_sign2b, save_sign3b, save_sign4b;
```

```
reg[19:0] p1b,p2b,p3b,p4b;
wire[35:0] p1b_all,p2b_all,p3b_all,p4b_all;
reg toggleB;
reg[19:0] dct2d_int1,dct2d_int2;
reg[19:0] dct_2d_int;
wire[11:0] dct_2d_rnd;


/*  1D-DCT BEGIN */

// store  1D-DCT constant coeeficient values for multipliers */

always @ (posedge RST or posedge CLK)
  begin
  if (RST)
     begin
     memory1a <= 8'd0; memory2a <= 8'd0; memory3a <= 8'd0; memory4a <= 8'd0;
     end
  else
     begin
            case (indexi)
       0 : begin memory1a <= 8'd91;
              memory2a <= 8'd91;
              memory3a <= 8'd91;
              memory4a <= 8'd91;end
       1 : begin memory1a <= 8'd126;
              memory2a <= 8'd106;
              memory3a <= 8'd71;
              memory4a <= 8'd25;end
       2 : begin memory1a <= 8'd118;
              memory2a <= 8'd49;
              memory3a[7] <= 1'b1; memory3a[6:0] <= 7'd49;//-8'd49;
              memory4a[7] <= 1'b1; memory4a[6:0] <= 7'd118;// end -8'd118;end
              end
       3 : begin memory1a <= 8'd106;
              memory2a[7] <= 1'b1; memory2a[6:0] <= 7'd25;//-8'd25;
              memory3a[7] <= 1'b1; memory3a[6:0] <= 7'd126;//-8'd126;
              memory4a[7] <= 1'b1; memory4a[6:0] <= 7'd71;end//-8'd71;end
       4 : begin memory1a <= 8'd91;
              memory2a[7] <= 1'b1; memory2a[6:0] <= 7'd91;//-8'd91;
              memory3a[7] <= 1'b1; memory3a[6:0] <= 7'd91;//-8'd91;
              memory4a <= 8'd91;end
       5 : begin memory1a <= 8'd71;
              memory2a[7] <= 1'b1; memory2a[6:0] <= 7'd126;//-8'd126;
              memory3a <= 8'd25;
              memory4a <= 8'd106;end
```

```
        6 : begin memory1a <= 8'd49;
                memory2a[7] <= 1'b1; memory2a[6:0] <= 7'd118;//-8'd118;
                memory3a <= 8'd118;
                memory4a[7] <= 1'b1; memory4a[6:0] <= 7'd49;end//-8'd49;end
        7 : begin memory1a <= 8'd25;
                memory2a[7] <= 1'b1; memory2a[6:0] <= 7'd71;//-8'd71;
                memory3a <= 8'd106;
                memory4a[7] <= 1'b1; memory4a[6:0] <= 7'd126;end//-8'd126;end
      endcase
     end
end


/* 8-bit input shifted 8 times thru a shift register*/
always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     xa0_in <= 8'b0; xa1_in <= 8'b0; xa2_in <= 8'b0; xa3_in <= 8'b0;
     xa4_in <= 8'b0; xa5_in <= 8'b0; xa6_in <= 8'b0; xa7_in <= 8'b0;
     end
  else
     begin
     xa0_in <= xin; xa1_in <= xa0_in; xa2_in <= xa1_in; xa3_in <= xa2_in;
     xa4_in <= xa3_in; xa5_in <= xa4_in; xa6_in <= xa5_in; xa7_in <= xa6_in;
     end
  end

/* shifted inputs registered every 8th clk (using cntr8)*/

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     cntr8 <= 4'b0;
     end
  else if (cntr8 < 4'b1000)
     begin
     cntr8 <= cntr8 + 1;
     end
  else
     begin
     cntr8 <= 4'b0001;
     end
  end
```

```
always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     xa0_reg <= 9'b0; xa1_reg <= 9'b0; xa2_reg <= 9'b0; xa3_reg <= 9'b0;
     xa4_reg <= 9'b0; xa5_reg <= 9'b0; xa6_reg <= 9'b0; xa7_reg <= 9'b0;
     end
  else if (cntr8 == 4'b1000)
     begin
     xa0_reg <= {xa0_in[7],xa0_in}; xa1_reg <= {xa1_in[7],xa1_in};
     xa2_reg <= {xa2_in[7],xa2_in}; xa3_reg <= {xa3_in[7],xa3_in};
     xa4_reg <= {xa4_in[7],xa4_in}; xa5_reg <= {xa5_in[7],xa5_in};
     xa6_reg <= {xa6_in[7],xa6_in}; xa7_reg <= {xa7_in[7],xa7_in};
     end
  else
     begin
     end
  end

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     toggleA <= 1'b0;
     end
  else
     begin
     toggleA <= ~toggleA;
     end
  end


/* adder / subtractor block */

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     add_sub1a <= 10'b0; add_sub2a <= 10'b0; add_sub3a <= 10'b0; add_sub4a <= 10'b0;
     end
  else
     begin
     if (toggleA == 1'b1)
               begin
        add_sub1a <= (xa7_reg + xa0_reg);
        add_sub2a <= (xa6_reg + xa1_reg);
```

```
        add_sub3a <= (xa5_reg + xa2_reg);
        add_sub4a <= (xa4_reg + xa3_reg);
        end
      else if (toggleA == 1'b0)
              begin
              add_sub1a <= (xa7_reg - xa0_reg);
        add_sub2a <= (xa6_reg - xa1_reg);
        add_sub3a <= (xa5_reg - xa2_reg);
        add_sub4a <= (xa4_reg - xa3_reg);
              end
      end
  end

/* multiply the outputs of the add/sub block with the 8 sets of stored coefficients */
/* The inputs are shifted thru 8 registers in 8 clk cycles. The ouput of the shift
registers are registered at the 9th clk. The values are then added or subtracted at the 10th
clk. The first mutiplier output is obtained at the 11th clk. Memoryx[0] shd be accessed
at the 11th clk*/

/*wait state counter */

always @ (posedge RST or posedge CLK)
  begin
  if (RST)
     begin
     i_wait <= 2'b01;
     end
  else  if (i_wait != 2'b00)
    begin
    i_wait  <= i_wait - 1;
    end
  else
    begin
    i_wait  <= 2'b00;
    end
  end

// First valid add_sub appears at the 10th clk (8 clks for shifting inputs,
// 9th clk for registering shifted input and 10th clk for add_sub
// to synchronize the i value to the add_sub value, i value is incremented
// only after 10 clks using i_wait
/* sign and magnitude separated here. magnitude of 9 bits is stored in *comp */

always @ (posedge RST or posedge CLK)
begin
   if (RST)
```

```
      begin
        addsub1a_comp <= 9'b0; save_sign1a <= 1'b0;
      end
    else
      begin
      case (add_sub1a[9])
      1'b0: begin
            addsub1a_comp <= add_sub1a; save_sign1a <= 1'b0;
            end
      1'b1: begin
            addsub1a_comp <= (-add_sub1a) ; save_sign1a <= 1'b1;
            end
      endcase
      end
end

always @ (posedge RST or posedge CLK)
begin
   if (RST)
      begin
        addsub2a_comp <= 9'b0; save_sign2a <= 1'b0;
      end
   else
      begin
      case (add_sub2a[9])
      1'b0: begin
            addsub2a_comp <= add_sub2a; save_sign2a <= 1'b0;
            end
      1'b1: begin
            addsub2a_comp <= (-add_sub2a) ; save_sign2a <= 1'b1;
            end
      endcase
      end
end

always @ (posedge RST or posedge CLK)
begin
   if (RST)
      begin
        addsub3a_comp <= 9'b0; save_sign3a <= 1'b0;
      end
   else
      begin
      case (add_sub3a[9])
      1'b0: begin
            addsub3a_comp <= add_sub3a; save_sign3a <= 1'b0;
```

```
            end
       1'b1: begin
            addsub3a_comp <= (-add_sub3a); save_sign3a <= 1'b1;
            end
       endcase
       end
end

always @ (posedge RST or posedge CLK)
begin
   if (RST)
     begin
      addsub4a_comp <= 9'b0; save_sign4a <= 1'b0;
     end
   else
     begin
     case (add_sub4a[9])
     1'b0: begin
            addsub4a_comp <= add_sub4a; save_sign4a <= 1'b0;
            end
     1'b1: begin
            addsub4a_comp <= (-add_sub4a); save_sign4a <= 1'b1;
            end
     endcase
     end
end

    assign p1a_all = addsub1a_comp * memory1a[6:0];/* 9 bits * 7 bits = 16 bits*/
    assign p2a_all = addsub2a_comp * memory2a[6:0];
    assign p3a_all = addsub3a_comp * memory3a[6:0];
    assign p4a_all = addsub4a_comp * memory4a[6:0];



always @ (posedge RST or posedge CLK)
 begin
   if (RST)
     begin
      p1a <= 18'b0; p2a <= 18'b0; p3a <= 18'b0; p4a <= 18'b0; indexi<= 7;
     end /* p*a is extended to one more bit to take into acoount the sign */
   else if (i_wait == 2'b00)
     begin


      p1a <= (save_sign1a ^ memory1a[7]) ? (-p1a_all[15:0]) :(p1a_all[15:0]);
      p2a <= (save_sign2a ^ memory2a[7]) ? (-p2a_all[15:0]) :(p2a_all[15:0]);
```

```
      p3a <= (save_sign3a ^ memory3a[7]) ? (-p3a_all[15:0]) :(p3a_all[15:0]);
      p4a <= (save_sign4a ^ memory4a[7]) ? (-p4a_all[15:0]) :(p4a_all[15:0]);


      if (indexi == 7)
        indexi <= 0;
      else
        indexi <= indexi + 1;
      end
  end

/* Final adder. Adding the ouputs of the 4 multipliers */

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     z_out_int1 <= 19'b0; z_out_int2 <= 19'b0; z_out_int <= 19'b0;
     end
  else
     begin
     z_out_int1 <= (p1a + p2a);
     z_out_int2 <= (p3a + p4a);
     z_out_int <= (z_out_int1 + z_out_int2);
     end
  end

// rounding of the value

assign z_out_rnd = z_out_int[7] ? (z_out_int[18:8] + 1'b1) : z_out_int[18:8];

/* 1 sign bit, 11 data bit */
assign z_out = z_out_rnd;

/* 1D-DCT END */

/* tranpose memory to store intermediate Z coeeficients */
/* store the 64 coeeficients in the first 64 locations of the RAM */
/* first valid adder output is at the 15th clk. (input reg + 8 bit SR + add_sub + comp.
Signal + reg prod
+ 2 partial prod adds) So the RAM is enabled at the 15th clk)*/

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
```

```
      cntr12 <= 4'b0;
      end
   else
      begin
      cntr12 <= cntr12 + 1;
      end
   end

/* enable RAM at the 14th clk after RST goes inactive */

assign en_ram1 = RST ? 1'b0 : (cntr12== 4'b1101) ? 1'b1 : en_ram1;

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
            en_ram1reg <= 1'b0;
      end
   else
      begin
      en_ram1reg <= en_ram1 ;
      end
   end

/* After the RAM is enabled, data is written into the RAM1 for 64 clk cycles. Data is
written in into
each consecutive location . After 64 locations are written into, RAM1 goes into read
mode and RAM2 goes into
write mode. The cycle then repeats.
For either RAM, data is written into each consecutive location. However , data is read in
a different order. If data
is assumed to be written in each row at a time, in an 8x8 matrix, data is read each column
at a time. ie., after
the first data is read out, every eight data is read out . Then the 2nd data is read out
followed be every 8th.

the write is as follows:
1w(ram_locn1) 2w(ram_locn2) 3w(ram_locn3) 4w(ram_locn4) 5w(ram_locn5)
6w(ram_locn6) 7w(ram_locn7) 8w(ram_locn8)
9w(ram_locn9) 10w(ram_locn10) 11w(ram_locn11) 12w(ram_locn12) 13w(ram_locn13)
14w(ram_locn14) 15w(ram_locn15) 16w(ram_locn16)
.....
57w(ram_locn57) 58w(ram_locn58) 59w(ram_locn59) 60w(ram_locn60)
61w(ram_locn61) 62w(ram_locn62) 63w(ram_locn63) 64w(ram_locn64)

the read is as follows:
```

1r(ram_locn1)  9r(ram_locn2) . . . 57r(ram_locn8)
2r(ram_locn9) 10r(ram_locn10) . . . 58r(ram_locn16)
3r(ram_locn17) 11r(ram_locn18) . . . 59r(ram_locn24)
4r(ram_locn25) 12r(ram_locn26) . . . 60r(ram_locn32)
5r(ram_locn33) 13r(ram_locn34) . . . 61r(ram_locn40)
6r(ram_locn41) 14r(ram_locn42) . . . 62r(ram_locn48)
7r(ram_locn49) 15r(ram_locn50) . . . 63r(ram_locn56)
8r(ram_locn57) 16r(ram_locn58) . . . 64r(ram_locn64)

where "xw" is the xth write and "ram_locnx" is the xth ram location and "xr" is the xth read. Reading
is advanced by the read counter rd_cntr, nd writing by the write counter wr_cntr. */

```
always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
          rd_cntr[5:3] <= 3'b111;
     end
  else
     begin
          if (en_ram1reg == 1'b1)
             rd_cntr[5:3] <= rd_cntr[5:3] + 1;
          end
  end

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
          rd_cntr[2:0] <= 3'b111;
     end
  else
     begin
          if (en_ram1reg == 1'b1 && rd_cntr[5:3] == 3'b111)
             rd_cntr[2:0] <= rd_cntr[2:0] + 1;
     end
  end

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
          rd_cntr[6] <= 1'b1;
     end
```

```
  else
     begin
     if (en_ram1reg == 1'b1 && rd_cntr[5:0] == 6'b111111)
       rd_cntr[6] <= ~rd_cntr[6];
     end
  end


always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     wr_cntr <= 7'b1111111;
     end
  else begin
     if (en_ram1reg == 1'b1)
       wr_cntr <= wr_cntr + 1;
           else
             wr_cntr <= 7'b0;
     end
  end


initial
begin
ram2_mem[0] <= 16'b0; ram2_mem[1] <= 16'b0; ram2_mem[2] <= 16'b0; ram2_mem[3]
<= 16'b0; ram2_mem[4] <= 16'b0;
ram2_mem[5] <= 16'b0; ram2_mem[6] <= 16'b0; ram2_mem[7] <= 16'b0; ram2_mem[8]
<= 16'b0; ram2_mem[9] <= 16'b0;
ram2_mem[10] <= 16'b0; ram2_mem[11] <= 16'b0; ram2_mem[12] <= 16'b0;
ram2_mem[13] <= 16'b0; ram2_mem[14] <= 16'b0;
ram2_mem[15] <= 16'b0; ram2_mem[16] <= 16'b0; ram2_mem[17] <= 16'b0;
ram2_mem[18] <= 16'b0; ram2_mem[19] <= 16'b0;
ram2_mem[20] <= 16'b0; ram2_mem[21] <= 16'b0; ram2_mem[22] <= 16'b0;
ram2_mem[23] <= 16'b0; ram2_mem[24] <= 16'b0;
ram2_mem[25] <= 16'b0; ram2_mem[26] <= 16'b0; ram2_mem[27] <= 16'b0;
ram2_mem[28] <= 16'b0; ram2_mem[29] <= 16'b0;
ram2_mem[30] <= 16'b0; ram2_mem[31] <= 16'b0; ram2_mem[32] <= 16'b0;
ram2_mem[33] <= 16'b0; ram2_mem[34] <= 16'b0;
ram2_mem[35] <= 16'b0; ram2_mem[36] <= 16'b0; ram2_mem[37] <= 16'b0;
ram2_mem[38] <= 16'b0; ram2_mem[39] <= 16'b0;
ram2_mem[40] <= 16'b0; ram2_mem[41] <= 16'b0; ram2_mem[42] <= 16'b0;
ram2_mem[43] <= 16'b0; ram2_mem[44] <= 16'b0;
ram2_mem[45] <= 16'b0; ram2_mem[46] <= 16'b0; ram2_mem[47] <= 16'b0;
ram2_mem[48] <= 16'b0; ram2_mem[49] <= 16'b0;
```

```
ram2_mem[50] <= 16'b0; ram2_mem[51] <= 16'b0; ram2_mem[52] <= 16'b0;
ram2_mem[53] <= 16'b0; ram2_mem[54] <= 16'b0;
ram2_mem[55] <= 16'b0; ram2_mem[56] <= 16'b0; ram2_mem[57] <= 16'b0;
ram2_mem[58] <= 16'b0; ram2_mem[59] <= 16'b0;
ram2_mem[60] <= 16'b0; ram2_mem[61] <= 16'b0; ram2_mem[62] <= 16'b0;
ram2_mem[63] <= 16'b0;

ram1_mem[0] <= 16'b0; ram1_mem[1] <= 16'b0; ram1_mem[2] <= 16'b0; ram1_mem[3]
<= 16'b0; ram1_mem[4] <= 16'b0;
ram1_mem[5] <= 16'b0; ram1_mem[6] <= 16'b0; ram1_mem[7] <= 16'b0; ram1_mem[8]
<= 16'b0; ram1_mem[9] <= 16'b0;
ram1_mem[10] <= 16'b0; ram1_mem[11] <= 16'b0; ram1_mem[12] <= 16'b0;
ram1_mem[13] <= 16'b0; ram1_mem[14] <= 16'b0;
ram1_mem[15] <= 16'b0; ram1_mem[16] <= 16'b0; ram1_mem[17] <= 16'b0;
ram1_mem[18] <= 16'b0; ram1_mem[19] <= 16'b0;
ram1_mem[20] <= 16'b0; ram1_mem[21] <= 16'b0; ram1_mem[22] <= 16'b0;
ram1_mem[23] <= 16'b0; ram1_mem[24] <= 16'b0;
ram1_mem[25] <= 16'b0; ram1_mem[26] <= 16'b0; ram1_mem[27] <= 16'b0;
ram1_mem[28] <= 16'b0; ram1_mem[29] <= 16'b0;
ram1_mem[30] <= 16'b0; ram1_mem[31] <= 16'b0; ram1_mem[32] <= 16'b0;
ram1_mem[33] <= 16'b0; ram1_mem[34] <= 16'b0;
ram1_mem[35] <= 16'b0; ram1_mem[36] <= 16'b0; ram1_mem[37] <= 16'b0;
ram1_mem[38] <= 16'b0; ram1_mem[39] <= 16'b0;
ram1_mem[40] <= 16'b0; ram1_mem[41] <= 16'b0; ram1_mem[42] <= 16'b0;
ram1_mem[43] <= 16'b0; ram1_mem[44] <= 16'b0;
ram1_mem[45] <= 16'b0; ram1_mem[46] <= 16'b0; ram1_mem[47] <= 16'b0;
ram1_mem[48] <= 16'b0; ram1_mem[49] <= 16'b0;
ram1_mem[50] <= 16'b0; ram1_mem[51] <= 16'b0; ram1_mem[52] <= 16'b0;
ram1_mem[53] <= 16'b0; ram1_mem[54] <= 16'b0;
ram1_mem[55] <= 16'b0; ram1_mem[56] <= 16'b0; ram1_mem[57] <= 16'b0;
ram1_mem[58] <= 16'b0; ram1_mem[59] <= 16'b0;
ram1_mem[60] <= 16'b0; ram1_mem[61] <= 16'b0; ram1_mem[62] <= 16'b0;
ram1_mem[63] <= 16'b0;

end

always @ (posedge CLK)
if (en_ram1reg == 1'b1 && wr_cntr[6] == 1'b0)
ram1_mem[wr_cntr[5:0]] <= z_out;


always @ (posedge CLK)
if (en_ram1reg == 1'b1 && wr_cntr[6] == 1'b1)
ram2_mem[wr_cntr[5:0]] <= z_out;
```

```
always @ (posedge CLK)
begin
if (en_ram1reg == 1'b1 && rd_cntr[6] == 1'b0)
  data_out <= ram2_mem[rd_cntr[5:0]];

else if (en_ram1reg == 1'b1 && rd_cntr[6] == 1'b1)
  data_out <= ram1_mem[rd_cntr[5:0]];
else data_out <= 11'b0;
end


/* END MEMORY SECTION */

/* 2D-DCT implementation same as the 1D-DCT implementation */

/* First dct coeeficient appears at the output of the RAM1 after
15 + 64 clk cycles. So the 2nd DCT operation starts after 79 clk cycles. */

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
      cntr79 <= 7'b0;
      end
   else
      begin
      cntr79 <= cntr79 + 1;
     end
   end

assign en_dct2d = RST ? 1'b0 : (cntr79 == 7'b1001111) ? 1'b1 : en_dct2d;

always @ (posedge CLK or posedge RST)
       begin
         if (RST)
            begin  en_dct2d_reg <= 1'b0; end
         else
            begin  en_dct2d_reg <= en_dct2d ; end
       end

assign data_out_final[10:0] = data_out;

always @ (posedge CLK or posedge RST )
   begin
   if (RST)
      begin
```

```
      xb0_in <= 11'b0; xb1_in <= 11'b0; xb2_in <= 11'b0; xb3_in <= 11'b0;
      xb4_in <= 11'b0; xb5_in <= 11'b0; xb6_in <= 11'b0; xb7_in <= 11'b0;
      end
  else if (en_dct2d_reg == 1'b1)
      begin
      xb0_in <= data_out_final; xb1_in <= xb0_in; xb2_in <= xb1_in; xb3_in <= xb2_in;
      xb4_in <= xb3_in; xb5_in <= xb4_in; xb6_in <= xb5_in; xb7_in <= xb6_in;
      end
  else if (en_dct2d_reg == 1'b0)
      begin
      xb0_in <= 11'b0; xb1_in <= 11'b0; xb2_in <= 11'b0; xb3_in <= 11'b0;
      xb4_in <= 11'b0; xb5_in <= 11'b0; xb6_in <= 11'b0; xb7_in <= 11'b0;
      end
  end

/* register inputs, inputs read in every eighth clk*/

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
      begin
      xb0_reg <= 12'b0; xb1_reg <= 12'b0; xb2_reg <= 12'b0; xb3_reg <= 12'b0;
      xb4_reg <= 12'b0; xb5_reg <= 12'b0; xb6_reg <= 12'b0; xb7_reg <= 12'b0;
      end
  else if (cntr8 == 4'b1000)
      begin
      xb0_reg <= {xb0_in[10],xb0_in}; xb1_reg <= {xb1_in[10],xb1_in};
      xb2_reg <= {xb2_in[10],xb2_in}; xb3_reg <= {xb3_in[10],xb3_in};
      xb4_reg <= {xb4_in[10],xb4_in}; xb5_reg <= {xb5_in[10],xb5_in};
      xb6_reg <= {xb6_in[10],xb6_in}; xb7_reg <= {xb7_in[10],xb7_in};
      end
  end

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
      begin
      toggleB <= 1'b0;
      end
  else
      begin
      toggleB <= ~toggleB;
      end
  end

/* adder / subtractor block */
```

```
always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     add_sub1b <= 12'b0; add_sub2b <= 12'b0; add_sub3b <= 12'b0; add_sub4b <=
12'b0;
     end
  else
     begin
              if (toggleB == 1'b1)
                begin
          add_sub1b <= (xb0_reg + xb7_reg); add_sub2b <= (xb1_reg + xb6_reg);
          add_sub3b <= (xb2_reg + xb5_reg); add_sub4b <= (xb3_reg + xb4_reg);
          end
              else if (toggleB == 1'b0)
                begin
                add_sub1b <= (xb7_reg - xb0_reg); add_sub2b <= (xb6_reg - xb1_reg);
          add_sub3b <= (xb5_reg - xb2_reg); add_sub4b <= (xb4_reg - xb3_reg);
                end
     end
  end

always @ (posedge RST or posedge CLK)
begin
   if (RST)
     begin
       addsub1b_comp <= 11'b0; save_sign1b <= 1'b0;
     end
   else
     begin
     case (add_sub1b[11])
     1'b0: begin
           addsub1b_comp <= add_sub1b; save_sign1b <= 1'b0;
           end
     1'b1: begin
           addsub1b_comp <= (-add_sub1b) ; save_sign1b <= 1'b1;
           end
     endcase
     end
end

always @ (posedge RST or posedge CLK)
begin
   if (RST)
     begin
```

```
        addsub2b_comp <= 11'b0; save_sign2b <= 1'b0;
      end
    else
      begin
      case (add_sub2b[11])
      1'b0: begin
            addsub2b_comp <= add_sub2b; save_sign2b <= 1'b0;
            end
      1'b1: begin
            addsub2b_comp <= (-add_sub2b) ; save_sign2b <= 1'b1;
            end
      endcase
      end
end

always @ (posedge RST or posedge CLK)
begin
   if (RST)
      begin
        addsub3b_comp <= 11'b0; save_sign3b <= 1'b0;
      end
   else
      begin
      case (add_sub3b[11])
      1'b0: begin
            addsub3b_comp <= add_sub3b; save_sign3b <= 1'b0;
            end
      1'b1: begin
            addsub3b_comp <= (-add_sub3b) ; save_sign3b <= 1'b1;
            end
      endcase
      end
end

always @ (posedge RST or posedge CLK)
begin
   if (RST)
      begin
        addsub4b_comp <= 11'b0; save_sign4b <= 1'b0;
      end
   else
      begin
      case (add_sub4b[11])
      1'b0: begin
            addsub4b_comp <= add_sub4b; save_sign4b <= 1'b0;
            end
```

```
      1'b1: begin
            addsub4b_comp <= (-add_sub4b) ; save_sign4b <= 1'b1;
            end
      endcase
      end
end

    assign p1b_all = addsub1b_comp * memory1a[6:0];
    assign p2b_all = addsub2b_comp * memory2a[6:0];
    assign p3b_all = addsub3b_comp * memory3a[6:0];
    assign p4b_all = addsub4b_comp * memory4a[6:0];

always @ (posedge RST or posedge CLK)
  begin
    if (RST)
      begin
        p1b <= 20'b0; p2b <= 20'b0; p3b <= 20'b0; p4b <= 20'b0;
      end
    else if (i_wait == 2'b00)
      begin


       p1b <= (save_sign1b ^ memory1a[7]) ? (-p1b_all[17:0]) :(p1b_all[17:0]);
       p2b <= (save_sign2b ^ memory2a[7]) ? (-p2b_all[17:0]) :(p2b_all[17:0]);
       p3b <= (save_sign3b ^ memory3a[7]) ? (-p3b_all[17:0]) :(p3b_all[17:0]);
       p4b <= (save_sign4b ^ memory4a[7]) ? (-p4b_all[17:0]) :(p4b_all[17:0]);
       end
  end

/* multiply the outputs of the add/sub block with the 8 sets of stored coefficients */

/* Final adder. Adding the ouputs of the 4 multipliers */

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
      dct2d_int1 <= 20'b0; dct2d_int2 <= 20'b0; dct_2d_int <= 20'b0;
      end
   else
      begin
      dct2d_int1 <= (p1b + p2b);
      dct2d_int2 <= (p3b + p4b);
      dct_2d_int <= (dct2d_int1 + dct2d_int2);
      end
   end
```

```
assign dct_2d_rnd = dct_2d_int[19:8];
assign dct_2d = dct_2d_int[7] ? (dct_2d_rnd + 1'b1) : dct_2d_rnd;

/* The first 1D-DCT output becomes valid after 14 +64 clk cycles. For the first
2D-DCT output to be valid it takes 78 + 1clk to write into the ram + 1clk to
write out of the ram + 8 clks to shift in the 1D-DCT values + 1clk to register
the 1D-DCT values + 1clk to add/sub + 1clk to take compliment + 1 clk for
multiplying +  2clks to add product. So the 2D-DCT output will be valid
at the 94th clk. rdy_out goes high at 93rd clk so that the first data is valid
for the next block*/

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     cntr92 <= 8'b0;
     end
  else if (cntr92 < 8'b1011110)
     begin
     cntr92 <= cntr92 + 1;
     end
  else
     begin
     cntr92 <= cntr92;
     end
  end

assign rdy_out = (cntr92 == 8'b1011110) ? 1'b1 : 1'b0;

endmodule
```

```
//IDCT module from Xilinx applications notes
module idct ( CLK, RST, rdy_in, dct_2d,idct_2d);
output [7:0] idct_2d;
input CLK, RST,rdy_in;
input[11:0] dct_2d;
wire [7:0] idct_2d;

/* constants */
reg[7:0] memory1a, memory2a, memory3a, memory4a;
reg[7:0] memory5a, memory6a, memory7a, memory8a;

/* 1D section */
/* The max value of a pixel after processing (to make their expected mean to zero)
is 2047. If all the values in a row are 2047, the max value of the product terms
would be (127*2)*23170 and that of z_out_int would be (2047*8)*23170=235,407,20
which
is a 25 bit binary. This value divided by 2raised to 16
is equivalent to ignoring the 16 lsb bits of the value */
reg[11:0] xa0_in, xa1_in, xa2_in, xa3_in, xa4_in, xa5_in, xa6_in, xa7_in;
reg[11:0] xa0_reg, xa1_reg, xa2_reg, xa3_reg, xa4_reg, xa5_reg, xa6_reg, xa7_reg;
reg[10:0] xa0_reg_comp, xa1_reg_comp, xa2_reg_comp, xa3_reg_comp,
        xa4_reg_comp, xa5_reg_comp, xa6_reg_comp, xa7_reg_comp;
reg xa0_reg_sign, xa1_reg_sign, xa2_reg_sign, xa3_reg_sign,
          xa4_reg_sign, xa5_reg_sign, xa6_reg_sign, xa7_reg_sign;
reg[21:0] p1a,p2a,p3a,p4a,p5a,p6a,p7a,p8a;
wire[35:0] p1a_all,p2a_all,p3a_all,p4a_all,p5a_all,p6a_all,p7a_all,p8a_all;
reg[21:0] z_out_int1, z_out_int2, z_out_int3;
reg[21:0] z_out_int4;
reg[21:0] z_out_int;
wire[10:0] z_out_rnd;
wire[10:0] z_out;
reg[2:0] indexi_val;

/* clks and counters */
reg[3:0] cntr11 ;
reg[3:0] cntr8, prod_en1;
reg[6:0] cntr80;
reg[6:0] wr_cntr,rd_cntr;
reg[10:0] ram1_mem[63:0],ram2_mem[63:0]; // add the following to infer block RAM in
synlpicity
                                //    synthesis syn_ramstyle = "block_ram"

/* memory section */
reg[10:0] data_out;
reg[10:0] data_out_pipe1;
wire en_ram1,en_dct2d;
```

```
reg en_ram1reg,en_dct2d_reg;

/* 2D section */
wire[10:0] data_out_final;
reg[10:0] xb0_in, xb1_in, xb2_in, xb3_in, xb4_in, xb5_in, xb6_in, xb7_in;
reg[10:0] xb0_reg, xb1_reg, xb2_reg, xb3_reg, xb4_reg, xb5_reg, xb6_reg, xb7_reg;
reg[9:0] xb0_reg_comp, xb1_reg_comp, xb2_reg_comp, xb3_reg_comp,
        xb4_reg_comp, xb5_reg_comp, xb6_reg_comp, xb7_reg_comp;
reg xb0_reg_sign, xb1_reg_sign, xb2_reg_sign, xb3_reg_sign,
    xb4_reg_sign, xb5_reg_sign, xb6_reg_sign, xb7_reg_sign;

reg[15:0] p1b,p2b,p3b,p4b,p5b,p6b,p7b,p8b;
wire[35:0] p1b_all,p2b_all,p3b_all,p4b_all,p5b_all,p6b_all,p7b_all,p8b_all;
reg[19:0] idct_2d_int1,idct_2d_int2,idct_2d_int3,idct_2d_int4;
reg[19:0] idct_2d_int;
//wire[7:0] idct_2d_rnd;

/*  1D-DCT BEGIN */

// store  1D-DCT constant coeeficient values for multipliers */

always @ (posedge RST or posedge CLK)
  begin
  if (RST)
     begin
     memory1a <= 8'd0; memory2a <= 8'd0; memory3a <= 8'd0; memory4a <= 8'd0;
     memory5a <= 8'd0; memory6a <= 8'd0; memory7a <= 8'd0; memory8a <= 8'd0;
     end
  else
     begin
            case (indexi_val)
     3'b000 : begin memory1a <= 8'd91; memory2a <= 8'd126;
                memory3a <= 8'd118; memory4a <= 8'd106;
                memory5a <= 8'd91; memory6a <= 8'd71;
                memory7a <= 8'd49; memory8a <= 8'd25;end

     3'b001 : begin memory1a <= 8'd91; memory2a <= 8'd106;
                memory3a <= 8'd49;
                memory4a[7] <= 1'd1; memory4a[6:0] <= 7'd25;
                memory5a[7] <= 1'd1;memory5a[6:0] <= 7'd91;
                memory6a[7] <= 1'd1;memory6a[6:0] <= 7'd126;
                memory7a[7] <= 1'd1;memory7a[6:0] <= 7'd118;
                memory8a[7] <= 1'd1;memory8a[6:0] <= 7'd71;end

     3'b010 : begin memory1a <= 8'd91; memory2a <= 8'd71;
                memory3a[7] <= 1'd1;memory3a[6:0] <= 7'd49;
```

```
            memory4a[7] <= 1'd1;memory4a[6:0] <= 7'd126;
            memory5a[7] <= 1'd1;memory5a[6:0] <= 7'd91;
            memory6a <= 8'd25;
            memory7a <= 8'd118; memory8a <= 8'd106;end

3'b011 : begin memory1a <= 8'd91; memory2a <= 8'd25;
            memory3a[7] <= 1'd1;memory3a[6:0] <= 7'd118;
            memory4a[7] <= 1'd1;memory4a[6:0] <= 7'd71;
            memory5a <= 8'd91;
            memory6a <= 8'd106;
            memory7a[7] <= 1'd1;memory7a[6:0] <= 7'd49;
            memory8a[7] <= 1'd1;memory8a[6:0] <= 7'd126;end

3'b111 : begin memory1a <= 8'd91;
            memory2a[7] <= 1'd1;memory2a[6:0] <= 7'd126;
            memory3a <= 8'd118;
            memory4a[7] <= 1'd1;memory4a[6:0] <= 7'd106;
            memory5a <= 8'd91;
            memory6a[7] <= 1'd1;memory6a[6:0] <= 7'd71;
            memory7a <= 8'd49;
            memory8a[7] <= 1'd1;memory8a[6:0] <= 7'd25;end

3'b110 : begin memory1a <= 8'd91;
            memory2a[7] <= 1'd1;memory2a[6:0] <= 7'd106;
            memory3a <= 8'd49;
            memory4a <= 8'd25;
            memory5a[7] <= 1'd1;memory5a[6:0] <= 7'd91;
            memory6a <= 8'd126;
            memory7a[7] <= 1'd1;memory7a[6:0] <= 7'd118;
            memory8a <= 8'd71;end

3'b101 : begin memory1a <= 8'd91;
            memory2a[7] <= 1'd1;memory2a[6:0] <= 7'd71;
            memory3a[7] <= 1'd1;memory3a[6:0] <= 7'd49;
            memory4a <= 8'd126;
            memory5a[7] <= 1'd1;memory5a[6:0] <= 7'd91;
            memory6a[7] <= 1'd1;memory6a[6:0] <= 7'd25;
            memory7a <= 8'd118;
            memory8a[7] <= 1'd1;memory8a[6:0] <= 7'd106;end

3'b100 : begin memory1a <= 8'd91;
            memory2a[7] <= 1'd1;memory2a[6:0] <= 7'd25;
            memory3a[7] <= 1'd1;memory3a[6:0] <= 7'd118;
            memory4a <= 8'd71;
            memory5a <= 8'd91;
            memory6a[7] <= 1'd1;memory6a[6:0] <= 7'd106;
```

```
                memory7a[7] <= 1'd1;memory7a[6:0] <= 7'd49;
                memory8a <= 8'd126;end

      endcase
     end
end


/* 8-bit input shifted 8 times thru a shift register*/
always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     xa0_in <= 12'b0; xa1_in <= 12'b0; xa2_in <= 12'b0; xa3_in <= 12'b0;
     xa4_in <= 12'b0; xa5_in <= 12'b0; xa6_in <= 12'b0; xa7_in <= 12'b0;
     end
  else if (rdy_in == 1'b1)
     begin
     xa0_in <= dct_2d; xa1_in <= xa0_in; xa2_in <= xa1_in; xa3_in <= xa2_in;
     xa4_in <= xa3_in; xa5_in <= xa4_in; xa6_in <= xa5_in; xa7_in <= xa6_in;
     end
  else
     begin
     xa0_in <= 12'b0; xa1_in <= 12'b0; xa2_in <= 12'b0; xa3_in <= 12'b0;
     xa4_in <= 12'b0; xa5_in <= 12'b0; xa6_in <= 12'b0; xa7_in <= 12'b0;
     end
  end

/* shifted inputs registered every 8th clk (using cntr8)*/

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     cntr8 <= 4'b0;
     end
  else if(rdy_in == 1'b1)
     begin
     if (cntr8 < 4'b1000)
        begin
        cntr8 <= cntr8 + 1;
        end
     else
        begin
        cntr8 <= 4'b001;
        end
```

```
      end
   else  cntr8 <= 4'b0;

   end

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
      prod_en1 <= 4'b0000;
      end
   else if(rdy_in == 1'b1)
      begin
         if (prod_en1 < 4'b1001)
         begin
         prod_en1 <= prod_en1 + 1;
         end
      else
         begin
         prod_en1 <= 4'b1001;
         end
      end
   end

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
      xa0_reg <= 12'b0; xa1_reg <= 12'b0; xa2_reg <= 12'b0; xa3_reg <= 12'b0;
      xa4_reg <= 12'b0; xa5_reg <= 12'b0; xa6_reg <= 12'b0; xa7_reg <= 12'b0;
      end
   else if (cntr8 == 4'b1000)
      begin
      xa0_reg <= xa0_in; xa1_reg <= xa1_in; xa2_reg <= xa2_in; xa3_reg <= xa3_in;
      xa4_reg <= xa4_in; xa5_reg <= xa5_in; xa6_reg <= xa6_in; xa7_reg <= xa7_in;
      end
   else
      begin
      end
   end
/* take absolute value of signals */

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
```

```
      xa0_reg_comp <= 11'b0; xa1_reg_comp <= 11'b0; xa2_reg_comp <= 11'b0;
xa3_reg_comp <= 11'b0;
      xa4_reg_comp <= 11'b0; xa5_reg_comp <= 11'b0; xa6_reg_comp <= 11'b0;
xa7_reg_comp <= 11'b0;
      xa0_reg_sign <= 1'b0; xa1_reg_sign <= 1'b0; xa2_reg_sign <= 1'b0; xa3_reg_sign
<= 1'b0;
      xa4_reg_sign <= 1'b0; xa5_reg_sign <= 1'b0; xa6_reg_sign <= 1'b0; xa7_reg_sign
<= 1'b0;
      end
  else
      begin
      xa0_reg_sign <= xa0_reg[11];
      xa0_reg_comp[10:0] <= (xa0_reg[11]) ? (-xa0_reg) : xa0_reg[10:0];
      xa1_reg_sign <= xa1_reg[11];
      xa1_reg_comp[10:0] <= (xa1_reg[11]) ? (-xa1_reg) : xa1_reg[10:0];
      xa2_reg_sign <= xa2_reg[11];
      xa2_reg_comp[10:0] <= (xa2_reg[11]) ? (-xa2_reg) : xa2_reg[10:0];
      xa3_reg_sign <= xa3_reg[11];
      xa3_reg_comp[10:0] <= (xa3_reg[11]) ? (-xa3_reg) : xa3_reg[10:0];
      xa4_reg_sign <= xa4_reg[11];
      xa4_reg_comp[10:0] <= (xa4_reg[11]) ? (-xa4_reg) : xa4_reg[10:0];
      xa5_reg_sign <= xa5_reg[11];
      xa5_reg_comp[10:0] <= (xa5_reg[11]) ? (-xa5_reg) : xa5_reg[10:0];
      xa6_reg_sign <= xa6_reg[11];
      xa6_reg_comp[10:0] <= (xa6_reg[11]) ? (-xa6_reg) : xa6_reg[10:0];
      xa7_reg_sign <= xa7_reg[11];
      xa7_reg_comp[10:0] <= (xa7_reg[11]) ? (-xa7_reg) : xa7_reg[10:0];
      end
  end


/* multiply the outputs of the add/sub block with the 8 sets of stored coefficients */
/* The inputs are shifted thru 8 registers in 8 clk cycles. The ouput of the shift
registers are registered at the 9th clk. The values are then added or subtracted at the 10th
clk. The first mutiplier output is obtained at the 11th clk. Memoryx[0] shd be accessed
at the 11th clk*/

/*wait state counter */
// First valid add_sub appears at the 10th clk (8 clks for shifting inputs,
// 9th clk for registering shifted input and 10th clk for add_sub
// to synchronize the i value to the add_sub value, i value is incremented
// only after 10 clks using i_wait
/* max value for p1a = 2047*126. = 18 bits */

    assign p1a_all = xa7_reg_comp[10:0] * memory1a[6:0];/*11bits * 7bits = 18bits */
    assign p2a_all = xa6_reg_comp[10:0] * memory2a[6:0];
```

```
    assign p3a_all = xa5_reg_comp[10:0] * memory3a[6:0];
    assign p4a_all = xa4_reg_comp[10:0] * memory4a[6:0];
    assign p5a_all = xa3_reg_comp[10:0] * memory5a[6:0];
    assign p6a_all = xa2_reg_comp[10:0] * memory6a[6:0];
    assign p7a_all = xa1_reg_comp[10:0] * memory7a[6:0];
    assign p8a_all = xa0_reg_comp[10:0] * memory8a[6:0];

always @ (posedge RST or posedge CLK)
  begin
    if (RST)
      begin
        p1a <= 21'b0; p2a <= 21'b0; p3a <= 21'b0; p4a <= 21'b0;
        p5a <= 21'b0; p6a <= 21'b0; p7a <= 21'b0; p8a <= 21'b0;
        indexi_val <= 3'b000;
      end
    else if (rdy_in == 1'b1 && prod_en1 == 4'b1001)
        begin
        p1a <= (xa7_reg_sign ^ memory1a[7])?(-p1a_all[17:0]):(p1a_all[17:0]);
        p2a <= (xa6_reg_sign ^ memory2a[7])?(-p2a_all[17:0]):(p2a_all[17:0]);
        p3a <= (xa5_reg_sign ^ memory3a[7])?(-p3a_all[17:0]):(p3a_all[17:0]);
        p4a <= (xa4_reg_sign ^ memory4a[7])?(-p4a_all[17:0]):(p4a_all[17:0]);
        p5a <= (xa3_reg_sign ^ memory5a[7])?(-p5a_all[17:0]):(p5a_all[17:0]);
        p6a <= (xa2_reg_sign ^ memory6a[7])?(-p6a_all[17:0]):(p6a_all[17:0]);
        p7a <= (xa1_reg_sign ^ memory7a[7])?(-p7a_all[17:0]):(p7a_all[17:0]);
        p8a <= (xa0_reg_sign ^ memory8a[7])?(-p8a_all[17:0]):(p8a_all[17:0]);
        if (indexi_val == 3'b111)
          indexi_val <= 3'b000;
        else
          indexi_val <= indexi_val + 1'b1;
        end
   else
        begin
        p1a <= 21'b0; p2a <= 21'b0; p3a <= 21'b0; p4a <= 21'b0;
        p5a <= 21'b0; p6a <= 21'b0; p7a <= 21'b0; p8a <= 21'b0;
        end
  end


/* Final adder. Adding the ouputs of the 4 multipliers */
/* max value for z_out_int = 2047*126*8 = 2063376 = 21 bits */
always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
      z_out_int1 <= 21'b0; z_out_int2 <= 21'b0; z_out_int3 <= 21'b0;
      z_out_int4 <= 21'b0; z_out_int <= 21'b0;
```

```
      end
   else
      begin
      z_out_int1 <= (p1a + p2a);
      z_out_int2 <= (p3a + p4a);
      z_out_int3 <= (p5a + p6a);
      z_out_int4 <= (p7a + p8a);
      z_out_int  <= (z_out_int1 + z_out_int2 + z_out_int3 + z_out_int4);
      end
   end

// rounding of the value
/* max value for a 1D-DCT output is "11111111"*126*8/256=1004.
To represent this we need only 10 bits, plus 1 bit for sign */

assign z_out_rnd = z_out_int[18:8];
assign z_out = z_out_int[7] ? (z_out_rnd + 1'b1) : z_out_rnd;

/* 1D-DCT END */

/* tranpose memory to store intermediate Z coefficients */
/* store the 64 coefficients in the first 64 locations of the RAM */
/* first valid final (product) adder ouput is at the 13th clk. 8clk SR
+ 1 clk reg + 1 clk comp + 1 clk prod. + 2 clks summing.
So the RAM is enabled at the 11th clk) */

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
      cntr11 <= 4'b0;
      end
   else if (rdy_in == 1'b1)
      begin
      cntr11 <= cntr11 + 1;
      end
   end

/* enable RAM at the 14th clk after RST goes inactive */

assign en_ram1 = RST ? 1'b0 : (cntr11== 4'b1100) ? 1'b1 : en_ram1;

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
```

```
            en_ram1reg <= 1'b0;
      end
   else
      begin
      en_ram1reg <= en_ram1 ;
      end
   end
```

/* After the RAM is enabled, data is written into the RAM1 for 64 clk cycles. Data is written in into
each consecutive location . After 64 locations are written into, RAM1 goes into read mode and RAM2 goes into
write mode. The cycle then repeats.
For either RAM, data is written into each consecutive location. However , data is read in a different order. If data
is assumed to be written in each row at a time, in an 8x8 matrix, data is read each column at a time. ie., after
the first data is read out, every eight data is read out . Then the 2nd data is read out followed be every 8th.

the write is as follows:
1w(ram_locn1) 2w(ram_locn2) 3w(ram_locn3) 4w(ram_locn4) 5w(ram_locn5)
6w(ram_locn6) 7w(ram_locn7) 8w(ram_locn8)
9w(ram_locn9) 10w(ram_locn10) 11w(ram_locn11) 12w(ram_locn12) 13w(ram_locn13)
14w(ram_locn14) 15w(ram_locn15) 16w(ram_locn16)
.....
57w(ram_locn57) 58w(ram_locn58) 59w(ram_locn59) 60w(ram_locn60)
61w(ram_locn61) 62w(ram_locn62) 63w(ram_locn63) 64w(ram_locn64)

the read is as follows:
1r(ram_locn1)  9r(ram_locn2) . . . 57r(ram_locn8)
2r(ram_locn9) 10r(ram_locn10) . . . 58r(ram_locn16)
3r(ram_locn17) 11r(ram_locn18) . . . 59r(ram_locn24)
4r(ram_locn25) 12r(ram_locn26) . . . 60r(ram_locn32)
5r(ram_locn33) 13r(ram_locn34) . . . 61r(ram_locn40)
6r(ram_locn41) 14r(ram_locn42) . . . 62r(ram_locn48)
7r(ram_locn49) 15r(ram_locn50) . . . 63r(ram_locn56)
8r(ram_locn57) 16r(ram_locn58) . . . 64r(ram_locn64)

where "xw" is the xth write and "ram_locnx" is the xth ram location and "xr" is the xth read. Reading
is advanced by the read counter rd_cntr, nd writing by the write counter wr_cntr. */

```
always @ (posedge CLK or posedge RST)
   begin
   if (RST)
```

```
      begin
            rd_cntr[5:3] <= 3'b111;
      end
   else
      begin
            if (en_ram1reg == 1'b1)
               rd_cntr[5:3] <= rd_cntr[5:3] + 1;
            end
   end

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
            rd_cntr[2:0] <= 3'b111;
      end
   else
      begin
            if (en_ram1reg == 1'b1 && rd_cntr[5:3] == 3'b111)
               rd_cntr[2:0] <= rd_cntr[2:0] + 1;
      end
   end

always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
            rd_cntr[6] <= 1'b1;
      end
   else
      begin
      if (en_ram1reg == 1'b1 && rd_cntr[5:0] == 6'b111111)
        rd_cntr[6] <= ~rd_cntr[6];
      end
   end


always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
      wr_cntr <= 7'b1111111;
      end
   else begin
      if (en_ram1reg == 1'b1)
        wr_cntr <= wr_cntr + 1;
```

```
            else
              wr_cntr <= 7'b0;
        end
    end

initial
begin
ram2_mem[0] <= 16'b0; ram2_mem[1] <= 16'b0; ram2_mem[2] <= 16'b0; ram2_mem[3]
<= 16'b0; ram2_mem[4] <= 16'b0;
ram2_mem[5] <= 16'b0; ram2_mem[6] <= 16'b0; ram2_mem[7] <= 16'b0; ram2_mem[8]
<= 16'b0; ram2_mem[9] <= 16'b0;
ram2_mem[10] <= 16'b0; ram2_mem[11] <= 16'b0; ram2_mem[12] <= 16'b0;
ram2_mem[13] <= 16'b0; ram2_mem[14] <= 16'b0;
ram2_mem[15] <= 16'b0; ram2_mem[16] <= 16'b0; ram2_mem[17] <= 16'b0;
ram2_mem[18] <= 16'b0; ram2_mem[19] <= 16'b0;
ram2_mem[20] <= 16'b0; ram2_mem[21] <= 16'b0; ram2_mem[22] <= 16'b0;
ram2_mem[23] <= 16'b0; ram2_mem[24] <= 16'b0;
ram2_mem[25] <= 16'b0; ram2_mem[26] <= 16'b0; ram2_mem[27] <= 16'b0;
ram2_mem[28] <= 16'b0; ram2_mem[29] <= 16'b0;
ram2_mem[30] <= 16'b0; ram2_mem[31] <= 16'b0; ram2_mem[32] <= 16'b0;
ram2_mem[33] <= 16'b0; ram2_mem[34] <= 16'b0;
ram2_mem[35] <= 16'b0; ram2_mem[36] <= 16'b0; ram2_mem[37] <= 16'b0;
ram2_mem[38] <= 16'b0; ram2_mem[39] <= 16'b0;
ram2_mem[40] <= 16'b0; ram2_mem[41] <= 16'b0; ram2_mem[42] <= 16'b0;
ram2_mem[43] <= 16'b0; ram2_mem[44] <= 16'b0;
ram2_mem[45] <= 16'b0; ram2_mem[46] <= 16'b0; ram2_mem[47] <= 16'b0;
ram2_mem[48] <= 16'b0; ram2_mem[49] <= 16'b0;
ram2_mem[50] <= 16'b0; ram2_mem[51] <= 16'b0; ram2_mem[52] <= 16'b0;
ram2_mem[53] <= 16'b0; ram2_mem[54] <= 16'b0;
ram2_mem[55] <= 16'b0; ram2_mem[56] <= 16'b0; ram2_mem[57] <= 16'b0;
ram2_mem[58] <= 16'b0; ram2_mem[59] <= 16'b0;
ram2_mem[60] <= 16'b0; ram2_mem[61] <= 16'b0; ram2_mem[62] <= 16'b0;
ram2_mem[63] <= 16'b0;

ram1_mem[0] <= 16'b0; ram1_mem[1] <= 16'b0; ram1_mem[2] <= 16'b0; ram1_mem[3]
<= 16'b0; ram1_mem[4] <= 16'b0;
ram1_mem[5] <= 16'b0; ram1_mem[6] <= 16'b0; ram1_mem[7] <= 16'b0; ram1_mem[8]
<= 16'b0; ram1_mem[9] <= 16'b0;
ram1_mem[10] <= 16'b0; ram1_mem[11] <= 16'b0; ram1_mem[12] <= 16'b0;
ram1_mem[13] <= 16'b0; ram1_mem[14] <= 16'b0;
ram1_mem[15] <= 16'b0; ram1_mem[16] <= 16'b0; ram1_mem[17] <= 16'b0;
ram1_mem[18] <= 16'b0; ram1_mem[19] <= 16'b0;
ram1_mem[20] <= 16'b0; ram1_mem[21] <= 16'b0; ram1_mem[22] <= 16'b0;
ram1_mem[23] <= 16'b0; ram1_mem[24] <= 16'b0;
ram1_mem[25] <= 16'b0; ram1_mem[26] <= 16'b0; ram1_mem[27] <= 16'b0;
ram1_mem[28] <= 16'b0; ram1_mem[29] <= 16'b0;
```

```
ram1_mem[30] <= 16'b0; ram1_mem[31] <= 16'b0; ram1_mem[32] <= 16'b0;
ram1_mem[33] <= 16'b0; ram1_mem[34] <= 16'b0;
ram1_mem[35] <= 16'b0; ram1_mem[36] <= 16'b0; ram1_mem[37] <= 16'b0;
ram1_mem[38] <= 16'b0; ram1_mem[39] <= 16'b0;
ram1_mem[40] <= 16'b0; ram1_mem[41] <= 16'b0; ram1_mem[42] <= 16'b0;
ram1_mem[43] <= 16'b0; ram1_mem[44] <= 16'b0;
ram1_mem[45] <= 16'b0; ram1_mem[46] <= 16'b0; ram1_mem[47] <= 16'b0;
ram1_mem[48] <= 16'b0; ram1_mem[49] <= 16'b0;
ram1_mem[50] <= 16'b0; ram1_mem[51] <= 16'b0; ram1_mem[52] <= 16'b0;
ram1_mem[53] <= 16'b0; ram1_mem[54] <= 16'b0;
ram1_mem[55] <= 16'b0; ram1_mem[56] <= 16'b0; ram1_mem[57] <= 16'b0;
ram1_mem[58] <= 16'b0; ram1_mem[59] <= 16'b0;
ram1_mem[60] <= 16'b0; ram1_mem[61] <= 16'b0; ram1_mem[62] <= 16'b0;
ram1_mem[63] <= 16'b0;

end

always @ (posedge CLK)
if (en_ram1reg == 1'b1 && wr_cntr[6] == 1'b0)
ram1_mem[wr_cntr[5:0]] <= z_out;


always @ (posedge CLK)
if (en_ram1reg == 1'b1 && wr_cntr[6] == 1'b1)
ram2_mem[wr_cntr[5:0]] <= z_out;

always @ (posedge CLK)
begin
if (en_ram1reg == 1'b1 && rd_cntr[6] == 1'b0)
  data_out <= ram2_mem[rd_cntr[5:0]];

else if (en_ram1reg == 1'b1 && rd_cntr[6] == 1'b1)
  data_out <= ram1_mem[rd_cntr[5:0]];
else data_out <= 11'b0;
end


/* END MEMORY SECTION */

/* 2D-DCT implementation same as the 1D-DCT implementation */

/* First dct coeeficient appears at the output of the RAM1 after 13clk + 1clk ram in + 64
clk cycles + 1clk ram out. including the 1 pipestage, the 2nd DCT operation starts after
16 +  64 clk cycles. Pipe stages are added to match the timing with the cntr8 counter */

always @ (posedge CLK or posedge RST)
```

```
  begin
  if (RST)
     begin
     data_out_pipe1 <= 11'b0;
     end
  else
     begin
     data_out_pipe1 <= data_out;
     end
  end

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
     begin
     cntr80 <= 7'b0;
     end
  else if (rdy_in ==1'b1)
     begin
     cntr80 <= cntr80 + 1;
    end
  end

assign en_dct2d = RST ? 1'b0 : (cntr80== 7'b1001111) ? 1'b1 : en_dct2d;

always @ (posedge CLK or posedge RST)
       begin
         if (RST)
            begin  en_dct2d_reg <= 1'b0; end
         else
            begin  en_dct2d_reg <= en_dct2d ; end
       end

assign data_out_final[10:0] = data_out_pipe1;

always @ (posedge CLK or posedge RST )
  begin
  if (RST)
     begin
     xb0_in <= 11'b0; xb1_in <= 11'b0; xb2_in <= 11'b0; xb3_in <= 11'b0;
     xb4_in <= 11'b0; xb5_in <= 11'b0; xb6_in <= 11'b0; xb7_in <= 11'b0;
     end
  else if (en_dct2d_reg == 1'b1)
     begin
     xb0_in <= data_out_final; xb1_in <= xb0_in; xb2_in <= xb1_in; xb3_in <= xb2_in;
     xb4_in <= xb3_in; xb5_in <= xb4_in; xb6_in <= xb5_in; xb7_in <= xb6_in;
```

```
      end
  else if (en_dct2d_reg == 1'b0)
      begin
      xb0_in <= 11'b0; xb1_in <= 11'b0; xb2_in <= 11'b0; xb3_in <= 11'b0;
      xb4_in <= 11'b0; xb5_in <= 11'b0; xb6_in <= 11'b0; xb7_in <= 11'b0;
      end
  end

/* register inputs, inputs read in every eighth clk*/

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
      begin
      xb0_reg <= 11'b0; xb1_reg <= 11'b0; xb2_reg <= 11'b0; xb3_reg <= 11'b0;
      xb4_reg <= 11'b0; xb5_reg <= 11'b0; xb6_reg <= 11'b0; xb7_reg <= 11'b0;
      end
  else if (cntr8 == 4'b1000)
      begin
      xb0_reg <= xb0_in; xb1_reg <= xb1_in; xb2_reg <= xb2_in; xb3_reg <= xb3_in;
      xb4_reg <= xb4_in; xb5_reg <= xb5_in; xb6_reg <= xb6_in; xb7_reg <= xb7_in;
      end
  end

/* take absolute value of signals */

always @ (posedge CLK or posedge RST)
  begin
  if (RST)
      begin
      xb0_reg_comp <= 10'b0; xb1_reg_comp <= 10'b0; xb2_reg_comp <= 10'b0;
xb3_reg_comp <= 10'b0;
      xb4_reg_comp <= 10'b0; xb5_reg_comp <= 10'b0; xb6_reg_comp <= 10'b0;
xb7_reg_comp <= 10'b0;
      xb0_reg_sign <= 1'b0; xb1_reg_sign <= 1'b0; xb2_reg_sign <= 1'b0; xb3_reg_sign
<= 1'b0;
      xb4_reg_sign <= 1'b0; xb5_reg_sign <= 1'b0; xb6_reg_sign <= 1'b0; xb7_reg_sign
<= 1'b0;
      end
  else
      begin
      xb0_reg_sign <= xb0_reg[10];
      xb0_reg_comp[9:0] <= (xb0_reg[10]) ? (-xb0_reg) : xb0_reg[9:0];
      xb1_reg_sign <= xb1_reg[10];
      xb1_reg_comp[9:0] <= (xb1_reg[10]) ? (-xb1_reg) : xb1_reg[9:0];
      xb2_reg_sign <= xb2_reg[10];
```

```
      xb2_reg_comp[9:0] <= (xb2_reg[10]) ? (-xb2_reg) : xb2_reg[9:0];
      xb3_reg_sign <= xb3_reg[10];
      xb3_reg_comp[9:0] <= (xb3_reg[10]) ? (-xb3_reg) : xb3_reg[9:0];
      xb4_reg_sign <= xb4_reg[10];
      xb4_reg_comp[9:0] <= (xb4_reg[10]) ? (-xb4_reg) : xb4_reg[9:0];
      xb5_reg_sign <= xb5_reg[10];
      xb5_reg_comp[9:0] <= (xb5_reg[10]) ? (-xb5_reg) : xb5_reg[9:0];
      xb6_reg_sign <= xb6_reg[10];
      xb6_reg_comp[9:0] <= (xb6_reg[10]) ? (-xb6_reg) : xb6_reg[9:0];
      xb7_reg_sign <= xb7_reg[10];
      xb7_reg_comp[9:0] <= (xb7_reg[10]) ? (-xb7_reg) : xb7_reg[9:0];
      end
  end

/* 10 bits * 7 bits = 16 bits */
    assign p1b_all = xb7_reg_comp[9:0] * memory1a[6:0];
    assign p2b_all = xb6_reg_comp[9:0] * memory2a[6:0];
    assign p3b_all = xb5_reg_comp[9:0] * memory3a[6:0];
    assign p4b_all = xb4_reg_comp[9:0] * memory4a[6:0];
    assign p5b_all = xb3_reg_comp[9:0] * memory5a[6:0];
    assign p6b_all = xb2_reg_comp[9:0] * memory6a[6:0];
    assign p7b_all = xb1_reg_comp[9:0] * memory7a[6:0];
    assign p8b_all = xb0_reg_comp[9:0] * memory8a[6:0];


/* multiply the outputs of the add/sub block with the 8 sets of stored coefficients */

always @ (posedge RST or posedge CLK)
  begin
    if (RST)
      begin
        p1b <= 19'b0; p2b <= 19'b0; p3b <= 19'b0; p4b <= 19'b0;
        p5b <= 19'b0; p6b <= 19'b0; p7b <= 19'b0; p8b <= 19'b0;
      end
    else if (rdy_in == 1'b1 && prod_en1 == 4'b1001)
        begin
        p1b <= (xb7_reg_sign ^ memory1a[7])?(-p1b_all[15:0]):(p1b_all[15:0]);
        p2b <= (xb6_reg_sign ^ memory2a[7])?(-p2b_all[15:0]):(p2b_all[15:0]);
        p3b <= (xb5_reg_sign ^ memory3a[7])?(-p3b_all[15:0]):(p3b_all[15:0]);
        p4b <= (xb4_reg_sign ^ memory4a[7])?(-p4b_all[15:0]):(p4b_all[15:0]);
        p5b <= (xb3_reg_sign ^ memory5a[7])?(-p5b_all[15:0]):(p5b_all[15:0]);
        p6b <= (xb2_reg_sign ^ memory6a[7])?(-p6b_all[15:0]):(p6b_all[15:0]);
        p7b <= (xb1_reg_sign ^ memory7a[7])?(-p7b_all[15:0]):(p7b_all[15:0]);
        p8b <= (xb0_reg_sign ^ memory8a[7])?(-p8b_all[15:0]):(p8b_all[15:0]);
        end
    else
```

```
        begin
        p1b <= 19'b0; p2b <= 19'b0; p3b <= 19'b0; p4b <= 19'b0;
        p5b <= 19'b0; p6b <= 19'b0; p7b <= 19'b0; p8b <= 19'b0;
        end

  end


always @ (posedge CLK or posedge RST)
   begin
   if (RST)
      begin
      idct_2d_int1 <= 20'b0; idct_2d_int2 <= 20'b0; idct_2d_int3 <= 20'b0;
      idct_2d_int4 <= 20'b0; idct_2d_int <= 20'b0;
      end
   else
      begin
      idct_2d_int1 <= (p1b + p2b);
      idct_2d_int2 <= (p3b + p4b);
      idct_2d_int3 <= (p5b + p6b);
      idct_2d_int4 <= (p7b + p8b);
      idct_2d_int <= (idct_2d_int1 + idct_2d_int2 + idct_2d_int3 + idct_2d_int4);
      end
   end

/* max value for a input signal to dct is "11111111".
To represent this we need only 8 bits, plus 1 bit for sign */

assign idct_2d = idct_2d_int[15:8];
//assign idct_2d = {idct_2d_int[19],idct_2d_int[14:8]};
//assign idct_2d = idct_2d_int[7] ? (idct_2d_rnd + 1'b1) : idct_2d_rnd;
//assign idct_2d = idct_2d_int[19:8];
Endmodule
```

Sanjay Jhaveri
6.111 Final Project Code

```
//
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT
// memories for video display.  Video input from the NTSC digitizer is
// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is used
// as the video frame buffer, with 8 bits used per pixel (black & white).
//
// Since the ZBT is read once for every four pixels, this frees up time for
// data to be stored to the ZBT during other pixel times.  The NTSC decoder
// runs at 27 MHz, whereas the XGA runs at 65 MHz, so we synchronize
// signals between the two (see ntsc2zbt.v) and let the NTSC data be
// stored to ZBT memory whenever it is available, during cycles when
// pixel reads are not being performed.
//
// We use a very simple ZBT interface, which does not involve any clock
// generation or hiding of the pipelining.  See zbt_6111.v for more info.
//
// switch[7] selects between display of NTSC video and test bars
// switch[6] is used for testing the NTSC decoder
// switch[1] selects between test bar periods; these are stored to ZBT
//           during blanking periods
// switch[0] selects vertical test bars (hardwired; not stored in ZBT)



//////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
```

```
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
//////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
```

```
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
//////////////////////////////////////////////////////////////////////////////

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  keyboard_clock, keyboard_data;
       inout mouse_clock, mouse_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_clk = 1'b0;
  // assign ram1_cen_b = 1'b1;

  assign ram1_ce_b = 1'b0;
  assign ram1_oe_b = 1'b0;
 // assign ram1_we_b = 1'b1;
  assign ram1_bwe_b = 4'h0;
         assign ram1_adv_ld = 1'b0;

  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
 // 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


//create 130 mhz clock for the ram….allows a read and write to happen on the same clock
//cycle.

  wire clock_130mhz_unbuf,clock_130mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_130mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 5
   // 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_130mhz),.I(clock_130mhz_unbuf));
// use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)

         wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk3(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk3 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk3 is 24
   // synthesis attribute CLK_FEEDBACK of vclk3 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk3 is 37
   BUFG vclk4(.O(clock_65mhz),.I(clock_65mhz_unbuf));
         reg parity;
         reg parity1;
         reg parity2;
         wire shootmode;
         wire viewmode;
         wire livemode;
         reg ramtoread;

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

         reg [35:0] vram_write_data1;
   wire [35:0] vram_read_data1;
   reg [18:0] vram_addr1;
   reg       vram_we1;

         reg [35:0] vram_read_data2;

         wire enable;
         wire shoot;
         wire view;
         wire live;
//create parity bits to control when to read and when to write.


         always @ (posedge clock_130mhz) begin
                  parity <= ~parity;
          end

         always @ (posedge clock_130mhz) begin
                  parity2 <= parity;
          end
```

```
        wire clk = clock_130mhz;
//      wire [35:0] vram_read_data_disp = ramtoread ? vram_read_data :
vram_read_data2;
                reg [35:0] vram_read_data_disp;
                always @ (posedge clk)
                        if (~parity) vram_read_data_disp <= ramtoread ? vram_read_data :
vram_read_data2;




  // 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;
  debounce db1(power_on_reset, clk, ~button_enter, user_reset);
  assign reset = user_reset | power_on_reset;

  // display module for debugging


        // generate basic XVGA video signals
  wire [10:0] hcount;
  wire [9:0]  vcount;
  wire hsync,vsync,blank;
  xvga xvga1(parity, clk,hcount,vcount,hsync,vsync,blank);
        //wire [11:0] mx, my;

//writing to both on board srams

        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);

        zbt_6111 zbt2(clk, 1'b1, vram_we1, vram_addr1,
                vram_write_data1, vram_read_data1,
                ram1_clk, ram1_we_b, ram1_address, ram1_data, ram1_cen_b);

         // wire [63:0] dispdata = {mx,my,1'b0,btn_click,36'b0};
  display_16hex d1(reset, clk, 63'b0,
                   disp_blank, disp_clock, disp_rs, disp_ce_b,
```

```
                        disp_reset_b, disp_data_out);

  // generate pixel value from reading ZBT memory
  wire [29:0]   vr_pixel;
        wire [29:0] ycrcb;      // video data (luminance, chrominance)
        wire [29:0] huhs;
//this function sets the user inputs
        capture_input ci1(clk, ~button_enter, shoot, view, live, shootmode, viewmode,
livemode);


        assign led = {~livemode, ~viewmode, ~shootmode, 2'b11, ~live, ~view, ~shoot};
  wire [29:0]   pixel;



 vram_display vd1(shootmode, parity, reset, clk, hcount, vcount, pixel,
vram_read_data_disp);

        // 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 [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));



        Divider div(shootmode, clk, reset, enable);
  // code to write NTSC data to video memory


        wire [18:0] ntsc_addr;
  wire [35:0] ntsc_data;
  wire        ntsc_we;

  wire [9:0] row;
```

```
        wire [9:0] col;



        wire [29:0] huhsdata;
        wire [18:0] shot_addr;
        wire [35:0] shot_read_data;
        wire [9:0] vcountshift = vcount + 21;

         ntsc_to_zbt n2z(parity, clk, tv_in_line_clock1, fvh, dv, ycrcb,
                    ntsc_addr, ntsc_data, ntsc_we, switch[6], shootmode);


        write_to_huhs wth(shootmode, clock_27mhz, shotaddr, shot_read_data,
huhsdata);

//this always statement controls state machine according to user inputs
 always @ (posedge clk) begin
                    vram_addr <= (parity) ? ntsc_addr : {vcount[8:0], hcount[9:0]};
                     vram_write_data <= ntsc_data;

                     vram_addr1 <= vram_addr;
                     vram_write_data1 <= vram_write_data;
                     vram_read_data2 <=  vram_read_data1;

                    if(~button_enter) begin
                              vram_we1 <= parity2;
                              vram_we <= (parity);
                              ramtoread <=1;
                     end
                    else if (shootmode)     begin
                     if (ramtoread) begin
                              vram_we1 <= parity2;
                              vram_we <= 0;
                              ramtoread <=1;
                      end
                      else begin
                              vram_we1 <= 0;
                              vram_we <= parity;
                              ramtoread <=0;
                      end
                    end
                    else if (livemode) begin

                              if (ramtoread) begin
```

```
                        vram_we1 <= parity2;
                        vram_we <= 0;
                        ramtoread <=0;
                    end
                    else begin
                        vram_we1 <= 0;
                        vram_we <= parity;
                        ramtoread <=1;
                    end
                end
                else if (viewmode) begin
                    if (ramtoread) begin
                        vram_we1 <= 0;
                        vram_we <= parity;
                        ramtoread <=0;
                    end
                    else begin
                        vram_we1 <= parity2;
                        vram_we <= 0;
                        ramtoread <=1;
                    end
                end
                else begin
                        vram_we1 <= parity2;
                        vram_we <= (parity);
                        ramtoread <=1;
                        end
        end



        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);

        reg [29:0] gpixel;
        reg [29:0] gpixel1;
        wire [29:0] bpixel;

        Background BG(clk, hcount, vcount, bpixel, mouse_clock, mouse_data, shoot,
view, live);

        wire [2:0] pixelShoot;
        wire [29:0] pixela;
```

```
	reg [29:0] pixela1;
	reg [29:0] pixela2;
	parameter Shootword = {8'd83, 8'd104, 8'd111, 8'd111, 8'd116, 8'd33};
	char_string_display cd(clock_65mhz, hcount, vcount, pixelShoot, Shootword,
10'd825, 9'd40);
		defparam cd.NCHAR = 6;

	assign pixela =  {pixelShoot, 3'b000, 3'b000, pixelShoot, pixelShoot, pixelShoot,
pixelShoot, pixelShoot, 3'b000, 3'b000};



	wire [2:0] pixelView;
	wire [29:0] pixelb;
	reg [29:0] pixelb1;
	reg [29:0] pixelb2;
	parameter Viewword = {8'd86, 8'd105, 8'd101, 8'd119};
	char_string_display cd2(clock_65mhz, hcount, vcount, pixelView, Viewword,
10'd838, 9'd130);
		defparam cd2.NCHAR = 4;

	assign pixelb =  {pixelView, 3'b000, 3'b000, pixelView, pixelView, pixelView,
pixelView, pixelView, 3'b000, 3'b000};

	wire [2:0] pixelLive;
	wire [29:0] pixelc;
	reg [29:0] pixelc1;
	reg [29:0] pixelc2;
	parameter Liveword = {8'd76, 8'd105, 8'd118, 8'd101};
	char_string_display cd3(clock_65mhz, hcount, vcount, pixelLive, Liveword,
10'd838, 9'd220);
		defparam cd3.NCHAR = 4;

	assign pixelc =  {pixelLive, 3'b000, 3'b000, pixelLive, pixelLive, pixelLive,
pixelLive, pixelLive, 3'b000, 3'b000};


//latch all pixel data

	always @(posedge clk) begin
			pixela2 <= pixela;
			pixelb2 <= pixelb;
			pixelc2 <= pixelc;
			pixela1 <= pixela2;
			pixelb1 <= pixelb2;
			pixelc1 <= pixelc2;
```

```
                if (~parity) begin
                        gpixel <= (pixel | bpixel | pixela1 | pixelb1 | pixelc1);
                end
        end


        wire [7:0] R;
        wire [7:0] G;
        wire [7:0] B;

        YCrCb2RGB conv(R, G, B, clk, ~button_enter, gpixel[29:20], gpixel[19:10],
gpixel[9:0]);
  // VGA Output.  In order to meet the setup and hold times of the
  // AD7125, we send it ~clock_65mhz.

        assign vga_out_red =  R ;
  assign vga_out_green = G;
        assign vga_out_blue =  B;
  assign vga_out_sync_b = 1'b1;    // not used
  assign vga_out_pixel_clock = ~clk;
  assign vga_out_blank_b = ~b;
  assign vga_out_hsync = hs;
  assign vga_out_vsync = vs;

  // debugging



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(parity, clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw,
shoot);
input shoot;
input parity;
input clk; // system clock
input vclk; // video clock from camera
input [2:0] fvh;
input dv;
input [29:0] din;
output reg [18:0] ntsc_addr;
output reg [35:0] ntsc_data;
output  ntsc_we; // write enable for NTSC data
input sw; // switch which determines mode (for debugging)
parameter COL_START = 10'd0;
parameter ROW_START = 10'd0;
// 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 [29:0] vdata = 0;
//reg [18:0] myaddr;
reg vwe;
reg old_dv;
reg even_odd; // decode interlaced frame to this wire
wire frame = fvh[2];
reg [18:0] counter;
always @ (posedge vclk) begin
        old_dv <= dv;
        vwe <= dv && ~old_dv; // if data valid, write it
        even_odd <= fvh[2];
        row <= (fvh[1] && !fvh[2]) ? ROW_START :
                                (!fvh[1] && fvh[0] && (row < 255)) ? row + 1 : row;

        col <= fvh[0] ? COL_START :
                        (!fvh[1] && dv && (col < 721)) ? col + 1 : col;

        vdata <= dv ? din  : vdata;
end

        // synchronize with system clock
        reg [9:0] x[1:0];
        reg [9:0] y[1:0];
        reg [29:0] 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

reg old_we;
wire we_edge = we[1] & ~old_we;
always @(posedge clk) begin
        old_we <= we[1];
        end
reg [35:0] mydata;
always @(posedge clk)
        if (we_edge)
                mydata <= {6'b000000, data[1]};


wire ntsc_we = we_edge;
always @(posedge clk) begin
          if (ntsc_we)   begin
                ntsc_data <= mydata;
        ntsc_addr <=  {y[1][7:0], eo[1], x[1][9:0]};
                end
end


endmodule // ntsc_to_zbt

/*************************************************************************
***
 **
 ** Module: ycrcb2rgb
 **
 ** Generic Equations:

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

module YCrCb2RGB( R, G, B, clk, rst, Y, Cr, Cb );

output [7:0]  R, G, B;

input clk,rst;
input[9:0] Y, Cr, Cb;
```

```
wire [7:0] R,G,B;
reg [20:0] R_int,G_int,B_int,X_int,A_int,B1_int,B2_int,C_int;
reg [9:0] const1,const2,const3,const4,const5;
reg[9:0] Y_reg, Cr_reg, Cb_reg;

//registering constants
always @ (posedge clk)
begin
 const1 = 10'b 0100101010; //1.164 = 01.00101010
 const2 = 10'b 0110011000; //1.596 = 01.10011000
 const3 = 10'b 0011010000; //0.813 = 00.11010000
 const4 = 10'b 0001100100; //0.392 = 00.01100100
 const5 = 10'b 1000000100; //2.017 = 10.00000100
end

always @ (posedge clk or posedge rst)
  if (rst)
    begin
    Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
    end
  else
    begin
          Y_reg <= Y; Cr_reg <= Cr; Cb_reg <= Cb;
    end

always @ (posedge clk or posedge rst)
  if (rst)
    begin
     A_int <= 0; B1_int <= 0; B2_int <= 0; C_int <= 0; X_int <= 0;
    end
  else
    begin
    X_int <= (const1 * (Y_reg - 'd64)) ;
    A_int <= (const2 * (Cr_reg - 'd512));
    B1_int <= (const3 * (Cr_reg - 'd512));
    B2_int <= (const4 * (Cb_reg - 'd512));
    C_int <= (const5 * (Cb_reg - 'd512));
    end

always @ (posedge clk or posedge rst)
  if (rst)
    begin
     R_int <= 0; G_int <= 0; B_int <= 0;
    end
  else
```

```
    begin
    R_int <= X_int + A_int;
    G_int <= X_int - B1_int - B2_int;
    B_int <= X_int + C_int;
    end



/*always @ (posedge clk or posedge rst)
  if (rst)
    begin
     R_int <= 0; G_int <= 0; B_int <= 0;
    end
  else
    begin
    X_int <= (const1 * (Y_reg - 'd64)) ;
    R_int <= X_int + (const2 * (Cr_reg - 'd512));
    G_int <= X_int - (const3 * (Cr_reg - 'd512)) - (const4 * (Cb_reg - 'd512));
    B_int <= X_int + (const5 * (Cb_reg - 'd512));
    end

*/
/* limit output to 0 - 4095, <0 equals o and >4095 equals 4095 */
assign R = (R_int[20]) ? 0 : (R_int[19:18] == 2'b0) ? R_int[17:10] : 8'b11111111;
assign G = (G_int[20]) ? 0 : (G_int[19:18] == 2'b0) ? G_int[17:10] : 8'b11111111;
assign B = (B_int[20]) ? 0 : (B_int[19:18] == 2'b0) ? B_int[17:10] : 8'b11111111;

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    19:50:04 04/24/06
// Design Name:
// Module Name:    xvga
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
```

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

module xvga(parity, vclock,hcount,vcount,hsync,vsync,blank);
  input vclock, parity;
  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
        if (parity) 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
          end
endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:20:28 04/05/06
// Design Name:
// Module Name:    Background
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module Background(clk130, x, y, YCRCB, mouse_clock, mouse_data, capture, view,
live);
    input [10:0] x;
    input [9:0] y;
            inout mouse_clock;
            inout mouse_data;
            input clk130;
           output  capture;
           output view;
           output live;
           output [29:0] YCRCB;
           wire [29:0] YCRCB;
           wire [29:0] YCRCB1;
           wire [29:0] YCRCB2;
           wire [29:0] YCRCB3;
           wire [29:0] YCRCB4;
           wire [29:0] YCRCB5;
           wire [29:0] YCRCB6;
           wire [29:0] YCRCB7;
           wire [29:0] YCRCB8;
```

```
        wire [29:0] YCRCB9;
        wire [29:0] YCRCB10;
        wire [29:0] YCRCB11;
        wire [29:0] YCRCB12;
        wire [29:0] YCRCB13;
        wire [29:0] YCRCB14;
        wire [29:0] YCRCB15;
        wire [29:0] YCRCB16;
        wire [29:0] YCRCB17;

        parameter POSX = 800;
        parameter POSY = 25;
        parameter color = 30'b111111111100000000000000000000;
        wire [11:0] mx,my;
 wire [2:0]  btn_click;
        ps2_mouse_xy m1(clk130, reset, mouse_clock, mouse_data, mx, my, btn_click);
 defparam    m1.MAX_X = 1023;    // max - blob size
 defparam    m1.MAX_Y = 767;

        wire overshoot = ((mx < 950) & (mx > 800) & (my > 25) & (my < 75));
        wire capture = btn_click[2] & overshoot;

        wire overview = ((mx < 950) & (mx > 800) & (my > 115) & (my < 165));
        wire view = btn_click[2] & overview;

        wire overlive = ((mx < 950) & (mx > 800) & (my > 205) & (my < 255));
        wire live = btn_click[2] & overlive;

        rect k1(x, y, 7, 22, 4, 457, YCRCB1);
                defparam k1.COLOR = color;
        rect k2(x, y, 715, 22, 4, 457, YCRCB2);
                defparam k2.COLOR = color;
rect k3(x, y, 7, 22, 712, 4, YCRCB3);
                defparam k3.COLOR = color;
rect k4(x, y, 7, 477, 712, 4, YCRCB4);
                defparam k4.COLOR = color;

        rect R1(x, y, POSX, POSY, 150, 5, YCRCB5);
                defparam R1.COLOR = color;
         rect R2( x, y, POSX, POSY + 5, 5, 40, YCRCB6);
                defparam R2.COLOR = color;
         rect R3( x, y, POSX+145, POSY + 5, 5, 40, YCRCB7);
                defparam R3.COLOR = color;
         rect R4( x, y, POSX, POSY+45, 150, 5, YCRCB8);
                defparam R4.COLOR = color;
```

```
        rect S1(x, y, POSX, POSY+90, 150, 5, YCRCB9);
                defparam S1.COLOR = color;
        rect S2( x, y, POSX, POSY + 95, 5, 40, YCRCB10);
                defparam S2.COLOR = color;
        rect S3( x, y, POSX+145, POSY + 95, 5, 40, YCRCB11);
                defparam S3.COLOR = color;
        rect S4( x, y, POSX, 160, 150, 5, YCRCB12);
                defparam S4.COLOR = color;

         rect A1(x, y, POSX, 205, 150, 5, YCRCB13);
                defparam A1.COLOR = color;
        rect A2( x, y, POSX, 210, 5, 40, YCRCB14);
                defparam A2.COLOR = color;
        rect A3( x, y, POSX+145, 210, 5, 40, YCRCB15);
                defparam A3.COLOR = color;
        rect A4( x, y, POSX, 250, 150, 5, YCRCB16);
                defparam A4.COLOR = color;


        blob ball(mx[9:0],my[9:0],x[9:0],y,YCRCB17);
 //  wire up to ZBT ram



       assign YCRCB = (YCRCB1 | YCRCB2 | YCRCB3 | YCRCB4 | YCRCB5 |
YCRCB6 | YCRCB7 | YCRCB8 | YCRCB9 | YCRCB10 | YCRCB11 | YCRCB12 |
YCRCB13 | YCRCB14 | YCRCB15 | YCRCB16 | YCRCB17);

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:40:18 04/05/06
// Design Name:
// Module Name:    rect
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
```

```
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module rect( x, y, posx, posy, width, height, YCRCB);
        parameter COLOR = 30'b000001110100011011000011111111;
   input [9:0] x;
   input [9:0] y;
        input [9:0] posx, posy;
   input [9:0] width;
   input [9:0] height;
   output reg [29:0] YCRCB;

        always @ (posx or x or width or posy or y or height)
        begin
                if ((posx <= x) && (x < (posx + width)) && (posy <= y) && (y < (posy +
height)))
                        YCRCB = COLOR;
                else
                        YCRCB = 30'd0;
        end

endmodule

//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//

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

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

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

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

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

  reg [1:0]   we_delay;

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

  // create two-stage pipeline for write data

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

  // wire to ZBT RAM signals

  assign      ram_we_b = ~we;
  assign      ram_clk = ~clk;     // RAM is not happy with our data hold
                                  // times if its clk edges equal FPGA's
                                  // so we clock it on the falling edges
                                  // and thus let data stabilize longer
        assign ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
     assign ram_we_b = ~we;
    assign ram_address= addr;
```

```
        always @ (posedge clk) begin
      read_data <= ram_data;
            end

endmodule // zbt_6111

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:43:24 05/07/06
// Design Name:
// Module Name:    capture_input
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module capture_input(clk, reset, shoot, view, live, shootmode, viewmode, livemode);
input clk;
input reset;
input shoot;
input view;
input live;
output reg shootmode;
output reg viewmode;
output reg livemode;

always @ (posedge clk) begin
        if (reset) begin
                shootmode <= 0;
                viewmode <= 0;
                livemode <= 0;
        end
        else if (shoot) begin
                shootmode <= 1;
                viewmode <= 0;
                livemode <= 0;
```

```
        end
  else if (view) begin
                shootmode <= 0;
                viewmode <= 1;
                livemode <= 0;
        end
        else if (live) begin
                shootmode <= 0;
                viewmode <= 0;
                livemode <= 1;
        end
end



endmodule

 module blob(x,y,hcount,vcount,pixel);

  parameter WIDTH = 10;
  parameter HEIGHT = 10;
  parameter COLOR = 30'b111111111100000000000000000000;

  input [9:0] x,hcount;// x,y specifies upper left corner
  input [9:0] y,vcount;
  output [29:0] pixel;

  reg [29:0]     pixel;
  always @(x or y or hcount or vcount) begin
    if ((hcount >= x && hcount < (x+WIDTH)) &&
          (vcount >= y && vcount < (y+HEIGHT)))
        pixel = COLOR;
    else pixel = 30'd0;
  end
endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:17:11 05/14/06
// Design Name:
// Module Name:    huhs_to_sram
// Project Name:
// Target Device:
```

```
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module huhs_to_sram(startfeed, clk, input_data, output_addr, output_data, comp_we,
finished);
input startfeed, clk;
output  [18:0] output_addr;
input [35:0] input_data;
output  [35:0] output_data;
output comp_we;
output finished;
reg [10:0] hccount;
reg [9:0] vccount;
reg start;

reg finished;
wire rightedge;
wire bottom;
wire move_top_corner_left;
wire stop;
wire move_top_corner_right;
wire rightedgeblock;
wire bottomblock;
wire bringerback;
assign  rightedge = (hccount == 718);
assign  bottom = (vccount == 479);
assign rightedgeblock = (hccount[2:0] == 3'b110);
assign bottomblock =  (vccount[2:0] == 3'b111);
assign  move_top_corner_left = (rightedge & ~bottom);
assign  stop = (rightedge & bottom);

assign bringerback = (rightedgeblock & ~bottomblock);
assign  move_top_corner_right = (rightedgeblock & bottomblock & ~stop &
~move_top_corner_left);

assign output_data = input_data;
assign output_addr = {vccount, hccount};
```

```
always @ (posedge clk) begin
        if (finished) begin
                hccount <= 10'd7;
         vccount <= 9'd24;
                finished <=1;
                start <=0;
        end
        else if (~startfeed) begin
      hccount <= 10'd7;
              vccount <= 9'd24;
                      start <=0;
                      finished <= 0;
        end
        else if (startfeed) begin
                      start <=1;
        end

        if (start) begin
                if (bringerback) begin
                       hccount <= hccount - 7;
                       vccount <= vccount + 1;

                end
                else if (move_top_corner_right) begin
        hccount <= (hccount + 1);
                      vccount <= vccount-7;
                end
                else if (move_top_corner_left) begin
                      hccount <= 10'd7;
                      vccount <= vccount + 1;
                end
                else if (stop) begin
                      hccount <= 10'd7;
                      vccount <= 9'd24;
                      finished <=1;
                      start<=0;
                end
                else begin
                       hccount <= hccount + 1;
                       vccount <= vccount;

                end
        end
end
```

```
endmodule

//
// File:   cstringdisp.v
// Date:   24-Oct-05
// Author: I. Chuang, C. Terman
//
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//
// INPUTS:
//
//   vclock       - video pixel clock
//   hcount       - horizontal (x) location of current pixel
//   vcount       - vertical (y) location of current pixel
//   cstring      - character string to display (8 bit ASCII for each char)
//   cx,cy        - pixel location (upper left corner) to display string at
//
// OUTPUT:
//
//   pixel        - video pixel value to display at current location
//
// PARAMETERS:
//
//   NCHAR        - number of characters in string to display
//   NCHAR_BITS   - number of bits to specify NCHAR
//
// pixel should be OR'ed (or XOR'ed) to your video data for display.
//
// Each character is 8x12, but pixels are doubled horizontally and vertically
// so fonts are magnified 2x.  On an XGA screen (1024x768) you can fit
// 64 x 32 such characters.
//
// Needs font_rom.v and font_rom.ngo
//
// For different fonts, you can change font_rom.  For different string
// display colors, change the assignment to cpixel.


//////////////////////////////////////////////////////////////////////////////
//
// video character string display
//
//////////////////////////////////////////////////////////////////////////////
```

```
module char_string_display (vclock, hcount,vcount,pixel,cstring,cx,cy);

  parameter NCHAR = 1;      // number of 8-bit characters in cstring
  parameter NCHAR_BITS = 7; // number of bits in NCHAR
  input vclock;// 65MHz clock
  input [10:0] hcount; // horizontal index of current pixel (0..1023)
  input [9:0]   vcount; // vertical index of current pixel (0..767)
  output [2:0] pixel;   // char display's pixel
  input [NCHAR*8-1:0] cstring;       // character string to display
  input [10:0] cx;
  input [9:0]   cy;
//        input parity;
  // 1 line x 8 character display (8 x 12 pixel-sized characters)

  wire [10:0]   hoff = hcount-1-cx;
  wire [9:0]    voff = vcount-cy;
  wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // <
NCHAR
  wire [2:0]    h = hoff[3:1];          // 0 .. 7
  wire [3:0]    v = voff[4:1];              // 0 .. 11

  // look up character to display (from character string)
  reg [7:0]  char;
  integer  n;
  always @( *) begin
                  //if (parity) begin
                  for (n=0 ; n<8 ; n = n+1 )                // 8 bits per character (ASCII)
         char[n] <= cstring[column*8+n];
        //        end
                  end
  // look up raster row from font rom
  wire reverse = char[7];
  wire [10:0] font_addr = char[6:0]*12 + v;    // 12 bytes per character
  wire [7:0]  font_byte;
  font_rom f(font_addr,vclock,font_byte);

  // generate character pixel if we're in the right h,v area
  wire [2:0] cpixel = (font_byte[7 - h] ^ reverse) ? 7 : 0;
  wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*16)
                    & (vcount < cy + 24));
  wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule

// ps2_mouse_xy gives a high-level interface to the mouse, which
// keeps track of the "absolute" x,y position (within a parameterized
```

```
// range) and also returns button presses.

module ps2_mouse_xy(clk, reset, ps2_clk, ps2_data, mx, my, btn_click);

  input clk, reset;
  inout ps2_clk, ps2_data;      // data to/from PS/2 mouse
  output [11:0] mx, my;         // current mouse position, 12 bits
  output [2:0]  btn_click;      // button click: Left-Middle-Right

  // module parameters
  parameter     MAX_X = 1023;
  parameter     MAX_Y = 767;

  // low level mouse driver

  wire [8:0]    dx, dy;
  wire [2:0]    btn_click;
  wire  data_ready;
  wire  error_no_ack;
  wire [1:0]    ovf_xy;
  wire  streaming;

  ps2_mouse m1(clk,reset,ps2_clk,ps2_data,dx,dy,ovf_xy, btn_click,
                  data_ready,streaming);

  // Update "absolute" position of mouse

  reg [11:0]  mx, my;
  wire       sx = dx[8];          // signs
  wire       sy = dy[8];
  wire [8:0]  ndx = sx ? {0,~dx[7:0]}+1 : {0,dx[7:0]};        // magnitudes
  wire [8:0]  ndy = sy ? {0,~dy[7:0]}+1 : {0,dy[7:0]};

  always @(posedge clk) begin
    mx <= reset ? 0 :
          data_ready ? (sx ? (mx>ndx ? mx - ndx : 0)
                           : (mx < MAX_X - ndx ? mx+ndx : MAX_X)) : mx;
    // note Y is flipped for video cursor use of mouse
    my <= reset ? 0 :
          data_ready ? (sy ? (my < MAX_Y - ndy ? my+ndy : MAX_Y)
                           : (my>ndy ? my - ndy : 0))  : my;
//          data_ready ? (sy ? (my>ndy ? my - ndy : 0)
//                           : (my < MAX_Y - ndy ? my+ndy : MAX_Y)) : my;
  end

endmodule
```

```
//////////////////////////////////////////////////////////////////////////////////
//  PS/2 MOUSE
//
//  6.111 Fall 2005
//
//  NOTE:  make sure to change the mouse ports (mouse_clock, mouse_data) to
//                  bi-directional 'inout' ports in the top-level module
//
//  specifically, labkit.v should have the line
//
//     inout  mouse_clock, mouse_data;
//
// This module interfaces to a mouse connected to the labkit's PS2 port.
// The outputs provided give dx and dy movement values (9 bits 2's comp)
// and three button click signals.
//
// NOTE: change the following parameters for a different system clock
//         (current configuration for 50 MHz clock)
//                  CLK_HOLD                        : 100 usec hold to bring PS2_CLK
low
//                  RCV_WATCHDOG_TIMER_VALUE : (PS/2 RECEIVER) 2 msec count
//                  RCV_WATCHDOG_TIMER_BITS  :                      bits needed for
timer
//                  INIT_TIMER_VALUE                : (INIT process) 0.5 sec count
//                  INIT_TIMER_BITS          :                      bits needed for timer
//////////////////////////////////////////////////////////////////////////////////
//
// Nov-8-2005: Registered the outputs (dout_dx, dout_dy, ovf_xy, btn_click) in
[ps2_mouse]
//                          Added output "streaming"
// Nov-9-2005: synchronized ps2_clk to local clock for transmitter in [ps2_interface]
//                          Programmed watchdog_timer for [ps2] (receiver module)
//                          Programmed init_timer for [ps2_mouse] (resets initialization)
//////////////////////////////////////////////////////////////////////////////////

module ps2_mouse(clock, reset, ps2_clk, ps2_data, dout_dx,
                  dout_dy, ovf_xy, btn_click, ready, streaming);

  input clock, reset;
  inout ps2_clk, ps2_data;              //data to/from PS/2 mouse
  output [8:0] dout_dx, dout_dy;        //9-bit 2's compl, indicates movement of mouse
  output [1:0] ovf_xy;                  //==1 if overflow: dx, dy
  output [2:0] btn_click;               //button click: Left-Middle-Right
  output ready;                         //synchronous 1 cycle ready flag
  output streaming;                             //==1 if mouse is in stream mode
```

```
  ////////////////////////////////////////////////////
  // PARAMETERS
  // # of cycles for clock=50 MHz
  parameter CLK_HOLD = 13000;                                    //100 usec hold to
bring PS2_CLK low
  parameter RCV_WATCHDOG_TIMER_VALUE = 260000;       // For PS/2
RECEIVER : # of sys_clks for 2msec.
  parameter RCV_WATCHDOG_TIMER_BITS  = 18;       //
: bits needed for timer
  parameter INIT_TIMER_VALUE = 65000000;              // For INIT process : sys_clks
for 0.5 sec.(SELF-TEST phase takes several miliseconds)
  parameter INIT_TIMER_BITS  = 28;                        //                          :
bits needed for timer
  ////////////////////////////////////////////////////
  wire reset_init_timer;


  ////////////////////////////////////////////////////
  //CONTROLLER:
  //-initialization process:
  //       Host:  FF  Reset command
  //       Mouse: FA  Acknowledge
  //       Mouse: AA  Self-test passed
  //       Mouse: 00  Mouse ID
  //       Host:  F4  Enable
  //       Mouse: FA  Acknowledge
  parameter SND_RESET = 0,  RCV_ACK1 = 1,  RCV_STEST = 2, RCV_ID = 3;
  parameter SND_ENABLE =4,  RCV_ACK2 = 5,  STREAM = 6;
  reg [2:0] state;

  wire send, ack;
  wire [7:0] packet;
  wire [7:0] curkey;
  wire key_ready;

  //NOTE: no support for scrolling wheel, extra buttons
  always @(posedge clock) begin
    if (reset || reset_init_timer) state <= SND_RESET;
    else case (state)
      SND_RESET:     state <= ack ? RCV_ACK1 : state;
      RCV_ACK1:      state <= (key_ready && curkey==8'hFA) ? RCV_STEST : state;
      RCV_STEST:     state <= (key_ready && curkey==8'hAA) ? RCV_ID : state;
      RCV_ID:        state <= (key_ready) ? SND_ENABLE : state;                  //any
device type
      SND_ENABLE:  state <= ack ? RCV_ACK2 : state;
```

```
    RCV_ACK2:       state <= (key_ready && curkey==8'hFA) ? STREAM :state;
    STREAM:                 state <= state;
        default:        state <= SND_RESET;
  endcase
end

assign send = (state==SND_RESET) || (state==SND_ENABLE);
assign packet = (state==SND_RESET) ? 8'hFF :
                        (state==SND_ENABLE) ? 8'hF4 :
                                                        8'h00;
assign streaming = (state==STREAM);


// Connect PS/2 interface module
ps2_interface ps2_mouse(.reset(reset), .clock(clock),
                                        .ps2c(ps2_clk), .ps2d(ps2_data),
                                        .send(send), .snd_packet(packet), .ack(ack),
                                        .rcv_packet(curkey), .key_ready(key_ready)  );
defparam ps2_mouse.CLK_HOLD                     = CLK_HOLD;
defparam ps2_mouse.WATCHDOG_TIMER_VALUE =
RCV_WATCHDOG_TIMER_VALUE;
defparam ps2_mouse.WATCHDOG_TIMER_BITS  =
RCV_WATCHDOG_TIMER_BITS;

//////////////////////////////////////////////////////////
// DECODER
//http://www.computer-engineering.org/ps2mouse/
//                      bit-7                           3                       bit-0
//Byte 1:  Y-ovf  X-ovf  Y-sign  X-sign  1  Btn-M  Btn-R  Btn-L
//Byte 2:  X movement
//Byte 3:  Y movement
reg [1:0] bindex, old_bindex;
reg [7:0] status, dx, dy;               //temporary storage of mouse status
reg [8:0] dout_dx, dout_dy;             //Clock the outputs
reg [1:0] ovf_xy;
reg [2:0] btn_click;
wire ready;

always @(posedge clock) begin
        if (reset) begin
          bindex <= 0;
          status <= 0;
          dx <= 0;
          dy <= 0;
        end else if (key_ready && state==STREAM) begin
          case (bindex)
```

```
                2'b00:  status <= curkey;
                2'b01:  dx <= curkey;
                2'b10:  dy <= curkey;
                default:        status <= curkey;
            endcase

            bindex <= (bindex==2'b10) ? 0 : bindex + 1;
            if (bindex == 2'b10) begin                                  //Now, dy is ready
                dout_dx   <= {status[4], dx};                           //2's compl 9-bit
                dout_dy   <= {status[5], curkey};                       //2's compl 9-bit
                ovf_xy    <= {status[6], status[7]};                    //overflow: x, y
                btn_click <= {status[0], status[2], status[1]};  //button click: Left-Middle-
Right
        end
            end     //end else-if (key_ready)
  end

  always @(posedge clock)
    old_bindex <= bindex;

  assign ready = (bindex==2'b00) && old_bindex==2'b10;

  ////////////////////////////////////////////////////////
  // INITIALIZATION TIMER
  //      ==> RESET if processs hangs during initialization
  reg [INIT_TIMER_BITS-1:0] init_timer_count;
  assign reset_init_timer = (state != STREAM) &&
(init_timer_count==INIT_TIMER_VALUE-1);
  always @(posedge clock)
  begin
        init_timer_count <= (reset || reset_init_timer || state==STREAM) ?
                                                                    0 : init_timer_count +
1;
  end

endmodule



////////////////////////////////////////////////////////////////////////
// PS/2 INTERFACE: transmit or receive data from ps/2

module ps2_interface(reset, clock, ps2c, ps2d, send, snd_packet, ack, rcv_packet,
key_ready);
   input clock,reset;
   inout ps2c;                          // ps2 clock     (BI-DIRECTIONAL)
```

```
  inout ps2d;                          // ps2 data       (BI-DIRECTIONAL)
  input send;                          //flag: send packet                    _
  output ack;                          // end of transmission                | for
transmitting
  input [7:0] snd_packet;      // data packet to send to PS/2  _|
  output [7:0] rcv_packet;     //packet received from PS/2          _
  output key_ready;            // new data ready (rcv_packet)       _| for receiving

  ////////////////////////////////////////////////////////////////////////
  // MAIN CONTROL
  ////////////////////////////////////////////////////////////////////////
  parameter CLK_HOLD = 13005;                          //hold PS2_CLK low for 100
usec (@50 Mhz)
  parameter WATCHDOG_TIMER_VALUE = 260000;     // Number of sys_clks for
2msec. _
  parameter WATCHDOG_TIMER_BITS  = 18;     // Number of bits needed for timer
        _| for RECEIVER

  wire serial_dout;                    //output (to ps2d)
  wire rcv_ack;                                //ACK from ps/2 mouse after data
transmission

  wire we_clk, we_data;

  assign ps2c = we_clk  ? 0 : 1'bZ;
  assign ps2d = we_data ? serial_dout : 1'bZ;

  assign ack = rcv_ack;


  ////////////////////////////////////////////////////////////////////////
  // TRANSMITTER MODULE
  ////////////////////////////////////////////////////////////////////////

  //////////////////////////////////////////////////////
  // COUNTER: 100 usec hold
  reg [15:0] counter;
  wire en_cnt;
  wire cnt_ready;
  always @(posedge clock) begin
    counter <= reset ? 0 :
                       en_cnt ? counter+1 :
                                0;
  end
  assign cnt_ready = (counter>=CLK_HOLD);
```

```
  ////////////////////////////////////////////////////
  // SEND DATA
  //      hold CLK low for at least 100 usec
  //      DATA low
  //      Release CLK
  //      (on negedge of ps2_clock) - device brings clock LOW
  //              REPEAT:       SEND data
  //      Release DATA
  //      Wait for device to bring DATA low
  //      Wait for device to bring CLK low
  //      Wait for device to release CLK, DATA
  reg [3:0] index;

  // synchronize PS2 clock to local clock and look for falling edge
  reg [2:0] ps2c_sync;
  always @ (posedge clock) ps2c_sync <= {ps2c_sync[1:0],ps2c};
  wire falling_edge = ps2c_sync[2] & ~ps2c_sync[1];

  always @(posedge clock) begin
        if (reset) begin
              index <= 0;
        end
        else if (falling_edge) begin                    //falling edge of ps2c
          if (send) begin                                       //transmission mode
              if (index==0)
                    index <= cnt_ready ? 1 : 0;             //index=0: CLK low
              else
                    index <= index + 1;                     //index=1:
snd_packet[0], =8: snd_packet[7],
                                                                  //      9: odd
parity, =10: stop bit
                                                                  //          11:
wait for ack
          end else
              index <= 0;
        end else
          index <= (send) ? index : 0;
  end

  assign en_cnt = (index==0 && ~reset && send);
  assign serial_dout = (index==0 && cnt_ready) ? 0 :                    //bring
DATA low before releasing CLK
                                    (index>=1 && index <=8) ? snd_packet[index-1] :
                                    (index==9) ? ~(^snd_packet) :
        //odd parity
```

```
                                                    1;
                    //including last '1' stop bit

  assign we_clk = (send && !cnt_ready && index==0);
        //Enable when counter is counting up
  assign we_data = (index==0 && cnt_ready) || (index>=1 && index<=9);//Enable after
100usec CLK hold
  assign rcv_ack = (index==11 && ps2d==0);
        //use to reset RECEIVER module


  ////////////////////////////////////////////////////////////////////////
  // RECEIVER MODULE
  ////////////////////////////////////////////////////////////////////////
  reg [7:0]   rcv_packet;                 // current keycode
  reg             key_ready;              // new data
  wire       fifo_rd;                     // read request
  wire [7:0]  fifo_data;                  // data from mouse
  wire       fifo_empty;                  // flag: no data
  //wire       fifo_overflow;    // keyboard data overflow

  assign      fifo_rd = ~fifo_empty;    // continuous read

  always @(posedge clock)
  begin
        // get key if ready
        rcv_packet <= ~fifo_empty ? fifo_data : rcv_packet;
        key_ready  <= ~fifo_empty;
  end

  ////////////////////////////////////////////////////
  // connect ps2 FIFO module
  reg [WATCHDOG_TIMER_BITS-1 : 0] watchdog_timer_count;
  wire [3:0] rcv_count;                                 //count incoming data bits from ps/2
(0-11)

  wire watchdog_timer_done =
watchdog_timer_count==(WATCHDOG_TIMER_VALUE-1);
  always @(posedge clock)
  begin
        if (reset || send || rcv_count==0) watchdog_timer_count <= 0;
        else if (~watchdog_timer_done)
          watchdog_timer_count <= watchdog_timer_count + 1;
  end
```

```
   ps2 ps2_receiver(.clock(clock), .reset(!send && (reset || rcv_ack) ),         //RESET on
reset or End of Transmission
                                        .ps2c(ps2c), .ps2d(ps2d),
                                        .fifo_rd(fifo_rd), .fifo_data(fifo_data),
          //in1, out8
                                        .fifo_empty(fifo_empty) , .fifo_overflow(),
          //out1, out1
                                        .watchdog(watchdog_timer_done),
.count(rcv_count) );

endmodule



////////////////////////////////////////////////////////////////////////
// PS/2 FIFO receiver module (from 6.111 Fall 2004)

module ps2(reset, clock, ps2c, ps2d, fifo_rd, fifo_data, fifo_empty,fifo_overflow,
watchdog, count);
  input clock,reset,watchdog,ps2c,ps2d;
  input fifo_rd;
  output [7:0] fifo_data;
  output fifo_empty;
  output fifo_overflow;
  output [3:0] count;

  reg [3:0] count;      // count incoming data bits
  reg [9:0] shift;      // accumulate incoming data bits

  reg [7:0] fifo[7:0];   // 8 element data fifo
  reg fifo_overflow;
  reg [2:0] wptr,rptr;   // fifo write and read pointers

  wire [2:0] wptr_inc = wptr + 1;

  assign fifo_empty = (wptr == rptr);
  assign fifo_data = fifo[rptr];

  // synchronize PS2 clock to local clock and look for falling edge
  reg [2:0] ps2c_sync;
  always @ (posedge clock) ps2c_sync <= {ps2c_sync[1:0],ps2c};
  wire sample = ps2c_sync[2] & ~ps2c_sync[1];

  reg timeout;
  always @ (posedge clock) begin
    if (reset) begin
```

```
      count <= 0;
      wptr <= 0;
      rptr <= 0;
      timeout <= 0;
      fifo_overflow <= 0;
    end else if (sample) begin
      // order of arrival: 0,8 bits of data (LSB first),odd parity,1
      if (count==10) begin
        // just received what should be the stop bit
        if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
          fifo[wptr] <= shift[8:1];
          wptr <= wptr_inc;
             fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
        end
        count <= 0;
        timeout <= 0;
      end else begin
        shift <= {ps2d,shift[9:1]};
        count <= count + 1;
      end
    end else if (watchdog && count!=0) begin
      if (timeout) begin
        // second tick of watchdog while trying to read PS2 data
           count <= 0;
        timeout <= 0;
      end else timeout <= 1;
    end

    // bump read pointer if we're done with current value.
    // Read also resets the overflow indicator
    if (fifo_rd && !fifo_empty) begin
      rptr <= rptr + 1;
      fifo_overflow <= 0;
    end
  end
endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    19:52:09 04/24/06
// Design Name:
// Module Name:    vram_display
// Project Name:
```

```
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
// 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(shoot, parity, reset, clk, hcount, vcount, pixel, vram_read_data);
input reset, clk, parity, shoot;
input [10:0] hcount;
input [9:0] vcount;
output [29:0] pixel;
input [35:0] vram_read_data;
parameter HSHIFT = 14;
parameter VSHIFT = 35;
parameter HMID = 9'd367; // The horizontal center of the image in MEMORY
parameter HSTART = HMID-9'd256; // The horizontal counter decrements!!!
parameter VMID = 9'd287; // The vertical center of the image in MEMORY
parameter VSTART = VMID-9'd192;
wire [10:0] hcountshift;
wire [9:0] vcountshift;

wire hc2 = hcount[0];
assign vcountshift = vcount + VSHIFT;
assign hcountshift = hcount + HSHIFT;

 wire [18:0] vram_addr = {vcount[8:0], hcount[9:0]};

reg [29:0] vr_pixel;
reg [35:0] vr_data_latched;
reg [35:0] last_vr_data;
reg [29:0] pixel;
```

```
always @(posedge clk) begin

        if (~parity) begin
                vr_pixel <=  vr_data_latched[29:0] ;
          vr_data_latched <=  vram_read_data ;
        end
end

always @ (posedge clk) begin
        if ((vcount<22)  || (vcount>480) || (hcount < 6) || (hcount>717))
                 pixel <= 30'b000100000100000000001000000000;
        else pixel<= vr_pixel;
end


endmodule // vram_display

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:34:36 05/12/06
// Design Name:
// Module Name:    write_to_huhs
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module write_to_huhs(shoot,clk27, shotaddr, shot_read_data, huhsdata, start, done);

input clk27, shoot;
output reg start;
output reg done;
output  [29:0] huhsdata;
output  [18:0] shotaddr;
input [35:0] shot_read_data;
```

```
reg topcorner [18:0];
reg [9:0] hmcount;
reg [8:0] vmcount;
wire rightedge;
wire bottom;
wire move_top_corner_left;
wire stop;
wire move_top_corner_right;
wire rightedgeblock;
wire bottomblock;
wire bringerback;
assign  rightedge = (hmcount == 718);
assign  bottom = (vmcount == 479);
assign rightedgeblock = (hmcount[2:0] == 3'b110);
assign bottomblock =  (vmcount[2:0] == 3'b111);
assign  move_top_corner_left = (rightedge & ~bottom);
assign  stop = (rightedge & bottom);

assign bringerback = (rightedgeblock & ~bottomblock);
assign  move_top_corner_right = (rightedgeblock & bottomblock & ~stop &
~move_top_corner_left);

assign huhsdata = shot_read_data[29:0];
assign shotaddr = {vmcount, hmcount};

always @ (posedge clk27) begin
          if (done) begin
                    hmcount <= 10'd7;
           vmcount <= 9'd24;
                    done <=1;
   end
          else if (~shoot) begin
       hmcount <= 10'd7;
                  vmcount <= 9'd24;
                            done <=0;
          end
          else if (shoot) begin
                    start <= 1;
          end
          if (start) begin
                    if (bringerback) begin
                              hmcount <= hmcount - 7;
                              vmcount <= vmcount + 1;

                    end
                    else if (move_top_corner_right) begin
```

```
        hmcount <= (hmcount + 1);
                        vmcount <= vmcount-7;
                end
                else if (move_top_corner_left) begin
                        hmcount <= 10'd7;
                        vmcount <= vmcount + 1;
                end
                else if (stop) begin
                        hmcount <= 10'd7;
                        vmcount <= 9'd24;
                        done <=1;
                end
                else begin
                         hmcount <= hmcount + 1;
                         vmcount <= vmcount;

                end
         end
end


endmodule
```